July 22, 2007

COMPUTER ENGINEERING DEPARTMENT

ICS 233

COMPUTER ARCHITECTURE & ASSEMBLY LANGUAGE

Major Exam I

Summer Semester (063)

Time: 7:00-9:30 PM

Student Name : KEY____

Student ID. : ____

Question	Max Points	Score
Q1	40	
Q2	15	
Q3	18	
Q4	12	
Q5	15	
Total	100	

Dr. Aiman El-Maleh

[40 Points]

(Q1) Indicate whether the following is **true** or **false**, and if it is false **<u>correct</u>** it (correct the answer and not the question):

(1) (True, **<u>False</u>**) The smallest (negative) number that can be represented using 8-bit 2`s complement in hexadecimal is FF and the largest positive number in hexadecimal is 7F.

The smallest (negative) number that can be represented using 8-bit 2`s complement in hexadecimal is **80** and the largest positive number in hexadecimal is **7F**.

(2) (True, **<u>False</u>**) Assume that the CPU has just read a 32-bit instruction from the address 0x00400000. Then, the address of the next instruction that this CPU is going to read is 0x00400001.

The address of the next instruction that this CPU is going to read is 0x00400000+4=0x00400004.

(3) (**<u>True</u>**, False) Assuming 8-bit representation of numbers, the binary number 10100100 is equal to -36 in sign-magnitude representation, -91 in 1`s complement representation, and -92 in 2`s complement representation.

(4) (True, **<u>False</u>**) The following assembler directive allocates 1 word initialized by 10.

```
X: .word 1:10
```

It allocates 10 words all initialized by 1.

(5) (True, **<u>False</u>**) With a 32-bit address bus and 32-bit data bus, the maximum memory size than can be accessed by a processor is 4MByte and the maximum number of bytes that can be read or written in a single cycle is 8 Bytes.

The maximum memory size than can be accessed by a processor is 2^{32}=4GByte and the maximum number of bytes that can be read or written in a single cycle is 32/8=4 Bytes

(6) (**<u>True,</u>** False) Assuming variable Array is defined as shown below:

Array: .word 0x00000010, 0x00000020

The content of register $t0 after executing the following sequence of instructions is 0x00000020.

```
la $t0, Array
lw $t0, 4($t0)
```

(7) (True, **<u>False</u>**) The instruction set architecture of a processor consists of its control unit, data path, memory, and the instruction set.

The instruction set architecture of a processor consists of its instruction set, **programmer-accessible registers** and memory.

(8) (True, **<u>False</u>**) Given a magnetic disk with the following properties:

- Rotation speed = 7200 RPM (rotations per minute)
- Average seek = 8 ms, Sector = 512 bytes, Track = 200 sectors

The average time to access a block of 64 consecutive sectors is 13.5 ms.

Average access time= Seek Time + Rotation Latency + Transfer Time
Rotations per second=7200/60 =120 RPS
Rotation time in milliseconds=1000/120=8.33 ms
Time to transfer 64 sectors=(64/200)* 8.33=2.67 ms
Average access time=8 + 4.17 + 2.67 =**14.84 ms**.

(9) (True, **False**) Assuming the following data segment, and assuming that the first variable X is given the address **0x10010000**, then the address for variable Y will be **0x10010005.**

```
.data
X:      .byte  10, 11, 12, 13, 14
Y:      .word  15
```

The address for variable Y will be **0x10010008** as it has to be word aligned i.e. multiple of 4.

(10) (True, **False**) The code given below prints the statement: *Exam1*

```
MSG: .ascii  "Exam1"
        .asciiz " ICS 233"

          li $v0, 4
          la $a0, MSG
          syscall
```

It will print ExamI ICS233

(11) (True, **False**) Assume that the instruction j NEXT is at address 0x00400020 in the text segment, and the label NEXT is at address 0x00400010. Then, the address stored in the assembled instruction for the label NEXT is 0x0400010.

The address stored in the assembled instruction for the label NEXT is 0x0400010/4=0x0100004.

(12) (**True**, False) Assume that the instruction beq $t0, $t1, NEXT is at address 0x00400020 in the text segment, and the label NEXT is at address 0x00400010. Then, the address stored in the assembled instruction for the label NEXT is 0xfffb.

The address stored in the assembled instruction for the label NEXT is (Next-(PC+4))/4=(0x00400010-0x00400024)/4=0xffffffec/4=0xfffb

(13) (True, **False**) After executing the instruction sll $t0, $t0, 2, the content of register $t0 is equal to 2*$t0, for both signed and unsigned content.

The content of register $t0 is equal to 4*$t0, for both signed and unsigned content.

(14) (True, **False**) The code given below implements the conditional statement

if (($t0 < 1) AND ($t1 > 100)) Then $t2=0.

```
        slti $t3, $t0, 1
        bne $t3, $zero,  Zero_index
        li $t3, 100
        slt $t3, $t3, $t1
        beq $t3, $zero, End_if
Zero_index:
        xor $t2, $t2, $t2
End_if:
```

The code given below implements the conditional statement

if (($t0 < 1) OR ($t1 > 100)) Then $t2=0.

(15) (**True**, False) Assuming that $a0 contains an Alphabetic character, the instruction *andi $a0, $a0, 0xdf* will guarantee that the character in $a0 is an upper case character. Note that the ASCII code of character ‘A’ is 0x41 while that of character ‘a’ is 0x61.

(16) (True, **False**) Assume you are in a company that will market a certain IC chip. The cost per wafer is $4000, and each wafer can be diced into 2000 dies. The die yield is 60%. Then the cost per good die is $2.

The cost per good die = $4000/(2000*.6)=$4000/1200=$3.33

(17) (**<u>True</u>**, False) Assume that $t0=0xffff8111 and $t1=0xfffff265. Executing the instruction *subu $t0, $t0, $t1* produces correct result in $t0 assuming both signed and unsigned number representation.

(18) (True, **<u>False</u>**) The difference between *add* and *addu* instructions is that *add* should be used for addition of signed numbers while *addu* should be used for addition of unsigned numbers.

The difference is that when *add* instruction is used and the addition operation produces an overflow, this will cause arithmetic exception while in *addu* it will be ignored.

(19) (**<u>True</u>**, False) Executing the following sequence of instructions produces the value 0x0000009b in $s2.

```
li      $s1, 5
sll     $s2, $s1, 5
subu    $s2, $s2, $s1
```

(20) (True, **<u>False</u>**) Assuming that $s1=0xabcd1234 and $s2=0xffff0000, executing the instruction *nor $s0,$s1,$s2* produces the value 0xffff1234 in $s0.

It produces the value 0x0000edcb in $s0.

[15 Points]

(Q2) Using only basic MIPS instructions, write the shortest sequence of instructions to implement each of the following pseudo instructions:

1. *li $t0, 0x12345678* #$t0 is loaded with the immediate value 0x12345678

```
lui $at, 0x1234
ori $t0, $at, 0x5678
```

2. *not $t0, $t1* #$t0 is loaded with the 1's complement value of $t1

```
nor $t0, $t1, $zero
```

3. *bge $t0, $t1, Next* # branch to Next if $t0 is greater than or equal $t1

```
slt $at, $t0, $t1
beq $at, $zero, Next
```

4. *abs $t0, $t1* #$t0 is loaded with the absolute value of $t1

```
sra $at, $t1, 31
xor $t0, $at, $t1
sub $t0, $t0, $at
```

5. *rol $t0, $t0, 5* #$t0 is rotated to the left by 5 bits and stored in $t0

```
srl $at, $t0, 27
sll $t0, $t0, 5
or $t0, $t0, $at
```

[18 Points]

(Q3) Answer the following questions. Show how you obtained your answer:

(i) Given that **TABLE** is defined as: **TABLE: .ascii "Ahmad Ali Anas"**

Determine the content of register **$t0** after executing the following code:

```
        xor $t0, $t0, $t0
        li $t1, 14
        la $t2, TABLE
        addi $t2, $t2, -1
Next:   beq $t1, $zero, ENL
        addi $t2, $t2, 1
        lbu $t3, ($t2)
        ori $t3, $t3, 0x20
        li $t4, 'a'
        addi $t1, $t1, -1
        bne $t3, $t4, Next
        addi $t0, $t0, 1
        j Next
ENL:
```

The content of register $t0 after executing the program will be 5, which the number of characters 'A' or 'a' found in TABLE.

(ii) Given that **TABLE** is defined as shown below**:**

TABLE: .space 33

Determine the output produced after executing the following code:

```
            li $t0, 0xabcde765
            li $t1, 32
            la $t2, TABLE
AGAIN:      li $t3, '0'
            rol $t0, $t0, 1
            andi $t4, $t0, 1
            add  $t3, $t3, $t4
            sb $t3, ($t2)
            addi $t2, $t2, 1
            addi $t1, $t1, -1
            bne $t1, $zero, AGAIN
            la $a0, TABLE
            li $v0, 4
            syscall
```

The output produce will be the binary content of register $t0 which is
10101011110011011110011101100101

(iii) Given that TABLE is defined as shown below, determine the content of TABLE after executing the following code:

TABLE: .word 1, 2, 3, 4, 5, 6, 7, 8

```
        la $t0,  TABLE
        addi $t1, $t0, 28
        li $s0, 4
Again:
        lw $t2, ($t0)
        lw $t3, ($t1)
        sw $t2, ($t1)
        sw $t3, ($t0)
        addi $t0, $t0, 4
        addi $t1, $t1, -4
        addi $s0, $s0, -1
        bne $s0, $zero, Again
```

The content of TABLE after executing the program will be:

TABLE: .word 8, 7, 6, 5, 4, 3, 2, 1

[12 Points]

(Q4) Write a MIPS assembly program to do the following using the smallest possible number of instructions. Ask the user to enter two integers and then display their sum according to the format given below.

```
Sample Execution:
Enter an integer: -2
Enter another integer: 20
The sum of -2 and 20 = 18
```

```
.DATA
msg: .asciiz "Enter an integer:"
msg2: .asciiz "Enter another integer:"
msg3: .asciiz "The sum of "
msg4: .asciiz " and "
msg5: .asciiz " = "

.TEXT
.GLOBL      main

main:

# Reading the first integer
        la $a0, msg
        li $v0, 4
        syscall

        li $v0, 5
        syscall
        move $t0, $v0

# Reading the second integer
        la $a0, msg2
        li $v0, 4
        syscall

        li $v0, 5
        syscall
        move $t1, $v0

# Displaying the sum
        addu $t2, $t0, $t1

        la $a0, msg3
        li $v0, 4
        syscall
```

```
li $v0, 1
move $a0, $t0
syscall

la $a0, msg4
li $v0, 4
syscall

li $v0, 1
move $a0, $t1
syscall

la $a0, msg5
li $v0, 4
syscall

li $v0, 1
move $a0, $t2
syscall

li $v0, 10        # Exit program
syscall
```

[15 Points]

(Q5) Write a MIPS assembly program to sort an array of integers (i.e. 32-bit signed numbers) in an **ascending** order using **BubbleSort** algorithm. Minimize the number of instructions used.

The pseudocode for the **BublleSort** algorithm is given below:

```
BubbleSort (ArrayPointer, ArraySize)
        Status = Unsorted
        #comprisons = ArraySize-1
        while (#comparisons<>0 AND status = Unsorted)
                Status = Sorted
                for (i= 0 to #comparisons)
                        if (Array[i] > Array[i+1])
                                swap ith and (i+1)th elements of the array
                                Status = Unsorted
                        end if
                end for
        #comparisons = #comparisons – 1
        end while
end BubbleSort
```

Clearly indicate the registers used for each variable. Store the array to be sorted in variable Array as defined below.

Array: .word 10, 2, 0, 15, 25, 30, 7, 22

```
.DATA
Array: .word 10, 2, 0, 15, 25, 30, 7, 22
.TEXT
.GLOBL      main

main:

        la $a0, Array           # Array address
        li $a1, 8               # Size of the array

        xor $t0, $t0, $t0       # Status=Unsorted
        addi $t1, $a1, -1       # #comprisons = ArraySize-1

While:
        beq $t1, $zero, EndWhile   #while (#comparisons<>0 AND status = Unsorted)
        bne $t0, $zero, EndWhile
        li $t0, 1               # Status = Sorted
        xor $t3, $t3, $t3       # for (i= 0 to #comparisons)
ForLoop:
        beq $t3, $t1, EndFor
        sll $t4, $t3, 2         # if (Array[i] > Array[i+1])
        add $t4, $a0, $t4
        lw  $t5, ($t4)
        lw $t6, 4($t4)
        ble $t5, $t6, EndIf
        sw  $t5, 4($t4)         # swap ith and (i+1)th elements of the array
        sw $t6,  ($t4)
        xor $t0, $t0, $t0       # Status=Unsorted
EndIf:
        addi $t3, $t3, 1
        j ForLoop
EndFor:
        addi $t1, $t1, -1       # #comparisons = #comparisons – 1
        j While
EndWhile:

# Return to operating system
        li      $v0, 10         # Load the system call number.
        syscall                 # Return.
```