ECSE 324 LAB REPORT #2

Name: Ameer Ibrahim Osman

ID: 260682723

Name: Ahmed Abdel-Magied

ID: 260653515

Throughout this lab we were able to successfully deliver 5 algorithms, first of all we learned how to use subroutines and the stack, program in C, and call code written in assembly from code written in C. We worked with the DE1-SoC Computer System, which is composed of an ARM Cortex-A9 processor and peripheral components located on the FPGA found on the DE1-SoC board. We have also used an IDE, Intel FPGA Monitor Program 16.1 which was used to ‘compile’ the code and load it on the board. The FPGA Monitor Program also has a debugging tool that has allowed us to step through the code through several stages and verify that our code works. It also allowed us to check the memory so we can verify that our registers had the correct values in any given point.

1 Subroutines:

* 1. The stack

Within this part of the lab we implemented the PUSH and POP instructions. We realized that pushing and popping can be implemented without using the PUSH and POP instructions but by using other ARM instructions. So basically, for this part of the lab we applied the “push” and “pop” instructions to access the stack. The process is very simple what we did is that we pushed values found in the same register onto the stack and then we popped these values. We implemented the push function by using the STR function and before each push of each value, we already know that the register’s address is set to the highest address in the stack initially, then we decrement that address by 4 bytes and stored a new element in it. Then we implemented the pop function by using the LDR function and then we loaded the value at the current address of the register ‘sp’ to another separate register and then we incremented the address of ‘sp’ by 4 bytes. And then after we pushed and popped all values the code ENDs.

Improvements:

This is the shortest way we were able to push and pop values on and off the stack and no further improvements can be made.

1.2 The subroutine calling convention The convention which we used for calling a subroutine in ARM assembly is as follows. First of all, the caller must, move arguments into R0 through R3, then we had to call the subroutine using BL. Through the callee method we were able to move the return value into R0, also we were able to ensure that the state of the processor is restored to what it was before the subroutine call, and then we used BX LR to return to the calling code. We used our program from Lab 1 for finding the max of an array into a program which uses a subroutine. And then we programmed our subroutine program to return the max in R0. This was done by saving and restoring the state by pushing R4 through LR onto the stack in the start of the subroutines then popping R4 through LR from the stack when the subroutine is finished.

Improvements:

We could have implemented the “MAX” and “MIN” counters into one counter and used one loop to find both values. This would have made our code twice as faster and would have decreased our overall code execution time. Another improvement is that we could have added more loops into our code, using the link register and stack to branch between these loops.

* 1. Fibonacci calculation using recursive subroutine calls

Within this part of the lab we were able to write an assembly program which computes the nth Fibonacci number. Through our program we were able to implement a main section which calls the Fibonacci subroutine recursively for the following pseudocode:

Fi b (n): **if** n >= 2 : **return** Fi b (n−1) + Fi b (n−2)

**if** n < 2 :

**return** 1

This code works as a recursive function that calls upon itself on Fib(n-1) and Fib(n-2). The program initially computes the sum of Fib(n-1) and then moves onto Fib(n-2). R0 is used to store F(n), R1 is used to store F(n-1) and R2 is used to store F(n-2). Then we commuted the Fibonacci number of every value before the given n, decrementing using n-1 and n-2 until we reach Fib (0) and Fib (1). Then we added all the values that were pushed onto the stack in order to present the final product of the Fibonacci calculation and that was done using the recursive subroutine calls.

Improvements:

We could have implemented the ‘memoization’ technique, by storing some of the Fibonacci numbers. This would have our code twice as faster.

2. C Programming

2.1 Pure C

**int** main () {

**int** a[5] = {1, 20, 3, 4, 5}

**int** max.val;

**return** max.val;

}

Through this lab we were introduced to code in C language. What we did is that we created a new project by performing the same steps as you performed for an assembly project. The only difference is that when the New Project Wizard asks what program type you would like we selected CProgram. Then we clicked the box next to “Include a sample program with the project”, and then we selected the “Getting Started” program. After that we deleted all the code in “gettingstarted.c” and replace it with our C program. We initialized an array that contained 5 integer elements and an int variable called max\_val that stored the maximum array of elements. Then we implemented a for loop that iterated through the array in order to find the maximum number. What this array did is that it went through the array elements once. Inside the loop, we assigned max\_val to the first array element and then compared it to the next element, If the next was greater than the current value then we swapped both elements, else then it stayed the same. This procedure keeps on going until the maximum number was found.

Improvements:

We could have implemented the max value function in many different ways which could have been more efficient in terms of time or memory depending on the system requirements.

2.2 Calling Assembly Subroutine From C

For this part of the lab we implemented part 2.1 however using assembly code, which was basically a simple translation from C to assembly. The difference was that the code used in assembly was far less that the code in C which means that it was far more efficient and took less time to compute.

Improvements:

We could have implemented the max value function in many different ways which could have been more efficient in terms of time or memory depending on the system requirements.