**ECSE 324 Lab 1 Report**

**Part 1A: Factorial**

Approach

|  |
| --- |
| 2 |
| LR line 23 |
| 3 |
| LR line 23 |
| 4 |
| LR line 6 |
| *Figure 1* |

I stored the input n in R1 and called the “fact” subroutine with BL (subroutine calling convention). If the input was less than 2 using CMP, it would go to the base case. The base case sets R0, the output, as 1 and then branches to the link register LR. If the original input was 0 or 1, it would not go any further in the code, because LR would be pointing to B END.

For the non-base case, I pushed the link register and R1, decremented R1, then ran “BL fact” within the fact subroutine. In C, that’s basically calling fact(k – 1) within fact(k), k ≤ n. When it reached the base case from the non-base case, it would call “BX LR”, where LR would contain the address for the POP command. It would pop the R1 that was pushed, multiply it to the output R0 through “MUL R0, R0, R1”, pop the next LR that was pushed, then branch to it. That new LR would also point to the POP command, and the cycle would continue until we reached the last LR in the stack, which points to “B END”, line 6 in the code. Figure 1 shows the stack when fact(4) is called and reaches the base case. The 2, 3, 4 on the stack are popped to multiply with R0 = 1 to get R0 = 24.

As per convention, the callee stores the output in R0 and returns to the calling code with BX LR. Registers R4-12 were not used to begin with.

Challenges

This was definitely a program I simply could not devise on my own. The iterative version is trivial but calling the same function within the function was a major roadblock (technically, I should be calling them subroutines). In the end, I found this [Quora post](https://www.quora.com/How-do-I-create-a-recursion-in-ARM-architecture-assembly-language) that had the code for it. After understanding how it worked, I rewrote it without looking at it, and mines has 4 less commands. I then told the professor about it, and he said the code I wrote was fine in terms of not plagiarizing. In short, the hardest part is visualizing how it should work recursively.

This lab threw us into the cold water with recursive functions and deeply nested loops. We really had to apply ourselves. Perhaps it is to compensate for the fact that we do not have to work with a physical board. I do wish tutorial 3 covered recursive functions.

Shortcomings/Improvements

* It is only certified to handle non-negative integer input. If the output is over 32 bits, you will not get the answer that you want.
* It must take R1 as its input and not R0, that way the callee has two registers to work with. If I had used R0 as my parameter, I would have to store the argument into R1 to make way for R0 as the output. Then I would be obligated by convention to return R1 to what the caller intended.
* The iterative form of factorial will always be more efficient.

**Part 1B: Fibonacci**

Approach

Looking back, I think the structure is not too dissimilar from part 1A, so I will be concise. R0 is the input, but it also becomes the output when all the “fib” subroutines are called (for calling convention purposes, and like part 1A, it satisfies all the requirements). R1 stores k, k ≤ n, while R2 stores fib(k – 1). Whenever fib is called, it pushes R1, R2, and LR to be used later. The base case occurs when R0 < 3, it sets R0 to 1, the base case value. Otherwise, the program goes to the first recursive case, fib(k – 1). The base case also pops LR and branches to it. If it was recently called by fib(k – 1), it will go to the second recursive case, fib(k – 2). If it was recently called by the second recursive case, then it proceeds to add fib(k – 1) and fib(k – 2). Afterwards, it pops the stack again, effectively going up a level.

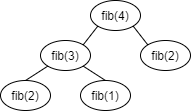
To visualize the process, imagine doing postorder traversal on a Fibonacci tree (figure 2). In other words, for n = 4, we go bottom left, find fib(2), then fib(1). Next, we add them up and go up a level, synonymous with popping. Then the fib(2) on the right side of the tree is computed, which is then added to the existing result.

Figure 2

|  |
| --- |
| … |
| R1 when fib(3) was called |
| R2 when fib(3) was called |
| LR line X |
| R1 when fib(4) was called |
| R2 when fib(4) was called |
| LR line X |
| *Figure 3* |

I will not mention this in the future, but for all my programs in this lab, I end with an infinite loop. This prevents the program from repeating itself or going through unintended addresses. Without the ending loop, the emulator will create a breakpoint at the end, but it is better safe than sorry.

If you look at the stack at any given moment, it should look like figure 3.

Challenges

If the previous subpart was hard to visualize into code, this subpart is even more challenging! I knew the structure would be similar to the factorial function, but I didn’t know how to modify it to be fibonacci. After looking at [a solution by Cambridge University](https://www.cl.cam.ac.uk/teaching/2002/CompDesig/lecture4.pdf), I was able to figure it out. Like the previous part, I then closed the webpage and made modifications to my factorial code to turn it into Fibonacci.

Shortcomings/Improvements

* There are more efficient ways of calculating Fibonacci: iterative, closed-form, dynamic programming/memoization.
* Two extra parameters to decide the starting values.

**Part 2: 2D convolution**

Approach

I copied the matrices from the C code. Then I initialized space for the output, with the same amount of memory as the fx image. I assigned a register for each necessary variable as I went along, using comments to remind myself which variable does R6 represent for example. Having written two nested loops for part 3, I used the same method to write four nested loops, with each loop declaring a loop variable beforehand, comparing on some condition, and branching to the end if it fails. At the end of the loop, it increments the loop variable before branching to an outer loop. In the inner most loop, for the if condition, I used four consecutive CMP and branch commands. Because my arrays in assembly are not actually 2D, I used “(row \* width/height + column) \* 4” to get the offset. I also stored the address of kx and fx with R11 and R12, and with it I could get the address of any element in the arrays. Finally, with basic arithmetic, I was able to figure out the sum for each pixel and write it onto the memory addresses of the gx array. Make sure to skip past the fx and kx memory addresses to see the output in the memory tab.

In terms of the algorithm itself, it was important for me to understand how it works from a non-coding perspective. After watching a Computerphile video on it, I realized that that the kernel should have odd dimensions, and that the if condition in the most inner loop is meant to handle pixels towards the edge of the matrix.

Challenges

This one wasn’t that hard but it was a headache to work with 4 nested loops. I started out by writing only the loop code and testing that the counters were incremented properly; that extra effort paid off with less debugging down the road. The other headache was rationing my registers. Because I was not familiar with stacks, I decided not to go down the function route. First, I stored fx width and height in one variable. I did the same for the kernel (kwh). Kernel width/height stride can be derived by shifting to the right the kwh register. When I got to calculating gx[x][y], the values in the registers in the further inner loops were not needed anymore, so I overwrote R0 and R3. When debugging, I kept getting a misalignment error until I figured out that my offset was not multiplied by 4 to account for the fact that the memory is word-aligned (4 bytes). I also thought that shifting by 4 multiplies by 4, that was embarrassing.

Shortcomings/Improvements

* I must store the matrix dimensions separately; they can’t be inferred. If there’s a mismatch, expect it to not work. It would be handy to check the dimensions of the image and kernel to confirm that it’s square.
* I should test my code more, but preparing test data for this part specifically is cumbersome.
* It could be turned into a function for modularity (and for part 3). Then filters can be composed.
* A feature would be a parameter to decide how to handle the border pixels, but I don’t think it’s a big enough improvement to be worth my time unfortunately.
* Make a 1D or 3D version!

**Part 3: Bubble sort**

Approach

Like the part before, I needed to use access the memory, because I was dealing with arrays. So I stored the array and its length in memory. They are declared at the top of my code, so that they can be easily modified, like parameters in a function. To convert the C code to assembly, I converted the for loops to while loops. I had to make sure to subtract size (R1) by 1 at the beginning, because bubble sort deals with size – 1 in its for loops (to prevent the left or right pointers from going 1 slot out of bounds). As for the looping, I had to make sure that for every loop, it began with comparison and branching to the end, and ended with incrementing a counter variable and then branching to the loop above (or in the case of the most outside loop, to the end). I didn’t need a temporary variable for swapping while sorting, because I used registers 6-9 to store the pointer addresses and values, using this syntax LDR R8, [R6] for dereferencing. If the value at the left pointer is greater than the value at the right pointer, then I swap the values in the memory locations through two store commands.

Figure 4 does a good job of visualizing how I wrote the loops. The “// …” is where you can place identically structured nested loops or non-boilerplate code.

|  |  |  |
| --- | --- | --- |
| for (i = 0; i < limit; i++)  // ... | int i = 0  while (i < limit)  // ...  i++ | mov r1, #0  loop:  cmp r1, r2  bge loop\_done  mov r2, #0  // ...  add r1, r1, #1  b loop  loop\_done: |

*Figure 4*

Challenges

This was the first part I started on, because I did not know about stacks yet and part 2 looked frightening. Looking back, it was a great warmup to part 2. Being new, I wasted a lot of time figuring out why “BGE R5, inner\_loop\_done” does not work. When debugging, it was not working because I forget either “B inner\_loop” or “B loop”; as a result, my array was only partially sorted. I also worked hard to get my code to structurally look like the C code (with the indentions; it paid off with easy debugging. I did have to invert the conditions from the C code but it is a small sacrifice. It’s interesting that the actual sorting was only 8 lines; the rest is looping and other boilerplate.

Shortcomings/Improvements

* Since it is easy to make it sort in descending order (just change BLT to BGT in my code), an easy improvement would be to include an extra parameter for deciding this, maybe 0 for ascending, 1 for descending.
* Like the previous part, I need a separate variable to store the length of the array (I take Java and Javascript for granted).
* Bubble sort’s time complexity is not as good as merge sort.