ECSE 324 Lab 2 Report

Giuseppe Lipari (260742811)
Alex Masciotra (260746829)

February 23, 2018

Introduction

In this lab, we continued to use the ARM processor and further advanced our knowledge in ARM assembly language while learning how to call subroutines. In addition, we advanced our knowledge in C coding language by programming four routines. The first routine was a stack. However, there was a limitation. That was that the push and pop ARM assembly code had to be implemented without using the generic push and pop functions and the use of the Stack Pointer in the ARM assembly language. The second routine was an assembly code that computes the max using a subroutine. The third was an assembly code which did the Fibonacci calculation using a subroutine. Finally, the last routine was to create the same max code (from the assembly code), but in C and calling an assembly written subroutine into the C main.

1.1 Stack

The primary purpose of the stack is to store temporary data. In the case of Assembly language, it is helpful when there aren't enough registers for a program to store. The stack can be used for additional storage. The way a stack works is last in, first out. What this means is the last thing that is pushed onto a stack will be the first one to be popped. Creating the stack was tricky considering we were not allowed to use the generic push and pop calls in the ARM assembly code. We first initialize a register with a value (the first number in the set of numbers) and then branched to the Push label. We re-initialized the same register with the next value in the array by using an offset and branched to Push again and once again repeated this process for the third number with another offset. In order to push, a stack was implemented using a register which did the same as the stack pointer (SP, which is register 13). Since we were not allowed to use the SP, we used another register to serve the same purpose.This essentially holds the address of the top value in the stack. Afterwards, in order to store the numbers in the register, we used the *STR* (store) label in order to store the current into the memory address of "SP". We did this process three times in order to store the three numbers. In order to Pop the stack, we used the *LDR* command which loaded the register R0. We then incremented the "SP" and repeated this process again for the second and third number, changing the address of the "SP" register after each pop.

This task was one of the easiest out of the four. To make it simpler, we could have used the push and pop instructions preset in Assembly language which would have made the code much shorter and simpler. However, this was not allowed in order for us to fully understand what is happening in a stack. Another way we could have made this code better would have been to simplify the instructions we used for push and pop, instead of being repetitive. We could have branched the Push instruction so that we did not need to write it out three times for each number. Same can be said for the pop instruction. We could have used a function which loads multiple registers and

then increments the “SP” after. Also, pre and post indexed modes could have been used in order to eliminate some lines of code.

1.2 Subroutine calling

In this section of the lab, we needed to create a subroutine that finds the maximum number out of an array of numbers. We began this process by creating a start (or a main method). This contained the starting address of the array, the number of elements in the array and the final result. These three elements are all in separate registers. We then use our Max subroutine. This is quite simple and is essentially the same max as in the previous lab. At first the max is the first number in the array list. Afterwards the current max number goes into a loop. The loop is decremented with a loop counter. The loop basically compares the current max number with the next number in the array, then adjusts the max if necessary. It does this until all the numbers are compared. Finally the max is stored in a register that was loaded back and held the final result.

This part of the lab was quite simple and really helped us understand how subroutines can be useful in assembly coding and how to be able to link the start, max, loop and end all together. All while learning the convention of using subroutines and why we push and pop at the beginning and end of each subroutine and became familiar with the LR register.

1.3 Fibonacci Calculation using recursive subroutine calls

This section of the lab had us find the nth Fibonacci number using recursive calls. In order to do this we followed the basic arithmetic given to us in the lab report instructions:

```
Fib(n):
    if n >= 2:
        return Fib(n-1) + Fib(n-2)
    if n < 2:
        return 1
```

Basically, this means that to calculate the Fibonacci number, we need to call the previous two fibonacci numbers together. The best way to do this is using recursion since recursion always keeps the current number updated. This was perhaps the most difficult section of the lab and was the hardest to get a firm grasp around.

At first, a start (or main) is created which points to a location of a Fibonacci number. A slot is also created for the final answer that acts as a counter. Afterwards, we jump into our subroutine. After, the code checks if the number is less than 2, it moves to

the *ELSE* routine. If it is less than 2, a 1 is stored in the result representing the bottom of a leaf for that iteration, we can add 1 to the final result which is initialized to 0. If the number is not less than 2, then 1 is subtracted. We then loop again within our subroutine and subtract 1 again from that number until we are less than 2. Then the link register remembers where it was in the code before going to the ELSE statement and continues to our n-2 step. Each time the *ELSE* routine is used a 1 is added to the final result which eventually computes the final answer as the Fibonacci result (adding the leafs of a recursive tree).

This section of the lab was perhaps the most challenging since recursion is a tricky idea to grasp beyond coding in assembly language. We used the document provided on mycourses to help us on recursion and looked for help a lot online with how to go about it. This took us the most time but was very satisfying when it worked.

There aren't many improvements that could have been done in this section. Recursion is still a shaky point for us and we can't think of any simpler ways that we could have done this.

2.1 Pure C

The purpose of this section of the lab was to reproduce the max function we created in assembly but in a higher language such as C. This was done quite easily considering the prior knowledge we had from our ECSE 202 class which focused on C coding.

Firstly, we initialized the array and a max value and created a variable to iterate through the array. This was implemented using a for loop. We set our maximum value to the first index of the array (this acts as a temporary maximum value). In the loop, we compare the next value in the array with our temporary maximum value, if the value in the array is greater than our temporary max, it will update our temporary maximum to hold our maximum value. The program iterates through the entire array doing this method of comparison. The program ends by returning the maximum variable.

This section was relatively simple since we had previous C code knowledge and the pseudocode for this section was almost identical to that of the assembly code max. It was also a lot easier since we knew the size of the array and we did not need to design the C code in a manner that would work for any sized array.

2.2 Calling an Assembly Subroutine from C

This last part put everything together by having us integrate subroutine in assembly and C code for the main. We were required to create a code in C that called an assembly subroutine. This section was not too difficult considering that most of the C code and assembly code was given to us. The idea was simple in the sense that we were again finding a max but using an assembly subroutine. The assembly subroutine worked by

comparing the current number in the array and the next number.The two numbers being compared were stored in registers R0 and R1. The results are once again stored in a temporary variable. Finally the program ends by returning the maximum number. The only tricky part of this section was knowing what parameters to give when calling the subroutine, we knew we had to give two numbers, a current element in the array and the one next to it. This made it possible to load R0 and R1 with numbers in order to compare the two.

This part was quite straightforward and there weren't many challenges with it. We tried looking at how the software converted the C code into assembly code but some parts of it were too complicated to understand.