Contents

1 Introduction to ARM **3**
1 ARM Overview . . . 3
1.1 Computation . . . 3
1.2 Commands . . . 4

2 Digital Logic Design **4**
2 Circuits and Transistors . . . 4
2.1 Circuits . . . 4
2.2 Transistors . . . 7
3 More Complex Components . . . 7
3.1 Decoder . . . 7
3.2 Multiplexor . . . 8
3.3 Latches . . . 8
3.4 Register File . . . 9
4 RAM, FSM, Data Representation . . . 11
4.1 Random Access Memory . . . 11
4.2 Finite State Machine . . . 12
4.3 Data Representation . . . 13
5 ALU, Floating Point . . . 14
5.1 ALU . . . 14
5.2 Floating Point . . . 15

3 Single Cycle Processing **15**
6 Single Cycle Processor . . . 15
6.1 Datapath Overview . . . 15
6.2 Specifics . . . 17
7 Single Cycle cont.; Multicycle Datapath . . . 18
7.1 More on Single Cycle Processor . . . 18

7.2 Multicycle Processor Datapath . . . 18

4 Pipelining **20**

8 Pipelined Datapath . . . 20

9 Control Hazards . . . 21

9.1 Data hazards . . . 21

9.2 Branch Hazards . . . 22

9.3 Branch Predicting . . . 22

5 Memory **23**

10 Cache . . . 23

11 Virtual Memory . . . 25

11.1 TLB . . . 26

12 Extras . . . 26

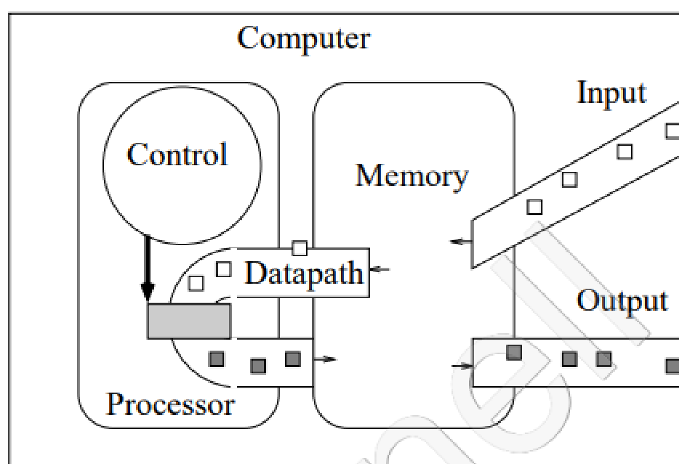
Unit 1 Introduction to ARM

Week 1 ARM Overview

1.1 Computation

High level languages are compiled to assembly languages such as ARM (think PRIMPL / MMIX) which are in turn assembled into binary code.

Computers work with current / voltage ($V = IR$). These quantities are continuous, manipulable values, however, they are difficult to accurately measure. Hence, we digitize them to discrete, binary values (High: 1, Low: 0). Binary values are used as they are simpler.



Memory goes from input to the registers, from the registers the processor, where an operation is performed on it, back to the registers, and then to output or wherever is necessary.

Definition. ARM: ARM stands for Advanced RISC (Reduced Instruction Set Computer) Machine. It is an assembly language with few, small, quick to execute commands.

Definition. Register: There are 32 general registers in ARM holding 64 bits each. Each register is denoted Xi, i.e. X0 to X31, however, X31 is reserved and holds only 0.

Note. Memory: Memory is segmented into 4 byte (word) and 8 byte (double-word) blocks. Instructions hold a word of memory, where as data holds a double-word of memory.

Definition. Program Counter (PC): Special register which holds the address of the next instruction in the program.

Note: Addresses in a program increment by 4 each time as an instruction is generally 32 bits, i.e. 4 bytes, called a word. Memory access is by accessing a byte in memory, hence addresses increase by 4 per instruction. After each instruction, $PC \leftarrow PC + 4$.

Note: Data in ARM is either an address (e.g. a register like `X5`) or an immediate (e.g. an integer like `#55`).

Remark. Instruction: Instructions come in 5 formats:

- R-Format: Inst Addr, Addr, Addr; e.g. `ADD X1,X2,X3`
- D-Format: Inst Addr, [Addr, Imm]; e.g. `LDUR X1, [X2, #20]`
- I-Format: Inst Addr Addr, Imm; e.g. `ADDI X1, X2, #100`
- B-Format: Inst Imm; e.g. `B #28`
- CB-Format: Inst Addr, Imm; e.g. `CBZ X1,#8`

1.2 Commands

Note: We use $M[x]$ to represent the memory held at the xth location, for instance $M[4]$ is the memory held at the second word, 4th byte.

Branches ARM has no if statements or loops, rather it uses conditional and unconditional branching. An unconditional branch is of the form `B Imm` and immediately sets PC to $PC \leftarrow PC + Imm \times 4$.

There are multiple conditional branches, for instance `CBZ Addr, Imm` which sets $PC \leftarrow PC + Imm \times 4$ if $Addr \neq 0$. Similarly is `CBNZ` which branches when the address is non-zero.

Memory accessing To load into a register, we use `LDUR Dst, [Src, Imm]`. This has the effect of $Dst \leftarrow M[Src + Imm]$ where Dst and Src are registers.

To save a register's value in memory, we use `STUR Src, [Dst, Imm]`. This has the effect of $M[Dst + Imm] \leftarrow Src$ where Dst and Src are registers.

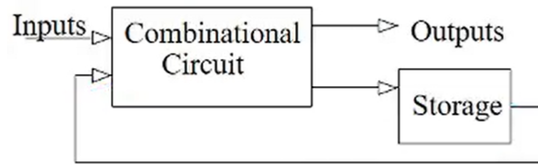
Unit 2 Digital Logic Design

Week 2 Circuits and Transistors

2.1 Circuits

Definition. Combinational Circuit: A combinational circuit consumes n Boolean inputs and returns m Boolean outputs. Recall a Boolean 1 is high and 0 is low voltage.

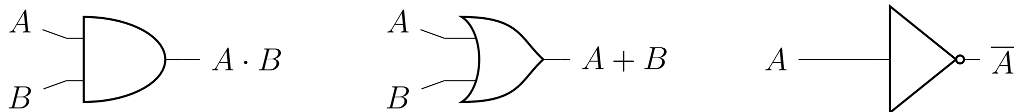
Definition. Sequential Circuit: A sequential circuit has inputs passed to a combination circuit and some of the outputs of the combinational circuit are passed along to output, while others are kept in storage. In the next clock cycle, the information in storage is passed to the combinational circuit along with inputs.



Remark: We use Boolean algebra in circuits, i.e. arithmetic in $\mathbb{Z}_2$. For instance for $X, Y \in \mathbb{Z}_2$, we defined X and Y as $X \cap Y = X \cdot Y = XY$ and X or Y as $X \cup Y = X + Y$. Further, we denote the complement as $\overline{X} = 1 - X$.

Definition. Minterm: A minterm or minimal term is a combination of inputs, for instance each combination of XYZ is a minterm. Notice we tend to use shorthand and to denote the minterm of $X = 1, Y = 0, Z = 1$, we write $X\overline{Y}Z$. Notice also, each row of a truth table has a corresponding minterm and we write the rows in increasing order by the binary representation of the minterm (i.e. $X\overline{Y}Z = 101 = 5$).

Remark. Circuit Representations:



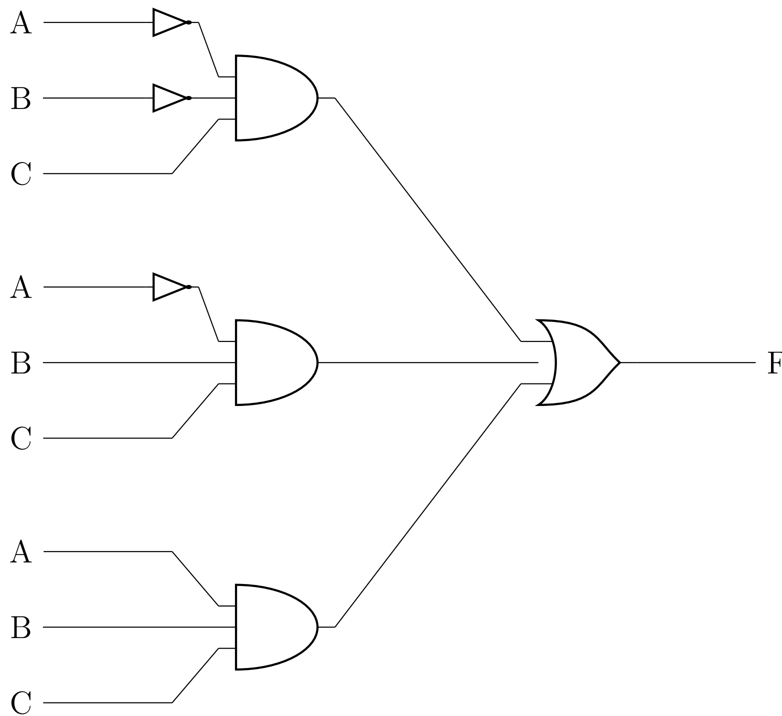
Remark. Truth Tables:

OR		
A	B	$A+B$
0	0	0
0	1	1
1	0	1
1	1	1

AND		
A	B	$A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

NOT	
A	$\neg A$
0	1
1	0

Example: Example of the logic circuit $F = \overline{AB}C + A\overline{B}C + ABC$.

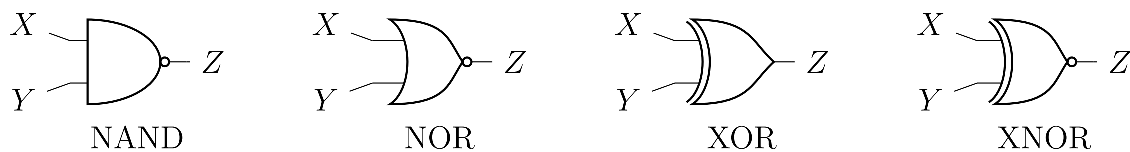


Remark. Don't Cares: In truth tables we occasionally use an X in the place of a 0 or 1 in input to indicate that regardless of the value the variable has, the output is the same. If the X occurs in the output, then it indicates we don't care about the output for this input.

Note. Laws of Boolean Algebra: Following mostly from the requirements for a field:

Rule	Dual Rule	Name
$\overline{\overline{X}} = X$		
$X + 0 = X$	$X \cdot 1 = X$	Identity
$X + 1 = 1$	$X \cdot 0 = 0$	Zero/one
$X + X = X$	$XX = X$	Absorption
$X + \overline{X} = 1$	$X\overline{X} = 0$	Inverse
$X + Y = Y + X$	$XY = YX$	Commutative
$X + (Y + Z) = (X + Y) + Z$	$X(YZ) = (XY)Z$	Associative
$X(Y + Z) = XY + XZ$	$X + YZ = (X + Y)(X + Z)$	Distributive
$\overline{X + Y} = \overline{X} \cdot \overline{Y}$	$\overline{XY} = \overline{X} + \overline{Y}$	DeMorgan

Remark: We introduce now the remaining simple logic gates. The NAND gate is $\overline{XY}$, NOR is $\overline{X + Y}$, XOR is $X\overline{Y} + \overline{X}Y$, and XNOR is $XY + \overline{XY}$. The following ar their gates



2.2 Transistors

Definition. Transistor: An electrically controlled switched, i.e. lets current pass when on, doesn't otherwise. An NMOS transistor ("n-transistor") behaves like a transistor but transmits strong 0's and weak 1's (expected 5v, gets 3-4v). A PMOS transistor ("p-transistor") is the opposite, weak 0's strong 1's.

Input	NMOS	PMOS
1	L	H
0	H	L

Definition. CMOS: A CMOS circuit uses p and n transistors to make circuits with clean paths to power and ground. This means there are no weak signals.

Remark. Circuit Analysis: To analyze a circuit, for each possible input, determine the resistance of each transistor. If the output has a low resistance path to power and high to ground than it is 1, if it has a low resistance path to ground and high to power it is 0. In the case that there is a low resistance path to power and ground, it is a *short circuit* marked by a star $\star$. In the case that there is a high resistance path to power and ground, it is a *float state* disconnected from both, marked by a dash $-$. A circuit with only 0 and 1 outputs is said to be stable, otherwise it is unstable.

Remark: An AND gate is just a NAND gate followed by a NOT gate.

Remark. Number of Transistors per Gate: A NOT gate has 2 transistors. A NAND gate has 2 transistors per input (minimum 4 total). A NOR gate has 2 transistor per input (minimum 4 total).

Week 3 More Complex Components

3.1 Decoder

Definition. Decoder: A component of a digital circuit, it has n inputs and 2^n outputs. In particular, it translates the n-bit input into a signal in the index that corresponds to the binary value of the input. Note this means one and only one output will ever be activated at once.

Remark: The decoder can be used for instance to determine which piece of hardware, which bit, etc. to activate in any given clock cycle.

Example: An example of a usage of the decoder is for expanding memory by a process called chip select. For instance, if we have 1 GB or 2^{30} B memory chips, each byte being addressed using a 2^{30} bit input, we can quadruple the memory into 4 GB by using a decoder. You pass instead 2^{32} bits, the decoder takes the first 2 bits to determine which memory chip to send the reference to.

3.2 Multiplexor

Definition. Multiplexor (MUX): It has n select bits/lines and 2^n lines for input, and a single output. In particular, it outputs the line with binary representation equal to select bits. That is, it chooses one of the input lines to pass through.

Definition. Bit Order: When considering bits for binary representations, we say S_0 is the lower order bit and it appears at the far right. S_n is the higher order bit and it appears at the far left. For instance, 2 in binary is 10, hence we have higher order bit $S_1 = 1$ and lower order bit $S_0 = 0$.

Remark. Multiple Bit Multiplexors: To use a (parent) multiplexor which consumes k n-bit inputs and returns an n-bit output, we use an array of (child) multiplexors. In particular, each bit of each of the k inputs is connected to a single multiplexor (n total multiplexors, 1 per bit), all sharing the same select bit, and return the output of each multiplexor in order.

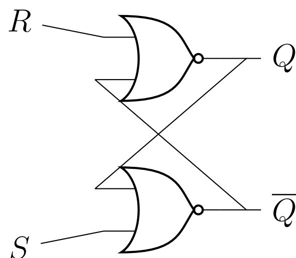
Definition. Bus: A slash across an input or an output is used to indicate a bus, a collection of data lines that are treated as a single logical signal.

Example: 64-bit multiplexors can be useful for instance in getting the computer to execute operations before receiving them. If it is unsure which arithmetic operation ($+$, $-$, $\times$, $\div$) then it'll compute all of them and use a 64-bit multiplexor to choose the appropriate output.

3.3 Latches

Definition. Synchronous: A synchronous (sequential) circuit has a clock pulse which is fed in every cycle as a clock bit. An asynchronous circuity does not have this clock pulse, making it potentially faster and less power-hungry, but harder to design and analyze.

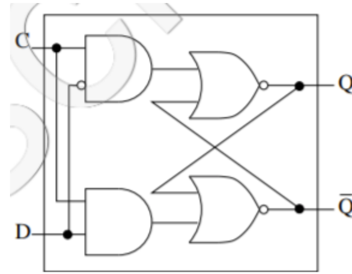
Definition. Set/Reset Latch: A Set/Reset (SR) latch holds a bit of information.



When $S, R = 0, 0$ the value of Q is unmodified. When $S, R = 0, 1$ the value of Q is set to 1. When $S, R = 0, 1$ the value of Q is set to 0. When $S, R = 1, 1$, the value of $Q, \overline{Q}$ oscillates from $0, 0$ to $1, 1$ indefinitely. We implement safety measures before the SR latch to avoid S, R every being $1, 1$.

Definition. Falling/Rising Edge: The rising edge of a signal is the edge of where the signal goes from low to high. The falling edge is where the signal goes from high to low.

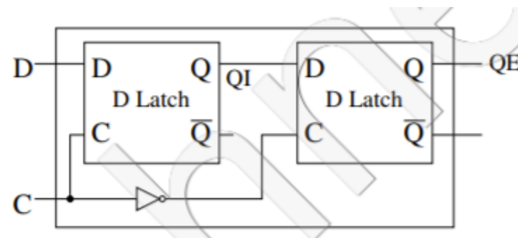
Definition. D-Latch: A D-Latch implements a clock bit to copy the value from an input D so long as the clock bit is high.

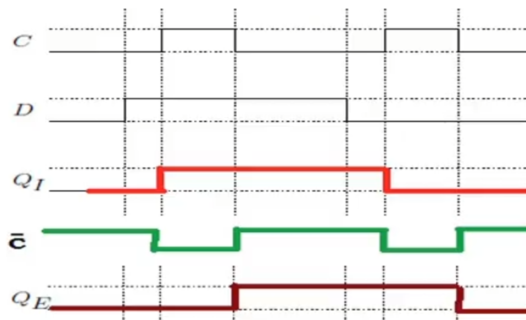


When the clock bit is high, Q will receive the value of D, when the clock bit is low Q will remain unchanged. We consider the changes to be instantaneous, though it is not in reality. In particular, a change to D on the falling edge of C has no impact.

Remark: In the case that D changes during a clock pulse, so will Q, this is generally an undesirable trait for a memory bit.

Definition. D Flip-Flop: A D flip-flop implements two D-latches in a master-slave to design to make it so that memory is only updated on the clock's falling edge. This means a D flip-flop is more stable as if Q changed during a clock cycle it doesn't matter. See below.

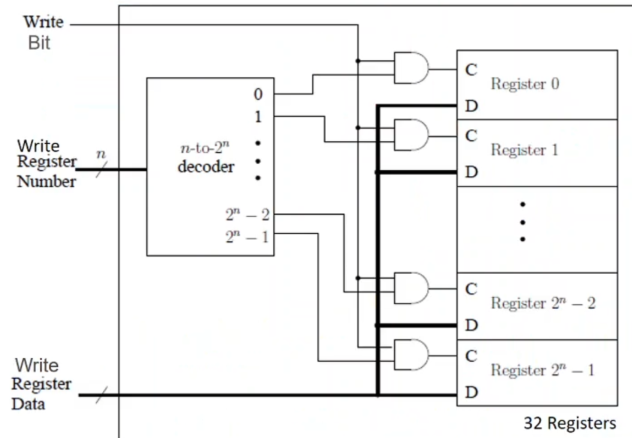




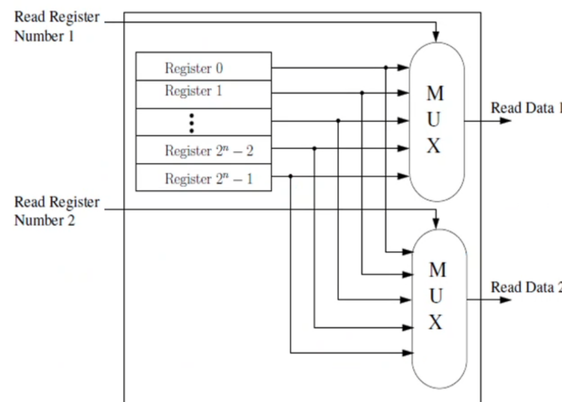
3.4 Register File

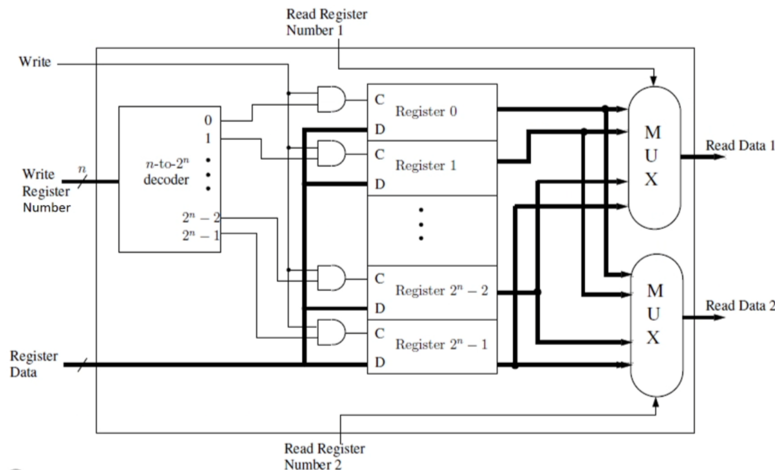
Definition. Register File: The register file is the set of all registers, implemented as D flip-flops. In particular, it has 5 inputs and 2 outputs. The first register number to read from, the second to read from, the register number to write to, the write data and the write bit. In the case of an ARM based processor, there are 32 registers, hence the register numbers are 5 bits wide. Since each register contains 64 bits, the write data is 64 bits wide. The write bit indicates whether the operation is a read (0) or write (1).

Remark. Writing to Register: Recall that in writing to a register, 3 inputs are needed. The write bit to permit writing, the write register number and the write register data. The register number is passed to a decoder which, along with the write bit, opens a register for writing. The data is then passed to each register, however, only one (the desired register) will commit anything. Recall a register is an abstraction of 64 (or more) D flip-flops.



Remark. Reading from Register: Recall that in reading from a register, 2 inputs are needed, the register read numbers. The information from each register is passed along to two multiplexors and each register read number is sent to one multiplexor. The register read number selects which register's information is allowed to pass through the multiplexor. Note that in reality this is 64 multiplexors per read number, one for each bit of data.





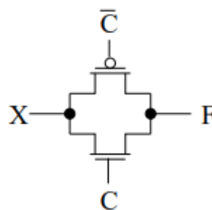
Remark: The second image is the combined register file, both read and write.

Week 4 RAM, FSM, Data Representation

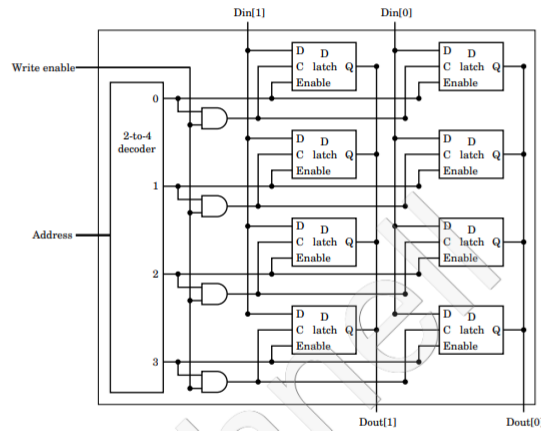
4.1 Random Access Memory

Definition. Random Access Memory: Abbreviated RAM, it is memory where any byte can be accessed randomly, i.e. not in sequential order. You provide an address and get the corresponding data.

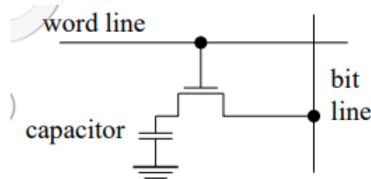
Definition. Three-State Buffer: Has 2 input values and outputs one of a 0, a 1, or a float state. It works by taking an input C and when $C = 1$ then transistors are all low allowing the other input, X to pass through to the output F. If $C = 0$ however, then X passes through two high resistances creating a float state. We usually abstract a 3 state buffer to a triangle with input at its base, output at its tip and control bit into one of the sides.



Definition. Static RAM: Abbreviated SRAM, it has 5 inputs and 1 output. There's an input for address of byte, chip select, output enable, write enable, and data in. It uses D-Latches to store memory instead of flip-flops as there is no clock. The chip select is a binary number representing which RAM chip to index. Importantly, since all chips are connected to the same output, the output number decides which chip to enable, turning each other output into a float state using three-state buffers. Otherwise it is similar to a register file, one column per bit, one row per word. The following is a 4 word 2 bit SRAM cell



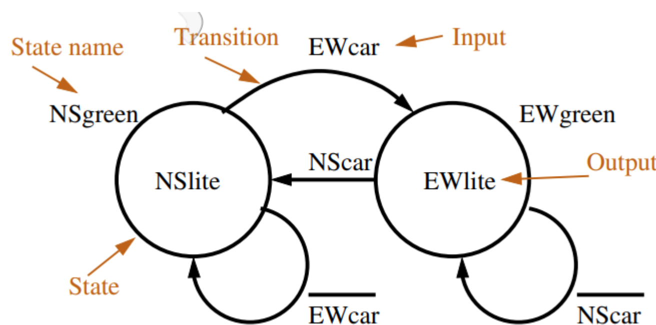
Definition. Dynamic RAM: Abbreviated DRAM, it is similar to SRAM but uses capacitors and a single transistor to store a bit. To write a bit, the word line is sent a charge and each bit in the word line is sent the new value through the bit line. The word line opens the transistor to allow the charge to leak if the bit sent is 0 or to charge it to 1 if the bit sent is 1. To read the word line is sent a 1 and the bit line is sent a value of $\frac{1}{2}$. A charge in the capacitor will slightly increase the voltage of the bit line, a lack of charge will decrease it. This change in the charge will be detected and written back. However, charges slowly leak and so they must be recharged thousands of times per second.



4.2 Finite State Machine

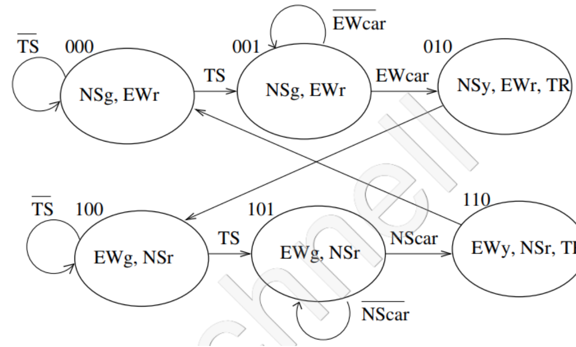
Definition. Finite State Machine: Abbreviated FSM is a circuit whose outputs depend on the state and the state depends on the state and the input. In particular, if we get a specific input, we may change our state. These are often complemented by a clock where a minimum period of time spent in one state is enforced.

Example. FSM Traffic Lights:



In this example for traffic lights, the east west lights are on so long as we are in the EWgreen state. So long as there is no north south car we stay in EWgreen, however, once there is a north south car we transition to NSgreen. So on so forth.

Example. FSM Traffic Lights + Yellow:



current state	inputs			next state	current state	inputs			next state
	NS-	EW-				NS-	EW-		
$S_2S_1S_0$	car	car	TS	$S_2'S_1'S_0'$	$S_2S_1S_0$	car	car	TS	$S_2'S_1'S_0'$
0 0 0	X	X	0	0 0 0	1 0 0	X	X	0	1 0 0
0 0 0	X	X	1	0 0 1	1 0 0	X	X	1	1 0 1
0 0 1	X	0	X	0 0 1	1 0 1	0	X	X	1 0 1
0 0 1	X	1	X	0 1 0	1 0 1	1	X	X	1 1 0
0 1 0	X	X	X	1 0 0	1 1 0	X	X	X	0 0 0
0 1 1	X	X	X	X X X	1 1 1	X	X	X	X X X

Remark: Notice the truth table used above, this is a common way to depict details of a FSM. In particular, from this table we are able to easily find minterms to generate simple combinatorial circuits.

Remark. Details of FSM: In an FSM, the state bits are held in a D Flip-Flop hooked up to the clock so as to ensure they cannot be updated until after the clock cycle has passed. These states are held inside the circuit.

4.3 Data Representation

Definition. Unsigned Binary: An unsigned binary number can represent natural numbers up to 2^k where k is the number of bits used. In particular, the binary number $d_3d_2d_1d_0$ translates to $2^0d_0 + 2^1d_1 + 2^2d_2 + d^3d_3$. Common powers to remember are that 2^{10} is $1K$ (kilobyte), 2^{20} is $1M$ (megabyte), and 2^{30} is $1G$ (gigabyte).

Definition. Signed Binary: Binary numbers used to be given a sign as the top bit (signed magnitude). I.e. if the top bit is 1 then the following number is negative but this leads to two representations of 0. We switched since to two's complement, taking the first bit as negative and the remaining bits as positive. I.e. the binary number $d_3d_2d_1d_0$ translates to $2^0d_0 + 2^1d_1 + 2^2d_2 - 2^3d_3$. This method uniquely represents each number.

Remark. Adding Unsigned Binary: Adding unsigned binary is as normal. Go column by column carrying if necessary. If the last column results in a carry, then there is an overflow, as in the number is too large to be represented in binary.

Remark. Adding Signed Binary: Adding signed (two's complement) binary is different. If both number are positive, proceed as above but note that if the sign bit changes, then an overflow occurred. If one of the numbers is negative, throw out the final carry bit no matter what it is.

Remark. Circuit for Addition: You can build a 1 bit adder with carry in by following the truth table. To perform addition on larger number, e.g. 8 bits, we use a ripple carry adder where the bits of each number are each put into a ripple adder. A carry bit of 0 is put into the lowest bit's adder and the carry out of this adder is the carry in of the next bit, so on so forth.

Definition. Bit Shifts: There are two common types of bit shifts: rotates and arithmetic shifts. A left rotate moves each bit one place to the left, moving the top most bit into the lower most bit, a right rotate is similar, going the opposite direction. A left arithmetic shift throws away the top most bit and copies a zero into the lower most bit. A right arithmetic shift copies the top bit and throws away the lower most bit.

Remark. Arithmetic Shift: Arithmetic shifts are special in that they double or halve the number they shift. A left shift doubles whereas a right shift halves and truncates. The truncate means rounding down numbers by throwing out the decimal part.

Definition. Bitwise Operations: Bitwise logical operations are taking a logical operation on each bit of two numbers. For instance we can apply the OR operation to each bit of two different binary numbers. A useful application is taking the NOR of a number with 0 inverts the number. Bitwise operations can be useful as they can be a faster way to work around a problem, especially when dealing with many booleans.

Week 5 ALU, Floating Point

5.1 ALU

Note. Multi-input XOR: An XOR with multiple inputs check for parity. It is true if there are an odd number of 1s, false otherwise (odd parity checker). It can also be applied between each bit sequentially.

Definition. Arithmetic Logic Unit: Abbreviated ALU, takes in two binary numbers A and B, operations bits, B invert bit and a carry in bit and performs a given operation (specified by operation bits) on the numbers. The ALU actually performs all the possible operations simply selects one of them to return using a MUX.

Remark: ALUs permitting multi-bit numbers actually simply ripple the operations accross multiple 1 bit ALUs. Note also that ALUs do not permit subtraction, but rather if you invert each bit of B and have a carry bit of 1 then subtraction is performed.

Definition. Indicator Bit: In ARM and other architectures, indicator bits are often used in ALUs to quickly check common properties of the result. For instance there is a bit to see if the number if all zeros, if it is negative, if there was a carry out, overflow, etc.

Remark: In a true ALU, it is often the case that the B invert along with an additional A invert bit is abstracted away into the operation bits. I.e. if there are normally 2 operation bits then you would have a total of 4 operation bits, the first corresponding to A invert, the second to B invert. This allows subtraction as shown above and the use of De Morgan's laws for NOR and NAND.

5.2 Floating Point

Definition. Binary Point: A binary point represents fractional numbers in binary. In particular, the number $d_3d_2.d_1d_0$ represents $2^1d_3 + 2^0d_2 + 2^{-1}d_1 + 2^{-2}d_0$.

Definition. Floating Point: The IEEE standard for floating points is the most common way of representing decimal numbers. A floating point is separated into 3 sections. The first bit is the sign (1 is negative, 0 is positive), the next 8 bits represent the exponent, and the last 23 represent the mantissa or significand, the decimal points.

We always assume the number has a leading one in its integer portion (i.e. exactly a 1) in scientific notation as if it doesn't we can just shift the number and its exponent until satisfied. We then consider the mantissa as the numbers after the binary point, led by a 1 in the integer portion and multiplied by $2^{exponent}$.

The exponent however is a biased binary integer as opposed to a two's complement integer. We take the positive value of the integer minus 127. This makes sorting easier.

In conclusion. If we have our sign s as a binary number, mantissa m as a binary number, and exponent e as a binary number, the floating point is represented as $(-1)^s \cdot 1.m \cdot 2^{e-127}$.

Remark: The all 0 representation is reserved for the number 0. No number's exponent can be all 0 or all 1.

Unit 3 Single Cycle Processing

Week 6 Single Cycle Processor

6.1 Datapath Overview

Note: The datapath is also occasionally called the CPU, it is the flow of information in computation.

Remark. High Level View of ARM Datapath: The datapath works in a fetch-execute cycle. It first fetches the instruction, computes $PC + 4$ and reads memory as necessary (governed by instruction) and then it executes the instruction and updates memory as necessary

(governed by instruction). This instruction may update PC (i.e. branch) and so the value of $PC + 4$ is not immediately used.

Remark. Instruction Bits: The bits of an instruction can usually be separated into specific groups. The highest order bits (variable length) of an instruction are always the opcode, specifying the instruction to be executed.

Remark. R-Format Bit Groups: Bits 31 – 21 specify the opcode, bits 20 – 16 specify the second input register (Rm), bits 15 – 10 specify the shift amount (for logical shift operations) (shamt), bits 9 – 5 specify the first input register (Rn), and bits 4 – 0 specify the register destination (Rd). I.e. `ADD Rd,Rn,Rm`. The opcode of R-format instructions contains 6 function bits which specify which specific operation (e.g. add, sub, etc.) is being performed.

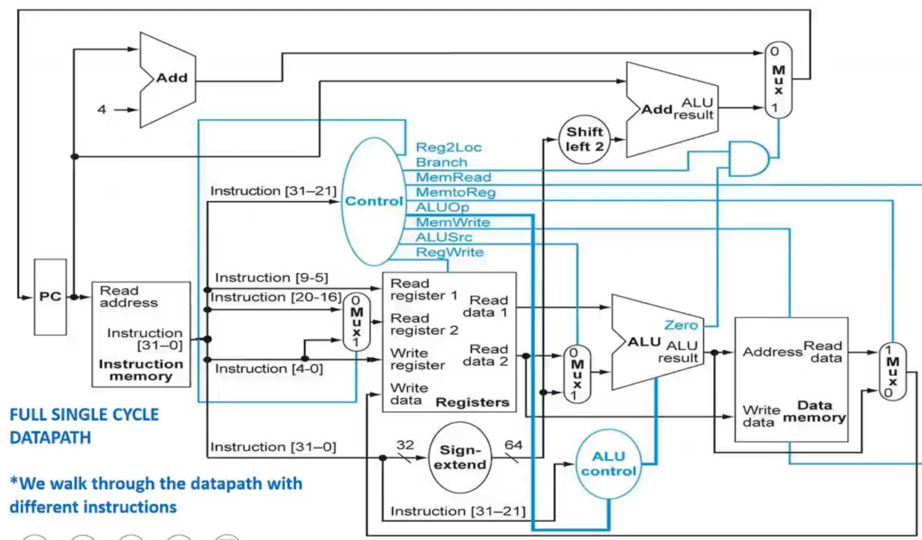
Remark. I-Format Bit Groups: Bits 31 – 22 specify the opcode, bits 21 – 10 specify the ALU Immediate (Imm), bits 9 – 5 specify the input register (Rn), and bits 4 – 0 specify the register destination (Rd). I.e. `ADDI Rd,Rn,#Imm`.

Remark. D-Format Bit Groups: Bits 31 – 21 specify the opcode, bits 20 – 12 specify the memory address offset as an immediate, bits 11 – 10 are ignored in this course, bits 9 – 5 specify the read register to which the offset is applied, bits 4 – 0 specify the write register.

Remark. B-Format Bit Groups: Bits 31 – 26 specify the opcode, bits 25 – 0 specify the branch relative address.

Remark. CB-Format Bit Groups: Bits 31 – 24 specify the opcode, bits 23 – 5 specify the branch relative address, bits 4 – 0 specify the register to check.

Remark. Control Unit: The opcode of each instruction is sent to a special ALU called the control unit. This control unit will decide which instruction to execute through the use of multiplexors and such. All instructions of the same format (e.g. R-format) start with the same specific bits. The remaining bits are different, specifying the specific instruction. The control unit generates from the opcode several control bits which are passed to ALUs, multiplexors, etc. For instance, the register 2 location, the branch value, the read/write memory if any (not register), the ALU operation, the source, etc.



Remark. ALU Control: The ALU control unit is the control unit which governs the operations in the main ALU. It receives the opcode as well 2 ALU op control bits (provided by main control unit) which dictate what is going on. In particular, 00 represents an add, 01 represents pass B, 10 represents an R-format instruction, and 11 represents a sub. In the case where the ALU op bits are 10, the ALU control will look at the 6 function bits of the instruction to determine what to do.

6.2 Specifics

Note. Control Bits:

- Reg2Loc: bit which specifies which bits of the instruction represent the second register address. In particular, the address might be saved in one of either bits $20-16$ or $4-0$ depending on the instruction.
- ALUSrc: bit which specifies if the second read register or the part of the instruction which would be used as an immediate should be used in the ALU. For instance, in I-format instructions we want the latter to occur.
- MemtoReg: bit which specifies whether the memory read (if any) or the result from the ALU should be written to the destination register (if writing occurs).
- RegWrite: bit which allows or disallows writing to the register file (write bit).
- MemRead: bit which allows or disallows reading from memory and specifies if the address passed to memory would be a read location or write destination.
- MemWrite: bit which allows or disallows writing to memory (write bit).
- Branch: bit which specifies if a branch instruction is (possibly) occurring. If set to 0, it will always execute $PC \leftarrow PC + 4$.
- ALUOp: 2 bits which are passed the ALU control unit specifying how it should act. E.g. perform R-format instruction, force addition, force subtraction, pass B.

Remark: An example of where we want to use bits $4-0$ as read address 2 is in the case of `STUR`. We use those bits to specify the register whose data is written to memory.

Remark. Sign Extend: The sign extend unit determines from the opcode of the instruction how to extend the immediate part of the instruction. For instance, if the opcode is that of an I-format instruction, it extracts the 12 relevant bits and 0 pads them so it becomes a 64 bit number. If it is D-format instructions, it extracts the 9 relevant bits and pads it with the most significant bit since it is 2's complement. Similar results hold for B-format and CB-format instructions.

Remark. Branching: When performing a branch, after sign extending the relative address, it is shifted left by 2 bits to multiply it by 4. This number is then added to PC and passed to the MUX for branching. In a conditional branch, you pass the register to check (bits 4

– 0) to the ALU and the ALU control performs a "pass B" instruction, getting the value of the register. The ALU has a zero indicator bit which is passed to the and gate before the MUX for branching. Hence if the value of the register is zero and the branch bit is one, then it will take the new PC location instead.

Remark. Timing: We use exaggerated timings for each process of datapath to ensure that each instruction has sufficient time to execute. In general, we associate reading instruction memory with 200ps (picosecond), reading register memory with 100ps, writing to register memory with 100ps, performing an ALU operation with 200ps, and reading or writing to memory with 200ps. This means R-format instructions take 600ps whereas an instruction like `LDUR` takes 800ps.

Remark. PC Update Timing: Notice the adder to computer $PC + 4$ takes 200ps, however, this is negligible since it occurs in parallel with reading from instruction memory. Notice the adder used for relative branching takes 200ps, but starts at the same time as a register read. Hence it is spread across the register and main ALU operation, making it negligible.

Week 7 Single Cycle cont.; Multicycle Datapath

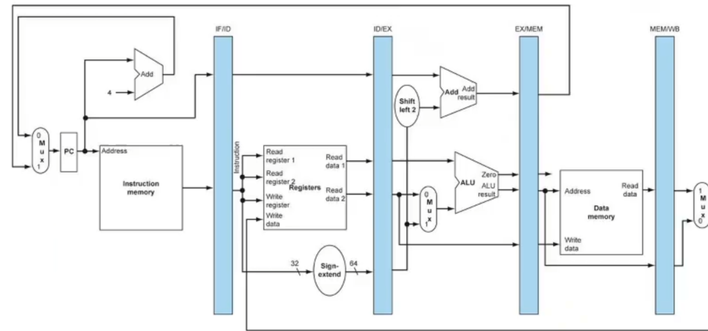
7.1 More on Single Cycle Processor

Remark. Control unit timing: The control unit takes 10ps, but produces the Reg2Loc bit which is required before doing a register read on register 2, hence it must occur before a register read, adding 10ps to all instructions which use register 2 (e.g. R-format). In particular, the bit is always required to read from register 2 which always takes place, but in the case that the information at register 2 isn't used, the data is thrown away and so we didn't wait for the read to occur.

Note. Different Types of Branching: To implement unconditional branches, simply add a new control bit to the main control unit which forces the branch to occur regardless of the value of reg 2. To implement `CBNZ`, modify the control unit to provide a CBNZ bit which is fed to a mux where the zero bit and the inverse of the zero bit are passed to it, controlled by the CBNZ bit. To implement `BREL Xn` which has the effect of $PC \leftarrow PC + 4 \times Xn$ where Xn is a register, add a new control bit for BREL and pass the register information and and opcode to a mux where BREL selects the register data to be multiplied by 4 and added to PC. Then use the previous unconditional branch bit to force select the new value.

7.2 Multicycle Processor Datapath

Note. Multicycle Datapath: This datapath is similar to a single cycle datapath, except instructions are processed in 5 steps. These 5 steps each require a single clock cycle to execute, however, these clock cycles are shorter.



Remark: Multicycle processors are actually an intermediate datapath before we implement the more powerful Pipelining.

Remark. Multicycle Registers: Between each step, memory which needs to be used in the proceeding step is stored in D flip flops. This is convenient since it will only ever write on the falling edge of a clock cycle.

Definition. Multicycle Steps:

1. Step 1, Instruction Fetch (IF). The next instruction is fetched from memory and $PC+4$ is computed.
2. Step 2, Instruction Decode (ID). The instruction is passed to the control unit, register memory is read and the sign extend is computed.
3. Step 3, instruction EXecute (EX). The instruction is executed, including computing a different value of PC and performing the main ALU step.
4. Step 4, read/write to MEMory (MEM). Any memory reading and writing instructions are committed to memory. If there is no reading or writing taking place, this is idle time. In this step the new value of PC is asserted, it either $PC \leftarrow PC + 4$ or the new value computed in the previous step as determined by a mux and control bits.
5. Step 5, Write Back (WB). In this step if it is necessary to write to the register file, the write occurs, otherwise this is idle time.

Definition. Multicycle Control: Multicycle processors use a finite state machine referred to as the multicycle control or MCC to determine which step is being executed. This determines when to allow writing to the intermediate registers and when to give components their control bits.

Remark. Multicycle Efficiency: In the implementation of a multicycle datapath, every step must be completed. Further, the execution of each step must be sequential and must be the only step be execute (i.e. no multiple instructions at a time). Since each step takes at most 200ps to execute, the clock cycle for each step is 200ps, meaning a total of 1000ps per instruction. This is slower than the 800ps (time to execute `LDUR`) clock cycle of a single cycle datapath.

However, multicycle datapaths are more efficient since steps can often be ended early. For instance, `CBZ` only needs 3 steps to execute. That is, the implementation of a multicycle datapath is more sophisticated. In particular, if an instruction only needs 3 steps to execute, then the next instruction can begin execution after 600ps, where as in a single cycle processor it would be necessary to wait for the entire 800ps clock cycle to pass.

Note. Pipelining: Pipelining is an optimized version of multicycle processing where steps are execute in parallel. This allows for multiple instructions to be computed at once.

Unit 4 Pipelining

Week 8 Pipelined Datapath

Definition. Pipeline Register: The registers between the steps in the pipeline are now referred to as pipeline registers.

Remark: The idea of pipelining is that more than one instruction can be executed at a time on a single processor. For instance, while one instruction is in the IF stage, another may be in the EX stage. However, we run into issues where for instance we try to do `ADDI X2,X31,#5` then `SUBI X3,X2,#2`. The next instruction depends on the execution (register writing) of the first.

Definition. Structural Hazard: When two or more components of the datapath have to be used by separate instructions. E.g. one instruction writes to register while another reads from it.

Definition. Data Hazard: When the result of one instruction is required by the next instruction. E.g. one instruction updates X1 and the next uses X1.

Definition. Control Hazard: When a (conditional) branch instruction changes the sequence of instructions executed.

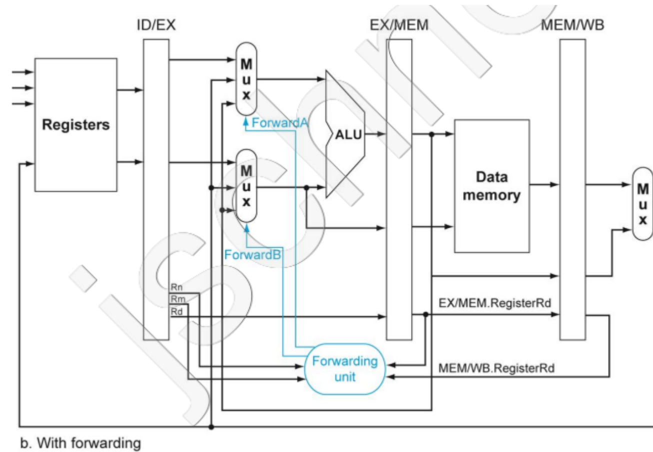
Remark. Solving Parallel Accessing: To solve this data hazard, first make it so that the register file is accessed twice in a clock cycle. The first to be written to and the second to be read from.

Remark. Solving Data Hazards (Naive): The naive approach to solving data hazards between other steps, is to have the compiler throw in `NOP` (no operation) instructions where necessary to offset the instructions enough to avoid a data hazard. Each `NOP` adds a 1 clock cycle delay.

Remark. Solving Data Hazards (Smart): The smarter approach, however, is to notice that the required data is actually available as soon as the instruction is executed. Hence, before the next instruction needs to execute the correct value is known. At this point we pass the instruction to the next execution through a process known as forwarding.

Definition. Forwarding: A method to prevent data hazards, the result of the computation in stage WB and in stage MEM are fed to multiplexors before the ALU in the EX stage.

Each result and the default values for the multiplexors are passed to both the mux for A and B in the ALU. A forwarding unit then decides which result to pick based on if the Rn and Rm values conflict with the Rd values of any of the other steps.



Remark. Problems with Forwarding: Forwarding is not a perfect solution, and still does not solve the problem with `LDUR` instructions. We can't take the output from reading memory in the MEM stage as input since we won't have a stable value. That is, we don't want to pass memory from middle of clock cycle to middle of clock cycle. Instead, we are forced to use the naive approach and place a `NOP` instruction between instructions that load and those that depend on memory.

Note. Forwarding Unit: To check if forwarding is required from MEM to EX, check if EX/MEM regWrite bit is on, and register Rd is not 31, and if EX/MEM register Rd is equal to ID/EX register Rn, in this case ForwardA=10. For ForwardB just compare against ID/EX register Rm. To check if forwarding is required from WB to EX, do the same as above, but also check that forwarding is not occurring from MEM.

Week 9 Control Hazards

9.1 Data hazards

Note. Load Use Stall: To solve the problem mentioned with `LDUR` instructions, we must use a `NOP` instruction. This method of placing a `NOP` after a `LDUR` instruction is called Load Use Stall. To know when to use it, check if reading from memory in the ID/EX stage and if the ID/EX register Rd is equal to either IF/ID's register Rn or Rm.

Remark. Run-time NOPs (Stalls): To add a NOP at run time (in particular at ID stage), zero out the current instructions control bits, effectively turning it into a NOP. Then block PC and IF/ID registers from being written to, thereby forcing the current memory to be decoded again.

9.2 Branch Hazards

Remark. CB in MEM Stage: Note when a CBZ instruction is executed, we do not know until the MEM stage whether or not to branch. This means, since determining whether to branch, 3 instructions have been started. To remove them, we must flush the pipeline.

Definition. Pipeline Flush: Flushing the pipeline converts all instructions before a certain point into NOPs. To flush instructions in ID and EX stages, it suffices to zero out their control bits. To flush an instruction in the IF stage, it is necessary to directly modify its instruction bits, e.g. into an R-Format instruction using only X31.

Remark. CB in ID Stage: while branching in MEM is the most straightforward, we can do better. Firstly, checking if the branch requirements are met (zero and branch bit) is relatively fast. It can therefore be pushed up further. Then when reading from a register, we can immediately check its bits against zero and skips the use of the ALU in EX. This means in ID stage, using the fact instruction memory reading is slow and separated into reading and writing, we can apply a CBZ. We have then only one instruction to flush.

Definition. Branch Data Hazard: The above implementation of evaluating CB in the ID stage comes with a catch. If the instruction before the CB instruction modifies the Rm value of the CB instruction (e.g. `ADDI X1, X1, #1; CBZ X1, #5`) then we can't implement data forwarding like usual. We can't forward from the EX stage to the ID stage since this would involve unstable bit (reading at end of clock cycle). Therefore, we are forced to place a NOP between any such instructions. We also then need forwarding unit / hardware from MEM to ID.

Remark. Code Rearrangement: With all the load swapping, branch flushing, etc. we want to optimize code as much as possible. Some of this work can be done by compiler by rearranging code to make code more efficient. For instance, if a NOP must be implemented between lines 100 and 104 and then 108 increments a different variable, swapping lines 104 and 108 makes the code functionally identical, but requires no additional NOP.

9.3 Branch Predicting

Definition. Branch Predicting: Branch predicting is the technique of predicting the address of a branch and fetching its instruction immediately. These come in multiple forms of varying sophistication. Below follow some techniques

Definition. Predict Branch Not Taken: Predict branch not taken is a method of prediction where $PC \leftarrow PC + 4$ each branch and we flush the instruction if the prediction was wrong. I.e. we always predict there will be no branch when fetching.

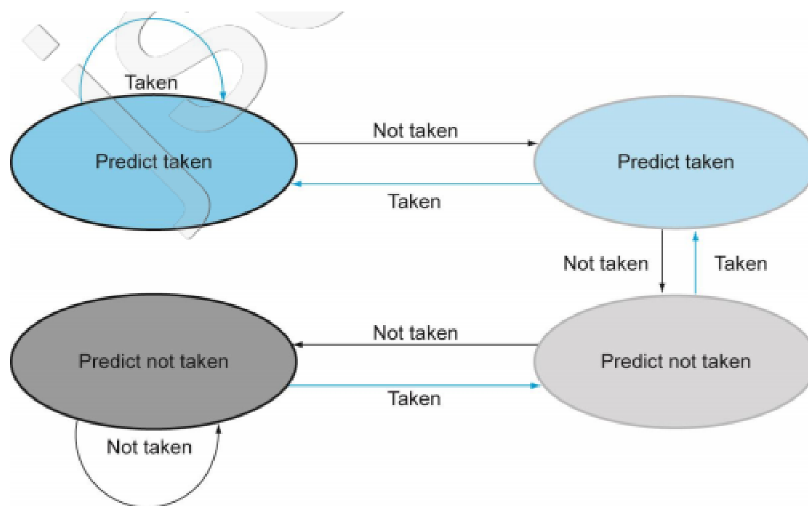
Definition. Predict Branch Taken: The opposite method to the above is implemented. Each branch PC is updated to the BTA of the branch and the next instruction at this address is loaded. If the branch was wrong, flush the instruction and rollback the location.

Definition. 1 bit Prediction: A "smarter" technique is to implement an additional predict bit. When the predict bit is on, each branch predicts taking a branch, when it is off none

predict taking a branch. When it predicts are a branch but gets a non-branch or vice-versa, the predict bit is flipped. This technique should be more consistent than the above two.

Definition. 2 bit Prediction: An even "smarter" technique is to implement two predict bits. You use these predict bits to create strong and weak predictions. Both a weak and strong prediction will cause the CPU to predict a branch and a weak and strong no predict will cause the CPU to not predict a branch.

A finite state machine is used to control the two bits. Bits 00 are strong no predict, 01 are weak no predict, 10 are weak predict, and 11 are strong predict. Taking a branch (regardless of the prediction) causes the binary number to increase by 1 and not taking a branch causes it to decrease by 1. I.e. with a strong prediction, must be wrong twice before changing prediction.



Definition. Branch Destination Address Table: The branch table is what allows our predictions to occur. It is a table of instruction lines to their branch target addresses. If the instruction being read isn't in the table, we wait for its address to be computed and then add it to the table. Otherwise, if we are predicting a branch and also have the BTA in the table, we update PC to the BTA. We can also add predict bits to each command separately, allowing more tailored predictions.

Remark: Unconditional branches require a flushed instruction the first time they run. However, in subsequent encounters they do not requiring flushing by using the branch table and prediction bits.

Unit 5 Memory

Week 10 Cache

Remark. Memory Hierarchy: There is a hierarchy of memory accessing with registers being the fastest but most expensive and secondary storage being the slowest but cheapest.

In between we have RAM which is quite fast, but still significantly slower than registers for instance. That said, RAM is relatively inexpensive, so we want another intermediate which is even faster. This comes in the form caches, which are faster but smaller than RAM. Memory is moved from the RAM to caches whereupon they can be worked with.

Definition. Direct Mapped Cache: A direct mapped cache takes memory from RAM and places it into its own higher access speed cache. It does so by taking the lower order bits of memory index and turning it into its own indices (like a hash map). When memory is queried, it is first looked in the cache (based on the lower order bits) and if the memory at that location is valid (validation bit is turned on) and the tag (higher order bits of the index) is matching, the memory is retrieved, this is a cache hit. Otherwise, it must first be read from RAM and then written to the cache, an expensive process called a cache miss.

In the case that the requested index has a different tag at the (valid) specified location, the memory must be read from RAM and written to that location. I.e. we must effectively kick out the old memory

Definition. Fully Associative Cache: A cache which works like an association list. Each index in the cache must be compared to the index requested to search for a match (plus check its valid).

Definition. Set Associative Cache: A cache which combines the above ideas. There are a small number (e.g. 4) of locations associated with the same index (like a direct mapped cache). Among these locations, we compare tags to find a match. When all the memory for an index is occupied, we kick out the least recently used memory.

Definition. Block Caching: A more sophisticated method of caching is block caching. In block caching specify a block size and we separate memory in to groups of double words (as many as the block size specifies).

When we request memory, we instead of retrieving one item retrieve the entire block of memory. This is brought to the cache, where they share the same index and tag, but are marked by their block bits. This is to implement temporal locality, hoping if you need this memory, you probably need the next memory too.

Remark. `STUR` with Caching: When modifying memory, it is first necessary to read the memory which will be modified to the cache. While this step is unimportant in a block size of 1, for all other sizes this is necessary to ensure the entire block is brought.

To ensure that main memory is updated, we implement a "dirty bit." This bit specifies that the memory in the cache location has been updated and is no longer a copy of main memory. When writing to a location with its dirty bit on, we must write back the contents of the location to main memory before writing to the location. At this point, (assuming no other dirty locations) main memory and the cache agree on memory content. Note that this write back requires an additional writing step, effectively doubling the time of the miss.

Remark: We can mix set associative and block caches quite easily to make even larger caches. This a common technique as it combines the best of both worlds.

Definition. AMAT: Much of the design around caches is built to minimize miss rates to

try and make them as fast as possible. This leads to the AMAT (average memory access time) measure. AMAT is calculated as time for a hit + miss rate $\times$ miss penalty.

Example: If we take a computer with a 100ps clock speed, 0.05 miss rate, and 2000ps miss penalty, our AMAT is $100 + 0.05 \times 2000 = 200$ps. If we decrease this miss rate to 0.01 we get an AMAT of $100 + 0.01 \times 2000 = 120$ps. Now if we make our CPU ten times faster with a 10ps clock speed, our AMATs become $10 + 0.05 \times 2000 = 110$ps and $10 + 0.05 = 30$ps respectively. So with a high miss rate, making our CPU ten times faster doesn't make our computer even twice as fast. This has real world impacts on algorithms such as quick sort for instance.

Week 11 Virtual Memory

Definition. Virtual Page: Since fetching from disk is incredibly expensive, we fetch an entire virtual page at a time. These virtual pages have a set size, called the page size, which is generally 4KB. Generally a virtual page corresponds to a portion of data of a single program.

Definition. Physical Page: A physical page is a section of RAM capable of fitting a single virtual page in its entirety. Virtual pages swap in and out of physical pages at run time, giving the illusion to a program that its memory is unlimited. When a program requests a virtual page which is not currently in a physical page (i.e. like a cache miss), we call this a page fault.

Definition. Page Table: When a program is loaded into physical memory (RAM), a page table is made for it. This table is a mapping of the program's virtual page number (VPN) to the physical page number (PPN) it occupies in physical memory. Further when the program is loaded, the processor is given a pointer to the page table so that it may access it. Now when requesting virtual page n address k, the page table will translate to physical page m address k (note address or offset doesn't change). If the page table cannot translate because the virtual page is not in memory, then we have a page fault.

Definition. Memory Address: When requesting an instruction, the processor has a specific virtual address in mind. This is a 64 bit number where bits $63 - 12$ correspond to the virtual page number and bits $11 - 00$ correspond to the offset of the instruction. This gets translated to a physical address by the page table. This is a 32 bit number where bits $31 - 12$ correspond to the physical page number and bits $11 - 00$ correspond to the same offset.

Remark: The page table contains one row per virtual page (associated with the program), each with a valid bit and possibly a number. The valid bit is 1 when the page is loaded in physical memory, in this case it has a physical page number associated with it. Otherwise the page is not in physical memory and so its physical page number is considered garbage data.

Definition. Reference Bit: In the page table, there is also a reference bit for each virtual page. This reference bit just means that the physical page has been recently used, and the OS will regularly clear all the reference bits. When a virtual page's memory is accessed, its

reference bit is turned back on. When a physical page must be overwritten due to lack of space, the system will prefer to overwrite physical pages whose reference bit is off.

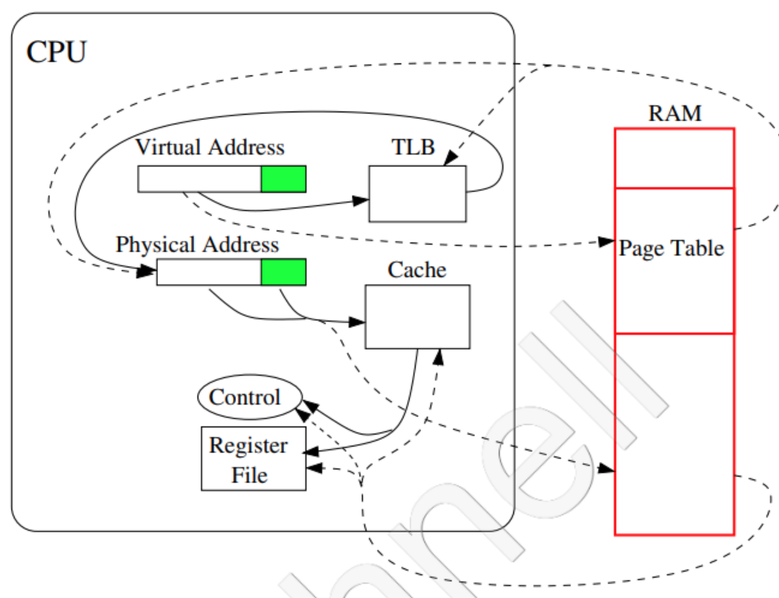
Remark: Just like a cache, the page number also keeps track of a dirty bit for each virtual page. When the dirty bit is on, this denotes that the physical page has been written to more recently then the virtual page on the disk and the it is necessary to write the entire page to the disk.

Note: For incredibly large programs, it may be necessary to treat page tables themselves as pages. E.g. if a program has 5000 pages, we may want 5 pages of page tables, each with 1000 entries.

11.1 TLB

Definition. Translation Look Aside Buffer: Abbreviated TLB, it aims to increase efficiency in the data path. Due to the fact that accessing the page table in RAM is expensive, we aim to keep a small number of the most popular pages in cache in the TLB. We then first check to translate a virtual address using the TLB and if it's not in the TLB, we check the page table. Since the TLB only points to popular pages, every page will be in physical memory, never on disk.

Remark: The TLB is fully associative and each time that there is a TLB miss, the corresponding page table entry is added to the TLB.



Week 12 Extras

Definition. Swap Space: As mentioned earlier, it is expensive to run multiple programs, so we run a single program for some number of cycles and then switch to another program. This process is called time slicing the CPU and when going from one program to another

(context switching), we save all important information of the current process to a swap space. The swap space saves the register values, program counter, and page table register (pointer). The cache and TLB can refilled when the process starts up again.

Remark: Linux saves its swap space in a dedicated partition of the disk, whereas windows has a hidden file dedicated to the swap space.