ECSE 324

LAB 1 REPORT

Martin Kruchinski Almeida

260915767

Part 1:

Exercise 1-

Introduction:

The first problem was to create an iterative algorithm that calculates the inputted Fibonacci number the user wants to get. This algorithm had 2 special cases and a general case. If the **input = 0**, the **fib(0) = 0** and if the **input = 1** or **2, fib(1) = fib(2) = 0**. For those special cases I did a comparison of **#0** and **#2** with **n** and then branched to the corresponding case.

```
ldr R2, n //load n memory address in r2
mov R6, #2 //Initialize i
cmp R2, #0
beq zero
cmp R2, #2
ble one
```

Fig 1: Special cases

```
LOOP:
cmp R6, R2 //compare the values of n and i
bgt store // if i is greater than n then branch to end
add R5, R1, R0 // f(i) = f(i-1) + f(i-2)
add R6, R6, #1 //increase the value of i
mov R0, R1  //change the value of f(i-2)
mov R1, R5 ////change the value of f(i-1)
mov R7, #1
ble LOOP //if less or equal branch to loop again
```

Fig 2: General cases

```
store:
mov R0, r5 //store the final value in r0
b end

zero: //case 0
mov R0, #0
b end

one: mov R0, #1 //case n=2 or n=1
b end
end: b end
```

Fig 3: Dealing with special cases

For the rest of the cases I created a for loop that compares **i** with **n**. If **i** is greater than **n** the program stores the result in R0 and branches to an infinite loop, and if it is less or equal it computes **f(i) = f(i-1) + f(i-2)** , updates the values and branches to the loop again at the end. I implemented subroutines by using for loops and branches between different labels, to make the program iterative.

Challenges and improvement:

I had no challenges for this part, but I would like to improve my program by making the code shorter and simpler.

Exercise 2-

Introduction

The second problem was to create a recursive version of the Fibonacci algorithm. As the last one, this algorithm had 2 special cases and a general case. If the **input <= 2**, the algorithm would branch to a label that would take care of the special cases. Because it is a recursive algorithm, these labels were also reused by the normal cases.

```
fib:
    push {lr} //push address of next instruction
    cmp R0, #2 //if it is less than, then the result is 2 or 1
    BLE twoorless //branch if n is two or less (cause is either 0 or 1)
```

Fig 5: labels for special cases

```
twoorless:
    cmp R0, #0 //check if n=0
    BEQ iszero //branch if n=0
    mov R0, #1 //if n is not 0 then fib(n)=1
    pop {lr} //pop value of lr to go backwards after we finished recursion
    bx lr //branch to lr's address

iszero:
    mov R0, #0 //fib(0)=0
    pop {lr} //pop value of lr to go backwards after we finished recursion
    bx lr //branch to lr's address
```

Fig 4: checking for special cases

```
twoorless:
    cmp R0, #0 //check if n=0
    BEQ iszero //branch if n=0
    mov R0, #1 //if n is not 0 then fib(n)=1
    pop {lr} //pop value of lr to go backwards after we finished recursion
    bx lr //branch to lr's address

iszero:
    mov R0, #0 //fib(0)=0
    pop {lr} //pop value of lr to go backwards after we finished recursion
    bx lr //branch to lr's address
```

Figure 6: Dealing with special cases and using stack calls

The body of the program:

We took a subroutine and stack approach for this program. In fact, we used instructions such as **push {lr}** (that pushes the address of the next instruction into the stack), as well as **pop {register}** (that pops the value that is on top of the stack) and **bl** (that copies the address of the next instruction into the link register and **bx** (that branches to the lr address). The reason I chose this approach was because when I did subroutines, I wanted the program to remember when I was before doing a recursive call, that is why I push the next instruction into lr. When I am done with the recursion, and I computed all the Fibonacci values, I pop those addresses that were piled in the stack to go backwards and add every Fibonacci number to compute the one we want to store in R0.

```
bl fib //recursive call
pop {R3}    //pop fib(n-2)
push {R0}   //save the value that was returned
mov R0, R3  //moving the value of fib(n-2) into R0 so we can do recursion
BL fib  //recursive call
mov R3,R0   //move the returned value into R3, because R0 is going to store the sum of R2+R1
pop {R2}    //Pop the value that was previously returned from the first recursion into R2
add R0, R2, R3 // f(n) = f(n-1) + f(n-2)
pop {lr}
bx lr
```

Fig 7: stack calls and subroutines instructions

Challenges and improvement:

At first, I had a hard time understanding how subroutines and stack calls were related, but later I was able to understand it. The code could be more readable.

Part 2:

Introduction:

In part 2 of the laboratory, I worked on a 2d convolution algorithm, which is usually used in image processing applications. The convolution algorithm is used to implement image filters and to detect objects using machine learning.

The program takes two main inputs: a 2d array **fx [10][10]** and a 2d array **kx [5][5]** called the kernel and returns a 2d array **gx[x][x]** in memory as the output. Each value in the 2d arrays is a word. For the output **gx [5][5]** a space of 40 bytes (filled with 0's) was allocated in memory and meant to be filled once the program is done computing the algorithm. The program has also some variables like **iw, ih** (representing the width and height of the image), **kw, kh** (representing the width and height of the kernel) and **ksw, khw** (representing the Kernel width and height stride). All these variables were initialized as words.

```
//initialize fx
fx_input: .word 183, 207, 128, 30, 109, 0, 14, 52, 15, 210
          .word 228, 76, 48, 82, 179, 194, 22, 168, 58, 116
          .word 228, 217, 180, 181, 243, 65, 24, 127, 216, 118
          .word 64, 210, 138, 104, 80, 137, 212, 196, 150, 139
          .word 155, 154, 36, 254, 218, 65, 3, 11, 91, 95
          .word 219, 10, 45, 193, 204, 196, 25, 177, 188, 170
          .word 189, 241, 102, 237, 251, 223, 10, 24, 171, 71
          .word 0, 4, 81, 158, 59, 232, 155, 217, 181, 19
          .word 25, 12, 80, 244, 227, 101, 250, 103, 68, 46
          .word 136, 152, 144, 2, 97, 250, 47, 58, 214, 51

//initialize kernel
kernel_input: .word 1,1,0,-1,-1
              .word 0,1,0,-1,0
              .word 0,0,1,0,0
              .word 0,-1,0,1,0
              .word -1,-1,0,1,1

//initialize gx
output_gx: .space 40
           .space 40
           .space 40
           .space 40
           .space 40
           .space 40
           .space 40
           .space 40
           .space 40
           .space 40

//initialize variables
int_iw: .word 10
int_ih: .word 10
int_kw: .word 5
int_kh: .word 5
int_ksw: .word 2
int_khw: .word 2
```

Fig 8: initialization of inputs an outputs

The body of the program:

The body of the program consists of 4 for-loops where we compare the initial variables **iw, ih, kw and kh** to different for loop variables x, y, i, and j (also of size word). Also, the values that were going to be stored in each index of the output array, must be first stored in **sum**, a variable that would compute **sum = sum + kx[j][i] * fx [x+j -ksw] [y+I -khw].** Before computing the sum, a series of conditions would need to be met for the sum to be finally computed. First **y < ih, x < iw, I < kw, and j < kh**, then **0 <=** x+j **-ksw <= 9, and 0 <=** y+I **-khw <= 9.** The reason for those values is because fx is a 10*10 array and the indices **x+j -ksw and** y+I **-khw** need to be able to be in bounds with the array to fetch the correct value. The same reason is valid for j and i that both need to be between 0 and 4 because they are the indices of kx and 5 * 5 array. When the conditions were not met, I implemented subroutines to increase the value of the variables and branch back to the previous loop to restart the iteration.

```
for_loop1:
    ldr R1, int_ih //ih =10
    cmp R0, R1 // y < ih
    bge end //branch to end, end of the program
    mov R2, #0 // x = 0
for_loop2:
    ldr R1, int_iw //iw = 10
    cmp R2, R1 // x < iw
    bge increase_y //y++
    mov R3, #0 // i = 0
    mov R10, #0 //sum = 0
```

Figure 9: For loops

```
for_loop4:
    ldr R1, int_kh //kh = 5
    cmp R4, R1 // j < kh
    bge increase_i // i++
    ldr R1, int_ksw //ksw = 2
    add R5, R2, R4 // temp1 = x+ j
    sub R5, R5, R1 // temp1 = x+j -ksw
    ldr R1, int_khw //khw = 2
    add R6, R0, R3 // temp2 = y+i
    sub R6, R6, R1 // temp2 = y+i -khw

    //if statement
    cmp R5, #0 // check temp1>=0
    blt increase_j
    cmp R5, #9 //check temp1<=9
    bgt increase_j
    cmp r6, #0 // check temp2>=0
    blt increase_j
    cmp R6, #9 //check temp2<=9
    bgt increase_j
```

Figure10: check conditions

```
ldr R9, =kernel_input //load first address of kernel
mov R1, #5
mla R11, R4, R1 , R3 //multiply j by number of rows and add
mov R1, #4
mul R11, R11, R1 //total offset
add R9, R9, R11 //address of element we want to get
ldr r8, [R9] //value of element we want to get (override R2
ldr R12, =fx_input
mov R1, #10
mla R11, R5, R1 , R6 //multiply temp1 by number of rows and
mov R1, #4
mul R11, R11, R1 //total offest
add R12, R12, R11 //address of element we want to get
ldr R7, [R12] //value of fx[temp1][temp2]
mul R8, R8, R7 // kx[j][i] * fx[temp1][temp2]
add R10, R10, R8 // sum = sum + kx[j][i] * fx [temp1][temp2]
```

Figure11:computesum

Challenges and improvements:

The challenge I had with this program was understanding how to get the value from a specific index of a 2d array. Later, with some research, I discovered that to get the address of the index **[j][i]** of the array **kx [j] [i]** I had to compute **j * size of array + I** and then multiply that by 4 because each word in the array takes 4 spaces in memory. I would like to make this program simpler because I think it is too long and hard to read sometimes. Also, I think it could be implemented with less computations and no need to reuse registers.

Part 3:

Introduction:

In the third part of my lab, I implemented the bubble sort algorithm, which sorts an input array in ascending and descending order. The user enters as input an array of size “size” and the program returns the same array in ascending or descending order.

The body:

For this algorithm, I implemented subroutines using for loops. The program has two for loops that allows to iterate through and compare the values to be sorted. The program uses pointers to point to the address in memory of the different indices of the array and load their values into registers. The ascending sorter, for example, checks if (*(ptr + i) > *(ptr + i + 1)) to see if the current element in the specified index is bigger than the next element in the list, and would need to be swapped.

```
loop1: //first for loop
ldr R0, =array //allows us to point to first element when starting every new iteration
ldr R3, int_i // i = 0
cmp R2, R4 //  step < size-1
bge end // loop ends if step >= size - 1

loop2: //second for loop
sub R5, R4, R2 //size - 1 - step
cmp R3, R5 // i < size - 1 - step
beq increase_step //increase step if equal
```

Figure 12: implementation of for loops

If the two values need to be swapped, I stored them in **registers R6 and R7**, instead of using a **temporary value** to make the swap. The value that was **in register R7** was stored **index i** and the value that was stored in register **R6** in **index i + 1** (to access the next index, I added 4 to the current address).

Once the values were swapped, I branched to a label to increase the for-loop variable i, incremented the value in the pointer by 4 so that we could start the comparison in the next index and branched back to the loop to continue with the iteration as we can see in figure 14.

```
str R7, [R0] //swap R7 and R6
str R6, [R0, #4] //swap R7 and R6
b increase_i //increase i when loop ends
```

Figure 13: swap of elements

```
increase_i:
add R3, R3, #1 // i = i + 1
add R0, R0, #4 //add 4 to the pointer
b loop2 //branch to second for loop
```

Figure 14: increase i and make pointer point to the next index

Challenges and improvements:

The only challenge I had was to understand how to use the pointers. Pointers in C have always been hard for me, but programming in Assembly, that constantly works with addresses in memories, made me have a better understanding of this concept. I would like to see if I could improve my code by making it simpler, as I use many labels that could be avoided.