Lab 2 Report
ECSE 324 — Nada Marawan (260720514)
Prof. Davis — Imad Dodin (260713381)
Demo Date: 22/02/19 — **Due Date: 01/03/19**

1 Subroutines

1.1 The Stack

This task requires us to implement two Stack operations in ARM: **PUSH{R0}** and **POP{R0-R2}**. We note the following:

- The Stack is an address in memory pointed to by a special register known as SP - the *Stack Pointer.*
- The stack stores item in decreasing address order (i.e. if item A is on top of item B on the Stack, it is stored 4 bytes (1 word) below B in memory.

Knowing this we can make the following translations into ARM instructions for the provided PUSH and POP operations:

- **PUSH{R0} - STR R0, [SP, -4]!** - *Store the contents of R0 onto the next slot in Stack, we must perform a pre-decrement to point the stack pointer to next slot, first.*
- **POP{R0 - R2} - LDMIA SP!, {R0 - R2}** - *We use the Load Multiple instruction to POP multiple values from the stack. The **IA** (Increment After) suffix allows us to move the stack pointer to the next most bottom element after retrieving a given element and storing it into its respective register (LDM stores into registers in order of ascending indices).*

1.2 The subroutine calling convention

This task requires us to implement and call a subroutine to find the maximum value within an array. Note that the logic from the previous Lab demo does not require more than four arguments, we may simply pass arguments to the subroutine using the R0-R3 registers, as stipulated by the calling convention. We pass the first element in the list *by reference*, the count of elements in the list *by value* and the result variable *by reference.*

We use the Stack to save the state of the processor at the beginning of the main section of our code and at the beginning of the subroutine (i.e *by pushing the values of registers R4-R7, which are the only registers used during the subroutine, onto the stack.*) As discussed previously, we use the store operation to push onto the stack - for simplicity's sake we can use the *Store Multiple* instruction to do so, decrementing our reference address before each store operation. In essence we then have **STMDB SP!, {LR, R4-R7}** operation which handles saving the state of our processor.

We then restore the state of the processor by popping off of the stack at the end of the main section and of the subroutine - we use the **LDMIA SP!, {LR, R4-R7}** to do so. Note that we may maintain the order of the registers that we provide as STM stores from registers in order of descending indices, and LDR stores into registers in order of ascending indices. We store the return value of the subroutine into **R0**, as required by the calling convention, directly before restoring the state of our processor with the Pop operation.

1.2.1 Improvements

An error that we made stemmed from our misunderstanding of the calling convention: in essence, we passed arguments into the subroutine with registers **R1 - R3**, assuming that the call convention gives us freedom to do so, however, the call convention requires us to pass arguments in *sequential registers, starting from* ***RO***.

1.3 Fibonacci calculation using recursive subroutine calls

We again use a sandwich to save and restore the state of our processor to and from the Stack. The main section of our code loads the desired Fibonacci index from memory. The subroutine begins by saving the state of the processor, **including the Link Register** (crucial for further recursive calls). The subroutine checks whether our base-case has been reached, if it has then it returns a value of 1 via the **R0** register. Otherwise, we initiate a subroutine that makes recursive calls for **N-1** and **N-2**, passing the values into **R1** (should be R0), when making calls. We sum the return values of the recursive calls and return the result into **R0**.

1.3.1 Improvements

We can improve the simplicity and thus efficiency and readability of our code by implementing the recursive calls and checking of the base cases in a single subroutine (as opposed to doing them in separate subroutines). This reduces the total overhead associated with making each subroutine call (e.g. memory use associated with pushing onto the Stack).

Another improvement that could be made is implementing a Dynamic Programming approach to the problem - in essence, this means, that we store the Fibonacci result for a given **N** and retrieve this result in any case where we would need to recompute the Fibonacci result for that **N** (this is known as the elimination of re computations of overlapping sub-problems). This technically increases the *Space Complexity* of our solution, however, this solution proves beneficial when also taking into account the *space overhead associated with each recursive call.*

2 C Programming

2.1 Pure C

We implement a simple C program that finds the maximum value within an array. We begin by assigning the first element of the array as the maximum value. We then do a linear scan of the remainder of the array, checking whether any value exceeds the current maximum value, updating the maximum value accordingly. After the loop terminates, we return the maximum value.

2.1.1 Improvements

This task was fairly trivial to implement, given our previous knowledge of the C programming language. Nonetheless, it was interesting to see how we can implement the same functionality while using calls to Assembly subroutines in the next task.

2.2 Calling an assembly subroutine from C

This task required us to integrate Assembly and C to find the maximum element in an array (the same as the previous task).

We begin by declaring a comparison subroutine that takes in two integer arguments, within the C program. We use the **extern** keyword to indicate to the compiler that this is defined externally and must be resolved. We give it the name MAX. We define the subroutine in an assembly file, and use the **.global** keyword to, likewise, name the subroutine MAX_2. The subroutine code is provided, and takes in argument values in the **R0** and **R1** registers, as is required by the C calling convention, and returns the result in the **R0** register, again following the calling convention.

2.2.1 Improvements

There were no real improvements to be made to our solution, however, we noted how the C calling convention works in the provided subroutine. Indeed, the first argument was passed in **R0**, and the second in **R1** and the return value was stored in **R0**.