OVERVIEW OF 8086 MICROPROCESSOR

In April 1978, Intel introduced its first 16 bit microprocessor.

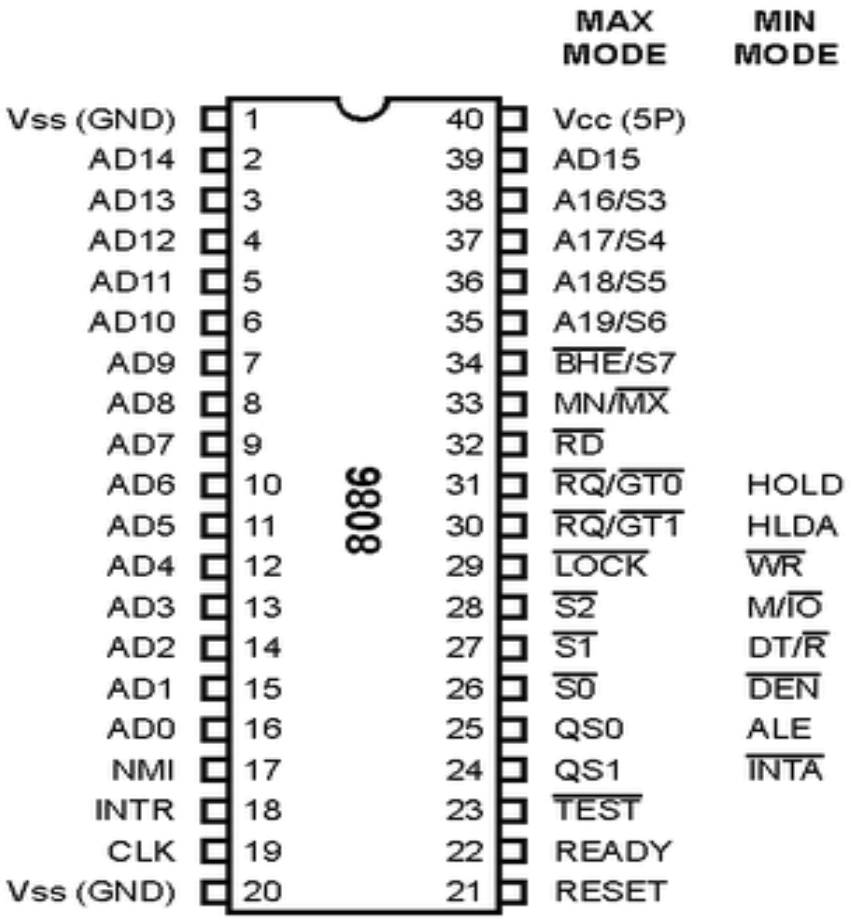


Fig. : PIN diagram of 8086 Microprocessor IC.

FEATURES OF 8086

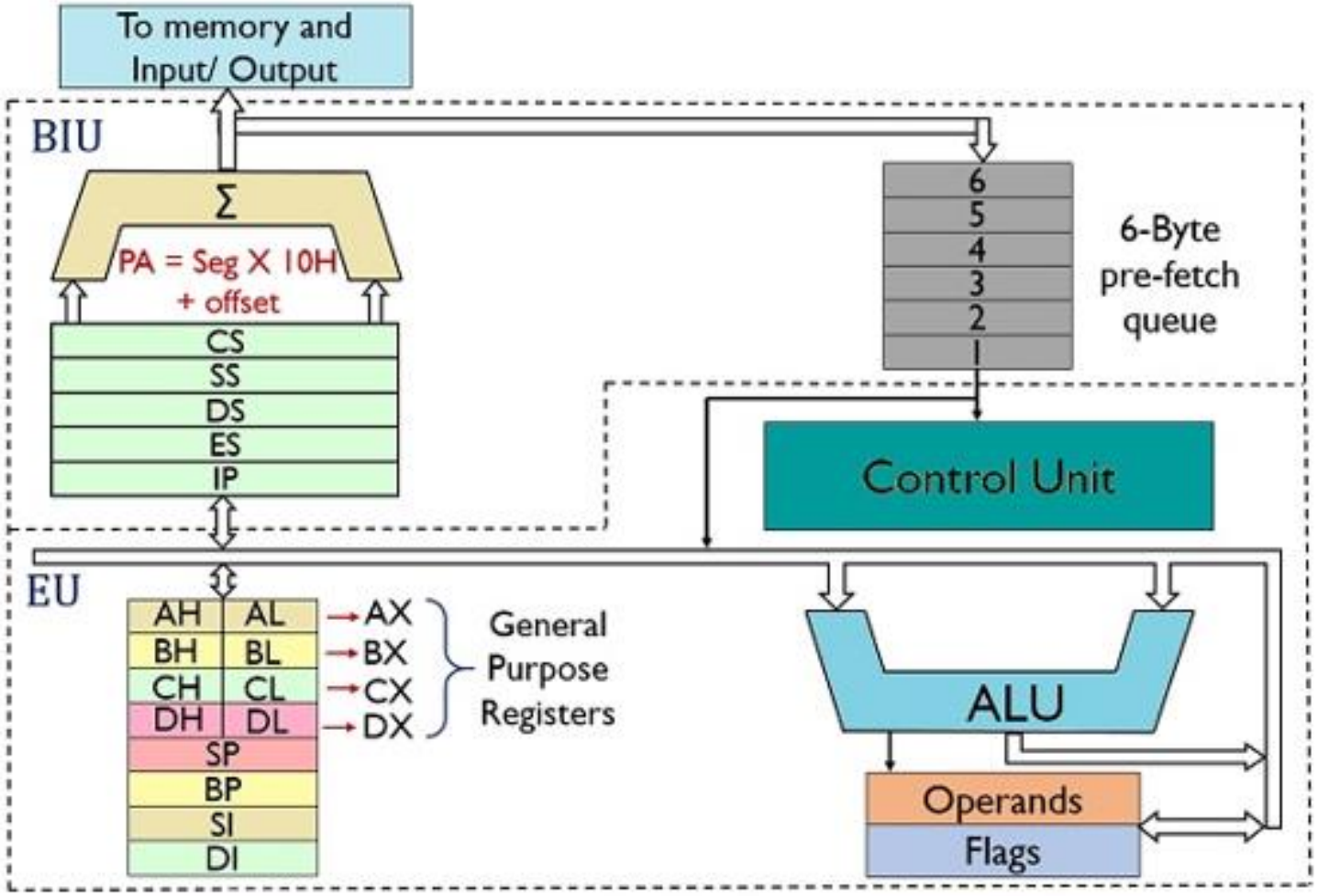
The most prominent features of a 8086 microprocessor are as follows:

- ✓ It is a 40 pin dual in line package IC.
- ✓ It is a 16-bit microprocessor.
- ✓ 8086 has a 20-bit address bus and can access up to 2^20 (1 MB) memory locations.
- ✓ It can support up to 64K I/O ports.
- ✓ It provides 14, 16-bit registers.
- ✓ Word size is 16 bits and double word size is 4 bytes.
- ✓ It has multiplexed address and data bus AD0-AD15 and A16-A19.

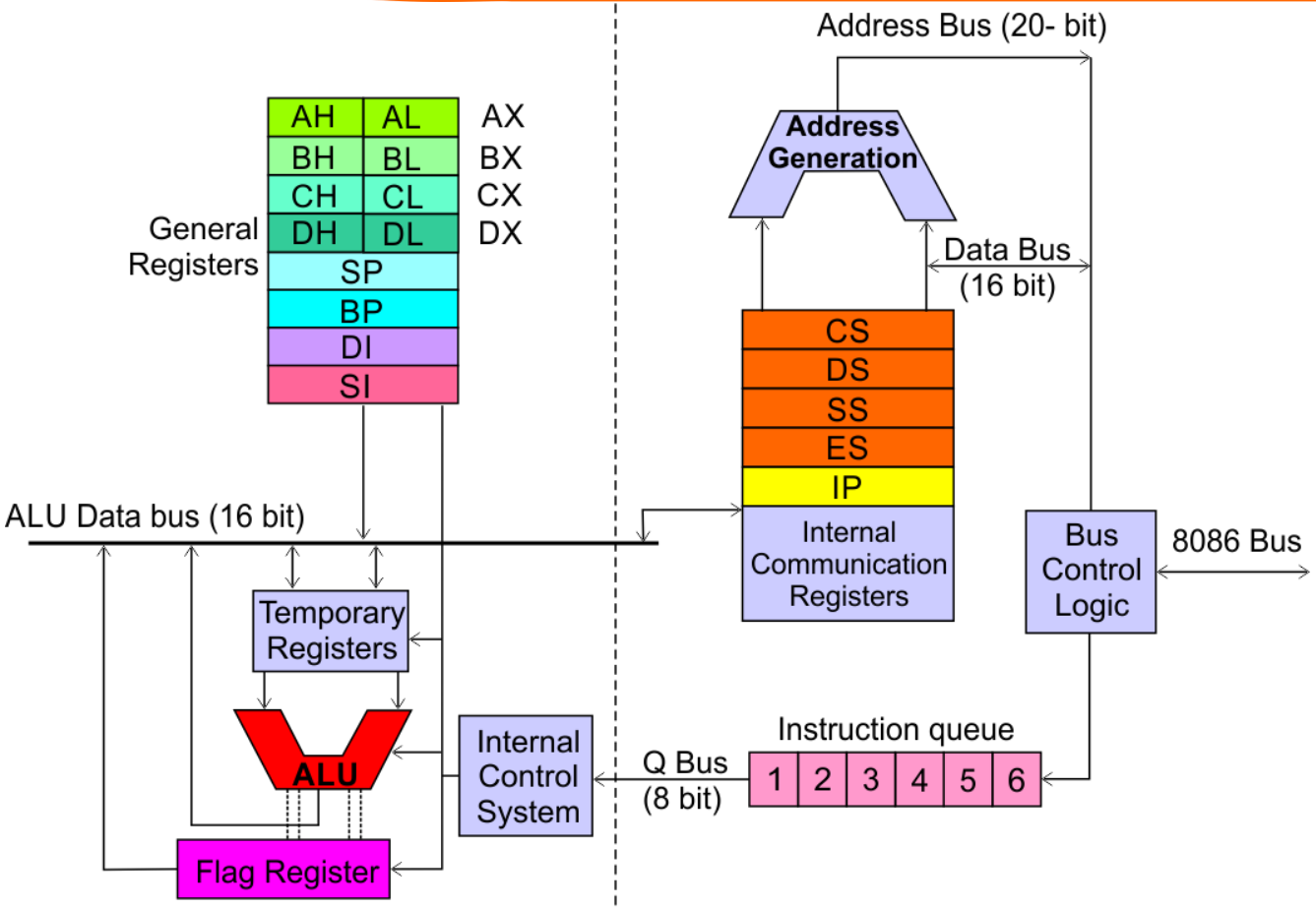
Contd.

- ✓ It requires +5V power supply.
- ✓ It can pre-fetch up to 6 instruction bytes from memory and queues them in order to speed up instruction execution.
- ✓ It has multiplexed address and data bus AD0-AD15 and A16-A19.
- ✓ It requires single phase clock with 33% duty cycle to provide internal timing.
- ✓ Address ranges from 00000H to FFFFFH.
- ✓ Memory is byte addressable – every byte has a separate address.
- ✓ 8086 is designed to operate in two modes: Minimum and Maximum.

8086 Architecture



Architecture



Execution Unit (EU)

EU executes instructions that have already been fetched by the BIU.

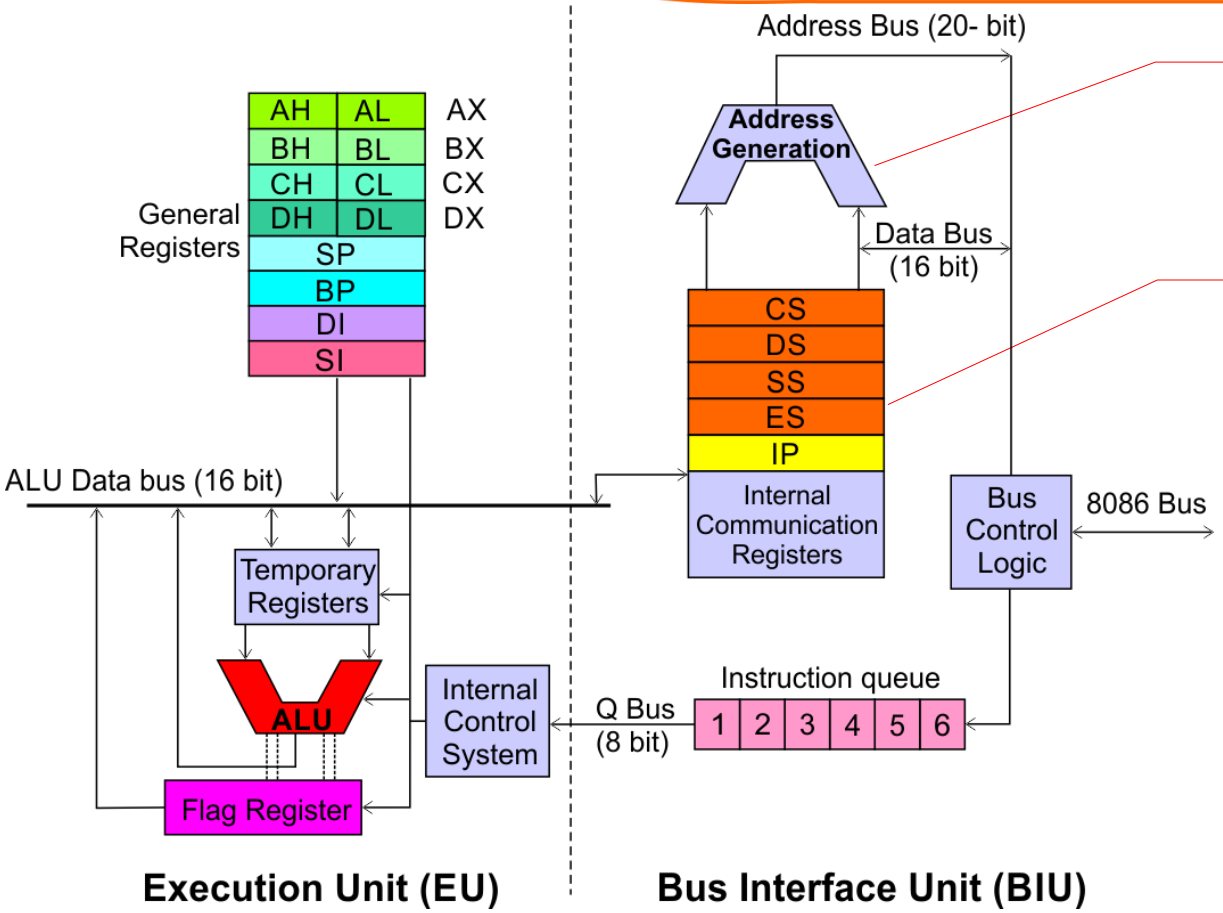
BIU and EU functions separately.

Bus Interface Unit (BIU)

BIU fetches instructions, reads data from memory and I/O ports, writes data to memory and I/ O ports.

Architecture

Bus Interface Unit (BIU)



Dedicated Adder to generate 20 bit address

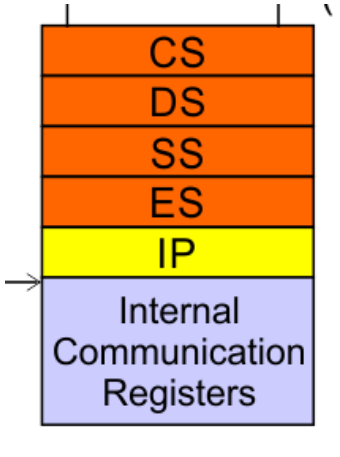
Four 16-bit segment registers

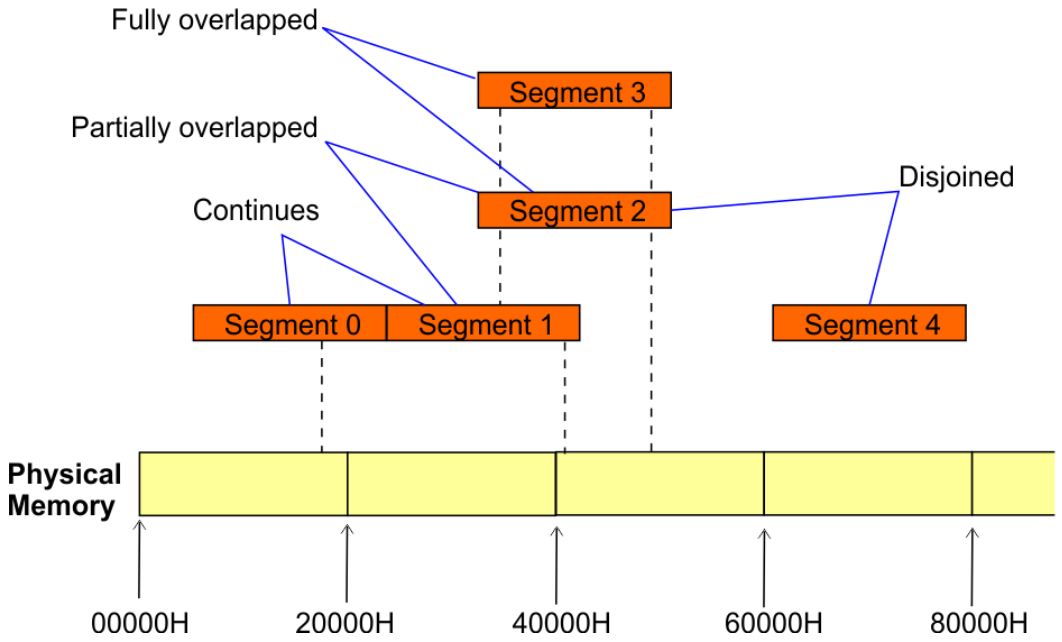
Code Segment (CS)
Data Segment (DS)
Stack Segment (SS)
Extra Segment (ES)

Architecture

Bus Interface Unit (BIU)

Segment Registers



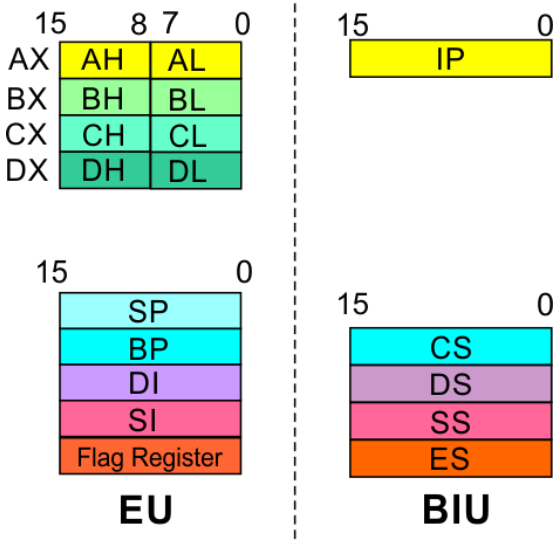


- **8086's 1-megabyte memory is divided into segments of up to 64K bytes each.**
- **The 8086 can directly address four segments (256 K bytes within the 1 M byte of memory) at a particular time.**
- **Programs obtain access to code and data in the segments by changing the segment register content to point to the desired segments.**

Segment Registers

Code Segment Register

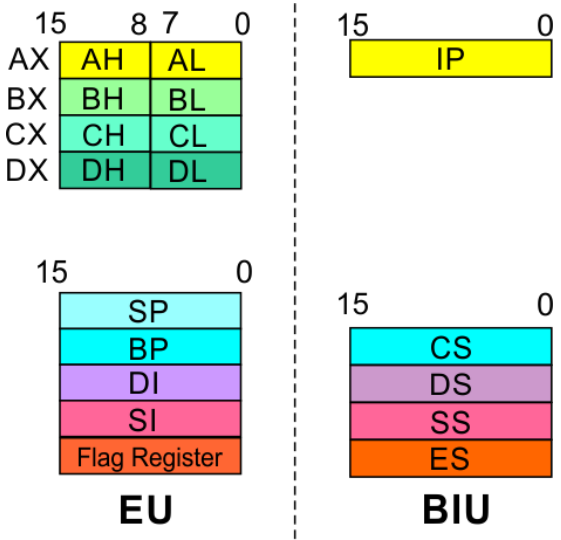
- 16-bit
- CS contains the base or start of the current code segment; IP contains the distance or offset from this address to the next instruction byte to be fetched.
- BIU computes the 20-bit physical address by logically shifting the contents of CS 4-bits to the left and then adding the 16-bit contents of IP.
- That is, all instructions of a program are relative to the contents of the CS register multiplied by 16 and then offset is added provided by the IP.



Segment Registers

Data Segment Register

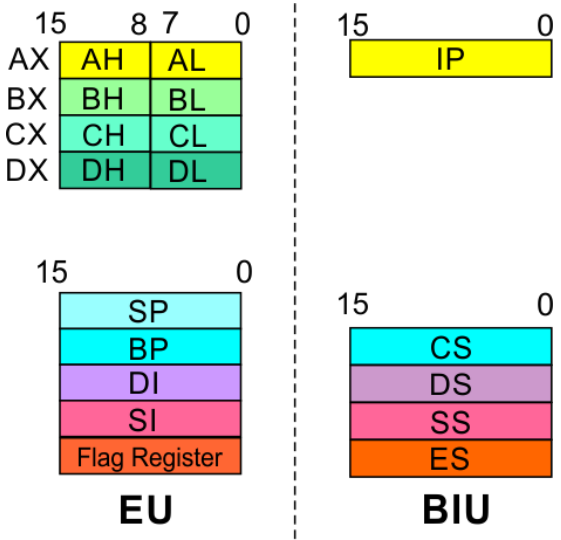
- **16-bit**
- **Points to the current data segment; operands for most instructions are fetched from this segment.**
- **The 16-bit contents of the Source Index (SI) or Destination Index (DI) or a 16-bit displacement are used as offset for computing the 20-bit physical address.**



Segment Registers

Stack Segment Register

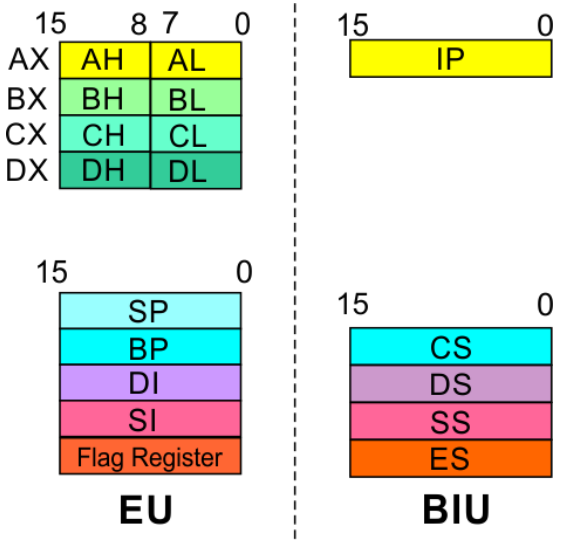
- 16-bit
- Points to the current stack.
- The 20-bit physical stack address is calculated from the Stack Segment (SS) and the Stack Pointer (SP) for stack instructions such as **PUSH** and **POP**.
- In based addressing mode, the 20-bit physical stack address is calculated from the Stack segment (SS) and the Base Pointer (BP).



Segment Registers

Extra Segment Register

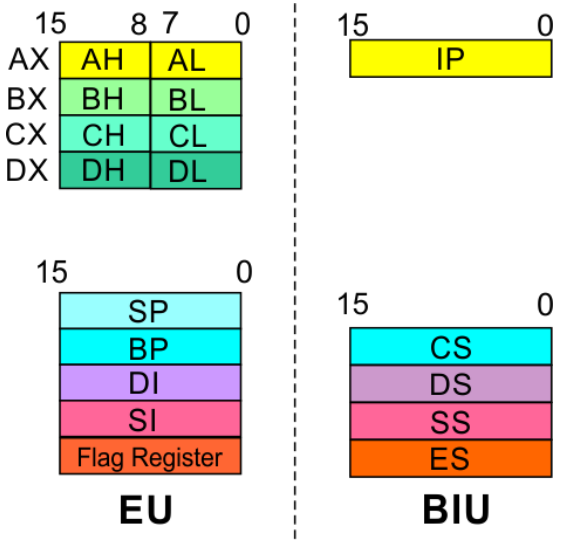
- 16-bit
- Points to the extra segment in which data (in excess of 64K pointed to by the DS) is stored.
- String instructions use the ES and DI to determine the 20-bit physical address for the destination.



Segment Registers

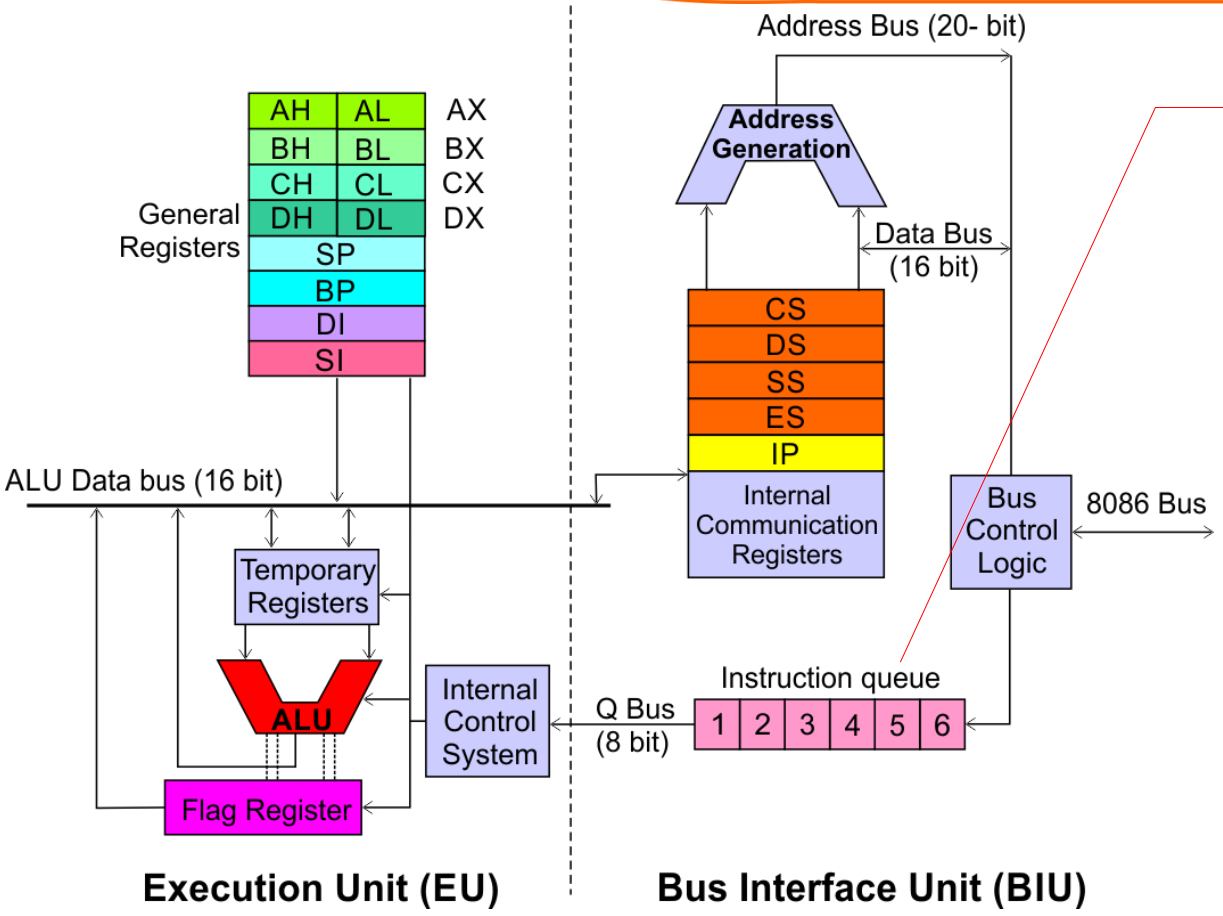
Instruction Pointer

- 16-bit
- Always points to the next instruction to be executed within the currently executing code segment.
- So, this register contains the 16-bit offset address pointing to the next instruction code within the 64Kb of the code segment area.
- Its content is automatically incremented as the execution of the next instruction takes place.



Architecture

Bus Interface Unit (BIU)



Instruction queue

- **A group of First-In-First-Out (FIFO) in which up to 6 bytes of instruction code are pre fetched from the memory ahead of time.**
- **This is done in order to speed up the execution by overlapping instruction fetch with execution.**
- **This mechanism is known as pipelining.**

Architecture

Execution Unit (EU)

EU decodes and executes instructions.

A decoder in the EU control system translates instructions.

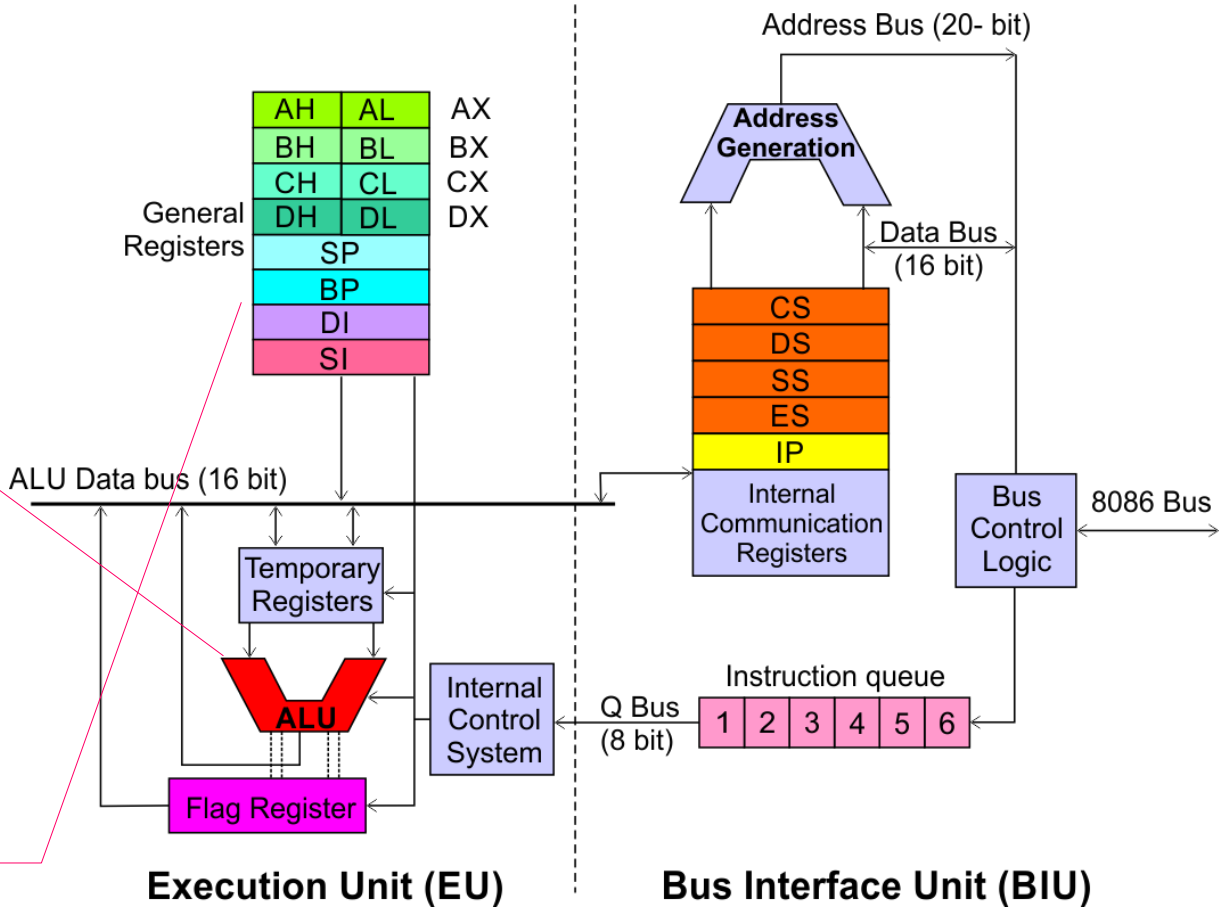
16-bit ALU for performing arithmetic and logic operation

Four general purpose registers(AX, BX, CX, DX);

Pointer registers (Stack Pointer, Base Pointer);

and

Index registers (Source Index, Destination Index) each of 16-bits



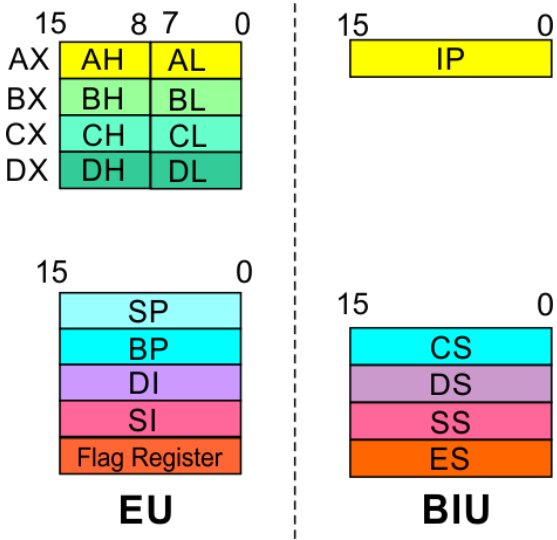
Some of the 16 bit registers can be used as two 8 bit registers as :

AX can be used as AH and AL
BX can be used as BH and BL
CX can be used as CH and CL
DX can be used as DH and DL

EU Registers

Accumulator Register (AX)

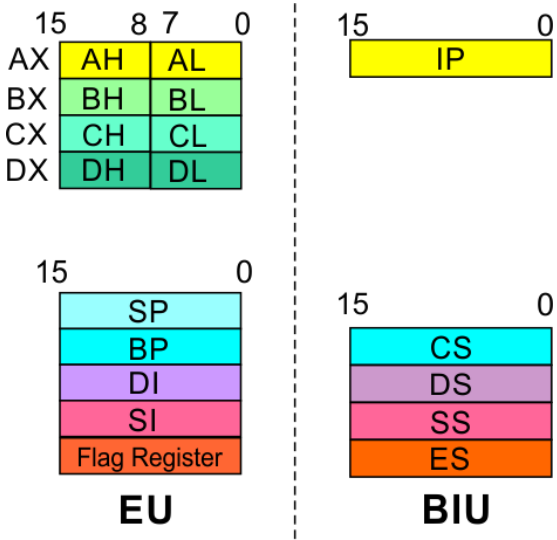
- Consists of two 8-bit registers AL and AH, which can be combined together and used as a 16-bit register AX.
- AL in this case contains the low order byte of the word, and AH contains the high-order byte.
- The I/O instructions use the AX or AL for inputting / outputting 16 or 8 bit data to or from an I/O port.
- Multiplication and Division instructions also use the AX or AL.



EU Registers

Base Register (BX)

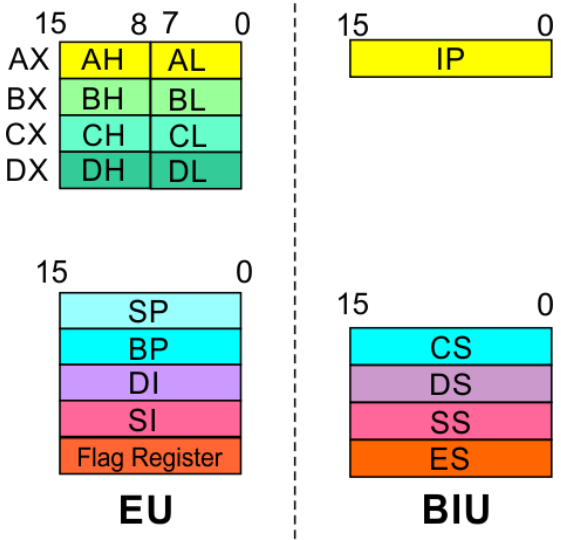
- Consists of two 8-bit registers BL and BH, which can be combined together and used as a 16-bit register BX.
- BL in this case contains the low-order byte of the word, and BH contains the high-order byte.
- This is the only general purpose register whose contents can be used for addressing the 8086 memory.
- All memory references utilizing this register content for addressing use DS as the default segment register.



EU Registers

Counter Register (CX)

- Consists of two 8-bit registers CL and CH, which can be combined together and used as a 16-bit register CX.
- When combined, CL register contains the low order byte of the word, and CH contains the high-order byte.
- Instructions such as **SHIFT**, **ROTATE** and **LOOP** use the contents of CX as a counter.



Example:

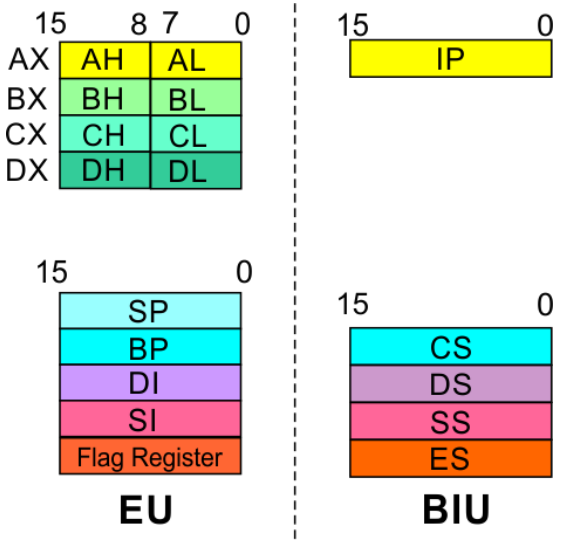
The instruction LOOP START automatically decrements CX by 1 without affecting flags and will check if [CX] = 0.

If it is zero, 8086 executes the next instruction; otherwise the 8086 branches to the label START.

EU Registers

Data Register (DX)

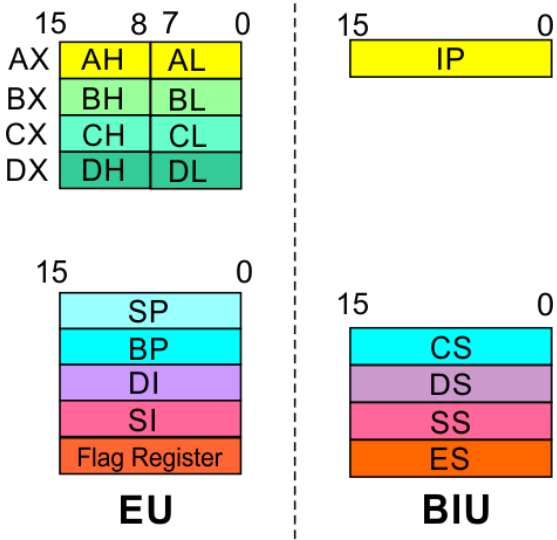
- Consists of two 8-bit registers DL and DH, which can be combined together and used as a 16-bit register DX.
- When combined, DL register contains the low order byte of the word, and DH contains the high-order byte.
- Used to hold the high 16-bit result (data) in 16 X 16 multiplication or the high 16-bit dividend (data) before a 32 ÷ 16 division and the 16-bit reminder after division.



EU Registers

Stack Pointer (SP) and Base Pointer (BP)

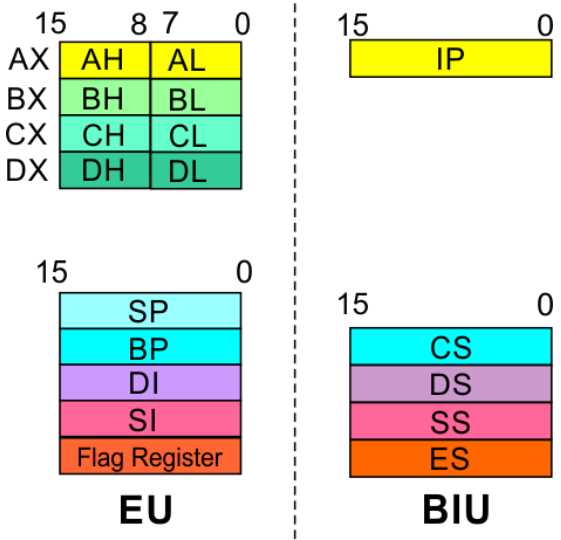
- **SP and BP are used to access data in the stack segment.**
- **SP is used as an offset from the current SS during execution of instructions that involve the stack segment in the external memory.**
- **SP contents are automatically updated (incremented/ decremented) due to execution of a POP or PUSH instruction.**
- **BP contains an offset address in the current SS, which is used by instructions utilizing the based addressing mode.**



EU Registers

Source Index (SI) and Destination Index (DI)

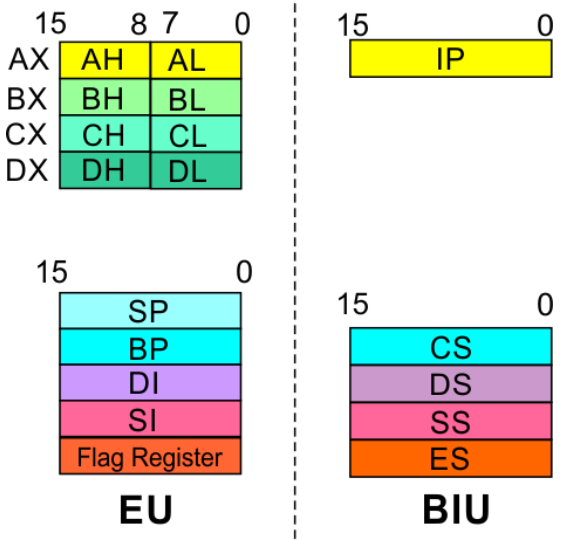
- **Used in indexed addressing.**
- **Instructions that process data strings use the SI and DI registers together with DS and ES respectively in order to distinguish between the source and destination addresses.**



EU Registers

Source Index (SI) and Destination Index (DI)

- **Used in indexed addressing.**
- **Instructions that process data strings use the SI and DI registers together with DS and ES respectively in order to distinguish between the source and destination addresses.**



Flag Register

Auxiliary 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.

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.

Sign Flag

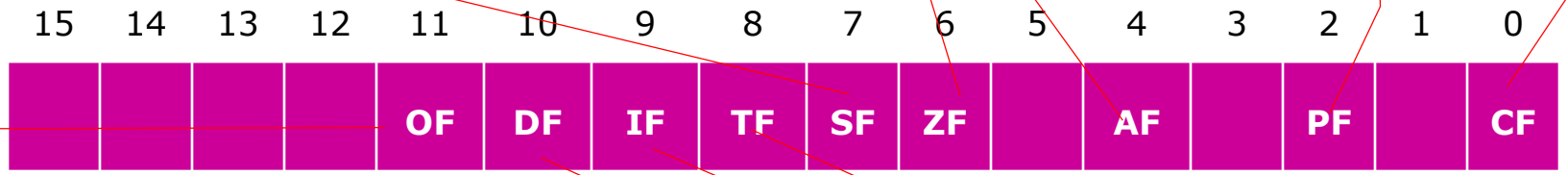
This flag is set, when the result of any computation is negative

Zero Flag

This flag is set, if the result of the computation or comparison performed by an instruction is zero

Parity Flag

This flag is set to 1, if the lower byte of the result contains even number of 1's ; for odd number of 1's set to zero.



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, then the overflow will be set.

Tarp Flag

If this flag is set, the processor enters the single step execution mode by generating internal interrupts after the execution of each instruction

Direction Flag

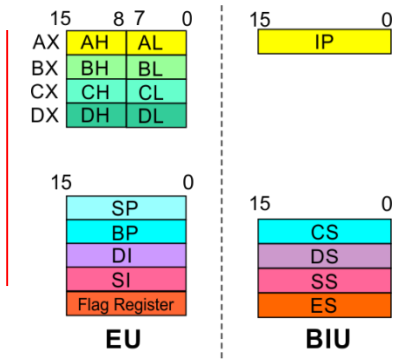
This is used by string manipulation instructions. If this flag bit is '0', the string is processed beginning from the lowest address to the highest address, i.e., auto incrementing mode. Otherwise, the string is processed from the highest address towards the lowest address, i.e., auto incrementing mode.

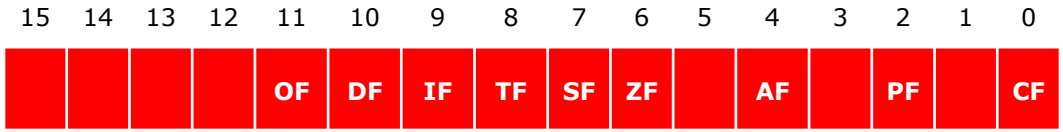
Interrupt Flag

Causes the 8086 to recognize external mask interrupts; clearing IF disables these interrupts.

Architecture

8086 registers categorized into 4 groups





Sl.No.	Type	Register width	Name of register
1	General purpose register	16 bit	AX, BX, CX, DX
		8 bit	AL, AH, BL, BH, CL, CH, DL, DH
2	Pointer register	16 bit	SP, BP
3	Index register	16 bit	SI, DI
4	Instruction Pointer	16 bit	IP
5	Segment register	16 bit	CS, DS, SS, ES
6	Flag (PSW)	16 bit	Flag register

Architecture

Registers and Special Functions

Register	Name of the Register	Special Function
AX	16-bit Accumulator	Stores the 16-bit results of arithmetic and logic operations
AL	8-bit Accumulator	Stores the 8-bit results of arithmetic and logic operations
BX	Base register	Used to hold base value in base addressing mode to access memory data
CX	Count Register	Used to hold the count value in SHIFT, ROTATE and LOOP instructions
DX	Data Register	Used to hold data for multiplication and division operations
SP	Stack Pointer	Used to hold the offset address of top stack memory
BP	Base Pointer	Used to hold the base value in base addressing using SS register to access data from stack memory
SI	Source Index	Used to hold index value of source operand (data) for string instructions
DI	Data Index	Used to hold the index value of destination operand (data) for string operations

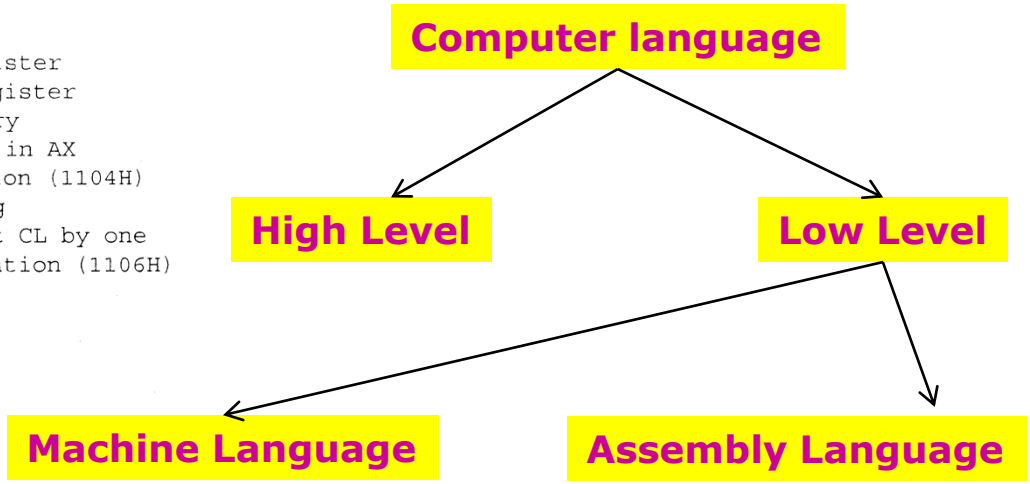
ADDRESSING MODES
&
Instruction set

Introduction

```
;PROGRAM TO ADD TWO 16-BIT DATA (METHOD-1)

DATA SEGMENT              ;Assembler directive

        ORG 1104H         ;Assembler directive
        SUM DW 0          ;Assembler directive
        CARRY DB 0        ;Assembler directive

DATA ENDS                 ;Assembler directive

CODE SEGMENT              ;Assembler directive

        ASSUME CS:CODE    ;Assembler directive
        ASSUME DS:DATA    ;Assembler directive
        ORG 1000H         ;Assembler directive

        MOV AX,205AH      ;Load the first data in AX register
        MOV BX,40EDH      ;Load the second data in BX register
        MOV CL,00H        ;Clear the CL register for carry
        ADD AX,BX         ;Add the two data, sum will be in AX
        MOV SUM,AX        ;Store the sum in memory location (1104H)
        JNC AHEAD         ;Check the status of carry flag
        INC CL            ;If carry flag is set,increment CL by one
AHEAD:  MOV CARRY,CL      ;Store the carry in memory location (1106H)
        HLT

CODE ENDS                 ;Assembler directive
END                       ;Assembler directive
```

Program
A set of instructions written to solve a problem.

Instruction
Directions which a microprocessor follows to execute a task or part of a task.



- **Binary bits**

- **English Alphabets**
- **'Mnemonics'**
- **Assembler**
  Mnemonics → Machine Language

ADDRESSING MODES

Addressing Modes

- Every instruction of a program has to operate on a data.
- The different ways in which a source operand is denoted in an instruction are known as addressing modes.

1. Register Addressing 2. Immediate Addressing	Group I : Addressing modes for register and immediate data
3. Direct Addressing 4. Register Indirect Addressing 5. Based Addressing 6. Indexed Addressing 7. Based Index Addressing 8. String Addressing	Group II : Addressing modes for memory data
9. Direct I/O port Addressing 10. Indirect I/O port Addressing	Group III : Addressing modes for I/O ports
11. Relative Addressing	Group IV : Relative Addressing mode
12. Implied Addressing	Group V : Implied Addressing mode

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing

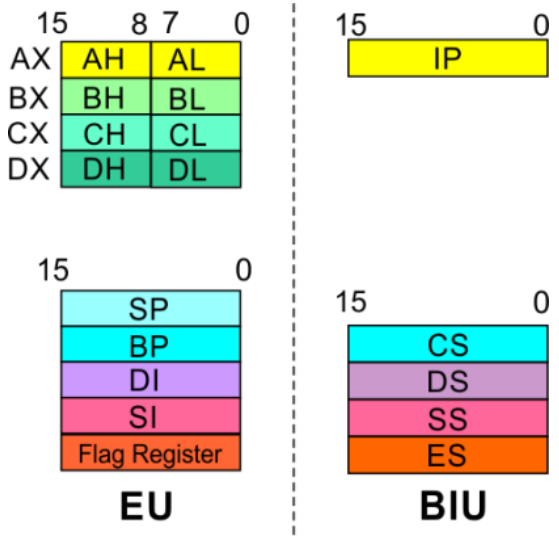
The instruction will specify the name of the register which holds the data to be operated by the instruction.

Example:

MOV CL, DH

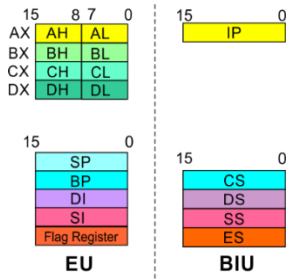
The content of 8-bit register DH is moved to another 8-bit register CL

(CL) ← (DH)



Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



In immediate addressing mode, an 8-bit or 16-bit data is specified as part of the instruction

Example:

MOV DL, 08H

The 8-bit data (08_H) given in the instruction is moved to DL

(DL) ← 08_H

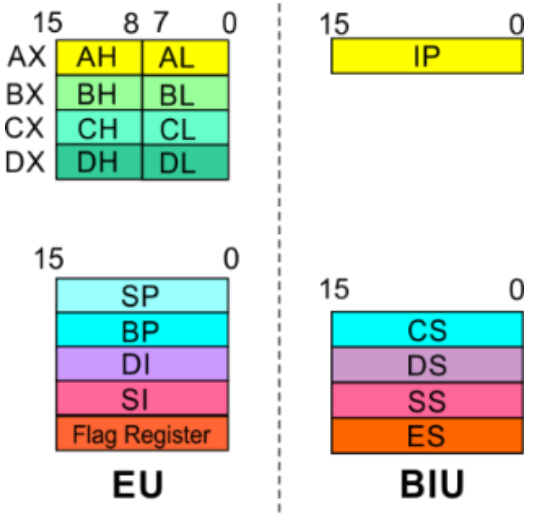
MOV AX, 0A9FH

The 16-bit data ($0A9F_H$) given in the instruction is moved to AX register

(AX) ← $0A9F_H$

Addressing Modes : Memory Access

- **20 Address lines ⇒ 8086 can address up to 2^{20} = 1M bytes of memory**
- **However, the largest register is only 16 bits**
- **Physical Address will have to be calculated**
  Physical Address : Actual address of a byte in memory. i.e. the value which goes out onto the address bus.
- **Memory Address represented in the form –**
  Seg : Offset (Eg - 89AB:F012)
- **Each time the processor wants to access memory, it takes the contents of a segment register, shifts it one hexadecimal place to the left (same as multiplying by 16_{10}), then add the required offset to form the 20- bit address**

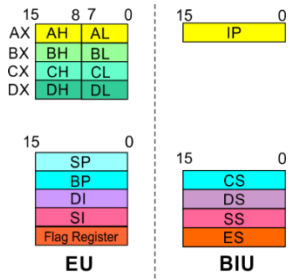


16 bytes of contiguous memory

89AB : F012 → 89AB → 89AB0 (Paragraph to byte → 89AB x 10 = 89AB0)
F012 → 0F012 (Offset is already in byte unit)
+ -------
98AC2 (The absolute address)

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



Here, the effective address of the memory location at which the data operand is stored is given in the instruction.

The effective address is just a 16-bit number written directly in the instruction.

Example:

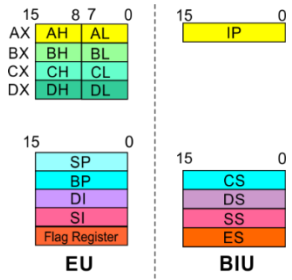
```
MOV  BX, [1354H]
MOV  BL, [0400H]
```

The square brackets around the 1354_H denotes the contents of the memory location. When executed, this instruction will copy the contents of the memory location into BX register.

This addressing mode is called direct because the displacement of the operand from the segment base is specified directly in the instruction.

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



In Register indirect addressing, name of the register which holds the effective address (EA) will be specified in the instruction.

Registers used to hold EA are any of the following registers:

BX, BP, DI and SI.

Content of the DS register is used for base address calculation.

Example:

MOV CX, [BX]

Note : Register/ memory enclosed in brackets refer to content of register/ memory

Operations:

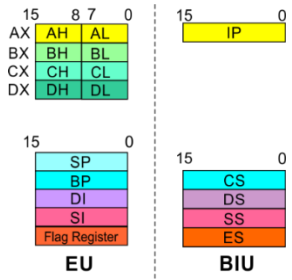
EA = (BX)
BA = (DS) x 16_{10}
MA = BA + EA

(CX) ← (MA) or,

(CL) ← (MA)
(CH) ← (MA +1)

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



In Based Addressing, BX or BP is used to hold the base value for effective address and a signed 8-bit or unsigned 16-bit displacement will be specified in the instruction.

In case of 8-bit displacement, it is sign extended to 16-bit before adding to the base value.

When BX holds the base value of EA, 20-bit physical address is calculated from BX and DS.

When BP holds the base value of EA, BP and SS is used.

Example:

MOV AX, [BX + 08H]

Operations:

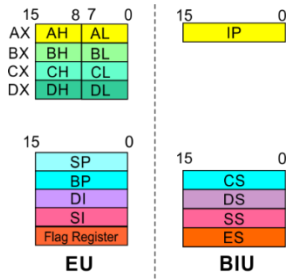
$0008_H \leftarrow 08_H$ (Sign extended)
EA = (BX) + 0008_H
BA = (DS) x 16_{10}
MA = BA + EA

(AX) ← (MA) or,

(AL) ← (MA)
(AH) ← (MA + 1)

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



SI or DI register is used to hold an index value for memory data and a signed 8-bit or unsigned 16-bit displacement will be specified in the instruction.

Displacement is added to the index value in SI or DI register to obtain the EA.

In case of 8-bit displacement, it is sign extended to 16-bit before adding to the base value.

Example:

MOV CX, [SI + 0A2H]

Operations:

$FFA2_H \leftarrow A2_H$ (Sign extended)

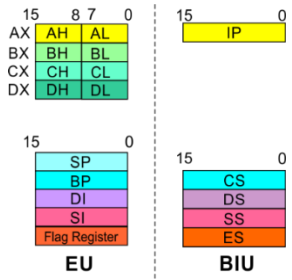
$EA = (SI) + FFA2_H$
$BA = (DS) \times 16_{10}$
$MA = BA + EA$

$(CX) \leftarrow (MA)$ or,

$(CL) \leftarrow (MA)$
$(CH) \leftarrow (MA + 1)$

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



In Based Index Addressing, the effective address is computed from the sum of a base register (BX or BP), an index register (SI or DI) and a displacement.

Example:

MOV DX, [BX + SI + 0AH]

Operations:

$000A_H \leftarrow 0A_H$ (Sign extended)

$EA = (BX) + (SI) + 000A_H$
$BA = (DS) \times 16_{10}$
$MA = BA + EA$

$(DX) \leftarrow (MA)$ or,

$(DL) \leftarrow (MA)$
$(DH) \leftarrow (MA + 1)$

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing

Note : Effective address of the Extra segment register

Employed in string operations to operate on string data.

The effective address (EA) of source data is stored in SI register and the EA of destination is stored in DI register.

Segment register for calculating base address of source data is DS and that of the destination data is ES

Example: MOVS BYTE

Operations:

Calculation of source memory location:

EA = (SI) BA = (DS) x 16_{10} MA = BA + EA

Calculation of destination memory location:

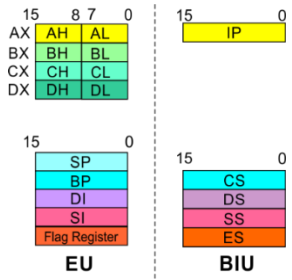
EA_E = (DI) BA_E = (ES) x 16_{10} MA_E = BA_E + EA_E

(MAE) ← (MA)

If DF = 1, then (SI) ← (SI) – 1 and (DI) = (DI) - 1
If DF = 0, then (SI) ← (SI) +1 and (DI) = (DI) + 1

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied 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.

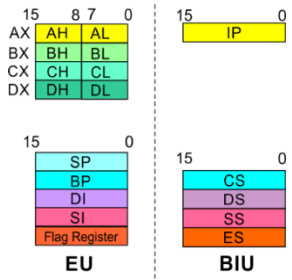
Example: **IN AL, [09H]**

Operations: $PORT_{addr} = 09_H$
$(AL) \leftarrow (PORT)$

Content of port with address 09_H is moved to AL register

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



In this addressing mode, the effective address of a program instruction is specified relative to Instruction Pointer (IP) by an 8-bit signed displacement.

Example: JZ 0AH

Operations:

$000A_H \leftarrow 0A_H$ **(sign extend)**

If ZF = 1, then

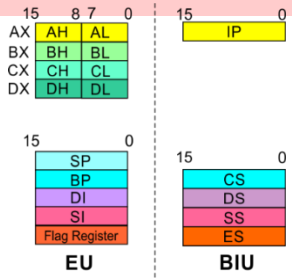
$EA = (IP) + 000A_H$
$BA = (CS) \times 16_{10}$
$MA = BA + EA$

If ZF = 1, then the program control jumps to new address calculated above.

If ZF = 0, then next instruction of the program is executed.

Addressing Modes

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



Instructions using this mode have no operands. The instruction itself will specify the data to be operated by the instruction.

Example: **CLC**

This clears the carry flag to zero.

INSTRUCTION SET

Instruction Set

8086 supports 6 types of instructions.

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

Instruction Set

8086 supports 6 types of instructions-

1. Data Transfer Instructions

Mnemonics: MOV, XCHG, PUSH, POP, IN, OUT

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC,MUL,DIV,CMP

3. Logical Instructions

Mnemonics: AND, OR, XOR, TEST, SHR, SHL, RCR, RCL

Instruction Set

4. String Manipulation Instructions

Mnemonics: REP, MOVS,

5. Processor Control Instructions

Mnemonics: STC, CMC, STD, CLD

6. Control Transfer Instructions

Mnemonics: CALL, RET, JMP

Instruction Set

1. Data Transfer Instructions

Instructions that are used to transfer data/ address in to registers, memory locations and I/O ports.

Generally involve two operands: Source operand and Destination operand of the same size.

Source: Register or a memory location or an immediate data
Destination : Register or a memory location.

The size should be a either a byte or a word.

A 8-bit data can only be moved to 8-bit register/ memory and a 16-bit data can be moved to 16-bit register/ memory.

Instruction Set

1. Data Transfer Instructions

Mnemonics: MOV, XCHG, PUSH, POP, IN, OUT …

MOV reg2/ mem, reg1/ mem	
MOV reg2, reg1	**(reg2) ← (reg1)**
MOV mem, reg1	**(mem) ← (reg1)**
MOV reg2, mem	**(reg2) ← (mem)**
MOV reg/ mem, data	
MOV reg, data	**(reg) ← data**
MOV mem, data	**(mem) ← data**

XCHG reg2/ mem, reg1	
XCHG reg2, reg1	**(reg2) ↔ (reg1)**
XCHG mem, reg1	**(mem) ↔ (reg1)**

Instruction Set

1. Data Transfer Instructions

Mnemonics: **MOV, XCHG, PUSH, POP, IN, OUT ...**

PUSH reg16/ mem	
PUSH reg16	**(SP) ← (SP) – 2** **MA_S = (SS) x 16_{10} + SP** **(MA_S ; MA_S + 1) ← (reg16)**
PUSH mem	**(SP) ← (SP) – 2** **MA_S = (SS) x 16_{10} + SP** **(MA_S ; MA_S + 1) ← (mem)**
POP reg16/ mem	
POP reg16	**MA_S = (SS) x 16_{10} + SP** **(reg16) ← (MA_S ; MA_S + 1)** **(SP) ← (SP) + 2**
POP mem	**MA_S = (SS) x 16_{10} + SP** **(mem) ← (MA_S ; MA_S + 1)** **(SP) ← (SP) + 2**

Instruction Set

1. Data Transfer Instructions

Mnemonics: **MOV, XCHG, PUSH, POP, IN, OUT ...**

IN A, [DX]	
IN AL, [DX]	**$PORT_{addr}$ = (DX)** **(AL) ← (PORT)**
IN AX, [DX]	**$PORT_{addr}$ = (DX)** **(AX) ← (PORT)**
IN A, addr8	
IN AL, addr8	**(AL) ← (addr8)**
IN AX, addr8	**(AX) ← (addr8)**

OUT [DX], A	
OUT [DX], AL	**$PORT_{addr}$ = (DX)** **(PORT) ← (AL)**
OUT [DX], AX	**$PORT_{addr}$ = (DX)** **(PORT) ← (AX)**
OUT addr8, A	
OUT addr8, AL	**(addr8) ← (AL)**
OUT addr8, AX	**(addr8) ← (AX)**

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

ADD reg2/ mem, reg1/mem	
ADC reg2, reg1	**(reg2) ← (reg1) + (reg2)**
ADC reg2, mem	**(reg2) ← (reg2) + (mem)**
ADC mem, reg1	**(mem) ← (mem)+(reg1)**
ADD reg/mem, data	
ADD reg, data	**(reg) ← (reg)+ data**
ADD mem, data	**(mem) ← (mem)+data**
ADD A, data	
ADD AL, data8	**(AL) ← (AL) + data8**
ADD AX, data16	**(AX) ← (AX) +data16**

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

ADC reg2/ mem, reg1/mem	
ADC reg2, reg1	(reg2) ← (reg1) + (reg2)+CF
ADC reg2, mem	(reg2) ← (reg2) + (mem)+CF
ADC mem, reg1	(mem) ← (mem)+(reg1)+CF
ADC reg/mem, data	
ADC reg, data	(reg) ← (reg)+ data+CF
ADC mem, data	(mem) ← (mem)+data+CF
ADDC A, data	
ADD AL, data8	(AL) ← (AL) + data8+CF
ADD AX, data16	(AX) ← (AX) +data16+CF

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

SUB reg2/ mem, reg1/mem **SUB reg2, reg1** **SUB reg2, mem** **SUB mem, reg1**	 **(reg2) ← (reg1) - (reg2)** **(reg2) ← (reg2) - (mem)** **(mem) ← (mem) - (reg1)**
SUB reg/mem, data **SUB reg, data** **SUB mem, data**	 **(reg) ← (reg) - data** **(mem) ← (mem) - data**
SUB A, data **SUB AL, data8** **SUB AX, data16**	 **(AL) ← (AL) - data8** **(AX) ← (AX) - data16**

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

INC reg/ mem	
INC reg8	**(reg8) ← (reg8) + 1**
INC reg16	**(reg16) ← (reg16) + 1**
INC mem	**(mem) ← (mem) + 1**
DEC reg/ mem	
DEC reg8	**(reg8) ← (reg8) - 1**
DEC reg16	**(reg16) ← (reg16) - 1**
DEC mem	**(mem) ← (mem) - 1**

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

MUL reg/ mem	
MUL reg	**For byte : (AX) ← (AL) x (reg8)** **For word : (DX)(AX) ← (AX) x (reg16)**
MUL mem	**For byte : (AX) ← (AL) x (mem8)** **For word : (DX)(AX) ← (AX) x (mem16)**
IMUL reg/ mem	
IMUL reg	**For byte : (AX) ← (AL) x (reg8)** **For word : (DX)(AX) ← (AX) x (reg16)**
IMUL mem	**For byte : (AX) ← (AX) x (mem8)** **For word : (DX)(AX) ← (AX) x (mem16)**

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

DIV reg/ mem	
DIV reg	**For 16-bit :- 8-bit :** **(AL) ← (AX) :- (reg8) Quotient** **(AH) ← (AX) MOD(reg8) Remainder** **For 32-bit :- 16-bit :** **(AX) ← (DX)(AX) :- (reg16) Quotient** **(DX) ← (DX)(AX) MOD(reg16) Remainder**
DIV mem	**For 16-bit :- 8-bit :** **(AL) ← (AX) :- (mem8) Quotient** **(AH) ← (AX) MOD(mem8) Remainder** **For 32-bit :- 16-bit :** **(AX) ← (DX)(AX) :- (mem16) Quotient** **(DX) ← (DX)(AX) MOD(mem16) Remainder**

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

IDIV reg/ mem	
IDIV reg	**For 16-bit :- 8-bit :** **(AL) ← (AX) :- (reg8) Quotient** **(AH) ← (AX) MOD(reg8) Remainder** **For 32-bit :- 16-bit :** **(AX) ← (DX)(AX) :- (reg16) Quotient** **(DX) ← (DX)(AX) MOD(reg16) Remainder**
IDIV mem	**For 16-bit :- 8-bit :** **(AL) ← (AX) :- (mem8) Quotient** **(AH) ← (AX) MOD(mem8) Remainder** **For 32-bit :- 16-bit :** **(AX) ← (DX)(AX) :- (mem16) Quotient** **(DX) ← (DX)(AX) MOD(mem16) Remainder**

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

CMP reg2/mem, reg1/ mem	

CMP reg2, reg1	Modify flags ← (reg2) – (reg1)
	
	If (reg2) > (reg1) then CF=0, ZF=0, SF=0
	If (reg2) < (reg1) then CF=1, ZF=0, SF=1
	If (reg2) = (reg1) then CF=0, ZF=1, SF=0

CMP reg2, mem	Modify flags ← (reg2) – (mem)
	
	If (reg2) > (mem) then CF=0, ZF=0, SF=0
	If (reg2) < (mem) then CF=1, ZF=0, SF=1
	If (reg2) = (mem) then CF=0, ZF=1, SF=0

CMP mem, reg1	Modify flags ← (mem) – (reg1)
	
	If (mem) > (reg1) then CF=0, ZF=0, SF=0
	If (mem) < (reg1) then CF=1, ZF=0, SF=1
	If (mem) = (reg1) then CF=0, ZF=1, SF=0

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

CMP reg/mem, data	

CMP reg, data	Modify flags ← (reg) – (data)
	
	If (reg) > data then CF=0, ZF=0, SF=0
	If (reg) < data then CF=1, ZF=0, SF=1
	If (reg) = data then CF=0, ZF=1, SF=0

CMP mem, data	Modify flags ← (mem) – (mem)
	
	If (mem) > data then CF=0, ZF=0, SF=0
	If (mem) < data then CF=1, ZF=0, SF=1
	If (mem) = data then CF=0, ZF=1, SF=0

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...

CMP A, data	
CMP AL, data8	**Modify flags ← (AL) – data8** **If (AL) > data8 then CF=0, ZF=0, SF=0** **If (AL) < data8 then CF=1, ZF=0, SF=1** **If (AL) = data8 then CF=0, ZF=1, SF=0**
CMP AX, data16	**Modify flags ← (AX) – data16** **If (AX) > data16 then CF=0, ZF=0, SF=0** **If (mem) < data16 then CF=1, ZF=0, SF=1** **If (mem) = data16 then CF=0, ZF=1, SF=0**

Instruction Set

3. Logical Instructions

Mnemonics: AND, OR, XOR, TEST, SHR, SHL, RCR, RCL …

AND A, data AND AL, data8	(AL) ← (AL) & data8
AND AX, data16	(AX) ← (AX) & data16

AND reg/mem, data AND reg, data	(reg) ← (reg) & data
AND mem, data	(mem) ← (mem) & data

Instruction Set

3. Logical Instructions

Mnemonics: AND, OR, XOR, TEST, SHR, SHL, RCR, RCL …

OR reg2/mem, reg1/mem OR reg2, reg1	(reg2) ← (reg2) | (reg1)
OR reg2, mem	(reg2) ← (reg2) | (mem)
OR mem, reg1	(mem) ← (mem) | (reg1)

OR reg/mem, data OR reg, data	(reg) ← (reg) | data
OR mem, data	(mem) ← (mem) | data

OR A, data OR AL, data8	(AL) ← (AL) | data8
OR AX, data16	(AX) ← (AX) | data16

Instruction Set

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL …**

XOR reg2/mem, reg1/mem	
XOR reg2, reg1	(reg2) ← (reg2) ^ (reg1)
XOR reg2, mem	(reg2) ← (reg2) ^ (mem)
XOR mem, reg1	(mem) ← (mem) ^ (reg1)

XOR reg/mem, data	
XOR reg, data	(reg) ← (reg) ^ data
XOR mem, data	(mem) ← (mem) ^ data

XOR A, data	
XOR AL, data8	(AL) ← (AL) ^ data8
XOR AX, data16	(AX) ← (AX) ^ data16