Review

What is a pseudo-instruction?
What true MIPS instruction is equivalent to:

- **move $t0, $s1**?

- **Assume Registers:**

<table>
<thead>
<tr>
<th>0</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>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Explain what the following instructions do …

- `sb $t0, 4($1)`
- `lw $t1, 2($2)`
- `sh $t2, -500($2)`

**Load Byte:** `lb reg, addr`  
```
? ? ? [M[addr]]
```

From Last Time: MIPS Control

- **MIPS’s control-flow instructions**
  - `j` # for unconditional jumps
  - `bne` and `beq` # for conditional branches
  - `slt` and `slti` # set if less than (w/o and w/ immediate)

- **As in**
  - `j line_label`  
  - `bne $4, $7, line_label` #skip to next part
  - `slt $4, $7, $8` #test $7 less than $8

- **For example, compute |$8| … first test, then branch**
  - `slt $9, $8, $0` #set $9 to 1 if $8 < 0
  - `beq $9, notNeg` #branch if $9 not set
  - `sub $8, $0, $8` #flip sign

  **notNeg:**
Pseudo-Branches

• The MIPS processor only supports two branch instructions, beq and bne, but to simplify your life the assembler provides the following other branches:

  \[
  \begin{align*}
  \text{blt} & \; \text{t0}, \; \text{t1}, \; \text{Lab1} & \# \text{Branch if } \text{t0} < \text{t1} \\
  \text{ble} & \; \text{t0}, \; \text{t1}, \; \text{Lab2} & \# \text{Branch if } \text{t0} <= \text{t1} \\
  \text{bgt} & \; \text{t0}, \; \text{t1}, \; \text{Lab3} & \# \text{Branch if } \text{t0} > \text{t1} \\
  \text{bge} & \; \text{t0}, \; \text{t1}, \; \text{Lab4} & \# \text{Branch if } \text{t0} >= \text{t1}
  \end{align*}
  \]

• There are also immediate versions of these branches, where the second source is a constant instead of a register

• Later this quarter we’ll see how supporting just beq and bne simplifies the processor design

Implementing Pseudo-Branches

• Most pseudo-branches are implemented using slt. Consider a branch-if-less-than instruction

  \[
  \begin{align*}
  \text{blt} & \; \text{a0}, \; \text{a1}, \; \text{Label} & \text{is translated into} \\
  \text{slt} & \; \text{at}, \; \text{a0}, \; \text{a1} & // \text{at} = 1 \text{ if } \text{a0} < \text{a1} \\
  \text{bne} & \; \text{at}, \; \text{0}, \; \text{Label} & // \text{Branch if } \text{at} != 0
  \end{align*}
  \]

• This supports immediate branches, which are also pseudo-instructions. For example,

  \[
  \begin{align*}
  \text{blti} & \; \text{a0}, \; 5, \; \text{Label} & \text{is translated into} \\
  \text{slti} & \; \text{at}, \; \text{a0}, \; 5 & // \text{at} = 1 \text{ if } \text{a0} < 5 \\
  \text{bne} & \; \text{at}, \; \text{0}, \; \text{Label} & // \text{Branch if } \text{a0} < 5
  \end{align*}
  \]

• All pseudo-branches need a register to save the result of slt, even though it’s not needed afterwards

  ❖ MIPS assemblers use register $1, or $at, for temporary storage.

  ❖ You should be careful in using $at in your own programs, as it may be overwritten by assembler-generated code.
Register Correspondences: First View

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>$0</td>
<td>Zero</td>
</tr>
<tr>
<td>$at</td>
<td>$1</td>
<td>Assembler temp</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>$2-3</td>
<td>Value (return from fcn)</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>$4-7</td>
<td>Argument (to fcn)</td>
</tr>
<tr>
<td>$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>$26-27</td>
<td>Kernel (OS) Registers</td>
</tr>
<tr>
<td>$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>

Translating an if-then Statement

We can use branch instructions to translate if-then statements into MIPS assembly code

\[
\begin{align*}
&v0 = a0; \\
&\text{if (v0 < 0)} \\
&\quad v0 = -v0; \\
&\quad v1 = v0 + v0;
\end{align*}
\]

\[
\begin{align*}
&\text{move $v0, $a0} \\
&\text{bge $v0, $0, Label} \\
&\text{sub $v0, $0, $v0} \\
&\text{Label: add $v1, $v0,$v0}
\end{align*}
\]

Sometimes it’s easier to invert the original condition.

- In this case, we changed “continue if v0 < 0” to “skip if v0 >= 0”.
- This saves one or two instructions in the resulting assembly code.
If-Then-Else

In an If-Then-Else there must be branching to the else and around the else
Increase the magnitude of v0 by one

\[
\text{if (v0 < 0)} \\
\quad v--; \\
\text{else} \\
\quad v++; \\
\text{v1 = v0;}
\]

\[
\begin{align*}
\text{bge } & \$v0, \$\text{zero, } E \\
\text{subi } & \$v0, \$v0, 1 \\
\text{j L} \\
\text{E: addi } & \$v0, \$v0, 1; \\
\text{L: move } & \$v1, \$v1;
\end{align*}
\]

What Does This Code Do?

\[
\begin{align*}
\text{label: sub } & \$a0, \$a0, 1 \\
\text{bne } & \$a0, \$\text{zero, label}
\end{align*}
\]
### Encoding Loop Structure

```c
define loop
for (i = 0; i < 4; i++) {
    // stuff
}
    add $t0, $zero, $zero  # initialize i to 0 $t0 = 0
Loop:   slti $t1, $t0, 4              # $t1 = 1 if i < 4
    beq $t1, $zero, EoL     # Exit if i >= 4
    // stuff
    addi $t0, $t0, 1              # i ++
    j Loop           #continue?
EoL:
```

### Computing With A Loop

Let's write a program to count the 1 bits in a 32-bit word

```c
int count = 0;
for (int i = 0 ; i < 32 ; i ++) {
    int bit = input & 1;
    if (bit != 0) {
        count ++;
    }
    input = input >> 1;
}
```

```c
text main:
    ...  
    li $t0, 0                         ## int count = 0;
    li $t1, 0                         ## for (int i = 0
    ...  
main_loop:
    bge $t1, 32, main_exit ## exit loop if i >= 32
    andi $t2, $a0, 1            ## bit = input & 1
    beq $t2, $0, main_skip ## skip if bit == 0
    addi $t0, $t0, 1              ## count ++
main_skip:
    srl $a0, $a0, 1              ## input = input >> 1
    add $t1, $t1, 1              ## i ++
    j main_loop
main_exit:
    ...  
```
Assembly vs. machine language

- So far we’ve been using assembly language.
  - We assign names to operations (e.g., add) and operands (e.g., $t0)
  - Branches and jumps use labels instead of actual addresses
  - Assemblers support many pseudo-instructions
- Programs must eventually be translated into machine language, a binary format that can be stored in memory and decoded by the CPU
- MIPS machine language is designed to be easy to decode
  - Each MIPS instruction is the same length, 32 bits
  - There are only three different instruction formats, which are very similar to each other
- Studying MIPS machine language will also reveal some restrictions in the instruction set architecture, and how they can be overcome.

R-type format

- Register-to-register arithmetic instructions are R-type

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

- This format includes six different fields
  - op is an operation code or opcode that selects a specific operation
  - rs and rt are the first and second source registers
  - rd is the destination register
  - shamt is “shift amount” and is only used for shift instructions
  - func is used together with op to select an arithmetic instruction
- See the text’s inside back cover or the Green Card for opcodes and function codes for all MIPS instructions
About the registers

- We have to encode register names as 5-bit numbers from 00000 to 11111
  - For example, $t8$ is register $24$, which is represented as 11000
  - The complete mapping is given on page B-24 in the book
- The number of registers available affects the instruction length
  - Each R-type instruction references 3 registers, which requires a total of 15 bits in the instruction word
  - We can't add more registers without either making instructions longer than 32 bits, or shortening other fields like op and possibly reducing the number of available operations

I-type format

- Load, store, branch, & immediate instructions are I-type

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

- For uniformity, op, rs and rt are in the same positions as in the R-format
- The meaning of the register fields depends on the instruction
  - rs is a source register—an address for loads and stores, or an operand for branch and immediate arithmetic instructions
  - rt is a source register for branches, but a destination register for the other I-type instructions
- The address is a 16-bit signed two’s-complement value
  - Its range is [-32,768, +32,767]
  - But that's not always enough!
Larger constants

- Larger constants can be loaded 16 bits at a time
  - The load upper immediate instruction `lui` loads the highest 16 bits of a register with a constant, and clears the lowest 16 bits to 0s
  - An immediate logical OR, `ori`, then sets the lower 16 bits
- To load 32-bit value `0000 0000 0011 1101 0000 1001 0000 0000`:
  ```
  lui $s0, 0x003D
  ori $s0, $s0, 0x0900
  # $s0 = 003D 0900
  ```

- This illustrates a Principle: Make the common case fast
  - Most of the time, 16-bit constants are enough
  - It's still possible to load 32-bit constants, but at the cost of two instructions and one temporary register
- Pseudo-instructions may contain large constants, which the assembler translates correctly