**ECE 260: Fundamentals of Computer Engineering – Lab #4  
Branching and Conditional Assembly**

**Name \_\_\_\_Le Quient Lewis\_\_\_\_\_\_ Lab Partner(s) \_\_\_\_\_\_\_N/A\_\_\_\_\_\_\_\_\_\_\_**

**1. Introduction**: This lab provides students practice with writing conditional and looping programs in assembly. Students will also explore various pseudo-instructions that can be used for this purpose.

**2. Background**: In lecture we discussed two new types of instructions *conditional branches* and *unconditional branches*. Conditional branches (*beq* and *bne*) will move the program counter to a specified memory location based upon a == or != comparison. These instructions take two registers as operands and a *label* that correlates to a memory location in the *text* segment of memory. A jump ( *j* ) instruction is an unconditional branch that immediately jumps to a location within the program.

Conditional and unconditional branch instructions can be used to implement *if-statements*, *while-loops*, and *for-loops*. Two additional instructions are also useful to aid in implementing looping operations, *set-on-less-than* (*slt* rd, rs, rt) and *set-on-less-than-immediate* (*slti* rt, rs, imm). Each operation does a comparison (rs < rt) or (rs < imm) and sets the destination register to 1 if the condition is true, or to 0 otherwise. These operations will be helpful in implementing comparisons such as (i<3) or (i>=x). Consult the MIPS Green Sheet for more information on the unsigned variations of this instruction.

In this lab you will implement conditional operations. Chapter 2.7 of your text book may be helpful in your efforts.

**3. Procedure**

This lab has several parts. Each part is designed to show you how to implement various conditional operations. In general, you may not use pseudo-instructions unless explicitly permitted. You can distinguish a “real” or native instruction from a pseudo-instruction in several ways: (1) pseudo-instructions are explicitly listed on the MIPS Green Sheet as such and (2) when compiled pseudo-instructions are converted into one or more “real” instructions (shown in the Basic column of the Execute window).

**3.1 Implementing *if-statements* Using the SLT Instruction**

**Programming Tasks**

1) In MARS, open the file called lab04\_part1.asm. Write an assembly program to implement the *if-else-statement* shown in Figure 1. You must use the *slt* instruction for this part, **do not** use the *slti* instruction (you’ll do that later). You will see two load immediate (*li*) pseudo-instructions have already been provided to pre-load the integer values 7 and 5 into registers **$t0** and **$t1** respectively. **Keep your final result for *value* in register $t0.** You may use any other registers you feel necessary to finish implementing your code.

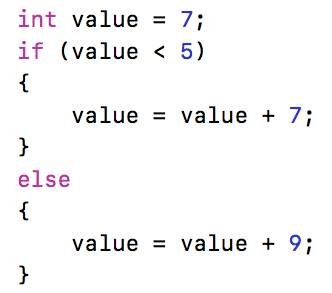


Figure 1: *if-else-statement* for Section 3.1 and 3.2

2) After you have implemented your program, test out your solution by trying different initial values in lieu of 7. Using the initial values in Table 1, fill in the expected output values based on the C-code, and the actual output value that your MIPS code produces.

Table 1: Results for *less-than* (<) *if-else-statement*

|  |  |  |
| --- | --- | --- |
| **Initial Value** | **Expected Value (after execution)** | **Real Value (after execution)** |
| **3** | 10 | 10 |
| **4** | 11 | 11 |
| **5** | 14 | 14 |
| **6** | 15 | 15 |

**Test Your Code**

You should test your code with different initial values to ensure that it satisfies the requirements described above. Then, test your code using the supplied unit tests. To run the supplied unit tests, open a Cygwin shell and type the following:

cd h:  
cd ECE260  
cd Lab04\_Branching\_and\_Conditional\_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 Implementing *if-statements* Using the SLTI Instruction**

1) In MARS, open the file called lab04\_part2.asm. Once again, write an assembly program to implement the *if-else-statement* shown in Figure 1. However, this time use the *slti* instruction. You will see a single load immediate (*li*) pseudo-instruction has already been provided to pre-load the integer value 7 into register **$t0**. **As before, keep your final result for *value* in register $t0.** There is one fewer load immediate (*li*) instruction in the provided code than there was in the previous part. One fewer load instruction means faster code. It also frees up a register that can be used for other data.

2) After you have implemented your program, test out your solution by trying different initial values in lieu of 7. Run your new program with the values in Table 1 to check that your new program works the same as your program from section 3.1.

3) Compare your implementations from part 1 and part 2 – one uses the slt instruction and the other uses the slti instruction. How many instructions does each implementation require (yes, you should count the *li* instructions that were provided for you)?

|  |
| --- |
| [There were 6 instructions in part 1 and 5 in part 2] |

**Test Your Code**

You should test your code with different initial values to ensure that it satisfies the requirements described above. Then, test your code using the supplied unit tests. To run the supplied unit tests, open a Cygwin shell and type the following:

cd h:  
cd ECE260  
cd Lab04\_Branching\_and\_Conditional\_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 Implementing *if-statements* with *less-than-or-equal-to* (<=) Using the BEQ/BNE Instructions**

1) In MARS, open the file called lab04\_part3.asm. Write an assembly program to implement the *if-else-statement* shown in Figure 2. As was demonstrated in part 1, use the load immediate (*li*) pseudo-instruction to initialize registers **$t0** and **$t1** with the values 7 and 13 respectively. **Do not use any pseudo-instructions, other than *li* for this part**. The initialization code should be placed under the “initialize” label, but before the “ece260\_main” label. Don’t put any additional code in the “initialize” section. Put the remainder of your code below the “ece260\_main” label. **As before, keep your final result for *value* in register $t0.** Your entire solution should require only 8 instructions.

**HINT:** To implement the less than or equal to condition (<=) it is recommended that you break that statement into two different tests – one test for equality and one test for less than. You will want to branch to the TRUE *if* block if either of the tests pass and simply fall into the FALSE *else* block if both tests fail. Alternatively, you can take the inverse of <= and branch to the else statement when the inverse is TRUE.

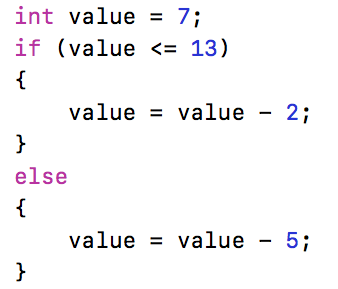


Figure 2: *if-statement* for Section 3.3 and 3.4

2) After you have implemented your program, test out your solution by trying different initial values in lieu of 7. Using the initial values in Table 2, fill in the expected output values based on the C-code, and the actual output value that your MIPS code produces.

Table 2: Results for *less-than-or-equal*-to (<=) *if-else-statement*

|  |  |  |
| --- | --- | --- |
| **Initial Value** | **Expected Value (after execution)** | **Real Value (after execution)** |
| **12** | 10 | 10 |
| **13** | 8 | 8 |
| **14** | 9 | 9 |
| **15** | 10 | 10 |

**Test Your Code**

You should test your code with different initial values to ensure that it satisfies the requirements described above. Then, test your code using the supplied unit tests. To run the supplied unit tests, open a Cygwin shell and type the following:

cd h:  
cd ECE260  
cd Lab04\_Branching\_and\_Conditional\_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 Implementing *if-statements* with *less-than-or-equal-to* (<=) Using the BLE Pseudo-Instruction**

1) In MARS, open the file called lab04\_part4.asm. Once again, write an assembly program to implement the *if-else-statement* shown in Figure 2. However, this time use the BLE (branch if less than or equal) pseudo-instruction. The structure of your code may be similar to part 3.3 (i.e. your *else* block may come first in your code, and you will need to branch to the *if* block). As was demonstrated in earlier parts, use the load immediate (*li*) pseudo-instruction to initialize registers **$t0** and **$t1** with the values 7 and 13 respectively. The initialization code should be placed under the "initialize" label, but before the "ece260\_main" label. Don't put any additional code in the "initialize" section. Put the remainder of your code below the "ece260\_main" label. **As before, keep your final result for value in register $t0.**

2) After you have implemented your program, test out your solution by trying different initial values in lieu of 7. Run your new program with the values in Table 2 to check that your new program works the same as your program from section 3.3.

3) Compare the number of instructions between your implementation in Section 3.3 and this implementation. How many instructions does each solution use?

|  |
| --- |
| [10 instructions in part 1 and 8 instructions in part 2] |

4) In the *Execute* tab of MARS, examine the instructions produced by the BLE pseudo-instruction. How many instructions are produced? How do these new instructions compare to the original C-code?

|  |
| --- |
| [There are 10 instructions in part 1 and 9 instructions in part 2] |

**Test Your Code**

You should test your code with different initial values to ensure that it satisfies the requirements described above. Then, test your code using the supplied unit tests. To run the supplied unit tests, open a Cygwin shell and type the following:

cd h:  
cd ECE260  
cd Lab04\_Branching\_and\_Conditional\_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 Implementing *if-statements* with *greater-than-or-equal-to* (>=) Using the SLTI/BEQ Instructions**

1) In MARS, open the file called lab04\_part5.asm. Write an assembly program to implement the *if-else-statement* shown in Figure 3. As was demonstrated in previous parts, use the load immediate (*li*) instruction to initialize register **$t0** with the value 7. **Do not use any pseudo-instructions, other than *li* for this part**. The initialization code should be placed under the “initialize” label, but before the “ece260\_main” label. Don’t put any additional code in the “initialize” section. Put the remainder of your code below the “ece260\_main” label. **As before, keep your final result for *value* in register $t0.**

**HINT:**  Since *greater-than-or-equal* is the logical negation of *less-than* it is straightforward to implement >= using the *slti* and *beq* instructions.

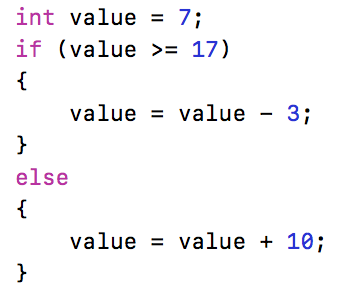


Figure 3: *if-statement* for Section 3.5

2) After you have implemented your program, test out your solution by trying different initial values in lieu of 7. Using the initial values in the Table 3, fill in the expected output values based on the C-code, and the actual output value that your MIPS code produces.

Table 3: Results for *greater-than-or-equal-to* (>=) *if-else-statement*

|  |  |  |
| --- | --- | --- |
| **Initial Value** | **Expected Value (after execution)** | **Real Value (after execution)** |
| **15** | 25 | 25 |
| **16** | 26 | 26 |
| **17** | 14 | 14 |
| **18** | 15 | 15 |

**Test Your Code**

You should test your code with different initial values to ensure that it satisfies the requirements described above. Then, test your code using the supplied unit tests. To run the supplied unit tests, open a Cygwin shell and type the following:

cd h:  
cd ECE260  
cd Lab04\_Branching\_and\_Conditional\_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.

**3.6 Implementing a *for-loop* to Find the Average of an Array of Integers**

By this point you have seen the various ways to express conditional statements using assembly. All conditionals, whether *if-statements*, *for-loops*, or *while-loops*, are variations on the examples that you implemented in the previous sections. In this final section of the lab you must write assembly code to take the average of an array of integer values. You may use any instructions available within MARS (native instructions or pseudo-instructions).

1) In MARS, open the file called lab04\_part6.asm. In this file, write an assembly program to implement the *for-loop* shown in Figure 4. For the example code shown in Figure 4, the final *average* value should be **5**. First, create a word-sized integer value called *len* to represent the number of integers that will be stored in an array called *nums*. Also, in the .data section, create the array of word-sized integers called *nums* using the values shown in Figure 4.

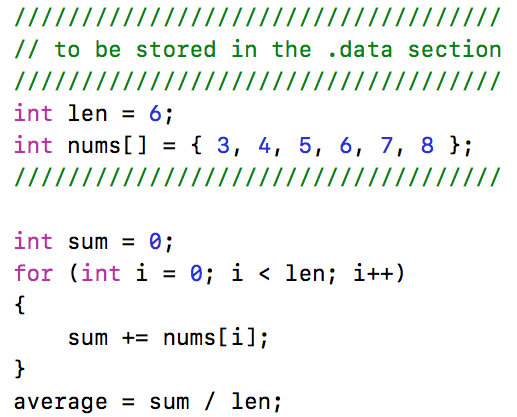


Figure 4: *for-loop* for Section 3.6

2) Note that there are three labels already provided for you in the .text section of the lab04\_part6.asm file, the “initialize” label, the “loop” label, and the “end” label. You will likely need at least one more label to which you can branch when you want to terminate your *for-loop*. **Do NOT put any instructions after the “end” label**.

3) Load the value for *len* and the base address of *num* into registers the “initialize” section. Also initialize the values for *sum* and your loop variable *i* in the “initialize” section. To ensure that the automated tests perform as expected, use the following registers for each of these values: store *len* in **$s0**, store the base address of *nums* in **$s1**, store the final *average* in **$s2**, store *sum* in **$t0**, and store *i* in **$t1**.

**HINT:** You will want to look up and use the load address (*la*) instruction. It can be used to load the base address of your *nums* array into register **$s1**. Additionally, it can be used immediately before a load word (*lw*) instruction to load the value of *len* from memory into register **$s0**.

**ANOTHER HINT:**  You will find it easiest to use the logical negation of the less-than condition (i.e. greater than or equal to) to branch out of the loop when necessary. Use the *bge* pseudo-instruction to simplify your code.

**YET ANOTHER HINT:**  Look up and use the divide (*div*) instruction to compute the final value for *average*.

4) Are you sure that your code is correct? Try setting *len* to 0 and stepping through your code to ensure that the body of your *for-loop* never executes. Of course, when *len* is 0, your program when eventually crash when you attempt to divide by 0… unless you check for that sort of thing.

**Test Your Code**

You should test your code with different initial values to ensure that it satisfies the requirements described above. Then, test your code using the supplied unit tests. To run the supplied unit tests, open a Cygwin shell and type the following:

cd h:  
cd ECE260  
cd Lab04\_Branching\_and\_Conditional\_Assembly  
make test\_part6

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 Lab04\_Branching\_and\_Conditional\_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.**