SOLUTION:

CSC390 (Computer Org. & Arch.)
Exam#2- Spring 2018

Name:

Student ID:

#Q1. Consider the following C procedure and its corresponding MIPS assembly code which simply swap two adjacent memory locations contents. Note that the given assembly code works only for integer data type. The parameters, v and k, will be found in registers $a0 and $a1. The only other variable temp is associated with register $t0. [10 pts]

```
void swap(int v[], int k)
{
 int temp;
 temp = v[k];
 v[k] = v[k+1];
 v[k+1] = temp;
}
```

Procedure body

```
swap: sll   $t1, $a1, 2      # reg $t1 = k * 4
      add   $t1, $a0, $t1    # reg $t1 = v + (k * 4)
                             # reg $t1 has the address of v[k]
      lw    $t0, 0($t1)      # reg $t0 (temp) = v[k]
      lw    $t2, 4($t1)      # reg $t2 = v[k + 1]
                             # refers to next element of v
      sw    $t2, 0($t1)      # v[k] = reg $t2
      sw    $t0, 4($t1)      # v[k+1] = reg $t0 (temp)
```

What changes you would make in the above MIPS Assembly code so that it will work for **single precision floating point** data type? Write down the modified code. Include Syscall function to display the results.

Solution Next page:

Modified Code:

```
# Exam#2 Q1 Swap Algorithm
# Array= [100.00 50.00 75.00 -1.00 -50.00 500.00 20.00 40.00]

.data
Array: .double 100.00, 50.00, 75.00, -1.00, -50.00, 500.00, 20.00, 40.00

.text
la $a0, Array #load the base address of Array into the parameter register $a0
li $s1, 5 # Index of the element that will be swapped with next element

         move $a1, $s1           # Index K
         jal  swap               # call swap procedure

        #Printing the value at index 5 after swapping
        sll $t1, $a1, 3    # $t1 = k * 8
        add $t1, $a0, $t1 # $t1 = v+(k*8); #(address of v[k])

         l.d $f12, 0($t1) # Loading the value in $f12 pair

         li $v0,3                # for Syscall
         syscall
         j END

swap: sll $t1, $a1, 3    # $t1 = k * 8

      add $t1, $a0, $t1 # $t1 = v+(k*8)
                        #   (address of v[k])

      l.d $f2, 0($t1)     # $f2 (temp) = v[k]
      l.d $f4, 8($t1)     # $f4 = v[k+1]
      s.d $f4, 0($t1)     # v[k] = $f4 (v[k+1])
      s.d $f2, 8($t1)     # v[k+1] = $f2 (temp)
      jr $ra             # return to calling routine
END:
nop
```

```
Mars Messages | Run I/O

20.0
-- program is finished running (dropped off bottom) --

Clear
```

Q2. The following sequence of MIPS codes can detect overflow in unsigned addition of two unsigned numbers. Registers $t1 and $t2 contain the signed numbers.

```
addu $t0, $t1, $t2     # $t0 = sum
nor  $t3, $t1, $zero   # $t3 = NOT $t1
                       # (2's comp - 1: 2^32 - $t1 - 1)
sltu $t3, $t3, $t2     # (2^32 - $t1 - 1) < $t2
                       # ⇒ 2^32 - 1 < $t1 + $t2
bne $t3,$zero,Overflow # if(2^32-1<$t1+$t2) goto overflow
```

Consider, the code is working on two four-bit unsigned numbers stored in $t1 and $t2. For the following two pairs of $t1 and $t2, find if the code will detect any overflow in the addition operation. **Explain your results with the values of registers $t0 and $t3** as the program executes the sequence of the above MIPS assembly code (i.e. show the values of $t0 and $t3 for every line of the codes and justify your results).

[10 points]

a) $t1 = 0101;
   $t2 = 0010;

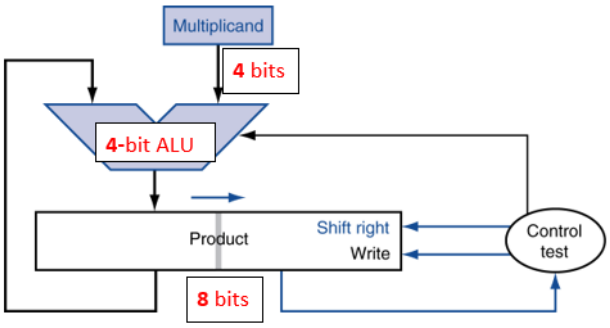
b) $t1=0110;
   $t2=0100;

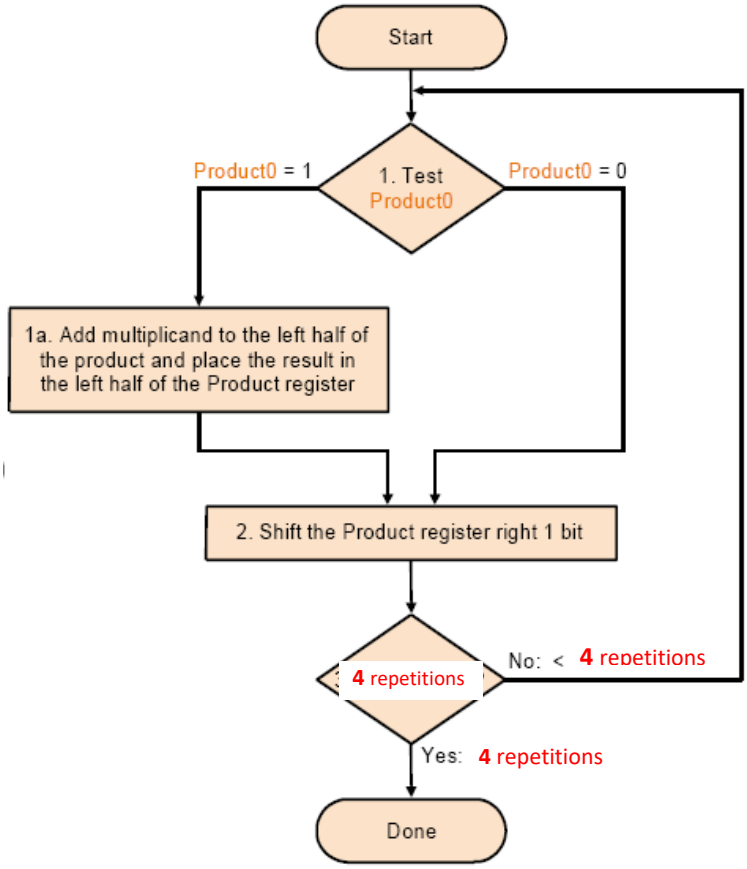
Solution:

(a) No overflow. Explanation below:
$t0 = 0111 ; addu $t0, $t1, $t2
$t3 = 1010 ; nor $t3, $t1, $zero
$t3 = 0000; sltu $t3, $t3, $t2
Not satisfied; bne $t3, $zero, No_overflow

(b) No Overflow detected. Explanation below:
$t0 = 1010 ; addu $t0, $t1, $t2
$t3 = 1001 ; nor $t3, $t1, $zero
$t3 = 0000; sltu $t3, $t3, $t2
Not satisfied; bne $t3, $zero, No_overflow

Q3. (a) The following Circuit performs the multiplication of two 32-bit unsigned binary numbers and produces a 64-bit result. Figure 1 shows the three steps multiplication algorithm that the circuit performs to produce the result. Suppose, you are assigned to design a 4-bit multiplier circuit which will perform the multiplication of the two 4-bit unsigned numbers A=$(1101)_2$ and B=$(1011)_2$ and produce an 8-bit result. **Modify hardware and the flow chart of the algorithm**. Tell me **how many addition and shifting operations** would be **required** to perform the above multiplication. [10 pts]





Since B=$(1011)_2$, there will be 3 addition and 4 shifting operations in the multiplication process

Q3. (b) The circuit in part(a) required several processing steps to perform the multiplication. To improve the execution speed of the above multiplier circuits, a Fast Multiplication Hardware, as shown in figure 3(b), is designed using (n-1) adder circuits, which perform the multiplication of two n-bit numbers and produces the 2n-bit result. Suppose, you are assigned to design a 4-bit Fast Multiplication Hardware circuit which will perform the multiplication of the two 4-bit unsigned numbers A=$(1101)_2$ and B=$(1011)_2$ and produce a 8-bit result. **Draw the necessary hardware** and show how the result is produced in the multiplication process. [10 pts]

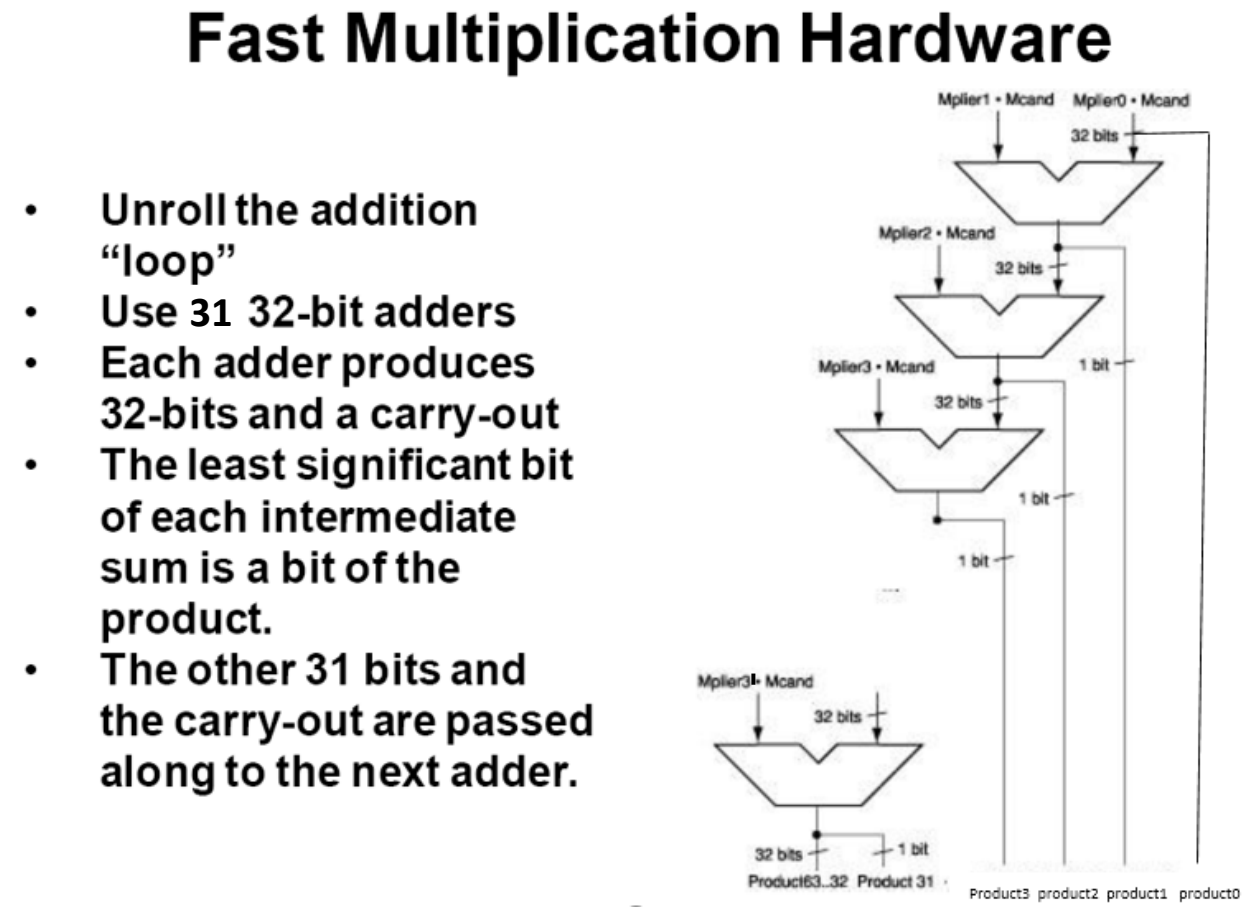


Figure 1(b)

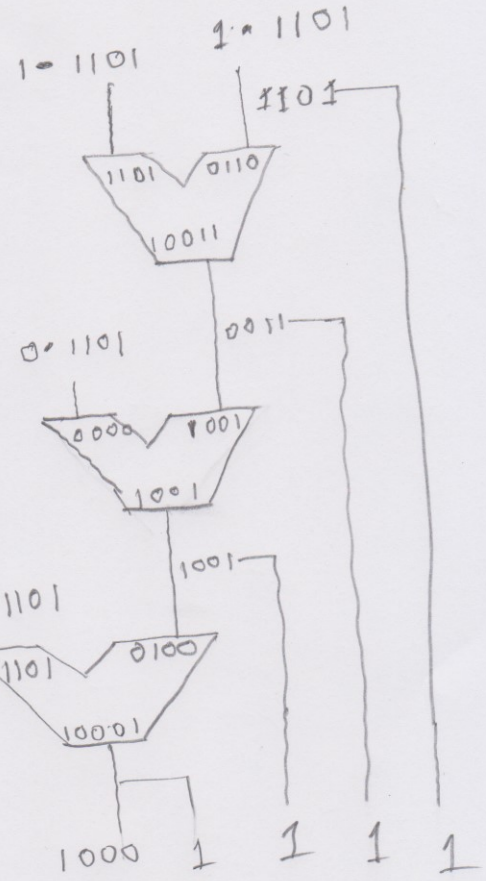
$(1101)_2 = (13)_{10}$

$(1011)_2 = (11)_{10}$

13
11

13
13×

143



1000 1 1 1 1

$= 128 + 15 = (143)_{10}$

Q4. (a) IEEE 754 binary representation of floating point numbers is widely used in today's computer systems. Consider the decimal number $(-0.09375)_{10}$, represent it in a single precision IEEE 754 format. Show all the steps of your calculation and the final results in Hex format. [8 pts]

$(-0.09375)_{10} = (-0.00011)_2$

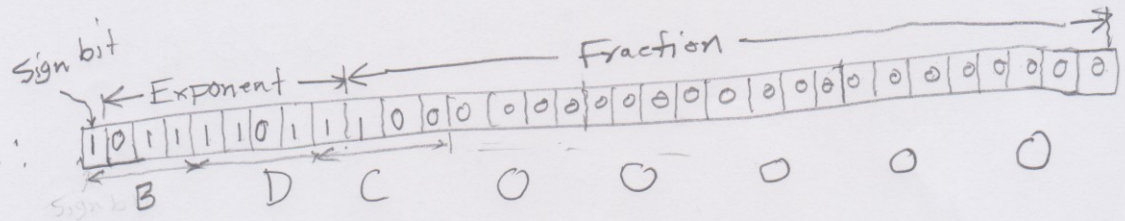
$\therefore$ Normalize $(-1.1) \times 2^{-4}$

Exponent = Actual exponent + Bias

$= -4 + 127 = (123)_{10}$

$= (01111011)_2$

Fraction: 1000 ... 000 (23 bits)



$(BDC00000)_{16}$

7

Q4. (b) Let's consider a 11-bit binary floating-point number system, in which -like IEEE 754 number system- 1 sign bit, 5 Exponent bits and 5 fraction bits are used to represent a floating-point number. Two such floating point numbers, as shown on figure 4(b), are added following the flow chart of Figure 4. The circuit that implements the flow chart is shown in figure 4(b).

Note that the Exponent = actual exponent + Bias, where Bias = $2^{n-1} - 1$; n = number of bits in Exponent and like the IEEE 754 system, the decimal equivalent of the binary floating point number is represented by: $(-1)^S x(1+\text{Fraction})x2^{(\text{Exponent} - \text{Bias})}$ (**ref:** Section 3.5 of your Text)

(a) Without doing any calculation tell me which floating point number has the smaller value and then calculate the corresponding decimal value. [7 pts]

(b) Write a short note on the operations of the controller circuit (i.e. oval shape control box) of Figure 4(b). Specifically discuss its role in the different steps of the floating-point addition process. [8 pts]

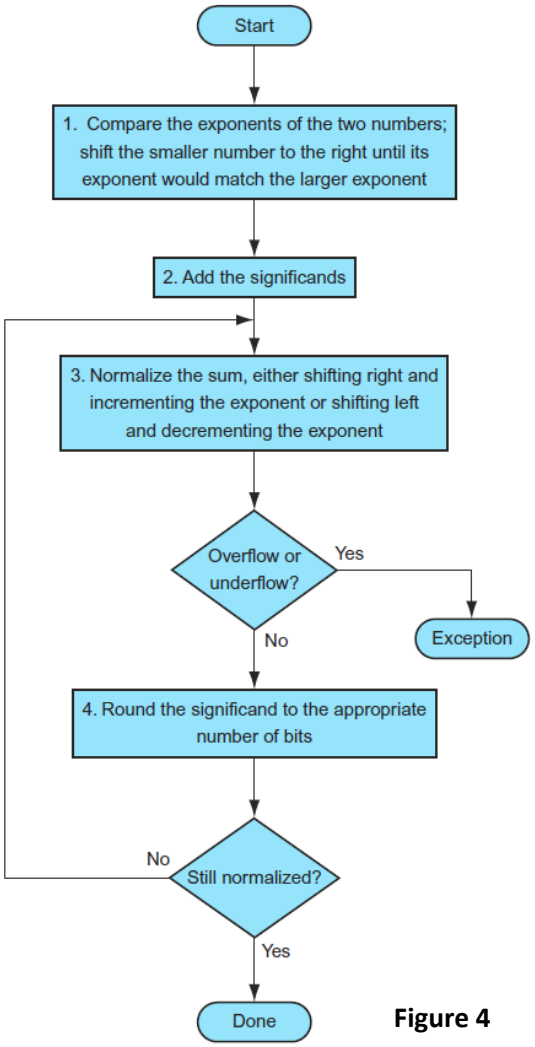


Figure 4

10-bit Binary floating point numbers:

0 01101 11101	0 10110 11 0 10

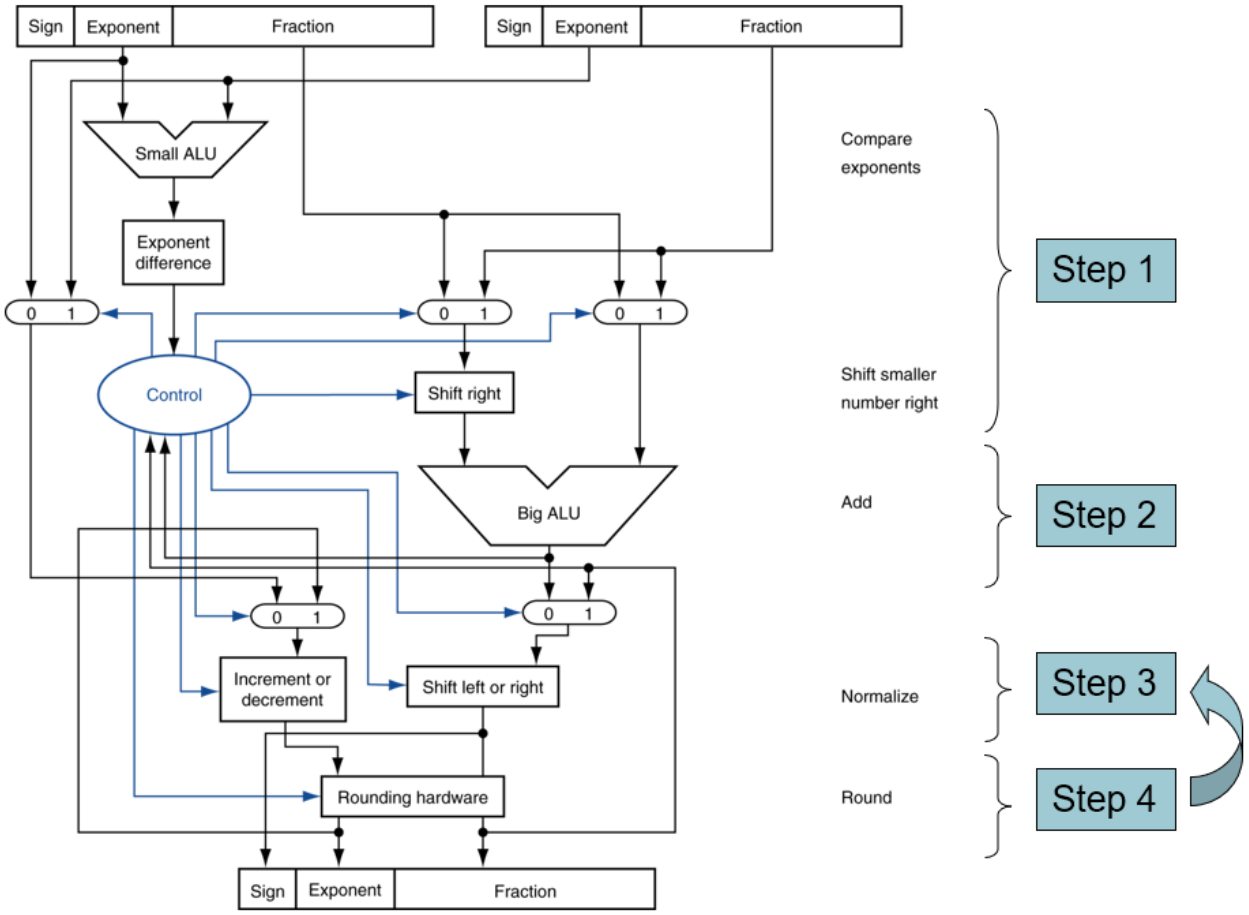


Figure 4(b)

Solution:

(a) The number on the left hand side, 0 0 1 1 0 1 1 1 1 0 1 , is smaller since it has lower exponent value compared to the other number.

$(-1)^0 \times (1 + 2^{-1} + 2^{-2} + 2^{-3} + 2^{-5}) \times 2^{(13-15)} = 0.4766$

(b) Controller functions:

In Step#1, after getting the exponent difference, the controller send appropriate input values to the three Multiplexers to select largest exponent, to channel the fraction of the smaller numbers in to the right-shift box, the fraction of the larger exponent is also channeled to the other input of the Big Alu.

-send proper shift_right value to right shift the fraction of the smaller number so that the exponent of the smaller number becomes same as the exponent of the other (largest) number.

At the beginning of step #3, the controller also selects the proper value of the multiplexors inputs so that it can keep the results normalized by incrementing or decrementing the exponent and shifting left or right the fraction bits.

During the iterative process the controller adjusts the exponent and the fractions of the result to keep it as normalized after rounding operation.

The controller also sends appropriate rounding signal to the rounding hardware to fit the results in the desired format.

Q5. Consider, X[i][j] is a **double precision type** matrix of size 8x8. The following code converts the 2D indices (i.e. [i][j]) of the matrix to the corresponding linear indexes of the memory where the matrix data is stored. Note that the registers **$S0** and **$S1** correspond to the **i** and **j** indexes, respectively. Also, register **$a0** contains the base address (i.e. starting memory address) of X.

```
sll  $t2, $s0, 3    # $t2 = i * 8 (size of row of x)
addu $t2, $t2, $s1  # $t2 = i * size(row) + j
sll  $t2, $t2, 3    # $t2 = physical offset of [i][j]
addu $t2, $a0, $t2  # $t2 = physical address of x[i][j]
```

(a) Given, **i = 2, j = 3**, and the base address of X, i.e. **$a0** contains 200_{10}, calculate the corresponding physical address of the memory location. [5 pts]

(b) What changes would you make in the above codes so that the modified code would work for a matrix X[i][j] of size **16x16** and each element is of **single precision floating point** type? [12 pts]

Solution:

(a) The physical address= [(linearized index of the matrix element) X (number of bytes in the data type)] + (base address of the matrix)
Linearized index of the matrix element= (i x row length)+j = 2 x 8+3=19
Thus **the physical address**= (19x8 + 200) = **352**

(b)

```
sll  $t2, $s0, 4    # $t2 = i * 16 (size of row of x)
addu $t2, $t2, $s1  # $t2 = i * size(row) + j
sll  $t2, $t2, 2    # $t2 = byte offset of [i][j]
addu $t2, $a0, $t2  # $t2 = byte address of x[i][j]
```