ECSE 324 Lab 2 Report

ARCHIT GUPTA (260720757) *KAI LUO (260479742)*

PART 1.1: THE STACK

To successfully create this program, we had to implement our own version of the push and pop instructions without actually using the short push and pop keywords. This part was pretty straightforward as we got our inspiration from the book.

We defined our stack pointer as R12, added the values that we wanted in the stack and then decremented our pointer (by one word). Decrementing the pointer was done instead of incrementing because of the way stacks work, in a LIFO approach. Once all the values are loaded onto the stack (in our case 3), we move over to the pop instruction. The popping was executed by the use of just one statement LDMDA R12!, {R0-R2}. This popped all the 3 values that we earlier pushed at once into registers R0, R1 and R2. The DA stands for 'Decrement After' and the '!' tells the compiler to decrement R12 by one word after each pop (in order to get to the next pop).

The only challenge we faced in this part was understanding how LDMDA works (and its variants). I believe that the only way this could have been improved was by using the actual push and pop instructions.

PART 1.2: THE SUBROUTINE

This part took us quite a bit of time as it was our first introduction to subroutines. We reused the code from part 1 of Lab 1 as was stated in the document. The concept was not very hard. It was only the implementation that was tricky.

We used an array of 4 numbers (which were pushed as the subroutine parameters) but we only pushed 3 of those numbers onto the stack as we wanted to leave the 1st one for comparison purposes (R0). This value in R0 was defined to be the current 'max_value'. After the pushing (R0-R3) was done, we created a loop. The loop iterated through the numbers of the stack and laid them out for comparison. The first number was popped and was compared to the number in our reference register (the current max). If the popped number was greater than the 'max_value' we swapped the values. If not, we let the contents of the registers remain as they were and moved ahead with the popping. After all the numbers have been popped and compared, we can be certain that the maximum number is stored in R0.

It was difficult to improve on this part as the algorithm used was efficient. Plus this was also a good introduction to subroutine calling which would serve us well in the next part.

PART 1.3: FIBONACCI CALCULATION

This had to be the toughest program we've implemented in this course so far and it certainly turned some hair grey. As we were just introduced to the concept of subroutines and on top of that, we had to use recursion.

We had to initialize a bunch of stuff: the number n whose Fibonacci sequence we had to calculate, the number 2 for initial comparison and the number 1 for the first value (base case) that is used in calculating a Fibonacci sequence using recursion (as seen in the pdf). At the starting of the subroutine, the initial comparison checks if the number is less than 2. If so, it exits the loop and the sequence is outputted.

There are two loops in the subroutine: the outer loop and the inner loop. The inner loop calls the previous values, adds them and stores the result in memory. After that, the inner counter is incremented. The outer loop checks if the subroutine has been called for the nth value of the sequence and following that, the outer counter is reset. We made use of the push and pop instructions to properly implement the inner loop.

The main difficulty in implementing this program was understanding how to use recursion to calculate the Fibonacci sequence. Also it was quite difficult to keep track of all the different counters used in the program.

Part 2.1: PURE C

This part was the easiest in this lab and took us very less time to complete. The main body was given and we just had to implement a simple for loop. We defined the maximum element to be stored at the 1st index of the array of numbers. Then we used the 'for' loop to go over the array and compare each number with the current max. If the number was bigger than the current max, we swapped the values. If not, we just continued the iteration. Finally, we got the maximum number in the register where we wanted it to be. Simple!

Part 2.2: CALLING ASSEMBLY SUBROUTINE FROM C

This part was also not too hard to implement as we were already given the method subroutine implementation in assembly. We were also given the partial implementation of the C code in which we just had to call that method (from the assembly) by using the method name and passing the arguments. The 'extern int MAX_2 keyword' was used to link the assembly file to the C Program. After passing in the arguments through the method name, the assembly language program would compute the result and return the bigger of the 2 numbers (contained inside a ‘for’ loop). As this step was done for each iteration, at the end of the program we had the maximum number of the array in our pre-defined location. Overall, not a very tough part as the lab pdf made it quite simple enough to understand and implement what’s going on.

Thank you for reading this.