EE2028 Group 19 Assignment 1 Report

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 using the following instruction `LDR R2, [R0].`

The address for the (N+k)-th element is (X+(k-1)*4).

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

When executing a subroutine, the program uses registers for loading and storing its own contents. 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?

We shall highlight 2 possible approaches in the event that one has used up all the general purpose registers and needs to store some more values during processing.

The first possible approach is to use and re-use the registers in a systematic way. We can do so by implementing 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.

The second possible approach is to create a user-defined stack. In a user-defined stack, the memory address in a register (for example, R1) is decremented by 4 bytes and the value is stored in that memory location (R1 - 4) and the R1 is updated to hold the memory location (R1 - 4). Subsequently, if one needs to store more elements, the register R1 can be decremented by 4 bytes and the value will be stored at the memory location of (R1 - 8) and the R1 is updated to

hold the memory location (R1 - 8). This demonstrates a full descending user-defined stack operation. It is a full stack meaning the stack pointer (R1) points to the most recent item in the stack, i.e the location of the last item pushed into the stack. A full descending stack grows downwards meaning R1 starts at a high memory address and as items are pushed onto it, the stack pointer progresses to lower memory addresses.

Improvements done:

We have implemented a “0 swap counter” using R7. The number of 0 swaps in the array will be tracked in this register, and the length of the array will be compared with the "0 swap counter". This will make sure that once an array is completely sorted, bubble sort.s will end and enter main.c right away, and allow the sorting process to end faster.

We reduced 2 registers, from 8 to 6 registers used in total. Firstly, we removed one register which was used as a temporary location to hold the values, now we store the values directly into the memory location instead of storing in a register, thereby saving one register. Secondly, we used 2 different registers for our subroutine loop counter, however by pushing and popping R2 in the stack, we were able to reuse R2 in 2 different subroutines and hereby reducing one register required to count both loops.

Future improvements:

We can take different approaches into consideration instead of bubble-sort to provide quicker sorting algorithms. As can be observed, our software uses two loops to finish the array sort, which inevitably makes the program more time-consuming to execute the array swap. For a simple five elements array we have to loop a total of 5^2 times in the worst case scenario and the number of loops increases with more elements in the array. Hence, for larger arrays it will take a longer time due to the nature of the algorithm.

One 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.

Further Discussion:

We followed the philosophy as described in the Zen of Python. For example,

```
Explicit is better than implicit.
Readability counts.
Beautiful is better than ugly.
```

We developed the code such that it should be understandable to everyone, including those with little or no prior assembly knowledge. We avoided using confusing function names to conceal code functionality and kept our code as simple as we could. Our focus was to write code which is simple to perceive, comprehend, or interpret. We also included comments for almost each line detailing the purpose of each instruction.

```
Simple is better than complex.
Complex is better than complicated.
```

We divided the implementation into 2 subroutines OUTER_LOOP and INNER_LOOP. This enables us to debug and test each subroutine easily. In conclusion, we closely followed industry standard’s practical guidelines and coding best practices so as to develop a robust codebase

Discussion of program logic using flowchart: