27/10/2018

ECSE 324 Lab 2 Group 47:

Nihal Pakis- 260733837
Abdallah Basha- 260719132
Fouad El Bitar- 260719196

---

1. Subroutines

1.1 The Stack

- ***Description:***
    - For this section of the assignment we were asked to use the stack data structure for a situation where there are not enough registers for the program to store data of the subroutines (usually after four arguments). For this section we had to implement the “push” and “pop” instructions to access the stack. We pushed the different values of the same register into the stack, then we popped the values. We implemented the push and pop functions in three different ways, using STMIA, STM, STR for push and LDMDB, LDR, and LDM for the pop. Before each ‘push’ R0, we increment it. Hence, when we are pop R0, we decrement its value using only the stack.
- ***Problems and Improvements:***
    - For this subsection we did not come across any problems.
    - We could have tried another version of stack. Instead of using STMIA and LDMBD, we could have used STMBD and LDMIA, which would structure the stack differently. This would cause the the memory location to decrease each time we push instead of increasing it, therefore extending the stack in the opposite direction. Whether this is an ‘improvement’ depends on the application of the code.
    - There could have also been a 4th representation of push and pop. We could have incremented or decremented the SP first and then Load or Store the LR on stack.

1.2 The Subroutine Calling Convention

- ***Description:***
    - For this subsection we were asked to display our knowledge of the 'Caller-Callee' convention in which the program is responsible for *saving and restoring* the value in a register across a call. For this case we were asked to "move arguments into R0-R3" using the stack when needed as explained in section 1.1. The Subroutine call is done using BL as per usual. In terms of the callee we were asked to "move the return value into R0" whilst "insuring the state of the processor is restored to what it was before the call" using BX LR to return to calling code.
    - We turned the finding max program completed in Lab 1 for finding the max of an array into one that uses the aforementioned 'Caller-Callee' convention for subroutines.
    - 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.
- ***Problems and Improvements :***
    - For this subsection we did not run into any issues as we were simply putting together functioning code we already wrote.
    - We could have added more loops, using the link register and stack to branch between these loops. Although this uses more processing power, it results in more 'modulated', separate code which could be useful for later usage of the code and to understand its functions more easily if another user is trying to understand it.
    - We could have also stored the current max value in the stack so we could access it at any point of the code before it finishes running, which could be useful for debugging larger projects which use this code.

1.3 Fibonacci Calculation Using Recursive Subroutine Calls

- ***Description:***
    - For this subsection we were asked to compute the *n*th Fibonacci number using a recursive subroutine which calls itself where F_0=1, F_2=2, F_3=3, etc.
    - The Fibonacci sequence is found by dividing a number in the series by the number that follows it. However, when practically implemented recursively, this is computed differently by having the

bottom of each branch as either ‘1’ or ‘0’ and adding these values until the Fibonacci value is found. This is shown in the lab descriptions given pseudocode.
    - The aforementioned ‘Stack’ and ‘Caller-Callee’ methods were implemented in order to compute the recursive functions subroutines while insuring the values of the registers across the calls are saved and restored.
    - R0 is used to store F(n), R1 is used to store F(n-1) and R2 is used to store F(n-2).
    - We know that F(0) and F(1) returns 1, therefore we wrote a loop t return 1. Before we Save R0 as 1, we either push R0 to stack or move it to other register.
- ***Problems and Improvements:***
    - It took us a lot of time to figure this one out.
    - After the compilation and running R0 returns us the fibonacci number. We could have stored it in memory after it is calculated instead incase R0 is used for something else as a part of a larger program.
    - We can implement the ‘memoization’ technique, by storing some of the fibonacci numbers. For example if we want to calculate F(8), and if we already stored F(7), then we only need to calculate F(6) and then we can add F(7) and F(6) to obtain F(8). Calculating F(6) is much more faster then calculating F(8). Although this is a faster method, it takes up more memory.

2. C Programming

2.1 Pure C

- ***Descriptions:***
    - For this subsection of the Lab we were asked to make a program in C *only* that finds the largest value of the given array. We did this by looping through the array and manually storing then comparing each element one by one.
- ***Problems and Improvements:***
    - This task was quite simple and didn't have any problems.
    - 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. For

example we could have used adjusted versions or parts of different sort methods such as insertion, heap, merge, selection, quicksort, etc. Some are faster in terms of runtime, others are more conservative in terms of memory usage.

2.2 Calling Assembly Subroutine From C

- ***Description:***
    - For this lab we did the same thing as part 2.1, however we called the aforementioned find max assembly code, completed and discussed in earlier tasks, to find the max instead of comparing each element one by one in a loop. The amount of assembly code used with this method is far less than that produced in section 2.1 only using ‘Pure C’.
- ***Problems and Improvements:***
    - Again, all the code was either done before or given, there were no issues.
    - The same improvements as the ones in section 1.2 could be implemented here as the code used in assembly is the same, it is just called from the C program.