Group 47 Lab 1 Report

Within this lab, four different algorithms were to be implemented, these algorithms are listed and described below. This was to be done using the Intel FPGA monitor program IDE and an ARM cortex located within the FPGA. A tool that was used throughout the lab was the debug mode on Intel, which allowed us to run certain parts of our code and check the updated values and addresses in our register and in memory. This proved as a useful tool in showing whether the code was doing what it was expected to.

Largest integer

Writing an algorithm to find the largest integer in a list was simple. The code consisted of one loop which would iterate through all contents of the list whilst updating the largest integer which is stored in a memory location called RESULT. The algorithm would compare each value in the list to the current RESULT and update RESULT if current value pointed at is larger than RESULT. The number of loop iterations was determined from the size of the list provided, N, if our N was larger than 0 the loop will branch back, and N is decremented. The only challenges faced whilst writing this algorithm was getting used to writing code in ARM, getting used to the syntax and understanding how memory locations and registers worked in specific.

Standard deviation

For this algorithm, we found that using the equation for standard deviation provided in the lab guidelines and shown below was the simplest approach.

$$\sigma \cong \frac{x_{max} - x_{min}}{4}$$

Like our previous algorithm, this one consisted of a loop checking for the max value in our list, however this time storing it in a register. In addition to checking for max value at each iteration of the loop if the value we just checked was not larger than MAX, we would branch out of the loop and compare to our current MIN (also stored in a register), if it was less than min, we would update current value of min and branch back to the loop until the loop has iterated N times i.e. N has reached 0 after decrementing at each loop iteration.
Once we have updated the Max and Min value and have been through the whole list of data, we would subtract max from min and last use LSR (logical shift right) instruction by 2 on our current max-min which would shift the current value right by 2, equivalent to dividing an integer by 4. The value after the shift was stored in memory called RESULT. Here, the challenge was getting used to branching out of the loop to check for a condition, the instruction BGE (branch greater than or equal to) was used.

Centering

For this part of the lab, we were to ‘center’ an array, i.e. subtract the average of all values in the array from each element in the array. The challenge here was to figure out how to divide the sum of all the numbers by number of elements, ensuring the algorithm wasn't hard coded and worked for an array with an arbitrary number of elements. The way this was done was by continuously dividing both the sum of all the elements and the number of elements (N), by 2 in a loop. We would also continuously check

whether N=1 which was the condition to terminate the loop. For example, if the sum (S) was 16 and the number of elements (N) was 4, we would simultaneously divide both numbers by 2 (shifting by 1) until N was equal to 1. For this example, we would have after the first loop iteration S=8 and N=2, after the second loop iteration S=4 and N=1. Here N=1 therefore we already have the average which is our other value, S. After this was done, we subtracted the average from each value in the array using a loop.

Bubble sort

To sort the list of elements we used the bubble sort algorithm that was provided to us. This method involves having two loops to traverse the array. The inner loop is where the swapping of elements occurs, each element is compared to the previous one and swapped if it's smaller, then the indices advance by one. Once the array has been run through once the outer loop will increment. This sort finishes when an entire traversal of the array is performed and no swaps are made, we have a register, R4, that holds the 'sorted' status of the array, 1 for sorted, 0 for unsorted. This register is set to 1, sorted, every time the outer loop runs, and is only set back to 0, unsorted, when a swap is made. In a normal software program, swaps of two variables are generally made using a temporary variable to hold one of the values so that data is preserved. However, in ARM we can use the fact that registers and memory are distinct to perform swaps without the need for a temporary variable. Since we had to have the values we are comparing loaded into our registers already we simply store them in the other one's memory address. One of the first issues we came across was that our outer loop would only run once, all swaps were being made but only for one pass through. We realized that if we wanted to use the same registers to act as pointers to elements in the array we would have to reset them at the start of each iteration of the loop. Once that was fixed we ran the code and noticed that the entire memory was getting moved around and swapped, not just the array, even the random values were being compared. This was caused by the same issue; our pointers were not being reset so they were constantly incrementing through the memory. The easiest was we found to debug these errors was to look at the memory tab to see if the results were as expected and if not then implement breakpoints at crucial parts of the code to see what the interim values were. If even that did not clear up the source of the error we would resort to stepping into the code line by line to detect any anomalies. The final problem we had is that the last element would never get compared, we eventually realized that it was because the inner loop always terminated 1 element early, caused by an off-by-one error in our loop structure. Since we incremented out counter at the beginning of the loop the loop would terminate early. There are two ways to fix this, increase N by 1 for the purposes of the loop, or increment the loop at the tail-end, for simplicity we chose the first method.

Possible Improvements

It would have been useful to have an instruction in ARM that would divide 2 arbitrary numbers by each other, therefore, an algorithm wouldn't be needed to do so and centering an array would have been much simpler. We could utilize this in future labs now that we know how to implement this function and subroutines. Our code clarity could also be improved using aliases for register, being able to label them more descriptively than just R#.