**Introduction**

The purpose of design project 3 is to introduce PIC assembly language programming. This is facilitated through the use of MPLAB, which simulates a PIC16F84 micro controller. We were required to design a program which would compute the max value and the mean value of an array of 8 integers that are in 2’s complement form. The design of our program had several restrictions, outlined below:

* The program must start at 0x00 memory location. This should be accomplished using the “org” directive.
* Indirect addressing must be used to read values from the data array
* The program must use a loop to cycle through the array
* After the program completes, it must enter an infinite loop

Additionally, we were required to have at least 2 subroutines. The numbers in the data array are stored in register locations with the first beginning at 0x30 and the last at 0x37. These cannot be changed. The final average is stored in location 0x20 and the max number is stored in location 0x21. Locations 0x22 through 0x2F are allowed to be used as temporary storage for numbers. No other locations are to be used. In computing the average, overflow did not have to be accounted for. In computing the mean, any partial values did not have to be accounted for.

**Design**

The first step in writing the program was to decide how to create the loop. I decided to use a counter set initially to 8 since there are 8 values in the array. The number is placed into position 0x2D which is the designated loop counter position. Using decfsz, I could decrease the loop counter each time until it hit zero, which would then break the loop.

The location of the first number is first placed into the FSR. The first value itself is automatically placed into the max value position since it will be the first maximum. The program then enters the loop. Using INDF, indirect addressing is used to move the contents of the position stored in the FSR. Thus, the contents of position 0x30 are placed into the working register instead of the number 0x30. This number is then placed into the first temporary position since the original values cannot be changed.

Once the value is placed into the temporary position, the first operations can then occur. I decided to calculate the total of the numbers first and then calculate the average value outside of the loop since the average does not need to be calculated more than once. However, there are 2 cases that can occur when trying to find the total number, one for if the number is negative and one for if the number is positive. I broke down these two cases into two additional subroutines, neg\_op and pos\_op. In order to distinguish between these two cases I used the btfsc and btfss commands on bit 7 of the number to be added. Btfsc skips the next command if the bit is clear, thus indicating a negative. Btfss does the opposite, skipping if the bit is clear, thus indicating a negative. Either neg\_op or pos\_op is called after evaluating the number. Since the number is in 2’s complement form, neg\_op undoes the 2’s complement and then subtracts from the total. The resulting number is placed back into total. Pos\_op simply adds the number to the total since undoing 2’s complement is not required. Once the new total is determined, the compute\_max subroutine is called.

The compute\_max subroutine is used to determine which of 2 numbers is larger. I also based this algorithm on condition testing. The number is first preserved using the second register since the old register will have operations run on it. There are three cases that could occur. The first is if one number is positive and the other negative, the second is if both numbers are positive, and the third is if both numbers are negative. In order to determine the case, btfss and btfsc are used again to test for negativity. For the first case, the positive number is automatically moved into the max value position since a positive number will always be greater than a negative number. For the second case, the numbers are subtracted and the bit 7 is tested. This is done because if the resulting number is negative, this must mean that the first number was smaller than the second and vice versa. This subroutine is called 8 times, one for each number.

Once the max value was determined and placed into its position, the loop counter was decremented by one and the loop started again. When the loop counter hit zero, then the goto loop command would be skipped, essentially breaking out of the loop. From here, compute\_avg is called. The compute\_avg subroutine computes the average using the total value that was recorded during the loop. To compute the average, I decided to use rrf, which would circularly rotate the register. Since an arithmetic right shift is division by 2, I needed to modify rrf so that it would become an arithmetic shift. This was done through the manipulation of the carry bit. I first checked to see if bit 7 of the number in question was set. I used btfss and btfsc in conjunction with bsf and bcf to change the carry bit. Bsf and bcf set or cleared a certain bit. In my case, this would be the carry bit. Depending if bit 7 of the number was set or clear, then the carry bit would be set or cleared every time, thus creating the arithmetic right shift. Since I needed to divide by 8 and 2^3 = 8, I looped the arithmetic shift 3 times and then placed it into the average value position. The last two lines of code move the literal value 0x00 into the total position. This resets the total value and all the registers do not need to be reset to do another average.

**Obstacles**

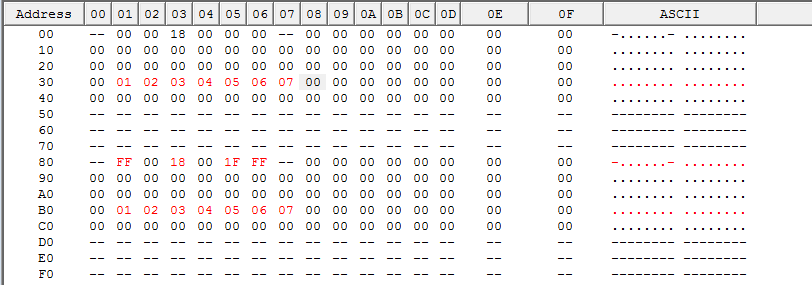
While there were no actual unrecoverable problems encountered during the design process, there were many obstacles. The most difficult part of this project was learning MPLAB. Assembly language programming is very different from higher level languages such as C++, thus requiring much more adaptation. The btfss and btfsc commands were also very confusing. I would frequently get them confused and use the opposite of what I wanted. Loops were also a little hard to figure out, since they are “manual” loops. In higher level programming, loops are uses by directly specifying an end condition and are generally all within the loop command. In assembly, there is a lot more required manipulation of variables and registers and everything is more specific, such as the use of the goto command in loops.

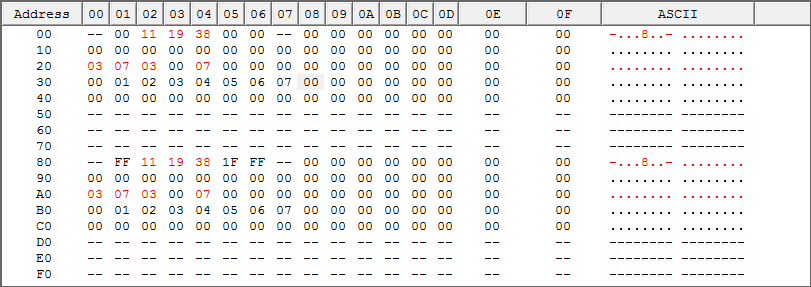
**Conclusion**

The results of this design project speak for themselves. I would say that the learning curve for assembly language programming is in the midrange level. Anyone transitioning from higher level programming languages will have a somewhat difficult time simply because assembly language programming is similar in some places but completely different in others. However, I now have a much clearer understanding of assembly language and its programming. My program is also able to successfully calculate the average and max of various arrays of 8 integers, so the design is deemed successful.

**Test 1**

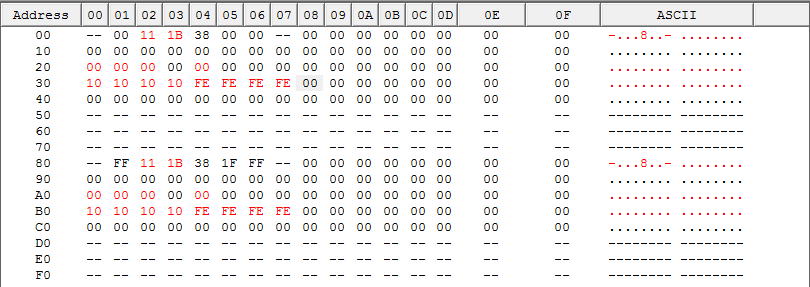
**Memory Before Computation**

****

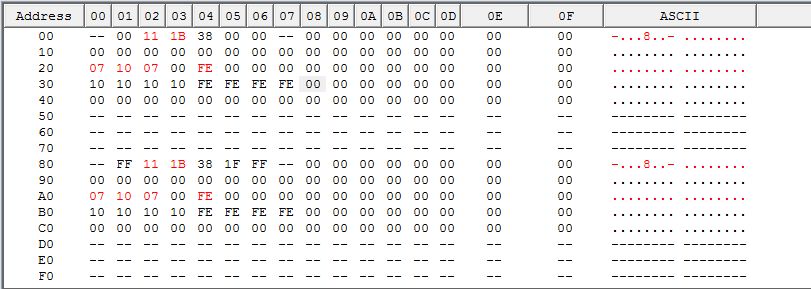
**Memory After Computation**

**Test 2**

**Memory Before Computation**

****

**Memory After Computation**

****