ISTANBUL TECHNICAL UNIVERSITY
COMPUTER ENGINEERING DEPARTMENT

BLG 212E - Microprocessor Systems
Homework 2

Name - ITU ID : Kemal Tahir Bıcılıoğlu - 150210083
ITU mail : bicilioglu21@itu.edu.tr

FALL 2024

Contents

1 INTRODUCTION

→ We are asked to implement a bubble sort algorithm in Arm Cortex M0+ assembly language and compare the time elapsed measured by the interrupts with the previously implemented merge sort for the time complexity analysis. Default configurations give us the variables defined in the memory as a linked list structure consisting of num and next, and the given merge sort is sorting those variables by changing the next pointers of the nodes. Additionally, the default configuration can measure the time elapsed for the merge sort algorithm using a timing.c file that utilizes the system core clock and counts the interrupt cycles in every 10 microseconds. Similarly, we are asked to fill the timing_asm.s file and use our handler function for the interrupts by deleting the given handler function and removing comment in the assembly file.

2 METHODS

In this part of the report, I will explain the functions I implemented detaily. First the timer assembly file and the bubble sort assembly file. Lastly the testing methods I used for the homework.

2.1 System Timer

The Figure 2 given in the homework document shows the registers that should be changed to enable the interrupt with the correct time intervals.

- **Systick Start** assembly function should first clear the ticks value, then should write 249 to the LOAD register because of the formula:

$$\text{Reload value} = \frac{\text{Time interval}}{\text{Clock period}} - 1$$

$$\text{where} \quad \text{Time interval} = 10\,\mu\text{s}, \quad \text{Clock frequency} = 25\,\text{MHz}$$

  I found the Clock frequency by looking at the SystemCoreClock defined in system_ARMCM0plus.c and the time interval by looking at what the default configuration of the homework is using as a time interval for the interrupt. Next it should clear Systic Val register to 0 and it should enable the Control register by moving 7 (0b111) to the register.

- **Systick handler** assembly function will be called when an interrupt occurs so we should basically increment the tick value in that function.

- **Systick stop** assembly function should disable the interrupt control register, interrupt load, and interrupt value. To be able to disable the interrupt, we should basically load 0 to all the registers including the ticks variable excluding the value register.

Since we are counting the ticks in every 10μs we cannot precisely measure the time in micro units. To do that we can use the value register also. Incrementing the value register 250 times gives 10 microseconds so 1 time increment equals $10/250\mu$s which is 0.4μs

2.2 Sorting Algorithm

We are asked to implement the bubble sort algorithm. We are given 2 parameters, first one is the address of aux which keeps the head of the linked list and the second one is the function pointer for the function which compares two given values.

- **The main logic** of the bubble sort algorithm I implemented is keeping a register which indicates if a swap operation is done in the current loop. If no swap operation is done, it means that the linked list is sorted.

- **Swap logic** of the nodes is keeping 3 addresses for the 3 nodes which are previous, current and the next nodes. We should make the next of the current as current→next→next. Since we are going to also make the next of the next node as the current, we should not lose the current→next→next and keep in another register. After that we should also update the previous node's next as the next node.

- I provided an image which shows how the next pointers should be after swap operation in Figure 3 and Figure 4.

- The tricky parts about arranging the next pointers was as the following. If the swap operation needed for the first node, we should also be aware that head pointer should be changed. If swap does not occur, we can easily keep continue the loop by making previous = previous→next, current = current→next. However if swap occurs, current node should be also the current node and previous should be the next node for the next iteration to maintain the loop. One can observe this by looking the figures I provided. After the sorting is done, aux value should be changed to the head since the head of the linked list might have been changed.

- There are 2 ways to optimize the bubble sort and I did both of them. First one is keeping a swapped flag and if the inner loop did not swap any element we finish sorting. Second one is not iterating all the elements in the inner loop since bubble

sort puts the i'th max element in the i'th iteration. So when the i'th iteration is done, we no longer need to check the last index so we iterate until $size - i - 1$ for i'th outer loop iteration in the inner loop.

- I used the utility function provided named **ft_lstsize** to find the linked list size.

2.3 BigO Analysis

The following table shows the measured time data for both merge and bubble sort.

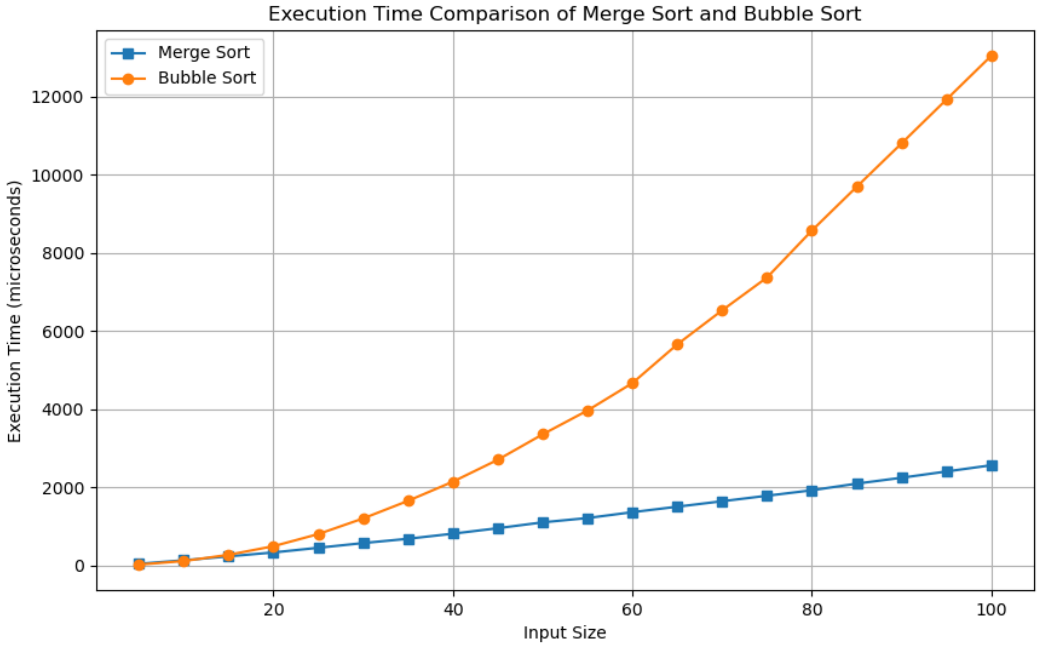

Figure 1: Bubble Sort Merge Sort Graphic

One can see the actual measured time from the Figure 5. Also I provided an example debug image in the Figure 6

1. Bubble Sort
   (a) Time Complexity
      i. Best Case: $\Omega(n)$
      ii. Average Case: $\theta(n^2)$
      iii. Worst Case: $\mathcal{O}(n^2)$
      iv. Space Complexity: $\mathcal{O}(1)$

(b) The number of iterations can be calculated as the following for the worst-case scenario:

$$n + (n-1) + (n-2) + \cdots + 1 = \frac{n(n+1)}{2}$$

This case occurs when the data is sorted in the reverse way and there is always a swap in all the inner loops. The multiplication gives n^2 which makes the worst-case time complexity $\mathcal{O}(n^2)$. The best case is $\Omega(n)$ which occurs when the linked list is already sorted. It achieves this by keeping a swapped flag. Also bubble sort gives reasonable time for nearly sorted data since it holds a swapped flag and whenever the data is sorted at any time in the iteration, it will finish the algorithm.

2. Merge Sort

   (a) Time Complexity

      i. Best Case: $\Omega(n \log n)$

      ii. Average Case: $\theta(n \log n)$

      iii. Worst Case: $\mathcal{O}(n \log n)$

      iv. Space Complexity: $\mathcal{O}(n)$

   (b) Merge sort is $\mathcal{O}(n \log n)$ algorithm since it uses divide and conquer strategy. It basically divides the array in half until the dividend array size is 1, then merges the divided subarrays accordingly. Because we always do the same process in merge sort, the time complexities for best, average and worst case is the same.

2.4 Testing

I write a C function which takes a parameter of a linked list's head, and returns 1 if the linked list is sorted otherwise 0. I passed the aux parameter after the sort is done to my function to see if it is sorted. It can be seen from the debug screen example in the Figure 6. is_sorted variable is the return value for the merge sort, is_sorted_asm is for my implementation of linked list bubble sort tested by my function.

3 RESULTS

- Systick registers given in the homework pdf:

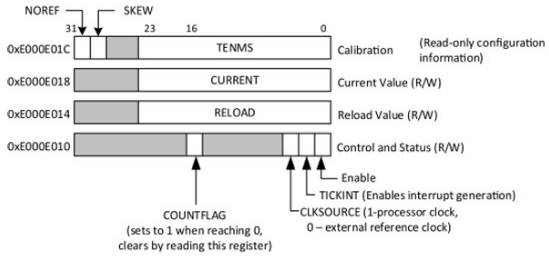

Figure 2: SysTick Registers

■ Bubble Sort swapping logic explained visually:

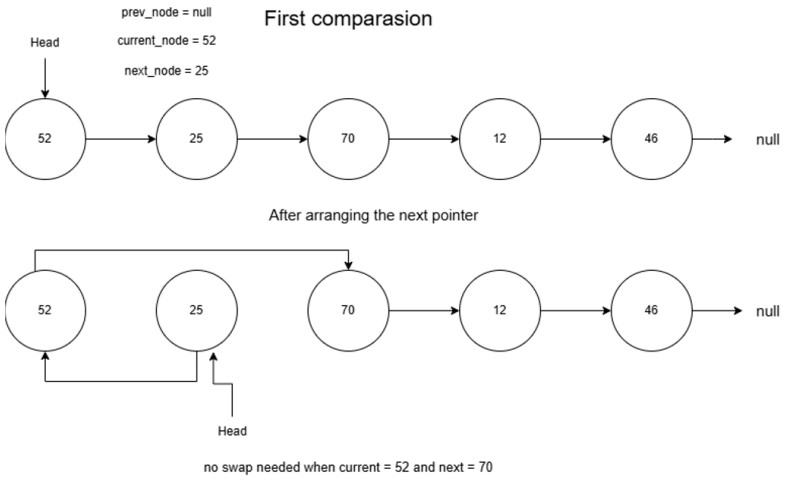

Figure 3: Bubble Sort Example 1

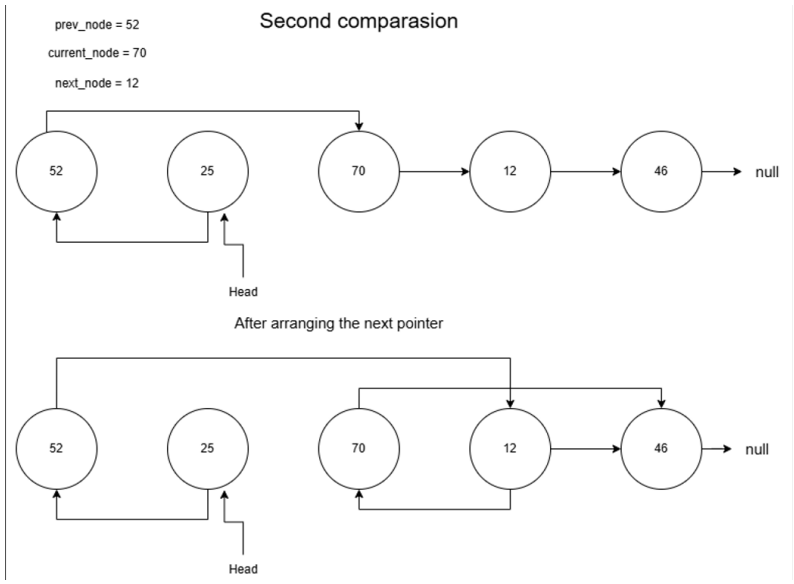

Figure 4: Bubble Sort Example 2

■ Time measured data for the sorting algorithms:

SIZE/ micro seconds	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100
merge sort	50	140	240	340	460	580	690	820	960	1110	1220	1370	1510	1650	1790	1930	2100	2250	2410	2570
bubble sort	30	120	280	500	810	1210	1660	2150	2710	3360	3970	4670	5660	6530	7370	8570	9690	10800	11920	13040

Figure 5: Bubble Sort Merge Sort Time Measured

■ Example figure taken from the debugging screen:

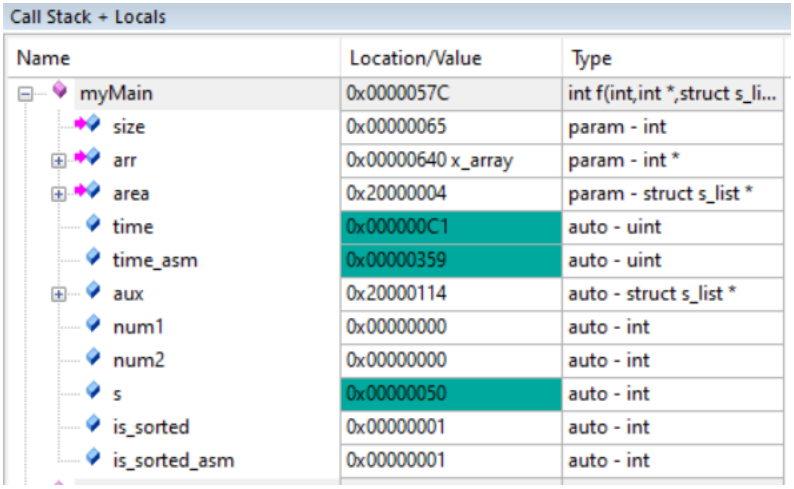

Figure 6: Debug Screen