ECSE 324 - Computer Organization
Lab session 001
Winter 2020

Lab Assignment 1 - Introduction to ARM Programming

Group 05 :
Narry Zendehrooh Kermani 260700556
Imane Chafi 260847716

Largest Integer

The first section of the lab is mostly about getting familiar with assembly language in the manner that the code is provided to us, and we only needed to understand the logic of the code. This first code is an algorithm to find the largest integer in the list. In the beginning, the code stores the elements, the number of elements and a slot for the result in memory. Then, we load these contents from memory into the registers so that we can do computations on them. The code used a loop to iterate through all the elements while updating the largest element. In the loop, each element is compared with the register that has the largest value.

If the value of this element is larger than the value of the result register, then we load that value in the result as the new largest number, and proceed to the next element. If it is not larger than the result register, then we proceed with the next element without any specific action. The counter is set to the number of elements so that when the counter becomes 0, we exit the loop and store the result register which is the largest element into the memory. A challenge that we faced while writing this code in ARM was understanding the new structure, as it took time for us to get used to the syntax and to analyze the memory locations. An improvement that we could have made on this code is to push each number of the list inside of a stack and pop one number at a time, compare that number with the largest integer stored in the result register and update it if it is smaller than the number popped. This would use a lot less registers to code and would make the code more reusable.

Standard Deviation

In the second part, we are provided with the equation below so that we can calculate the average value of a signal by only using the maximum and minimum values of a given set of numbers. Here is the formula that was provided to us to compute the standard deviation :

$$\sigma = \frac{(x_{max} - x_{min})}{4}$$

Where σ is the standard deviation, and x_{max}, x_{min} are the maximum and minimum values of the signal, respectively. Similar to the code for finding the largest integer, this code consists of a loop that iterates through all the elements to compare them to find the maximum and minimum values. In the first version of the code, we used 2 loops, one for the maximum element and one for the minimum element in the list. However, we quickly realized that the 2 loops could be merged into 1. Indeed, we only used a single loop that checks whether the element is smaller than the register value that is already assigned to the smallest element. If it is smaller, then we branch to the “XMINIMUM” branch to load the element as the new smallest element and branch back to the beginning of the loop.

Otherwise, it will check if the element is greater than the register value for the largest element. If it is greater, then it will be assigned as the new largest value. Otherwise, it does nothing. In both cases, it will branch back to the loop in the end. The loop will end once the counter, which is equal to the number of elements, becomes 0. Then, we simply substitute the largest value from the minimum value that we got from the loop. For the division by 4, we can make a right shift operation of the result by 2. This operation gives us the standard deviation of the list. We can thus store the result into a resulting register. One of the challenges was to apply two comparisons in the same loop. It was a new approach for us to branch out from a loop, update values from registers and branch back to the loop. Indeed, another challenge was to figure out the division by 4 in ARM, which could simply be done with a LRS, logical right shift operation.

Centering an array

To center a signal, it is important to be able to calculate its average value and subtract it from each input. Indeed, to better understand the notion of centering a signal, we were able to apply this methodology to ARM Programming. To make our calculations more simple, we've only taken into account signal lengths of powers of 2, since we'll need to make an average of all values, which means that we'll have to divide by the number of elements in the array and then subtract that average from each input value in memory. This can be easily done with a right shift operation for numbers in powers of 2. For our methodology, we've started by listing the 4 different branches that would hold the logic behind adding numbers for the average, checking if we've looped through each element, dividing by the number of elements in the array and subtracting the average from each signal. At the end of our code, we've also added a memory assignment for the number of entries in the signal list in variable "N", and the list of input data in the variable "NUMBERS". We know that a word is 4 bytes in ARM, which is enough to hold the integers we needed for both the length of the array and the array of each signal, as shown by the ".word" specification. The "_start:" branch starts the execution of our code by simply assigning our length and signal inputs to registers. Some registers hold the length of the array of inputs to be used as conditions for making for loops with branches. To add each element together, we used a branch that first subtracts 1 from the decrement register which holds the number of elements in to act as a for loop. Once the decrement register, R2, is equal to 0, then we would have looped through the branch as many times as there are elements in the array of inputs and we can go to the next branch.

Once all elements have been added to a single register, R0, we will need to divide R0 by the number of elements in the array to have our average. We've thus created a branch, "AVERAGE_VALUE", that loops through the number of elements in the array and divides R0 by the number of elements in the array using an ASR, arithmetic shift right operation. Finally, the code branches back to the "SIGNAL_SUB" branch that subtracts the average from each signal value and storing the result in the memory location of the signal. In terms of challenges faced, finding a way to loop through branches was particularly difficult in the example since we wanted to optimize the code as much as possible by using the least amount of registers and reusing some, but it ended up making our code much more difficult to read so we decided on using 3 registers, all holding the number of elements in the array, so that we could use them in the branches as conditions for exiting the branches once the registers are equal to 0 with a decrement of 1 each time it goes into the loop. Another important challenge was finding out how to store the final values in the code for each signal by subtracting the average. We needed to add a couple of lines of code carefully such that we only go to the next value once the previous result has been stored in the input's memory location using an STR operation.

An improvement that can be made on this code is to simply use the register convention for the first inputs of the length of the signal and its values by loading them in registers R0 and R1. We could also have easily used a stack by adding a PUSH instruction to store the value of the length of the array and not have to think about when registers holding this value would be overridden, and popping this value when we need it. These changes would make our code more readable and easier to reuse in future ARM programs.

Sorting

This part of the assignment asked us to create an ARM code for sorting a list of integers from smallest to greatest, including negative numbers. This sorting code would be using the simple bubble sort algorithm, and could be easily implemented using branches. Indeed, we started our code by listing at the bottom the list of parameter inputs to implement this sorting code : the number of elements to sort as the "N" variable, and the unsorted list of elements that we needed to sort as the "NUMBERS" list. An example of the numbers array would be 1, 4, 2, 3 and it would need to be sorted to 1, 2, 3, 4. Furthermore, we listed under the "_start" branch the loading operations to add the number of elements in the list and the first number of the list in an array. Our code is separated into 3 branches, the outer loop, the inner loop and the branch that swaps 2 numbers together. The outer loop loads elements for the 2 numbers that need to be compared, and moves from the first number to the last once they have been compared in the second loop.

The second loop compares the 2 numbers listed in the registers from the outer loop and if the second number is smaller than the first number, then the code is branched out to the third branch that swaps the 2 numbers together. Once the 2 numbers have been swapped and their new values stored into memory with a temporary variable, we can move both registers 1 numbers to the right and load back into the 2 registers the numbers that now need to be compared. Then, we are brought back to the inner loop and this process goes on until we have gone through all the elements. Then, we are brought back to the outer loop and we need to loop through the outer loop as many times as there are elements in the array "NUMBERS". This is not the most efficient method of sorting, as it gives a O(n^2) complexity, which is pretty slow. However, it is a simple way of using branches to compute a bubble sort algorithm.

One of the challenges we encountered was figuring out how to code the outer/inner loop mechanism. Indeed, at first, we only used one loop which made branching back into the code once the first iteration of swapping was over very difficult to manage. We ended up creating a second loop, using 2 registers to hold the number of elements in the list and decrement each time the branch is passed until they got to 0 and needed to go back to the outer loop, or to simply end the program. Another challenge was to use a temporary register, which we realized only after seeing that one of our registers was overridden everytime we went through the swap branch. Indeed a temporary variable to store the value of the first register once we set its value to the value of the second register was crucial to making our code work. One of the improvements that we could have made on the code is to simply have a stack and push and pop the value of the first register instead of using a temporary register to store it, which saves us from using all of our registers as we don't have many registers to spare. Another improvement on this code would be to pass the length of the array and the values as parameters by using the program counter and the link register such that we could link out after the outer loop and use the program recursively. We could have also used the callee-save convention and created a merge-sort function instead using the BL operation to save the positions of the values in the stack before popping them in the right order and storing them into registers, which would increase our code efficiency and have a code complexity of O(nlogn) instead of O(n^2).