Karl Rouhana 260948648

ECSE 324 Lab 1 Report

Part 1.1: *Exponential function*

1. My approach:

I first started by initializing my constants x and n and loading them in R0 and R1, then proceeded to call my function. Since I'm going to do some calculation to find the exponential of x, I need another register to store my temporary results , hence I'm initializing R2 to 1 using MOV. In my loop branch, the first step is to check using CMP if my counter R1 (n in this case), is 0. If R1 is greater than 0 then we should continue our program by going into another branch to continue with the calculations (This is the recursive step, using B to branch). In this branch, I first down count R1 by 1 and I store the result back in R1 (equivalent of doing n = n-1), I then multiply R2 with R0, which is just multiplying the previous result of our exponential, again with x, and storing it in R2 for the next iteration (result = result * x). After these 2 steps, I branch back to the loop (where it was called) using LR to check again if n is 0 and continue with my program. When n is finally 0, our calculations are done, so we return the result from R2 to R0, and we branch back to the start using LR, since it's the last instruction.
When the function should terminate, we change the address of R2 to the address allocated for the result, and we store the value returned in R0, to the result address. Since we must do another function call with different values, I re-initialized my different constants in R0 and R1 and redid the same process again. The result of the second function call will be stored in memory address result + 4.

2. Challenges faced:

I honestly thought this was going to be as easy as lab 0, but I was unfortunately wrong. There was a lot of information coming my way : between the C-code and the Assembly instructions, I quickly felt overwhelmed. So, I took notes of each step and each instruction that needed to be done and converted them to Assembly instructions, which was of course the biggest challenged I faced. I also did not expect to deal with that many areas in the code: The memory, the disassembly, the registers table, but especially LR.
At first, the debugging of the code was difficult, but after reading back the examples in class and given by the prof, I was more confident with coding in assembly, which also helped me to realize I had to "declare" the variables before the start and allocating space in memory for them to be loaded later in the program.

3. Possible improvements:

I'm sure that my code is not optimal. I do not think my code for this part is optimized, but I am sure that if I come back at the later this semester, I will find more efficient and simple ways of writing this code.

I also realised that anything to the power of 0 will yield a wrong result, but there's no way around that since we must copy the C code, which is why the exponential by squaring method is better.

Part 1.2: *Factorial function*

1. My approach:

I first started by initializing my constant n and loading it in R0, then proceeded to call my function with BL. Since I'm going to do some calculation to find the factorial of n, I need another register to store my temporary results , hence I'm initializing R1 to 1 using MOV. In my loop branch, the first step is to check using CMP if n (R0 in this case), is less than 2, if so, then we go to the base case branch to store back the result in R0 and terminate the program. Otherwise, we should multiply n with n and store the result temporarily in R1 and decrease n by 1 and run the program again recursively using LR.
When the function should terminate, we change the address of R1 to the address allocated for the result, and we store the value returned in R0, to the result address. Since we must do another function call with different values, I re-initialized my different constant in R0 and redid the same process again. The result of the second function call will be stored in memory address result + 4.

2. Challenges faced:

After having done the first part, I felt confident implementing a basic recursive program and hence, I did not face any new important challenges.

3. Possible improvements:

Like the first part, I am sure that the code is not optimal and can be even more improved. I tried to use as few register and function calls as possible. Having said that, I am not sure about any other possible improvements.

Part 1.3: *Exponential by Squaring*

1. My approach:

Like the other parts, I must initialize my variables x and n in R0 and R1, and then proceeded to call my function using BL. I also need a temporary register to store the result, hence I use MOV to R2 and set it to 1. I now must check if n is 0 or 1 (using CMP), if it's 1, I branch to the base case to store my result in R0 and return the result, if n is 0, then the result should be 1, hence I'm assigning 0 in R0 in the caseOfZero branch and returning the value in main.
If n is not 1 or 0, then I must check if my number x is odd or even, so we must check the LSB only by performing AND on R1 with the value 1 and storing the result in a temporary

register R3. I will now CMP R3 with 0. If it's equal, then R1 is even, and we go to the even branch. Else, since we know that R3 is not 0, then it must be 1, and we go to the odd branch. In the even branch, I must multiply x with itself, hence R0 and then storing it back in R0, and do a shift of bits of n by 1.
In the odd branch, it's the same steps as the even branch, but first I must multiply my temporary result with x and update it (R2).
For both branches, after the steps of the current branch are done, I go back to where my branch was called, to check again for the base cases recursively.
When the function should terminate, we change the address of R2 to the address allocated for the result, and we store the value returned in R0, to the result address. Since we must do another function call with different values, I re-initialized my different constants in R0 and R1 and redid the same process again. The result of the second function call will be stored in memory address result + 4.

2. Challenges faced:

This part was a bit harder than the first two since it was longer and had to think about multiple cases. Since it was exponential, I thought I could just copy the code of part 1.1 and change it a bit, but it was way different since the cases are more in depth, and this time you had to worry about a number to the power of 0.

3. Possible improvements:

Again, I'm sure my code is not optimized, but I tried my best using the least number of function calls and registers as possible. I now realize that I could have not used R3, I could have used R2 and use the stack to save the value in R2 and then popping it before branching out, I could have saved a register place.

Part 2: *Quick Sort*

1. My approach:

I tried to follow the C code as closely as possible. The main function started by initializing the length in R1, the address of the array in R0 and the start variable in R2 by using MOV. I then proceeded to decrease the length variable by 1 and storing it in R3, then I pushed these registers and LR and called my function.
First, to avoid clobbered registers problems, I pushed from R4 to LR. I started by comparing start and end, if it's bigger than the function call should return recursively to where it was called. Otherwise, we continue the program by assigning pivot, i and j to various temporary registers. After that, we check if i<j true, if it is we branch to the while loop and directly go for the loop to increment i. In this branch, since I'm going to use R7 and R8 as temporary registers, I have to push/pop them. I check the various conditions to increment i, if they all pass, I stay in the loop and increment, else I branch to the other while loop to increment j, and same process, except that if they don't pass, I branch to the next part of the initial while

loop. Here, if i < j, I swap the required elements, else I branch back to the start of the while loop. After swapping another time, I save the old value of end by pushing R3 onto the stack, then called back the quicksort function with the new end (j – 1). After this call, I restore the old value of end by popping it off the stack and calling quick sort once again with my new start (j +1). When the computations are done, all function call are going back to their callers since I save LR and then I branch back with BX LR.
The sorted array is still in the address of R0.

2. Challenges faced:

By far the hardest part, I did not realize it was going to be that hard. Replicating the program was not the hardest challenge for me though, it was to fix the clobbered registers problem I had. With little to no help on the internet, I had to figure it out. The worst part is that the problem worked fine if I ignore the exception raised at runtime, which was frustrating. After a lot of debugging, a post on Teams and some change in my code, I finally fixed the program and now it works perfectly, and I am very proud :)

3. Possible improvements:

Here, I'm sure that my code is not optimized. I used the stack a lot, so that is a lot of push/pop instructions, I have a lot of calls to other branches which is expensive, I also could have saved some registers. In addition, the prof made a post on Teams about the runtime and overall complexity of the program, and after comparing my result with that of other students, it was a confirmation that my code can be further optimized, but I'm going to leave that for another time, I'm just happy that it works.