ECEN 651

LAB NO. 4

MIPS Assembly and Simulation

TA: Mr. Ye Wang

DATE: 09/25/2019

Student Name: Dhiraj Dinesh Kudva

UIN: 829009538

**Objective:** The objectives of this lab are

1. To familiarize with the MIPS Instruction set architecture.
2. Executing the MIPS instruction set using MARS IDE compiler- This lab will focus on programming in the assembly language and checking the changes in register that occurs after each instruction.
3. Studying different techniques of bit level multiplication.

**Design:**

Part 1: In this part we will be performing a simple addition program, to be familiar with the MIPS assembly language.

1. To open the MARS, type the following command in the terminal window.

>java -jar /<location where the MARS file is downloaded>/Mars\_4\_1.jar

1. Then create a new file and use .asm extension.
2. Enter the following code.

#The original source code with pseudo instructions

.text #text section

.globl main #call main

main:

addi $t1,$0,8 #load immediate value(8) into $t1

addi $t2,$0,9 #load immediate value(9) into $t2

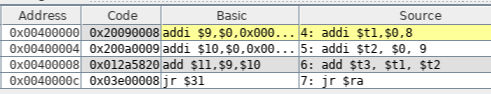
add $t3, $t1, $t2 #add two numbers into $t3

#jr $ra #return from main; return address stored in $ra

f. The MIPS instruction of the original source code: The basic column highlights this.

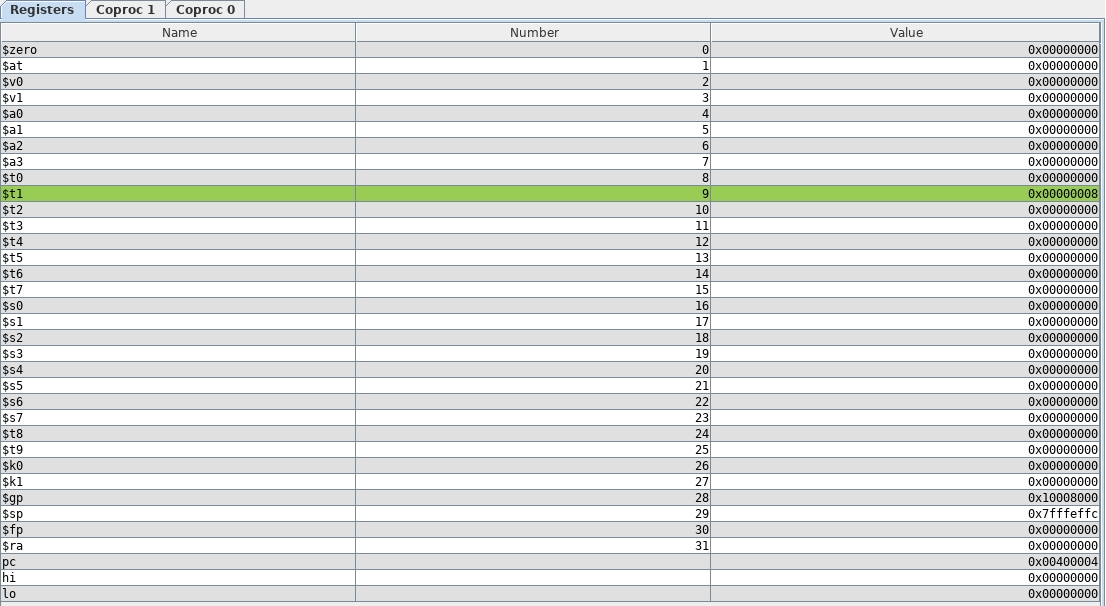
The machine code that MIPS translate to is given by the Code Column.

The address where each machine instruction is located is provided by the address column.

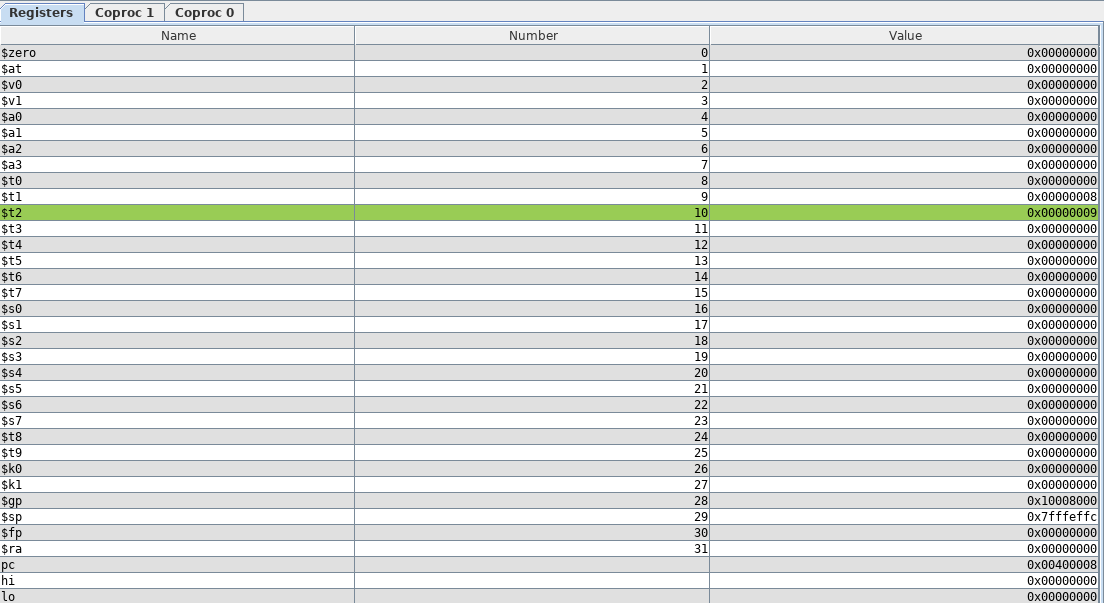


g. The above program just loads two numbers in $t1 and $t2 register by adding with $0 register. These two values are then added and stored in $t3 register. The jr instruction jumps back the control to return address.

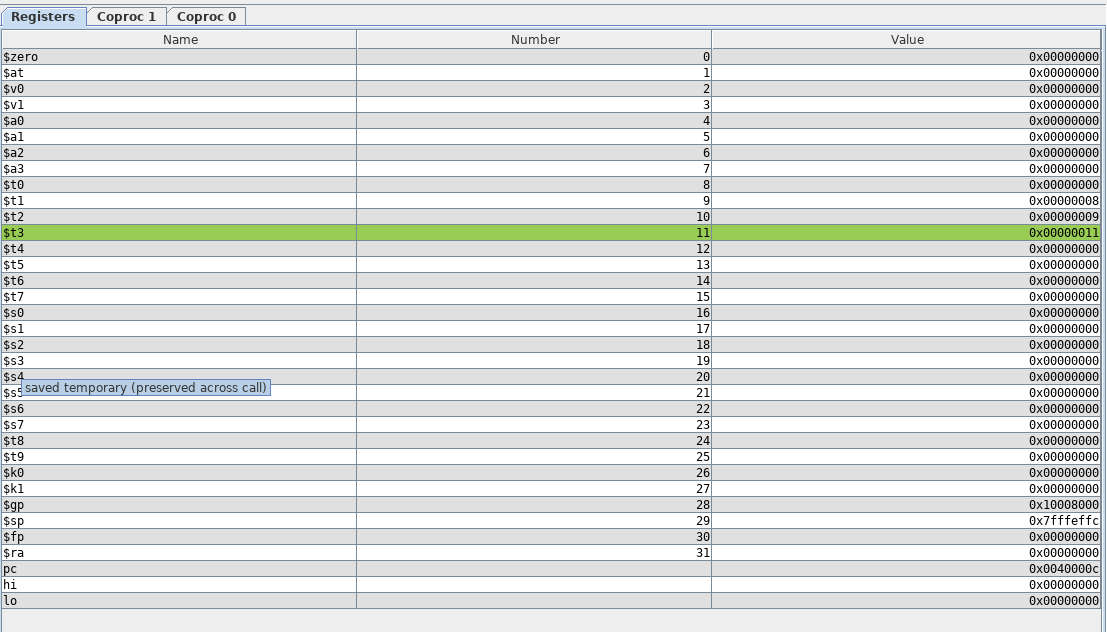
h.



In the above image, the number 8 is loaded in $t1 register.



In this image, the number 9 is loaded in $t2 register.



Here the output is stored in $t3. The result 8 and 9 is 17. The hex equivalent of 17 is stored in the $t3 register.

Part 2:

In this program, we will be carrying on shifting operation on the number entered by the user through console.

The source code of this is:

.data

msg1: .asciiz "Please enter an integer number:"

msg2: .asciiz "\t First Result : "

msg3: .asciiz "\t SEcond Result: "

.text

.globl main

# Inside main thereare some calls (syscall) which will change the

#value in register $ra which initially contains the return

#address from main.This needs to be saved.

main:

addu $s0, $ra, $0 #save $31 in $16

li $v0,4 #system call for print\_str

la $a0, msg1 #address of string to print

syscall

#now get an integer from the user

li $v0,5 #system call for read\_int

syscall #the integer placed in $v0

#do some computations here with the integer

addu $t0, $v0, $0 #move the number is $v0 to $t0

sll $t1, $t0,2 #computation 1, result is in $t1

srl $t2, $t0,2 #computation 2, result is in $t2

#print the first result

li $v0,4 #system call for print\_str

la $a0,msg2 #address of string to print

syscall

li $v0, 1 #system call for print\_int

addu $a0, $t1, $0 #move number to print in $a0

syscall

#print the second result

li $v0,4 #system call for print\_str

la $a0,msg3 #address of string to print

syscall

li $v0, 1 #system call for print\_int

addu $a0, $t2, $0 #move numberto print in $a0

syscall

#restore now the return address in $ra and return from main

addu $ra, $0, $s0 #return address back in $31

#jr $ra #return from main

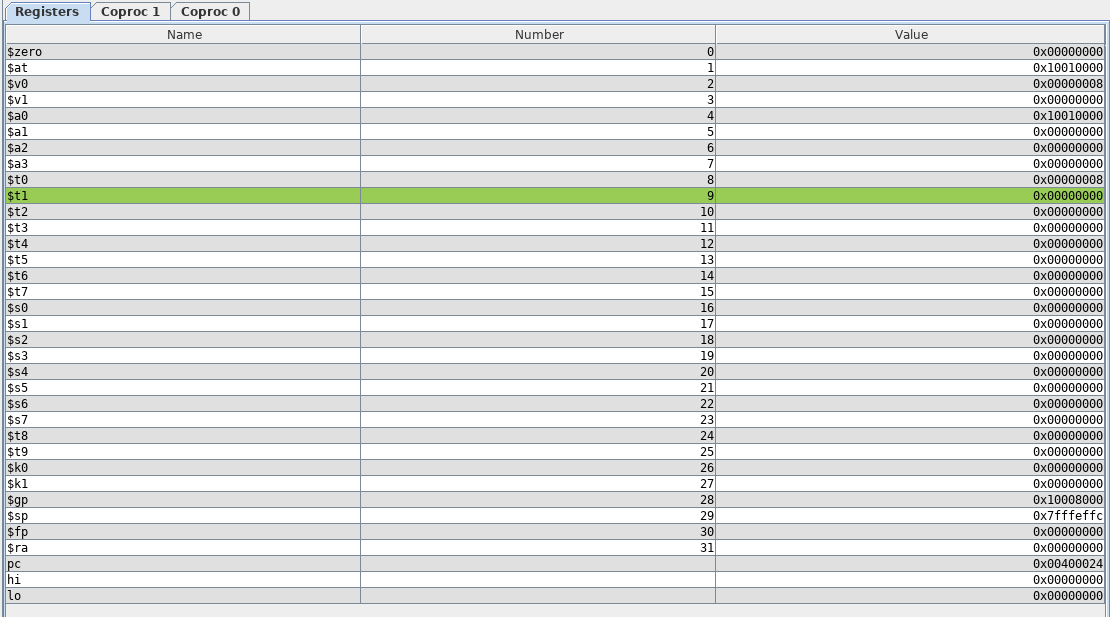
b. In this code, the program requests the user to enter two numbers and then multiply the number by 4 and divide the number by 4 and display the result. The multiplication is done by using shift logical left ‘sll’ instruction and division is done by using ‘srl’ command.

c. The output of the console.

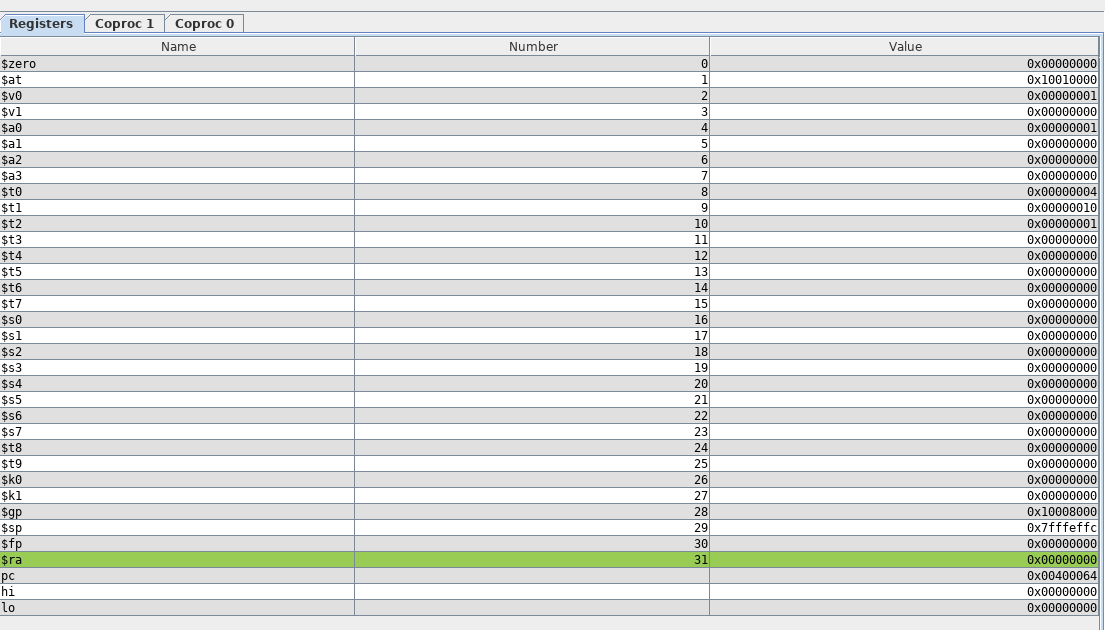


d. The text segment contains the pseudo code, its MIPS translation, the machine code and the addresses where it is stored. The data segment contains value at different memory locations. We can go to that address and see the value stored at that particular address.

e. The output when we run at single stepping are as follows:



In this stage, the $t1, $t2 and $t3 do not contain any values. $a0 contains the machine code of the message that is to be printed.



In this stage, the $t0 contains the user entered value and $t1 contains the hexadecimal value of the number when multiplied by 4 and $t2 contains the hexadecimal equivalent number when divided by 4.

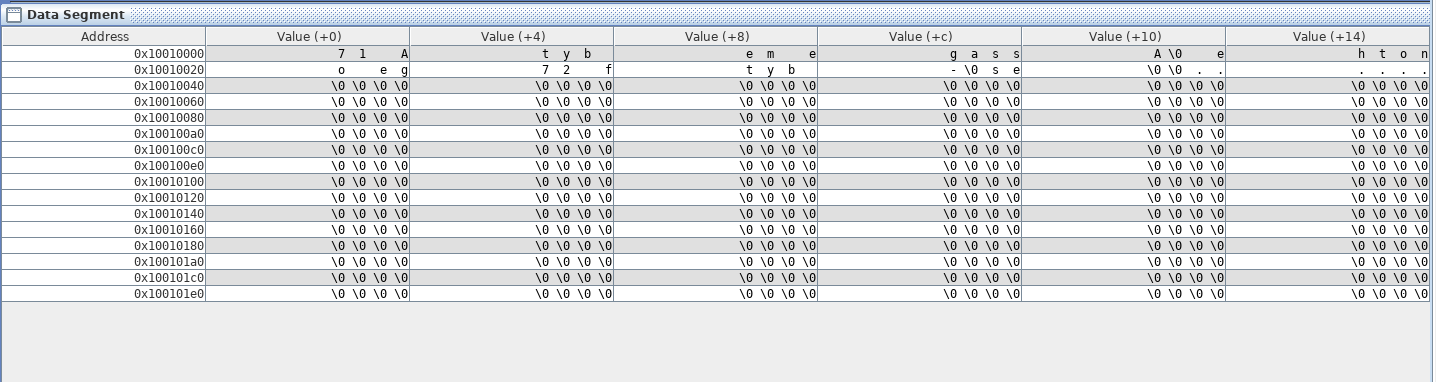
f. The operation of the program counter PC register is that it points to the next instruction in line. It stores the address of the next instruction.

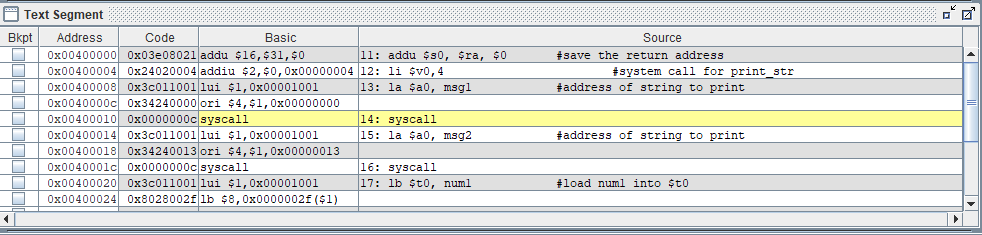
Part 3:

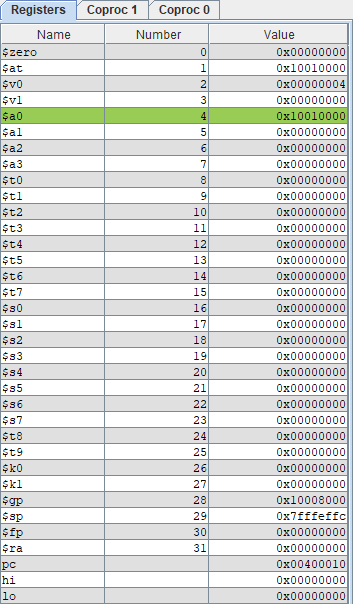
In this part, we are storing a string and display the string. The byte, word and half are also stored in the registers and then the same is seen in the data segment and registers part.

b.

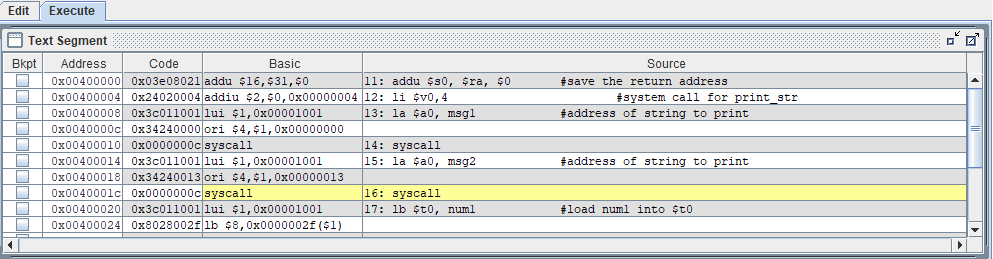
The contents of the data segment is shown in the below figure. The address is also shown in the first column.

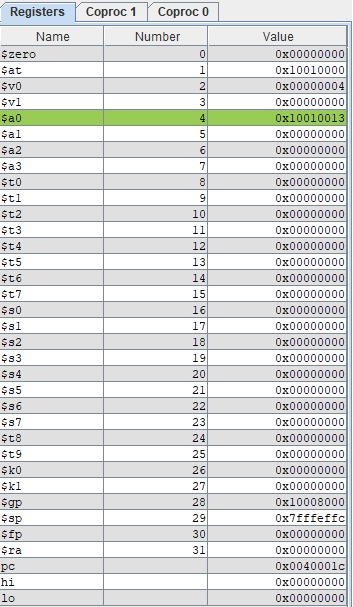


c. The contents of the register window is shown below: 

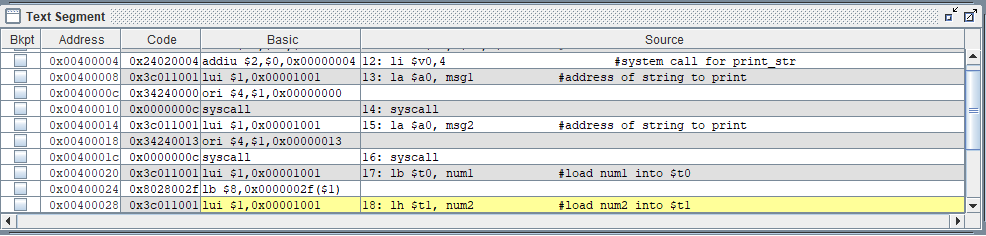


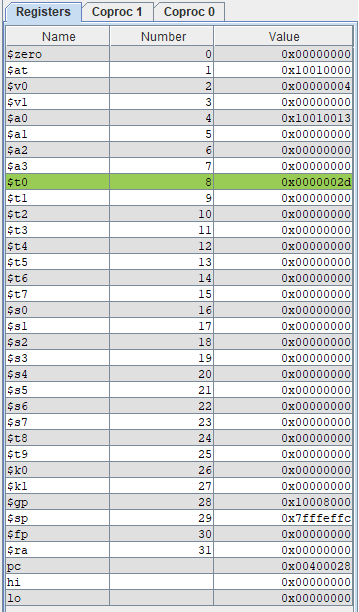
The value of msg1 which is to be printed is stored in $a0.



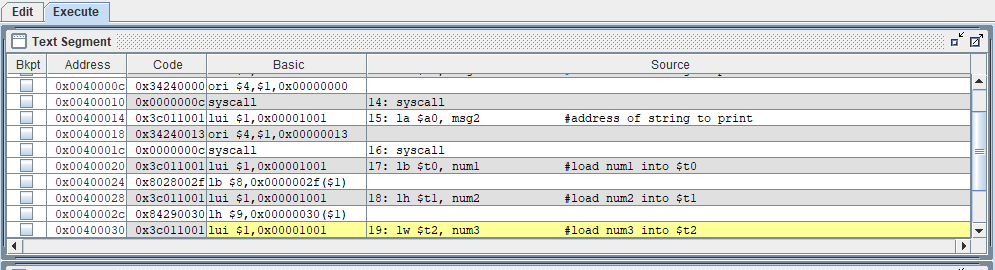


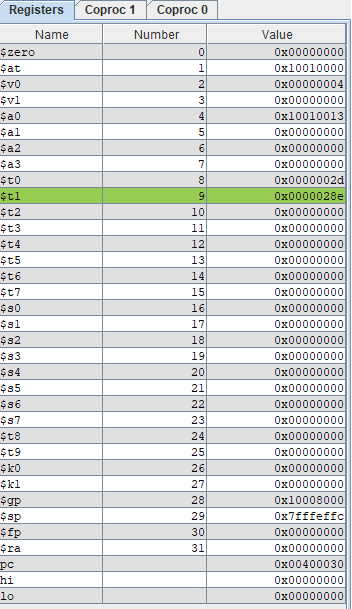
The value of second message to printed is stored in $a0.



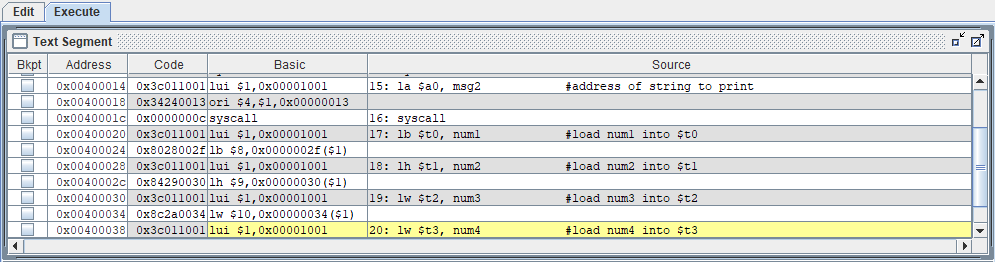


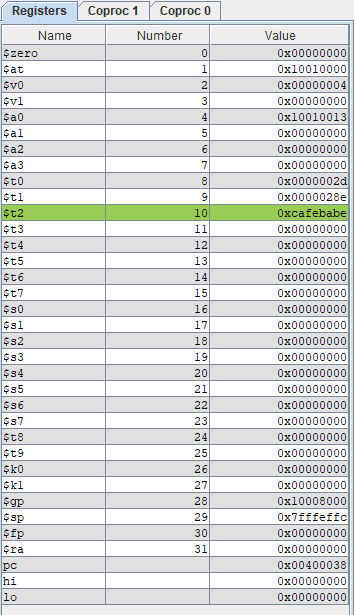
The value of the first number to be stored is shown in $t0.



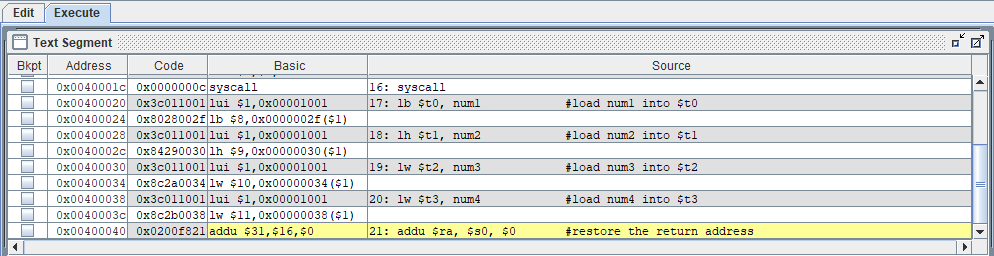


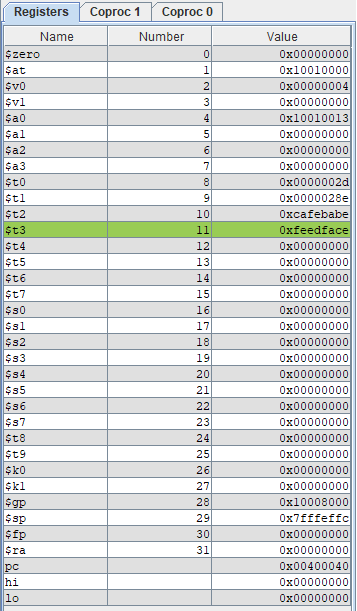
The value of the second number is stored in $t1.





The value of third number is stored in $t2.





The value of third register is stored in $t3.

The output in the console



d. In this program, the entered decimal number is compared and a corresponding hexadecimal number is provided.

The source code for this is :

.data

hextable: .ascii "0123456789abcdef"

msg1: .asciiz"Your number in Hex is: "

.text

.globl main

main:

#addu $s0, $0.$ra #save the return address

li $v0, 5 #syscall for read\_int

syscall

add $s1, $v0, $0

li $v0,4 #syscall for print\_str

la $a0, msg1

syscall

la $a1, hextable

srl $t0, $s1,4 #get upper 4 bits

add $a2, $a1, $t0 #get address in hextable

lb $a0,0($a2) #get character

li $v0,11 #syscall for print\_char

syscall

andi $t0, $s1, 0xf #get lower 4 bits

add $a2, $a1,$t0 #get address in hextable

lb $a0,0($a2) #get character

li $v0, 11 #syscall for print\_str

syscall

addu $ra, $s0, $0 #restore return address

#jr $ra #return from main

e. The output of the console is as shown below:



f. In this program, the user is asked to enter a number. That number is compared with the hextable written in the program. The comparison is done in the following way:

1. The entered number is split into two halves, upper bit and lower bit. The upper bit is obtained by right shifting the value in the register and the lower bit is obtained by anding with 0x0F.
2. These upper and lower halves are then individually compared with the hex table. The comparison is done by mapping the location of both the halves in the hex table.

**Part 4:**

In this part, two numbers asked from the user are multiplied and then the result is displayed using console.

The source code is as follows:

.data

msg1: .asciiz "Enter the first number: \n"

msg2: .asciiz "Enter the second number: \n"

msg: .asciiz " The product \n"

.text

.globl main

.globl my\_mul

main:

addi $sp, $sp, -8 #make room for $ra and $fp on the stack

sw $ra, 4($sp) #push $ra

sw $fp, 0($sp) #push $fp

la $a0, msg1 #load address of msg1 into $a0

li $v0, 4

syscall #print msg1

li $v0,5

syscall #read\_int

add $t0, $v0, $0 #put in $t0

la $a0, msg2 #load address of msg2 into $a0

li $v0, 4

syscall #print msg2

li $v0,5

syscall #read\_int

add $a1, $v0, $0 #put in $a1

add $a0, $t0,$0 #put first number in $a0

add $fp, $sp, $0 #set fp to top of stack pointer

#to function call

jal my\_mul #do mul, result is in $v0

add $t0, $v0, $0 #save the result in $t0

la $a0, msg

li $v0,4

syscall #print msg

add $a0, $t0, $0 #put computation result in $a0

li $v0,1

syscall #print result number

lw $fp, 0($sp) #restore (pop) $fp

lw $ra, 4($sp) #restore (pop) $ra

addi $sp, $sp, 8 #adjust $sp

jr $ra #return

my\_mul: #multiply $a0 with $a1

#does not handle negative $a1!

#Note: This is an inefficient way to multiply!

addi $sp, $sp,-4 #make room for $s0 on the stack

sw $s0, 0($sp) #push $s0

add $s0, $a1, $0 #set $s0 equal to $a1

add $v0,$0,$0 #set $v0 to 0

mult\_loop:

beq $s0, $0, mult\_eol

add $v0, $v0, $a0

addi $s0, $s0, -1

j mult\_loop

mult\_eol:

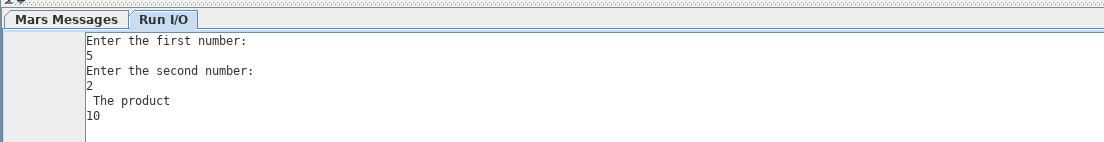
lw $s0,0($sp) #pop $s0

jr $ra

c. The run speed of 2 inst/sec executes 2 instruction per sec and we can see the value of the register changing at a slow speed. In this mode we can easily study which instruction is causing the value in the register to change.

d. In this program, we take two inputs from the user to multiply. This is done using addition method. The first number is added to itself a number of times, which is equal to the second number. This is executed using jump and beq instructions.

The two numbers are read from the console and stored in $t0 and $a1 respectively. Then the value in $a1 is stored in $s0. The $t0 is added $s0 times in the multiply loop and the value of $s0 is decremented at the steps of 1. When the value of $s0 is zero loop is exited and the value of $t0 is loaded in $a0 and the result is displayed.



e. The multiplication using Shift and add method source code is shown below:

.data

msg1: .asciiz "Enter the first number: \n"

msg2: .asciiz "Enter the second number: \n"

msg: .asciiz " The product \n"

.text

.globl main

.globl my\_mul

main:

addi $sp, $sp, -8 #make room for $ra and $fp on the stack

sw $ra, 4($sp) #push $ra

sw $fp, 0($sp) #push $fp

la $a0, msg1 #load address of msg1 into $a0

li $v0, 4

syscall #print msg1

li $v0,5

syscall #read\_int

add $t0, $v0, $0 #put in $t0

la $a0, msg2 #load address of msg2 into $a0

li $v0, 4

syscall #print msg2

li $v0,5

syscall #read\_int

add $a1, $v0, $0 #put in $a1

add $a0, $t0,$0 #put first number in $a0

add $fp, $sp, $0 #set fp to top of stack pointer

#to function call

jal my\_mul #do mul, result is in $v0

add $t0, $v0, $0 #save the result in $t0

la $a0, msg

li $v0,4 #print msg

syscall

add $a0, $t0, $0 #put computation result in $a0

li $v0,1

syscall #print result number

lw $fp, 0($sp) #restore (pop) $fp

lw $ra, 4($sp) #restore (pop) $ra

addi $sp, $sp, 8 #adjust $sp

j exit

my\_mul:

addi $sp, $sp,-4 #make room for $s0 on the stack

sw $s0, 0($sp) #push $s0

add $s0, $a1, $0 #set $s0 equal to $a1

add $v0,$0,$0 #set $v0 to 0

mult\_loop:

beq $s0, $0, mult\_eol

andi $t2, $s0,1 #checking if the last bit is 1

bne $t2,1,name1 #if last bit is not 1, then jump to label name1

#skipping the add step

#if last bit is 1, then add the second number to first number

add $v0, $v0, $a0

name1:

sll $a0,$a0,1 #shifting the first number left by 1.

srl $s0,$s0,1 #shifting the second number right by 1.

j mult\_loop

mult\_eol:

lw $s0,0($sp)

jr $ra

exit:

The shift add method is more efficient than the earlier only write method. In this the number of loop required for multiplication are less and also it performs correct results when the second number is negative. The earlier method is not useful for negative numbers. It will not produce any result in the earlier method, but in the second method correct result will be obtained.

**Questions:**

1. Explain how the target address in a branch instruction is calculated. How about for a jump instruction? What is the different between jump and jump and link (i.e. jal)?

Ans. The branch instruction in MIPS has 16 bits offset to determine the next instruction. The program counter PC gets updated during the next fetch cycle to PC +4. Since this PC is word aligned, we sign extend the offset and multiply it by 4. Then add this word aligned to the PC+4. This is the branch instruction address. For jump instruction, the offset is of 26 bits. This offset is multiplied by 4 and added to PC+4. This becomes the jump instruction address. The jump instruction loads an immediate value into the program counter (PC) register. The jump and link (jal) is similar to jump instruction, except that it also stores the address of the next instruction, i.e. the instruction immediately after the jump, in the return address register ($ra, $31).

1. List the registers that must be preserved during the execution of a procedure. Which registers do not have to be preserved?

Ans: The registers which are not preserved by the compiler implicitly and hence needs to be preserved explicitly by the programmer during the execution of a procedure are :gp (Global pointer) ,sp(Stack Pointer) ,ra (Return address) and t0-t9 (temporary register).

The registers which do not have to be preserved during the execution are those which are preserved by the compiler itself. For example, $s0-$s7 are saved registers that are saved by the compiler.

1. What are the steps necessary to call a procedure? Be sure to include the interaction with the stack and frame pointers.

Ans: Whenever a function call is made, in MIPS it is made either using jal (jump after link) or jump. In both the cases the PC is first updated with the branch address and in jal, the address of the next instruction is stored in the return address register $ra.

The following are the steps to call a procedure:

1. Firstly, as the stack pointer can be changed during a procedure call, we first allocate space for frame pointer and return address. This is done by lowering the address of stack pointer. As in the multiplication program, we lower the stack pointer address by $sp-8 and then push ra and fp.
2. Before a function is called, the stack pointer is copied to the frame pointer as the stack pointer will point to the function once it is called and hence it value will be changed.
3. Then if the command is jal, the address of the next instruction is stored in the return address.
4. After the procedure is over, the stack pointer is again restored and the value of fp and ra are popped out.