**CMPE-250 Laboratory Exercise 4**

**Iterations and Subroutines**

By submitting this report, I attest that its contents are wholly my individual writing about this exercise and that they reflect the submitted code. I further acknowledge that permitted collaboration for this exercise consists only of discussions of concepts with course staff and fellow students; however, other than code provided by the instructor for this exercise, all code was developed by me.

Joel Yuhas

Performed 2/17/2016

Submitted 2/25/2016

Lab Section 02

Instructor: Alessandro Sarra

TA: Peter Muller

Connor Goss

Stephen Moore

Lecture Section 250.01

Professor: Alessandro Sarra

**Abstract.**

Subroutines were observed, constructed, and tested using methods that would not cause any side effects to the main program. Specifically, an unsigned, integer divider subroutine was created that followed a set of guidelines and handled special cases, such divide by 0. The integer divider was implemented by following a special algorithm that consisted of continually subtracting the dividend by the divider until the dividend reached 0 or had a remainder less than the divider. Loops, C flag clears/set, conditional branching, as well as pushing and popping registers were also used during the creation of the subroutine. This was done in order to familiarize the user with subroutines, iterators, conditional branching, and C flag setting/clearing. The final product successfully implemented the subroutine correctly and produced results that were expected.

**Procedure.**

First, the unsigned integer divider, which was promptly named DIVU, was designed on paper. This was used to layout the code implementation of the algorithm, which would be used to divide a number, the “dividend”, by another number, the “divider”. The actual algorithm consisted of taking the dividend and then continually subtracting the divider until the dividend was either less than the divider or 0. The algorithm also called for a “Quotient” or tally, which kept track of how many times the divider was subtracted from the dividend. In theory, the quotient would be the final result of the divide operation, alongside any reminder that there may have been.

After the structure of the code had been laid out on paper, it was then implemented. A new file was created and the source code that was already provided was copied and pasted into the locations specified. Once the base file was created, a number of other guild lines needed to be followed before the bulk of the code could be programmed. This included creating a MAX\_DATA EQUate that was set to 25, allocating the variables P (Dividend) to R0, Q (Divider) to R1 and Results to contain a value of 2 x MAX\_DATA. Results would also have to be multiplied again by 4, because the size of the data being implemented was 4 bytes. The code also could not use registers R6 and R7 as these would later be used by the provided code to check if the program was executed correctly.

Next, the DIVU subroutine was created. This involved loading the variable P and Q appropriately and then pushing the R2 register, which would serve as the quotient. R2 was “pushed” in order to save the current value and free up the register for use of the subroutine, so it would be able to use R2 without changing its original value. The subroutine would then check if R0(Q) was zero, as this would signify that there was a divide by zero. If there was, then the code would use a conditional branch and skip over the rest of the DIVU subroutine and manually set the C flag to 1. Setting the C flag to 1 was important because it was the trigger for another routine that was called after DIVU, which would handle the divide by zero functionality. If C was set, then the subroutine FLAG\_SET would go in and manually set the values of P and Q to 0xFFFFFFFF.

If the equation was not a divide by 0, then the subroutine would check if the dividend was 0. If it was, then the program would skip the rest of DIVU and manually clear the C flag. This was done because if the dividend was 0, then the C flag would be set to 1 by the one of the CMP functions. To ensure that FLAG\_SET was not triggered, the C flag had to be manually be set to 0.

Finally, if neither the Dividend nor the Divider were zero, the subroutine would continue as expected. This would execute a while loop where R1 (P/the dividend) would be subtracted by R0 (Q/divider) until R1 was either 0 or less than R0. Meanwhile, R2 would be used to store how many times the while loop was executed. Once the operation was complete, R2 would be moved to R0, and R1 would contain the remainder. P and Q would then be reloaded, this time however, P would be loaded into R1, and Q would be loaded into R0, which was done for the purpose of compatibility with the provided code. R2 would then be popped off of the stack, reverting it back to its original state, and the subroutine would end.

After the DIVU subroutine was completed, the main program was written, which mainly consisted of calling other subroutines in the provided code that would test DIVU. This was done by calling the subroutine InitData, which initialized some data values in the memory that would be used to test DIVU. Next the data was loaded using the provided LoadData subroutine. The variables P and Q were then instantiated and DIVU was called. Directly following DIVU,   
FLAG\_SET was called, which would handle a divide by zero case given that the carry flag was set. P and Q were then reloaded and stored into memory and then program proceeded onto the TestData subroutine. After checking the results that DIVU had now stored, the program repeated itself and jumped back to the beginning, where it would load in a new set of data and then repeat the cycle until eventually coming to a stop when TestData eventually called for it.

The program was translated and assembled and then stepped through with a debugger. Two break points were set, one before DIVU was called and one after TestData was called. This was so the program could be ran and stopped just in time in order to observe the two numbers that were loaded into R1 and R0, and then observed the results that were then stored in memory before the program looped again.

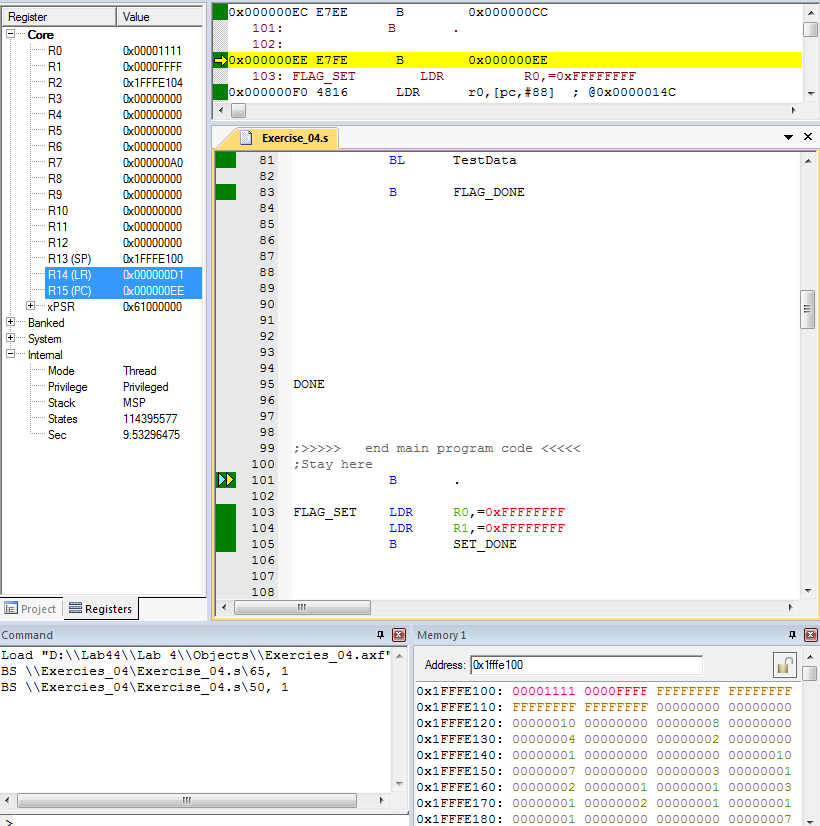
Once one cycle of the program had ran, the values of R6 and R7 were noted. R6 stored the number of errors that the TestData subroutine ran into while using DIVU, ideally the value of R6 should would be 0. R7 stored the number of cycles that had been ran, and incremented in values of 4. The program was deemed a success if, when the program stopped running, R6 read 0 and R7 read A0.

In order to further check the results, the individual register values of the first couple of runs were noted in order to ensure that the program was actually completing the desired operations.

If the results were not correct, and did not match a predetermined set of results, the program was looked over until errors could be assessed. Once the results were correct, a screen capture was taken and the testing was complete.

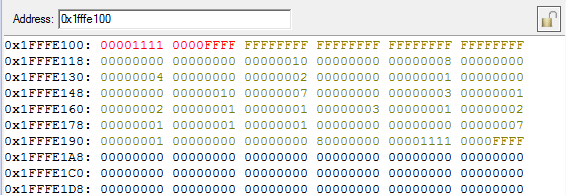
**Results**

Figure 1 shows the final simulation results of the debugger. Notice how R6 is 0 and R7 is A0, signifying there were 0 mistakes that the TestData subroutine picked up on and how the program ran A0/4 cycles. Also notice how the bottom right shows the data values stored in memory, and how they are the direct result of the DIVU operations. This is explained and shown in Figure 2.



**Figure 1.**

Figure 2 shows the results that were determined by performing the DIVU subroutine using the data that was provided in the LoadData subroutine. The results of these operations were stored in the memory as displayed, and were crossed referenced and the end by an instructor to ensure that the results were correct.



**Figure 2.**

**Conclusion**

Subroutines were observed, constructed, and tested in a basic way by creating an unsigned dividing subroutine. This was done with special cases in mind such divide by zero and dividing a zero. The implemented subroutine DIVU used a special algorithm were the dividend was subtracted by the divider until the divided reached either 0 or had a remainder less than the divider. A number of different of techniques were used to accomplish this, such as loops, conditional branches and C flag setting/clearing. These methods worked in tandem with the provided code in order to create a program that successfully implemented a DIVU subroutine and tested it. Ultimately the program was a success and by testing DIVU with the provided code, the function could be tested appropriately and deemed successful by providing expected results.