**ECSE 324 – Computer Organization (Fall 2018)**

**Lab 2 – Stacks, Subroutines and C**

**Group 12: Alain Daccache-260714615 ; Michel Abdel Nour-260725050**

In this lab, we worked with the DE1-SoC Computer System and delved deeper into how to program subroutines and how to make subroutine calls in the C language. The system’s architecture consists of an ARM Cortex-A9 processor and peripheral components of the FPGA on the DE1-SoC board. We used the Intel FPGA Monitor Program 16.1 as our IDE.

Part 1.1 – Stack:

This subpart consisted in understanding how push and pop pseudo-instructions work by using other instructions.

* To push an element in the stack, we first decrement the stack pointer so that it points to a new top of stack (SUB SP, SP, #4), leaving space to subsequently store the value of the element in the word referenced by the pointer (STR R0, [SP]).
* To pop an element from the stack, we first load the contents of the top of the stack (LDR R0, [SP]), then increment the stack pointer (ADD SP, SP, #4) to point to a new top of

In our program, we first pushed three elements contained in R0, R1, and R2 (0, 1, 2 successively) in the stack, and then popped them back off the stack. We can observe the contents of R0 and R2 swapped, while R1 stayed the same.

**Improvements:** We could have implemented a stack ourselves, instead of using the stack pointer SP. Basically what we could do is clear an area of contiguous words in memory and have a pointer point to the first element. To push, we store the value in the address pointed to by the pointer, then increment the pointer by a word. Idem for pop.

Part 1.2 – Calling Convention:

To follow convention, we respected the following rules:

* We are passing the list of numbers and number of elements as arguments in R0 and R1 (no need to push arguments in the stack since no. of arguments < 4)
* When in the subroutine, the first instruction pushes the registers that will be modified (in the subroutine) in the stack, then popped back to the same registers after the subroutine finishes executing.
* After the subroutine, we pop the previously pushed registers back to their initial state before the call of the subroutine.

The logic inside the subroutine itself is the same as the max function implemented in Lab1. R1 that holds the value N of the number of elements to loop through before defining the maximum. R0 acts as a pointer to the list of numbers from memory that allows to loop through all the elements in the list, and the contents of which are store in R3, and then compared with the previous element stored in R2. If contents of R3 are bigger than contents of R2, we update the maximum element in R2. If not, we branch back to LOOP (core of FIND\_MAX subroutine).

**Challenges:** Since the registers should be pushed onto the stack at the beginning of the subroutine, we found it hard to loop through the subroutine without performing the push operation at every iteration. A simple solution required the introduction of a **LOOP** branch into the subroutine. Also, since we need to pop the contents of the stack at the end of the subroutine, this would reset the value of R2, which

Part 1.3 – Fibonacci:

This subpart of the lab consisted of writing a subroutine “FIBONACCI” that would calculate the *n*th Fibonacci number. After writing it in the editor, we saved it in a file ‘fib\_rec.s’ and added it as a source file in a new project ‘G12\_Lab2’. The approach taken to solve this part of the lab is the following:

A register R1 hold the number N of the *n*th Fibonacci number. Before calling the Fibonacci subroutine, we push the contents of R4 and the Link Register onto the stack because we will manipulate R6’s contents. After the function call, we restore the contents of R4.

In order to account for the base case where N has a minimum value 2, we branch to Base that sets the return value to 1. The result is always stored in R0 at the end, which sum is initialized in the Base branch. R1 allows to keep track of the base case, and get the values of fib(n-2) and fib(n-1) to be stored in R4 and R0 respectively and added up at the end into R0 to give fib(n).

Pushing and popping the LR allows to keep track of the stage of the recursion and keep comparing the value in R1 with 2 to ensure the base case is still verified.

**Challenges:**This part of the assignment was quite similar to the recursive factorial example we have done in the tutorial. The main difference is that the we had to do recursion twice (fib(n-1) and fib(n-2)) rather than once (fact(n-1)) so knowing at which section we had to start accumulating the sum was challenging.

**Improvements:** While recursion is usually elegant, it is not very intuitive to come up with an assembly equivalent to a high-level language recursive function. Therefore, an iterative approach is preferable in such cases.

Part 2 – Pure C:

This subpart consists of the C code in getting\_started.c in which we follow a very simple logic of iterating through an array of numbers to find the maximum element. In the C code, we initialize an array a in memory, and using a for loop, we iterate through elements of the array calling the MAX\_2 subroutine implemented in subroutine.s to compare elements of the array 2 by 2 and updating the maximum element max\_val. At the end, we print the maximum value in Terminal and return max\_val to memory.

In this process, we have to make sure that the subroutine MAX\_2 is declared as a global … so that it is visible to other files in the project, and thus able to be called in the C file.

The subroutine MAX\_2 just compares 2 values in R0 and R1, then updates the new max in R0 by moving the contents of R1 to R0 if they are larger in value than those of R0, or exits the branch and links back to the Link Register.

**Challenges:**This part of the assignment was quite simple and straightforward. The only challenge would be the way we integrated the subroutine MAX\_2 in the C code as a method comparing two elements of the array.

**There are not really any improvements that could be done to this part of the lab since the C code is very basic and MAX\_2 subroutine has been implemented and in a clear and simple way.**