

## Laboratory Exercise #4

### MIPS Assembly and Simulation

#### Objective

The fourth laboratory's objective is to assemble and simulate basic assembly language programs to understand and familiarize with the MIPS Instruction Set Architecture. For the code simulation we use the MIPS Assembler and Runtime Simulator (MARS) to execute our code and understand the flow of the data through the MIPS machine which helps us strengthen our understanding of the processor and its functioning.

#### Design

##### 1. Launch and install MIPS Assembler and Runtime Simulator (MARS)

- a) Start MARS by first opening a terminal window, then invoking the following command:  
    >java -jar /homes/grad/atarghe1/ECEN651/Mars\_4\_1.jar
- b) From within the IDE, select File→New to create a new source file.

##### 2. Enter the specific code into the editor within MARS:

- a) Save the file as prog1.asm in our lab4 directory and click on Run→Assemble.
- b) Select the Execute tab in MARS
- c) Now select Run→Step to single step through the program. Examine the state of the machine provided in the Registers window on the right as we single step through your program.
- d) Examine the Text Segment and Data Segments under the Execute tab in MARS
- e) Repeat the same procedures for executing other programs.

##### 3. Source Files:

- a) \*\*\*\*ASSEMBLY PROGRAM - ADDITION\*\*\*\*

```
.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
```

- b) \*\*\*\*ASSEMBLY PROGRAM – SLL & SRL OPERATION\*\*\*\*

```
.data                # start a group of variable declarations
                    # $tns are all temporary registers
msg1 : .asciiz "Please enter an integer number: "
msg2 : .asciiz "\ t First result "
msg3 : .asciiz "\t Second result"
.text                # start a group of assembly language instructions
.globl main
                    # inside main there are some calls (syscall) which will change the value
```

# value in register \$ra which initially contains the return  
#address from main. This needs to be saved.

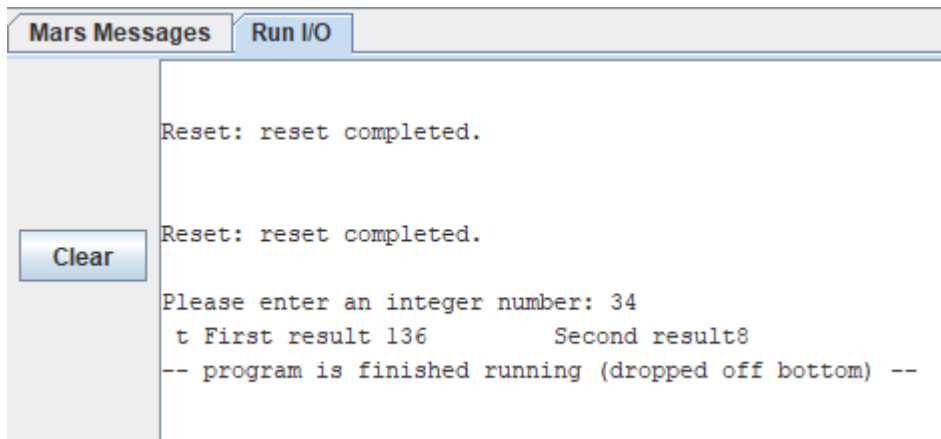
```
main:
addu $s0, $ra, $0          #save $31 in $16
li $v0,4                   # system call for print_str
la $a0, msg1               #a d d r e s s o f s t r i n g t o p r i n t
syscall                   # message1 printed
                           #now get an integer from the user
li $v0, 5                  #system call for read_int
syscall                   #the integer placed in $v0
```

# do some computation here with the integer  
addu \$t0,\$v0,\$0 # move the number in \$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, load immediate
la $a0,msg2                #address of string to print, load address
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 number to 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 back to the operating system
```



c) \*\*\*\*ASSEMBLY PROGRAM – LOAD NUMBERS\*\*\*\*

```
.data
msg1: .asciiz "A 17 byte message"
msg2: .asciiz "another message of 27n bytes"
num1: .byte 45
num2: .half 654
num3: .word 0xcafebabe
num4: .word 0xfeedface
.text
.globl main
main:
addu $s0,$ra,$0          #save the return address
li $v0,4                  #system call for print_string
la $a0,msg1               #address of string to be printed
syscall

la $a0,msg2               #address of string to print
syscall

lb $t0,num1               #load num1 in $t0
lh $t1,num2               #load num2 in $t1
lw $t2,num3               #load num3 in $t2
lw $t3,num4               #load num4 in $t3
addu $ra,$s0,$0          #restore the return address
#jr $ra                   #return from main
```

d) \*\*\*\*ASSEMBLY PROGRAM – HEXADECIMAL OUTPUT\*\*\*\*

#the following program outputs the hexadecimal value only for integer values between 0 & 255  
 # i.e. 8 bit values only

```

.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           #save the data in v0 in s0
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) \*\*\*\*\*ASSEMBLY PROGRAM – MULTIPLICATION\*\*\*\*\*

```

.data
msg1: .asciiz "enter the first number\n"
msg2: .asciiz "enter the second number \n"
msg: .asciiz "The product is "
.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

```

```

li $v0,4
syscall #print msg1
li $v0,5
syscall #read_int
add $t0,$v0,$0 #put int in $t0
la $a0,msg2 #load address
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 prior
                # 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

```

```

lw $fp,0($sp) #restore (pop) $ fp
lw $ra,4($sp) #restore (pop) $ra
addi $sp,$sp,8 #adjust $sp
#jr $ra #return to main
J exit

```

```

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

```

```

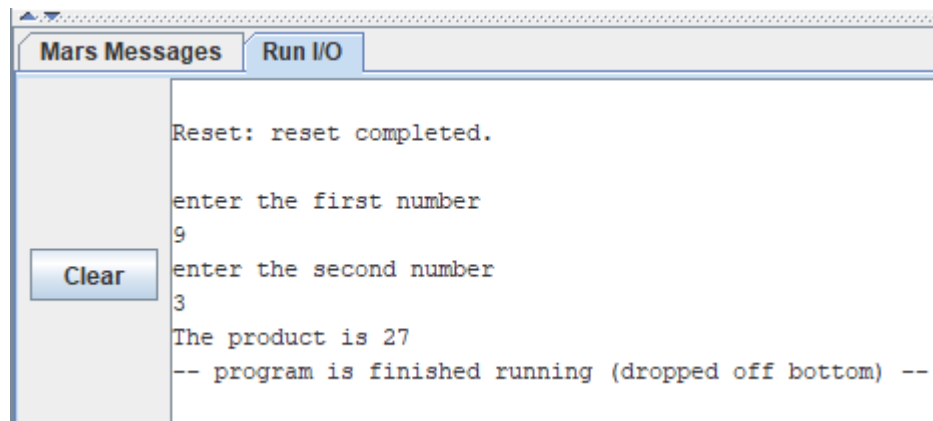
mult_eol :

```

```
lw $s0,0($sp) #pop $s0
jr $ra
```

exit:

OUTPUT:



f) (I)\*\*\*\*ASSEMBLY PROGRAM – MULTIPLICATION SHIFT & ADD\*\*\*\*

#APPROACH -1 USING SHIFT LEFT LOGICAL(SLL) AND 32-BIT COUNTER:

```
.data
msg1: .asciiz "enter the first number\n"
msg2: .asciiz "enter the second number \n"
msg: .asciiz "The product is "
.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
li $v0,4
syscall                  #print msg1
li $v0,5
syscall                  #read_int
add $t0,$v0,$0           #put int in $t0
la $a0,msg2              #load address
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 prior
                        # 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

```

```

lw $fp,0($sp)          #restore (pop) $ fp
lw $ra,4($sp)          #restore (pop) $ra
addi $sp,$sp,8         #adjust $sp
#jr $ra                #return to main
j exits

```

```

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 $s1,$a0,$0 # set $s1 equal to $a0
add $v0,$0,$0 #set $v0 to 0
li $s3, 1 # Mask for extracting bit!
li $t1, 0 # Counter for loop.

```

```

mult_loop:
    beq $t1, 31, exit
    and $t0, $s1, $s3
    sll $s3, $s3, 1

    beq $t0, 0, multiply_inc
    add $v0, $v0, $s0

```

```

multiply_inc:
    sll $s0, $s0, 1
    addi $t1, $t1, 1
    j mult_loop

```

```

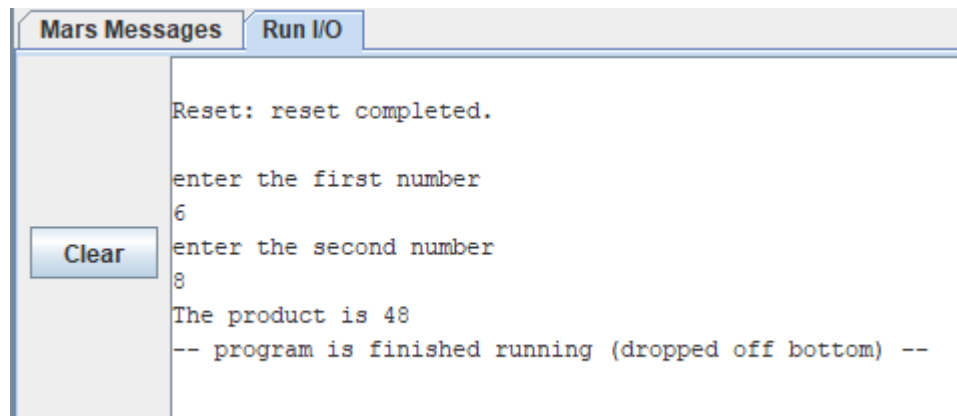
exit:

```

```
lw $s0,0($sp)      #pop $s0
jr $ra
```

exits:

OUTPUT:



The screenshot shows the 'Mars Messages' window with a 'Run I/O' button. The output text is as follows:

```
Reset: reset completed.
enter the first number
6
enter the second number
8
The product is 48
-- program is finished running (dropped off bottom) --
```

A 'Clear' button is visible on the left side of the window.

#APPROACH -2 USING SHIFT LEFT LOGICAL(SLL) AND SHIFT RIGHT LOGICAL(SRL):

```
.data
msg1: .asciiz "enter the first number\n"
msg2: .asciiz "enter the second number \n"
msg: .asciiz "The product is "
.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
li $v0,4
syscall #print msg1
li $v0,5
syscall #read_int
add $t0,$v0,$0 #put int in $t0
la $a0,msg2 #load address
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 prior
                # 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

```

```

lw $fp,0($sp) #restore (pop) $ fp
lw $ra,4($sp) #restore (pop) $ra
addi $sp,$sp,8 #adjust $sp
#jr $ra #return to main
j exit
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
    and $t0,$s0,0x1
    beq $t0,$0,here
    add $v0,$v0,$a0
here:  sll $a0,$a0,1
        srl $s0,$s0,1
        j mult_loop

```

```

mult_eol :
    lw $s0,0($sp) #pop $s0
    jr $ra

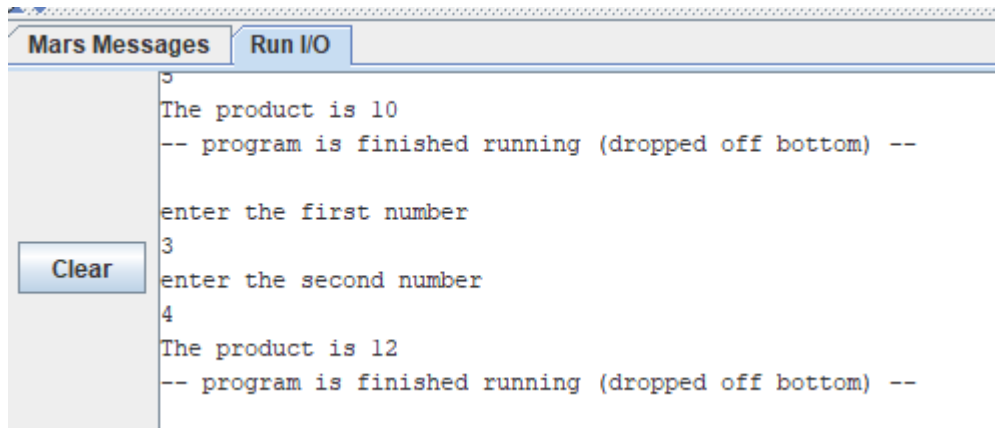
```

```

exit:

```

OUTPUT:



The screenshot shows the Mars Messages window with two tabs: 'Mars Messages' and 'Run I/O'. The 'Run I/O' tab is active, displaying the following text: 'The product is 10', '-- program is finished running (dropped off bottom) --', 'enter the first number', '3', 'enter the second number', '4', 'The product is 12', and '-- program is finished running (dropped off bottom) --'. A 'Clear' button is visible on the left side of the window.

## QUESTIONS

- (a) 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)?

MIPS architecture's branch instruction has only 16 bits offset to determine next instruction. So, we use a 16-bit register to add to this 16-bit value in order to determine next instruction. This register is implied by the architecture. It is a PC register and hence PC gets updated ( $PC+4$ ) during the fetch cycle so that it holds the address of the next instruction. Hence to calculate a branch target address in a branch instruction we do the following:

- Sign extend the 16-bit offset value to preserve its value.
- Multiply resulting value with 4. This is done because if we are going to branch some address, and PC is already word aligned, then the immediate value must be word-aligned as well. However, it makes no sense to make the immediate word-aligned because we would be wasting low two bits by forcing them to be 00.
- Now we have 32-bit address. Add this value to  $PC + 4$  and the value we get will be our branch address.

In order to calculate jump target instruction, we must understand that jump instruction is loaded in the PC with a 32-bit address.

Here is the machine language form of the instruction:

6            26  
000010 00000000000000000000000000000000 -- fields of the instructions

opcode          target                  -- meaning of the fields

There is room in the instruction for a 26-bit address. The 26-bit target address field is transformed into a 32-bit address. This is done at run-time, as the jump instruction is executed.

Instructions always start on an address that is a multiple of four (they are word-aligned). So, the low order two bits of a 32-bit instruction address are always "00". Shifting the 26-bit target left two places results in a 28-bit word-aligned address (the low-order two bits become "00".)

After the shift, we need to fill in the high-order four bits of the address. These four bits come from the high-order four bits in the PC. These are concatenated to the high-order end of the 28-bit address to form a 32-bit address. So, the steps to be followed are as follows:

- Multiply 26-bit value with 4.
- Since we are jumping relative to PC value, concatenate first four bits of PC value to left of our jump address.
- Resulting address is the jump value.

The MIPS processor has two instructions that enable us to call functions from anywhere and return back to where we called the function from – jr and jal:

### ➔ Jump and Link (jal)

#### Jal label

This instruction copies the address of the next instruction (the one immediately after the jump) into the return address register \$ra (register 31) and then jumps to the address label. This allows a subroutine to return to the main body routine after completion. The **jal** instruction and register \$31 provide the hardware support necessary to elegantly implement subroutines.

#### jalr label

This instruction loads the PC register with a value stored in a register and the return address is loaded into a specified register (or \$ra if not specified).

### ➔ Jump (j)

#### J label

The jump instructions load a new value into the PC register, which stores the value of the instruction being executed. This causes the next instruction read from memory to be retrieved from a new location.

The **j** instruction loads an immediate value into the PC register. This immediate value is either a numeric offset or a label (and the assembler converts the label into an offset).

#### Jr label

The **jr** instruction loads the PC register with a value stored in a register. As such, the jr instruction can be called as such:

jr \$t0

assuming the target jump location is in \$t0

**(b) List the registers that must be preserved during the execution of a procedure. Which registers do not have to be preserved?**

Rules for register-use are called procedure call conventions which help us understand how registers are supposed to be used and how stack frames should be laid out in memory so that a piece of software code does not conflict with other. To compile a procedure, the compiler must know which registers need to be preserved and which can be modified without worry.

**Preserved across procedure calls** means that the value stored in the register will not be changed by a procedure. It is safe to assume that the value in the register after the procedure call is the same as the value in the register before the call i.e. it remains unchanged.

The following table lists the registers that must be preserved during the execution of a procedure across subroutine calls:

Register number	Register name	Usage
\$16-\$23	\$s0-\$s7	The Saved Registers.
\$28	\$gp	The Global Pointer (pointer to global area) used for addressing static global variables.
\$29	\$sp	The Stack Pointer
\$30	\$fp (or \$s8)	The Frame Pointer. Programs that do not use an explicit frame pointer can use register \$30 as another saved register. Not recommended however.
\$31	\$ra	The Return Address in a subroutine call.
	\$f20 - \$f30	Saved floating point values.

**Not preserved across procedure calls** means that the register may change value if a procedure is called. If some value is stored in that register before the procedure is called, then we may not assume that the same value will be in the register at the return from the procedure.

The following table lists the registers that need not be preserved during the execution of a procedure:

Register number	Register name	Usage
\$0	\$0	Always 0
\$1	\$at	Reserved for the assembler. The Assembler Temporary used by the assembler in expanding pseudo-ops
\$2-\$3	\$v0-\$v1	Expression evaluation and procedure return results. These registers contain the Returned Value of a subroutine; if the value is 1 word only, \$v0 is significant.

\$4-\$7	\$a0-\$a3	The Argument registers, these registers contain the first 4 argument values for a subroutine call.
\$8-\$15, \$24, \$25	\$t0-\$t9	The Temporary Registers.
\$26-\$27	\$k0-\$k1	The Kernel Reserved registers
	\$f4 - \$f10	Temporary registers.
	\$f12 - \$f14	The first two floating point parameters
	\$f16 - \$f18	Temporary floating-point registers.

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

Procedures (or functions) are a crucial program structuring mechanism. To support procedures, we need to define a calling convention: a sequence of steps followed by the calling procedure and the called procedure to assure correct passage of parameters, results, and control flow.

The processor supports procedure calls in two different ways:

- CALL and RET instructions.
- ENTER and LEAVE instructions, in conjunction with the CALL and RET instructions.

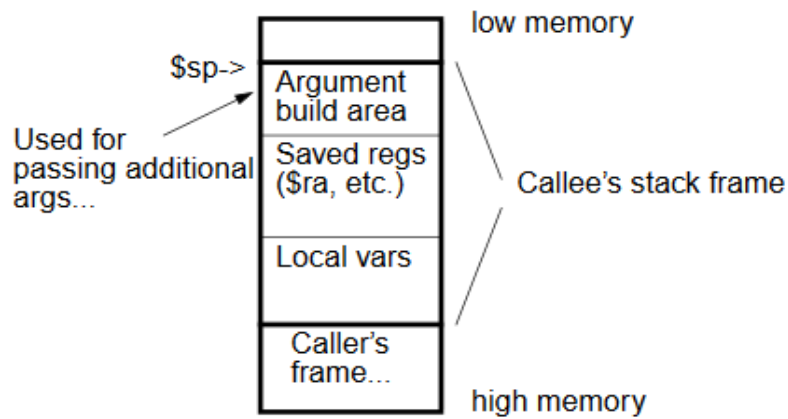
Both of these procedure call mechanisms use the procedure stack, to save the state of the calling procedure, pass parameters to the called procedure, and store local variables for the currently executing procedure.

### **Program Stack:**

A stack is a dynamic data structure that is accessed in a LIFO manner and is automatically allocated by the OS when the program starts up. The register \$sp (register 29 on the MIPS) is automatically loaded to point to the first empty slot on the top of the stack. By convention, the stack grows towards lower memory addresses. To allocate space on the stack, we decrement \$sp. To free old stack space, we increment \$sp.

Procedure Stack Frame is a block of memory on the stack that is used for:

- Passing arguments
- Saving registers
- Space for local variables



While executing the procedure, we deal with two calls:

- 1) Callee – The procedure that is called from the main () function. The procedure is the callee.
- 2) Caller – The procedure that does the calling - main () function. The caller calls the procedure.

In any general procedural call, the following sequence of steps are expected to be performed:

1. Caller must pass the return address (where to continue execution after the call) to the callee.
2. Caller must pass parameters to callee.
3. Caller must save registers that the callee might want to use.
4. Jump to the first instruction of the callee.
5. Callee must allocate space for local variables, and possibly save registers.
6. Callee function execution.
7. Callee has to restore registers (possibly) and return to caller
8. Caller function execution continued.

For a general multiplication code, execution of steps is explained as follows:

```
.data
msg1: .asciiz "enter the first number\n"
msg2: .asciiz "enter the second number \n"
msg: .asciiz "The product is "
.text
.globl main
.globl my_mul
```

# Here the caller function is the main function and the callee function is my\_mul which does the multiplication operation.

main:

# As PC increments by 4, increment stack pointer by 8 to reserve space for register ra and frame pointer.

Frame pointer separates one frame from another.

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

#Now we can use \$ra and \$fp in the body of the procedure. Save them on stack.

```
sw $ra,4($sp)                                #push $ra to stack
sw $fp,0($sp)                                #push $fp to stack

la $a0,msg1                                  #load address of message1
li $v0,4
syscall                                      #print msg1
li $v0,5
syscall                                      #read_int
add $t0,$v0,$0                               #put int in $t0
la $a0,msg2                                  #load address of message 2
li $v0,4
syscall                                      #print msg2
li $v0,5
syscall                                      #read_int
add $a1,$v0,$0                               #put second number in $a1
add $a0,$t0,$0                               #put first number in $a0
add $fp,$sp,$0                               #set fp to top of stack prior
                                           # to function call
```

#The caller gives the callee arguments at this stage. For this code we don't have any arguments to be passed to callee. The jal to the procedure keeps track of the instruction after the jal so that we can continue at the right place when we are done with the procedure. It stores the return address (PC+4) in \$ra (R31) before jumping to the function.

```
jal my_mul                                    #do mul,result is in $v0. Calls function my_mul

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
```

