Brandon Rullamas – brullama

Lab section: 4 TTH TA: Shea Ellerson

Due: 5/5/2013

Lab Partner: None

Title:

Lab 3: Introduction to LC3

Purpose:

The purpose of this lab is to learn how assembly code in LC-3 works and all of the different opcodes that can be used to make programs, such as the ‘calculator’ in this program.

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 program started off by using the echo.asm code that was given to us and looking at how it can be transformed into the calculator program that we needed it to be. It was deemed that the starting code was very basic and would 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. This was taken straight from the echo.asm code and was very trivial. Then, the subtraction subroutine was made. This was done by loading in the two inputs that the user had input, making the second input negative by NOT’ing it and adding one (2’s complement) and then adding the two inputs together. This created the subtracted result, which was then returned to the main subroutine and printed out.

The multiplication subroutine was then created. This was much simple at first, but became much more complicated later on once division was finished and the need to print out two-digit numbers was at hand. To start off, the two user inputs were loading in and a counter was created starting at the second input’s value. The first value was then added to a variable X which started at zero for the number of times equal to the second input’s value. So, if you input 4 and 2, you would add 4 to 0 two times because the second input is 0 and the first input is 4, creating 8. The process to print out two-digit numbers will be discussed later.

The division subroutine was created after the basic multiplication subroutine was made. This was similar to the subtraction and multiplication subroutine, except that it required a lot more checking and a bit different order of operations. The two inputs were loaded in and then the subroutine utilized the same operations as the subtraction subroutine in terms of inverting and adding one to a specific number. This was done for a specific number of times equal to the second input's value, similar to the way the multiplication subroutine worked. However, in order to break out of this loop instead of simply stopping at zero, statements to see if the quotient was negative, zero, or positive were made. If it was negative, the counter went back one and the quotient regained its original value by adding the value of the divided quotient and the value of the second input. It then went into the 'zero' loop (which could also be reached if the quotient was 0 to begin with), which added 48 to the quotient and remainder for proper printing. These values were then returned to the main subroutine for printing.

Finally, the ability to print out two-digit numbers was implemented. This was done in a nearly identical way to the division subroutine, following the same negative-zero-positive checking routine. However, the main difference in this part of the program was that -10 was subtracted from the number each time instead of a specific value input by the user. This allowed 8 to turn into 0.8 and 12 to turn into 1.2. These results were used later in the subroutine because a loop to check if the quotient was zero. If it was zero, it skipped straight to the remainder for printing, making it seem like 8 is 8 (as it removed the 0.), and if it was not zero, it printed it, making 12 seem like 12 (by printing out 1 and then 2).

Algorithms and other data:

There 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

For results that went over 9, a function very similar to division was used, which is now described. The division subroutine is very similar to both the 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 the subtraction subroutine (trivial). Three different loop conditions were used to check the condition of the loop. It first passed through a negative condition, which would go back one in the counter and add back the value if the quotient turned up to be negative. The second condition was to see if it was equal to zero, and if so, it would simply add in the values into the appropriate registers and print it out as necessary. The final condition was to see if the value was still positive, in which it would continue with the loop. In addition, below is a sample output of the program with everything working:

```
Please enter a 1-digit number: 7

Please enter a second 1-digit number: 3

Subtracted: 4

Multiplicated: 21

Divided: 2

Remainder: 1
```

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 multiplication, division, and printing out two-digit numbers, more tinkering was needed with the assembly code in order to fine tune the code. For example, during the multiplication subroutine, the values were originally being added to the original value, causing 4x2 to return 12 instead of 8, because it would add 4 twice to 4. Obviously, this was fixed by adding the two 4's to zero, instead of the existing value.

Other information:

How does a branch instruction in LC3 work?

A branch instruction works by specifying one or more condition codes. It uses the flow of control and the last used bits to check the condition. The three different conditions are negative, zero, and positive. If a branch instruction exists that is BRnzp, then the branch always executes due to the value always being true. However, if the branch is not taken, then the next sequential instruction is executed.

Which store instruction (ST, STI, STR) did you use to store your result to the correct memory location? Why did you use that particular instruction and not the other two?

ST was used to store my result to the correct memory location, because it stores a value/score label into a destination register. This allowed me to use a label to store my value, allowing for me to call it later. I did not use STR because it uses a source register to put a value into a destination register (an immediate offset could also be used, which wasn't necessary). I did not use STI because it treats its immediate value source/label as a memory address and stores that value into a destination register, which was not needed. ST was simply the storing instruction that fit my program the best.

What is an addressing mode? What are the five LC3 addressing modes and give an example of each one?

An addressing mode is an instruction which defines how the instructions in the processor identify the operand/operands of each instruction. The five addressing modes are Registers, Immediate, Base+Offset, PC-Relative, and Indirect. The following are examples:

Register

R1 <- R1 + R2

Immediate

R1 <- R1 + -2

Base+Offset

R1 <- Memory[R2+4]

PC-Relative

R1 <- Memory[PC+6]

Indirect

R1 <- Memory[Memory[R2+4]]

Which register is used as a place to put return value for TRAP instructions?

Register 7 is used as a place to put return value for TRAP instructions.

By stepping through your program, what does PUTS trap do? Please explain the entire process of how PUTS is executed.

PUTS trap subroutines print out the stringz statements that are coded into the program. It works by getting the value specified in the trap vector, which is located in the memory at the absolute address given by the immediate field. The values that are needed are given through instructions such as TRAP x04 and TRAP xFD. The subroutine uses several branch instructions as well in order to print out all of the characters that are needed by the program.

Conclusion:

The lab had a nice mix of difficulty to it. It started out as a very large problem, but once it was broken down into smaller parts and looked into more deeply, the problems became very manageable and fun to solve. No problem caused too much disruption in the progress of the program and the lab as a whole served very well in helping me understand how the LC3 operates.

Extra:

The logic that the TAs helped provide in terms of the division and multiplication subroutines were useful for getting started on the program, and they helped clear up a lot of small differences, such as when to use ST, STI, and STR. The weirdest part about the lab was going from programming languages that I’ve used in other classes that make division and multiplication so simple yet complicated since you’re programming it into the computer, to the LC3 where you actually had to go through all of the logic for division and multiplication yourself. It was a nice experience, though.