Student Grade Management System using 8086 Assembly Language

Project submitted to the

SRM University – AP, Andhra Pradesh

for the partial fulfillment of the requirements to award the degree of

Bachelor of Technology

In

Computer Science and Engineering

School of Engineering and Sciences

Submitted by

Bandi Srinivasa Karthikeya [AP23110010124]

Kothuri Siva Jathin Chakravarthy [AP23110010007]

Maydiga Surya Sri [AP23110010302]

Fizan Baig Mohammed [AP23110010065]

L. Charme Srija [AP23110010099]

Under the Guidance of

Dr. Deepthi K C

Kakumani Assistant

Professor,

Department of Literature and Languages [April, 2025]

Abstract

This project outlines the design and implementation of a Student Grade Management System developed using 8086 Assembly Language. In the field of Computer Organization and Architecture, hands-on experience with low-level programming is crucial for understanding how computers operate below high-level abstractions. Working directly with the 8086-instruction set not only fosters an appreciation for data flow and memory management but also shows how careful handling of registers and memory can greatly affect system performance.

The system manages the academic data of 25 students, providing a structured way to record, organize, and evaluate their grades. It uses parallel arrays, one for student roll numbers and another for their corresponding grades. This method keeps the data synchronized and easily accessible while ensuring efficient memory use, which is important when working within the limits of the 8086 architecture.

A key part of the project is the implementation of an optimized Bubble Sort algorithm, which arranges student records in descending order of merit. While Bubble Sort is a straightforward algorithm in theory, implementing and refining it in assembly deepens understanding of loops, flag registers, comparisons, and branching instructions.

To improve usability, the system features a menu-driven interface that allows options to categorize students by grade classifications from A to F. This simplifies the interpretation of overall class performance. The application also includes a linear search feature, enabling users to quickly find a student's grade by providing their roll number.

In conclusion, this project showcases the practical use of key architectural concepts such as memory segmentation, register-level data processing, instruction set usage, and software interrupt handling on the Intel 8086 processor. By creating a functional system entirely in assembly language, the project connects theoretical knowledge with real-world implementation and emphasizes the precision and problem-solving skills needed for programming at the hardware level.

Contents

1 Introduction

Computer Organization and Architecture is crucial for understanding how software communicates with hardware. Modern high-level programming languages provide many layers of abstraction that protect developers from dealing with processor instructions, memory addresses, or register management. In contrast, Assembly Language removes these conveniences. It forces the programmer to engage with the basic mechanics of computation. Tasks like moving data between memory and registers, managing memory segmentation, and controlling program flow must be done explicitly. This hands-on experience fosters a better understanding of how each instruction impacts the processor's behavior, making assembly programming a valuable learning tool for anyone who wants to grasp computing at its core.

The main goal of this project is to design and implement a fully functional Student Grade Management System using 8086 Assembly Language. Managing student grades might seem simple in high-level languages, where developers can use predefined functions for sorting, searching, or categorizing data. However, doing the same operations in assembly language is much more complex. Every task, from reading a grade to comparing values or swapping elements, needs to be broken down into basic processor instructions. This approach gives students a direct view of how data is actually stored, moved, and compared inside the Central Processing Unit (CPU).

This project connects theoretical architectural ideas with practical applications. It turns concepts like memory segmentation, instruction execution, and register manipulation from abstract subjects into tangible processes. By working directly with the Data Segment (DS) to store student information and using the Code Segment (CS) to implement algorithms, the program closely mirrors the core functions of a lightweight database management system.

The system carries out three main operations: Sorting, Categorization, and Retrieval. Sorting is achieved through a carefully designed version of the Bubble Sort algorithm, tailored for the limitations of the 8086 architecture. Categorization allows users to group students based on grade ranges, showcasing conditional logic and controlled branching. Retrieval is accomplished through a linear search method, illustrating how data traversal operates when no built-in search functions are available.

In the end, this project highlights the importance of understanding what occurs beneath the abstractions provided by high-level languages. By working at the assembly level, the programmer gains a clearer insight into machine-level operations, processor behavior, and memory organization. The project not only enhances foundational knowledge but also develops the precision, logical thinking, and problem-solving skills necessary for programming at the hardware level.

2 Background and Description of the Problem

2.1 Background

Data sorting is one of the most important and commonly used tasks in computer science. In most high-level programming languages, developers rarely consider the internal mechanics of sorting because efficient, pre-built library functions take care of it. These built-in routines simplify the process, allowing programmers to focus on higher-level logic instead of individual data movements.

In low-level or embedded environments, like those defined by the 8086 microprocessor, the situation is quite different. Here, the responsibility falls entirely on the programmer. They must design and implement the sorting logic from scratch. This requires direct interaction with memory, careful navigation of data structures, and explicit handling of tasks like byte comparison, conditional branching, and value swapping.

Working at this level requires a deeper understanding of how data is stored and accessed in the system. Every comparison, increment, and memory reference must be managed with registers, instructions, and addressing modes specific to the 8086 architecture. Because of this, implementing even a basic sorting algorithm becomes a valuable learning experience. It deepens one's understanding of processor behavior, reinforces concepts like memory addressing and loop control, and emphasizes the precision needed when working close to the hardware.

2.2 Problem Description

The specific challenge addressed in this project involves a dataset representing a class of 25 students. The data presents two distinct issues:

1. **Data Association:** We possess two separate arrays. The first contains Student Roll Num- bers (1–25), and the second contains their respective Grades (random integers between 0–100).
2. **Data Integrity:** When sorting the grades to determine rank, the associated student roll number must move in synchronization with the grade. Failure to do so would result in students being assigned incorrect grades.

The system must allow a user to:

- View the data only after it has been processed (sorted).
- Search for a specific student's result using their unique roll number.
- Assign a letter grade (Category) based on specific numerical thresholds.

3 Problem Statement and Methodology

3.1 Problem Statement

A classroom has 25 students, and we need to arrange them by their grades in the Microprocessor course from highest to lowest. The inputs include two arrays: the first array has the student roll numbers, and the second one contains their corresponding grades. The outputs are also two arrays: the first array hows the student roll numbers sorted by their grades, and the second array lists the grades in descending order. The system also offers an easy-to-use interface for categorizing grades (A–F), displaying a ranked table of students, and allowing users to look up individual records by roll number. This setup ensures easy access to overall class performance and specific student information.

3.2 Methodology

3.2.1 Data Structures

The system uses **Parallel Arrays** stored in the **Data Segment** to maintain student records:

- **students:** A byte array (db) storing student roll numbers from **1 to 25**.
- **grades:** A byte array (db) storing the corresponding **grades** for each student.

This structure ensures that the roll numbers and grades remain synchronized during sorting and retrieval operations, allowing efficient access and manipulation of student data in memory.

3.2.2 Algorithmic Approach: Bubble Sort

To rank students in **descending order of grades**, the **Bubble Sort algorithm** is implemented due to its simplicity in assembly language. Although its **time complexity is $O(n^2)$**, it is acceptable for the small dataset of 25 students.

The algorithm follows these steps:

1. **Comparison:** The processor loads a grade from the grades array into the **AL register** and compares it with the next grade in memory.
2. **Conditional Swap:** If the current grade is **less than the next grade**, a swap is performed to maintain descending order.
3. **Parallel Swapping:** Whenever a swap occurs in the grades array, the **students array is also swapped** at the same index. This maintains the correct association between roll numbers and grades.

3.2.3 Grading Logic

After sorting, the system classifies grades into categories using **CMP instructions** to compare grade values against thresholds:

- **A:** ≥ 90
- **B:** 80 – 89
- **C:** 70 – 79

- **D:** 60 – 69
- **F:** < 60

This logic allows the program to display a **ranked table** showing **student number, grade, and category**, and also supports **individual record lookup** by roll number.

4 Program Code

Below is the complete source code for the Student Grade Management System developed for the EMU8086 emulator.

```
; -------------------------------------------------------------
; PROJECT: SORT 25 STUDENTS BY GRADES (DESCENDING)
; SUBJECT: COMPUTER ORGANISATION AND ARCHITECTURE
; -------------------------------------------------------------

data segment
    ; 25 STUDENT NUMBERS
    students db  1,2,3,4,5,6,7,8,9,10
             db  11,12,13,14,15,16,17,18,19,20
             db  21,22,23,24,25

    ; 25 GRADES (out of 100)
    grades   db  88,92,75,69,85,91,73,95,60,84
             db  77,66,82,94,58,90,87,72,80,89
             db  65,70,83,93,78

    sorted   db 0       ; flag: 0=not sorted, 1=sorted

    ; Menu
    menu1    db 0Dh,0Ah,'======================================$'
    menu2    db 0Dh,0Ah,'   STUDENT GRADE MANAGEMENT SYSTEM   $'
    menu3    db 0Dh,0Ah,'======================================$'
    menu4    db 0Dh,0Ah,'  1. Sort Students by Grade$'
    menu5    db 0Dh,0Ah,'  2. Display Sorted Table$'
    menu6    db 0Dh,0Ah,'  3. Search by Roll Number$'
    menu7    db 0Dh,0Ah,'  4. Exit$'
    menu8    db 0Dh,0Ah,'======================================$'
    menu9    db 0Dh,0Ah,'Enter choice (1-4): $'

    ; Messages
    sorting_msg  db 0Dh,0Ah,'Sorting students... Please wait...$'
    sorted_done  db 0Dh,0Ah,'Sorting completed successfully!$'
    not_sorted_msg db 0Dh,0Ah,'Error: Please sort the data first
(Option 1)!$'
    invalid_msg  db 0Dh,0Ah,'Invalid choice! Please enter 1-4.$'

    ; Table formatting
    header1                                                    db
0Dh,0Ah,0Dh,0Ah,'======================================$'
    header2  db 0Dh,0Ah,'  SORTED STUDENT GRADES TABLE       $'
    header3  db 0Dh,0Ah,'======================================$'
    header4  db 0Dh,0Ah,' Rank | Student | Grade | Category $'
    header5  db 0Dh,0Ah,'--------------------------------------$'
```

```
    footer   db 0Dh,0Ah,'======================================$'

    ; Search prompts
    search_hdr                                                  db
0Dh,0Ah,0Dh,0Ah,'======================================$'
    search_title db 0Dh,0Ah,'      SEARCH STUDENT BY ROLL NO      $'
    search_line  db 0Dh,0Ah,'======================================$'
    prompt_msg   db 0Dh,0Ah,'Enter student number (1-25): $'
    result_msg   db 0Dh,0Ah,0Dh,0Ah,'Student #$'
    grade_msg    db ' has Grade: $'
    category_msg db ' [Category: $'
    not_found    db 0Dh,0Ah,'Student not found!$'

    press_key    db 0Dh,0Ah,0Dh,0Ah,'Press any key to continue...$'
    exit_msg     db 0Dh,0Ah,'Thank you for using the system!$'

    rank_num db 1
ends

stack segment
    dw   128  dup(0)
ends

code segment
start:
    ; Set segment registers
    mov ax, data
    mov ds, ax
    mov es, ax

; ---------------------------------------------------------
; MAIN MENU LOOP
; ---------------------------------------------------------
main_menu:
    ; Display menu
    lea dx, menu1
    mov ah, 09h
    int 21h

    lea dx, menu2
    mov ah, 09h
    int 21h

    lea dx, menu3
    mov ah, 09h
    int 21h

    lea dx, menu4
    mov ah, 09h
    int 21h

    lea dx, menu5
    mov ah, 09h
    int 21h
```

```
    lea dx, menu6
    mov ah, 09h
    int 21h

    lea dx, menu7
    mov ah, 09h
    int 21h

    lea dx, menu8
    mov ah, 09h
    int 21h

    lea dx, menu9
    mov ah, 09h
    int 21h

    ; Get user choice
    mov ah, 01h
    int 21h

    cmp al, '1'
    je option_sort
    cmp al, '2'
    je option_display
    cmp al, '3'
    je option_search
    cmp al, '4'
    je option_exit

    ; Invalid choice
    lea dx, invalid_msg
    mov ah, 09h
    int 21h

    call wait_key
    jmp main_menu

; ------------------------------------------------------------
; OPTION 1: SORT STUDENTS
; ------------------------------------------------------------
option_sort:
    lea dx, sorting_msg
    mov ah, 09h
    int 21h

    ; Bubble sort
    mov cx, 24

outer_loop:
    push cx
    mov bx, 0
    mov si, cx
```

```
inner_loop:
    mov al, grades[bx]
    mov dl, grades[bx+1]

    cmp al, dl
    jae no_swap

    ; SWAP GRADES
    mov grades[bx], dl
    mov grades[bx+1], al

    ; SWAP STUDENT NUMBERS
    mov al, students[bx]
    mov dl, students[bx+1]
    mov students[bx], dl
    mov students[bx+1], al

no_swap:
    inc bx
    dec si
    jnz inner_loop

    pop cx
    loop outer_loop

    ; Mark as sorted
    mov sorted, 1

    lea dx, sorted_done
    mov ah, 09h
    int 21h

    call wait_key
    jmp main_menu

; ------------------------------------------------------------
; OPTION 2: DISPLAY TABLE
; ------------------------------------------------------------
option_display:
    ; Check if sorted
    cmp sorted, 0
    je not_sorted_error

    ; Print headers
    lea dx, header1
    mov ah, 09h
    int 21h

    lea dx, header2
    mov ah, 09h
    int 21h

    lea dx, header3
    mov ah, 09h
```

```
    int 21h

    lea dx, header4
    mov ah, 09h
    int 21h

    lea dx, header5
    mov ah, 09h
    int 21h

    ; Print rows
    mov cx, 25
    xor si, si
    mov byte ptr rank_num, 1

print_rows:
    ; New line
    mov dl, 0Dh
    mov ah, 02h
    int 21h
    mov dl, 0Ah
    mov ah, 02h
    int 21h

    ; Rank
    mov dl, ' '
    mov ah, 02h
    int 21h

    mov al, rank_num
    call print_num_fixed
    inc rank_num

    ; Separator
    mov dl, ' '
    mov ah, 02h
    int 21h
    int 21h
    mov dl, '|'
    int 21h
    mov dl, ' '
    int 21h
    int 21h

    ; Student number
    mov al, students[si]
    call print_num_fixed

    ; Separator
    mov dl, ' '
    mov ah, 02h
    int 21h
    int 21h
    int 21h
```

```
    mov dl, '|'
    int 21h
    mov dl, ' '
    int 21h
    int 21h

    ; Grade
    mov al, grades[si]
    push si
    call print_num_fixed

    ; Separator
    mov dl, ' '
    mov ah, 02h
    int 21h
    int 21h
    mov dl, '|'
    int 21h
    mov dl, ' '
    int 21h
    int 21h

    ; Category
    pop si
    mov al, grades[si]
    call print_category

    inc si
    loop print_rows

    ; Footer
    lea dx, footer
    mov ah, 09h
    int 21h

    call wait_key
    jmp main_menu

not_sorted_error:
    lea dx, not_sorted_msg
    mov ah, 09h
    int 21h
    call wait_key
    jmp main_menu

; ------------------------------------------------------------
; OPTION 3: SEARCH STUDENT
; ------------------------------------------------------------
option_search:
    ; Check if sorted
    cmp sorted, 0
    je not_sorted_error

    lea dx, search_hdr
```

```
    mov ah, 09h
    int 21h

    lea dx, search_title
    mov ah, 09h
    int 21h

    lea dx, search_line
    mov ah, 09h
    int 21h

    lea dx, prompt_msg
    mov ah, 09h
    int 21h

    ; Get input
    call get_number
    mov bl, al

    ; Search
    mov cx, 25
    xor si, si

search_loop:
    mov al, students[si]
    cmp al, bl
    je found_student
    inc si
    loop search_loop

    ; Not found
    lea dx, not_found
    mov ah, 09h
    int 21h
    jmp search_done

found_student:
    ; Display result
    lea dx, result_msg
    mov ah, 09h
    int 21h

    mov al, bl
    call print_num_fixed

    lea dx, grade_msg
    mov ah, 09h
    int 21h

    mov al, grades[si]
    call print_num_fixed

    lea dx, category_msg
    mov ah, 09h
```

```
    int 21h

    mov al, grades[si]
    call print_category

    mov dl, ']'
    mov ah, 02h
    int 21h

search_done:
    call wait_key
    jmp main_menu

; ------------------------------------------------------------
; OPTION 4: EXIT
; ------------------------------------------------------------
option_exit:
    lea dx, exit_msg
    mov ah, 09h
    int 21h

    mov ax, 4c00h
    int 21h

; ------------------------------------------------------------
; WAIT FOR KEY PRESS
; ------------------------------------------------------------
wait_key:
    lea dx, press_key
    mov ah, 09h
    int 21h

    mov ah, 01h
    int 21h
    ret

; ------------------------------------------------------------
; GET NUMBER INPUT
; ------------------------------------------------------------
get_number:
    push bx
    push cx
    push dx

    mov ah, 01h
    int 21h

    cmp al, 0Dh
    je input_done

    sub al, '0'
    mov bl, al

    mov ah, 01h
```

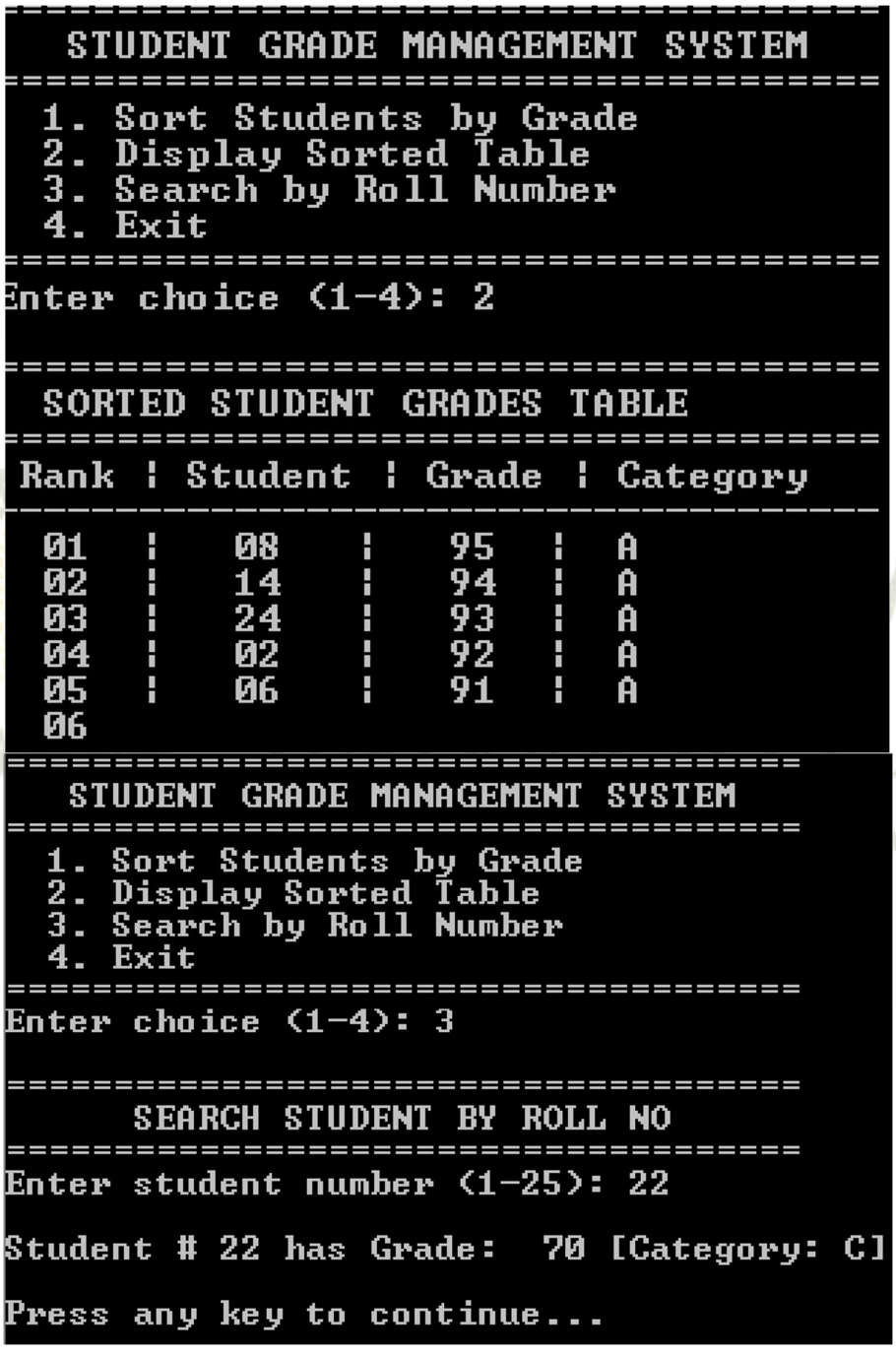
```
    int 21h

    cmp al, 0Dh
    je single_digit

    sub al, '0'
    mov bh, al
    mov al, bl
    mov cl, 10
    mul cl
    add al, bh
    jmp input_done

single_digit:
    mov al, bl

input_done:
    pop dx
    pop cx
    pop bx
    ret

; ------------------------------------------------------------
; PRINT CATEGORY
; ------------------------------------------------------------
print_category:
    push ax
    push dx

    cmp al, 90
    jae grade_A
    cmp al, 80
    jae grade_B
    cmp al, 70
    jae grade_C
    cmp al, 60
    jae grade_D
    jmp grade_F

grade_A:
    mov dl, 'A'
    jmp print_cat
grade_B:
    mov dl, 'B'
    jmp print_cat
grade_C:
    mov dl, 'C'
    jmp print_cat
grade_D:
    mov dl, 'D'
    jmp print_cat
grade_F:
    mov dl, 'F'
```

```
print_cat:
    mov ah, 02h
    int 21h

    pop dx
    pop ax
    ret

; ------------------------------------------------------------
; PRINT NUMBER FIXED WIDTH
; ------------------------------------------------------------
print_num_fixed:
    push ax
    push dx
    push bx

    xor ah, ah
    mov bl, 100
    div bl

    push ax

    cmp al, 0
    je print_space_h
    add al, '0'
    mov dl, al
    mov ah, 02h
    int 21h
    jmp after_h

print_space_h:
    mov dl, ' '
    mov ah, 02h
    int 21h

after_h:
    pop ax
    mov al, ah
    xor ah, ah
    mov bl, 10
    div bl

    push ax

    add al, '0'
    mov dl, al
    mov ah, 02h
    int 21h

    pop ax
    mov dl, ah
    add dl, '0'
    mov ah, 02h
    int 21h
```

```
    pop bx
    pop dx
    pop ax
    ret

ends
end start
```

5 Results

Upon execution in the EMU8086 environment, the project successfully performed all intended functions.



6 Conclusion

The Student Grade Management System was successfully designed and implemented using Assembly language, demonstrating how low-level programming can effectively handle structured data operations. This project highlights the strength of Assembly in providing direct control over hardware resources, memory allocation, and data manipulation. By structuring the system around parallel arrays for storing student roll numbers and their corresponding grades, the program ensures data consistency and supports efficient record retrieval and ranking.

The integration of the Bubble Sort algorithm further showcases how classical sorting techniques can be translated into step-by-step Assembly logic. Despite its $O(n^2)$ time complexity, Bubble Sort is well-suited for the system's dataset of 25 students. The project illustrates how comparisons, conditional jumps, and swapping operations can be coordinated to maintain synchronized arrays, preserving the correct mapping of students to their grades. This reinforces an essential concept in system-level programming: maintaining data integrity through precise, low-level control.

Additionally, the grading classification component demonstrates how logical constraints can be enforced using CMP instructions and conditional branching. The system accurately maps numerical scores to grade categories (A, B, C, D, F), mirroring real-world academic evaluation processes. This logical structure enables the program not only to compute rankings but also to provide meaningful interpretation of student performance.

Overall, the project confirms that Assembly language remains a valuable tool for understanding core architectural behavior. Through this implementation, key concepts such as memory addressing, register operations, control flow, and data linkage become clearer. The system also reflects how menu-driven interfaces and search functionalities can be constructed even in a low-level environment, emphasizing the versatility of Assembly when used systematically.

In conclusion, this project provides a strong foundation for understanding how complex systems can be built from basic operational instructions. It enhances both theoretical knowledge and practical skill, demonstrating that efficient data processing is achievable even at the lowest level of programming abstraction.