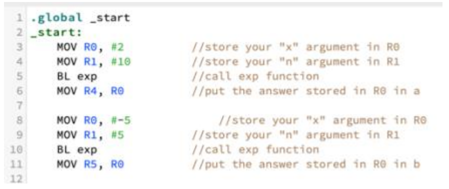
Part 1: Function calls

1. *Exponential function:*

In this first part of the lab, we got familiar with the concept of calling a function in assembly language, but also how recursive functions could be implemented. We started by writing a simple function that calculates x^n (with x and n as the function parameters) that can be called by the program more than once.

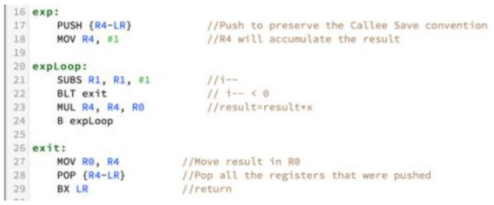
To respect the subroutine calling convention, the main program (which is the caller here) must first move the arguments into the registers R0 through R3. In this case, the function takes for first argument x and for second argument n, so the caller stores "x" in R0 and "n" in R1 before calling the function. Since we are following the lab description, we first call this function with x=2 and n=10. We therefore respectively store 2 and 10 in R0 and R1. Moreover, the main program calls the exp function a second time, with x=-5 and n=5. In both cases, we call the subroutine function using the branch and link instruction BL, which stores the address of the next instruction in the link register LR.



```
.global _start
_start:
    MOV R0, #2          //store your "x" argument in R0
    MOV R1, #10         //store your "n" argument in R1
    BL exp              //call exp function
    MOV R4, R0          //put the answer stored in R0 in a

    MOV R0, #-5             //store your "x" argument in R0
    MOV R1, #5          //store your "n" argument in R1
    BL exp              //call exp function
    MOV R5, R0          //put the answer stored in R0 in b
```

Figure 1: Assembly code of the main program that calls the exp function

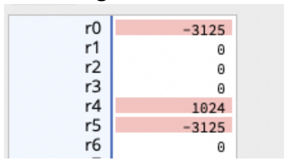
After branching to the exp function, we need to follow the callee save convention by saving the content of the registers used by this function. Indeed, the function might use the registers R4 through LR and modify their original content, which we might need after exiting the function. To preserve the state of the processor, we therefore push R4 through LR onto the stack at the beginning of the exp function. This is done using the PUSH instruction as shown on line 17 in Figure 2.



```
16 exp:
17     PUSH {R4-LR}            //Push to preserve the Callee Save convention
18     MOV R4, #1              //R4 will accumulate the result
19
20 expLoop:
21     SUBS R1, R1, #1         //i--
22     BLT exit                // i-- < 0
23     MUL R4, R4, R0          //result=result*x
24     B expLoop
25
26 exit:
27     MOV R0, R4          //Move result in R0
28     POP {R4-LR}         //Pop all the registers that were pushed
29     BX LR               //return
```

Figure 2: Screenshot showing the implementation of the exp function

We then started to write the function based on the C code provided. We represented the "for loop" in assembly using a label: "expLoop". The loop in C uses an integer variable that starts at 0 and increment its value after each iteration by 1, until it's equal to n-1. Since R1 contains n, we loop by subtracting one to it in each iteration and exit the loop if this result is lower than 0. Doing this avoids using a separate register for the variable 'i'. We use the instruction SUBS to subtract one from n in each iteration of the loop instead of a CMP instruction because we need R1 to store the new value of 'i'. Once the for loop is over, we need to respect the convention of putting the result returned by the function in register R0. In this case, R4 accumulated the result so we put its content into R0. We then restore the state of the processor by pushing all register from R4 to LR and we branch back to the caller using "BX LR". When we go back to the caller, we save the returned value stored in R0 in a different register in case we use R0 for another function call. At the end of the program, R4 stores the result of the first function call and R5 the result of the second one. We then successfully compiled and loaded the program and ran it to get the results shown in Figure 3.

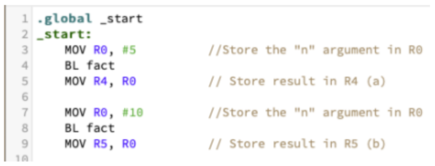


```
r0   -3125
r1       0
r2       0
r3       0
r4    1024
r5   -3125
r6       0
```

Figure 3: Results obtained after running the assembly program part1-exp

2. *Factorial function:*

Writing this assembly program was more challenging since the factorial function uses recursion. We respected the subroutine calling convention by putting the argument of the factorial function in the register R0 at the start of the program before calling it. Our main program calls the fact method twice: once passing 5 as its argument and a second time passing 10 as shown on Figure 4. The function is called by branching to it using the "BL" instruction which stores the address of the next instruction in the LR register.



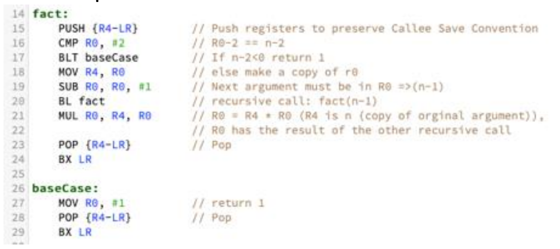
```
.global _start
_start:
    MOV R0, #5          //Store the "n" argument in R0
    BL fact
    MOV R4, R0          // Store result in R4 (a)

    MOV R0, #10         //Store the "n" argument in R0
    BL fact
    MOV R5, R0          // Store result in R5 (b)
```

Figure 4: Screenshot illustrating how the main program calls the fact function twice.

When entering the function, we preserved the state of the processor by pushing all registers from R4 through LR. We then implemented the given C function by first comparing n to 2 using the CMP instruction.

-If n< 2, the function must return 1, so we branch to the baseCase label shown on line 26 of Figure 5. The assembly program follows the convention by moving the returned result in R0, popping registers R4 through LR to restore the state of the processor, and branches to the next instruction (which is stored in LR) by using "BX LR".

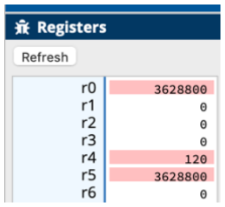
-If n >= 2, the function must return n * fact(n-1). The program calls the fact function with argument "n-1". This argument must be stored in R0 by following the convention we adopted. Since we need the current value of n to multiply it by fact(n-1), we store this value in R4. We then store "n-1" in R0 and branch back to the function. We use the "BL" instruction again to store the next instruction's address in LR, so that the function keeps track of what instruction to execute next when returning from the recursive calls. The function is executed again but R0 now contains n-1. This function might write this "n-1" value in R4 before calling itself back again with "n-2", but the previous value of n we placed in R4 is not lost, it was saved on the stack when we entered fact(n-1). This illustrates how important the stack is in recursive calls.



```
14 fact:
15     PUSH {R4-LR}        // Push registers to preserve Callee Save Convention
16     CMP R0, #2          // R0-2 == n-2
17     BLT baseCase        // If n-2<0 return 1
18     MOV R4, R0          // else make a copy of r0
19     SUB R0, R0, #1      // Next argument must be in R0 =>(n-1)
20     BL fact             // recursive call: fact(n-1)
21     MUL R0, R4, R0      // R0 = R4 * R0 (R4 is n (copy of orginal argument)),
22                         // R0 has the result of the other recursive call
23     POP {R4-LR}         // Pop
24     BX LR
25
26 baseCase:
27     MOV R0, #1          // return 1
28     POP {R4-LR}         // Pop
29     BX LR
```

Figure 5: Assembly program for the factorial function.

The fact function calls itself until an argument n lower than 2 is passed to the function, which will make it branch to the baseCase. The baseCase returns 1 and the program goes back to the instruction stored in LR. The result is always stored in R0 to respect the convention, and the function always pops the registers off the stack before returning to the address stored in LR. This address stored in LR might refer to the instruction at line 21 of a previous fact function call if we are in a recursive call, or to the instruction at a line in the main program if the function is done. Figure 6 shows the result from running the program: "5!" is sorted in R4 and "10!" in R5.



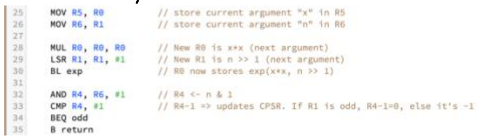
```
r0   3628800
r1         0
r2         0
r3         0
r4       120
r5   3628800
r6         0
```

Figure 6: Results obtained after running the assembly program part1-fact

One of the main challenges in writing that function was to translate the recursion call from the C program to assembly language. This was understood after thinking about the subroutine calling conventions and how to use the stack to preserve values stored in current registers.

3. *Exponential function(recursive):*

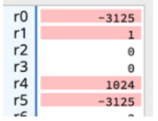
This second implementation of the exp function is very similar to the first one. The main difference was to implement the recursion and the different return statements at the beginning of the function. The exp function respects the subroutine calling convention by pushing R4 through LR at the beginning of the function and by popping them when returning. To facilitate the return statements, we implemented a return label, that makes the program pop R4 through LR to restore the state of the processor and branch back to the instruction stored in LR. At the beginning of the program, we branch to the baseCase label if "n" is 0, which stores 1 in R0 and return, or we branch to return directly if “n” is 1 because we must return x, which is already in R0. If “n” is greater than 1, the function can return two different values depending on if “n” is odd or even. In both cases, the function must call itself again as exp(x*x, n>>1). We therefore store the current values of x and n we might need later in the program in R5 and R6 respectively. Afterwards, we store the next arguments in R0 and R1 before calling the function again using the “BL” instruction. After the function returned from this recursive call, the result is stored in R0 because the program followed the subroutine calling convention. The program then checks if n is odd, if so, it branches to the “odd” label which returns x*exp(x*x, n>>1). Otherwise, the program just returns exp(x*x, n>>1) which is stored in R0 already so it branches to return.



```
    MOV R5, R0              // store current argument "x" in R5
    MOV R6, R1              // store current argument "n" in R6

    MUL R0, R0, R0          // New R0 is x*x (next argument)
    LSR R1, R1, #1          // New R1 is n >> 1 (next argument)
    BL exp                  // R0 now stores exp(x*x, n >> 1)

    AND R4, R6, #1          // R4 <- n & 1
    CMP R4, #1              // R4-1 => updates CPSR. If R1 is odd, R4-1=0, else it's -1
    BEQ odd
    B return
```

Figure 7: Assembly program implementing the recursion call for the recursive exponential function.

After successfully compiling, loading and running the program, we obtained the results shown in figure 8. R4 contains the result of $2^{10}$and R5 the result of -5^5.

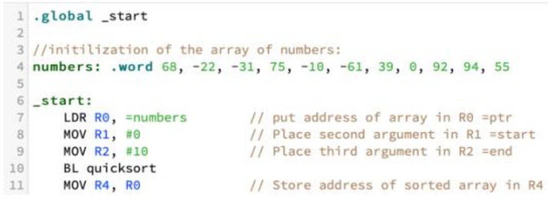


```
r0    -3125
r1        1
r2        0
r3        0
r4     1024
r5    -3125
```

Figure 8 : Results obtained after running the assembly program part1-expbysq

Part 2: Sorting arrays

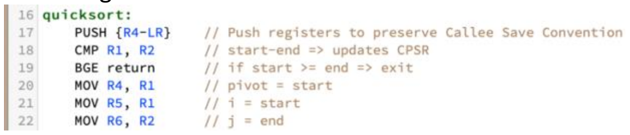
In the second part of the lab, we implemented the quicksort function. We started writing the program by initializing the array of number in memory with the numbers provided in the C code. We then put the arguments for the quicksort function into the registers R0 to R2 to respect the subroutine calling convention. We placed the address in memory of the first element in the array in R0, we then specified the low index in R1, which is 0 and the high index in R2 which is the size of the array-1 (9 in this case). We could have initialized a variable in memory that contains the length of the array and then store it in R2 using a LDR instruction, but we followed the C program that initialized a separate variable for the length. The quicksort function is then called by branching to it using the “BL” instruction which stores the address of the next instruction in the LR register to make sure the program executes the correct instruction after the function returns. When this function returns, R0 will contain the address of the first element in the array, which we store into R4 in case we call another function in the main program later. This first part of the assembly code is shown in Figure 9.



```
.global _start

//initilization of the array of numbers:
numbers: .word 68, -22, -31, 75, -10, -61, 39, 0, 92, 94, 55

_start:
    LDR R0, =numbers        // put address of array in R0 =ptr
    MOV R1, #0              // Place second argument in R1 =start
    MOV R2, #10             // Place third argument in R2 =end
    BL quicksort
    MOV R4, R0              // Store address of sorted array in R4
```

Figure 9: Assembly code of the main program that calls the quicksort function

We then saved the state of the processor when entering the quicksort function by pushing all registers from R4 to LR. We then compared the start and end variables stored in R1 and R2 respectively using the CMP command which updates the CPSR. The instruction shown at line 19 of figure 10 then evaluates the CPSR and branches to the “return” label if start>=end, otherwise the program continues to the next instructions which assigns the values for the pivot, i and j variables. As a design decision, we implemented this “return” label to facilitate the return statements. Its code simply restores the saved state of the processor by popping R4 to LR and branching to the next instruction saved in LR with “BX LR”.

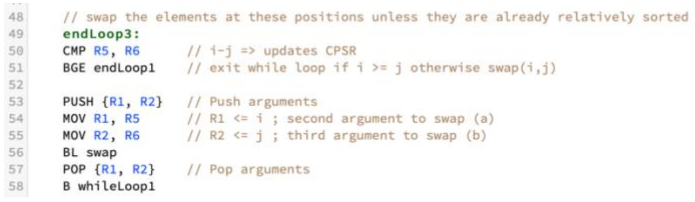


```
16 quicksort:
17     PUSH {R4-LR}     // Push registers to preserve Callee Save Convention
18     CMP R1, R2       // start-end => updates CPSR
19     BGE return       // if start >= end => exit
20     MOV R4, R1       // pivot = start
21     MOV R5, R1       // i = start
22     MOV R6, R2       // j = end
```

Figure 10: Assembly code of the first if statement in quicksort

We then enter the first while loop of the code labelled whileLoop1, compare i and j and exit to the endLoop1 label if i>=j. Otherwise, the program enters the second while loop labeled whileLoop2, which increments the value of i until a number greater than the one stored at the pivot index is found in the array. We exit to a third loop if the conditions for the second one are not met. The third while loop decrements the value of j until a number smaller than the one stored at the pivot index is found in the array. If the conditions are not met, we branch to the exitLoop1 label.

Once we exited the third loop, we check whether "i<j". If this is false, we exit the first while loop we are still in by branching to the endLoop1 label. Otherwise, we call the helper method "swap" to swap elements at indices "i" and "j" in the array. Since we need to respect the subroutine calling convention, we need to store the three arguments of swap in R0, R1 and R2 respectively. Since swap requires a pointer to the first element of the array as the first argument and does not return anything, R0 does not need to change since it already stores it and will not be erased by the function. However, R1 and R2's contents will be erased and are needed later in the quicksort function. We therefore push them onto the stack before we modify their contents, and pop them after we return from the swap function. We also could have stored their contents in one of the other registers we push when entering the swap function. This process is illustrated in Figure 11:

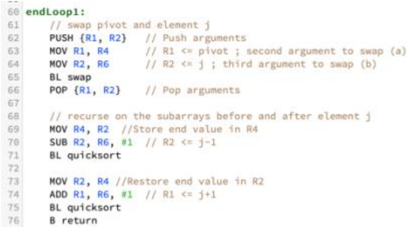


```
    // swap the elements at these positions unless they are already relatively sorted
    endLoop3:
    CMP R5, R6      // i-j => updates CPSR
    BGE endLoop1    // exit while loop if i >= j otherwise swap(i,j)

    PUSH {R1, R2}   // Push arguments
    MOV R1, R5      // R1 <= i ; second argument to swap (a)
    MOV R2, R6      // R2 <= j ; third argument to swap (b)
    BL swap
    POP {R1, R2}    // Pop arguments
    B whileLoop1
```

Figure 11: Assembly code in the quicksort function to call the swap helper method

We implemented the swap function as described in the C program but without using a temporary variable. The elements are swapped directly in the array using the load and store instructions. Algorithmically speaking, this is a big advantage of quicksort since the sorting is done in place, we don't have to copy the content of the array in another memory location. The swap function then branches to the "return" label which will pop the registers R4 through LR and branch back to the next instruction stored in LR.

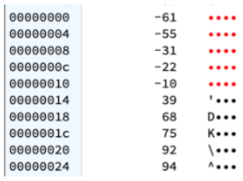
After the end of the first while loop, quicksort calls "swap" again to swap the element at pivot with the one at j, and then recurse on the subarrays before and after element at j by calling itself twice. The first call takes for input the same pointer to the beginning of the array which is already in R0, the start value which is already in R1 and the end value which is now "j-1". We therefore make a copy of the end value (needed for the second call to quicksort) in R4 and move "j-1" in R2. After this first call to quicksort, we put the original end value in R2 and put "j+1" in R1 before calling quicksort a second time. Note we don't have to store the original start value since we don't use it later in the function. After this second call, the function branches to the "return" label and the program will execute the next statement stored in LR.



```
endLoop1:
    // swap pivot and element j
    PUSH {R1, R2}   // Push arguments
    MOV R1, R4      // R1 <= pivot ; second argument to swap (a)
    MOV R2, R6      // R2 <= j ; third argument to swap (b)
    BL swap
    POP {R1, R2}    // Pop arguments

    // recurse on the subarrays before and after element j
    MOV R4, R2  //Store end value in R4
    SUB R2, R6, #1  // R2 <= j-1
    BL quicksort

    MOV R2, R4 //Restore end value in R2
    ADD R1, R6, #1  // R1 <= j+1
    BL quicksort
    B return
```

Figure 12: Assembly code at the end of the quicksort function

After compiling and loading this assembly program, we ran it and obtained the output shown in Figure 13. Note that this output is in memory.



```
00000000        -61   ••••
00000004        -55   ••••
00000008        -31   ••••
0000000c        -22   ••••
00000010        -10   ••••
00000014         39   '•••
00000018         68   D•••
0000001c         75   K•••
00000020         92   \•••
00000024         94   ^•••
```

Figure 13: Results obtained after running the assembly program part2

Quicksort was a more challenging function to implement, but if the subroutine calling conventions are always respected the function can easily be implemented.