Stacks, Subroutines, and C - Laboratory #2 Report

ECSE 324 - Computer Organization

October 28th, 2018

Sarah Swanson - 260660847

Bryan Jay - 260738764

Group 19

Introduction

The purpose of the second laboratory is to learn to use subroutines, the stack, to program in C, and to be able to call code written in assembly from code written in C.

Part 1 - Subroutines

1.1 The Stack

The purpose of part 1.1 of laboratory 2 is to understand the stack and when to use it in ARM assembly. It is important to use a stack when there are not enough registers for a program and also for saving the state of the code outside of the subroutine when calling subroutines. Our challenge is to write PUSH and POP instructions without using the PUSH and POP methods.

The way we wrote the PUSH method is to decrement the stack pointer by 4 to point to the first spot in the stack and then load the number from the array and store it at the top of the stack. Then decrement the stack pointer by 4 again to point to the next spot in the stack to be able to push another number onto the stack. We used a '!' to ensure the stack pointer gets moved before pushing. The way we wrote the POP method is to load the number at the top of the stack (stack pointer is pointing to it) into a register and then increment the stack pointer by 4 to be able to pop the next element.

There were no challenges faced and nothing we could have improved because we learned how to PUSH and POP using our own method during the helpful tutorial.

1.2 The Subroutine Calling Convention

The purpose of part 1.2 of laboratory 2 is to understand the use of the subroutine calling convention in ARM assembly. The basics of caller and callee are explained when using subroutines. In order to call, the caller must move arguments into registers R0-R3. If there are more than four arguments, the caller should use the PUSH method to push arguments onto the stack. Then the caller must call the subroutine using BL. The callee has to move the return value into R0. The callee must also make sure that the state of the processor is restored to what is was before the subroutine call. Finally, the callee must use the BX LR to be able to return to the calling code.

Our challenge was to use our program to find the max of an array in laboratory 1 and insert a subroutine in it. We used registers R0-R3 in the subroutine and pushed the

register contents on the stack at the beginning of the subroutine in order to save the state of the registers. We popped them off at the end of the subroutine. We found the max value as explained in laboratory report #1, and the value was stored in the stack pointer location. After the subroutine, the branch link is called and the max value is stored into the memory location RESULT. The link register and stack pointer are restored.

A challenge faced was understanding the branch loop back to the link register. Once we understood this, implementing the code was simple when using subroutines. An improvement for our code could have been to make it a lot cleaner because we used stack pushes that could have been avoided.

1.3 Fibonacci Calculation using Recursive Subroutine Calls

The purpose of part 1.3 of laboratory 2 was to write the fibonacci sequence for a number ‘n’ in assembly language. To do this, a recursive subroutine was used to find the two fibonacci numbers that add up to the ‘n’ number.

The way we executed the fibonacci code was to have a loop that printed out all of the N-1 numbers down the fibonacci tree. The numbers were pushed onto the stack one by one to keep track of which iteration of code we were on for the N-1 fibonacci number. This loop would continue until it went through the full counter. After this, we would decrement for the N-2 section, which was going back up the tree to fill in the remaining numbers. To do this, we used a recursive call to go through the Fib loop again, working up the tree until all the numbers were filled in for the given ‘N’ fibonacci number. At the end, we used the final value stored in the stack to add both of those numbers together.

A challenge faced while writing this code was properly using the branch link statements. The way branch link works is when you branch into a subroutine, the next instructions are saved into the link register. Another challenge was the properly use the stack pointer and when was the right time to push and pop teh registers. An improvement could have been to have cleaner code and subroutines because we redid the same process several times.

Part 2 - C Programming

2.1 Pure C

The purpose of part 2.1 of laboratory 2 was to get used to using C and understand how the compiler translates the code from C into assembly language using the disassembly viewer. Our challenge was to initialize a C project within the ARM Processor and to fill code for a loop within the code provided. The loop iterates through the array to find the maximum number.

We implemented this in C by finding the length of the array, then iterating through the array until we checked the full length, and checking if each value is greater than the maximum value. We then return the maximum value.

A challenge faced was switched from coding in Assembly to C. It was difficult to find the array size in the beginning, but we used the function " int length = sizeof(a) / sizeof(a[0]) " which is defined as the way of finding the array length. To make this code more effective, a bubble sort system could have been used. This would have decreased the amount of iterations, therefore speeding up the time required to find the maximum number within the array.

2.2 Calling an Assembly Subroutine from C

The purpose of part 2.2 of laboratory 2 was to understand how to mix C and assembly language. Our challenge was to write a C program that calls an assembly subroutine. The C code will call an assembly subroutine to find the maximum of two numbers. We were also told to put breakpoints to find the main and MAX_2 section in order to see the processor run the two sections of the program.

To write this program, we used C to find the length of the array, iterated through the array and compared each number with the maximum value by using a subroutine to go to MAX_2 (external function, using a global directive) which finds the maximum using assembly language. It then sends you back to the C code and stores the maximum value. The subroutine will know where the result is stored and its data can be retrieved in the C program.

A challenge for this was to understand how to link the programs, which we learned was done by making an external function to connect to MAX_2 at the top of the C program.