ECSE 324 Lab 1 Report

Group 16
Yuxiang Ma 260714506
Chris Li 260708306

1. Set Up the Tool

1.1 Description
The first part of the lab is designed for us to get familiar with the tool. The steps include creating a file, editing a file, configuring the project, loading the code and compiling the code.

1.2 Approach
Firstly, we followed the instruction to create a file with the given sample code (part1.s). The code is designed to find the maximum value in a given list. The code loads the memory location of result into R4, the number of elements in the list into R2, the memory location of the first element into R3 and the value of the first element into R0. The loop will compare all remaining elements in the list with the current maximum value (initially, the current max is the first element). If the current maximum value is smaller than the current element, the current element will be stored into R0 and become the current maximum value. After the loop, we can make sure all elements are traversed and the current maximum value must be the maximum value of the list. In the last step, We store the final maximum value into the memory location of the result.
After figuring out the logic of the code, we followed the steps to configure and execute the code. The code gave the correct answer 8. Then, we changed the N and NUMBERS and tested the code to ensure the code works well for all instances.

```
RESULT:     .word 0
N:          .word 7
NUMBERS:    .word 4, 5, 3, 6
            .word 1, 8, 2
```

	+0x0	+0x4	+0x8	+0xc
0x00000000	E59F4054	E5942004	E2843008	E5930000
0x00000010	E2522001	0A000005	E2833004	E5931000
0x00000020	E1500001	AAFFFFF9	E1A00001	EAFFFFF7
0x00000030	E5840000	EAFFFFFE	00000008	00000007

Given the test case, the result should be 8 and stored into 0x00000038. We got the expected result, so the correctness of the code is verified.

1.3 Possible Improvement
The sample code is concise and well-commented, so it is easy for us to follow. We learnt grammar by manipulating the code, which is very helpful. However, We can merge instructions of loading neighbour addresses to register using post or pre indexing to reduce number of lines.

1.4 Challenges
Because we do not have previous experience with assembly, we took some time to get used to its grammar. We overcame this by reading course slides and the instruction manuals.

2. Fast Standard Deviation Computation (stddev.s)

2.1 Description
The fast standard deviation computation calculates the maximum and minimum value in a list then divide the difference of them by 4 to find the approximate standard deviation. The calculation is straight forward and the key point is to find the maximum and minimum value efficiently.

2.2 Approach

To implement the function, we need to find the maximum and minimum value in the list. We followed the same logic as the sample code, but we had two registers to store the current maximum and current minimum separately. In the loop, we kept track both the current minimum and current minimum and updated them when necessary. By doing so, we could find both the maximum value and the minimum value in one loop. After the loop, the code will go to the DONE label to calculate the final result and save it.

```
RESULT: .word 0
N:      .word 5
NUM:    .word 10, 8, 6, 4, 2
```

0x00000050	00000002	00000005	0000000A	00000008
0x00000060	00000006	00000004	00000002	00000050
0x00000070	00000000	00000000	00000000	00000088
0x00000080	00000000	00000000	00000000	00000000

Given the test case, the result should be (10-2)/4=2 and stored into 0x00000050. We got the expected result, so the correctness of the code is verified.

2.3 Possible Improvement

We optimized the code by using one loop to find both minimum and maximum value in the list. Since we know the only situation that the minimum value equals the maximum value is when all elements in the list are same, we only need to compare the current element with the current minimum value if the current value is not greater than the current maximum value.

2.4 Challenges

There is no convenient division in the assembly as in high-level languages, so instead, we use bit shifting. ASR is the arithmetic shift right operation, and we use 'ASR R0, R0, #2' to achieve division of 4. The differences between the assembly and high-level languages should be noticed.

3. Centring an Array (center.s)

3.1 Description

The logic of the program is to find the average of a list and subtract the average from each element in the list to get a new list. Also, the operation is in place, so the updated value should be put back to the same memory location.

3.2 Approach

Firstly, we loaded the memory location of the result, the number of elements and the memory location of the first element in the list into registers. Then, we made a loop to sum up all the elements. In the loop, we kept updating current elements and the sum. We used a counter (R2) to track the loop processes. The average value was calculated with the ARS operation and stored into R5. We reset the loop counter and array counter to prepare for updating the list. The MAP label is the operations to subtract the average from each element and save updated elements back to corresponding memory locations.

```
RESULT: .word 0
N:      .word 8
NUMBER: .word 2, 9, 11, 5, 7, 6, 8, 7
```

0x00000060	00000008	FFFFFFFC	00000003	00000005
0x00000070	FFFFFFFF	00000001	00000000	00000002
0x00000080	00000001	0000005C	00000000	00000000
0x00000090	00000000	000000A0	00000000	00000000

Given the test case, the result should be a list of -4, 3, 5, -1, 1, 0, 2, 1 and stored into 0x00000064 to 0x00000080. We got the expected result, so the correctness of the code is verified.

3.3 Possible Improvement

We tried to make the code concise and efficient, but we think it is not possible to finish all operations in one loop because the update is based on the average value. We had a memory location labelled RESULT, but it is not necessary if we do not need to store the average value.

3.4 Challenges

To update elements in place can be challenging, as we need to start over to traverse the list. We reset the counter and the array pointer to solve this problem.

4. Sorting(sort.s)

4.1 Description
This part we implemented the given sorting algorithm. We keep swapping elements until the list is ordered. There is a sentinel to track the status of the list, so the loop will not run infinitely.

4.2 Approach
Firstly, we loaded the number of elements in the list and used R0 to store variable ‘sorted’ (0 represents not sorted). Then, we nested a for-loop into the while-loop to write the program. In the for-loop, we compared the variable with 0 to see if the list is sorted or not. If the list is already sorted, we can finish the loop; if not, we reset the counter and the array pointer and go to the for-loop. In the for loop, we went from N down to 2 and compared elements two by two. If the current one is smaller than the previous one, we swap them.

```
N:      .word 8
NUM:    .word 9, 5, 4, 3, 2, -1, -99, -100
```

0x00000060	FFFFFF9C	FFFFFF9D	FFFFFFFF	00000002
0x00000070	00000003	00000004	00000005	00000009
0x00000080	0000005C	00000000	00000000	00000000
0x00000090	00000098	00000000	49FD1FDD	FE19D7DB

Given the test case, the result should be a list of -100, -99, -1, 2, 3, 4, 5, 9 and stored into 0x00000060 to 0x0000007c. We got the expected result, so the correctness of the code is verified.

4.3 Possible Improvement
In the for loop, we changed the pseudo code and made the loop from N down to 2. This alternative method gives an identical result, but in terms of efficiency, we need further observation.

4.4 Challenges
The implementation of this part is more complex than previous programs, so we took longer to make the pseudo code to assembly. Also, there are three branches in the for-loop label, so we need to truly understand the logic behind.