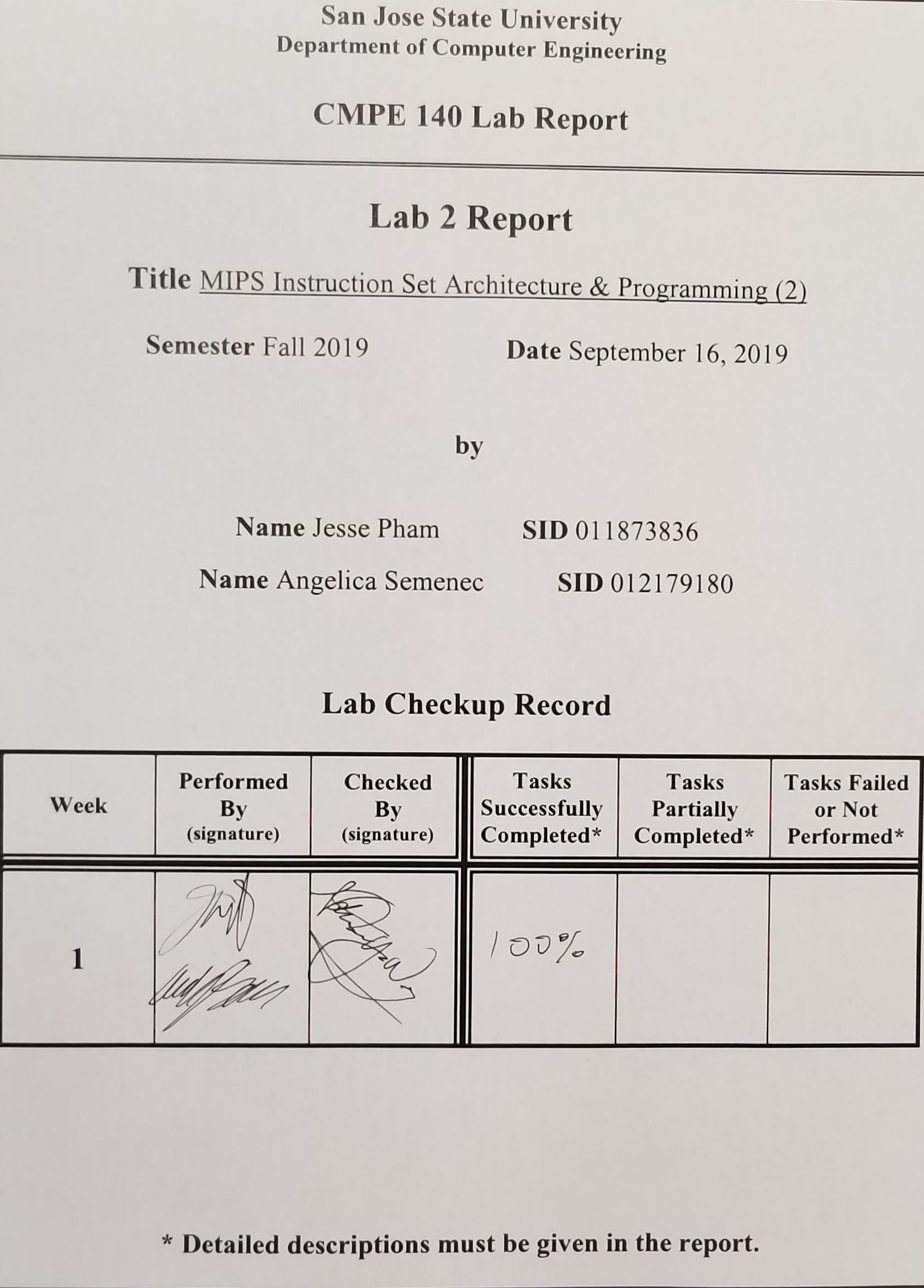
****

**Purpose**

The purpose of this lab was to utilize the MARS IDE to become familiar with the MIPS instruction set architecture by assembling, simulating, and analyzing the given code shown in the source code section of this document under *mipstest.asm*.

**Design Methodology**

The source code shown below, *mipstest.asm*, was provided within the assignment instructions. The code shows the function of various MIPS instructions. By single-stepping through the given code, the address of the instruction, machine code, program counter, and registers utilized within the given code can be determined to gain an understanding of the instruction set architecture and functionality. With each step, the changes in the registers, program counter, and the memory content were recorded in a table. The actual machine code was also recorded to compare it to the expected machine code.

**Tasks Successfully Accomplished**

1. Successfully installed the MIPS Assembler/Simulator software
2. Assembled the MIPS code in a file called “mipstest.asm.”
3. Single-stepped through the execution of the instructions and recorded the relevant contents into a test log table.

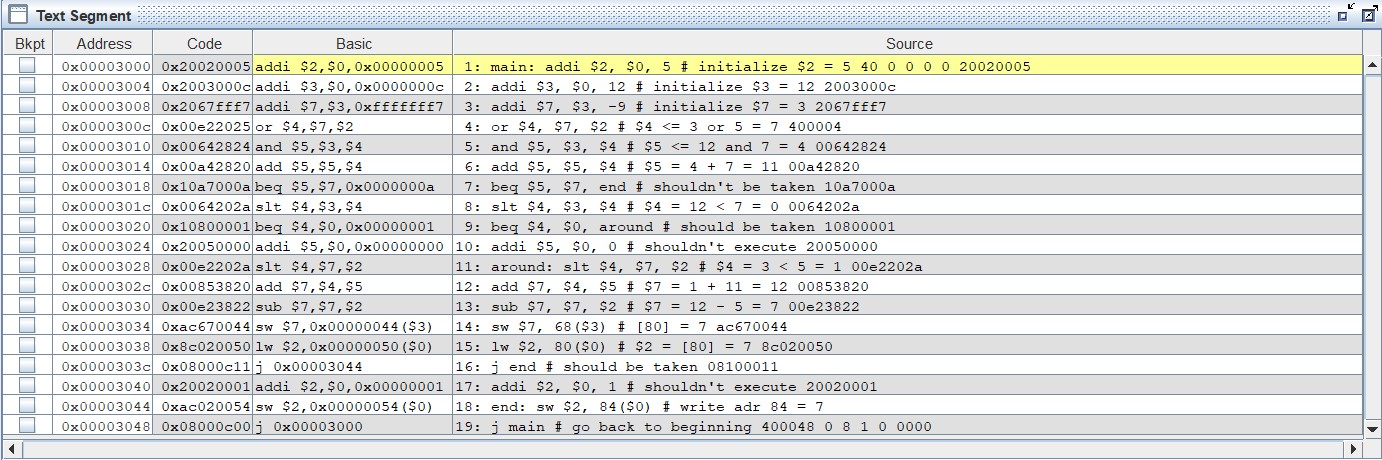
**Source Code**

|  |
| --- |
| **mipstest.asm** |
| *# mipstest.asm*  *# Test the following MIPS instructions.*  *# add, sub, and, or, slt, addi, lw, sw, beq, j*  *# Assembly Description Address Machine*  main: addi $2, $0, 5 *# initialize $2 = 5 4000 20020005 00*  addi $3, $0, 12 *# initialize $3 = 12 2003000c*  addi $7, $3, -9 *# initialize $7 = 3 2067fff7*  or $4, $7, $2 *# $4 <= 3 or 5 = 7*  and $5, $3, $4 *# $5 <= 12 and 7 = 4 00642824*  add $5, $5, $4 *# $5 = 4 + 7 = 11 00a42820*  beq $5, $7, end *# shouldn't be taken 10a7000a*  slt $4, $3, $4 *# $4 = 12 < 7 = 0 0064202a*  beq $4, $0, around *# should be taken 10800001*  addi $5, $0, 0 *# shouldn't execute 20050000*  around: slt $4, $7, $2 *# $4 = 3 < 5 = 1 00e2202a*  add $7, $4, $5 *# $7 = 1 + 11 = 12 00853820*  sub $7, $7, $2 *# $7 = 12 - 5 = 7 00e23822*  sw $7, 68($3) *# [80] = 7 ac670044*  lw $2, 80($0) *# $2 = [80] = 7 8c020050*  j end *# should be taken 08100011*  addi $2, $0, 1 *# shouldn't execute 20020001*  end: sw $2, 84($0) *# write adr 84 = 7*  j main *# go back to beginning 400048 08100000* |

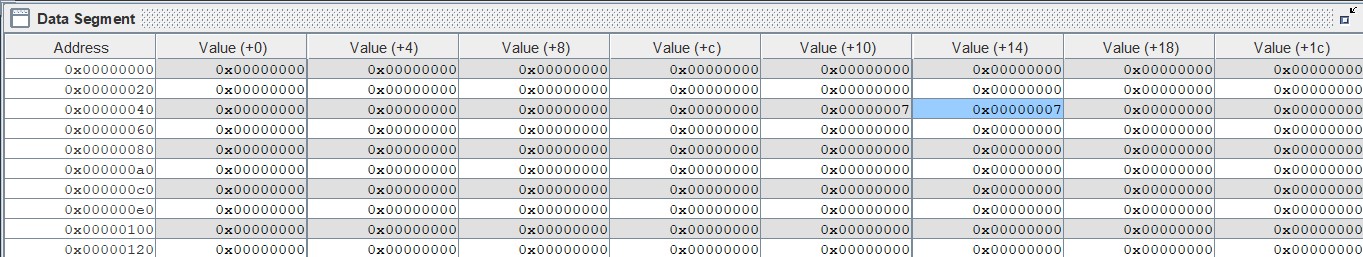
**Test Results**

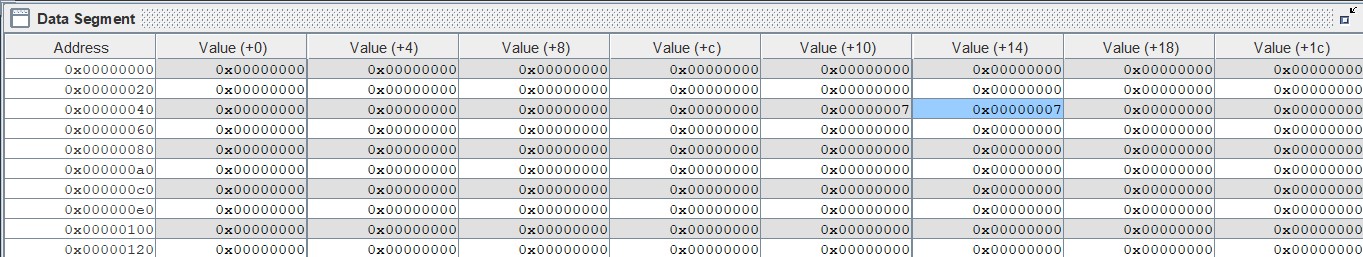
Table 1: The test log table shows the changes in value within registers, memory contents, the program counter, and the actual machine code when single-stepping through each execution of instruction.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Adr | Expected Machine Code | Actual Machine Code | PC | Registers | | | | | Memory Content | |
| $v0 | $v1 | $a0 | $a1 | $a3 | [80] | [84] |
| 400000 | 20020005 | 20020005 | 4 | 5 | 0 | 0 | 0 | 0 | 0 | 0 |
| 400004 | 2003000c | 2003000c | 8 | 5 | c | 0 | 0 | 0 | 0 | 0 |
| 400008 | 2067fff7 | 2067fff7 | c | 5 | c | 0 | 0 | 3 | 0 | 0 |
| 40000c | 00e22025 | 00e22025 | 10 | 5 | c | 7 | 0 | 3 | 0 | 0 |
| 400010 | 00642824 | 00642824 | 14 | 5 | c | 7 | 4 | 3 | 0 | 0 |
| 400014 | 00a42820 | 00a42820 | 18 | 5 | c | 7 | b | 3 | 0 | 0 |
| 400018 | 10a7000a | 10a7000a | 1c | 5 | c | 7 | b | 3 | 0 | 0 |
| 40001c | 0064202a | 0064202a | 20 | 5 | c | 0 | b | 3 | 0 | 0 |
| 400020 | 10800001 | 10800001 | - | - | - | - | - | - | - | - |
| 400024 | 20050000 | 20050000 | 28 | 5 | c | 0 | b | 3 | 0 | 0 |
| 400028 | 00e2202a | 00e2202a | 2c | 5 | c | 1 | b | 3 | 0 | 0 |
| 40002c | 00853820 | 00853820 | 30 | 5 | c | 1 | b | c | 0 | 0 |
| 400030 | 00e23822 | 00e23822 | 34 | 5 | c | 1 | b | 7 | 0 | 0 |
| 400034 | ac670044 | ac670044 | 38 | 5 | c | 1 | b | 7 | 7 | 0 |
| 400038 | 8c020050 | 8c020050 | 3c | 7 | c | 1 | b | 7 | 7 | 0 |
| 40003c | 08100011 | 08000c11 | - | - | - | - | - | - | - | - |
| 400040 | 20020001 | 20020001 | 44 | 7 | c | 1 | b | 7 | 7 | 0 |
| 400044 | ac020054 | ac020054 | 48 | 7 | c | 1 | b | 7 | 7 | 7 |
| 400048 | 08100000 | 08000c00 | 0 | 7 | c | 1 | b | 7 | 7 | 7 |

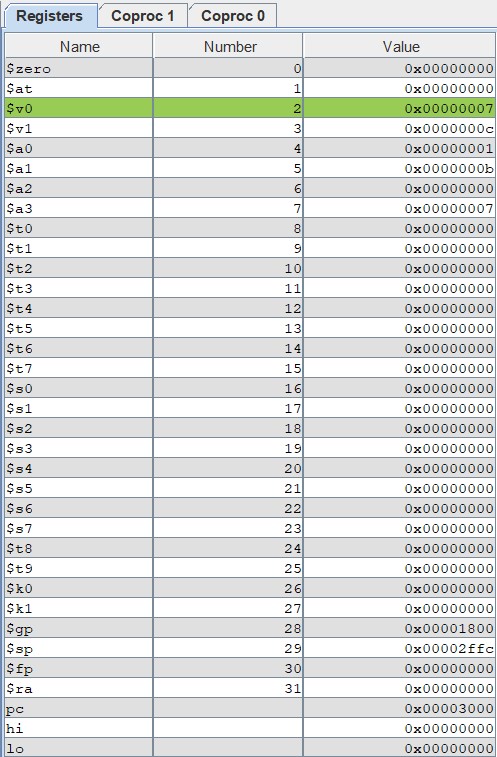
**Screen Capture**

*Figure 1: Showing the Text Segment after assembling the code. This segment displays the Machine Code and the address of each instructions.*

****

****

*Figure 2: Showing the Data segment after reaching the end of the code. The value at address 0x00000050 and 0x00000054 are both 7.*

****

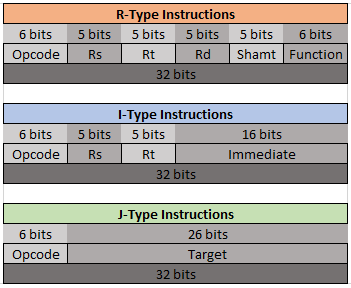
*Figure 3: Showing the registers and their corresponding values as well as the value within the program counter after the code has been executed.*

**Discussion/ Conclusion**

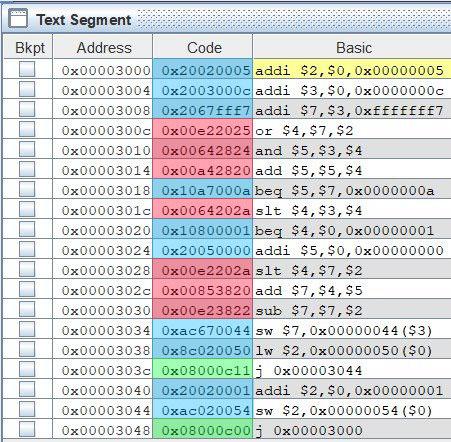
The MIPS code was assembled and ran as expected. Initially, when the actual machine code differed from the expected machine code at addresses 40003c and 400048, we thought the experiment had failed, however, upon discussing the reasons behind their differences, it was concluded that the machine codes were not the same due to their point of reference in memory allocation. Before configuring the memory settings to start the data segment at 0, the beginning address used was 0x00400000, which fits with what we were expecting. After configuring the memory settings to start the Data Segment at 0, the beginning address for the instructions was 0x00003000. This leads to a difference in Machine Code whenever absolute addressing is used instead of relative addressing. For more context, the BRANCH command utilizes relative addressing, which means that the address that it is branching to is relative to its initial position within the code. This doesn’t change the opcode because it will branch a certain memory location distance every time. However, when using the JUMP command, absolute addressing will be used. This instruction will have the program jump to a specific address in the code. This means that when we changed our memory configuration to have the beginning address start at 0x00003000, the machine code for the JUMP instruction changed since the address for the beginning and end of the code were different.

There are three different types of instructions in the MIPS language: R-type, I-type, and J-type. R-type instructions are used when the data values being accessed are located in the registers. These instruction types are generally identified by their opcode being of value 0. I-type instructions are used when operating on an immediate value and a register value. Their opcodes are anything greater than 3. J-type instructions are used when a JUMP instruction is being called. The J-type instructions have the most space for immediate values due to addresses being large numbers. Their opcodes are generally identified with the value 2 and 3. The format for these three different types of instructions are shown below in *Figure 4*. For the experiment, the R-type instructions are shaded red, the I-type instructions are shaded blue, and the J-type instructions are shaded green in *Figure 5* below to differentiate which instruction types were being used.

Overall, the experiment provided a thorough walk-through of how each instruction is executed. It provided an opportunity to become familiar with MIPS instruction set architecture through the use of the MARS IDE. With each instruction, the machine code was discussed to differentiate their instruction types, and the register values were analyzed to ensure that the code was working as intended. The test log table helped keep track of what was changing and to ensure the experiment was properly executed.



*Figure 4: Showing the format for R-type, I-type, and J-type instructions.*



*Figure 5: Showing the different types of instructions being used for each instruction. The instructions that are shaded red represents an R-type instruction, blue represents an I-type instruction, and green represents a J-type instruction.*