Computer Architecture
(ENE1004)

Lec – 4: Instructions: Language of the Computer (Chapter 2) - 3

Review: Representing MIPS Instructions

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
addi	I	8	18	17	100			addi $s1,$s2,100
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

- R-type and I-type
- Each type has different fields with fixed sizes
- Opcode (+funct) is unique for each operation
- Registers as source/destination operands are specified their numbers
- Notice: the order of register fields is different from that in an instruction

Review: Example

- Assume that $t1 has the base address of A and $s2 corresponds to h
- **A[300] = h + A[300];** is compiled to

```
lw  $t0,  1200($t1)    # temporary reg $t0 gets A[300]
add  $t0,  $s2,  $t0   # temporary reg $t0 gets h + A[300]
sw  $t0,  1200($t1)    # store h + A[300] back into A[300]
```

Op	rs	rt	rd	address/ shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

MIPS Instructions for Logical Operations: Shifts

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl

- Moving all the bits in a word to the left or right, filling the emptied bits with 0s
  - Shift left (**sll**) and shift right (**srl**)
  - Example of shifting 9 by 4
    - 9 = 0000 0000 0000 0000 0000 0000 0000 1001
    - (To the left) 9 << 4 = 0000 0000 0000 0000 0000 0000 1001 0000 = 144
    - (To the right) 9 >> 4 = 0000 0000 0000 0000 0000 0000 0000 0000 = 0
- Example: **sll $t2, $s0, 4 # reg $t2 = reg $s0 << 4 bits**
  - The original value is in $s0 and the result of shifting left by 4 goes into $t2

	op	rs	rt	rd	shamt	funct
R-type	0	0	16	10	4	0

  - (*op*=0 + *funct*=0) for **sll**, *rd*=10 for $t2, *rt*=16 for $s0, and *shamt*=4 (*rs* is unused and set to 0)

MIPS Instructions for Logical Operations: AND/OR

Logical operations	C operators	Java operators	MIPS instructions
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit NOT	~	~	nor

- Example
  - $t2 = 0000 0000 0000 0000 0000 1101 1100 0000 = 3,520
  - $t1 = 0000 0000 0000 0000 0011 1100 0000 0000 = 15,360
- AND: sets 1 in the bit only if both the corresponding bits of the operands are 1
  - **and $t0, $t1, $t2 # reg $t0 = reg $t1 & reg $t2**
  - $t0 = 0000 0000 0000 0000 0000 1100 0000 0000 = 3,072
- OR: sets 1 in the bit if either operand bit is a 1
  - **or $t0, $t1, $t2 # reg $t0 = reg $t1 | reg $t2**
  - $t0 = 0000 0000 0000 0000 0011 1101 1100 0000 = 15,808
- Immediate instructions are supported for AND and OR
  - **andi** (and immediate), **ori** (or immediate)

MIPS Instructions for Logical Operations: NOR

Logical operations	C operators	Java operators	MIPS instructions
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit NOT	~	~	nor

- NOT: takes one operand and places a 1 in the bit if one operand bit is a 0, and vice versa
  - MIPS designers did their best for keeping with the three-operand format
  - MIPS designers decided to include NOR (NOT OR) instead of NOT
  - A NOR 0 = NOT (A OR 0) = NOT (A)
  - If one operand is zero, NOR is equivalent to NOT
- Example: **nor $t0, $t1, $t3 # reg $t0 = ~ (reg $t1 | reg $t3)**
  - $t1 = 0000 0000 0000 0000 0011 1100 0000 0000 = 15,360
  - $t3 = 0000 0000 0000 0000 0000 0000 0000 0000 = 0
  - $t1 OR $t3 = 0000 0000 0000 0000 0011 1100 0000 0000
  - ~($t1 OR $t3) = 1111 1111 1111 1111 1100 0011 1111 1111 = NOT $t1
- Immediate NOR is not supported, since its main use is to invert the bits (NOT)

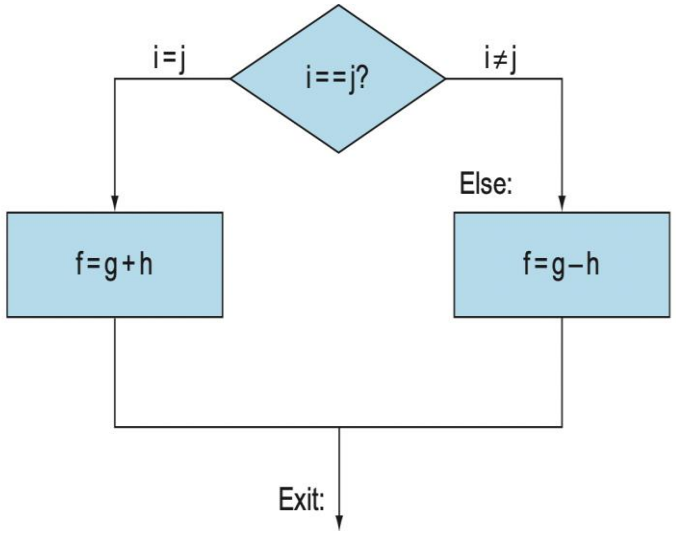
MIPS Decision-making Instructions

- Computers have an ability to make decisions
  - Based on input data or values created during computation, different instructions are executed
  - Decision making is commonly represented in programming languages using if statements
  - MIPS includes two decision-making instructions
- Branch if equal: **beq register1, register2, L1**
  - L1 is a label that indicates a statement in the code; recall goto statement in C
  - It jumps to the statement labeled L1 if the value in register1 equals the value in register2
- Branch if not equal: **bne register1, register2, L1**
  - It jumps to the statement labeled L1 if the value in register1 does not equal the value in register2
- We call the two instructions “conditional branches”
- “Unconditional branch” jump: **j L1**
  - There is no condition
  - It always jumps to the statement labeled L1

MIPS Decision-making Instructions: **if-else**

- **if (i == j) f = g + h; else f = g - h;**
  - f corresponds to $s0, g to $s1, h to $s2, i to $s3, j to $s4
  - is translated into

```
  bne  $s3,  $s4,  Else    # go to Else if i ≠ j
  add  $s0,  $s1,  $s2     # f = g + h (skipped if i ≠ j)
  j  Exit                  # go to Exit
Else:                      # here is the label Else
  sub  $s0,  $s1,  $s2     # f = g - h (skipped if i = j)
Exit:                      # here is the label Exit
```



  - Without the unconditional branch (j Exit), both add and sub are executed
- An alternative: **beq $s3, $s4, Then # go to Then if i = j**
  - If you like this, where do you locate the case of where i ≠ j?
  - In general, the code will be more efficient if we test for the opposite condition to branch over the code that performs the subsequent *then* part of the if

MIPS Decision-making Instructions: **while**

- Decisions are also important for iterating a computation, which are found in loops
- **while (save[i] == k) i += 1;**
  - i corresponds to $s3, k to $s5, the base of the array save is in $s6
  - is translated into

```
Loop:
   sll   $t1,  $s3,  2      # temp reg $t1 = i *4 (for the index in bytes)
   add  $t1,  $t1,  $s6     # $t1 = address of save[i]
   lw    $t0,  0($t1)       # temp reg $t0 = save[i]
   bne  $t0,  $s5,  Exit    # go to Exit if save[i] ≠ k
   addi $s3,  $s3,  1       # i = i + 1
   j  Loop                  # go to Loop
Exit:                       # here is the label Exit
```

- Notice: Shift left logical “m” by “n” bits generates m * 2^n

MIPS Decision-making Instructions: **for**

- **for (i = 0; i < 4; i++) { //do something }**
  - Do you have any idea for handling "i < 4"?
  - It would be useful to see if a variable is less than another variable
- Two useful instructions, which can tell if a variable is less than another variable
  - Set less than: **slt reg1, reg2, reg3 # reg1 = 1 if reg2 < reg3**
    - It compares two registers (reg2 and reg3), and sets a register (reg1) to 1 if reg2 < reg3
    - Otherwise (if reg2 >= reg3), it sets reg1 to 0
  - Set less than immediate: **slti reg1, reg2, const #reg1=1 if reg2<const**
    - Immediate version of slt
    - It compares reg2 and constant, and sets a register (reg1) to 1 if reg2 < constant
- **for (i = 0; i < 4; i++) { //do something }**
  - Do you have any idea for initializing a variable "i = 0"?
  - MIPS includes a register, named "$zero", which always holds a value of 0
  - **add $s0, $zero, $zero # $s0 will hold a value of 0**

register	assembly name	Comment
r0	$zero	Always 0

MIPS Decision-making Instructions: **for**

- **for (i = 0; i < 4; i++) { //do something }**
  - i corresponds to $t0
  - is translated into

```
   add  $t0,  $zero,  $zero  # i is initialized to 0, $t0 = 0
Loop:

   # do something

   addi  $t0,  $t0,  1          # i ++
   slti   $t1,  $t0,  4          # $t1 = 1 if i < 4
   bne   $t1,  $zero,  Loop   # go to Loop if i < 4
```

MIPS Decision-making Instructions: **switch/case**

- A straightforward way to implement switch is using a chain of if-then-else statements
  - Would you try to write a switch statement?
- An alternative is to take advantage of jump (address) table
  - We will discuss this later