ECSE 324 - Lab #2: Stacks, Subroutines, and C

Alex Hale (260672475) and Tre Mansdoerfer (260680162)

October 20, 2017

Section 1)a): The Stack

To create a program that demonstrates the functionality of a stack, we recreated the push and pop functionality. We used R12 as our permanent stack pointer, and we operated on it in the same way as the usual stack pointer (SP). We then found the length of the list, and iterated over that list to push each element onto the stack. The instruction STR R0, [R12, #4]! was used to insert the value in register R0 to the top of the stack, then to increment R12 (our stack pointer) by one memory address. Once all the elements are loaded into the stack, we exit the loop. Finally, we use the instruction LDMDA R12!, {R0-R2} to pop the top three values off the stack and into registers R0, R1, and R2, decrementing R12 by one memory address after each pop.

This program was relatively straightforward to write, especially because the instructions required were found in the lecture slides. One challenge was ensuring the loop iterated the correct number of times. Another challenge was researching the difference between LDMIA (load multiple, increment after) and LDMDA (load multiple, decrement after) - we had to determine the direction our stack was expanding and choose to decrement.

This program could be improved by having the program demonstrate that values were successfully pushed from R0 to the top of the stack - the way we wrote it, the program ends by popping the top three values, so the program must be stepped through one line at a time if you want to see the values being pushed onto the stack. A breakpoint could be used, or, since we recreated our own stack functionality, we could have created a separate stack entirely.

Section 1)b): Subroutine Calling Convention

When creating the stackmax program, the structure of the part1 program from Lab 1 was reused. The array was four numbers long because that is the conventional maximum number of values that can be passed into a subroutine as parameters. First, the four numbers were loaded into registers R0 to R3 so they could be passed to the subroutine. Once the subroutine is entered, the values in R1 to R3 are pushed onto the stack, while the value in R0 is left in place as the initial max value. Then, the three values are sequentially popped off the stack and compared to the current max value. Finally, the max value is moved to R0, and returned to the earlier point in the program, where it is stored to the predefined memory location.

There were few challenges with this program, because the structure is very similar to the given code in Part 1 of Lab 1. After reading the hints provided in the lab manual and debugging a few small typos and logical errors, the program was ready to go.

This program could be improved by using a while loop in the subroutine instead of a for loop. Instead of storing the loop counter in R10, a flag could be pushed onto the stack before any of the numbers in the list. Then, the top value of the stack could be popped and compared

with the current max until the flag was reached, at which point the current max could be returned in R0.

Section 1)c): Fibonacci Using Recursive Calls

The Fibonacci program uses a recursive subroutine to find the n^{th} number of the Fibonacci sequence. In creating this program, it was important to write an efficient subroutine that would exit when needed. We first had to initialize the n^{th} number, the number value 2, and the number value 1. The n^{th} number is needed to iterate through the subroutine, 2 is used for the initial compare command, and the number 1 is used for the inner section of the subroutine and as the first values of the Fibonacci sequence. The initial compare statement checks the condition to see if n is less than 2 - if so, the initial value of the sequence is outputted.

Our approach to the recursive subroutine is as follows. There is an outer portion of the subroutine that checks if the subroutine has been recursed for the n^{th} value of the sequence. The inner portion of the subroutine calls the previous values stored in memory, adds them together, then pushes it into the memory at the new specified pointer value. The inner counter increments and the outer counter is reset.

In designing this code, we desired to make it easy to comprehend and follow how it works. This shows in the inner loop, where we write out exactly what defines push and pop commands and implement them. By manually changing the pointer, we illustrated how the push, pop, and pointer commands in assembly should function.

One difficulty is making sense of how to program something like a recursive Fibonacci sequence. It was difficult to come up with the concept of the outer and inner loop and make them work. Once that was solved, the rest of the code was simple. A minor difficulty was accidentally iterating to n+1 when we initially ran the program. However, this was also an easy fix by increasing the inner loop increment by 1.

Section 2)a): Pure C

The pure C program creates a max array finder in C, instead of assembly. A lot of the starter code was originally provided, and our task was to use a for loop and compare statements in C. In creating the for loop comparison, I first set the initial max value as a[0]. The for loop itself iterates from a[1] to a[4], comparing the values individually to the max. The max value is kept until the list iterates to the final element - once this happens the max value is returned.

There were a lot of difficulties coding this, since we have never coded in C before. We were not properly flagging our code, so it was hard to gauge which registers equated to the max value in the code. From trial and error, it was found that R4 represents the max value of the array.

Another difficulty was trying to debug while looking through the disassembly mode. Disassembly is extremely difficult to comprehend without flags, since the code is not similar to the C code. It was hard to debug when not using flags and only using disassembly as the basis for debugging. With flags, this code would have been much simpler to debug.

There is not too much to improve in this code. It is a very basic problem that we are addressing, and our code is straightforward in solving it.

Section 2)b): Calling an Assembly Subroutine in C

To demonstrate the capability of using an assembly program within a C program, we again turn to the problem of finding the maximum value in an array of numbers. First, the assembly program is referenced with its input parameters, using the code provided. Next, the array and working variables are declared, and the first value of the array is loaded into variable c. The length of the array is calculated, then the loop iterates the appropriate number of times to check each value of the array. On each iteration of the loop, the current max value and the current value in the array are passed to the assembly subroutine. The assembly subroutine takes the two input values, compares them, and returns the higher one. Back in the C program, the loop moves to the next iteration. After the last iteration, the value of c is returned.

Since most of this code was again provided, we experienced few challenges in the completion of this project. In fact, the biggest challenge we faced was deciding whether the provided code said `MAX_2` or `MAX.2`! Another challenge was learning how to find the size of an array in C.

This program could be improved by using the first spot in the array to store the current maximum value - this way an extra variable would not have to be created.