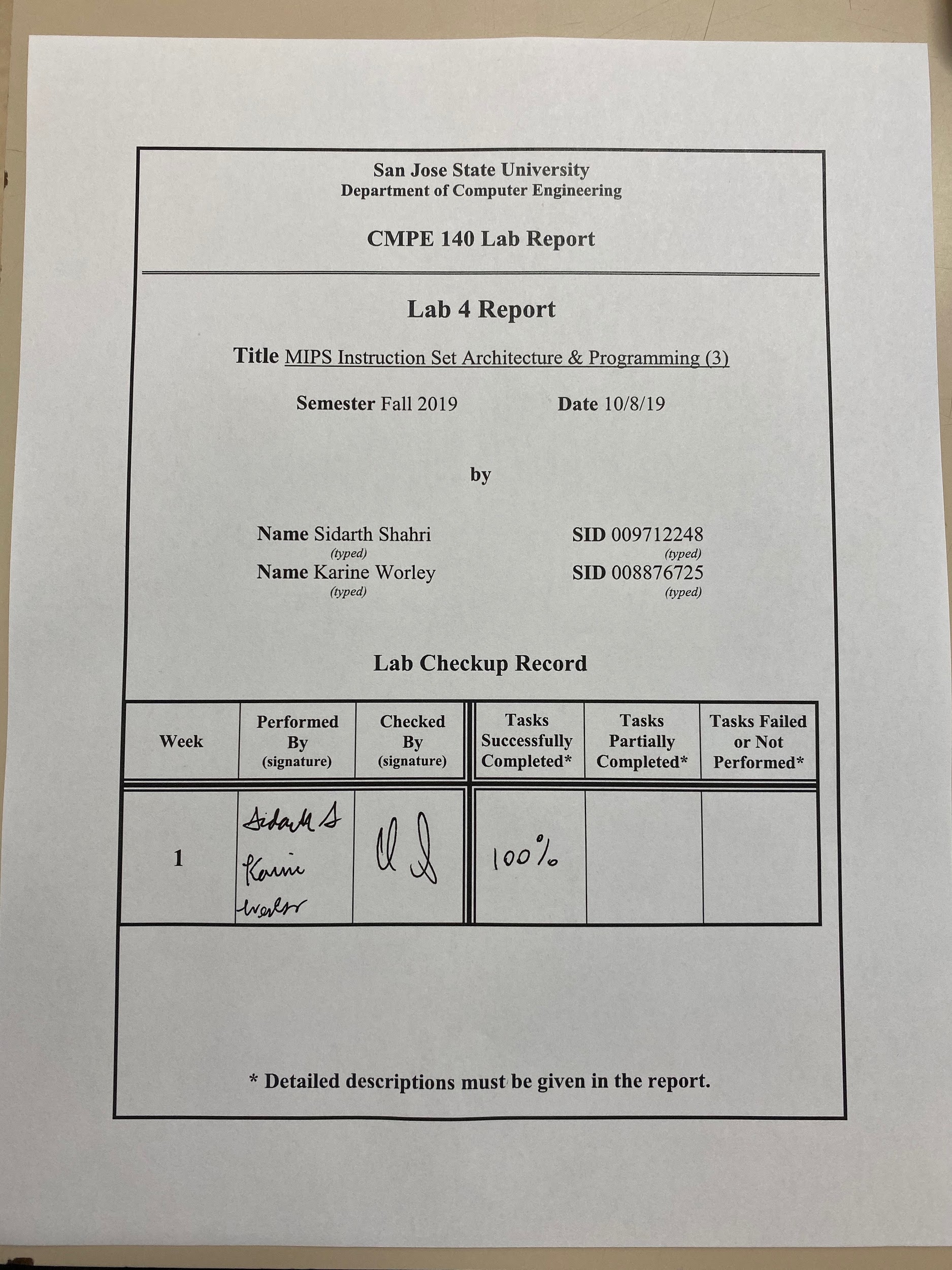
* 

### 

### Purpose

The purpose of this assignment was to learn how to use MIPS to create an array with a starting base address given by the lab requirements. Additionally, the assignment aimed to familiarize us with how MIPS implements stacks, arrays, procedures, and recursive procedures. Like the previous few assignments, this assignment also aimed to help us practice our MIPS programming skills and learn how to use the MARS assembler tool. The tasks for this assignment included writing code to create a 50-element array starting at base address 0x100, initializing all elements of the array, and calculating n! using recursive procedures. All results were logged in a test log and a stack sketch diagram was created to illustrate how the stack was used in the recursive procedure.

### Approach

For task 1, we were asked to write a MIPS assembly program to perform arithmetic on array elements from a previously initialized array. Additionally, we were asked to calculate the factorial of the calculated value from the previous step using a recursive procedure. This task was completed successfully with the help of the provided code.

The provided assembly code initialized a 50-element array at base address 0x100. The array was initialized with multiples of 3 such that array[0] = 0, array[1] = 3, array[2] = 6, and so on. After the array was filled, we were asked to perform the following arithmetic: n = (my\_array[25]+my\_array[30])/30. To do this, we first loaded my\_array[25] and my\_array[30] from memory into separate registers. The address of my\_array[25] was calculated by multiplying 4 and 25 and adding the result to the address stored in $a0. Likewise, the address of my\_array[30] was calculated by multiplying 4 and 30 and adding the result to the address stored in $a0. Then, we performed an add instruction and stored the result in $a1 or n as dictated by the requirements. Next, we loaded the immediate value 30 into a temporary register. Finally, we performed the division and pulled the result from the $LO register into $a1. $a1’s final value was 5.

Next, we began writing code for the recursive procedure to calculate n! where n = $a1 = 5. The requirements for this part of the task included using $a1 as n and $s0 as n!, implementing the factorial function as a recursive procedure, and storing the value of n at memory location 0x00 and the value of n! at memory location at 0x10. To accomplish this part of the task, we used the given pseudocode to design our assembly function. First, the function creates room on the stack by subtracting 8 from the stack pointer. Then, the new space is filled with the return address and value of n ($a1). Next, a comparison was performed to find if n <= 1. If true, then a recursive call was performed and the above steps repeated until n <= 1 was false. Once false, the function begins to return, restoring the return address and value of n from the previous function call to the relevant registers. The stack pointer is incremented by 8 for the next return call. Additionally, the multiplication of n and the previous return value is performed and stored in $v0. The final value for n! is stored in $s0 after the function returns to the original point of being called using the return address. Then, the program jumps to the end label where the final values of n and n! are stored to memory. Figure 1 shows how the stack pointer changes throughout each iteration of the recursive factorial function.

For task 2, the test log was filled out after single stepping through each line of code. This test log can be found in Table 1 in Appendix B. Additionally, a stack sketch diagram can be found below in Figure 1. The final value at 0x00 is 0x5 and the final value at 0x10 is 0x78.



Figure 1. Stack Diagram showing how the stack pointer moves as code executes through the recursive factorial loop. Additionally, a table shows how the relevant registers are updated after each iteration.

### Accomplished Tasks

1. Task 1: Wrote MIPS code for Recursive Factorial
2. Task 2: Assembled the MIPS code and filled in the test logs
   1. Drew diagram of stack status regarding:
      1. The relevant addresses
      2. Stack pointer location
      3. $a1 and $ra contents
      4. Final value of 0x00 - 0x03
      5. Final value of 0x10 - 0x13
3. Sketched the stack status diagram depicting addresses, and the location of the stack pointer for each iteration

### 

### Conclusion

The purpose of the lab was accomplished successfully. A foundation for implementing arrays with MIPS was built upon. Additionally, the lab provided a familiarity of how to perform arithmetic calculations in addition to recursive functions, procedures, and stacks. When writing the MIPS version of the C++ code, we learned that verifying the calculations beforehand would lead to further efficiency when checking the output to the registers. When assembling the MIPS code, we learned that recursiveness can be easily validated when cycling through each iteration. Sketching the stack status diagram also allowed further verification of the expected results of the MIPS version. With each task accomplished successfully, the purposes of the lab were fulfilled.

### 

### Appendix A: MIPS Assembly Code

|  |
| --- |
| **recursive\_factorial.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 # store 3 for use in for loop  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:  # n = (my\_array[25] + my\_array[30])/30  lw $t0, 100($a0) # loading my\_array[25]  lw $t1, 120($a0) # loading my\_array[30]  add $a1, $t0, $t1 # add my\_array[25] and my\_array[30]  ori $t0, $0, 30 # load imm 30 for division  div $a1, $t0 # divide n by 30  mflo $a1 # store result into n = 5    # factorial computation  jal factorial # call procedure  add $s0, $v0, $0 # return value  j end  factorial:  addi $sp, $sp, -8 # make room on the stack  sw $a1, 4($sp) # store $a1  sw $ra, 0($sp) # store $ra  addi $t0, $0, 2 # $t0 = 2  slt $t0, $a1, $t0 # n <= 1?  beq $t0, $0, else # no: go to else (recursion)  addi $v0, $0, 1 # yes: return 1  addi $sp, $sp, 8 # restore $sp  jr $ra # return  else: addi $a1, $a1, -1 # n = n - 1  jal factorial # recursive call  lw $ra, 0($sp) # restore return address  lw $a1, 4($sp) # restore input  addi $sp, $sp, 8 # restore $sp  mul $v0, $a1, $v0 # n \* factorial(n-1)  jr $ra # return    end: sw $a1, 0($0) # store n in 0x00  sw $s0, 0x10($0) # store n! in 0x10  # finish |

### Appendix B: Test Logs

Table 1: Test Log for Task 1

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Addr | MIPS Instruction | Machine Code | Registers | | | | Memory Content | |
| $a1 | $sp | $ra | $v0 | [0x00] | [0x10] |
| 3034 | lw $8, 100($4) | 0x8c880064 | 0x00000032 | 0x00002ffc | 0x0 | 0x0 | 0x0 | 0x0 |
| 3038 | lw $9, 120($4) | 0x8c890078 | x00000032 | 0x00002ffc | 0x0 | 0x0 | 0x0 | 0x0 |
| 303c | add $a1, $t0, $t1 | 0x01092820 | 0x000000a5 | 0x00002ffc | 0x0 | 0x0 | 0x0 | 0x0 |
| 3040 | ori $t0, $0, 30 | 0x3408001e | 0x000000a5 | 0x00002ffc | 0x0 | 0x0 | 0x0 | 0x0 |
| 3044 | div $a1, $t0 | 0x00a8001a | 0x000000a5 | 0x00002ffc | 0x0 | 0x0 | 0x0 | 0x0 |
| 3048 | mflo $a1 | 0x00002812 | 0x00000005 | 0x00002ffc | 0x0 | 0x0 | 0x0 | 0x0 |
| 304c | jal factorial | 0x0c000c16 | 0x00000005 | 0x00002ffc | 0x00003050 | 0x0 | 0x0 | 0x0 |
| 3050 | add $s0, $v0, $0 | 0x00408020 | 0x00000005 | 0x00002ffc | 0x00003050 | 0x00000078 | 0x0 | 0x0 |
| 3054 | j end | 0x08000c26 | 0x00000005 | 0x00002ffc | 0x00003050 | 0x00000078 | 0x0 | 0x0 |
| 3058 | addi $sp, $sp, -8 | 0x23bdffff8 | 0x00000005 | 0x00002ff4 | 0x00003050 | 0x0 | 0x0 | 0x0 |
| 305c | sw $a1, 4($sp) | 0xafa50004 | 0x00000005 | 0x00002ff4 | 0x00003050 | 0x0 | 0x0 | 0x0 |
| 3060 | sw $ra, 0($sp) | 0xafbf0000 | 0x00000005 | 0x00002ff4 | 0x00003050 | 0x0 | 0x0 | 0x0 |
| 3064 | addi $t0, $0, 2 | 0x20080002 | 0x00000005 | 0x00002ff4 | 0x00003050 | 0x0 | 0x0 | 0x0 |
| 3068 | slt $t0, $a1, $t0 | 0x00a8402a | 0x00000005 | 0x00002ff4 | 0x00003050 | 0x0 | 0x0 | 0x0 |
| 306c | beq $t0, $0, else | 0x11000003 | 0x00000005 | 0x00002ff4 | 0x00003050 | 0x0 | 0x0 | 0x0 |
| 3070 | addi $v0, $0, $1 | 0x20020001 | 0x00000001 | 0x00002fd4 | 0x00003084 | 0x00000001 | 0x0 | 0x0 |
| 3074 | addi $sp, $sp, 8 | 0x23bd0008 | 0x00000001 | 0x00002fdc | 0x00003084 | 0x00000001 | 0x0 | 0x0 |
| 3078 | jr $ra | 0x03e00008 | 0x0000001 | 0x00002fdc | 0x00003084 | 0x00000001 | 0x0 | 0x0 |
| 307c | addi $a1, $a1, -1 | 0x20a5ffff | 0x00000004 | 0x00002ff4 | 0x00003050 | 0x0 | 0x0 | 0x0 |
| 3080 | jal factorial | 0x0c000c16 | 0x00000004 | 0x00002ff4 | 0x00003084 | 0x0 | 0x0 | 0x0 |
| 3084 | lw $ra, 0($sp) | 0x8fbf0000 | 0x00000001 | 0x00002fdc | 0x00003084 | 0x00000001 | 0x0 | 0x0 |
| 3088 | lw $a1, 4($sp) | 0x8fa50004 | 0x00000002 | 0x00002fdc | 0x00003084 | 0x00000001 | 0x0 | 0x0 |
| 308c | addi $sp, $sp, 8 | 0x23bd0008 | 0x00000002 | 0x00002fe4 | 0x00003084 | 0x00000001 | 0x0 | 0x0 |
| 3090 | mul $v0, $a1, $v0 | 0x70a21002 | 0x00000002 | 0x00002fe4 | 0x00003084 | 0x00000002 | 0x0 | 0x0 |
| 3094 | jr $ra | 0x03e00008 | 0x00000002 | 0x00002fe4 | 0x00003084 | 0x00000002 | 0x0 | 0x0 |
| 3098 | end: sw $a1, 0($0) | 0xac050000 | 0x00000005 | 0x00002ffc | 0x00003050 | 0x00000078 | 0x00000005 | 0x0 |
| 309c | sh $16, 10($0) | 0xa410000a | 0x00000005 | 0x00002ffc | 0x00003050 | 0x00000078 | 0x00000005 | 0x00780000 |
| 3100 |  |  |  |  |  |  |  |  |
| 3104 |  |  |  |  |  |  |  |  |

### Appendix C: Screenshots

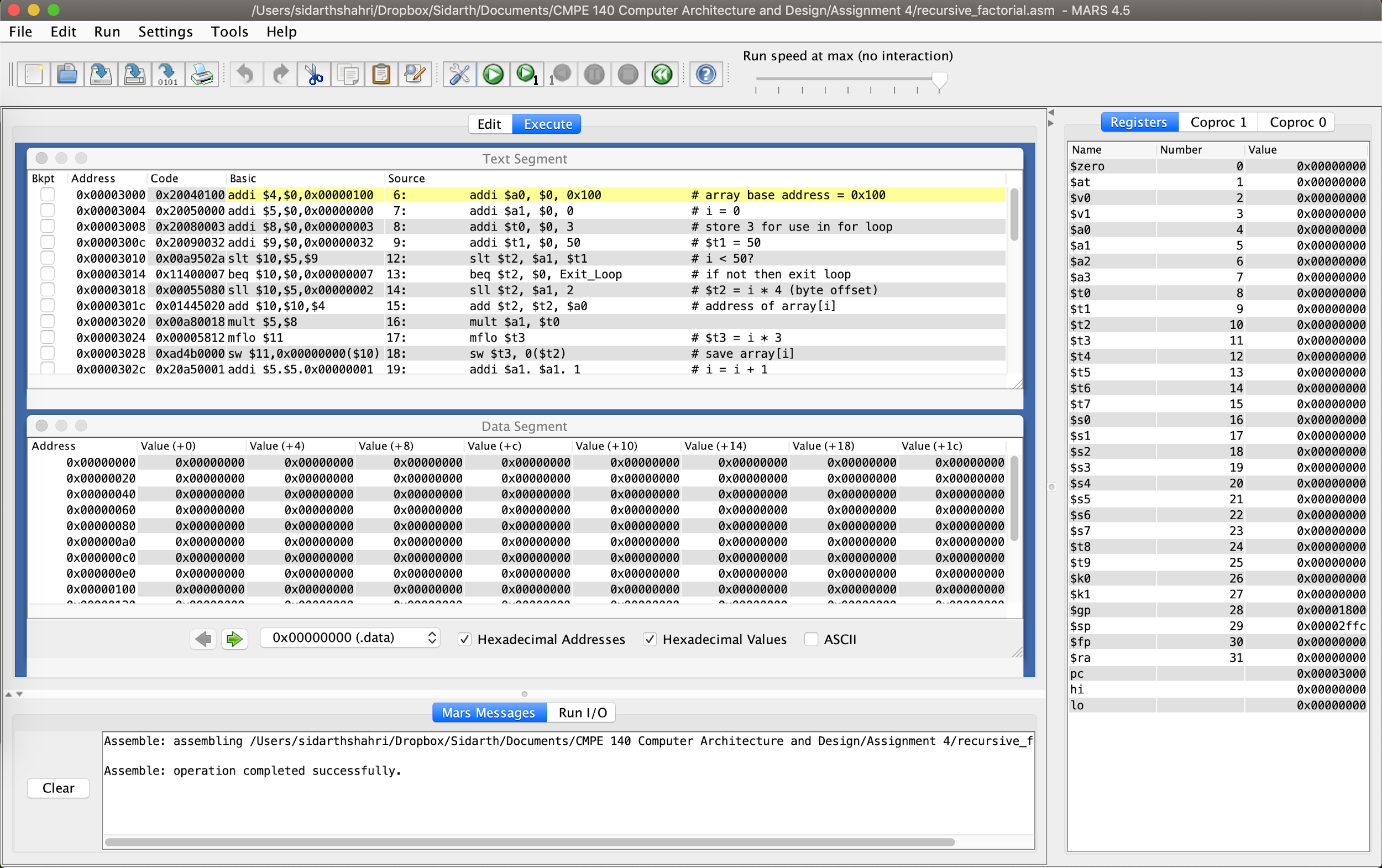
****

Figure 2. Screenshot showing assembled code and contents of memory and registers before any code is executed.

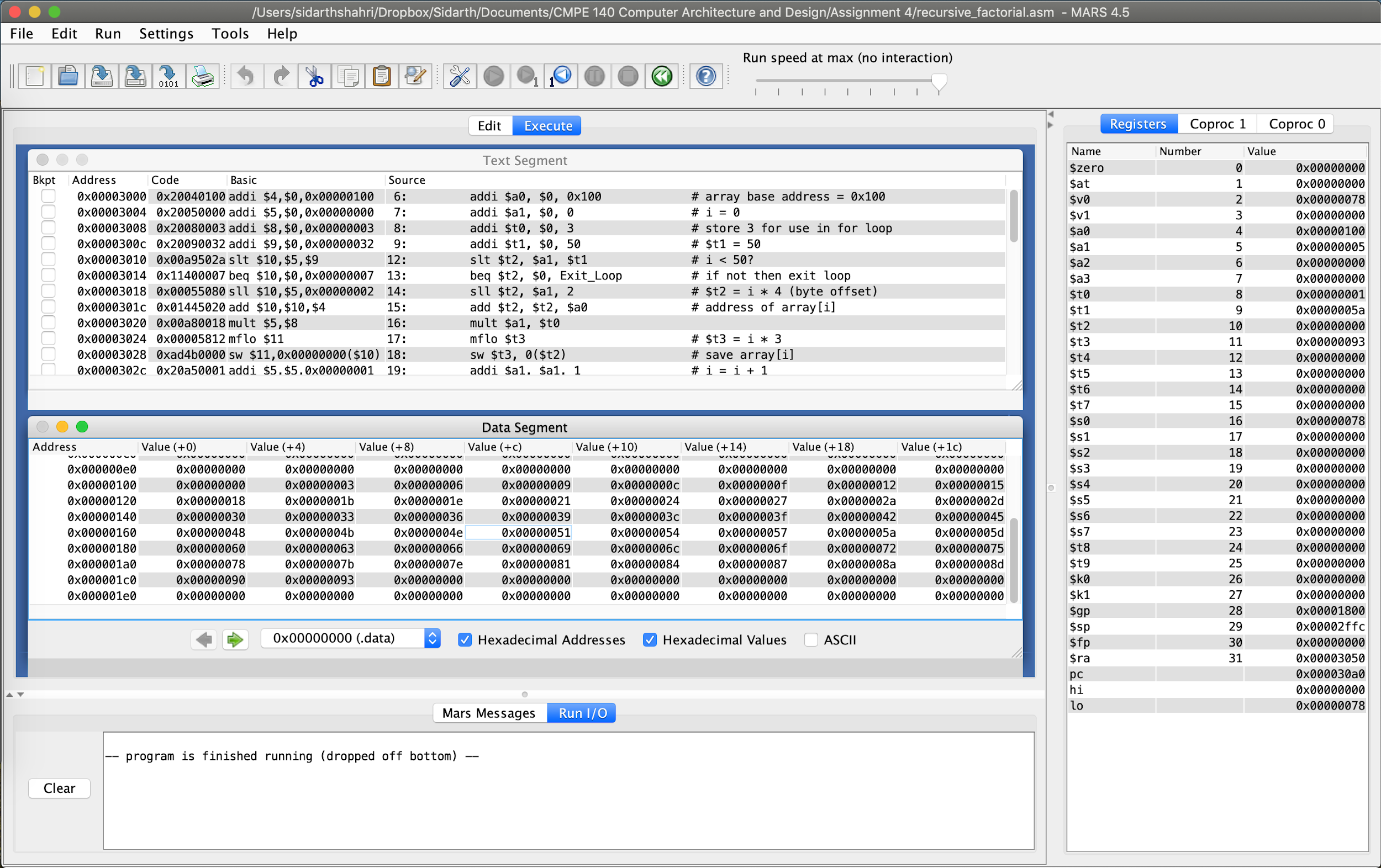


Figure 3. Screenshot showing contents of array after initialization.

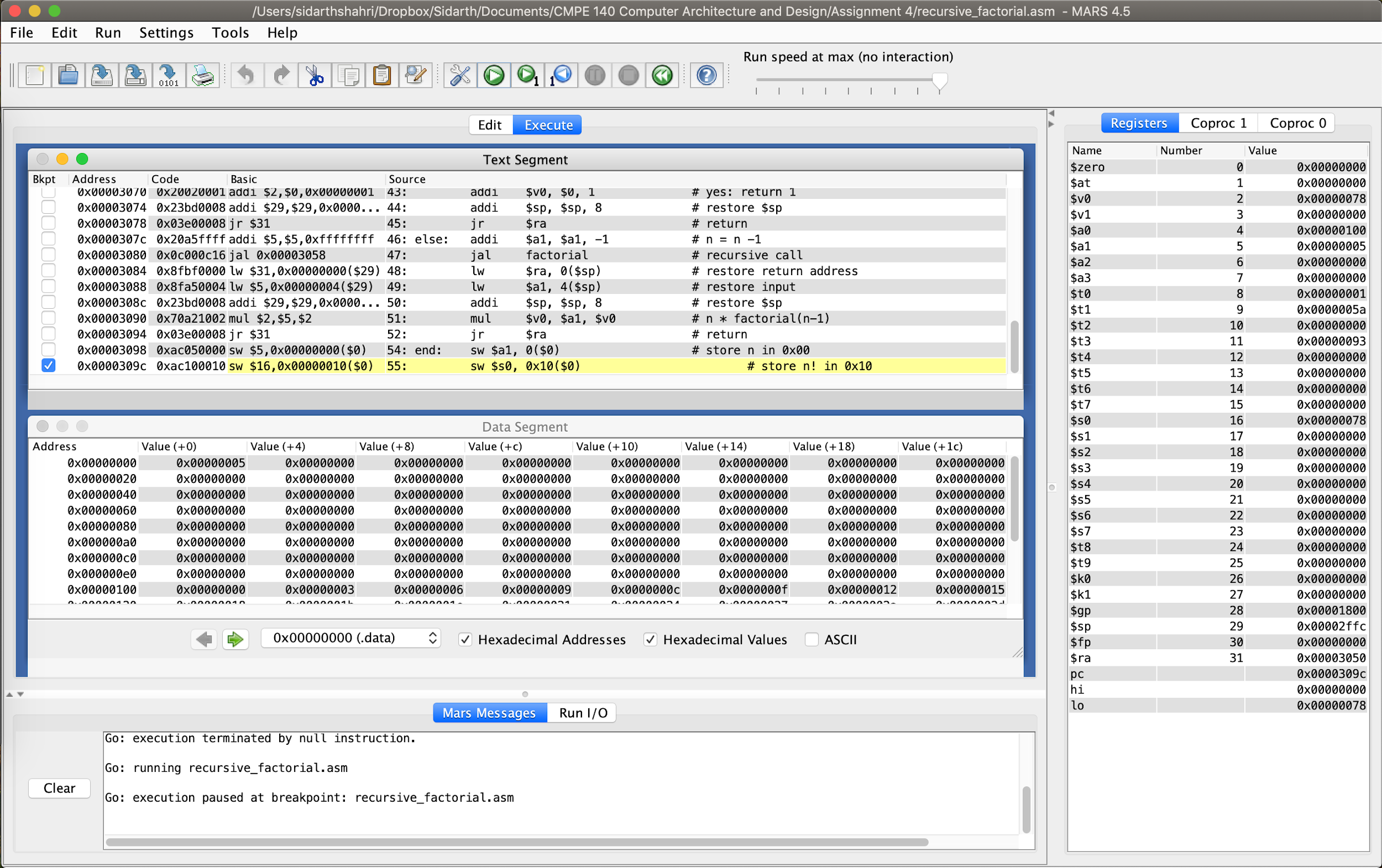


Figure 4. Screenshot showing assembled code and contents of registers and memory before the final result of n! is stored to memory.

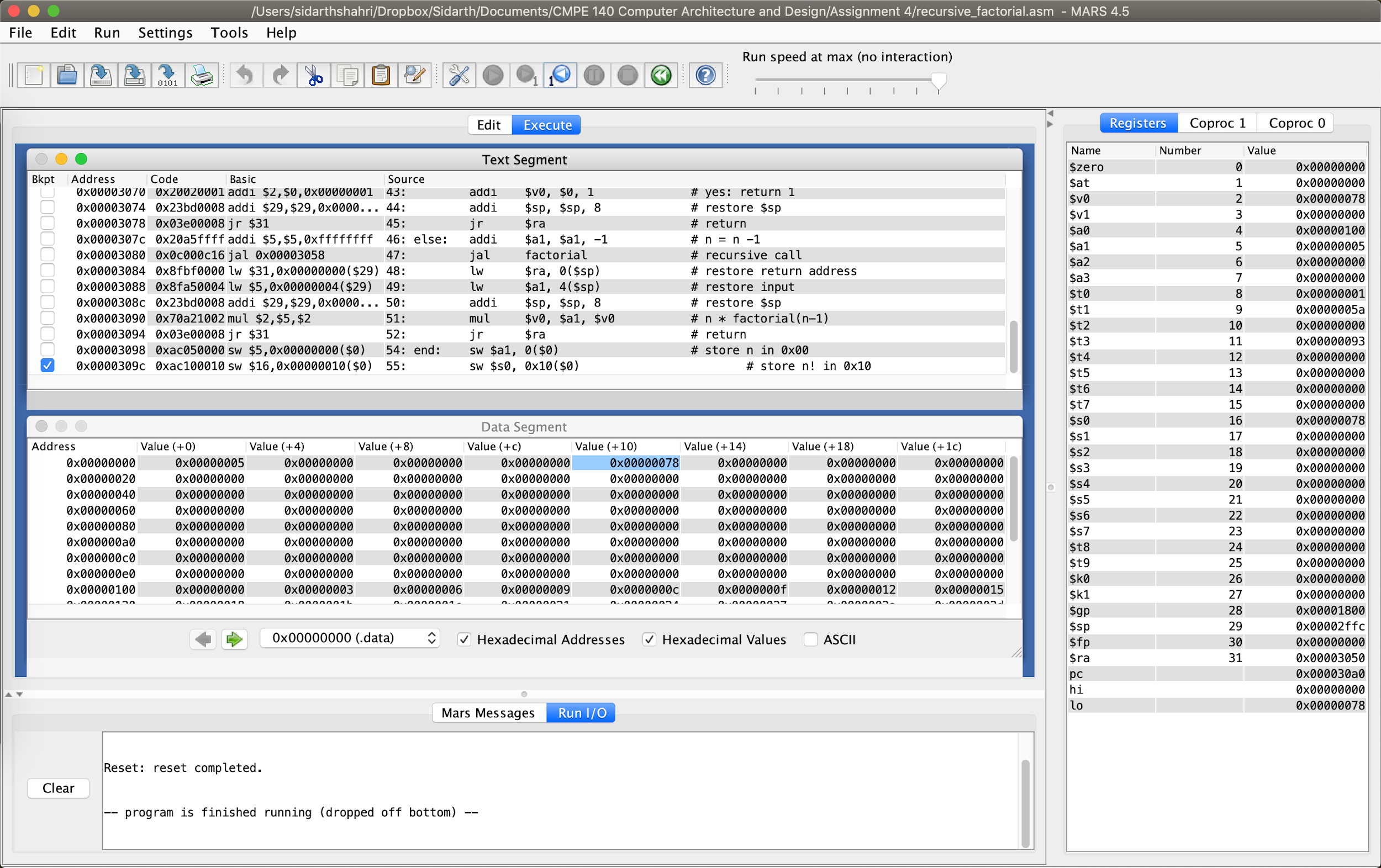


Figure 5. Screenshot showing assembled code and final contents of registers and memory after n! is stored to 0x10.