Decision making instructions
– alter the control flow,
– i.e., change the “next” instruction to be executed

MIPS conditional branch instructions (I–type):
\[
\begin{align*}
  \text{bne} & \quad $t0, $t1, \text{ Label} \\
  \text{beq} & \quad $t0, $t1, \text{ Label}
\end{align*}
\]

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

Assembly Code:
\[
\begin{align*}
  \text{bne } & \quad $s0, $s1, \text{ Label} \\
  \text{add} & \quad $s3, $s0, $s1 \\
  \text{Label:} & \quad ...
\end{align*}
\]

MIPS unconditional branch instructions:
\[
  \text{j} \quad \text{label}
\]

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

Example:
\[
\begin{align*}
  \text{if } (i! = j) \quad & \quad \text{beq } $s4, $s5, \text{Lab1} \\
  h & = i + j; \quad \text{add } $s3, $s4, $s5 \\
  \text{else} \quad & \quad \text{j} \quad \text{Lab2} \\
  h & = i - j; \quad \text{Lab1:} \quad \text{sub } $s3, $s4, $s5 \\
  \text{Lab2:} & \quad ...
\end{align*}
\]
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
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $s1,$s2,$s3</td>
<td>$s1 = $s2 + $s3</td>
</tr>
<tr>
<td>sub $s1,$s2,$s3</td>
<td>$s1 = $s2 - $s3</td>
</tr>
<tr>
<td>lw $s1,100($s2)</td>
<td>$s1 = Memory[$s2+100]</td>
</tr>
<tr>
<td>sw $s1,100($s2)</td>
<td>Memory[$s2+100] = $s1</td>
</tr>
<tr>
<td>bne $s4,$s5,L</td>
<td>Next instr. is at Label if $s4 != $s5</td>
</tr>
<tr>
<td>beq $s4,$s5,L</td>
<td>Next instr. is at Label if $s4 == $s5</td>
</tr>
<tr>
<td>j Label</td>
<td>Next instr. is at Label</td>
</tr>
</tbody>
</table>

- Formats:

<table>
<thead>
<tr>
<th>Format</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td></td>
<td>16 bit address</td>
</tr>
<tr>
<td>J</td>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>26 bit address</td>
</tr>
</tbody>
</table>

Control Flow – Branch if less than

- We have: beq, bne, what about Branch-if-less-than?
- New instruction:
  
  ```
  if  $s1 < $s2 then
   $t0 = 1
  else
   $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)
   go to Less;
  ```

- Less:
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 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</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
$\text{sk}0, \text{sk}1 = \text{Register #26, 27} – \text{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:
  - addi $29, $29, 4
  - slti $8, $18, 10
  - andi $29, $29, 6
  - ori $29, $29, 4
- How do we make this work?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>lui</td>
<td></td>
<td>$t0, 1010101010101010</td>
</tr>
<tr>
<td>addi</td>
<td></td>
<td>$t0, 0000000000000000</td>
</tr>
<tr>
<td>ori</td>
<td></td>
<td>$t0, 0000000000000000</td>
</tr>
</tbody>
</table>

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

- Then must get the lower order bits right, i.e.,
  ori $t0, $t0, 0000000000000000
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

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?
- “Basic block”
  - Sequence of instructions __________________________
    except possibly __________________________-

Looping Example

Goal: Provide the comments # to the assembly language

C Code
do {
  g = g + A[i];  // vars g to j in $s1 to $s4
  i = i + j;    // $s5 holds base add of A
} while (i < h)

Assembly Language
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 #