Laboratory 1: Introduction to ARM Programming

ECSE 324- Winter 2020

Lab Section 002 - Group 43
Dafne Culha (260785524)
Majd Antaki (260731626)

Culha Dafne (26078524)
Majd Antaki (260731626)

Introduction:

For this lab, we practiced to work with an ARM processor and the basics of ARM assembly by programming some common routines. Tools that were used were DE1-SoC Computer System and Intel FPGA monitor program and an ARM cortex located within the FPGA. Three different algorithms were implemented: Fast standard deviation computation, centering an array and sorting. Our logic for implementing each algorithm is described in this report. For each part, we used the debug mode on Intel which allowed us to run certain parts of our code and check the updated values and addresses in the register and the memory to see if our code was working as expected.

Part 1 - Fast standard deviation computation

For this part, we used the formula provided in the lab guide which was $\sigma = \frac{x_{max} - x_{min}}{4}$. We use a loop to find the maximum. Based on the code we already have to find the maximum, it was rather an easy task to find the minimum. By switching the branch condition from BGE to BLE, the same algorithm can be used to find the minimum instead of the maximum. By having both the maximum and the minimum, and moving them in two different registers, applying SUBS command to calculate the difference to get the numerator of the final [quotient. In order to get the approximated standard deviation, dividing by 4 without using a division algorithm was merely performing a logical right shift of 2 bits, since each logical shift is equivalent to a division by two.

Part 2 - Centering an array

Centering an array combines two distinct operations; finding the average, and subtracting each number in the array. Finding the average involves adding all the elements in the array. This operation is fairly simple with ARM. A loop was implemented to go through the elements and keep the sum in a given register. This loop needs to add the current element to the total sum as well as increment the address in order to get the next element, while decrementing the number of elements remaining in order to know when to stop the iterations. After that, a division of the sum by the number of elements in the array is needed to get an average. Since the number of elements n in the array was assumed to be a power of 2, division, just like in Part 1, can be done through logical right shift of the sum of $log_2(n)$ bits. Even though the logarithmic operation sounds hard to perform using ARM, but we used the fact that the number of logical shift performed on a number that is power of 2 to reach to number one is equal to applying log_2. Subtracting the

Culha Dafne (26078524)
Majd Antaki (260731626)

average from each element of the array was done in place, where the subtraction happens in the registers and then saved in the address it was loaded from.

Part 3 - Sorting

To sort the array, we used the bubble sort algorithm that was provided to us in the lab guide. We use 2 loops, an outer and an inner loop. The outer loop is meant to scan the whole list of size at most n times. In each iteration, at least one element will be at its terminal position. Bubble sort algorithm has a specific feature that needs to be implemented, namely the ability to stop sorting whenever the sorting process is over. This feature can be fully implemented with a boolean variable, that starts as true at the beginning of every iteration. This boolean is changed to false whenever the bubble sort changes the position of some element. As long as the boolean is false we repeat. Since booleans cannot be represented directly in ARM, a register was used to hold the value of 0 meaning that the list is sorted and one otherwise. The inner loop is the real sorting step. It goes through the array, comparing every two neighbouring elements. If any of these neighbours are in the wrong order, they are flipped. This guarantees that at the end of each run, the maximum/minimum element of the unsorted part of the list will go to its position when sorting from small to large or large to small, respectively.

Possible Improvements

Firstly, to improve our code clarity, we could use aliases for our registers instead of labelling them all as R#.

Secondly, a better sorting algorithm than bubble sort could have been implemented. We could have tried implementing merge sort or quicksort which are more efficient in sorting arrays and have smaller complexities than bubble sort.

Restoring the state of the program throughout the run might be some major issue. Registers should have been pushed before applying the sorting and popped right back when done.