Overview

- Mini-review: Integer representations
- syscall
- QtSpim and pseudoinstructions
- Branches
- Memory introduction
Mini-review: Integer Representations
How do I tell the processor to do some operation in hex?
How do I tell the processor to do some operation in hex?

Answer: you’d have to code it yourself, but it doesn’t matter anyway.
To a Processor

- To a processor, everything is in binary
  - Cannot directly say to do an operation in hex
- *Could* write a program to add in hex, but it would ultimately go down to binary
- **Everything** is just a bunch of bits
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
QtSpim
.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
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?
  
  • 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

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

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

- swap.asm
- negate.asm
- mult80.asm
- div80.asm
Branches
Conditionals

• Using all the instructions learned so far, how might we code up the following?

```c
if (x == 0) {
    printf("x is zero");
}
```
Conditionals

• Using all the instructions learned so far, how might we code up the following?

```c
if (x == 0) {
    printf("x is zero");
}
```

Answer: We can’t (realistically).
Handling Conditionals

• What do we need to implement this?

```c
if (x == 0) {
    printf("x is zero");
}
```
Handling Conditionals

• What do we need to implement this?
  • A way to compare numbers
  • A way to conditionally execute code

```c
if (x == 0) {
    printf("x is zero");
}
```
Relevant Instructions

• Comparing numbers: set-less-than (\texttt{slt})
• Conditional execution: branch-on-equal (\texttt{beq}) and branch-on-not-equal (\texttt{bne})
• Do we need anything else?
Relevant Instructions

• Comparing numbers: set-less-than (\texttt{slt})

• Conditional execution: branch-on-equal (\texttt{beq}) and branch-on-not-equal (\texttt{bne})

• Do we need anything else?
  • This is sufficient
if (x == 0) {
    printf("x is zero");
}

.data
x_is_zero:  
    .asciiz "x is zero"

.text
    bne $t0, $zero, after_print
    li $v0, 4
    la $a0, x_is_zero
    syscall
after_print:  
    li $v0, 10
    syscall
• How might we translate the following to assembly?

    sum = 0;
    while (n != 0) {
        sum = sum + n;
        n--;  
    }
Control Structure
Examples

- max.asm
- sort2.asm
- add_0_to_n.asm
Memory
Accessing Memory

• Two base instructions: load-word ($lw$) and store-word ($sw$)

• MIPS lacks instructions that do more with memory than access it (e.g., retrieve something from memory and add)
  • Mark of RISC architecture
Global Variables

- Typically, global variables are placed directly in memory, not registers
- Why might this be?
Global Variables

• Typically, global variables are placed directly in memory, not registers

• Why might this be?
  • Not enough registers