ECSE 324 Lab Report 2

Marie-Lynn Mansour - 260770547
Haluk Calin - 260790895

Part 1: Subroutines

Part 1.1: Implementing a Stack

The first part of this lab was to implement a stack in assembly language. The goal was to push the numbers 2, 3, and 4 onto a stack using the R0 register then pop these values from the stack into the R1, R2, and R3 registers. We implemented this by using the Stack Pointer (R13), which points to the top of the stack.

The loop (LOOP) starts by decrementing the loop counter, branching to DONE and ending the loop if the counter is less than 0. A register R0 is then loaded with the contents of R3, which is the first number in the list. R3 is then incremented by adding 4 to it, meaning it now points to the next item in the list. The stack pointer (SP) is updated by incrementing it by 4 and the new top element is set by storing the contents of SP in R0. We then branch back to the loop and restart until all the elements of the list have been pushed.

```
LOOP:       SUBS R2, R2, #1
            BLT DONE
            LDR R0, [R3]
            ADD R3, R3, #4
            SUBS SP, SP, #4
            STR R0, [SP]
            B LOOP
```

figure 1: loop to push the numbers 2,3, and 4 onto the stack

When LOOP has terminated, it goes to DONE. The top element of the stack (SP) is popped into R1 by loading the contents of SP into R1. The SP is updated by incrementing it by 4. The new top element is then popped into R2 by loading the contents of SP into R2. The SP is updated again by incrementing it by 4 and the new top element is popped into R3 by loading the contents of SP into R3. SP is updated once more by incrementing it by 4, returning it to its original state. The final result is the value 4 in register R1, the value 3 in register R2, and the value 2 in register R3, as stacks work as “last in, first out”.

```
DONE:       LDR R1, [SP]
            ADD SP, SP, #4
            LDR R2, [SP]
            ADD SP, SP, #4
            LDR R3, [SP]
            ADD SP, SP, #4
```

figure 2: popping the elements of the stack into registers R1, R2, and R3

A possible improvement would have been to wrap our implementation in a function. We would simply call the function with a list of registers to push or pop. The function would take care of branching back to the caller once it’s done. This would make the pushing and popping of multiple registers much simpler.

Part 1.2: Subroutine Call to Find the Minimum Value of an Array

The second part of this lab was to find the minimum value of an array of three numbers, modifying the algorithm implemented in Lab 1, implementing it as a subroutine. In a nutshell, we moved the loop that found the minimum outside of the main function and then called it from the main function. The caller places the arguments into arrays R0-R3. The subroutine was then called using BL. The callee moves the return value, which is the minimum value of the array, into R0. Since the subroutine was only called once, we were able to overwrite the values stored in the registers and therefore not use the push and pop commands. The caller then restores the state of the processor then uses BX and LR to return to the calling code. The snippet of code shown in *figure 3* illustrates this:

```
SUBROUTINE:     PUSH {R0-R3}
                LDR R1, [SP, #20]
                LDR R2, [SP, #16]
                LDR R0, [R2]

LOOP:           SUBS R1, R1,#1
                BEQ end
                ADD R2, R2, #4
                LDR R3, [R2]
                CMP R0, R3
                BLE LOOP
                MOV R0, R3
                B LOOP

end:            STR R0, [SP, #20]
                BX LR
```

figure 3: subroutine and loop to find the minimum value of the array

A possible improvement would have been to streamline the function call - instead of passing three arguments, we could have simply passed the address of where the array is stored in memory as an argument. Everything else would have been handled in the function and we would still have the minimum value stored in R0 once the function returns.

Part 1.3: Recursive Subroutine to Compute the Factorial of an Integer

The third part of this lab was to use a recursive subroutine to compute the factorial of an integer, which is 3 in our case. Register R0 points to the result. Register R1 holds N, which is 3. The value of the accumulator is held in register R2. BL FACTORIAL calls the subroutine FACTORIAL by branching to it.

In the subroutine itself, registers R0, R1, and R2 are pushed onto the stack. The values R1 and R2, which hold the values 3 and 2, respectively, are multiplied together and placed in R2. The counter is decremented, meaning R1 now holds the value 2. We then branch back to FACTORIAL if we do not branch to EXIT (BEQ) and restart the recursive subroutine. The number 3 is continuously decremented and multiplied together until we branch to EXIT. The result R2 is then stored in R0, which is 6 and we branch to the instruction in the Link Register (LR) - it goes to BL FACTORIAL then branches to END.

```
_start:
            LDR R0, =RESULT
            LDR R1,[R0,#4]
            MOV R2, #1
            BL FACTORIAL
            B END


FACTORIAL:
            PUSH {R0-R2}
            MUL R2,R2,R1
            SUBS R1,R1,#1
            BEQ EXIT
            B FACTORIAL

EXIT:
            LDR R0, =RESULT
            MOV R0, R2
            BX LR
```

figure 4: recursive subroutine FACTORIAL that computes the factorial of an integer

Part 2: C Programming

Part 2.1: Getting Started C Program - Finding the Minimum of an Array Using C

This part of the lab was a "fill in the blank" of a given C program to find the minimum of an array. Our implementation was to simply add a for loop that iterates through each element in the array, comparing every element with the current minimum value. If the current minimum value was larger than the current element, the current element became the new minimum value. The for loop we implemented is shown in *figure 4*. A possible improvement would have been to check if the end of an array character is present. This extra check would eliminate the possibility of iteratively over elements after the array, preventing wrong calculations of the minimum value.

```
for(k=1; k<5; k++) {
    if(min_val > a[k]) {
        min_val = a[k];
    }
```

figure 5: for loop that iterates through an array to find the minimum value

Part 2.2: Calling an Assembly Subroutine from C to Find the Minimum Value of a List

The final part of the lab was to mix assembly language and C. We had to rewrite a given C program in order to find the minimum value of an array using the given MIN_2 subroutine. We implemented a for loop that finds the minimum value by calling the MIN_2 subroutine, passing the arguments min_val, which is the current minimum value, and a[i], the current element i in the array.

```
    int i;
    for (i=1; i<5; i++) {
        min_val = MIN_2(min_val, a[i]);
    }

    return min_val;
}
```

figure 6: finding the minimum value by calling the subroutine MIN_2

Similar to Part 2.1, a possible improvement would have been to check if the end of an array character is present. This extra check would eliminate the possibility of iteratively over elements after the array, preventing wrong calculations of the minimum value.