

## Purpose

In Assignment 2, we were asked to install the MARS tool on our personal computers to run sample assembly code. The goal behind the lab was to obtain familiarity of MIPS instructions. By aiming to grasp an overall familiarity of MIPS, students established additional aims of learning how the instructions can function (*add, sub, and, or, slt, addi, lw, sw, beq, j*) and how the assembler can generate machine code for them. Furthermore, the lab required examination of the sample assembly code, recorded output of running the code, and generated machine code.

## MARS Tool

The MARS tool has proven to be simple to use yet powerful for examining how MIPS works. For example, the “Help” section of the program has a convenient guide for all MIPS instructions and a guide on how to use the MARS tool. Both guides were helpful in approaching and verifying the usages of each instruction. The guide regarding the instructions provided a description for each relevant line. Being able to switch between hex and decimal values for the register and data segment addresses also proved to be extremely useful when examining the sample assembly code. The sample assembly code was well commented to aid in understanding what each instruction was doing. Through examination, we discovered the difference between R-Type, J-Type, and I-Type instructions.We learned further about how *and, or,* and *slt* are R-Type instructions. The R-Type instructions only involve registers in their computation. An example of a J-Type instruction is *j*. J-Type instructions jump unconditionally. When including *addi*, we learned that the instruction adds an immediate value to a second value in a register (the zero register for this purpose) and stores it into a destination register. The *addi* was an example of an I-Type instruction as it uses an immediate value.

Meanwhile, during the lab various instructions were implemented. The *add* instructions adds the value from two registers and stores it in a register. The s*ub* instruction subtracts the value from two registers and stores it in a register. The *and* instructionobtains values from two registers, performs a bitwise AND, and stores the result in a register. The *or* instruction retrieves values from two registers, performs a bitwise OR, and stores the result in a register. Meanwhile, *slt* obtains values from the register source, shifts it left by the value from the register target, and stores the result into a register destination. The *addi* instructionperforms the same operation as *add.* However, the operation is performed with an immediate value and the value from only one register. The *lw* instruction loads a word (16 bits) from a memory address which is calculated by adding an immediate value to the value in a register source and stores it into a register destination. The *sw* instruction performs similarly to *lw.* However, it saves a value from a register to a calculated memory address. The *beq* instruction branches to a location if the values from two registers are equal. The *j* instruction jumps to a location by loading the new location into the program counter.

We also learned from our lab TA that *beq* is an example of an I-Type instruction. The instruction does not branch directly to a hard-coded location in memory as the actual location of the program in memory may change. Instead *beq* branches to an immediate value that is calculated by the assembler at the time of assembly. That value is calculated by finding the label to jump to and counting how many bytes to add to the program counter to reach that label. We also learned that *beq* performs a comparison by subtracting the values in the register parameters. If the calculated value is 0, a branch occurs. Otherwise, the *beq* instruction does nothing.

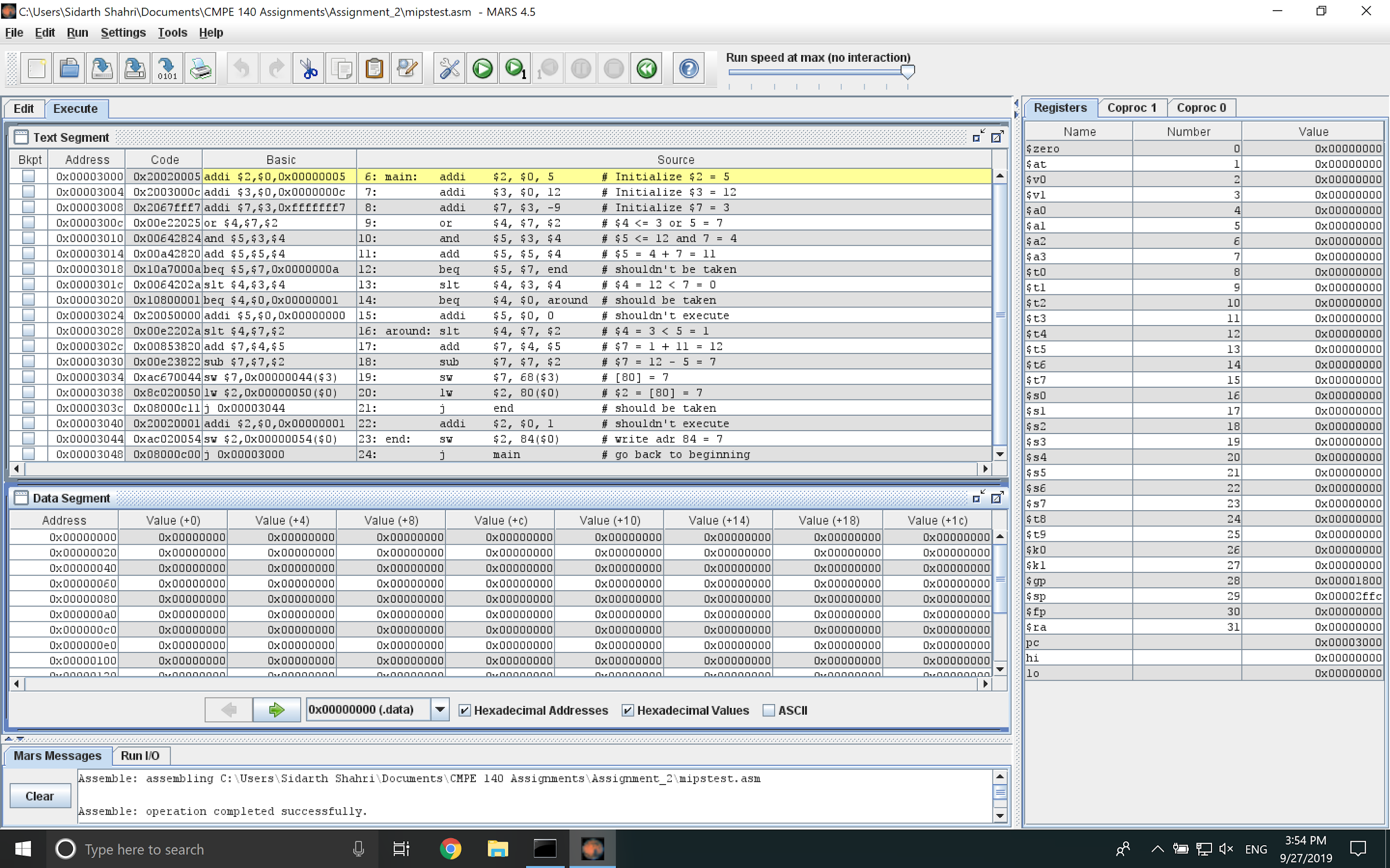


Figure 1. The MARS window after assembling the source assembly code.

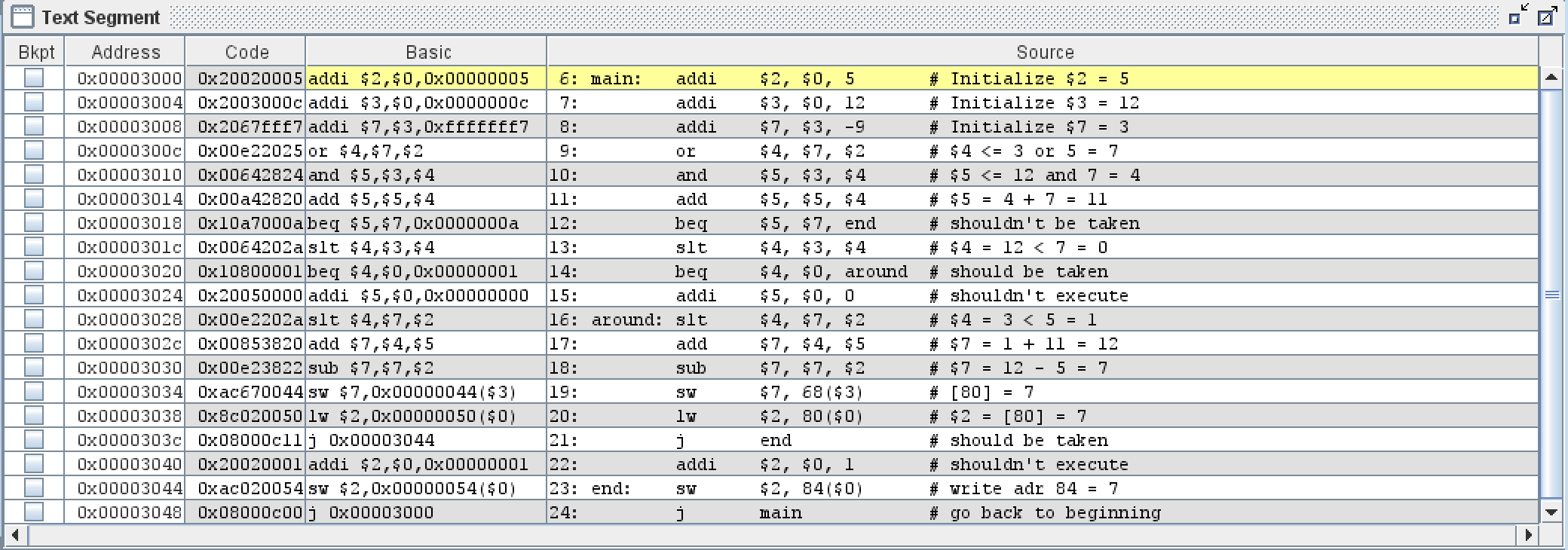


Figure 2. Text Segment depicting relevant assembly code and generated machine code.

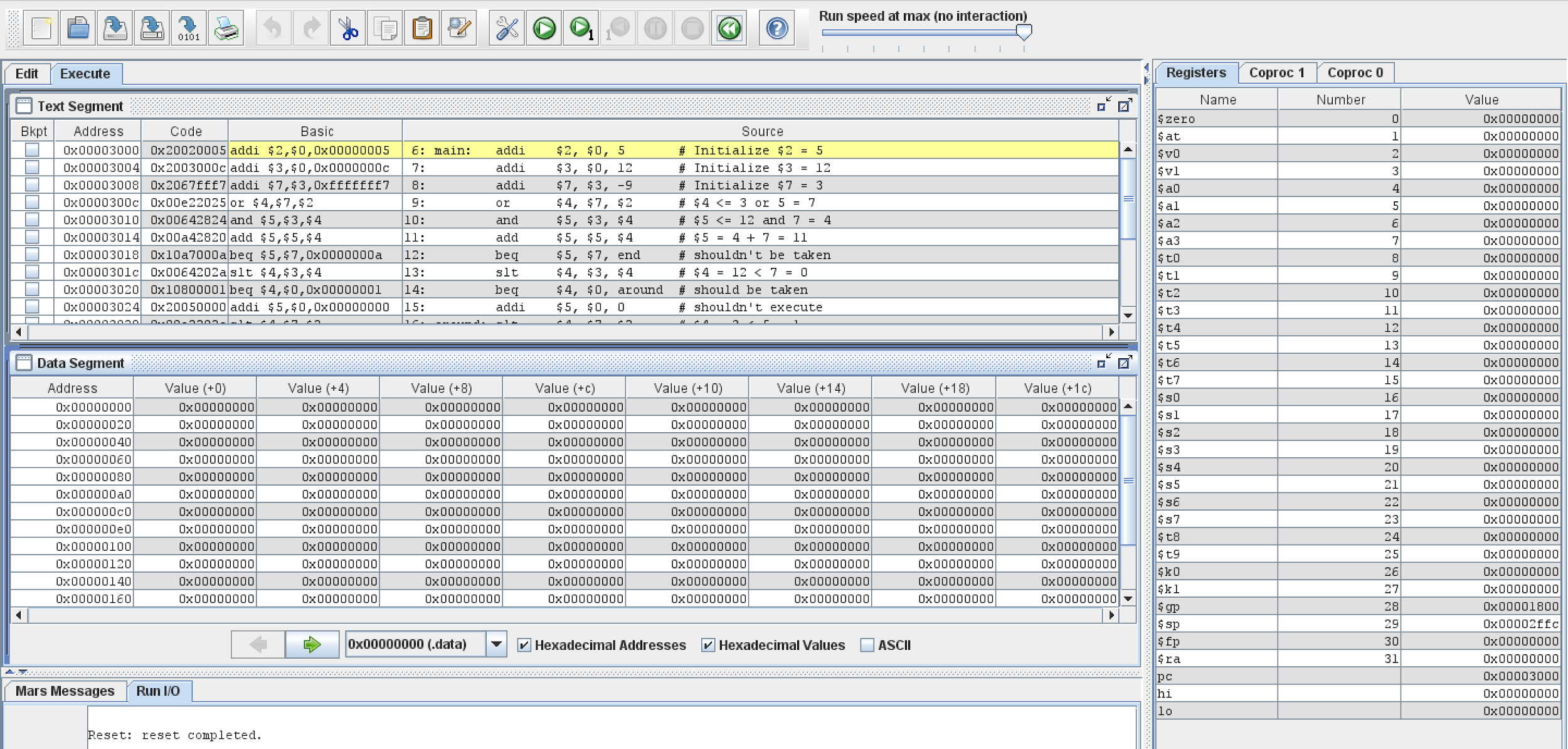


Figure 3. Before running assembled code.

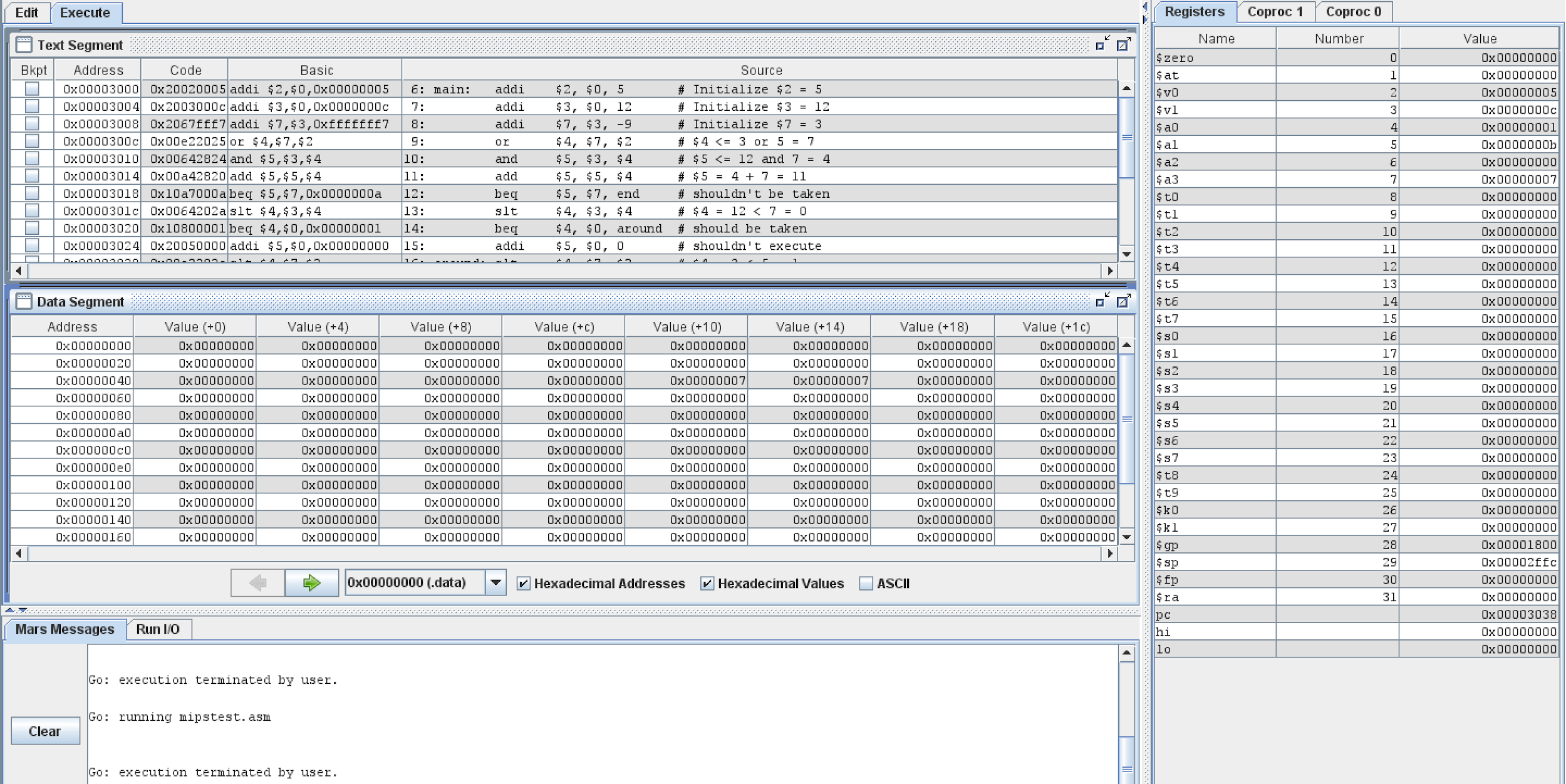


Figure 4. After running assembled code

The assembled source code is depicted in Appendix A. Figure 1 depicts the initial windows after assembling the source code. Figure 2 depicts the Text Segment for the generated assembly code. Figure 3 depicts the generated contents before running the assembled code. Figure 4 depicts generated contents after running each instruction. The test log depicts the expected outputs for the contents of machine code, program counter, registers, and memory as shown in Table 1.

Table 1. Test Log

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Adr | Expected Machine Code (hex) | Actual Machine Code (hex) | PC | Registers (values in hex) | | | | | Memory Content (values in hex) | |
| $v0 | $v1 | $a0 | $a1 | $a3 | [80] | [84] |
| 3000 | 20020005 | 20020005 | 3004 | 5 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3004 | 2003000c | 2003000c | 3008 | 5 | c | 0 | 0 | 0 | 0 | 0 |
| 3008 | 2067fff7 | 2067fff7 | 300c | 5 | c | 0 | 0 | 3 | 0 | 0 |
| 300c | 00e22025 | 00e22025 | 3010 | 5 | c | 7 | 0 | 3 | 0 | 0 |
| 3010 | 00642824 | 00642824 | 3014 | 5 | c | 7 | 4 | 3 | 0 | 0 |
| 3014 | 00a42820 | 00a42820 | 3018 | 5 | c | 7 | b | 3 | 0 | 0 |
| 3018 | 10a7000a | 10a7000a | 301c | 5 | c | 7 | b | 3 | 0 | 0 |
| 301c | 0064202a | 0064202a | 3020 | 5 | c | 0 | b | 3 | 0 | 0 |
| 3020 | 10800001 | 10800001 | 3028 | 5 | c | 0 | b | 3 | 0 | 0 |
| 3024 | 20050000 | 20050000 | skips | skips | skips | skips | skips | skips | skips | skips |
| 3028 | 00e2202a | 00e2202a | 302c | 5 | c | 1 | b | 3 | 0 | 0 |
| 302c | 00853820 | 00853820 | 3030 | 5 | c | 1 | b | c | 0 | 0 |
| 3030 | 00e23822 | 00e23822 | 3034 | 5 | c | 1 | b | 7 | 0 | 0 |
| 3034 | ac670044 | ac670044 | 3038 | 5 | c | 1 | b | 7 | 7 | 0 |
| 3038 | 8c020050 | 8c020050 | 303c | 7 | c | 1 | b | 7 | 7 | 0 |
| 303c | 08100011 | 08000c11 | 3044 | 7 | c | 1 | b | 7 | 7 | 0 |
| 3040 | 20020001 | 20020001 | skips | skips | skips | skips | skips | skips | skips | skips |
| 3044 | ac020054 | ac020054 | 3048 | 7 | c | 1 | b | 7 | 7 | 7 |
| 3048 | 08100000 | 08000c00 | 0 | 7 | c | 1 | b | 7 | 7 | 7 |

## 

## Accomplished Tasks

1. *Installed the MIPS software*
2. *Assembled the source code and compared with machine code*
3. *Stepped through each instruction and verified register and memory values*

## Conclusion

The purpose of the lab was fulfilled successfully with all tasks accomplished. A foundation of how to use MIPS instructions with the MARS tool was built upon. In Assignment 2, we learned that the program counter is incremented by 4 for each instruction (unless branching or jumping to a new location) to keep the program running. We also learned that machine code is generated depending on the type of instruction being assembled. For example, for a R-Type instruction, the first 6 bits are the opcode, the next 5 bits are the register source (rs), the next 5 bits are the register target (rt), the next 5 bits are the register destination (rd), the next 5 bits are the shift amount (shamt), and the final 6 bits are the function. This varies depending on the type of instruction and can be found in the MIPS guide posted on Canvas. The lab provided a solid foundation for how to apply MIPS instructions with future labs.

## Appendix A

Table 2. MIPS Assembled Code

|  |
| --- |
| **mipstest.asm** |
| # mipstest.asm  # Test the following MIPS instructions.  # add, sub, and, or, slt, addi, lw, sw, beq, j  # Assembly Description  main: addi $2, $0, 5 # Initialize $2 = 5  addi $3, $0, 12 # Initialize $3 = 12  addi $7, $3, -9 # Initialize $7 = 3  or $4, $7, $2 # $4 <= 3 or 5 = 7  and $5, $3, $4 # $5 <= 12 and 7 = 4  add $5, $5, $4 # $5 = 4 + 7 = 11  beq $5, $7, end # shouldn't be taken  slt $4, $3, $4 # $4 = 12 < 7 = 0  beq $4, $0, around # should be taken  addi $5, $0, 0 # shouldn't execute  around: slt $4, $7, $2 # $4 = 3 < 5 = 1  add $7, $4, $5 # $7 = 1 + 11 = 12  sub $7, $7, $2 # $7 = 12 - 5 = 7  sw $7, 68($3) # [80] = 7  lw $2, 80($0) # $2 = [80] = 7  j end # should be taken  addi $2, $0, 1 # shouldn't execute  end: sw $2, 84($0) # write adr 84 = 7  j main # go back to beginning |