ECSE 324 – Lab Report #2 – Group 34

Jia Wei Sun, 260866206

Karl Michel Koerich, 260870321

27 October 2019

Overview

For this lab, we worked with the DE1-SoC Computer System and learned how to use subroutines and stack. Also, we used code written in C to call code written in assembly. After reading the instructions carefully and assessing the material covered in class, we managed to implement all the desired functionalities of lab 2.

Firstly, we followed the steps from section #1: Subroutines. By the end, we had generated and ran a simple code that pushes and pops values from the stack, a code that finds the minimum number from a list of “numbers” with length “3” using subroutines, and a code that computes the factorial of a value using recursive subroutine calls. In the next section, we were introduced to C programming with assembly. We adapted the code given to find the maximum value from a list. Then, for the last task, we rewrote the previous code to make it use subroutines to find the maximum value from a list.

Overall, we still faced some difficulties regarding the assembly language and the board itself, but we managed to perform all the proposed challenges and we were satisfied with the final results.

Approaches

1.1

After reading the requirements of part 1.1 (Stack), we decided to push and then pop all the numbers one by one. First of all, we load the first number at the address R4 and then load its value to R0. We then performed a push operation with R0, and then make R4 point to the next number. We push R0 2 more times until all numbers have

been pushed, then we performed pop R0 operation 3 times and load the popped value into R1, R2, R3 respectively for each value popped.

1.2

In part 1.2, we used stack operations to determine the minimum value in a list of 3 numbers. First of all, we set R2 to the number of elements in the list, R0 to the minimum value of the list, which we set to the value of the 1st number of the list at first. We then push R1, R2, R3 to the stack and branch link to our minimum-value-determining loop (MIN). The next instruction POP {R1-R3} is stored in the link register. In MIN, we decrement the counter (R2) at the start of each iteration and branch to the loop FINAL when the counter equals 0. At each iteration that the counter is not 0, we point R3 to the next number and load its value into R4; we compare R4 to R0 (minimum value), and we branch back to MIN if R4 is bigger than the smallest value of the list, or we update R0 with R4 if R4 is smaller than the minimum value. At the end of MIN loop, we branch to FINAL and we branch exchange to the instruction stored in the link register, which pops R1, R2, R3 and R4 from the stack.

1.3

In part 2.1, we wrote a program that calculates the Fibonacci sum of a positive natural number. We first set R1 pointing to the address of NUMBER, which is the number we want to find the Fibonacci sum of. We load the value of R1 into R0, and we branch link to the FUNC_FAC loop. In the loop, we push R1 and its next instruction into the stack. We move R0 to R1, and we compare R0 to 0; if R0 is not equal to 0, we branch into IF_NOT_0 loop. At the start of IF_NOT_0 loop, we decrement the counter by 1, and we branch link to the previous loop. This set of operations is repeated until R0 is equal to 0, then we will move the value 1 to R0 because we use R0 to calculate the Fibonacci sum and we don't want to multiply a 0 to the rest of the numbers. We then branch to final, where we pop every element in the stack and each time branching to the instruction stored in the link register of the popped element. For every pop operation

except that for the last element in the stack, we multiply the R0 with R1 and we store the partial product into R0. At each iteration, we multiply the new R0 to the previous pushed version of R1 and we store the partial product into R0 again, until the stack is empty.

2.1

In part 2.1, we wrote a C program that finds the minimum value of a list of numbers. In the main function, we set up an array of integers. We initiate *min_val* and *i*, both integers, and we set *min_val* to the first number of the list. We then wrote a for loop to iterate through all elements of the list, and update new integer *b* with the next iterated number. Then we compare it with the *min_val*, and we update *min_val* with the current iterated number if that number is smaller than *min_val*.

2.2

In part 2.2, we use a combination of a C program and an assembly program to find the maximum value of a list of integers. First of all, we have 2 files; one .c file, and one .s file. In the C program, the approach is very similar to part 2.1; the only difference is in the for loop, we have integer *max* instead of *min* in the previous program. We set the integer *max* to the result of the function MAX_2(*max, b*), which refers to the function of the same name in assembly language, and then we update the new integer *b* with the next iterated number. The MAX_2 function takes in the value of *max* and *b* and compares them. We branch exchange to the address stored in the link register if R0 is greater than R1, then we move R1 to R0. We finish with another branch exchange instruction to the address of the instruction stored in the link register, without condition. At the end of the C program, we return the *max.*

Challenges

The main challenge we had was while programming the factorial problem with subroutine and stacks. At first, the idea of making the program recursive was not clear.

To understand the logic, we first sketched in pseudocode the main of the recursive factorial. We understood that somehow the previous states would have to be stored and that the call for factorial happened in a sort of cascade shape. When it reaches the end case, the go on the opposite side of the cascade until the original state in which the factorial was first called. After understanding the order in which the operations happen, it became clear that the previous states would have to be pushed to the stack until the final condition was not met, and then the states would have to be popped to return to the original position and have the factorial computed.

Other challenges encountered during this lab include understanding certain commands, like BX LR, that were new to us. After asking for help from the TAs and consulting online resources, we understood the commands and we were able to use them correctly in the code.

Conclusion

The main objective of this lab was to introduce us to stacks, subroutines, and C programming combined with assembly language. The programs implemented include: a simple code that pushes and pops values from the stack, a code that finds the minimum number from a list using subroutines, a code that computes the factorial of a value using recursive subroutine calls, a C program that finds the maximum value from a list, and a code that uses subroutines to find the maximum value from a list. his lab required more from us in terms of hours, logic testing and research. After reading the guidelines, consulting TAs and online sources, we were able to implement all proposed problems and we were satisfied with the results. For future labs, we will try to identify edge cases and run more tests for them before to make the code more robust and avoid losing marks while demoing.