CPSC 330
Computer Organization

Lecture 2c - Instructions: Language of the computer

Adapted from CS152, CS 61C lecture notes and notes by Morgan Kaufmann Publishers, Copyright 2004.

---

Instruction Format - Review

<table>
<thead>
<tr>
<th>R</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS Instructions - Review

- **Instruction**
- **Meaning**
  - `add $s1,$s2,$s3` $s1 = $s2 + $s3
  - `sub $s1,$s2,$s3` $s1 = $s2 - $s3
  - `addi $s1,$s1,4` $s1 = $s1 + 4
  - `slt $t0,$s0,$s1` if ($s0 < $s1) then $t0 gets 1 otherwise $t0 gets 0
  - `lw $s1,100($s2)` $s1 = Memory[$s2+100]
  - `sw $s1,100($s2)` Memory[$s2+100] = $s1
  - `bne $s4,$s5,L` Next instr. is at Label if $s4 ≠ $s5
  - `beq $s4,$s5,L` Next instr. is at Label if $s4 = $s5
  - `jr $t1` jump via register: go to the address specified by $t1
  - `j Label` go to the target address
  - `mult $t1,$t2` {Hi,Lo} = $t1*$t2
  - `div $t1,$t2` Lo = $t1/$t2; Hi = $t1%$t2
  - `mflo $a0` move from Lo to $s0

MIPS Register Convention

- **Name**
- **Register number**
- **Usage**
  - `$zero` 0 the constant value 0
  - `$v0-$v1` 2-3 values for results and expression evaluation
  - `$a0-$a3` 4-7 arguments
  - `$t0-$t7` 8-15 temporaries
  - `$s0-$s7` 16-23 saved
  - `$t8-$t9` 24-25 more temporaries

---
R - Format Example - Review

- MIPS Example
  sub $t1, $s1, $s2
  opcode = 0 (lookup in table)
  funct = 34 (lookup in table)
  rs = 17 (first operand)
  rt = 18 (second operand)
  rd = 8 (destination)
  shamt = 0 (no shift)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>17</th>
<th>18</th>
<th>8</th>
<th>0</th>
<th>34</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>000000</td>
<td>10001</td>
<td>10010</td>
<td>01000</td>
<td>00000</td>
<td>100010</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
</table>

I - Format Example - Review

- MIPS Example
  lw $t0, 32($s2) # $t0 = addr $s2 + 32 (offset)
  opcode = 35 (lookup in table)
  rs = 18 (first operand)
  rt = 9 (second operand)

<table>
<thead>
<tr>
<th></th>
<th>35</th>
<th>18</th>
<th>9</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100011</td>
<td>10010</td>
<td>01001</td>
<td>0000000000100000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16-bit number</th>
</tr>
</thead>
</table>

J - Format Example - Review

- MIPS Example
  j Loop # go to location Loop
  No register operands.
  Operand destination address is signed word offset from PC.

<table>
<thead>
<tr>
<th></th>
<th>000010</th>
<th>00000000000000000000100000</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>op</th>
<th>26-bit number</th>
</tr>
</thead>
</table>

Exercise

Scoreboard the operation of the program by single-stepping through the MIPS code and recording results in the trace table. Then, go back and annotate the instructions by adding meaningful comments on the end of each line of the program. Describe in one sentence what the program has done.

<table>
<thead>
<tr>
<th>Step</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400:a</td>
<td>addi $t0, $zero, 3</td>
<td></td>
</tr>
<tr>
<td>0x404:b</td>
<td>add $v0, $t0, $t0</td>
<td></td>
</tr>
<tr>
<td>0x408:c</td>
<td>addi $t0, $t0, -1</td>
<td></td>
</tr>
<tr>
<td>0x410:d</td>
<td>bne $t0, $zero, b</td>
<td></td>
</tr>
<tr>
<td>0x410:e</td>
<td>j</td>
<td></td>
</tr>
</tbody>
</table>

Trace Table

<table>
<thead>
<tr>
<th>Step</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<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></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></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>
### Addressing Modes

Everything has an address

- Computers need to know
  - Where to go next?
  - Where to get data?

- Program execution either...
  - Continues sequentially or...
  - Branches (near) or jumps (far) somewhere else.

- Programs get data either from...
  - Registers (fast and inside CPU) or...
  - Memory (maybe near, maybe far)...
  - (or sometimes directly from IO devices).

### MIPS assembler

What is an assembler?

- An assembler translates...
  “assembly language”
  (human readable instructions)
  into...
  “machine language”
  (binary instructions, “1”s & “0”s).

- It’s a program, similar to a compiler, only it uses a language specific to a computer’s ISA.

### Decoding Machine Language

What is the assembly language statement corresponding to the this machine instruction: \((00af8020)_{16}\)?

**Fields:**

- Op: 6 bits
- rs, rt, rd, shamt: 5 bits
- Funct: 6 bits

**Answer:** `add $s0, $a1, $t7`

### MIPS Assembler

How does it work?

- It reads an input text file (a program written in assembly language), then it parses a line of program at a time and “assembles” them into machine code instructions.

- Lines contain either:
  - Single instructions
  - Assembler “directives”
  - Comments (lines preceded with “#” or just blank)

- It produces an output binary file that contains machine code, data, and startup info.
Directives tell the assembler to do something “else” besides assembling.

- `.text <address>`: Store subsequent program instructions or static data at address specified (if present) in the text segment
- `.data <address>`: Indicates that subsequent data items should be put in data segment

### More “directives”

- `.align <n>`: Aligns next data to next natural boundary “n”, for `.half(2)`, `.word(4)`, `.double(8)`, `.float(8)`, not necessary for `.byte(1)`
- `.byte <b>,…`: Store bytes in data segment
- `.half <h>,…`: Store half-words (16bit)
- `.word <w>,…`: Store words (32bit)
- `.double <d>,…`: Store double words (64bit)
- `.float <f>,…`: Store floating point numbers
- `.ascii <string>`: Store string in memory
- `.asciiz <string>`: Store string zero terminated

### More “directives”

- `.extern <sym> <size>`: Allocates space for the symbol in global space
- `.global <sym>`: Defines a global symbol.

---

**Assembler & SPIM: Starting a program**

1. **C program**
2. **Compiler**
3. **Assembly language program**
4. **Object: Machine language module**
5. **Linker**
6. **Linked Object: Machine language program**
7. **Loader**
8. **Exe file: Machine language program**

---

**MIPS Assembler**

*What are “directives”?*

- Directives tell the assembler to do something “else” besides assembling.
- `.text <address>`: Store subsequent program instructions or static data at address specified (if present) in the text segment
- `.data <address>`: Indicates that subsequent data items should be put in data segment
The CD that accompanies the book contains Software: HDL simulators, MIPS simulator, and FPGA design tools.

Tutorials: SPIM, Verilog, and VHDL.

Content/Software/Spim/Pcspim

Using SPIM

- Simulator -> Settings…:
  In the Display section check only the first two items Save window positions and General registers in hexadecimal

- Simulator -> Set Value…: to load PC with address of first instruction enter Address or Register Name as "PC" and enter Value as "0x00400000"
  Reason: the text area of memory, where programs are stored, starts here.

- Simulator -> Go : run loaded program → F5
  Click the OK button in the Run Parameters pop-up window if the StartingAddress value is "0x00400000"

- Simulator -> Break : stop execution

- Simulator -> Clear Registers and Reinitialize : clean-up before new run

Using SPIM

- Simulator → Reload : load file again after editing
- Simulator → Single Step (F10) or Multiple Step (F11): stepping to debug
- Simulator → Breakpoints: set breakpoints
Arithmetic Example for SPIM

# Using SPIM, write and debug a program to input two numbers, then computes and prints their sum and their difference. Use the SPIM system calls read_int (5, $a0), print_int (1, $v0).

# Inputs:
# single precision floating values after syscall
# Outputs:
# sum, diff.
# Register Usage:
#$ t 2  =  arg1
#$ t 4  =  arg1
#$v0 = syscall function code
#$a0 = syscall string pointer
#
.text
.globl main # Entry point for SPIM linkage.

Example (continued)

compute:
 li $v0, 4 # Load syscall code for print_str.
la $a0, sumstr # Load address of string.
syscall # Print string.
li $v0, 4 # Load syscall code for print_str.
la $a0, difstr # Load address of string.
syscall # Print string.
li $v0, 1 # Load syscall code for print_int.
syscall # Print
la $a0, $t2, $t4 # Add them.
syscall # Load syscall code for print_int.
li $v0, 4 # Subtract them
la $a0, $t2, $t4 # Load syscall code for print_int.
syscall # Print

Example (continued)

main: nop
li $v0, 4 # Load syscall code for print_str.
la $a0, askstr1 # Load address of string
syscall # Print string
li $v0, 5 # Load syscall code for read
syscall # Get arg1
li $v0, 4 # Load syscall code for print_str.
la $a0, askstr2 # Load address of string
syscall # Print string
li $v0, 5 # Load syscall code for read
syscall # Get arg2
move $t4, $v0 # Save arg2
li $v0, 4 # Load syscall code for print_str.
la $a0, askstr1 # Load address of string
syscall # Print string
li $v0, 5 # Load syscall code for read
syscall # Get arg1
move $t2, $v0 # Save arg1
li $v0, 4 # Load syscall code for print_str.
la $a0, askstr2 # Load address of string
syscall # Print string
li $v0, 5 # Load syscall code for read
syscall # Get arg2

compute:
 li $v0, 4 # Load syscall code for print_str.
la $a0, sumstr # Load address of string.
syscall # Print string.
li $v0, 4 # Load syscall code for print_str.
la $a0, difstr # Load address of string.
syscall # Print string.
li $v0, 1 # Load syscall code for print_int.
syscall # Print
la $a0, $t2, $t4 # Add them.
syscall # Load syscall code for print_int.
li $v0, 4 # Subtract them
la $a0, $t2, $t4 # Load syscall code for print_int.
syscall # Print

Example (continued)

double floating point registers

Example (continued)

save log file (ctrl+s) in file
Next time...

- Wrapping-up chapter 2
- Branching and Looping
- Alternative Architectures
- IA-32