CCCN221 – Computer Architecture
Lab 8 – Floating points in MIPS

In this lab, we will go through Floating-Point Number Representation (IEEE 754 Standard), have the basic understanding of MIPS Floating-Point Unit. Will write down programs using the MIPS Floating-Point Instructions that will have the input and output as the floating point numbers.

Floating-Point Number Representation

Floating-point numbers have been defined as follows

S	**E = Exponent**	**F = Fraction**

The Sign bit **S** is zero (positive) or one (negative).

For single-precision the Exponent field **E** has 8 bits and for double-precision, 11 bits. The exponent field is biased. The Bias is 127 for single-precision and 1023 for double-precision.

The Fraction field **F** is 23 bits for single-precision and 52 bits for double-precision. Floating-point numbers are normalized (except when **E** is zero). There is an implicit **1.** (not stored) before the fraction **F**. Therefore, the value of a normalized floating-point number is:

$$\textbf{Value} = \square\ (1.F)_2 \times 2^{E - Bias}$$

The QTSPIM simulator has a floating-point representation tool that illustrates single-precision floating-point numbers. The figure 1 shows the floating point representation.

Now use the tool to check the binary format and the decimal value of floating-point numbers.

For example, the decimal value of: **0 10000001 10110100000000000000000** is **6.75**.

Similarly, the 32-bit representation of: **-2.7531** is **1 10000000 01100000011001011001010**.

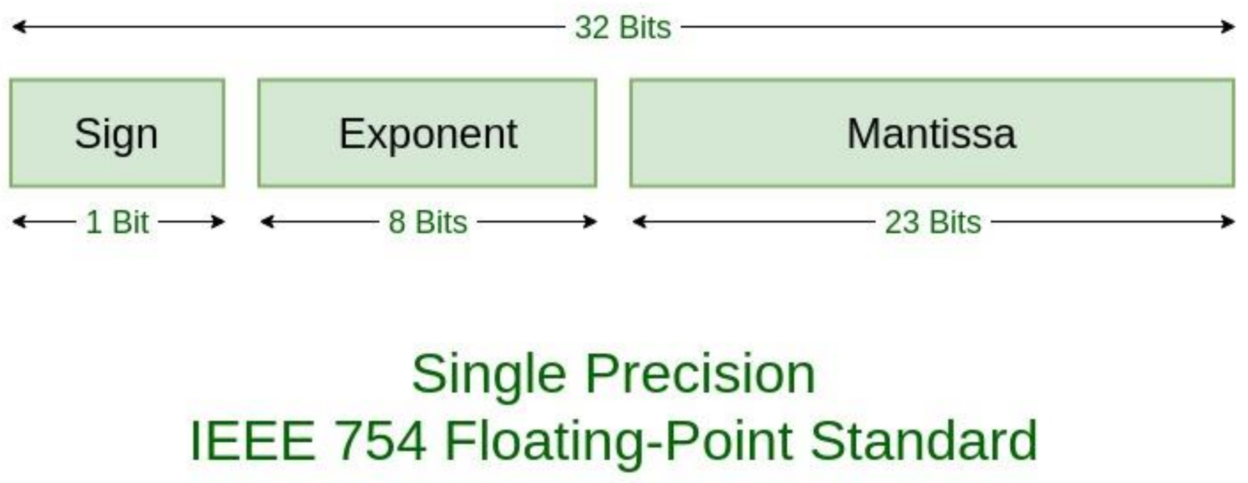
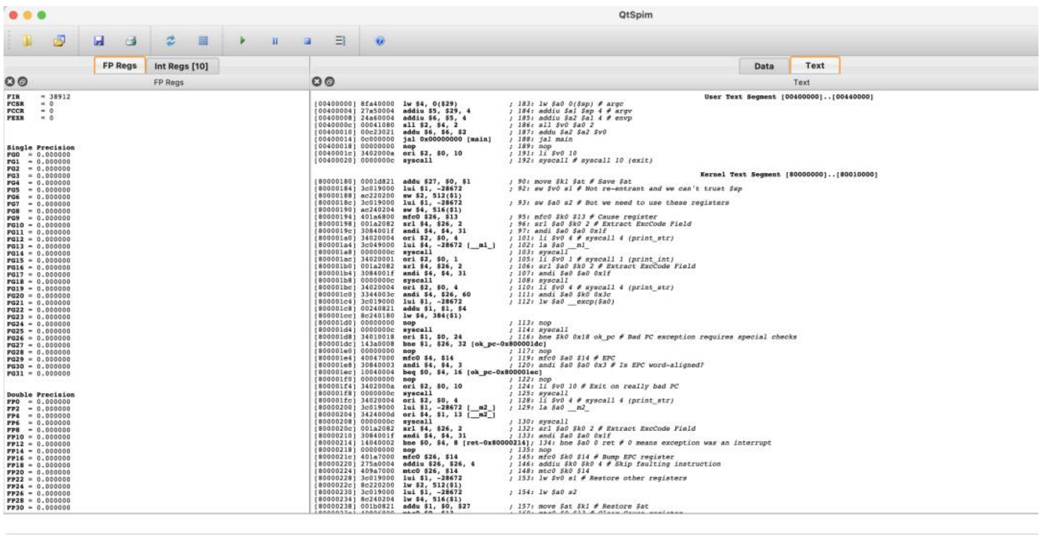


Figure 9.1: Floating-Point Representation

MIPS Floating-Point Registers

The floating-point unit (called coprocessor 1) has 32 floating-point registers. These registers are numbered as **$f0**, **$f1**, …, **$f31**. Each register is 32 bits wide. Thus, each register can hold one single-precision floating-point number. How can we use these registers to store 64-bit double- precision floating-point numbers? The answer is that the 32 single-precision registers are grouped into 16 double-precision registers. The double-precision number is stored in an even-odd pair of registers, but we only refer to the even-numbered register. For example, when we store a double- precision number in **$f0**, it is actually stored in registers **$f0**and **$f1**.

In addition, there are 8 condition flags, numbered from 0 to 7. These condition flags are used by floating-point compare and branch instructions. These are shown in Figure 9.2.



```
FIR    = 38912
FCSR   = 0
FCCR   = 0
FEXR   = 0


Single Precision
FG0  = 0.000000
FG1  = 0.000000
FG2  = 0.000000
FG3  = 0.000000
FG4  = 0.000000
FG5  = 0.000000
FG6  = 0.000000
FG7  = 0.000000
FG8  = 0.000000
FG9  = 0.000000
FG10 = 0.000000
FG11 = 0.000000
FG12 = 0.000000
FG13 = 0.000000
FG14 = 0.000000
FG15 = 0.000000
FG16 = 0.000000
FG17 = 0.000000
FG18 = 0.000000
FG19 = 0.000000
FG20 = 0.000000
FG21 = 0.000000
FG22 = 0.000000
FG23 = 0.000000
FG24 = 0.000000
FG25 = 0.000000
FG26 = 0.000000
FG27 = 0.000000
FG28 = 0.000000
FG29 = 0.000000
FG30 = 0.000000
FG31 = 0.000000

Double Precision
FP0  = 0.000000
FP2  = 0.000000
FP4  = 0.000000
FP6  = 0.000000
FP8  = 0.000000
FP10 = 0.000000
FP12 = 0.000000
FP14 = 0.000000
FP16 = 0.000000
FP18 = 0.000000
FP20 = 0.000000
FP22 = 0.000000
FP24 = 0.000000
FP26 = 0.000000
FP28 = 0.000000
FP30 = 0.000000
```

```
                                    User Text Segment [00400000]..[00440000]
[00400000] 8fa40000  lw $4, 0($29)            ; 183: lw $a0 0($sp) # argc
[00400004] 27a50004  addiu $5, $29, 4         ; 184: addiu $a1 $sp 4 # argv
[00400008] 24a60004  addiu $6, $5, 4          ; 185: addiu $a2 $a1 4 # envp
[0040000c] 00041080  sll $2, $4, 2            ; 186: sll $v0 $a0 2
[00400010] 00c23021  addu $6, $6, $2          ; 187: addu $a2 $a2 $v0
[00400014] 0c000000  jal 0x00000000 [main]    ; 188: jal main
[00400018] 00000000  nop                      ; 189: nop
[0040001c] 3402000a  ori $2, $0, 10           ; 191: li $v0 10
[00400020] 0000000c  syscall                  ; 192: syscall # syscall 10 (exit)

                                    Kernel Text Segment [80000000]..[80010000]
[80000180] 0001d821  addu $27, $0, $1         ; 90: move $k1 $at # Save $at
[80000184] 3c019000  lui $1, -28672           ; 92: sw $v0 s1 # Not re-entrant and we can't trust $sp
[80000188] ac220200  sw $2, 512($1)
[8000018c] 3c019000  lui $1, -28672           ; 93: sw $a0 s2 # But we need to use these registers
[80000190] ac240204  sw $4, 516($1)
[80000194] 401a6800  mfc0 $26, $13            ; 95: mfc0 $k0 $13 # Cause register
[80000198] 001a2082  srl $4, $26, 2           ; 96: srl $a0 $k0 2 # Extract ExcCode Field
[8000019c] 3084001f  andi $4, $4, 31          ; 97: andi $a0 $a0 0x1f
[800001a0] 34020004  ori $2, $0, 4            ; 101: li $v0 4 # syscall 4 (print_str)
[800001a4] 3c049000  lui $4, -28672 [__m1_]   ; 102: la $a0 __m1_
[800001a8] 0000000c  syscall                  ; 103: syscall
[800001ac] 34020001  ori $2, $0, 1            ; 105: li $v0 1 # syscall 1 (print_int)
[800001b0] 001a2082  srl $4, $26, 2           ; 106: srl $a0 $k0 2 # Extract ExcCode Field
[800001b4] 3084001f  andi $4, $4, 31          ; 107: andi $a0 $a0 0x1f
[800001b8] 0000000c  syscall                  ; 108: syscall
[800001bc] 34020004  ori $2, $0, 4            ; 110: li $v0 4 # syscall 4 (print_str)
[800001c0] 3344003c  andi $4, $26, 60         ; 111: andi $a0 $k0 0x3c
[800001c4] 3c019000  lui $1, -28672           ; 112: lw $a0 __excp($a0)
[800001c8] 00240821  addu $1, $1, $4
[800001cc] 8c240180  lw $4, 384($1)
[800001d0] 00000000  nop                      ; 113: nop
[800001d4] 0000000c  syscall                  ; 114: syscall
[800001d8] 34010018  ori $1, $0, 24           ; 116: bne $k0 0x18 ok_pc # Bad PC exception requires special checks
[800001dc] 143a0008  bne $1, $26, 32 [ok_pc-0x800001dc]
[800001e0] 00000000  nop                      ; 117: nop
[800001e4] 40047000  mfc0 $4, $14             ; 119: mfc0 $a0 $14 # EPC
[800001e8] 30840003  andi $4, $4, 3           ; 120: andi $a0 $a0 0x3 # Is EPC word-aligned?
[800001ec] 10040004  beq $0, $4, 16 [ok_pc-0x800001ec]
[800001f0] 00000000  nop                      ; 122: nop
[800001f4] 3402000a  ori $2, $0, 10           ; 124: li $v0 10 # Exit on really bad PC
[800001f8] 0000000c  syscall                  ; 125: syscall
[800001fc] 34020004  ori $2, $0, 4            ; 128: li $v0 4 # syscall 4 (print_str)
[80000200] 3c019000  lui $1, -28672 [__m2_]   ; 129: la $a0 __m2_
[80000204] 3424000d  ori $4, $1, 13 [__m2_]
[80000208] 0000000c  syscall                  ; 130: syscall
[8000020c] 001a2082  srl $4, $26, 2           ; 132: srl $a0 $k0 2 # Extract ExcCode Field
[80000210] 3084001f  andi $4, $4, 31          ; 133: andi $a0 $a0 0x1f
[80000214] 14040002  bne $0, $4, 8 [ret-0x80000214]; 134: bne $a0 0 ret # 0 means exception was an interrupt
[80000218] 00000000  nop                      ; 135: nop
[8000021c] 401a7000  mfc0 $26, $14            ; 145: mfc0 $k0 $14 # Bump EPC register
[80000220] 275a0004  addiu $26, $26, 4        ; 146: addiu $k0 $k0 4 # Skip faulting instruction
[80000224] 409a7000  mtc0 $26, $14            ; 148: mtc0 $k0 $14
[80000228] 3c019000  lui $1, -28672           ; 153: lw $v0 s1 # Restore other registers
[8000022c] 8c220200  lw $2, 512($1)
[80000230] 3c019000  lui $1, -28672           ; 154: lw $a0 s2
[80000234] 8c240204  lw $4, 516($1)
[80000238] 001b0821  addu $1, $0, $27         ; 157: move $at $k1 # Restore $at
```

Figure 9.2: QTSPIM Floating-Point Registers and Condition Flags

MIPS Floating-Point Instructions

The FPU supports several instructions including floating-point load and store, floating-point arithmetic operations, floating-point data movement instructions, convert, and branch instructions. We start this section with the floating-point load and store instructions. These instructions load into or store a floating-point register. However, they use the same base-displacement addressing mode used with integer instructions. Notice that the base address register is an integer (not a floating- point) register.

Instruction	Example	Meaning
lwc1 or l.s	lwc1 $f1,0($sp)	Load a word from memory to a single-precision floating-point register: $f1 = MEM[$sp]
ldc1 or l.d	ldc1 $f2,8($t1)	Load a double word from memory to a double-precision register: $f2 = MEM[$t1+8]

Instruction	Example	Meaning
swc1 or s.s	swc1 $f5,4($t2)	Store a single-precision floating-point register in memory: MEM[$t2+4] = $f5
sdc1 or s.d	sdc1 $f6,16($t3)	Store a double-precision floating-point register in memory: MEM[$t3+16] = $f6

The floating-point arithmetic instructions are listed next. The **.s** extension is used for single-precision arithmetic instructions, while the **.d** is used for double-precision instructions.

Instruction	Example	Meaning
add.s	add.s $f0,$f2,$f4	$f0 = $f2 + $f4 (single-precision)
add.d	add.d $f0,$f2,$f4	$f0 = $f2 + $f4 (double-precision)
sub.s	sub.s $f0,$f2,$f4	$f0 = $f2 - $f4 (single-precision)
sub.d	sub.d $f0,$f2,$f4	$f0 = $f2 - $f4 (double-precision)
mul.s	mul.s $f0,$f2,$f4	$f0 = $f2 × $f4 (single-precision)
mul.d	mul.d $f0,$f2,$f4	$f0 = $f2 × $f4 (double-precision)
div.s	div.s $f0,$f2,$f4	$f0 = $f2 / $f4 (single-precision)
div.d	div.d $f0,$f2,$f4	$f0 = $f2 / $f4 (double-precision)
sqrt.s	sqrt.s $f0, $f2	Square root (single-precision)
sqrt.d	sqrt.d $f0, $f2	Square root (double-precision)
abs.s	abs.s $f0, $f2	Absolute value (single-precision)
abs.d	abs.d $f0, $f2	Absolute value (double-precision)
neg.s	neg.s $f0, $f2	Negative value (single-precision)
neg.d	neg.d $f0, $f2	Negative value (double-precision)

The data movement instructions move data between general-purpose and floating-point registers, or between floating-point registers.

Instruction	Example	Meaning
mfc1	mfc1 $t0, $f2	Move data from a floating-point register to a general-purpose register.
mtc1	mfc1 $t0, $f2	Move data from a general-purpose register to a floating-point register.
mov.s	mov.s $f0, $f1	Move single-precision data between two floating-point registers.
mov.d	mov.d$f0, $f2	Move double-precision data between two floating-point registers (move even-odd pair of registers).

The convert instructions convert the format of data in floating-point registers. Three data formats are supported: **.s** = single-precision float, **.d** = double-precision, and .w = integer word.

Instruction	Example	Meaning
cvt.s.w	cvt.s.w $f0,$f2	**$f0** = convert **$f2** from word to single-precision
cvt.s.d	cvt.s.d $f0,$f2	**$f0** = convert **$f2** from double to single-precision
cvt.d.w	cvt.d.w $f0,$f2	**$f0** = convert **$f2** from word to double-precision
cvt.d.s	cvt.d.s $f0,$f2	**$f0** = convert **$f2** from single to double-precision
cvt.w.s	cvt.w.s $f0,$f2	**$f0** = convert **$f2** from single-precision to word
cvt.w.d	cvt.w.d $f0,$f2	**$f0** = convert **$f2** from double-precision to word
ceil.w.s	ceil.w.s $f0,$f2	**$f0** = Integer ceiling of single-precision float in **$f2**
ceil.w.d	ceil.w.d $f0,$f2	**$f0** = Integer ceiling of double-precision float in **$f2**
floor.w.s	floor.w.s $f0,$f2	**$f0** = Integer floor of single-precision float in **$f2**
floor.w.d	floor.w.d $f0,$f2	**$f0** = Integer floor of double-precision float in **$f2**
trunc.w.s	trunc.w.s $f0,$f2	**$f0** = Truncate single-precision float in **$f2**
trunc.w.d	trunc.w.d $f0,$f2	**$f0** = Truncate double-precision float in **$f2**

The floating-point compare instructions compare floating-point registers for equality, less than, and less than or equal. The FP compare instructions set the condition flags **0** to **7** to true (1) or false(0).

Instruction	Example	Meaning
c.eq.s	c.eq.s $f2,$f3	if **($f2 == $f3)** set flag **0** to true else false
c.eq.d	c.eq.s 3,$f4,$f6	Compare equal double-precision. Result in flag **3**
c.lt.s	c.eq.s 4,$f5,$f8	if **($f5 < $f8)** set flag **4** to true else false
c.lt.d	c.lt.d 7,$f4,$f6	Compare less-than double. Result in flag **7**
c.le.s	c.le.s $f10,$f11	if **($f10 <= $f11)** set flag **0** to true else false
c.le.d	c.le.d $f14,$f16	Compare less or equal double. Result in flag **0**

The floating-point branch instructions (**bc1t** and **bc1f**) branch to the target address based on the value of the specified condition flag (true or false).

Instruction	Example	Meaning
bc1t	bc1t label	Branch to label if condition flag **0** is true
bc1t	bc1t 1, label	Branch to label if condition flag **1** is true
bc1f	bc1f label	Branch to label if condition flag **0** is false
bc1f	bc1f 4, label	Branch to label if condition flag **4** is false

System Call Services for Floating-Point Numbers

The MARS tool provides the following **Syscall** service numbers (passed in **$v0**) to print and read single-precision and double-precision floating-point numbers:

Service	$v0	Arguments	Result
Print Float	**2**	**$f12** = float to print	
Print Double	**3**	**$f12** = double to print	
Read Float	**6**		Float is returned in $f0
Read Double	**7**		Double is returned in **$f0**

MIPS Floating-Point Register Usage Convention

Compilers follow the MIPS register usage convention when translating functions and procedures into MIPS assembly-language code. The following table shows the MIPS software convention for floating-point registers. Not following the MIPS software usage convention can result in serious bugs when passing parameters, getting results, or using registers across function calls.

Registers	Usage
$f0 - $f3	Floating-point procedure results
$f4 - $f11	Temporary floating-point registers, NOT preserved across procedure calls
$f12 - $f15	Floating-point parameters, NOT preserved across procedure calls. Additional floating-point parameters should be pushed on the stack.
$f16 - $f19	More temporary registers, NOT preserved across procedure calls.
$f20 - $f31	Saved floating-point registers. Should be preserved across procedure calls.

Contents

Lab Tasks

Task 1

1. Convert by hand the number **-123456789** into its 32-bit single-precision binary representation, and Show your work for a full mark.

First. We start with the positive version of the number |-123456789| = 123456789

Second. We divide the number repeatedly by 2

division = quotient + remainder;

123 456 789 ÷ 2 = 61 728 394 + 1;

61 728 394 ÷ 2 = 30 864 197 + 0;

30 864 197 ÷ 2 = 15 432 098 + 1;

15 432 098 ÷ 2 = 7 716 049 + 0;

7 716 049 ÷ 2 = 3 858 024 + 1;

3 858 024 ÷ 2 = 1 929 012 + 0;

1 929 012 ÷ 2 = 964 506 + 0;

964 506 ÷ 2 = 482 253 + 0;

482 253 ÷ 2 = 241 126 + 1;

241 126 ÷ 2 = 120 563 + 0;

120 563 ÷ 2 = 60 281 + 1;

60 281 ÷ 2 = 30 140 + 1;

30 140 ÷ 2 = 15 070 + 0;

15 070 ÷ 2 = 7 535 + 0;
7 535 ÷ 2 = 3 767 + 1;
3 767 ÷ 2 = 1 883 + 1;
1 883 ÷ 2 = 941 + 1;
941 ÷ 2 = 470 + 1;
470 ÷ 2 = 235 + 0;
235 ÷ 2 = 117 + 1;
117 ÷ 2 = 58 + 1;
58 ÷ 2 = 29 + 0;
29 ÷ 2 = 14 + 1;
14 ÷ 2 = 7 + 0;
7 ÷ 2 = 3 + 1;
3 ÷ 2 = 1 + 1;
1 ÷ 2 = 0 + 1;

Then we Construct the base 2 representation of the positive number.
123 456 789 (decimal) = 111 0101 1011 1100 1101 0001 0101 (binary)

After that. We Normalize the binary representation of the number.
111 0101 1011 1100 1101 0001 0101 (binary) = 1.1101 0110 1111 0011 0100 0101 01 (binary) × 2^{26}
Now. We adjust the exponent. Exponent (unadjusted) = 26
Exponent (adjusted) = Exponent (unadjusted) + 2(8-1) - 1 = 26 + 2(8-1) - 1 = (26 + 127)(10)
= 153(10)

Then we use the same technique by dividing the number repeatedly by 2
division = quotient + remainder;
153 ÷ 2 = 76 + 1;
76 ÷ 2 = 38 + 0;
38 ÷ 2 = 19 + 0;
19 ÷ 2 = 9 + 1;
9 ÷ 2 = 4 + 1;
4 ÷ 2 = 2 + 0;
2 ÷ 2 = 1 + 0;
1 ÷ 2 = 0 + 1;
Exponent (adjusted) = 153 (decimal) = 1001 1001 (binary)

After that. We normalized the mantissa by:

A- Remove the leading bit
B- Adjust its length to 23 by removing the excess bits

1.1101 0110 1111 0011 0100 0101 01 (binary) = 110 1011 0111 1001 1010 0010

Finally. The three elements that make up 32 bit single precision IEE 754 binary floating point are
Sign (1 bit) = 1

Exponent (8 bits) = 1001 1001 (binary)
Mantissa (23 bits) = 110 1011 0111 1001 1010 0010
= 1 10011001 11010110111100110100010

Task 2

2. Convert by hand the floating-point number **1 10010100 10011000001100000000000** (shown in binary) into its corresponding decimal value. Show your work for a full mark.

Sign bit: 1 (negative)

Exponent: 10010100 (binary) = 148 (decimal)

Mantissa: 10011000001100000000000 (binary)

First. we subtract 127 from the exponent to get the true exponent: 148 – 127 = 21

Second. we convert the mantissa to decimal, by multiplying by $2^{\text{N of number}}$:

$1 \times 2^{-1} + 0 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} + 1 \times 2^{-5} + 0 \times 2^{-6} + 0 \times 2^{-7} + 0 \times 2^{-8} + 0 \times 2^{-9} + 0 \times 2^{-10} + 1 \times 2^{-11} + 1 \times 2^{-12} + 0 \times 2^{-13} + 0 \times 2^{-14} + 0 \times 2^{-15} + 0 \times 2^{-16} + 0 \times 2^{-17} + 0 \times 2^{-18} + 0 \times 2^{-19} + 0 \times 2^{-20} + 0 \times 2^{-21} + 0 \times 2^{-22} + 0 \times 2^{-23} = 0.594482421875$ (decimal)

Then we will add 1 to the mantissa: 0.594482421875 (decimal) + 1 = 1.594482421875 (decimal)

Finally, the floating-point value is then equal to $-1 \times 2^{21} \times 1.594482421875 = -3343872$

Task 3

3. Trace the following program by hand to determine the values of registers **$f0** thru **$f9**. Notice that **array1** and **array2** have the same elements, but in a different order. Comment on the sums of **array1** and **array2** elements computed in registers **$f4** and **$f9**, respectively. Now use the QTSPIM tool to trace the execution of the program and verify your results. What conclusion can be made from this exercise?

```
.data
array1:  .float  5.6e+20,  -5.6e+20,  1.2
array2:  .float  1.2,  5.6e+20,  -5.6e+20
.text
.globl main

main:
```

```
la      $t0,  array1
lwc1    $f0,  0($t0)
lwc1    $f1,  4($t0)
lwc1    $f2,  8($t0)
add.s   $f3,  $f0,  $f1
add.s   $f4,  $f2,  $f3
la      $t1,  array2
lwc1    $f5,  0($t1)
lwc1    $f6,  4($t1)
lwc1    $f7,  8($t1)
add.s   $f8,  $f5,  $f6
add.s   $f9,  $f7,  $f8

li $v0, 10          # To terminate the program
syscall

.end main
```

$f0 = 5.6e+20

$f1 = -5.6e+20

$f2 = 1.2

$f3 = $f0 + $f1 = 0

$f4 = $f2 + $f3 = 1.2

$f5 = 1.2

$f6 = 5.6e+20

$f7 = -5.6e+20

$f8 = $f5 + $f6 = 5.6e+20

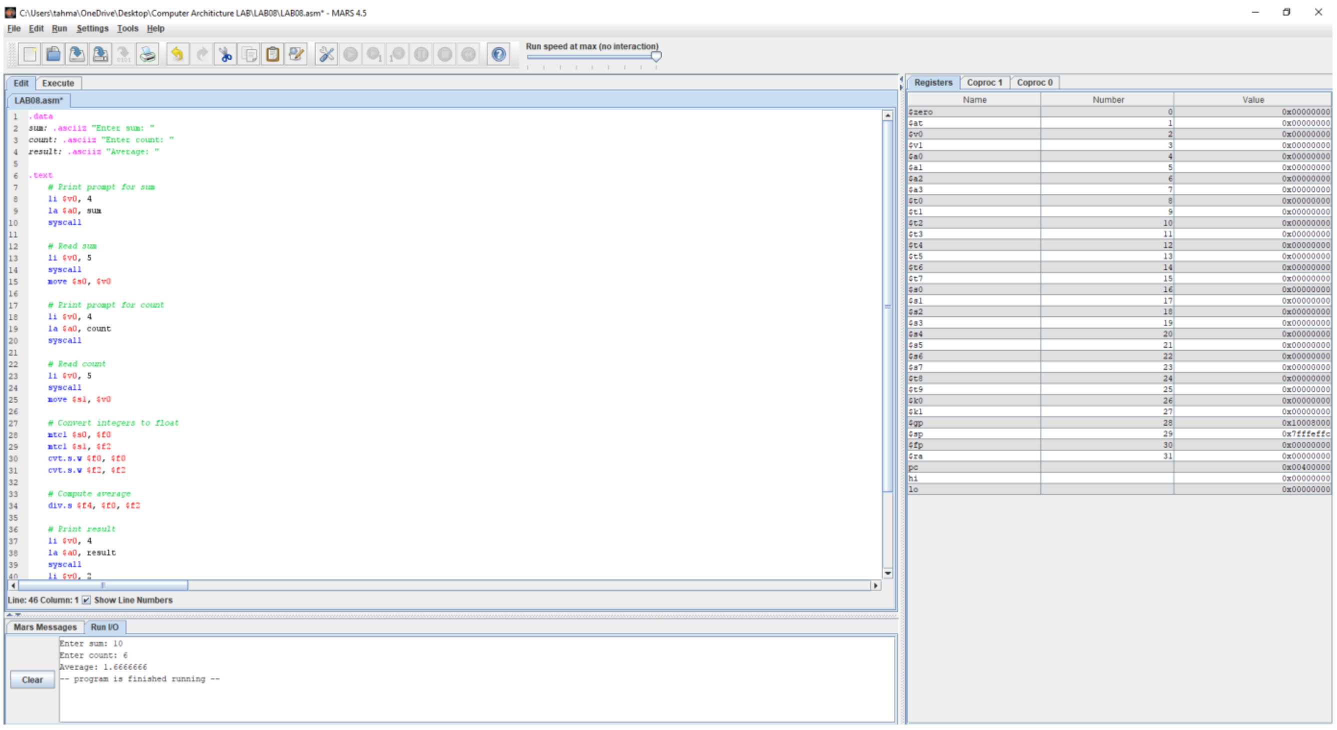
$f9 = $f7 + $f8 = 0

The sum of elements of "array1" is 1.2 and the sum of elements of "array2" is 0. This shows that the order of elements in a floating-point addition operation can affect the result due to the limited precision of floating-point numbers.

Task 4

4. Write an interactive program that inputs an integer **sum** and an integer **count**, computes, and displays the **average = (float) sum / (float) count** as a single-precision floating- point number. Hint: use the proper convert instruction to convert **sum** and **count** from integer word into single-precision float.

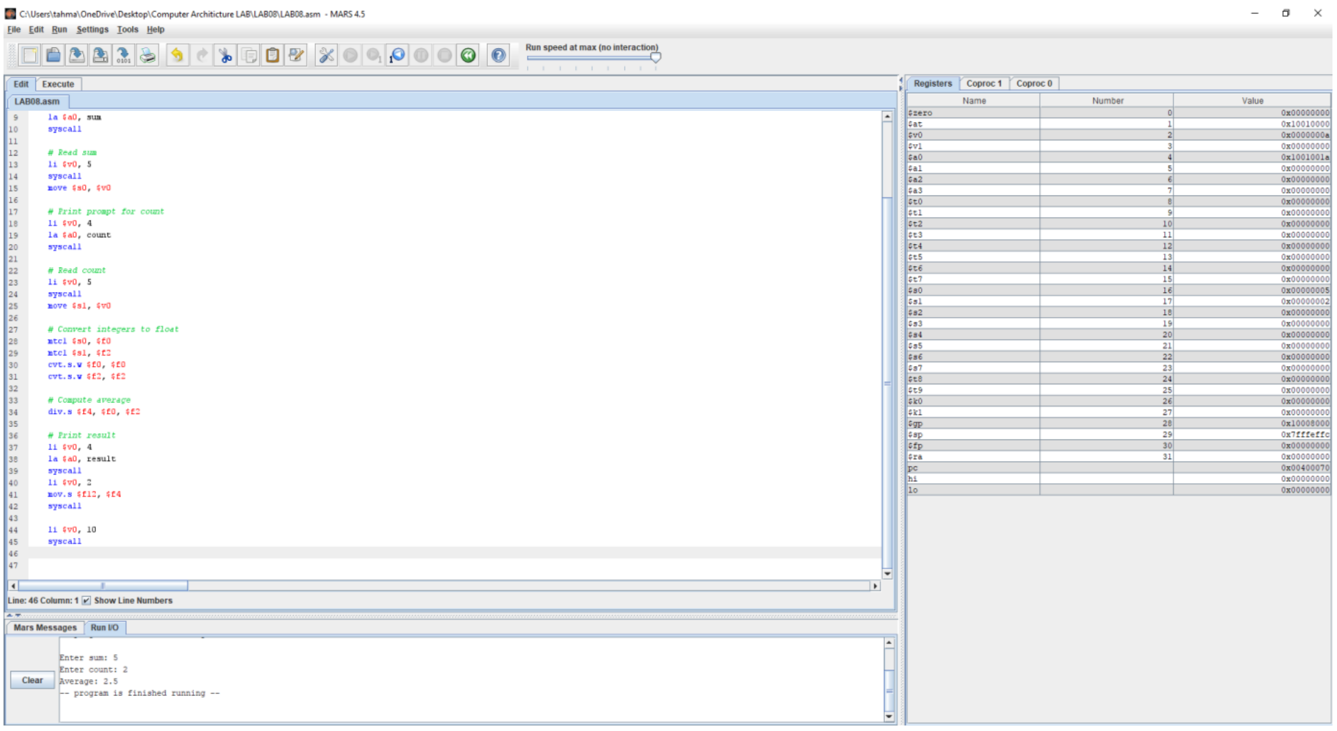
PICTURES OF THE CODE:


C:\Users\tahma\OneDrive\Desktop\Computer Architicture LAB\LAB08\LAB08.asm* - MARS 4.5
File Edit Run Settings Tools Help
Run speed at max (no interaction)
Edit Execute
LAB08.asm*
Line: 46 Column: 1 Show Line Numbers
Mars Messages Run I/O
Clear
Registers Coproc 1 Coproc 0

```
.data
sum: .asciiz "Enter sum: "
count: .asciiz "Enter count: "
result: .asciiz "Average: "

.text
    # Print prompt for sum
    li $v0, 4
    la $a0, sum
    syscall

    # Read sum
    li $v0, 5
    syscall
    move $s0, $v0

    # Print prompt for count
    li $v0, 4
    la $a0, count
    syscall

    # Read count
    li $v0, 5
    syscall
    move $s1, $v0

    # Convert integers to float
    mtc1 $s0, $f0
    mtc1 $s1, $f2
    cvt.s.w $f0, $f0
    cvt.s.w $f2, $f2

    # Compute average
    div.s $f4, $f0, $f2

    # Print result
    li $v0, 4
    la $a0, result
    syscall
    li $v0, 2
```

```
Enter sum: 10
Enter count: 6
Average: 1.6666666
-- program is finished running --
```

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000


C:\Users\tahma\OneDrive\Desktop\Computer Architicture LAB\LAB08\LAB08.asm - MARS 4.5
File Edit Run Settings Tools Help
Run speed at max (no interaction)
Edit Execute
LAB08.asm
Line: 46 Column: 1 Show Line Numbers
Mars Messages Run I/O
Clear
Registers Coproc 1 Coproc 0

```
    la $a0, sum
    syscall

    # Read sum
    li $v0, 5
    syscall
    move $s0, $v0

    # Print prompt for count
    li $v0, 4
    la $a0, count
    syscall

    # Read count
    li $v0, 5
    syscall
    move $s1, $v0

    # Convert integers to float
    mtc1 $s0, $f0
    mtc1 $s1, $f2
    cvt.s.w $f0, $f0
    cvt.s.w $f2, $f2

    # Compute average
    div.s $f4, $f0, $f2

    # Print result
    li $v0, 4
    la $a0, result
    syscall
    li $v0, 2
    mov.s $f12, $f4
    syscall

    li $v0, 10
    syscall
```

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x1001001a
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000005
$s1	17	0x00000002
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400070
hi		0x00000000
lo		0x00000000

```
Enter sum: 5
Enter count: 2
Average: 2.5
-- program is finished running --
```

CODE:

```
.data
sum: .asciiz "Enter sum: "
count: .asciiz "Enter count: "
result: .asciiz "Average: "

.text
    # Print prompt for sum
    li $v0, 4
    la $a0, sum
    syscall

    # Read sum
    li $v0, 5
    syscall
    move $s0, $v0

    # Print prompt for count
    li $v0, 4
    la $a0, count
    syscall

    # Read count
    li $v0, 5
    syscall
    move $s1, $v0
```

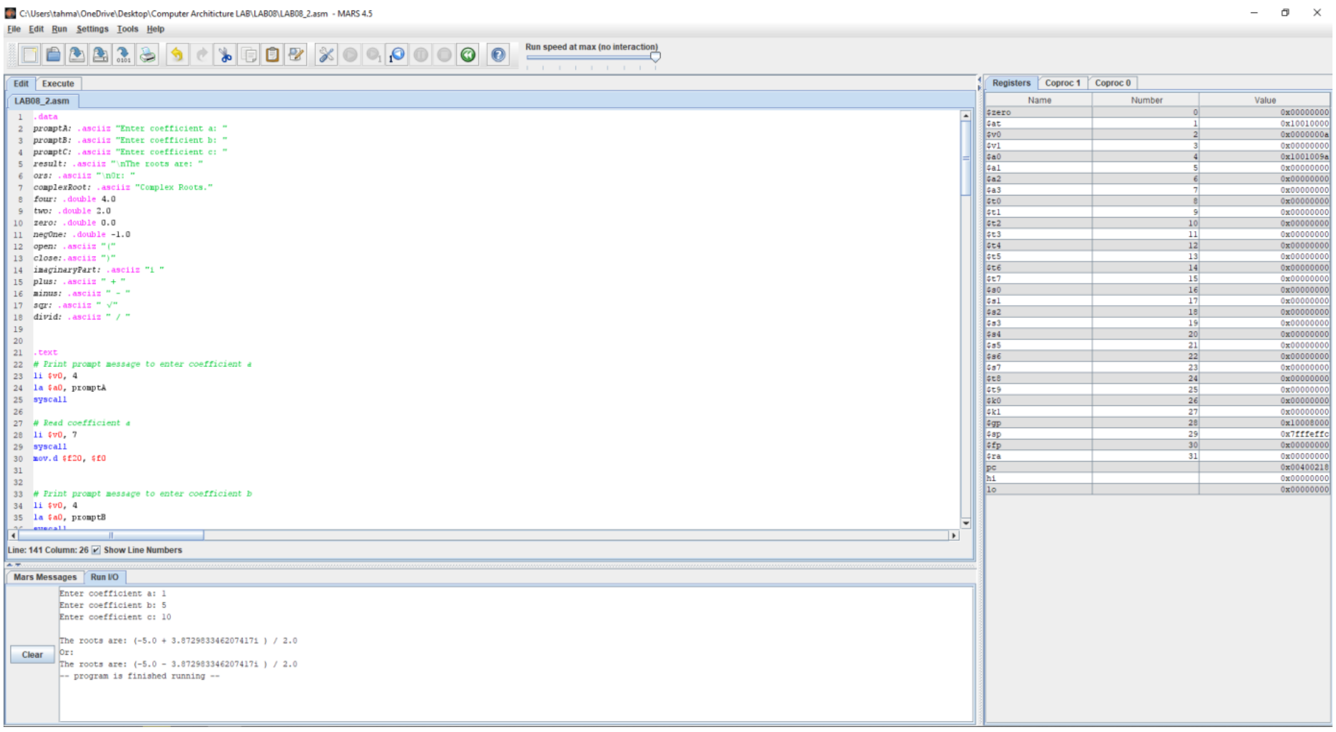
```
# Convert integers to float
mtc1 $s0, $f0
mtc1 $s1, $f2
cvt.s.w $f0, $f0
cvt.s.w $f2, $f2

# Compute average
div.s $f4, $f0, $f2

# Print result
li $v0, 4
la $a0, result
syscall
li $v0, 2
mov.s $f12, $f4
syscall

li $v0, 10
syscall
```

Task 5

5. Write an interactive program that inputs the coefficient of a quadratic equation, computes, and displays the roots of the quadratic equation. All input, computation, and output should be done using double-precision floating-point instructions and registers. The program should handle the case of complex roots and displays the results properly.

$$x = -b \pm \frac{\sqrt{b^2 - 4ac}}{2a}$$

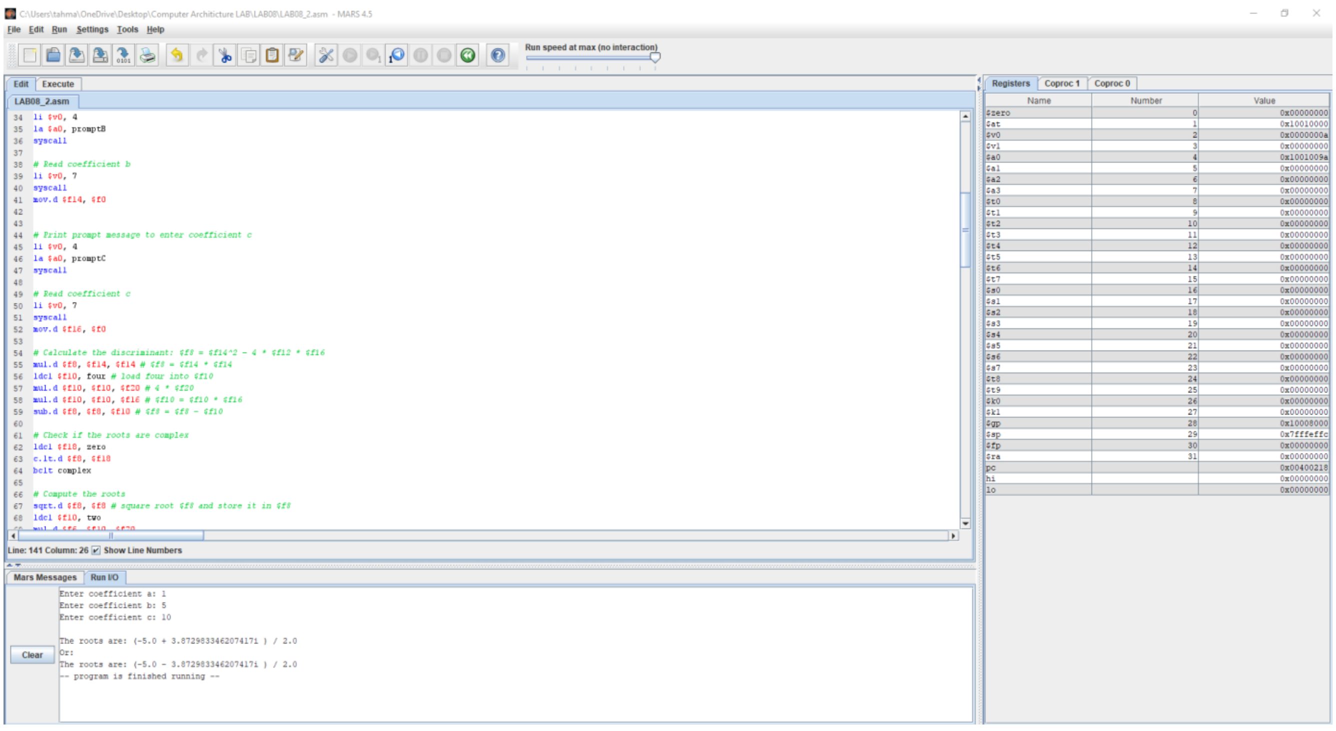
PICTURES OF THE CODE:



```
.data
promptA: .asciiz "Enter coefficient a: "
promptB: .asciiz "Enter coefficient b: "
promptC: .asciiz "Enter coefficient c: "
result: .asciiz "\nThe roots are: "
ors: .asciiz "\nOr: "
complexRoot: .asciiz "Complex Roots."
four: .double 4.0
two: .double 2.0
zero: .double 0.0
negOne: .double -1.0
open: .asciiz "("
close:.asciiz ")"
imaginaryPart: .asciiz "i "
plus: .asciiz " + "
minus: .asciiz " - "
sqr: .asciiz " √"
divid: .asciiz " / "


.text
# Print prompt message to enter coefficient a
li $v0, 4
la $a0, promptA
syscall

# Read coefficient a
li $v0, 7
syscall
mov.d $f20, $f0


# Print prompt message to enter coefficient b
li $v0, 4
la $a0, promptB
```

```
Enter coefficient a: 1
Enter coefficient b: 5
Enter coefficient c: 10

The roots are: (-5.0 + 3.8729833462074171 ) / 2.0
Or:
The roots are: (-5.0 - 3.8729833462074171 ) / 2.0
-- program is finished running --
```

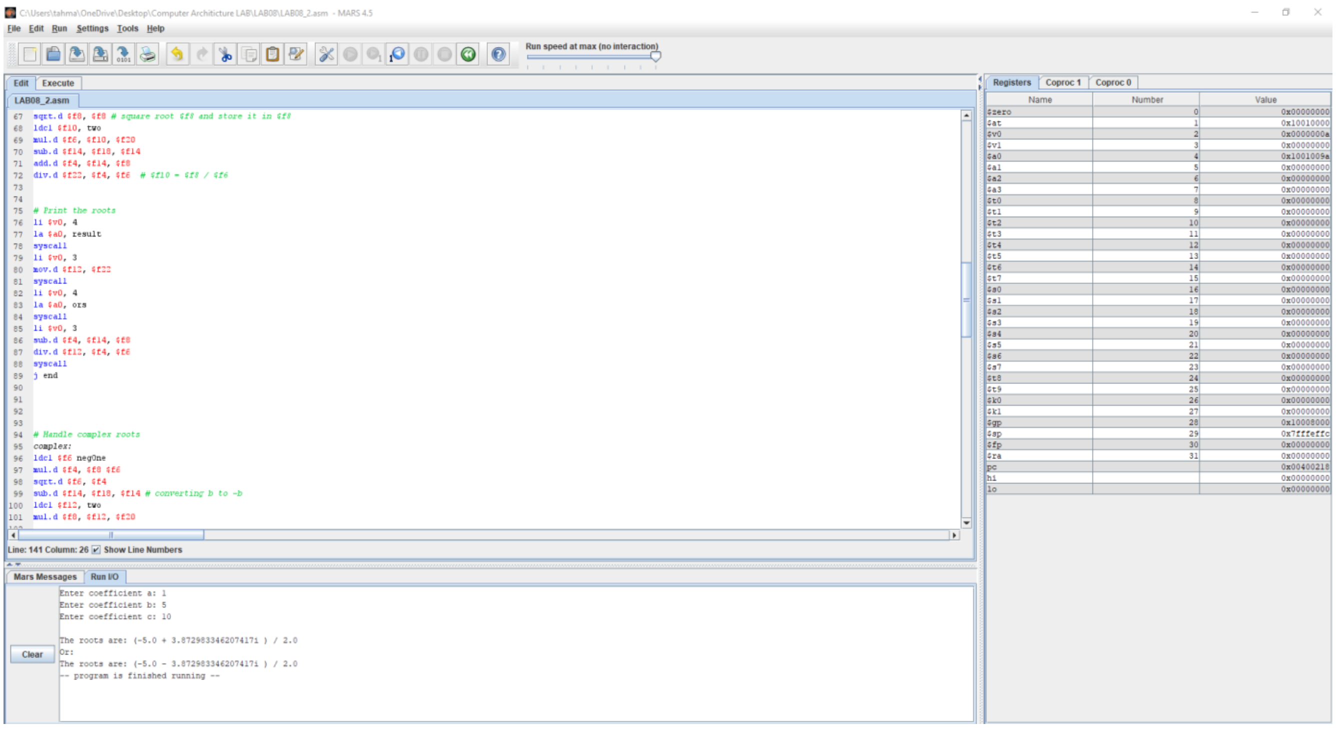
Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x1001009a
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400218
hi		0x00000000
lo		0x00000000



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

# Read coefficient b
li $v0, 7
syscall
mov.d $f14, $f0


# Print prompt message to enter coefficient c
li $v0, 4
la $a0, promptC
syscall

# Read coefficient c
li $v0, 7
syscall
mov.d $f16, $f0

# Calculate the discriminant: $f8 = $f14^2 - 4 * $f12 * $f16
mul.d $f8, $f14, $f14 # $f8 = $f14 * $f14
ldc1 $f10, four # load four into $f10
mul.d $f10, $f10, $f20 # 4 * $f20
mul.d $f10, $f10, $f16 # $f10 = $f10 * $f16
sub.d $f8, $f8, $f10 # $f8 = $f8 - $f10

# Check if the roots are complex
ldc1 $f18, zero
c.lt.d $f8, $f18
bc1t complex

# Compute the roots
sqrt.d $f8, $f8 # square root $f8 and store it in $f8
ldc1 $f10, two
```

```
Enter coefficient a: 1
Enter coefficient b: 5
Enter coefficient c: 10

The roots are: (-5.0 + 3.8729833462074171 ) / 2.0
Or:
The roots are: (-5.0 - 3.8729833462074171 ) / 2.0
-- program is finished running --
```

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x1001009a
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400218
hi		0x00000000
lo		0x00000000



C:\Users\tahma\OneDrive\Desktop\Computer Architicture LAB\LAB08\LAB08_2.asm - MARS 4.5

Run speed at max (no interaction)

LAB08_2.asm

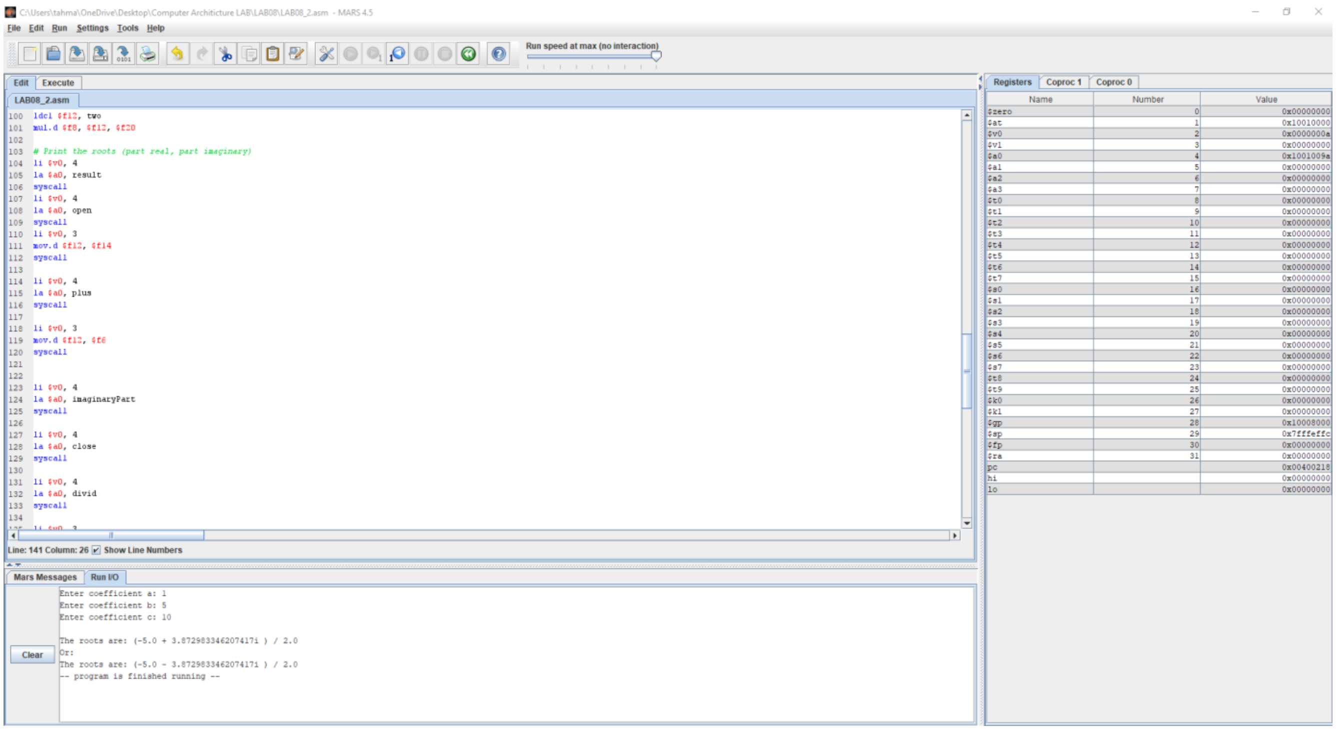
```
sqrt.d $f8, $f8 # square root $f8 and store it in $f8
ldc1 $f10, two
mul.d $f6, $f10, $f20
sub.d $f14, $f18, $f14
add.d $f4, $f14, $f8
div.d $f22, $f4, $f6  # $f10 = $f8 / $f6


# Print the roots
li $v0, 4
la $a0, result
syscall
li $v0, 3
mov.d $f12, $f22
syscall
li $v0, 4
la $a0, ors
syscall
li $v0, 3
sub.d $f4, $f14, $f8
div.d $f12, $f4, $f6
syscall
j end




# Handle complex roots
complex:
ldc1 $f6 negOne
mul.d $f4, $f8 $f6
sqrt.d $f6, $f4
sub.d $f14, $f18, $f14 # converting b to -b
ldc1 $f12, two
mul.d $f8, $f12, $f20
```

Line: 141 Column: 26 Show Line Numbers

Run I/O

```
Enter coefficient a: 1
Enter coefficient b: 5
Enter coefficient c: 10

The roots are: (-5.0 + 3.872983346207417i ) / 2.0
Or:
The roots are: (-5.0 - 3.872983346207417i ) / 2.0
-- program is finished running --
```

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x1001009a
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400218
hi		0x00000000
lo		0x00000000



C:\Users\tahma\OneDrive\Desktop\Computer Architicture LAB\LAB08\LAB08_2.asm - MARS 4.5

Run speed at max (no interaction)

LAB08_2.asm

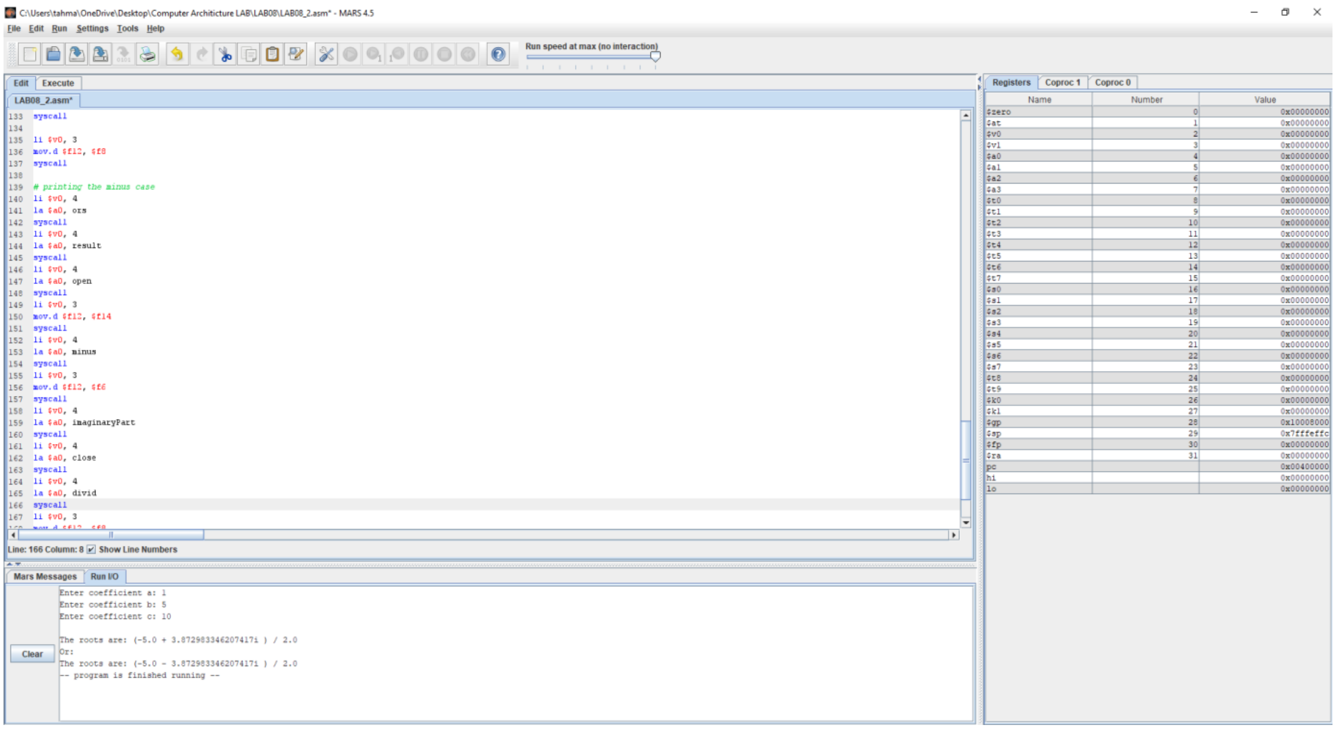
```
ldc1 $f12, two
mul.d $f8, $f12, $f20

# Print the roots (part real, part imaginary)
li $v0, 4
la $a0, result
syscall
li $v0, 4
la $a0, open
syscall
li $v0, 3
mov.d $f12, $f14
syscall

li $v0, 4
la $a0, plus
syscall

li $v0, 3
mov.d $f12, $f6
syscall


li $v0, 4
la $a0, imaginaryPart
syscall

li $v0, 4
la $a0, close
syscall

li $v0, 4
la $a0, divid
syscall

```

Line: 141 Column: 26 Show Line Numbers

Run I/O

```
Enter coefficient a: 1
Enter coefficient b: 5
Enter coefficient c: 10

The roots are: (-5.0 + 3.872983346207417i ) / 2.0
Or:
The roots are: (-5.0 - 3.872983346207417i ) / 2.0
-- program is finished running --
```

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x1001009a
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400218
hi		0x00000000
lo		0x00000000


C:\Users\tahma\OneDrive\Desktop\Computer Architicture LAB\LAB08\LAB08_2.asm* - MARS 4.5

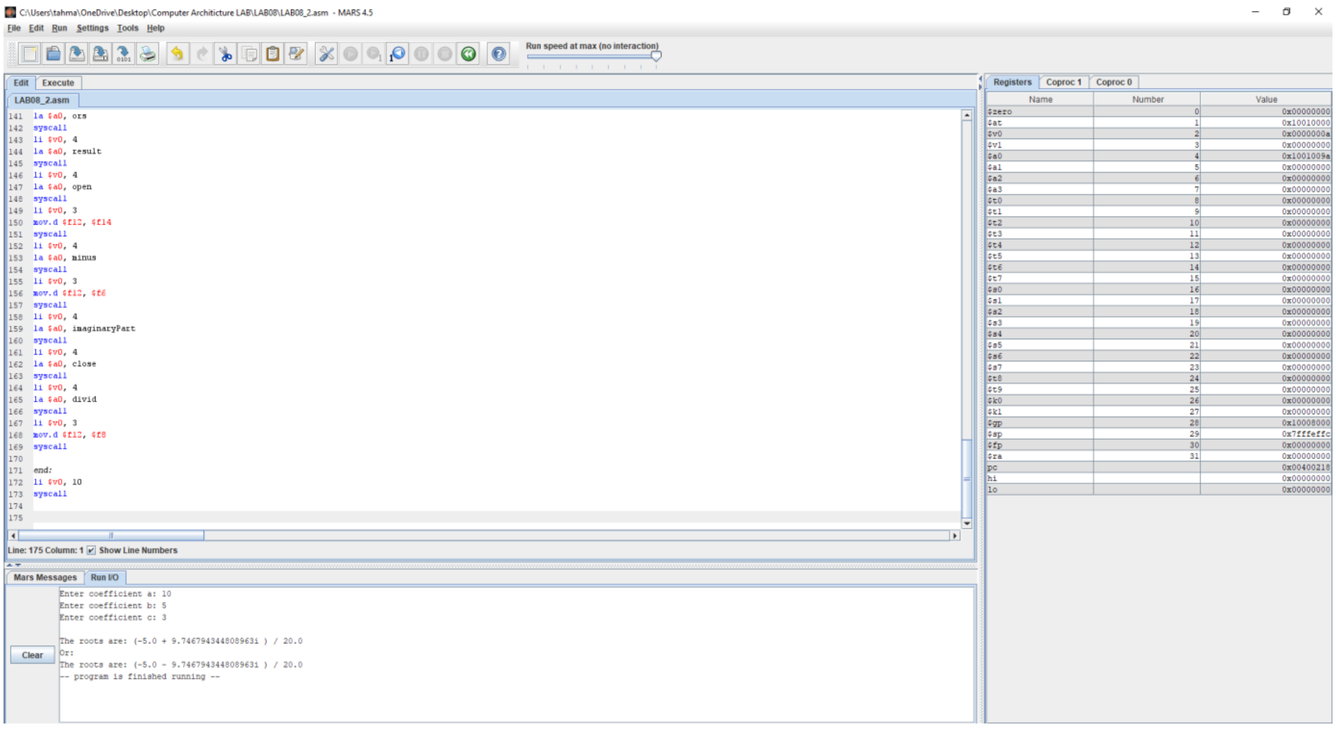
LAB08_2.asm*

```
syscall

li $v0, 3
mov.d $f12, $f8
syscall

# printing the minus case
li $v0, 4
la $a0, ors
syscall
li $v0, 4
la $a0, result
syscall
li $v0, 4
la $a0, open
syscall
li $v0, 3
mov.d $f12, $f14
syscall
li $v0, 4
la $a0, minus
syscall
li $v0, 3
mov.d $f12, $f6
syscall
li $v0, 4
la $a0, imaginaryPart
syscall
li $v0, 4
la $a0, close
syscall
li $v0, 4
la $a0, divid
syscall
li $v0, 3
```

Line: 166 Column: 8 Show Line Numbers

```
Enter coefficient a: 1
Enter coefficient b: 5
Enter coefficient c: 10

The roots are: (-5.0 + 3.872983346207417i ) / 2.0
Or:
The roots are: (-5.0 - 3.872983346207417i ) / 2.0
-- program is finished running --
```

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000


C:\Users\tahma\OneDrive\Desktop\Computer Architicture LAB\LAB08\LAB08_2.asm - MARS 4.5

LAB08_2.asm

```
la $a0, ors
syscall
li $v0, 4
la $a0, result
syscall
li $v0, 4
la $a0, open
syscall
li $v0, 3
mov.d $f12, $f14
syscall
li $v0, 4
la $a0, minus
syscall
li $v0, 3
mov.d $f12, $f6
syscall
li $v0, 4
la $a0, imaginaryPart
syscall
li $v0, 4
la $a0, close
syscall
li $v0, 4
la $a0, divid
syscall
li $v0, 3
mov.d $f12, $f8
syscall

end:
li $v0, 10
syscall


```

Line: 175 Column: 1 Show Line Numbers

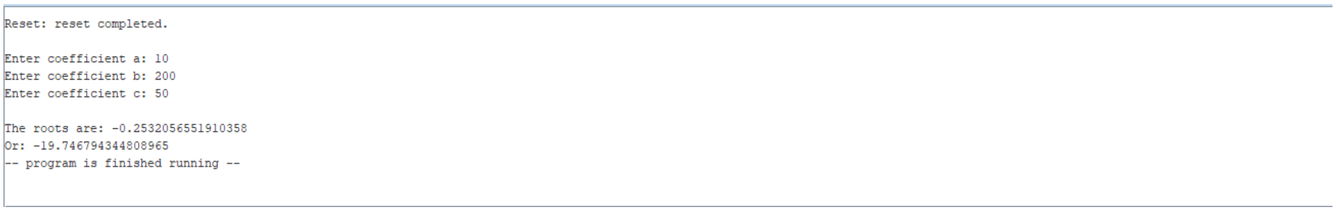
```
Enter coefficient a: 10
Enter coefficient b: 5
Enter coefficient c: 3

The roots are: (-5.0 + 9.746794344808963i ) / 20.0
Or:
The roots are: (-5.0 - 9.746794344808963i ) / 20.0
-- program is finished running --
```

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x1001009a
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400218
hi		0x00000000
lo		0x00000000

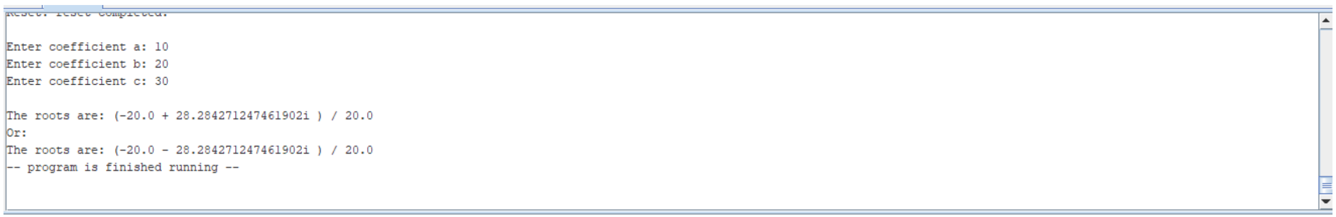
OUTPUTS

```
Reset: reset completed.

Enter coefficient a: 25
Enter coefficient b: -30
Enter coefficient c: 9

The roots are: 0.6
Or: 0.6
-- program is finished running --
```

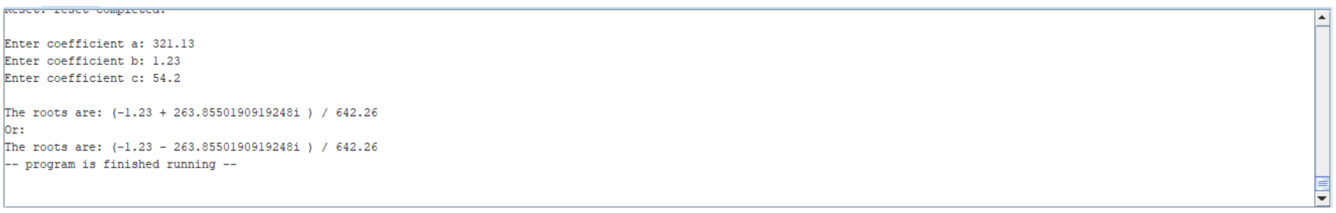
```
Reset: reset completed.

Enter coefficient a: 1
Enter coefficient b: 5
Enter coefficient c: 1

The roots are: -0.20871215252208009
Or: -4.7912878474779195
-- program is finished running --
```



```
Reset: reset completed.

Enter coefficient a: 10
Enter coefficient b: 200
Enter coefficient c: 50

The roots are: -0.2532056551910358
Or: -19.746794344808965
-- program is finished running --
```



```
Enter coefficient a: 10
Enter coefficient b: 20
Enter coefficient c: 30

The roots are: (-20.0 + 28.284271247461902i ) / 20.0
Or:
The roots are: (-20.0 - 28.284271247461902i ) / 20.0
-- program is finished running --
```



```
Enter coefficient a: 321.13
Enter coefficient b: 1.23
Enter coefficient c: 54.2

The roots are: (-1.23 + 263.8550190919248i ) / 642.26
Or:
The roots are: (-1.23 - 263.8550190919248i ) / 642.26
-- program is finished running --
```

CODE:

```
.data
promptA: .asciiz "Enter coefficient a: "
promptB: .asciiz "Enter coefficient b: "
promptC: .asciiz "Enter coefficient c: "
result: .asciiz "\nThe roots are: "
ors: .asciiz "\nOr: "
complexRoot: .asciiz "Complex Roots."
four: .double 4.0
two: .double 2.0
zero: .double 0.0
negOne: .double -1.0
open: .asciiz "("
close:.asciiz ")"
imaginaryPart: .asciiz "i "
plus: .asciiz " + "
minus: .asciiz " - "
sqr: .asciiz " √"
divid: .asciiz " / "


.text
# Print prompt message to enter coefficient a
li $v0, 4
la $a0, promptA
syscall

# Read coefficient a
li $v0, 7
syscall
mov.d $f20, $f0


# Print prompt message to enter coefficient b
li $v0, 4
la $a0, promptB
syscall
```

```
# Read coefficient b
li $v0, 7
syscall
mov.d $f14, $f0


# Print prompt message to enter coefficient c
li $v0, 4
la $a0, promptC
syscall

# Read coefficient c
li $v0, 7
syscall
mov.d $f16, $f0

# Calculate the discriminant: $f8 = $f14^2 - 4 * $f12 * $f16
mul.d $f8, $f14, $f14 # $f8 = $f14 * $f14
ldc1 $f10, four # load four into $f10
mul.d $f10, $f10, $f20 # 4 * $f20
mul.d $f10, $f10, $f16 # $f10 = $f10 * $f16
sub.d $f8, $f8, $f10 # $f8 = $f8 - $f10

# Check if the roots are complex
ldc1 $f18, zero
c.lt.d $f8, $f18
bc1t complex

# Compute the roots
sqrt.d $f8, $f8 # square root $f8 and store it in $f8
ldc1 $f10, two
mul.d $f6, $f10, $f20
sub.d $f14, $f18, $f14
add.d $f4, $f14, $f8
div.d $f22, $f4, $f6  # $f10 = $f8 / $f6


# Print the roots
li $v0, 4
la $a0, result
syscall
li $v0, 3
mov.d $f12, $f22
syscall
li $v0, 4
la $a0, ors
syscall
li $v0, 3
sub.d $f4, $f14, $f8
div.d $f12, $f4, $f6
syscall
j end




# Handle complex roots
```

```
complex:
ldc1 $f6 negOne
mul.d $f4, $f8 $f6
sqrt.d $f6, $f4
sub.d $f14, $f18, $f14 # converting b to -b
ldc1 $f12, two
mul.d $f8, $f12, $f20

# Print the roots (part real, part imaginary)
li $v0, 4
la $a0, result
syscall
li $v0, 4
la $a0, open
syscall
li $v0, 3
mov.d $f12, $f14
syscall

li $v0, 4
la $a0, plus
syscall

li $v0, 3
mov.d $f12, $f6
syscall


li $v0, 4
la $a0, imaginaryPart
syscall

li $v0, 4
la $a0, close
syscall

li $v0, 4
la $a0, divid
syscall

li $v0, 3
mov.d $f12, $f8
syscall

# printing the minus case
li $v0, 4
la $a0, ors
syscall
li $v0, 4
la $a0, result
syscall
li $v0, 4
la $a0, open
syscall
li $v0, 3
mov.d $f12, $f14
syscall
li $v0, 4
```

```
la $a0, minus
syscall
li $v0, 3
mov.d $f12, $f6
syscall
li $v0, 4
la $a0, imaginaryPart
syscall
li $v0, 4
la $a0, close
syscall
li $v0, 4
la $a0, divid
syscall
li $v0, 3
mov.d $f12, $f8
syscall

end:
li $v0, 10
syscall
```

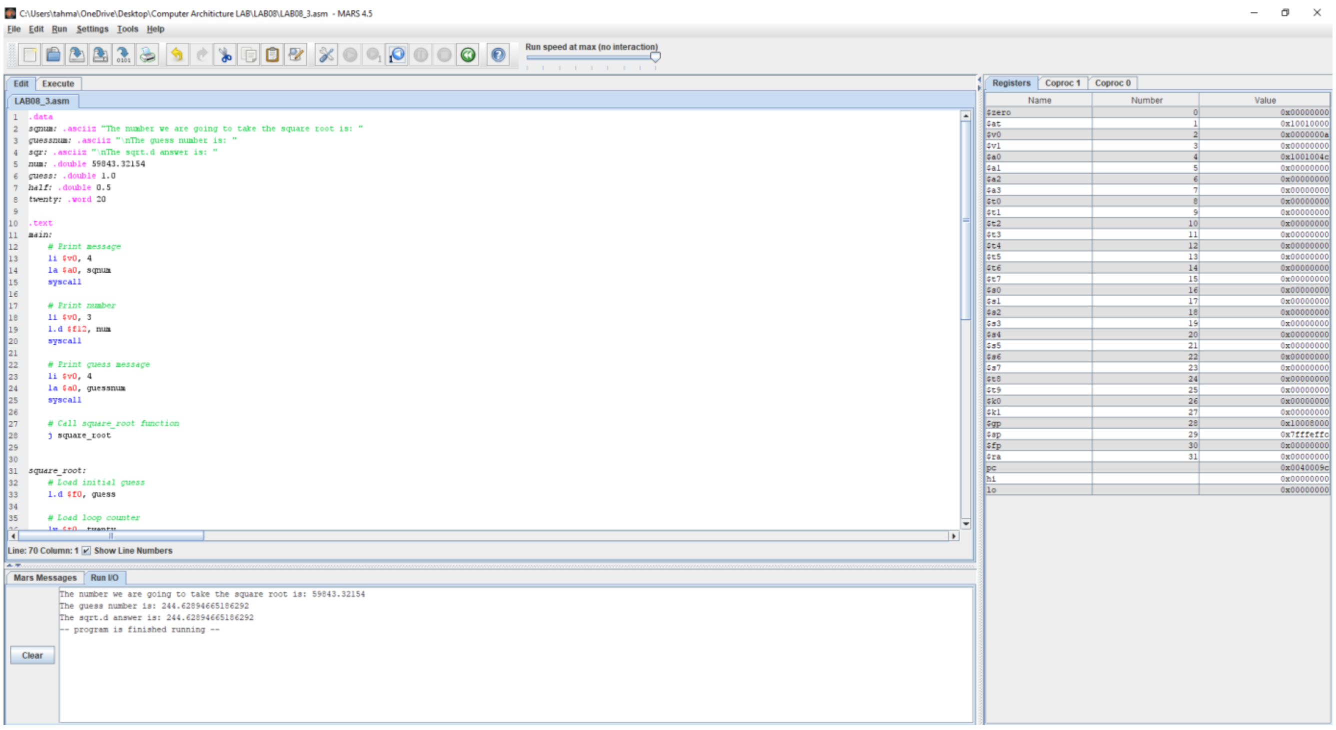
Task 6

6. Square Root Calculation: Newton's iterative method can be used to approximate the square root of a number **x**. Let the initial **guess** be **1**. Then each new **guess** can be computed as follows:

   guess = ((x/guess) + guess) / 2;

   Write a function called **square_root** that receives a double-precision parameter **x**, computes, and returns the approximated value of the square root of **x**. Write a loop that repeats 20 times and computes 20 **guess** values, then returns the final **guess** after 20 iterations. Use the MIPS floating-point register convention to pass the parameter **x** and to return the function result. All computation should be done using double-precision floating-point instructions and registers. Compare the result of the **sqrt.d** instruction against the result of your **square_root** function. What is the error in absolute value?

PICTURES OF THE CODE:



C:\Users\tahma\OneDrive\Desktop\Computer Architicture LAB\LAB08\LAB08_3.asm - MARS 4.5

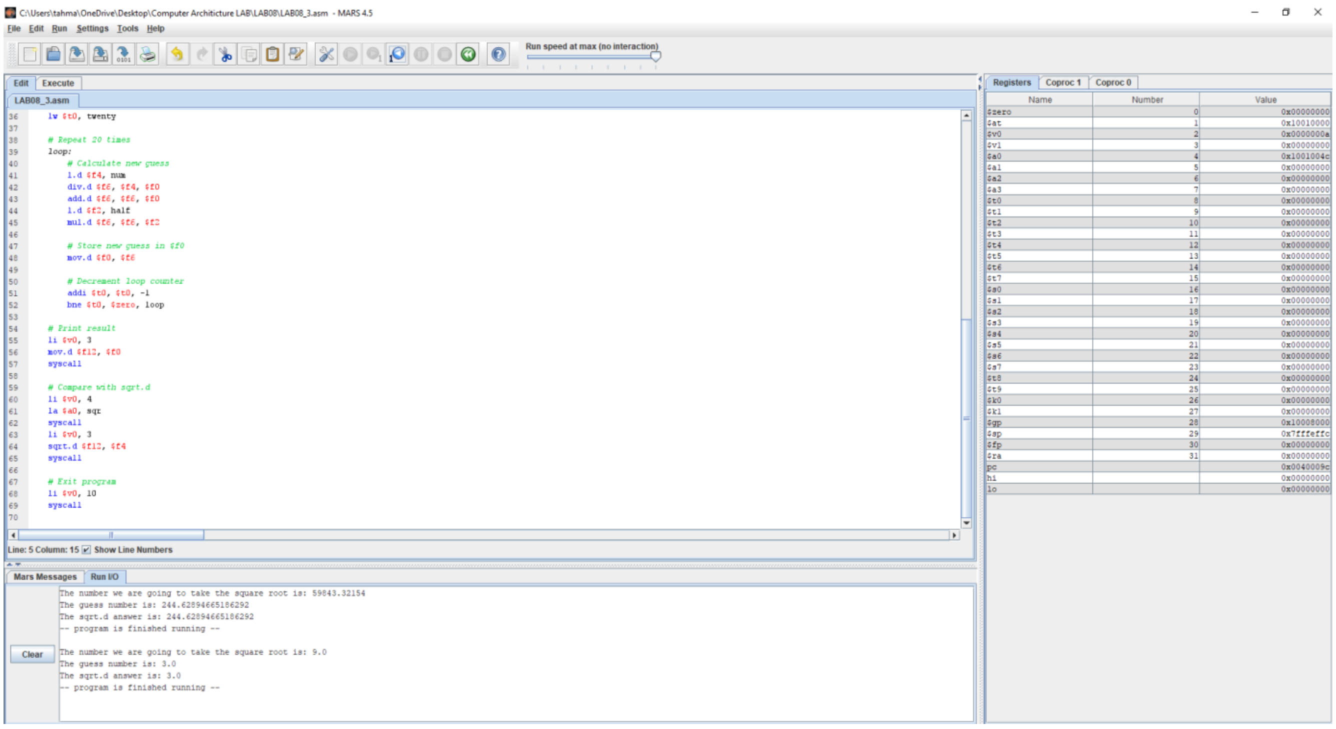
File Edit Run Settings Tools Help
Run speed at max (no interaction)
Edit Execute
LAB08_3.asm

```
.data
sqnum: .asciiz "The number we are going to take the square root is: "
guessnum: .asciiz "\nThe guess number is: "
sqr: .asciiz "\nThe sqrt.d answer is: "
num: .double 59843.32154
guess: .double 1.0
half: .double 0.5
twenty: .word 20

.text
main:
    # Print message
    li $v0, 4
    la $a0, sqnum
    syscall

    # Print number
    li $v0, 3
    l.d $f12, num
    syscall

    # Print guess message
    li $v0, 4
    la $a0, guessnum
    syscall

    # Call square_root function
    j square_root


square_root:
    # Load initial guess
    l.d $f0, guess

    # Load loop counter
```

Line: 70 Column: 1 ☑ Show Line Numbers

```
The number we are going to take the square root is: 59843.32154
The guess number is: 244.62894665186292
The sqrt.d answer is: 244.62894665186292
-- program is finished running --
```

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x1001004c
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x0040009c
hi		0x00000000
lo		0x00000000



C:\Users\tahma\OneDrive\Desktop\Computer Architicture LAB\LAB08\LAB08_3.asm - MARS 4.5

File Edit Run Settings Tools Help
Run speed at max (no interaction)
Edit Execute
LAB08_3.asm

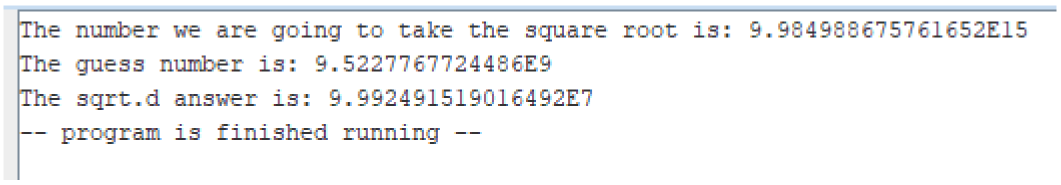
```
    lw $t0, twenty

    # Repeat 20 times
    loop:
        # Calculate new guess
        l.d $f4, num
        div.d $f6, $f4, $f0
        add.d $f6, $f6, $f0
        l.d $f2, half
        mul.d $f6, $f6, $f2

        # Store new guess in $f0
        mov.d $f0, $f6

        # Decrement loop counter
        addi $t0, $t0, -1
        bne $t0, $zero, loop

    # Print result
    li $v0, 3
    mov.d $f12, $f0
    syscall

    # Compare with sqrt.d
    li $v0, 4
    la $a0, sqr
    syscall
    li $v0, 3
    sqrt.d $f12, $f4
    syscall

    # Exit program
    li $v0, 10
    syscall

```

Line: 5 Column: 15 ☑ Show Line Numbers

```
The number we are going to take the square root is: 59843.32154
The guess number is: 244.62894665186292
The sqrt.d answer is: 244.62894665186292
-- program is finished running --

The number we are going to take the square root is: 9.0
The guess number is: 3.0
The sqrt.d answer is: 3.0
-- program is finished running --
```

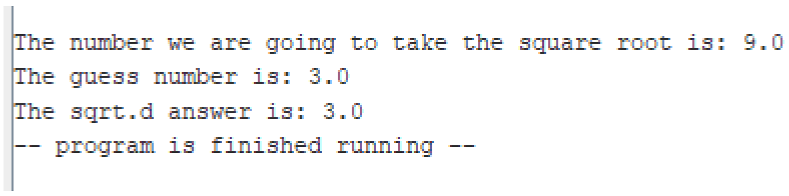
Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x1001004c
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x0040009c
hi		0x00000000
lo		0x00000000

COMPARING RESULTS:



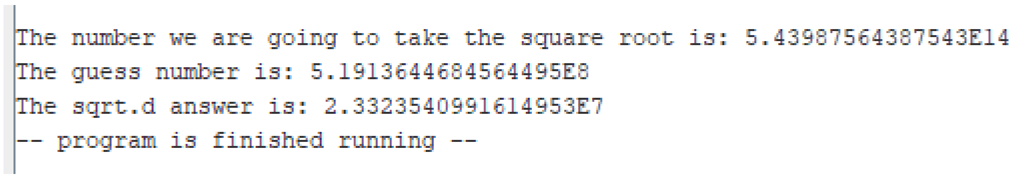
```
The number we are going to take the square root is: 9.984988675761652E15
The guess number is: 9.5227767724486E9
The sqrt.d answer is: 9.992491519016492E7
-- program is finished running --
```

Error = 9.5227767724486E9 - 9.992491519016492E7 = 852929148061692



```
The number we are going to take the square root is: 9.0
The guess number is: 3.0
The sqrt.d answer is: 3.0
-- program is finished running --
```

Error = 3.0 – 3.0 = 0.0



```
The number we are going to take the square root is: 5.43987564387543E14
The guess number is: 5.1913644684564495E8
The sqrt.d answer is: 2.3323540991614953E7
-- program is finished running --
```

Error = 2.3323540991614953E7 - 5.1913644684564495E8 = 4.858010370295 01E8

The difference gets bigger when the number is big. We can fix that by increasing the loops from 20 to something bigger.

CODE:

```
.data
sqnum: .asciiz "The number we are going to take the square root is: "
guessnum: .asciiz "\nThe guess number is: "
sqr: .asciiz "\nThe sqrt.d answer is: "
num: .double 5.43987564387543E14
guess: .double 1.0
half: .double 0.5
twenty: .word 20

.text
main:
        # Print message
        li $v0, 4
        la $a0, sqnum
        syscall

        # Print number
        li $v0, 3
        l.d $f12, num
        syscall

        # Print guess message
        li $v0, 4
        la $a0, guessnum
        syscall

        # Call square_root function
        j square_root
```

```
square_root:
        # Load initial guess
        l.d $f0, guess

        # Load loop counter
        lw $t0, twenty

        # Repeat 20 times
        loop:
                # Calculate new guess
                l.d $f4, num
                div.d $f6, $f4, $f0
                add.d $f6, $f6, $f0
                l.d $f2, half
                mul.d $f6, $f6, $f2

                # Store new guess in $f0
                mov.d $f0, $f6

                # Decrement loop counter
                addi $t0, $t0, -1
                bne $t0, $zero, loop

        # Print result
        li $v0, 3
        mov.d $f12, $f0
        syscall

        # Compare with sqrt.d
        li $v0, 4
        la $a0, sqr
        syscall
        li $v0, 3
        sqrt.d $f12, $f4
        syscall

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