Karl Doumar 260733160 February 23th, 2018

He Qian Wang 260688073

Group 32

Lab 2: Stacks, Subroutines and C

**Part 1 – Subroutines**

**Part 1.1 – The stack**

This part of the lab asked us to create a PUSH{R0} and POP {R0-R2} functions that uses the stack without using the predefined ARM instructions. Basically, this part teaches us and makes us understand how the stack operations and operations should work in the background. From what we understood of the question, we must push the contents of register R0 onto the stack and pop the contents of the stack onto register R0-R2.

The way we have chosen to do this part of the lab is to define four different numbers in memory. The program will fetch the contents of the memory locations using a loop and each time the loop iterates, the next number will be stored in register R0 and it will be pushed onto the stack. The stack pointer will also be updated manually. When the loop is done, we simply load the first three contents of the stack onto register R0, R1 and R2. At each step, the stack pointer will be updated by the value of #4 so that we can accurately pinpoint where we are on the stack.

**Part 1.2 – Subroutine**

For this part of the lab, the purpose was to teach us how to use subroutines in ARM assembly programs. We had to convert the find max of an array from Lab 1. We initialize the program using with 4 numbers in memory along with the number N which is the number of elements in the array. Firstly, we declare our arguments and store the information onto registers R0, R1 and we push the contents onto the stack. Then we branch out to our subroutine using BL MAX.

Onto the subroutine, we free up the contents of the registers R0-R3 to be able to work with these registers. We use the instructions LDR R1, [SP, #20] and LDR R2, [SP, #16] to fetch the contents that we pushed onto the stack initially. Following this part, we simply use the same logic as in Lab 1 to find the max of the array. Basically, it is a loop that uses a temporary variable that stores the current maximum to any other values. If bigger, we update the value of the temporary variable. After this is executed, we simply use BX LR to branch back to the first location in the program. When we have branched back, we simply fetch the information that we pushed on the stack after the subroutine.

**Part 1.3 – Fibonacci**

This part of the lab asked us to make a Fibonacci program that recursively calls itself to find the Fibonacci sum. The purpose is really to help us understand how to do recursion in ARM assembly programs. There are two locations in memory that we defined already which are result and N. Result stores the Fibonacci sum and then N stores the number of elements that we add to the sum. Our program stores the requirements onto registers R0 and R1 and we push it on the stack. The subroutine will be called using BL Fib and we move on to the next part.

The subroutine frees up {R0-R3} to have registers with which we can work with. We initialize a running sum using register R0 and we first check if our value N is less than 2. If yes, we move to the EQUAL statement and we set it as equal to 1. Each time we need to recursively call the Fibonacci, we use BL fib and it only stops if N=1. After everything is added together, we branch back to the main program and we use BX LR. In the end, we store our result in memory using the STR statement.

**Part 2 – C Programming**

**Part 2.1 – Pure C**

In order to have code that runs quickly, one would need further abstraction beyond assembly language. As we know, C takes a few lines of code and compiles them to become many lines of assembly code. The disadvantage to using C as opposed to assembly, is that C compiles into sometimes unnecessarily large amounts of code. During this laboratory session, we used C on its own, then learned to augment C with assembly to reduce the amount of calculations that need to be done. The objective of this section was to write a simple C program to find the maximum number in a list. We approached this problem by inserting a for loop to iterate through the array as instructed in the lab manual. The next step was to write an if-statement comparing our return variable to the value at the current array index. If greater, the return variable would be overwritten. In terms of improving this relatively simple program, we could not think of a way to do so while staying within the bounds of writing in C exclusively. However, as we will see in part 2.2, there is a way to improve the code using calls to assembly subroutines.

**Part 2.2 – Calling an assembly subroutine from C**

For this section of the lab, the goal was to implement the same function as in part 2.1 however using the assembly subroutine handed out in the lab manual. We saved the subroutine file to the same folder as the main C method in our memory for everything to work properly. The subroutine’s advantage is that it is only four instructions long, which could already save us a lot of computational time. We used the disassembly panel of the development software to understand how and where the subroutine was incorporated. The main challenge we had with implementing this algorithm was deciding where to put the call to a subroutine. What we did is keep the for loop we had written for part 2.1 and replace the if-statement. Implementing the max subroutine involved two calls to it. The first was to compare the n+1th element of the array to the nth element. After that, the max of those two elements would be compared to our “running max”. This way we did not overwrite our maximum value. To improve this code in the future we would possibly want to write as much code as possible in assembly, considering this is a fairly simple function. If we had some arbitrary condition to still use C, our main method could simply be calling our assembly subroutine.