**ECE 260: Fundamentals of Computer Engineering – Lab #3  
Introduction to MIPS Assembly and MARS Simulator**

**Name \_\_\_**

**Le Quient Lewis\_\_\_\_\_ Lab Partner(s) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**1. Introduction**: This lab provides a hands-on exploration of the MIPS architecture and assembly language. Students will learn how instructions and data are stored in memory, the role of pseudo-instructions, and how to write basic programs.

**2. Background**: Several concepts are important for this lab. First, both *data* and *instructions* are stored in the same memory within the computer. In this lab, we will only use statically declared variables that will be stored in the *.data* section. Instructions are stored into their own memory location called the *.text* section. Each section represents blocks of memory that are reserved within RAM. Several MIPS instructions will be used in this lab, including *add, sub, lw,* and *sw*. Please refer to your notes and the MIPS ‘Green Sheet’ to recall how to use these instructions. You can find can find a copy of the Green Sheet on the Resources page of the course web site [here](https://ycpcs.github.io/ece260-spring2019/resources/index.html). There, you will also find a simplified ‘Cheat Sheet’ created by David Broman from the KTH Royal Institute of Technology.

When writing MIPS, our primary software environment will be the MARS simulator. This software is available as a Java JAR executable and is located on the resources page of the course web site [here](https://ycpcs.github.io/ece260-spring2019/resources/index.html). **Please download and save this executable JAR to the C:\temp drive on a KEC computer (there may already be a copy on your computer)**. If you attempt to save this JAR file elsewhere on a KEC computer, it will not work properly. Assuming you saved the JAR file to the correct location, you should be able to execute the program like any other program -- by double-clicking on the icon.

Once launched you will see the development environment as shown in Figure 1. We will cover many aspects of this environment throughout the lab. Figure 1 shows the environment when in *Editor mode* -- the mode where you write programs. Figure 2 shows the environment in *Execute mode* -- the mode that displays when running and debugging programs.

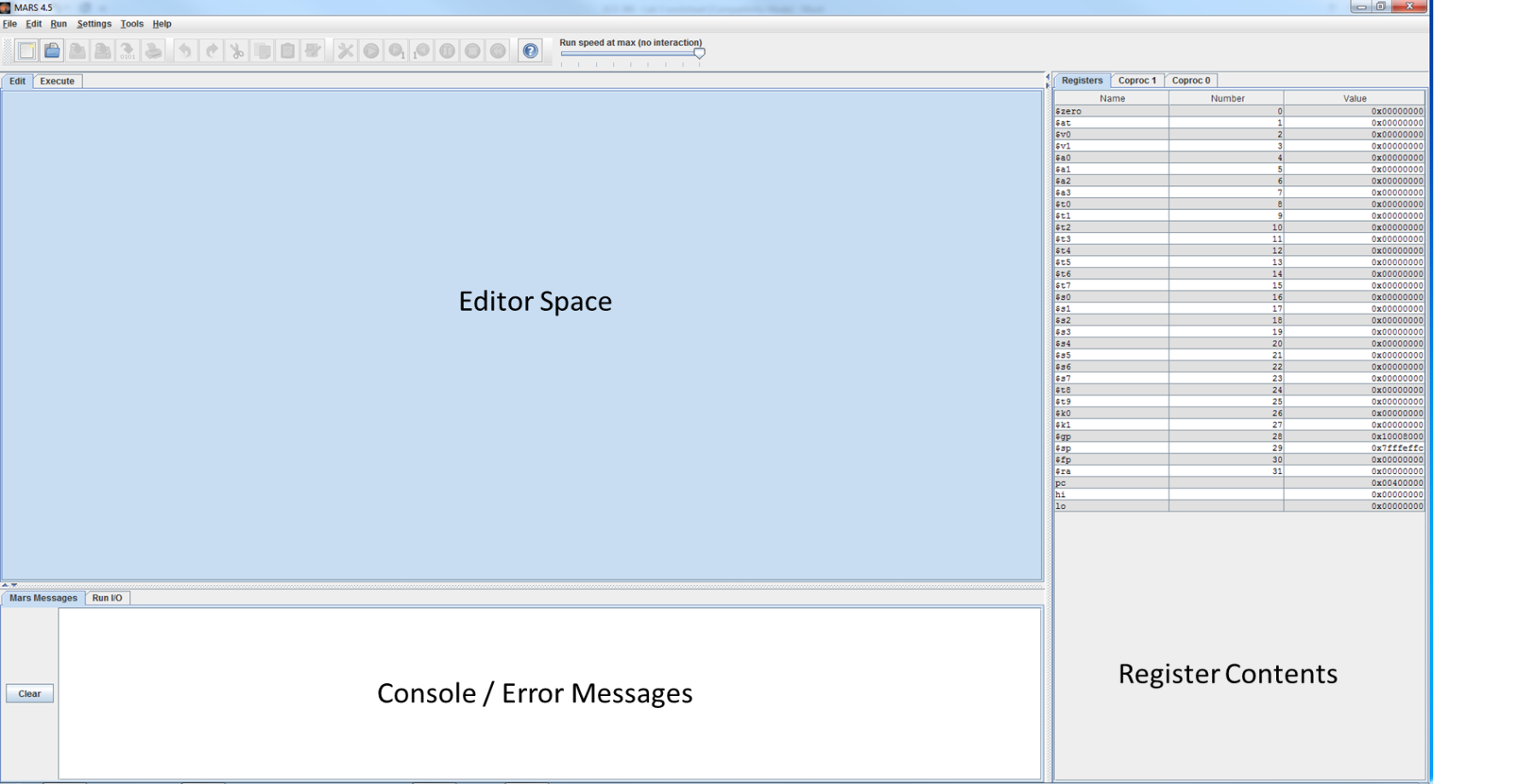


Figure 1: MARS Development Environment in Edit Mode

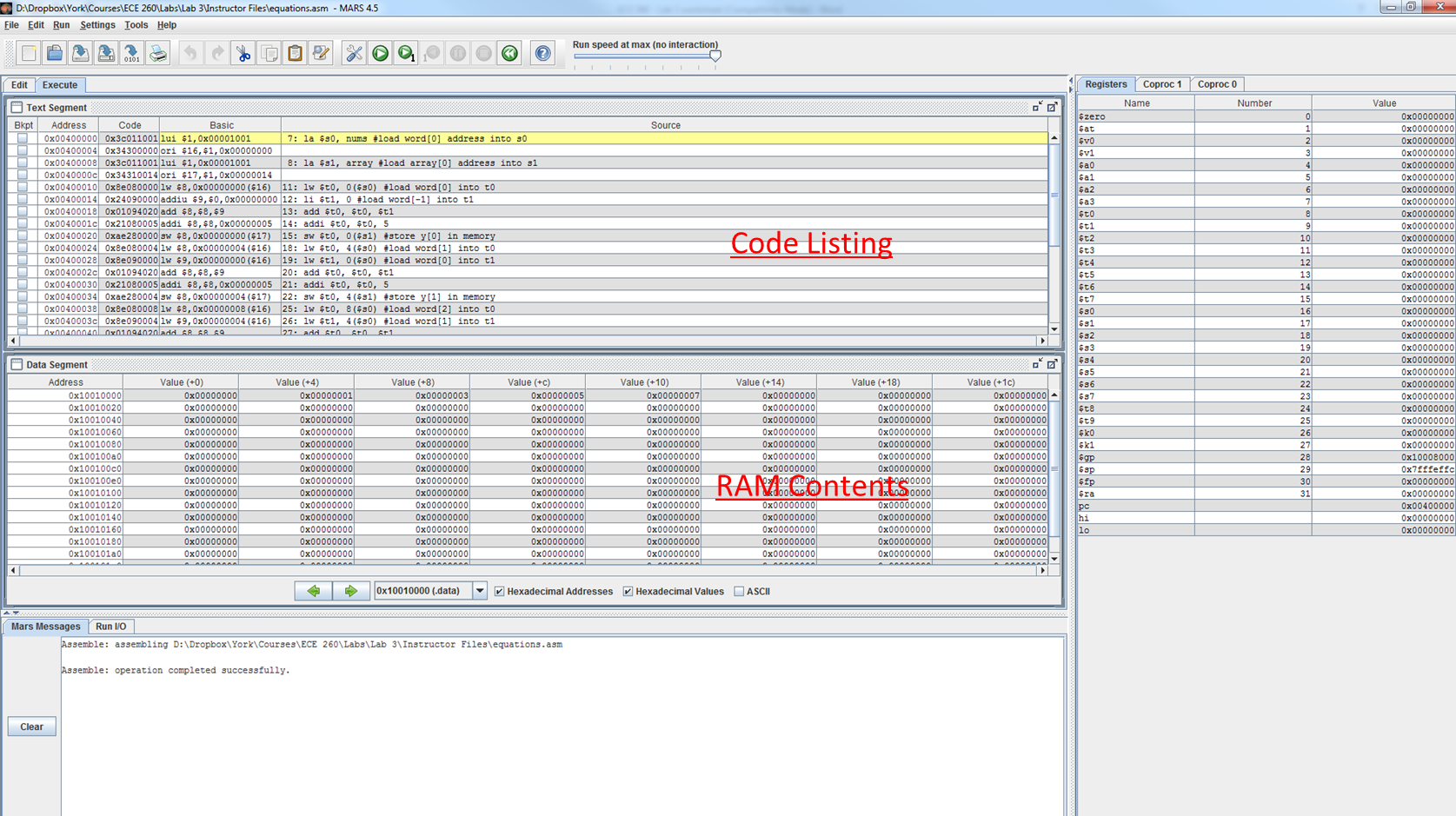


Figure 2: MARS Environment in Execute Mode

When writing MIPS assembly programs in Editor mode the environment will try to be “helpful” and suggest command completions. Note that some of the suggested instructions are ***pseudo-instructions*** and are not real MIPS instructions. Pseudo-instructions are covered in more detail later in this lab.

Once your program is written you can assemble and run it from the “Run” menu (or press F3). Any errors will be printed in the “Mars Messages” console window. When executing, the “Text Segment” window displays how your program is stored in memory, each instruction formatted as hexadecimal, and a representation of the code that you wrote. Note that, if you use pseudo-instructions when writing your assembly code, those pseudo-instructions will display in the “Source” column of the “Text Segment” window. The “Basic” column will display what your pseudo-instructions were converted into by the assembler.

After assembling you can run your program directly (F5) or debug step-by-step (F7). Until you are **EXTREMELY** proficient with MIPS assembly, you should use F7 to run your programs step-by-step. The contents of your registers and RAM will change as your program runs. All values in the registers or RAM can be represented as ASCII, hex, or decimal values by selecting the checkboxes at the bottom of the “Data Segment” window (shown in Figure 3).

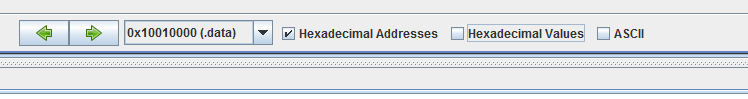


Figure 3: Checboxes to change data and address representation

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **3. Procedure:** This lab has multiple parts. For each part, write your code in the specified file (all of the required .asm files are in the ***src* directory**). Writing your code in the wrong file will cause problems when you test your code.  **3.1: Introduction to MIPS: Adding Two Numbers**  **Programming Tasks**   1. Open the file called lab03\_part1.asm in the MARS Editor. 2. At the top of the file you will see the “.globl” directive. It exports the “ece260\_main” label to make it accessible outside of this file. The “.text” section is where all of your instructions go. Finally, the label “ece260\_main”, serves as an easy way to reference a section of code (similar to a function, but different). 3. In the “.text” section, use **two** *addi* instructions to perform the operation 200 + 100. You should not need any additional instructions, only the two *addi* instructions. Store the result in the **$s1** register. **DO NOT MODIFY ANY OTHER REGISTER VALUES.** Note that registers have unknown initial value, so only use a register as a source operand if you know the contents. HINT: Consider using the **$zero** register to get the first value into a register. 4. Assemble and run your program and answer the following questions:  |  | | --- | | **Question 1:** Examine your code after compilation in the Execute tab. Specifically, look at the “Basic” column to see how MARS translated your code. What value does register name $s1 get translated into? How does this compare with the “Register Name” table at the bottom of page 1 on your MIPS Green Sheet?  [$s1 gets translated into $17. It matches the assigned number in the table, which is $17.] | | **Question 2:** Examine the “Text Segment” window for your assembled program. What is the origin address of your program in memory? That is, at what memory address is your first instruction stored? Where is the second instruction stored?  [The origin address of the program is 0x00400000. The second instruction is stored at 0x00400004.] | | **Question 3:**  Execute your program. What is the value of $s1 after execution? Write the value in hex. Does $s1 contain the value you expect? HINT: You can use the checkboxes at the bottom of the “Data Segment” window to change how data is displayed.  [The value is 0x201100c8. Yes, it does contain the value I would expect] |   **Test Your Code**  After you have answered the above questions, you can test your code to ensure that it satisfies the requirements described above. To run the supplied unit tests, open a Cygwin shell and type the following:  cd h: cd ECE260 cd Lab03\_Intro\_to\_MIPS\_Assembly make test\_part1  You will see output that indicates if your code passed or failed the included unit tests. If your code did NOT pass the units tests, address any errors and try running the unit tests again.  **3.2: Introduction to MIPS: Adding Signed Numbers**  **Programming Tasks**   1. Open the file called lab03\_part2.asm in the MARS Editor. 2. In the “.text” section, use **two** *addi* instructions to perform the operation -200 + 100 (just like part1, except now negative 200). You should not need any additional instructions, only the two *addi* instructions. Store the result in the **$s1** register. **DO NOT MODIFY ANY OTHER REGISTER VALUES.** Note that registers have unknown initial value, so only use a register as a source operand if you know the contents. HINT: Consider using the **$zero** register to get the first value into a register. 3. Assemble and run your program and answer the following questions:  |  | | --- | | **Question 4:** After replacing the value 200 with -200 in your program, what is the new value in $s1 in hexadecimal? Enter this hex value in your windows calculator and convert. Does this value result in -100? Why or why not.  [The value is 0xFFFFFF9C, which is negative 100 because if you use 2’s Complement and add 1 to the converted binary value you get 100.] |   **Test Your Code**  After you have answered the above questions, you can test your code to ensure that it satisfies the requirements described above. To run the supplied unit tests, open a Cygwin shell and type the following:  cd h: cd ECE260 cd Lab03\_Intro\_to\_MIPS\_Assembly make test\_part2  You will see output that indicates if your code passed or failed the included unit tests. If your code did NOT pass the units tests, address any errors and try running the unit tests again.  **3.3: Instruction to Pseudo-instructions**  **Programming Tasks**   1. Open the file called lab03\_part3.asm in the MARS Editor. 2. Copy your code from lab03\_part1.asm into lab03\_part3.asm. 3. Replace the first *addi* instruction with a “Load Immediate” instruction. The “Load Immediate” instruction has the format “*li destReg, imm”*. Assemble and run this new program. It should behave exactly as before.  |  | | --- | | **Question 5**: Examine your newly assembled program in the “Basic” column of the “Execute” tab. What happened to your *li* instruction? What instruction was it translated into? Use the MIPS Green Sheet to explain what happened.  [The li instruction is being translated to an add immediate operation because loading immediate is instructionally the same as adding immediate.] | | **Question 6**: From a human perspective, describe why it is useful to have a *li* pseudo-instruction.  [It’s helps lessen the number of variables and/or registers to keep track of.] | | **Question 7:** From a computer perspective, why is it useful to not have the *li* instruction and simply re-use *addi*?  [your answer here] |   **Test Your Code**  After you have answered the above questions, you can test your code to ensure that it satisfies the requirements described above. To run the supplied unit tests, open a Cygwin shell and type the following:  cd h: cd ECE260 cd Lab03\_Intro\_to\_MIPS\_Assembly make test\_part3  You will see output that indicates if your code passed or failed the included unit tests. If your code did NOT pass the units tests, address any errors and try running the unit tests again.  **3.4: Loading Data into Memory**  **Programming Tasks**   1. Open the file called lab03\_part4.asm in the MARS Editor. 2. At the top of this file there exists a space to create data values. The “.data” section provides a location in your code where you can populate memory with data values (i.e. arrays, strings, variables, etc.). These values will be stored in memory using the exact size and order that you specify. Replicate the “.data” segment shown below in lab03\_part4.asm.   .data  intarray: .word 0 1 3 5 7 11 13 17 19 23  empty: .space 10  hello: .asciiz "Hello World!"  goodbye: .asciiz “Goodbye”   |  | | --- | | **Question 8:** In the code above, each term that ends with a “:” is called a “label”. Labels are similar to pointers in that you can create labels with names of your own choosing. Each label references a memory location. Each of the terms that is prepended with a “.” is called a “directive” (e.g. .word, .asciiz, etc.). Use the MARS help file to determine what each of the above directives do. Press F1 on your keyboard to access help and then select “Directives” under the MIPS tab.   * [.word Store the listed value(s) as 32 bit words on word boundary] * [.space Reserve the next specified number of bytes in Data segment] * [.asciiz Store the string in the Data segment and add null terminator] | | **Question 9:** Assemble your program and use the Data Segment window to identify the starting address of intarray, empty, hello, and goodbye*.* Try toggling the “ASCII” checkbox at the bottom of the Data Segment window to help you find hello. Express these addresses in hexadecimal below.  [(intarray) 0x10010000]  [(empty) 0x10010020]  [(hello) 0x10010030]  [(goodbye) 0x10010038] | | **Question 10:** The directive “.space” allocates memory for the label named empty. How many bytes are being reserved? If intarray was copied into the space reserved for empty would there be sufficient space for the copy? Explain why or why not.  [10 bytes are being reserved and that would not sufficient to hold for intarray, which is in the .array directory. .word allocates 32 bits of memory.] | | **Question 11:** Compare how hello and intarray are stored in memory. You should notice a *significant* difference in byte ordering. Use [this article](http://teaching.idallen.com/cst8281/10w/notes/110_byte_order_endian.html) to identify the *endianness* of integers and character arrays in the MARS simulator.  [your answer here] | | **Question 12:** For this question, you will be adding some code to the .text section of lab03\_part4.asm. Use the pseudo-instruction *la* (load address) to load the address of intarrayinto $t0. Assemble this new program. Into what MIPS instructions does the assembler convert the pseudo-instruction *la* (check the Basic column on the Execute tab)? Step through the code one line at a time to get a better understanding (F3 to assemble, F7 to step).  [your answer here] | | **Question 13:** For the instructions generated in Question 12 (*lui* and *ori*) use your MIPS Green Sheet to explain what is going on. Why can’t we directly load a 32-bit memory address into a register using a single instruction? (Hint: the answer is related to word-size).  [your answer here] | | **Question 14:** In the .text segment that you created earlier, use three more *la* pseudo-instruction to load the addresses of empty, hello, and goodbye into $t1, $t2, and $t3 respectively. Assemble and run this new program. What value is loaded into registers $t1, $t2, and $t3? Do these values match your answer for question #9? If not, why not?  [your answer here] |   **Test Your Code**  After you have answered the above questions, you can test your code to ensure that it satisfies the requirements described above. To run the supplied unit tests, open a Cygwin shell and type the following:  cd h: cd ECE260 cd Lab03\_Intro\_to\_MIPS\_Assembly make test\_part4  You will see output that indicates if your code passed or failed the included unit tests. If your code did NOT pass the units tests, address any errors and try running the unit tests again.  **3.5: Writing a More Complete MIPS Program**  Use the abilities that you have learned in the previous sections of this lab to implement the program described below.  **Programming Tasks**   1. Open the file called lab03\_part5.asm in the MARS Editor. 2. Store the following integer sequence in memory as an array of words called nums, {1, 3, 5, 7, 9} 3. Reserve a space in memory called array immediately after nums that can hold five **words** 4. Store the hexadecimal value 0xDEADBEEF as a word called flag immediately after array 5. For each element (i=0 to i=4) in nums, perform the following calculation:   You do NOT need to write any type of loop, just write a simple block of code for each value of i. 6. When i=0, replace nums[i-1] in the above equation with 0 since *nums*[i-1] does not exist when i=0.   **HINTS:**   1. Start your code by using the *la* pseudo-instruction to store the addresses of nums and array into registers $s0 and $s1 respectively. You can then use these registers throughout the rest of your code as base registers for storing and loading data to and from memory. 2. Select another register (a register other than $s0 or $s1) and use that register as an accumulator. Sum up the values for the calculation in that register prior to writing the result to the array space memory. 3. The label nums is not labeled as an “array”. However, those label names are arbitrary anyway. The data specified by nums is stored in memory one word after another. Treat nums like an array -- access the data in nums as if it were an array. 4. Since this is your first assembly program, your code doesn’t need to be flexible or even support the ability to change the size of the nums array. Approach this program by figuring out how to do the computation on a single index value for i. Then you can copy/paste the code and make the appropriate changes for the other indices. That is, you do not need to create a loop.   Complete your program and list the contents of the Data segment in the table below. All cells should have a value except those that are grayed out. Write your answers in decimal.  Table 1: Data segment for MIPS program   |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | | **Address** | **0** | **+4** | **+8** | **+c** | **+10** | **+14** | **+18** | **+1c** | | **0x10010000** | ?? | ?? | ?? | ?? | ?? | ?? | ?? | ?? | | **0x10010020** | ?? | ?? |  |  |  |  |  |  | |

**Test Your Code**

After you have answered the above questions, you can test your code to ensure that it satisfies the requirements described above. To run the supplied unit tests, open a Cygwin shell and type the following:

cd h:  
cd ECE260  
cd Lab03\_Intro\_to\_MIPS\_Assembly  
make test\_part5

You will see output that indicates if your code passed or failed the included unit tests. If your code did NOT pass the units tests, address any errors and try running the unit tests again.

**4. Submission**

When you have finished your lab, demo your program for your instructor. Write your answers to the above questions electronically in this document. To submit your lab assignment, make sure your files have all been saved (*including this file*). In a Cygwin window type the commands:

cd h:  
cd ECE260  
cd Lab03\_Intro\_to\_MIPS\_Assembly  
make submit

Enter your Marmoset username and password (which you should have received by email). Note that your password will not be echoed to the screen. Make sure that after you enter your username and password, you see a message indicating that the submission was successful. Log into [Marmoset](https://cs.ycp.edu/marmoset/login) via the web to check the files you submitted to ensure they are correct.

**DO NOT MANUALLY ZIP YOUR PROJECT AND SUBMIT IT TO MARMOSET.  
YOU MUST USE THE make submit COMMAND.**