Brandon Rullamas – brullama

Lab section: 4 TTH TA: Shea Ellerson

Due: 5/19/2013

Lab Partner: Susie Kim

Title:

Lab 4: Floating Point Multiplication

Purpose:

The purpose of this lab is to learn more about how assembly code in LC-3 works and all of the different opcodes that can be used to make programs, with a more specific purpose being to convert two-digit base 10 numbers into half-precision floating-point numbers and then to multiply them.

Procedure:

The procedure for this lab consisted of a lot of trial and error, with renditions of the assembly code being made after every trial to fix small errors such as loading the wrong number into specific places, etc. The first thing to do was to create the flowcharts that were mandated by the lab. The process for this was relatively quick as the creation of flowcharts is much easier than the code itself to properly execute the algorithms. The program was then started on.

The program started off by using the lab3.asm code that was from the previous lab and looking at how it can be transformed into the program that we needed it to be. It was deemed that the starting code was semi-sufficient and held a lot of algorithms that would be used in this program (such as multiplication and clearing registers), but it would still require a lot of work to turn into the program we needed it to be.

To start, all of the variables that were thought to be needed were made (variables were later added on) and the greetings prompting the user for input were created. A lot of the variables that were needed already existed in lab3.asm and were simply renamed to fit this new program. Then, the conversion subroutine was made. This was done by first creating the two numbers that were needed to make the program work. Because the LC-3 cannot read in two-digit numbers, the numbers were broken up and read individually in order to get the correct values. This created four stored values: the two digits of the first number, and the two digits of the second number. The 'CONV' (conversion) method was then called on both numbers to properly convert them to a decimal value. This was very similar to a multiplication algorithm in that the first digit of the number was added 9 times to create the ten's place, and then the second value was added to the value that has been added 9 different times. This was then returned to the main subroutine.

The now two decimal values were converted to floating point numbers. This was primarily done through masks in order to correctly get the values of the appropriate locations. The exponent was done first, as the layout of the floating point numbers is as follows: [SIGN][EXPONENT][MANTISSA]. Assuming that the sign is positive, the next one to do is the exponent. This was done by creating a count of 25 and decrementing it each time the values of the number were moved to the left. This got the values of the exponent in the appropriate location. After this, the fraction/mantissa's were made. This was done by inverting the mask that was used and 'AND'ing it with the number that was originally created. The exponents and fractions were then stored in appropriate locations.

The next appropriate thing to do was to combine the exponent and fraction to create a floating point number. This was done using a counter that would tick 10 times as well has having the exponent in R0 and the fraction in R1. The exponent was added to itself 10 times to get it into the correct location and then then two values were added together. The overall algorithm was pretty trivial, but the number needed to get the exponent into the correct location took some verification to ensure that the rest of the number would be correct.

The floating point numbers that existed would then be printed out for the user to see if they were correct. This was done by creating three different counters (to correctly print the spaces) and going through the numbers of the floating point number to see if 0 or 1 had to be printed. The main point of interest here was to be sure to convert the number back to ASCII after deciding on the value so that the number could be printed. The counters were decremented each time (holding values of 1, 6, and 16) and printed at the appropriate times. The next thing to do was to multiply the two values together.

The exponents were relatively simple to create. The exponents were added together through simple addition, and then as noted in the lab, the double bias was removed. In this case, the bias was 15, so 15 was subtracted from the number and stored into 'EXPOFIN'. In order to create the fraction/mantissa, the first fraction/mantissa was loaded in. A leading 1 was added to this, and then shifted four times in order to account for the 4 zeros at the end of fractions/mantissas as clarified in lab. This was then done for the second fraction/mantissa with the same process. These two fractions/mantissas were then added together through the following algorithm: first, the first fraction was added to an empty R3. 1 was then subtracted from the second fraction. After this, the first fraction was added to itself based on how big R1 was. The leading one was then checked to see if it was contained within the mask. If it was, it would skip the following algorithm: 1 would be added to the exponent and the two shifts would occur. If not, only one shift would occur and the leading 1 would be removed from the result.

The final process to occur in this program is a combination of all of the above to form the final product. The exponent and fraction numbers were combined using the 'COMB' subroutine, and the three counters to check where the spaces were were made again so that the spaces could be printed properly in the final subroutine, ‘PRIFLO’. The specific algorithms in these subroutines can be found above.

Algorithms and other data:

There were two main algorithms that were needed for this lab: multiplication and division. Multiplication required a visit back to the very learning of multiplication and how it works. A number X is added to itself a certain number of times Y. In order to accomplish this, one register was loaded with the value of 0, another with the value of x, and finally another with the value of y. A loop was then made using branch that looped through, adding X to the register holding 0, and then subtracting 1 from the Y. When Y reached 0, the loop stopped. This would lead to the following results if you were to use 4 and 2 (The register holding 0 will be referred to as R0):

```
R0: 0   X: 4     Y: 2

R0: 4   X: 4     Y: 1

R0: 8   X: 4     Y: 0
```

Despite most of the multiplication in this being used with much more complicated numbers, the basic idea of the multiplication algorithm still works.

The second algorithm needed was something similar to a division subroutine which is very similar to both subtraction and multiplication subroutines, forming nearly a combination of the two. The division subroutine followed the same process in that it used three different values, using one as a counter and one as a base, while the other held the main value. It was then subtracted from the main value using the same logic as subtraction (trivial). This process was primarily used to shift over the numbers to get them into the correct locations. For example, the layout of a floating point number is as follows: [SIGN][EXPONENT][FRACTION]. If the exponent was to be 01010, it would originally be added to 16 0’s and create 0000000000101010. To get the exponent into the correct place, the number was divided by 2 until 0010101000000000 was created.

The third and relatively simple concept that needed for this lab was the concept of masking. For example, if you ‘AND’ 00000 and 11111 together, you will get 00000. This process can be used to obtain the value of a number at a specific location, as such: 00100(11111)->00100. This concept of masking was used several times, and although it was relatively simple, it is still an important concept needed for this lab. The main processes for other algorithms were described very thoroughly in the procedure. As a final note, here is a sample output:

```
Enter 1st number: 12

Enter 2nd number: 04

Floating Point 1: 0 10010 1000000000
```

Floating Point 2: 0 10001 0000000000

Product: 0 10100 1000000000

The program then repeats indefinitely by calling on the ‘START’ subroutine (start of the program) at the end of the instructions.

What went wrong or what were the challenges?

The biggest, and probably slowest, challenge of this lab was stepping through the LC-3 multiple times to see what was wrong with the code. This was not often at first, as it was usually just problems with wrong opcodes being used or a typo somewhere. However, as the program progressed into combining the numbers and trying to get them into the correct locations, more tinkering was needed with the assembly code in order to fine tune the code. For example, in order to combine the two fractions, the second decimal number was subtracted one by one in order to get the correct value. So, if 86 was put into the program as the second number, you would have to click 258 times to get through that one process (86x3, because there were 3 lines of code in that loop). Also, due to the length of the program, it took a very long time to step through all of the code to get to the specific point where a problem was identified.

Other information:

What is the largest number that you can represent in half-precision floating-point format?

The largest number that you can represent in half-precision floating-point format would be 1.111...11 x 2^15.

What is the smallest positive number that can be represented in half-precision floating-point format?

The smallest positive number that can be represented in half-precision floating-point format would be 0.000…01 x 2^0. This is a denormalized number because the exponent would be all zeros, making the first number in the fraction 0 instead of 1 (0.000 instead of 1.000). This number multiplied out is basically zero due to how small the number is, but still holds a positive value that is very tiny.

How does a JSR and RET instruction work in LC3?

A JSR instruction jumps to a location (for example, an unconditional branch), and then saves the address of the next instruction to R7. It allows the processor to go to different locations in the code at different times in order to execute different parts of code. A RET instruction brings the processor back to the calling routine by doing an execution similar to JMP R7. Both instructions utilize different values to operate. JSR uses arguments which are values passed into the subroutine in order to function, and RET uses return values in order to bring back to the main subroutine that shows that the subroutine did its job.

Enumerate the subroutines you implemented. For each subroutine, include its function, the register(s) that holds the argument values, and the register(s) that holds the return values.

The following subroutines were implemented: CLEAR, CONV, FLOAT, COMB, PRIFLO, PRINT1, PRINT0, PRINTSPA1, PRINTSPA2, and SHIFT.

Name: CLEAR [Clear]

Function: This subroutine clears all of the registers for use.

Argument Registers: R0, R1, R2, R3, R4, R5, R6.

Return Registers: R0, R1, R2, R3, R4, R5, R6.

Name: CONV [Convert]

Function: Converts two separate integers into one, two-digit number by adding one number to itself several different times to make a tens place, and then adding in the second number at the end.

Argument Registers: R1, R2, R6.

Return Registers: R4.

Name: FLOAT [Floating Point Numbers]

Function: Uses a counter that will decrement itself and add numbers in order to move that number to the left in order to appropriately create a floating point number.

Argument Registers: R1, R2, R4.

Return Registers: R1, R2, R4.

Name: COMB [Combine]

Function: Sets up the exponent for combination by adding the exponent to itself several times until R2, the counter, reaches zero, so that the exponent is in the appropriate location.

Argument Registers: R0, R2.

Return Registers: R0, R2.

Name: PRIFLO [Print Floating Point Numbers]

Function: Prints floating point numbers by looping through itself to get every value of each location of the floating point number and printing a 0, 1, or space depending on what is needed.

Argument Registers: R1, R2, R3, R4,

Return Registers: None – prints out values.

Name: PRINT1 [Print One]

Function: Prints out a 1 if the value of the location in the floating point number calls for a 1.

Argument Registers: R0.

Return Registers: None – prints out values.

Name: PRINT0 [Print Zero]

Function: Prints out a 0 if the value of the location in the floating point number calls for a 0.

Argument Registers: R0.

Return Registers: None – prints out values.

Name: PRINTSPA1 [Print First Space]

Function: Prints out a space if the value of the location in the floating point number calls for a space. It is primarily used for the first space needed in the floating point number, between the sign and the exponent.

Argument Registers: R0.

Return Registers: None – prints out values.

Name: PRINTSPA2 [Print Second Space]

Function: Prints out a space if the value of the location in the floating point number calls for a space. It is primarily used for the second space needed in the floating point number, between the exponent and the fraction.

Argument Registers: R0.

Return Registers: None – prints out values.

Name: SHIFT [Shift Numbers]

Function: Shifts the numbers in the argument to the left based on the values of the arguments passed to it.

Argument Registers: R0, R1.

Return Registers: R0, R1.

Conclusion:

This lab was very difficult in comparison to all of the previous labs. The jump in difficulty was very surprising. Although it served its purpose in teaching us how to multiply half-precision floating-point numbers, making the code to do so proved to be a very difficult task and took a lot of work in comparison to the previous labs. There were several problems that came up through the process of the lab that took a long time to figure out, and having a partner that was unclear of what was going on did not assist in making the difficulty of the lab any easier.

Extra:

The algorithms and tips given by the TAs were helpful, although slightly confusing at times. Each TA and tutor had their own perspective on how to approach the problem and these approaches often overlapped to produce confusing results. However, the TAs were still very useful. If anything, this lab made me appreciate high level languages a lot more due to how more simple this lab would have been through the use of a language like Python or Java.