Basic Assembly Programming in MIPS

CS 64: Computer Organization and Design Logic
Lecture #4

Ziad Matni
Dept. of Computer Science, UCSB
This Week on “Didja Know Dat?!”

Small corrections to the programmed sequence could be done by punching over portions of the paper tape and re-punching the holes in that section.

Image courtesy of the Smithsonian Archives Center.
Administrative

• Class is full – no more adds

• Lab 2 is this Monday (1/23)

• Assignments #2 AND #3 are given simultaneously
  – Assignment #2: Due Monday 1/30
  – Assignment #2: Due Friday 2/3
  – I RECOMMEND THAT YOU PARTNER UP!
    • Remember: you can change partners from one assignment to the next

• Midterm #1 is on Feb. 2\textsuperscript{nd}
  – Material included is: all lectures, all readings (see syllabus), all assignments turned in before 2/2.
  – Will do a review on the lesson before the midterm.
Lecture Outline

• MIPS core processing blocks
• Basic programming in assembly
• Arithmetic programs
• Control logic programs
Assembly

• The code that you see 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)

• An assembler takes assembly code and changes it into the actual 1’s and 0’s for machine code
  – Analogous to a compiler for HL code
Machine Code/Language

• 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 \textit{R format}
1/22/17

Matni, CS64, Wi17

---

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

Program Counter

```
4
```

Memory

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

Arithmetic Logic Unit

```
0 + 4 = 4
```
Instruction Register

li $t1, 7

Registers

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

Program Counter

4

Memory

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

Arithmetic Logic Unit

?
Instruction Register

`li $t1, 7`

Registers

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

Program Counter

4

Memory

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

Arithmetic Logic Unit

?
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

??
**Memory**

0: li $t0, 5

4: li $t1, 7

8: add $t3, $t0, $t1

**Instruction Register**

```
add $t3, $t0, $t1
```

**Registers**

$t0$: 5

$t1$: 7

$t2$: ?

**Program Counter**

8

**Arithmetic Logic Logic Unit**

```
5 + 7 = 12
```
add $t3, $t0, $t1

$t0: 5
$t1: 7
$t2: 12

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

5 + 7 = 12
Adding More Functionality

• We also need a way to display the result
• What would this entail?
  – Engaging with Input / Output.
  – i.e. talking to devices
    • What usually handles this? the operating system!

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

• We are going to be running on MIPS emulators called SPIM through a program called **QtSPIM** (GUI based)

• We’re not actually running our commands on a MIPS processor!!!
  ...so, in other words... we’re “faking it”
  – 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 (i.e. in the real world), 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?

- The OS/emulator has access to the registers

- Put special values in the registers to indicate what you want
  - These are codes that can’t be used for anything else, so they’re understood to be just for syscall
Program Files for MIPS Assembly

• The files have to be text

• Typical file extension type is .asm

• To leave comments, use # at the start of the line
(Finally) Printing an Integer

• For SPIM, if register $v0$ contains 1, then it will print whatever integer is stored in register $a0$
  – Other values in $v0$ indicate other types to syscall
    • Example $v0 = 3$ means double, $v0 = 4$ means string
  – Remember, the usual syntax to load immediate a value into a register is:
    \[ \text{li} \ <\text{register}>\ ,\ \ <\text{value}> \]
  – To make sure that the register $a0$ has the value of what you want to print out, use the move command:
    \[ \text{move} \ <\text{to register}>\ ,\ <\text{from register}> \]

• Note that $v0$ and $a0$ are distinct from $t0$ - $t9$
Ok... So About Those Registers
MIPS has 32 registers, each 32 bits

<table>
<thead>
<tr>
<th>NAME</th>
<th>NUMBER</th>
<th>USE</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>The Constant Value 0</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>Assembler Temporary</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>Values for Function Results and Expression Evaluation</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>Arguments</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>Temporaries</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>Saved Temporaries</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>Temporaries</td>
</tr>
<tr>
<td>$k0-$k1</td>
<td>26-27</td>
<td>Reserved for OS Kernel</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>Global Pointer</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>Frame Pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>Return Address</td>
</tr>
</tbody>
</table>
Augmenting with Printing

# Main program
li $t0, 5
li $t1, 7
add $t3, $t0, $t1

# Print an integer to std.output
li $v0, 1
move $a0, $t3
syscall
Exiting an Assembly Program in SPIM

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

- How might this be done?
  - Issue a syscall with a special value in $v0 (specifically 10 decimal)
Augmenting with Exiting

# Main program
li $t0, 5
li $t1, 7
add $t3, $t0, $t1

# Print to standard output
li $v0, 1
move $a0, $t3
syscall

# End program
li $v0, 10
syscall
MIPS System Services

<table>
<thead>
<tr>
<th>Service</th>
<th>System Call Code</th>
<th>Arguments</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_int</td>
<td>1</td>
<td>$a0 = integer</td>
<td></td>
</tr>
<tr>
<td>print_float</td>
<td>2</td>
<td>$f12 = float</td>
<td></td>
</tr>
<tr>
<td>print_double</td>
<td>3</td>
<td>$f12 = double</td>
<td></td>
</tr>
<tr>
<td>print_string</td>
<td>4</td>
<td>$a0 = string</td>
<td></td>
</tr>
<tr>
<td>read_int</td>
<td>5</td>
<td></td>
<td>integer (in $v0)</td>
</tr>
<tr>
<td>read_float</td>
<td>6</td>
<td></td>
<td>float (in $f0)</td>
</tr>
<tr>
<td>read_double</td>
<td>7</td>
<td></td>
<td>double (in $f0)</td>
</tr>
<tr>
<td>read_string</td>
<td>8</td>
<td>$a0 = buffer, $a1 = length</td>
<td></td>
</tr>
<tr>
<td>sbrk</td>
<td>9</td>
<td>$a0 = amount</td>
<td>address (in $v0)</td>
</tr>
<tr>
<td>exit</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>print_character</td>
<td>11</td>
<td>$a0 = character</td>
<td></td>
</tr>
<tr>
<td>read_character</td>
<td>12</td>
<td>$a0 = filename</td>
<td>character (in $v0)</td>
</tr>
<tr>
<td>open</td>
<td>13</td>
<td>$a0 = filename, $a1 = flags, $a2 = mode</td>
<td>file descriptor (in $v0)</td>
</tr>
<tr>
<td>read</td>
<td>14</td>
<td>$a0 = file descriptor, $a1 = buffer, $a2 = count</td>
<td>bytes read (in $v0)</td>
</tr>
<tr>
<td>write</td>
<td>15</td>
<td>$a0 = file descriptor, $a1 = buffer, $a2 = count</td>
<td>bytes written (in $v0)</td>
</tr>
<tr>
<td>close</td>
<td>16</td>
<td>$a0 = file descriptor</td>
<td>0 (in $v0)</td>
</tr>
<tr>
<td>exit2</td>
<td>17</td>
<td>$a0 = value</td>
<td></td>
</tr>
</tbody>
</table>
Now Let’s Make it a Full Program (almost)

- We need to tell the assembler (and the assembler simulator) which bits should be placed where in memory
  - Bits?
  (remember: everything ends up being a bunch of 1’s and 0’s!)

*Allocated as program RUN*

*Allocated at program LOAD*

*Constants to be used in the program (like strings)*

*Mutable global variables*

*The text of the program*
Marking the Code

• You’ll need a `.text` directive to specify code

```assembly
.text

# Main program
li $t0, 5
li $t1, 7
add $t3, $t0, $t1

# Print to standard output
li $v0, 1
move $a0, $t3
syscall

# End program
li $v0, 10
syscall
```

- Constants to be used in the program (like strings)
- Allocated as program RUN
- Allocated at program LOAD
- `the text of the program`
- `mutable global variables`
The *move* instruction

- The move instruction does not actually show up in SPIM

- It is a *pseudo-instruction* for us to use, but it’s actually translated into an actual instruction

**ORIGINAL:**  move $a0, $t3

**ACTUAL:**  addu $a0, $zero, $t3
Why Pseudocodes?
And what’s this $zero??

• $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

• Why have move as a pseudo-instruction instead of as an actual instruction?
  – It’s one less instruction to worry about
  – One design goal of RISC is to cut out redundancy
  – move isn’t the only one! li is one too!
<table>
<thead>
<tr>
<th>Name, Mnemonic</th>
<th>Format</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>add</td>
<td>R</td>
</tr>
<tr>
<td>Add Immediate</td>
<td>addi</td>
<td>I</td>
</tr>
<tr>
<td>Add Imm. Unsigned</td>
<td>addiu</td>
<td>I</td>
</tr>
<tr>
<td>Add Unsigned</td>
<td>addu</td>
<td>R</td>
</tr>
<tr>
<td>And</td>
<td>and</td>
<td>R</td>
</tr>
<tr>
<td>And Immediate</td>
<td>andi</td>
<td>I</td>
</tr>
<tr>
<td>Branch On Equal</td>
<td>beq</td>
<td>I</td>
</tr>
<tr>
<td>Branch On Not Equal</td>
<td>bne</td>
<td>I</td>
</tr>
<tr>
<td>Jump</td>
<td>j</td>
<td>J</td>
</tr>
<tr>
<td>Jump And Link</td>
<td>jal</td>
<td>J</td>
</tr>
<tr>
<td>Jump Register</td>
<td>jr</td>
<td>R</td>
</tr>
<tr>
<td>Load Byte Unsigned</td>
<td>lbu</td>
<td>I</td>
</tr>
<tr>
<td>Load Halfword Unsigned</td>
<td>lhu</td>
<td>I</td>
</tr>
<tr>
<td>Load Linked</td>
<td>li</td>
<td>I</td>
</tr>
<tr>
<td>Load Upper Imm.</td>
<td>lui</td>
<td>I</td>
</tr>
<tr>
<td>Load Word</td>
<td>lw</td>
<td>I</td>
</tr>
<tr>
<td>Nor</td>
<td>nor</td>
<td>R</td>
</tr>
<tr>
<td>Or</td>
<td>or</td>
<td>R</td>
</tr>
<tr>
<td>Or Immediate</td>
<td>ori</td>
<td>I</td>
</tr>
<tr>
<td>Set Less Than</td>
<td>slt</td>
<td>R</td>
</tr>
<tr>
<td>Set Less Than Imm.</td>
<td>slti</td>
<td>I</td>
</tr>
<tr>
<td>Set Less Than Imm. Unsigned</td>
<td>sltiu</td>
<td>I</td>
</tr>
<tr>
<td>Set Less Than Unsg.</td>
<td>sltu</td>
<td>R</td>
</tr>
<tr>
<td>Shift Left Logical</td>
<td>sll</td>
<td>R</td>
</tr>
<tr>
<td>Shift Right Logical</td>
<td>srl</td>
<td>R</td>
</tr>
<tr>
<td>Store Byte</td>
<td>sb</td>
<td>I</td>
</tr>
<tr>
<td>Store Conditional</td>
<td>sc</td>
<td>I</td>
</tr>
<tr>
<td>Store Halfword</td>
<td>sh</td>
<td>I</td>
</tr>
<tr>
<td>Store Word</td>
<td>sw</td>
<td>I</td>
</tr>
<tr>
<td>Subtract</td>
<td>sub</td>
<td>R</td>
</tr>
<tr>
<td>Subtract Unsigned</td>
<td>subu</td>
<td>R</td>
</tr>
</tbody>
</table>

Matni, CS64, Wi17
List of the Arithmetic Core Instructions in MIPS

<table>
<thead>
<tr>
<th>NAME, MNEMONIC</th>
<th>FOR-MAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch On FP True</td>
<td>bct</td>
</tr>
<tr>
<td>Branch On FP False</td>
<td>bcft</td>
</tr>
<tr>
<td>Divide</td>
<td>div</td>
</tr>
<tr>
<td>Divide Unsigned</td>
<td>divu</td>
</tr>
<tr>
<td>FP Add Single</td>
<td>add.s</td>
</tr>
<tr>
<td>FP Add Double</td>
<td>add.d</td>
</tr>
<tr>
<td>FP Compare Single</td>
<td>crs*</td>
</tr>
<tr>
<td>FP Compare Double</td>
<td>crd*</td>
</tr>
<tr>
<td>FP Divide Single</td>
<td>div.s</td>
</tr>
<tr>
<td>FP Divide Double</td>
<td>div.d</td>
</tr>
<tr>
<td>FP Multiply Single</td>
<td>mou.s</td>
</tr>
<tr>
<td>FP Multiply Double</td>
<td>mou.d</td>
</tr>
<tr>
<td>FP Subtract Single</td>
<td>sub.s</td>
</tr>
<tr>
<td>FP Subtract Double</td>
<td>sub.d</td>
</tr>
<tr>
<td>Load FP Single</td>
<td>lwcl</td>
</tr>
<tr>
<td>Load FP Double</td>
<td>ldcl</td>
</tr>
<tr>
<td>Move From Hi</td>
<td>mghi</td>
</tr>
<tr>
<td>Move From Lo</td>
<td>mflo</td>
</tr>
<tr>
<td>Move From Control</td>
<td>mfc0</td>
</tr>
<tr>
<td>Multiply</td>
<td>mull</td>
</tr>
<tr>
<td>Multiply Unsigned</td>
<td>multu</td>
</tr>
<tr>
<td>Shift Right Arith.</td>
<td>sra</td>
</tr>
<tr>
<td>Store FP Single</td>
<td>swcl</td>
</tr>
<tr>
<td>Store FP Double</td>
<td>sadcl</td>
</tr>
</tbody>
</table>
List of all Pseudoinstructions in MIPS

<table>
<thead>
<tr>
<th>PSEUDOINSTRUCTION SET</th>
<th>NAME</th>
<th>MNEMONIC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Branch Less Than</td>
<td>blt</td>
</tr>
<tr>
<td></td>
<td>Branch Greater Than</td>
<td>bgt</td>
</tr>
<tr>
<td></td>
<td>Branch Less Than or Equal</td>
<td>ble</td>
</tr>
<tr>
<td></td>
<td>Branch Greater Than or Equal</td>
<td>bge</td>
</tr>
<tr>
<td></td>
<td>Load Immediate</td>
<td>li</td>
</tr>
<tr>
<td></td>
<td>Move</td>
<td>move</td>
</tr>
</tbody>
</table>

ALL OF THIS AND MORE IS ON YOUR HANDY “MIPS REFERENCE CARD” FOUND ON THE FRONT PAGE OF THE CLASS WEBSITE
MIPS Peculiarity: NOR used a NOT

• How to make a NOT function using **NOR** instead
• NOR = NOT OR
• Truth-Table:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>A NOR B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Note that:

0 NOR \( x \) = NOT \( x \)

• So, in the absence of a NOT function, use a NOR with a 0 as one of the inputs
Let’s Run This Program Already! Using SPIM

- We’ll call it `add2.asm`
- Run it on CSIL as: `$ spim -f add2.asm`

- We’ll also run other arithmetic programs and explain them as we go along
  – TAKE NOTES!
YOUR TO-DOs

• Tuesday (1/24): MIPS memory
  – Do the reading (check the syllabus!)

• Lab 2 is this Monday (1/23)

• Assignments #2 **AND** #3
  – Assignment #2: Due Monday 1/30
  – Assignment #2: Due Friday 2/3
  – I RECOMMEND THAT YOU PARTNER UP!