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

Design a system that searches a character in a string using 8086 assembly language

<u>**Submitted by**</u>

Name 01: Siva Kumar Dash	**Regd No.: 2241004186**
Name 02: Smruti Sikha Dash	**Regd No.: 2241004187**
Name 03: Soumyadeep Dash	**Regd No.: 2241004188**
Name 04: Subha Samikshya Dash	**Regd No.: 2241004191**
Name 05: Shakiba Fatima	**Regd No.: 2241006002**

B. Tech. CSE 4th Semester (Section - 2241029)

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 AND ENGINEERING)** Department hereby declare that we own the full responsibility for the information, results etc. provided in this PROJECT titled "**(*Design a system that searches a character in a 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.

SIVA KUMAR DASH	**SMRUTI SIKHA DASH**
Registration No.: 2241004186	**Registration No.: 2241004187**
SOUMYADEEP DASH	**SUBHA SAMIKSHYA DASH**
Registration No.: 2241004188	**Registration No.: 2241004191**
SHAKIBA FATIMA	
Registration No.: 2241006002	

DATE:

PLACE: BHUBANESHWAR

Abstract

This project aims to design a pattern searching system in 8086 assembly language. The system will take input from the user for a string and a character to be searched within that string. Using a Brute Force algorithm, the system will scan the string character by character to find occurrences of the character. Upon successful detection, it will return the position(s) where the character is found within the string. The implementation will utilize low-level programming constructs such as memory manipulation and conditional branching to achieve efficient character search. The final system will provide a foundational tool for string manipulation tasks on legacy systems or environments where 8086 assembly language is still relevant.

Contents

1. Introduction

In the realm of low-level programming, where efficiency and resource optimization are paramount, the task of character searching in a string holds significant importance. In this project, we delve into the realm of 8086 assembly language, a foundational language for many early computing systems, to design a system for character searching within a given string. Leveraging the architecture's capabilities and constraints, we aim to construct an efficient and robust solution that can locate occurrences of a specified character within a provided string.

Character searching is a fundamental problem encountered across various domains, ranging from text processing and data mining to bioinformatics and compiler construction. By implementing such functionality in assembly language, we gain insights into the intricacies of low-level computation, memory management, and algorithmic design.

Throughout this project, we will navigate the complexities of 8086 assembly language, devising algorithms that optimize both time and space efficiency within the constraints of the architecture. Our goal is to develop a reliable character searching system that demonstrates the power and versatility of assembly language programming, showcasing its relevance even in contemporary computing scenarios. Through meticulous design and rigorous testing, we aim to construct a solution that not only meets the specified requirements but also serves as a testament to the enduring legacy of assembly language in the realm of computer science.

2. Problem Statement

The objective of this project is to design and implement a character searching system using 8086 assembly language. The system should be capable of efficiently locating a specified character within a given string and reporting its position if found, or indicating if the character is not present in the string.

This project aims to develop a character searching system in 8086 assembly language, addressing the need for efficient pattern search capabilities within the constraints of the 8086 architectures. Through this project, the goal is to provide a practical solution for character searching tasks in legacy systems or educational contexts where assembly language programming is still relevant.

3. Methodology

The provided assembly language program written for an x86 architecture in real mode uses various components and concepts. Here's a breakdown of the elements used in the program:

Directives and Segments:

1. .model small: Specifies the memory model. The 'small' model uses one segment each for code and data.

2. .stack 100h: Allocates 256 bytes (100h in hexadecimal) for the stack.

Data Segment:

1. text db 'Hello world': Defines a byte array (string) with the text "Hello world".

2. count dw 13: Defines a word (2 bytes) that stores the length of the string (13 characters).

3. search db 'x': Defines a byte with the character 'x', which is the character to search for in the string.

4. found db 'String Found$': Defines a string that will be displayed if the search character is found.

5. notfound db 'String NOT Found$': Defines a string that will be displayed if the search character is not found.

Code Segment:

1. begin: Label marking the start of the code.

Registers and Instructions:

1. mov ax,@data: Loads the address of the data segment into the AX register.

2. mov ds,ax and mov es,ax: Sets the DS (data segment) and ES (extra segment) registers to the address of the data segment.

3. mov cx,count: Loads the value of 'count' (13) into the CX register, which will be used as a counter for the string length.

4. mov di,offset text: Loads the offset address of the 'text' string into the DI register.

5. mov al,search: Loads the search character 'x' into the AL register.

6. repne scasb: Repeats the 'scasb' instruction (scan string for byte in AL) while CX is not zero and the byte in AL is not found in the string.

7. jz yes: Jumps to the 'yes' label if the zero flag (ZF) is set, indicating the character was found.

8. mov dx,offset notfound: Loads the offset address of the 'notfound' string into the DX register.

9. mov ah,09: Sets the AH register to 09h, the DOS function code for displaying a string.

10. int 21h: Calls DOS interrupt 21h to execute the function (display string).

11. jmp over: Jumps to the 'over' label, skipping the 'yes' label block.

12. yes: Label for the block of code executed if the character is found.

13. mov dx,offset found: Loads the offset address of the 'found' string into the DX register.

14. mov ah,09: Sets the AH register to 09h for displaying the string.

15. int 21h: Calls DOS interrupt 21h to display the 'found' string.

16. over: Label marking the end of the conditional blocks.

17. mov ah,4ch: Sets the AH register to 4Ch, the DOS function code for program termination.

18. int 21h: Calls DOS interrupt 21h to terminate the program.

Concepts:

1. String Manipulation: Scanning through a string to find a specific character.

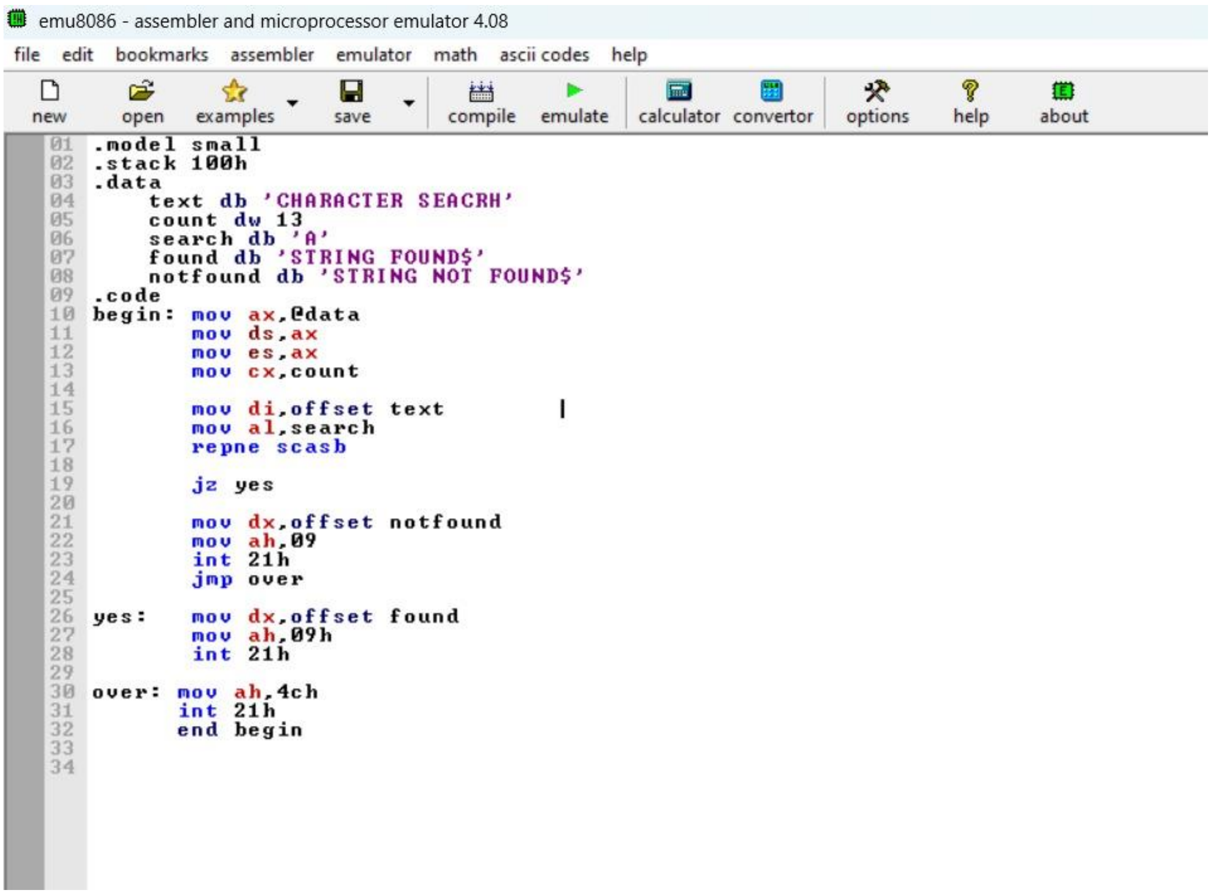
2. Interrupts: Using DOS interrupts to display strings and terminate the program.

3. Control Flow: Conditional branching based on the result of the string search.

Summary:

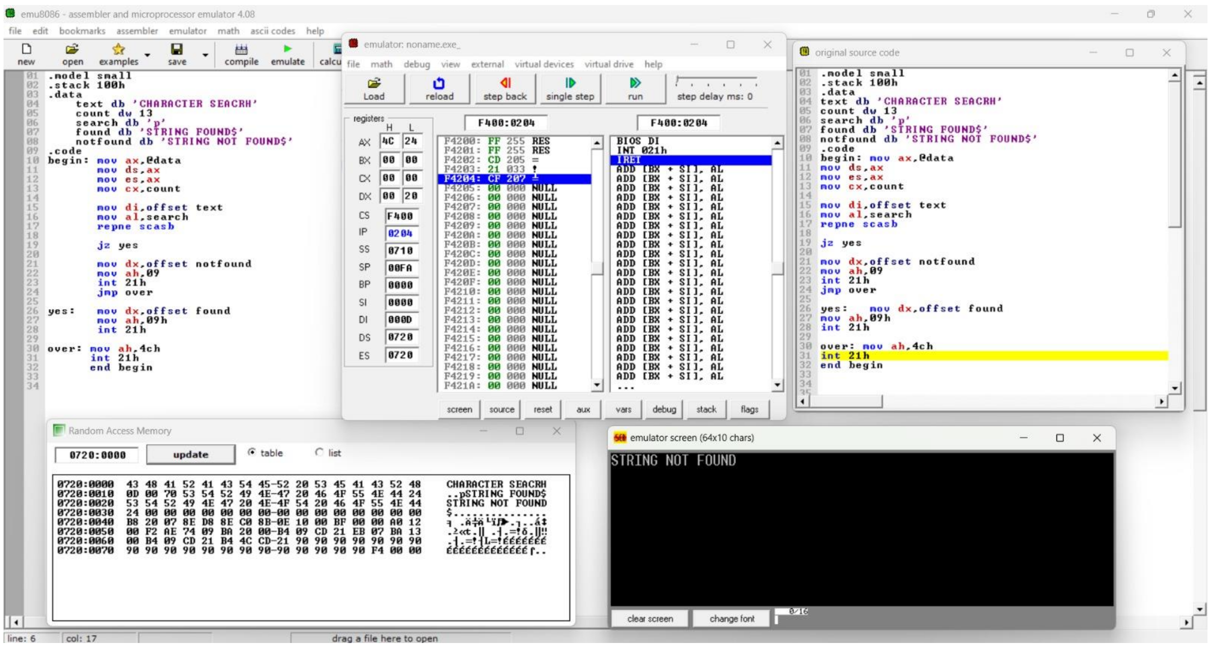
The program initializes the data segment, searches for the character 'x' in the string "Hello world", and displays a message indicating whether the character was found or not. Finally, it terminates the program.

4. Implementation


emu8086 - assembler and microprocessor emulator 4.08
file edit bookmarks assembler emulator math ascii codes help
new open examples save compile emulate calculator convertor options help about

```
.model small
.stack 100h
.data
    text db 'CHARACTER SEACRH'
    count dw 13
    search db 'A'
    found db 'STRING FOUND$'
    notfound db 'STRING NOT FOUND$'
.code
begin: mov ax,@data
       mov ds,ax
       mov es,ax
       mov cx,count

       mov di,offset text
       mov al,search
       repne scasb

       jz yes

       mov dx,offset notfound
       mov ah,09
       int 21h
       jmp over

yes:   mov dx,offset found
       mov ah,09h
       int 21h

over: mov ah,4ch
      int 21h
      end begin
```

5. Results & Interpretation



```
.model small
.stack 100h
.data
    text db 'CHARACTER SEACRH'
    count dw 13
    search db 'p'
    found db 'STRING FOUND$'
    notfound db 'STRING NOT FOUND$'
.code
begin: mov ax,@data
       mov ds,ax
       mov es,ax
       mov cx,count

       mov di,offset text
       mov al,search
       repne scasb

       jz yes

       mov dx,offset notfound
       mov ah,09
       int 21h
       jmp over

yes:   mov dx,offset found
       mov ah,09h
       int 21h

over: mov ah,4ch
      int 21h
      end begin
```

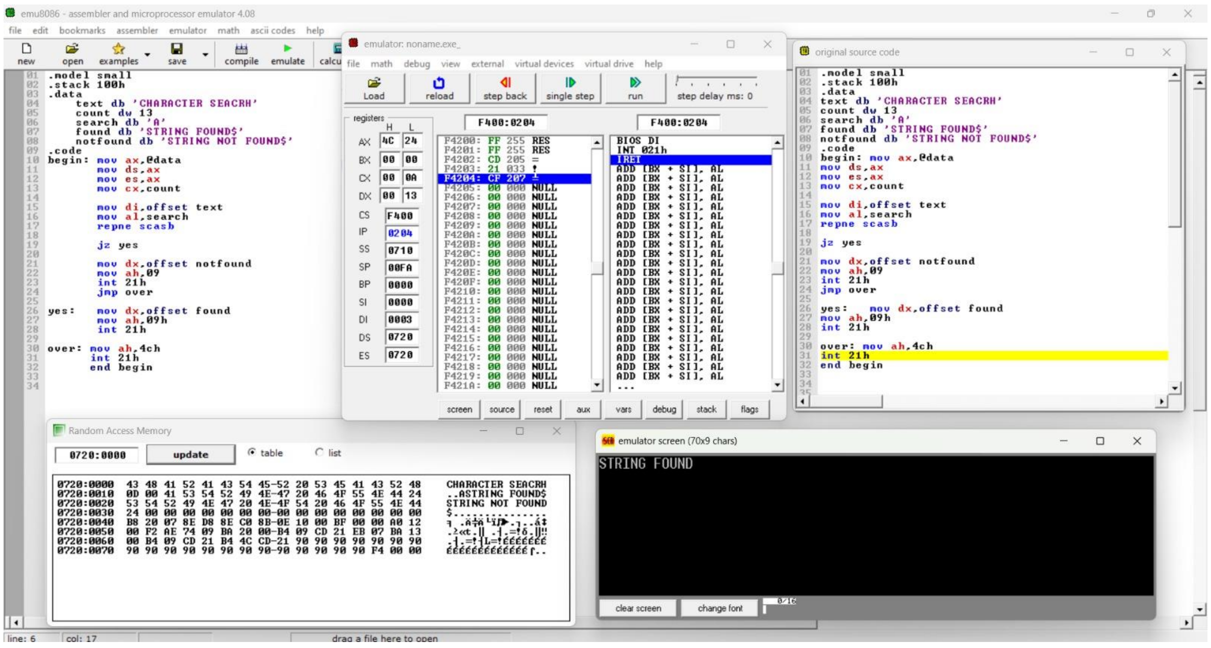
```
registers   H   L
AX         4C  24
BX         00  00
CX         00  00
DX         00  20
CS   F400
IP   0204
SS   0710
SP   00FA
BP   0000
SI   0000
DI   000D
DS   0720
ES   0720
```

```
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            ADD [BX + SI], AL
F4216: 00 000 NULL            ADD [BX + SI], AL
F4217: 00 000 NULL            ADD [BX + SI], AL
F4218: 00 000 NULL            ADD [BX + SI], AL
F4219: 00 000 NULL            ADD [BX + SI], AL
F421A: 00 000 NULL            ...
```

```
0720:0000
0720:0000  43 48 41 52 41 43 54 45-52 20 53 45 41 43 52 48   CHARACTER SEACRH
0720:0010  0D 00 70 53 54 52 49 4E-47 20 46 4F 55 4E 44 24   ..pSTRING FOUND$
0720:0020  53 54 52 49 4E 47 20 4E-4F 54 20 46 4F 55 4E 44   STRING NOT FOUND
0720:0030  24 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   $...............
0720:0040  B8 20 07 8E D8 8E C0 8B-0E 10 00 BF 00 00 A0 12   ╕ .Ä╪Ä└ï♫►.┐..á↕
0720:0050  00 F2 AE 74 09 BA 20 00-B4 09 CD 21 EB 07 BA 13   .≥«t.║ .┤.═!δ.║‼
0720:0060  00 B4 09 CD 21 B4 4C CD-21 90 90 90 90 90 90 90   .┤.═!┤L═!ÉÉÉÉÉÉÉ
0720:0070  90 90 90 90 90 90 90 90-90 90 90 90 90 F4 00 00   ÉÉÉÉÉÉÉÉÉÉÉÉÉ⌠..
```

emulator screen (64x10 chars)

```
STRING NOT FOUND
```



```
.model small
.stack 100h
.data
    text db 'CHARACTER SEACRH'
    count dw 13
    search db 'A'
    found db 'STRING FOUND$'
    notfound db 'STRING NOT FOUND$'
.code
begin: mov ax,@data
       mov ds,ax
       mov es,ax
       mov cx,count

       mov di,offset text
       mov al,search
       repne scasb

       jz yes

       mov dx,offset notfound
       mov ah,09
       int 21h
       jmp over

yes:   mov dx,offset found
       mov ah,09h
       int 21h

over: mov ah,4ch
      int 21h
      end begin
```

```
registers   H   L
AX         4C  24
BX         00  00
CX         00  0A
DX         00  13
CS   F400
IP   0204
SS   0710
SP   00FA
BP   0000
SI   0000
DI   0003
DS   0720
ES   0720
```

```
0720:0000
0720:0000  43 48 41 52 41 43 54 45-52 20 53 45 41 43 52 48   CHARACTER SEACRH
0720:0010  0D 00 41 53 54 52 49 4E-47 20 46 4F 55 4E 44 24   ..ASTRING FOUND$
0720:0020  53 54 52 49 4E 47 20 4E-4F 54 20 46 4F 55 4E 44   STRING NOT FOUND
0720:0030  24 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   $...............
0720:0040  B8 20 07 8E D8 8E C0 8B-0E 10 00 BF 00 00 A0 12   ╕ .Ä╪Ä└ï♫►.┐..á↕
0720:0050  00 F2 AE 74 09 BA 20 00-B4 09 CD 21 EB 07 BA 13   .≥«t.║ .┤.═!δ.║‼
0720:0060  00 B4 09 CD 21 B4 4C CD-21 90 90 90 90 90 90 90   .┤.═!┤L═!ÉÉÉÉÉÉÉ
0720:0070  90 90 90 90 90 90 90 90-90 90 90 90 90 F4 00 00   ÉÉÉÉÉÉÉÉÉÉÉÉÉ⌠..
```

emulator screen (70x9 chars)

```
STRING FOUND
```

6. Conclusion

In conclusion, the designed system for character searching in a string using 8086 assembly language provides a foundational framework for implementing such functionality. By breaking down the problem into distinct steps including input, search algorithm, and output, the system offers a structured approach to solving the character search problem.

The search algorithm, utilizing a simple yet effective Brute Force method, demonstrates the fundamental principles of character matching in assembly language. While this approach may not be the most efficient for larger datasets, it serves as a starting point for understanding the intricacies of character searching at a low level.

Furthermore, the system showcases the integration of assembly language with DOS interrupts for input/output operations, emphasizing the versatility of assembly language in interacting with the underlying system environment.

Overall, while the system provides a functional solution, there is room for improvement and optimization, such as implementing more advanced character matching algorithms or incorporating error handling mechanisms. Nevertheless, the system lays a solid foundation for exploring further enhancements and serves as a valuable learning resource for those interested in assembly language programming.

References

(as per the IEEE recommendations)

1. **COMPUTER ORGANIZATION AND ARCHITECTURE** Designing for Performance Tenth Editon by **WILLIAM STALLINGS**
2. https://youtu.be/V3qXAMiQmNQ?si=NC-uHwEVpalSrz2z
3. **Computer Organization and Design** by **David A. Patterson and John L. Hennessy**
4. **Structured Computer Organization** by **Andrew S. Tanenbaum**
5. **Computer Systems: A Programmer's Perspective** by **Randal E. Bryant and David R. O' Hallaron**

Appendices

8086 MICROPROCESSOR

The microprocessor 8085 followed by 8080, with a few more added features to it's architecture, which resulted in a functionally complete microprocessor.

• The main limitations of 8-bit microprocessor were

– Low speed,

– Low memory addressing capability

– Limited number of general purpose registers

– Less powerful instruction set

• All these limitations lead to the launching of 8086 microprocessor.

• In the family of 16 bit microprocessors, Intel's 8086 was the first one to be launched in 1978.

•The 8086 microprocessor has a much more powerful instruction set along with the architectural developments which imparts substantial programming flexibility and improvement in speed over the 8bit microprocessor.

• The peripheral chips designed earlier for 8085 were compatible with microprocessor 8086 with slight or no modifications.

Register organization of 8086

• 8086 has a powerful set of registers known as general purpose registers and special purpose registers.

• All of them are 16 bit registers.

• The general purpose 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 other purposes like a counter or for storing offset address for some particular addressing modes etc.

• The special purpose registers are used as segment registers, pointers, index registers or as offset storage registers for particular addressing modes.

• The register set is categorized into four groups, as follows:

– General data registers

– Segment registers

– Pointers and index registers

– Flag register

General data registers:

• Figure 1.1 shows the register organization of 8086.

• The registers AX, BX, CX and DX are the general purpose 16 bit registers. AH, AL

• AX is used as 16 bit accumulator   (AH, AL)

• AL can be used as an 8 bit accumulator for 8 bit operations. This is the most important general purpose register having multiple functions.

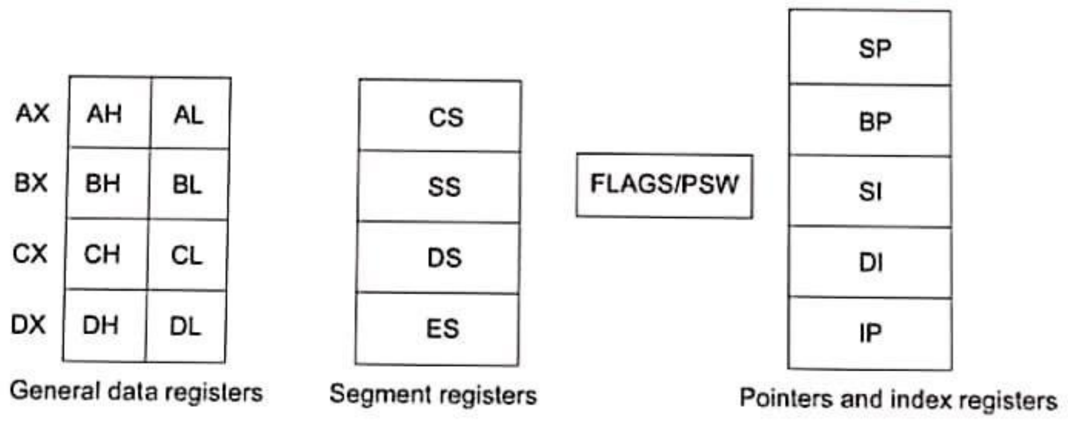


Fig. 1.1 ***Register organisation of 8086***

• Usually L and H specify the lower and higher bytes of a particular register.

• AX – accumulator,

BX – offset storage,

CX – counter,

DX – to store data

Segment Registers:

• Unlike 8085, the 8086 addresses segmented memory.

• The complete 1 megabyte memory, which the 8086 addresses, is divided into 16 logical segments.

• Each segment thus contains 64 k bytes of memory.

• There are 4 segment registers:- Code segment register (CS)   Code,

Data segment register (DS)   Data,

Extra segment register (ES)   data ,

Stack segment register (SS) Stack related data

• The CPU uses the stack for temporarily storing important data.

• While addressing any location in the memory bank, the physical address is calculated from two parts, the first is segment address and the second is offset.

• The segment registers contain 16 bit segment base addresses, related to different segments

• Any of the pointers and index registers or BX may contain the offset of the location to be addressed

• The advantage of this scheme is that instead of maintaining a 20 bit register for a physical address, the processor just maintains two 16 bit registers which are within the word length capacity of the machine.

• It may be noted that all these segments are logical segments.

Architecture of 8086

The architecture of 8086 supports a 16 bit ALU, a set of 16 bit registers and provides the segmented memory addressing capability, a rich instruction set, powerful interrupt structure, fetched instruction queue for overlapped fetching and execution etc.

• The internal block diagram, shown in Fig 1.2, describes the overall organization of different units inside the chip.

• The complete architecture of 8086 can be divided into two parts.

– Bus interface unit

– Execution Unit

• The bus interface unit contains the circuit for physical address calculations and a predecoding instruction byte queue (6 bytes long)

• The bus interface unit makes the system's bus signals available for external interfacing of the devices.

• In other words, this unit is responsible for establishing communications with external devices and peripherals including memory via the bus.

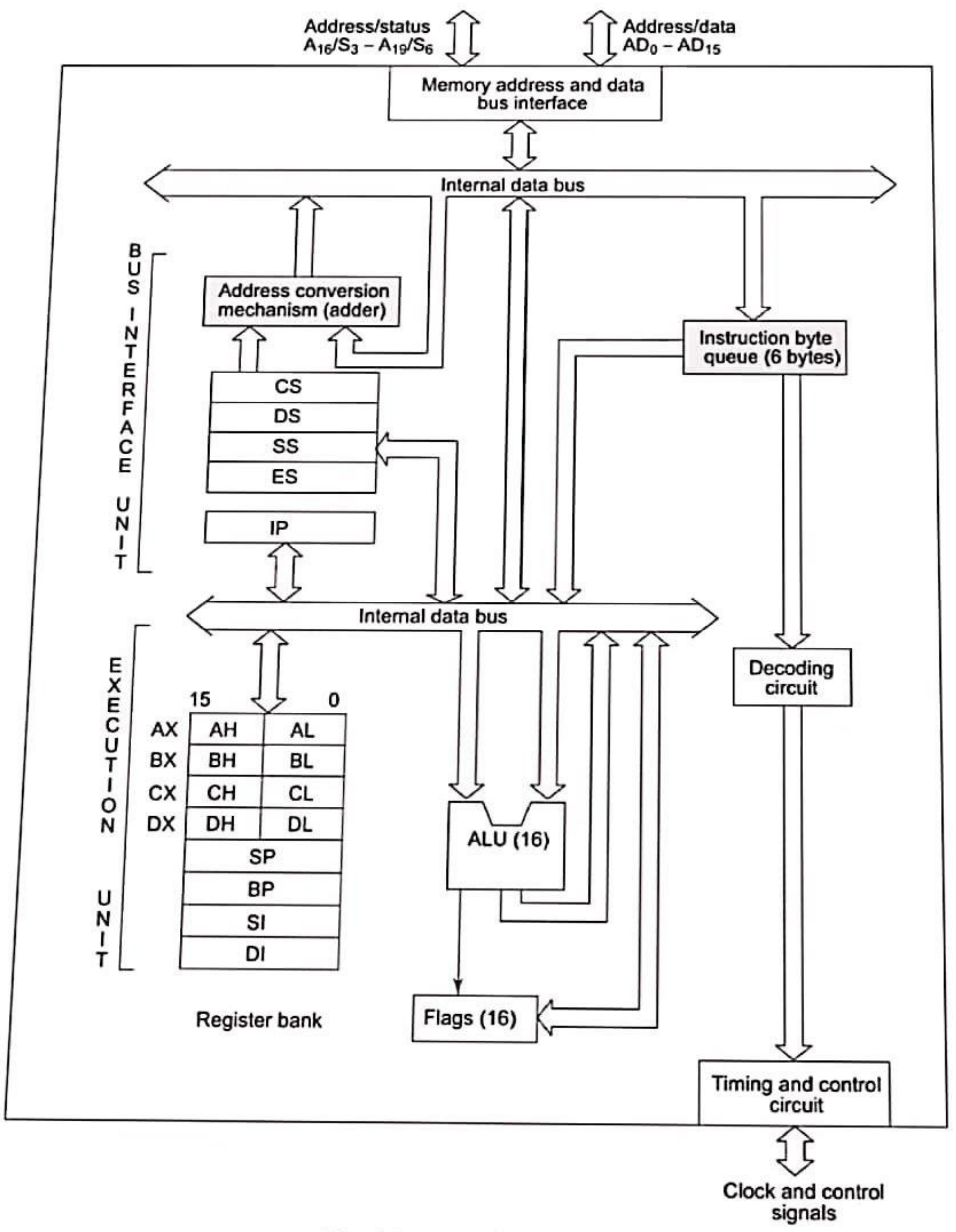
• As already stated, the 8086 addresses a segmented memory. The complete physical address which is 20 bits long is generated using segment and offset registers, each 16 bit long.

• For generating a physical address from contents of these two registers, the content of a segment register also called as segment address is shifted left bit-wise four times and to this result, content of an offset register also called as offset address is added, to produce a 20 bit physical address.

• For example, if the segment address is 1005H and the offset is 5555H, then the physical address is calculated as below 1005H□segment address 5555H□offset address Physical address = 1005 * 10 + 5555 = 155A5H

• Thus, the segment addressed by the segment value 1005H can have offset values from 0000H to FFFFH within it i.e. maximum 64 K locations may be accommodated in the segment.

• Thus, the segment register indicates the base address of a particular segment, while the offset indicates the distance of the required memory location in the segment from the base address.



Address/status $A_{16}/S_3 - A_{19}/S_6$
Address/data $AD_0 - AD_{15}$
Memory address and data bus interface
Internal data bus
BUS INTERFACE UNIT
Address conversion mechanism (adder)
Instruction byte queue (6 bytes)
CS
DS
SS
ES
IP
Internal data bus
EXECUTION UNIT
15
0
AX
BX
CX
DX
AH
AL
BH
BL
CH
CL
DH
DL
SP
BP
SI
DI
Register bank
ALU (16)
Flags (16)
Decoding circuit
Timing and control circuit
Clock and control signals

Fig. 1.2 *8086 Architecture*

- Since the offset is a 16-bit number, each segment can have a maximum of 64k locations.
- The bus interface unit has a separate adder to perform this procedure for obtaining a physical address while addressing a memory.
- The segment address value is to be taken from an appropriate segment register depending upon whether code, data or stack are to be accessed, while the offset may be the content of IP, BX, SI, DI, SP, BP or an immediate 16-bit value, depending upon the addressing mode.
- In case of 8085, once the op-code is fetched and decoded, the external bus remains free for some time, while processor internally executes the instruction.

• The time slot is utilized in 8086 to achieve the overlapped fetch and execution cycles.

• While the fetched instruction is executed internally, the external bus is used to fetch the machine code of the next instruction and arrange it in a queue known as pre-decoded instruction byte queue. It is a 6 byte long, first-in first-out structure.

• The instructions from the queue are taken for decoding sequentially.

• Once a byte is decoded, the queue is rearranged by pushing it out and the queue status is checked for the possibility of the next op-code fetch cycle.

• While the op-code is fetched by the interface unit (BIU), the execution unit (EU) executes the previously decoded instruction concurrently.

• The BIU along with EU thus forms a pipeline.

• The bus interface unit, thus manages the complete interface of execution unit with memory and I/O devices, of-course, under the control of the timing and control unit.

• The execution unit contains the register set of 8086 except segment register and IP.

• It has a 16-bit ALU, able to perform arithmetic and logical operation.

• The 16-bit flag register reflects the results of execution by the ALU.

• The decoding unit decodes the op-code bytes issued from the instruction byte queue.

• The timing and control unit derives the necessary control signals to execute the instruction op-code received from the queue, depending upon the information made available by the decoding circuit.

• The execution unit may pass the results to bus interface unit for storing them in memory.

Memory Addresses

• As 8086 has got 20 address lines, it's addressing capability is 1 M Byte memory locations.

• The physical address is calculated from segment address and offset address as given below Physical address=10*segment addr + offset addr

Memory Segmentation

• The memory in an 8086 based system is organized as segmented memory.

• In this scheme, the complete physically available memory may be divided into a number of logical segments.

• Each segment is 64 Kbytes in 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.

• To address a specific memory location within a segment, we need an offset address.

• The offset address is also 16 bit long so that the maximum offset value can be FFFFH, and the maximum size of any segment is thus 64 K locations.

• To emphasize this segmented memory concept, we will consider an example of a housing colony containing say, 100 houses

– Numbering the houses sequentially

– Numbering the houses matrix wise (rows X columns) 10 X 10

• In the second scheme, the efforts required for finding the same house will be too less.

• This second scheme in our example is analogous to the segmented memory scheme, where the addresses are specified in terms of segment addresses analogous to rows and offset addresses analogous to columns

• The CPU 8086 is able to address 1 Mbytes of physical memory.

• The complete 1 Mbytes memory can be divided into 16 segments, each of 64 Kbytes size.

• The offset address values are from 0000H and FFFFH so that the physical addresses range from 00000H to FFFFFH.

• In the above said case, the segments are called non-overlapping segments which are shown in Figure
1.3a.

• In some cases, however, the segments are overlapping.

• Suppose a segment starts at a particular address and its maximum size can be 64 Kbytes

• But, if another segment starts before this 64 kbytes locations of the first segment, the two segments are said to be overlapping segments.

• The area of memory from the start of the second segment to the possible end of the first segment is called an overlapped segment area • Figure 1.3b explains the phenomenon more clearly.

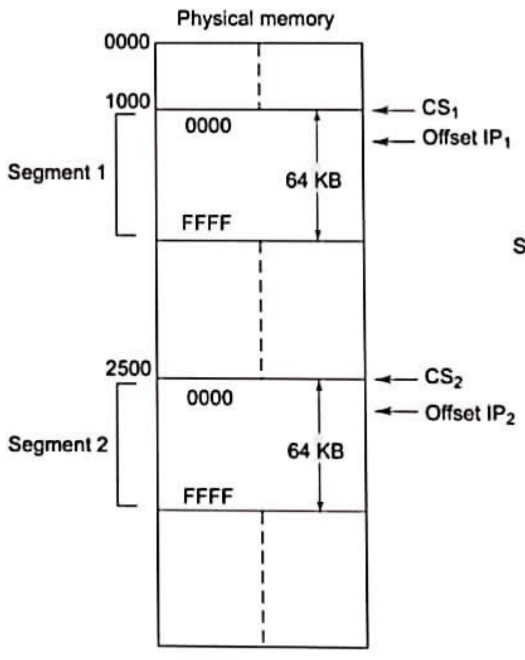


Fig. 1.3(a) ***Non-overlapping Segments***

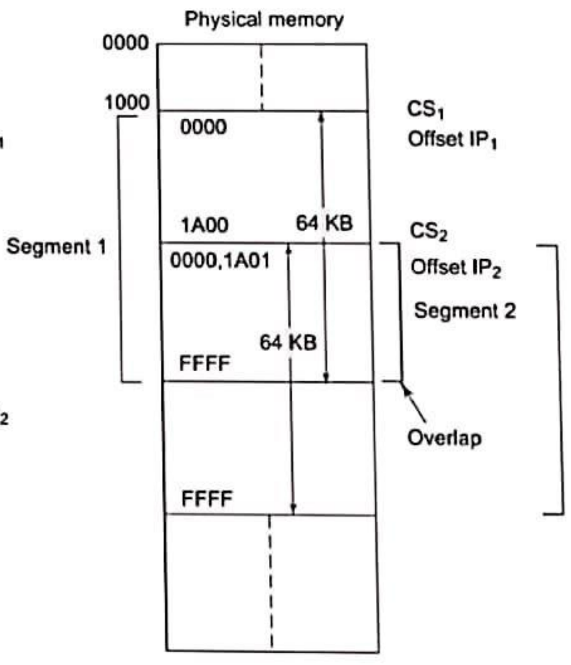


Fig. 1.3(b) ***Overlapping Segments***

• The locations lying in the overlapped area may be addressed by the same physical address generated from two different sets of segment and offset addresses.

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

– 1 Allows the memory capacity to be 1 Mbytes although the actual addresses to be handled are of 16 bit size.

– 2 Allows the placing of the code, data, and stack portions of the same program in different parts 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.

• In the overlapped Area locations physical address =CS1+ IP1 = CS2 + IP2 where + indicates the procedure of physical address formation