Writing an Assembler: Part 2
Writing an Assembler with Labels

• For our first assembler, we made the simplifying assumption that every line contains exactly one instruction or .word directive.
• Let's remove this assumption, and also add support for labels.
• The general format of a MIPS line is [labels] [instruction] [comment]
  • All three parts are optional and can be omitted, but must appear in order.
    • A line can be completely blank.
    • A line can have only labels, only an instruction/directive, or only a comment.
    • A line can have labels+instruction, labels+comment, or instruction+comment.
    • A line can have all three parts (in the specified order only).

  label: HELLO: meow241: beq $0, $0, meow241 ; three labels!
Extending Our Scanner

label: HELLO: meow241: beq $0, $0, meow241 ; three labels!

• This line illustrates the new syntax our scanner needs to support:
  • **Label definitions**: A sequence of alphanumeric characters (uppercase or lowercase) that *starts with an alphabet letter*, followed by a colon.
  • **Label uses**: Like a label definition, but without the colon.
  • **Comments**: A semicolon starts a comment. All characters up until the end of the line are part of the comment. *(Discarded by the scanner)*

• The changes to our DFA should be fairly straightforward...

• A minor issue: Instruction identifiers are also valid label names.
  beq: beq $0, $0, beq
Extending Our Scanner

label: HELLO: meow241: beq $0, $0, meow241 ; three labels!

- Instruction identifiers are also valid label names, so to support labels, we will extend the definition of an identifier token:
  - An **Identifier** is a sequence of alphanumerical characters (uppercase or lowercase) that starts with an alphabet letter.
  - A **Label-Def** token is an **Identifier** followed by a colon.
  - Note this means that **label uses** are represented by **Identifier** tokens.
  - For consistency, **Dot-Identifiers** will follow the same rules as **Identifier** tokens except that they begin with a dot.

- A full MIPS scanning DFA is provided as a starter file for Project 1.
Extending our Parser

• Parsing MIPS instruction lines was fairly straightforward before.
  • Look up the required syntax based on the instruction name.
  • Match tokens against the required syntax to check for errors.
  • Extract important information for the synthesis phase.

• The scanner discards comments, but we do need to worry about blank lines and label definitions during parsing.

• If a line contains zero tokens, skip it.

• Otherwise, check if the first token is a label definition.
  • If so, process all labels on the line first. Then, check for an instruction or .word, and if one exists, process that as before.
Processing Labels

• Before we can explain what "processing labels" means, we need to back up and look at the big picture.
  
  label: HELLO: meow241: beq $0, $0, meow241

• A line can have multiple label definitions before the instruction.

• Each label must have a unique name, not just within the line, but within the whole program. (No duplicate label definitions)

• Labels can be used in some instructions. For example, \texttt{beq $0, $0, meow241} uses the label called \texttt{meow241}.
  • Using a label that has no definition is an error! (No undefined label uses)
Labels and .word Directives

• Aside from beq and bne, labels can also be used as an argument to the .word directive.

• A line of the form ".word label" is equivalent to writing ".word i" where i is the memory address where the label is defined.
  • This address is computed assuming the program is loaded at address 0.

• What value does the program below place in $3?

```
lis $3
.word end
end: jr $31
```

• Answer: The address of the jr $31 instruction, which is 8 (because there are two 4-byte words preceding it).
Labels and Machine Language

• How do you encode a label in machine language?

• There is not enough space in our 32-bit instruction words to do something like encoding the name in ASCII.

• All labels need to be converted to the appropriate integer values.

```
loop: lis $7
    .word pool
    beq $0, $0, loop
pool:  jr $31
```

• The program on the left and the program on the right get translated into the same machine code.
Summary of Label Problems

• We need to keep track of which labels have been defined.
• One reason is to check for duplicate label definition errors.
• Another is to check for undefined label use errors.
• We also need to keep track of where each label was defined to compute the corresponding numeric values for encoding.
• More precisely, assuming the program is loaded at address 0, for each label, we want to keep track of the memory address of the corresponding location in the program.
• We will create a symbol table that maps label names to addresses.
Symbol Table Example

main:  lis $2
      .word beyond
      lis $1
      .word 2
; Hello, I'm a comment
      add $3, $0, $0

top:  begin: ; Two labels on the same line
      add $3, $3, $2
      sub $2, $2, $1
      bne $2, $0, top ; Go to top
      jr $31
beyond: ; Label after last instruction
Symbol Table Example

main: lis $2
    .word beyond
lis $1
    .word 2
; Hello, I'm a comment
add $3, $0, $0

top: begin: ; Two labels on the same line
    add $3, $3, $2
    sub $2, $2, $1
    bne $2, $0, top ; Go to top
    jr $31

beyond: ; Label after last instruction

Label definitions highlighted.
# Symbol Table Example

<table>
<thead>
<tr>
<th>Address</th>
<th>Label</th>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>main:</td>
<td><code>lis $2</code></td>
<td></td>
</tr>
<tr>
<td>0x04</td>
<td></td>
<td><code>.word beyond</code></td>
<td></td>
</tr>
<tr>
<td>0x08</td>
<td></td>
<td><code>lis $1</code></td>
<td></td>
</tr>
<tr>
<td>0x0c</td>
<td></td>
<td><code>.word 2</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>;</td>
<td><code>Hello, I'm a comment</code></td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td></td>
<td><code>add $3, $0, $0</code></td>
<td></td>
</tr>
</tbody>
</table>

Each line with an **instruction or** `.word` takes up 4 bytes (32 bits) in memory once assembled. *Other lines do not take up space in memory!!*

<table>
<thead>
<tr>
<th>Address</th>
<th>Label</th>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x14</td>
<td>top: begin:</td>
<td><code>add $3, $3, $2</code></td>
<td>; Two labels on the same line</td>
</tr>
<tr>
<td>0x18</td>
<td></td>
<td><code>sub $2, $2, $1</code></td>
<td></td>
</tr>
<tr>
<td>0x1c</td>
<td></td>
<td><code>bne $2, $0, top</code></td>
<td>; Go to top</td>
</tr>
<tr>
<td>0x20</td>
<td>beyond:</td>
<td><code>jr $31</code></td>
<td>; Label after last instruction</td>
</tr>
</tbody>
</table>
Symbol Table Example

Two views of label addresses:

- It's the address where the next instruction after the label is (or would be) located.
- Count the number of instruction lines before the definition and multiply by 4.

```
0x00    main:   lis $2
0x04    .word  beyond
0x08    lis $1
0x0c    .word  2
         ; Hello, I'm a comment
0x10    add $3, $0, $0

top:    begin: ; Two labels on the same line
0x14    add $3, $3, $2
0x18    sub $2, $2, $1
0x1c    bne $2, $0, top ; Go to top
0x20    jr $31

beyond: ; Label after last instruction
```
Symbol Table Example

<table>
<thead>
<tr>
<th>Address</th>
<th>Label</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>main:</td>
<td>lis $2</td>
</tr>
<tr>
<td>0x04</td>
<td>.word</td>
<td>beyond</td>
</tr>
<tr>
<td>0x08</td>
<td>lis</td>
<td>$1</td>
</tr>
<tr>
<td>0x0c</td>
<td>.word</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>; Hello, I'm a comment</td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td>add</td>
<td>$3, $0, $0</td>
</tr>
<tr>
<td>0x14</td>
<td>top:</td>
<td>begin:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>; Two labels on the same line</td>
</tr>
<tr>
<td>0x18</td>
<td>add</td>
<td>$3, $3, $2</td>
</tr>
<tr>
<td>0x1c</td>
<td>sub</td>
<td>$2, $2, $1</td>
</tr>
<tr>
<td>0x20</td>
<td>bne</td>
<td>$2, $0, top ; Go to top</td>
</tr>
<tr>
<td></td>
<td>jr</td>
<td>$31</td>
</tr>
<tr>
<td>0x24</td>
<td>beyond:</td>
<td>; Label after last instruction</td>
</tr>
</tbody>
</table>

Two views of label addresses:
- It's the address where the next instruction after the label is (or would be) located.
- Count the number of instruction lines before the definition and multiply by 4.

<table>
<thead>
<tr>
<th>Label</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>0x00 (0 x 4 = 0)</td>
</tr>
<tr>
<td>top</td>
<td>0x14 (5 x 4 = 20)</td>
</tr>
<tr>
<td>begin</td>
<td>0x14 (5 x 4 = 20)</td>
</tr>
<tr>
<td>beyond</td>
<td>0x24 (9 x 4 = 36)</td>
</tr>
</tbody>
</table>
Symbol Table Implementation

• Keep track of a counter which you start at 0 and increment by 4 each time you encounter a line with an instruction or .word.
  • You can think of this as where PC will be when the instruction executes.

• When you encounter a label definition during parsing, add the pair (label name, PC counter) to the symbol table.

• Check first if the pair already exists in the table. If so, this is a duplicate label definition error.

• Although you catch this error while parsing lines, it is an example of a semantic error (an error in meaning rather than syntax).
Symbol Table Data Structures

- Use a data structure with efficient lookups.
- A vector or list of pairs will be slow.
  - Lookups are linear time, so the performance could be quadratic in the number of labels.
- A map (using a hash table or binary tree) is a good choice.
  - In C++, std::map (probably) uses a binary tree, and std::unordered_map uses a hash table.
  - In Racket, you can use (hash ...) or (make-hash ...)
Replacing Labels

- Assembling ".word label" is equivalent to assembling ".word i", where i is the location of the label.
- Assembling "beq $0, $0, label" is trickier because we need to convert the label to an offset from PC.
- Let's think about what happens when we branch.
  - Desired effect: PC is set to the address of the label (PC = address).
  - Branching does PC += offset * 4, so we want: PC + (offset * 4) = address.
  - Basic algebra gives the formula: offset = (address − PC) / 4.
  - Conveniently, when we read the branch instruction, the counter we use to determine label addresses will contain the required value of PC!
Synthesis with Labels

• Synthesis works just as before once labels have been translated into the appropriate numeric values.

• There are two issues:
  • What if we try to assemble an instruction that uses a label, and the label turns out to be undefined? (See the following slides.)
  • When assembling a branch instruction, what if the formula \((\text{address} - \text{PC}) / 4\) gives a value that is outside the 16-bit two's complement range?

• The out-of-range issue requires a rather large program to occur, but it is possible! You need to check for this and report an error.
The Forward Reference Problem

• We still haven't seen how to deal with this very common situation:
  
  \[
  \text{bne } $1, $2, \text{ notEqual} \\
  \text{...} \\
  \text{notEqual:}
  \]

• Here the label definition appears \textbf{after} the branch, so it's not in our symbol table when we first encounter the branch instruction.

• In C/C++ this is solved by requiring "forward declarations" of symbols that are used before their definition.

• But MIPS assembly allows labels to be freely defined anywhere.
Two-Pass Assembly

• A simple way of dealing with forward references, i.e., references to names that are not yet defined, is to do two passes over the source.

• **Pass 1:** Build a complete symbol table.

• **Pass 2:** Resolve label references.
  • In Pass 1, parse the instructions and store the parsed lines in a data structure. Pass 2 is over the contents of this data structure.

• The two passes are *not* an "analysis pass" and a "synthesis pass".
  • Duplicate label are checked in Pass 1 while the symbol table is being built.
  • Undefined label errors and out-of-range branch offset errors must be checked in Pass 2 since this requires knowing label addresses.
Dealing with Labels: Summary

• Implement the assembler in **two passes**.
• The first pass builds a **symbol table** containing a mapping of label names to label locations.
• The second pass uses this table for error checking and synthesis.
• Duplicate label errors are caught in the first pass, undefined label errors and branch offset range errors are caught in the second pass.
• Analysis can otherwise be split across both passes as you see fit.
• For `.word` with labels, the label is replaced with its value in the symbol table. For branching with labels, use a formula to compute the offset.
Assembler Complete!

• We've covered enough to write a complete MIPS assembler that supports labels.
• In Project 2, you will implement such an assembler yourself.
• Now that we can write complex programs more conveniently, we'll dive even deeper into MIPS programming.
• We'll first discuss arrays in more detail, then learn how to implement procedures (not in this lecture).
• We will see how to manage the associated memory by using part of RAM as a stack.
• You will see how recursion actually works at a low level!
MIPS Assembly Language Programming: Part 2
Arrays Revisited

• An array like "int A[3] = {1,2,3};" would be stored as follows in memory.

• To access element A[i], you need to multiply i by 4 and add it to the starting address of the array A.

• The Load Word (lw) and Store Word (sw) instructions read or write entire words.

<table>
<thead>
<tr>
<th>A[0]</th>
<th>MEM[address of A]</th>
<th>00000000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MEM[address of A + 1]</td>
<td>00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 2]</td>
<td>00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 3]</td>
<td>00000001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MEM[address of A + 5]</td>
<td>00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 6]</td>
<td>00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 7]</td>
<td>00000010</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MEM[address of A + 9]</td>
<td>00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 10]</td>
<td>00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 11]</td>
<td>00000011</td>
</tr>
</tbody>
</table>
Tracing an Array Loop

• The following program sets all elements of an array to 0.

```assembly
; $1 contains the starting address of A
; $2 contains the size of A
lis $11
.word 1
lis $4
.word 4
add $5, $0, $0 ; $5 holds i

PC for:
  slt $6, $5, $2 ; $6 is 1 if i < n, 0 otherwise
  beq $6, $0, end ; Go to end if i < n is false
  mult $5, $4
  mflo $6 ; $6 = i * 4
  add $6, $1, $6 ; $6 = address of A[i]
  sw $0, 0($6) ; A[i] = 0
  add $5, $5, $11 ; i += 1
  beq $0, $0, for ; back to top of loop

end: jr $31
```

<table>
<thead>
<tr>
<th></th>
<th>MEM[address of A]</th>
<th>MEM[address of A + 1]</th>
<th>MEM[address of A + 2]</th>
<th>MEM[address of A + 3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0]</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>000000001</td>
</tr>
<tr>
<td>A[1]</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>000000000</td>
</tr>
</tbody>
</table>

$1 = address of A[0]   $2 = 3
$6 = ?                 $5 = 0
Tracing an Array Loop

• The following program sets all elements of an array to 0.

```asm
; $1 contains the starting address of A
; $2 contains the size of A
lis $11
.word 1
lis $4
.word 4
add $5, $0, $0 ; $5 holds i
for:  slt $6, $5, $2 ; $6 is 1 if i < n, 0 otherwise
     beq $6, $0, end ; Go to end if i < n is false
     mult $5, $4
     mflo $6 ; $6 = i * 4
     add $6, $1, $6 ; $6 = address of A[i]
     sw $0, 0($6) ; A[i] = 0
     add $5, $5, $11 ; i += 1
     beq $0, $0, for ; back to top of loop
end:  jr $31
```

<table>
<thead>
<tr>
<th>Example Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0]</td>
<td></td>
</tr>
<tr>
<td>A[1]</td>
<td></td>
</tr>
<tr>
<td>A[2]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$1 = address of A[0]</th>
<th>$2 = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$6 = 1</td>
<td>$5 = 0</td>
</tr>
</tbody>
</table>

$1 = address of A[0]  $2 = 3
$6 = 1  $5 = 0
Tracing an Array Loop

• The following program sets all elements of an array to 0.

```assembly
lis $11
.word 1
lis $4
.word 4
add $5, $0, $0 ; $5 holds i
for: slt $6, $5, $2 ; $6 is 1 if i < n, 0 otherwise
    beq $6, $0, end ; Go to end if i < n is false
    mult $5, $4
    mflo $6  ; $6 = i * 4
    add $6, $1, $6 ; $6 = address of A[i]
    sw $0, 0($6) ; A[i] = 0
    add $5, $5, $11 ; i += 1
    beq $0, $0, for ; back to top of loop
end: jr $31
```

<table>
<thead>
<tr>
<th>A[0]</th>
<th>MEM[address of A]</th>
<th>00000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM[address of A + 1]</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>MEM[address of A + 2]</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>MEM[address of A + 3]</td>
<td>00000001</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM[address of A + 5]</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>MEM[address of A + 6]</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>MEM[address of A + 7]</td>
<td>00000010</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM[address of A + 9]</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>MEM[address of A + 10]</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>MEM[address of A + 11]</td>
<td>00000011</td>
<td></td>
</tr>
</tbody>
</table>

$1 = address of A[0]  $2 = 3
$6 = 1  $5 = 0
Tracing an Array Loop

• The following program sets all elements of an array to 0.

; $1$ contains the starting address of $A$
; $2$ contains the size of $A$
lis $11$
.word 1
lis $4$
.word 4
add $5$, $0$, $0$ ; $5$ holds $i$
for:  slt $6$, $5$, $2$ ; $6$ is 1 if $i < n$, 0 otherwise
    beq $6$, $0$, end ; Go to end if $i < n$ is false
    mult $5$, $4$
    add $6$, $0$, $6$ ; $6$ = address of $A[i]$
    sw $0$, 0($6) ; A[i] = 0
    add $5$, $5$, $11$ ; $i += 1$
    beq $0$, $0$, for ; back to top of loop
end:  jr $31$

$1 = \text{address of } A[0]$  $2 = 3$
$6 = 1$  $lo = 0$
$5 = 0$

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0]</td>
<td>00000000</td>
</tr>
<tr>
<td>A[0] + 1</td>
<td>00000000</td>
</tr>
<tr>
<td>A[0] + 2</td>
<td>00000000</td>
</tr>
<tr>
<td>A[0] + 3</td>
<td>00000001</td>
</tr>
<tr>
<td>A[1]</td>
<td>00000000</td>
</tr>
<tr>
<td>A[1] + 1</td>
<td>00000000</td>
</tr>
<tr>
<td>A[1] + 2</td>
<td>00000000</td>
</tr>
<tr>
<td>A[1] + 3</td>
<td>00000010</td>
</tr>
</tbody>
</table>

A[0] = MEM[address of A]
A[0] + 1 = MEM[address of A + 1]
A[0] + 2 = MEM[address of A + 2]
A[0] + 3 = MEM[address of A + 3]
A[1] + 1 = MEM[address of A + 5]
Tracing an Array Loop

• The following program sets all elements of an array to 0.

```assembly
; $1 contains the starting address of A
; $2 contains the size of A
lis $11
.word 1
lis $4
.word 4
add $5, $0, $0 ; $5 holds i
for: slt $6, $5, $2 ; $6 is 1 if i < n, 0 otherwise
   beq $6, $0, end ; Go to end if i < n is false
   mult $5, $4
   mflo $6 ; $6 = i * 4
   add $6, $1, $6 ; $6 = address of A[i]
   sw $0, 0($6) ; A[i] = 0
   add $5, $5, $11 ; i += 1
   beq $0, $0, for ; back to top of loop
end: jr $31
```

<table>
<thead>
<tr>
<th></th>
<th>$1 = address of A[0]</th>
<th>$2 = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$6 = 0</td>
<td>$5 = 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM[address of A]</td>
<td>00000000</td>
</tr>
<tr>
<td>MEM[address of A + 1]</td>
<td>00000000</td>
</tr>
<tr>
<td>MEM[address of A + 2]</td>
<td>00000000</td>
</tr>
<tr>
<td>MEM[address of A + 3]</td>
<td>00000001</td>
</tr>
<tr>
<td>MEM[address of A + 4]</td>
<td>00000000</td>
</tr>
<tr>
<td>MEM[address of A + 5]</td>
<td>00000000</td>
</tr>
<tr>
<td>MEM[address of A + 6]</td>
<td>00000000</td>
</tr>
<tr>
<td>MEM[address of A + 7]</td>
<td>00000010</td>
</tr>
<tr>
<td>MEM[address of A + 8]</td>
<td>00000000</td>
</tr>
<tr>
<td>MEM[address of A + 9]</td>
<td>00000000</td>
</tr>
<tr>
<td>MEM[address of A + 10]</td>
<td>00000000</td>
</tr>
<tr>
<td>MEM[address of A + 11]</td>
<td>00000011</td>
</tr>
</tbody>
</table>
Tracing an Array Loop

- The following program sets all elements of an array to 0.

```
; $1 contains the starting address of A
; $2 contains the size of A
lis $11
.word 1
lis $4
.word 4
add $5, $0, $0; $5 holds i
for:  slt $6, $5, $2; $6 is 1 if i < n, 0 otherwise
   beq $6, $0, end; Go to end if i < n is false
   mult $5, $4
   mflo $6; $6 = i * 4
   add $6, $1, $6; $6 = address of A[i]
   sw $0, 0($6); A[i] = 0
   add $5, $5, $11; i += 1
   beq $0, $0, for; back to top of loop
end:   jr $31
```

<table>
<thead>
<tr>
<th></th>
<th>MEM[address of A]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0]</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>A[1]</td>
<td>00000000</td>
<td></td>
</tr>
</tbody>
</table>

$1 = address of A[0]  $2 = 3
$6 = address of A[0]  $5 = 0
• The following program sets all elements of an array to 0.

```assembly
; $1 contains the starting address of A
; $2 contains the size of A
lis $11
.word 1
lis $4
.word 4
add $5, $0, $0 ; $5 holds i
for:  slt $6, $5, $2 ; $6 is 1 if i < n, 0 otherwise
    beq $6, $0, end ; Go to end if i < n is false
    mult $5, $4
    mflo $6 ; $6 = i * 4
    add $6, $1, $6 ; $6 = address of A[i]
    sw $0, 0($6) ; A[i] = 0
A[0]  MEM[address of A]   00000000
      MEM[address of A + 1]  00000000
      MEM[address of A + 2]  00000000
      MEM[address of A + 3]  00000000
PC    add $5, $5, $11 ; i += 1
    beq $0, $0, for ; back to top of loop
end:  jr $31

$1 = address of A[0]  $2 = 3
$6 = address of A[0]  $5 = 0
```
Tracing an Array Loop

• The following program sets all elements of an array to 0.

```
; $1 contains the starting address of A
; $2 contains the size of A
lis $11
.word 1
lis $4
.word 4
add $5, $0, $0 ; $5 holds i
for: slt $6, $5, $2 ; $6 is 1 if i < n, 0 otherwise
    beq $6, $0, end ; Go to end if i < n is false
    mult $5, $4
    mflo $6 ; $6 = i * 4
    add $6, $1, $6 ; $6 = address of A[i]
    sw $0, 0($6) ; A[i] = 0
    add $5, $5, $11 ; i += 1
PC beq $0, $0, for ; back to top of loop
end: jr $31
```

<table>
<thead>
<tr>
<th></th>
<th>MEM[address of A]</th>
<th>MEM[address of A + 1]</th>
<th>MEM[address of A + 2]</th>
<th>MEM[address of A + 3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0]</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>A[1]</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000010</td>
</tr>
</tbody>
</table>

$1 = address of A[0]  $2 = 3  $6 = address of A[0]  $5 = 1
Tracing an Array Loop

• The following program sets all elements of an array to 0.

```assembly
; $1 contains the starting address of A
; $2 contains the size of A
lis $11
.word 1
lis $4
.word 4
add $5, $0, $0 ; $5 holds i
PC for:
slt $6, $5, $2 ; $6 is 1 if i < n, 0 otherwise
beq $6, $0, end ; Go to end if i < n is false
mult $5, $4
mflo $6 ; $6 = i * 4
add $6, $1, $6 ; $6 = address of A[i]
sw $0, 0($6) ; A[i] = 0
add $5, $5, $11 ; i += 1
beq $0, $0, for ; back to top of loop
end: jr $31
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0]</td>
<td>00000000</td>
</tr>
<tr>
<td>A[1]</td>
<td>00000000</td>
</tr>
</tbody>
</table>

- $1 = address of A[0]  
- $2 = 3  
- $6 = address of A[0]  
- $5 = 1
Tracing an Array Loop

• The following program sets all elements of an array to 0.

```assembly
; $1 contains the starting address of A
; $2 contains the size of A
lis $11
.word 1
lis $4
.word 4
add $5, $0, $0 ; $5 holds i
for:  slt $6, $5, $2 ; $6 is 1 if i < n, 0 otherwise
      beq $6, $0, end ; Go to end if i < n is false
      mult $5, $4
      mflo $6 ; $6 = i * 4
      add $6, $1, $6 ; $6 = address of A[i]
      sw $0, 0($6) ; A[i] = 0
      add $5, $5, $11 ; i += 1
      beq $0, $0, for ; back to top of loop
end:  jr $31
```

<table>
<thead>
<tr>
<th>MEM[address of A]</th>
<th>00000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM[address of A + 1]</td>
<td>00000000</td>
</tr>
<tr>
<td>MEM[address of A + 2]</td>
<td>00000000</td>
</tr>
<tr>
<td>MEM[address of A + 3]</td>
<td>00000000</td>
</tr>
<tr>
<td>MEM[address of A + 4]</td>
<td>00000000</td>
</tr>
<tr>
<td>MEM[address of A + 5]</td>
<td>00000000</td>
</tr>
<tr>
<td>MEM[address of A + 6]</td>
<td>00000000</td>
</tr>
<tr>
<td>MEM[address of A + 7]</td>
<td>00000010</td>
</tr>
<tr>
<td>MEM[address of A + 8]</td>
<td>00000000</td>
</tr>
<tr>
<td>MEM[address of A + 9]</td>
<td>00000000</td>
</tr>
<tr>
<td>MEM[address of A + 10]</td>
<td>00000000</td>
</tr>
<tr>
<td>MEM[address of A + 11]</td>
<td>00000011</td>
</tr>
</tbody>
</table>

$1 = address of A[0]  $2 = 3
$6 = 1  $5 = 1
Tracing an Array Loop

• The following program sets all elements of an array to 0.

; $1$ contains the starting address of $A$
; $2$ contains the size of $A$
lis $11$
.word 1
lis $4$
.word 4
add $5$, $0$, $0$ ; $5$ holds $i$
for:  slt $6$, $5$, $2$ ; $6$ is 1 if $i < n$, 0 otherwise
    beq $6$, $0$, end ; Go to end if $i < n$ is false
    mult $5$, $4$ ; $6 = i \times 4$
    mflo $6$
    add $6$, $1$, $6$ ; $6$ = address of $A[i]$
    sw $0$, 0($6$) ; $A[i] = 0$
    add $5$, $5$, $11$ ; $i += 1$
    beq $0$, $0$, for ; back to top of loop
end: jr $31$

$1 = \text{address of } A[0]$  $2 = 3$
$6 = 1$  $5 = 1$
Tracing an Array Loop

• The following program sets all elements of an array to 0.

; $1$ contains the starting address of $A$
; $2$ contains the size of $A$
lis $11$
.word 1
lis $4$
.word 4
add $5$, $0$, $0$ ; $5$ holds $i$
for: slt $6$, $5$, $2$ ; $6$ is 1 if $i < n$, 0 otherwise
   beq $6$, $0$, end ; Go to end if $i < n$ is false
   mult $5$, $4$
   add $6$, $1$, $6$ ; $6$ = address of $A[i]$
   sw $0$, 0($6) ; A[i] = 0
   add $5$, $5$, $11$ ; $i$ += 1
   beq $0$, $0$, for ; back to top of loop
end: jr $31$

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A[0]$</td>
<td>00000000</td>
</tr>
<tr>
<td>$A[1]$</td>
<td>00000000</td>
</tr>
</tbody>
</table>

$1 = \text{address of } A[0]$  $2 = 3$
$6 = 1$  $lo = 4$  $5 = 1$
Tracing an Array Loop

• The following program sets all elements of an array to 0.

```
; $1 contains the starting address of A
; $2 contains the size of A
lis $11
.word 1
lis $4
.word 4
add $5, $0, $0 ; $5 holds i
for: slt $6, $5, $2 ; $6 is 1 if i < n, 0 otherwise
    beq $6, $0, end ; Go to end if i < n is false
    mult $5, $4
    mflo $6 ; $6 = i * 4
    add $6, $1, $6 ; $6 = address of A[i]
    sw $0, 0($6) ; A[i] = 0
    add $5, $5, $11 ; i += 1
    beq $0, $0, for ; back to top of loop
end: jr $31
```

<table>
<thead>
<tr>
<th>$1 = address of A[0]</th>
<th>$2 = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$6 = 4</td>
<td>$5 = 1</td>
</tr>
</tbody>
</table>
Tracing an Array Loop

- The following program sets all elements of an array to 0.

```assembly
; $1 contains the starting address of A
; $2 contains the size of A
lis $11
.word 1
lis $4
.word 4
add $5, $0, $0 ; $5 holds i
for: slt $6, $5, $2 ; $6 is 1 if i < n, 0 otherwise
beq $6, $0, end ; Go to end if i < n is false
mult $5, $4
mflo $6 ; $6 = i * 4
add $6, $1, $6 ; $6 = address of A[i]
sw $0, 0($6) ; A[i] = 0
add $5, $5, $11 ; i += 1
beq $0, $0, for ; back to top of loop
end: jr $31
```

**Table:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Memory Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0]</td>
<td>MEM[address of A]</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 1]</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 2]</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 3]</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 4]</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 5]</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 6]</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 7]</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 8]</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 9]</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 10]</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 11]</td>
</tr>
</tbody>
</table>

$1 = \text{address of A[0]}$    $2 = 3$
$6 = \text{address of A[1]}$    $5 = 1$
Tracing an Array Loop

The following program sets all elements of an array to 0.

```assembly
; $1 contains the starting address of A
; $2 contains the size of A
lis $11
.word 1
lis $4
.word 4
add $5, $0, $0 ; $5 holds i
for: slt $6, $5, $2 ; $6 is 1 if i < n, 0 otherwise
    beq $6, $0, end ; Go to end if i < n is false
    mult $5, $4
    mflo $6 ; $6 = i * 4
    add $6, $1, $6 ; $6 = address of A[i]
    sw $0, 0($6) ; A[i] = 0
    PC add $5, $5, $11 ; i += 1
    beq $0, $0, for ; back to top of loop
end: jr $31
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0]</td>
<td>00000000</td>
</tr>
<tr>
<td>A[1]</td>
<td>00000000</td>
</tr>
</tbody>
</table>

|$1 = \text{address of } A[0]$ | $2 = 3$
|$6 = \text{address of } A[1]$ | $5 = 1$
Tracing an Array Loop

• The following program sets all elements of an array to 0.

```assembly
; $1 contains the starting address of A
; $2 contains the size of A
lis $11
.word 1
lis $4
.word 4
add $5, $0, $0 ; $5 holds i
for: slt $6, $5, $2 ; $6 is 1 if i < n, 0 otherwise
    beq $6, $0, end ; Go to end if i < n is false
    mult $5, $4
    mflo $6 ; $6 = i * 4
    add $6, $1, $6 ; $6 = address of A[i]
    sw $0, 0($6) ; A[i] = 0
    add $5, $5, $11 ; i += 1
PC
    beq $0, $0, for ; back to top of loop
end: jr $31
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 = address of A[0]</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000011</td>
</tr>
<tr>
<td>$2 = 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$6 = address of A[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$5 = 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Tracing an Array Loop

• The following program sets all elements of an array to 0.

```
; $1 contains the starting address of A
; $2 contains the size of A
lis $11
.word 1
lis $4
.word 4
add $5, $0, $0 ; $5 holds i
PC for:

slt $6, $5, $2 ; $6 is 1 if i < n, 0 otherwise
beq $6, $0, end ; Go to end if i < n is false
mul $5, $4
mflo $6 ; $6 = i * 4
add $6, $1, $6 ; $6 = address of A[i]
sw $0, 0($6) ; A[i] = 0
add $5, $5, $11 ; i += 1
beq $0, $0, for ; back to top of loop
end: jr $31
```
Tracing an Array Loop

• The following program sets all elements of an array to 0.

; $1$ contains the starting address of $A$
; $2$ contains the size of $A$
lis $11$
.word 1
lis $4$
.word 4
add $5$, $0$, $0$ ; $5$ holds $i$
for: slt $6$, $5$, $2$ ; $6$ is 1 if $i < n$, 0 otherwise
  beq $6$, $0$, end ; Go to end if $i < n$ is false
  mult $5$, $4$
  mflo $6$ ; $6 = i \times 4$
  add $6$, $1$, $6$ ; $6 = \text{address of } A[i]$
  sw $0$, 0($6) ; A[i] = 0
  add $5$, $5$, $11$ ; $i += 1$
  beq $0$, $0$, for ; back to top of loop
end: jr $31$

<table>
<thead>
<tr>
<th>$A[0]$</th>
<th>MEM[address of A]</th>
<th>00000000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MEM[address of A + 1]</td>
<td>00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 2]</td>
<td>00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 3]</td>
<td>00000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MEM[address of A + 5]</td>
<td>00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 6]</td>
<td>00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 7]</td>
<td>00000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MEM[address of A + 9]</td>
<td>00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 10]</td>
<td>00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 11]</td>
<td>00000011</td>
</tr>
</tbody>
</table>

$1 = \text{address of } A[0]$  $2 = 3$
$6 = 1$  $5 = 2$
Tracing an Array Loop

• The following program sets all elements of an array to 0.

```assembly
; $1 contains the starting address of A
; $2 contains the size of A
lis $11
.word 1
lis $4
.word 4
add $5, $0, $0 ; $5 holds i
for: slt $6, $5, $2 ; $6 is 1 if i < n, 0 otherwise
    beq $6, $0, end ; Go to end if i < n is false
    mult $5, $4
    mflo $6 ; $6 = i * 4
    add $6, $1, $6 ; $6 = address of A[i]
    sw $0, 0($6) ; A[i] = 0
    add $5, $5, $11 ; i += 1
    beq $0, $0, for ; back to top of loop
end: jr $31
```

<table>
<thead>
<tr>
<th>PC</th>
<th>A[0]</th>
<th>MEM[address of A]</th>
<th>00000000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MEM[address of A + 1]</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 2]</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 3]</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 4]</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 5]</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 6]</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 7]</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 8]</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 9]</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 10]</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 11]</td>
<td>00000111</td>
<td></td>
</tr>
</tbody>
</table>

$1 = address of A[0]   $2 = 3
$6 = 1                  $5 = 2
Tracing an Array Loop

• The following program sets all elements of an array to 0.

```assembly
; $1 contains the starting address of A
; $2 contains the size of A
lis $11
.word 1
lis $4
.word 4
add $5, $0, $0 ; $5 holds i
for:  slt $6, $5, $2 ; $6 is 1 if i < n, 0 otherwise
    beq $6, $0, end ; Go to end if i < n is false
    mult $5, $4
    add $6, $0, $6 ; $6 = address of A[i]
    sw $0, 0($6) ; A[i] = 0
    add $5, $5, $11 ; i += 1
    beq $0, $0, for ; back to top of loop
end:  jr $31
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0]</td>
<td>MEM[address of A]</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 1]</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 2]</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 3]</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 5]</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 6]</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 7]</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 8]</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 9]</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 10]</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 11]</td>
</tr>
<tr>
<td>$1 = address of A[0]</td>
<td>$2 = 3</td>
</tr>
<tr>
<td>$6 = 1</td>
<td>lo = 8</td>
</tr>
<tr>
<td>$5 = 2</td>
<td></td>
</tr>
</tbody>
</table>
Tracing an Array Loop

• The following program sets all elements of an array to 0.

; $1 contains the starting address of A
; $2 contains the size of A
lis $11
.word 1
lis $4
.word 4
add $5, $0, $0 ; $5 holds i
for: slt $6, $5, $2 ; $6 is 1 if i < n, 0 otherwise
   beq $6, $0, end ; Go to end if i < n is false
   mult $5, $4
   mflo $6 ; $6 = i * 4
   add $6, $1, $6 ; $6 = address of A[i]
   sw $0, 0($6) ; A[i] = 0
   add $5, $5, $11 ; i += 1
   beq $0, $0, for ; back to top of loop
end: jr $31

A[0]
| MEM[address of A]       | 00000000 |
| MEM[address of A + 1]   | 00000000 |
| MEM[address of A + 2]   | 00000000 |
| MEM[address of A + 3]   | 00000000 |

A[1]
| MEM[address of A + 4]   | 00000000 |
| MEM[address of A + 5]   | 00000000 |
| MEM[address of A + 6]   | 00000000 |
| MEM[address of A + 7]   | 00000000 |

A[2]
| MEM[address of A + 8]   | 00000000 |
| MEM[address of A + 9]   | 00000000 |
| MEM[address of A + 10]  | 00000000 |
| MEM[address of A + 11]  | 00000011 |

$1 = address of A[0]  $2 = 3
$6 = 8  $5 = 2
Tracing an Array Loop

- The following program sets all elements of an array to 0.

```assembly
; $1 contains the starting address of A
; $2 contains the size of A
lis $11
.word 1
lis $4
.word 4
add $5, $0, $0 ; $5 holds i
for: slt $6, $5, $2 ; $6 is 1 if i < n, 0 otherwise
    beq $6, $0, end ; Go to end if i < n is false
    mult $5, $4
    mflo $6 ; $6 = i * 4
    add $6, $1, $6 ; $6 = address of A[i]
    sw $0, 0($6) ; A[i] = 0
    add $5, $5, $11 ; i += 1
    beq $0, $0, for ; back to top of loop
end: jr $31
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0]</td>
<td>MEM(address of A) 00000000</td>
</tr>
<tr>
<td>A[1]</td>
<td>MEM(address of A + 4) 00000000</td>
</tr>
<tr>
<td>A[2]</td>
<td>MEM(address of A + 8) 00000000</td>
</tr>
</tbody>
</table>

$1 = address of A[0] $2 = 3
$6 = address of A[2] $5 = 2
Tracing an Array Loop

• The following program sets all elements of an array to 0.

```assembly
; $1 contains the starting address of A
; $2 contains the size of A
lis $11
.word 1
lis $4
.word 4
add $5, $0, $0 ; $5 holds i
for: slt $6, $5, $2 ; $6 is 1 if i < n, 0 otherwise
  beq $6, $0, end ; Go to end if i < n is false
  mult $5, $4
  mflo $6 ; $6 = i * 4
  add $6, $1, $6 ; $6 = address of A[i]
  sw $0, 0($6) ; A[i] = 0
  add $5, $5, $11 ; i += 1
  beq $0, $0, for ; back to top of loop
end: jr $31
```

```
<table>
<thead>
<tr>
<th>A[0]</th>
<th>MEM[address of A]</th>
<th>00000000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MEM[address of A + 1]</td>
<td>00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 2]</td>
<td>00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 3]</td>
<td>00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 5]</td>
<td>00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 6]</td>
<td>00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 7]</td>
<td>00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 9]</td>
<td>00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 10]</td>
<td>00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 11]</td>
<td>00000000</td>
</tr>
</tbody>
</table>
```

$1 = address of A[0]  $2 = 3
$6 = address of A[2]  $5 = 2
Tracing an Array Loop

• The following program sets all elements of an array to 0.

```assembly
; $1 contains the starting address of A
; $2 contains the size of A
lis $11
.word 1
lis $4
.word 4
add $5, $0, $0 ; $5 holds i
for: slt $6, $5, $2 ; $6 is 1 if i < n, 0 otherwise
    beq $6, $0, end ; Go to end if i < n is false
    mult $5, $4
    mflo $6 ; $6 = i * 4
    add $6, $1, $6 ; $6 = address of A[i]
    sw $0, 0($6) ; A[i] = 0
    add $5, $5, $11 ; i += 1
PC
end: jr $31
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0]</td>
<td>MEM[address of A] 00000000</td>
</tr>
<tr>
<td>$1 = address of A[0]</td>
<td>$2 = 3</td>
</tr>
<tr>
<td>$6 = address of A[2]</td>
<td>$5 = 3</td>
</tr>
</tbody>
</table>
Tracing an Array Loop

• The following program sets all elements of an array to 0.

```assembly
; $1 contains the starting address of A
; $2 contains the size of A
lis $11
.word 1
lis $4
.word 4
add $5, $0, $0 ; $5 holds i
PC for: slt $6, $5, $2 ; $6 is 1 if i < n, 0 otherwise
    beq $6, $0, end ; Go to end if i < n is false
    mult $5, $4
    mflo $6 ; $6 = i * 4
    add $6, $1, $6 ; $6 = address of A[i]
    sw $0, 0($6) ; A[i] = 0
    add $5, $5, $11 ; i += 1
    beq $0, $0, for ; back to top of loop
end: jr $31
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0]</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>A[1]</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>A[2]</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
</tbody>
</table>

$1 = address of A[0] $2 = 3 $6 = address of A[2] $5 = 3
Tracing an Array Loop

• The following program sets all elements of an array to 0.

```assembly
; $1 contains the starting address of A
; $2 contains the size of A
lis $11
.word 1
lis $4
.word 4
add $5, $0, $0  ; $5 holds i
for:
slt $6, $5, $2  ; $6 is 1 if i < n, 0 otherwise
beq $6, $0, end  ; Go to end if i < n is false
mult $5, $4
mflo $6  ; $6 = i * 4
add $6, $1, $6  ; $6 = address of A[i]
sw $0, 0($6)  ; A[i] = 0
add $5, $5, $11  ; i += 1
beq $0, $0, for  ; back to top of loop
end: jr $31
```

A[0]

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0]</td>
<td>00000000</td>
</tr>
<tr>
<td>A[1]</td>
<td>00000000</td>
</tr>
</tbody>
</table>

A[1]

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0]</td>
<td>00000000</td>
</tr>
<tr>
<td>A[1]</td>
<td>00000000</td>
</tr>
</tbody>
</table>

A[2]

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0]</td>
<td>00000000</td>
</tr>
<tr>
<td>A[1]</td>
<td>00000000</td>
</tr>
</tbody>
</table>

$1 = address of A[0]  $2 = 3
$6 = 0  $5 = 3
Tracing an Array Loop

• The following program sets all elements of an array to 0.

```assembly
; $1 contains the starting address of A
; $2 contains the size of A
lis $11
.word 1
lis $4
.word 4
add $5, $0, $0       ; $5 holds i
for:   slt $6, $5, $2 ; $6 is 1 if i < n, 0 otherwise
       beq $6, $0, end ; Go to end if i < n is false
       mult $5, $4
       mflo $6        ; $6 = i * 4
       add $6, $1, $6 ; $6 = address of A[i]
       sw $0, 0($6)  ; A[i] = 0
       add $5, $5, $11 ; i += 1
       beq $0, $0, for ; back to top of loop
PC end: jr $31
```

<table>
<thead>
<tr>
<th>Index</th>
<th>Address Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0]</td>
<td>MEM[address of A] 00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 1] 00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 2] 00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 3] 00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 5] 00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 6] 00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 7] 00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 9] 00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 10] 00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 11] 00000000</td>
</tr>
</tbody>
</table>

$1 = address of A[0]  $2 = 3
$6 = 0                $5 = 3
Tracing an Array Loop

• The following program sets all elements of an array to 0.

```assembly
; $1 contains the starting address of A
; $2 contains the size of A
lis $11
.word 1
lis $4
.word 4
add $5, $0, $0       ; $5 holds i
for: slt $6, $5, $2  ; $6 is 1 if i < n, 0 otherwise
    beq $6, $0, end ; Go to end if i < n is false
    mult $5, $4
    mflo $6         ; $6 = i * 4
    add $6, $1, $6  ; $6 = address of A[i]
    sw $0, 0($6)    ; A[i] = 0
    add $5, $5, $11 ; i += 1
end: jr $31
```

### Memory State

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0]</td>
<td>MEM[address of A] 00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 1] 00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 2] 00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 3] 00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 5] 00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 6] 00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 7] 00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 9] 00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 10] 00000000</td>
</tr>
<tr>
<td></td>
<td>MEM[address of A + 11] 00000000</td>
</tr>
</tbody>
</table>

### Variables

- $1 = address of A[0]
- $2 = 3
- $6 = 0
- $5 = 3
Allocating Arrays

• How do we actually get an array into memory so we can work with it?
• We can allocate it **statically**: the array is allocated before running the program at a fixed location.
• In C/C++, we can allocate arrays **dynamically** (at runtime) on either "the stack" or "the heap".
• We will learn how "the heap" works near the end of the course.
• We will learn about "the stack" today!
• But first, let's take a look at what static allocation looks like.
Example: A Statically Allocated Array

; returns 1 in $3 if $7 is in the courses array, returns 0 otherwise

PC 0x00  lis $4
    0x04  .word 4
    0x08  lis $1
    0x0c  .word courses ; equivalent to .word 0x38
    $1 = ?
    0x10  lis $2
    0x14  .word endArray ; equivalent to .word 0x44
    $3 = ?
    0x18  add $3, $0, $0
          $4 = ?
    0x1c  loop: beq $1, $2, end ; go to end if $1 == endArray
          $6 = ?
    0x20  lw $6, 0($1) ; load A[i] into $6
          $7 = 241
    0x24  add $1, $1, $4 ; $1 = address of A[i+1]
    0x28  bne $6, $7, loop ; continue loop if $7 != A[i]
    0x2c  lis $3
    0x30  .word 1 ; return $3 = 1 if $7 == A[i]
    0x34  end: jr $31
    0x38  courses: .word 240 ; array A starts here
    0x3c  .word 241
    0x40  .word 251
    0x44  endArray: ; array A ends here
Example: A Statically Allocated Array

; returns 1 in $3 if $7 is in the courses array, returns 0 otherwise

0x00    lis $4
0x04    .word 4
PC  0x08    lis $1
0x0c    .word courses ; equivalent to .word 0x38
0x10    lis $2
0x14    .word endArray ; equivalent to .word 0x44
0x18    add $3, $0, $0
0x1c    loop: beq $1, $2, end ; go to end if $1 == endArray
0x20    lw $6, 0($1) ; load A[i] into $6
0x24    add $1, $1, $4 ; $1 = address of A[i+1]
0x28    bne $6, $7, loop ; continue loop if $7 != A[i]
0x2c    lis $3
0x30    .word 1 ; return $3 = 1 if $7 == A[i]
0x34    end: jr $31
0x38    courses: .word 240 ; array A starts here
0x3c    .word 241
0x40    .word 251
0x44    endArray: ; array A ends here
Example: A Statically Allocated Array

; returns 1 in $3 if $7 is in the courses array, returns 0 otherwise

0x00  lis $4
0x04  .word 4
0x08  lis $1
0x0c  .word courses ; equivalent to .word 0x38

$1 = 0x38 (courses)

PC  0x10  lis $2 $2 = ?
0x14  .word endArray ; equivalent to .word 0x44
0x18  add $3, $0, $0 $4 = 4
0x1c  loop: beq $1, $2, end ; go to end if $1 == endArray $6 = ?
0x20  lw $6, 0($1) ; load A[i] into $6 $7 = 241
0x24  add $1, $1, $4 ; $1 = address of A[i+1]
0x28  bne $6, $7, loop ; continue loop if $7 != A[i]
0x2c  lis $3
0x30  .word 1 ; return $3 = 1 if $7 == A[i]
0x34  end: jr $31

$1 0x38 courses: .word 240 ; array A starts here
0x3c  .word 241
0x40  .word 251
0x44  endArray: ; array A ends here
Example: A Statically Allocated Array

; returns 1 in $3 if $7 is in the courses array, returns 0 otherwise

```
0x00  lis $4
0x04  .word 4
0x08  lis $1
0x0c  .word courses  ; equivalent to .word 0x38
0x10  lis $2
0x14  .word endArray  ; equivalent to .word 0x44
0x18  add $3, $0, $0  ; $4 = 4
0x1c  loop: beq $1, $2, end  ; go to end if $1 == endArray
0x20  lw $6, 0($1)  ; load A[i] into $6
0x24  add $1, $1, $4  ; $1 = address of A[i+1]
0x28  bne $6, $7, loop  ; continue loop if $7 != A[i]
0x2c  lis $3
0x30  .word 1  ; return $3 = 1 if $7 == A[i]
0x34  end: jr $31

$1 0x38 courses: .word 240  ; array A starts here
    .word 241
    .word 251
0x40  endArray:  ; array A ends here
```
Example: A Statically Allocated Array

; returns 1 in $3 if $7 is in the courses array, returns 0 otherwise

```
0x00    lis $4
0x04    .word 4
0x08    lis $1
0x0c    .word courses ; equivalent to .word 0x38 $1 = 0x38 (courses)
0x10    lis $2 $2 = 0x44 (endArray)
0x14    .word endArray ; equivalent to .word 0x44 $3 = 0
0x18    add $3, $0, $0 $4 = 4
PC 0x1c loop: beq $1, $2, end ; go to end if $1 == endArray $6 = ?
0x20    lw $6, 0($1) ; load A[i] into $6 $7 = 241
0x24    add $1, $1, $4 ; $1 = address of A[i+1]
0x28    bne $6, $7, loop ; continue loop if $7 != A[i]
0x2c    lis $3
0x30    .word 1 ; return $3 = 1 if $7 == A[i]
0x34    end: jr $31
```

$1 0x38 courses: .word 240 ; array A starts here
0x3c    .word 241
0x40    .word 251
0x44    endArray: ; array A ends here
Example: A Statically Allocated Array

; returns 1 in $3 if $7 is in the courses array, returns 0 otherwise

0x00    lis $4
0x04    .word 4
0x08    lis $1
0x0c    .word courses ; equivalent to .word 0x38
0x10    lis $2
0x14    .word endArray ; equivalent to .word 0x44
0x18    add $3, $0, $0
0x1c    loop: beq $1, $2, end ; go to end if $1 == endArray

PC 0x20    lw $6, 0($1) ; load A[i] into $6
0x24    add $1, $1, $4 ; $1 = address of A[i+1]
0x28    bne $6, $7, loop ; continue loop if $7 != A[i]
0x2c    lis $3
0x30    .word 1 ; return $3 = 1 if $7 == A[i]
0x34    end: jr $31

$1 0x38 courses: .word 240 ; array A starts here
0x3c    .word 241
0x40    .word 251
0x44    endArray: ; array A ends here

0x00 lis $4
0x04 .word 4
0x08 lis $1
0x0c .word courses ; equivalent to .word 0x38
0x10 lis $2
0x14 .word endArray ; equivalent to .word 0x44
0x18 add $3, $0, $0
0x1c loop: beq $1, $2, end ; go to end if $1 == endArray

PC 0x20 lw $6, 0($1) ; load A[i] into $6
0x24 add $1, $1, $4 ; $1 = address of A[i+1]
0x28 bne $6, $7, loop ; continue loop if $7 != A[i]
0x2c lis $3
0x30 .word 1 ; return $3 = 1 if $7 == A[i]
0x34 end: jr $31

0x00 lis $4
0x04 .word 4
0x08 lis $1
0x0c .word courses ; equivalent to .word 0x38
0x10 lis $2
0x14 .word endArray ; equivalent to .word 0x44
0x18 add $3, $0, $0
0x1c loop: beq $1, $2, end ; go to end if $1 == endArray

PC 0x20 lw $6, 0($1) ; load A[i] into $6
0x24 add $1, $1, $4 ; $1 = address of A[i+1]
0x28 bne $6, $7, loop ; continue loop if $7 != A[i]
0x2c lis $3
0x30 .word 1 ; return $3 = 1 if $7 == A[i]
0x34 end: jr $31

0x00 lis $4
0x04 .word 4
0x08 lis $1
0x0c .word courses ; equivalent to .word 0x38
0x10 lis $2
0x14 .word endArray ; equivalent to .word 0x44
0x18 add $3, $0, $0
0x1c loop: beq $1, $2, end ; go to end if $1 == endArray

PC 0x20 lw $6, 0($1) ; load A[i] into $6
0x24 add $1, $1, $4 ; $1 = address of A[i+1]
0x28 bne $6, $7, loop ; continue loop if $7 != A[i]
0x2c lis $3
0x30 .word 1 ; return $3 = 1 if $7 == A[i]
0x34 end: jr $31

0x00 lis $4
0x04 .word 4
0x08 lis $1
0x0c .word courses ; equivalent to .word 0x38
0x10 lis $2
0x14 .word endArray ; equivalent to .word 0x44
0x18 add $3, $0, $0
0x1c loop: beq $1, $2, end ; go to end if $1 == endArray

PC 0x20 lw $6, 0($1) ; load A[i] into $6
0x24 add $1, $1, $4 ; $1 = address of A[i+1]
0x28 bne $6, $7, loop ; continue loop if $7 != A[i]
0x2c lis $3
0x30 .word 1 ; return $3 = 1 if $7 == A[i]
0x34 end: jr $31

0x00 lis $4
0x04 .word 4
0x08 lis $1
0x0c .word courses ; equivalent to .word 0x38
0x10 lis $2
0x14 .word endArray ; equivalent to .word 0x44
0x18 add $3, $0, $0
0x1c loop: beq $1, $2, end ; go to end if $1 == endArray

PC 0x20 lw $6, 0($1) ; load A[i] into $6
0x24 add $1, $1, $4 ; $1 = address of A[i+1]
0x28 bne $6, $7, loop ; continue loop if $7 != A[i]
0x2c lis $3
0x30 .word 1 ; return $3 = 1 if $7 == A[i]
0x34 end: jr $31

0x00 lis $4
0x04 .word 4
0x08 lis $1
0x0c .word courses ; equivalent to .word 0x38
0x10 lis $2
0x14 .word endArray ; equivalent to .word 0x44
0x18 add $3, $0, $0
0x1c loop: beq $1, $2, end ; go to end if $1 == endArray

PC 0x20 lw $6, 0($1) ; load A[i] into $6
0x24 add $1, $1, $4 ; $1 = address of A[i+1]
0x28 bne $6, $7, loop ; continue loop if $7 != A[i]
0x2c lis $3
0x30 .word 1 ; return $3 = 1 if $7 == A[i]
0x34 end: jr $31

0x00 lis $4
0x04 .word 4
0x08 lis $1
0x0c .word courses ; equivalent to .word 0x38
0x10 lis $2
0x14 .word endArray ; equivalent to .word 0x44
0x18 add $3, $0, $0
0x1c loop: beq $1, $2, end ; go to end if $1 == endArray

PC 0x20 lw $6, 0($1) ; load A[i] into $6
0x24 add $1, $1, $4 ; $1 = address of A[i+1]
0x28 bne $6, $7, loop ; continue loop if $7 != A[i]
0x2c lis $3
0x30 .word 1 ; return $3 = 1 if $7 == A[i]
0x34 end: jr $31

0x00 lis $4
0x04 .word 4
0x08 lis $1
0x0c .word courses ; equivalent to .word 0x38
0x10 lis $2
0x14 .word endArray ; equivalent to .word 0x44
0x18 add $3, $0, $0
0x1c loop: beq $1, $2, end ; go to end if $1 == endArray

PC 0x20 lw $6, 0($1) ; load A[i] into $6
0x24 add $1, $1, $4 ; $1 = address of A[i+1]
0x28 bne $6, $7, loop ; continue loop if $7 != A[i]
0x2c lis $3
0x30 .word 1 ; return $3 = 1 if $7 == A[i]
0x34 end: jr $31

0x00 lis $4
0x04 .word 4
0x08 lis $1
0x0c .word courses ; equivalent to .word 0x38
0x10 lis $2
0x14 .word endArray ; equivalent to .word 0x44
0x18 add $3, $0, $0
0x1c loop: beq $1, $2, end ; go to end if $1 == endArray

PC 0x20 lw $6, 0($1) ; load A[i] into $6
0x24 add $1, $1, $4 ; $1 = address of A[i+1]
0x28 bne $6, $7, loop ; continue loop if $7 != A[i]
0x2c lis $3
0x30 .word 1 ; return $3 = 1 if $7 == A[i]
0x34 end: jr $31

0x00 lis $4
0x04 .word 4
0x08 lis $1
0x0c .word courses ; equivalent to .word 0x38
0x10 lis $2
0x14 .word endArray ; equivalent to .word 0x44
0x18 add $3, $0, $0
0x1c loop: beq $1, $2, end ; go to end if $1 == endArray

PC 0x20 lw $6, 0($1) ; load A[i] into $6
0x24 add $1, $1, $4 ; $1 = address of A[i+1]
0x28 bne $6, $7, loop ; continue loop if $7 != A[i]
0x2c lis $3
0x30 .word 1 ; return $3 = 1 if $7 == A[i]
0x34 end: jr $31
Example: A Statically Allocated Array

; returns 1 in $3 if $7 is in the courses array, returns 0 otherwise

0x00  lis $4
0x04  .word 4
0x08  lis $1
0x0c  .word courses  ; equivalent to .word 0x38
0x10  lis $2
0x14  .word endArray ; equivalent to .word 0x44
0x18  add $3, $0, $0
0x1c  $2 = 0x44 (endArray)
$3 = 0
0x20  lw $6, 0($1)
0x24  ; load A[i] into $6
$7 = 241
0x28  bne $6, $7, loop  ; continue loop if $7 != A[i]
0x2c  lis $3
0x30  .word 1  ; return $3 = 1 if $7 == A[i]
0x34  end:  jr $31

$1 0x38 courses: .word 240  ; array A starts here
0x3c  .word 241
0x40  .word 251
0x44  endArray:  ; array A ends here
Example: A Statically Allocated Array

; returns 1 in $3 if $7 is in the courses array, returns 0 otherwise

0x00  lis $4
0x04  .word 4
0x08  lis $1
0x0c  .word courses ; equivalent to .word 0x38
0x10  lis $2
0x14  .word endArray ; equivalent to .word 0x44
0x18  add $3, $0, $0
0x1c  loop: beq $1, $2, end ; go to end if $1 == endArray
     lw $6, 0($1) ; load A[i] into $6
     add $1, $1, $4 ; $1 = address of A[i+1]
0x24  bne $6, $7, loop ; continue loop if $7 != A[i]
     lis $3
0x30  .word 1 ; return $3 = 1 if $7 == A[i]
0x34  end: jr $31
0x38  courses: .word 240 ; array A starts here

$1 0x3c  (courses+4)
0x40  .word 251
0x44  endArray: ; array A ends here
Example: A Statically Allocated Array

; returns 1 in $3 if $7 is in the courses array, returns 0 otherwise

0x00    lis $4
0x04    .word 4
0x08    lis $1
0x0c    .word courses ; equivalent to .word 0x38
0x10    lis $2
0x14    .word endArray ; equivalent to .word 0x44
0x18    add $3, $0, $0

PC 0x1c  loop: beq $1, $2, end ; go to end if $1 == endArray
0x20    lw $6, 0($1) ; load A[i] into $6
0x24    add $1, $1, $4 ; $1 = address of A[i+1]
0x28    bne $6, $7, loop ; continue loop if $7 != A[i]
0x2c    lis $3
0x30    .word 1 ; return $3 = 1 if $7 == A[i]
0x34    end: jr $31
0x38    courses: .word 240 ; array A starts here

$1 0x3c    .word 241
0x40    .word 251
0x44    endArray: ; array A ends here
Example: A Statically Allocated Array

; returns 1 in $3 if $7 is in the courses array, returns 0 otherwise

0x00 lis $4
0x04 .word 4
0x08 lis $1
0x0c .word courses ; equivalent to .word 0x38 $1 = 0x3c (courses+4)
0x10 lis $2 $2 = 0x44 (endArray)
0x14 .word endArray ; equivalent to .word 0x44 $3 = 0
0x18 add $3, $0, $0 $4 = 4
0x1c loop: beq $1, $2, end ; go to end if $1 == endArray $6 = 240
PC 0x20 lw $6, 0($1) ; load A[i] into $6 $7 = 241
0x24 add $1, $1, $4 ; $1 = address of A[i+1]
0x28 bne $6, $7, loop ; continue loop if $7 != A[i]
0x2c lis $3
0x30 .word 1 ; return $3 = 1 if $7 == A[i]
0x34 end: jr $31
$1 0x3c .word 240 ; array A starts here
0x38 courses: .word 240
0x40 .word 251
0x44 endArray: ; array A ends here
Example: A Statically Allocated Array

; returns 1 in $3 if $7 is in the courses array, returns 0 otherwise

0x00  lis $4
0x04  .word 4
0x08  lis $1
0x0c  .word courses  ; equivalent to .word 0x38
       $1 = 0x3c (courses+4)
0x10  lis $2
0x14  .word endArray ; equivalent to .word 0x44
       $2 = 0x44 (endArray)
0x18  add $3, $0, $0
       $3 = 0
0x1c  loop: beq $1, $2, end  ; go to end if $1 == endArray
       $4 = 4
0x20  lw $6, 0($1)   ; load A[i] into $6
       $6 = 241
0x24  bne $6, $7, loop  ; continue loop if $7 != A[i]
       $7 = 241
0x28  lis $3
0x2c  .word 1          ; return $3 = 1 if $7 == A[i]
0x30  end: jr $31
0x34  courses: .word 240 ; array A starts here
0x38  .word 241
0x40  .word 251
0x44  endArray:        ; array A ends here
Example: A Statically Allocated Array

; returns 1 in $3 if $7 is in the courses array, returns 0 otherwise

0x00    lis $4
0x04    .word 4
0x08    lis $1
0x0c    .word courses ; equivalent to .word 0x38
0x10    lis $2
0x14    .word endArray ; equivalent to .word 0x44
0x18    add $3, $0, $0
0x1c    loop: beq $1, $2, end ; go to end if $1 == endArray
0x20    lw $6, 0($1) ; load A[i] into $6
0x24    add $1, $1, $4 ; $1 = address of A[i+1]
0x28    bne $6, $7, loop ; continue loop if $7 != A[i]
0x2c    lis $3
0x30    .word 1 ; return $3 = 1 if $7 == A[i]
0x34    end: jr $31
0x38    courses: .word 240 ; array A starts here
0x3c    .word 241
$1 0x40 .word 251
0x44    endArray: ; array A ends here
Example: A Statically Allocated Array

; returns 1 in $3 if $7 is in the courses array, returns 0 otherwise

0x00  lis $4
0x04  .word 4
0x08  lis $1
0x0c  .word courses ; equivalent to .word 0x38
0x10  lis $2
0x14  .word endArray ; equivalent to .word 0x44
0x18  add $3, $0, $0
0x1c  loop: beq $1, $2, end ; go to end if $1 == endArray
0x20  lw $6, 0($1) ; load A[i] into $6
0x24  add $1, $1, $4 ; $1 = address of A[i+1]
0x28  bne $6, $7, loop ; continue loop if $7 != A[i]

PC 0x2c  lis $3
0x30  .word 1 ; return $3 = 1 if $7 == A[i]
0x34  end: jr $31
0x38  courses: .word 240 ; array A starts here
0x3c  .word 241

$1 0x40  .word 251
0x44  endArray: ; array A ends here
Example: A Statically Allocated Array

; returns 1 in $3 if $7 is in the courses array, returns 0 otherwise

0x00    lis $4
0x04    .word 4
0x08    lis $1
0x0c    .word courses ; equivalent to .word 0x38 $1 = 0x40 (courses+8)
0x10    lis $2 $2 = 0x44 (endArray)
0x14    .word endArray ; equivalent to .word 0x44 $3 = 1
0x18    add $3, $0, $0 $4 = 4
0x1c    loop: beq $1, $2, end ; go to end if $1 == endArray $6 = 241
0x20    lw $6, 0($1) ; load A[i] into $6 $7 = 241
0x24    add $1, $1, $4 ; $1 = address of A[i+1]
0x28    bne $6, $7, loop ; continue loop if $7 != A[i]
0x2c    lis $3
0x30    .word 1 ; return $3 = 1 if $7 == A[i]
PC 0x34 end: jr $31
0x38 courses: .word 240 ; array A starts here
0x3c    .word 241
$1 0x40  endArray: ; array A ends here
Example: A Statically Allocated Array

; returns 1 in $3 if $7 is in the courses array, returns 0 otherwise
0x00   lis $4
0x04   .word 4
0x08   lis $1
0x0c   .word courses ; equivalent to .word 0x38
0x10   lis $2
0x14   .word endArray ; equivalent to .word 0x44
0x18   add $3, $0, $0
0x1c   loop: beq $1, $2, end ; go to end if $1 == endArray
0x20   lw $6, 0($1)       ; load A[i] into $6
0x24   add $1, $1, $4   ; $1 = address of A[i+1]
0x28   bne $6, $7, loop ; continue loop if $7 != A[i]
0x2c   lis $3
0x30   .word 1           ; return $3 = 1 if $7 == A[i]
0x34   end: jr $31
0x38   courses: .word 240 ; array A starts here
0x3c   .word 241
0x40   .word 251
0x44   endArray:         ; array A ends here

$1 = 0x40 (courses+8)  $2 = 0x44 (endArray)  $3 = 1
$4 = 4                 $6 = 241
$7 = 241

Stack Allocation

• When you run a program, it does not take up all available memory. Instead, the program is allocated a specific chunk of memory.

• In our MIPS emulator, programs are always allocated the region of memory from 0x00000000 (inclusive) to 0x01000000 (exclusive).
  • Accessing addresses 0x01000000 or higher gives an "out of bounds" error.

• The program code itself is at address 0, the "lower end" of memory.

• We will use the opposite end of program memory as a stack.

• To facilitate this, $30 is initialized to 0x01000000, and we treat $30 as the stack pointer, the memory address of the top of the stack.
Using the Stack

• Note that the initial value of the stack pointer $30 is an out of bounds address. This represents the stack being *empty*.  

• All addresses higher than 0x01000000 are out of bounds. So the stack actually grows "backwards" from high to low addresses.  
  • It's a little confusing. When we push to the stack, we say the stack gets "higher", but the address of the top of the stack *decreases* numerically.

• To push something on the stack, there are two steps.  
  • *Store* the word at the address 4 bytes before the stack pointer. 
  • *Decrement* the stack pointer by 4, so it points to the new top of the stack.

• Popping from the stack is similar, but reversed: increment then load.
Pushing and Popping: Examples

• Push value in $3 onto the stack:
  
  sw $3, -4($30)
  lis $3
  .word 4
  sub $30, $30, $3

• Pop value from top of stack into $3:
  
  lis $3
  .word 4
  add $30, $30, $3
  lw $3, -4($30)

• There are other ways of doing this, but the methods shown here are fairly safe and general. Alternative approaches might require care.
Pushing and Popping: Examples

• Push value in $3 onto the stack:

```assembly
sw $3, -4($30)
lis $3
.word 4
sub $30, $30, $3
```

• Pop value from top of stack into $3:

```assembly
lis $3
.word 4
add $30, $30, $3
lw $3, -4($30)
```

• There are other ways of doing this, but the methods shown here are fairly safe and general. Alternative approaches might require care.
Pushing and Popping: Examples

• Push value in $3 onto the stack:
  sw $3, -4($30)
  lis $3
  .word 4
  sub $30, $30, $3

• Pop value from top of stack into $3:
  lis $3
  .word 4
  add $30, $30, $3
  lw $3, -4($30)

• There are other ways of doing this, but the methods shown here are fairly safe and general. Alternative approaches might require care.
Pushing and Popping: Examples

• Push value in $3 onto the stack:

```
sw $3, -4($30)
lis $3
.word 4
sub $30, $30, $3
```

• Pop value from top of stack into $3:

```
lis $3
.word 4
add $30, $30, $3
lw $3, -4($30)
```

• There are other ways of doing this, but the methods shown here are fairly safe and general. Alternative approaches might require care.
Pushing and Popping: Examples

• Push value in $3 onto the stack:

sw $3, -4($30)
lis $3
.word 4
sub $30, $30, $3

• Pop value from top of stack into $3:

lis $3
.word 4
add $30, $30, $3
lw $3, -4($30)

• There are other ways of doing this, but the methods shown here are fairly safe and general. Alternative approaches might require care.
Pushing and Popping: Examples

• Push value in $3 onto the stack:
  sw $3, -4($30)
  lis $3
  .word 4
  sub $30, $30, $3

• Pop value from top of stack into $3:
  lis $3
  .word 4
  add $30, $30, $3
  lw $3, -4($30)

• There are other ways of doing this, but the methods shown here are fairly safe and general. Alternative approaches might require care.
Pushing and Popping: Examples

• Push value in $3 onto the stack:
  
  sw $3, -4($30)
  lis $3
  .word 4
  sub $30, $30, $3

• Pop value from top of stack into $3:
  
  lis $3
  .word 4
  add $30, $30, $3
  lw $3, -4($30)

• There are other ways of doing this, but the methods shown here are fairly safe and general. Alternative approaches might require care.
Pushing and Popping: Examples

• Push value in $3 onto the stack:
  
  \[
  \text{sw } \$3, \ -4(\$30) \\
  \text{lis } \$3 \\
  .\text{word } 4 \\
  \text{sub } \$30, \ $30, \ $3 
  \]

• Pop value from top of stack into $3:
  
  \[
  \text{lis } \$3 \\
  .\text{word } 4 \\
  \text{add } \$30, \ $30, \ $3 \\
  \text{lw } \$3, \ -4(\$30) 
  \]

• There are other ways of doing this, but the methods shown here are fairly safe and general. Alternative approaches might require care.
Pushing and Popping: Examples

• Push value in $3 onto the stack:
  
  sw $3, -4($30)
  lis $3
  .word 4
  sub $30, $30, $3

• Pop value from top of stack into $3:
  
  lis $3
  .word 4
  add $30, $30, $3
  lw $3, -4($30)

• There are other ways of doing this, but the methods shown here are fairly safe and general. Alternative approaches might require care.
Pushing and Popping: Examples

• Push value in $3 onto the stack:
  sw $3, -4($30)
  lis $3
  .word 4
  sub $30, $30, $3

• Pop value from top of stack into $3:
  lis $3
  .word 4
  add $30, $30, $3
  lw $3, -4($30)

• There are other ways of doing this, but the methods shown here are fairly safe and general. Alternative approaches might require care.
Examples of Pitfalls

• "I don't like the use of -4. Why not just use offset 0 instead?"

```
sw $3, 0($30)
lis $3
.word 4
sub $30, $30, $3
```
Examples of Pitfalls

• "I don't like the use of -4. Why not just use offset 0 instead?"

```assembly
sw $3, 0($30)
lis $3
.word 4
sub $30, $30, $3
```

• Initially, $30 is an out-of-bounds address, so this will crash.
Examples of Pitfalls

• "I don't like the use of -4, so I will decrement first, then use offset 0."

lis $3
.word 4
sub $30, $30, $3
sw $3, 0($30)
Examples of Pitfalls

• "I don't like the use of -4, so I will decrement first, then use offset 0."

  lis $3
  .word 4
  sub $30, $30, $3
  sw $3, 0($30)

• $3 gets overwritten with 4, so you need to use a different register for the decrement, or you lose the value of $3.
Examples of Pitfalls

- "I don't like the use of -4, so I will decrement first, then use offset 0."
  
  ```
  lis $4
  .word 4
  sub $30, $30, $4
  sw $3, 0($30)
  ```

- This is fine, as long as you weren't using $4 for anything important. Being able to use the same register for increment/decrement and store/load is kind of nice.
Allocating an Array on the Stack

• To allocate an array of \( n \) words, simply decrement the stack pointer by \( 4n \) rather than just 4.

; assume $2 contains \( n \), the size of the array
lis $4
.word 4
mult $2, $4
mflo $5          ; \( \}$5 contains \( 4 \times n \)$
sub $30, $30, $5  ; decrement stack pointer
add $1, $30, $0
; now \$1 contains the starting address of the array

• Notice that we didn't do anything to "initialize" the array! We just made space for it. You can initialize it using a loop as discussed earlier.

• To \textit{deallocate}, increment the stack pointer by \( 4n \).
Coming Up Next

• Now that we understand how to use part of memory as a stack, we will use this to implement *procedures* in MIPS assembly.
• You may be familiar with the concept of the "call stack" that forms when procedures call each other.
• We will see the precise details of how this is implemented using our stack pointer in $30!
• We will learn about the final instruction in our version of MIPS, Jump and Link Register (jalr) which is used for procedure calls.
• We'll even be able to write recursive procedures!