

San Jose State University
Department of Computer Engineering

CMPE 140 Lab Report

Lab 4 Report

Title MIPS Instruction Set Architecture & Programming (3)

Semester Spring 2019

Date 03/06/19

by



Name Nickolas Schiffer

SID 012279319

Name Salvatore Nicosia

SID 012013599

Lab Checkup Record

Week	Performed By (signature)	Checked By (signature)	Tasks Successfully Completed*	Tasks Partially Completed*	Tasks Failed or Not Performed*
1	 SN		100%		

*** Detailed descriptions must be given in the report.**

I. INTRODUCTION

The purpose of this lab is to familiarize ourselves with the MIPS implementation of arrays, stacks, procedures, and recursive procedures. A MIPS assembly program was created to use a 50-entry array to perform arithmetic calculations and use the results as input arguments for the factorial function.

II. TESTING PROCEDURE

The MIPS assembly program for the factorial function was written with the provided C++ pseudo code shown in Figure 1.

```
void main()
{
    int n, f;
    int my_array[50];
    // Create the array
    for(i=0; i<50; i=i+1)
    {
        my_array[i] = i*3;
    }
    /*You will write MIPS code for the following parts*/
    // Arithmetic calculation
    n = (my_array[25]+ my_array[30])/30;
    // Factorial
    f = Factorial(n);
    return;
}

// Recursive factorial procedure
int Factorial(int n)
{
    if (n <= 1)
        return 1;
    else
        return (n*Factorial(n-1));
}
```

Figure 1. C++ pseudo code

The registers \$a1 and \$s0 were used to hold the value of n and n! respectively. First, MIPS assembly code is written to build a 50-entry array with the base address 0x100 stored in \$a0. This array was used to perform arithmetic calculations to calculate the value of n. This result was then used in the recursive factorial procedure which calculates the factorial of n. The code for this program file named *recursive_factorial.smd* can be found in the appendix. The MIPS assembly code was then assembled and executed to verify the correct behavior of each instruction and verify the contents of the relevant registers. After verifying the correct behavior of the algorithm, the execution results and the values of the memory addresses 0x00, and 0x10 were recorded in the test log table (see *Table 1*) in the testing result section. In addition, a stack status diagram (see *Table 2*) was created to show the values of \$a1 and \$ra for each recursive call and where \$sp (stack pointer) was pointing after each call.

III. TESTING RESULTS

The values recorded in *Table 1* for the factorial computation show the content of the registers \$a1, \$sp, \$ra, and \$v0 after the execution of each instruction and the relative content at memory address 0x00 (n) and 0x01 (n!). In addition, *Table 2* represents the stack status diagram showing where \$sp is pointing after each recursive call.

Table 1. Test Log

Addr	MIPS Instruction	Machine Code	Registers				Memory Content	
			\$a1	\$sp	\$ra	\$v0	[0x00]	[0x10]
34	# lw \$t1, 100(\$a0) (\$t1 = mem[\$a0 + 100])	0x8C890064	0x32	0x200	0	0	0	0
38	# lw \$t2, 120(\$a0) (\$t2 = mem[\$a0 + 120])	0x8C8A0078	0x32	0x200	0	0	0	0
3c	# addu \$t1, \$t1, \$t2 (\$t1 = \$t1 + \$t2)	0x012A4821	0x32	0x200	0	0	0	0
40	# addiu \$t2, \$zero, 30 (\$t2 = 30)	0x240A001E	0x32	0x200	0	0	0	0
44	# divu \$t1, \$t2 (\$Hi = \$t1 div \$t2; \$Lo = \$t1 mod \$t2)	0x012A001B	0x32	0x200	0	0	0	0
48	# mflo \$a1 (\$a1 = \$lo)	0x00002812	0x32	0x200	0	0	0	0
4c	# sw \$a1, 0(\$zero) (mem[\$zero + 0] = \$a1)	0xAC050000	0x5	0x200	0	0	0	0
50	# jal 0x0017 (jump & link to addr 0x005C)	0x0C000017	0x5	0x200	0x54	0	0x5	0
54	# add \$s0, \$v0, \$zero (\$s0 = \$v0)	0x00408020	0x5	0x200	0x54	0x78	0x5	0
58	# sw \$s0, 16(\$zero) (mem[\$zero + 16] = \$s0)	0xAC100010	0x5	0x200	0x54	0x78	0x5	0x78
5c	# j 0x000	0x08000000	0x5	0x200	0x54	0x78	0x5	0x78
60	# addi \$sp, \$sp, -8 (\$sp = \$sp + -8)	0x23BDFF8	0x1	0x1D8	0x88	0	0x5	0
64	# sw \$a1, 4(\$sp) (mem[\$sp + 4] = \$a1)	0xAFA50004	0x1	0x1D8	0x88	0	0x5	0
68	# sw \$ra, 0(\$sp) (mem[\$sp + 0] = \$ra)	0xAFBF0000	0x1	0x1D8	0x88	0	0x5	0
6c	# slti \$RD, \$a1, 2 (if (\$a1 < 2) #RD = 1 else #RD = 0)	0x28A80002	0x1	0x1D8	0x88	0	0x5	0
70	# beq \$zero, \$t0, 3 (if (\$zero == \$t0) goto 3)	0x10080003	0x1	0x1D8	0x88	0	0x5	0
74	# addi \$v0, \$zero, 1 (\$v0 = 1)	0x20020001	0x1	0x1D8	0x88	0x1	0x5	0
78	# addi \$sp, \$sp, 8 (\$sp = \$sp + 8)	0x23BD0008	0x1	0x1E0	0x88	0x1	0x5	0
7c	# jr \$ra (jump \$ra)	0x03E00008	0x1	0x1E0	0x88	0x1	0x5	0
80	# addi \$a1, \$a1, -1 (\$a1 = \$a1 + -1)	0x20A5FFFF	0x1	0x1E0	0x88	0	0x5	0
84	# jal 0x0017 (jump & link to addr 0x005C)	0x0C000017	0x1	0x1E0	0x88	0	0x5	0
88	# lw \$ra, 0(\$sp) (\$ra = mem[\$sp + 0])	0x8FBF0000	0x5	0x1F8	0x54	0x18	0x5	0
8c	# lw \$a1, 4(\$sp) (\$a1 = mem[\$sp + 4])	0x8FA50004	0x5	0x1F8	0x54	0x18	0x5	0
90	# addi \$sp, \$sp, 8 (\$sp = \$sp + 8)	0x23BD0008	0x5	0x1F8	0x54	0x18	0x5	0
94	# mult \$v0, \$a1 (\$Hi = High(\$v0 * \$a1); \$Lo = Low(\$v0 * \$a1))	0x00450018	0x5	0x200	0x54	0x18	0x5	0
98	# mflo \$v0 (\$v0 = \$lo)	0x00001012	0x5	0x200	0x54	0x78	0x5	0
9c	# jr \$ra (jump \$ra)	0x03E00008	0x5	0x200	0x54	0x78	0x5	0
100								
104								

Table 2. Stack Diagram

Stack during Recursive Call (for 5!)		
Address	Data	
0x1FC	\$a1 (0x5)	
0x1F8	\$ra (0x54)	<- \$sp after the 1st recursive call
0x1F4	\$a1 (0x4)	
0x1F0	\$ra (0x84)	<- \$sp after the 2nd recursive call
0x1EC	\$a1 (0x3)	
0x1E8	\$ra (0x84)	<- \$sp after the 3rd recursive call
0x1E4	\$a1 (0x2)	
0x1E0	\$ra (0x84)	<- \$sp after the 4th recursive call
0x1DC	\$a1 (0x1)	
0x1D8	\$ra (0x84)	<- \$sp after the 5th recursive call

Figures 1, 2, 3, and 4 show the execution of each instructions from the creation of the array to calculation of the factorial and the result stored in memory.

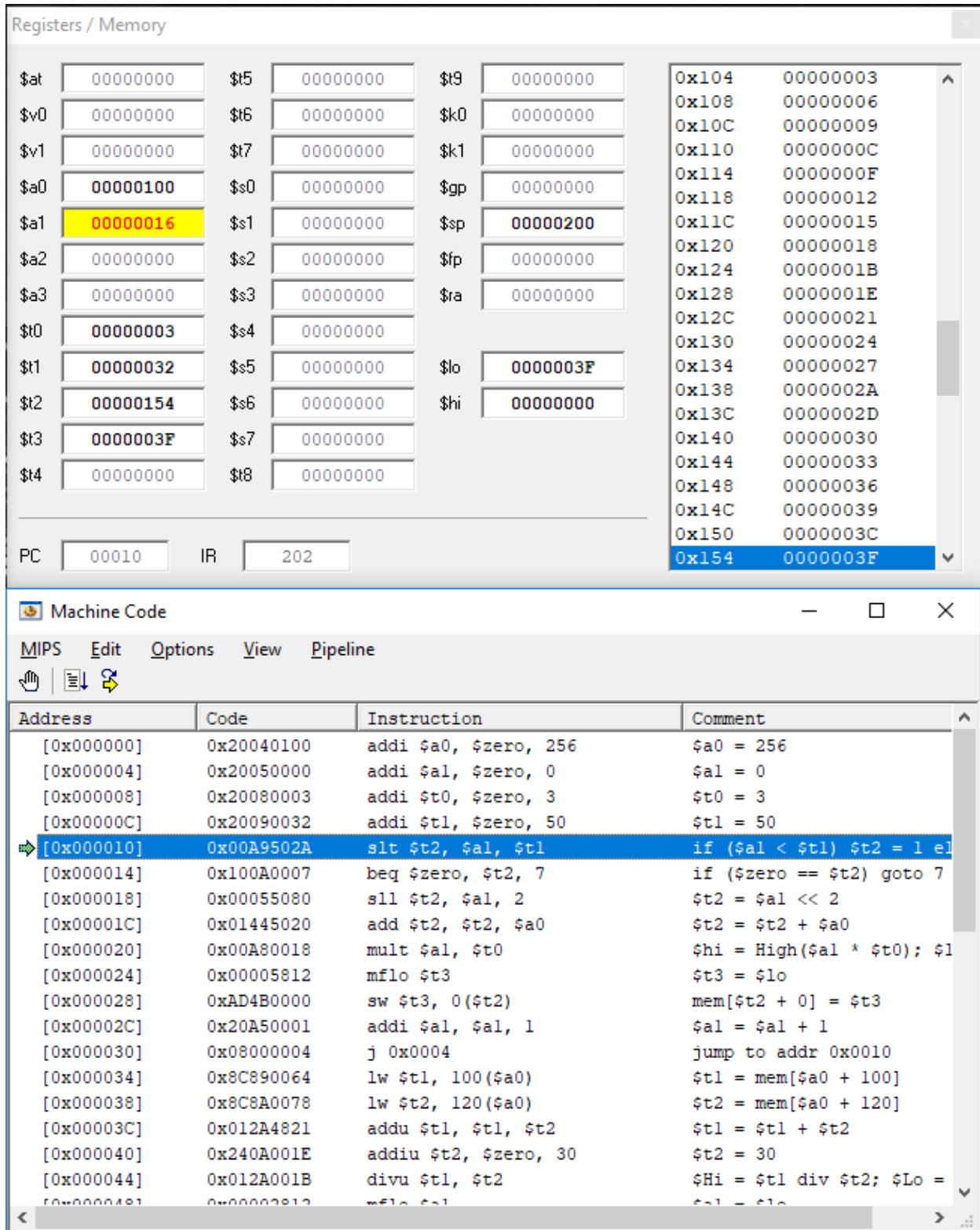


Figure 1: Execution result showing the array getting created in memory.

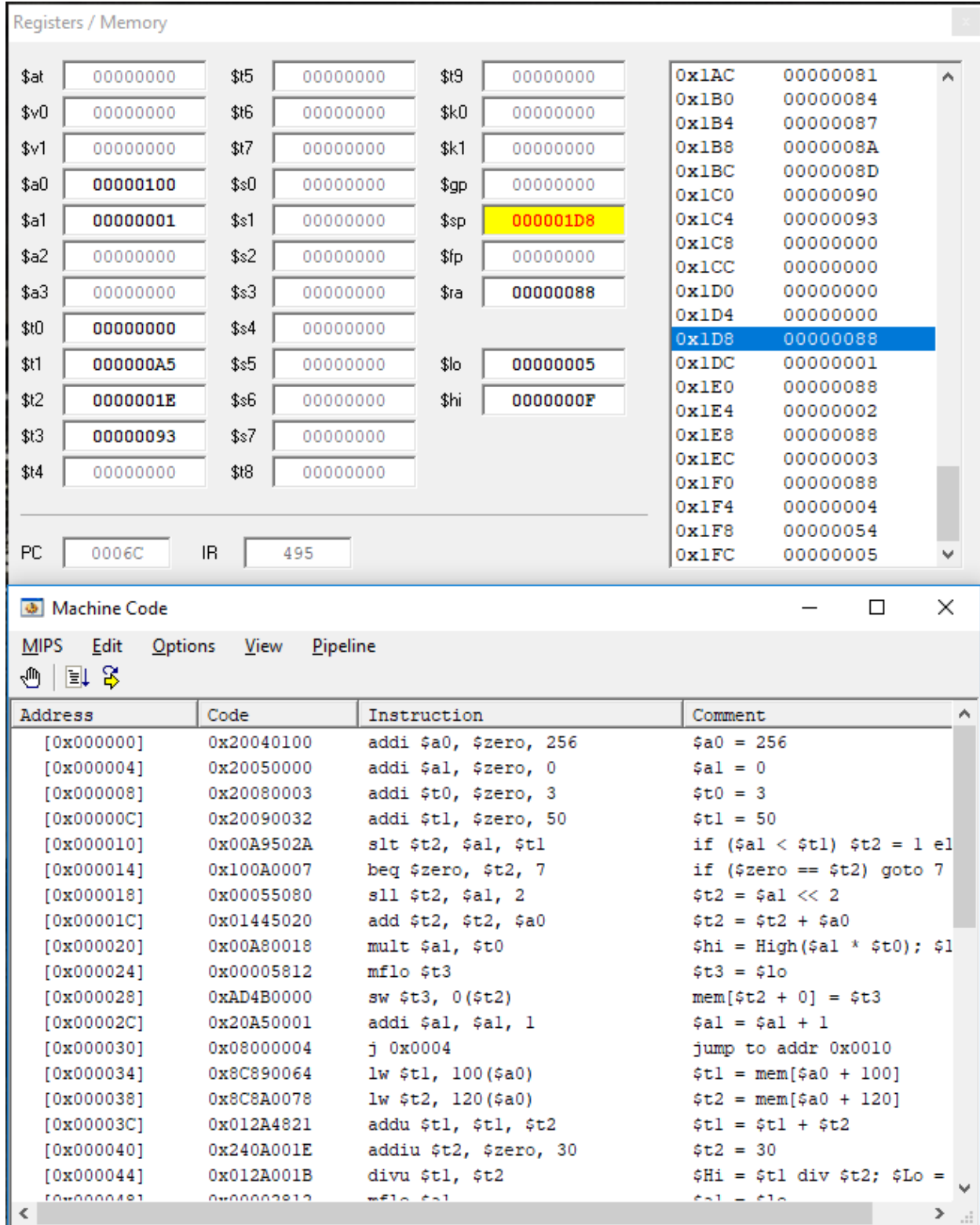


Figure 2: Execution result showing the stack and the values of \$a1 and \$s0 in memory as well as where \$sp is pointing.

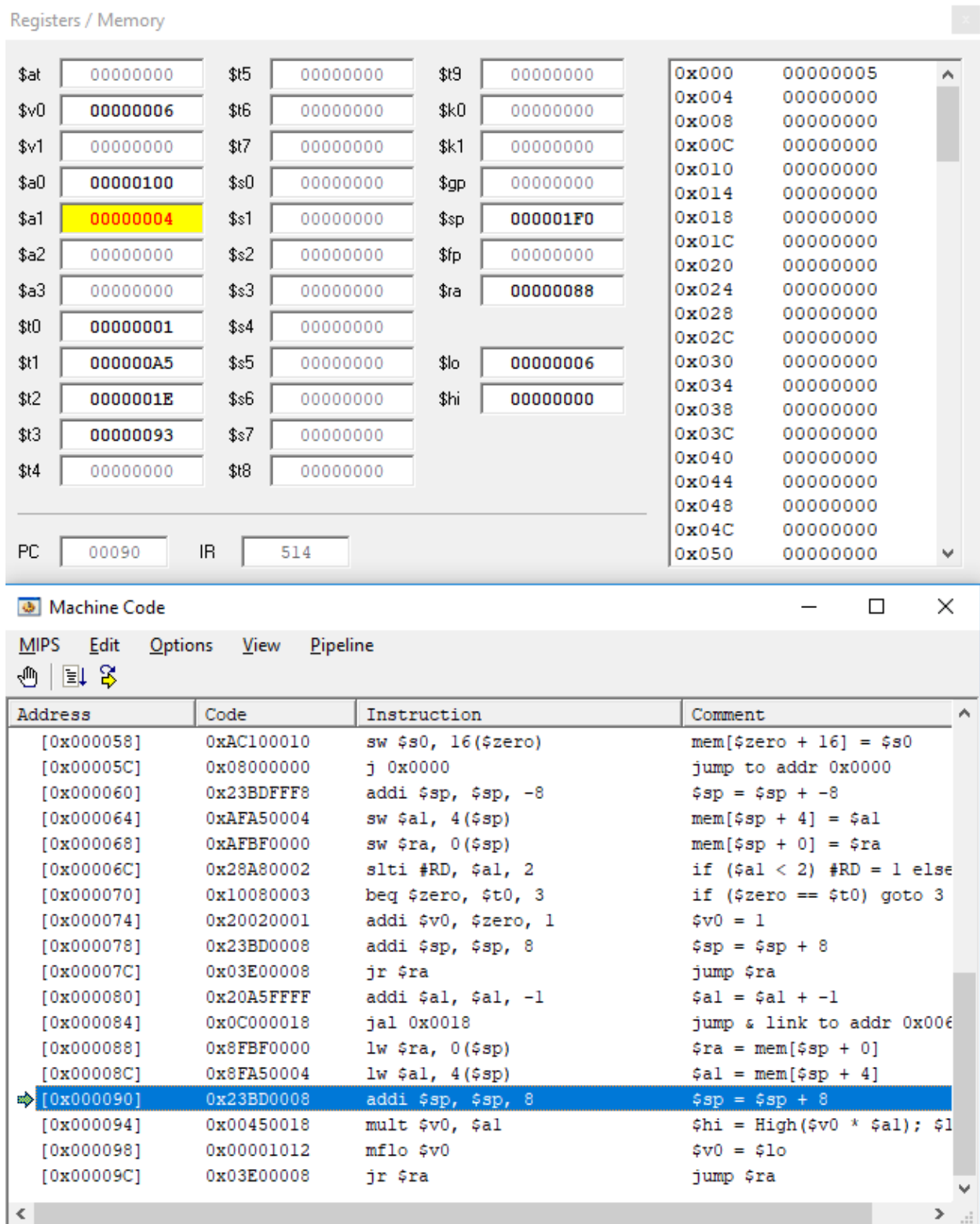


Figure 3. Execution result showing the values getting used from the stack to calculate the factorial of 5 as well as the value of $n = 5$ in memory at address 0x000.

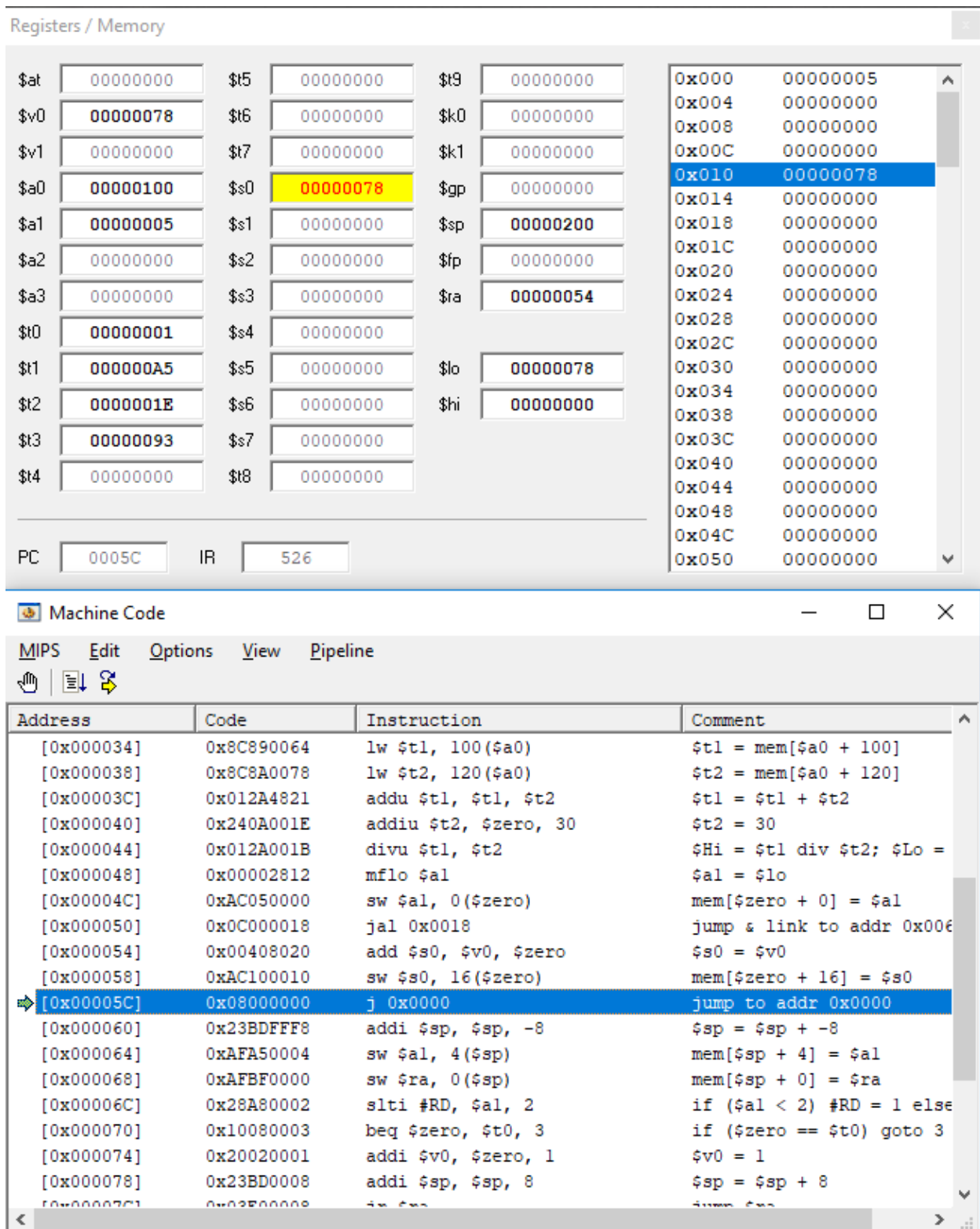


Figure 4. Execution result showing the value of the factorial of 5 getting stored in memory at address 0x10 with value 78.

IV. CONCLUSION

This lab further solidified how to use the MIPS instruction set to implement algorithms using arrays, stacks, procedures, and recursive procedures. It also illustrated how the stack is created in memory and how the stack pointer changes with each recursive call.

V. SUCCESSFUL TASKS

1. Implement the recursive factorial with the MIPS instruction set.
2. Generate the correct stack after each recursive call.
3. Verification of the register's content and memory value for each execution of the MIPS instructions.
4. Recorded execution results in the test log table.
5. Sketch of stack status diagram.

VI. APPENDIX

A. SOURCE CODE:

<i>recursive_factorial.smd</i>	
# \$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	
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:	
# arithmetic calculation	
lw \$t1, 100(\$a0)	# t1 = my_array[25]
lw \$t2, 120(\$a0)	# t2 = my_array[30]
addu \$t1, \$t1, \$t2	# t1 = my_array[25] + my_array[30]
addiu \$t2, \$zero, 30	
divu \$t1, \$t2	# (my_array[25] + my_array[30])/30
mflo \$a1	# a1 = (my_array[25] + my_array[30])/30
sw \$a1, 0(\$zero)	# store n at 0x00
jal factorial	# call procedure
add \$s0, \$v0, \$0	# return value
sw \$s0, 0x10(\$zero)	# store n! in 0x10
j Main	# reset
factorial:	


```

# factorial computation
addi $sp, $sp, -8      # make room on the stack
sw   $a1, 4($sp)       # store $a1
sw   $ra, 0($sp)       # store $ra
slti $t0, $a1, 2       # a1 <= 1 ?
beq  $t0, $zero, else  # no: go to else (recursion)
addi $v0, $zero, 1     # yes: return 1
addi $sp, $sp, 8       # restore sp
jr   $ra               # return
else:
    addi $a1, $a1, -1   # n -= 1
    jal  factorial      # recursive call
    lw   $ra, 0($sp)    # restore $ra
    lw   $a1, 4($sp)    # restore $a1
    addi $sp, $sp, 8     # restore $sp
    mult $v0, $a1        # n * factorial(n-1)
    mflo $v0
    jr   $ra            # return

```