CMPE 12 Lab Report # 4

John Allard
Steve Lessard
Lab Section #2

November 15th, 2014

1 Overview

This laboratory had us once again working with LC3 simulator to write a program in assembly language. The goal for this laboratory was to write a program in LC3 assembly that allows the user to encode and decode Morse code messages. Although Morse code is a very old technology, this was still no simple task. This is because there is no standard function that takes in an ascii character and converts it to it's Morse code equivilent, or visa-versa. This meant that we would have to hard-code the conversions into a table in our program, then use that table to look up the translations to both encode and decode our messages. There was no single correct way to accomplish this, and so the length and difficulty of this project was largly determined by the creativity of the student. In the end, our program was supposed to be able to accomplish two tasks, taking an ascii string and turning it into morse code, and taking in a morse code string and turning it into an ascii string.

2 User Input

The first task that we needed to accomplish was to gather some input from the user. To start, the user needed to choose if they wanted to encode or decode a given message. To choose encode, they would press 'E', to decode they would press 'D'. The only other piece of information that we need from the user is the string to process. If the user chose encode in the last step, they are expected to enter in a string of ASCII characters (1-9, A-Z strict uppercase). This string can be up to 30 characters long, and must be ended with a newline character (by pressing the enter key). If they user tries to enter too many letters, the character-entering loop is restarted and they must re-enter their entire string. If they enter in a string of valid length and consisting of valid characters, we move on to deal with encoding the message. If the user however chose to decode a message, they will still be expected to enter in an ASCII string of up to 30 characters, but this time the characters must be valid HEX symbols (1-9, A-F strictly uppercase). If the user enters in an invalid character, they will be informed of such and will have to re-enter their entire query again.

3 Encoding

The first task that I decided to tackle was encoding a string of ASCII characters into their morse-code equivilent. Performing this task was quite complicated. and thus completion of the task involves numerous subroutine calls. Instead of having the user enter their entire ASCII string, then go over the entire string and convert it to morse code, what I do instead is convert each individual ASCII character right after the user enters it and before I prompt the user for another character. This means as the user is entering is data to be stored in addresses `x3201 - x3220`, we are simultaniously looking up the morse code equivilent for the entered character and storing it in the addresses `x3221 - x3240`. I do this by simply looking up the morse code equivilent of the entered character in the lookup table that we were required to define.

So after the user has entered in their entire string of ASCII characters to be encoded, my program already has the Morse code encoding stored in memory, but their is still another step to carry out before the program is finished. Even though we know the morse code equivilent as a single byte of data (example 'B' = 01111000), we still do not know what the hexidecimal representation of that byte is in ASCII. In other worse, even though 'B' = 01111000 = x78, we still have to turn the value 01111000 = x78 into the ASCII string '78', consisting of the two characters, '7' and '8'. To perform this action, I used another small lookup table that converts a given hex value to it ASCII representation (B = 11 = 'B' = 66 in ASCII). For this method to work, I had to take a given morse code byte and separate it based on it's upper and lower 4 bits, because each of these sets of 4 bits represents a single hex digit. What made this task

even more difficult was the fact that the LC3 does not have any bit-shift commands, and this operations called for right-shifts, which are much harder to implement then left-shifts. To get the first hex digit for the mrose-code byte, I would go through the byte of data, and perform a bitwise AND operation with specific number to extract the data I needed. For instance, if `sum1` represents the high bits and `sum2` is a variable for the low bits :

$$\text{B} = 01111000 = \text{x78}, \text{sum1} = 0, \text{sum2} = 0$$

$$\text{sum2} = \text{B} \wedge 00001111$$

$$\text{if}(\text{B} \wedge 00010000 == 00010000)\ \text{sum1} \mathrel{+}= 1;$$

$$\text{if}(\text{B} \wedge 00100000 == 00100000)\ \text{sum1} \mathrel{+}= 2;$$

$$\text{if}(\text{B} \wedge 01000000 == 01000000)\ \text{sum1} \mathrel{+}= 4;$$

$$\text{if}(\text{B} \wedge 10000000 == 10000000)\ \text{sum1} \mathrel{+}= 8;$$

$$\text{sum1} = 00000111 = 7$$

$$\text{sum2} = 1000 = 8$$

Thus at the end, every bit gets shifted 4 bits to the right, allowing us the 4 high-bits of the morse code into the lower 4-bits of a different byte altogether. We then run this byte through our symbol table that converts a hex digit to the corresponding ascii character, and print this character to the screen. We then mask the morse-code byte with `x0F`, turning off all of the high-bits and leaving the low-bits in their original state. We then send this number through the hex digit to ASCII symbol table, and print that character to the screen.

In summary, if the user asks to encode a message, we enter a loop that both takes in his character, saves it to memory, translates that character into the morse-code equivilent, and saves that new byte to memory. We then go to a new loop that loop over every single more-code byte, and turns it into two ASCII characters representing the hex digits for both the high and low 4-bits of the morse code byte.

4 Decoding

Decoding was a task that I actually found a bit simpler than encoding. It starts nearly the same wy, after the user enters 'D' to choose decode, they are prompted to enter in a sequence of up-to 30 ASCII characters, always terminated by a newline character. The different between decoding and encoding is that for decoding they are restricted to entering in the character 1-9 and A-F, which represent valid hex values. The loop that allows that user to enter in the characters is a bit different for decoding than for encoding. For decoding, we let the user enter in two characters in sequence before we perform any data-validation or decoding. This is because we need 2 hex digits to represent a single more code character, so until we know what both digits are we aren't able to perform any meaningful computation on the data. For each iteration of the input loop, we take in two characters, store them to memory, then call a decode subroutine. This subroutine takes these two ASCII characters representing the upper and lower bits of a morse code byte, and first turns them into a single byte. To do this, we use a HEX to ASCII symbol table that converts the input characters to their corresponding hex value (A = 10, B = 11, etc). Using these two values, we can construct a single 8 bit number by adding 16 for every value in the high digit, and 1 for every value in the low digit. This leaves us will a single number that represent the 8-bit morse code sequence that the user wants to decode. For example, if the user entered in the two ASCII characters '7''8', then we would have first translated this into the two numbers 00000111 and 00001000. Then we combine these numbers, bit shifting the first number to the left by 4 bits, and adding it to the second number to get 01111000. Now

all we have to do is go to our lookup table and find what ASCII character this byte of Morse code corresponds to (if any). When a corresponding ASCII character is found, we write this character to memory. Finally, after all of the users input has been gathered, we go through our array of decoded values and print them to the screen.

5 Conclusion

This program was long, mine came out to over 700 lines of code including extensive commenting and whitespace. I'm nearly positive that there are more effective ways to write a program that accomplishes the same tasks, but as far as my knowledge of assembly goes (and the limitations of the LC3's ISA) this is a decent solution that satisfies all requirements outlines in the instruction handout.