# C335 Homework #3

| | |
|---|---|
| Points: | : 40 points |
| Due Date: | : Feb. 17th (before the start of the class) |
| Submissions: | : For F-2-F students, hardcopy (type or write your solution clearly)<br>For online students, e-copy to Canvas |

## PART I (10 POINTS)

In the class lecture, we have compiled the following C code:

```
          while (save[i] == k)
               i += 1;
```

into the following assembly language:

```
     Loop:     sll  $t1, $s3, 2
               add  $t1, $t1, $s6
               lw   $t0, 0($t1)
               bne  $t0, $s5, Exit
               addi $s3, $s3, 1
               j    Loop
     Exit:     . . .
```

Assume the memory address for the first instruction (Loop: …) is 0x00400020.

Now, you are expected to assemble the program (except the last line of the assembly code) into machine code. Your solution should contain three columns: the first column being the memory address of the instruction, the second column being the binary representation of the machine code, and the third column being the hexadecimal representation of the machine code.

| Memory Address | Machine Code (Bin) | Machine Code (Hex) |
|---|---|---|
| 0x00400020 | 000000, 00000, 10011, 01001, 00010, 000000 | 0x00134880 |
| | | |
| | | |
| | | |
| | | |
| | | |

## PART II (9 POINTS)

For the following C code segment, write a code segment in MIPS assembly language to do the same thing. Assume **i** is in $s0, **x** is in $s1, and **y** is in $s2. Don't forget to comment you code.

```
          for (i=0; i<x; i=i+1)
               y = y + i;
```

## PART III (11 POINTS)

We have given some examples of pseudoinstructions in the lecture notes. For each pseudoinstruction in the following table, produce a minimal sequence of actual MIPS instructions to accomplish the same thing. You may need to use **$at** for some of the sequences. In the following table, ***big*** refers to a specific number that requires 32 bits to represent and ***small*** to a number that can fit in 16 bits.
(**Hint:** you can use upper(big) / lower(big) to represent the upper/lower half of the immediate big respectively.)

| Pseudoinstruction | What it accomplishes | Equivalent MIPS instructions |
|---|---|---|
| move $t1, $t2 | $t1 = $t2 | |
| clear $t0 | $t0 = 0 | |
| li $t1, small | $t1 = small | |
| li $t2, big | $t2 = big | |
| addi $t0, $t2, big | $t0 = $t2 + big | |
| beq $t2, big, L | If($t2 = big) go to L | |
| beq $t1, small, L | If ($t1 = small) go to L | |
| ble $t3, $t5, L | If ($t3 <= $t5) go to L | |
| bgt $t4, $t5, L | If ($t4 > $t5) go to L | |
| bge $t5, $t3, L | If ($t5 >= $t3) go to L | |
| lw $t5, big(t2) | $t5 = Memory[$t2 + big] | |

## PART IV (10 POINTS)

In the class lecture, we have compiled the following C code:

```
int fact (int n) {
        if (n < 1) return 1;
                else return (n * fact (n-1)); }
```

into the following assembly code:

```
fact:     addi $sp, $sp, -8         #adjust stack pointer
          sw   $ra, 4($sp)          #save return address
          sw   $a0, 0($sp)          #save argument n
          slti $t0, $a0, 1          #test for n < 1
```

```
            beq  $t0, $zero, L1       #if n >=1, go to L1
            addi $v0, $zero, 1        #else return 1 in $v0
            addi $sp, $sp, 8          #adjust stack pointer
            jr   $ra                  #return to caller
L1:         addi $a0, $a0, -1         #n >=1, so decrement n
            jal  fact                 #call fact with (n-1)
                                 #this is where fact returns
bk_f:       lw   $a0, 0($sp)          #restore argument n
            lw   $ra, 4($sp)          #restore return address
            addi $sp, $sp, 8          #adjust stack pointer
            mul  $v0, $a0, $v0        #$v0 = n * fact(n-1)
            jr   $ra                  #return to caller
```

Assume the memory address for the first instruction (fact: …) is 0x00400020.

Now, you are expected to assemble the program (***SKIP the instruction: mul $v0, $a0, $v0***) into machine code. Your solution should contain three columns: the first column being the memory address of the instruction, the second column being the binary representation of the machine code, and the third column being the hexadecimal representation of the machine code.