Assignment 1: Implementation of Merge Sort in ARM Cortex-M0+ Assembly

1 Assignment Description

In this assignment, students are required to develop a single assembly program that performs sorting on an unsorted array using the merge sort algorithm. The merge sort algorithm should be examined in detail from the following reference:
`https://www.geeksforgeeks.org/dsa/merge-sort/`.

The program must:

- Implement the merge sort algorithm using a recursive approach.
- Be fully written in ARM Cortex-M0+ assembly language.
- Follow the Thumb-1 instruction set constraints.
- Produce a sorted array in ascending order.

2 Program Requirements

The program must compile and run successfully on **Keil μVision IDE v5** using the default project configuration. Every line of assembly code must include a clear explanatory comment; lines without comments will not be accepted.

2.1 Part 1

The test cases must contain exactly **five unsorted elements**. Initially, these unsorted elements should be placed in registers `R0--R4`. The program should then recursively apply the *divide and conquer* strategy to split the array into two halves until each sub-array contains only one element. Once the recursive division is complete, the elements must be merged in sorted order.

An illustrative example is provided in Figure 1. After sorting is completed, the resulting sorted values must again be stored in registers `R0--R4`. Refer to Table 1 for the expected placement of values.

Please write the requirements for Part 1 into the `part1.s` file in the requested format.

As shown in Table 2, the register values update gradually during each merge operation. Each time a merge operation occurs, the registers should appear as follows:

Table 1: Unsorted and sorted values in registers R0–R4

Register	Unsorted (decimal/hex)	Sorted (decimal/hex)
R0	38 / 0x26	10 / 0x0A
R1	27 / 0x1B	27 / 0x1B
R2	43 / 0x2B	38 / 0x26
R3	10 / 0x0A	43 / 0x2B
R4	55 / 0x37	55 / 0x37

Iteration	R0	R1	R2	R3	R4
0	27	38	43	10	55
1	27	38	43	10	55
2	27	38	43	10	55
3	27	38	43	10	55
4	27	38	43	10	55
5	10	38	43	27	55
6	10	27	43	38	55
7	10	27	38	43	55
8	10	27	38	43	55
9	10	27	38	43	55

Table 2: Merge sort register states (R0–R4) at each iteration

2.2 Part 2

At the beginning of the program, **the 8-element array** `(13, 27, 10, 7, 22, 56, 28, 2)` must be placed in memory, and its base address must be loaded into register **R7**. You are required to sort this array using the Merge Sort algorithm. During the Merge Sort procedure, all partial merge results must be temporarily stored in **R4**. For example, when merging `[10, 27]` with `[13]`, the intermediate merged sequence `[10, 13, 27]` should be written into **R4** before continuing the algorithm.

After the Merge Sort process is fully completed, the final sorted array must be stored across registers **R0**–**R7**. Accordingly, the final register assignment becomes **R0=2, R1=7, R2=10, R3=13, R4=22, R5=27, R6=28, R7=56**.

Please write the requirements for Part 2 into the `part2.s` file in the requested format.

Warnings

- **Every part of your code must include a clear explanation.** Any code submitted without accompanying explanations will not be accepted.

- **Function names and label names must exactly match the required reference format and order: `main` (the main function), `My_MergeSort` (the divide function), and `My_Merge` (the merge function).** The naming convention must follow the structure illustrated in the reference diagram (see Figure 2).

- **All comments must be written in your own words.** Reusing or paraphrasing comments from the sample code or from any other student will be treated as duplicated work.

- **Submitting code that is identical or highly similar to that of a classmate is strictly prohibited.** Register usage, function flow, memory handling, and overall structure must be entirely original for each student.

- **Please write the requirements for Part 1 into the `part1.s` and Part 2 into the `part2.s` files in the requested format.**

3 Merge Sort Assembly Grading Rubric

Table 3: First Part Merge Sort Assembly Rubric (Total: 60 points)

Divide Phase (Recursive Splitting)	10	The student must correctly implement the divide phase: computing the midpoint, splitting the array into two halves, and making recursive calls for the left and right segments. This phase must correctly structure the recursion tree without performing merging.
Conquer Phase (Merge Logic)	15	The student must properly merge two sorted subarrays: computing the sizes, creating temporary buffers, copying the elements, comparing items from each side, inserting in correct order, and handling remaining elements.
Stack Frame and Function Organization	10	Correct preservation of registers, proper PUSH/POP usage, correct LR handling, stack pointer adjustments, and clean functional structure.
Temporary Buffer Usage	10	The merge routine must allocate and use a temporary buffer (stack-based or equivalent), with correct offset arithmetic and accurate handling of buffer boundaries.
Base Case	10	Correct termination condition (e.g., left $\geq$ right) to prevent unnecessary recursion or invalid memory access.
Register Requirement (R0–R4)	5	Initial unsorted values must be placed in registers R0–R4, and the final sorted results must again be returned in R0–R4 after completion.
Total	**60**	

Table 4: Second Part Merge Sort Assembly Rubric (Total: 40 points)

Divide Phase (Recursive Splitting)	10	Correct implementation of the recursive divide step: computing the midpoint, splitting the RAM-based array `arrayB` into two segments, and making recursive calls on both halves without performing merge operations.
Conquer Phase (Merge Logic)	15	Proper implementation of the merge logic in `my_Merge`: loading elements, comparing values, writing them in ascending order, and draining the leftover values from both subarrays.
Stack Frame, Registers, and Function Organization	5	Correct use of `PUSH`/`POP`, restoring `LR`, adjusting `SP`, and maintaining clear separation between sorting and merging routines.
Temporary Buffer Usage	5	Correct stack allocation of the temporary buffer, accurate index/offset calculations (e.g., `TMP_OFF`), and safe access within the allocated bounds.
Base Case Handling and Final Output	5	Correct handling of the base case ($\texttt{p} \geq \texttt{r}$) and correct loading of the final sorted `arrayB` values into `R0--R7` for verification.
Total	**40**	

If you have any questions regarding Assignment 1, you may contact the teaching assistant of the course at: `altunelnu@itu.edu.tr`.

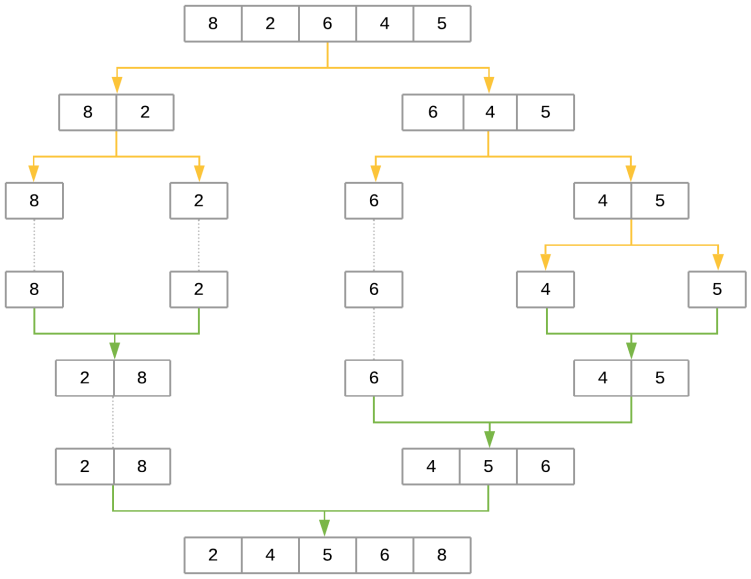

Figure 1: Illustration of the merge sort process.

```
1
2           AREA MergeSort_M0, CODE, READONLY
3           THUMB
4           ENTRY
5
6           EXPORT  main
7           EXPORT  mergeSort
8           EXPORT  merge
9
10  main
11  ;FILL THE MAIN FUNCTION
12
13  MergeSort
14  ;FILL THE mergSort FUNCTION
15
16  merge
17  ;FILL THE merge FUNCTION
```

Figure 2: Reference merge function structure required for naming conventions.