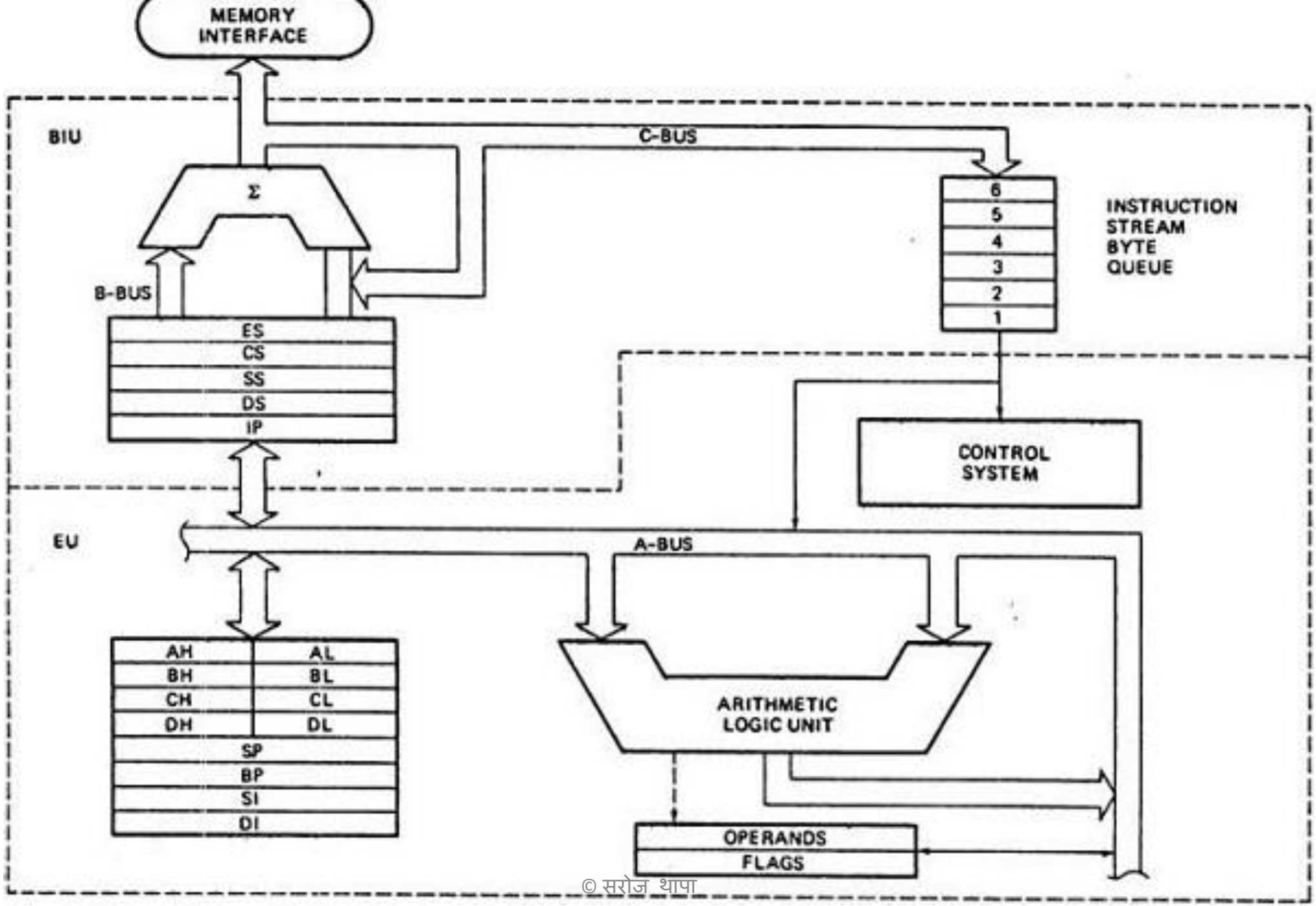
8086 Microprocessor

THE 8086 MICROPROCESSOR OVERVIEW

- The Intel 8086 is a 16 bit Microprocessor that is intended to be used as the CPU in a Microcomputer.
- The term 16 bit means that it's ALU, its internal registers, and most of its instructions are designed to work with 16 bit binary words.
- The 8086 has a 16 bit data bus, so it can read data from or write data to memory and ports either 16 bits or 8 bits at a time.
- The 8086 has a 20 bit address bus, so it can address 2^20 or 1,048576 memory locations.
- It uses two stages of pipelining, i.e. Fetch Stage and Execute Stage, which improves performance.
- Fetch stage can prefetch up to 6 bytes of instructions and stores them in the queue.


MEMORY INTERFACE
BIU
C-BUS
Σ
B-BUS
6
5
4
3
2
1
INSTRUCTION STREAM BYTE QUEUE
ES
CS
SS
DS
IP
CONTROL SYSTEM
EU
A-BUS
AH
AL
BH
BL
CH
CL
DH
DL
SP
BP
SI
DI
ARITHMETIC LOGIC UNIT
OPERANDS
FLAGS

Explanation

- The 8086 CPU is divided into two independent functional parts :

  BIU (Bus Interface Unit) and EU (Execution Unit)

- Dividing the work between these units speeds up the processing
- The BIU send out address, fetches instructions from memory, reads data from ports and memory, and writes data to ports and memory.
- In other words BIU handles all transfers of data and addresses on the buses for the execution unit.
- The EU of the 8086 tells the BIU where to fetch the instructions and data from, decodes instructions and executes instructions.

A.THE EXECUTION UNIT

- The EU contains control circuitry which directs internal operations.
- A decoder in EU translates instructions fetched from memory into a series of actions which the EU carries out.
- The EU has a 16 bit ALU which can add subtract, AND, OR, increment, decrement, complement or shift binary number

1. GENERAL PURPOSE REGISTERS

- The EU has eight general purpose registers, labeled AH, AL, BH, BL, CH, CL, DH and DL.
- These registers can be used individually for temporary storage of 8 bit data.
- The AL register is also called accumulator
- It has some features that the other general purpose registers do not have.
- Certain pairs of these general purpose registers can be used together to store 16 bit words.
- The acceptable register pairs are AH and AL,BH and BL,CH and CL,DH and DL
- The AH-AL pair is referred to as the AX register, the BH-BL pair is referred to as the BX register, the CH-CL pair is referred to as the CX register, and the DH-DL pair is referred to as the DX register.
    - AX = Accumulator Register
    - BX = Base Register
    - CX = Count Register
    - DX = Data Register

2. FLAG REGISTER

- A Flag is a flip-flop which indicates some condition produced by the execution of an instruction or controls certain operations of the EU.
- A 16 bit flag register in the EU contains 9 active flags.
- Figure below shows the location of the nine flags in the flag register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
U	U	U	U	OF	DF	IF	TF	SF	ZF	U	AF	U	PF	U	CF

Figure: 8086 Flag Register Format

U = UNDEFINED

CONDITIONAL FLAGS

CF = CARRY FLAG [Set by Carry out of MSB]
PF = PARITY FLAG [Set if Result has even parity]
AF= AUXILIARY CARRY FLAG FOR BCD
ZF = ZERO FLAG [Set if Result is 0]
SF = SIGN FLAG [MSB of Result]
OF = OVERFLOW FLAG

CONTROL FLAG

TF = SINGLE STEP TRAP FLAG
IF = INTERRUPT ENABLE FLAG
DF = STRING DIRECTION FLAG

Flags Cont..

- The six conditional flags in this group are the CF,PF,AF,ZF,SF and OF
- The three remaining flags in the Flag Register are used to control certain operations of the processor.
- The six conditional flags are set or reset by the EU on the basis of the result of some arithmetic or logic operation.
- The Control Flags are deliberately set or reset with specific instructions you put in your program.
- The three control flags are the TF,IF and DF.
- Trap Flag is used for single step control and allows the user to execute one instruction at a time for debugging
- The Interrupt Flag is used to allow or prohibit the interruption of a program.
- The Direction Flag is used with string instructions. As the name suggests when it is set then string bytes are accessed from the higher memory address to the lower memory address and vice-a-versa.

3. POINTER REGISTERS

- The 16 bit Pointer Registers are IP,SP and BP respectively
- SP and BP are located in EU whereas IP is located in BIU

3.1 STACK POINTER (SP)

- The 16 bit SP Register provides an offset value, which when associated with the SS register (SS:SP)

3.2 BASE POINTER (BP)

- The 16 bit BP facilitates referencing parameters, which are data and addresses that a program passes via the stack.
- The processor combines the addresses in SS with the offset in BP.
- BP can also be combined with DI and SI as a base register for special addressing.

4. INDEX REGISTERS

- The 16 bit Index Registers are SI and DI

4.1 SOURCE INDEX (SI) REGISTER

- The 16 bit Source Index Register is required for some string handling operations
- SI is associated with the DS Register.

4.2 DESTINATION INDEX (DI) REGISTER

- The 16 bit Destination Index Register is also required for some string operations.
- In this context, DI is associated with the ES register.

B. THE BUS INTERFACE UNIT

1. SEGMENT REGISTERS

1.1 CS REGISTER

- It contains the starting address of a program's code segment.
- This segment address plus an offset value in the IP register indicates the address of an instruction to be fetched for execution

1.2 DS REGISTER

- It contains the starting address of a program's data segment
- Instruction uses this address to locate data.
- This address plus an offset value in an instruction causes a reference to a specific byte location in the data segment.

Cont..

1.3 SS REGISTER

- It stands for Stack Segment.
- Permits the implementation of a stack in memory
- It stores the starting address of a program's stack segment the SS register.
- This segment address plus an offset value in the Stack Pointer (SP) register indicates the current word in the stack being addressed.

1.4 ES REGISTER

- It stands for Extra segment.
- It is used by some string operations to handle memory addressing.
- ES Register is associated with the DI Register

2. INSTRUCTION POINTER (IP)

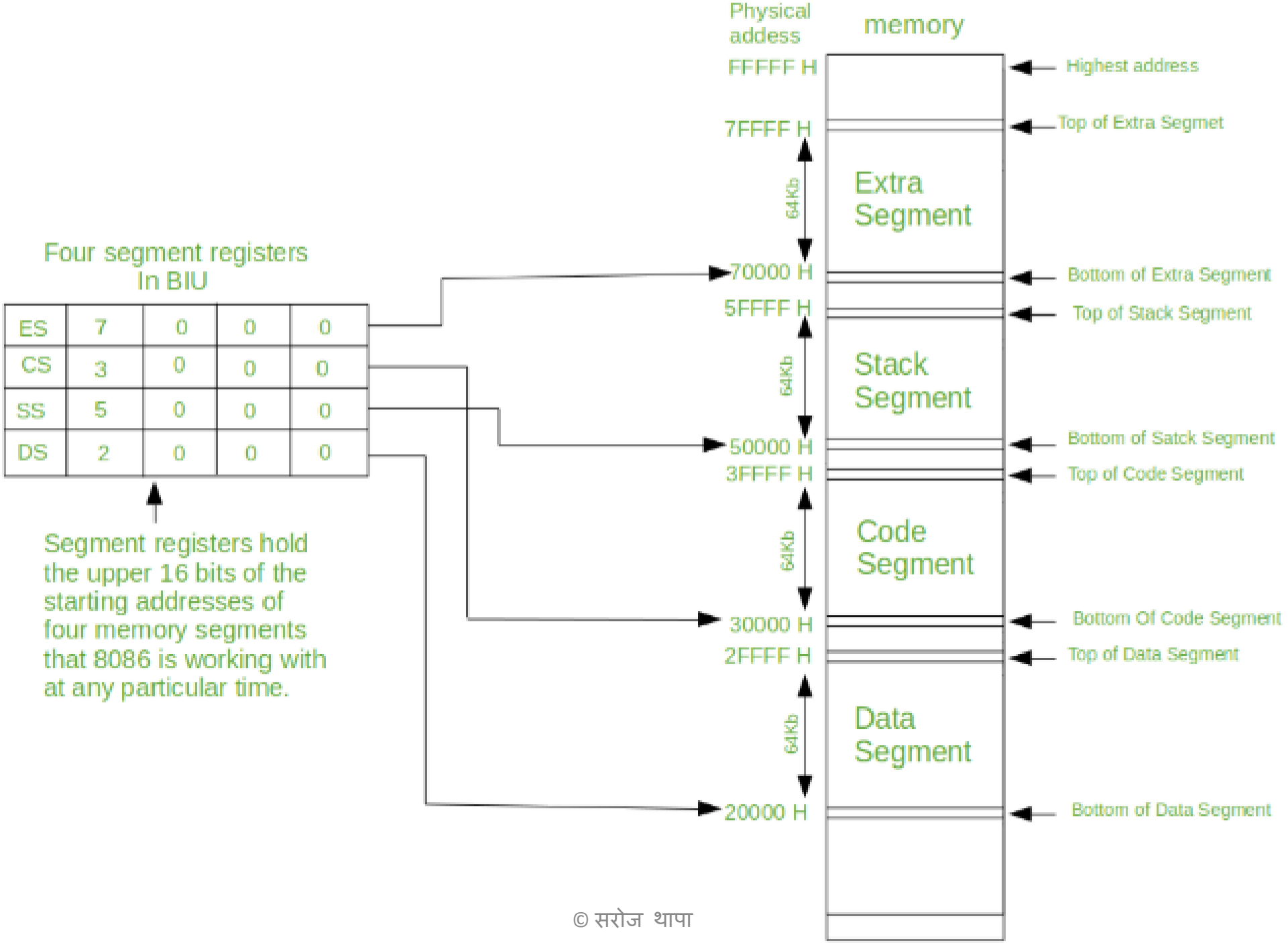
- The 16 bit IP Register contains the offset address of the next instruction that is to execute.
- IP is associated with CS register as (CS:IP)
- For each instruction that executes, the processor changes the offset value in IP so that IP in effect directs each step of execution.

3. THE QUEUE

- While the EU is decoding an instruction or executing an instruction which does not require use of the buses, the BIU fetches up to six instructions bytes for the following instructions.
- The BIU Stores pre-fetched bytes in First in First out register set called a queue.
- When the EU is ready for its next instruction, it simply reads the instruction bytes for the instruction from the queue in the BIU.
- This is much faster than sending out an address to the system memory and waiting for memory to send back the next instruction bytes or bytes.
- Fetching the next instruction while the current instruction executes is called pipeling.

Memory Segmentation in 8086 Microprocessor

- **Segmentation** is the process in which the main memory of the computer is divided into different segments and each segment has its own base address.
- It is basically used to enhance the speed of execution of the computer system, so that processor is able to fetch and execute the data from the memory easily and fast.
- The **four segment registers** actually contain the upper 16 bits of the starting addresses of the four memory segments of 64 KB each with which the 8086 is working at that instant of time.
- A segment is a logical unit of memory that may be up to 64 kilobytes long. Each segment is made up of contiguous memory locations. It is independent, separately addressable unit. Starting address will always be changing. It will not be fixed.


Physical addess
memory
FFFFF H
Highest address
7FFFF H
Top of Extra Segmet
64Kb
Extra Segment
70000 H
Bottom of Extra Segment
5FFFF H
Top of Stack Segment
64Kb
Stack Segment
50000 H
Bottom of Satck Segment
3FFFF H
Top of Code Segment
64Kb
Code Segment
30000 H
Bottom Of Code Segment
2FFFF H
Top of Data Segment
64Kb
Data Segment
20000 H
Bottom of Data Segment
Four segment registers In BIU
ES 7 0 0 0
CS 3 0 0 0
SS 5 0 0 0
DS 2 0 0 0
Segment registers hold the upper 16 bits of the starting addresses of four memory segments that 8086 is working with at any particular time.

Advantages of the Segmentation

The main advantages of segmentation are as follows:

- It provides a powerful memory management mechanism.
- Data related or stack related operations can be performed in different segments.
- Code related operation can be done in separate code segments.
- It allows to processes to easily share data.
- It allows to extend the address ability of the processor, i.e. segmentation allows the use of 16 bit registers to give an addressing capability of 1 Megabytes. Without segmentation, it would require 20 bit registers.
- It is possible to enhance the memory size of code data or stack segments beyond 64 KB by allotting more than one segment for each area.

INSTRUCTION SETS OF 8086

Instruction set in 8086

1. **Data Transfer Instructions**
2. **Arithmetic Instructions**
3. **Logical Instructions**
4. **String manipulation Instructions**
5. **Process Control Instructions**
6. **Control Transfer Instructions**

Data transfer Instructions

MOV des,src

- This instruction copies a word or a byte of data from some source to a destination. The destination can be a register or a memory location. The source can be a register, a memory location, or an immediate number.
- MOV AX,BX
- MOV AX,5000H
- MOV AX,[SI]
- MOV [734AH],BX
- MOV DS,CX
- MOV CL,[357AH]

Cont..

PUSH: Push to Stack

- This instruction pushes the contents of the specified register/memory location on to the stack. The stack pointer is decremented by 2, after each execution of the instruction.

E.g. PUSH AX

PUSH DS

PUSH [5000H]

POP : Pop from Stack

- This instruction when executed, loads the specified register/memory location with the contents of the memory location of which the address is formed using the current stack segment and stack pointer.
- The stack pointer is incremented by 2 Eg. POP AX

POP DS

POP [5000H]

Cont..

XCHG : Exchange byte or word

- This instruction exchange the contents of the specified source and destination operands
- Eg. XCHG [5000H], AX

  XCHG BX, AX

Cont..

IN:

- Copy a byte or word from specified port to accumulator. Eg. IN AL,03H
- IN AX,DX

OUT:

- Copy a byte or word from accumulator specified port. Eg. OUT 03H, AL
- OUT DX, AX

LEA :

- Load effective address of operand in specified register. [reg] offset portion of address in DS
- Eg. LEA reg, offset

LDS:

- Load DS register and other specified register from memory.
- Eg. LDS reg, mem

LES:

- Load ES register and other specified register from memory.
- Eg. LES reg, mem

Arithmetic Instructions:

ADD :

- The add instruction adds the contents of the source operand to the destination operand.
- Eg. ADD AX, 0100H ADD AX, BX ADD AX, [SI] ADD AX, [5000H]
- ADD [5000H], 0100H ADD 0100H

ADC : Add with Carry

- This instruction performs the same operation as ADD instruction, but adds the carry flag to the result.
- Eg. ADC 0100H ADC AX, BX ADC AX, [SI] ADC AX, [5000]
- ADC [5000], 0100H

SUB : Subtract

- The subtract instruction subtracts the source operand from the destination operand and the result is left in the destination operand.
- Eg. SUB AX, 0100H SUB AX, BX SUB AX, [5000H]

SBB : Subtract with Borrow

- The subtract with borrow instruction subtracts the source operand and the borrow flag (CF) which may reflect the result of the previous calculations, from the destination operand
- Eg. SBB AX, 0100H SBB AX, BX SBB AX, [5000H]
- SBB [5000H], 0100H

Cont..

DEC : Decrement

- The decrement instruction subtracts 1 from the contents of the specified register or memory location.
- Eg. DEC AX
- DEC [5000H]

NEG : Negate

- The negate instruction forms 2's complement of the specified destination in the instruction. The destination can be a register or a memory location. This instruction can be implemented by inverting each bit and adding 1 to it.
- Eg. NEG AL

CMP : Compare

- This instruction compares the source operand, which may be a register or an immediate data or a memory location, with a destination operand that may be a register or a memory location
- Eg. CMP BX, 0100H
- CMP AX, 0100H

Cont..

MUL :Unsigned Multiplication Byte or Word

- This instruction multiplies an unsigned byte or word by the contents of AL.
- Eg. MUL BH
- MUL CX

DIV : Unsigned division

- This instruction is used to divide an unsigned word by a byte or to divide an unsigned double word by a word.
- Eg. DIV CL

[Word in AX / byte in CL] [Quotient in AL, remainder in AH]

- DIV CX

[Double word in DX and AX / word in CX]

and Quotient in AX, remainder in DX

AAA : ASCII Adjust After Addition

- The AAA instruction is executed after an ADD instruction that adds two ASCII coded operand to give a byte of result in AL. The AAA instruction converts the resulting contents of Al to a unpacked decimal digits.
- Eg. ADD CL, DL ; [CL] = 32H = ASCII for 2

  ; [DL] = 35H = ASCII for 5

  ; Result [CL] = 67H

  MOV AL, CL ; Move ASCII result into AL since AAA adjust only [AL]

  AAA ; [AL]=07, unpacked BCD for 7

Addressing modes in 8086 MPU

Addressing modes refer to the different methods of addressing the operands. Addressing modes of 8086 are as follows:

1.Immediate addressing mode-

In **this** mode, the operand is specified in the instruction itself. Instructions are longer but the operands are easily identified.

Example:

MOV CL, 12H

This instruction moves 12 immediately into CL register. CL ← 12H

2. Register addressing mode-

In this mode, operands are specified using registers. This addressing mode is normally preferred because the instructions are compact and fastest executing of all instruction forms.

Registers may be used as source operands, destination operands or both.

Example:

MOV AX, BX

This instruction copies the contents of BX register into AX register. AX ← BX

3. Direct memory addressing mode-

- In this mode, address of the operand is directly specified in the instruction. Here only the offset address is specified, the segment being indicated by the instruction.

Example:

MOV CL, [4321H]

- This instruction moves data from location 4321H in the data segment into CL.
- The physical address is calculated as

DS * 10H + 4321

Assume DS = 5000H

∴PA = 50000 + 4321 = 54321H

∴CL ← [54321H]

4. Register based indirect addressing mode-

- In this mode, the effective address of the memory may be taken directly from one of the base register or index register specified by instruction. If register is SI, DI and BX then DS is by default segment register.
- If BP is used, then SS is by default segment register.

Example:

MOV CX, [BX]

5. Register relative addressing mode-

- In this mode, the operand address is calculated using one of the base registers and an 8 bit or a 16 bit displacement.

Example:

MOV CL, [BX + 04H]

- This instruction moves a byte from the address pointed by BX + 4 in data segment to CL.

CL ← DS: [BX + 04H]

- Physical address can be calculated as DS * 10H + BX + 4H.

6. Base indexed addressing mode-

- Here, operand address is calculated as base register plus an index register.
- Example:

  MOV CL, [BX + SI]

- This instruction moves a byte from the address pointed by BX + SI in data segment to CL.
- CL ← DS: [BX + SI]
- Physical address can be calculated as DS * 10H + BX + SI

7. Relative based indexed addressing mode-

- In this mode, the address of the operand is calculated as the sum of base register, index register and 8 bit or 16 bit displacement.
- Example:

  MOV CL, [BX + DI + 20]

- This instruction moves a byte from the address pointed by BX + DI + 20H in data segment to CL.
- CL ← DS: [BX + DI + 20H]
- Physical address can be calculated as DS * 10H + BX + DI + 20H.

8. Direct I/O port Addressing

9. Indirect I/O port Addressing

- These addressing modes are used to access data from standard I/O mapped devices or ports.
- In direct port addressing mode, an 8-bit port address is directly specified in the instruction.
- Example: IN AL, [09H]
- In indirect port addressing mode, the instruction will specify the name of the register which holds the port address. In 8086, the 16-bit port address is stored in the DX register.

- Example: OUT [DX], AX

8. Implied addressing mode-

- In this mode, the operands are implied and are hence not specified in the instruction.
- Example:

  STC

- This sets the carry flag

Assembly level programming

Coding in Assembly language: Assembly language programming language has taken its place in between the machine language (low level) and the high level language.

- High level language's one statement may generate many machine instructions.
- Low level language consists of either binary or hexadecimal operation. One symbolic statement generates one machine level instructions.

Advantage of ALP

- They generate small and compact execution module.
- They have more control over hardware.
- They generate executable module and run faster.

Disadvantages of ALP:

- Machine dependent.
- Lengthy code
- Error prone (likely to generate errors).

Assembly language features:

- The main features of ALP are program comments, reserved words, identifies, statements and directives which provide the basic rules and framework for the language.

Program comments:

- The use of comments throughout a program can improve its clarity.
- It starts with semicolon (;) and terminates with a new line.
- E.g. ADD AX, BX **; Adds AX & BX**

Reserved words:

- Certain names in assembly language are reserved for their own purpose to be used only under special conditions and includes
- Instructions : Such as MOV and ADD (operations to execute)
- Directives: Such as END, SEGMENT (information to assembler)
- Operators: Such as FAR, SIZE
- Predefined symbols: such as @DATA, @ MODEL

Identifiers:

- An identifier (or symbol) is a name that applies to an item in the program that expects to reference.
- Two types of identifiers are **Name and Label**.
- Name refers to the address of a data item such as **NUM1 DB 5, COUNT DB 0**
- Label refers to the address of an instruction.

E. g: **MAIN PROC FAR**

L1: ADD BL, 73

Directives:

- The directives are the number of statements that enables us to control the way in which the source program assembles and lists.
- These statements called directives act only during the assembly of program and generate no machine-executable code. The different types of directives are:

1) The page and title listing directives:

- The page and title directives help to control the format of a listing of an assembled program.
- This is their only purpose and they have no effect on subsequent execution of the program.
- The page directive defines the maximum number of lines to list as a page and the maximum number of characters as a line.

PAGE [Length] [Width]

Default : Page [50][80]

TITLE gives title and place the title on second line of each page of the program.

TITLE text [comment]

2) SEGMENT directive

- It gives the start of a segment for **stack, data and code**.

  Seg-name Segment [align]+[combine]+[‘class’]
  Seg-name ENDS

- Segment name must be present, must be unique and must follow assembly language naming conventions.
- An ENDS statement indicates the end of the segment.
- Align option indicates the boundary on which the segment is to begin; PARA is used to align the segment on paragraph boundary.
- Combine option indicates whether to combine the segment with other segments when they are linked after assembly. COMMON, PUBLIC, etc are combine types.
- Class option is used to group related segments when linking. The class code for code segment, stack for stack segment and data for data segment.

Example:STACK SEGMENT PARA ‘Stack’

DATA SEG SEGMENT PARA ‘Data

CODE SEG SEGMENT PARA ‘Code’

3) PROC Directives

- The code segment contains the executable code for a program, which consists of one or more procedures, defined initially with the PROC directives and ended with the ENDP directive.

PROC - name PROC [FAR/NEAR]

…….

…….

…….

PROC - name ENDP

- FAR is used for the first executing procedure and rest procedures call will be NEAR.
- Procedure should be within segment.

4) END Directive

- An END directive ends the entire program and appears as the last statement.
- ENDS directive ends a segment and ENDP directive ends a procedure. END PROC-Name

5) ASSUME Directive

- An .EXE program uses the SS register to address the stack, DS to address the data segment and CS to address the code segment.
- Used in conventional full segment directives only.
- Assume directive is used to tell the assembler the purpose of each segment in the program.
- Assume **SS: Stack name, DS: Data Segname CS: codesegname**

6) Processor directive

- Most assemblers assume that the source program is to run on a basic 8086 level computer.
- Processor directive is used to notify the assembler that the instructions or features introduced by the other processors are used in the program.
- E.g. **.386** - program for 386 protected mode.

7) Dn Directive (Defining data types)

- Assembly language has directives to define data syntax [name] Dn expression
- The Dn directive can be any one of the following:

DB Define byte 1 byte

DW Define word 2 bytes

DD Define double 4 bytes

DF defined farword 6 bytes

DQ Define quadword 8 bytes

DT Define 10 bytes 10 bytes

8) The EQU directive

- It can be used to assign a name to constants.
- E.g. **FACTOR EQU 12**

  MOV BX, FACTOR ; MOV BX, 12
- It is short form of equivalent.
- Do not generate any data storage; instead the assembler uses the defined value to substitute in.

9) DUP Directive

- It can be used to initialize several locations to zero.

e. g. SUM DW 4 DUP(0)

Reserves four words starting at the offset sum in DS and initializes them to Zero.

- Also used to reserve several locations that need not be initialized. In this case (?) is used with DUP directives.
- E. g. **PRICE** DB 100 DUP(?)

Reserves 100 bytes of uninitialized data space to an offset PRICE

ALP written in simplified segment directives:

Page 60, 132

```
TITLE Sum program to add two numbers.
 .MODEL SMALL
 .STACK 64
.DATA
        NUM1 DW 3241
        NUM 2 DW 572
SUM DW ?
.CODE
 MAIN PROC FAR
        MOV AX, @ DATA
        MOV DS, AX
        MOV AX, NUM1
        ADD AX, NUM2
        MOV SUM, AX
        MOV AX, 4C00H
        INT 21H
 MAIN ENDP ; End of procedure
 END MAIN
```

Assembling, Linking and Executing

1) Assembling:

- Assembling converts **source program into object program** if syntactically correct and generates an intermediate **.obj** file or module.
- It calculates the offset address for every data item in data segment and every instruction in code segment.
- Assembler complains about the syntax error if any and does not generate the object module.
- Assembler creates **.obj .lst** and **.crf** files and last two are optional files that can be created at run time.
- For short programs, assembling can be done manually where the programmer translates each mnemonic into the machine language using **lookup table(??)**.
- Assembler reads each assembly instruction of a program as ASCII character and translates them into respective machine code.

Assembler Types:

- There are two types of assemblers:
- **a) One pass assembler:**
- This assembler scans the assembly language program once and converts to object code at the same time.
- This assembler has the program of defining forward references only.
- The jump instruction uses an address that appears later in the program during scan, for that case the programmer defines such addresses after the program is assembled.

b) Two pass assembler

- This type of assembler scans the assembly language twice.
- First pass generates symbol table of names and labels used in the program and calculates their relative address.
- This table can be seen at the end of the list file and here user need not define anything.
- Second pass uses the table constructed in first pass and completes the object code creation.
- This assembler is more efficient and easier than earlier

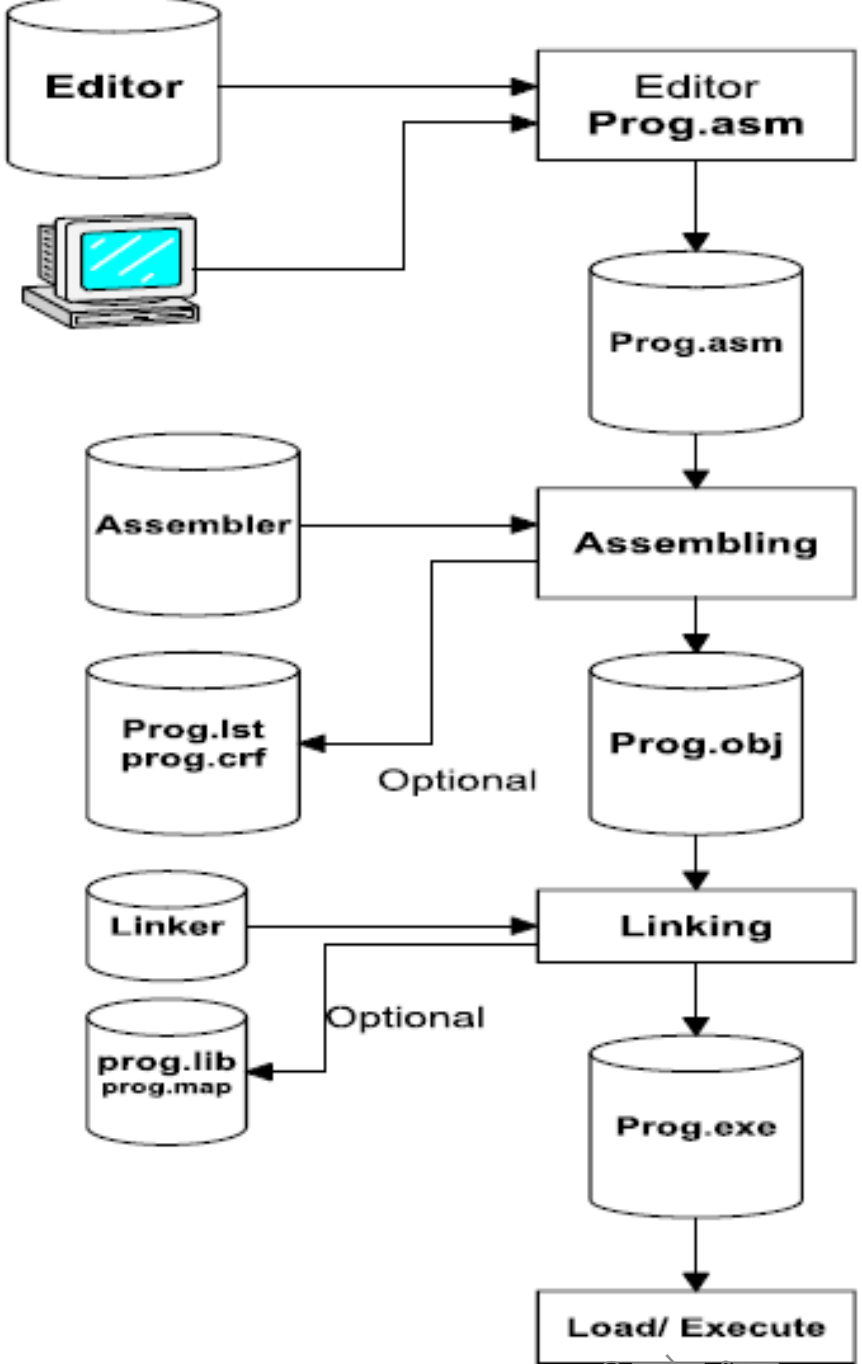


Fig: Steps in assembling, linking & Executing

2) Linking:

- This involves the converting of **.OBJ** module into **.EXE**(executable) module i.e. executable machine code.
- It completes the address left by the assembler.
- It combines separately assembled object files.
- Linking creates **.EXE**, **.LIB**, **.MAP** files among which last two are optional files.

3) Loading and Executing:

- It Loads the program in memory for execution.
- It resolves remaining address.
- This process creates the program segment prefix (PSP) before loading.
- It executes to generate the result.
- Sample program is assembled to create object file which are linked together to create an executable program

Macro Assembler

- A macro is an instruction sequence that appears repeatedly in a program assigned with a specific name.
- The macro assembler replaces a macro name with the appropriate instruction sequence each time it encounters a macro name
- When same instruction sequence is to be executed repeatedly, macro assemblers allow the macro name to be typed instead of all instructions provided the macro is defined.
  - Macro are useful for the following purposes:
  - To simplify and reduce the amount of repetitive coding.
  - To reduce errors caused by repetitive coding.
  - To make an assembly language program more readable.
  - Macro executes faster because there is no need to call and return.

- **Basic format of macro definition:**
- Macro name MACRO [Parameter list] ; Define macro

…….

…….

[Instructions] ; Macro body

…….

…….

ENDM ; End of macro

E.g. Addition MACRO

IN AX, PORT

ADD AX, BX

OUT PORT, AX

ENDM