Review

What is a pseudo-instruction?

What true MIPS instruction is equivalent to:

- ❖ `move $t0, $s1`?

- Assume Registers:

0	0
1	1
2	2000
…	

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]

1

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:
```

2

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:

```
blt $t0, $t1, Lab1      # Branch if $t0 < $t1
ble $t0, $t1, Lab2      # Branch if $t0 <= $t1
bgt $t0, $t1, Lab3      # Branch if $t0 > $t1
bge $t0, $t1, Lab4      # Branch if $t0 >= $t1
```

- 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

3

Implementing Pseudo-Branches

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

```
blt $a0, $a1, Label        is translated into
  slt $at, $a0, $a1          // $at = 1 if $a0 < $a1
  bne $at, $0, Label         // Branch if $at != 0
```

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

```
blti $a0, 5, Label         is translated into
  slti $at, $a0, 5           // $at = 1 if $a0 < 5
  bne $at, $0, Label         // Branch if $a0 < 5
```

- 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.

4

Register Correspondences: First View

$zero	$0	Zero	
$at	$1	Assembler temp	
$v0-$v1	$2-3	Value (return from fcn)	
$a0-$a3	$4-7	Argument (to fcn)	
$t0-$t7	$8-15	Temporaries	
$s0-$s7	$16-23	Saved Temporaries	Saved
$t8-$t9	$24-25	Temporaries	
$k0-$k1	$26-27	Kernel (OS) Registers	
$gp	$28	Global Pointer	Saved
$sp	$29	Stack Pointer	Saved
$fp	$30	Frame Pointer	Saved
$ra	$31	Return Address	Saved

5

Translating an if-then Statement

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

```
v0 = a0;
if (v0 < 0)
    v0 = -v0;
v1 = v0 + v0;
```

```
        move $v0, $a0
        bge  $v0, $0, Label
        sub  $v0, $0, $v0
Label:  add  $v1, $v0,$v0
```

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.

```
        move $v0, $a0
        blt  $v0, $0, L1
        j    Label
L1:     sub  $v0, $0, $v0
Label:  add  $v1, $v0,$v0
```

6

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

```
if (v0 < 0)
   v--;
else
   v++;
v1 = v0;
```

```
      bge  $v0, $zero, E
      subi $v0, $v0, 1
      j L
E: addi $v0, $v0, 1;
L: move $v1, $v1;
```

7

What Does This Code Do?

```
label: sub $a0, $a0, 1
       bne $a0, $zero, label
```

8

Encoding Loop Structure

```
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:
```

9

Computing With A Loop

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

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

```
.text main:                    ## arg in $a0
        …
        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:
        …
```

10

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.

11

R-type format

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

op	rs	rt	rd	shamt	func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- 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

12

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

13

I-type format

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

op	rs	rt	address
6 bits	5 bits	5 bits	16 bits

- For uniformity, op, rs and rt are in the same positions as in the R-format
- The meaning of the register fields depends on 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!

14

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          # $s0 = 003D 0000 (in hex)
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

15