Overview

- What’s in a processor
- MIPS
- QtSpim
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:
\[
x = 5;
y = 7;
z = x + y;
\]
Implementation

• What do we need to implement this?

\[
\begin{align*}
x &= 5; \\
y &= 7; \\
z &= x + y;
\end{align*}
\]
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?
Back to Processors

• 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: ?

Instruction Register

?

Program Counter

0

Arithmetic Logic Unit

?
Memory

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

Instruction Register

x = 5;

Registers

x: ?
y: ?
z: ?

Program Counter

0

Arithmetic Logic Unit

?
Instruction Register
------------------------
\[ x = 5; \]

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

Program Counter
------------------
\[ 0 \]

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

Arithmetic Logic Unit
----------------------
?
Instruction Register
-----------------------------
\[
x = 5;
\]

Registers
-----------------------------
\[
\begin{align*}
x & : 5 \\
y & : ? \\
z & : ? 
\end{align*}
\]

Program Counter
-----------------------------
\[
1
\]

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

Arithmetic Logic Unit
-----------------------------
\[
0 + 1 = 1
\]
Instruction Register
-------------------------
y = 7;

Registers
---------
x: 5
y: ?
z: ?

Program Counter
-----------------
1

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

Arithmetic Logic Unit
----------------------
?
Memory

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

Instruction Register

y = 7;

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>

Program Counter

1

Instruction Register

y = 7;

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
Memory

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

Registers

x: 5
y: 7
z: ?

Instruction Register

z = x + y;

Program Counter

2

Arithmetic Logic Unit

?
Instruction Register

\[ z = x + y; \]

Registers

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

Program Counter

2

Memory

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

Arithmetic Logic Unit

\[ 5 + 7 = 12 \]
Instruction Register

\[ z = x + y; \]

Registers

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

Program Counter

2

Memory

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

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
<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>
Code on MIPS

Original

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

MIPS

\[ \text{li } \$t0, 5 \]
\[ \text{li } \$t1, 7 \]
\[ \text{add } \$t3, \$t0, \$t1 \]

**load immediate**: put the given value into a register

**\$t0**: temporary register 0
Code on MIPS

Original

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

MIPS

\[
\text{li } \$t0, 5 \\
\text{li } \$t1, 7 \\
\text{add } \$t3, \$t0, \$t1
\]

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

\texttt{\$t1: temporary register 1}
Code on MIPS

Original

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

MIPS

\[
\begin{align*}
\text{li } & \text{ } $t0, 5 \\
\text{li } & \text{ } $t1, 7 \\
\text{add } & \text{ } $t3, $t0, $t1
\end{align*}
\]

**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).

```assembly
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
```
Registers

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

Program Counter

0

Arithmetic Logic Unit

?

Memory

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

Instruction Register

?
Registers

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

Instruction Register

```
li $t0, 5
```

Program Counter

```
0
```

Memory

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

Arithmetic Logic Unit

```
?
```
Registers

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

Instruction Register

-li $\texttt{t0}, 5$

Program Counter

0

Memory

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

Arithmetic Logic Unit

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

Program Counter:
4

Instruction Register:
li $t0, 5

Arithmetic Logic Unit:
0 + 4 = 4

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

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

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

$t0: 5
$t1: 7
$t2: ?

Program Counter

4

Arithmetic Logic Unit

?

Memory

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

Instruction Register

li $t1, 7
Registers

$t0: 5
$t1: 7
$t2: ?

Program Counter

8

Arithmetic Logic Unit

4 + 4 = 8

Memory

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

Instruction Register

li $t1, 7
Instruction Register

---

add $t3, $t0, $t1

Registers

---

$t0: 5
$t1: 7
$t2: ?

Program Counter

---

8

Memory

---

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

Arithmetic Logic Unit

---

?
Registers

$t0: 5
$t1: 7
$t2: ?

Program Counter

8

Instruction Register

add $t3, $t0, $t1

Memory

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

Arithmetic Logic Unit

5 + 7 = 12
Registers

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

Program Counter

8

Arithmetic Logic Unit

\[5 + 7 = 12\]

Memory

\[
\begin{align*}
&0: \text{li } t0, 5 \\
&4: \text{li } t1, 7 \\
&8: \text{add } t3, t0, t1
\end{align*}
\]

Instruction Register

\[\text{add } t3, t0, t1\]
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.
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

Mutable Global Variables

Constants (e.g., strings)

Allocated as Program Runs

Stack

Free Memory

Heap

Initialized Data

Uninitialized Data (BSS)

Code

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)
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 \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?

- Hint: instructions and registers are both 32 bits long.
The \( \text{li} \) instruction does not actually show up in SPIM. It is a pseudoinstruction which is translated into actual instructions. Why might \( \text{li} \) 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

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

\[
\begin{align*}
\text{li} & \ \$t0, \ 5 \\
\text{li} & \ \$t1, \ 7 \\
\text{add} & \ \$t3, \ $t0, \ $t1
\end{align*}
\]
More Examples

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

- max.asm
- sort2.asm
- add_0_to_n.asm