IC220  
SlideSet #3: Control Flow  
(Section 2.7, plus pgs 86-87, 128-129,140-141)

**Conditional Control**

- Decision making instructions  
  - alter the control flow,  
  - i.e., change the "next" instruction to be executed

- MIPS *conditional* branch instructions (I – type):
  
  `bne $t0, $t1, Label`
  `beq $t0, $t1, Label`

- Example:  
  
  \[
  \text{if (i == j)} \\
  \text{h = i + j;}
  \]

- Assembly Code:
  
  ```
  bne $s0, $s1, Label
  add $s3, $s0, $s1
  Label: ....
  ```

**Unconditional Control**

- MIPS unconditional branch instructions:

  ```
  j label
  ```

- New type of instruction (*J*-type)  
  - op code is 2 (no function field)

- Example:
  
  ```
  if (i!=j)  
  beq $s4, $s5, Lab1
  h=$i+$j;  
  add $s3, $s4, $s5
  else  
  j Lab2
  h=$i-$j;  
  Lab1: sub $s3, $s4, $s5
  Lab2: ...
  ```
Example

• What is the MIPS assembly code for the following:
  if (i == j)  f = g + h;
  else         f = g - h;

Variables f to j are assigned to registers $s0 to $s4

So far:

• Instruction  Meaning
  add $s1,$s2,$s3   $s1 = $s2 + $s3
  sub $s1,$s2,$s3   $s1 = $s2 - $s3
  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
  j Label          Next instr. is at Label

• Formats:
  R | op | rs | rt | rd | shamt | funct
  I | op | rs | rt | 16 bit address
  J | op | 26 bit address

Control Flow – Branch if less than

• We have: beq, bne, what about Branch-if-less-than?
• New instruction:
  \[
  \text{if} \quad \text{\$s1 < \$s2 then} \\
  \quad \text{\$t0 = 1} \\
  \quad \text{slt \$t0, \$s1, \$s2} \\
  \text{else} \\
  \quad \text{\$t0 = 0}
  \]

• slt is a R-type instruction (function code 42)

Example

• What is the MIPS assembly code to test if variable a (\$s0) is less than variable b (\$s1) and then branch to Less: if the condition holds?

  if (a < b)
  \quad \text{go to Less;}
  \quad \text{....}
  \text{Less: \text{....}}

**Pseudoinstructions**

- Example #1: Use `slt` instruction to build "`blt $s1, $s2, Label`"
  - "Pseudoinstruction" that assembler expands into several real instructions
  - Note that the assembler needs a register to do this
  - What register should it use?
  - Why not make `blt` a real instruction?

- Example #2: "Move" instruction
  - "move $t0, $t1"
  - Implementation?

**Policy of Use Conventions**

<table>
<thead>
<tr>
<th>Name</th>
<th>Register number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>the constant value 0</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>values for result and expression evaluation</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>arguments</td>
</tr>
<tr>
<td>$t0-$t9</td>
<td>8-15</td>
<td>temporaries</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>saved</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>more temporaries</td>
</tr>
<tr>
<td>$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>

Sat = Register #1 – reserved for assembler
$K0, $K1 = Register #26, 27 – reserved for OS

**Constants**

- Small constants are used quite frequently
  - e.g., $A = A + 5;
  - $B = B + 1;
  - $C = C - 18;
- Possible solution
  - put 'typical constants' in memory and load them.
  - And create hard-wired registers for constants like zero, one.
- Problem?

- MIPS Instructions:
  - `add $t0, $t0, 4`
  - `and $t0, $t0, 6`
  - `ori $t0, $t0, 4`

- How do we make this work?

**How about larger constants?**

- We’d like to be able to load a 32 bit constant into a register
- Must use two instructions, new "load upper immediate" instruction
  - `lui $t0, 1010101010101010`
  - `ori $t0, $t0, 0000000000111111`
- Then must get the lower order bits right, i.e.,
  - `ori $t0, $t0, 0000000000000000`
  - `ori $t0, $t0, 0000000000001111`
Assembly Language vs. Machine Language

- Assembly provides convenient symbolic representation
  - much easier than writing down numbers
  - e.g., destination first
- Machine language is the underlying reality
  - e.g., destination is no longer first
- Assembly can provide 'pseudoinstructions'
- When considering performance you should count

Memory – Byte Order & Alignment

- **Endian**
  - Processors don’t care
  - Big: 0,1,2,3
  - Little: 3,2,1,0
  - Network byte order:

- **Alignment**
  - require that objects fall on address that is multiple of
  - Legal word addresses:
  - Legal byte addresses:

Looping

- We know how to make decisions, but:
  - Can we set up a flow that allows for multiple iterations?
  - What high level repetition structures could we use?
  - What MIPS instructions could we use?

Looping Example

**Goal:** Provide the comments # to the assembly language

**C Code**

```c
do {
    g = g + A[i];
    i = i + j;
} while (i < h)
```

**Assembly Language**

```assembly
Loop:
    add $t1, $s3, $s3  #
    add $t1, $t1, $t1  #
    add $t1, $t1, $s5  #
    lw $t0, 0($t1)  #
    add $s1, $s1, $t0  #
    add $s3, $s3, $s4  #
    bne $s3, $s2, Loop  #
```

Ex 2-21 to 2-23