Carlo D'Angelo 260803454
Mostafa Abdelwahab 260551876

Test Program with Rewritten PUSH and POP Instructions

The approach for this part of the lab was quite straightforward. Instead of using the PUSH and POP instructions, we used the STM and LDM instructions respectively.

1. PUSH {R0}

To test the rewritten version of this PUSH instruction, we used one register (R0) and loaded into it a value of 5 (value itself does not really matter). Then, we wrote the following instruction: **STMDB SP!, {R0}**. By writing the STM instruction in this form and using the stack pointer as the base register, we replicate the effects of PUSHing R0 onto the stack (memory). Using the DB (decrement before) suffix with the STM instructions allows the address stored in the stack pointer to be decremented before any data is stored, which corresponds with the properties of a PUSH instruction. Moreover, this form of the STM instruction is in pre-indexed mode meaning that the address stored in the stack pointer will be updated with a new address after every decrement.

Then, we added 1 to the value stored in R0 (making the value stored in the stack different than the value stored in R0). Finally, we wrote an LDM instruction using a similar format to simulate a POP instruction (i.e., POPing the value stored at the top of the stack back into R0), specifically, **LDMIA SP!, {R0}**. Using the IA (increment after) suffix allowed the address stored in the stack pointer to be incremented after a given data transfer, which also corresponds with the properties of a POP instruction. Now, if the value stored in R0 was the same as its original value in the beginning, we would know that the rewritten PUSH and POP instructions were indeed implemented correctly.

2. POP {R0 - R2}

To test the rewritten version of this POP instruction, we simply copied what we did in the first part, but used three registers instead of one.

Assembly Code which Computes the Max of an Array using an Assembly Subroutine

To complete this part of the lab, we converted the program from Lab 1 for finding the max of an array into a program which uses a subroutine and conforms to the calling convention specified in the Lab 2 instructions.

Before calling the subroutine, we placed two arguments in registers R0 and R1 respectively: a pointer to the data array and the size of the array minus 1 (i.e., N – 1). To call the subroutine, we used the BL instruction, which is a special type of branch instruction that stores the address of the next instruction in the program in the link register (LR). Then, to ensure that the state of the processor is restored to what it was before the subroutine call, we pushed R4 and R5 (registers

within R4 – LR that we change in the subroutine) onto the stack at the beginning of the subroutine and popped R4 and R5 off the stack at the end of the subroutine.

As for the subroutine itself, most of the code was reused from Lab 1. The most significant difference was that we returned the result (i.e., the max) in R0 before exiting the subroutine. We had to do this extra step because it was specified in the ARM assembly subroutine calling convention. Furthermore, to actually exit the subroutine, we used the BX instruction and wrote BX LR since LR holds an address pointing to the next instruction in the calling code.

Fibonacci Program with Recursive Subroutine

The file fibonacci.s is an assembly program that calls a subroutine that requires one parameter n (an integer stored in R0) and returns F(n) in R0. The program then stores the value in the main memory in an assigned location.

If the integer n is smaller than zero, the subroutine returns -1 in R0. If the integer n is smaller than two, the subroutine returns 1 in R0. If the integer is bigger than 1, the subroutine creates two local variables with the value (n-1) and (n-2) in R4 and R5 respectively. The values of R4, R5 and LR are then pushed to the stack. R0 takes the value of R4. We then call Fibonacci on the current value in R0 (n-1). After this call returns, we store the returned value (R0) to the top of the stack (equivalent of local R4), we then load into R0 the equivalent of local R5 (n-2) and call Fibonacci on it. After this second call, we can pop the values in the stack back to R4, R5, LR. We add the value of R4 (F(n-1)) to R0 (F(n-2)) in R0. Now R0 stores f(n-1) + f(n-2)=f(n). We can return the main caller.

This method to compute Fibonacci numbers involves many repeated computations. In fact, when we compute f(n) using this method, f(n-k) is calculated 2^{k-1} times. Also, each f(x) calculation call involves 3 stores in the stack. Eventually the stack can fill up pretty quickly: up to around 3n values ae staked at a given moment when we are computing f(n). A better solution does not involve stacking: building Fibonacci from the bottom up using an array that can be accesses by index. Another quick method (but requires precision) is using the direct formula involving the golden ratio Φ.

$F(n) = \frac{\Phi^n - (-\Phi)^n}{\sqrt{5}}$ where $\Phi = 0.5*(1+\sqrt{5})$

C Code which Computes the Max of an Array using C

The disassembly of pureC.c compilation shows us what a compiler really does. There are previous blocks of assembly instructions running before branching to the label main.

C Code which Computes the Max of an Array using an Assembly Subroutine

Calling assembly subroutines from within a c file shows us the importance of respecting the calling convention. The int max_val takes the value of the last value stored in R0 in the subroutine. Also the registers for the parameters used in the subroutine are automatically R0 then R1.