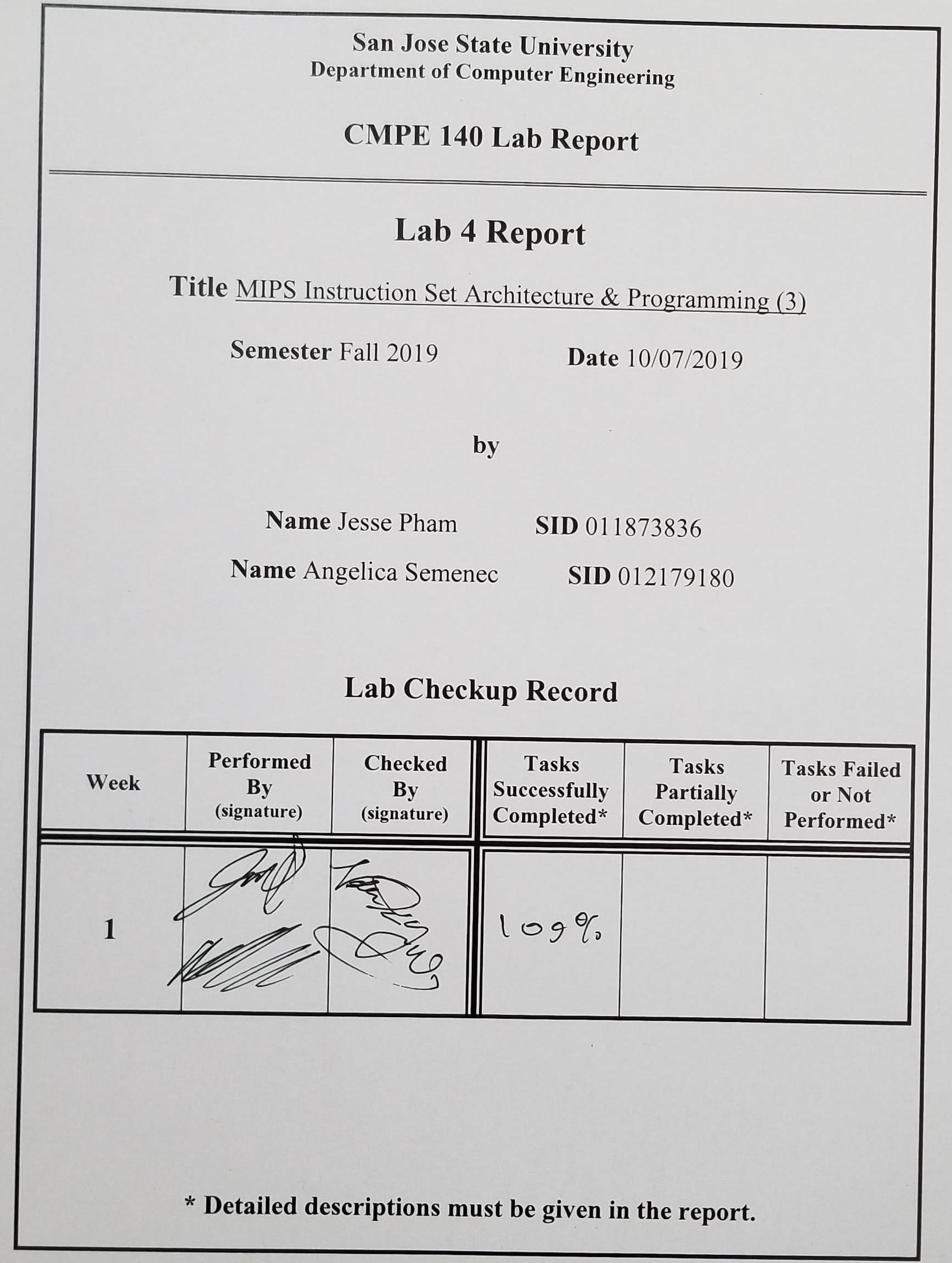
****

**Purpose**

The purpose of this lab experiment is to allow students to familiarize themselves with implementing arrays, stacks, and recursive procedures using MIPS ISA and assembly programming. By single-step executing through the instructions, the experiment will help students visualize and better understand the flow of the program.

**Design Methodology**

The code to be translated from C++ into the MIPS instructions was provided along with the code to set up the variables in main, create the array in memory, and the beginning instructions to adjust the stack pointer during the initial function call. The program to be created included a main function that called a recursive factorial function. To implement this in the MIPS language, jump and branch instructions were required for function execution as well as manipulation of the stack pointer. The function would recursively call itself, and so the return address of the function needed to be stored on the stack as well as the input parameter. When the function reached its end, the program would then return up the stack, popping off the return address and stored input value to be used in the factorial calculation. The input to the function as well as the resulting output were stored to the specified addresses in memory.

**Tasks Successfully Accomplished**

1. Properly implemented the factorial function as a recursive procedure
2. Wrote the final value of *n* to Memory Address 0x00
3. Wrote the final value of *n!* to Memory Address 0x10
4. Created a stack status diagram showing the addresses, stack pointer position, and values of $a1 and $ra
5. Recorded the execution results onto the Test Log Table

**Source Code**

|  |
| --- |
| **assignment4.asm** |
| # $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:  # your code goes in here...  # arithmetic calculation  addi $t0, $0, 25  sll $t0, $t0, 2  add $t0, $t0, $a0  lw $t1, 0($t0)  addi $t0, $0, 30  sll $t0, $t0, 2  add $t0, $t0, $a0  lw $t2, 0($t0)  add $t0, $t1, $t2  divu $t0, $t0, 30  mflo $a1  sw $a1, 0x0  # factorial computation  jal factorial # call procedure  add $s0, $v0, $0 # return value  j done  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  slti $t0, $a1, 2 #if n <= 1  beq $t0, $0, else #When greater than 1, got to else  addi $v0, $0, 1 #Return value = 1  addi $sp, $sp, 8 #restore stack pointer  jr $ra  else: addi $a1, $a1, -1 #n = n - 1  jal factorial #Call factorial  lw $ra, 0($sp) #load return address  lw $a1, 4($sp) #load n  addi $sp, $sp, 8 #Restore stack pointer  mul $v0, $v0, $a1  jr $ra  done: sw $s0, 0x10 |

**Stack Status Diagram**

Table 1: Showing the status of the stack pointer as well as the memory contents of the return address.

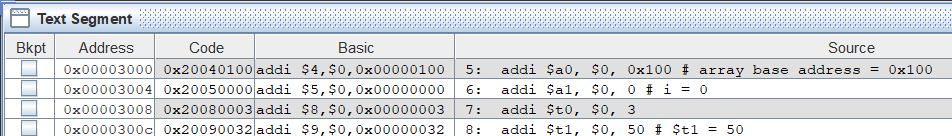
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | **Address** | **Memory Contents** | **Stack** | | **0x2ffc** | **0x0** | **← $sp0** | | **0x2ff8** | **$a1 = 5** |  | | **0x2ff4** | **$ra = 3070** | **← $sp1** | | **0x2ff0** | **$a1 = 4** |  | | **0x2fec** | **$ra = 30a0** | **← $sp2** | | **0x2fe8** | **$a1 = 3** |  | | **0x2fe4** | **$ra = 30a0** | **← $sp3** | | **0x2fe0** | **$a1 = 2** |  | | **0x2fdc** | **$ra = 30a0** | **← $sp4** | | **0x2fd8** | **$a1 = 1** |  | | **0x2fd4** | **$ra = 30a0** | **← $sp5** | | |  |  |  | | --- | --- | --- | | **Address** | **Memory Contents** | **Stack** | | **0x2ffc** | **0x0** | **← $sp10** | | **0x2ff8** | **$a1 = 5** |  | | **0x2ff4** | **$ra = 3070** | **← $sp9** | | **0x2ff0** | **$a1 = 4** |  | | **0x2fec** | **$ra = 30a0** | **← $sp8** | | **0x2fe8** | **$a1 = 3** |  | | **0x2fe4** | **$ra = 30a0** | **← $sp7** | | **0x2fe0** | **$a1 = 2** |  | | **0x2fdc** | **$ra = 30a0** | **← $sp6** | | **0x2fd8** | **$a1 = 1** |  | | **0x2fd4** | **$ra = 30a0** |  | |

**Test Logs**

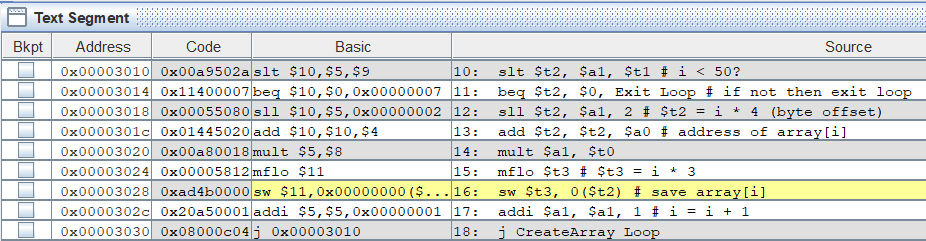
Table 2: Displaying the final values of each Register after executing each MIPS Instruction. The final value is loaded into memory address location 0x10 at the end of the program.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Addr | MIPS Instruction | Machine Code | Registers | | | | Memory Content | |
| $a1 | $sp | $ra | $v0 | [0x00] | [0x10] |
| 3034 | addi $8, $0, 0x00000019 | 0x20080019 | 32 | 2ffc | 0 | 0 | 0 | 0 |
| 3038 | sll $8, $8, 2 | 0x00084080 | 32 | 2ffc | 0 | 0 | 0 | 0 |
| 303c | add $8, $8, $4 | 0x01044020 | 32 | 2ffc | 0 | 0 | 0 | 0 |
| 3040 | lw $9, 0x0($8) | 0x8d090000 | 32 | 2ffc | 0 | 0 | 0 | 0 |
| 3044 | addi $8, $0, 0x1e | 0x2008001e | 32 | 2ffc | 0 | 0 | 0 | 0 |
| 3048 | sll $8, $8, 0x2 | 0x00084080 | 32 | 2ffc | 0 | 0 | 0 | 0 |
| 304c | add $8, $8, $4 | 0x01044020 | 32 | 2ffc | 0 | 0 | 0 | 0 |
| 3050 | lw $10, 0x0($8) | 0x8d0a0000 | 32 | 2ffc | 0 | 0 | 0 | 0 |
| 3054 | add $8, $9, $10 | 0x012a4020 | 32 | 2ffc | 0 | 0 | 0 | 0 |
| 3058 | addi $1, $0, 0x1e | 0x2001001e | 32 | 2ffc | 0 | 0 | 0 | 0 |
| 305c | divu $8, $1 | 0x0101001b | 32 | 2ffc | 0 | 0 | 0 | 0 |
| 3060 | mflo $8 | 0x00004012 | 32 | 2ffc | 0 | 0 | 0 | 0 |
| 3064 | mflo $5 | 0x00002812 | 5 | 2ffc | 0 | 0 | 0 | 0 |
| 3068 | sw $5, 0x0($0) | 0xac050000 | 5 | 2ffc | 0 | 0 | 5 | 0 |
| 306c | jal 0x3078 | 0x0c000c1e | 5 | 2ffc | 3070 | 0 | 5 | 0 |
| 3070 | add $16, $2, $0 | 0x408020 | 5 | 2ffc | 3070 | 78 | 5 | 0 |
| 3074 | j 0x30b4 | 0x8000c2d | 5 | 2ffc | 3070 | 78 | 5 | 0 |
| 3078 | addi $29, $29, 0xfffffff8 | 0x23bdfff8 | 1 | 2fd4 | 30a0 | 0 | 5 | 0 |
| 307c | sw $5, 0x00000004($29) | 0xafa50004 | 1 | 2fd4 | 30a0 | 0 | 5 | 0 |
| 3080 | sw $31, 0x00000000($29) | 0xafbf0000 | 1 | 2fd4 | 30a0 | 0 | 5 | 0 |
| 3084 | slti $8, $5, 0x00000002 | 0x28a80002 | 1 | 2fd4 | 30a0 | 0 | 5 | 0 |
| 3088 | beq $8, $0, 0x00000003 | 0x11000003 | 1 | 2fd4 | 30a0 | 0 | 5 | 0 |
| 308c | addi $2, $0, 0x00000001 | 0x20020001 | 1 | 2fd4 | 30a0 | 1 | 5 | 0 |
| 3090 | addi $29, $29, 0x00000008 | 0x23bd0008 | 1 | 2fdc | 30a0 | 1 | 5 | 0 |
| 3094 | jr $31 | 0x03e00008 | 1 | 2fdc | 30a0 | 1 | 5 | 0 |
| 3098 | addi $5, $5, 0xffffffff | 0x20a5ffff | 1 | 2fdc | 30a0 | 1 | 5 | 0 |
| 309c | jal 0x00003078 | 0x0c000c1e | 1 | 2fdc | 30a0 | 1 | 5 | 0 |
| 30a0 | lw $31, 0x0($29) | 0x8fbf0000 | 4 | 2ff4 | 3070 | 1 | 5 | 0 |
| 30a4 | lw $5, 0x4($29) | 0x8fa50004 | 5 | 2ff4 | 30a0 | 1 | 5 | 0 |
| 30a8 | addi $29, $29, 0x8 | 0x23bd0008 | 5 | 2fe4 | 30a0 | 1 | 5 | 0 |
| 30ac | mul $2, $2, $5 | 0x70451002 | 5 | 2ffc | 3070 | 78 | 5 | 0 |
| 30b0 | jr $31 | 0x3e00008 | 5 | 2ffc | 3070 | 78 | 5 | 0 |
| 30b4 | sw $16, 0x10($0) | 0xac100010 | 5 | 2ffc | 3070 | 78 | 5 | 78 |

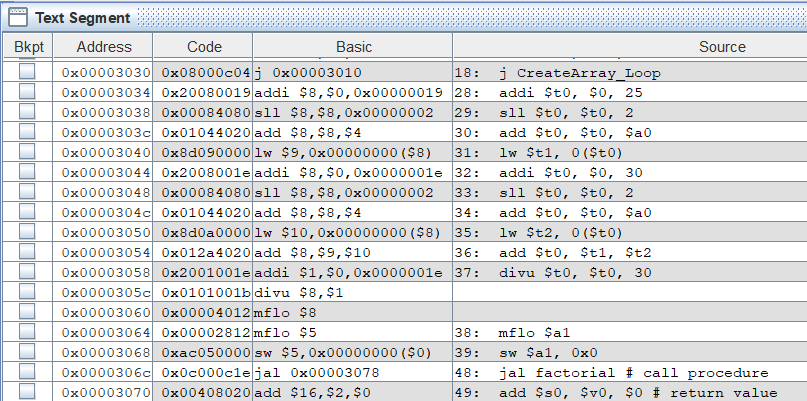
**Screen Captures**

**

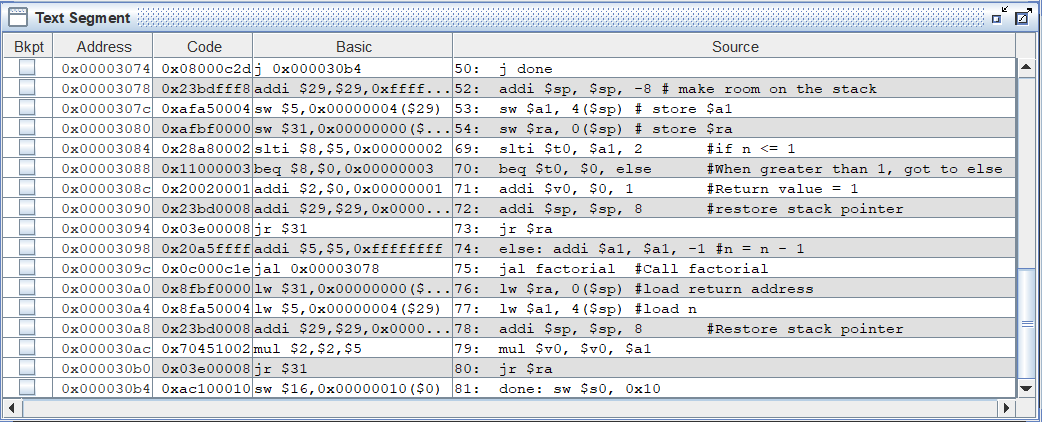
*Figure 1: Displaying the Text Segment for the Main function. This was given by the lab manual and it sets up the initial base address, the initial i value, and the value of the test variable.*

**

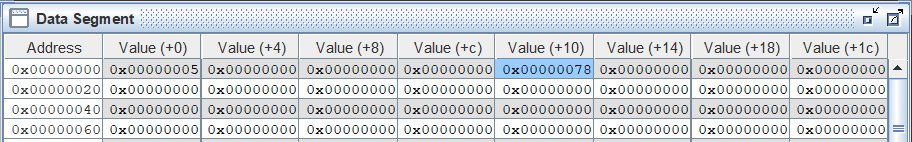
*Figure 2: Displaying the Text Segment for the CreateArray\_Loop function. This was also given by the lab manual. This function initializes the array from 0 to 49.*

**

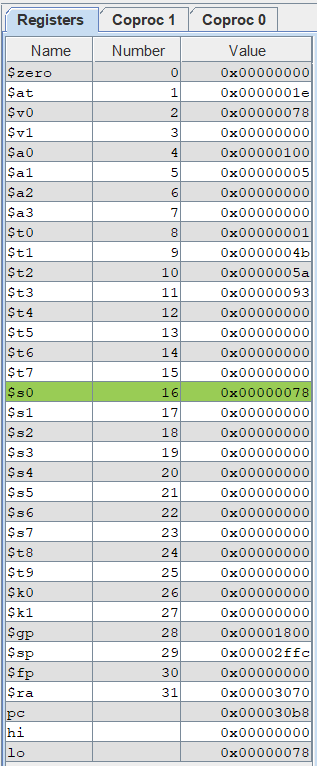
*Figure 3: Displaying the Text Segment for the Exit\_Loop function implementation.*

**

*Figure 4: Displaying the Text Segment for the Factorial function implementation. The initial Prologue of the function was provided in the lab manual. The Factorial function is recursive and the Epilogue is shown towards the end of the function through the use of the Load Word (lw) instruction.*

**

*Figure 5: Displaying the Data Segment after the MIPS program has run entirely through. The final output for the input value of 5! is stored in memory address location 0x10, which contains the value 0x78 (12010).*

**

*Figure 6: Displays the Register contents after the MIPS program has run through entirely.*

**Discussion/Conclusion**

The experimental results were successful. Although we took extra steps to load our registers from memory by using the *Shift Logical Left (sll)* and addition instructions to obtain the correct memory address, our result output was correct. It has been noted that a more efficient approach would be to directly use the *Load Word (lw)* command’s offset functionality to load from the desired memory address location rather than using commands *sll* and *addi* to obtain the memory address and then *lw* to load the contents to the register.

When the program first enters the *factorial* function, it first executes the prologue of the program, which includes the instructions for adjusting the stack pointer so that the return address and function parameters can be stored to the stack memory. This prevents the registers from being overwritten if another function call occurs. When the function reaches its end, it enters into the epilogue. The epilogue includes code that loads the appropriate values from the stack into their respect registers, which was the return address register and the function parameter registers. The stack pointer is then incremented so that the data that was previously stored in the function prologue is effectively popped off the stack. This, however, does not clear the data from the stack. The data that was loaded into the stack memory stays in memory rather than being reset. When a function call occurs again in the program, any existing data on the stack that is no longer in use will be overwritten by the new function call.

When the program first calls the *factorial* function, the jump and link (jal) instruction updates the program counter to the next address and stores that to the return address register before jumping to the function label. The initial function call places the return address 0x3070 on the stack, which is the address of the instruction directly after the initial function call. The *factorial* function is implemented recursively, and so there are multiple function calls within the function itself. These function calls store the return address 0x30a0 onto the stack for each recursive function call. This address returns the program to the instruction directly after the recursive *factorial* call so that the final value of the factorial can be calculated. When the recursive function calls have finished, the first return address, 0x3070, is loaded from the stack and places the program back to the line directly after the initial function call.