External Project Report on Computer Organization and Architecture (EET 2211)

Design a system that replace a character with a user input character in a given string using 8086 assembly language

Submitted by:

KIRTIBARDHAN GAAN	Reg No : 2241007002
BALAJI GANGULI	Reg No : 2241007004
CHINMAYI GANTAYAT	Reg No : 2241007006
DEEPAK RANJAN GIRI	Reg No : 2241007026

B. Tech. CSE 4th Semester (Section - 29)

INSTITUTE OF TECHNICAL EDUCATION AND RESEARCH

(FACULTY OF ENGINEERING)

SIKSHA 'O' ANUSANDHAN (DEEMED TO BE UNIVERSITY), BHUBANESWAR, ODISHA

Declaration

We, the undersigned students of B. Tech. of **Computer Science Engineering** Department hereby declare that we own the full responsibility for the information, results etc. provided in this PROJECT titled "**Design a system that replace a character with a user input character in a given string using 8086 assembly language**" submitted to **Siksha 'O' Anusandhan Deemed to be University, Bhubaneswar** for the partial fulfillment of the subject **Computer Organization and Architecture (EET 2211)**. We have taken care in all respect to honor the intellectual property right and have acknowledged the contribution of others for using them in academic purpose and further declare that in case of any violation of intellectual property right or copyright we, as the candidate(s), will be fully responsible for the same.

KIRTIBARDHAN GAAN

Registration No.: 2241007002

BALAJI GANGULI

Registration No.: 2241007004

CHINMAYI GANTAYAT

Registration No.: 2241007006

DEEPAK RANJAN GIRI

Registration No.: 2241007026

DATE:

PLACE: ITER

Abstract

The project aims to develop a robust system in 8086 assembly language capable of replacing characters within a string as specified by user input. In today's computing landscape, where low-level programming languages like assembly are still relevant in certain domains, such a system holds practical value in text processing tasks, data manipulation, and embedded systems programming. This project delves into the intricacies of assembly language programming, offering insights into string manipulation, user input handling, algorithm design, memory management, and error handling mechanisms.

At its core, the system solicits user input for the target string, the character to be replaced, and the replacement character. Upon receiving input, the system meticulously traverses the input string character by character, employing efficient algorithms to identify and replace occurrences of the specified character. Throughout the implementation process, emphasis is placed on optimizing memory usage, enhancing execution speed, and ensuring the system's correctness and reliability through rigorous testing.

The development process adheres to a systematic methodology encompassing various stages, starting with problem analysis and research. Understanding the requirements and constraints of the 8086 architecture, the system's design is meticulously planned to accommodate efficient memory allocation, algorithmic complexity, and user interaction. Implementation follows a modular approach, with individual components structured and coded in accordance with the design specifications.

Testing plays a pivotal role in validating the system's functionality and robustness. Test cases covering diverse input scenarios, including edge

cases and boundary conditions, are devised and executed to ensure comprehensive coverage. Additionally, optimization techniques such as loop unrolling, register allocation, and code refactoring are employed to enhance the system's performance and efficiency.

Documentation is integral to the project, providing detailed insights into the system's design, implementation, testing procedures, and usage instructions. Inline comments within the assembly code elucidate key functions, algorithms, and data structures, facilitating comprehension and maintenance.

In conclusion, the project culminates in the development of a sophisticated system that offers a practical solution to the task of character replacement within strings in the 8086 assembly language. Through meticulous planning, implementation, testing, and documentation, the project not only addresses the specified requirements but also serves as a testament to the intricacies and nuances of assembly language programming. As such, it contributes to the collective knowledge and understanding of low-level programming concepts and techniques, empowering programmers with valuable insights into the intricacies of assembly language development

Contents

1. Introduction

In the realm of computer programming, particularly in the realm of low-level programming languages like assembly language, manipulating strings and characters efficiently is essential. One common task encountered in programming is the need to replace occurrences of a specific character within a string with another character. This task is foundational in text processing applications, data manipulation, and various other domains.[1]

In this context, we'll delve into the intricacies of designing a system using 8086 assembly language to replace a character within a given string with another character. The 8086 assembly language, although vintage, is still relevant and widely used in specific embedded systems and legacy applications. Understanding its principles provides a solid foundation for comprehending computer architecture and low-level programming concepts. [3]

The task at hand involves several key components:

1. String Manipulation: Assembly language, being a low-level language, operates directly on memory and registers. Thus, string manipulation involves accessing memory locations, iterating over characters, and performing operations like comparison and replacement.

2. Character Replacement Algorithm: The algorithm to replace a character within a string typically involves iterating over each character in the string, checking if it matches the character to be replaced, and if so, replacing it with the desired character.

3. User Input Handling: The system must handle user input effectively, allowing users to specify the target character to be replaced and the character to replace it with.

4. Memory Management: Managing memory efficiently is crucial in assembly language programming. Allocating space for the string, temporary variables, and handling memory addresses accurately are essential considerations.

5. Error Handling: Robust error handling mechanisms need to be implemented to handle scenarios such as invalid input, insufficient memory, or unexpected program behavior.

Designing such a system requires careful consideration of the limitations and capabilities of the 8086 architecture, as well as an understanding of fundamental programming concepts.

Here's a high-level overview of the steps involved in designing the system:

1. Input Processing: Accept user input specifying the string, the character to be replaced, and the replacement character.

2. String Traversal: Iterate over each character in the string, comparing it with the character to be replaced.

3. Replacement Logic: If a match is found, replace the character with the replacement character.

4. Output Generation: Output the modified string.

5. Error Handling: Implement checks for potential errors and handle them gracefully.

Implementing these steps efficiently in 8086 assembly language requires careful attention to detail and a deep understanding of the architecture's nuances. Memory management, register usage, and instruction optimization are critical factors that influence the performance and correctness of the system.

In conclusion, designing a system to replace a character within a string using 8086 assembly language involves a combination of string manipulation, algorithm design, user input handling, memory management, and error handling. Mastery of these concepts equips programmers with the skills to tackle a wide range of text processing tasks in the realm of low-level programming.[2]

2. Problem Statement

Designing a system in 8086 assembly language to replace characters within a string with user-specified replacements involves several intricate steps and considerations:

In the realm of low-level programming, character manipulation within strings is a fundamental task with diverse applications. This project focuses on implementing such functionality using 8086 assembly language, a vintage but still relevant language in certain domains.

The primary objective is to create a system capable of replacing all occurrences of a specified character within a given string with another character. This involves accepting user input for the input string, the character to be replaced, and the replacement character, and then performing the replacement operation efficiently.[5]

User input is a crucial aspect of the system. It must prompt the user for the input string, ensuring proper termination with a null character ('\0'). Additionally, it should validate the input to prevent buffer overflow by enforcing a predefined maximum length for the input string.

The system should provide a user-friendly interface for inputting the string, the character to be replaced, and the replacement character. Clear instructions and error messages should guide the user through the input process.

Once the input is obtained, the system must traverse the input string character by character. This requires setting up appropriate loop structures and counter variables to iterate through the string.

Identifying occurrences of the specified character to be replaced and replacing them with the replacement character is the core algorithmic task. This involves comparing each character in the string with the specified character and replacing it when a match is found.

Given the constraints of the 8086 architecture, efficient memory management is crucial. The system should allocate memory dynamically as needed, ensuring optimal usage and avoiding memory leaks.[5]

Robust error handling mechanisms are essential to handle various scenarios, including invalid user input, insufficient memory, or unexpected program behavior. Error messages should provide informative feedback to the user.

Thorough testing is necessary to ensure the correctness and reliability of the system. Test cases covering different input scenarios should be devised and executed to validate the system's functionality.

Proper documentation and inline comments within the assembly code are vital for understanding the system's implementation. This includes explanations of key functions, data structures, and algorithms used.

Optimization techniques such as loop unrolling, register usage optimization, and code refactoring should be employed to enhance the system's performance and efficiency.

Designing the system in a modular and scalable manner facilitates future enhancements and modifications. Separating different components into reusable modules promotes code maintainability and extensibility.[5]

Upon successfully replacing all occurrences of the specified character within the string, the modified string should be displayed as output to the user. The output should be presented in a clear and readable format.

Summarize the objectives, requirements, and strategies outlined in the problem statement, emphasizing the importance of creating a robust and efficient system capable of handling various scenarios gracefully.

By elaborating on each aspect of the problem statement in detail, we ensure a comprehensive understanding of the task and lay the groundwork for designing and implementing a high-quality solution in 8086 assembly language.[4]

3. Methodology

INSTRUCTION SET USED IN THE CODE :

1. Data Movement Instructions:

- `MOV`: Used for moving data between registers, memory, and immediate values.
  - Examples: `MOV AX, data`, `MOV DS, AX`, `MOV [DI], BH`.

2. String Instructions:

- `LEA`: Load Effective Address, used for loading the effective address of a memory operand into a register.
  - Example: `LEA DX, STR1`.

3. Input/Output Instructions:

- `INT`: Software Interrupt, used for invoking DOS interrupts to perform input/output operations and other system services.
  - Examples: `INT 21H` for displaying strings and reading character input.

4. Arithmetic Instructions:

- `INC`: Increment, used for incrementing the value of a register or memory location.
  - Example: `INC DI`.

5. Comparison Instructions:

- `CMP`: Compare, used for comparing two operands.
  - Example: `CMP [DI], AL`.

6. Control Transfer Instructions:

- `JZ`: Jump if Zero, used for conditional branching based on the state of the Zero Flag (ZF).
  - Example: `JZ END_LOOP`.

7. Program Control Instructions:

- `END`: Marks the end of the program.

REGISTRES USED IN THE CODE :

1. General-Purpose Registers:

- `AX`: Accumulator register, used for general arithmetic operations and data movement.
- `BX`: Base register, often used as a pointer or for addressing memory locations.
- `CX`: Count register, commonly used as a loop counter.
- `DX`: Data register, used for data operations and I/O port access.

2. Index Register:

- `DI`: Destination Index register, commonly used as a pointer for string operations and memory manipulation.

3. Segment Registers:

- `DS`: Data Segment register, used to point to the segment where the data is stored.
- `ES`: Extra Segment register, used as an additional segment register for certain memory operations.

4. Status Register:

- `ZF`: Zero Flag, used to indicate the result of arithmetic and logical operations (set if the result is zero).

5. Flags Register:

   - Other flags may be implicitly used during execution, such as the Carry Flag (CF), Sign Flag (SF), etc., but they are not explicitly manipulated in the provided code.

ADDRESSING MODES USED IN THE CODE :

1. Immediate Addressing Mode:

   - This mode is used when immediate data is directly specified in the instruction.

   - Example: `MOV AH, 09H`, `MOV AL, BL`, `MOV AL, 01H`.

2. Register Addressing Mode:

   - In this mode, the operand is the content of a register.

   - Examples: `MOV AX, data`, `MOV DS, AX`, `MOV BL, AL`.

3. Direct Addressing Mode:

   - The operand is located at a specific memory address.

   - Example: `MOV [DI], BH`, `CMP [DI], AL`.

4. Register Indirect Addressing Mode:

   - The effective address is determined by the content of a register.

   - Example: `LEA DX, STR1`, `MOV [DI], BH`.

5. Based Addressing Mode:

- The effective address is formed by adding an offset value to the content of a base register.

- Example: `LEA DI, STR1 + 2`.

6. Indexed Addressing Mode:

- The effective address is formed by adding an offset value to the content of an index register.

- Example: `INC DI`.

7. Based Indexed Addressing Mode:

- The effective address is formed by adding an offset value to the content of a base register and an index register.

- Example: `MOV [DI], BH`.

8. Relative Addressing Mode:

- The effective address is determined relative to the current instruction pointer or program counter.

- This mode is not explicitly used in the provided code.

BRANCH LOOPING INSTRUCTIONS USED IN THIS CODE :

1. Conditional Jump Instructions:

- `JZ`: Jump if Zero

  - This instruction jumps to a specified location if the Zero Flag (ZF) is set, indicating that the result of the previous operation was zero.

  - Used in the `CMP CX, 0` instruction to check if the loop counter `CX` is zero.

2. Unconditional Jump Instructions:

- `JMP`: Jump

  - This instruction unconditionally jumps to a specified location.

  - Not explicitly used in the provided code, as the code flows sequentially, but could be used for branching in more complex programs.

3. Looping Instructions:

- `LOOP`: Loop

  - This instruction decrements the loop counter (`CX`) and jumps to the specified location until the loop counter becomes zero.

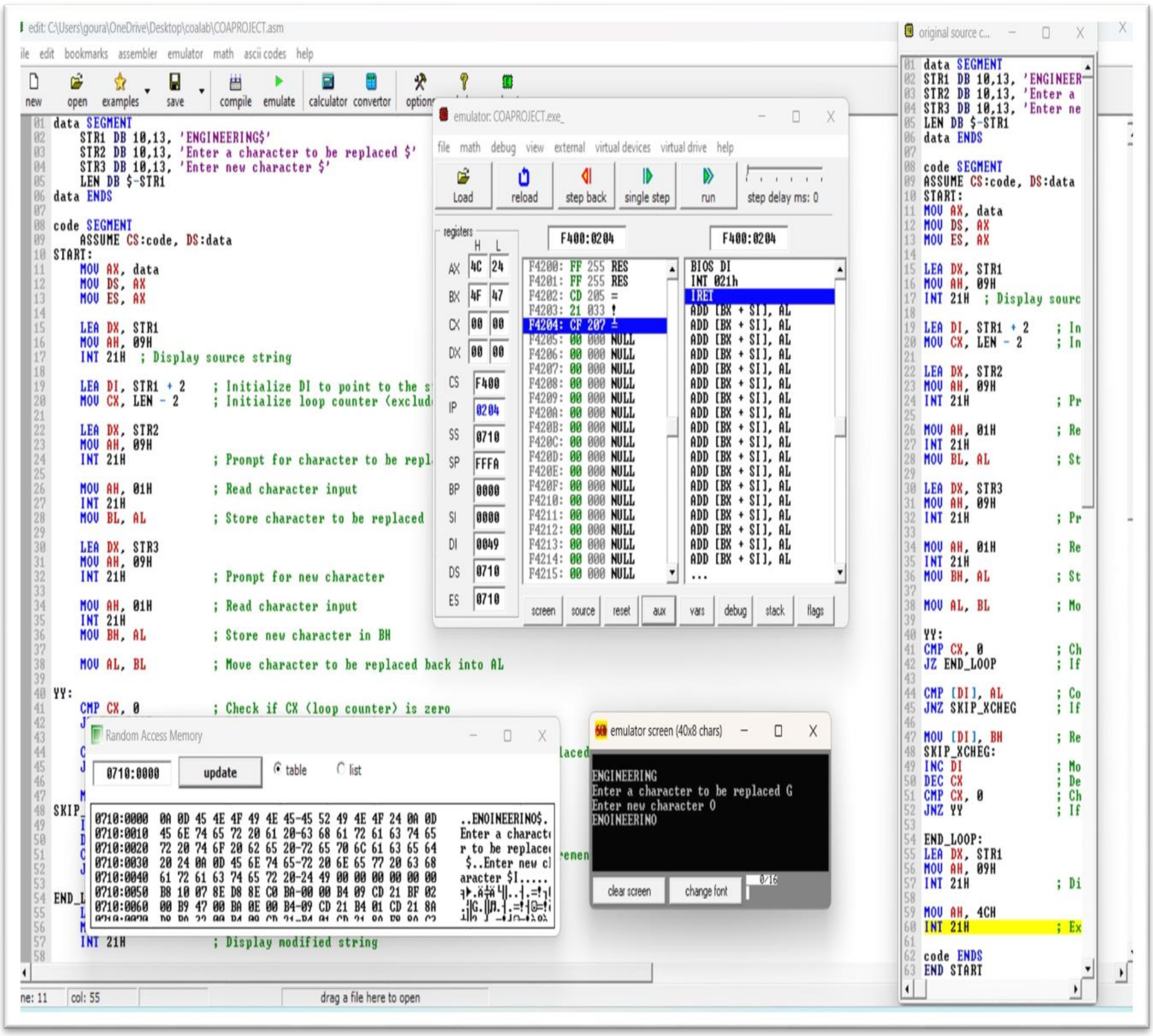
  - Used implicitly in the provided code with the combination of decrementing `CX` and conditional jump instruction (`JZ END_LOOP`) to create a loop for character replacement.

4. Implementation

CODE :

```
data SEGMENT
    STR1 DB 10,13, 'ENGINEERING$'
    STR2 DB 10,13, 'Enter a character to be replaced $'
    STR3 DB 10,13, 'Enter new character $'
    LEN DB $-STR1
data ENDS

code SEGMENT
    ASSUME CS:code, DS:data
START:
    MOV AX, data
    MOV DS, AX
    MOV ES, AX

    LEA DX, STR1
    MOV AH, 09H
    INT 21H  ; Display source string

    LEA DI, STR1 + 2     ; Initialize DI to point to the start of
STR1
    MOV CX, LEN - 2     ; Initialize loop counter (exclude CR and LF
characters)

    LEA DX, STR2
    MOV AH, 09H
    INT 21H             ; Prompt for character to be replaced

    MOV AH, 01H         ; Read character input
    INT 21H
    MOV BL, AL          ; Store character to be replaced in BL

    LEA DX, STR3
    MOV AH, 09H
    INT 21H             ; Prompt for new character

    MOV AH, 01H         ; Read character input
    INT 21H
    MOV BH, AL          ; Store new character in BH

    MOV AL, BL          ; Move character to be replaced back into AL

YY:
    CMP CX, 0           ; Check if CX (loop counter) is zero
    JZ END_LOOP         ; If it's zero, end the loop
    CMP [DI], AL          ; Compare current character with character
to be replaced
    JNZ SKIP_XCHEG      ; If not equal, skip replacement

    MOV [DI], BH        ; Replace character
SKIP_XCHEG:
    INC DI              ; Move to the next character
    DEC CX              ; Decrement loop counter
    CMP CX, 0             ; Check if CX (loop counter) is zero again
after decrement
    JNZ YY              ; If it's not zero, repeat loop

END_LOOP:
    LEA DX, STR1
    MOV AH, 09H
    INT 21H             ; Display modified string

    MOV AH, 4CH
    INT 21H             ; Exit program

code ENDS
END START
```

5. Results & Interpretation



edit: C:\Users\goura\OneDrive\Desktop\coalab\COAPROJECT.asm

```
data SEGMENT
    STR1 DB 10,13, 'ENGINEERING$'
    STR2 DB 10,13, 'Enter a character to be replaced $'
    STR3 DB 10,13, 'Enter new character $'
    LEN DB $-STR1
data ENDS

code SEGMENT
    ASSUME CS:code, DS:data
START:
    MOV AX, data
    MOV DS, AX
    MOV ES, AX

    LEA DX, STR1
    MOV AH, 09H
    INT 21H  ; Display source string

    LEA DI, STR1 + 2      ; Initialize DI to point to the s
    MOV CX, LEN - 2       ; Initialize loop counter (exclud

    LEA DX, STR2
    MOV AH, 09H
    INT 21H               ; Prompt for character to be repl

    MOV AH, 01H           ; Read character input
    INT 21H
    MOV BL, AL            ; Store character to be replaced

    LEA DX, STR3
    MOV AH, 09H
    INT 21H               ; Prompt for new character

    MOV AH, 01H           ; Read character input
    INT 21H
    MOV BH, AL            ; Store new character in BH

    MOV AL, BL            ; Move character to be replaced back into AL

YY:
    CMP CX, 0             ; Check if CX (loop counter) is zero
    J
    C                                                   laced
    J
    M
SKIP_
    I
    D
    C                                                   emen
    J

END_L
    L
    M
    INT 21H               ; Display modified string
```

line: 11   col: 55   drag a file here to open

emulator: COAPROJECT.exe_

```
registers    H   L
AX          4C  24
BX          4F  47
CX          00  00
DX          00  00
CS          F400
IP          0204
SS          0710
SP          FFFA
BP          0000
SI          0000
DI          0049
DS          0710
ES          0710
```

```
F400:0204                   F400:0204
F4200: FF 255 RES           BIOS DI
F4201: FF 255 RES           INT 021h
F4202: CD 205 =             IRET
F4203: 21 033 !             ADD [BX + SI], AL
F4204: CF 207 ±             ADD [BX + SI], AL
F4205: 00 000 NULL          ADD [BX + SI], AL
F4206: 00 000 NULL          ADD [BX + SI], AL
F4207: 00 000 NULL          ADD [BX + SI], AL
F4208: 00 000 NULL          ADD [BX + SI], AL
F4209: 00 000 NULL          ADD [BX + SI], AL
F420A: 00 000 NULL          ADD [BX + SI], AL
F420B: 00 000 NULL          ADD [BX + SI], AL
F420C: 00 000 NULL          ADD [BX + SI], AL
F420D: 00 000 NULL          ADD [BX + SI], AL
F420E: 00 000 NULL          ADD [BX + SI], AL
F420F: 00 000 NULL          ADD [BX + SI], AL
F4210: 00 000 NULL          ADD [BX + SI], AL
F4211: 00 000 NULL          ADD [BX + SI], AL
F4212: 00 000 NULL          ADD [BX + SI], AL
F4213: 00 000 NULL          ADD [BX + SI], AL
F4214: 00 000 NULL          ADD [BX + SI], AL
F4215: 00 000 NULL          ...
```

Random Access Memory — 0710:0000 — update — table / list

```
0710:0000  0A 0D 45 4E 4F 49 4E 45-45 52 49 4E 4F 24 0A 0D   ..ENOINEERINO$.
0710:0010  45 6E 74 65 72 20 61 20-63 68 61 72 61 63 74 65   Enter a characte
0710:0020  72 20 74 6F 20 62 65 20-72 65 70 6C 61 63 65 64   r to be replaced
0710:0030  20 24 0A 0D 45 6E 74 65-72 20 6E 65 77 20 63 68    $..Enter new ch
0710:0040  61 72 61 63 74 65 72 20-24 49 00 00 00 00 00 00   aracter $I......
0710:0050  B8 10 07 8E D8 8E C0 BA-00 00 B4 09 CD 21 BF 02
0710:0060  00 B9 47 00 BA 0E 00 B4-09 CD 21 B4 01 CD 21 8A
```

emulator screen (40x8 chars)

```
ENGINEERING
Enter a character to be replaced G
Enter new character O
ENOINEERINO
```

original source c...

```
data SEGMENT
STR1 DB 10,13, 'ENGINEER
STR2 DB 10,13, 'Enter a
STR3 DB 10,13, 'Enter ne
LEN DB $-STR1
data ENDS

code SEGMENT
ASSUME CS:code, DS:data
START:
MOV AX, data
MOV DS, AX
MOV ES, AX

LEA DX, STR1
MOV AH, 09H
INT 21H  ; Display sourc

LEA DI, STR1 + 2    ; In
MOV CX, LEN - 2     ; In

LEA DX, STR2
MOV AH, 09H
INT 21H             ; Pr

MOV AH, 01H         ; Re
INT 21H
MOV BL, AL          ; St

LEA DX, STR3
MOV AH, 09H
INT 21H             ; Pr

MOV AH, 01H         ; Re
INT 21H
MOV BH, AL          ; St

MOV AL, BL          ; Mo

YY:
CMP CX, 0           ; Ch
JZ END_LOOP         ; If

CMP [DI], AL        ; Co
JNZ SKIP_XCHEG      ; If

MOV [DI], BH        ; Re
SKIP_XCHEG:
INC DI              ; Mo
DEC CX              ; De
CMP CX, 0           ; Ch
JNZ YY              ; If

END_LOOP:
LEA DX, STR1
MOV AH, 09H
INT 21H             ; Di

MOV AH, 4CH
INT 21H             ; Ex

code ENDS
END START
```

6. Conclusion

In the realm of low-level programming, the development of a system in 8086 assembly language to replace characters within a string represents a significant undertaking that touches upon various facets of software engineering. This project has been an exploration into the complexities of assembly language programming, offering valuable insights into string manipulation, algorithm design, memory management, and error handling mechanisms.

Throughout the project lifecycle, from problem analysis to implementation, testing, and documentation, several key learnings have emerged. One of the foremost realizations is the importance of meticulous planning and design in the development process. By thoroughly analyzing the problem statement, understanding the requirements, and devising a structured methodology, we laid the groundwork for a systematic approach to system development.[4]

The implementation phase brought to light the intricacies of working within the constraints of the 8086 architecture. Efficient memory management, optimal register usage, and algorithmic efficiency became paramount considerations in achieving a functional and performant solution. Techniques such as loop unrolling, register allocation, and code optimization were employed to enhance the system's efficiency and speed.

Testing played a crucial role in validating the system's functionality, correctness, and robustness. Through comprehensive test case development and execution, we ensured that the system performed as expected under various input scenarios, including edge cases and boundary conditions. This rigorous testing regimen not only validated the system's reliability but also unearthed areas for improvement and optimization.[3]

Documentation emerged as a cornerstone of the project, providing comprehensive insights into the system's design, implementation details, and usage instructions. Clear and concise documentation, supplemented by inline comments in the assembly code, facilitated understanding, maintenance, and future enhancements of the system.

In conclusion, the completion of this project signifies more than just the development of a character replacement system in 8086 assembly language. It represents a journey of exploration, learning, and growth in the realm of low-level programming. The project has equipped us with a deeper understanding of assembly language programming concepts and techniques, honed our problem-solving skills, and instilled a sense of discipline and rigor in software development practices.

Moving forward, the knowledge and experience gained from this project will serve as a solid foundation for tackling more complex programming challenges and exploring advanced topics in low-level programming. Moreover, the project's outcomes, including the developed system and accompanying documentation, stand as a testament to our capabilities as software engineers and contribute to the collective body of knowledge in assembly language programming.

In essence, this project has been a journey of discovery and achievement, showcasing the power of determination, creativity, and technical acumen in overcoming challenges and achieving success in the world of low-level programming. As we conclude this project, we look forward to applying our newfound skills and insights to future endeavors, continuing our pursuit of excellence in software engineering.[2]

References

1. M. Rafiquzzaman, "Fundemetals of Digital Logic and Microcomputer Design (5th Ed.)". John Wiley & Sons, Inc, 2005.
2. Ahmed, A., & Farook, O. (1988, October). Software-firmware design for 8088/8086 microprocessor based systems utilizing XT/AT compatible personal computers. In *Proceedings Frontiers in Education Conference* (pp. 448-453). IEEE.
3. J. Yu, "Construction of Assembly Language Three-tier Experimental System Based on EMU8086," ISCTT 2021; 6th International Conference on Information Science, Computer Technology and Transportation, Xishuangbanna, China, 2021, pp. 1-5.
4. Y. Joshi, Y. Vyas, T. Ghone, V. Soni and L. D'Mello, "Emulation of Intel's 8086 Microprocessor using Rust and Web Assembly," 2022 3rd International Conference for Emerging Technology (INCET), Belgaum, India, 2022, pp. 1-8, doi: 10.1109/INCET54531.2022.9824078. keywords: {Performance evaluation;Operating systems;Microprocessors;Linux;Memory management;Emulation;User experience;Intel 8086;Microprocessor;Web Assembly;Rust;Online Compiler},
5. G. Mostafa, "Development of a 16-bit microprocessor learning system using Intel 8086 architecture," 2013 2nd International Conference on Advances in Electrical Engineering (ICAEE), Dkaka, Bangladesh, 2013, pp. 146-153, doi: 10.1109/ICAEE.2013.6750323. keywords: {Electronics packaging;Random access memory;Registers;Decoding;Microprocessors;Keyboards;Hardware;MicroTalk-8086 Trainer;Learning System;SDK-86;Monitor Program},

Appendices

Appendix A: Sample Input and Output

Sample Input:

- Input String: "Hello, World!"
- Character to be Replaced: 'l'
- Replacement Character: 'x'

Sample Output:

- Modified String: "Hexxo, Worxd!"

Appendix B: Error Handling

The system incorporates robust error handling mechanisms to address various potential issues, ensuring a smooth user experience and reliable operation. Common errors and their corresponding error messages include:

1. Invalid Input:
   - Error Message: "Invalid input. Please enter a valid string."
   - This message is displayed when the user enters an empty string or provides input that does not conform to the specified format.

2. Invalid Character:
   - Error Message: "Invalid character. Please enter a single character."
   - This message is shown when the user inputs a character other than a single character to be replaced or the replacement character.

3. String Exceeds Maximum Length:
   - Error Message: "String exceeds maximum length. Please enter a shorter string."

- This error occurs if the length of the input string exceeds the predefined maximum length, preventing buffer overflow.

4. Memory Allocation Failure:
   - Error Message: "Memory allocation failed. Please try again later."
   - If the system fails to allocate memory dynamically due to insufficient resources, this error message is displayed, prompting the user to retry later.

Appendix C: Optimization Techniques

To enhance the efficiency and performance of the system, several optimization techniques are employed:

1. Loop Unrolling:
   - Loop unrolling is applied to reduce loop overhead and improve execution speed. Instead of processing one character at a time, multiple characters are processed within each iteration of the loop.

2. Register Allocation:
   - Efficient register allocation minimizes memory accesses and reduces latency. Critical variables and loop counters are stored in registers, optimizing access times and improving overall performance.

3. Code Refactoring:
   - Code refactoring techniques are utilized to streamline the implementation, eliminate redundant operations, and improve code readability. Complex algorithms are decomposed into smaller, more manageable functions, enhancing maintainability and facilitating future optimizations.

Appendix D: Test Cases

A variety of test cases are devised to validate the system's functionality under different input scenarios:

1. Basic Test Case:
   - Input: "Hello, World!", 'l', 'x'
   - Expected Output: "Hexxo, Worxd!"

2. Edge Case - Empty String:
   - Input: "", 'a', 'b'
   - Expected Output: ""

3. Edge Case - String Without Target Character:
   - Input: "Hello, World!", 'x', 'y'
   - Expected Output: "Hello, World!"

4. Boundary Case - Maximum Length String:
- Input: (Maximum Length String), 'a', 'b'
   - Expected Output: (Modified Maximum Length String)

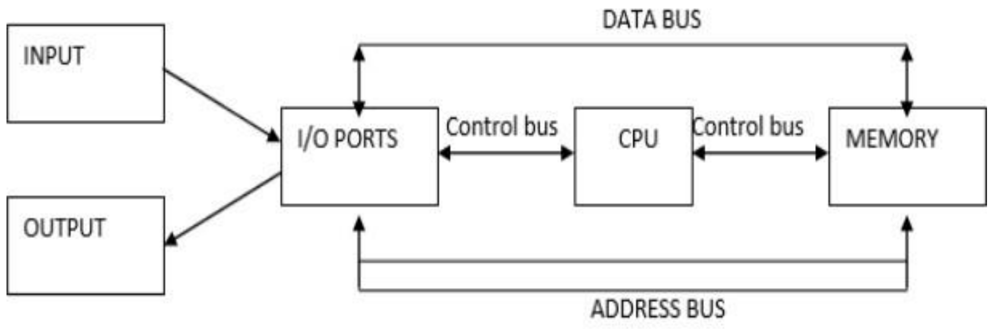
Appendix E: Assembly Code Snippets

Selected snippets of assembly code illustrating key components of the system, including input handling, string traversal, character replacement algorithm, and output generation, are provided for reference and insight into the implementation details.

INTRODUCTION TO MICROPROCESSOR:

OVERVIEW OF A SIMPLE MICRO COMPUTER:

The major parts are the central processing unit or CPU, memory, and the input and output circuitry or I/O. Connecting these parts together are three sets of parallel lines called buses. The three buses are the address bus, the data bus, and the control bus.



Block diagram of simple computer or microcomputer.

i) **MEMORY:** The memory section usually consists of a mixture of RAM and ROM. It may also have magnetic floppy disks, magnetic hard disks, or laser optical disks. Memory has two purposes. The first purpose is to store the binary codes for the sequence of instructions you want the computer to carry out. When you write a computer program, what you are really doing is just writing a sequential list of instructions for the computer. The second purpose of the memory is to store the binary-coded data with which the computeris going to be working.

ii) **INPUT/OUTPUT:** The input/output or I/O section allows the computer to take in data from the outside world or send data to the outside world. These allow the user and the computer to communicate with each other. The actual physical devices used to interface the computer buses to external systems are oftencalled ports.

iii) **CPU:** The central processing unit or CPU controls the operation of the computer. It fetches binary-coded instruction of the computer. It fetches binary-coded instructions from memory, decodes the instructions into a series of simple actions, and carries out these actions. The CPU contains an arithmetic logic unit, or ALU. Which can perform add, subtract, OR, AND, invert, or exclusive-OR operations on binary words when instructed to do so. The CPU also contains an address counter which is used to hold the address of the next instruction or data to be fetched from memory, general-purpose registers which are used for temporary storage of binary data, and circuitry which generates the control bus signals.

iv) **ADDRESS BUS:** The address bus consists of 16, 20, 24, or more parallel signal lines. On these lines the CPU sends out the address of the memory location that is to be written to or read from. The number of address lines determines the number of memory locations that the CPU can address. If the CPU has N address lines then it can directly address 2 N memory locations.

v) **DATA BUS:** The data bus consists of 8, 16, 32 or more parallel signal lines. As indicated by the double- ended arrows on the data bus line, the data bus lines are bi-directional. This means that the CPU can read data in on these lines from memory or from a port as well as send data out on these lines to memory location or to a port. Many devices in a system will have their outputs connected to the data bus, but the outputs of only one device at a time will be enabled.

vi) **CONTROL BUS:** The control bus consists of 4-10 parallel signal lines. The CPU sends out signals on the control bus to enable the outputs of addressed memory devices or port devices. Typical control bus signals are memory read, memory write, I/O read, and I/O writer. To read a byte of data from a memory location, for example, the CPU sends out the address of the desired byte on the address bus and then sends out a memory read signal on the control bus.

What is a Microprocessor?

The word comes from the combination micro and processor.

- ✓ Processor means a device that processes numbers, specifically binary numbers, 0’s and 1’s. –
- ✓ Micro is a new addition.
- ✓ In the late 1960’s, processors were built using discrete elements.
- ✓ These devices performed the required operation, but were too large and too slow.
- ✓ In the early 1970’s the microchip was invented. All of the components that made up the processor were now placed on a single piece of silicon. The size became several thousand times smaller and thespeed became several hundred times faster.
- ✓ The “Micro” Processor was born.

Definition of Microprocessor: ¬

- ✓ Microprocessor is a multipurpose, programmable device that accepts digital data as input, processes it according to instructions stored in its memory, and provides results as output. or
- ✓ A microprocessor is a multipurpose, programmable, clock-driven, register-based electronic device that reads binary instructions from a storage device called memory accepts binary data as input and processes data according to instructions, and provides result as output.

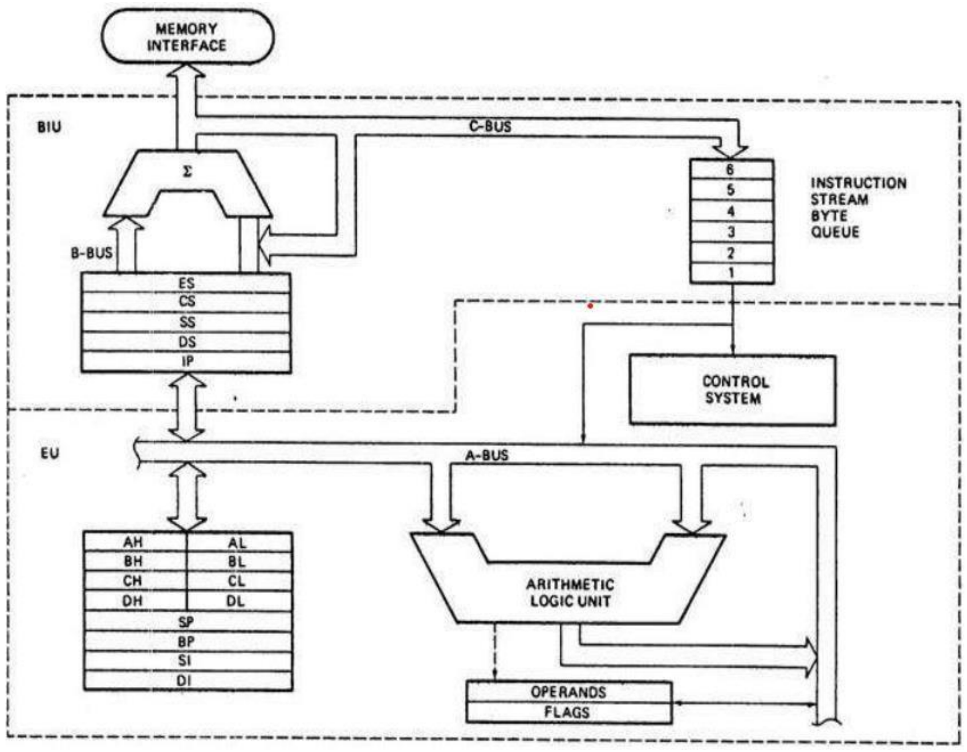
8086 microprocessor features:

- ✓ It is 16-bit microprocessor.
- ✓ It has a 16-bit data bus, so it can read data from or write data to memory and ports either 16-bit or8-bit at a time.
- ✓ It has 20-bit address bus and can access up to 220 memory locations (1 MB).
- ✓ It can support up to 64K I/O ports.
- ✓ It provides 14, 16-bit registers
- ✓ It has multiplexed address and data bus AD0-AD15 & A16-A19.
- ✓ It requires single phase clock with 33% duty cycle to provide internal timing.
- ✓ Prefetches up to 6 instruction bytes from memory and queues them in order to speed up theprocessing.
- ✓ 8086 supports 2 modes of operation:Minimum mode and Maximum mode

Architecture of 8086 microprocessor:

As shown in the below figure, the 8086 CPU is divided into two independent functional parts:

- ✓ Bus Interface Unit (BIU)
- ✓ Execution Unit (EU)



- ***The Execution Unit (EU):***

- ✓ The execution unit of the 8086 tells the BIU where to fetch instructions or data from, decodes instructions,and executes instructions.
- ✓ The EU contains control circuitry, which directs internal operations.
- ✓ A decoder in the EU translates instructions fetched from memory into a series of actions, which the EUcarries out.
- ✓ The EU has a 16-bit arithmetic logic unit (ALU) which can add, subtract, AND, OR, XOR, increment,decrement, complement or shift binary numbers.
- ✓ The main functions of EU are:
- ➢ Decoding of Instructions
- ➢ Execution of instructionsSteps:
  - ♣ EU extracts instructions from top of queue in BIU
  - ♣ Decode the instructions

♣ Generates operands if necessary

♣ Passes operands to BIU & requests it to perform read or write bus cycles to memory or I/O

♣ Perform the operation specified by the instruction on operands.

- ***Bus Interface Unit (BIU):***

✓ The BIU sends out addresses, fetches instructions from memory, reads data from ports and memory, andwrites data to ports and memory
In simple words, the BIU handles all transfers of data and addresses on the buses for the execution unit.

Register organization:

- 8086 has a powerful set of registers known as general purpose registers and special purpose registers.
- All of them are 16-bit registers.
- General purpose registers:

✓ These registers can be used as either 8-bit registers or 16-bit registers.

✓ They may be either used for holding data, variables and intermediate results temporarily or for other purposes like a counter or for storing offset address for some particular addressing modes etc.

- Special purpose registers:

✓ These registers are used as segment registers, pointers, index registers or as offset storage registersfor particular addressing modes.

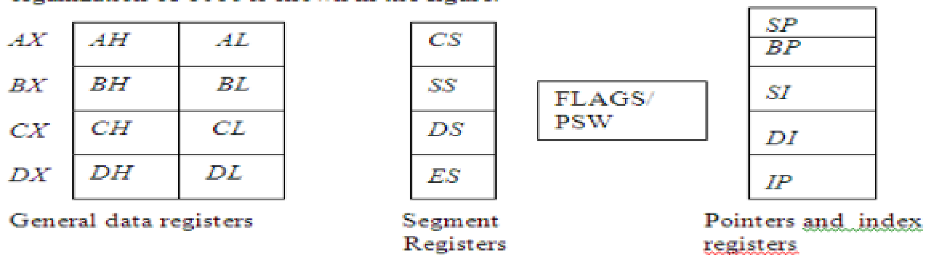
✓ The 8086 registers are classified into the following types:

General Data Registers

Segment Registers

Pointers and Index RegistersFlag Register

The register set of 8086 can be categorized into 4 different groups. The register organization of 8086 is shown in the figure.



Register organization of 8086

1. *General Data Registers:*

✓ The registers AX, BX, CX and DX are the general purpose 16-bit registers.

✓ AX is used as 16-bit accumulator. The lower 8-bit is designated as AL and higher 8-bit is designated as AH. AL can be used as an 8-bit accumulator for 8-bit operation.

✓ All data register can be used as either 16 bit or 8 bit. BX is a 16 bit register, but BL indicates thelower 8-bit of BX and BH indicates the higher 8-bit of BX.

✓ The register BX is used as offset storage for forming physical address in case of certain addressing modes.

✓ The register CX is used default counter in case of string and loop instructions.

✓ DX register is a general purpose register which may be used as an implicit operand or destinationin case of a few instructions.

2. *Segment Registers:*

- ✓ There are 4 segment registers. They are: Code Segment Register(CS) , Data Segment Register(DS)
  , Extra Segment Register(ES) , Stack Segment Register(SS) .
- ✓ The 8086 architecture uses the concept of segmented memory. 8086 able to address a memorycapacity of 1 megabyte and it is byte organized. This 1 megabyte memory is divided into 16 logical segments. Each segment contains 64 kbytes of memory.
- ✓ Code segment register (CS): is used for addressing memory location in the code segment of thememory, where the executable program is stored.
- ✓ Data segment register (DS): points to the data segment of the memory where the data is stored.
- ✓ Extra Segment Register (ES) : also refers to a segment in the memory which is another datasegment in the memory.
- ✓ Stack Segment Register (SS): is used for addressing stack segment of the memory. The stacksegment is that segment of memory which is used to store stack data.
- ✓ While addressing any location in the memory bank, the physical address is calculated from twoparts:
  Physical address= segment address + offset address
- ✓ The first is segment address, the segment registers contain 16-bit segment base addresses,related to different segment.
- ✓ The second part is the offset value in that segment.

3. *Pointers and Index Registers:*

- ✓ The index and pointer registers are given below:

  IP—Instruction pointer-store memory location of next instruction to be executed. BP—Base pointer

  SP—Stack pointer SI—Source index DI—Destination index
- ✓ The pointers registers contain offset within the particular segments. The pointer register IP contains offset within the code segment. The pointer register BP contains offset within the data segment. The pointer register SP contains offset within the stack segment.
- ✓ The index registers are used as general-purpose registers as well as for offset storage in case of indexed, base indexed and relative base indexed addressing modes.
- ✓ The register SI is used to store the offset of source data in data segment.
- ✓ The register DI is used to store the offset of destination in data or extra segment.
- ✓ The index registers are particularly useful for string manipulation. 8086 flag register and itsfunctions.
- ✓ The 8086 flag register contents indicate the results of computation in the ALU. It also containssome flag bits to control the CPU operations.

  A 16-bit flag register is used in 8086. It is divided into two parts: Condition code or status flags

  Machine control flags
- ✓ The condition code flag register is the lower byte of the 16-bit flag register. The condition code flag register is identical to 8085 flag register, with an additional overflow flag

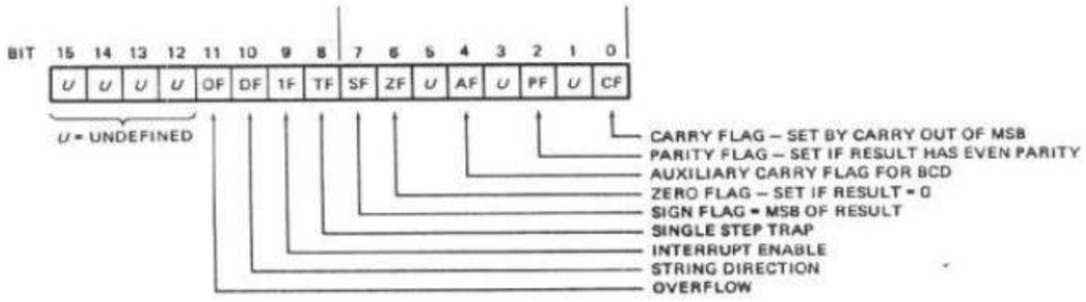
4. *8086 flag register and its functions:*

- ✓ The 8086 flag register contents indicate the results of computation in the ALU. It also containssome flag bits to control the CPU operations.
- ✓ A 16-bit flag register is used in 8086. It is divided into two parts: Condition code or status flags

  Machine control flags

- ✓ The condition code flag register is the lower byte of the 16-bit flag register. The condition code flag register is identical to 8085 flag register, with an additional overflow flag.
- ✓ The control flag register is the higher byte of the flag register. It contains three flags namelydirection flag (D), interrupt flag (I) and trap flag (T).

Flag register configuration


BIT
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
U
U
U
U
OF
DF
1F
TF
SF
ZF
U
AF
U
PF
U
CF
U = UNDEFINED
CARRY FLAG – SET BY CARRY OUT OF MSB
PARITY FLAG – SET IF RESULT HAS EVEN PARITY
AUXILIARY CARRY FLAG FOR BCD
ZERO FLAG – SET IF RESULT = 0
SIGN FLAG = MSB OF RESULT
SINGLE STEP TRAP
INTERRUPT ENABLE
STRING DIRECTION
OVERFLOW

The description of each flag bit is as follows:

***SF*- Sign Flag:** This flag is set, when the result of any computation is negative. For signed computations the sign flag equals the MSB of the result.

***ZF*- Zero Flag:** This flag is set, if the result of the computation or comparison performed by the previous instruction is zero.

***PF*- Parity Flag:** This flag is set to 1, if the lower byte of the result contains even number of 1's.

***CF*- Carry Flag:** This flag is set, when there is a carry out of MSB in case of addition or a borrow in case of subtraction.

***AF*-Auxilary Carry Flag:** This is set, if there is a carry from the lowest nibble, i.e, bit three during addition, or borrow for the lowest nibble, i.e, bit three, during subtraction.

***OF*- Over flow Flag:** This flag is set, if an overflow occurs, i.e, if the result of a signed operation is large enough to accommodate in a destination register. The result is of more than 7-bits in size in case of 8-bit signed operation and more than 15-bits in size in case of 16-bit sign operations, and then the overflow will be set.

***TF*- Tarp Flag:** If this flag is set, the processor enters the single step execution mode. The processor executes the current instruction and the control is transferred to the Trap interrupt service routine.

***IF*- Interrupt Flag:** If this flag is set, the mask able interrupts are recognized by the CPU, otherwise they are ignored.

***D*- Direction Flag:** This is used by string manipulation instructions. If this flag bit is '0', the string is processed beginning from the lowest address to the highest address, i.e., auto incrementing mode. Otherwise, the string is processed from the highest address towards the lowest address, i.e., auto decrementing mode.

Memory Segmentation:

- ➢ The memory in an 8086 based system is organized as segmented memory.
- ➢ The CPU 8086 is able to access 1MB of physical memory. The complete 1MB of memory can be divided into 16 segments, each of 64KB size and is addressed by one of the segment register.
- ➢ The 16-bit contents of the segment register actually point to the starting location of a particular segment. The address of the segments may be assigned as 0000H to F000h respectively.
- ➢ To address a specific memory location within a segment, we need an offset address. The offset address values are from 0000H to FFFFH so that the physical addresses range from 00000H to FFFFFH.

Physical address is calculated as below:

Ex: Segment address -------→ 1005H

Offset address ----------→ 5555H

Segment address -------→ 1005H ----- 0001 0000 0000 0101

Shifted left by 4 Positions------ 0001 0000 0000 0101 0000

+

Offset address --- 5555H ------ 0101 0101 0101 0101

Physical address -------155A5H 0001 0101 0101 1010 0101

Physical address = Segment address * 10H + Offset address.

The main advantages of the segmented memory scheme are as follows:

1. Allows the memory capacity to be 1MB although the actual addresses to be handled are of 16-bit size.
2. Allows the placing of code, data and stack portions of the same program in different parts (segments) of memory, for data and code protection.
3. Permits a program and/or its data to be put into different areas of memory each time the program is executed, i.e., provision for relocation is done.

Overlapping and Non-overlapping Memory segments:

➢ In the overlapping area locations physical address = CS1+IP1 = CS2+IP2. Where '+' indicates the procedure of physical address formation.

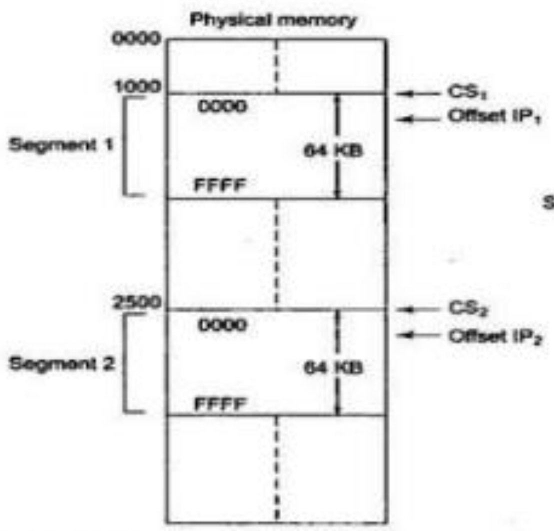


Fig. 1.3(a) *Non-overlapping Segments*

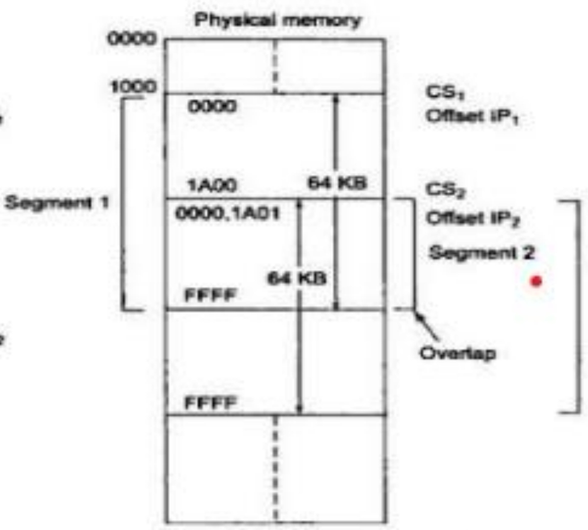


Fig. 1.3(b) *Overlapping Segments*

Addressing modes of 8086:

- Addressing mode indicates a way of locating data or operands.
- The addressing modes describe the types of operands and the way they are accessed for executing an instruction.
- According to the flow of instruction execution, the instructions may be categorized as
  i) Sequential control flow instructions and
  ii) Control transfer instructions

Sequential control flow instructions are the instructions, which after execution, transfer control to the next instruction appearing immediately after it (in the sequence) in the program. For example, the arithmetic, logic, data transfer and processor control instructions are sequential control flow instructions.

The **control transfer instructions**, on the other hand, transfer control to some predefined address or the address somehow specified in the instruction, after their execution. For example, INT, CALL, RET and JUMP instructions fall under this category.

The addressing modes for sequential control transfer instructions are:

1. **Immediate:** In this type of addressing, immediate data is a part of instruction and appears in the form of successive byte or bytes.

Ex: MOV AX, 0005H

In the above example, 0005H is the immediate data. The immediate data may be 8-bit or 16-bit in size.

2. **Direct:** In the direct addressing mode a 16-bit memory address (offset) is directly specified in the instruction as a part of it.

Ex: MOV AX, [5000H]

Here, data resides in a memory location in the data segment, whose effective address may be completed using 5000H as the offset address and content of DS as segment address. The effective address here, is 10H * DS + 5000H.

3. **Register:** In register addressing mode, the data is stored in a register and is referred using the particular register. All the registers, except IP, may be used in this mode.

Ex: MOV BX, AX

4. **Register Indirect:** Sometimes, the address of the memory location, which contains data or operand, is determined in an indirect way, using the offset register. This mode of addressing is known as register indirect mode. In this addressing mode, the offset address of data is in either BX or SI or DI register. The default segment is either DS or ES. The data is supposed to be available at the address pointed to by the content of any of the above registers in the default data segment.

Ex: MOV AX, [BX]

Here, data is present in a memory location in DS whose offset address is in BX. The effective address of the data is given as 10H * DS+[BX].

5. **Indexed:** In this addressing mode, offset of the operand is stored in one of the index registers. DS and ES are the default segments for index registers, SI and DI respectively. This is a special case of register indirect addressing mode.

Ex: MOV AX, [SI]

Here, data is available at an offset address stored in SI in DS. The effective address, in this case, is computed as 10*DS+[SI].

6. **Register Relative:** In this addressing mode, the data is available at an effective address formed by adding an 8-bit or 16-bit displacement with the content of any one of the registers BX, BP, SI and DI in the default (either DS or ES) segment.

Ex: MOV AX, 50H[BX]

Here, the effective address is given as 10H *DS+50H+[BX]

7. **Based Indexed:** The effective address of data is formed, in this addressing mode, by adding content of a base register (any one of BX or BP) to the content of an index register (any one of SI or DI). The default segment register may be ES or DS.

Ex: MOV AX, [BX][SI]

Here, BX is the base register and SI is the index register the effective address is computed as 10H * DS + [BX] + [SI].

8. **Relative Based Indexed:** The effective address is formed by adding an 8 or 16-bit displacement with the sum of the contents of any one of the base register (BX or BP) and any one of the index register, in a default segment.

Ex: MOV AX, 50H [BX] [SI]

Here, 50H is an immediate displacement, BX is base register and SI is an index register the effective address of data is computed as

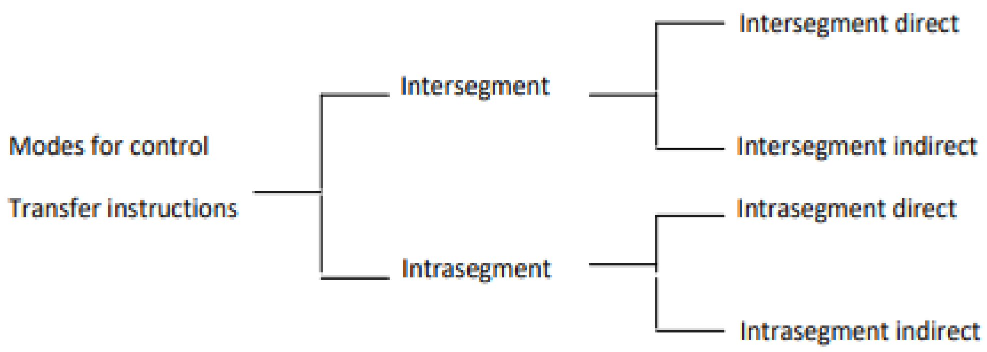
10H * DS + [BX] + [SI] + 50H

For control transfer instructions, the addressing modes depend upon whether the destination is within the same segment or different one. It also depends upon the method of passing the destination address to the processor.

Basically, there are two addressing modes for the control transfer instructions, **intersegment** addressing and **intrasegment** addressing modes.

If the location to which the control is to be transferred lies in a different segment other than the current one, the mode is called intersegment mode.

If the destination location lies in the same segment, the mode is called intrasegment mode.



Addressing modes for Control Transfer Instructions

9. **Intrasegment Direct Mode:** In this mode, the address to which the control is to be transferred lies in the same segment in which the control transfer instruction lies and appears directly in the instruction as an immediate displacement value. In this addressing mode, the displacement is computed relative to the content of the instruction pointer IP.

The effective address to which the control will be transferred is given by the sum of 8 or 16-bit displacement and current content of IP. In the case of jump instruction, if the signed displacement (d) is of 8-bits (i.e –128<d<+128) we term it as short jump and if it is of 16-bits (i.e-32, 768<d<+32,768) it is termed as long jump.

10. **Intrasegment Indirect Mode:** In this mode, the displacement to which the control is to be transferred, is in the same segment in which the control transfer instruction lies, but it is passed to the instruction indirectly. Here, the branch address is found as the content of a register or a memory location. This addressing mode may be used in unconditional branch instructions.

11. **Intersegment Direct:** In this mode, the address to which the control is to be transferred is in a different segment. This addressing mode provides a means of branching from one code segment to another code segment. Here, the CS and IP of the destination address are specified directly in the instruction.

12. **Intersegment Indirect:** In this mode, the address to which the control is to be transferred lies in a different segment and it is passed to the instruction indirectly, i.e contents of a memory block containing four bytes, i.e IP (LSB), IP(MSB), CS(LSB) and CS (MSB) sequentially. The starting address of the memory block may be referred using any of the addressing modes, except immediate mode.