COSC 211 Lab 3 – Logical Operators and Decision Making

(DUE October 6th, 2023 – end of day)

0.Introduction

In this lab, you will have the opportunity to explore using the different logical operations available on the MIPS architecture to isolate and manipulate data. You will also gain experience with branching instructions that allow you to build loops and decision statements. Additionally, there are a series of questions from the text book. There is a total of 47 marks available for this assignment.

The assignment is broken into two parts. The written submission can be hand written and scanned or typed. Work must be clear with all work shown. The coding part of the assignment requires two submissions. Your code and written submission are to be submitted through Canvas by **the end of the day on Friday, October 6th**. Submissions will be accepted earlier. Late submissions will not be accepted. Even if you do not complete all questions, please try them all and submit what you have for partial marks.

1.Programming Assignment

Bit Manipulation

In assembly programming, logical operations are a key set of tools. These operations allow for low level manipulation and determination of the values of individual bit fields. Masking is a powerful technique that allows a programmer to isolate the value of an individual bit. This is useful when you want to decompose a number. Consider the following binary number:

```
1101 0001   (original number)
```

In order to determine if the value of the 3rd bit is set to 1 or 0, we can mask out the rest of the bits using a bit-by-bit AND operation and a shift left operation. To start with, we build a mask which has a 1 in the position of the bit we want to isolate and 0's in every other position. In this case, we wish to isolate the 3rd bit so our mask would appear as:

```
0000 1000   (mask)
```

To isolate the bit, we and the mask with our original number which results in all other bits being set to zero giving results as:

```
0000 0000   (final)
```

At this point, if the value is greater than 0, then you know that the value was set to 1 or it the value is 0, then the bit was zero. A safer strategy is to introduce another operation where we take the final value and shift it to the left by the number of positions that you wanted to isolate. This will move the bit of interest into the 0 bit

position and you can simply check to see if the value is equal to 0 if it was not set. Conversely, the value will be equal to 1 if the bit is set. This can be done using the `beq` instruction as shown in the following code segment:

```
                li $t0, 24              # assume that this is the number that you
                                        # want to test

                li $t1, 8               # load your mask (bit 3)
                and $s0,$t0,$t1         # mask out the bit of interest

                srl $s1,$s0,3           # now isolate the bit by shifting it to
                                        # the left

                #and test
                beq $s1,$zero,isclr     #if it is zero, branch to message

isset:          #do something if it is set
                j cont                  # and exit statement

isclr:          # do something else if the bit is not set(cleared)

cont:           #carry on with your program
```

This framework demonstrates how simple logical and shift commands can be used to isolate a bit, but we can also use them to set or clear a specific bit in a number. If we want to set a specific bit in a number, we can use the masking concept again but this time we will use the bit-by-bit OR operation. If we want to set the value of the 2nd bit in a number

```
1101 0001   (original number)
```

We can again use our mask with a 1 in the second bit place (starting from 0):

```
0000 0100   (mask)
```

but we OR the two numbers together. This will result in a final value of

```
1101 0101   (final number)
```

Conversely, we may need to clear a bit in a number which involved a bit more work. A mask is still used but requires some inversion. If we want to set the value of the 4th bit in a number

```
1101 0001   (original number)
```

we again use our mask with a 1 in the fourth bit place (starting from 0):

```
0001 0000   (mask)
```

but before we can use this mask for clearing a bit we need to invert is using the NOT operation. This will cause all the bits to switch state leaving our mask as

```
1110 1111   (inverted mask)
```

This inverted mask can then be AND'd with the original value resulting in the final value of

```
1100 0001   (final number)
```

In summary, bitwise logical operations and shifting are powerful tools for the manipulation and interpretation of bit values in assembly.

Branching

In assembly programming, branching is the primary way to control the flow direction of your program. There are two different types of branching that we encounter: **conditional branching** and **unconditional branching**. Conditional branching is analogous to a decision making statement in a higher level language such as an *if-else* statement. In reality, it is more closely aligned to an *if* statement combined with a *goto* statement. In the MIPS architecture there are two conditional branching statements that are available, that being the **`beq`** (branch if equal) and the **`bne`** (branch if not equal). Both of these instructions compare the contents of two registers and then branch to a destination label in your code base on the evaluation condition.

For example, if you wanted to evaluate two registers and branch to a location if they were equal to zero, the code would look like this:

```
here:       beq   $t0,$t1,there     #compares $t0 and $t1
                                    #and branches to there if
                                    #they are equal
            #more code...

there:      #code at destination
```

If $t0 and $t1 are equal, then the code will continue to execute at the **there** label, otherwise the code will carry on at the next instruction after the `beq`. The same can be accomplished with the `bne` operation except that the branch will occur if the two registers are not equal:

```
here:       bne   $t0,$t1,there     #compares $t0 and $t1
                                    #and branches to there if
                                    #they are not equal
            #more code...

there:      #code at destination
```

You can use these instructions to build an *if-else* statement but you will need one more instruction. In the above two examples, regardless of the branch of execution taking, the code will always run the code at the **there** label. This is generally not how an *if-else* statement functions. With an *if-else* statement you execute one branch of code or the other, and then regroup at a common point. To support this functionality, you also need to use an unconditional branch to jump around the 'other' section of code. The statement to do this is **`j`** (jump). It is called unconditional because it will always jump to the destination. An *if-else* can be constructed as in the following example:

```
if:         bne   $t0,$t1,else      #compares $t0 and $t1
                                    #and branches to there if
                                    #they are not equal
            #if code
            j exit
else:       #else code

exit:       #the rest of the program
```

The last common thing that can be done with conditional and unconditional branches is loops. You can use the conditional branch to determine if it is time to exit the loop and the unconditional branch to jump back to the evaluation point after the code has run. Here is an example:

```
loop:       beq   $t0,$zero,exit    #compares $t0 and $zero (0)
                                    #and branches to there if
                                    #they are equal
            subi  $t0,$t0,1         #subtract 1
            #some code….
            j loop

exit:       #code at destination
```

There are many ways to formulate *loops* and *if-else* statements with assembly. Some offer better performance speeds due to better register utilisation or fewer instructions and it will take practice to recognise the patterns. When dealing with complex instructions, it is best to plan out how your code is going to work and then build up your code is blocks which will make it easier for testing.

Programming Exercise

1. (/22) Write a MIPS program that reads from the keyboard an 8-bit binary number in ASCII where the only input is 0 or 1. The program will have to convert the string you have read in, into a numeric value. When a string is read in, it is stored as a numeric value in memory using the ASCII mapping (http://en.wikipedia.org/wiki/ASCII). Keep in mind that the character 0 has as ASCII value of 48_{10} or 0x30. The character 1 has as ASCII value of 49_{10} or 0x31. You do not have to worry about handling any other characters. The input will always be 8 characters long. The program will then output the value in both base 10 and base 16 representations. Assume the value is unsigned. The sample output should appear as:

   ```
   Enter your number (1 or 0): 11111111
   The number in base 16 is: 0xFF
   The number in base 10 is: 255
   ```

   In you will have to use logical operators, shifts and branching to successfully implement this program. Include a program comment header that has your name, student number and program name. **Name your file excercise1.s.**

2. (/12) Write a MIPS program that reads from the keyboard an integer number where the only input is between 0 and 255. The program will then output the value as a binary number represented as a string of 0's and 1's. Assume the value is unsigned. The sample output should appear as:

   ```
   Enter your number (base 10): 36
   The number in base 2 is: 00100100
   ```

   In you will have to use logical operators, shifts and branching to successfully implement this program. Include a program comment header that has your name, student number and program name. **Name your file excercise2.s.**

Programming Submission

For questions 1 and 2, submit both your .s files through Canvas. All files should be in a single .zip file. Submit your files through Canvas before the assignment due date. Paper copies of the .s files only are due along with the written assignment on the above indicated due date.

2. Written Assignment

3. (/13) Instructions: Language of the Computer (Show all work. Part marks will be assigned for all questions)

Complete the following questions (Textbook, pages 164-174):

i. 2.8 (1 mark)
ii. 2.12.1 (1 mark)
iii. 2.12.2 (1 marks)
iv. 2.12.3 (1 mark)
v. 2.12.4 (1 marks)
vi. 2.12.5 (1 mark)
vii. 2.12.6 (1 marks)
viii. 2.14 (2 mark)
ix. 2.15 (1 mark)
x. 2.19.1 (1 mark)
xi. 2.19.2 (1 mark)
xii. 2.19.3 (1 mark)

Written Submission

Submit your written assignment in a text file or scanned on the above indicated due date. Please take care and effort to make your assignment answers as clear as possible.