# ABOUT THE AUTHOR

I am Saiteja and I am from Hyderabad, India. I completed my bachelors in Electronics and Communication Engineering from Gokaraju Rangaraju Institute of Engineering and Technology with a CGPA of 9.97/10 in the year 2019. I received a Gold Medal for the academic excellence from my university. Right after that, I got an opportunity to work for Tata Consultancy Services as a Systems Engineer and have worked for 3 years until July 2022. I speak English, Hindi, and Telugu where the last of these is my first language. My interests include but not limited to Cricket, Table-Tennis and learning new languages. Currently I am pursuing my master's in Computer Engineering at San Jose State University.

# TABLE OF CONTENTS

# LIST OF FIGURES

## INTRODUCTION:

This activity is used to design and develop an assembly code to build a 50-entry array with the base address 0x100 in MARS and gain some familiarity on the MIPS implementation of arrays, stacks, procedures, and recursive procedures by logging few selected registers' values and memory content of selected addresses in a table for the given task.

## MY GROUP:

**Name:** Student_Team 6
**Members:** Tirumala Saiteja Goruganthu (016707210), Harish Marepalli (016707314)

## GIVEN TASK:

The task is to write a MIPS assembly program to perform an arithmetic computation on the given array elements by accessing them and use the result as input to find the factorial using the recursive procedure.
Following are the requirements presented for the task:

a. The MIPS code should be under the line "#your code goes in here…"
b. Register assignments must be $a1 ← n, $a0 ← array-base-address, and $s0 ← n!
c. Your factorial function must be implemented as a recursive procedure.
d. The final value of n obtained from the arithmetic calculation must be written to the memory location at address 0x00.
e. The factorial n! must be written to the memory location at address 0x10.

## STEPS TAKEN TO COMPLETE THE TASK:

a. Spent time to understand the given question carefully.
b. Imported the given assembly file into the MARS simulator.
c. Revised few topics related to arrays, function calls, procedures, and recursive procedures from the power-point presentations discussed in class.
d. Edited the file by making changes at the right place according to the given task.
e. Assembled the edited code in the MARS simulator and executed each line of the code step-by-step to check the purpose of that line.
f. If found some issues, went back to editing the code and re-assembled it again.
g. Check if the output is as expected after the program execution.
h. If the output has no errors, then proceed by executing each instruction step-by-step and simultaneously taking the log of the contents of the registers.
i. This log is finally dumped into the given test log table.
j. Finally, snippets are taken to show the current output along with the screenshots of the stack.

**TEST LOG:**

**Programmer's Names:** Tirumala Saiteja Goruganthu **Date:** 10/02/2022

| Addr | MIPS Instruction | Machine Code | Registers | | | | Memory Content | |
|---|---|---|---|---|---|---|---|---|
| | | | $a1 | $sp | $ra | $v0 | [0x00] | [0x10] |
| 3034 | lw $t0, 100($a0) | 0x8c880064 | 0x00000032 | 0x00002ffc | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 3038 | lw $t1, 120($a0) | 0x8c890078 | 0x00000032 | 0x00002ffc | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 303c | add $t1, $t0, $t1 | 0x01094820 | 0x00000032 | 0x00002ffc | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 3040 | addi $t0, $0, 30 | 0x2008001e | 0x00000032 | 0x00002ffc | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 3044 | divu $t1, $t0 | 0x0128001b | 0x00000032 | 0x00002ffc | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 3048 | mflo $a1 | 0x00002812 | 0x00000005 | 0x00002ffc | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 304c | sw $a1, 0($0) | 0xac050000 | 0x00000005 | 0x00002ffc | 0x00000000 | 0x00000000 | 0x00000005 | 0x00000000 |
| 3050 | jal factorial | 0x0c000c19 | 0x00000005 | 0x00002ffc | 0x00003054 | 0x00000000 | 0x00000005 | 0x00000000 |
| 3054 | add $s0, $v0, $0 | 0x00408020 | 0x00000005 | 0x00002ffc | 0x00003054 | 0x00000078 | 0x00000005 | 0x00000000 |
| 3058 | sw $s0, 16($0) | 0xac100010 | 0x00000005 | 0x00002ffc | 0x00003054 | 0x00000078 | 0x00000005 | 0x00000078 |
| 305c | li $v0, 10 | 0x2402000a | 0x00000005 | 0x00002ffc | 0x00003054 | 0x0000000a | 0x00000005 | 0x00000078 |
| 3060 | syscall | 0x0000000c | 0x00000005 | 0x00002ffc | 0x00003054 | 0x0000000a | 0x00000005 | 0x00000078 |
| 3064 | addi $sp, $sp, -8 | 0x23bdfff8 | 0x00000001 | 0x00002fd4 | 0x00003090 | 0x00000000 | 0x00000005 | 0x00000000 |
| 3068 | sw $a1, 4($sp) | 0xafa50004 | 0x00000001 | 0x00002fd4 | 0x00003090 | 0x00000000 | 0x00000005 | 0x00000000 |
| 306c | sw $ra, 0($sp) | 0xafbf0000 | 0x00000001 | 0x00002fd4 | 0x00003090 | 0x00000000 | 0x00000005 | 0x00000000 |
| 3070 | addi $t0, $0, 2 | 0x20080002 | 0x00000001 | 0x00002fd4 | 0x00003090 | 0x00000000 | 0x00000005 | 0x00000000 |
| 3074 | slt $t0, $a1, $t0 | 0x00a8402a | 0x00000001 | 0x00002fd4 | 0x00003090 | 0x00000000 | 0x00000005 | 0x00000000 |
| 3078 | beq $t0, $0, else | 0x11000003 | 0x00000001 | 0x00002fd4 | 0x00003090 | 0x00000000 | 0x00000005 | 0x00000000 |
| 307c | addi $v0, $0, 1 | 0x20020001 | 0x00000001 | 0x00002fd4 | 0x00003090 | 0x00000001 | 0x00000005 | 0x00000000 |
| 3080 | addi $sp, $sp, 8 | 0x23bd0008 | 0x00000001 | 0x00002fdc | 0x00003090 | 0x00000001 | 0x00000005 | 0x00000000 |
| 3084 | jr $ra | 0x03e00008 | 0x00000001 | 0x00002fdc | 0x00003090 | 0x00000001 | 0x00000005 | 0x00000000 |
| 3088 | addi $a1, $a1, -1 | 0x20a5ffff | 0x00000001 | 0x00002fdc | 0x00003090 | 0x00000000 | 0x00000005 | 0x00000000 |
| 308c | jal factorial | 0x0c000c19 | 0x00000001 | 0x00002fdc | 0x00003090 | 0x00000000 | 0x00000005 | 0x00000000 |
| 3090 | lw $ra, 0($sp) | 0x8fbf0000 | 0x00000004 | 0x00002ff4 | 0x00003054 | 0x00000018 | 0x00000005 | 0x00000000 |
| 3094 | lw $a1, 4($sp) | 0x8fa50004 | 0x00000005 | 0x00002ff4 | 0x00003054 | 0x00000018 | 0x00000005 | 0x00000000 |
| 3098 | addi $sp, $sp, 8 | 0x23bd0008 | 0x00000005 | 0x00002ffc | 0x00003054 | 0x00000018 | 0x00000005 | 0x00000000 |
| 309c | mul $v0, $a1, $v0 | 0x70a21002 | 0x00000005 | 0x00002ffc | 0x00003054 | 0x00000078 | 0x00000005 | 0x00000000 |
| 30a0 | jr $ra | 0x03e00008 | 0x00000005 | 0x00002ffc | 0x00003054 | 0x00000078 | 0x00000005 | 0x00000000 |
| 30a4 | | | | | | | | |

**SCREEN CAPTURES:**

a. Final assembly code in the MARS editor (*Fig6.a*).

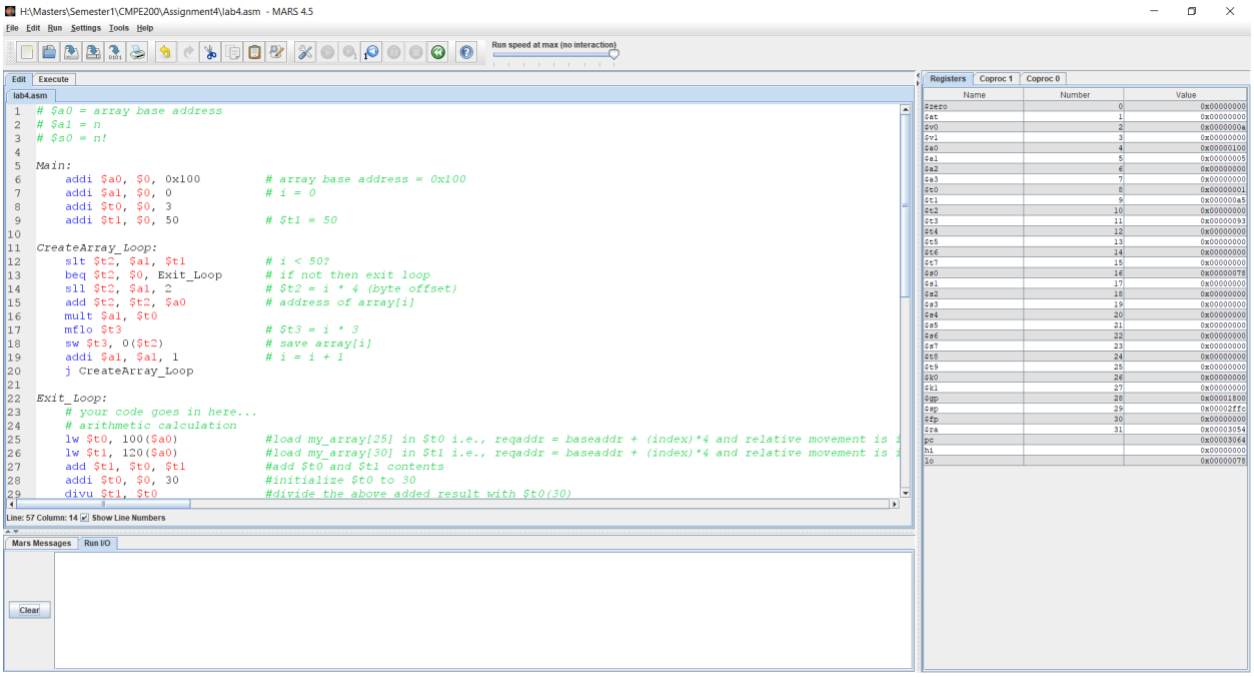


**Fig6.a – Snippet of the Final assembly code in the MARS editor**

b. Register and memory contents before execution of the program (*Fig6.b*).

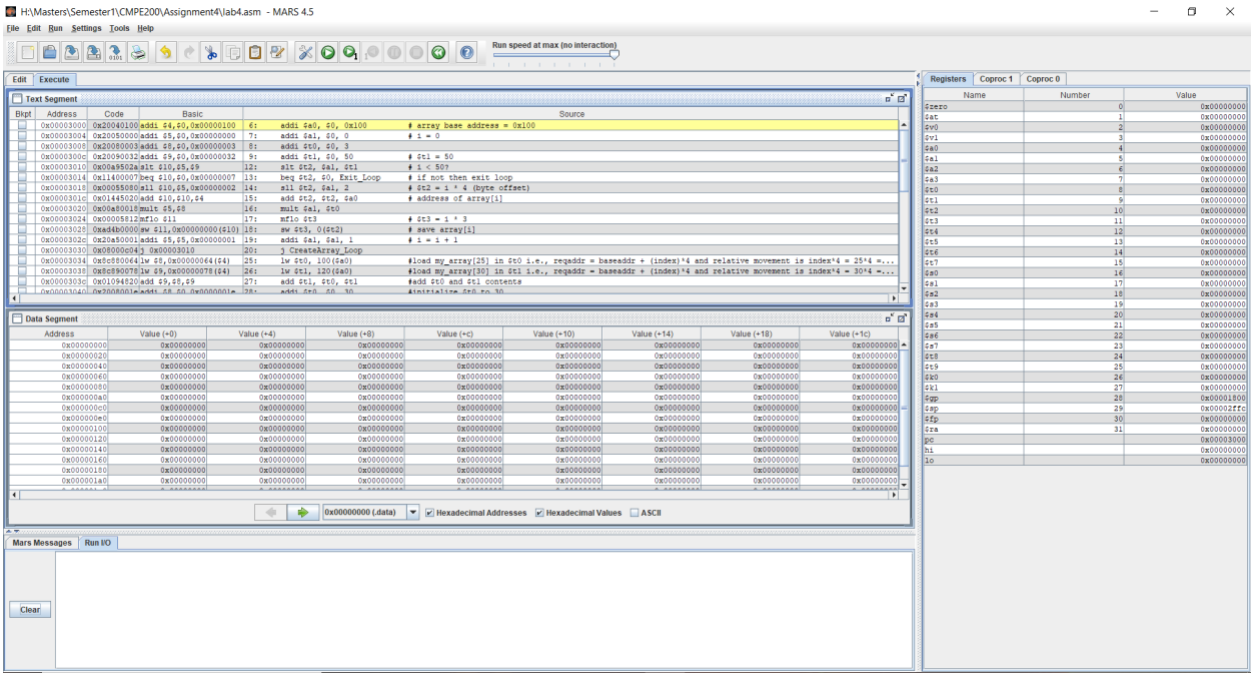


**Fig6.b – Snippet of the register and memory contents before execution of the program**

c. Register and memory contents after execution of the program (highlighted are the desired outputs in *Fig6.c*).

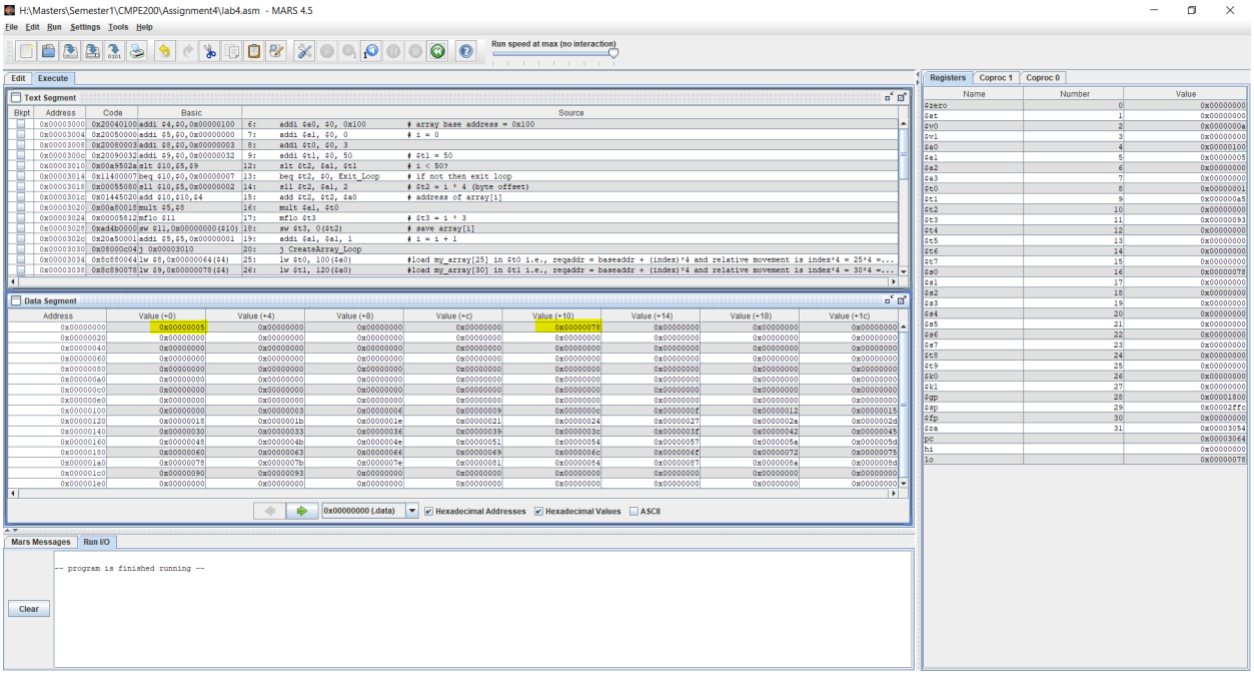


**Fig6.c – Snippet of the register and memory contents after execution of the program**

d. Stack memory contents after the execution of the program (*Fig6.d*).

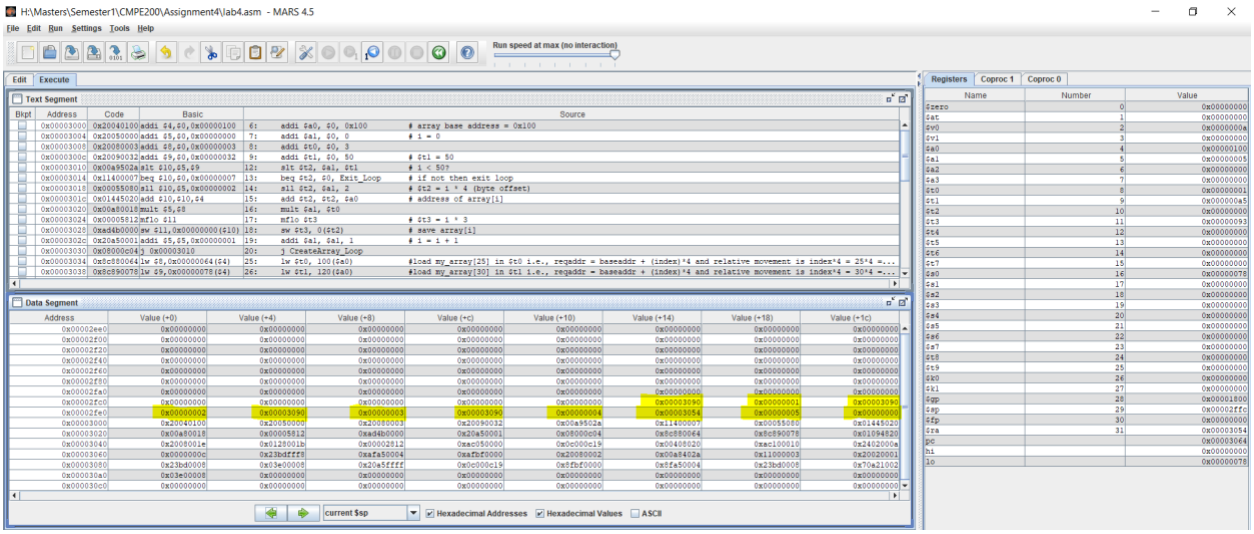


**Fig6.d – Snippet of the stack memory after execution of the program**

**DISCUSSION SECTION:**

Following is the explanation behind the arithmetic calculation of the given array elements:

a. Since we are talking about the integer arrays, each integer in the array occupies 4 bytes.

b. To access the array element at the index 25, we need to use the load word (lw) instruction by pointing to the correct index.

c. In general, the target address is calculated using the below formula

   *Target Address = Base Address + Relative value*
   *Where Relative value = (Index)*(Size of the object)*

   In our case,
   the size of the integer object is 4 bytes
   Index is 25
   Base address is 0x100

   Therefore,
   Target Address = $(0x100)_h + (25*4)_d$
   ⇨ $(0x100)_h + (0x64)_h$
   ⇨ $(0x164)_h$

d. But, while writing the lw instruction, we need to specify the increment/decrement in decimal form relative to the base address. So, we need to calculate only the relative value for the purpose of writing the code.
   i.e., relative value = 25*4 = 100
   => lw $t0, 100($a0) where $a0 contains the base address.

Following are the observations with respect to the stack when executing the recursive part of the final assembly code:

a. When n = 5, the stack memory is as follows

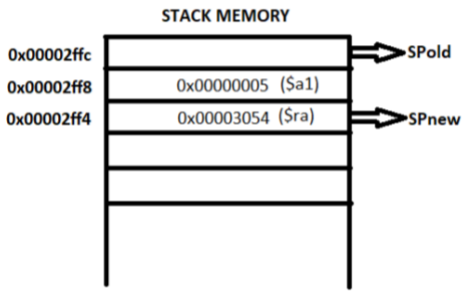


**Fig7.a – Stack memory with n=5 case**

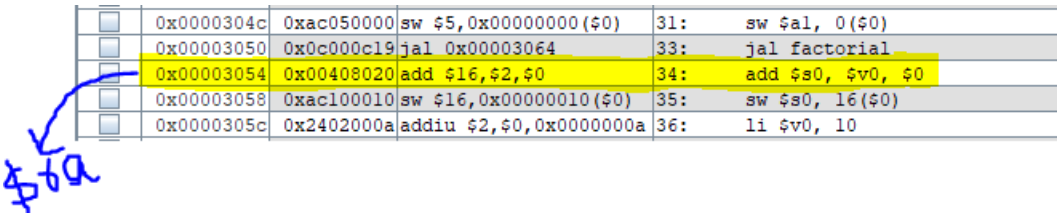


**Fig7.b – Highlighted return address is stored in the stack (n=5 case)**

Recursive functions must store their input parameters and return addresses to the stack because all the recursion iterations will try to the same registers for all operations. Hence, in the *Fig7.a*, the stack pointer is incremented down two memory locations which are used to store the input value (n) and return address ($ra). In the given task, the input value is 0x00000005 and the return address (0x00003054) is shown in the *Fig7.b*.

b. When n=4 , the stack memory is as follows

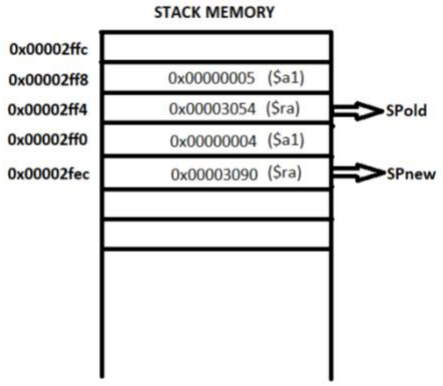


**Fig7.c – Stack memory with n=4 case**

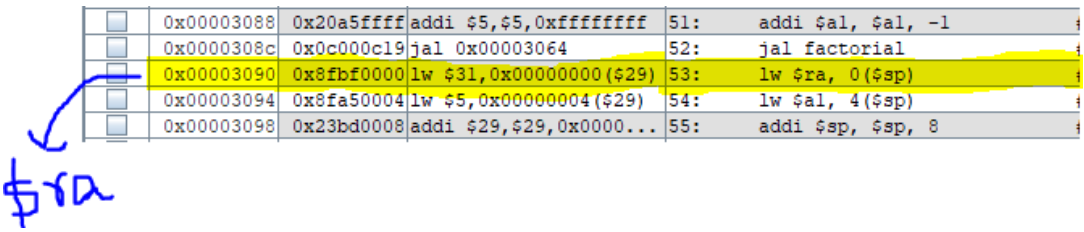


**Fig7.d – Highlighted return address is stored in the stack (n=4 case)**

In the next iteration of the recursion, the updated input parameter and the return address will be stored by again incrementing the stack pointer to create two new memory locations. As shown in the *Fig7.c*, the old stack pointer ($SP_{old}$) is incremented to the new stack pointer ($SP_{new}$) from the address 0x00002ff4 to address 0x00002fec. One of the new locations will store the updated input parameter (0x00000004) and the other location will store the return address (0x00003090) as shown in *Fig7.d*.

c. When n=3, the stack memory will be as follows

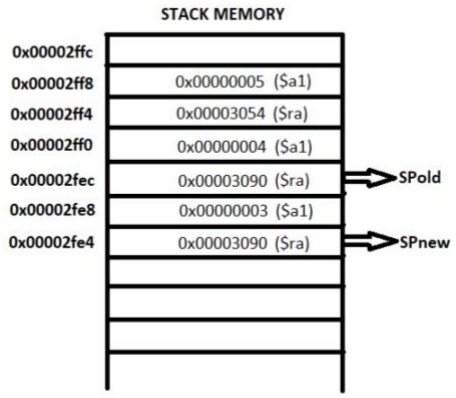


**Fig7.e – Stack memory in n=3 case**

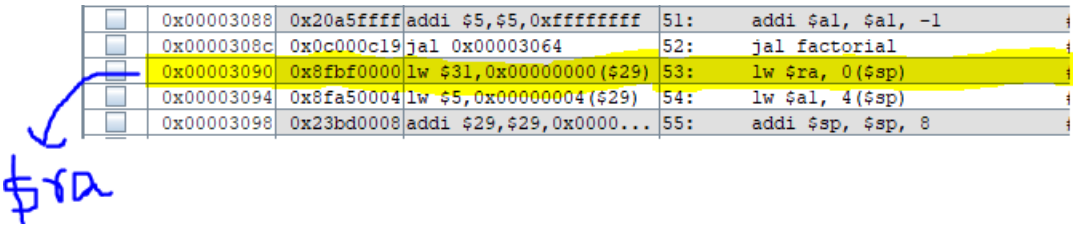


**Fig7.f – Highlighted return address is stored in the stack (n=3 case)**

In the next iteration of the recursion, the updated input parameter and the return address will be stored by again incrementing the stack pointer to create two new memory locations. As shown in the *Fig7.e*, the old stack pointer ($SP_{old}$) is incremented to the new stack pointer ($SP_{new}$) from the address 0x00002fec to address 0x00002fe4. One of the new locations will store the updated input parameter (0x00000003) and the other location will store the return address (0x00003090) as shown in *Fig7.f*.

d. When n=2, the stack memory is as follows

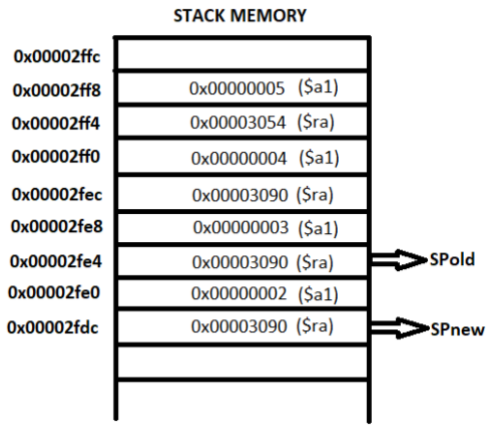


**Fig7.g – Stack memory in n=2 case**

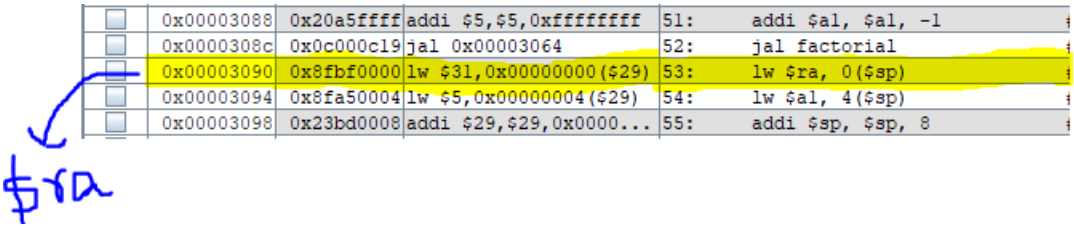


**Fig7.h – Highlighted return address is stored in the stack (n=2 case)**

In the next iteration of the recursion, the updated input parameter and the return address will be stored by again incrementing the stack pointer to create two new memory locations. As shown in the *Fig7.g*, the old stack pointer ($SP_{old}$) is incremented to the new stack pointer ($SP_{new}$) from the address 0x00002fe4 to address 0x00002fdc. One of the new locations will store the updated input parameter (0x00000002) and the other location will store the return address (0x00003090) as shown in *Fig7.h*.

e. When n=1, the stack memory is as follows

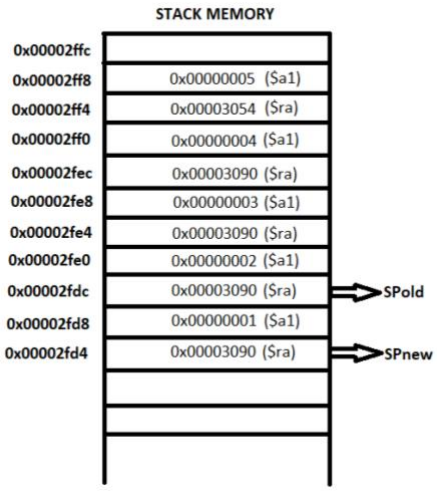


**Fig7.i – Stack memory in n=1 case**

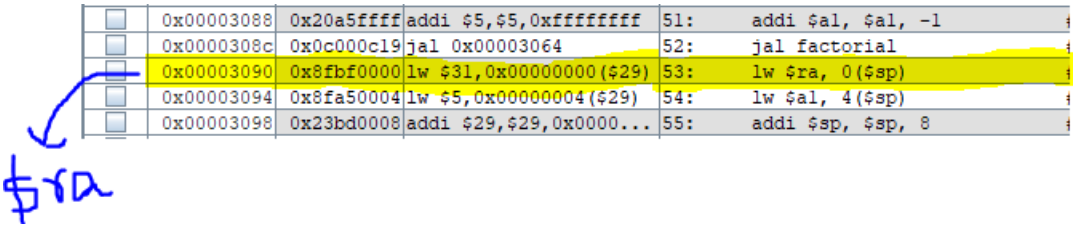


**Fig7.j – Highlighted return address is stored in the stack (n=1 case)**

In the next iteration of the recursion, the updated input parameter and the return address will be stored by again incrementing the stack pointer to create two new memory locations. As shown in the *Fig7.i*, the old stack pointer ($SP_{old}$) is incremented to the new stack pointer ($SP_{new}$) from the address 0x00002fdc to address 0x00002fd4. One of the new locations will store the updated input parameter (0x00000001) and the other location will store the return address (0x00003090) as shown in *Fig7.j*.

f. When n=0, two more locations are allocated, but due to the n>=1 condition failure, these two locations are again deallocated.
g. After the deallocation, the jr $ra instruction is executed recursively backwards where the input parameter (n) and the return address ($ra) are popped out of the stack with each iteration and the n is multiplied with the current iterative value i.e., (n-1). Similarly, all the n and $ra pairs are popped out of the stack and the deallocation of the stack is done using the stack pointer.
   Therefore, the factorial is => n*(n-1)*(n-2)*….1 stored in $v0 register.
h. Finally, the factorial value is copied into $a0 register and into the memory location 0x10 according to the given requirement.

**COLLABORATION SECTION:**

1. For the given C++ pseudo code, developed corresponding assembly codes in the MARS software by collaborating with each other.
2. Worked together to execute the code and to observe all the operations and values.
3. Collaborated in understanding the concepts such as arrays, stacks, procedures along with the recursive procedures and discussed and realized few basics of recursion, stack pointer register ($sp).
4. By working together, we debugged each line to know the contents of the relevant registers and recorded the memory values at certain addresses in the test log table.

**CONCLUSION:**

In conclusion, we end the report by understanding the arrays, accessing arrays using MIPS assembly, procedures such as function calls, nested function calls, and finally the recursive procedure by taking factorial program as an example. We also gained insight into the concept of jump and link (jal) and jump return (jr) instructions along with the usage of the stack pointer register ($sp).

**APPENDIX:**

The source code of the given task is as follows

```
# $a0 = array base address
# $a1 = n
# $s0 = n!

Main:
   addi $a0, $0, 0x100        # array base address = 0x100
   addi $a1, $0, 0            # i = 0
   addi $t0, $0, 3            #initialize $t0 to 3
   addi $t1, $0, 50           # $t1 = 50

CreateArray_Loop:
   slt $t2, $a1, $t1          # i < 50?
   beq $t2, $0, Exit_Loop     # if not then exit loop
   sll $t2, $a1, 2            # $t2 = i * 4 (byte offset)
   add $t2, $t2, $a0          # address of array[i]
   mult $a1, $t0
   mflo $t3                   # $t3 = i * 3
   sw $t3, 0($t2)             # save array[i]
   addi $a1, $a1, 1           # i = i + 1
   j CreateArray_Loop

Exit_Loop:
   # your code goes in here...
   # arithmetic calculation
   lw $t0, 100($a0)           #load my_array[25] in $t0 i.e., reqaddr = baseaddr + (index)*4
                              and relative movement is index*4 = 25*4 = 100
   lw $t1, 120($a0)           #load my_array[30] in $t1 i.e., reqaddr = baseaddr + (index)*4
                              and relative movement is index*4 = 30*4 = 120
   add $t1, $t0, $t1          #add $t0 and $t1 contents
   addi $t0, $0, 30           #initialize $t0 to 30
   divu $t1, $t0              #divide the above added result with $t0(30)
   mflo $a1                   #capture the quotient from $lo to $a1
   sw $a1, 0($0)              #store the content of $a1 (n) at 0x00 location
   # factorial computation
   jal factorial              # call procedure
   add $s0, $v0, $0           # return value
   sw $s0, 16($0)             #store the content of $s0 (n!) at 0x10 location
   li $v0, 10                 #load the $v0 register with value 10
   syscall                    #call the syscall to exit the program
```

```
factorial:
   addi $sp, $sp, -8          # make room on the stack
   sw $a1, 4($sp)             # store $a1
   sw $ra, 0($sp)             # store $ra
   # your code goes in here
   addi $t0, $0, 2            #initialize $t0 to 2
   slt $t0, $a1, $t0          #check for n<2 => n<=1
   beq $t0, $0, else          #branch to label 'else' if above condition is false
   addi $v0, $0, 1            #initialize $v0 to 1
   addi $sp, $sp, 8           #deallocate the stack
   jr $ra                     #jump to the caller using $ra
else:
   addi $a1, $a1, -1          #decrement the n value
   jal factorial              #recursive call to the label 'factorial'
   lw $ra, 0($sp)             #recursively load back the return address from stack
   lw $a1, 4($sp)             #recursively load back the n value from stack
   addi $sp, $sp, 8           #recursively deallocate the stack memory
   mul $v0, $a1, $v0          #recursively multiply the n value to its previous value i.e.,(n-1)
   jr $ra                     #jump to the caller using $ra
```