ECSE 324 – Computer Organization
Lab 2 – Stacks, Subroutines, and C

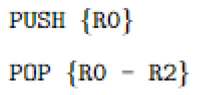
Ryan Servera 260741736
Gabrielle Doucette-Poirier 260738518

Part 1: Subroutines

The stack:

Brief description

The program consisted of rewriting assembly functions PUSH and POP in terms of other assembly functions.



```
PUSH {R0}

POP {R0 - R2}
```

Approach taken

The program would rewrite PUSH and POP by manipulating the stack pointer and then writing into the appropriate address in the stack. Seeing as how the stack in ARM is in full descending, if POP was called the stack pointer would be in a larger address than it was previously in and if PUSH was called the stack pointer would be in a smaller address than originally. As for the test, it would take the first value of an array, PUSH it onto the stack and then POP it out of the stack, ensuring that both the argument registers and the current stack pointer location contained the appropriate contents.

Challenges

One challenge that we faced was coming up with a test that could check the functionality of the two methods, this was solved by inserting a value that was predetermined in the array and then checking if after a push or a pop the registers had the correct values in either stack pointer or the registers

Improvements

A more thorough test could have been written, one that incorporated multiple PUSHs and POPs in a row, with varying orders.

The subroutine calling convention:

Brief description

This program would find the maximum value of an array, but this would be done by calling a subroutine.

Approach taken

The program would start of the same way as originally, it would set registers to contain the address of the array, the size of the array and then address of the first element in the array. However, unlike in the original code, it would now perform iterative comparison of array element within a subroutine, and then once it was completed it would go back to value on the link register.

Challenges

One challenge that we faced was understanding the use of the pushing and popping elements onto the stack and understanding the use of a link register. Without these two concepts it was difficult to determine which parts of the code could be made into a subroutine.

Improvements

One possible improvement would be to use push and pop in the subroutines in order to use those argument registers within the subroutine instead of using more registers to serve as buffers.

Fibonacci calculation using recursive subroutine calls:

Brief description

The program consisted would compute the Fibonacci number, given an input.

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

Approach taken

Firstly, in this program, before calling the FIB subroutine, the current link register must be pushed onto the stack in case that information was useful (the current place in the program was called from a higher order routine). The argument is also pushed through the stack; this slot of memory is also used to push the return value which is the answer of the subroutine. The top elements on the stack after the execution of the routine is therefore the answer. Lastly, the previous link register can be popped back from the stack. Inside the subroutine, the registers R0-R3 are first pushed on the stack to preserve the state of the caller. The argument is then loaded in one of the registers. A compare instruction is run on the argument to see if it is a “base case.” That is, if the argument is indeed less or equal to two, the program jumps to the base case tag and moves the number 1 in R3 which is defined as the register where the answer is stored and jumps to the return tag. If the input was not a base case (greater than two), the program subtracts one from the register holding the argument and recursively calls the subroutine using the same calling convention by pushing the link register and the new argument before branching and linking. This is performed again after subtracting one from the argument once again. The answers for fib(n-1) and fib(n-2) are popped into R1 and R2 respectively. The sum of the contents of these two registers is then written in the return register R3 before going to the return tag. The instructions after the return tag is to overwrite the slot of memory in the stack where the argument was passed in by the value held by R3 which is the return register. Before branching back to the link register, the registers R0-R3 are restored to the values they were found before the subroutine being called by popping the 4 top values of the stack.

Challenges

One challenge was organizing all the code such that all the returns to the link registers would act in a way such that it follows the workflow that you intended. Without understanding the returns lead to a lot of unexpected branches. Another challenge was knowing what elements to push and knowing when to pop them.

Improvements

The code could have been more efficient by using less branches. It could also be improved by utilizing less PUSHs and POPs, which would improve organization.

Part 2: C Programming

Pure C:

Brief description

This program takes an array and finds the max value in that array.

Approach taken

This program would initialize the array with all the element then it would use a for loop to iterate through it. During each iteration, the max value variable would be compared to the current index element of the array. If the max value variable, was smaller then the element in the array, max value would be set to that element.

Challenges

One challenge that we faced was determining the size of the array, without this value we didn't have a counter to keep track of which index of the array we were currently at.

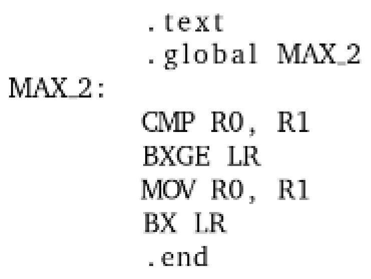
Improvements

There is a redundant if else statement that is in the for loop, which is meant to assign the max value to the value of the first element in the array at the first iteration of the for loop. This can be simplified by initializing the max value to the first element in the array before the for loop.

Calling an assembly subroutine from C:

Brief description

This is almost the same as the C code from the previous section, but instead of comparing values in C, an assembly subroutine would perform the comparison and then return the larger value.



```
        .text
        .global MAX_2
MAX_2:
        CMP R0, R1
        BXGE LR
        MOV R0, R1
        BX LR
        .end
```

Approach taken

This program at first sets the first element of the array as the max value. Next, it goes into a for loop. Within that loop the instead of comparing the current indexed element and the max value, it now makes a call to a MAX_2 subroutine. Once called that subroutine, compares the two inputted numbers and returns the larger one. Finally this is repeat for the entire array.

Challenges

The main challenged we faced in this part was understanding that R0 is used to pass from an assembly subroutine, which could then be used in the C program. This was necessary to comprehend, because without it we didn't know what was going to be returned after then assembly subroutine.

Improvements

One possible improvement would be to dynamically find the number of shifts needed to be performed based on the length of the array. Another improvement would be to use an actual division algorithm to calculate the average of collections which are not solely powers of two.