National University of Singapore

# Assignment 1

## Group members:

Ong Wei Heng [A0235044N]
Rani Dilipkumar Shiva Shankar [A0235167A]

**Question 1: How to access the elements in an 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 can load the elements from the array’s first memory address into a register.
LDR R2, [R0]

The address for the (N+k)-th element is (X+k*4).

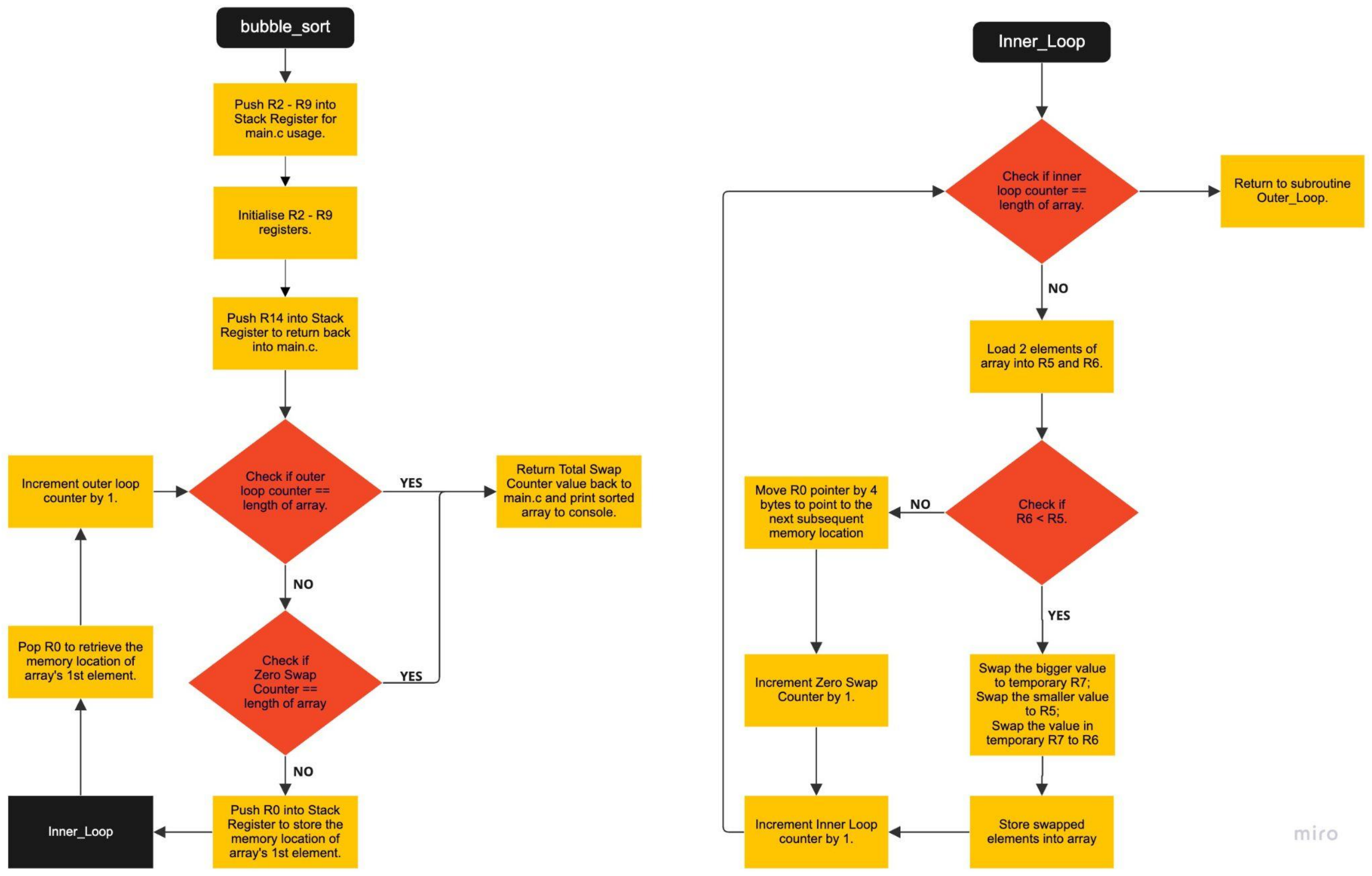
**Question 2: Describe what happens with and without PUSH and POP {R14}, explain why there is a difference.**

When executing a subroutine, registers are being used for the subroutine. Therefore, the PUSH and POP instruction allows the register to restore its original contents. In this example, the R14 (Link Register) is modified when the main.c program executes the bubble_sort.s sorting function. During the execution of bubble_sort.s R14 is being modified, without the PUSH and POP {R14} instruction the bubble_sort.s would not be able to return correctly to the main.c after it has completed the sorting function, as R14 has lost its address for linking back to main.c

**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?**

Use and re-use the registers in a systematic way. We can do so by doing a data table to record any registers that can be or should not be modified. By implementing PUSH and POP we can also retain the original data in the registers.

## Discussion of program logic using flowchart:


bubble_sort
Push R2 - R9 into Stack Register for main.c usage.
Initialise R2 - R9 registers.
Push R14 into Stack Register to return back into main.c.
Check if outer loop counter == length of array.
YES
Return Total Swap Counter value back to main.c and print sorted array to console.
NO
Check if Zero Swap Counter == length of array
YES
NO
Push R0 into Stack Register to store the memory location of array's 1st element.
Inner_Loop
Pop R0 to retrieve the memory location of array's 1st element.
Increment outer loop counter by 1.
Inner_Loop
Check if inner loop counter == length of array.
Return to subroutine Outer_Loop.
NO
Load 2 elements of array into R5 and R6.
Check if R6 < R5.
NO
Move R0 pointer by 4 bytes to point to the next subsequent memory location
YES
Swap the bigger value to temporary R7; Swap the smaller value to R5; Swap the value in temporary R7 to R6
Increment Zero Swap Counter by 1.
Store swapped elements into array
Increment Inner Loop counter by 1.
miro

**Improvements done:**

We have implemented a ‘0 swap counter’ using R9. This register will capture the number of no swaps in the array, and we will be comparing the counts with the length of the array. This can ensure that the array is fully sorted. It also allows bubble_sort.s to exit faster back into main.c without having to complete all the outer loop.

**Future improvements:**

To allow faster sorting algorithms we can consider other methods rather than bubble-sort. As it can be seen in our programme we are using two loops to complete the array sort, this inherently increases the time complexity of the programme to complete the array swap. For a simple five elements array we have to loop a total of $5^2$ times and the number of loops increases with more elements. Hence for larger arrays it will take a longer time due to the number of loops to be completed.

Another possible improvement is to use Merge sort. Merge sort uses divide and conquer approach to divide the array into smaller sub arrays. Perform recursion and sort the smaller sub arrays, finally combine the smaller sub arrays to get the sorted array. The time complexity of Merge Sort is O(Nlog(N)) in all 3 cases (worst, average, and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.