EECS 645

Hao Luo

Sep 18th

Homework 6

2.24 [5] Suppose the program counter (PC) is set to 0x2000 0000. Is it possible to use the jump (j) MIPS assembly instruction to set the PC to the address as 0x4000 0000? Is it possible to use the branch-on-equal (beq) MIPS assembly instruction to set the PC to this same address?

Solution: In jump operation, as the j-instruction format (6 for op code), there are only 26 bits to represent the address representation. On the running time, the low order of two bits “00” of 32-bits instruction will shift the 26-bits left result a 28-bit address. However, the 28-bits address is impossible to achieve a jump from 0x2000 0000 to 0x4000 0000, which actually need 30 bits.

For the branch-on-equal instruction, it uses the i-format instruction, which has 16 bits for address representation. On its running time, it will be upper 18 bits. It is impossible to reach 0x4000 0000 from 0x2000 0000.

2.25 The following instruction is not included in the MIPS instruction set: rpt $t2, loop # if(R[rs]>0) R[rs]=R[rs]−1, PC=PC+4+BranchAddr

2.25.1 [5] If this instruction were to be implemented in the MIPS instruction set, what is the most appropriate instruction format?

Solution: I-format is the most appropriate one.

2.25.2 [5] What is the shortest sequence of MIPS instructions that performs the same operation?

Solution:

```
Loop:  slt $t0, $0, $t2        #if ($0<$t2),$t0=1
       Beq $t0, $0, exit       #if ($t0=0), exit
       Subi $t2, $t2, 1        # R[rs]=R[rs]−1
       J loop                  # PC=PC+4+BranchAddr
exit:
```

2.26 Consider the following MIPS loop:

```
LOOP:  slt $t2, $0, $t1
       beq $t2, $0, DONE
       subi $t1, $t1, 1
       addi $s2, $s2, 2
       j LOOP
DONE:
```

2.26.1 [5] Assume that the register $t1 is initialized to the value 10. What is the value in register $s2 assuming $s2 is initially zero?

Solution: if $t1 = 10, then

Slt $t2, $0, $t1 => $t2 = 1

Beq $t2, $0, DONE => $t2! = 0

Subi $t1, $t1, 1 => $t1 = 9

Addi $s2, $s2, 2 => $s2 = 2

This loop will run 10 times until $t1 = 0, so $s2 will be 0 + 10 * 2 = 20

2.26.2 [5] For each of the loops above, write the equivalent C code routine. Assume that the registers $s1, $s2, $t1, and $t2 are integers A, B, i, and temp, respectively

Solution:

```
int i =10;
```

```
while (i > 0)
{
        i--;
        B +=2;
}
```

2.26.3 [5] For the loops written in MIPS assembly above, assume that the register $t1 is initialized to the value N. How many MIPS instructions are executed?

Solution: For each loop ($t2! =0), there are 5 MIPS instructions, therefore, before the loop end, 5N instructions are executed. For the last time, after determine $t1 = 0, we get $t2 = 0, then the program end in the beq instruction for $t2 because $t2 = 0.

Hence, there are 5N+2 instruction executed for $t1 = N.

2.27 [5] Translate the following C code to MIPS assembly code. Use a minimum number of instructions. Assume that the values of a, b, i, and j are in registers $s0, $s1, $t0, and $t1, respectively. Also, assume that register $s2 holds the base address of the array D.

```
for(i=0; i<a; i++)
        for(j=0; j<b;j++)
                D[4*j] = i + j;
```

```
Solution:   add $t0, $0, $0                 #initialize i
            Loop1:  Slt $t2, $t0, $s0       #if (i<a), $t2 = 1
                    Beq $t2, $0, exit       #if ($t2 = 0), exit
                    Addi $t0, $t0, 1        #i=i+1
                    Add $t1, $0, $0         #initialize j
                    J Loop1                 #back to loop1
            Loop2:  slt $t3, $t1, $s1       #if (j<b), $t3 = 1
```

```
                Beq $t3, $t0, Loop1 #if ($t3 = 0), back to loop1
                Add $t4, $t0, $t1   #get value of i+j
                Sll $t5, $t1, 4     #get j*4 for offset
                Add $t6, $t5, $s2   #offset address D[4*j]
                Sw $t4, 0($t6)      #D [4*j] = i+j
                Addi $t1, $t1, 1    #j = j+1
                J Loop2             #back to loop2
        Exit:
```

2.31 [5] Implement the following C code in MIPS assembly. What is the total number of MIPS instructions needed to execute the function?

```
int fib(int n){
    if (n==0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fib(n−1) + fib(n−2);}
```

Solution:

```
fib:
        Addi $sp, $sp, -12      #allocate stack of 12 bytes
        Sw $a0, 8($sp)          #save n argument
        Sw $ra, 4($sp)          #save return address
        Sw $s0, 0($sp)          #save $s0
        Bne $a0, $0, else1      #if ($a0! =0), go else1
        Add $v0, $a0, $0        #if ($a0 = 0), return 0
        J exit;                 #jump exit
ELSE1:
```

```
        slt $s0, $a0, 2          #if($a0<2), $s0=1
        Beq $s0, $0, ELSE2       #if ($s0=0), go else2
        Addi $v0, $a0, 1         #if ($s0=1), return 1
        J exit;                  #jump exit
ELSE2:
        Addi $a0, $a0, -1        #get value of n-1
        Jal fib                  #call function
        Add $s0, $v0, 0          #receive returned value
        Addi $a0, $a0, -1        #get value of n-2
        Jal fib                  #call function
        Add $v0, $s0, $s1        #add all the returned value
                                 together
EXIT:
        Lw $a0, 8($sp)           #load saved n argument
        Lw $ra, 4($sp)           #load saved return address
        Lw $s0, 0($sp)           #load saved $s0
        Addi $sp, $sp, 12        #free stack
        Jr $ra                   #return to caller
```

2.34 [5] Translate function f into MIPS assembly language. If you need to use registers $t0 through $t7, use the lower-numbered registers first. Assume the function declaration for func is “int f(int a, int b);”. The code for function f is as follows:

```
int f(int a, int b, int c, int d){
    return func(func(a,b),c+d);
}
```

```
Solution:   f:   addi $sp, $sp, -4    #allocate stack of 4 bytes
                 Sw $ra, 0($sp)       #save return address
```

```
Jal func              #call function
Add $a0, $0, $v0      #get return value of func(a,b)
Add $a1, $a2, $a3     #get value of c+d
Jal func              #call function
Lw $ra, 0($sp)        #restore return address
Addi $sp, $sp, 4      #free stack
Jr $ra                #return to caller
```

2.46 Assume for a given processor the CPI of arithmetic instructions is 1, the CPI of load/store instructions is 10, and the CPI of branch instructions is 3. Assume a program has the following instruction breakdowns: 500 million arithmetic instructions, 300 million load/store instructions, 100 million branch instructions.

2.46.1 [5] Suppose that new, more powerful arithmetic instructions are added to the instruction set. On average, through the use of these more powerful arithmetic instructions, we can reduce the number of arithmetic instructions needed to execute a program by 25%, and the cost of increasing the clock cycle time by only 10%. Is this a good design choice? Why?

Solution: CPU time = numbers of instructions * CPI * clock cycle time

The new arithmetic instructions = 500*0.75 = 375, and the new clock is 1.1 times than the original clock time.

Then, new CPU time = 0.75 *1*1.1 * 500 + 10*1.1*300 + 3*1.1*100

= 412.5 + 3300 + 330 = 4042.5 (ignoring the unit of instructions)

For the original CPU time, it = 500 + 3000 + 300 = 3800

4042.5>3800, it means the new design increase the CPU time, and make the execution time slower, therefore, it is not a good design choice.

2.46.2 [5] Suppose that we find a way to double the performance of arithmetic instructions. What is the overall speedup of our machine? What

if we find a way to improve the performance of arithmetic instructions by 10 times?

Solution: the total instructions = 500 + 300 + 100 = 900

Weighted average CPI = (1 * 500 + 10 * 300 + 3 * 100)/900 = 4.22

When performance of arithmetic instructions doubled, the CPI of arithmetic will be divided by 2, it = 1 / 2 =0.5

Then, new weighted average CPI = (0.5 * 500 + 10 * 300 + 3 * 100)/900 = 3.94

The speedup = old CPI/new CPI = 4.22/3.94 = 1.07

When performance of arithmetic instructions increased by 10 times, the CPI of arithmetic will be divided by 10, it = 1 / 10 =0.1

Then, new weighted average CPI = (0.1 * 500 + 10 * 300 + 3 * 100)/900 = 3.72

The speedup = old CPI/new CPI = 4.22/3.72 = 1.13