Carlo D'Angelo 260803454
Mostafa Abdelwahab 260551876

Lab 1: Introduction to ARM Programming

Largest Integer Program

In this case, part1.s was already provided in the lab instructions (gets the maximum value in a list). It first loads in a register the memory address to the final result. It also loads another pointer to the first element in the list as well as the first element value itself. It stores the number of elements in the list.

It then goes through a loop to scan the elements in the list: by subtracting each iteration 1 from the number of elements in the list and comparing it to zero. It loads and compares the next element in the list to the maximum (initially set as the first element). If the new element is bigger, then the register containing the maximum takes the value of the new element.

Finally, it stores into the RESULT main memory address, the value of the maximum and moves to an infinite loop so we can view the values in the main memory.

Standard Deviation Program

Since the standard deviation needed to be computed using the "range rule", the code implemented in this part is very similar to the code seen in the Largest integer program. For instance, to find the maximum and minimum values of the signal, we essentially reused the code in part1.s. In fact, the overall process for finding both values (min/max) were almost the same, except for one main difference. Instead of using the operation BGE when finding the maximum value, we used the operation BLE when finding the minimum value. Both operations execute a branch but based on different current settings of the Condition Code flags (due to suffixes – GE vs. LE). In other words, after the CMP operation compares the current max/min value (loaded in a register) and another value in the list (loaded in another register), the program will branch back to the top of the loop only if the value loaded in the first register is still greater/smaller than or equal to the value loaded in the other register. Therefore, in the code, the max/min value will only be updated when the program skips this instruction (due to wrong Condition Code flags being set).

As for the division by 4, we initially used a Logical Shift Right (LSR) operation and set the shift parameter to #2 since *n* shifts to the right is equivalent to a division by 2^n. Although it worked in some cases, we learned that it is better to use the Arithmetic Shift Right (ASR) operation because it will replicate the sign bit as the shifts are done and, thus, maintain the sign of the value.

Centering Program

The code to "center" a signal is divided into three main parts, all of which make use of loops.

The first part uses a loop to simply find the sum of all elements in the given list.

The second part uses a loop to find the *n* of 2^n which, according to the lab instructions, was a necessary constraint on the signal length. In other words, it was assumed that the signal length would be a multiple of two (made it easier to calculate the average). By placing this sum value (loaded in a register) and this *n* value (loaded in another register) as parameters in the ASR instruction, we were able to calculate the average value of the signal.

The third part uses a loop to actually "center" the signal. Using the calculated average value of the signal, we loop through the list and subtract this average from every value in the list while storing the corresponding result in place.

Although this method ended up working, it was not very efficient. One way to improve this code is to combine parts 2 and 3 into one loop instead of doing two separate loops. This will improve the complexity of the code from O(3N) to O(2N) where N is the signal length.

Sorting Program

Here, sort.s follows the pseudo code provided in the lab instructions to sort elements in the list in ascending order using bubble sort.

It first loads to a register a pointer to the list length. It then loads to another register the value 0 (which will behave as the Boolean sorted being false, 1 being true). It then starts iterating an "outer" loop as long as the list is not sorted (i.e. sorted is false at the start of each iteration). It loads to a register the number of elements in the list and to another register the pointe to the first element.

It then moves to the inner loop that scans the remaining elements of the list. At each iteration, it compares 2 consecutive elements (loading them first from the main memory). If the second element is bigger than the first, the register of the left element stores the value of the right element for the following iteration for the next iteration. If the second element is bigger, we set the sorted Boolean as false and we switch the values in the main memory.

At the end, it moves to an infinite loop, so we can view the results in the main memory.

The best-case scenario is that if the n-element list is already sorted in an ascending order, there will 1 iteration of the outer loop containing (n-1) iterations in the inner loop without any STR instructions executed. The worst-case scenario is that if the list is n-element is sorted in a descending order: it will iterate the outer-loop (n-1) times each containing (n-1) inner-loop iterations. Each inner iteration executes 2 STR instructions.

If the n-element list is sorted in an ascending order (except for the last element, which is also the smallest). It will have the same number of iterations as the worst-case scenario but with only 2 STR instructions executed per outer-loop iteration (instead of 2n-2).

We can improve the code by inserting an IF statement before the beginning of the outer-loop: if the number of elements is zero or one, branch directly to the END infinite loop.