**Answers to the questions asked in the manual** **Artem Moshnin (A0266642Y) | Li Zheng Ting (A252496W)**

**Question 1: How to access the elements in the array from the asm_fun? Given the N-th element's address is X, what would be the address of the (N+k)-th element's address?**

We know that (memory address) of (nth element) => X
Hence, (memory address) of ((n+k)th element) => X + 4k
Because, in a 32-bit architecture, each element is made up of 32 bits = 4 bytes, hence, we must shift the (memory address of (N)th element) by (4*k bytes) to get (address of (N+k)th element).

**Question 2: Describe what happens with and without PUSH {R14} and POP {R14}, explain why there is a difference.**

In our code, we need to utilise PUSH {R14} and POP {R14} in order to keep track of the values of the Link Register (LR) when utilising BL and BX. If we use BL Subroutine and another Branch with link within Subroutine, we will override the link register with the second BL address. By using Push and Pop, we are able to save the memory address from the first BL so we can exit Subroutine to the correct line of code.

**Question 3: What can you do if you have used up all the general purpose registers and you need to store some more values during processing?**

If we have used up all the general purpose registers and we still need to store some more values during processing, then for this purpose we can utilise the Stack to save and restore registers as needed. Meaning that we can push the values of registers onto the stack before overwriting them & pop them back when needed with instructions PUSH {R1} and POP {R1}.

```
PUSH {R4, R5, R6}  @ Push registers R4, R5, and R6 onto the stack
POP {R4, R5, R6}   @ Pop the values back into registers
```

**Discussions of your program logic (do not explain your code line by line)**

In this report, we will be frequently referring to the next unsorted number in the array as "new number"
First, we initialised a few registers with starting values. R0 to R5 and R8 will be used for our code. Here are the functions for each register:

Initial:
- **R0: (memory address)** stores the (memory address) of (1st element) in the (input array)
- **R1: (memory address)** stores the (number of elements) in the (input array) (= length of the input array)

Subsequent:
- **R0: (Decimal)** Counter for number of swaps. Returned as output to the C code.
- **R1: (Decimal)** Counter to check number of numbers left in the array. It is used at the start of every loop to check whether the program code has finished sorting the entire array. Each time a number is in its correct position, 1 is subtracted from R1, and once R1 reaches 0, it will branch back to the line of code after "BL SUBROUTINE" to end the code and return the output to C.
- **R2 & R3: (Decimal)** Used to store adjacent numbers for comparison to determine if they need to be swapped.
    - R2 => (left number of array) when comparing (two adjacent numbers)
    - R3 => (right number of array) when comparing (two adjacent numbers)
- **R4: (memory address)** Stores memory address of the "new number", so we can jump to it once finished sorting previous numbers. Jump is implemented by replacing the current address stored in R8 with R4.
- **R5: (memory address)** is a constant that stores the memory address of the first element in the array. It is used by the code to understand that it has reached the start of the array by comparing with R8 (when swapping a number towards the start of the array) (this must be smallest number so far ordered) and then knowing that it have positioned the current number into its correct position (in this case first in the array), it will jump back to compare the next value (by performing R8=R4). This prevents us from checking contents stored in the memory outside our array (from the left side).
- **R8: (memory address)** Register that stores the memory address of the number that we are currently comparing to the one at [R8, #-4].

Next, we enter a loop that iterates through the array while performing insertion sort on required numbers. Within the loop, we included code to check for certain conditions to break the loop early:

- We compared R8 with the constant stored in R5, to (end the current loop cycle and move on to the next iteration if the number that we are currently checking (at [R8]) is at the memory location of the first element in the array (at [R5]). This is needed because we compare (value stored at [R8]) with (value stored at [R8, #-4]). We want to prevent the program from comparing the value stored in a memory address location outside of the array.
- Every time we subtracted #1 from R1, we check if R1 is equal to #0. If equal, we know that we have finished iterating through the array and we can return the output to C
- If both conditions above are not satisfied, the core program will continue running indefinitely. The next part will explain the function of the core code.

This next part of the program will perform two major actions, one to check adjacent numbers and one to swap the two numbers if needed. Both actions branch back to the start of the loop after performing their respective functions. We first load the number stored in the memory location in [R8] and that stored in [R8, #-4] into R2 and R3 respectively. We then perform a compare operation on the two registers.

- If the number in R3 is greater than or equal to that in R2 (R3 >= R2), we will move on to check the next unchecked number in the array. We achieve this by incrementing our pointer “R4” by one word, and move that memory address to R8 and branch back to the start of the loop. This way, we can jump forward to the middle of the array without having to loop through the numbers that we have checked. Simultaneously, we decrease the counter “R1” by one. We then branch to the start of the loop
- If the number in R3 is smaller than that in R2 (R3 < R2), we will branch to the “swap” function to swap the two numbers. If this function is run, we will also increment the swap counter “R0” by one. We then branch back to the start of the loop.

Once the end condition is met (R1 == 0), we immediately stop the code by branching to the value stored in the link register, effectively ending the assembly_sort & returning the output to C.

## <u>Discussions of the improvement you have done or can be done in the future</u>

### Iteration (#1)

The first version of our code didn’t utilise the provided (length of the array in R1) and instead would stop iterating (from left side whenever it reaches a hard-coded value of an empty space) and (from right side whenever the index we kept track of would reach a value less than 0).
The main disadvantage of this approach was the fact that when we would swap a number downwards to its correct location in the ordered array, we would have to reiterate backwards one-by-one through already sorted values (which would just be a waste of resources) without performing direct jumping to the next potentially unsorted numbers.

### Iteration (#2)

We made substantial changes to the shortcomings of the first version, and utilised the register R4 to store the memory address of the “new number”. We strategically placed this new block of code to run it after we have either successfully swapped a number to its correct position in the sorted array thus far, or if that number does not need to be swapped. Both conditions signal that the current iteration of the swap is complete and we can move on to the “new number”. However, we noticed another problem when we have to swap small numbers. We did not have a check for the start of the array, so if the memory location right before the array contains a number greater than the number that we are sorting, we would sort past the start of the array. This is an undesirable outcome that causes the program to behave erratically in some cases.

**Iteration (#3)**
In this subsequent iteration our aim was to improve our code's ability to finally perform insertion sort with 100% correctness while at the same time attempting to maximise efficiency of our code. In the previous iteration, we had an edge-case bug when:

- (we're sorting downwards a small number into its correct position within the sorted array)
- & (turns out to be the smallest number in our sorted array so far (so moved towards first position of array)
- & the memory location right before the array contains a number greater than the number that we're sorting.

then, the previous program would begin sorting past the start of the array, making the result of the sorting completely wrong.

Therefore, to prevent this undesirable behaviour, we've firstly saved the (address of the first position in the array into R5) and then use it to perform a conditional check in the code on every loop that checks whether the (currently looked element in the array = R8) is equal to (first element in the array = R5), if satisfied, then we move R8 to the next element in the array and branch back to loop after which this condition will not get satisfied and following that, the checks for swap will see that the number are correctly sorted and then jump back to the next number in the unsorted array to sort.

Hence, via this edge-case check we prevented our algorithm from checking outside of the array. Specifically this block of code at the start of the 'check' label have enabled us to perform this checking functionality:

```
CMP R5, R8
        ITT EQ
        ADDEQ R8, #4
        BEQ loop
```

As you can see in the above block of code, we compare whether (R8 equal to R5), if they're then we prevent from going outside of the array by moving R4 onto the next number (R8=R8+4) and branching back to the 'loop' label, which will then jump to checking the next "new number".

**Iteration (#4)**
In this final version of the iteration, we performed some optimisations of the code.
In previous iterations, we included too many BL (branching with link), most of the branching happened within nested loops. As a result, we had to use PUSH {R14} and POP {R14} many times. We realised that some branches are redundant and after careful examination of the logic flow within the program, we managed to shorten the code efficiently, and only branch when needed. Another optimisation we implemented to iteration #3 was the removal of a redundant loop.In the previous iteration of our code, when facing the edge-case that we've explained in the paragraph about iteration (3), we had the functionality that first increments the address of R8, then branch to 'loop' and only after successfully skipping the various checks and loading we will then jump back to the next unsorted number - however, this behaviour will always be same in this scenario and therefore in order to avoid performing these unnecessary operations (various checks and loadings) we can directly move on to the next unsorted number (because know for a fact that if this condition is met, then the sorted array is fully sorted and we can move onto the next new unsorted number). To illustrate visually this specific modified block of code:
- In Iteration (3) it looked like that:

```
loop:
        CMP R1, #0
        BEQ fin
        BL check
        MOV R8, R4
        B loop
check:
        CMP R5, R8
        ITT EQ
        ADDEQ R8, #4
        BEQ loop
        LDR R2, [R8, #-4]
        LDR R3, [R8]
        CMP R3, R2
        BLT swap
        SUB R1, #1
        ADD R4, #4
        BX LR
```

- In Iteration (4) it looked like that:

```
        CMP R5, R8
        ITTT EQ
        SUBEQ R1, #1
        ADDEQ R4, #4
        MOVEQ R8, R4
```

As you can see, in the Iteration (4) we directly move on to (sorting the next unsorted number in the array) instead of (branching back to the ‘loop’ label to perform unnecessary checks & loads). Hence, optimising the efficiency and performance of our program.

With these optimisations implemented into our code, we believe that we have successfully used most aspects of what we have learnt in the past 6 weeks into our assignment. However there can still be certain improvements that can be made to the code. For example, in this final iteration of the code, we still included a 2-line part of the code that repeats itself in 2 different areas of the code. Specifically, it is this part:

```
        CMP R5, R8
        ITTT EQ
        SUBEQ R1, #1
        ADDEQ R4, #4
        MOVEQ R8, R4
```

**AND**

```
        SUB R1, #1
        ADD R4, #4
```

To improve the code readability and maintainability, we could theoretically encapsulate these two lines of code that is repeated throughout our code twice, however, this would involve utilising (BL and BX) operations at least twice, and therefore utilising the stack to PUSH and POP the (Register 14) to avoid losing the previous values, so we have decided that doing so would negatively trade-off the performance of our code and so decided to avoid this refactoring.

## Whatever efforts you have done and you want to let the teaching team know

After producing a code that managed to sort the ordered array in iteration #1, we decided that we were not satisfied with the capability of the current iteration and put effort into ensuring that it is maximally optimised for performance and ensure that the assembly code can be called from any variation of C code (because our most optimal solution in iteration 4 doesn’t rely on any hard-coded values as it does in iteration 1). We did not utilise the register R1 that provides us the number of elements in the array. As a result, we had to use a hard-coded value as the end-condition for the program to know when to stop and return its output to C. We thought that this approach with hard-coded value could potentially be not scalable to all the different ARM 32-bit microcontrollers and therefore decided to utilise the register R1 instead. Secondly, initially our program was performing redundant loops as our code had to cycle through already checked numbers after sorting a number to, for example, the start of the array. Therefore, to further improve the performance of our program we’ve added a secondary pointer and slightly changed the logic of our program such that redundancy does not happen.

Once we achieved the version of code we had sought after, we added comments to each line of the code to enable viewers to understand the function of each line more comprehensively. This helps us to keep track of the code easier as well.