ECSE 324 Lab 1

ID: 260889422

Part 1: Calling all functions

1. Exponential function:

Description:

We have to write a function which takes 2 inputs, x and n. It then calculates the value of x^n iteratively and returns the value.

Approach:

I used R0 as my x value and R1 as my n value as described in the lab document. I implemented the function very similarly to the C code provided. I used R3 as the register in which I store the accumulated value, and initially set it to a value of 1 as the C code did. I compare R1 (n) with 0 to check that n is greater than 0 since x^0=1 so if n=0 we will just return R3 which has a value of 1. If R1 (n) is greater than 0 then we branch to the for loop implemented within the method. In the for loop R1 (n) is used as the "i" value in the C code for loop and we multiply R3 with R1 (x) in every iteration of the loop, then decrement R1(n) by 1. When decrementing R1 (n) we use SUBS as our command so we don't have to call CMP to set flags. Since we decrement by 1 then when we call SUBS the CMP is with #1 and thus the loop terminates when R1=0 as we desire. At the end we call MOV to move our output in R3 into R0. Then we BX LR to return to start. The function was tested using the values in the main function in the C code provided in the lab document, i.e. (-5)^5 and 2^10. The correct results were obtained. After the first test, its value is moved to R2 to be stored before the second test is run.

Challenges:

None, this was fairly simple to implement as the assembly code is quite similar to the C code provided.

Shortcomings, possible improvements:

Program performs as desired. I am not sure how it can be improved.

2. Factorial Function:

Description:

We are asked to write a recursive factorial function which takes only 1 input, x

and calculates the factorial of x (x!).

Approach:

The input x is in R0 which is also where I return the result. Initially in start we push R4 and LR as these registers will be used and must be restored to their original state as per the caller-callee convention. Once we have BL'ed into the fact function R2 is pushed into the stack first as this register will be used within the function and needs to be restored at the end. We also push LR into the stack to store our return point back to start. First register R2 value is set as 1, for our base case. Then we BL into subroutine factorial. This is used because we don't want to keep setting R2 to 1 every time we recurse so we use a subroutine to store a value 1 so that when a base case is encountered we can simply return R2.

Inside the factorial subroutine we once again store LR to store our return point. At first we do CMP R0, #2 to set condition flags based on value of R0-#2. Then we call BGE to check if R0-#2>=0, i.e. if we have a base case. If not, we Branch to another subroutine called rec. If we do have a base case we simply move R2's value into R0 (our output) and return this value by popping from the top of the stack into LR as the top of the stack contains our LR, and then we do BX LR to return. If we do not have a base case then we enter the subroutine rec, where we first push R0 into the stack as we want to store this value since this will be the input to our future recursive calls and will be changed. We eventually want to get back this original R0 value to carry out the "n*fact(n-1)" part from the C code. Now that the R0 value is stored in the stack, we can change it appropriately for our recursive call's input. So we subtract 1 from R0 and then do BL factorial to carry out our recursive call. Since we use BL, and since the first line of "factorial" contains PUSH {LR} we can store our return point to eventually return back to this point. After recursing, since LR is on the top of the stack and since we do POP {LR} and BX LR to return, the top of our stack now has the original R0 value. So we pop into R3. R0 has the value returned by the recursion. So we do MUL R0, R0, R3 to carry out "n*fact(n-1)". We keep the product of the multiplication in R0 as this is our output register. Then we simply do POP {LR} and BX {LR} to return to the point where we had called BL factorial. Then we have to do POP {LR} again to get the address back to start. We also POP {R2} since we want to restore the value of that register and it was at the bottom of the stack since it was the first thing that got pushed. Lastly, we BX LR to return to start.

The function was tested using the 2 values provided in the lab document, i.e 5 and 10 respectively. Correct outputs were obtained for both tests. After the first test, the value at R0 (the output) is moved to R1 before the next test is started.

Challenges:

The main challenge for this part of the lab was understanding how to implement the recursive steps in ARM assembly and how values can be stored in the stack and then popped after returning from a recursive call to use the newly returned value in a

calculation with the stored value in the stack. Another challenge was understanding that LR is only updated when BL, BLGE, etc. is called and if we have BGE instead of BLGE for example, then LR will not be updated.

Shortcomings, possible improvements:

During the demo, I was told that my function was not recursive as it was not doing anything with the returned values of recursions and I had accidentally written the code in a way such that it was actually a loop. Since then, I have fixed my code to properly carry out recursion by pushing R0 into the stack before a recursive call and then popping after returning from the recursive call and using it for the multiplication, as previously I was doing this multiplication before the recursive step so it was not really a recursive function. This problem has helped me understand recursion as a concept a lot more clearly and made me more comfortable with coding recursive algorithms in ARM assembly.

3. Exponentiation by Squaring

Description:

In this part, we are asked to implement a function that carries out exponentiation except via squaring through recursion. Similar to our interactive exponentiation function, R0 will be our base x and R1 will be our exponent n. We are required to calculate the value of x^n. Our output will be in R0 once the function has finished and returned to start.

Approach:

At the start we use MOV to put in our inputs for x and n in R0 and R1 respectively. Then we push R2 and LR onto the stack as these 2 registers will be used in the function and thus, have their values changed so we want to be able to restore them to their original values at the end of the function. Then we call BL exp to enter the function.

At first we push LR onto the stack to save our point of return back to start. Then we do the appropriate comparisons for the 2 base cases. First, CMP R1, #0 to see if n=0. If n=0, we branch to a subroutine called base_case_1 where we set our output R0 as 1 and return the value (R0 is our output). Carrying out POP {LR} and BX LR respectively at this point will return us to start. If n is not 0 then we do CMP R1, #1 to check if n=1. We branch if n=1 using BEQ base_case_2 to the subroutine base_case_2. In this subroutine, we simply POP {LR} to set LR to our return point and then call BX LR as R0 already contains x and we will simply return the value of x.

In the case that a base case is not encountered, it is time to implement recursion. But before that we have to determine if n is odd or even. To do that we do TST R1, #1 which carries out R1&#1 and sets condition flags based on the result of the logical AND.

If the logical AND results in a value of 1 it will be because the least significant bit of n is 1. The least significant bit will be 1 only for odd numbers. Hence we do BGT n_odd to branch to the subroutine for odd scenarios as the result of the logical AND will be 1 when n is odd and 0 if even.

If n is indeed odd and we reach the n_odd subroutine then at first we push R0 onto the stack to store the current R0 value as R0 is our output as well as the input for our exp function so we will be changing R0 for our recursive calls and we need the original value later on after we return from our recursive step to be multiplied with our returned value (x*exp(x*x, n>>1). After this we do MUL R0, R0, R0 to change R0 value to x*x as it is for the input of the recursive call to exp. Then we change R1 by right carrying out a logical right shift by 1 one place via LSR R1, R1, #1. Now that our inputs have been properly set we can do BL exp to recurse through our function till we reach a base case. At the beginning of exp, we do PUSH {LR} so our LR value (which is updated when we call BL) is stored in the stack. When we do POP {LR} and BX LR after encountering one of the base cases, and return, our returned value will be in R0 as we have used this register for our output through the entire exp function so far. Since we have returned to this point, all the LR values have been popped off of the stack and the top of the stack now contains our stored R0 value. So we do POP {R2} to store our original R0 value in R2 and then call MUL R0, R2, R0 to carry out x*exp(x*x, n>>1).
In the end we simply do POP {LR} and BX LR to return to either start or our return point in case we are in a recursive step.

Now if n is odd, it would not enter the n_odd subroutine, instead it would go to the next line after BGT n_odd. Here we simply update R0 and R1 values without having to worry about storing the current value of R0 as we won't need to use it and we will be returning the value of our recursive step alone. Hence after updating R0 and R1, using MUL and LSR respectively, we call BL exp to recurse. Again, PUSH {LR} at the start of exp will make sure our return point, the updated LR, is stored. Once we return to this point, since we haven't pushed anything into the stack since entering the scenario where n is even, the top of the stack still contains our LR so we simply do POP {LR} and call BX LR.

Once we have finished recursion and called BX LR to return to start, we move our output value which is in R0 into R3, before updating inputs for our second test. The function was tested for 2^10 and (-5)^5 and provided the correct outputs.

Challenges:

The main challenge for this part of the lab, similar to factorial, was understanding how to implement the recursive steps in ARM assembly and how values can be stored in the stack and then popped after returning from a recursive call to use the newly returned value in a calculation with the stored value in the stack. Another challenge was understanding how carrying out n&1 tells us if n is odd or even.

Shortcomings, possible improvements:

During the demo, my function was written to return outputs in R2 instead of R0. This issue has been fixed for the code submission. The function may be further improved by optimizing it so that there are less lines in start so that some memory may be saved.

Part 2: Sort this out quickly

Description:

We are asked to implement a recursive quicksort algorithm which can sort any given array of integers, if provided with a pointer to the beginning of the array, the start index and the end index of the array. This function will also utilize a function called swap which swaps the values at 2 indexes of the array. The 2 indexes it will swap are provided as input to swap as well as a pointer to the start of the integer array. Our pointer to the start of the integer array is stored in R0, our starting index (int start) is stored in R1 which is the same as R0. Our ending index (int end) is stored in R2. To verify if our output is correct, after our program finishes running we will check the memory tab in the ARM simulator and check if the values stored starting from address 0x000 to the end of the array are in order from smallest to largest.

Approach:

For this problem, as the algorithm itself is quite complicated I tried to follow the C code as closely as possible and translate it to ARM as best as I could.

After pushing our inputs into registers R0 through R2 as stated in the Description above, we push registers R4-R9 and LR into the stack as we will use these registers within our function. Then we do BL quicksort to branch into our function. As usual, we push LR onto the stack at the start of our function since we just called BL. After this we push R4-R10 into the stack again as this method is recursive and we want to keep restoring R4-R10 at the end of every recursive step to avoid clobbering registers we use.

For the first if statement, I compared R1 (start) and R2(end) to set condition flags based on R1-R2 and then called BLT to branch to subroutine “if_statement” if R1-R2 is less than 0, or in other words, if R1 is less than R2 (start<end). The subroutine “if_statement” was named as such to help me identify where I am within the C code, or which part of the C code this is dealing with. In the C code, int i, j and pivot are initialized and used within the while loop. So here we initiate i, j and pivot by transferring appropriate values by calling MOV R6, R1, MOV R4, R1 and MOV R5, R2, where R6 is the pivot variable from the C code, R4 is i and R5 is j.

After this we enter a while loop within the C code so we make another subroutine to implement a while loop, aptly named “while_i_less_than_j” to help us identify where we are within the C code. Inside this subroutine, we simply compare R4 and R5 (i and j) to check if i<j and decide if we should terminate the loop. If the result of CMP R4, R5 (R4-R5) is less than 0 then i<j and we keep iterating through our while loop. So we do BLT inner_while_loop to branch

to the subroutine “inner_while_loop”, again aptly named to help us identify which part of the C code this subroutine will deal with. It is important to remember this BLT call as if this BLT call doesn’t lead to us branching then this means we are at the end of our primary while loop and it is time for the recursive call. So we will be returning to this point but for now let’s assume i<j so we entered the subroutine “inner_while_loop”.

In this inner loop we check if array[i] <= array[pivot] and also if i(R4)<end(R2). So first we get the value for array [i] and array [pivot] by doing LDR R7, [R0, R4, LSL#2] and LDR R8, [R0, R6, LSL #2] respectively. Then we compare the 2 values using CMP R9, R10 (R9-R10) and set condition flags. If R9<=R10 then R9-R10<=0 so we branch to the second condition of the while loop if the result of CMP is less than or equal to 0 so we call BLE second_condition to branch to the “second_condition” branch where we check if i<end which is relatively simple using CMP R4, R2. But if the initial condition array[i]<=array[pivot] is not satisfied then we branch to the second while loop using B second_inner_while_condition. The “second_inner_while_condition” subroutine checks conditions similar to the way we check conditions for array[i]<=array[pivot] but for array[j] > array[pivot] (here j = R5). If the first while loop is entered after satisfying the conditions inside “second_condition” then we branch to “first_inner_while” where we simply increment R5(i) and then branch back to the branch where we checked the conditions to enter this while loop(“inner_while_loop”) and keep checking conditions and incrementing till we step out of this while loop when the conditions are no longer satisfied. In this case we, branch to “second_inner_while_condition” like previously mentioned. Inside “second_inner_while_condition” we check conditions for the while loop as previously mentioned. If conditions are satisfied we branch to “second_inner_while” where we simply decrement R5(j) as in the C code, and then branch back to “second_inner_while_condition” to check conditions again and repeat reiterate through the loop. But in the case when conditions are not satisfied then we branch to “if_i_less_than_j”.

In this branch, we compare R4 and R5 and check if R4<R5. If R4 is indeed less than R5 then we branch to “store_in_stack” to set up our stack before we carry out swap. Otherwise, we simply reiterate through the while loop by branching back to “while_i_less_than_j”, since this is the start of our outer while loop.

Inside “store_in_stack” we push value of R2 into the stack as we will use this register as one of our inputs for swap, and after finishing the swap we want to restore this value. So we move R5 (j) into R2 after this and then proceed to branch into swap using BL swap so that LR is updated as we want to return to this point to restore R2.

Inside the method swap, we first push LR, R7 and R8 into the stack to maintain caller-callee convention and also to prevent clobbering registers as we will need to use registers outside R0-R3. R7 is used as our int temp variable and will contain value stored at index R3, since R3 is our input int a. Similarly, R8 will store our value at index R2 since R2 is our input int b. Then we simply call STR R8, [R0, R3, LSL#2] and STR R7, [R0, R2, LSL#2] to swap the values stored at indexes in R3 and R2 with the values we plugged into R7 and R8 at the beginning of swap. Then we pop R7, R8 and LR off the stack to restore their values and call BX LR to return to “store_in_stack” where we then pop into R2 to restore the value of R2 as well. Now we are done with swap and therefore the if statement, if(i<j). Hence we reiterate through the while loop so we branch back to “while_i_less_than_j” which was the start of our while loop.

When i>=j it is time to step out of the while loop and carry out our swap along with the recursive steps. So if we look at the subroutine “while_i_less_than_j”, if BLT “inner_while_loop” does not have the required condition flags to branch “inner_while_loop” is not entered and therefore we go to the next line where we prepare register R2 to carry out swap. First of all, R2 is pushed as we want to restore its value. Then we move appropriate values into R2 and R3 as these 2 registers are our inputs for swap. Then we branch and link to swap and carry out the function. After returning from swap we do POP {R2} to restore the value of R2. Now, similar to how we did swap, we need to carry out a recursive call to quicksort with a different value of R2(j-1) so we store R2 and R5 (j) to restore after we have returned from the recursive call. We change the value of R2 appropriately and the branch and link to quicksort recursively via BL quicksort. Then we pop values of R5 and R2 stored in the stack back into R5 and R2 by calling POP {R5} and POP {R2} respectively, once we return. We do the exact same thing for the next recursive quicksort call but this time we push R1 and R5 onto the stack.

Lastly, we have the subroutine return. We use this as the final subroutine where we POP {LR} and do BX LR. The point of using this subroutine is that whenever we branch to return we know the top of the stack contains the LR that will take us back to start, and is just something that helps to keep track of the stack. We also branch to return at the very beginning of quicksort if we do not satisfy the initial condition and have i>j (this is the base case).

At the end, we simply pop R4-R9 and LR from the stack and restore their original values to abide by the caller-callee convention.

Challenges:

It is quite a complicated algorithm to me, as I mentioned at the beginning and therefore the biggest challenge was navigating through the nested while loops and knowing when to branch to what, therefore the aptly named branches which point to which part of the C code we are working with.

Shortcomings, possible improvements:

At the time of the demo, my function took the memory address of start and stop as an input instead of the indexes. I have since fixed that for the code submission. However, I believe I could have improved my code’s performance significantly by using one PUSH and one POP statement to PUSH and POP multiple registers to and from the stack. This would have saved runtime and cut down a few lines of code. In addition, my swap method inputs are written to be as swap(R0, R3, R2) representing swap(int *array, int a, int b) from the C code. This could be rewritten to be swap (R0, R2, R3) to maintain the ascending order of registers.