ECSE 324
Computer Organization
Fall 2019

Date: October 27 2019

Lab 2 Report

Group 1

Preyansh Kaushik : 260790402
Elie Elia : 2607959306

1.1 The Stack

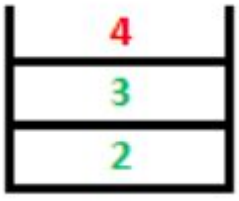
Description:

For this part, we had to simulate the data structure of a stack. Stack's work in a LIFO - Last In, First Out manner with two common operations: push and pop. The push operation adds an item onto the stack, while pop retrieves the top-most element from the stack. In this part of the lab, we created a stack and successfully pushed and popped items from it. To demonstrate a stack, we had to create an assembly program that would push the values 2,3, and 4 onto the stack, and then pop these elements.

Approach taken:

To do this, we decided to use the built-in mnemonics of PUSH and POP operations, although we could have implemented the stack with Load/Store instructions as well. We used a temporary register R0 to hold the value that was going to be pushed to the stack first. With the MOV operation, we load R0 with a particular value. For example, MOV R0, #2 would load R0 with the value 2. After loading the register, we used the PUSH mnemonic to push the contents of the register onto the stack. We did this by PUSH {R0}. This process was repeated with different values to push other items onto the stack, reusing the register R0.

By the end of the process, our stack resembles the following in memory:



After our push operation, our program jumps to the POP operations. We used the POP mnemonic, directing the popped value to be stored in registers R1, R2, and R3 sequentially. We achieved this by using POP {R1-R3}. By the end of the program, we had different values 4, 3, and 2 in the registers respectively. We could verify that the program worked properly since R1 had the top-most value (4) as the last in-first out sequence is implemented.

Challenges:

The main challenge with this part of the lab was being familiar with the stack operations PUSH and POP.

1.2 Min of Array Subroutine

Description:

For this part of the lab, we had to find the minimum of an input array of 3 elements using a subroutine and stack operations. The value of minimum could then be returned to R0.

Approach taken:
The code simply consisted of a caller and callee, the caller would move arguments from R0 through R3 and call the subroutine using BL DOMIN. The callee would move the return value into R0, ensure the state of the processor was restored, and finally, BX(branch and exchange instruction set) in addition to LR (Link Register) were used to return to the calling code.

The algorithm behind finding the minimum element consisted of setting R2 as our loop counter (the number of elements in our input array = 3). R1 acted as a pointer to point to the next element in the list and this value was stored in R3. A comparison between R3 and R0 would then take place to see if R3 held a smaller value than the already stored “minimum” value in R0. If so, the value of R3 would be moved into R3 and the loop would continue checking the rest of the input elements. Otherwise, the loop will have ended because the counter in R2 will have reached zero, and a return to the calling code would be executed.

Challenges:

The logic of the MIN subroutine was similar to the comparisons that were implemented in the algorithms used for Laboratory 1, and so were not a challenge to modify. The only challenge faced here was gaining familiarity with the calling of subroutines in ARM using the BL instruction and using the BX and LR instructions to return to the calling code.

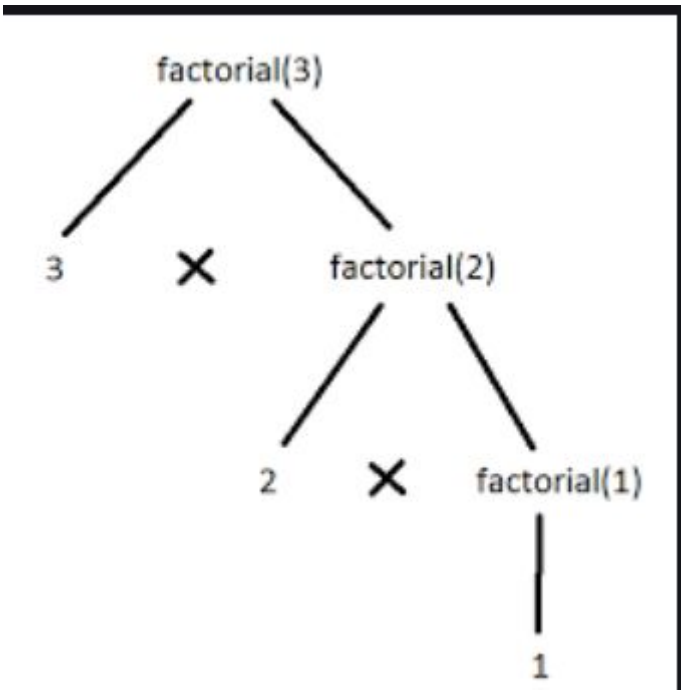
1.3. Recursive subroutine for factorial

Description:

For this part of the lab, we were asked to implement a recursive subroutine to calculate the factorial of an integer number. Our program had a main section which would call the subroutine recursively, which would implement the factorial recursive algorithm.

Approach taken:

Our approach for the factorial subroutine was to do it in a recursive manner by using Stacks. The expected behavior for our algorithm is best illustrated with the diagram below:



We put our code into three segments: Start, Factorial, and the End of the Factorial method. To start, we set the value in a register with the input that we are calculating the factorial for. In this example, since we are setting the register to 3, we use **MOV R0, #3.** After this instruction, we branch to the factorial part, **BL FACT.**

In the Factorial loop, we preserve the contents of Register R1 by pushing it into the Link Register. **PUSH {R1,LR}.** We have to take care of our base case, when the value in R0 goes to 0 as well. The base case for factorial for when n=0, should return 1 as 0! is 1. We handle this base case by comparing the value in register R0 with 0, and if it is equal we set R1 (result or R0!) to 1 and then branch to Done.

To do this, we use **CMP R0, #0** to check if the base case is reached. In the case the zero flag is set, **MOVEQ R1, #1** to set the result to 1. In all other cases, (i.e. in the recursive case), we preserve the value of R0 into R1 by **MOV R1, R0.** In the example of computing the factorial of 3, we will move 3 into R1 at this step. After this, we subtract R0 and recurse on that to find (n-1)!. We can subtract R0 by simply calling **SUB R0, R0, #1.** The recursion induced by **BL FACT** will now work on the contents of **R0** and **R1.**
The recursion will find the value of 2! in this case, and store that in R0. After this, we multiply R0 with R1. R1 is the value we have preserved, and R0 is the recursive answer of the (n-1) factorial.

Challenges:

The challenging part faced here was to figure out how to use the link register throughout the code with the aid of the stack in order to move down and up the recursion tree. An improvement to this algorithm would be to have a tree object within ARM, allowing for a much simpler recursive algorithm.

2.1. C Program

Description:

This part of the lab simply required us to write code in pure C to find the minimum value in an array. This was a very straightforward piece of code to write, as we had dealt with the C language extensively before in previous courses.

Approach taken:

The approach taken was to iterate through every element in the input array, comparing the current stored minimum value (initialized at 0) to current element pointed to in the array. If the value of the current element in the array was smaller than the stored minimum value, we would update the current minimum value. After iterating through every value in the array the variable holding the current minimum would be the minimum element in the array.

Challenges:

We have had extensive programming experience in the C language, making the implementation of our algorithm a simple task. This program however acted as an important setup for the integration of part 2.2 of our assignment.

In terms of improvements that could make the program more efficient, there doesn't seem to be any significant improvements for our code, as it only consists of basic C code to find the minimum element without the use of libraries.

We could have used Scanf to take in user input and make the program more dynamic. Rather than have the list hardcoded into an array this would allow the user to more easily input new lists to find the minimum value. We did not implement this as it wasn’t a requirement but it would be a great improvement to the program

2.2 C Program with assembly

Description:

This part of the lab was intriguing, it was interesting to see how you are able to call code written in ARM from code written in C, given some code in ARM which found the minimum of

2 numbers, we were to implement some code in C which called this code and found the minimum value in an array.

Approach taken:

This was fairly simple to do, a loop was used to iterate through the contents of the array whilst calling the MIN_2 subroutine written in ARM which would compare the current minimum to the current value in the array and return the minimum of the two.

Challenges:

No challenges were faced in this part of the assignment as we were given a thorough explanation of how to implement such an algorithm. No improvement could be made to this code as it seemed very straight forward and compact, it will be interesting to see in future labs a more in-depth example of how calling code in ARM from C works.