Shri Gajanan Shikshan Sanstha's

SHRI SANT GAJANAN MAHARAJ COLLEGE OF ENGINEERING

SHEGAON – 444203, DIST. BULDANA (MAHARASHTRA STATE), INDIA

"Recognized by A.I.C.T.E., New Delhi" Affiliated to Sant Gadge Baba Amravati University, Amravati

"Approved by the D.T.E., M.S. Mumbai", Institution Accredited by N.A.A.C. (UGC) Bangalore

DEPARTMENT OF INFORMATION TECHNOLOGY

BUBBLE SORT ALGORITHM (TASM)

ALP TEC PROJECT

PRESENTED BY: HRIDAY AMLE (2N-40)

ADITYA TALE (2N-29)

YEAR: SECOND (SEMESTER-3)

BRANCH: INFORMATION TECHNOLOGY

GUIDED BY: MS. P.P. BUTE

DATE: 20 NOVEMBER 2024

TABLE OF CONTENTS

Introduction

1.1 Overview of the Topic

Assembly language programming offers a deep understanding of computer architecture and memory management, making it a valuable skill for system-level programming. Unlike high-level programming languages that abstract away hardware details, assembly language gives programmers direct control over the system's hardware resources.

This project focuses on implementing a simple **sorting algorithm**, specifically **Bubble Sort**, using **Turbo Assembler (TASM)** for **x86 architecture** processors. By implementing the algorithm in assembly language, this project provides a hands-on approach to understanding basic programming concepts, memory handling, and the execution of low-level operations.

1.2 Problem Statement

Sorting is one of the fundamental tasks in computer science, used in a wide range of applications, from organizing data to optimizing performance in various algorithms. The objective of this project is to implement a sorting algorithm, **Bubble Sort**, using assembly language in **TASM**.

The array of integers will be sorted in **ascending order** to demonstrate how sorting algorithms are applied at the low-level, without relying on built-in functions or high-level language abstractions.

1.3 Objective of the Project

The primary objectives of this project are:

- To **implement the Bubble Sort algorithm** in **assembly language** using **Turbo Assembler (TASM)**.
- To **sort an array of integers** in **ascending order**, showcasing an understanding of algorithm design in assembly.
- To **demonstrate memory management techniques** in assembly, particularly the handling of arrays and basic input/output operations.
- To explore and understand **control flow**, **looping**, and **conditional logic** in assembly language programming.

1.4 Facilities Required

- **Hardware**:
  A **computer with an x86 architecture processor** is required to run TASM and execute assembly programs. The system should support **DOS** or a **DOS emulator** (like **DOSBox**) for running the assembly code. The minimum required system specifications include:
  - **At least 512 MB RAM**.
  - **x86 architecture processor** capable of running 16-bit or 32-bit assembly code.
- **Software**:
  - **Turbo Assembler (TASM)**: A powerful assembler for x86 processors that allows the writing and compilation of assembly programs.
  - **DOSBox** or any DOS-compatible emulator: Since modern operating systems do not natively support DOS applications, **DOSBox** is used to emulate the DOS environment for running TASM.
  - **Text Editor**: Any basic text editor (like **Notepad** or **TASM IDE**) to write and edit the assembly code.

1.5 Scope of the Study

This project explores the fundamental aspects of **assembly language programming** by implementing the **Bubble Sort algorithm**. The key focus areas of this project include:

- **Sorting Algorithm**: Implementing and understanding the Bubble Sort algorithm and its step-by-step execution.
- **Memory Management**: Demonstrating how arrays are managed in assembly, including the allocation, storage, and manipulation of data in memory.
- **Input/Output Operations**: Managing input and output operations in assembly, taking user input (array of integers) and displaying the sorted output.
- **Low-Level Control**: Understanding how loops, conditional statements, and branching are used to control the flow of the program at the assembly level.

The project will offer a comprehensive understanding of how sorting algorithms are implemented at the lowest levels of programming, providing valuable experience in memory management, program logic, and algorithm design in **assembly language**.

Theory Overview

2.1 About Bubble Sort

Bubble Sort is a simple comparison-based sorting algorithm used to arrange elements in a list or array in **ascending order**. The algorithm works by repeatedly stepping through the array, comparing each pair of adjacent elements and swapping them if they are in the wrong order. This process continues until the array is sorted.

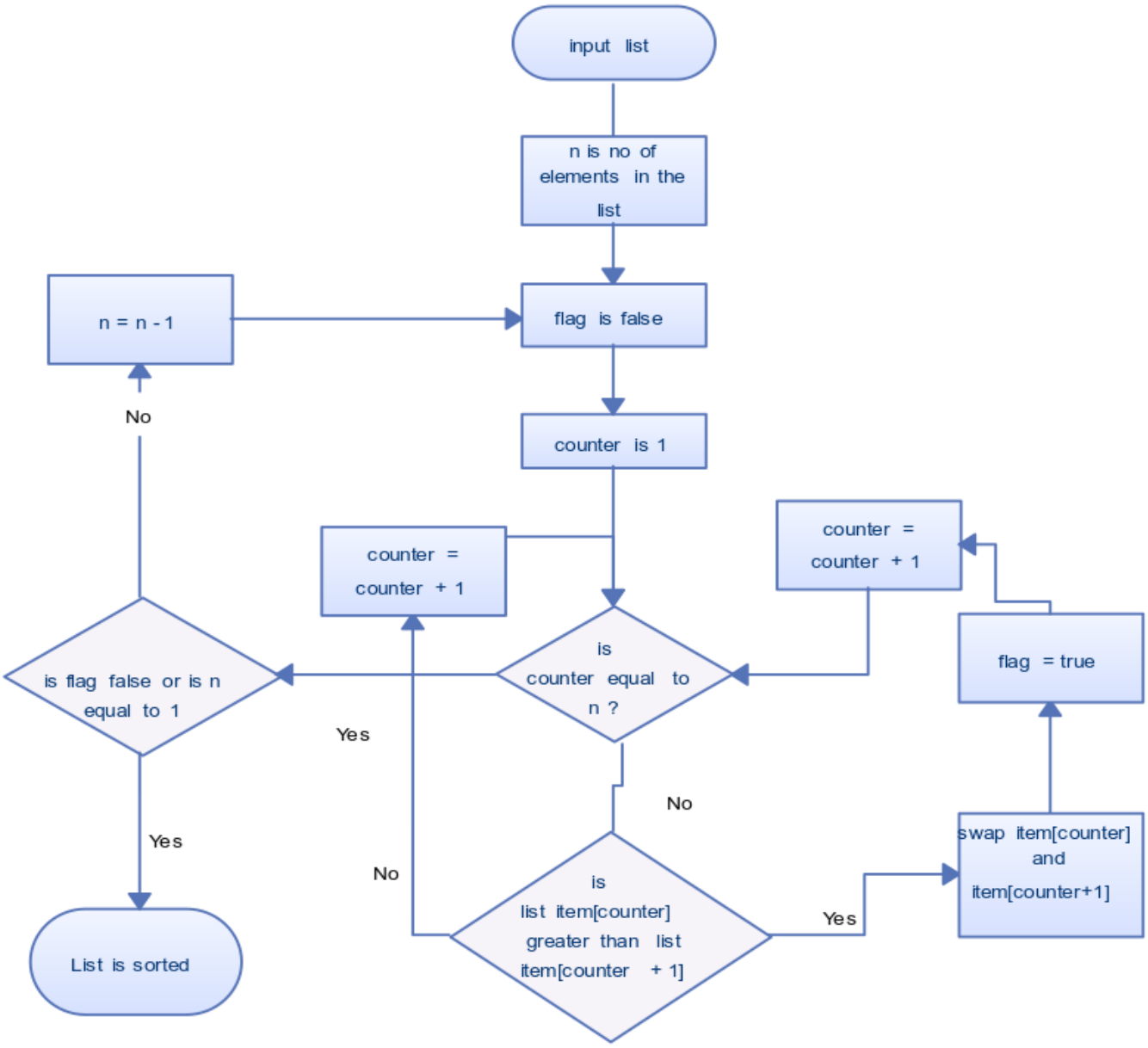

Fig: Bubble Sort Algorithm Flowchart

How Bubble Sort Works:

- **Comparison and Swapping**: In each iteration, adjacent elements are compared. If the first element is greater than the second, they are swapped. This comparison process "bubbles" the larger elements to the end of the array.

- **Multiple Passes**: After each pass, the largest element is placed at its correct position, and the algorithm continues to iterate over the unsorted portion of the array.
- **Termination**: The process continues until a complete pass through the array results in no swaps, indicating that the array is fully sorted.

Efficiency of Bubble Sort:

- **Time Complexity**: The time complexity of Bubble Sort is **$O(n^2)$** in both the average and worst-case scenarios, where **n** is the number of elements in the array. This quadratic time complexity makes it inefficient for large datasets but acceptable for small datasets or educational purposes.
- **Best Case**: If the array is already sorted, the best-case time complexity is **O(n)**, as the algorithm will make only one pass to confirm no swaps are needed.
- **Space Complexity**: Bubble Sort is an **in-place sorting algorithm**, meaning it uses **O(1)** additional space to sort the array, modifying the array directly.

Applications of Bubble Sort:

Though not suitable for large datasets due to its inefficiency, **Bubble Sort** is often used in educational contexts for teaching the basics of sorting algorithms and for sorting small datasets where performance is not a major concern.

2.2 Assembly Language Concepts

Assembly language is a low-level programming language that provides a symbolic representation of a computer's machine code instructions. Unlike high-level programming languages, assembly gives programmers direct control over the hardware, allowing for fine-grained management of memory and processor registers.

In this project, assembly language is used to implement the **Bubble Sort algorithm**, providing a hands-on understanding of how algorithms work at the hardware level, including memory management, control flow, and register manipulation.

Key Concepts in Assembly Language:

- **Registers**: Registers are small, fast storage locations within the CPU used to hold data during program execution. In **x86 architecture**, common registers include **AX**, **BX**, **CX**, and **DX**. These registers are used to store intermediate results, counters for loops, and memory addresses.
- **Instructions**: Assembly language consists of low-level instructions that directly correspond to machine code instructions. For example:
    - **MOV**: Moves data between registers or between registers and memory.
    - **CMP**: Compares two values.

- **JMP**: Performs an unconditional jump to another part of the program (used for loops and conditionals).
- **ADD, SUB, MUL, DIV**: Perform arithmetic operations.

- **Memory Management**: Assembly requires explicit management of memory. This includes allocating space for arrays and variables, as well as accessing and modifying them. In this project, we manually allocate memory for the array of integers to be sorted, and manipulate data directly in memory during the sorting process.
- **Control Flow**: Control flow in assembly is governed by conditional statements and loops, which are implemented using **jump** instructions like **JZ (jump if zero)** or **JNZ (jump if not zero)**. These instructions allow the program to branch based on the outcome of comparisons and control the flow of execution, which is crucial for implementing the sorting logic of **Bubble Sort**.
- **In-place Sorting**: Since assembly works directly with memory addresses, **Bubble Sort** is implemented in-place. This means that the algorithm does not require additional memory for storing a sorted copy of the array; instead, the array is modified directly, with elements being swapped within the same memory space.

Why Use Assembly for Sorting Algorithms?

Implementing **Bubble Sort** in assembly language is an educational exercise that demonstrates key programming concepts:

- **Low-Level Memory Management**: Working with arrays and registers directly gives insight into memory handling at the hardware level.
- **Efficient Control of Hardware**: Assembly provides the ability to optimize operations for performance, even in simple algorithms like Bubble Sort.
- **Understanding Hardware Operations**: By writing algorithms in assembly, we learn how the CPU processes data at the instruction level and how to use its resources effectively.

While **Bubble Sort** is not efficient for large datasets, it offers a clear understanding of fundamental sorting principles. By implementing it in **assembly language**, this project highlights key low-level concepts such as **registers**, **memory management**, and **control flow**, providing a deeper insight into how algorithms execute at the hardware level. This foundation is essential for tackling more complex algorithms and system-level programming in the future.

Algorithm Design & Explanation

3.1 Bubble Sort Algorithm

```
.MODEL SMALL
.STACK 100h

MYDATA SEGMENT
    ; Define the array of integers (single-digit)
    array DB 5, 3, 8, 6, 9
    arraySize DW 5

    ; Strings for output
    unsortedMsg DB 'Unsorted Array: $'
    sortedMsg   DB 'Sorted Array: $'
    newline     DB 0Dh, 0Ah, '$'
MYDATA ENDS

MYCODE SEGMENT
ASSUME CS:MYCODE,DS:MYDATA
MAIN PROC
    ; Initialize data segment
    MOV AX, MYDATA
    MOV DS, AX

    ; Display unsorted array
    LEA DX, unsortedMsg
    MOV AH, 09h
    INT 21h

    ; Print the unsorted array
    CALL PrintArray
```

```
    ; Bubble sort algorithm
    MOV CX, arraySize
    DEC CX                          ; We need to do n-1 passes
OuterLoop:
    MOV SI, 0                       ; Index j
    MOV DX, arraySize
    DEC DX                        ; Inner loop needs n-i-1 comparisons
InnerLoop:
    MOV AL, array[SI]               ; Load current element
    MOV AH, array[SI + 1]           ; Load next element
    CMP AL, AH
    JBE NoSwap                      ; If AL <= AH, no swap needed

    ; Swap elements
    XCHG AL, AH
    MOV array[SI], AL
    MOV array[SI + 1], AH

NoSwap:
    INC SI                          ; Move to the next element
    DEC DX                          ; Decrement inner loop counter
    JNZ InnerLoop                   ; Repeat inner loop if not done

    INC BX                          ; Increment outer loop counter
    DEC CX                          ; Decrement outer loop counter
    JNZ OuterLoop                   ; Repeat outer loop if not done

    ; Display sorted array
    LEA DX, sortedMsg
    MOV AH, 09h
```

```
    INT 21h

    ; Print the sorted array
    CALL PrintArray

    ; Exit program
    MOV AX, 4C00h
    INT 21h
MAIN ENDP

; Procedure to print the array
PrintArray PROC
    MOV CX, arraySize
    MOV SI, 0 ; Index

PrintLoop:
    MOV AL, array[SI]           ; Get the current element
    ADD AL, '0'           ; Convert to ASCII (only works for 0-9)
    MOV DL, AL                  ; Move to DL for printing
    MOV AH, 02h                 ; Function to print character
    INT 21h                     ; Print the character

    ; Print a space
    MOV DL, ' '
    INT 21h

    INC SI                      ; Move to the next element
    LOOP PrintLoop              ; Repeat until done

    ; Print a newline
    LEA DX, newline
```

```
    MOV AH, 09h
    INT 21h

    RET
PrintArray ENDP

MYCODE ENDS
END MAIN
```

3.2 Program Explanation

1. Specify the memory mode used for small programs using **.MODEL SMALL**
2. Allocate 256 bytes (100h in hexadecimal) for the stack (used for function calls & local variables) with **.STACK 100h**
3. Create a data segment named **MYDATA**.
4. Define an array of five single-digit numbers.
5. Define a word-sized variable for total count of array elements.
6. Define strings for output messages. The **$** character indicates the end of the string for the DOS interrupt.
7. End of the data segment.
8. Create a code segment named **MYCODE**.
9. Assume **DS, CS** as **MYDATA, MYCODE**
10. Create a **MAIN** procedure.
11. Initialize data segment using **MOV AX, MYDATA**
    MOV DS, AX
12. Load the address of the **unsortedMsg** string into the DX register.
13. Prepare the DOS interrupt for displaying a string.
14. Call the DOS interrupt to display the string.

15. Call the **PrintArray** procedure to print the contents of the array.
16. Load count of array elements into CX register as loop counter.
17. Decrease the loop counter for **n-1** passes.
18. Define a label **OuterLoop:**
    i. Initialize SI register to 0 as the index of the current element.
    ii. Load array element count into DX for the inner loop.
    iii. Decrease DX by one for the inner loop's comparisons.
19. Define a label **InnerLoop:**
    i. Load the current array element (say e1) into the AL register.
    ii. Load the next array element (say e2) to AH register.
    iii. Compare the two elements in AL and AH.
    iv. Jump to **NoSwap** label if AL is less than or equal to AH.
20. Swap the values in AL and AH using XCHG AL, AH.
21. Store the swapped value in AL back into the e1.
22. Store the swapped value in AH back into the e2.
23. Define a label **NoSwap:**
    i. Increase the index (SI) to move to the next element.
    ii. Decrease DX counting down the inner loop's remaining comparisons.
    iii. Jump back to **InnerLoop** is DX is not zero.
24. Increment BX for tracking outer loop passes.
25. Decrement CX to account completed passes.
26. Jump back to **OuterLoop** if CX is not zero.
27. Load the address of the **sortedMsg** string into the DX register.
28. Prepare the DOS interrupt for displaying a string.
29. Call the DOS interrupt to display the string.
30. Call the **PrintArray** procedure again to print the sorted array.
31. Prepare to terminate the program. The value **4C00h** indicates a normal exit.

32. Call the DOS interrupt to terminate the program.
33. End of the Main Procedure.
34. Define a **`PrintArray`** procedure to print array elements.
35. Load the size of the array into CX for iteration.
36. Initialize the index SI to 0 for accessing the array.
37. Create a label **`PrintLoop:`**
    i. Loads the current array element into AL.
    ii. Convert the integer to its ASCII representation.
    iii. Move the ASCII character to DL for printing.
    iv. Prepare for printing a character and call the DOS interrupt.
38. Load a space character into DL and call the DOS interrupt.
39. Increment the index to access the next element.
40. Decrement CX and jump back to **`PrintLoop`** if CX is not zero.
41. Load the address of the newline string into DX and prepare to print the newline.
42. Call the DOS Interrupt and Return from the **`PrintArray`** procedure.
43. End of the **`PrintArray`** procedure.
44. End of the **`MYCODE`** segment
45. End of the program and specify the Entry point of the program.

Output & Execution Process

4.1 Output of the Program

```
D:\>DEBUG SORTB.EXE
-G
Unsorted Array: 5 3 8 6 9
Sorted Array: 3 5 6 8 9

Program terminated normally
-Q
```

4.1 Execution Process

```
D:\>TASM SORTB.ASM
Turbo Assembler  Version 2.5  Copyright (c) 1988, 1991 Borland International

Assembling file:   SORTB.ASM
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  490k


D:\>LINK SORTB.OBJ

Microsoft (R) Overlay Linker  Version 3.60
Copyright (C) Microsoft Corp 1983-1987.  All rights reserved.

Run File [SORTB.EXE]:
List File [NUL.MAP]: SORTB.MAP
Libraries [.LIB]:

D:\>TYPE SORTB.MAP

 Start  Stop   Length Name                   Class
 00000H 00000H 00000H _TEXT                  CODE
 00000H 00000H 00000H _DATA                  DATA
 00000H 000FFH 00100H STACK                  STACK
 00100H 00129H 0002AH MYDATA
 00130H 0019AH 0006BH MYCODE

 Origin   Group
 0000:0   DGROUP

Program entry point at 0013:0000

D:\>DEBUG SORTB.EXE
-G
Unsorted Array: 5 3 8 6 9
Sorted Array: 3 5 6 8 9

Program terminated normally
-Q
```

Conclusion

5.1 Challenges

While implementing the **Bubble Sort** algorithm in **assembly language**, several challenges arose that required careful consideration and problem-solving:

1. **Memory Management**: In assembly, manual memory management is crucial, and handling arrays efficiently required an in-depth understanding of memory addressing. The array of integers was manually defined in the data segment, and each element needed to be accessed and modified directly in memory.
2. **Control Flow and Looping**: Implementing the sorting logic using **loops** and **conditional statements** in assembly posed a challenge due to the low-level nature of the language. The **inner and outer loops** required precise control using the **MOV**, **CMP**, and **JMP** instructions to ensure correct comparisons and swaps between elements.
3. **Character Output**: Displaying the unsorted and sorted arrays as output required converting integer values to their ASCII equivalents, which is a manual process in assembly. Ensuring proper formatting of the output, including spaces and newlines, was also a challenge, requiring careful attention to the use of interrupt calls.
4. **Debugging**: Since assembly language provides no high-level abstractions, debugging required a deep understanding of register values, memory addresses, and the execution flow. Identifying and correcting issues such as incorrect swapping or indexing often involved checking the contents of registers and memory locations step by step.

5.2 Key Achievements

Despite the challenges, the project achieved several important outcomes:

1. **Successful Implementation of Bubble Sort**: The **Bubble Sort** algorithm was successfully implemented in assembly language, with the array being sorted in **ascending order** as required. The algorithm effectively used nested loops and conditional jumps to compare and swap array elements.
2. **Effective Memory Management**: Through manual memory allocation in the **data segment**, the array was correctly stored and manipulated. The program successfully demonstrated low-level memory handling techniques, which is crucial in assembly language programming.
3. **Output Display**: The project included functions to display the **unsorted** and **sorted** arrays. Using **DOS interrupts**, the program printed the arrays to the console and ensured proper formatting with spaces and newlines for clarity.

4. **Enhanced Understanding of Assembly Language**: The project provided valuable experience with low-level programming concepts such as **register manipulation**, **conditional logic**, **loops**, and **in-place sorting**. It helped solidify an understanding of how sorting algorithms work at the hardware level, providing insight into the importance of efficient memory use and control flow.

5.3 Conclusion

This project provided a hands-on opportunity to explore assembly language programming and its application to algorithm design. By implementing the **Bubble Sort** algorithm in **Turbo Assembler (TASM)** for **x86 architecture**, the project offered practical experience in writing low-level programs that interact directly with the hardware.

The skills and knowledge gained during this project are essential for anyone interested in **system-level programming**, **hardware management**, and **algorithm optimization**. This project has laid a strong foundation for tackling more complex assembly language tasks and advanced algorithms in the future.