CMPE 12 Lab Report # 3

John Allard
Steve Lessard
Lab Section #2

October 16th, 2014

Contents

1 Lab Overview

For lab #3, our goal was to dive head-first into the world of machine language. For our last project, we built a simple ALU and learned how they work. We were able to enter OpCodes and inputs for the ALU, but we weren't able to actually run any programs. For this week's lab, we were instructed to use a Little Computer 3 (LC3) simulator. The LC3 is a very basic computer by modern standards, with only 65k of memory and a 16 bit word size, but it makes a perfect introductory tool for assembly and machine language programming. The purpose of this weeks lab is to build off of what we learned about individual operations in lab #2 and to learn to write entire programs in just 1's and 0's. This serves the purpose of showing us how those individual ALU calls can be combined together solve more complicated problems than would first seem appearent. The specific instructions for lab # 3 are listed below.

- Write a program in machine code for the LC3 that does the following
- Starts at memory address x3000
- Reads the value from memory address x3100 (called n for now).
- Start at address x3200, and for each of the n words following address x3200
- Go through the word, add up the number of 1's, and add this number to a global counter.
- Store the final value (number of 1's found in n words) at memory address x3101.

We were expected to accomplish everything listed above in less than 45 words of memory for both data and instructions, this means loops would have to be used instead of hardcoding cases.

2 High-Level Design

Translated to a high-level language like C or Java, the instructions listed in the prior section could be accomplished in about 6 lines of code. All one needs to do is contain 2 loops, the outer loop to go through each word of data, and the inner loop to go through the individual word and count the number of 1's. Because we are using machine code, there are no explicit loops, so we have to use conditional branch statements to achieve the same effect. Conditional branch statements can be a bit tricky, because they can only tell you if a result is positive, negative, or zero, you have to be careful in how to do your counting for your loops. I ended up counting up from 0, subtracting the final index I should stop at, and comparing that with the Z flag to tell if I should stop looping, but surely there are other ways to do this. Another semi-tricky part of this lab was figuring out how to sum up the number of high-bits in a single word. Although there are surely a few different ways of doing this, the way that I came used to to check if the N flag is set (means highest bit is a 1), increment the counter if it is, then bitshift the word left. After this step, repeat until you have shifted all of the original bits out of the word (16 iterations). I can then keep incrementing this counter for every word analyzed and by the end of the double loop I will have the number of 1's stored in a register. The original pseudo-code that I wrote while planning out the program can be seen below.

```
1  | Load constants into memory (16 = word size, x3100, x3101, x3200 = important addresses)
2  | Find the value stored at address x3101, this value (n) is how many words we will look through.
3  | Go to address x3200
4  | for each of the n words (x3200 - x3200+n)
5  |    Read in the value at that address.
6  |    for each bit in the word (16 total)
7  |       Check if the top bit is a 1 by checking negative condition flag.
8  |       If the N flag is set, increment a global counter by 1.
9  |       Add the value to itself, doubling it, leftshifting by 1.
10 |    write the final counter value to address x3101
```

The above reads almost like a C program, but adds in a little bit of the low-level machine-language description as well. After I finished writing and analyzing the psuedocode above for correctness, I translated the code to it's assembly code equivalent. Finally, I would compile the assembly code into a binary file, then comment each line of the binary file with a description of what is accomplished by that line. A in-depth description of the actual machine code is found in the next section.

3 Code Analysis

3.1 Loading and Initializing

My code can be found in the `lab3.bin` file. It is entirely commented in-line, so every line should have it's assembly language equivalent and a short description of it's purpose in laymens terms. I start off like my programs start off, defining constants, declaring variables, and initializing variables. In my case, the first thing I do is load 3 important values into memory. The first value loaded is the value `x3100`, which is then used as the address to use to look up another value, the number of words that need to be analyzed. This value is stored into register 1 and kept there for the remainder of the program. Next I load the constant value 16 into register 2. This value will be used to tell if the inner loop of the program has finished. Finally I load the value `x3200` into register 6. This will be the place in memory where we begin analyzing words from, and thus R6 will be incremented until we reach `x3200+n` where n is the number of words we need to search through. After loading these values in from memory, I go through and AND the rest of the registers with 0. This ensures we don't end up using a junk data left over from previous programs.

3.2 Outer Loop

The outer loop doesn't do much, it just goes linearly through a consecutive block of n words in memory starting from address `x3200`. In detail, it works like this :

1. Initialize R4 to zero, it will be the inner iteration counter.
2. Load the value from R6, which will be an address from `x3200 - x3200+n`.
3. Evaluate the memory at that address, and load it into R7. R7 contains the actual current data word that need to processed for ones.
4. Let the inner-loop parse R7 for data.
5. When the inner-loop ends, increment R6 to get the next data address.
6. Increment R4, subtract the value in R1 (number of words to analyze), see if the result is zero or positive. If true, then you have gone through all of the required words, which means we can exit the loop. Else, go back to step one.

So the main job in the outer-loop is just to store the appropriate data-word in register 7 for the inner-loop to perform an analysis on.

3.3 Inner Loop

The entire purpose of the inner-loop is to count the number of 1's present in the data word that is currently in register 7. If it finds a one, it increments a global counter and moves on. In detail, the inner loop works as such.

1. Check if the value in R7 (current data word) has a 1 in the high bit (check N condition flag).
2. If true, break to the increment section. This simply adds 1 to a global counter stored in R5.
3. If false, don't increment the counter.
4. Add R7 to itself, doubling the value, effectively left-shifting R7 by a single bit.
5. Increment the iteration counter.
6. Subtract 16 from the iteration counter (R3), if it is zero or positive we have gone through every bit, we can exit. If it is negative, we go back for another iteration.

So the inner loop takes 16 iterations to look for all of the 1's in a given word, but this is not actually optimal. A better solution would be to check if the word is equal to zero. If this is the case then you can stop looking for more ones because you already know every bit in the word has to be zero.

After the Inner loop is finished, we return to the outer loop which decides if we are finished or not. When the program is finished, the last operation is to write the total number of 1's to the address `x3101`.

4 Conclusion

This lab was an important bridging between ALU design and working with assembly language directly. It allowed us to actually run a real program instead of just single commands. At the end, I ended up using 9 unique opcodes through my entire program, counting all combinations of BRn/z/p as a single opcode. My program ended up being 35 words long, which seemed longer than the class average. This probably means there is a lot of room left to optimize my code. I ended up using all of the 8 registers that are available to me as a user at some different point in the program. The list of what each register is used for can be found in the header section of the `kab3.bin` file. It would have been quite convenient if the LC3 would come with a subtract instruction, or even better a 'Count 1s' instruction, although the former is obviously impracticle with a limited number of opcodes.