ECSE 324 - Computer Organization

Lab session 002

Winter 2020

Lab Assignment 2 – Stacks, Subroutines, and C

Group 05:
Narry Zendehrooh Kermani 260700556

Imane Chafi 260847716

The stack

The main purpose of a stack is to store data temporarily. In the assembly language, the number of registers is limited, so it is useful to use a stack for additional storage. Stack uses last in, first-out rule which means the last element pushed onto the stack will be the first element to be popped. The purpose of this part is to recreate the functionality of a stack without using generic PUSH and POP. We must find an equivalent for PUSH and POP instructions. Initially, we move three different values into R0, R1 and R2 registers. Then we push the value of each register into the stack by using STMDB instruction. STMDB instruction is equivalent to PUSH. STMDB stands for Store Multiple and Decrease Before.

It means that it will decrease the stack pointer first so that it points to an empty byte to prevent overwriting the existing value in the stack. Then it will store the value of the register in the stack. Then we pop all the values from R0 to R2 by using LDMIA instruction. LDMIA stands for Load multiple and Increase After. It means that first, it will load the value into the register and then increase the stack pointer to point to the next value in the stack instead of empty space. The challenge in this part was to find an alternative for PUSH and POP instructions. An improvement to this part is that we can put the entry values in an array then load from it so that we can use a smaller number of registers.

Finding maximum using subroutine

In this section of the lab, we create a subroutine that finds the maximum value from an array of numbers. The Callee-save convention is used in this section so that the subroutine saves registers it wants to use on the stack and then restore the original value to the registers after it is finished using them. First, we load the first element and the number of elements onto the registers. Then we push them together with the linked register into the stack. Then we call the subroutine MAX_ARRAY. In the subroutine, we push registers from R0 to R3 into the stack that we will use in the subroutine. Then FIND_MAX is a recursive function that compares 2 elements and keeps the greatest element in R0. We restore the registers from R0 to R3. Then we branch back to the start and store the value of R0 which is the maximum value, in the memory. The challenging part of this lab was learning the convention of the subroutine and getting familiar with the callee-save convention. An improvement is to have an instruction that could restore the state of the processor to what it was before the subroutine was called.

Fibonacci Calculation using recursive subroutine calls

In this section of the lab, we have to calculate the n^{th} number of the Fibonacci sequence. The Fibonacci sequence is implemented using the pseudo-code shown by figure 1. We use a recursive subroutine which calls itself on Fib(n-1) and Fib(n-2).

```
Fib(n):
    if n >= 2:
        return Fib(n-1) + Fib(n-2)
    if n < 2:
        return 1
```

Figure 1. Fibonacci pseudocode

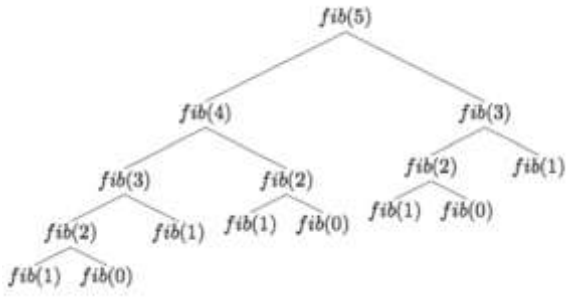

Figure 2. Fibonacci sequence tree

The recursive subroutine creates a tree, as shown on figure 2, that will first traverse through the right side of it and computes all the values and then goes to the right side of the tree. Then, it will move up the tree adding all values pushed onto the stack previously. Two registers keep the values for the result and the Fibonacci number. Then, we go into the FIBONACCI subroutine and load the link register back. We push R1 and R2 every time we do Fibonacci. The base case is 2 so that when Fib reaches there then we know that we are at the leaf, so we move up. Otherwise, we jump into the recursion instruction. We call FIBONACCI for n-1. We will keep calling FIBONACCI until we reach the base case.

Then we will do the same thing for n-2. In the end, we sum the values on both sides of trees and save it in R0 and branch back to the start and store the result and end the program. This section of the lab was perhaps the most challenging part since recursion is a concept. It was difficult to figure out how to use the link registers throughout the code in order to move down and up in the tree. Since recursion is the best method that we know to calculate Fibonacci number, there are not many improvements that could have been done in this section. However, an improvement could be that ARM can have a tree object so we can implement a recursive algorithm easier.

Pure C

The Pure C program simply finds the maximum value from the input array. The main purpose of this section is to reproduce the max function we created in assembly but in a higher language such as C. Most of the starter code is originally provided, and our task is to create a for

loop to compare the elements. We first initialize the array and a max value variable. Then we set the max value to the first element. The for loop then will iterate through the second element to the last element. Each element will be compared to the max value. If the element is greater than the max value, then the max value will be updated with that element to keep the greatest element. Once the loop is terminated, the max value will be returned as the greatest value in the array. For those who have never coded in C, this code might be a bit difficult, but for our team, it was an easy code to write. However, even though the code is a simple C code, the generated assembly code in disassembly mode was hard to understand. An improvement to this code is to combine C language with assembly language so that we can take care of the comparison in assembly which is a lower-level language and it can be done faster than C.

Calling an assembly subroutine from C

In this section again we go back to the problem of finding the maximum value, but this time it is easier because there are only two elements to be compared. We will compare these two numbers and return the greater number. However, this time we will demonstrate the capability of using an assembly code within a C program. At the beginning of the code, there is an external reference to the assembly code with its input. Then we declare ‘a’ for the first number, ‘b’ for the second number and ‘c’ to keep the result. We can set any random number for a and b. Then we will use the external reference which is Max_2 that is a code in assembly. It takes two inputs which in here are a and b and loads them into the registers. Then it compares these two numbers as we did in previous sections, and it will return the greater number. Then, the greater number will be stored in variable c. Then the C program will return c as the final result which is the greatest element. We did not face any challenges for this part since all the codes were provided to us. This code can be improved by taking inputs as an array rather than a single variable for each then by revising the assembly code a bit we can give an array as the input. Then, we can compare any number of elements we want. So, the code will be more flexible and functional.