Lab 1 Report

Samuel Proulx

Part 1.1

For the iterative approach, a simple loop with no subroutine was enough to execute the program.

The program thus consists of a setup section where some constants are moved into argument registers and the loop section that computes the Fibonacci. The loop first adjusts the argument register A3, then adds the content of A1, which corresponds to the result of the previous iteration of the loop, or *fib(n-1)*, with the content of A2, which corresponds to the result of the second previous operation, or *fib(n-2)*.

```
loop:
    mov a3, a1
    add a1, a1, a2
    mov a2, a3
    subs v1, v1, #1 @ i-- and update CPSR
    bgt loop
```

A mistake I did early on was to not reduce the number of iterations the loop has to do by 1. If the input *n* is not reduced by 1, the program will compute the next Fibonacci in the sequence.

In terms of improvements, I found no clear improvements that could be made to the logic of the program since the algorithm is already as simple as it can get. An improvement could be made to the structure of the program to make it more flexible, or in other words, make it so that it could run with other programs using some registers, would be to make the loop a subroutine. That would most likely be slower since a subroutine needs to *push* and *pop* values from the stack in memory. This is a marginal improvement to the program in terms of structure. It would also not improve the code readability as the simple loop shown above is very short and concise.

Part 1.2

For the recursive approach, a subroutine had to be used. Like part 1.1, I used register A1 (R0) as the one that would contain the answer and used registers A2 for *fib(n-1)* and A3 for *fib(n-2)*. This assures the positions of the arguments is known in every call of the subroutine. The subroutine is responsible for adding the two previous results in A1 then updating the two other arguments in what resembles a queue (A2 is next in line and A3 comes after). The subroutine also needs to keep track of how many iterations it has done and stop when it has done the required amount. This is done by decrementing a counter initialized at the given input. It works similarly to a static variable in a high-level programming language.

```
fib:
    add a1, a2, a3
    mov a3, a2
    mov a2, a1

    subs a4, a4, #1
    bgt fib

    bx lr
```

Planning the subroutine and how to handle results so that we know where previous results are located was challenging.

Part 2

The program initially declares the 2D array *fx* as a very large 1D array. The same goes for *kx*. A space of 400 bytes is allocated to the result array *gx* since its size is 10x10 in a 4-byte word architecture. I translated every loop in a subroutine. Every loop obviously incremented an index analogous to the ones found in the C program but also had to reset the one immediately nested into it. The y, x, and i loops are all very similar with the exception that the *x_loop* initializes the variable *sum* at 0.

```
x_loop:
        @ 0...iw-1
        cmp v2, a2
        @ >= 0 ? go to y_loop
        bxge lr
        @ callee-save convention
        push {v5}
        @ sum = 0
        mov v5, #0

        @ go to i_loop
        push {lr}
        bl i_loop
        pop {lr}

        pop {v5}

        @ reset 1st nested loop
        mov v3, #0
        add v2, v2, #1 @ x++
        b x_loop
```

The j loop is more complex in that it works like the other loops, but also has to compute the convolution. The j loop begins by computing *temp1* and *temp2*. It skips to a point further down the loop if *temp1* or *temp2* are outside the given boundaries. The program then fetches the elements in fx and kx at the corresponding positions.

```
@ callee-save convention
push {v6, v7, v8}
@ temp1 = x+j - kws
add v6, v2, v4
subs v6, v6, #kws
@ temp1 < 0
blt else
cmp v6, #9
@ temp1 > 9
bgt else

@ temp2 = y+i - khs
add v7, v1, v3
subs v7, v7, #khs
@ temp2 < 0
blt else
cmp v6, #9
@ temp2 > 9
bgt else
```

I had difficulties using subroutines in this part. I kept having problems of clobbered registers, which happen when the callee-save convention is not respected. My attempts to solve it where unsuccessful. This is something I could have improved on. Another solution would have been to not use subroutines, since organizing subroutines used quite some lines of code and added complexity to the program.

Part 3

For this part, I used a similar approach to that of part 2. Each loop was implemented using a subroutine. I followed the same structure for the loops as in part 2. To perform the comparison and the swap, I loaded the elements of the array to compare and loaded there addresses. If a swap was necessary, I stored the elements back in the array but I switched the addresses. If no swap was necessary, I skipped the part of the code that performed the swap by branching to a point after the swap.

```
second_loop:
        sub a2, a1, v1
        cmp v2, a2
        bxge lr

        @ v3 = (ptr + i) = #arr + (i*4)
        mov v3, #arr
        mov v4, #4
        mla v3, v2, v4, v3

        @ v4 = (ptr + i + 1) = v3 + 4
        add v4, v3, #4

        ldr v5, [v3]
        ldr v6, [v4]

        cmp v6, v5
        @ v5 <= v6 == ok
        bgt no_swap

        str v5, [v4]
        str v6, [v3]
```

The same problem arose as in part 2 with the clobbered registers. The same solution could be use, that is not use subroutine where it is not required or be more careful to respect the convention. Another minor challenge was the bounds of the inner for-loop since they were changing with the program. It turned out to require a little more thinking but the implementation was similar to that of a regular for-loop. Another obvious improvement would be to not use bubble sort, but that would go against the given requirements.