HW#9 (CSC390) **Due:** 04/07/2018 by 11:30PM

Q1. Let's consider a 10-bit binary floating-point number system, in which -like IEEE 754 number system- 1 sign bit, 4 Exponent bits and 5 fraction bits are used to represent a floating-point number. Two such floating point numbers, as shown on figure 2, are added following the flow chart of Figure 1. The circuit that implements the flow chart is shown in figure 2.
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) Convert the two binary floating-point numbers to their corresponding decimal equivalent numbers.
(b) Follow the flow chart of figure 1 and tabulate the values you would get after every step of the iterative process. Indicate the selection line values for the three Multiplexers (as shown in step – 1) to choose the proper values of the Exponent and Fractions. Verify that the adder circuit produces correct results.

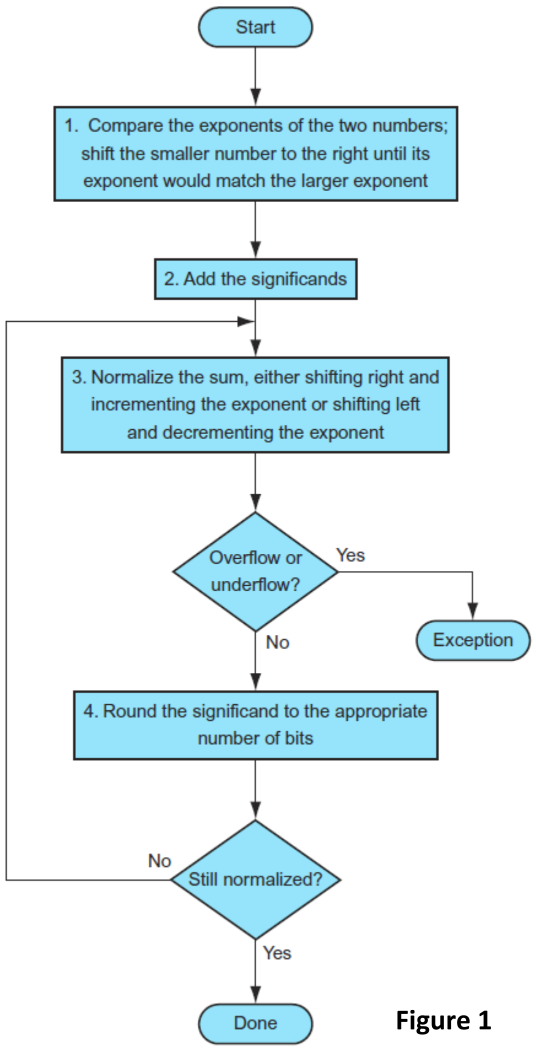

Start
1. Compare the exponents of the two numbers; shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
Done

Figure 1

10-bit Binary floating point numbers:

1	1101	11101

0	0110	11010

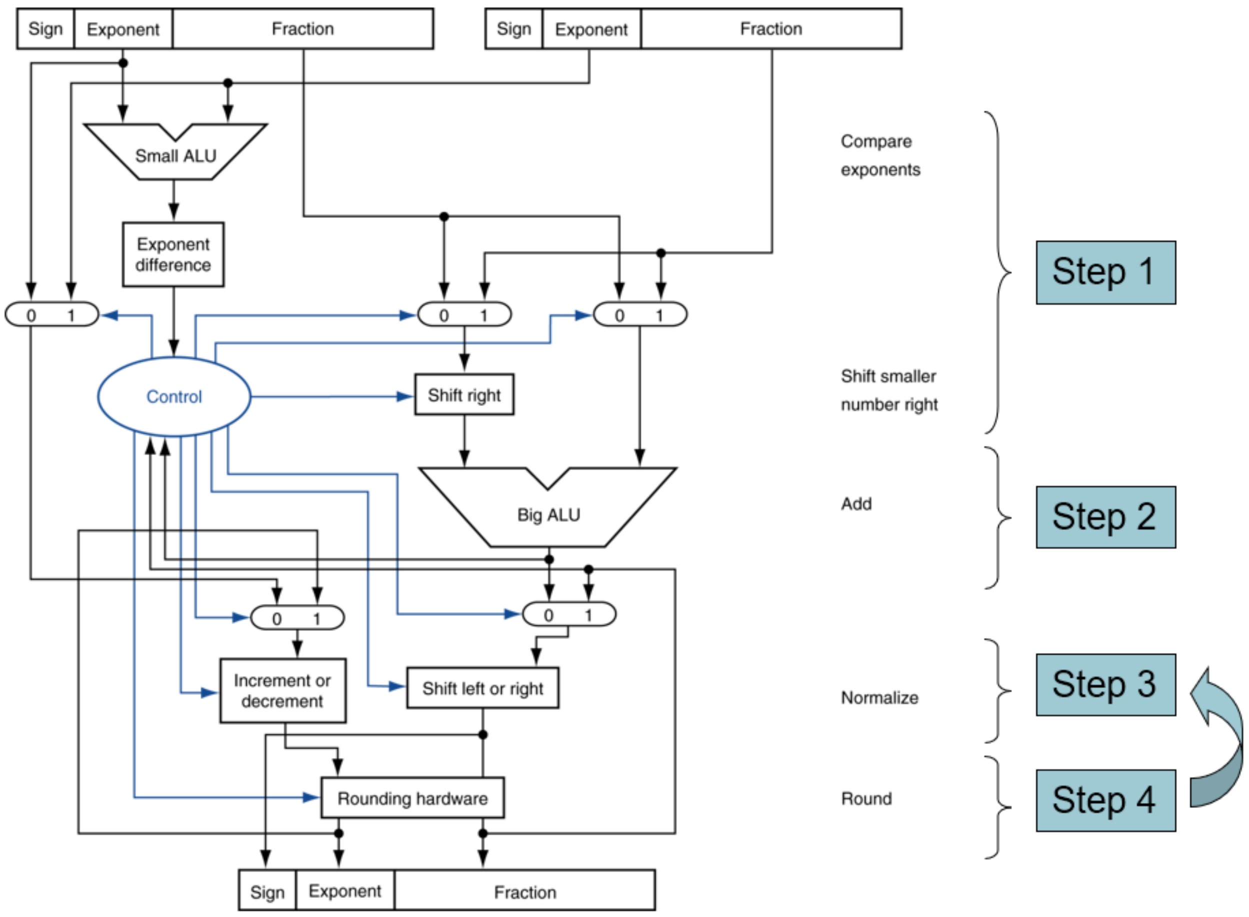


Figure 2

Q2. Now use the flow chart of Figure 3 to perform the floating-point multiplication of the Binary floating-point numbers given in Question 1 (figure 2). Clearly show the values you would get in every step of the iterative process.

Q2. (Continue)

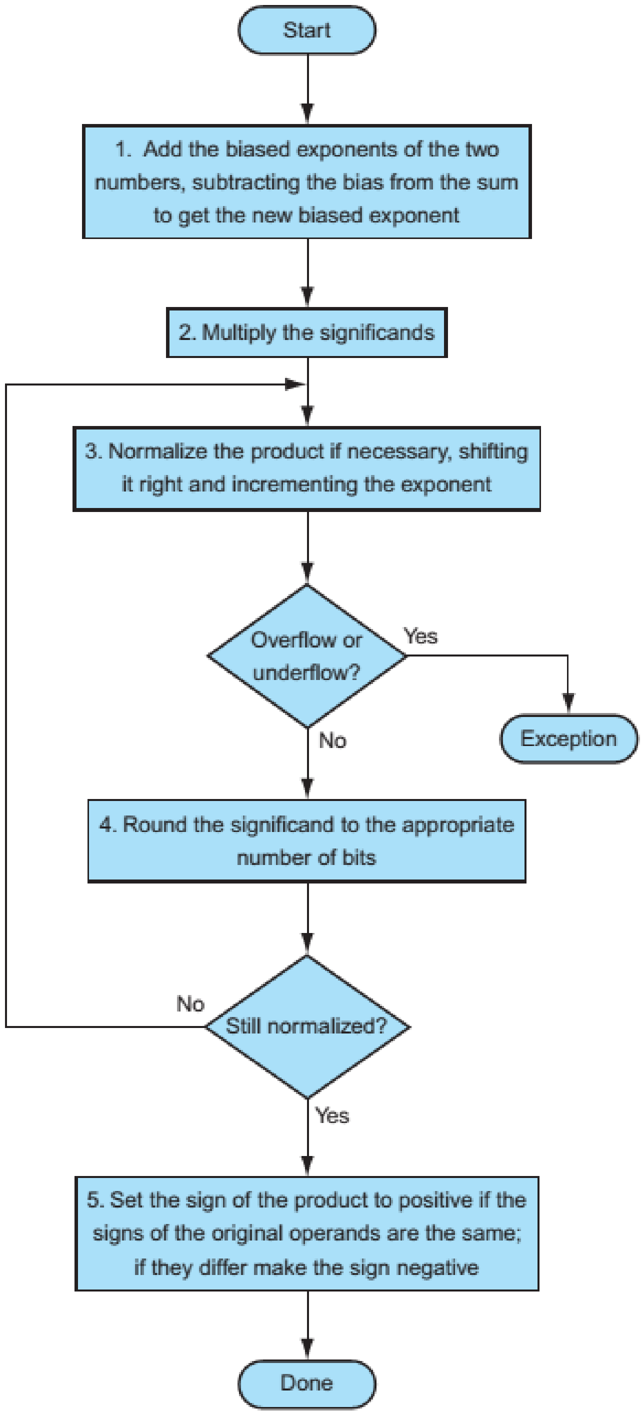
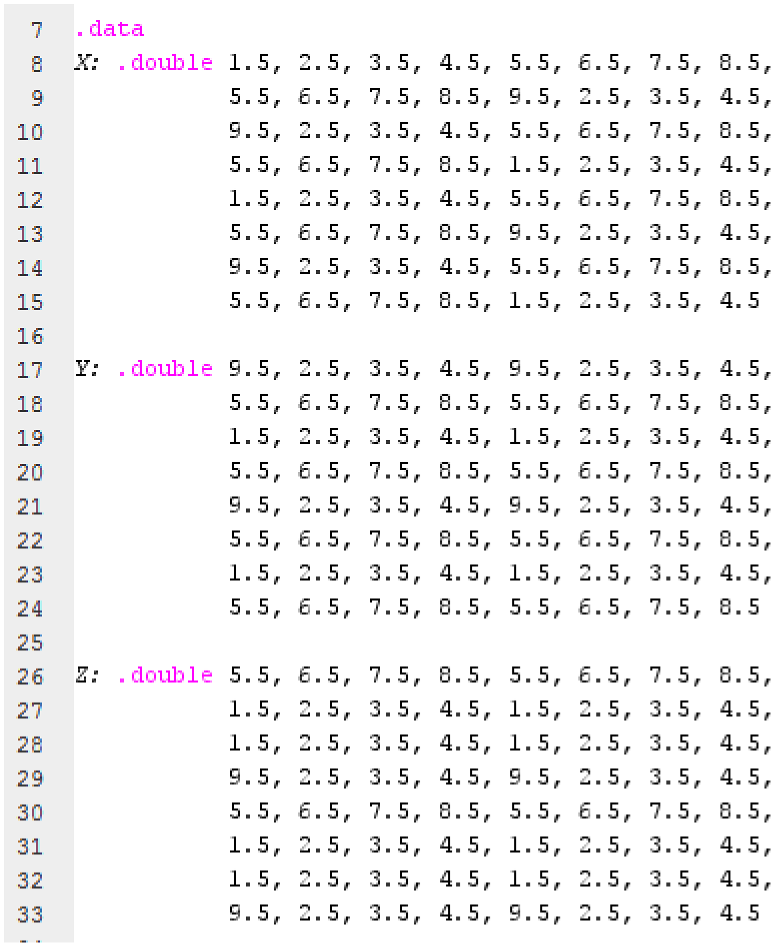

Start
1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent
2. Multiply the significands
3. Normalize the product if necessary, shifting it right and incrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
5. Set the sign of the product to positive if the signs of the original operands are the same; if they differ make the sign negative
Done

Figure 3

Q3. Write a MIPS assembly program to calculate the area of a rectangle using the floating point data (length and width of the rectangle) supplied from the keyboard. Your program should also display the results on the MIPS editor.
Hints: use "syscall" function discussed in our last class and example codes posted on the blackboard.

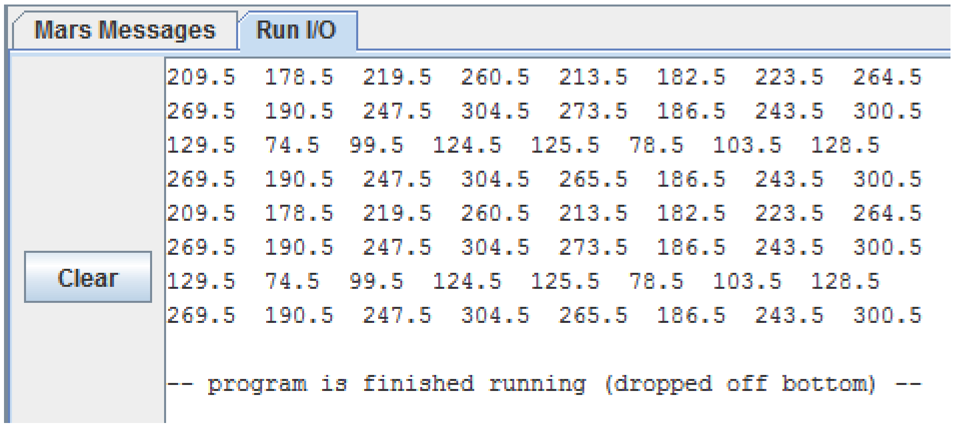
Q4. The following code (as shown in fig. 7, which is also posted on the blackboard) performs the matrix operation, X=X+Y*Z, and display the results on the MIPS editor as shown in Figure 8. Please note that X, Y, and Z are square matrixes of size **4x4** and the elements of the matrixes are double precision floating point number.

(a) Now, modify the code in such a way that it would perform:
X = X+Y*Z; where X, Y, and Z are **8x8 matrixes** (as shown in fig 4) and each element is represented by double precision number. Your program should also display the result as shown in figure 5.
(Note that results are verified using Matlab)



```
.data
X: .double 1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5, 8.5,
           5.5, 6.5, 7.5, 8.5, 9.5, 2.5, 3.5, 4.5,
           9.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5, 8.5,
           5.5, 6.5, 7.5, 8.5, 1.5, 2.5, 3.5, 4.5,
           1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5, 8.5,
           5.5, 6.5, 7.5, 8.5, 9.5, 2.5, 3.5, 4.5,
           9.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5, 8.5,
           5.5, 6.5, 7.5, 8.5, 1.5, 2.5, 3.5, 4.5

Y: .double 9.5, 2.5, 3.5, 4.5, 9.5, 2.5, 3.5, 4.5,
           5.5, 6.5, 7.5, 8.5, 5.5, 6.5, 7.5, 8.5,
           1.5, 2.5, 3.5, 4.5, 1.5, 2.5, 3.5, 4.5,
           5.5, 6.5, 7.5, 8.5, 5.5, 6.5, 7.5, 8.5,
           9.5, 2.5, 3.5, 4.5, 9.5, 2.5, 3.5, 4.5,
           5.5, 6.5, 7.5, 8.5, 5.5, 6.5, 7.5, 8.5,
           1.5, 2.5, 3.5, 4.5, 1.5, 2.5, 3.5, 4.5,
           5.5, 6.5, 7.5, 8.5, 5.5, 6.5, 7.5, 8.5

Z: .double 5.5, 6.5, 7.5, 8.5, 5.5, 6.5, 7.5, 8.5,
           1.5, 2.5, 3.5, 4.5, 1.5, 2.5, 3.5, 4.5,
           1.5, 2.5, 3.5, 4.5, 1.5, 2.5, 3.5, 4.5,
           9.5, 2.5, 3.5, 4.5, 9.5, 2.5, 3.5, 4.5,
           5.5, 6.5, 7.5, 8.5, 5.5, 6.5, 7.5, 8.5,
           1.5, 2.5, 3.5, 4.5, 1.5, 2.5, 3.5, 4.5,
           1.5, 2.5, 3.5, 4.5, 1.5, 2.5, 3.5, 4.5,
           9.5, 2.5, 3.5, 4.5, 9.5, 2.5, 3.5, 4.5
```

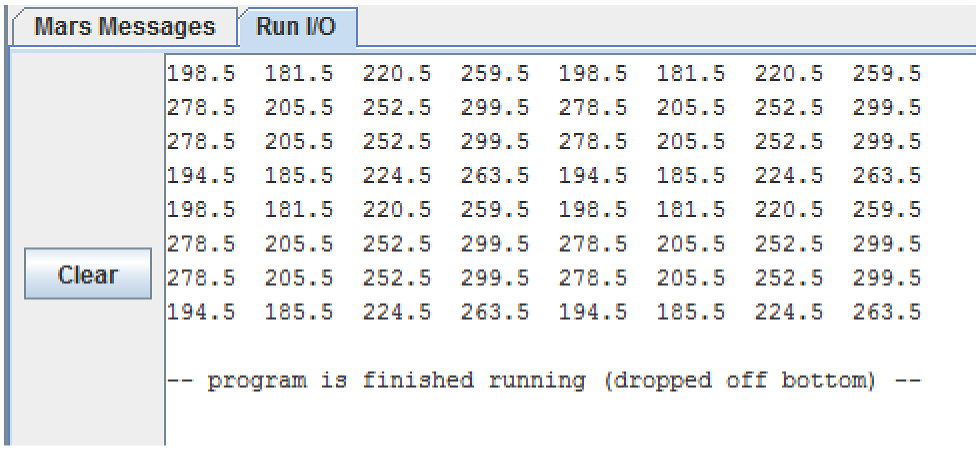
Figure 4



```
209.5  178.5  219.5  260.5  213.5  182.5  223.5  264.5
269.5  190.5  247.5  304.5  273.5  186.5  243.5  300.5
129.5  74.5  99.5  124.5  125.5  78.5  103.5  128.5
269.5  190.5  247.5  304.5  265.5  186.5  243.5  300.5
209.5  178.5  219.5  260.5  213.5  182.5  223.5  264.5
269.5  190.5  247.5  304.5  273.5  186.5  243.5  300.5
129.5  74.5  99.5  124.5  125.5  78.5  103.5  128.5
269.5  190.5  247.5  304.5  265.5  186.5  243.5  300.5

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

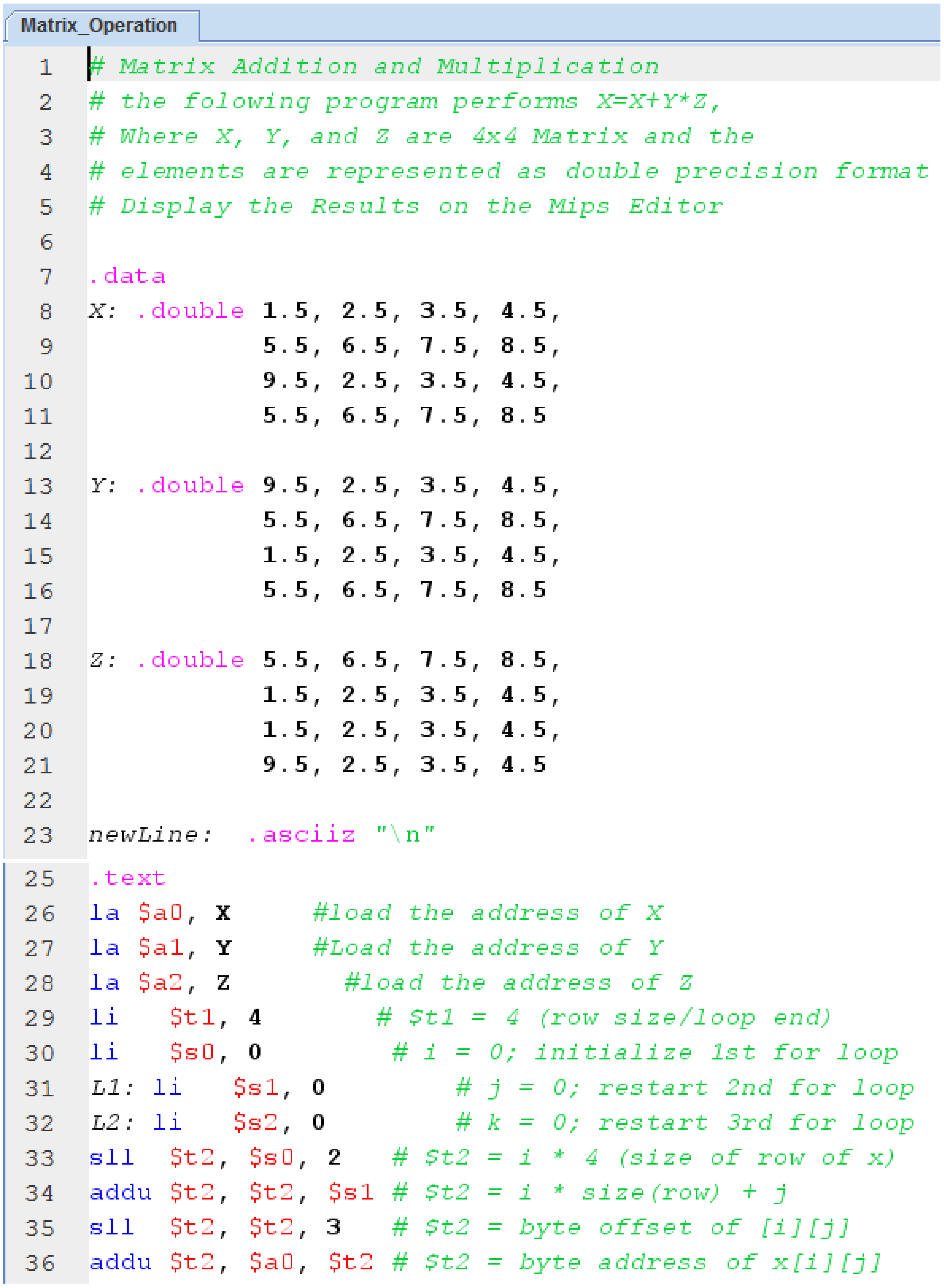
Figure 5

(b) Also, modify the code in such a way that it would perform:
Z = X*Y-Z; where X, Y, and Z are 8x8 matrixes (as shown in fig 4) and each element is represented by double precision number. Your program should also display the result as shown in figure 6.
(Note that this part would be more challenging than the first part. Please go through the pages 215-217 of your book. Pay attention to the Indexed operation and how the multiplication is done. You would need to use an extra instruction sub.d in addition to the add.d instruction. Add.d would be used to find each element of X*Y and sub.d would be used for X*Y-Z. Recall the Matrix multiplication operation. You may also need to use an extra floating point register to hold the X*Y value temporarily)

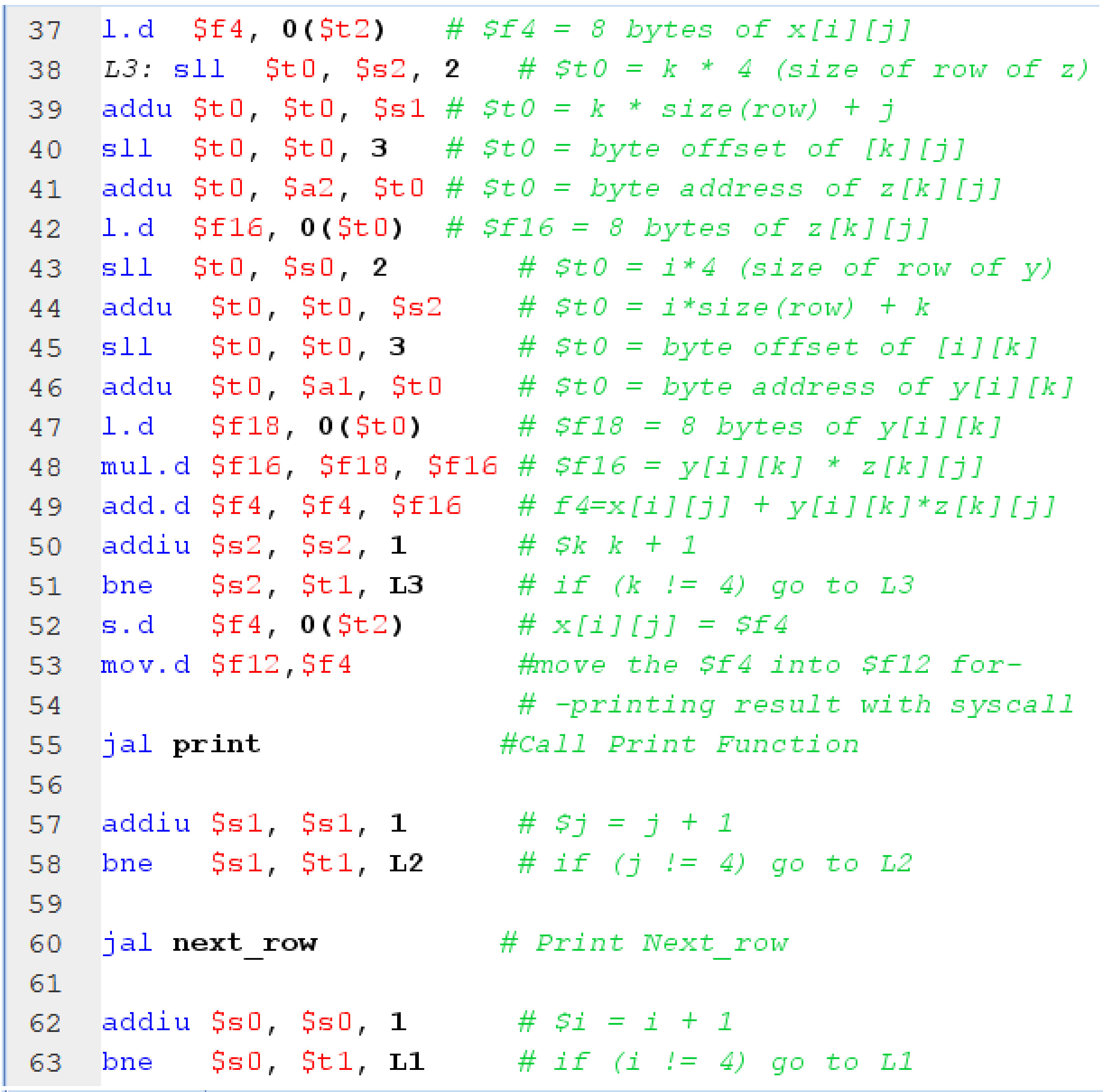


```
198.5  181.5  220.5  259.5  198.5  181.5  220.5  259.5
278.5  205.5  252.5  299.5  278.5  205.5  252.5  299.5
278.5  205.5  252.5  299.5  278.5  205.5  252.5  299.5
194.5  185.5  224.5  263.5  194.5  185.5  224.5  263.5
198.5  181.5  220.5  259.5  198.5  181.5  220.5  259.5
278.5  205.5  252.5  299.5  278.5  205.5  252.5  299.5
278.5  205.5  252.5  299.5  278.5  205.5  252.5  299.5
194.5  185.5  224.5  263.5  194.5  185.5  224.5  263.5

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

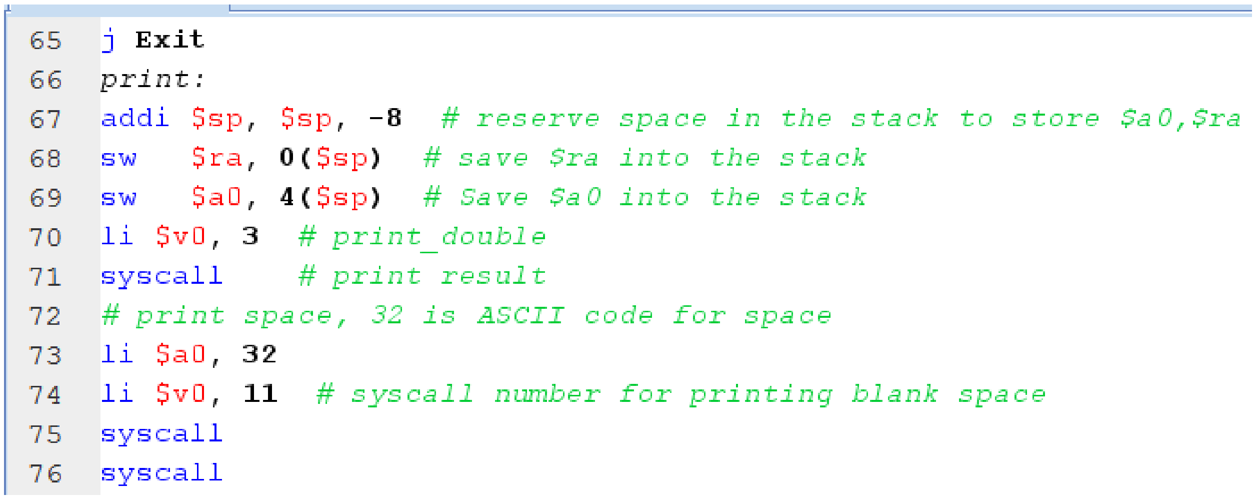
Figure 6



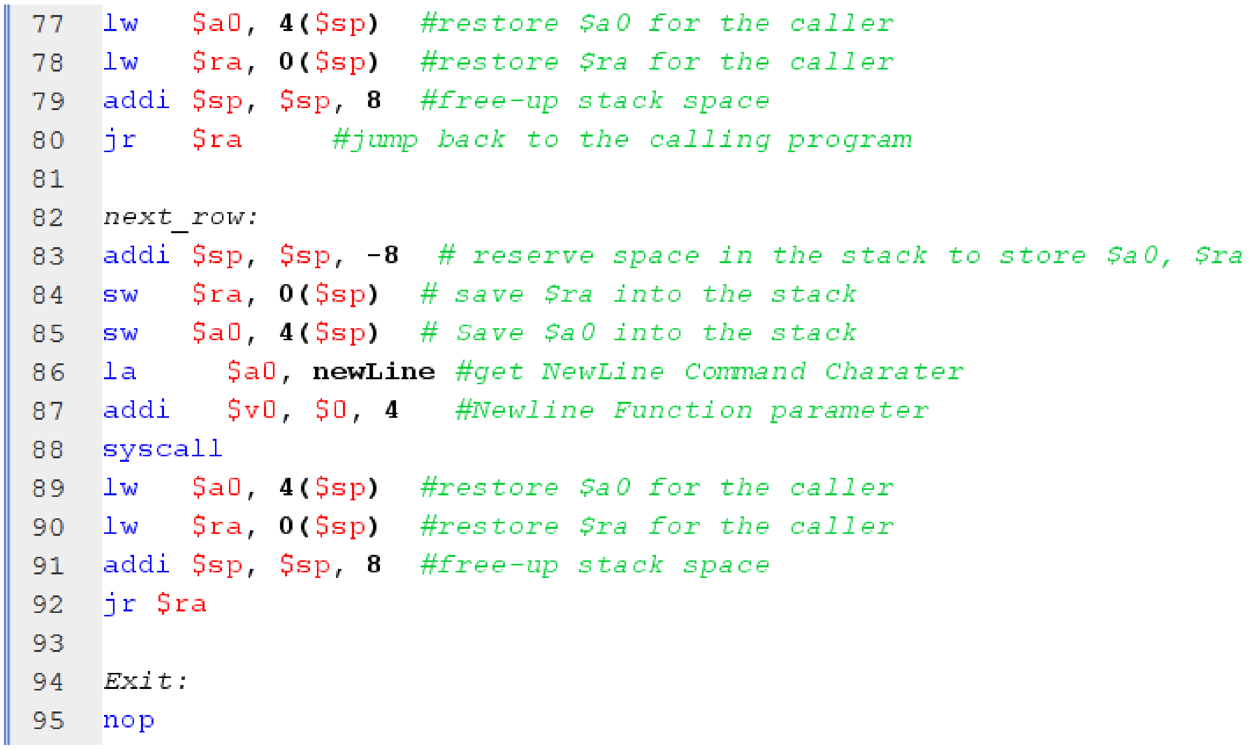
```
# Matrix Addition and Multiplication
# the folowing program performs X=X+Y*Z,
# Where X, Y, and Z are 4x4 Matrix and the
# elements are represented as double precision format
# Display the Results on the Mips Editor

.data
X: .double 1.5, 2.5, 3.5, 4.5,
           5.5, 6.5, 7.5, 8.5,
           9.5, 2.5, 3.5, 4.5,
           5.5, 6.5, 7.5, 8.5

Y: .double 9.5, 2.5, 3.5, 4.5,
           5.5, 6.5, 7.5, 8.5,
           1.5, 2.5, 3.5, 4.5,
           5.5, 6.5, 7.5, 8.5

Z: .double 5.5, 6.5, 7.5, 8.5,
           1.5, 2.5, 3.5, 4.5,
           1.5, 2.5, 3.5, 4.5,
           9.5, 2.5, 3.5, 4.5

newLine:  .asciiz "\n"
.text
la $a0, X      #load the address of X
la $a1, Y      #Load the address of Y
la $a2, Z        #load the address of Z
li   $t1, 4        # $t1 = 4 (row size/loop end)
li   $s0, 0         # i = 0; initialize 1st for loop
L1: li   $s1, 0         # j = 0; restart 2nd for loop
L2: li   $s2, 0         # k = 0; restart 3rd for loop
sll  $t2, $s0, 2    # $t2 = i * 4 (size of row of x)
addu $t2, $t2, $s1 # $t2 = i * size(row) + j
sll  $t2, $t2, 3    # $t2 = byte offset of [i][j]
addu $t2, $a0, $t2 # $t2 = byte address of x[i][j]
```



```
l.d   $f4, 0($t2)    # $f4 = 8 bytes of x[i][j]
L3: sll  $t0, $s2, 2    # $t0 = k * 4 (size of row of z)
addu $t0, $t0, $s1 # $t0 = k * size(row) + j
sll  $t0, $t0, 3    # $t0 = byte offset of [k][j]
addu $t0, $a2, $t0 # $t0 = byte address of z[k][j]
l.d  $f16, 0($t0)  # $f16 = 8 bytes of z[k][j]
sll  $t0, $s0, 2         # $t0 = i*4 (size of row of y)
addu  $t0, $t0, $s2      # $t0 = i*size(row) + k
sll   $t0, $t0, 3        # $t0 = byte offset of [i][k]
addu  $t0, $a1, $t0      # $t0 = byte address of y[i][k]
l.d   $f18, 0($t0)       # $f18 = 8 bytes of y[i][k]
mul.d $f16, $f18, $f16 # $f16 = y[i][k] * z[k][j]
add.d $f4, $f4, $f16     # f4=x[i][j] + y[i][k]*z[k][j]
addiu $s2, $s2, 1        # $k k + 1
bne   $s2, $t1, L3       # if (k != 4) go to L3
s.d   $f4, 0($t2)        # x[i][j] = $f4
mov.d $f12,$f4           #move the $f4 into $f12 for-
                         # -printing result with syscall
jal print               #Call Print Function

addiu $s1, $s1, 1        # $j = j + 1
bne   $s1, $t1, L2       # if (j != 4) go to L2

jal next_row            # Print Next_row

addiu $s0, $s0, 1        # $i = i + 1
bne   $s0, $t1, L1       # if (i != 4) go to L1
```



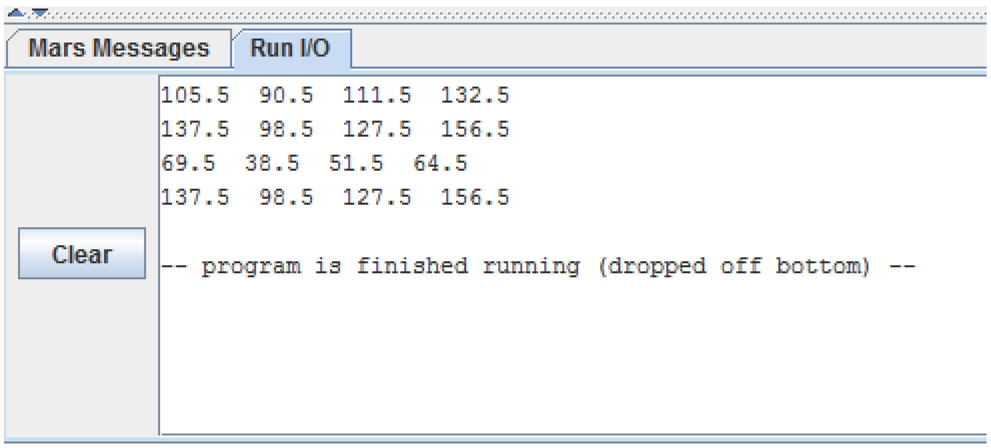
```
j Exit
print:
addi $sp, $sp, -8  # reserve space in the stack to store $a0,$ra
sw   $ra, 0($sp)  # save $ra into the stack
sw   $a0, 4($sp)  # Save $a0 into the stack
li $v0, 3  # print_double
syscall    # print result
# print space, 32 is ASCII code for space
li $a0, 32
li $v0, 11  # syscall number for printing blank space
syscall
syscall
```



```
lw    $a0, 4($sp)   #restore $a0 for the caller
lw    $ra, 0($sp)   #restore $ra for the caller
addi $sp, $sp, 8   #free-up stack space
jr    $ra      #jump back to the calling program

next_row:
addi $sp, $sp, -8   # reserve space in the stack to store $a0, $ra
sw    $ra, 0($sp)   # save $ra into the stack
sw    $a0, 4($sp)   # Save $a0 into the stack
la      $a0, newLine #get NewLine Command Charater
addi    $v0, $0, 4    #Newline Function parameter
syscall
lw    $a0, 4($sp)   #restore $a0 for the caller
lw    $ra, 0($sp)   #restore $ra for the caller
addi $sp, $sp, 8   #free-up stack space
jr $ra

Exit:
nop
```

Figure 7

Results:



```
105.5   90.5   111.5   132.5
137.5   98.5   127.5   156.5
69.5   38.5   51.5   64.5
137.5   98.5   127.5   156.5

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

Figure 8