ECSE 324: Computer Organization
Lab1 Report

Part 1: Recursive Function Calls
Part 2: Loops and Arrays
Part 3: Sorting

Grey Yuan (260857954)

Part 1: Recursive Function Calls

Part 1A: Factorial

Objective

The goal of this part is to convert the C code that calculates the factorial function of a given integer n with recursive function calls to ARMv7 assembly language.

Initialization

First thing I did is to think about the registers used to restore these values since we will be calculating factorials recursively. I decided to use R0 to restore the factorial equation (the values return by each function call), and this means we will get our final solution in R0. Then, I decided to use R1 as the place to input my value of n that we wish to calculate the factorial of. Also, R1 will act as a looping index to record how many times we need to call subroutine "factorial". It is important to remember to push the return address which is store in link register into the stack before calling our subroutine "factorial". We also need to make sure to pop it from the stack after calling and store our value in R0.

Approach

Now we need to implement subroutine that contains the assembly code converted from C to calculate factorial. In order to convert "if else" structure in C, we define subroutine "factorial", "then", and "else". In "factorial", we first negate the condition n<2, so we need to check if the input n stored in R1 is larger than or equal to 2. If n >=2, we switch to subroutine "else". Here R1 should be pushed into stack to restore the current input, then we should pass R1 = R1-1 to next subroutine call. We also need to multiply the current R1 with all the previous multiplications stored in R0 to get a new R0. We will keep doing these until n=1. In that case, we will go to subroutine "then" to move 1 to R1 and then go back to "factorial". In "factorial", we pop the last value from the stack, return the value in R0, and end the program. Therefore, all the value of R1will be popped and R0 expression will be like R1*(R1-1)*(R1-2)*…*1. That is exactly how to calculate factorial of n which is restored in R1.

Testing

As suggested in the lab description, I used Fact(4) = 24 to verify the correctness of my code. So, when I input n=4 to restore in R1, I should see the hexadecimal expression of 24 which is 18 show up in R0 for the register display.

Challenge

This factorial recursive call is not very complex and is well-covered in class, so not a lot challenges are encountered. One thing that has been solved by the professor but really bothered me for a long time is to use B stop instead of B end to terminate the program.

Improvement

I think a relatively easier way to implement the factorial function with ARMv7 is looping instead of recursive calls.

Part 1B: Fibonacci

Objective

The goal of this part is to convert the C code that calculates the nth Fibonacci number with recursive function calls to ARMv7 assembly language.

Initialization

Similar to my register choices in previous part, I chose R0 to store final answer of the nth Fibonacci number and R1 to input the value of n. According to the equation Fib(n) = Fib(n-1) + Fib(n-2), I chose to use R2 to keep track of Fib(n-1) and R3 to record to Fib(n-2). Also, similar to how we push LR to record the state of the program before calling subroutine "Fib", and pop to use it.

Approach

Similar subroutine setup to Factorial is applied here: we have "Fib", "then", and "else". The major issue is how to store our values from Fib(n-1) and Fib(n-2), so we may utilize the stack again to keep track of one while we need to compute the other. So, we will use stack when we progress to subroutine "else" to compute Fib(n) = Fib(n-1) + Fib(n-2). We push LR, R1, R2, and R3 into the stack before we do any calculation. Since we have input n to the subroutine, we pass n-1 to the next iteration always. These recursively process will be iterating until the input become 2 (when input reaches n<3 condition in the C code). In that case, we will branch to "then" to put 1 in R0. Then, we exit and save R0 in R2. Next, we will input n-2 instead to the subroutine and get the return value to save in R3. In general, R2 and R3 are used as temporary register to keep track of Fib(n-1) and Fib(n-2) respectively for each iteration. Afterwards, the values stored in them will be gathered and moved to R0 which will be final result of the nth Fibonacci number. It is important to remember to pop R3, R2, R1, LR in order as the last step.

Testing

I used n=5 (R1=5) as input to test the correctness of my algorithm. After we compile and run, we should see 5 in the register R0. Also, if we input 7 as R1, we should see d in R0 as output.

Challenge

I first do not know how to keep Fib(n-1) when computing Fib(n-2), but I figured out that we can use stack to solve almost all the value storage problem like this one. Also, it is confusing at first to know in what order PUSH{A,B,C,D} and POP{A,B,C,D} work at first, but I found resources online to learn about this.

Improvement

Similar to Part A, I think looping will be an easier way to tackle this question.

Part 2: Loops and Arrays

2D convolution

Objective

The goal of this part is to convert the C code that implements 2D convolution using deeply nested for loops to ARMv7 assembly language.

Initialization

To start, I first convert the variables given in the C code to assembly. Then, I start to think about how show I structure my subroutines and algorithm. According to the 4 nested loops, I decided to have 4 subroutines corresponding to 4 for loops and we would also have register R0, R1, R2, R3 correspond to y, x, i, j as looping indices. To add on that, y, x, i, j are respectively bonded by image height, image width, kernel width, and kernel height that are previously defined.

Approach

I have defined subroutines: “Loop1”, Loop2”,” Loop3”, and “Loop4”. In each of the four loops, for example in “Loop1”, I first decrement the index y stored in R0 by 1 in each iteration since I choose to start my index y from the upper bond. As long as the value is not less than 0, we will keep progress to the next nested loop which is “Loop2” as we are currently in “Loop1”. If the reach the end of the loop and the condition is not satisfied, we will terminate the program. In “Loop2”, I first find the total offset stored in R9 by summing row and column offset computed. I defined R7 to store column offset and R8 to store row offset. Then, I convert gx[x][y] = sum (sum is stored in R4) into assembly code since this snippet of code is in the second nested for loop in the lab description. “Loop3” has nothing besides the general condition check. In “Loop4”, after condition check, temp1 and temp2 are calculated first and stored in R5 and R6 respectively. As suggested in the C code, we need to check if (temp1>=0 && temp1<=9 && temp2>=0 && temp2<=9) is satisfied and break if it is not. If yes, we progress to complete the rest of the 2D array calculation which is similar to what did in “Loop 2”. We use R11 as pointer to the element in kernel and R12 as pointer to the element in 2D image. Check annotations in codes for implementation details.

Testing

We will find the memory address in stored in R11which should be 1f4, then we will search this address in the memory tab. In Settings, change format from hexadecimal to decimal signed. Next, we should see the expected output listed in the lab description show up on the screen from 656 to 517. Check to make sure those numbers match the desired output.

Challenge

When starting to write some of my codes, I have also encountered difficulty when dealing with the issue that how I should iterate my for loop. I come to realize that if we iterate starting from 0 as we usually do, we will have to register and store more variable to compare the maximum bond and the current iterating index. I searched online about this issue and learned that I could reverse the for loop direction (iterate from upper bond to 0) so that I can always compare with 0 which does not cost register to store.

Part 3: Sorting

Bubble sort

Objective

The goal of this part is to convert the C code that implements the bubble sort algorithm to ARMv7 assembly language.

Initialization

As a start, I first look at what is the input of the C function. So, I know to put the array and size of the array as my program input. Since I decided to use R1 to R5 as a display to show my result, R0 is used to load the array and R6 is used to load the size.

Approach

Then, started to convert the nested loops written in C to assembly. I set up two subroutines: “Loop1” corresponds to the outer loop and “Loop2” corresponds to the inner loop. In the outer loop, I store size-step-1 in R6 and compare it with 0 directly. If it is greater than 0, then we reset the index of the inner loop. Then, we set the pointer of array R0 to the address of the first element of the array. Next, in “Loop2”, we will compare R2 to R6. If the index of the inner loop is greater than the index of the outer loop, then the current element that the array pointer is pointing at will be compared with the element next to it (R3 and R4 are utilized for that). If one element is actually greater than its next element, they will be swapped. By

repeating these steps, our array will finally be sorted in ascending order. It is also important to remember to pop all the stored values they proceed to the print subroutine to output the result.

Testing

Set the input array to -1, 23, 1, 12, -7 and the input size to 5. After we compile and run, we should see the output array -7, -1, 0, 12, 23 from R1 to R5 in Registers. We should also check the memory and search address 00000000 to verify that the same sorted array is stored in the memory. In Settings, change hexadecimal to signed decimal expression if needed.