Matthew James Greci

Banner ID: 101634187

Laboratory 2:

More Basic Assembly Language Programming

February 24, 2015

**Introduction**

The goal of this laboratory is to practice more with coding in assembly language. This includes the assembly/linker/loader system, debugging, and other assembly level activities such as code equivalent to if/else statements and subroutines.

While it may make sense to simply ignore assembly level language and program at a higher level, such as C/C++, there will be times in an computer or electrical engineer’s career when a system may not have the ability to run C code. This is when it will become vital to know assembly code and be able to convert one’s C code to assembly. Because assembly is closer to pure hardware, creating assembly code could mean generating a faster, more efficient system. In order to do this, one must know how assembly works and how to translate pseudo-code to assembly.

**Methodology**

One of the most basic statements in C or C++ is the if/else statement. The if statement checks whether the condition (Z) given is true or false. Then it will jump to the true label or false label, corresponding with the condition (see Example 1). Assembly language does not have an equivalent statement. Instead, it was found that Example 2 was equivalent due to the abilities of assembly language. Both examples can be adjusted to allow for any kind of logical tests. The highlighted portion of Example 2 shows how it is equivalent to Example 1. The structure of an if/else statement was used in almost every problem in this laboratory.

**Example 1**

if (Z is true)

//Z is true

else

//Z is false

//if statement complete

**Example 2**

li $t0, 0 #t0 = 0

and $t0, $t0, (value of Z) # t0 = 0 && (value of Z)

bne $t0, $zero, TRUE # if Z != 0, jump to TRUE

nop # else, Z == 0, jump to FALSE

TRUE:

#Z is true

b DONE

nop

FALSE:

#Z is false

DONE:

nop # if statement complete

Problem 1 was solved with Example 2 and by making the needed modifications to test the value of Z. Then, code was added to test multiple value of Z (positive, negative, zero values) and then write each final result of W into memory.

Problem 2 used a loop that had a pointer to the address in memory where the 2,000 halfwords were stored. The loop incremented the pointer to look at each halfword and go over each test to determine what counter(s) should get incremented to match how many times each test returned true.

Problem 3 involved first finding how to talk to the LEDs. It was found that a bit pattern can be written to the least significant byte of address of PORTX, where X is the port the LED board was on. In this case, port ‘E’ was used so the address for the bit pattern was 0xBF886110. It was also found that the value in PORTA (0xBF886010) could be checked to see what (if any) buttons were pushed. Bit patterns can be placed into this address depending on the values of the buttons so a loop was created to continually check the buttons and then load in the next value for the LEDs.

Problem 4 had the most research. First, the UART had to be setup, which involved setting just certain bits of different UART registers. Next, test if there is anything in the UART receive register. If there is not, check again. If there is something, determine if the value is a new line ‘Enter’. If so, jump to the printNewLine routine and print a new line, the history of characters since the last ‘Enter’, and a new line. Otherwise store the value received in memory and transmit back the character that came in. This whole process loops continually so that the program is always echoing back to the terminal.

Problem 5 starts with the same UART setup as Problem 4. Instead, the program continually sends a single character to the terminal. This is done to view the waveform generated on an oscilloscope. See figures below.

Problem 6 starts with the same UART setup as Problem 4. Next, load the address of the string to print and call a subroutine to print just one char at a time. Then loop continues to load the next character to send and calls the subroutine until the character to send is the NULL.

Problem 7 was solved by converting a hexadecimal number to a base 10 number and print each digit. The PIC stores the input value as a hexadecimal number in memory and needs to be converted back to base 10 for printing. By diving by powers of 10, each digit can be printed (transmitted through UART), one at a time.

**Source Code**

(See attached pages)

**Testing**

Problem 1 had testing build in. The code was written with the idea of not having to change the value(s) of A and B then, rebuild, reprogram, and run. Instead multiple values for A and B are created to generate multiple values of Z. The tests are confirmed by looking at what values are written into memory at the set address and confirm they matched the expected result for each test.

Problem 2 was tested by doing a search on the file of 2,000 halfwords to see how many halfwords equaled 24,321 and -18,355. In order to confirm how many words were greater than 9,000 or smaller than -9,000, code in another language or a Unix script would be written. Otherwise, the TA confirmed the results.

Problem 3 was expected to have the LEDs rotate and change around one half-second intervals. Pressing the buttons on the board would either change directions of the LEDs or continue with the same direction. Multiple people tested this expected output and the TA confirmed this was correct.

Problem 4 had a simple output expectation, which was to echo the keys typed on the terminal onto the monitor. Then when ‘Enter’ was hit, a new line was enter, the history of keys typed since the last ‘Enter’ were repeated and a new line was enter again. Testing involved typing multiple lengths of characters, ranging from no characters to 20 characters, and hitting ‘Enter’.

Problem 5 testing means knowing what the generated waveform should look like. UART sends a start bit (1), 8 bits of data, and a stop bit (0). The 8 bits of data are inverted and this shows in the generated waveform. For example, sending 0xFF (11111111) results in a waveform pattern of 1(start) 00000000(data) 0(stop).

Problem 6 has a clear expected output. Using the UART setup and knowledge of how to send a single character from Problems 4 and 5 means the only thing to test is that the loop to load and call the subroutine performs correctly and terminates. The only condition to look for is when the character loaded is NULL. The TA confirmed the output.

Problem 7 was expected to print the input value given. This means confirmed that positive and negative numbers can be printed. The values 12345 and -12345 were tested and it was confirmed that both times, the results were as expected.

**Conclusion**

The goal of this laboratory was to be able to convert pseudo-code into assembly. The process of translating if/else statements was probably the most important and shows how almost every we know in higher languages such as C and C++ need to be translated. Also, the ability to have direct memory access from assembly really does allow the user to control LED board and use any sort of protocol such as UART or maybe I2C.

This laboratory makes one appreciate the high level of such languages but also allows one to better understand what the high language is translating to and why there may be errors with perform when translating. There are multiple ways to create the same if statement in assembly and thousands of statement with different ways of working. This is probably why coding at the assembly language can really boost performance of a system when the job of perform is left to the engineer.

**What I Learned**

By far, the most important lesson learned was how important writing at the assembly level can be. While coding at this level can be time consuming, the user is given the ability to control the performance of the system. The user can also directly access memory as well as utilize any protocol they choose to the system’s IO ports/pins.

I believe knowing how C code gets translated to assembly can really increase performance because of what I now know of what is going on underneath. At the same time, I can ignore certain obvious bits of code and take less time to code time by not having to completely rethink the problem just to create the UART setup. This chunk of code was constant and only changed if a different port were to be used. This can be written into a C function and the parameters given can determine which ports to use.