CS64 Week 2 Lecture 1

Kyle Dewey
Overview

• Two’s complement wrap-up
• Addition
• Subtraction
• Assembly
  • What’s in a processor
  • MIPS
  • QtSpim
Where Is Twos Complement From?

• Intuition: try to subtract 1 from 0, in decimal

• Involves borrowing from an invisible number on the left

• Twos complement is based on the same idea
Another View

- Modular arithmetic, with the convention that a leading 1 bit means negative

Denotes +1
Another View

- Modular arithmetic, with the convention that a leading 1 bit means negative.
Another View

- Modular arithmetic, with the convention that a leading 1 bit means negative

```
0  000
-1 111  001  1
-2 110  010  2
-3 101  011  3
-4 100
```

Denotes +1
Negation of 1

Diagram:

- 000
- 001
- 010
- 011
- 100
- 101
- 110
- 111

Arrows indicate the negation path.
Negation of 1

000

A circular diagram with nodes: 000, 111, 110, 101, 100. Arrows indicate the negation relationship between these nodes.
Negation of $1$
Negation of 1
Consequences

- What is the negation of 000?
Consequences

• What is the negation of 100?
Arithmetic Shift Right

• **Not exactly** division by a power of two
• Consider \(-3 / 2\)
Addition
Building Up Addition

• Question: how might we add the following, in decimal?

\[
\begin{array}{c}
986 \\
+123 \\
\hline
\end{array}
\]

?
Building Up Addition

- Question: how might we add the following, in decimal?

\[
\begin{align*}
986 \\
+123 \\
\hline
\end{align*}
\]

=?

\[
\begin{align*}
6 \\
+3 \\
\hline
?
\end{align*}
\]
# Building Up Addition

- Question: how might we add the following, in decimal?

\[
\begin{align*}
986 \\
+123 \\
\hline
? \\
\end{align*}
\]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>+2</td>
<td>+3</td>
</tr>
<tr>
<td></td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>?</td>
<td>9</td>
</tr>
</tbody>
</table>
Building Up Addition

Question: how might we add the following, in decimal?

\[
\begin{array}{c}
986 \\
+ 123 \\
\hline
? \\
\end{array}
\]

Carry: 1

\[
\begin{array}{c|c|c}
8 & +2 & 6 \\
0 & -- & +3 \\
\hline
0 & -- & 9
\end{array}
\]
Building Up Addition

- Question: how might we add the following, in decimal?

\[
\begin{align*}
986 & \\
+123 & \\
\hline
? & \\
\end{align*}
\]

\[
\begin{array}{c|c|c}
1 & 8 & 6 \\
9 & +2 & +3 \\
+1 & -- & -- \\
-- & 0 & 9 \\
? & & \\
\end{array}
\]
• Question: how might we add the following, in decimal?

\[
\begin{array}{c}
986 \\
+123 \\
\hline
? \\
\end{array}
\]

Carry: 1

<table>
<thead>
<tr>
<th>1</th>
<th>8</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>+2</td>
<td>+3</td>
</tr>
<tr>
<td>+1</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>--</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Building Up Addition

• Question: how might we add the following, in decimal?

\[
\begin{array}{c}
986 \\
+123 \\
\hline
? \\
\end{array}
\]

\[
\begin{array}{c}
1 \\
+0 \\
\hline
1 \\
\end{array} \\
\begin{array}{c}
1 \\
+1 \\
\hline
1 \\
\end{array} \\
\begin{array}{c}
8 \\
+2 \\
\hline
6 \\
\end{array} \\
\begin{array}{c}
6 \\
+3 \\
\hline
9 \\
\end{array}
\]
Core Concepts

• We have a “primitive” notion of adding single digits, along with an idea of carrying digits

• We can build on this notion to add numbers together that are more than one digit long
Now in Binary

- Arguably simpler - fewer one-bit possibilities

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>+0</td>
<td>+1</td>
<td>+0</td>
<td>+1</td>
</tr>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
Now in Binary

- Arguably simpler - fewer one-bit possibilities

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>+0</td>
<td>0</td>
<td>+0</td>
</tr>
<tr>
<td>--</td>
<td>+1</td>
<td>--</td>
<td>+0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Carry: 1
Chaining the Carry

- Also need to account for any input carry

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>+0</td>
<td>+1</td>
<td>+1</td>
<td>+0</td>
<td>+1</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>--</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>+0</td>
<td>+1</td>
<td>+1</td>
<td>+0</td>
<td>+1</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>--</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Carry: 1</td>
<td>0</td>
<td>Carry: 1</td>
<td>1</td>
<td>Carry: 1</td>
<td>1</td>
<td>Carry: 1</td>
</tr>
</tbody>
</table>
Adding Multiple Bits

• How might we add the numbers below?

\[
\begin{array}{c}
011 \\
+001 \\
\hline
011
\end{array}
\]
Adding Multiple Bits

• How might we add the numbers below?

```
  0
011
+001
--------
```
Adding Multiple Bits

• How might we add the numbers below?

```
  10
 011
+001
  ----
   0
```
Adding Multiple Bits

• How might we add the numbers below?

\[
\begin{array}{c}
110 \\
011 \\
+001 \\
\hline \\
00
\end{array}
\]
Adding Multiple Bits

• How might we add the numbers below?

```
  0110
+ 011
+ 001
-----
 100
```
• How might we add the numbers below?

```
0110
011
+001
---
100
```
Another Example

111
+001
-----
Another Example

\[\begin{align*}
0 \\
111 \\
+001 \\
\hline
\end{align*}\]
Another Example

\[
\begin{align*}
10 \\
111 \\
+001 \\
\hline \\
0
\end{align*}
\]
Another Example

\[
\begin{array}{c}
110 \\
111 \\
+001 \\
\hline
00
\end{array}
\]
Another Example

Output Carry Bit

---

Result Bits

1110
111
+001

000
Output Carry Bit Significance

• For unsigned numbers, it indicates if the result did not fit all the way into the number of bits allotted

• May be an error condition for software
Signed Addition

• Question: what is the result of the following operation?

\[
\begin{array}{c}
011 \\
+011 \\
\text{---} \\
? \\
\end{array}
\]
Signed Addition

• Question: what is the result of the following operation?

\[
\begin{array}{c}
011 \\
+ 011 \\
\hline
0111
\end{array}
\]
Overflow

• In this situation, overflow occurred: this means that both the operands had the same sign, and the result’s sign differed

```
  011
+011
----
 110
```

• Possibly a software error
Overflow vs. Carry

- These are **different ideas**
- Carry is relevant to **unsigned** values
- Overflow is relevant to **signed** values

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>011</td>
<td>111</td>
<td>001</td>
</tr>
<tr>
<td>+001</td>
<td>+011</td>
<td>+100</td>
<td>+001</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>110</td>
<td>011</td>
<td>010</td>
</tr>
<tr>
<td>No Overflow; Carry</td>
<td>Overflow; No Carry</td>
<td>Overflow; Carry</td>
<td>No Overflow; No Carry</td>
</tr>
</tbody>
</table>
Subtraction
Subtraction

- Have been saying to invert bits and add one to second operand
- Could do it this way in hardware, but there is a trick

\[
\begin{array}{c}
001 \\
-001 \\
--- \\
?
\end{array}
\quad \text{Hint: these two questions are equivalent}
\quad \begin{array}{c}
001 \\
+111 \\
--- \\
?
\end{array}
\]
Subtraction Trick

• Assume we can cheaply invert bits, but we want to avoid adding twice (once to add 1 and once to add the other result)

• How can we do this easily?
Subtraction Trick

• Assume we can cheaply invert bits, but we want to avoid adding twice (once to add 1 and once to add the other result)

• How can we do this easily?
  • Set the initial carry to $1$ instead of $0$
Subtraction Example

0101
-0011
----
Subtraction Example

0101
-0011
----

Invert 0011
Subtraction Example

0101
-0011
---

Invert 0011

1100
Subtraction Example

\[
\begin{array}{c}
0101 \\
-0011 \\
\hline
\end{array}
\quad \text{Invert} \quad 0011 \quad \text{Equivalent to} \quad 1100
\]
Subtraction Example

\[
\begin{array}{c}
0101 \\
-0011 \\
\hline
\end{array}
\quad \text{Invert} \quad \begin{array}{c}
0011 \\
\hline
1100 \\
\end{array}
\quad \text{Equivalent to} \quad \begin{array}{c}
0101 \\
+1100 \\
\hline
1
\end{array}
\]
Subtraction Example

\[ \begin{array}{c}
0101 \\
-0011 \\
\hline
\end{array} \]

Invert 0011

\[ \begin{array}{c}
1100 \\
\hline
\end{array} \]

Equivalent to

\[ \begin{array}{c}
11011 \\
0101 \\
+1100 \\
\hline
0010 \\
\end{array} \]
Assembly
What’s in a Processor?
Simple Language

- We have variables, integers, addition, and assignment
- Restrictions:
  - Can only assign integers directly to variables
  - Can only add variables, always two at a time

Want to say: \[ z = 5 + 7; \]

Translation:

\[
\begin{align*}
  x &= 5; \\
  y &= 7; \\
  z &= x + y;
\end{align*}
\]
Implementation

• What do we need to implement this?

\[
x = 5;
y = 7;
z = x + y;
\]
Core Components

- Some place to hold the statements as we operate on them
- Some place to hold which statement is next
- Some place to hold variables
- Some way to add numbers
Back to Processors

• Amazingly, these are all the core components of a processor
  • Why is this significant?
• Amazingly, these are all the core components of a processor
  • Why is this significant?
• Processors just reads a series of statements (instructions) forever. No magic.
Core Components

• Some place to hold the statements as we operate on them
• Some place to hold which statement is next
• Some place to hold variables
• Some way to add numbers
Core Components

• Some place to hold the statements as we operate on them - **memory**

• Some place to hold which statement is next - **program counter**

• Some place to hold variables - **registers**
  • Behave just like variables with fixed names

• Some way to add numbers - **arithmetic logic unit (ALU)**

• Some place to hold which statement is currently being executed - **instruction register (IR)**
Basic Interaction

• Copy instruction from memory at wherever the program counter says into the instruction register

• Execute it, possibly involving registers and the arithmetic logic unit

• Update the program counter to point to the next instruction

• Repeat
Basic Interaction

initialize();
while (true) {
    instruction_register = memory[program_counter];
    execute(instruction_register);
    program_counter ++;
}

Memory

0: x = 5;
1: y = 7;
2: z = x + y;

Registers

x: ?
y: ?
z: ?

Program Counter

0

Instruction Register

?

Arithmetic Logic Unit

?
Instruction Register
------------------------
x = 5;

Registers
---------
x: ?
y: ?
z: ?

Program Counter
-----------------
0

Memory
-------
0: x = 5;
1: y = 7;
2: z = x + y;

Arithmetic Logic Unit
----------------------
?
Instruction Register
-----------------------------
\texttt{x = 5;}

Registers
-----------------------------
\texttt{x: 5}
\texttt{y: ?}
\texttt{z: ?}

Program Counter
----------------------
0

Memory
-----------------------------
0: \texttt{x = 5;}
1: \texttt{y = 7;}
2: \texttt{z = x + y;}

Arithmetic Logic Unit
-----------------------------
?
Instruction Register

x = 5;

Registers

x: 5
y: ?
z: ?

Program Counter

1

Memory

0: x = 5;
1: y = 7;
2: z = x + y;

Arithmetic Logic Unit

0 + 1 = 1
Memory

0: x = 5;
1: y = 7;
2: z = x + y;

Instruction Register

y = 7;

Instruction Register

y = 7;

Instruction Register

y = 7;

Instruction Register

y = 7;

Registers

x: 5
y: ?
z: ?

Program Counter

1

Program Counter

1

Program Counter

1

Program Counter

1

Arithmetic Logic Unit

?

Arithmetic Logic Unit

?

Arithmetic Logic Unit

?
Instruction Register

\[ y = 7; \]

Registers

\begin{align*}
x & : 5 \\
y & : 7 \\
z & : ?
\end{align*}

Program Counter

\begin{align*}
1 
\end{align*}

Memory

\begin{align*}
0 & : x = 5; \\
1 & : y = 7; \\
2 & : z = x + y;
\end{align*}

Arithmetic Logic Unit
Memory

0: x = 5;
1: y = 7;
2: z = x + y;

Instruction Register

y = 7;

Registers

x: 5
y: 7
z: ?

Program Counter

2

Arithmetic Logic Unit

1 + 1 = 2
z = x + y;

Registers

x: 5
y: 7
z: ?

Instruction Register

z = x + y;

Program Counter

2

Registers

x: 5
y: 7
z: ?

Program Counter

2

Arithmetic Logic Unit

?
Instruction Register

\[ z = x + y; \]

 Registers

<table>
<thead>
<tr>
<th>x</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>7</td>
</tr>
<tr>
<td>z</td>
<td>?</td>
</tr>
</tbody>
</table>

Memory

0: \( x = 5; \)
1: \( y = 7; \)
2: \( z = x + y; \)

Program Counter

2

Arithmetic Logic Unit

\[ 5 + 7 = 12 \]
Instruction Register

\[ z = x + y; \]

Registers

\[ \begin{align*}
    x &: 5 \\
    y &: 7 \\
    z &: 12
\end{align*} \]

Program Counter

2

Memory

\[ \begin{align*}
    0 &: x = 5; \\
    1 &: y = 7; \\
    2 &: z = x + y;
\end{align*} \]

Arithmetic Logic Unit

\[ 5 + 7 = 12 \]
MIPS
Why MIPS?

- Relevant in the embedded systems domain
- All processors share the same core concepts as MIPS, just with extra stuff
- ...but most importantly...
It’s Simpler

• RISC (reduced instruction set computing)
  • Dozens of instructions as opposed to hundreds
  • Lack of redundant instructions or special cases
• Five stage pipeline versus 24 stages
## Code on MIPS

<table>
<thead>
<tr>
<th>Original</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 5;</td>
<td>li $t0, 5</td>
</tr>
<tr>
<td>y = 7;</td>
<td>li $t1, 7</td>
</tr>
<tr>
<td>z = x + y;</td>
<td>add $t3, $t0, $t1</td>
</tr>
</tbody>
</table>
## Code on MIPS

<table>
<thead>
<tr>
<th>Original</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = 5; )</td>
<td>( \text{li} \ $t0, 5 )</td>
</tr>
<tr>
<td>( y = 7; )</td>
<td>( \text{li} \ $t1, 7 )</td>
</tr>
<tr>
<td>( z = x + y; )</td>
<td>( \text{add} \ $t3, $t0, $t1 )</td>
</tr>
</tbody>
</table>

**MIPS**:
- **Load immediate**: put the given value into a register
- **$t0**: Temporary register 0
## Code on MIPS

<table>
<thead>
<tr>
<th>Original</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = 5; )</td>
<td>\textit{li} $\textit{t0}, 5$</td>
</tr>
<tr>
<td>( y = 7; )</td>
<td>\textit{li} $\textit{t1}, 7$</td>
</tr>
<tr>
<td>( z = x + y; )</td>
<td>\textit{add} $\textit{t3}, \textit{t0}, \textit{t1}$</td>
</tr>
</tbody>
</table>

\textbf{Load immediate:} put the given value into a register

\textbf{$\textit{t1}$: temporary register}
<table>
<thead>
<tr>
<th>Original</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x = 5;)</td>
<td>(\text{li } $t0, 5)</td>
</tr>
<tr>
<td>(y = 7;)</td>
<td>(\text{li } $t1, 7)</td>
</tr>
<tr>
<td>(z = x + y;)</td>
<td>(\text{add } $t3, $t0, $t1)</td>
</tr>
</tbody>
</table>

**add**: add the rightmost registers, putting the result in the first register

**\$t3**: temporary register 3
Available Registers

• 32 registers in all

• For the moment, we will only consider registers $t0 - t9$
Assembly

• The code that you see below is MIPS assembly

• Assembly is *almost* what the machine sees. For the most part, it is a direct translation to binary from here (known as machine code)

```
li $t0, 5
li $t1, 7
add $t3, $t0, $t1
```
Workflow

Assembly

li $t0, 5
li $t1, 7
add $t3, $t0, $t1

Assembler
(analogous to a compiler)

Machine Code

001101....
Machine Code

• This is what the process actually executes and accepts as input
• Each instruction is represented with 32 bits
• Three different instruction formats; for the moment, we’ll only look at the R format

add $t3, $t0, $t1
Instruction Register
--------------------
?

Registers
---------
$t0$: ?
$t1$: ?
$t2$: ?

Program Counter
----------------
0

Memory
------
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1

Arithmetic Logic Unit
---------------------
?
Instruction Register
----------------------------------
li $t0, 5

Registers
----------------------------------
$t0: ?
$t1: ?
$t2: ?

Program Counter
----------------------------------
0

Memory
----------------------------------
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1

Arithmetic Logic Unit
----------------------------------
?
Instruction Register

```
1i $t0, 5
```

Registers

```
$t0: 5
$t1: ?
$t2: ?
```

Program Counter

```
0
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1
```

Arithmetic Logic Unit

```
?
```
Registers

$t0: 5
$t1: ?
$t2: ?

Instruction Register

li $t0, 5

Program Counter

4

Memory

0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1

Arithmetic Logic Unit

0 + 4 = 4
Registers

$t0: 5
$t1: ?
$t2: ?

Instruction Register

li $t1, 7

Program Counter

4

Arithmetic Logic Unit

?

Memory

0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1

Arithmetic Logic Unit
Registers

$\text{t0}: 5$
$\text{t1}: 7$
$\text{t2}: ?$

Program Counter

4

Arithmetic Logic Unit

?

Instruction Register

`li $t1, 7`

Memory

0: `li $t0, 5`
4: `li $t1, 7`
8: `add $t3, $t0, $t1`
Registers

$\texttt{t0}: 5$
$\texttt{t1}: 7$
$\texttt{t2}: ?$

Program Counter

8

Arithmetic Logic Unit

$4 + 4 = 8$

Memory

0: \texttt{li \ $t0, 5}$
4: \texttt{li \ $t1, 7}$
8: \texttt{add \ $t3, \ t0, \ t1}$

Instruction Register

\texttt{li \ $t1, 7}$
Registers

$\text{t}_0: 5$
$\text{t}_1: 7$
$\text{t}_2: ?$

Program Counter

8

Instruction Register

add $\text{t}_3, \text{t}_0, \text{t}_1$

Memory

0: li $\text{t}_0, 5$
4: li $\text{t}_1, 7$
8: add $\text{t}_3, \text{t}_0, \text{t}_1$

Arithmetic Logic Unit

?
Instruction Register

\texttt{add \$t3, \$t0, \$t1}

Registers

\begin{align*}
\texttt{\$t0:} & \ 5 \\
\texttt{\$t1:} & \ 7 \\
\texttt{\$t2:} & \ ?
\end{align*}

Program Counter

\begin{align*}
\texttt{8}
\end{align*}

Memory

\begin{align*}
\texttt{0: li \$t0, 5} \\
\texttt{4: li \$t1, 7} \\
\texttt{8: add \$t3, \$t0, \$t1}
\end{align*}

Arithmetic Logic Unit

\begin{align*}
\texttt{5 + 7 = 12}
\end{align*}
Registers

\[
\begin{align*}
$t0: 5 \\
$t1: 7 \\
$t2: 12
\end{align*}
\]

Program Counter

8

Arithmetic Logic Unit

5 + 7 = 12

Instruction Register

add $t3, $t0, $t1

Memory

0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1
Adding More Functionality

- We need a way to display the result.
- What does this entail?
Adding More Functionality

• We need a way to display the result

• What does this entail?
  
  • Input / output. This entails talking to devices, which the operating system handles

  • We need a way to tell the operating system to kick in
Talking to the OS

• We are going to be running on a MIPS emulator, SPIM

• We cannot directly access system libraries (they aren’t even in the same machine language)

• How might we print something?
SPIM Routines

• MIPS features a syscall instruction, which triggers a software interrupt, or exception

• Outside of an emulator, these pause the program and tell the OS to check something

• Inside the emulator, it tells the emulator to check something
syscall

• So we have the OS/emulator’s attention. But how does it know what we want?
syscall

• So we have the OS/emulator’s attention. But how does it know what we want?
  • It has access to the registers
  • Put special values in the registers to indicate what you want
(Finally) Printing an Integer

- For SPIM, if register $v0 contains 1, then it will print whatever integer is stored in register $a0

- Note that $v0 and $a0 are distinct from $t0 - $t9
Augmenting with Printing

li $t0, 5
li $t1, 7
add $t3, $t0, $t1
li $v0 1
move $a0, $t3
syscall
Exiting

• If you are using SPIM, then you need to say when you are done as well

• How might this be done?
Exiting

- If you are using SPIM, then you need to say when you are done as well
- How might this be done?
  - syscall with a special value in $v0 (specifically 10 decimal)
Augmenting with Exiting

li $t0, 5
li $t1, 7
add $t3, $t0, $t1

li $v0 1
move $a0, $t3
syscall

li $v0, 10
syscall
Making it a Full Program

- Everything is just a bunch of bits
- We need to tell the assembler which bits should be placed where in memory
Allocated as Program Runs
Everything Below is Allocated at Program Load

\[ \text{Constants (e.g., strings)} \]

\[ \text{Mutable Global Variables} \]

\[ \text{Code} \]

\[ \text{Allocated as Program Runs} \]

\[ \text{Stack} \]

\[ \text{Free Memory} \]

\[ \text{Heap} \]

\[ \text{Initialized Data} \]

\[ \text{Uninitialized Data (BSS)} \]

\[ \text{Text} \]
Marking Code

- Use a `.text` directive to specify code

```
.text

li $t0, 5
li $t1, 7
add $t3, $t0, $t1
li $v0 1
move $a0, $t3
syscall

li $v0, 10
syscall
```
Running With SPIM
(add2.asm)
move Instruction

• The move instruction does not actually show up in SPIM

• It is a pseudoinstruction which is translated into an actual instruction

<table>
<thead>
<tr>
<th>Original</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>move $a0, $t3</td>
<td>addu $a0, $zero, $t3</td>
</tr>
</tbody>
</table>
$zero

• Specified like a normal register, but does not behave like a normal register
  • Writes to $zero are not saved
  • Reads from $zero always return 0
But why?

- Why have move as a pseudoinstruction instead of as an actual instruction?
But why?

- Why have move as a pseudoinstruction instead of as an actual instruction?
  - One less instruction to worry about
  - One design goal of RISC is to cut out redundancy
The `li` instruction does not actually show up in SPIM.

It is a *pseudoinstruction* which is translated into actual instructions.

Why might `li` work this way?

- Hint: instructions and registers are both 32 bits long.
load intermediate

- The \texttt{li} instruction does not actually show up in SPIM
- It is a \textit{pseudoinstruction} which is translated into actual instructions
- Why might \texttt{li} work this way?
  - Not enough room in one instruction to fit everything within 32 bits
  - I-type instructions only hold 16 bits
Assembly Coding Strategy

• Best to write it in C-like language, then translate down by hand

• This gets more complex when we get into control structures and memory

```
x = 5;
y = 7;
z = x + y;
```

```
li $t0, 5
li $t1, 7
add $t3, $t0, $t1
```
More Examples

• swap.asm
• negate.asm
• mult80.asm
• div80.asm
Control Structure Examples

• max.asm
• sort2.asm
• add_0_to_n.asm