McGill University
Computer Organization
ECSE 324

Lab #2

Group 20
Emmanuelle Coutu-Nadeau (260681550)
David Kronish (260870097)

February 10, 2020

Part 1: Subroutines

1. The stack

The first section of this lab was meant as an exercise to understand how the PUSH and POP operations are done in ARM Assembly when using a stack. More precisely, the first task of this lab was to make a program that would achieve the same result as PUSH{RO} and POP{R0-R2} without using these instructions directly.

To achieve this goal, we decided to divide our code into two subroutines called PUSH and POP. The approach for the PUSH operation was to iterate through the given list of numbers and store the current element at the location of the stack pointer. Once the push counter reaches 0, we know we have pushed all the elements in the original list. We then branch to POP. The pop subroutine is fairly simple. We load the value of the top of the stack into R0, then R1, and lastly R2. Here, we needed to be careful about using the correct addressing mode when updating the SP. We decided to use the post-index mode directly in our load instructions. This way, 4 is added to the SP after each "pop", so it points to the next element "below" in the stack.

A challenge we faced while coding the push operation was with decrementing the stack pointer. After trying out different solutions, we understood that we needed to decrement the SP before storing the new value to store on the stack. To do so, we used the pre-index mode when storing the current value of the list on top of the stack.

2. The subroutine calling convention

The second part of the lab asked to implement a subroutine to find the maximal value of a list of numbers. The constraint on this task was that we needed to implement this subroutine using the *callee-save* convention. In other words, the subroutine that is called must take care of restoring the registers it used during its execution.

In our program, we started by initializing registers to hold all elements in the list to registers above R3. The reason for this is that we wanted to keep R0 to R3 available for our subroutine's "arguments" and return value. The caller then pushes the elements of the list into the stack. This is a useful step if our list of numbers holds over 4 elements, since we can only pass variables through R0 to R3 in from the subroutine. After pushing, we BL to the MAX subroutine. The counter that was initialize to the number of elements in the list is passed in the "function" and decremented after every pass. We then pop the top of the stack and compare it to our first element of the list. If it is bigger, than we replace R0 with the new max, else we branch again. Once the counter reaches 0, we have gone through all the elements in the list and we then branch to DONE that branches to LR. LR is the location of the instruction after we branched to MAX the first time. The return value that was stored in R0 and the program terminates.

A challenge in this portion of the lab was to make sure we were correctly following the convention. At first, our program was working fine, but we were using registers not within R0-R3 as arguments. Also, restoring the values of the registers was quite puzzling. We needed to make sure that they were not re-initialized after each pass of the subroutine. The stack was quite useful in making this step easy. Pushing values onto the stack ensured that we were not overusing registers.

3. Fibonacci calculation using recursive subroutine calls

In the third section of the lab, we were asked to write an assembly program that uses a subroutine recursively to calculate a Fibonacci number. In the pseudocode that was given, we can understand what the subroutine needs to do more precisely. The main steps are to compare the input value (n) to 2 and make a recursive call if the value if over or equal to 2. The final step is the base case, where we return 1.

To implement this function in an Assembly subroutine, we started by initializing the input value (n) and storing it in the stack with LR. Then, we call FIB with BL. At this point, LR holds the address of the instruction after the first function call. In the subroutine, we start by storing this LR in the stack. If R0, our input, is bigger or equal, than we proceed to the first section of FIB. This block of code pushes all “n-2” values into the stack and call FIB again but with R0 now being n-1. Once R0 reaches 0, then we go to are base case. The base case will then overwrite the content of the R0 register with 1 and go the instruction POP{R2} that was previously stored as LR in the stack. Then the second block of FIB executes and calls itself again. Since R0 is 1, the base case gets called and pops the top of the stack into LR. This will go to the MOV R2, R0 instruction. We pop the value at the top of the stack into R1 and perform the addition of R1 and R2 into R0. This last sequence will repeat since we are BL to the MOV operation that was previously pushed onto the stack. Lastly, the function returns to the start of the program, stores R0 into a RESULT variable and restores the original values of the registers.

This program was very challenging to write for many reasons. First, the recursion itself can be difficult to understand in an Assembly program, particularly for new programmers. Understanding how to use the stack and linking operations can be difficult. In addition, the Fibonacci algorithm not only requires calling the function with n-1, but also with n-2. It took a long time to figure out how to properly push and pop the sequence on the stack to get the sequence of addition we want in the end. Although this was a difficult task, we have found some techniques that were very helpful, such as drawing the stack by hand.

Part 2: C Programming

1. Pure C

2. Calling an assembly subroutine from C