**Description, approach, challenge, and improvement**

**The stack**

The challenge is to implement “push” and “pop” operations of a stack without explicitly calling the PUSH and POP instructions in ARM. “Push” means to save a piece of data onto a region of memory. This region of memory, called a stack, is pointed to by the Stack Pointer, a special register in ARM. Specifically, the Stack Pointer points to the piece of data that is added the most recently. To implement the “push,” one needs to store the data to the memory address that is one word (4 bytes) lower than that pointed to by the Stack Pointer. Subsequently, one needs to decrement the Stack Pointer register so it stores the address of the newly added data. In our code, we push 10, 9, and 8 onto the stack in that order. The “pop” operation should allow us to pop multiple values at the same time. To load multiples values from the memory and put them to multiple registers, we use the LDMIA instruction (“Load Multiple Increment After”).



The code loads the data pointed by SP into R0, then it increments SP. It repeats the process and loads data to R1 and R2. Using the LDMIA instruction ensures compactness of the code.

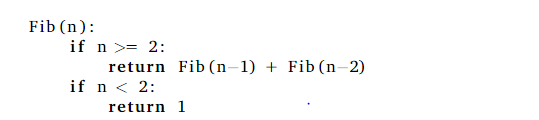
The picture on the left shows the numbers having been successfully popped off the stack in reverse order.

**Find Max in array using subroutine**

This challenge asks us to find the maximum integer in an array by calling a subroutine. We created a subroutine that takes in the number of element in list and pointer to the first element in list. This subroutine, FINDMAX, then returns the maximum number in the list. The logic used in FINDMAX is identical to that in lab 1: As the program goes through the loop, it begins by decrementing the loop counter. For each decrement, the number in the list to be evaluated is loaded into the register R1. The number is compared with the maximum, and if it is greater than the maximum, the value stored in the “maximum” register is replaced in the latter section of the loop. Otherwise, it is skipped over. When the loop counter hits zero, it means that all numbers in the list has been traversed, and the loop terminates. At the end, the value in the “maximum” register is saved. Since the number of parameters are less than four, we used registers to pass inputs. We also followed principle of the subroutine saving register values. There was no need to save LR in FINDMAX since there was no nested subroutines.

**Fibonacci using Subroutine**

The Fibonacci sequence is the sequence satisfying the recurrence relation Fn = Fn-1 + Fn-2 and F1 = 1 and F1 = 1. We intend to create an assembly program that calculates Fn. We use the following algorithm to calculate the sequence using its definition:



In assembly, we sought to create a subroutine called FIBONACCI that performs the algorithm above. This subroutine would compare its input parameter (the value stored in R0 at the beginning of subroutine) to the value 2, and, based on the result, either return 1 (by storing 1 to R0) or compute the sum of Fib (n-1) and Fib (n-2) by calling the subroutine FIBONACCI recursively.

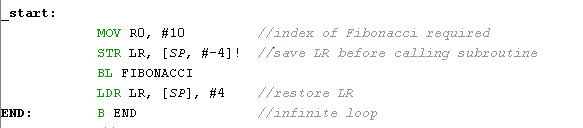
One difficulty was that the return values of the each Fib (n-1) call, stored in R0, would be overwritten as we compute Fib (n-2). In addition, the input parameter (n) for each recursive call would also be overwritten. We need to “remember” n because we need to pass n-2 to FIBONACCI to computer fib (n-2). To overcome these difficulties, we made extensive use of the stack.

In entering the subroutine, we save initial values of R1 to R3 to the stack, as these registers will be modified in the subroutine. In addition, we push the Link Register (LR) to the stack and stores the input parameter in R3. If the input is less than two, we return 1 by storing 1 in R0 and exiting the subroutine. Otherwise, we pass n-1 as to R0 and call another FIBONACCI subroutine to compute f (n-1), the result of which we save to R1. Then, we pass n-2 as to R0 and call another FIBONACCI subroutine to computer f(n-2), the result of which we save to R2. We add the results the f(n-1) and f(n-2), pass the sum to R0 as a return value, and exit the subroutine.

To “exit the subroutine” means to follow the standard procedure of popping the LR from the stack, restoring the value of registers before the subroutine by popping from the stack, and branching back to the link register.

It was important to pop the LR and the saved registers in reverse order as they were pushed.

Our program has a main section that calls the FIBONACCI subroutine.

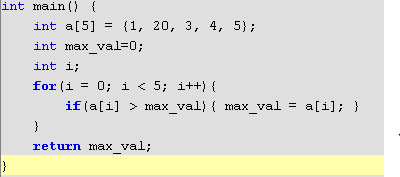


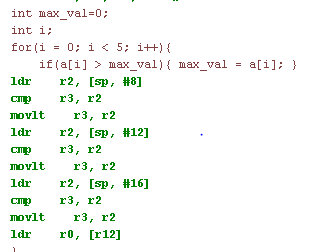
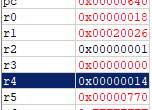
Example: when 10 is passed as input to subroutine, 0x59 is returned.

The current algorithm has an exponential run-time. Dynamic programming algorithms can be used to improve the run-time complexity.

**C-Programming – Pure C**

No significant challenge was encountered. The C program used to find the maximum number in an array is quite simple, and it involves replacing the maximum variable with any item that is larger:



The pictures above show the assembly code, the disassembly, and the result (in R4).

**Assembly routine from C**

This part asks us to use an assembly code segment that returns the max of two numbers. We must integrate this code segment with C code to find the maximum number in a list. The subroutine has the label MAX\_2. We find that a subroutine behaves like a function in C. In addition, the input to that function is passed automatically to R0 and R1, so if we change the registers in the assembly code, the code would fail to find the largest number. Our approach is that we repeatedly compare the maximum to different items of the list. If the list item is greater than the maximum, we change the maximum variable to the value of the item. It was notable that we must use an “extern” statement in the C code to tell the compiler where to look for MAX\_2.

When the input is , the output is , which is correct.