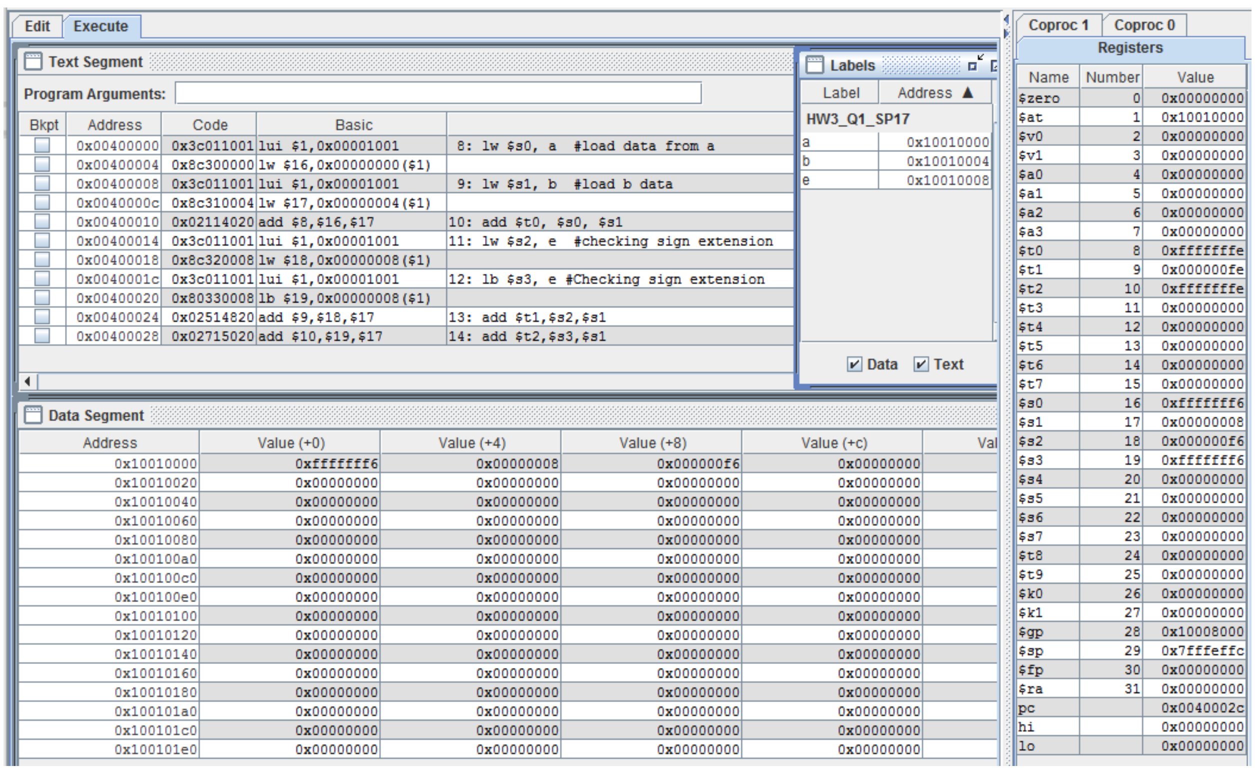
**Solution**

## HW#4 (CSC390-Spring 2018)
## Due: 02/16/2018 by 11:00PM

Q1. Consider the following MIPS assembly codes which perform a simple mathematical operation (-10+8) using the variables a, b, and e. The value -10 is assigned to variables a and e, as shown in lines 2 and 4, respectively.

```
1   .data    #declare data segment
2   a: .word -10
3   b: .word 8
4   e: .byte -10
5
6   .text #code segment
7
8   lw $s0, a  #load data from a
9   lw $s1, b  #load b data
10  add $t0, $s0, $s1
11  lw $s2, e  #checking sign extension
12  lb $s3, e #Checking sign extension
13  add $t1,$s2,$s1
14  add $t2,$s3,$s1
```

When we assembled the program, the first three locations of the Data-Segment are initialized with the values of a, b, and e, as shown in the following figure. Explain the questions on the next page.



i) Observe that for the same value of -10 the first location of the Data-Segment has 0xfffffff6 and the third location has 0x000000f6. Explain why?

**Answer:** For a word type (32 bits), 0xfffffff6 is the 2’s complement representation of -10. However for a byte type (8 bits), f6 is the 2’s complement representation of -10. Since in the data segment each column (in Fig2) has 32 bits the remaining bits are filled with zeros tha’s why we get 0x000000f6.

ii) Line 10 of the code is performing (-10+8) operation. Check whether you are getting the correct result in $t0.

**Answer:** (-10+8)=-2. In 2’s Complement 32 bit system -2 is 0xfffffffe. Since, after executing the program $t0 has 0xfffffffe, the result is correct.

iii) Lines 11 and 12 are loading the value of the variable “e” into the registers $s2 and $s3 respectively. Observe that after executing the program $s2 has 0x000000f6 and $s3 has 0xfffffff6. Explain Why?

Answer: In line 4, the variable “e” is defined as byte type. When a byte type data is loaded with “lw” then there will be no sign extension for negative number. That’s why in the $s2 we get 0x000000f6. However, when a byte type data is loaded with “lb” there will be sign extension of the negative number (that means all the remaining bits in the register will be filled with 1s). That’s why $s3 has 0xfffffff6.

iv) Observe that lines 13 and 14 are performing the same mathematical operation, (-10+8), and storing the results in $t1 and $t2, respectively. Check which one has the correct result. Explain why?

Answer: $t2 has the correct result. Since $s3 has -10 represented in 32-bit 2’s complement system, we get the correct results--i.e. -2 represented in 32-bit 2’s complement system. Observe that $s2 has a positive number (since there was no sign extension when -10 was loaded in $s2). Thus in line 13 basically two positive numbers are added, i.e. (f6+8)=fe. The results is not correct.

**Q2.** Write a MIPS assembly code to transfer data from register $s0 to $t0 without using Load and Store instructions. Solution:

**add $t0, $s0, $zero**

**Q3.** Write down the **machine code** of the following R-format instruction showing every instruction fields.

**add $t3, $S3, $S4**

**ANSWER:** for the above instruction following are the different fields we need to get the Machine.

Opcode: 000000 in binary

rs= $s3 (which is numbered as $19). Thus rs= 10011 in binary

rt=$s4 (which is numbered as $20). Thus, rt=10100 in binary

rd=$t3 (which is numbered as $11). Thus rd=01011 in binary

shamt = 00000 in binary (for add, sub instruction it is considered as zero)

funct=100000 in binary (for addition operation funct=32 in decimal)

Thus combining all the fields, we get:

0000 0010 0111 0100 0101 1000 0010 0000 = **02745820** in hexadecimal

The result is verified with MARS simulation

**Q4.** Write a MIPS assembly language program that calls a procedure, Add_Sub_Mul, which accept four parameters (g,h,i,j) and returns,

f = (g+h) if i >j; f= (g-h) if i < j; and f = g*h if i == j; the equivalent C function is shown below:

```
int Add_Sum_Mul (int g, int h, int i, int j) {
  int f;
 if  (i > j) {
        f=(g+h);}
  else if (i < j) {
        f=(g-h);}
 else if (i==j){
        f=g*h}
      }
```

Consider the variables g, h, i, j and f are initialized with some initial values in the data segment. Use $s0 as f in the function and also use $s0 to store the base address of f in the memory location. Clearly comment on the every instruction you use in your program. Specially, clearly show and describe the stack operation. Remember, resisters ($a0-$a2) are used for passing arguments in to the function and $v resisters are used to store the results in the function.

Solution:

```
# HW4_Q4
.data
g: .word 12
h: .word 6
i: .word 1
j: .word 0
f: .word  0 # store the result
.text
# put the data in the argument registers.
# arguments resisters are used to pass-parameters in the procedure (function)
lw $a0, g
lw $a1, h
lw $a2, i
lw $a3, j
la $s0, f # store the location of f to store results

# Call the Add_Sub_Mul
jal Add_Sub_Mul
sw $v0, 0($s0) # store the return value (i.e result) from the function into f
j halt
# Add_Sub_Mul Procedure
Add_Sub_Mul:
addi $sp, $sp, -4  # reserve space in the stack to store $s0
sw   $s0, 0($sp)  # save $s0 into the stack

#test if i==j
bne $a2, $a3, UnEq
mul $s0, $a0,$a1 # f=g*h
j store_result

#test condition if (i > j) or (i<j)
UnEq:
slt $t0, $a2, $a3  #set $t0 if (i < j)
beq $t0, $zero, GrTh # go to GrTh if (i>j)
sub $s0, $a0, $a1 # f = g-h
j store_result

GrTh:
add  $s0, $a0, $a1  # perform (g+h)
```

***Next Page:***

```
store_result:
add  $v0, $s0, $zero  # put the result in the return resister $v0
lw   $s0, 0($sp)  #restore $s0 for the caller
addi $sp, $sp, 4  #free-up stack space
jr   $ra     #jump back to the calling program

halt:
nop
```