8086 MICROPROCESSOR

Dr. M. Gopikrishna
Assistant Professor of Physics
Maharajas College
Ernakulam

Unit II

8086 Microprocessor (19 Hrs)

The Intel 8086 - architecture - MN/MX modes - 8086 addressing modes - instruction set- instruction format - assembler directives and operators - Programming with 8086 - interfacing memory and I/O ports - Comparison of 8086 and 8088 - Coprocessors - Intel 8087 - Familiarisation with Debug utility.

Program controlled semiconductor device (IC) which fetches (from memory), decodes and executes instructions.

It is used as CPU (Central Processing Unit) in computers.

MICROPROCESSOR

Fifth Generation **Pentium**

Fourth Generation
During 1980s
Low power version of HMOS technology (HCMOS)
32 bit processors
Physical memory space 2^{24} bytes = 16 Mb
Virtual memory space 2^{40} bytes = 1 Tb
Floating point hardware
Supports increased number of addressing modes

Intel 80386

Third Generation
During 1978
HMOS technology ⇒ Faster speed, Higher packing density
16 bit processors ⇒ 40/ 48/ 64 pins
Easier to program
Dynamically relatable programs
Processor has multiply/ divide arithmetic hardware
More powerful interrupt handling capabilities
Flexible I/O port addressing

Intel 8086 (16 bit processor)

Second Generation
During 1973
NMOS technology ⇒ Faster speed, Higher density, Compatible with TTL
4 / 8/ 16 bit processors ⇒ 40 pins
Ability to address large memory spaces and I/O ports
Greater number of levels of subroutine nesting
Better interrupt handling capabilities

Intel 8085 (8 bit processor)

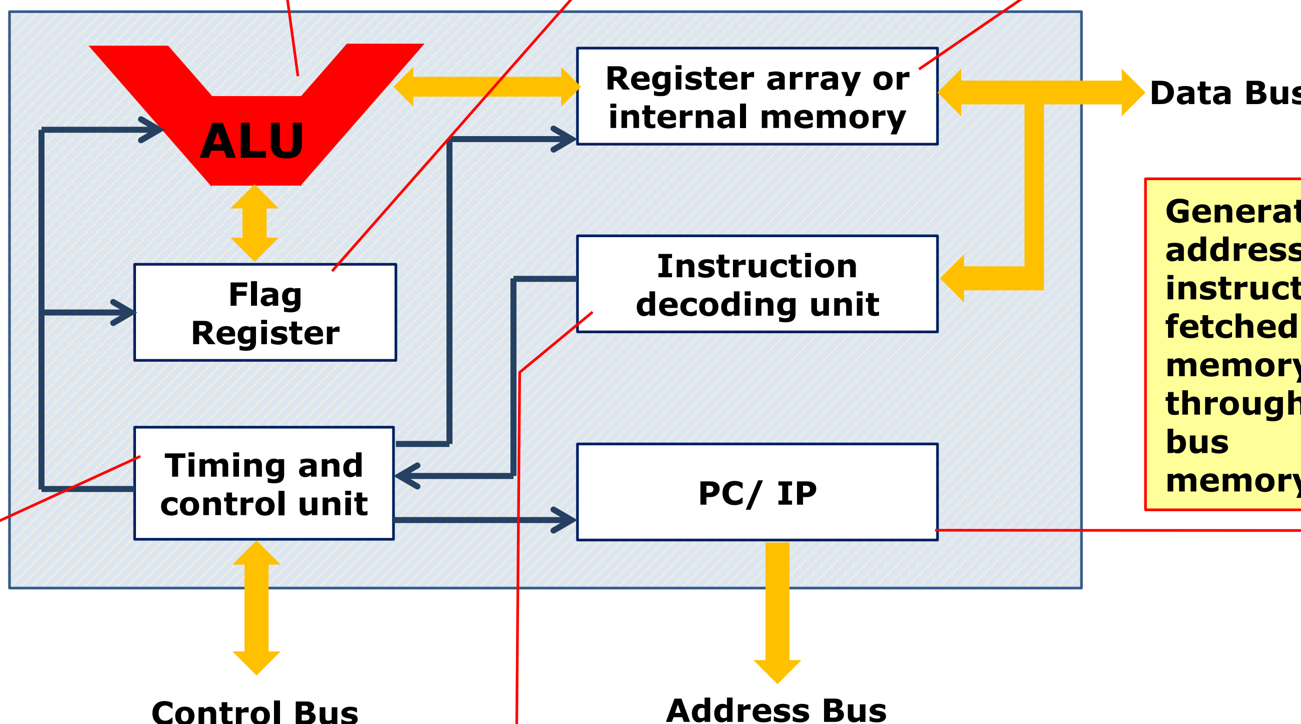
First Generation
Between 1971 – 1973
PMOS technology, non compatible with TTL
4 bit processors ⇒ 16 pins
8 and 16 bit processors ⇒ 40 pins
Due to limitations of pins, signals are multiplexed

Functional blocks

Computational Unit; performs arithmetic and logic operations

Various conditions of the results are stored as status bits called flags in flag register

Internal storage of data



Generates the address of the instructions to be fetched from the memory and send through address bus to the memory

Generates control signals for internal and external operations of the microprocessor

Decodes instructions; sends information to the timing and control unit

Overview

First 16- bit processor released by INTEL in the year 1978

Originally HMOS, now manufactured using HMOS III technique

Approximately 29, 000 transistors, 40 pin DIP, 5V supply

Does not have internal clock; external asymmetric clock source with 33% duty cycle

20-bit address to access memory ⇒ can address up to 2^{20} = 1 megabytes of memory space.

Addressable memory space is organized in to two banks of 512 kb each; Even (or lower) bank and Odd (or higher) bank. Address line A_0 is used to select even bank and control signal $\overline{BHE}$ is used to access odd bank

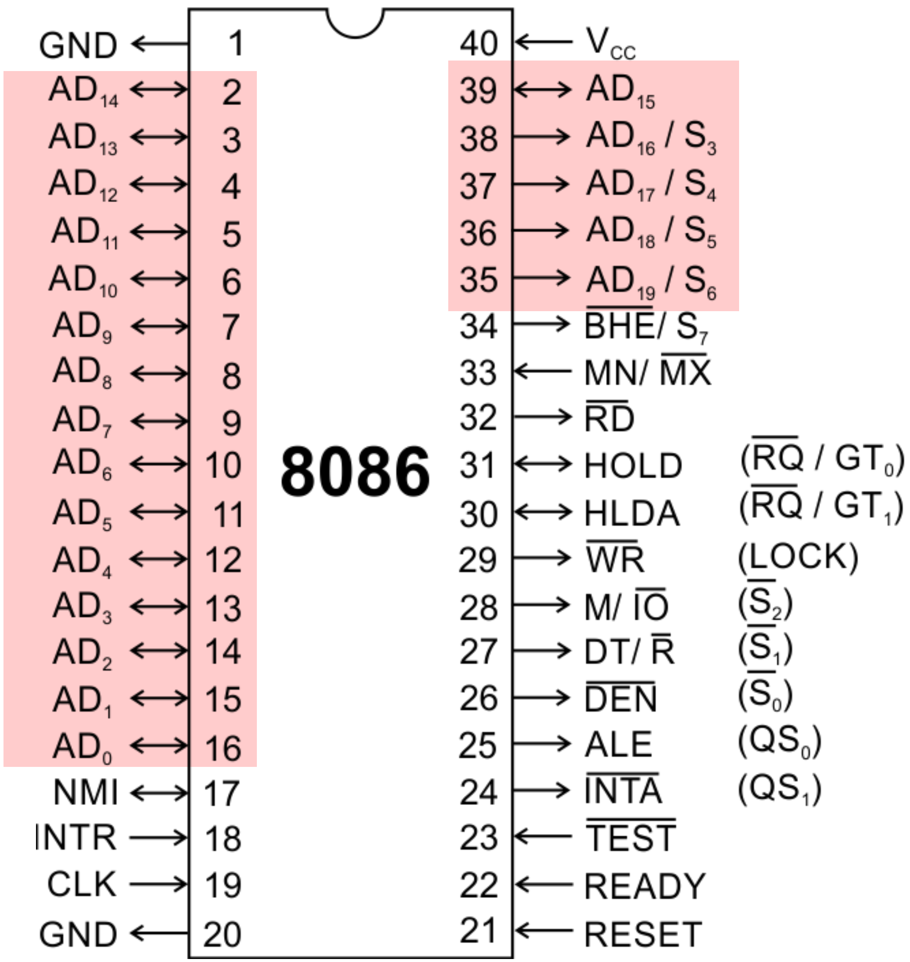
Uses a separate 16 bit address for I/O mapped devices ⇒ can generate 2^{16} = 64 k addresses.

Operates in two modes: minimum mode and maximum mode, decided by the signal at MN and $\overline{MX}$ pins.

Pins and Signals

Pins and Signals

Common signals



AD_0-AD_{15} (Bidirectional)

Address/Data bus

Low order address bus; these are multiplexed with data.

When AD lines are used to transmit memory address the symbol A is used instead of AD, for example A_0-A_{15}.

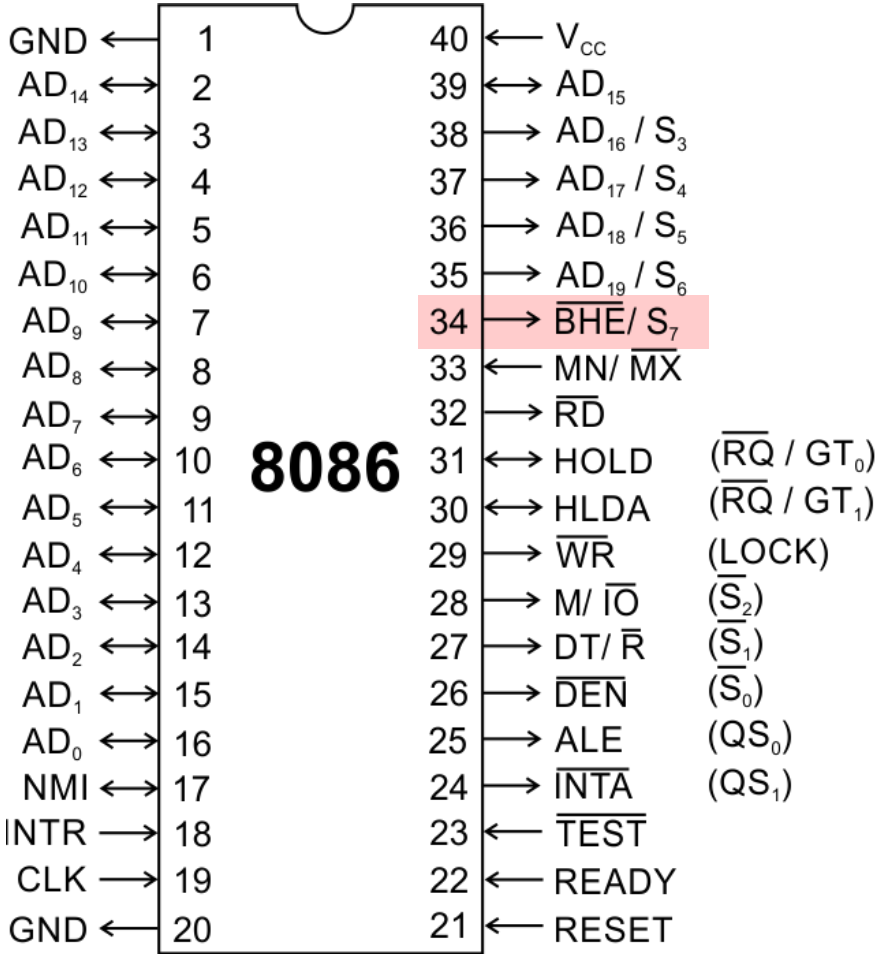
When data are transmitted over AD lines the symbol D is used in place of AD, for example D_0-D_7, D_8-D_{15} or D_0-D_{15}.

A_{16}/S_3, A_{17}/S_4, A_{18}/S_5, A_{19}/S_6

High order address bus. These are multiplexed with status signals

Pins and Signals

Common signals



BHE (Active Low)/S_7 (Output)

Bus High Enable/Status

It is used to enable data onto the most significant half of data bus, D_8-D_{15}. 8-bit device connected to upper half of the data bus use BHE (Active Low) signal. It is multiplexed with status signal S_7.

MN/ MX

MINIMUM / MAXIMUM

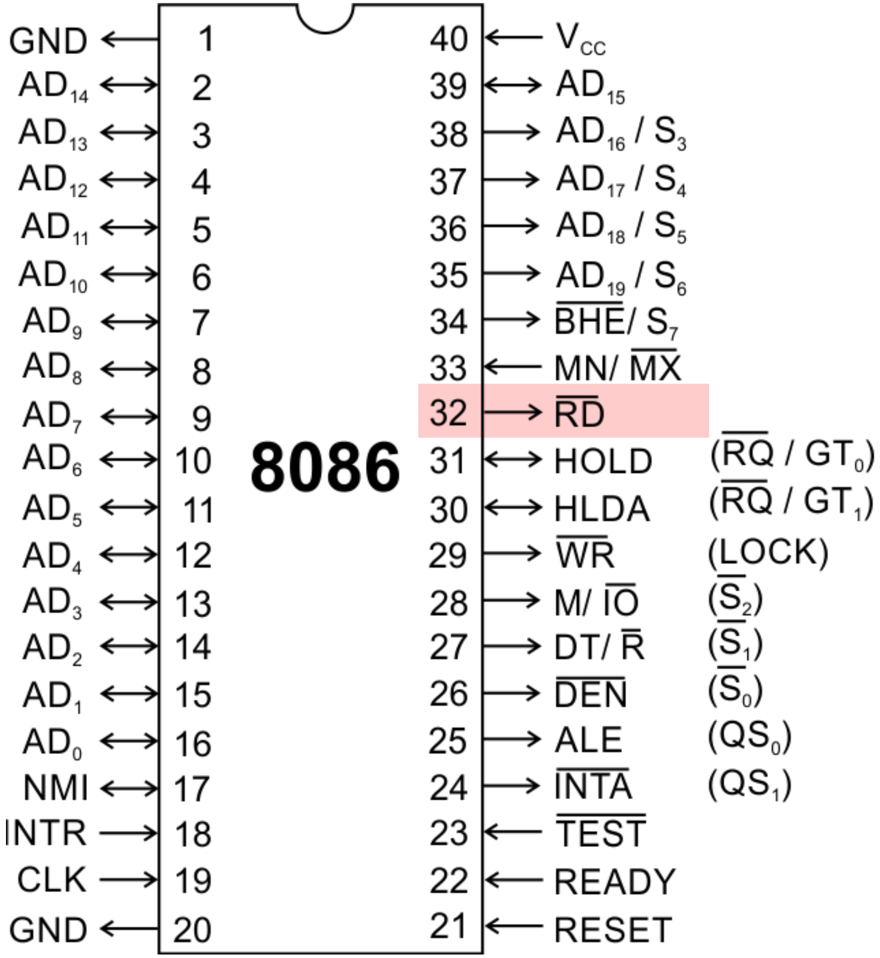
This pin signal indicates what mode the processor is to operate in.

RD (Read) (Active Low)

The signal is used for read operation.
It is an output signal.
It is active when low.

Pins and Signals

Common signals



TEST

$\overline{\text{TEST}}$ input is tested by the 'WAIT' instruction.

8086 will enter a wait state after execution of the WAIT instruction and will resume execution only when the $\overline{\text{TEST}}$ is made low by an active hardware.

This is used to synchronize an external activity to the processor internal operation.

READY

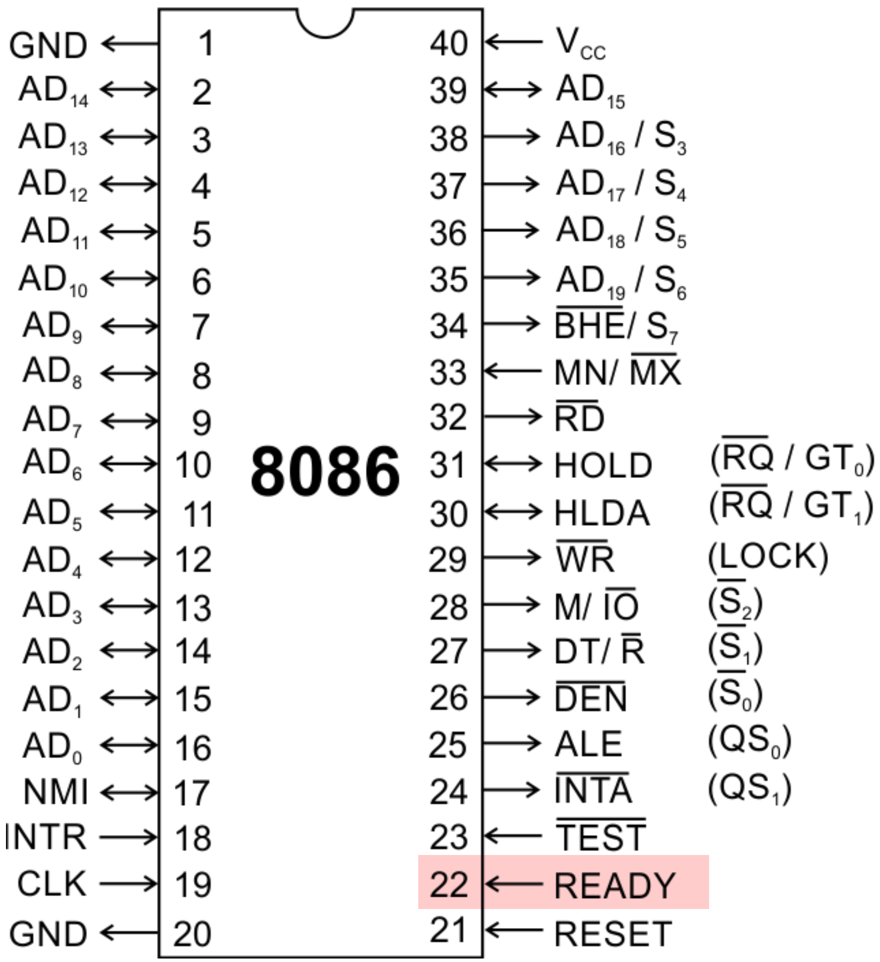
This is the acknowledgement from the slow device or memory that they have completed the data transfer.

The signal made available by the devices is synchronized by the 8284A clock generator to provide ready input to the 8086.

The signal is active high.

Pins and Signals

Common signals



RESET (Input)

Causes the processor to immediately terminate its present activity.

The signal must be active HIGH for at least four clock cycles.

CLK

The clock input provides the basic timing for processor operation and bus control activity. Its an asymmetric square wave with 33% duty cycle.

INTR Interrupt Request

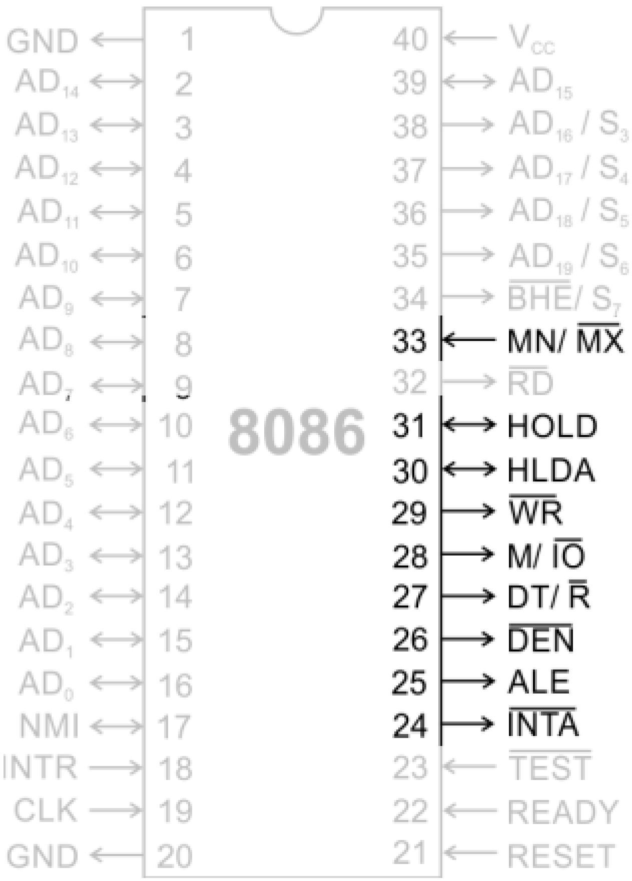
This is a triggered input. This is sampled during the last clock cycles of each instruction to determine the availability of the request. If any interrupt request is pending, the processor enters the interrupt acknowledge cycle.

This signal is active high and internally synchronized.

Pins and Signals

Min/ Max Pins

The 8086 microprocessor can work in two modes of operations : Minimum mode and Maximum mode.



In the <u>minimum mode</u> of operation the microprocessor <u>do not</u> associate with any co-processors and can not be used for multiprocessor systems.

In the <u>maximum mode</u> the 8086 <u>can work</u> in multi-processor or co-processor configuration.

Minimum or maximum mode operations are decided by the pin MN/ MX(Active low).

When this pin is <u>high</u> 8086 operates in <u>minimum mode</u> otherwise it operates in Maximum mode.

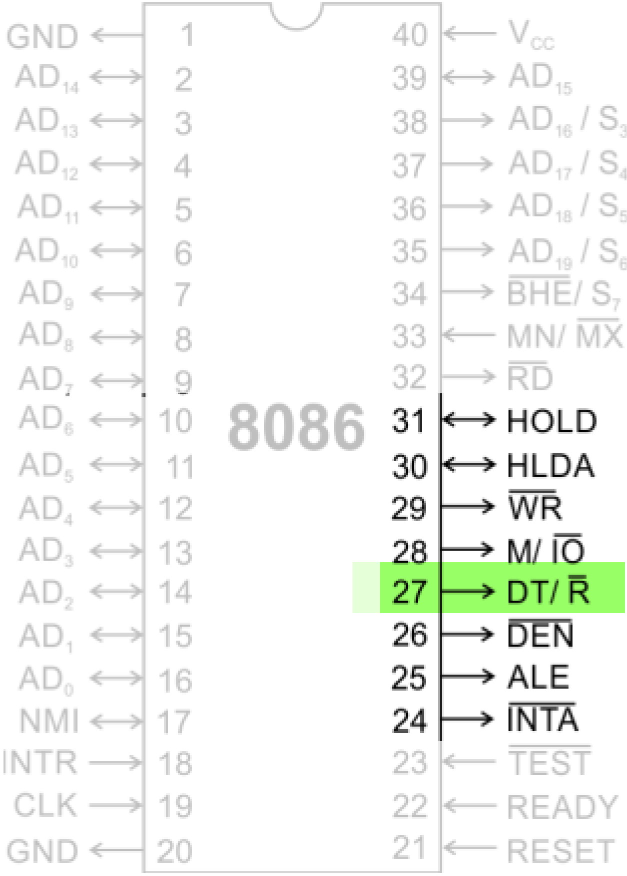
Pins and Signals

Minimum mode signals

Pins 24 -31

For minimum mode operation, the MN/ $\overline{MX}$ is tied to VCC (logic high)

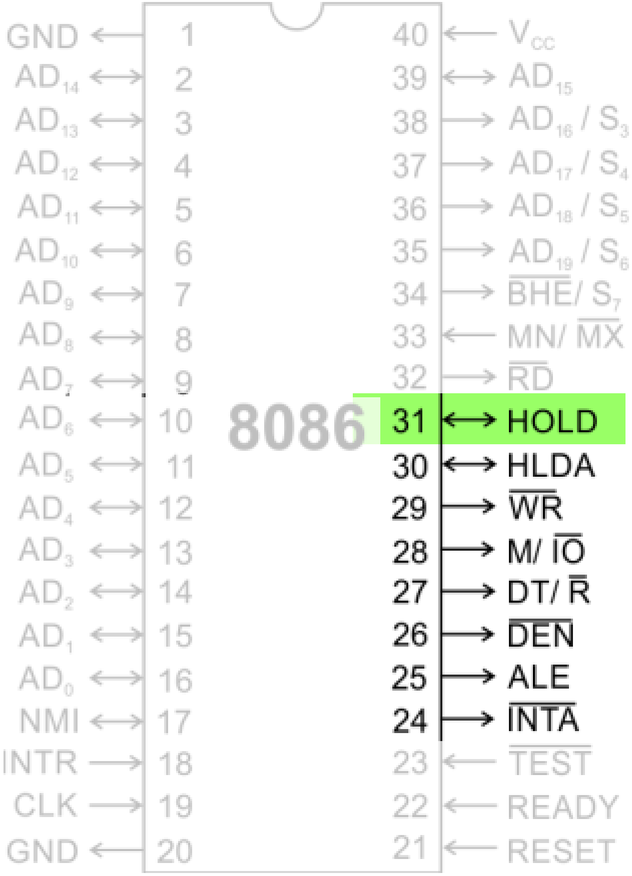
8086 itself generates all the bus control signals



DT/$\overline{R}$	**(Data Transmit/ Receive) Output signal from the processor to control the direction of data flow through the data transceivers**
$\overline{DEN}$	**(Data Enable) Output signal from the processor used as out put enable for the transceivers**
ALE	**(Address Latch Enable) Used to demultiplex the address and data lines using external latches**
M/$\overline{IO}$	**Used to differentiate memory access and I/O access. For memory reference instructions, it is high. For IN and OUT instructions, it is low.**
$\overline{WR}$	**Write control signal; asserted low Whenever processor writes data to memory or I/O port**
$\overline{INTA}$	**(Interrupt Acknowledge) When the interrupt request is accepted by the processor, the output is low on this line.**

Pins and Signals

Minimum mode signals



Pins 24 -31

For minimum mode operation, the MN/ $\overline{MX}$ is tied to VCC (logic high)

8086 itself generates all the bus control signals

HOLD — **Input signal to the processor form the bus masters as a request to grant the control of the bus.**

Usually used by the DMA controller to get the control of the bus.

HLDA — **(Hold Acknowledge) Acknowledge signal by the processor to the bus master requesting the control of the bus through HOLD.**

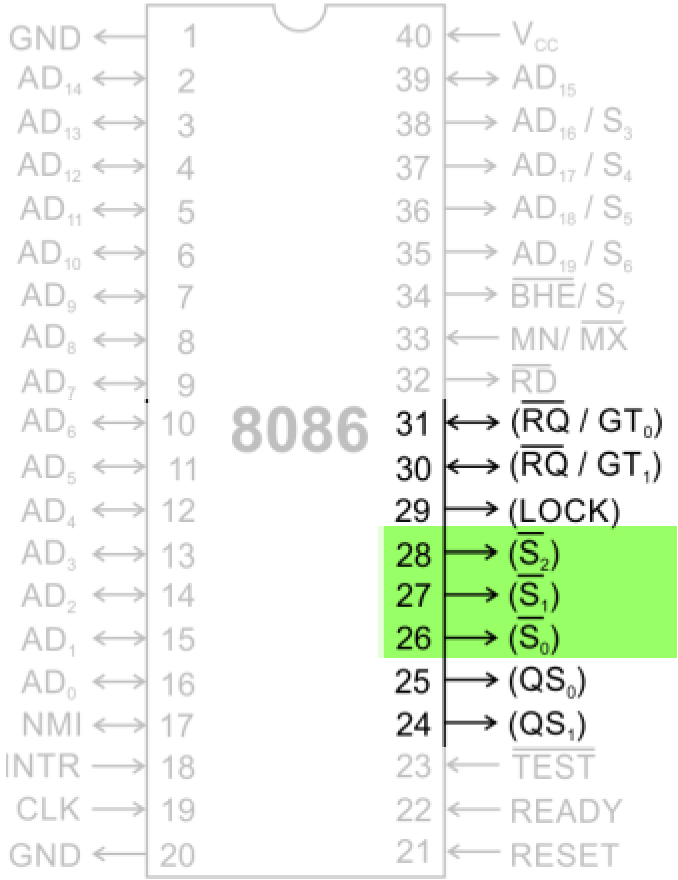
The acknowledge is asserted high, when the processor accepts HOLD.

Pins and Signals

Maximum mode signals

During maximum mode operation, the MN/ $\overline{MX}$ is grounded (logic low)

Pins 24 -31 are reassigned

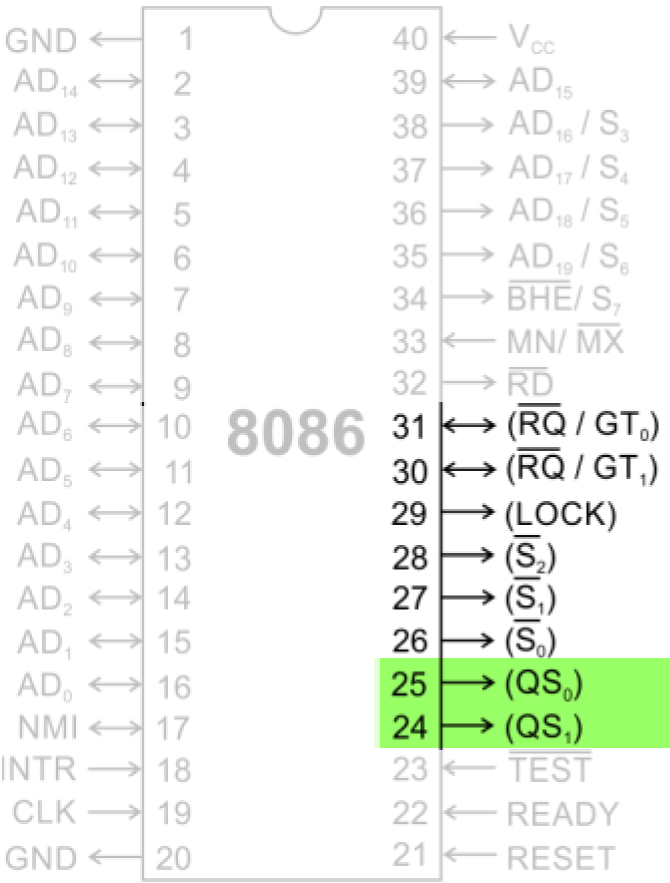


$\overline{S_0}, \overline{S_1}, \overline{S_2}$ **Status signals; used by the 8086 bus controller to generate bus timing and control signals. These are decoded as shown.**

Status Signal			Machine Cycle
$\overline{S}_2$	$\overline{S}_1$	$\overline{S}_0$	
0	0	0	Interrupt acknowledge
0	0	1	Read I/O port
0	1	0	Write I/O port
0	1	1	Halt
1	0	0	Code access
1	0	1	Read memory
1	1	0	Write memory
1	1	1	Passive/Inactive

During maximum mode operation, the MN/ $\overline{MX}$ is grounded (logic low)

Pins 24 -31 are reassigned



$\overline{QS_0}$, $\overline{QS_1}$

(Queue Status) The processor provides the status of queue in these lines.

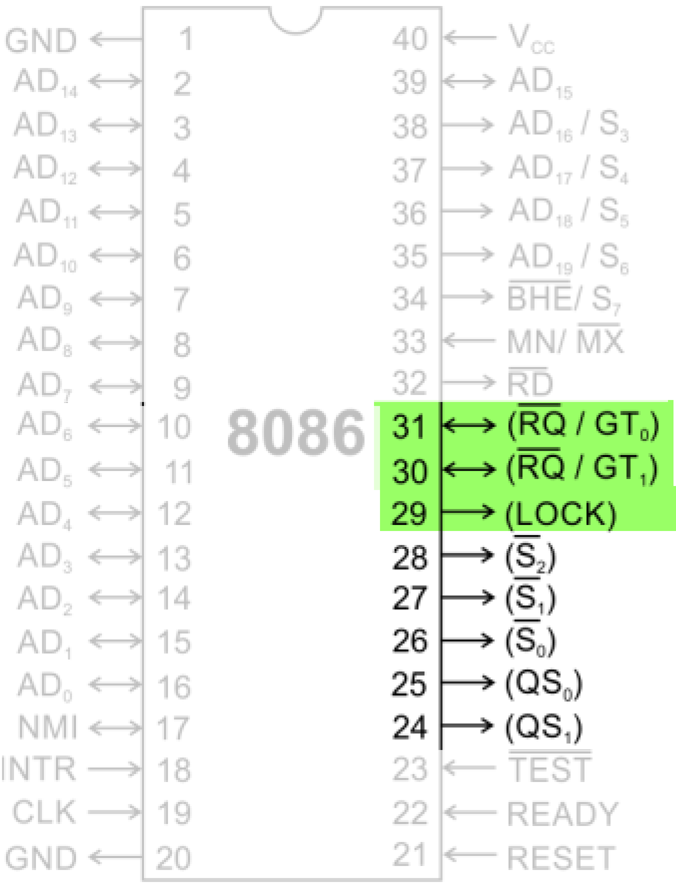
The queue status can be used by external device to track the internal status of the queue in 8086.

The output on QS_0 and QS_1 can be interpreted as shown in the table.

Queue status		Queue operation
QS_1	QS_0	
0	0	No operation
0	1	First byte of an opcode from queue
1	0	Empty the queue
1	1	Subsequent byte from queue

During maximum mode operation, the MN/ $\overline{MX}$ is grounded (logic low)

Pins 24 -31 are reassigned



$\overline{RQ}/\ \overline{GT_0}$, $\overline{RQ}/\ \overline{GT_1}$

(**Bus Request/ Bus Grant**) These requests are used by other local bus masters to force the processor to release the local bus at the end of the processor's current bus cycle.

These pins are bidirectional.

The request on $\overline{GT_0}$ will have higher priority than $\overline{GT_1}$

$\overline{LOCK}$

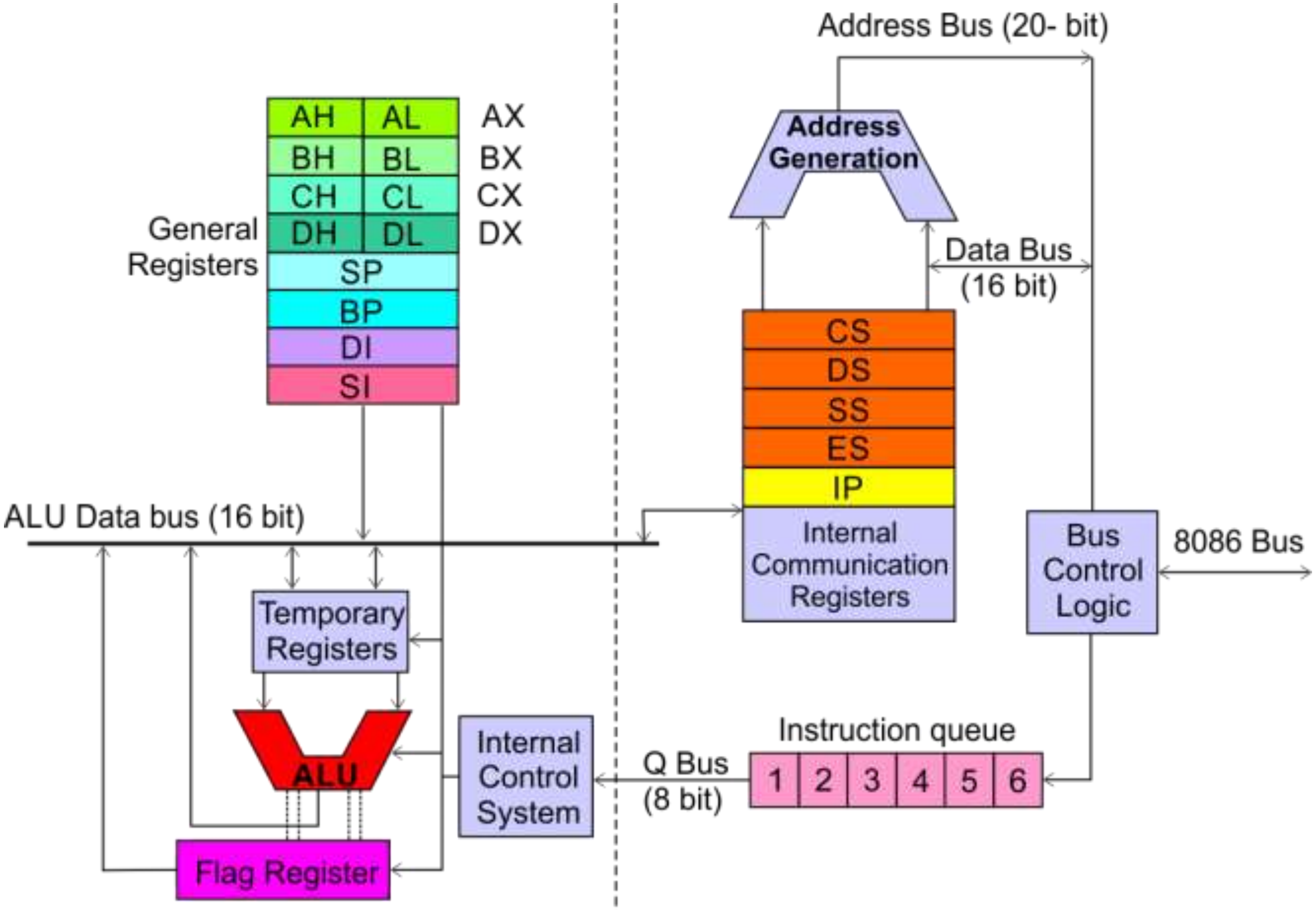
An output signal activated by the LOCK prefix instruction.

Remains active until the completion of the instruction prefixed by LOCK.

The 8086 output low on the $\overline{LOCK}$ pin while executing an instruction prefixed by LOCK to <u>prevent other bus masters from gaining control of the system bus.</u>

ARCHITECTURE

Architecture



Execution Unit (EU)

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

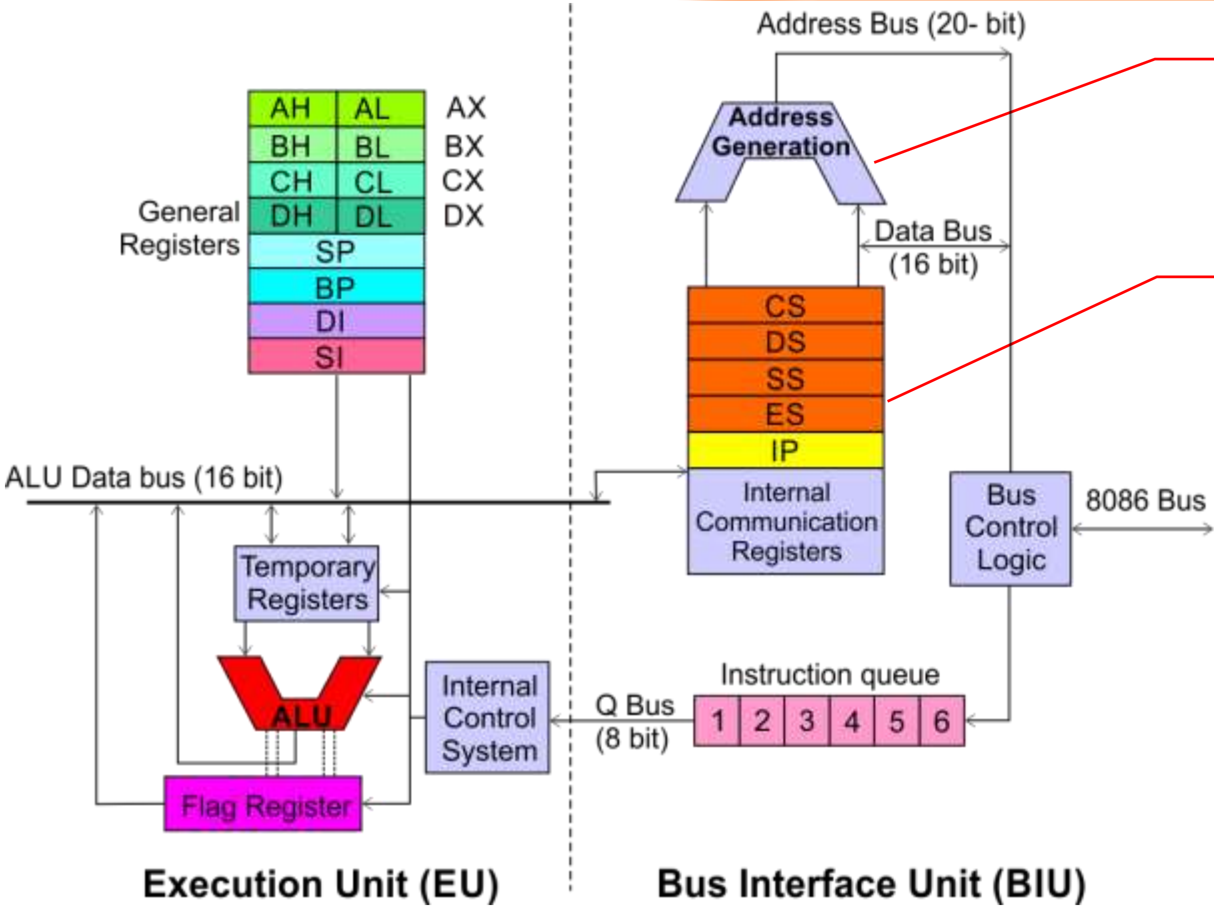
BIU and EU functions separately.

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)

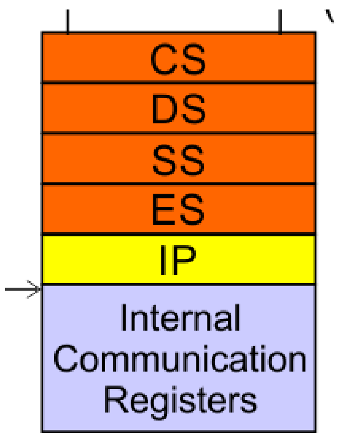


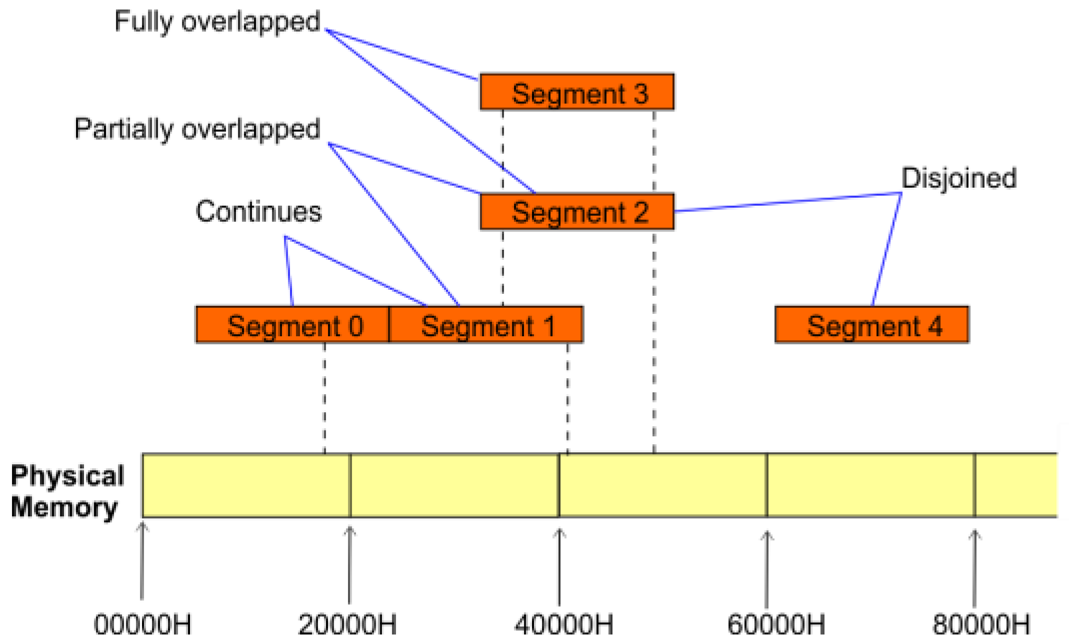
Dedicated Adder to generate 20 bit address

Four 16-bit segment registers

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

Segment Registers



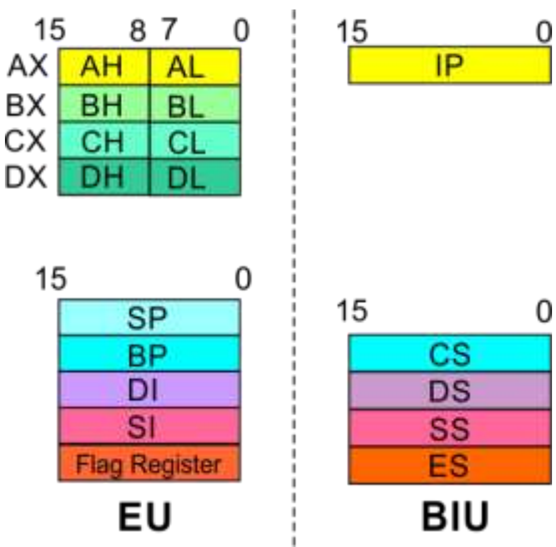


- **8086's 1-megabyte memory is divided into segments of up to 64K bytes each.**
- **The 8086 can directly address four segments (256 K bytes within the 1 M byte of memory) at a particular time.**
- **Programs obtain access to code and data in the segments by changing the segment register content to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

Segment Registers

Code Segment Register

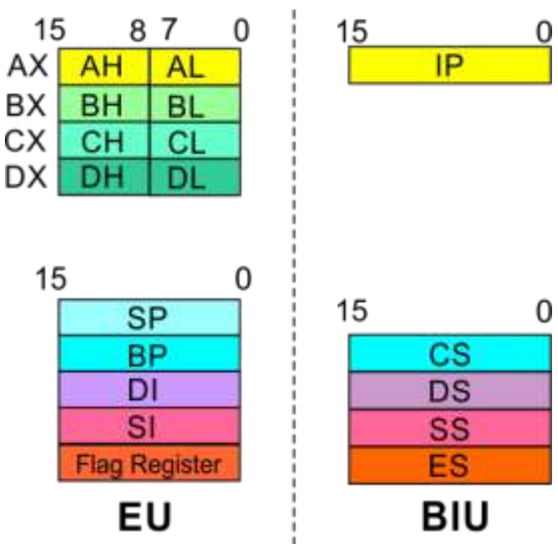
- **16-bit**
- **CS contains the base or start of the current code segment; IP contains the distance or offset from this address to the next instruction byte to be fetched.**
- **BIU computes the 20-bit physical address by logically shifting the contents of CS 4-bits to the left and then adding the 16-bit 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

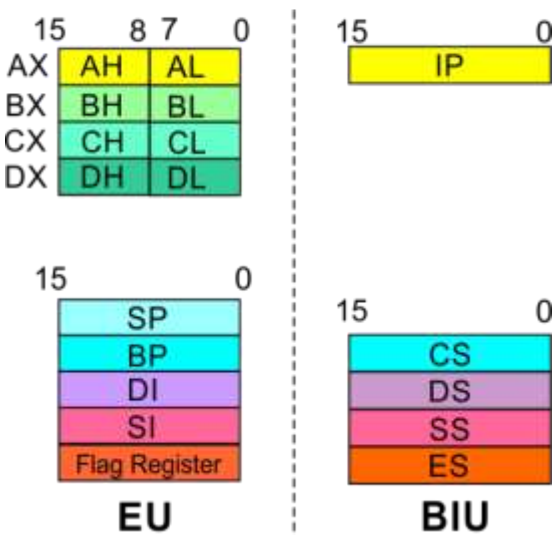
- **16-bit**
- **Points to the current data segment; operands for most instructions are fetched from this segment.**
- **The 16-bit contents of the Source Index (SI) or Destination Index (DI) or a 16-bit displacement are used as offset for computing the 20-bit physical address.**



Segment Registers

Stack Segment Register

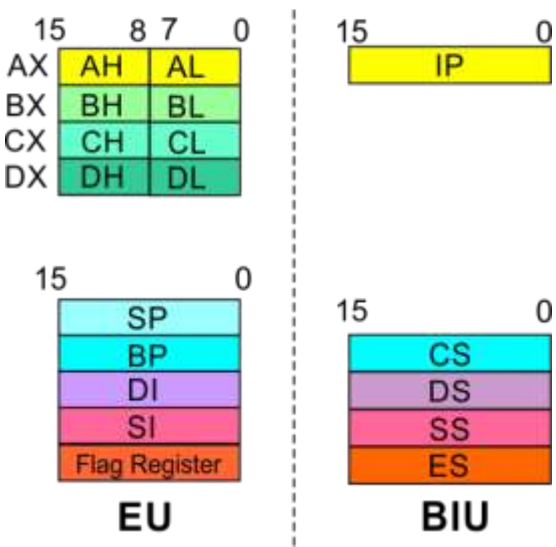
- **16-bit**
- **Points to the current stack.**
- **The 20-bit physical stack address is calculated from the Stack Segment (SS) and the Stack Pointer (SP) for stack instructions such as PUSH and POP.**
- **In based addressing mode, the 20-bit physical stack address is calculated from the Stack segment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



Segment Registers

Extra Segment Register

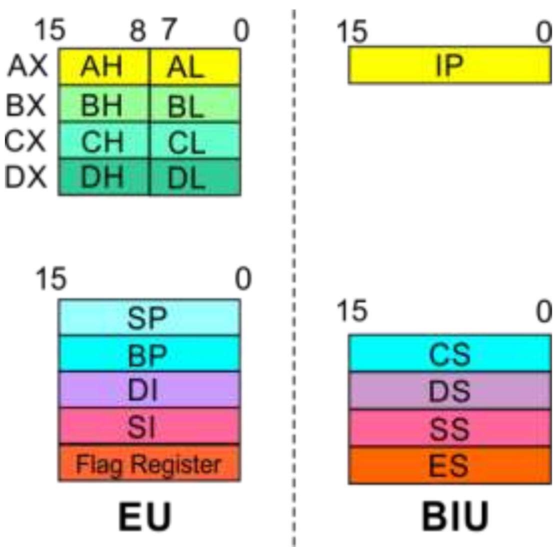
- **16-bit**
- **Points to the extra segment in which data (in excess of 64K pointed to by the DS) is stored.**
- **String instructions use the ES and DI to determine the 20-bit physical address for the destination.**



Segment Registers

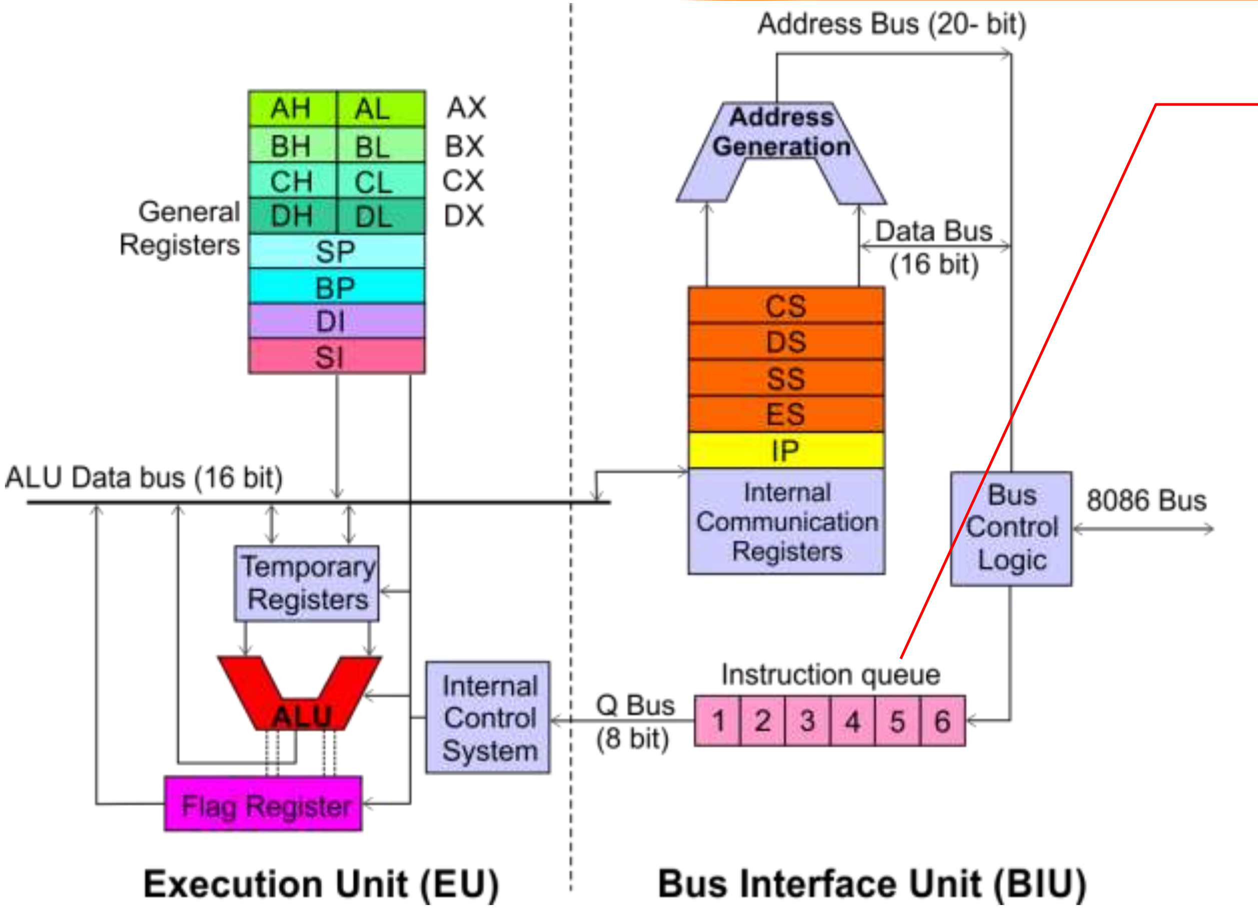
Instruction Pointer

- 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)



- 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 <u>pipelining</u>.

Architecture

Execution Unit (EU)

EU decodes and executes instructions.

A decoder in the EU control system translates instructions.

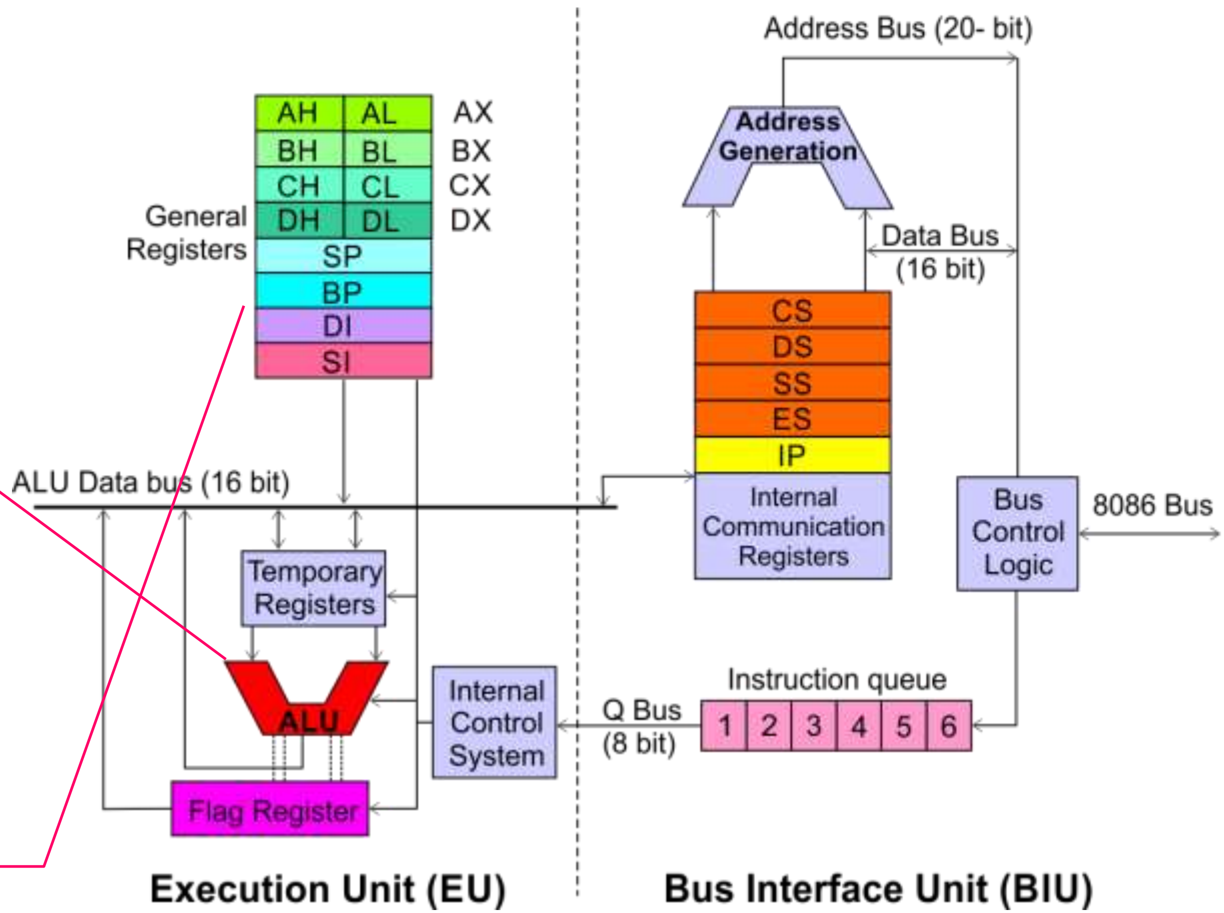
16-bit ALU for performing arithmetic and logic operation

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



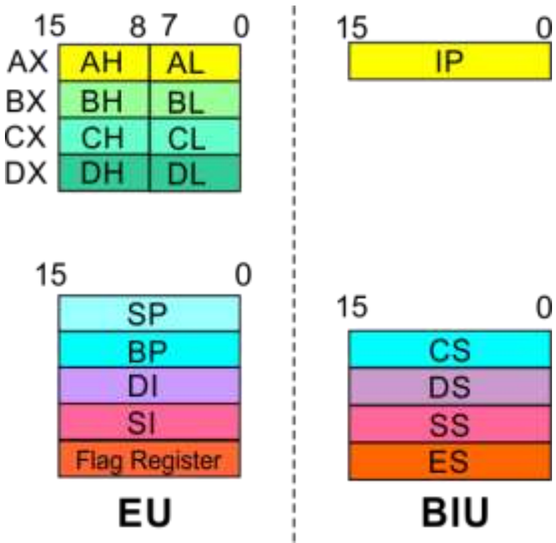
Some of the 16 bit registers can be used as two 8 bit registers as :

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)

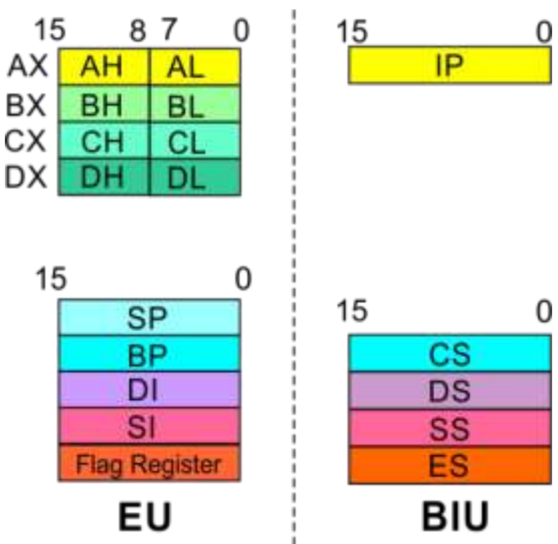
- **Consists of two 8-bit registers AL and AH, which can be combined together and used as a 16-bit register AX.**
- **AL in this case contains the low order byte of the word, and AH contains the high-order byte.**
- **The I/O instructions use the AX or AL for inputting / outputting 16 or 8 bit data to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



EU Registers

Base Register (BX)

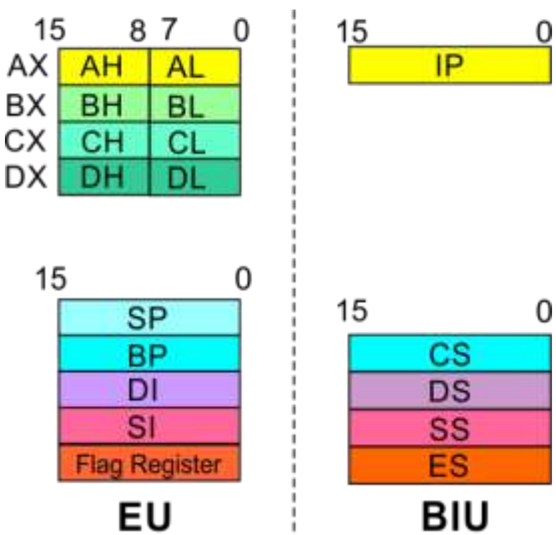
- **Consists of two 8-bit registers BL and BH, which can be combined together and used as a 16-bit register BX.**
- **BL in this case contains the low-order byte of the word, and BH contains the high-order byte.**
- **This is the only general purpose register whose contents can be used for addressing 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:

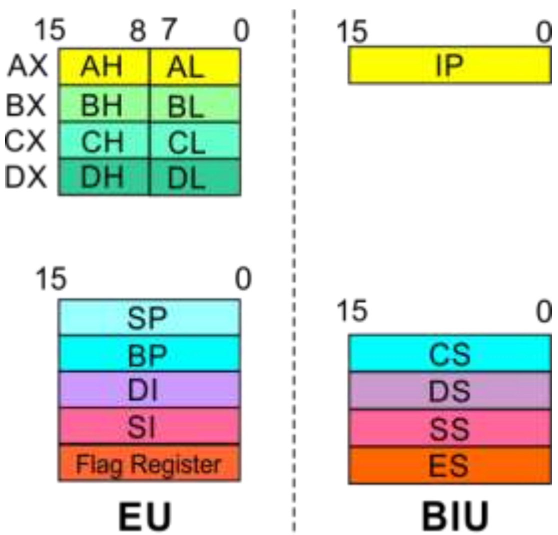
The instruction LOOP START automatically decrements CX by 1 without affecting flags and will check if [CX] = 0.

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

EU Registers

Data Register (DX)

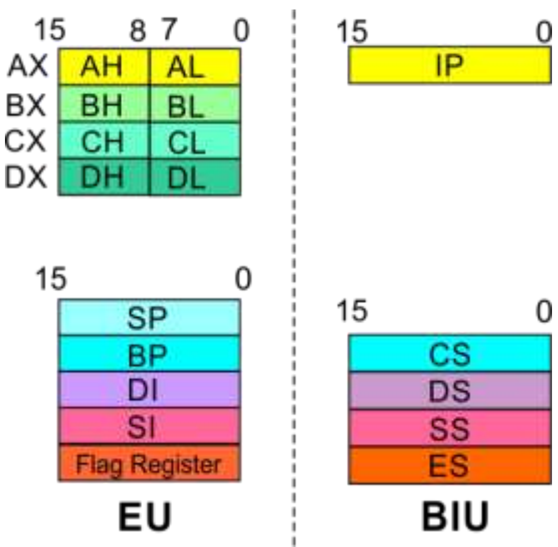
- **Consists of two 8-bit registers DL and DH, which can be combined together and used as a 16-bit register DX.**
- **When combined, DL register contains the low order byte of the word, and DH contains the high-order byte.**
- **Used to hold the high 16-bit result (data) in 16 X 16 multiplication or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

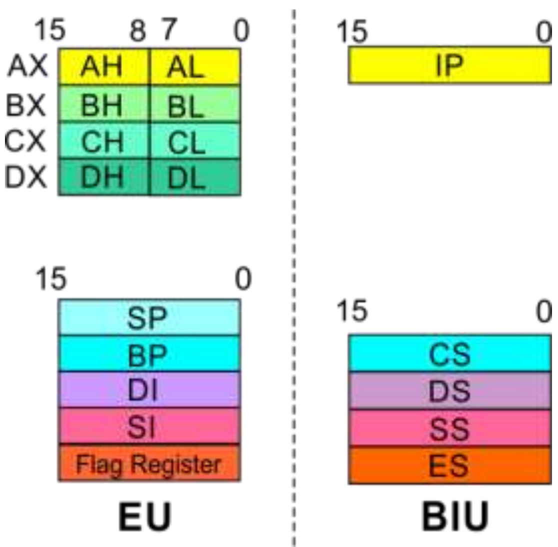
- **SP and BP are used to access data in the stack segment.**
- **SP is used as an offset from the current SS during execution of instructions that involve the stack segment in the external memory.**
- **SP contents are automatically updated (incremented/ decremented) due to execution of a POP or PUSH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

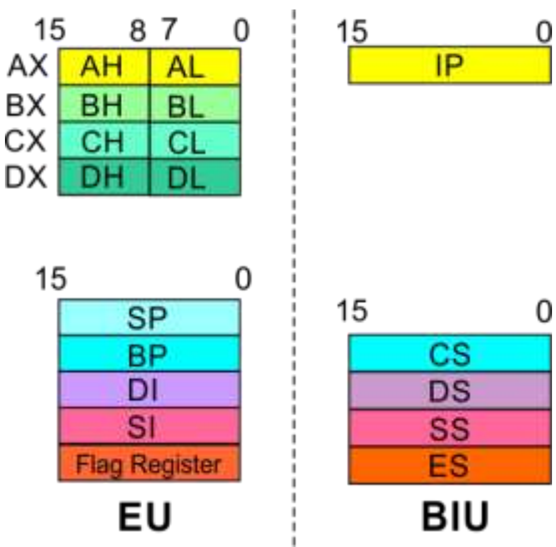
- **Used in indexed addressing.**
- **Instructions that process data strings use the SI and DI registers together with DS and ES respectively in order to distinguish between the source and destination addresses.**



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.

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

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

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.

Direction Flag

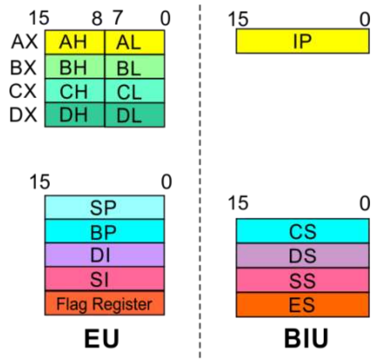
This is used by string manipulation instructions. If this flag bit is '0', the string is processed beginning from the lowest address to the highest address, i.e., auto incrementing mode. Otherwise, the string is processed from the highest address towards the lowest address, i.e., auto incrementing mode.

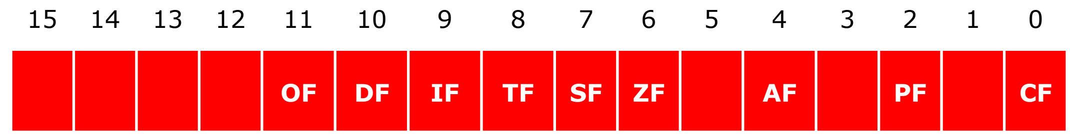
Interrupt Flag

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

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.

- **Computer language**
  - **High Level**
  - **Low Level**
    - **Machine Language**
      - **Binary bits**
    - **Assembly Language**
      - **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

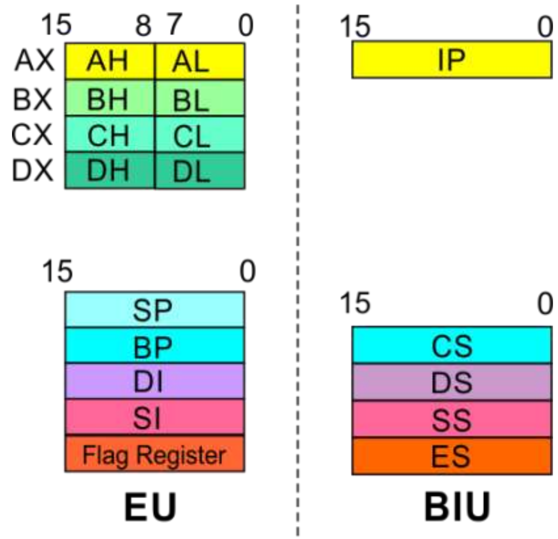
The instruction will specify the name of the register which holds the data to be operated by the instruction.

Example:

MOV CL, DH

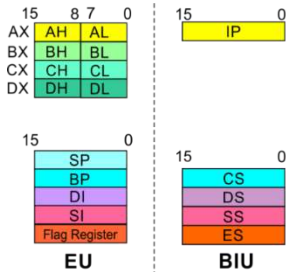
The content of 8-bit register DH is moved to another 8-bit register CL

(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

$(DL) \leftarrow 08_H$

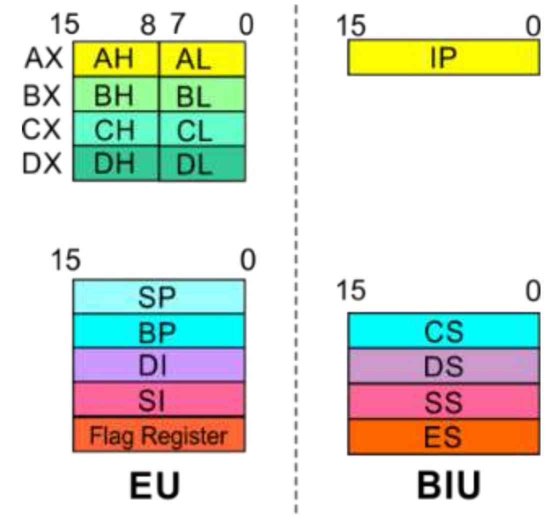
MOV AX, 0A9FH

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

$(AX) \leftarrow 0A9F_H$

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

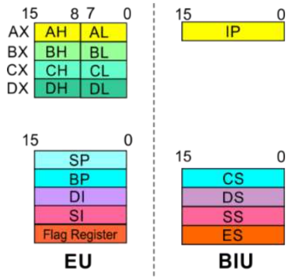


16 bytes of contiguous memory

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:

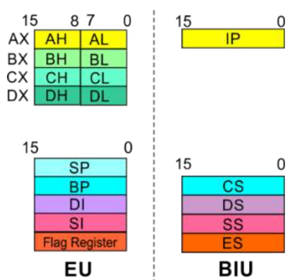
```
MOV  BX, [1354H]
MOV  BL, [0400H]
```

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:

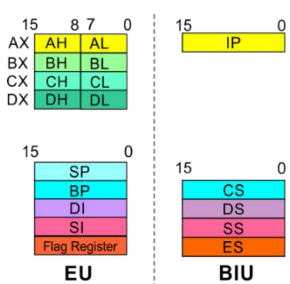
EA = (BX)
BA = (DS) x 16_{10}
MA = BA + EA

(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:

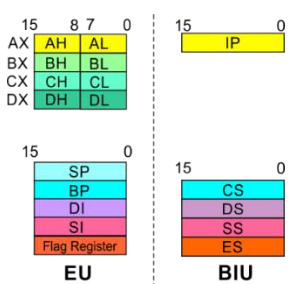
$0008_H \leftarrow 08_H$ (Sign extended)
$EA = (BX) + 0008_H$
$BA = (DS) \times 16_{10}$
$MA = BA + EA$

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

$(AL) \leftarrow (MA)$
$(AH) \leftarrow (MA + 1)$

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



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)

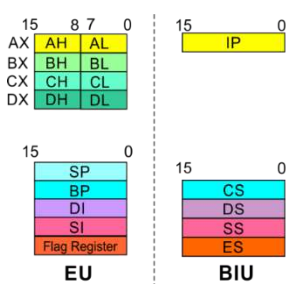
$EA = (SI) + FFA2_H$
$BA = (DS) \times 16_{10}$
$MA = BA + EA$

$(CX) \leftarrow (MA)$ 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

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:

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

Calculation of destination memory location:

$EA_E = (DI)$ $BA_E = (ES) \times 16_{10}$ $MA_E = BA_E + EA_E$

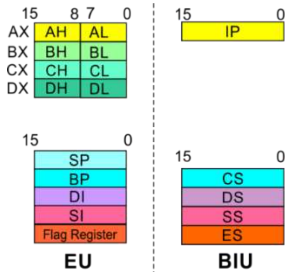
Note : Effective address of the Extra segment register

$(MAE) \leftarrow (MA)$

If DF = 1, then $(SI) \leftarrow (SI) - 1$ and $(DI) = (DI) - 1$

If DF = 0, then $(SI) \leftarrow (SI) + 1$ and $(DI) = (DI) + 1$

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]**

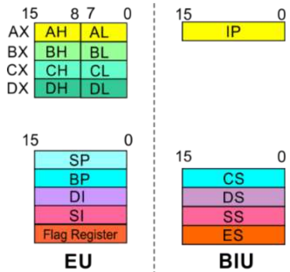
Operations: $PORT_{addr} = 09_H$

$(AL) \leftarrow (PORT)$

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

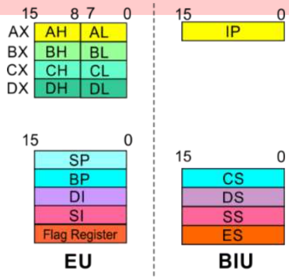
$EA = (IP) + 000A_H$
$BA = (CS) \times 16_{10}$
$MA = BA + EA$

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

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

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

2. Arithmetic Instructions

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

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

2. Arithmetic Instructions

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

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

SBB reg2/ mem, reg1/mem SBB reg2, reg1 SBB reg2, mem SBB mem, reg1	 (reg2) ← (reg1) - (reg2) - CF (reg2) ← (reg2) - (mem)- CF (mem) ← (mem) - (reg1) –CF
SBB reg/mem, data SBB reg, data SBB mem, data	 (reg) ← (reg) – data - CF (mem) ← (mem) - data - CF
SBB A, data SBB AL, data8 SBB AX, data16	 (AL) ← (AL) - data8 - CF (AX) ← (AX) - data16 - CF

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**

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)**

2. Arithmetic Instructions

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

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

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**

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**

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**

3. Logical Instructions

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

AND A, data AND AL, data8 AND AX, data16	$(AL) \leftarrow (AL)$ & data8 $(AX) \leftarrow (AX)$ & data16

AND reg/mem, data AND reg, data AND mem, data	(reg) $\leftarrow$ (reg) & data (mem) $\leftarrow$ (mem) & data

3. Logical Instructions

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

OR reg2/mem, reg1/mem OR reg2, reg1 OR reg2, mem OR mem, reg1	 (reg2) ← (reg2) | (reg1) (reg2) ← (reg2) | (mem) (mem) ← (mem) | (reg1)
OR reg/mem, data OR reg, data OR mem, data	 (reg) ← (reg) | data (mem) ← (mem) | data
OR A, data OR AL, data8 OR AX, data16	 (AL) ← (AL) | data8 (AX) ← (AX) | data16

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

3. Logical Instructions

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

TEST reg2/mem, reg1/mem	
TEST reg2, reg1	Modify flags ← (reg2) & (reg1)
TEST reg2, mem	Modify flags ← (reg2) & (mem)
TEST mem, reg1	Modify flags ← (mem) & (reg1)

TEST reg/mem, data	
TEST reg, data	Modify flags ← (reg) & data
TEST mem, data	Modify flags ← (mem) & data

TEST A, data	
TEST AL, data8	Modify flags ← (AL) & data8
TEST AX, data16	Modify flags ← (AX) & data16

3. Logical Instructions

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

SHR reg/mem

SHR reg

i) SHR reg, 1

ii) SHR reg, CL

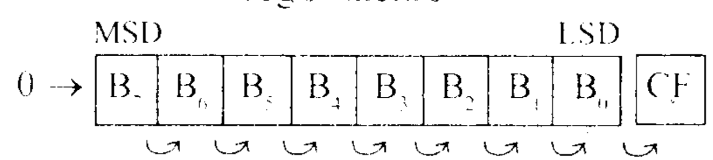
SHR mem

i) SHR mem, 1

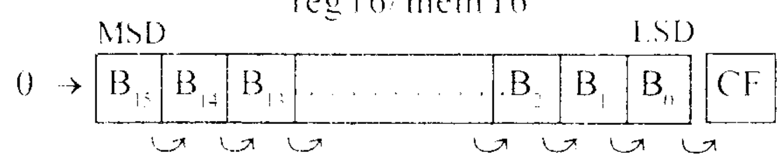
ii) SHR mem, CL

$CF \leftarrow B_{LSD}$; $B_n \leftarrow B_{n+1}$; $B_{MSD} \leftarrow 0$

reg 8 / mem 8



reg 16 / mem 16



3. Logical Instructions

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

SHL reg/mem or SAL reg/mem

SHL reg or SAL reg

i) SHL reg, 1 or SAL reg, 1

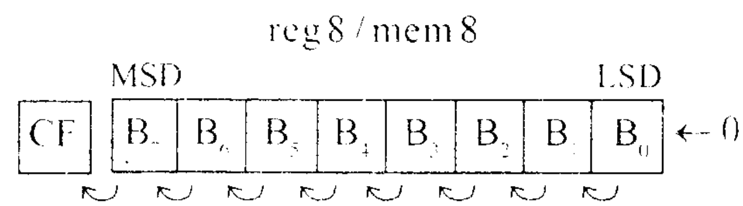
ii) SHL reg, CL or SAL reg, CL

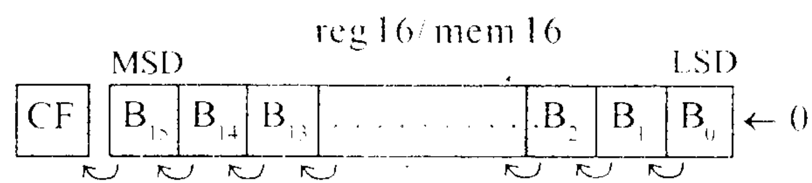
SHL mem or SAL mem

i) SHL mem, 1 or SAL mem, 1

ii) SHL mem, CL or SAL mem, CL

$CF \leftarrow B_{MSD}$; $B_{n+1} \leftarrow B_n$; $B_{LSD} \leftarrow 0$





3. Logical Instructions

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

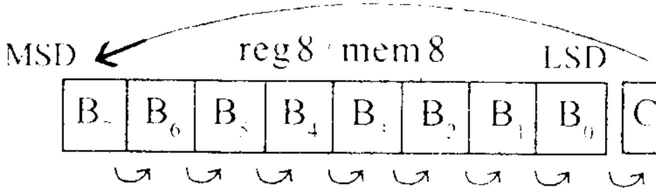
RCR reg/mem

$B_n \leftarrow B_{n-1}$; $B_{MSD} \leftarrow CF$; $CF \leftarrow B_{LSD}$

RCR reg

i) RCR reg, 1

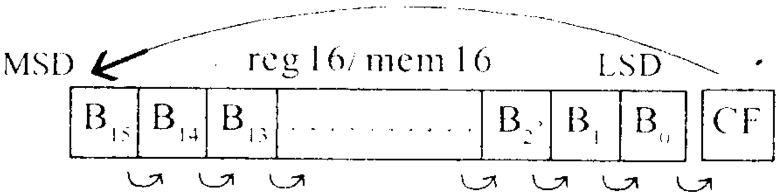
ii) RCR reg, CL



RCR mem

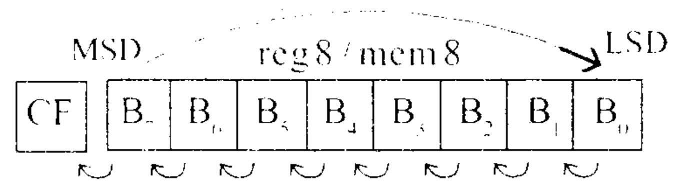
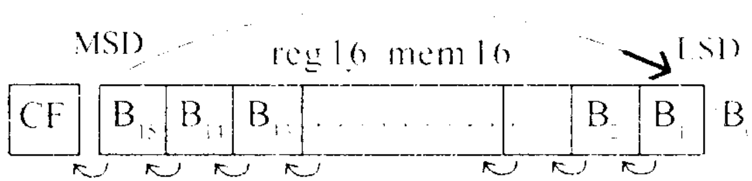
i) RCR mem, 1

ii) RCR mem, CL



3. Logical Instructions

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

ROL reg/mem	$B_{n+1} \leftarrow B_n ; CF \leftarrow B_{MSD} ; B_{LSD} \leftarrow B_{MSD}$
ROL reg i) ROL reg, 1 ii) ROL reg, CL	
ROL mem i) ROL mem, 1 ii) ROL mem, CL	

Instruction Set

4. String Manipulation Instructions

- String : Sequence of bytes or words
- 8086 instruction set includes instruction for string movement, comparison, scan, load and store.
- REP instruction prefix : used to repeat execution of string instructions
- String instructions end with S or SB or SW.
  S represents string, SB string byte and SW string word.
- Offset or effective address of the source operand is stored in SI register and that of the destination operand is stored in DI register.
- Depending on the status of DF, SI and DI registers are automatically updated.
- DF = 0 ⇒ SI and DI are incremented by 1 for byte and 2 for word.
- DF = 1 ⇒ SI and DI are decremented by 1 for byte and 2 for word.

4. String Manipulation Instructions

Mnemonics: **REP, MOVS, CMPS, SCAS, LODS, STOS**

REP	
REPZ/ REPE **(Repeat CMPS or SCAS until ZF = 0)**	**While CX ≠ 0 and ZF = 1, repeat execution of string instruction and (CX) ← (CX) – 1**
REPNZ/ REPNE **(Repeat CMPS or SCAS until ZF = 1)**	**While CX ≠ 0 and ZF = 0, repeat execution of string instruction and (CX) ← (CX) - 1**

4. String Manipulation Instructions

Mnemonics: **REP, MOVS, CMPS, SCAS, LODS, STOS**

MOVS	
MOVSB	$MA = (DS) \times 16_{10} + (SI)$ $MA_E = (ES) \times 16_{10} + (DI)$ $(MA_E) \leftarrow (MA)$ If $DF = 0$, then $(DI) \leftarrow (DI) + 1$; $(SI) \leftarrow (SI) + 1$ If $DF = 1$, then $(DI) \leftarrow (DI) - 1$; $(SI) \leftarrow (SI) - 1$
MOVSW	$MA = (DS) \times 16_{10} + (SI)$ $MA_E = (ES) \times 16_{10} + (DI)$ $(MA_E ; MA_E + 1) \leftarrow (MA; MA + 1)$ If $DF = 0$, then $(DI) \leftarrow (DI) + 2$; $(SI) \leftarrow (SI) + 2$ If $DF = 1$, then $(DI) \leftarrow (DI) - 2$; $(SI) \leftarrow (SI) - 2$

4. String Manipulation Instructions

Mnemonics: REP, MOVS, CMPS, SCAS, LODS, STOS

Compare two string byte or string word

CMPS	

CMPSB	$MA = (DS) \times 16_{10} + (SI)$
	$MA_E = (ES) \times 16_{10} + (DI)$
	
	Modify flags $\leftarrow$ (MA) - (MA_E)
	
	If (MA) > (MA_E), then CF = 0; ZF = 0; SF = 0
	If (MA) < (MA_E), then CF = 1; ZF = 0; SF = 1

CMPSW	If (MA) = (MA_E), then CF = 0; ZF = 1; SF = 0 **For byte operation** If DF = 0, then (DI) $\leftarrow$ (DI) + 1; (SI) $\leftarrow$ (SI) + 1 If DF = 1, then (DI) $\leftarrow$ (DI) - 1; (SI) $\leftarrow$ (SI) - 1 **For word operation** If DF = 0, then (DI) $\leftarrow$ (DI) + 2; (SI) $\leftarrow$ (SI) + 2 If DF = 1, then (DI) $\leftarrow$ (DI) - 2; (SI) $\leftarrow$ (SI) - 2

4. String Manipulation Instructions

Mnemonics: REP, MOVS, CMPS, SCAS, LODS, STOS

Scan (compare) a string byte or word with accumulator

SCAS	

SCASB	$MA_E = (ES) \times 16_{10} + (DI)$
	Modify flags $\leftarrow$ (AL) - (MA_E)
	
	If (AL) > (MA_E), then CF = 0; ZF = 0; SF = 0
	If (AL) < (MA_E), then CF = 1; ZF = 0; SF = 1
	If (AL) = (MA_E), then CF = 0; ZF = 1; SF = 0
	
	If DF = 0, then (DI) $\leftarrow$ (DI) + 1
	If DF = 1, then (DI) $\leftarrow$ (DI) – 1

SCASW	$MA_E = (ES) \times 16_{10} + (DI)$
	Modify flags $\leftarrow$ (AL) - (MA_E)
	
	If (AX) > (MA_E ; MA_E + 1), then CF = 0; ZF = 0; SF = 0
	If (AX) < (MA_E ; MA_E + 1), then CF = 1; ZF = 0; SF = 1
	If (AX) = (MA_E ; MA_E + 1), then CF = 0; ZF = 1; SF = 0
	
	If DF = 0, then (DI) $\leftarrow$ (DI) + 2
	If DF = 1, then (DI) $\leftarrow$ (DI) – 2

4. String Manipulation Instructions

Mnemonics: REP, MOVS, CMPS, SCAS, LODS, STOS

Load string byte in to AL or string word in to AX

LODS	
LODSB	**MA = (DS) x 16_{10} + (SI)** **(AL) ← (MA)** **If DF = 0, then (SI) ← (SI) + 1** **If DF = 1, then (SI) ← (SI) – 1**
LODSW	**MA = (DS) x 16_{10} + (SI)** **(AX) ← (MA ; MA + 1)** **If DF = 0, then (SI) ← (SI) + 2** **If DF = 1, then (SI) ← (SI) – 2**

4. String Manipulation Instructions

Mnemonics: **REP, MOVS, CMPS, SCAS, LODS, STOS**

Store byte from AL or word from AX in to string

STOS	
STOSB	$MA_E = (ES) \times 16_{10} + (DI)$ $(MA_E) \leftarrow (AL)$ If DF = 0, then $(DI) \leftarrow (DI) + 1$ If DF = 1, then $(DI) \leftarrow (DI) - 1$
STOSW	$MA_E = (ES) \times 16_{10} + (DI)$ $(MA_E ; MA_E + 1) \leftarrow (AX)$ If DF = 0, then $(DI) \leftarrow (DI) + 2$ If DF = 1, then $(DI) \leftarrow (DI) - 2$

Instruction Set

5. Processor Control Instructions

Mnemonics	Explanation
STC	Set CF ← 1
CLC	Clear CF ← 0
CMC	Complement carry CF ← CF/
STD	Set direction flag DF ← 1
CLD	Clear direction flag DF ← 0
STI	Set interrupt enable flag IF ← 1
CLI	Clear interrupt enable flag IF ← 0
NOP	No operation
HLT	Halt after interrupt is set
WAIT	Wait for TEST pin active
ESC opcode mem/ reg	Used to pass instruction to a coprocessor which shares the address and data bus with the 8086
LOCK	Lock bus during next instruction

6. Control Transfer Instructions

- Transfer the control to a specific destination or target instruction
- Do not affect flags

- 8086 Unconditional transfers

Mnemonics	Explanation
CALL reg/ mem/ disp16	Call subroutine
RET	Return from subroutine
JMP reg/ mem/ disp8/ disp16	Unconditional jump

6. Control Transfer Instructions

- 8086 signed conditional branch instructions
- 8086 unsigned conditional branch instructions

- Checks flags
- If conditions are true, the program control is transferred to the new memory location in the same segment by modifying the content of IP

6. Control Transfer Instructions

- **8086 signed conditional branch instructions**

Name	Alternate name
JE disp8 Jump if equal	**JZ disp8** Jump if result is 0
JNE disp8 Jump if not equal	**JNZ disp8** Jump if not zero
JG disp8 Jump if greater	**JNLE disp8** Jump if not less or equal
JGE disp8 Jump if greater than or equal	**JNL disp8** Jump if not less
JL disp8 Jump if less than	**JNGE disp8** Jump if not greater than or equal
JLE disp8 Jump if less than or equal	**JNG disp8** Jump if not greater

- **8086 unsigned conditional branch instructions**

Name	Alternate name
JE disp8 Jump if equal	**JZ disp8** Jump if result is 0
JNE disp8 Jump if not equal	**JNZ disp8** Jump if not zero
JA disp8 Jump if above	**JNBE disp8** Jump if not below or equal
JAE disp8 Jump if above or equal	**JNB disp8** Jump if not below
JB disp8 Jump if below	**JNAE disp8** Jump if not above or equal
JBE disp8 Jump if below or equal	**JNA disp8** Jump if not above

6. Control Transfer Instructions

- 8086 conditional branch instructions affecting individual flags

Mnemonics	Explanation
JC disp8	Jump if CF = 1
JNC disp8	Jump if CF = 0
JP disp8	Jump if PF = 1
JNP disp8	Jump if PF = 0
JO disp8	Jump if OF = 1
JNO disp8	Jump if OF = 0
JS disp8	Jump if SF = 1
JNS disp8	Jump if SF = 0
JZ disp8	Jump if result is zero, i.e, Z = 1
JNZ disp8	Jump if result is not zero, i.e, Z = 1

ASSEMBLER DIRECTIVES

Assemble Directives

- Instructions to the Assembler regarding the program being executed.

- Control the generation of machine codes and organization of the program; but no machine codes are generated for assembler directives.

- Also called 'pseudo instructions'

- Used to :
  - specify the start and end of a program
  - attach value to variables
  - allocate storage locations to input/ output data
  - define start and end of segments, procedures, macros etc..

Assemble Directives

- **DB**
- DW
- SEGMENT
- ENDS
- ASSUME
- ORG
- END
- EVEN
- EQU
- PROC
- FAR
- NEAR
- ENDP
- SHORT
- MACRO
- ENDM

- **Define Byte**
- **Define a byte type (8-bit) variable**
- **Reserves specific amount of memory locations to each variable**
- **Range : 00_H – FF_H for unsigned value;**
  00_H – $7F_H$ for positive value and
  80_H – FF_H for negative value
- **General form : variable DB value/ values**

Example:

LIST DB 7FH, 42H, 35H

Three consecutive memory locations are reserved for the variable LIST and each data specified in the instruction are stored as initial value in the reserved memory location

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- **Define Word**
- **Define a word type (16-bit) variable**
- **Reserves two consecutive memory locations to each variable**
- **Range : 0000_H – $FFFF_H$ for unsigned value;**
  0000_H – $7FFF_H$ for positive value and
  8000_H – $FFFF_H$ for negative value
- **General form : variable DW value/ values**

Example:

ALIST DW 6512H, 0F251H, 0CDE2H

Six consecutive memory locations are reserved for the variable ALIST and each 16-bit data specified in the instruction is stored in two consecutive memory location.

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- **SEGMENT : Used to indicate the beginning of a code/ data/ stack segment**
- **ENDS : Used to indicate the end of a code/ data/ stack segment**
- **General form:**

```
Segnam SEGMENT

   …
   …
   …
   …
   …
   …

Segnam ENDS
```

Program code
or
Data Defining Statements

User defined name of the segment

Assemble Directives

- DB
- DW
- SEGMENT
- ENDS
- **ASSUME**
- ORG
- END
- EVEN
- EQU
- PROC
- FAR
- NEAR
- ENDP
- SHORT
- MACRO
- ENDM

- **Informs the assembler the name of the program/ data segment that should be used for a specific segment.**
- **General form:**

ASSUME segreg : segnam, .. , segreg : segnam

Segment Register

User defined name of the segment

Example:

ASSUME CS: ACODE, DS:ADATA	Tells the compiler that the instructions of the program are stored in the segment ACODE and data are stored in the segment ADATA

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- **ORG** (Origin) is used to assign the starting address (Effective address) for a program/ data segment
- **END** is used to terminate a program; statements after END will be ignored
- **EVEN** : Informs the assembler to store program/ data segment starting from an even address
- **EQU** (Equate) is used to attach a value to a variable

Examples:

ORG 1000H	Informs the assembler that the statements following ORG 1000H should be stored in memory starting with effective address 1000_H
LOOP EQU 10FEH	Value of variable LOOP is $10FE_H$
_SDATA SEGMENT ORG 1200H A DB 4CH EVEN B DW 1052H _SDATA ENDS	In this data segment, effective address of memory location assigned to A will be 1200_H and that of B will be 1202_H and 1203_H.

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
ENDP
FAR
NEAR

SHORT

MACRO
ENDM

- **PROC** Indicates the beginning of a procedure
- **ENDP** End of procedure
- **FAR** Intersegment call
- **NEAR** Intrasegment call
- General form

```
procname PROC[NEAR/ FAR]

            ...
            ...
            ...

            RET

procname ENDP
```

Program statements of the procedure

Last statement of the procedure

User defined name of the procedure

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
ENDP
FAR
NEAR

SHORT

MACRO
ENDM

Examples:

ADD64 PROC NEAR … … … RET ADD64 ENDP	The subroutine/ procedure named ADD64 is declared as NEAR and so the assembler will code the CALL and RET instructions involved in this procedure as near call and return
CONVERT PROC FAR … … … RET CONVERT ENDP	The subroutine/ procedure named CONVERT is declared as FAR and so the assembler will code the CALL and RET instructions involved in this procedure as far call and return

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
ENDP
FAR
NEAR

SHORT

MACRO
ENDM

- **Reserves one memory location for 8-bit signed displacement in jump instructions**

Example:

JMP SHORT AHEAD	The directive will reserve one memory location for 8-bit displacement named AHEAD

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
ENDP
FAR
NEAR

SHORT

MACRO
ENDM

- **MACRO** Indicate the beginning of a macro
- **ENDM** End of a macro
- General form:

```
macroname MACRO[Arg1, Arg2 ...]
          ...
          ...
          ...
macroname ENDM
```

Program statements in the macro

User defined name of the macro

Unit II

8086 Microprocessor (19 Hrs)

The Intel 8086 - architecture - MN/MX modes - 8086 addressing modes - instruction set- instruction format - assembler directives and operators - Programming with 8086 - interfacing memory and I/O ports - Comparison of 8086 and 8088 - Coprocessors - Intel 8087 - Familiarisation with Debug utility.

INTERFACING MEMORY AND I/O PORTS

Memory

Memory

Store Programs and Data

Processor Memory

- Registers inside a microcomputer
- Store data and results temporarily
- No speed disparity
- Cost ↑

Primary or Main Memory

- Storage area which can be directly accessed by microprocessor
- Store programs and data prior to execution
- Should not have speed disparity with processor ⇒ Semi Conductor memories using CMOS technology
- ROM, EPROM, Static RAM, DRAM

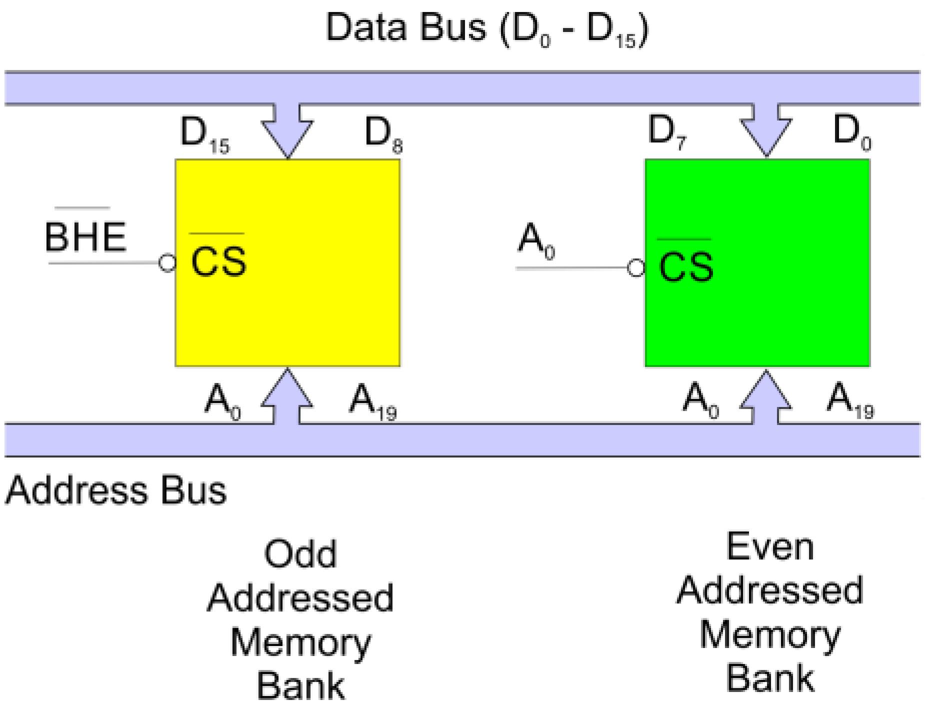
Secondary Memory

- Storage media comprising of slow devices such as magnetic tapes and disks
- Hold large data files and programs: Operating system, compilers, databases, permanent programs etc.

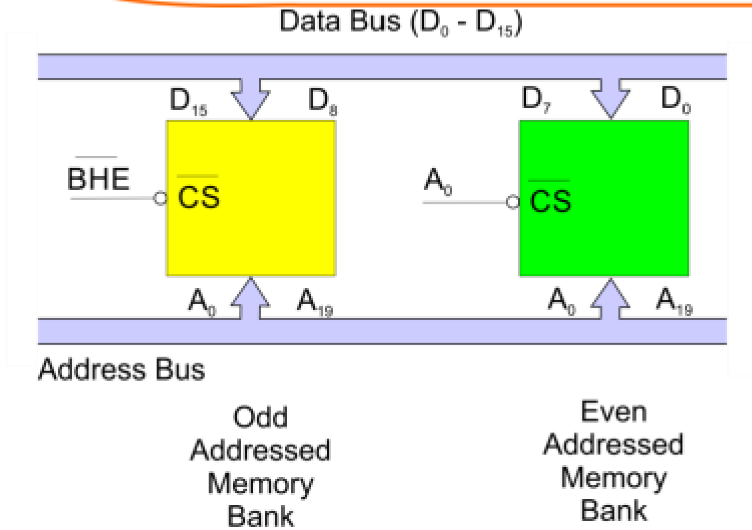
Memory organization in 8086

- Memory IC's : Byte oriented
- 8086 : 16-bit
- Word : Stored by two consecutive memory locations; for LSB and MSB
- Address of word : Address of LSB
- Bank 0 : $A_0 = 0$ $\Rightarrow$ Even addressed memory bank

  Bank 1 : $\overline{BHE} = 0$ $\Rightarrow$ Odd addressed memory bank



Memory organization in 8086



	Operation	$\overline{BHE}$	A_0	Data Lines Used
1	Read/ Write byte at an even address	1	0	D_7 – D_0
2	Read/ Write byte at an odd address	0	1	D_{15} – D_8
3	Read/ Write word at an even address	0	0	D_{15} – D_0
4	Read/ Write word at an odd address	0	1	D_{15} – D_0 in first operation byte from odd bank is transferred
		1	0	D_7 – D_0 in first operation byte from odd bank is transferred

Memory organization in 8086

- **Available memory space = EPROM + RAM**
- **Allot equal address space in odd and even bank for both EPROM and RAM**
- **Can be implemented in two IC's (one for even and other for odd) or in multiple IC's**

Interfacing SRAM and EPROM

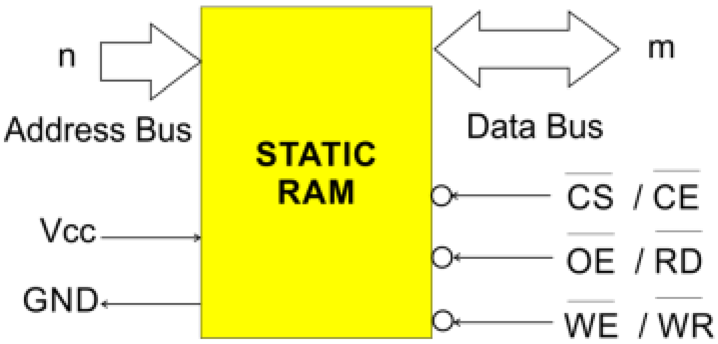
- Memory interface ⇒ Read from and write in to a set of semiconductor memory IC chip
- EPROM ⇒ Read operations
- RAM ⇒ Read and Write

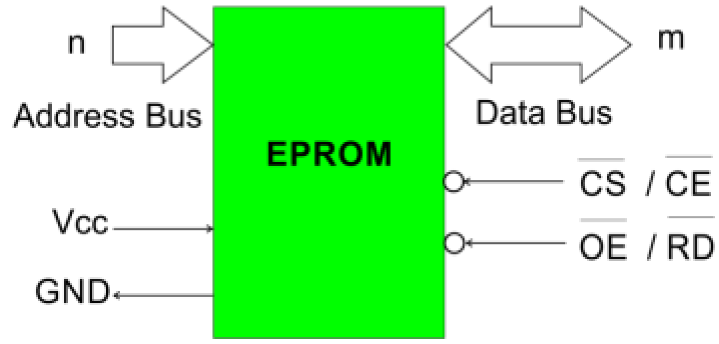
In order to perform read/ write operations,

- Memory access time < read / write time of the processor
- Chip Select (CS) signal has to be generated
- Control signals for read / write operations
- Allot address for each memory location

Interfacing SRAM and EPROM

Typical Semiconductor IC Chip





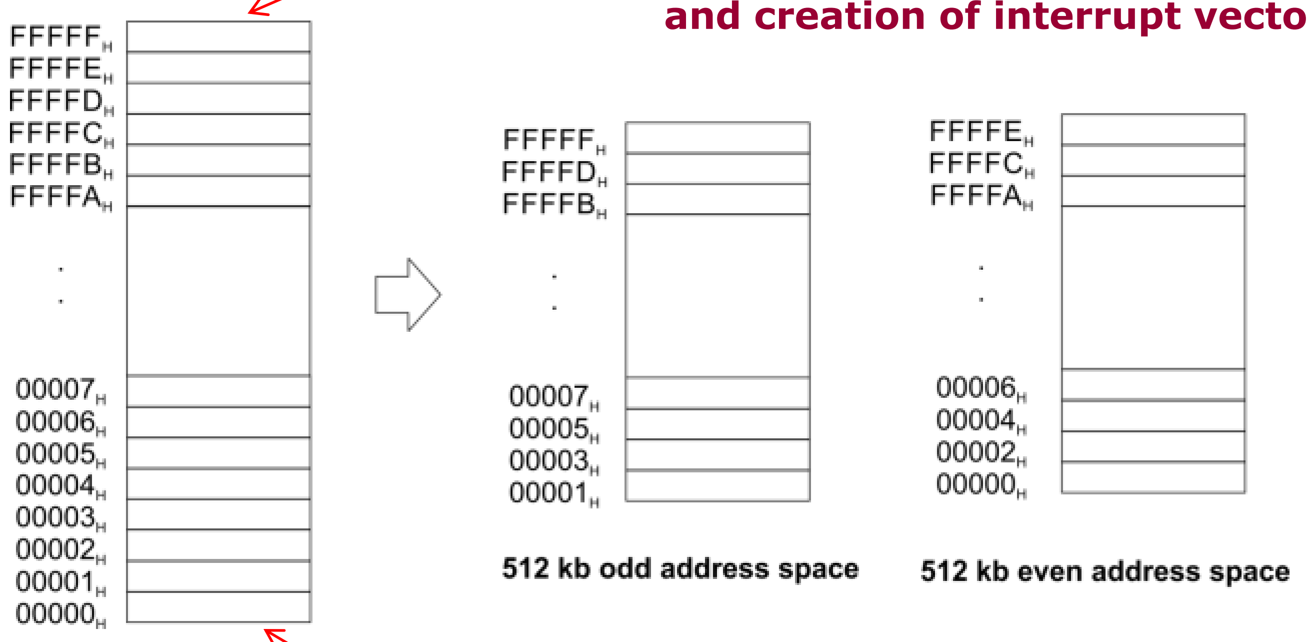
No of Address pins	Memory capacity			Range of address in hexa
	In Decimal	In kilo	In hexa	
20	2^{20}= 10,48,576	1024 k = 1M	100000	00000 to FFFFF

Interfacing SRAM and EPROM

- **Memory map of 8086**

EPROM's are mapped at $FFFFF_H$

⇒ **Facilitate automatic execution of monitor programs and creation of interrupt vector table**



RAM are mapped at the beginning; 00000H is allotted to RAM

Interfacing SRAM and EPROM

Monitor Programs

⇒ **Programing 8279 for keyboard scanning and display refreshing**

⇒ **Programming peripheral IC's 8259, 8257, 8255, 8251, 8254 etc**

⇒ **Initialization of stack**

⇒ **Display a message on display (output)**

⇒ **Initializing interrupt vector table**

Note :

8279	**Programmable keyboard/ display controller**
8257	**DMA controller**
8259	**Programmable interrupt controller**
8255	**Programmable peripheral interface**

Interfacing I/O and peripheral devices

I/O devices

⇒ For communication between microprocessor and outside world

⇒ Keyboards, CRT displays, Printers, Compact Discs etc.

⇒ Microprocessor ⟺ Ports / Buffer IC's (interface circuitry) ⟺ I/ O devices

⇒ Data transfer types

- **Programmed I/ O**
  - Data transfer is accomplished through an I/O port controlled by software
  - Memory mapped
  - I/O mapped
- **Interrupt driven I/ O**
  - I/O device interrupts the processor and initiate data transfer
- **Direct memory access**
  - Data transfer is achieved by bypassing the microprocessor

8086 and 8088 comparison

Memory mapping	I/O mapping
20 bit address are provided for I/O devices	8-bit or 16-bit addresses are provided for I/O devices
The I/O ports or peripherals can be treated like memory locations and so all instructions related to memory can be used for data transmission between I/O device and processor	Only IN and OUT instructions can be used for data transfer between I/O device and processor
Data can be moved from any register to ports and vice versa	Data transfer takes place only between accumulator and ports
When memory mapping is used for I/O devices, full memory address space cannot be used for addressing memory. ⇒ Useful only for small systems where memory requirement is less	Full memory space can be used for addressing memory. ⇒ Suitable for systems which require large memory capacity
For accessing the memory mapped devices, the processor executes memory read or write cycle. ⇒ $M / \overline{IO}$ is asserted high	For accessing the I/O mapped devices, the processor executes I/O read or write cycle. ⇒ $M / \overline{IO}$ is asserted low

8086 AND 8088 COMPARISON

8086 and 8088 comparison

8086	8088
Similar EU and Instruction set ; dissimilar BIU	
16-bit Data bus lines obtained by demultiplexing $AD_0 - AD_{15}$	8-bit Data bus lines obtained by demultiplexing $AD_0 - AD_7$
20-bit address bus	8-bit address bus
Two banks of memory each of 512 kb	Single memory bank
6-bit instruction queue	4-bit instruction queue
Clock speeds: 5 / 8 / 10 MHz	5 / 8 MHz
In MIN mode, pin 28 is assigned the signal $M / \overline{IO}$	In MIN mode, pin 28 is assigned the signal $IO / \overline{M}$
To access higher byte, $\overline{BHE}$ signal is used	No such signal required, since the data width is only 1-byte

8087 Coprocessor

Co-processor – Intel 8087

Multiprocessor system

- **A microprocessor system comprising of two or more processors**
- **Distributed processing: Entire task is divided in to subtasks**

Advantages

- **Better system throughput by having more than one processor**
- **Each processor have a local bus to access local memory or I/O devices so that a greater degree of parallel processing can be achieved**
- **System structure is more flexible.**
  One can easily add or remove modules to change the system configuration without affecting the other modules in the system

Co-processor – Intel 8087

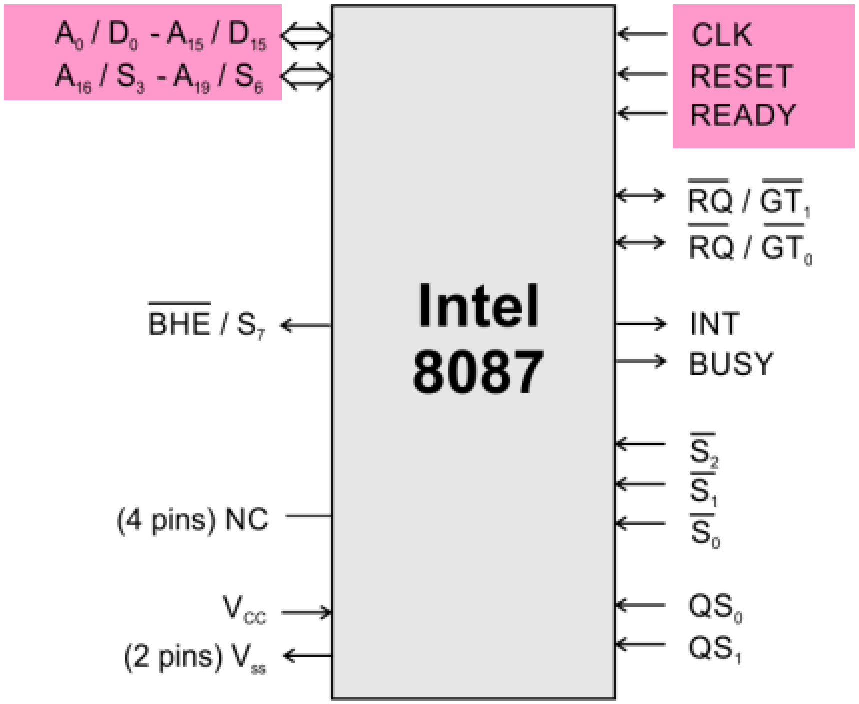
8087 coprocessor

- Specially designed to take care of mathematical calculations involving integer and floating point data
- "Math coprocessor" or "Numeric Data Processor (NDP)"
- Works in parallel with a 8086 in the maximum mode

Features

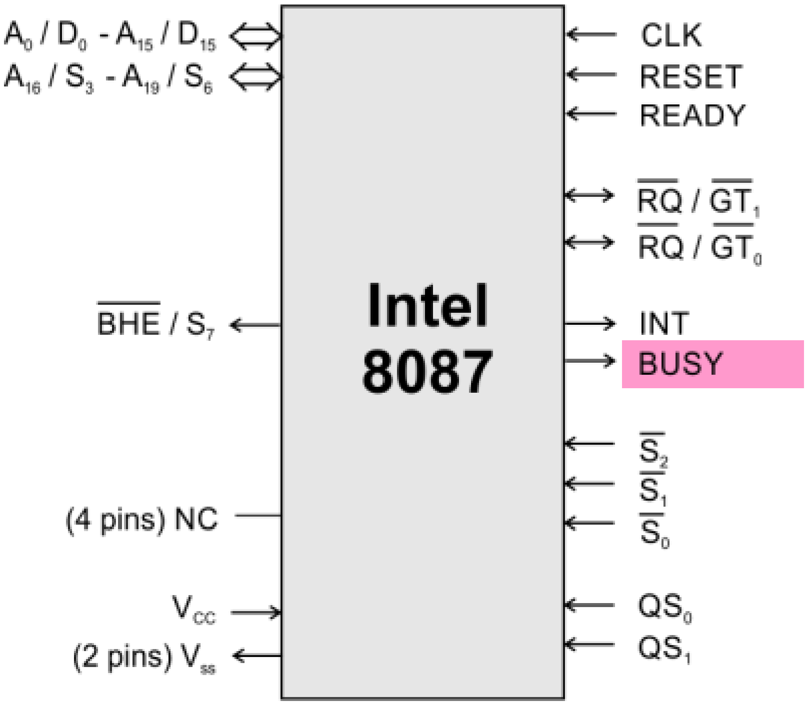
1) Can operate on data of the integer, decimal and real types with lengths ranging from 2 to 10 bytes
2) Instruction set involves square root, exponential, tangent etc. in addition to addition, subtraction, multiplication and division.
3) High performance numeric data processor $\Rightarrow$ it can multiply two 64-bit real numbers in about 27μs and calculate square root in about 36 μs
4) Follows IEEE floating point standard
5) It is multi bus compatible

Co-processor – Intel 8087



- **16 multiplexed address / data pins and 4 multiplexed address / status pins**
- **Hence it can have 16-bit external data bus and 20-bit external address bus like 8086**
- **Processor clock, ready and reset signals are applied as clock, ready and reset signals for coprocessor**

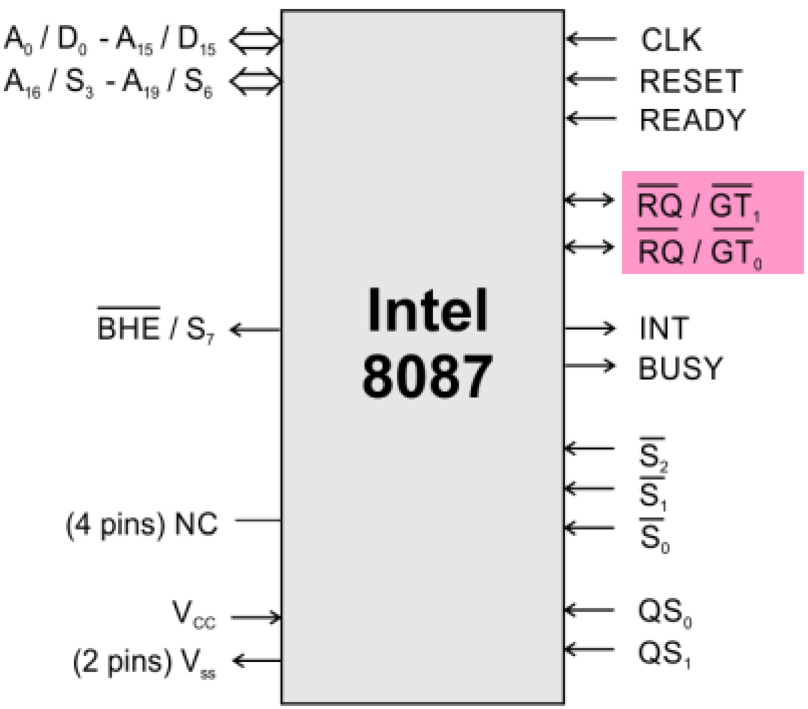
Co-processor – Intel 8087



BUSY

- **BUSY signal from 8087 is connected to the $\overline{TEST}$ input of 8086**
- **If the 8086 needs the result of some computation that the 8087 is doing before it can execute the next instruction in the program, a user can tell 8086 with a WAIT instruction to keep looking at its $\overline{TEST}$ pin until it finds the pin low**
- **A low on the BUSY output indicates that the 8087 has completed the computation**

Co-processor – Intel 8087



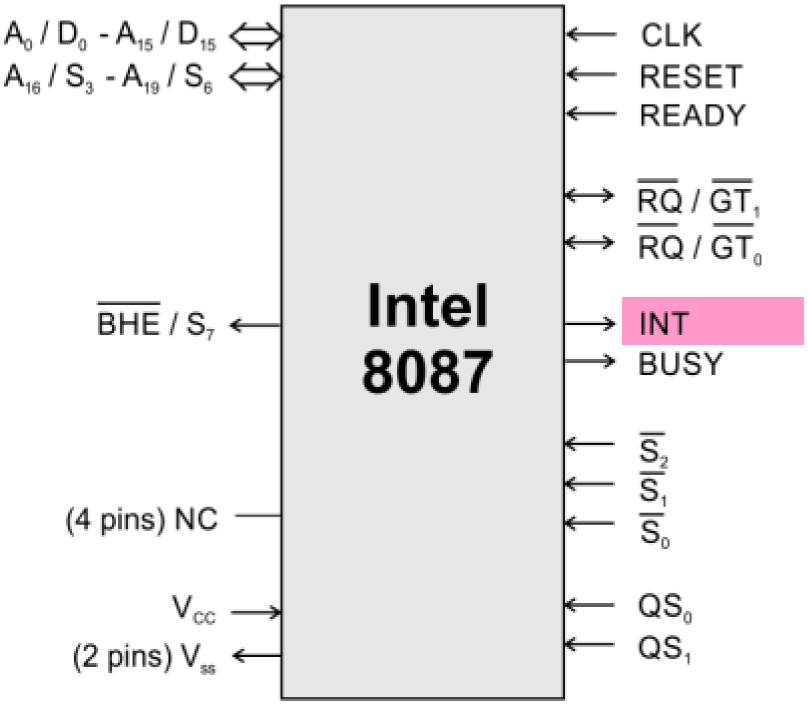
$\overline{RQ} / \overline{GT}_0$

- **The request / grant signal from the 8087 is usually connected to the request / grant ($\overline{RQ} / \overline{GT}_0$ or $\overline{RQ} / \overline{GT}_1$) pin of the 8086**

$\overline{RQ} / \overline{GT}_1$

- **The request / grant signal from the 8087 is usually connected to the request / grant pin of the independent processor such as 8089**

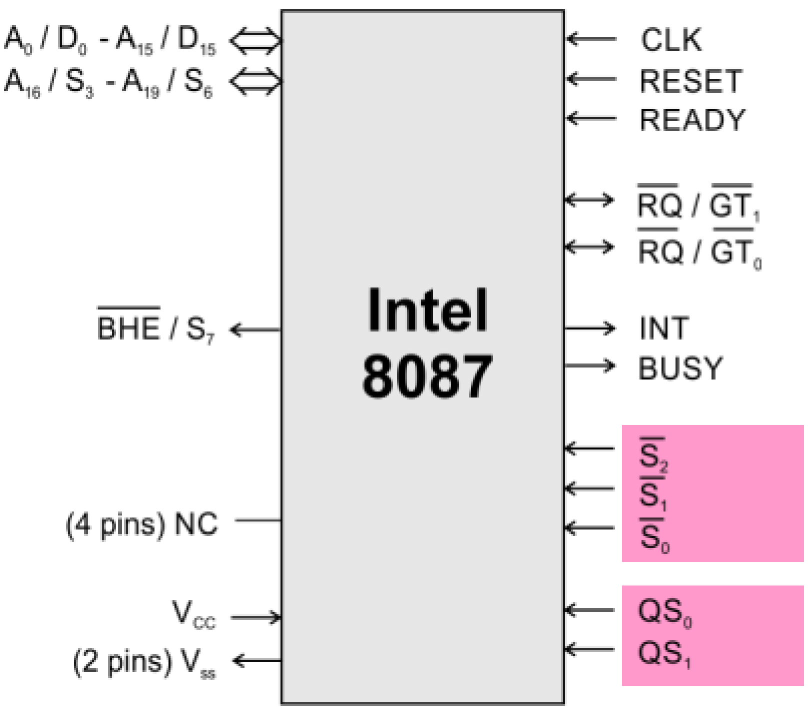
Co-processor – Intel 8087



INT

- **The interrupt pin is connected to the interrupt management logic.**
- **The 8087 can interrupt the 8086 through this interrupt management logic at the time error condition exists**

Co-processor – Intel 8087



$\bar{S}_0$ - $\bar{S}_2$

$\bar{S}_2$		$\bar{S}_1$	Status
1	0	0	Unused
1	0	1	Read memory
1	1	0	Write memory
1	1	1	Passive

QS_0 – QS_1

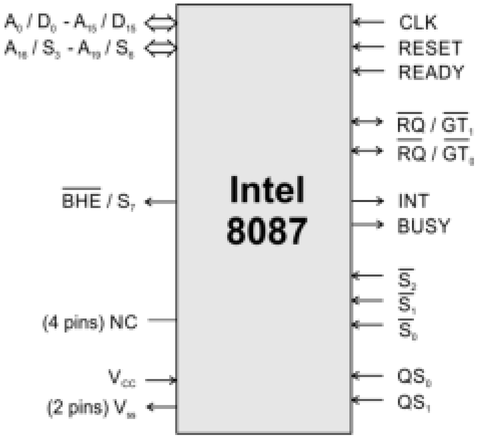
QS_0	QS_1	Status
0	0	No operation
0	1	First byte of opcode from queue
1	0	Queue empty
1	1	Subsequent byte of opcode from queue

Co-processor – Intel 8087

- **8087 instructions are inserted in the 8086 program**

- **8086 and 8087 reads instruction bytes and puts them in the respective queues**
- **NOP**
- **8087 instructions have 11011 as the MSB of their first code byte**

- **8087 keeps track for ESC instruction by monitoring $\overline{S_2}$ - $\overline{S_0}$ and AD_0 – AD_{15} of 8086.**
- **Also keeps track of QS_0 – QS_1.**
- **Q status 00; does nothing**
- **Q status 01; 8087 compares the five MSB bits with 11011**
- **If there is a match, then the ESC instruction is fetched and executed by 8087**
- **If there is error during decoding an ESC instruction, 8087 sends an interrupt request**



- **Memory read/ write**
- **Additional words : $\overline{RQ}$ - $\overline{GT_0}$**
- **8087 BUSY pin high**
- **$\overline{TEST}$**
- **WAIT**

Ref: Microprocessor, Atul P. Godse, Deepali A. Gode, Technical publications, Chap 11

Co-processor – Intel 8087