ECE 243 - Computer Organization
January 2024
Lab 3

Logic Instructions, Subroutines and Memory Mapped Output

Due date/time: During your scheduled lab period in the week of January 29, 2024. [Due date is not when Quercus indicates, as usual.]

Learning Objectives

The goal of this lab is to cover the following concepts/skills: using the logic and shift instructions to access single bits, understanding how to make a subroutine, and how transfer of control works, as well as parameter passing, what memory-mapped output is, and an introduction to *time* in the form of software delay loops. This, together with practice in basic assembly programming, accessing memory through loads and stores **and** those crucial debugging skills that are the key pathway to becoming an engineer!

What To Submit

You should hand in the following files prior to the end of your lab period. However, note that the grading takes place in-person, with your TA:

- The assembly code for Parts I,II, III and IV in the files `part1.s`, `part2.s`, `part3.s` and `part4.s`.

Part I

You are to write a Nios II assembly language program that will count the number of ones in the binary number representation of a given input 32-bit word. For example, if the word had the value `0x4a01fead` (hex/base 16) then the result would be 16 (in base 10). The "input" to your code will be set with an assembler directive (.word) as shown in the skeleton code given in Figure 1, where the assembler directive sets word at address `InputWord` to be have the value `0x4a01fead`. Your program must work correctly for any value placed there. The result (the number of 1's in the input), must be stored into the word `Answer`, which is also defined in Figure 1.

For preparation, you are to write, run and debug your code for the above problem on CPUlator. Submit your code to this Quercus assignment in a file named `part1.s`.

During the lab period, run your code on the DE1-Soc Nios II computer system (not CPUlator) and make sure it works. Show your TA that it works, by showing how the correct answer arrives in the memory location `Answer`.

```
.text
/* Program to Count the number of 1's in a 32-bit word,
located at InputWord */

.global _start
_start:

/* Your code here  */

endiloop: br endiloop

.data
InputWord: .word 0x4a01fead

Answer: .word 0
```

Figure 1: Skeleton Code for Part I

Part II - Your First Assembly Subroutine

In this part you are to make the code you've created in part I into a subroutine that is called from a main program. (The 'main program' is what you've been creating all along in your assembly programs; it is the part at the top, after which you would put things like this subroutine. The subroutine(s) come before the data section, typically.

1. Take the part I code which calculates the number of 1's in a word and make it into a subroutine called ONES. Have the subroutine use register `r4` to receive the input data to be processed and use register `r2` to return the resulting count. Be sure to do this, as we are carefully (but slowly) introducing you to rules that make subroutines (and subroutines that call subroutines) work correctly.

2. Now that you've created the subroutine ONES, use it to replicate the program from part I. That is, you will have a main program that simply transmits the value to be processed (have its 1's counted) in register `r4` and retrieves the result after that routine returns, and then places the result into memory as before.

Part III - Using the ONES Subroutine Twice in a Loop

In this part you'll use that same subroutine in a more complex way, and will call it twice, within a loop. One trouble you will likely run into is that your main program and the subroutine might be using the same registers for different purposes, and so you'll need to be careful not to over-write registers that will make your program fail.

We suggest using the stack to save and restore the registers that you use in the subroutine that are also used, for different purposes, in the main program. Later on in the course, we will describe a set of rules to apply to all of your main programs and subroutines that, when followed, prevent this kind of problem from happening.

1. Revise the main program part of your code from part II to have more words in memory (to be "measured") starting from a label called `TEST_NUM`, as shown in Figure 2. You can add as many words as you like, but include at least 10 words. Your program can detect the end of the list with a word value of 0, as illustrated in Figure 2, and provided to you in this lab in the file `part3_skel.s`. This is similar to the termination of the list of student numbers in Lab 2.

2. In your main program, call the subroutine in a loop for every word of data that you placed in memory. Keep track of the largest number of 1's in any of the words, and have this result placed in the memory location `LargestOnes` after the program ends.

3. In that same loop, make use of the same subroutine to count the number of 0's in the binary representation of these numbers. (Do not compute this value based on the output from ONEs, but rather change the input to the subroutine ONEs so that it directly computes the number of ZEROES. Hint: make use of the `xor` instruction to do this. Have this result placed in the memory location `LargestZeroes` after the program ends.

```
.text
/* Program to Count the number of 1's and Zeroes in a sequence of 32-bit words,
and determines the largest of each */

.global _start
_start:

/* Your code here  */

endiloop: br endiloop

.data
TEST_NUM:  .word 0x4a01fead, 0xF677D671,0xDC9758D5,0xEBBD45D2,0x8059519D
           .word 0x76D8F0D2, 0xB98C9BB5, 0xD7EC3A9E, 0xD9BADC01, 0x89B377CD
           .word 0  # end of list

LargestOnes: .word 0
LargestZeroes: .word 0
```

Figure 2: Skeleton Code for Part III

Put comments in your code to explain it. Submit your code on Quercus, in a file named `part3.s`.

During the lab, test and run your program on the DE1-SoC Nios II computer, and show it working to your TA.

Part IV - Display Output on LEDs

Modify your program for Part III to do the following: it should display the low-order 10 bits of the two resulting numbers (LargestOnes and LargesZeroes) on the LEDs, in an infinite loop, one after the other.

However, if you do that, you'll find there is a problem - that the numbers are displayed too quickly too see. To solve this you'll need to put what is called a *software delay loop* in between the statements that cause the output to appear on the LEDs, through memory-mapped output. This is a loop whose only purpose is to wait for an amount of time that makes these outputs visible. Put your delay loop into another subroutine that you can re-use, as you'll need to do to call it twice (likely).

Make this work well on CPUlator, by choosing a good value for the number of iterations of the delay loop. When you run your code on real hardware in the lab, you will likely find that it operates at a different speed than the simulation in CPUlator. So, you may have to change the number of loop iterations. Once you've done that, state which of the two (CPUlator or the real processor) is faster, and why. Include your answer to that question as part of your comments to the code you submit for this part, in the file `part4.s`.

Part V - Bonus Learning

Take a look at this video, of a car from a 1980s TV show with a funky-looking front LED scanning back and forth: https://www.youtube.com/watch?v=54O_1mOab4Y. Using the knowledge gained/used in this lab, make the LEDs on the DE1-SoC board do that. The bonus is that you'll learn some more, but there are no extra marks available for this.