**CMPE-250 Laboratory Exercise Four**

**Iteration 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.

John Judge

Performed 2/18/16

Submitted 2/25/16

Lab Section 02

Instructor: Dr.Shaaban

TAs: Peter Muller

Stephen Moore

Connor Goss

Lecture Section 01

Professor: Allesandro Sarra

**Abstract**

This exercise explored the use of iteration and subroutines by dividing one input variable by another. By using a template and given parameters the program developed could test a series of inputs and compare them to the correct results. A subroutine, called DIVU, was written using looping and branching techniques. After the program was run and the results and memory map were compared to the expected results the implementation was verified to be correct.

**Procedure**

In order to begin this exercise the given code was copy and pasted into the main program. This code contained the framework for the program as well as the test cases and correct results. It resulted in the program only needing a subroutine named “DIVU” and some minor statements and subroutine calls.

Once the given code was added to the project a few specifications needed to be met. The first of which were to create an equate statement called MAX\_DATA and set it equal to 25. Next a word array needed to be created. In order to work with the given code the array needed to be called RESULTS and have a size equal to. Then, variables were needed to hold values for the input variables P and Q. Both were specified to be 32-bits. From here the main program and the subroutine DIVU could be written.

To start the main code the subroutines INITDATA and LOADDATA. These routines were located in the given code and as their named imply, they initialized and loaded the test inputs and the correct results. To follow this, the variables P and Q were initialized, from here the main program could be written.

To start the subroutine DIVU was called. DIVU, which at this point was also written, divided two numbers. The functionality of this subroutine was specified by Equation (1.1).

As seen by Equation (1.1), DIVU essentially divided variable P by Q then stored the result in register R0 and the remainder in R1. Pushing and popping off a stack handled this, so that the contents of the registers when the subroutine was called would not be disturbed.

From there the functionality of the division was handled by a while loop. To start the dividend and the divisor were compared. If the dividend is less than the divisor then the while loop ended. If it was greater than the divisor was subtracted from the divided and the result was stored in the place of the dividend. A counter was implemented to see the number of times this subtraction took place. The value in this counter was then the result of the division once the dividend was smaller than the divisor.

In the situations in which the dividend was smaller than the divisor the carry flag had to be set and the value of the counter was then set as the result of the division. By setting the carry flag the results would indicate that the result was truncated to an integer value. To end the DIVU subroutine, regardless of the carry being set, the contents of the stack was popped and put back in the correct registers.

Then in the main program code the results needed to be stored. If the division was valid, (the carry wasn’t set), then the results were then stored in the way specified in Equation (1.1). If the carry was set, the result wasn’t valid, then the result and the remainder were both set to be FFFFFFFF and these values were stored. This code was written to repeat until all the test inputs had run.

The program was then built and translated. From here, the program was run and following some minor debugging, the results were verified to be correct.

**Results**

In this exercise, the given code was written in such a way that if an error took place in the division register R6 would be incremented. When the program was tested register R6 remained with no value in it, indicating the division was correctly implemented.

In order to make sure the rest of the program ran correctly the results and the memory map had to be checked.



Figure (1.1): Simulation Results

As seen in Figure (1.1), the simulation results matched the expected outcomes. The values listed in the memory matched the values in the given code. Register R6 did not increment in value, proving that DIVU correctly computed the correct values for the division.

Next the memory map was consulted to determine the exact memory ranges the program occupied.

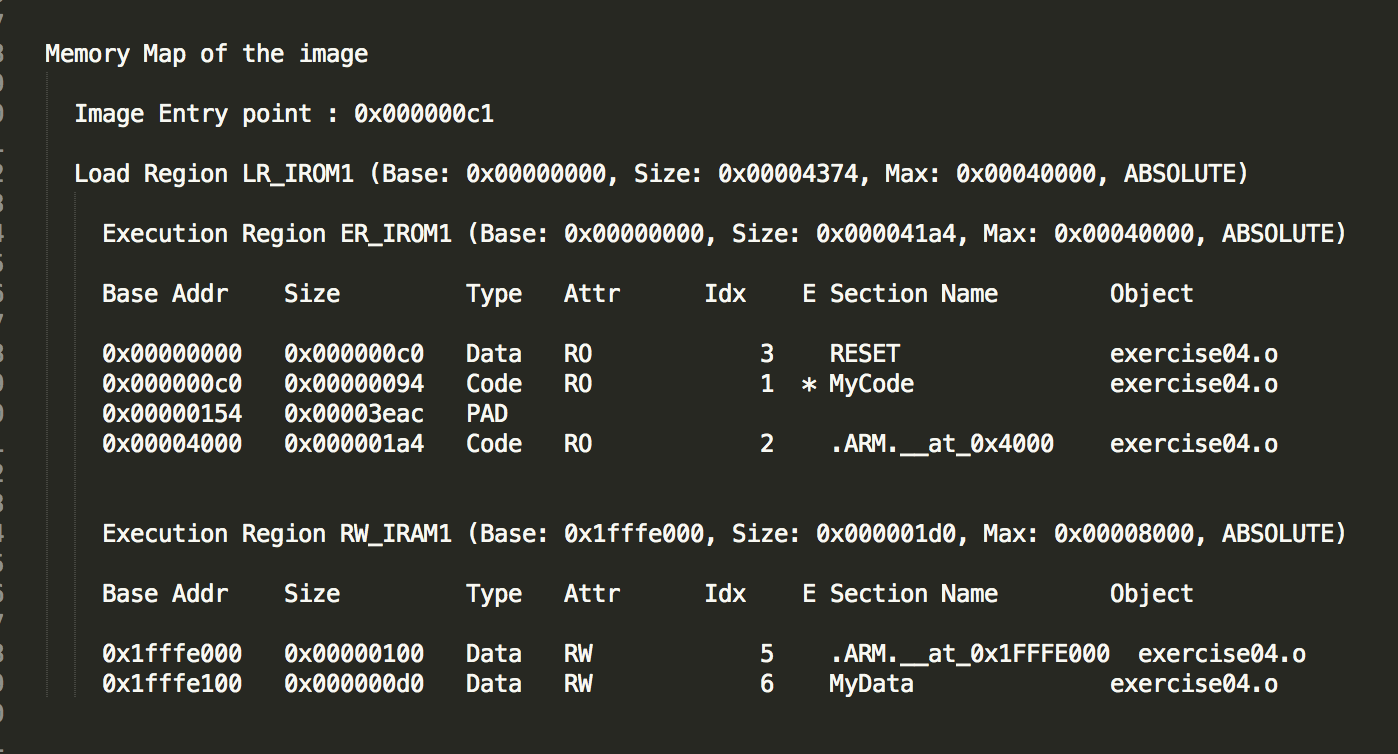


Figure (1.2): The Memory Map of the program

Figure (1.2) shows the memory map of the program generated in the listings file. As seen above, the executable code of the program occupied 148 bytes of data. As for the variables in the program, they occupied 208 bytes of RAM. The stack used 256 bytes of data. These values were found under the “size” column in both the RAM and ROM sub headings.

**Conclusion**

This exercise required the use of iteration and subroutines to solve division operations. The objective of the exercise was to develop a program that could test a series of given inputs and compare them to the correct results using subroutines and iteration. The program was developed using giving and original code. After testing the program and comparing the results to the expected outputs it was concluded the program executed correctly. The results of the program as well as the correct implementation techniques prove a successful exercise. In the future, this exercise can be looked back on as a correct example on basic subroutine and iteration techniques.