# ECSE 324 Lab 2 Report Group 11

Authors: Aleksas Murauskas, 260718389 and Brendan Kellam, 260718806

## 1. Rewritten Push/Pop

The file stack.s contains the code for this question. The _start assembler directive starts with setting R0, R1 and R2 with three arbitrary immediate values. A branch to “PUSHR0” represents the first portion of this question. The instruction “STR R0, [SP, #-4]!” is used to implement PUSH {R0}. Using the pre-index addressing mode, SP is first offset -4 bytes (allocation of memory) and then R0 is written to that location. The writeback suffix “!” is used to update the SP with the new address. R1 and R2 are then pushed on normally (PUSH {R1, R2}). The POPR0R2 sub-routine has a single instruction, “LDMIA SP!, {R0-R2}” analogous to POP {R0 - R2}. The base instruction is LDM (Load multiple) with an increment after (IA) addressing mode and the write-back suffix (!). Increment after (IA) is necessary since the stack **full-descending,** I.E. a **POP** will cause the SP to descend into lower addresses (SP will increase). Similar to the pusing, the write-back suffix (!) is necessary to update SP after elements have been popped.

## 2. Max Array assembly code

The file max.s contains the code for this deliverable. Within _start R4, R2 are set with the result location and the number of elements in the list respectively. R1 then points to the first member of the array. Label MAX is branched and linked. Within MAX, the state is saved by pushing R4 to R12 and the link register onto the stack. R0 is loaded with the first value of the list and will act as the maximum. The LOOP Label begins with decrementing R2. Once this value reaches zero, the program has looped through the whole array and branches MAXFIN. Within the loop. R1 is incremented to point at the next member of the array. R3 holds the value stored in R1 for comparison with R0 the maximum. If R0 is greater than R3, the code simply branches back to LOOP.  If R3 is greater, R0 is replaced by R3, the new maximum. The program returns to LOOP. Once the loop is terminated, the flow of execution goes to MAXFIN, which pops the stack and returns to link register. The code concludes in DONE by storing the found maximum in the R0 for return.

# 3. Fibonacci

As outlined in the lab 2 requirements, the implementation of the Nth fibonacci number algorithm was recursive rather than iterative. This proved to be challenging in two major aspects:

1. We had to determine how we would go about saving and loading a given recursive step's state (I.E creating/restoring it's stack frame)
2. How branching to the previous recursive step would function

We approached the problem by first looking to the pseudo code provided:

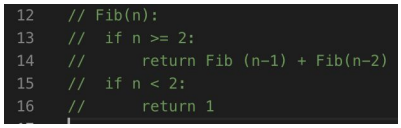


Figure 1: Fibonacci pseudo code

In order to follow the subroutine calling convention, R0 represents the argument n (Figure 1), the Nth fibonacci number to calculate. R4 & R5 are used to compute n-1 and n-2 repsectively. R0 is overwritten with the return value Fib(n-1) + Fib(n-2) **or** 1.

At the beginning of the sub-routine, we immediately push a new stack frame (R4-R12, LR) onto the top of the stack. This allows us to save the **current** state of the registers before making modifications to them. In order to balance the addition of a new stack frame, we also pop the upper-most stack frame at the end of the sub-routine, thus restoring the state to what it was before modification. **NOTE:** R0 is never pushed onto the stack because it's **not** unique to any given recursive step. R0 is analogous to a **static** variable, continually modified to be equal to:

- R4 (n-1) : When performing a branch to Fib(n-1)
- R5 (n-2) : When performing a branch to Fib(n-2)
- 1 : When the base case is reached (return 1)
- Fib(n-1) + Fib(n-2) : When the general case is reached (return Fib(n-1) + Fib(n-2)).

**NOTE:** LR needs to be included in the stack frame as it's **stateful.** For example, returning from any given recursive step, you may be returning to either Fib(n-1), Fib(n-2) **or** Fib(n). This implies that LR needs to be stored for each call of the Fib sub-routine.

We could have made this sub-routine more efficient in two ways:

1. Instead of allocating 40 bytes per stack frame (R4-R12, LR), we could have reduced this to 12 bytes by only storing R4, R5 and LR. These three registers (other than the argument registers, R0-R3) are the only ones being modified, with R6-R12 untouched.
2. We could have utilized memoization, a dynamic programing technique, to increase the runtime of our sub-routine. This would involve developing a hash-table data structure to store fibonacci calculations. Each subroutine call (with R0 representing n) would check against this hash-table to see if R0 is an existing key. If it is, the value represents the R0th fibonacci number, so return that. Else, compute the R0th fibonacci number normally.

# 4. Max Array C only

This program Would find the maximum of an array only using C. First the program sets the first member of the array as the maximum. Then it finds finds the size of the array using sizeof() and stores it in arr_size. Using a for loop, the program increments from 0 to arr_size, checking if the current member is larger than the maximum. If it is, then array member at x becomes the new maximum and the loop continues. The loop terminates once x is equivalent to arr_size. To conclude the program prints the found maximum and returns it.

The current process is O(n) with n being the size of the array. This program could be improved by sorting the array previously, where the maximum would either be at the front or rear of the array. This would make the time complexity of the retrieval of the maximum O(1).

# 5. C code Using Assembly subroutine

The flow of execution begins in the C file Max_Array_WithSub. Outside of main the MAX_ARR  must be defined as an external reference with set parameters, arr to hold the array pointer, and s to hold the size of the array. Within main(), first the array is set. Following that, the array size is found using sizeof() and stored inside arr_size. A for loop is declared to increment from 1 to arr_size. Then the subroutine MAX_2 with The maximum and the current member of the array. Switching to Assembly, the maximum and current member are stored in R0 and R1, respectively. MAX_2 compares R0 and R1. If R0 is greater, the code simply returns to the link register. Otherwise, R1 is greater and a new maximum

is needed. R0 stores the value in R1 and the program branches back to the link register. The result is stored in max_val. At the program's termination max_val is returned.