ECSE 324 Lab 1 Report Group 1

Authors: Aleksas Murauskas, 260718389 and Brendan Kellam, 260718806

1. Largest Integer

The Largest Integer deliverable was created by following the given instructions on MyCourses. The largest Integer is designed to search through an array and then the largest integer within the array is stored in a RESULT location in memory. The code first sets some key pointers; including the result, the members of the searched array, and the size of the array. The code loops through all members of an array sequentially, comparing the current member of the list with the currently held maximum. If it is not larger, the loop branch is called and the maximum remains unchanged. If it is larger, the maximum is replaced with the current array member the branch goes back to the loop. This loop continues until the variable R2, storing the length of the array, is decremented to 0.

This deliverable could be improved by introducing a sorting algorithm to sort the array, and then the maximum would simply be the first or last element. This would not be faster, as linear search completes in O(n) time whereas sorts take about O(nlogn), but if the array were needed again, sorting it in place could become helpful and decrease the time needed for other searches. In addition, we could improve the code by setting the initial max as the first member of the array instead of 0. If an array that held all negative values was searched through, the program would fail by storign 0 as the result.

2. Standard Deviation

This deliverable would store the standard deviation of an array,using a simplified version of the Standard Deviation Equation: $STD \approx (x_{max} - x_{min})/4$. We realized that the first step of finding the minimum and maximum could be easily achieved using similar code to that of Largest Integer. We had 4 key branches: LOOP, CHECKMAX, CHECKMIN, and DONE. LOOP is initially like it's counterpart in Largest Integer, a variable holding the length of the array [R2] is decremented with each call of the branch, calling DONE when R2 reaches 0. After a variable[R1] is assigned the current array member's value, the branch CHECKMAX is called. CHECKMAX will compare R1 and the maximum R0. If R0 is larger, no change is necessary and the code Branches to CHECKMIN. If, R1 is larger a new maximum is necessary and R0 is updated to hold the value R1 and

the code branches back to LOOP. CHECKMIN compares R1 with the minimum[R5]. If R1 is larger than R5, no change is necessary and branch back to LOOP. If R1 is smaller than R5, R5 takes on R1's value and the code branches back to LOOP. Once DONE is called, both the minimum and maximum have been found. R2 holds the value of the equation R0 subtracted by R5. Then R2 is divided by four, which is accomplished with a bit shift right by 2. The final result is stored at R4 in memory.

This deliverable could be improved similarly to Largest Integer. Sorting the array would make the time complexity of finding min and max of the sorted array O(1). As explained earlier this would not be faster if this is the only the time the array is interacted with, but would be useful the more the array is needed.

3. Centering Program

The Centering Program finds the average of a signal array, and then subtracts the average from every member of the array, functionally making the average 0. A prerequisite assumption we can make is that the signal's length will always be a power of 2. Our code has five key branches FINDSUM, LOG2, LOOP, AVG and CENTER. First FINDSUM operates by looping through the array and adding the member of the array to a sum. Once a variable[R0] initialized the size of the array is decremented to 0, the code branches to LOG2. Since we cannot use standard division, we must determine what amount of bits the sum found earlier must be divided by to find the average. The bit shift needed can be found using the equation: $Bits = Log_2(N).$ LOG2 sets the variables needed to determine what to bit shift the sum by, and LOOP executes by comparing the length of the signal [R0] with 1. A counter variable[R3] increments each time LOOP is called and R0 is bit shifted right by 1. The loop continues while R0-1 is greater than 0. After the loop terminates the R3 holds the values of bits needed to hold the length of the average, which is the same amount the sum will be shifted by to find the average. Then the code branches to AVG to compute the average. The sum is shifted by Log2(length) and stored in R0, then the code branches to CENTER. CENTER then increments through the array again, now subtracting each array member by the average stored in R0. The new value is then stored in the array members position. Once this has been done for every member, the program terminates.

4. Sorting Program

The sorting program uses the bubble sort algorithm to sort a set of N 32 bit (1 word) numbers in ascending order. We approached this problem by first implementing the algorithm in the C programming language to get a high-level

look at the problem and the general structure we would have to implement in assembly:

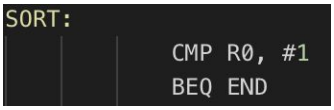
From this C program, we recognized there where 3 key components we would have to implement in assembly:

a. An outer loop that would iterate until a “sorted” flag was set to 1 (which would indicate the array is sorted).

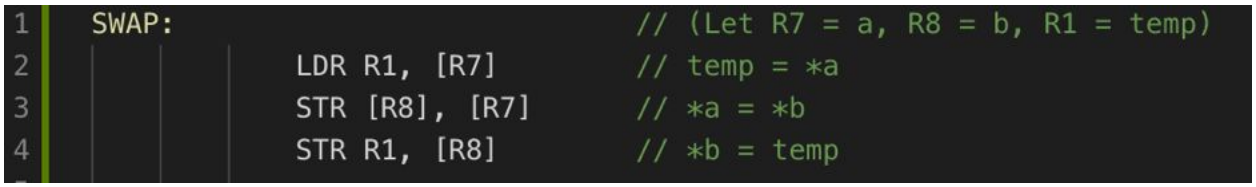
b. An inner loop that would iterate from the second element to the last, comparing the ith and (i-1)th pair to each other.

c. A swap sub-routine that could swap two values

(a) And (b) where quite trivial to implement and summarized here:

(a) while (sorted != 1)	 s.t R0 is the sorted flag [0: NOT SORTED, 1: SORTED]
(b) for (int i = 1; I < N; i++)	 s.t R2 is the number of elements in the list

(c) was allot trickier: we initially approached the swap sub-routine as a direct translation of the C function:



This assembler unfortunately generated errors for line 3. We found that due to ARM being RISC, the architecture did not support multiple memory accesses in a single instruction. We were attempting to get the value pointed to by R8 and then copy it to the location pointed to R7, thus requiring two memory accesses.

```
23          LDR R1, [R7]
24          LDR R3, [R8]
25          CMP R1, R3
26          BGT SWAP
```

```
32  SWAP:   STR R3, [R7]
33          STR R1, [R8]
34          MOV R0, #0
35          B INCR
```

Our solution was to break the statement up into two separate memory access operations: First load R1 and R3 with the values pointed to by R7 and R8 respectively and then, if a swap is needed, store R3 and R1 into R7 and R8 respectively.

We could have improved our sorting implementation by utilizing a quicker algorithm. Bubble sort runs at a comparatively slow speed (O(N^2)) compared to its highly efficient counterparts (Merge sort, quick sort, heap sort etc. all of which operate at O(Nlog(N)). The benefit of using quick sort is its simplicity: implementing it (especially in assembly) requires significantly less development time.