ECSE 324: Lab 1 Report

I. Largest integer

This program was provided to us at the start of the lab and was a good example of a tangible Assembly code with a purpose. It was also helpful because due to its lack of complexity, it allowed us to understand the process that was necessary for operations and blocks such as loops, incrementing, or storing a value to memory, which were all used in the subsequent codes.

II. Standard deviation

Given that the standard deviation "range rule" uses the notions of maximum and minimum value of a list of numbers, it was easy to see that the first code would be useful.

The first step of the standard deviation program being the computation of the maximum number, we essentially copied the relevant code from the part1.s file and made slight changes to it in terms of memory allocation space.

The second step was to compute the minimum number. This process, naturally, is almost identical to finding the maximum number, with the only change being the conditional instruction at the end of the loop. When computing the maximum number of a list, we define a variable that is our "temporary" maximum value, then loop through all the values of the list searching for a value larger than our temporary value. When computing the minimum, we also define a variable that will hold our temporary minimum value, however we update that value when we find a value smaller than our current minimum.

At this point in the code, we have therefore computed the required minimum and maximum values. We now need to subtract them (which is easily done with the SUB instruction), and finally divide by 4. The division by 4 was implemented with a shift instruction, shifting by 2 to the right, i.e. dividing by $2^2 = 4$.

III. Centering an array

The notion of centering an array means bringing its average back to 0. The way this is done is by calculating the average of the array, then subtracting that average value from every value in the array, effectively bringing that average down to 0. The operations used here are therefore: calculating an average (which is a sum and a division) and subtracting that average from every value (subtraction). The assumption that only signal lengths that are powers of two can be passed allows us to implement the division using shift instructions once again.

The first step is therefore to compute the sum of the array elements. This is simply done by assigning a memory space for the sum, and adding elements to that sum one by one with a loop.

The next step is to figure out what power of 2 the signal length is, to know by how much the sum variable should be shifted. This is done by dividing the signal length by 2 multiple times until it reaches a value of 1, and counting how many times we divided. For example, 8/2 = 4; 4/2 = 2; 2/2 = 1. We had to divide 8 by 2 three times to get to a value of 1, this tells us that $8 = 2^3$ and that to

divide by 8, we should shift to the right by 3. We therefore proceed with the shift with whatever power value was calculated.
Finally, we need to subtract that average value from all the array values. This is simply done by looping through the array and subtracting the stored average value.

IV. Sorting

This bubble sort implementation was the trickiest of the three programs we had to write, due to the presence of nested loops. It took us some time before properly understanding how we were to implement this.
The *nesting* loop is the while that keeps the program running until the array is successfully sorted. This is done by having a boolean variable that stores whether or not the array is sorted, and checking the value of that boolean at every iteration of the while loop. The boolean is set to true at every new iteration, and subsequently set to false if a swap was performed in the array. If no swaps are performed, the boolean variable remains true, and the current iteration is the last.
The *nested* loop is the for loop that checks every pair of numbers in the array, and checks for a pair where a number is larger than its next neighbour. This means the array is not sorted, and this leads to the two values being swapped (and the boolean variable to be set to false!). By swapping out of place pairs, we eventually bring every small number to the “left” (i.e. the beginning of the array) and every large number to the “right” (end of the array) (N.B. here “small” and “large” number are relative to the array).
The result is the array being fully sorted. This is not the most efficient sorting algorithm by far, but it is plenty enough for small arrays.