INTRODUCTION TO COMPUTER ORGANIZATION

CCIT4026

Chapter 4: Instruction Set Design (Part 3)

Outline

- Decision Making
- ‘Less Than’ Test
- Unconditional Jump
- Using Byte data in MIPS
- 2D array in MIPS

Decision Making

What distinguishes a computer from a simple calculator is its ability to **make decisions** based on input data or values obtained during the computation.

- In high-level programming languages, decision-making instruction:
  - `if` statement, then continue execute the following

- But in MIPS, decision-making instructions (or **conditional branches**):
  - **`beq`** ('branch if equal'):
    - e.g. **`beq reg1, reg2, L1`**
    - go to statement labeled **`L1`** if **`reg1`** and **`reg2`** have the same value

  - **`bne`** ('branch if not equal'):
    - e.g. **`bne reg1, reg2, L1`**
    - go to statement labeled **`L1`** if **`reg1`** and **`reg2`** do not have the same value

❖ Once branch is satisfied $\Longrightarrow$ *NOT continue, jump to the label*

Example #1

- In the following Python code segment, **i**, and **j** are variables:

```
if i == j:
        i = i+1
j = j-1
```

Assuming **$s1** stores **i** and **$s2** stores **j**, what is the compiled MIPS code?

```
    bne $s1,$s2, L1 # If $s1 != $s2, branch to L1 address
                    # (go to L1 if i != j)
    addi $s1,$s1,1  # $s1 = $s1+1     (i = i + 1)
L1: addi $s2,$s2,-1 # $s2 = $s2-1     (j = j - 1)
```

Unconditional Jump

- `j` ('jump'):
  - e.g.

```
        j   myLabel
        #...
myLabel:
        #...
```

  - Jump to the target address location specified by the label address in the instruction

Example #2

- In the following Python code segment, **f**, **g**, **h**, **i**, and **j** are variables:

```
if i == j:
        f = g + h     # if i == j, f = g + h
else:
        f = g - h     # if i != j, f = g - h
```

Assuming that the five variables **f** through **j** correspond to five registers **$s0** through **$s4**, what is the compiled MIPS code?

```
        # Assume $s0 = f, $s1 = g, $s2 = h
        #        $s3 = i, #s4 = j

        bne $s3,$s4, Else # go to Else if i != j
        add $s0,$s1,$s2   # f = g + h (if i == j)
        j Exit            # go to Exit
Else:   sub $s0,$s1,$s2   # f = g - h (if i != j)
Exit:   ……..
```

Example #3

- In the following Python code segment, **i**, **j** and **k** are variables:

```
if i == j and i == k:
        i = i + 1
else:
        j = j - 1
j = i + k
```

Assuming that the three variables **i**, **j** and **k** correspond to registers **$s1**, **$s2** and **$s3**, what is the compiled MIPS code?

```
        # Assume $s1 = i, $s2 = j, $s3 = k
        bne $s1,$s2, Else       # go to Else if i != j
        bne $s1,$s3, Else       # go to Else if i != k
        addi $s1,$s1,1          # if i==j & i==k: i=i+1
        j Next                  # go to Next
Else:   addi $s2,$s2,-1         # else j=j-1
Next:   add $s2,$s1,$s3         # j = i + k
```

Example #4

- In the following Python code segment, **i**, **j** and **k** are variables:

```
if i == j or i == k:
        i = i + 1
else:
        j = j - 1
j = i + k
```

Assuming that the three variables **i**, **j** and **k** correspond to registers **$s1**, **$s2** and **$s3**, what is the compiled MIPS code?

```
        # Assume $s1 = i, $s2 = j, $s3 = k
        beq $s1,$s2, IF          # go to IF if i == j
        bne $s1,$s3, Else        # go to Else if i != k
IF:     addi $s1,$s1,1           # if i==j or i==k: i=i+1
        j Next                   # go to Next
Else:   add $s2,$s2,-1           # else j=j-1
Next:   add $s2,$s1,$s3          # j = i + k
```

Loop

- Decisions are important both for
  - choosing between two alternatives–found in **if** statement
  - iterating a computation–found in **loops**
- In **loops**, decisions are needed to determine when to stop looping
- Commonly used loop constructs in high-level programming languages
  - **while**
  - **for**

While loop

- **while loop** in Python:

```
while <cond>:
      <while-body>
```

similar to:

```
L1: if <cond>:
      <if true-body>
      goto L1                    # unconditional jump
```

Example #5

```
while i < j:
        k = k + 1
        i = i * 2
```

Assuming that the three variables **i**, **j** and **k** correspond to registers **$s1**, **$s2** and **$s3**, what is the compiled MIPS code?

```
# Assume $s1 = i, $s2 = j, $s3 = k
L1:  bge  $s1,$s2, DONE      # branch if not (i < j)
     addi  $s3,$s3,1         # k = k + 1
     add  $s1,$s1,$s1        # i = i * 2
     j L1                    # loop back
DONE:
```

Example #6

- We want to write a program in MIPS assembly. The program can access an arbitrary element of an integer array. Here is the Python version and the MIPS version with an Boundary Checking function

```
// Python version

array = [10,11,12,13,14,15,16,17,18,19]

while 1:
    print ("i? ")
    i = int(input())
    if i >= 0 and i <= 9:
        print (array[i])
        break
```

Example #6

```
1  #------- Data Segment ----------
2  .data
3  # declare the string mesg
4  msg1: .asciiz "i? "
5  msg2: .asciiz "array[i] = "
6  newline: .asciiz "\n"
7  array: .word 10 11 12 13 14 15 16 17 18 19
8  size: .word 10
9
10 #------- Text Segment ----------
11 .text
12 .globl __start
13 __start:
14
15 # Print msg1: i?
16 Label:
17             la $a0, msg1
18             li $v0, 4
19             syscall
```

Example #6 (con't)

Format: `SLT rd, rs, rt`

Operation:

```
if GPR[rs] < GPR[rt] then
    GPR[rd] ← 0^(GPRLEN-1) || 1
else
    GPR[rd] ← 0^GPRLEN
endif
```

```
20
21  # Get the value i (store in $v0)
22              li $v0, 5
23              syscall
24              # Keep i in $t0: $t0 = 0 + $v0
25              add $t0, $zero, $v0
26
27  # Boundary Checking on i
28              lw $t4, size              # size is now in $t4
29              slt $t5, $t0, $t4         # set $t5 if $t0 (user input) is less than $t4 (10)
30              bne $t5, $zero, check_l # if result is not 0, means it is < 10, check lower boundry
31              j Label                   # jump to Label to get input again
32  check_l:
33              slt $t5, $t0, $zero       # set $t5 if $t0 (user input) is less than $zero (0)
34              beq $t5, $zero, output  # if result is 0, means it is not less than 0, skip to output
35              j Label                   # jump to Label to get input again
36
```

Example #6 (con't)

```
37  # Print msg2
38  output:
39              la $a0, msg2
40              li $v0, 4
41              syscall
42
43  # Find the address for array[i] and
44  # keep it in $t1 (i.e., $t1 = $t1 + $t0 x 4)
45              la $t1, array
46              add $t0, $t0, $t0
47              add $t0, $t0, $t0
48              add $t1, $t1, $t0
49
50  # Print result
51              lw $a0, 0($t1)
52              li $v0, 1
53              syscall
54  # Print a new line
55              la $a0, newline
56              li $v0, 4
57              syscall
58  # Terminate the program
59              li $v0, 10
60              syscall
```

for loop

- **for loop** in C-Language:

```
for( <init> ; <end> ; <increment>) {
        <for-body>
}
```

Equivalent to while loop:

```
<init>;
while (not <end>) {
    <while-body>
    <increment>
}
```

For Python

```
for i in range(0, 9, 2):
    print(i)
```

Example #7

```
for i in range(0, 10, 1):
    <for-body>
```

```
initialize:   add $s1, $zero, $zero   # i = $s1 = 0
              addi $s2, $zero, 10     # $s2 = 10
predicate:    slt $t0, $s1, $s2       # set $t0 = 1 if
                                      # i (=$s1) < 10 (=$s2)
              beq $t0, $zero, endfor  # if $t0=0, exit loop
consequent:   ……
               <for-body>
              …..
              addi $s1, $s1, 1        # i = i + 1
              j predicate             # Loop back
endfor:
```

'Less Than' Test

- Besides testing for equality or inequality, it is often useful to see if a variable is less than another variable.
  - e.g., want to exit from a loop when the array index is less than 0

- **`slt`** ('set on less than'):
  - **`slt reg1,reg2,reg3`**
  - register **`reg1`** is set to 1 if the value in **`reg2`** is less than the value in **`reg3`**; otherwise, register **`reg1`** is set to 0

- **`slti`** ('set on less than immediate')
  - **`slti reg1,reg2,imm`**
  - E.g. **`slti $t0, $s2, 10`** **`# if $s2<10,$t0=1,`** otherwise
    `# $t0=0`

‘Less Than’ Test

- MIPS compilers use **`beq`**, **`bne`**, **`slt`**, **`slti`** and the fixed value of 0 (always available by reading register **`$zero`**) to create all comparison operations:

  - **equal**
  - **not equal**
  - **less than**
  - **less than or equal**
  - **greater than**
  - **greater than or equal**

‘Less Than’ Test

- **Equal:** **beq**
- **not equal:** **bne**
- **less than:** **blt reg2, reg3, Label**

```
→ slt reg1, reg2, reg3      # if reg2 < reg3, set reg1 = 1 else 0
  bne reg1, $zero, Label    # jump to Label if reg1 != 0 (reg2 < reg3)
```

- **less than or equal:** **ble reg2, reg3, Label**

```
→ slt reg1, reg3, reg2      # if reg3 < reg2, set reg1 = 1 else 0 (reg3 >= reg2)
  beq reg1, $zero, Label    # jump to Label if reg1==0 (reg2 <= reg3)
```

- **greater than:** **bgt reg2, reg3, Label**

```
→slt reg1, reg3, reg2       # if reg3 < reg2, set reg1 = 1 else 0 (reg3 >= reg2)
  bne reg1, $zero, Label    # jump to Label if reg1!=0 (reg2 > reg3)
```

- **greater than or equal:** **bge reg2, reg3, Label**

```
→slt reg1, reg2, reg3       # if reg2 < reg3, set reg1 = 1 else 0 (reg2 >= reg3)
  beq reg1, $zero, Label    # jump to Label if reg1==0 (reg2 >= reg3)
```

Example #8

- What is the MIPS code that tests if variable **a** (corresponding to register **$s1**) is less than variable **b** (register **$s2**) and then branch to label **L** if the condition holds?

```
blt $s1,$s2, L              # branch if less than
translates to:
                            # $at0=1 if a < b else $at=0
                            # branch to L
```

- `slt $at,$s1,$s2    # $at=1 if a < b else $at=0`
- `bne $at,$zero, L   # branch to L`

- If results in a "1", it means it is less than, NOT EQUAL 0 is 1

- **Remarks**:
  - 'branch if less than' instruction is a pseudo instruction, and MIPS architecture does this operation using two faster MIPS instructions – similar for other conditional branches.

Example #9

Consider the program below: What are the values stored in Array1 after the program is executed?

```
          .data
Array1:  .word 4 8 12 16 20

         .text
         .globl __start
__start:

la $t0, Array1
lw $t1, 4($t0)
lw $t2, 8($t0)
la $s0, Label1
add $s0, $s0, $t1
jr $s0

Label1:
add $t1, $t1, $t1
add $t1, $t2, $t2
sw $t1, 12($t0)
```

i) What are the values of t1 & t2 ?

ii) If the instruction add $t1, $t1, $t1 stores at address 10000, what is the value of s0 & what jr $s0 does ?

Store at address 10000

iii) Where is this instruction stored ?

Example #9 (with remarks)

```
1   .data
2   Array1: .word 4 8 12 16 20
3   .text
4   .globl __start
5   __start:
6               la $t0, Array1          # $t0 points to beginning of Array1
7               lw $t1, 4($t0)          # $t1 = 8
8               lw $t2, 8($t0)          # $t2 = 12 (0xC)
9               la $s0, Label1          # $s0 points to Label1
10              add $s0, $s0, $t1       # $s0 points to Label1+8
11              jr $s0                  # jump to $s0, i.e., Label1+8
12
13  Label1:
14              add $t1, $t1, $t1
15              add $t1, $t2, $t2
16              sw $t1, 12($t0)         # this line is Label1+8, store $t1(8) into ($t0+12)
```

Put the code into MARS and single-step through it

Branch (summary)

Instructions	Explanation
`slt Rdest, Rsrc1, Src2` (Set on Less Than)	Set register *Rdest* to *1* if *Rsrc1* is less than R*Src2*, and to *0* otherwise.
`slti Rdest, Rsrc1, imm` (Set on Less Than immediate)	Set register *Rdest* to *1* if *Rsrc1* is less than *imm*, and to *0* otherwise.
`beq Rsrc1, RSrc2, label` (Branch on Equal)	Conditionally branch to the instruction at the label, if the contents of register *Rsrc1* equals R*Src2*.
`bne Rsrc1, RSrc2, label` (Branch on Not Equal)	Conditionally branch to the instruction at the label, if the contents of register *Rsrc1* doesn't equals R*Src2*.
`j label` (Unconditionally jump)	Jump to the instruction at the label unconditionally.

2D array in MIPS

- Two dimensional array is commonly used in programming. In fact, a 2D array with fixed dimension is implemented with 1D array.

- Suppose we have a array

a = [[11, 12, 5],[15, 6,10],[22, 8, 12],[7,15,8]]

Each integer is 4 bytes long. You realized that the address different between `a[i][j]` & `a[i][j+1]` is 4 bytes.

The difference between `a[i][j]` & `a[i+1][j]` is data_size*width = 4 byte * 3 = 12 bytes.

a = a[0]

a = a[1]

a = a[2]

a = a[3]

Address	Label	Content
a[0]	a[0][0]	11
a[0]+4	a[0][1]	12
a[0]+8	a[0][2]	5
a[0]+12	a[1][0]	15
a[0]+16	a[1][1]	6
a[0]+20	a[1][2]	10
a[0]+24	a[2][0]	22
a[0]+28	a[2][1]	8
a[0]+32	a[2][2]	12
a[0]+36	a[3][0]	7
a[0]+40	a[3][1]	15
a[0]+44	a[3][2]	8

2D array in MIPS

- In MIPS, you may declare the 2D array like the following format. Then, the "2D array" will be stored in memory like the table described above.

```
a:
a_0: .word 0 0 0
a_1: .word 0 0 0
a_2: .word 0 0 0
a_3: .word 0 0 0
```

Offset of a[row][col] is :
(row * (# of col) + col) * size of each element

- To access array `a[$t0][$t1]`, what you need is to find its address = address of `a + ($t0*3 + $t1)*4`. So, the method to access the 2D array is the similar to a simple array. To declare a 2D array in byte type, you can do the following:

```
B:
b_0: .byte 48 49 0
b_1: .byte 50 51 0
b_2: .byte 52 53 0
C:
c_0: .asciiz "01"      # array b & c are equivalent
c_1: .asciiz "23"
c_2: .asciiz "45"
```