Microprocessor

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

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)

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

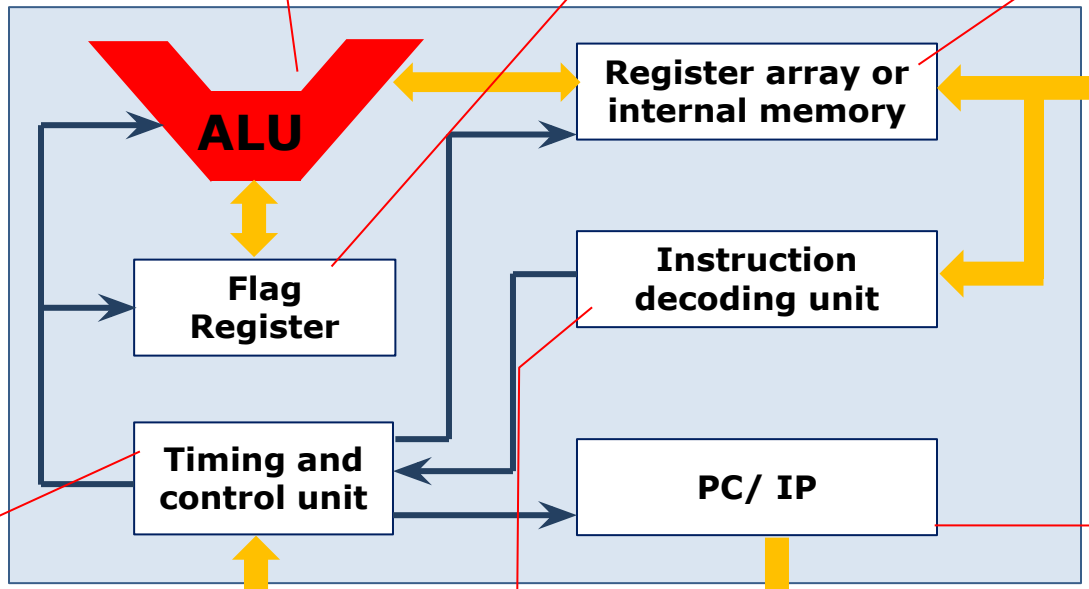
Intel 8086 (16 bit processor)

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

Fifth Generation Pentium

General Microprocessor 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.

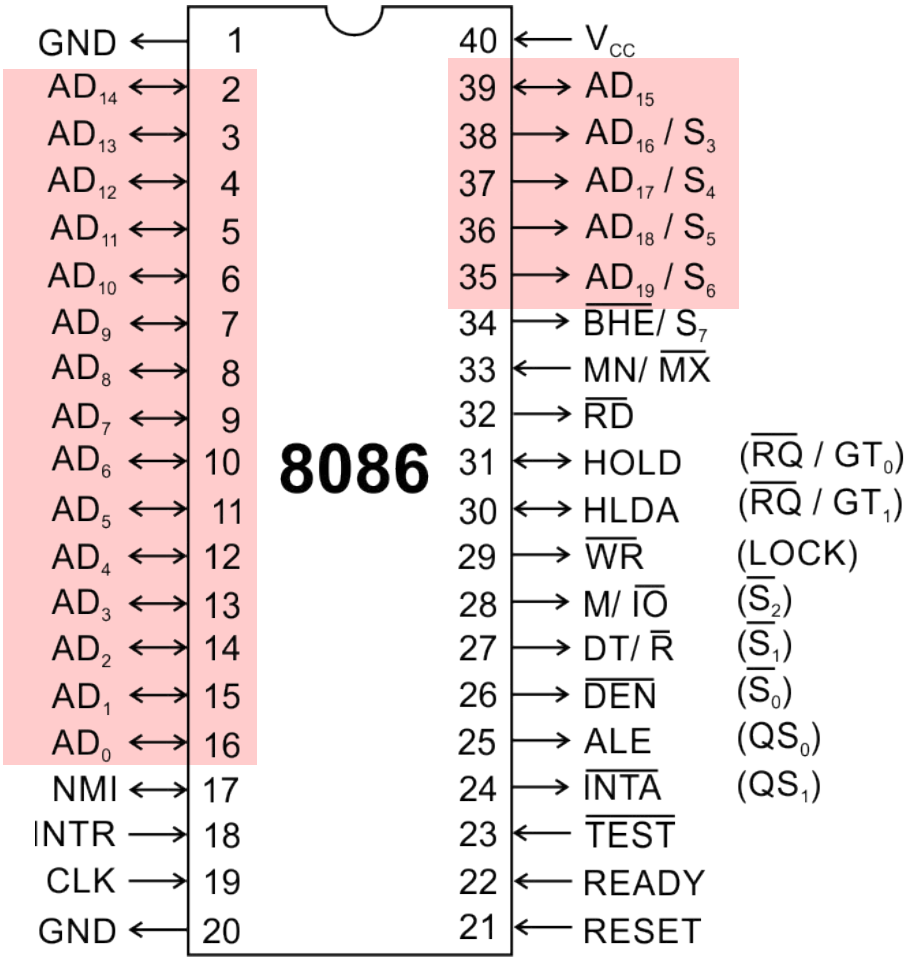
Addressable memory space is organized in to two banks of 512 KB each; Even (or lower) bank and Odd (or higher) bank. Address line A_0 is used to select even bank and control signal $\overline{BHE}$ is used to access odd bank

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.

Common signals

Pins and Signals



AD_0-AD_{15} (Bidirectional)

Address/Data bus

Low order address bus; these are multiplexed with data.

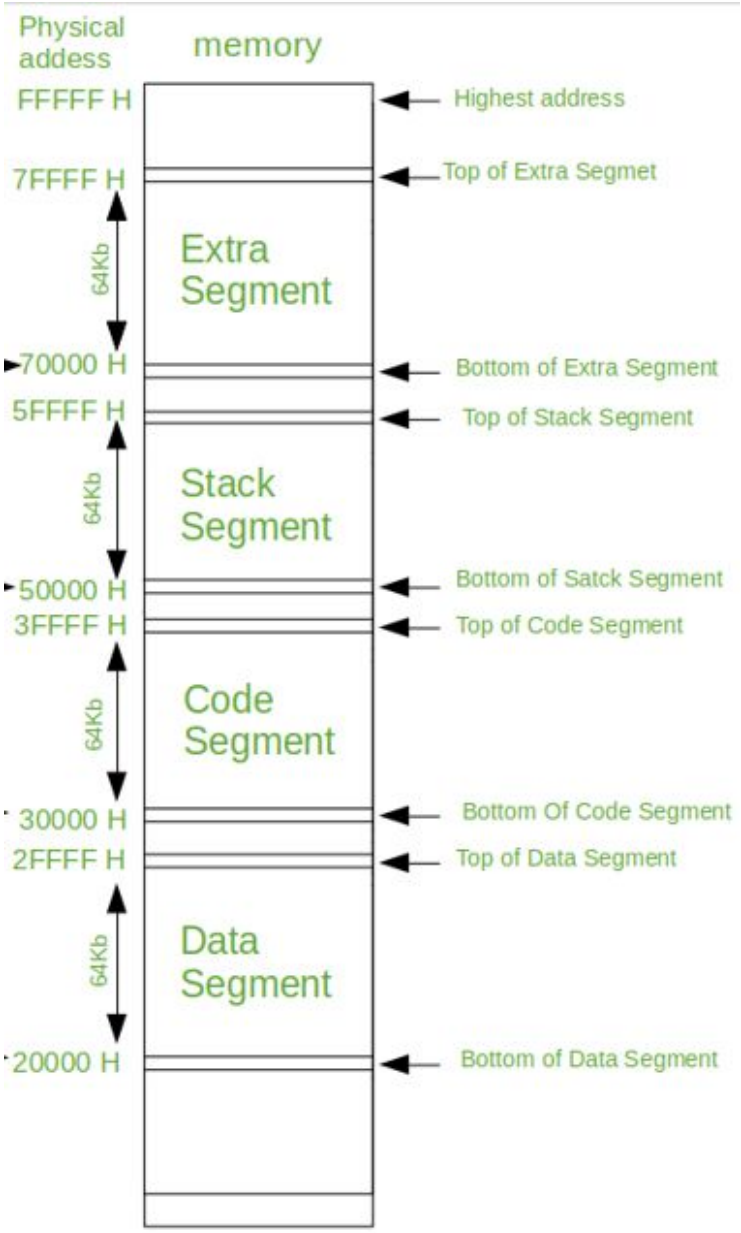
When AD lines are used to transmit memory address the symbol A is used instead of AD, for example A_0-A_{15}.

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

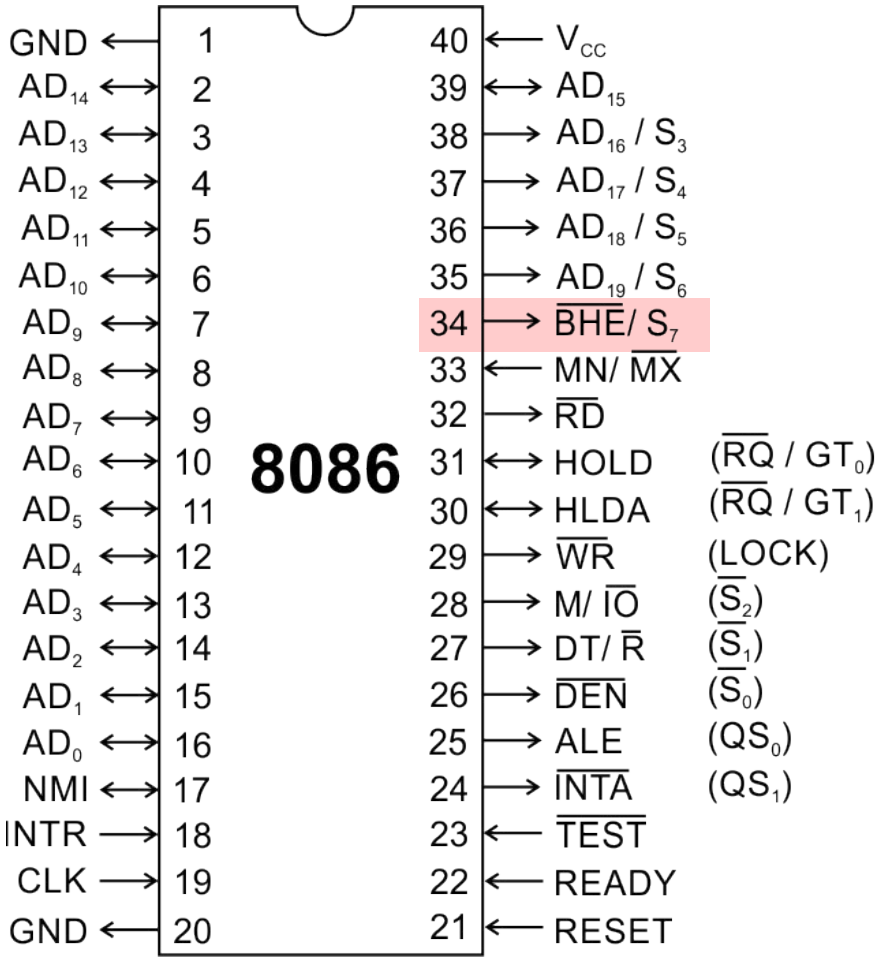
High order address bus. These are multiplexed with status signals

https://www.geeksforgeeks.org/pin-diagram-8086-microprocessor/


Physical addess
memory
FFFFF H
Highest address
7FFFF H
Top of Extra Segmet
64Kb
Extra Segment
70000 H
Bottom of Extra Segment
5FFFF H
Top of Stack Segment
64Kb
Stack Segment
50000 H
Bottom of Satck Segment
3FFFF H
Top of Code Segment
64Kb
Code Segment
30000 H
Bottom Of Code Segment
2FFFF H
Top of Data Segment
64Kb
Data Segment
20000 H
Bottom of Data Segment

Common signals

Pins and 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

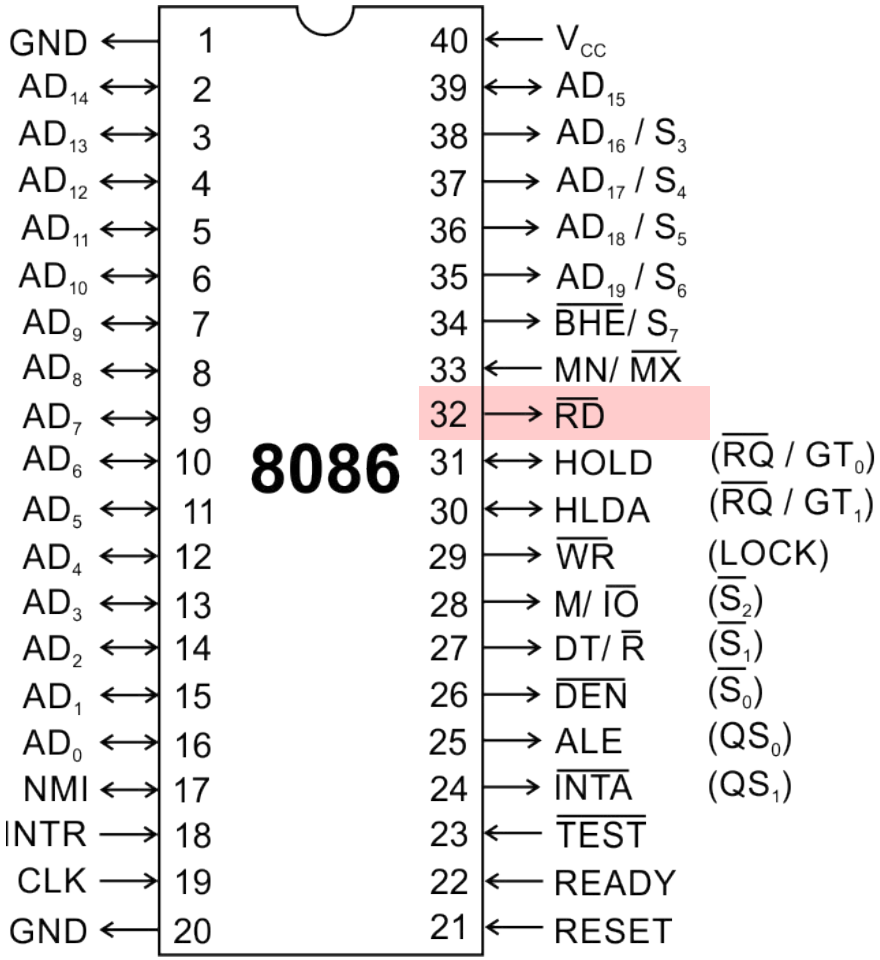
This pin signal indicates what mode the processor is to operate in.

RD (Read) (Active Low)

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

Common signals

Pins and 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.

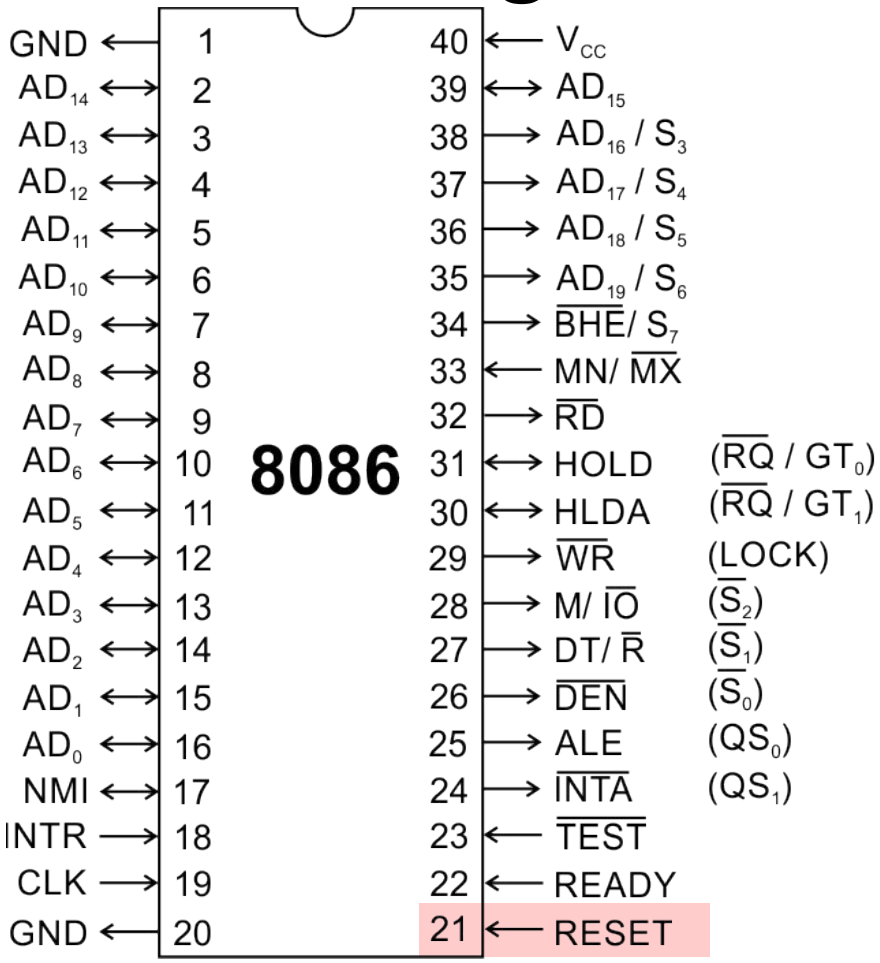
READY

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



- The 8086 does not have on-chip clock generation circuit. Hence the clock generator chip, 8284 is connected to the CLK pin of 8086.

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.

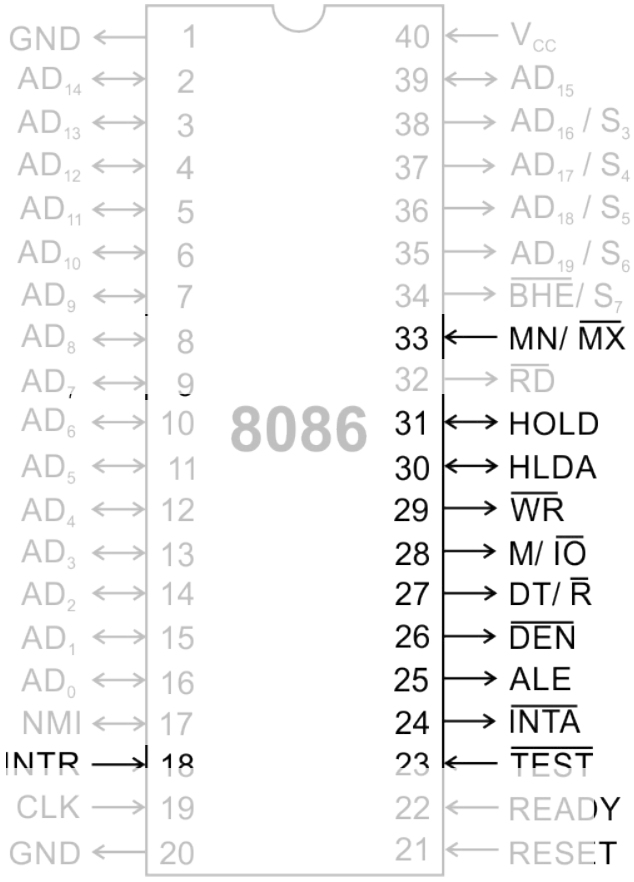
INTR Interrupt Request

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.

Min/ Max Pins

Pins and Signals



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

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

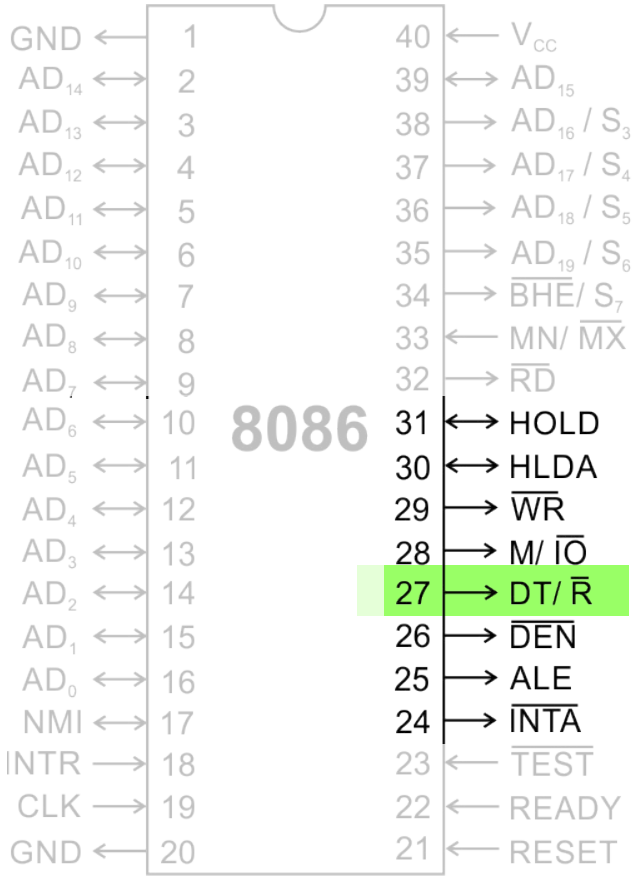
In the maximum mode the 8086 can work in multi-processor or co-processor configuration.

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

When this pin is high 8086 operates in minimum mode otherwise it operates in Maximum mode.

Minimum mode signals

Pins and Signals



Pins 24 -31

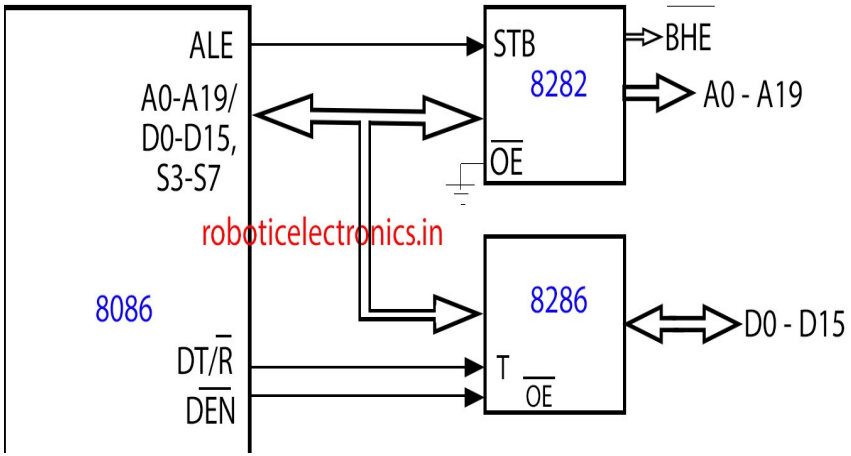
For minimum mode operation, the MN/ $\overline{MX}$ is tied to VCC (logic high)

8086 itself generates all the bus control signals

	(Data Transmit/ Receive) Output signal from the processor to control the direction of data flow through the data transceivers
	(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
	Used to differentiate memory access and I/O access. For memory reference instructions, it is high. For IN and OUT instructions, it is low.
	Write control signal; asserted low Whenever processor writes data to memory or I/O port
	(Interrupt Acknowledge) When the interrupt request is accepted by the processor, the output is low on this line.

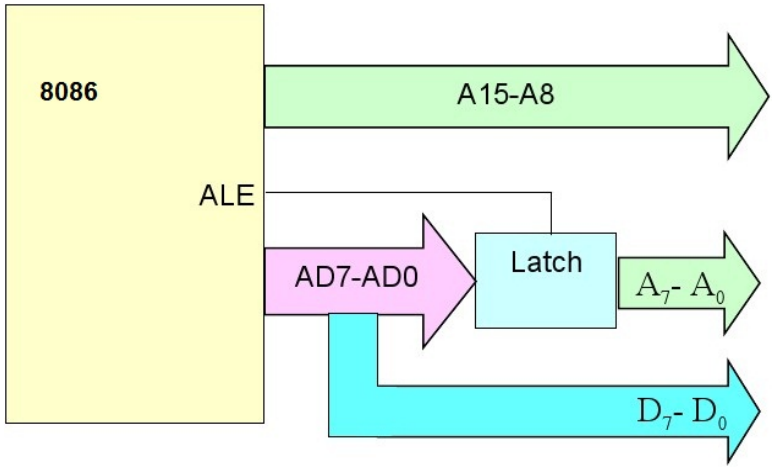
DEN

It stands for Data Enable and is available at pin 26. It is used to enable Transreceiver 8286. The transreceiver is a device used to separate data from the address/data bus.



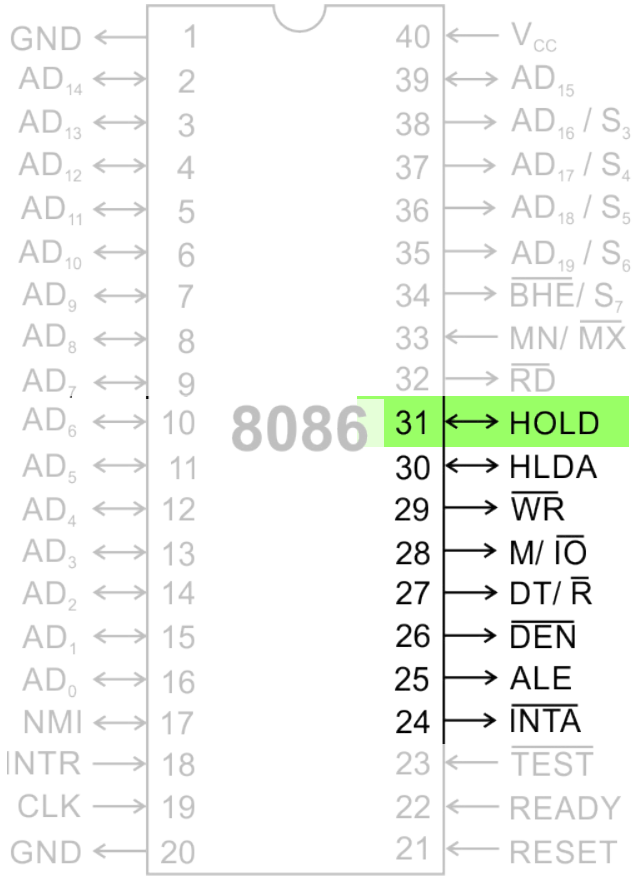
ALE

It stands for address enable latch and is available at pin 25. A positive pulse is generated each time the processor begins any operation. This signal indicates the availability of a valid address on the address/data lines.



Minimum mode signals

Pins and 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

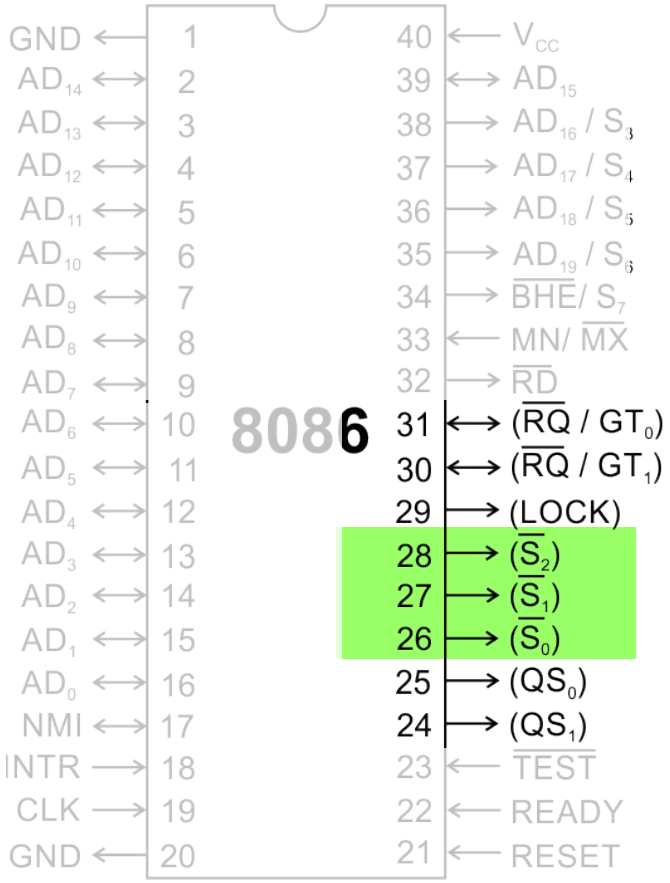
HOLD	**Input signal to the processor from the bus masters as a request to grant the control of the bus.** **Usually used by the DMA controller to get the control of the bus.**
HLDA	**(Hold Acknowledge) Acknowledge signal by the processor to the bus master requesting the control of the bus through HOLD.** **The acknowledge is asserted high, when the processor accepts HOLD.**

Maximum mode signals

Pins and Signals

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

Pins 24 -31 are reassigned



Status signals; used by the 8086 bus controller to generate bus timing and control signals. These are decoded as shown.

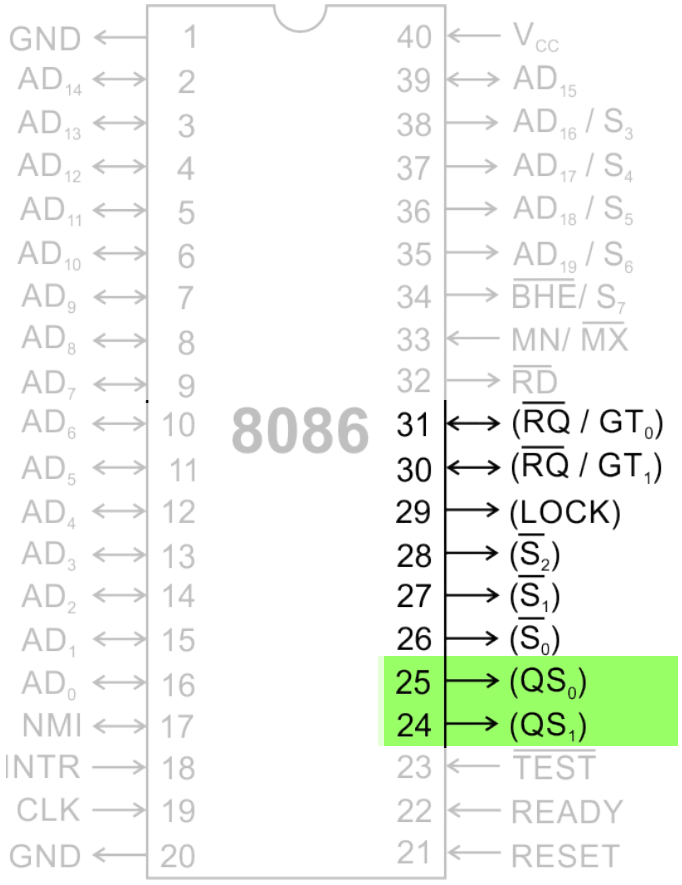
Status Signal			Machine Cycle
$\overline{S}_2$	$\overline{S}_1$	$\overline{S}_0$	
0	0	0	Interrupt acknowledge
0	0	1	Read I/O port
0	1	0	Write I/O port
0	1	1	Halt
1	0	0	Code access
1	0	1	Read memory
1	1	0	Write memory
1	1	1	Passive/Inactive

Maximum mode signals

Pins and Signals

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

Pins 24 -31 are reassigned



(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

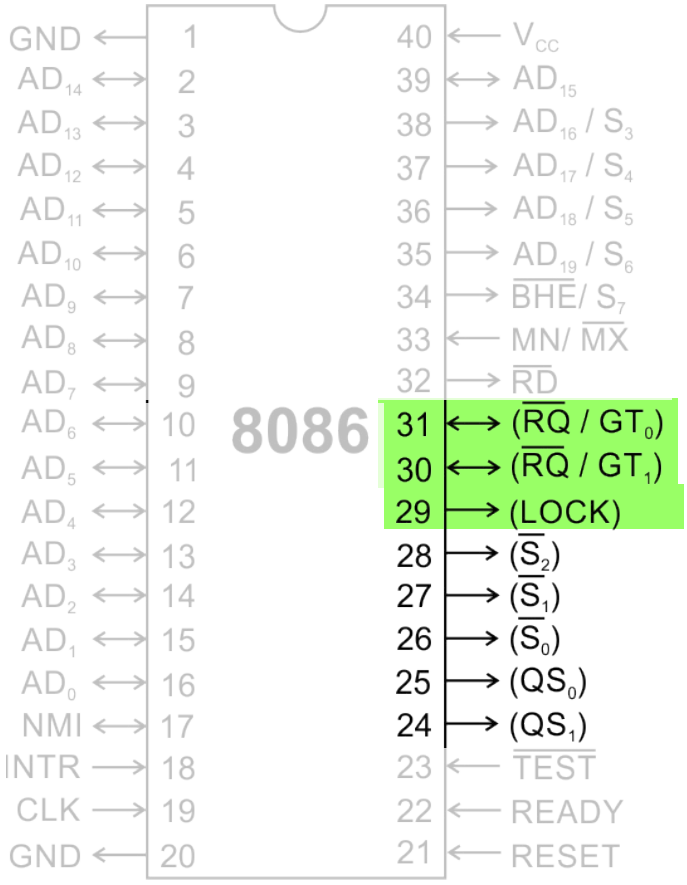
- These signals provide the status of instruction queue.

Maximum mode signals

Pins and Signals

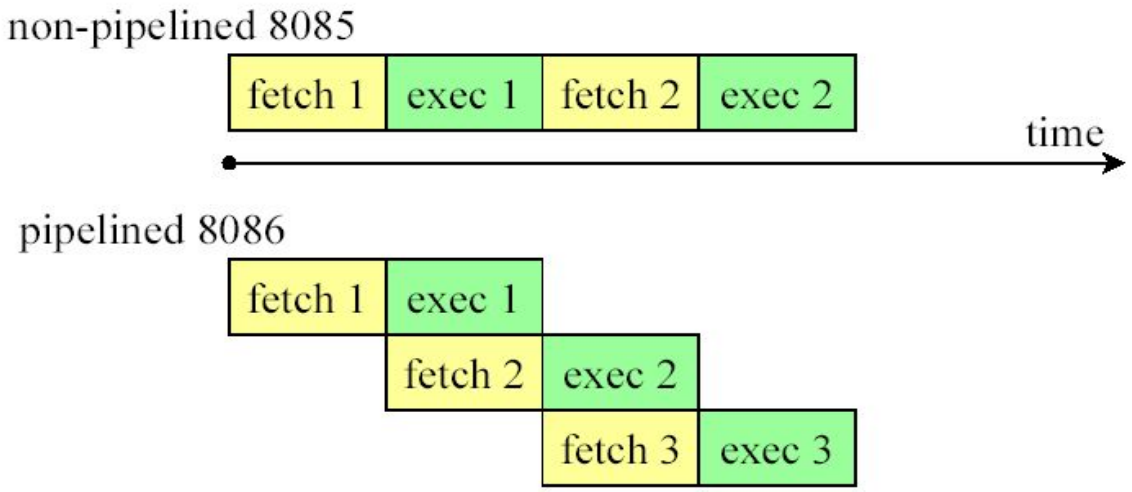
During maximum mode operation, the MN/ $\overline{MX}$ is grounded (logic low)

Pins 24 -31 are reassigned



Inside The 8088/8086…*pipelining*

- Pipelining
  - Two ways to make CPU process information faster:
    - Increase the working frequency – technology dependent
    - Change the internal architecture of the CPU
  - Pipelining is to allow CPU to fetch and execute at the same time



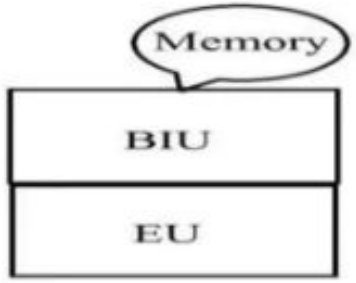
- 8086 contains two independent functional units. Bus Interface Unit (BIU) and Execution Unit (EU).
- The BIU handles transfer of data and addresses between the processor and memory I/O devices.
- The EU receives opcode of an instruction from the queue, decodes it and then executes it.
- While EU is decoding an instruction or executing an instruction, the BIU fetches instruction codes from the memory and stores them in the queue.
- The BIU and EU thus operate in parallel.

Generally a 8086 microprocessor is called a pipelined architecture. The process of fetching the next instruction when the present instruction is being executed is called as pipelining. Pipelining has become possible due to the use of queue. BIU (Bus Interfacing Unit) fills in the queue until the entire queue is full.

In 8086, to speed up the execution of program, the instructions fetching and execution of instructions are overlapped each other. This technique is known as pipelining. In pipelining, when the n^{th} instruction is executed, the $n+1^{th}$ instruction is fetched and thus the processing speed is increased.

Function of instruction queue in 8086:

In 8086, a 6-byte instruction queue is presented at the Bus Interface Unit (BIU). It is used to prefetching and store at the maximum of 6 bytes of instruction code from the memory. Due to this, overlapping instruction fetch with instruction execution increases the processing speed.



The functions of bus interface unit (BIU) in 8086:

(a) Fetch instructions from memory.

(b) Fetch data from memory and I/O ports.

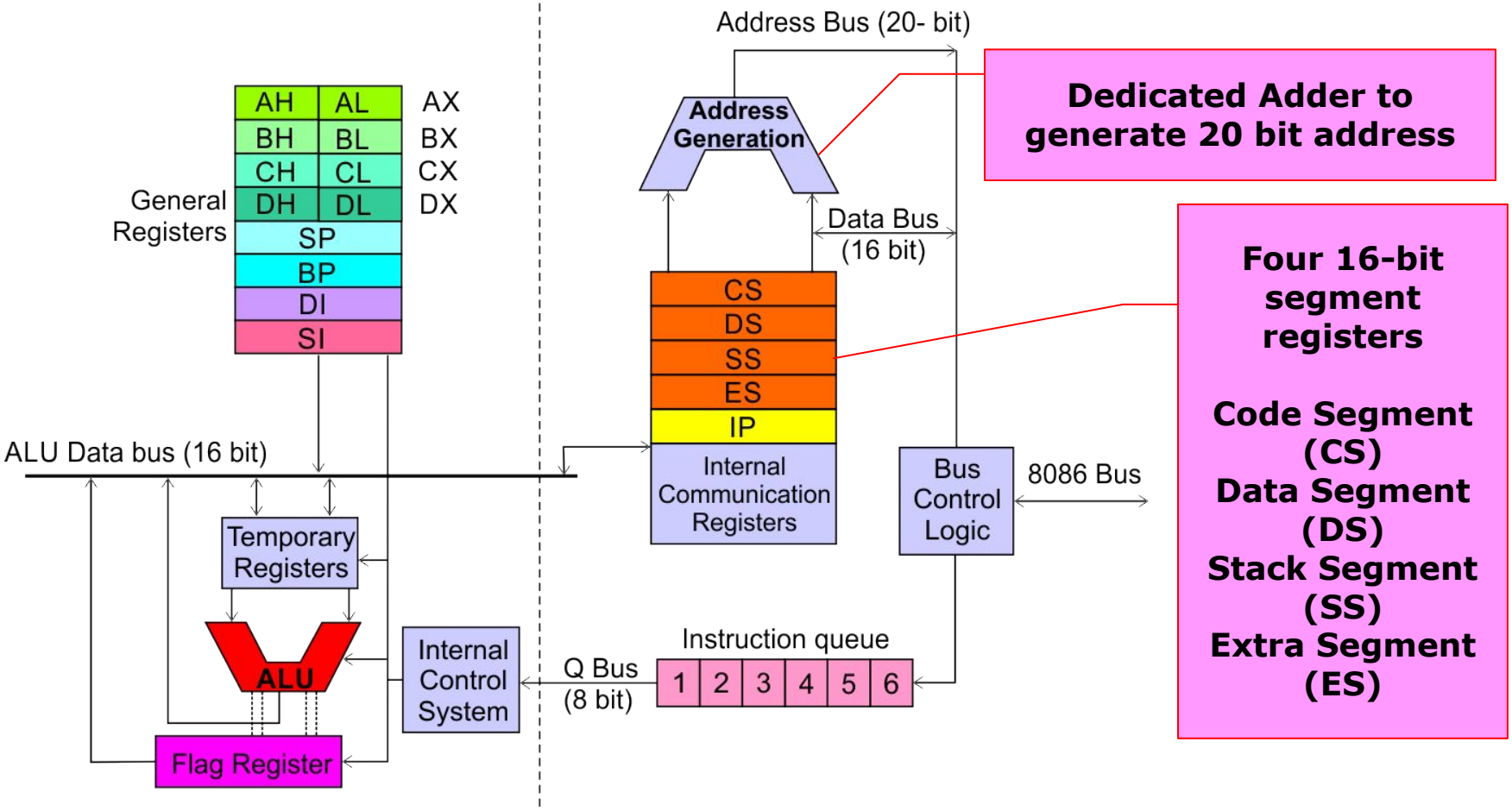
(c) Write data to memory and I/O ports.

(d) To communicate with outside world.

(e) Provide external bus operations and bus control signals.

- Pipeline in 8086 is a technique which is used in advanced microprocessors, where the microprocessor execute a second instruction before the completion of first. That is many instruction are simultaneously pipelined at different processing stage.
- The advantages of pipelining is performance improvement, we are able to pump more instructions and get improved in processor speed as we are able to execute parts of instructions in parallel to parts of other instruction.
- Disadvantage of pipeline is that it makes things complex, for example if we need to take care of branch penalty and forwarding, this become complex and several research problems are arise due to these complexity.

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.

Bus Interface Unit (BIU)

- It provides the interface of 8086 to external memory and I/O devices through the System Bus. It performs various machine cycles such as memory read, I/O read, etc. to transfer data between memory and I/O devices.
- BIU performs the following functions:
- It generates the 20-bit physical address for memory access.
- It fetches instructions from the memory.
- It transfers data to and from the memory and I/O.
- Maintains the 6-byte pre-fetch instruction queue(**supports pipelining**).
- BIU mainly contains the **4 Segment registers**, the **Instruction Pointer**, a pre-fetch queue, and an **Address Generation Circuit**.

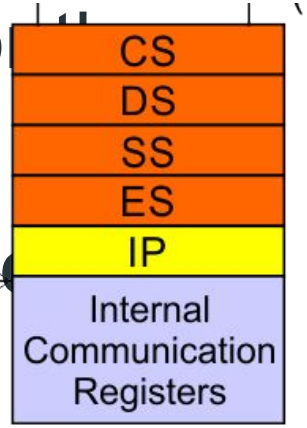
Instruction Pointer (IP):

- It is a *16-bit register*. It holds the offset of the next instructions in the *Code Segment (CS)*.
- IP is incremented after every instruction byte is fetched.
- IP gets a new value whenever a branch instruction occurs.
- CS is multiplied by 10H to give the 20-bit physical address of the Code Segment.
- The address of the next instruction is calculated by using the formula **CS x 10H + IP**

Example: If CS = 4321H and IP = 1000H
then CS x 10H = 43210H + offset = ***44210H,*** the address of the next instruction
Here Offset = Instruction Pointer(IP)

- **Code Segment register: (16 Bit register):** CS holds the base address for the Code Segment. All programs are stored in the Code Segment and accessed via the IP.
- **Data Segment register: (16 Bit register):** DS holds the base address for the Data Segment.
- **Stack Segment register: (16 Bit register):** SS holds the base address fo Stack Segment.
- **Extra Segment register: (16 Bit regist**

ES holds the base address for the Extra



- **Address Generation Circuit:**
- The BIU has a Physical Address Generation Circuit.
- It generates the 20-bit physical address using Segment and Offset addresses using the formula:
- ***Physical Address = Segment Address x 10H + Offset Address***

Execution Unit (EU)

- The main components of the EU are General purpose registers, the ALU, Special purpose registers, the Instruction Register and Instruction Decoder, and the Flag/Status Register.
- Functions are:

1. Fetches instructions from the Queue in BIU, decodes, and executes arithmetic and logic operations using the ALU.
2. Sends control signals for internal data transfer operations within the microprocessor.
3. Send request signals to the BIU to access the external module.
4. It operates with respect to T-states (clock cycles) and not machine cycles

- 8086 has four 16-bit general purpose registers: AX, BX, CX, and DX which store intermediate values during execution. Each of these has two 8-bit parts (higher and lower).
- **AX register: (Combination of A_L and A_H Registers)**
  It holds operands and results during multiplication and division operations. Also an accumulator during String operations.

- **BX register: (Combination of B_L and B_H Registers)**
  It holds the memory address (offset address) in indirect addressing modes.

- **CX register: (Combination of C_L and C_H Registers)**
  It holds the count for instructions like a loop, rotates, shifts and string operations.

- **DX register: (Combination of D_L and D_H Registers)**
  It is used with AX to hold 32-bit values during multiplication and division

- **Arithmetic Logic Unit (16-bit):** Performs **8 and 16-bit** arithmetic and logic operations.
- **Special purpose registers (16-bit):** Special purpose registers are called Offset registers also. Which points to specific memory locations under each segment.
- **Stack Pointer:** Points to Stack top. Stack is in Stack Segment, used during instructions like PUSH, POP, CALL, RET etc.
- **Base Pointer:** BP can hold the offset addresses of any location in the stack segment. It is used to access random locations of the stack.
- **Source Index:** It holds offset address in Data Segment during string operations.
- **Destination Index:** It holds offset address in Extra Segment during string operations.

- **Instruction Register and Instruction Decoder:**
- The EU fetches an opcode from the queue into the instruction register. The instruction decoder decodes it and sends the information to the control circuit for execution.
- **Flag/Status register (16 bits):** It has 9 flags that help change or recognize the state of the microprocessor.
- **6 Status flags:**

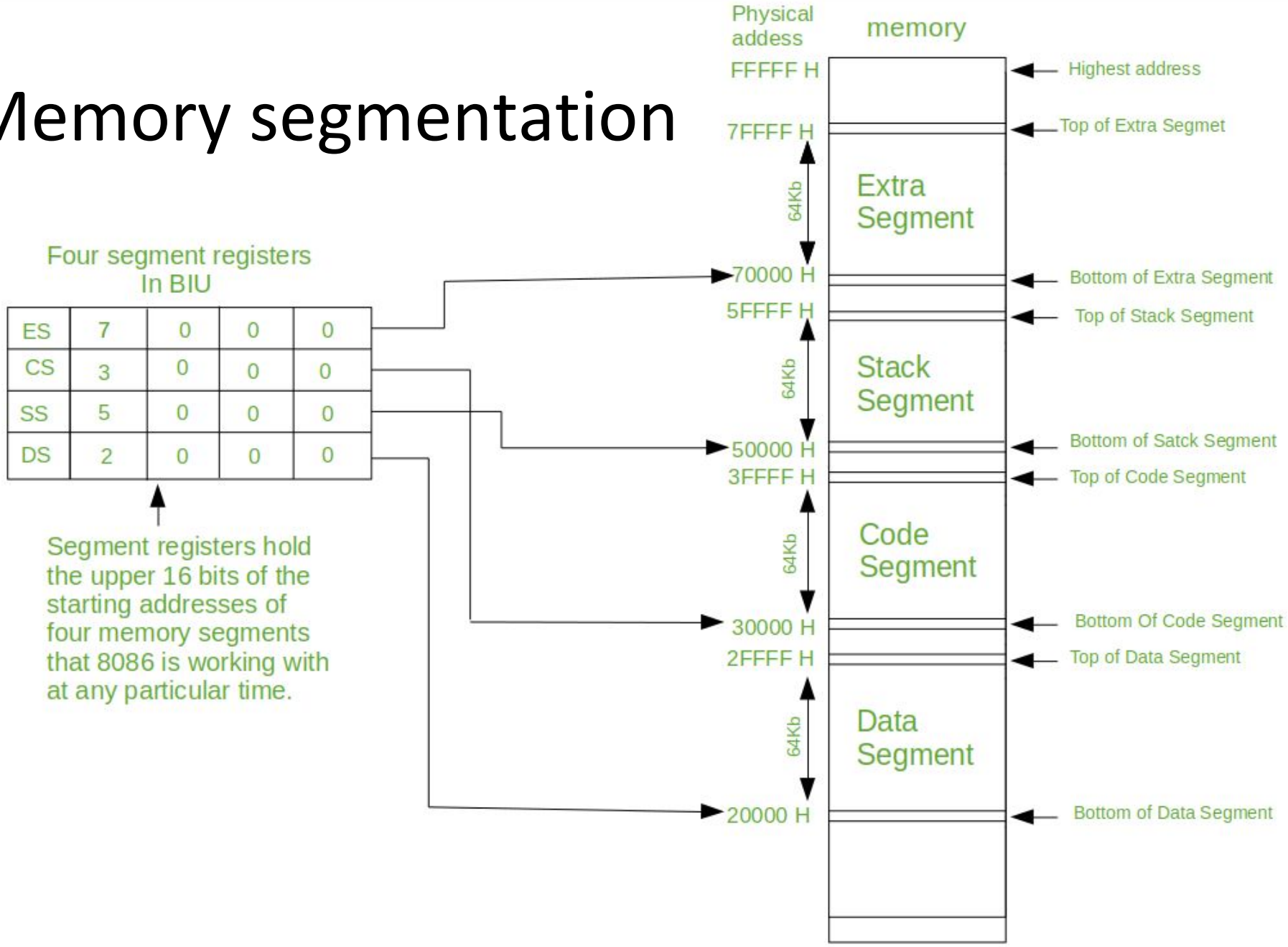
Carry flag(CF) Parity flag(PF) Auxiliary carry flag(AF) Zero flag(Z) Sign flag(S) Overflow flag (O)

- Status flags are updated after every arithmetic and logic operation.
- **3 Control flags:**

Trap flag(TF) Interrupt flag(IF) Direction flag(DF)

- These flags can be set or reset using control instructions

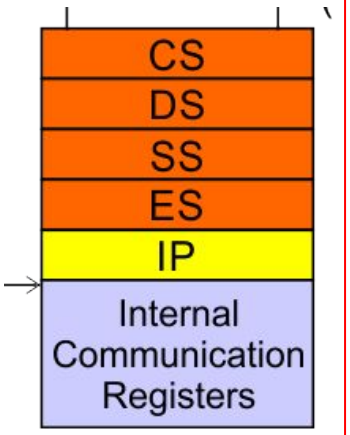
Memory segmentation

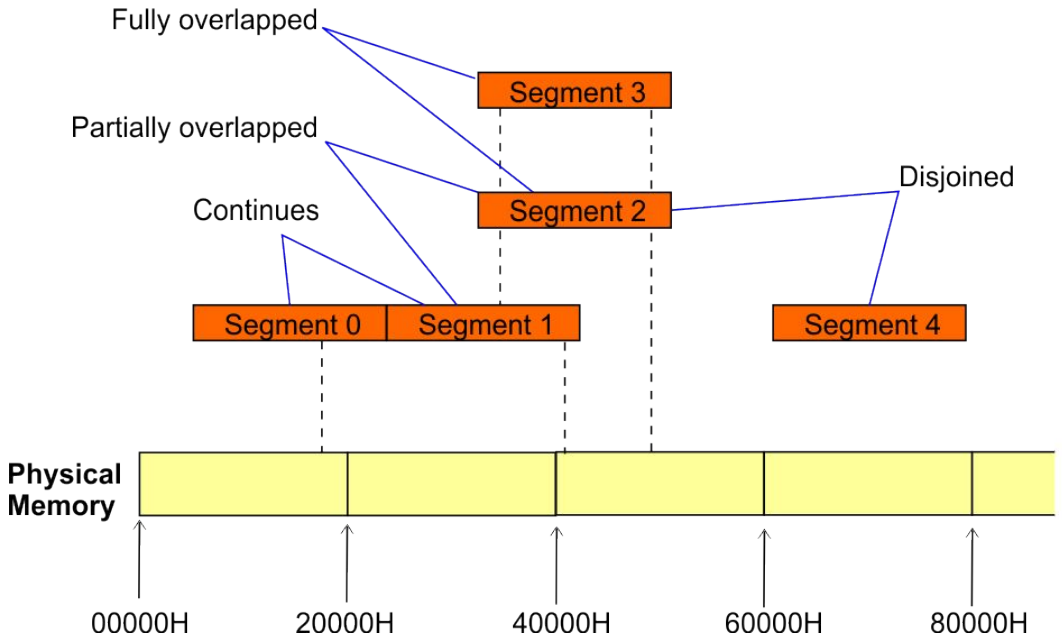


Architecture

Bus Interface Unit (BIU)

Segment Registers





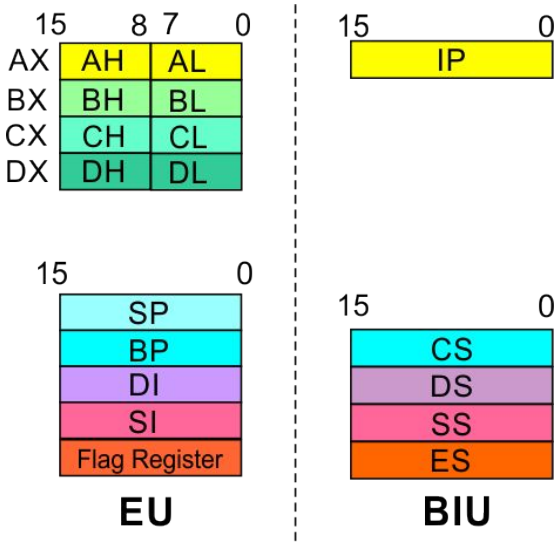
- **8086's 1-megabyte memory is divided into segments of up to 64K bytes each.**
- **The 8086 can directly address four segments (256 K bytes within the 1 M byte of memory) at a particular time.**
- **Programs obtain access to code and data in the segments by changing the segment register content to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

Architecture

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.

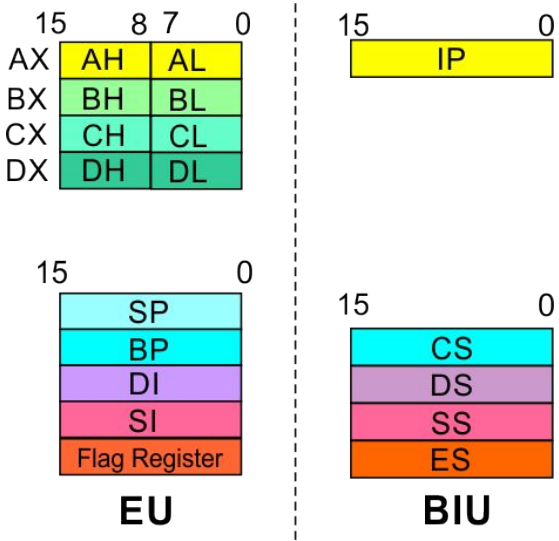


Architecture

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.

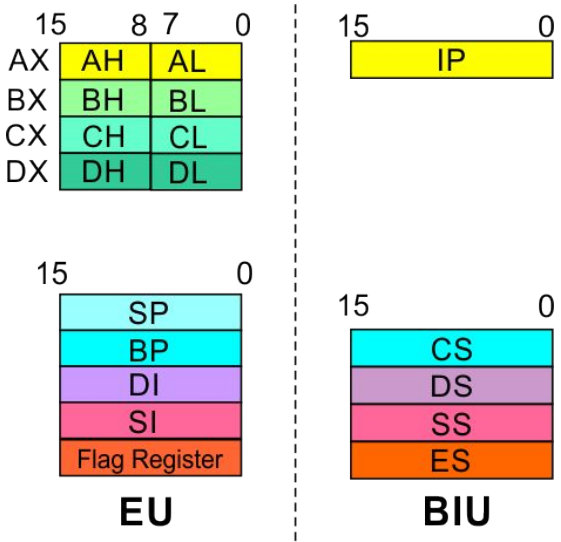


Architecture

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).

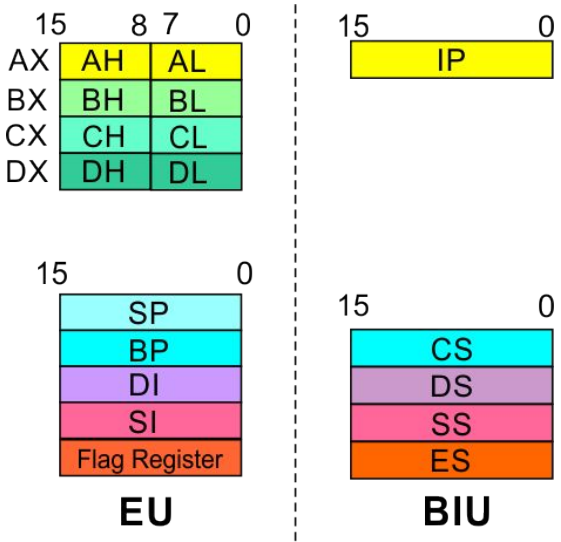


Architecture

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.**

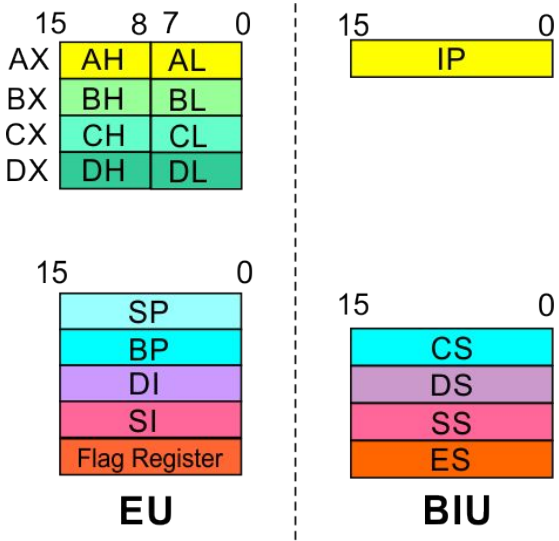


Architecture

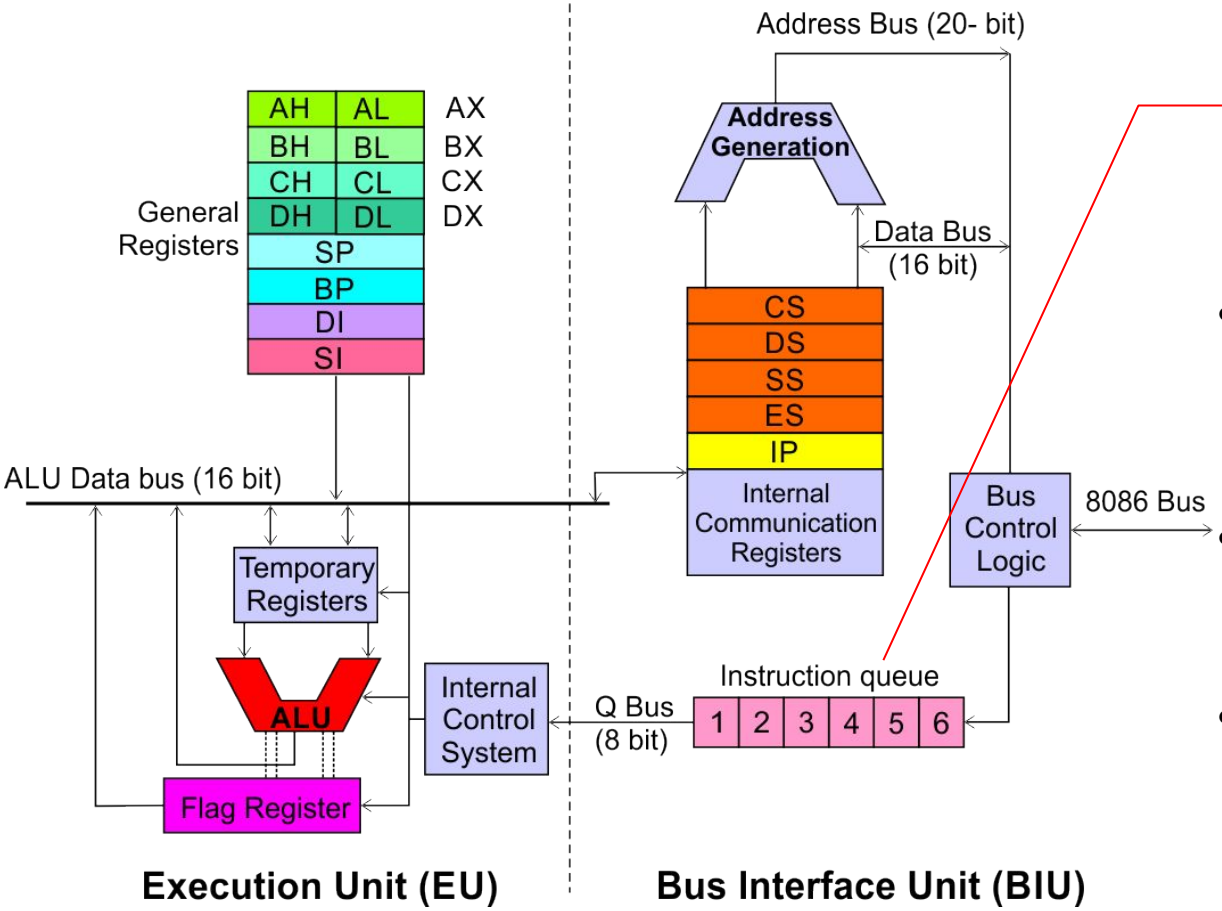
Segment Registers

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.**



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)

Architecture

EU decodes and executes instructions.

A decoder in the EU control system translates instructions.

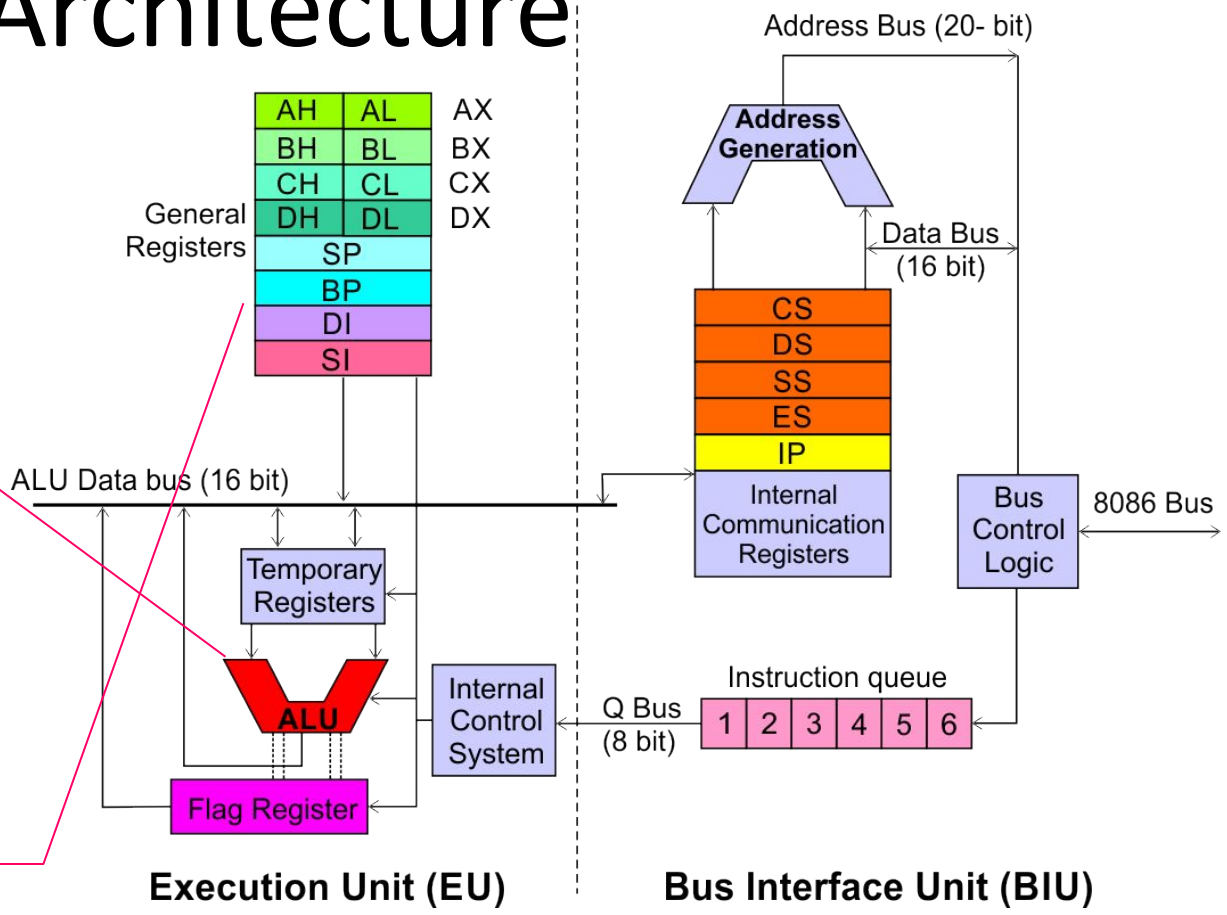
16-bit ALU for performing arithmetic and logic operation

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 :

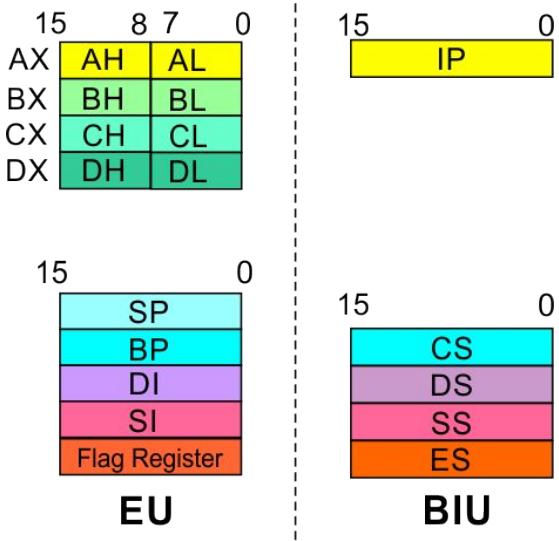
AX can be used as AH and AL
BX can be used as BH and BL
CX can be used as CH and CL
DX can be used as DH and DL

Architecture

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.**

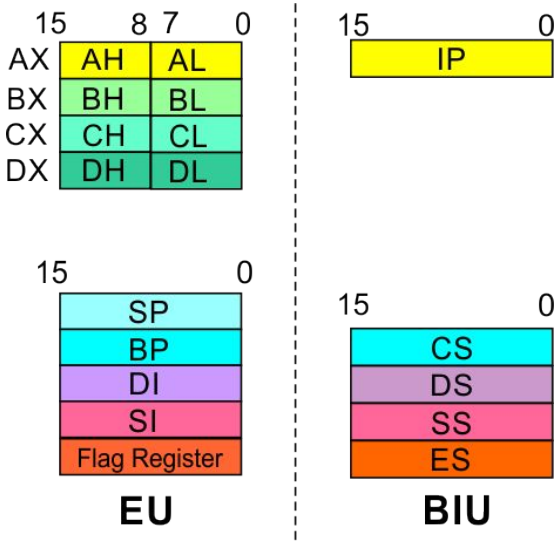


Architecture

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.**

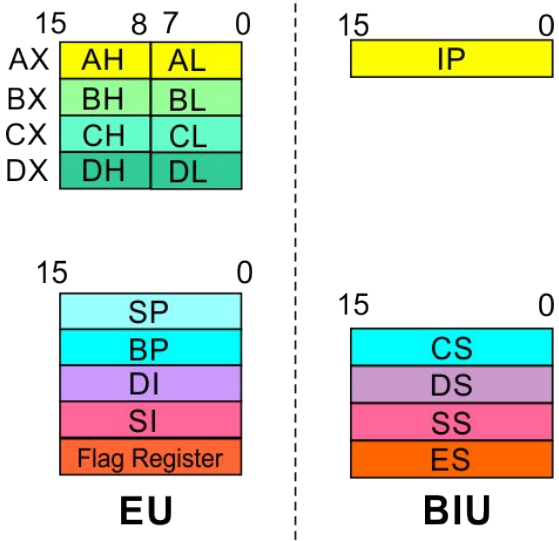


Architecture

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.

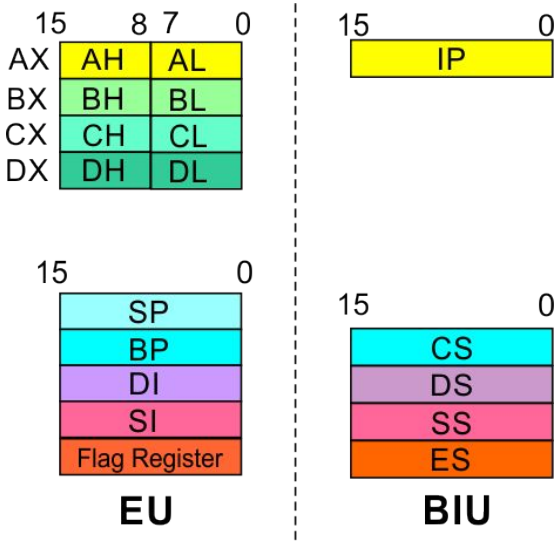
If it is zero, 8086 executes the next instruction; otherwise the 8086 branches to the label START.

Architecture

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.**

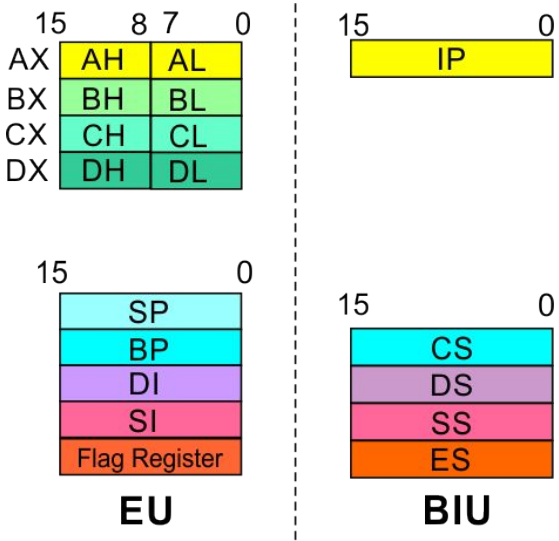


Architecture

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.**

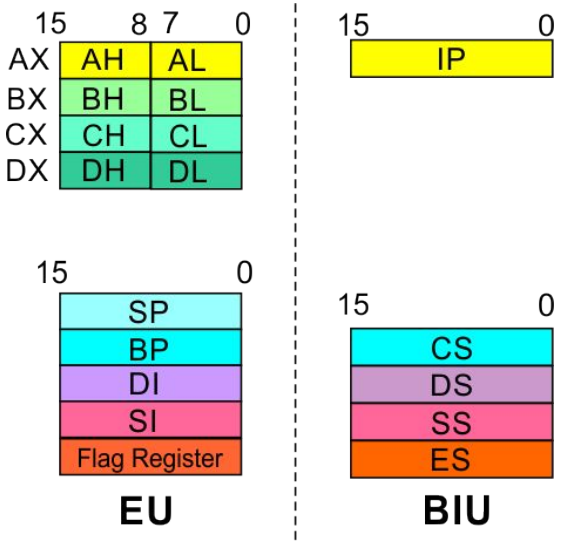


Architecture

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.**



Architecture

Execution Unit (EU)

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

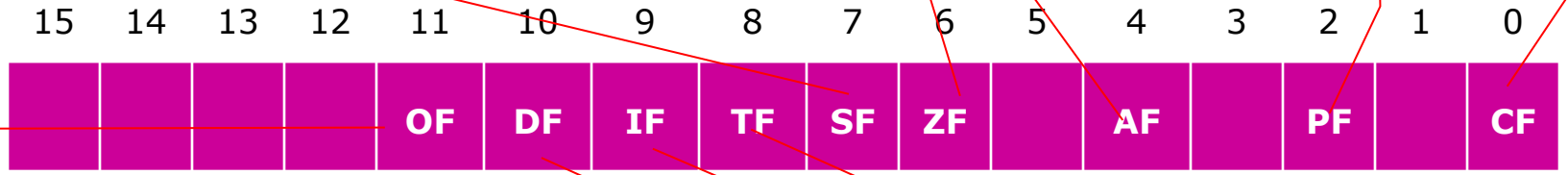
This flag is set, when the result of any computation is negative

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.

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

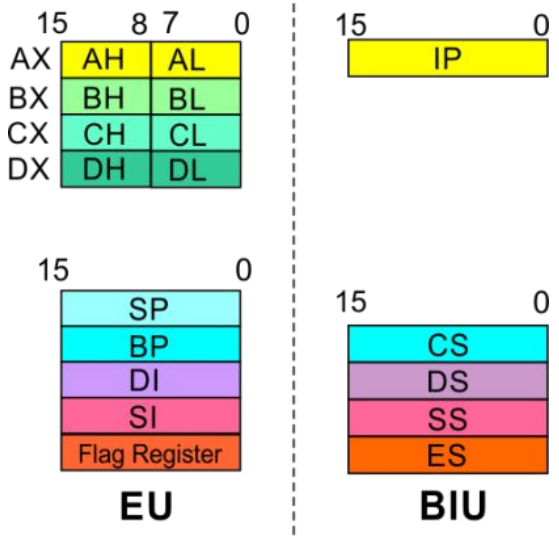
The instruction will specify the name of the register which holds the data to be operated by the instruction.

Example:

MOV CL, DH

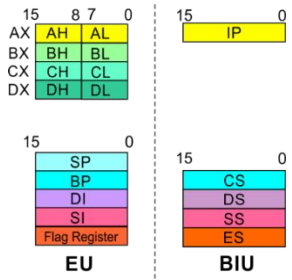
The content of 8-bit register DH is moved to another 8-bit register CL

(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

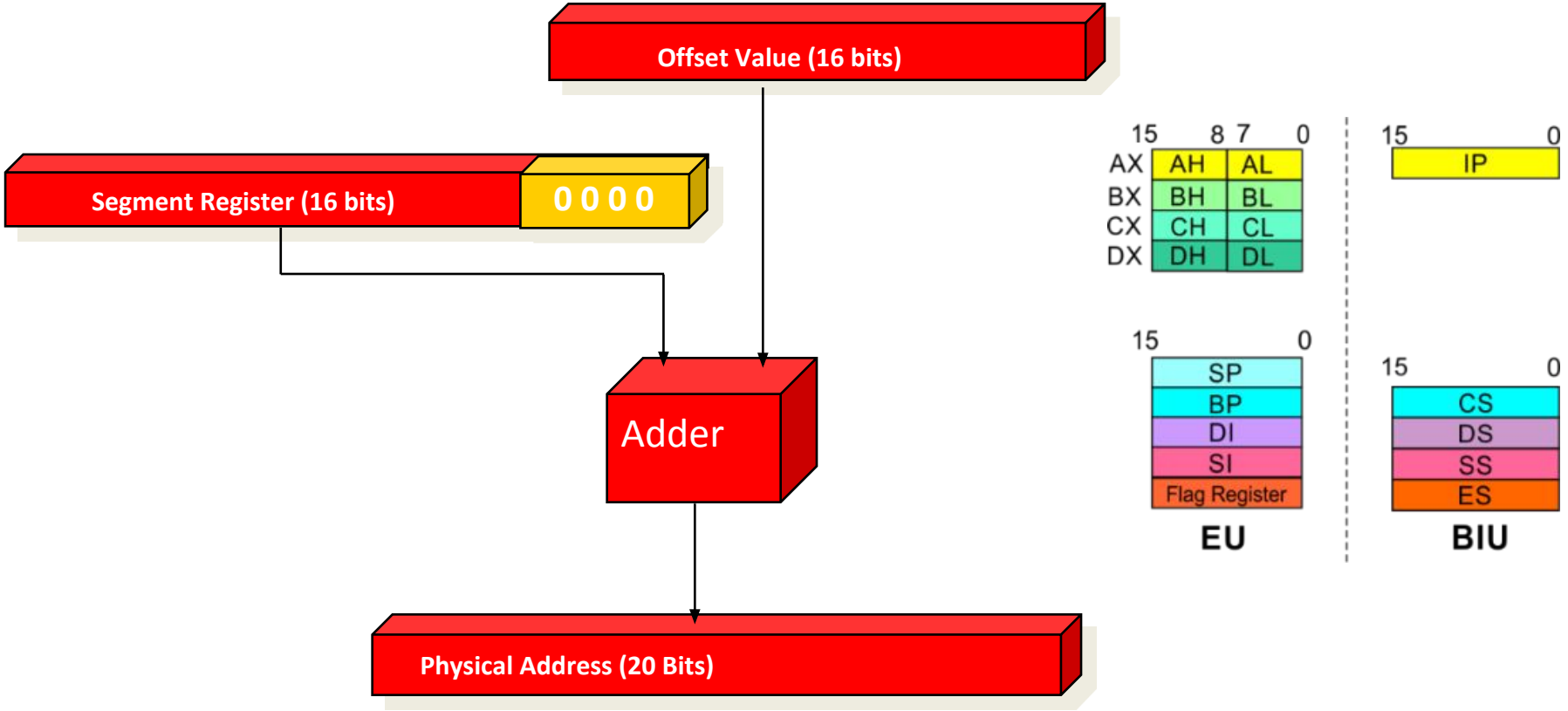
(DL) ← 08_H

MOV AX, 0A9FH

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

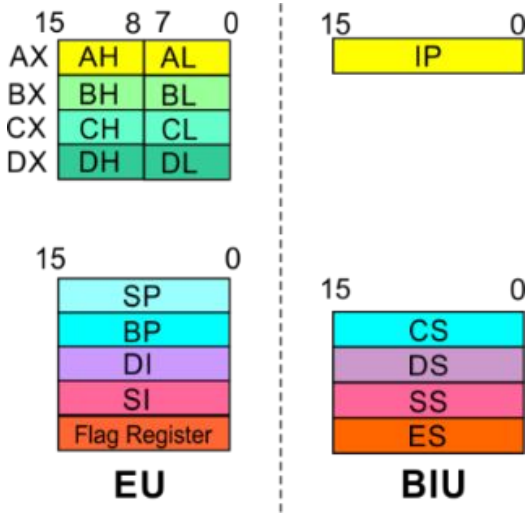
(AX) ← $0A9F_H$

Addressing Modes : Memory Access



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**

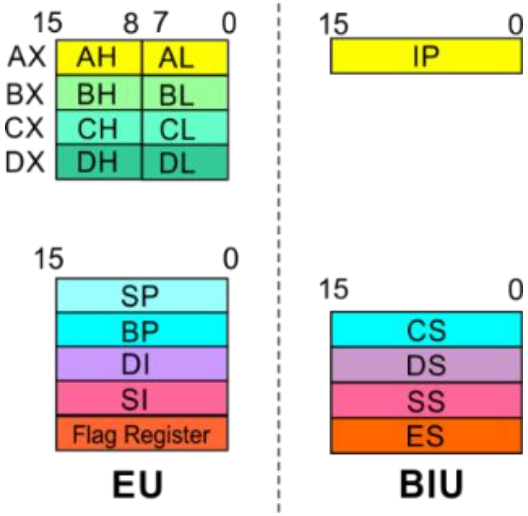


16 bytes of contiguous memory

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 : Memory Access

- **To access memory, we use these four registers: BX, SI, DI, BP**
- **Combining these registers inside [ ] symbols, we can get different memory locations (Effective Address, EA)**
- **Supported combinations:**

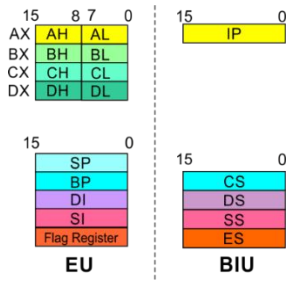


[BX + SI] [BX + DI] [BP + SI] [BP + DI]	[SI] [DI] d16 (variable offset only) [BX]	[BX + SI + d8] [BX + DI + d8] [BP + SI + d8] [BP + DI + d8]
[SI + d8] [DI + d8] [BP + d8] [BX + d8]	[BX + SI + d16] [BX + DI + d16] [BP + SI + d16] [BP + DI + d16]	[SI + d16] [DI + d16] [BP + d16] [BX + d16]

BX BP	SI DI	+ disp

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:

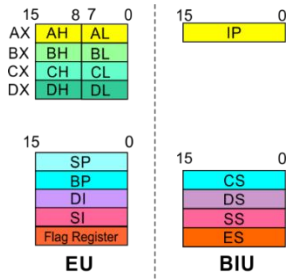
```
MOV  BX, [1354H]
MOV  BL, [0400H]
```

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]

Operations:

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

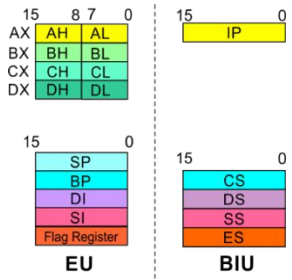
(CX) ← (MA) or,

(CL) ← (MA)
(CH) ← (MA +1)

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

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:

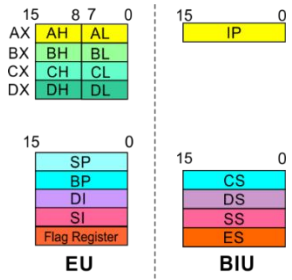
$0008_H \leftarrow 08_H$ (Sign extended)
EA = (BX) + 0008_H
BA = (DS) x 16_{10}
MA = BA + EA

(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)

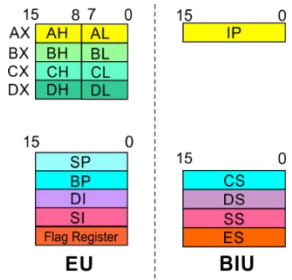
$EA = (SI) + FFA2_H$
$BA = (DS) \times 16_{10}$
$MA = BA + EA$

$(CX) \leftarrow (MA)$ 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

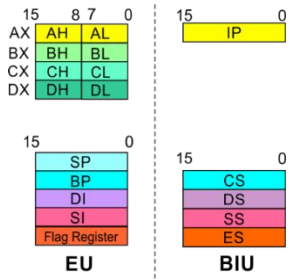
(MAE) ← (MA)

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) ← (PORT)

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

In indirect port addressing mode, the instruction will specify the name of the register which holds the port address. In 8086, the 16-bit port address is stored in the DX register.

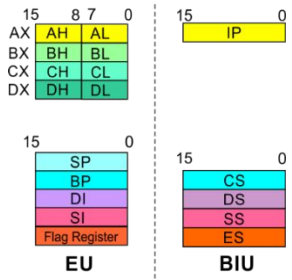
Example: OUT [DX], AX

Operations: $PORT_{addr} = (DX)$
(PORT) ← (AX)

Content of AX is moved to port whose address is specified by DX 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

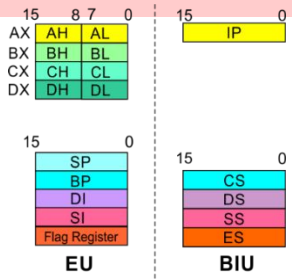
$EA = (IP) + 000A_H$
$BA = (CS) \times 16_{10}$
$MA = BA + EA$

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

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**

https://www.tutorialspoint.com/assembly_programming/assembly_logical_instructions.htm

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)**
PUSHF-pushes the contents of the FLAGS register onto the stack.	**(SP) ← (SP) – 2** **MA S = (SS) x 1610 + SP** **(MA S ; MA S + 1) ← (Flags)**
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, addr16	**(AX) ← (addr16)**

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 addr16, AX	**(addr16) ← (AX)**

Instruction Set

2. Arithmetic Instructions

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

ADD reg2/ mem, reg1/mem	
ADD reg2, reg1	(reg2) ← (reg1) + (reg2)
ADD reg2, mem	(reg2) ← (reg2) + (mem)
ADD 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
DAA (Decimal Adjust after Addition)	
ADD AL, data8	(AL) ← (AL) + data8 - Result in HEX
DAA	(AL) ← (AL) -Result in BCD

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
ADC A, data	
ADC AL, data8	(AL) ← (AL) + data8+CF
ADC 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) ← (reg2) - (reg1)** **(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) ← (reg2) - (reg1) - 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

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) ← (AL) 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) AL:Quotient; AH:Remainder** **(AH) ← (AX) MOD(reg8) Remainder** **For 32-bit :- 16-bit :** **(AX) ← (DX)(AX) :- (reg16) Quotient AX: Quotient, DX: Remainder** **(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…**

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) – (data)** **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 …** ROR, ROL

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 …** ROR, ROL

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 …** ROR, ROL

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

Instruction Set

3. Logical Instructions

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

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

The TEST instruction works same as the AND operation, but unlike AND instruction, it does not change the first operand. So, if we need to check whether a number in a register is even or odd, we can also do this using the TEST instruction without changing the original number.

Instruction Set

3. Logical Instructions

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

SHR reg/mem

SHR reg

i) SHR reg, 1

ii) SHR reg, CL

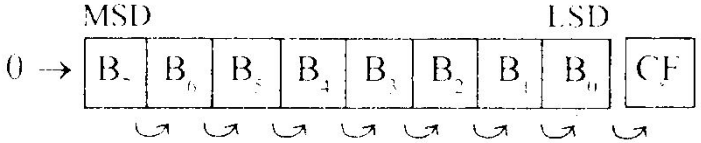
SHR mem

i) SHR mem, 1

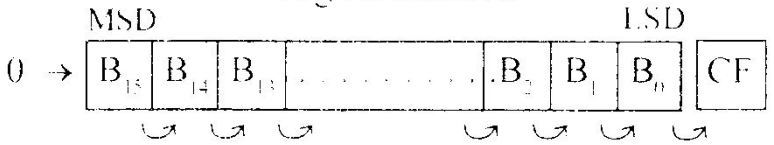
ii) SHR mem, CL

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

reg 8 / mem 8



reg 16 / mem 16



Instruction Set

3. Logical Instructions

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

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

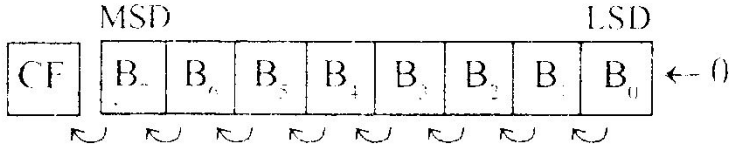
SHL mem or SAL mem

i) SHL mem, 1 or SAL mem, 1

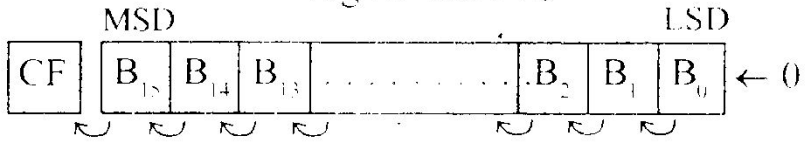
ii) SHL mem, CL or SAL mem, CL

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

reg 8 / mem 8



reg 16 / mem 16



Instruction Set

3. Logical Instructions

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

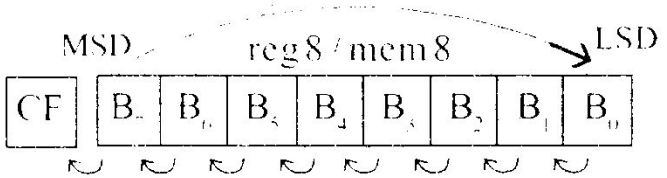
ROL reg/mem

ROL reg

i) ROL reg, 1

ii) ROL reg, CL

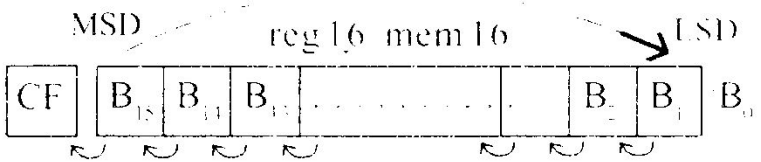
$B_{n+1} \leftarrow B_n : CF \leftarrow B_{MSD} ; B_{LSD} \leftarrow B_{MSD}$



ROL mem

i) ROL mem, 1

ii) ROL mem, CL



Instruction Set

3. Logical Instructions

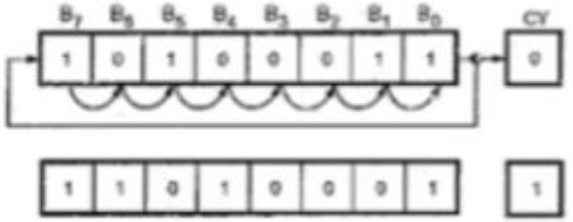
Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL …** ROR, ROL

ROR – Rotate right

Rotate right either by one bit or count specified by CL register.

MOV AL,56H ; AL=01010110 CF=1

ROR AL,01 ; AL=00101011 CF=0



Instruction Set

3. Logical Instructions

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

RCR reg/mem

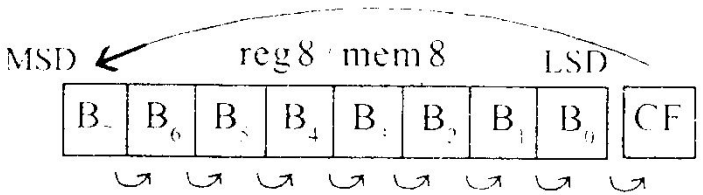
RCR reg

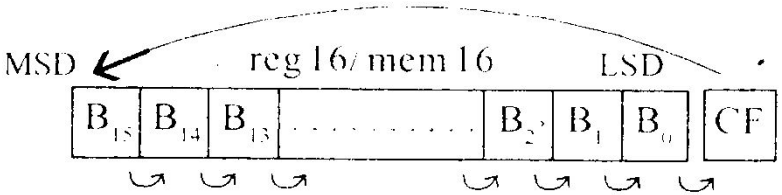
i) RCR reg, 1
ii) RCR reg, CL

RCR mem

i) RCR mem, 1
ii) RCR mem, CL

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



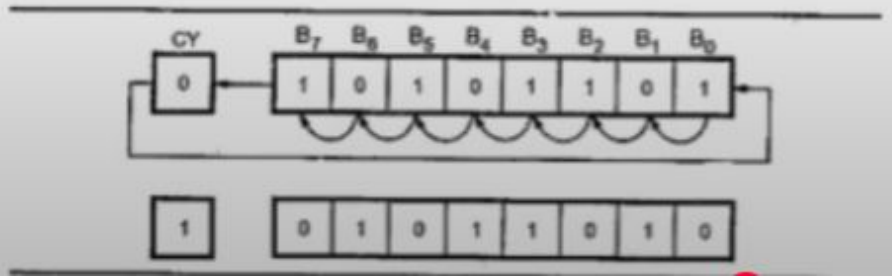


RCL – Rotate left through carry

- This 8086 Rotate Instructions all of the bits in a specified word or byte some number of bit positions to the left along with the carry flag. MSB is placed as a new carry and previous carry is placed as a new LSB

MOV AL,57H ; AL=01010110 CF=1

RCL AL,01 ; AL=10101101 CF=0



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.

Instruction Set

4. String Manipulation Instructions

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

REP	Result should be zero for condition true
REPZ/ REPE (Repeat CMPS or SCAS until ZF = 0)	While CX ≠ 0 and ZF = 1, repeat execution of string instruction and (CX) ← (CX) – 1
	Result should not be zero for condition true
REPNZ/ REPNE (Repeat CMPS or SCAS until ZF = 1)	While CX ≠ 0 and ZF = 0, repeat execution of string instruction and (CX) ← (CX) - 1

Ex: rep movsb

Note: Always **'REPZ'** instruction can be used in association with the string related operations.

Instruction Set

4. String Manipulation Instructions

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

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

Instruction Set

4. String Manipulation Instructions

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

Compare two string byte or string word

CMPS	

CMPSB	MA = (DS) x 16_{10} + (SI)
	MA_E = (ES) x 16_{10} + (DI)
	
	Modify flags ← (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
	If (MA) = (MA_E), then CF = 0; ZF = 1; SF = 0
	
	<u>For byte operation</u>
	If DF = 0, then (DI) ← (DI) + 1; (SI) ← (SI) + 1
	If DF = 1, then (DI) ← (DI) - 1; (SI) ← (SI) - 1
	
	<u>For word operation</u>
	If DF = 0, then (DI) ← (DI) + 2; (SI) ← (SI) + 2
	If DF = 1, then (DI) ← (DI) - 2; (SI) ← (SI) - 2

CMPSW	

Instruction Set

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) x 16_{10} + (DI)
	Modify flags ← (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) ← (DI) + 1
	If DF = 1, then (DI) ← (DI) – 1

SCASW	MA_E = (ES) x 16_{10} + (DI)
	Modify flags ← (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) ← (DI) + 2
	If DF = 1, then (DI) ← (DI) – 2

Instruction Set

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**

Instruction Set

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

Instruction Set

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

Instruction Set

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

Instruction Set

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
JPE disp8 Jump if even parity	JNP disp8 Jump if Odd parity

- 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

Note: Before these instructions comparison instruction is to be used for comparing two operands.

Instruction Set

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/JPE 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, ZF = 1
JNZ disp8	Jump if result is not zero, i.e, ZF = 0

Iteration Control Instructions

These instructions are used to execute the given instructions for number of times. Following is the list of instructions under this group −

LOOP − Used to loop a group of instructions until the condition satisfies, i.e., CX = 0

LOOPE/LOOPZ − Used to loop a group of instructions till it satisfies ZF = 1 & CX = 0

LOOPNE/LOOPNZ − Used to loop a group of instructions till it satisfies ZF = 0 & CX = 0

JCXZ − Used to jump to the provided address if CX = 0

```
Mov cx, 06h
label: add al, bl
Loop label
```

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 (16-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 (32-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

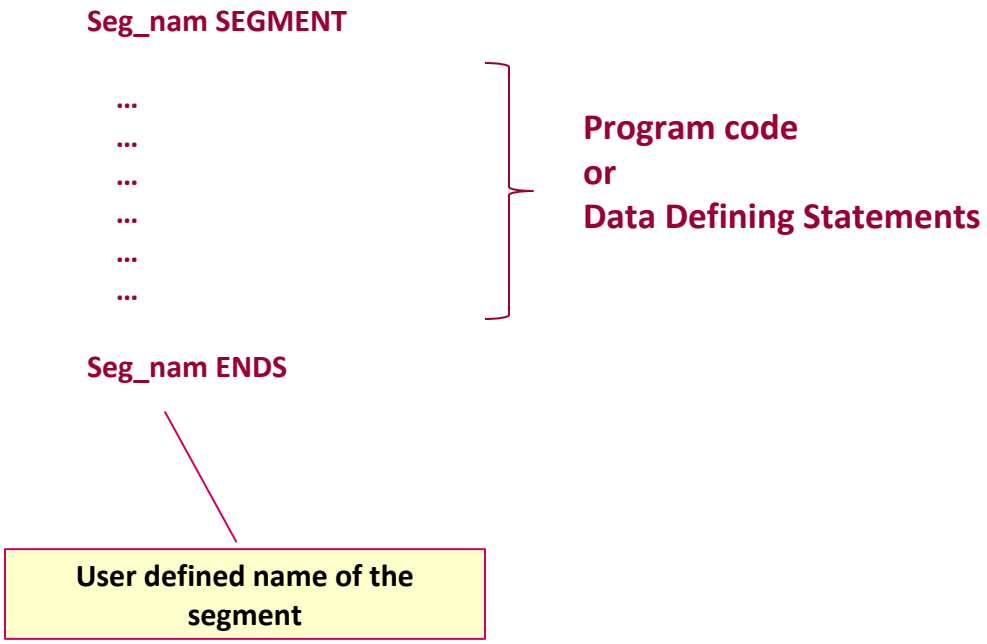
ORG
END
EVEN
EQU

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:**



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.

LENGTH: LENGTH is an operator, which tells the assembler to determine the number of elements in some named data item, such as a string or an array. When the assembler reads the statement MOV CX, LENGTH STRING1, for example, will determine the number of elements in STRING1 and load it into CX. If the string was declared as a string of bytes, LENGTH will produce the number of bytes in the string. If the string was declared as a word string, LENGTH will produce the number of words in the string.

LENGTH: Byte length of a label: This is used to refer to the length of a data array or a string. Ex : MOV CX, LENGTH ARRAY

OFFSET: offset of a label: When the assembler comes across the OFFSET operator along with a label, it first computing the 16-bit offset address of a particular label and replace the string 'OFFSET LABEL' by the computed offset address. Ex : MOV SI, offset list

LEA : Load Effective address : loads the address of variable.

Ex: Test DB 23H, 40H, 44H
LEA AX, Test

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

```
Proc_name PROC[NEAR/ FAR]

        ...
        ...
        ...

        RET

Proc_name 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:

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

Program statements in the macro

User defined name of the macro

Procedures and Macros:

http://www.snjb.org/polytechnic/up-images/downloads/chapter%206-MAPupFile_058d4fa990abaa.pdf.

Define procedure : A procedure is a group of instructions that usually performs one task. It is a reusable section of a software program stored in memory once but can be used as often as necessary. A procedure can be of two types. 1**) Near Procedure 2) Far Procedure**

Near Procedure: A procedure is known as NEAR procedure if is written(defined) in the same code segment that is calling that procedure. Only Instruction Pointer(IP register) contents will be changed in NEAR procedure.

FAR procedure : A procedure is known as FAR procedure if it is written (defined) in the different code segment than the calling segment. In this case both Instruction Pointer (IP) and the Code Segment (CS) register content will be changed.

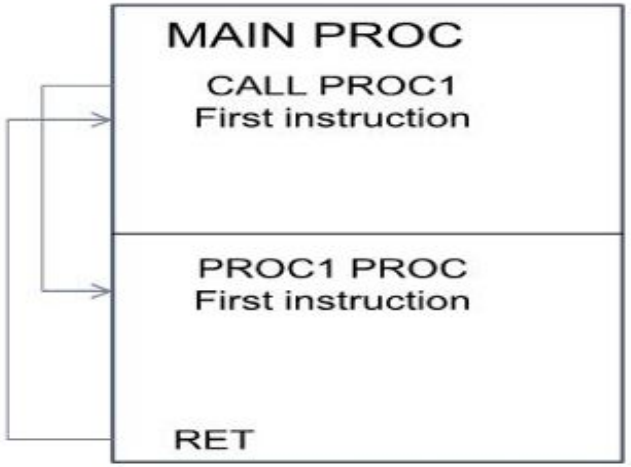
Directives used for procedure :
PROC directive: The PROC directive is used to identify the start of a procedure. The PROC directive follows a name given to the procedure. After that the term FAR and NEAR is used to specify the type of the procedure.
ENDP Directive: This directive is used along with the name of the procedure to indicate the end of a procedure to the assembler. The PROC and ENDP directive are used to bracket a procedure.

Creating Procedures

- Large problems can be divided into smaller tasks to make them more manageable
- A procedure is the assembly equivalent of a Java or C function
- Following is an assembly language procedure named `sample`:

```
sample  PROC
    .
    .
    ret
sample  ENDP
```

Procedure call and return



CALL and RET Instructions

- The CALL instruction calls a procedure
  - pushes offset of next instruction on the stack
  - copies the address of the called procedure into IP (Note: IP=Instruction Pointer)
- The RET instruction returns from a procedure
  - pops top of stack into IP

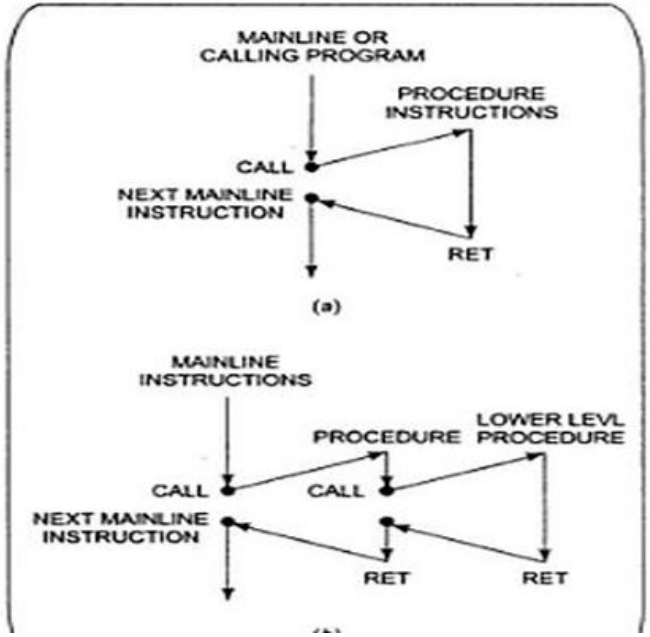
CALL instruction and RET instruction :

CALL instruction : The CALL instruction is used to transfer execution to a procedure. It performs two operation. When it executes, first it stores the address of instruction after the CALL instruction on the stack. Second it changes the content of IP register in case of Near call and changes the content of IP register and CS register in case of FAR call.

There are two types of calls. 1)Near Call or Intra segment call. 2) Far call or Inter Segment call

The CALL and RET instructions

Chart for CALL and RET instruction →



Operation for Near Call : When 8086 executes a near CALL instruction, it decrements the stack pointer by 2 and copies the IP register contents on to the stack. Then it copies address of first instruction of called procedure.

Operation of FAR CALL: When 8086 executes a far call, it decrements the stack pointer by 2 and copies the contents of CS register to the stack. It the decrements the stack pointer by 2 again and copies the content of IP register to the stack. Finally it loads CS register with base address of segment having procedure and IP with address of first instruction in procedure.

Near Call	Far Call
A near call refers a procedure which is in the same code segment.	A Far call refers a procedure which is in different code segment
It is also called Intra-segment call.	It is also called Inter-segment call
A Near Call replaces the old IP with new IP	A FAR replaces CS & IP with new CS & IP.
It uses keyword near for calling procedure.	It uses keyword far for calling procedure.
Less stack locations are required	More stack locations are required.

Procedure	Macro
Procedures are used for large group of instructions to be repeated.	**Procedures are used for small group of instructions to be repeated.**
Object code is generated only once in memory.	**Object code is generated everytime the macro is called.**
CALL & RET instructions are used to call procedure and return from procedure.	**Macro can be called just by writing its name.**
Length of the object file is less	**Object file becomes lengthy.**
Directives PROC & ENDP are used for defining procedure.	**Directives MACRO and ENDM are used for defining MACRO**
More time is required for it's execution	**Less time is required for it's execution**
Procedure can be defined as **Procedure_name PROC** **----** **------** **Procedure_name ENDP**	**Macro can be defined as** MACRO-name MACRO [ARGUMENT ,…… ARGUMENT N] ------ **-------** **ENDM**
For Example **Addition PROC near** **------** **Addition ENDP**	**For Example** **Display MACRO msg** **---------** **ENDM**

8086 INTERRUPTS

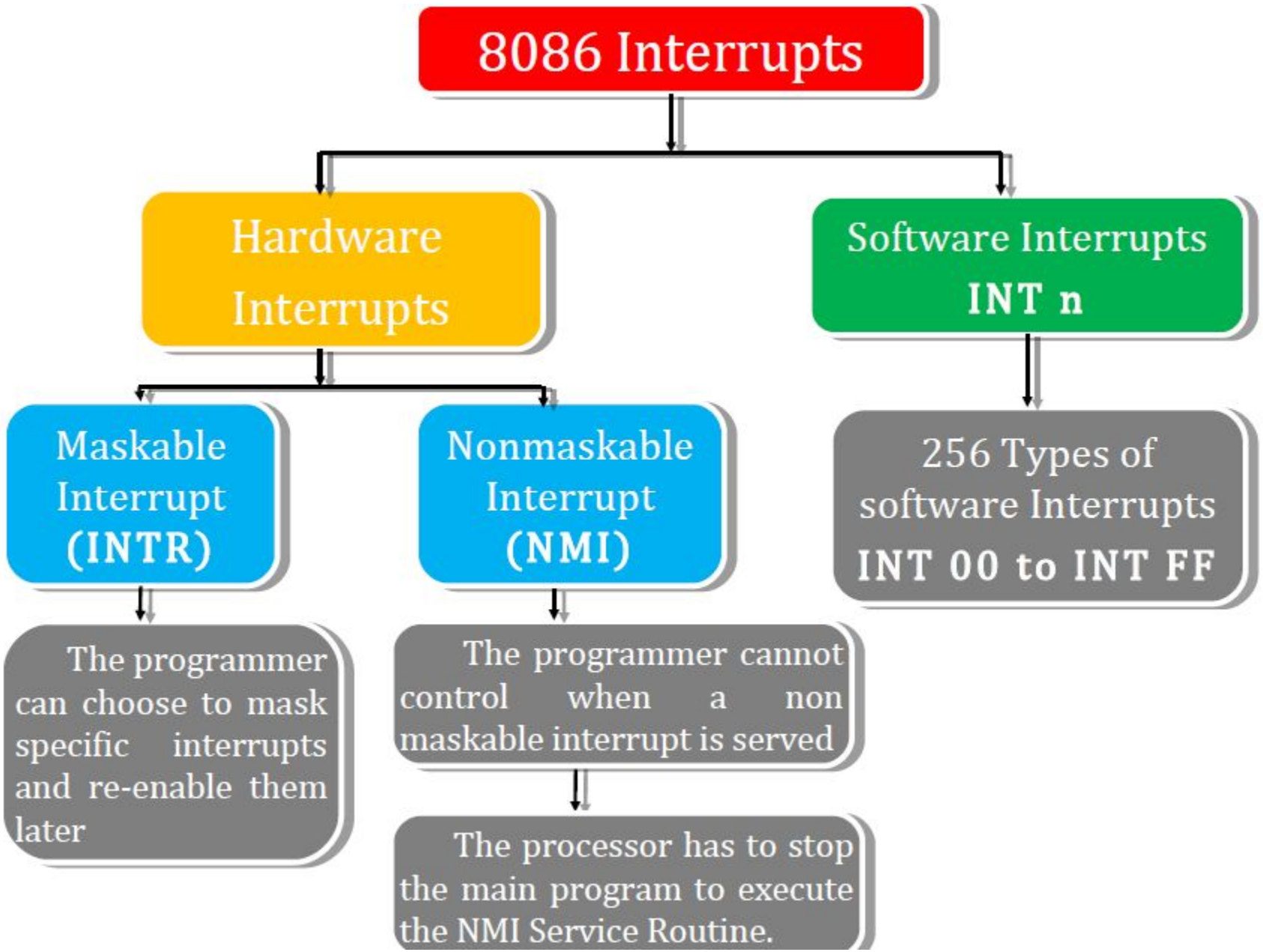
Sources of Interrupts in 8086:

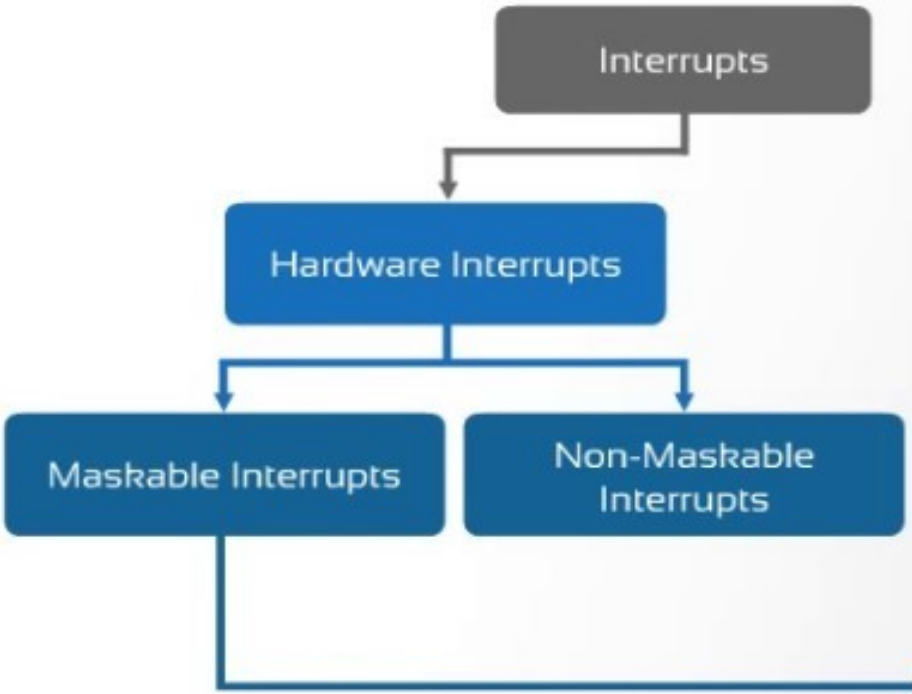
- Three types of interrupts sources are there in 8086:
- 1. An external signal applied to NMI or INTR input pin (Hardware interrupt)

2. Execution of INTn (n=00H-FFH) instruction (Software interrupt)

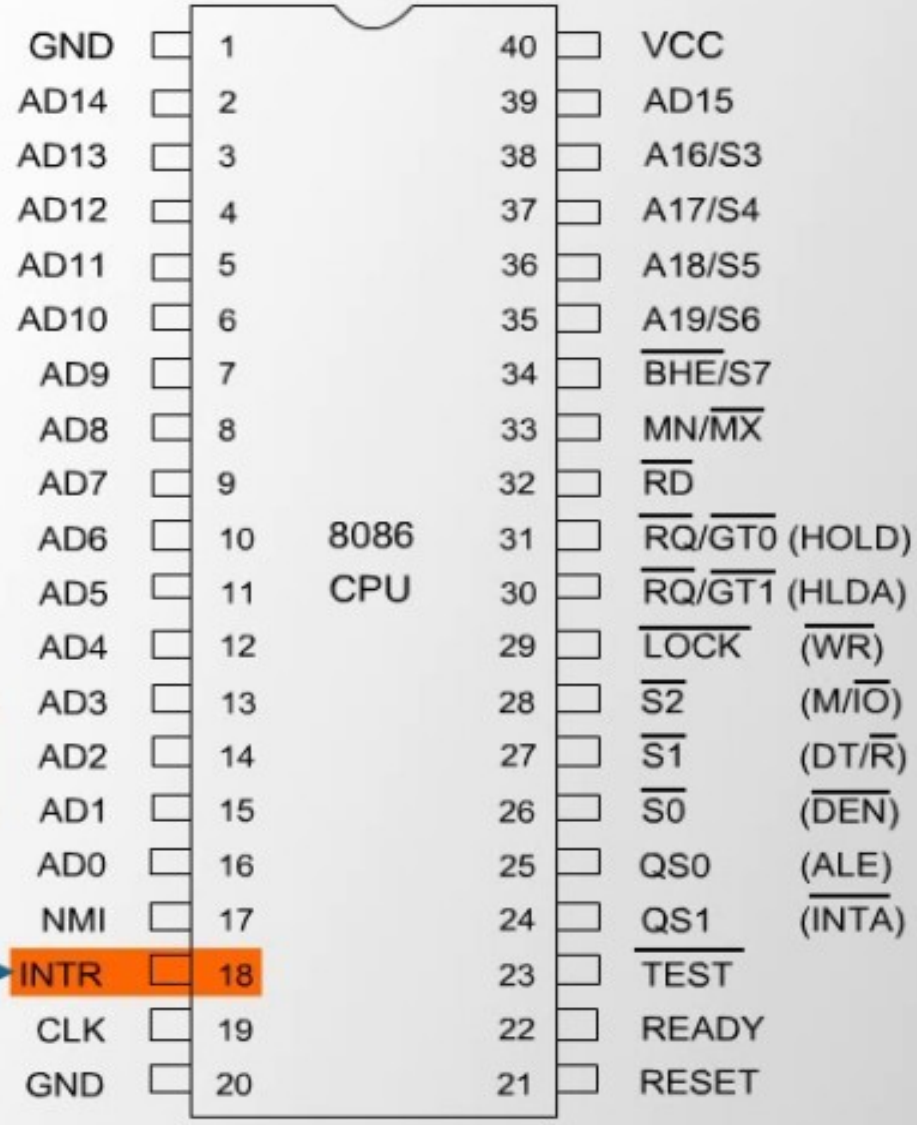
3. Interrupt caused by some error condition produced in 8086 instruction execution process.

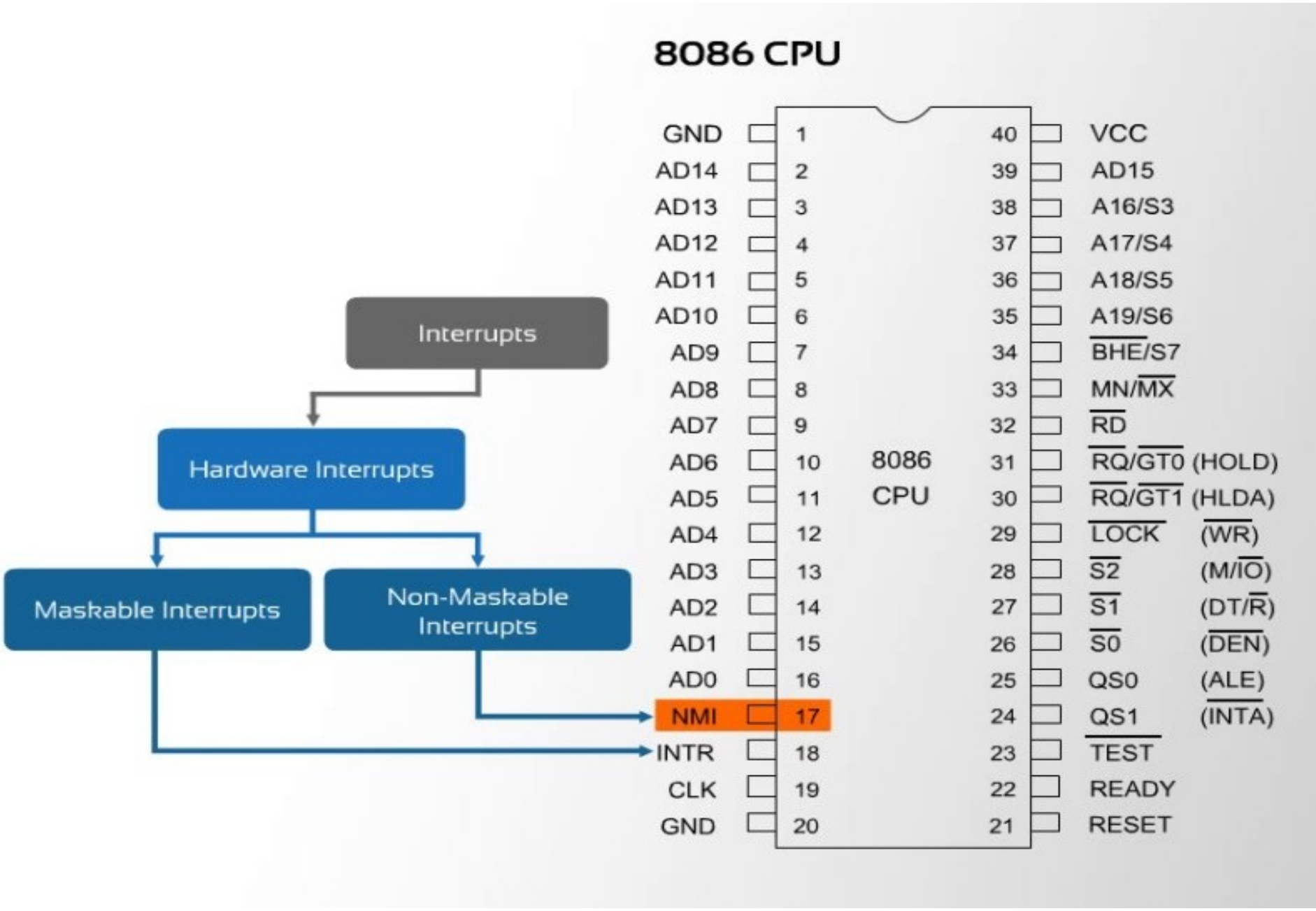
(Divide by zero, overflow errors etc)


8086 Interrupts
Hardware Interrupts
Software Interrupts
INT n
Maskable Interrupt (INTR)
Nonmaskable Interrupt (NMI)
256 Types of software Interrupts
INT 00 to INT FF
The programmer can choose to mask specific interrupts and re-enable them later
The programmer cannot control when a non maskable interrupt is served
The processor has to stop the main program to execute the NMI Service Routine.


Interrupts
Hardware Interrupts
Maskable Interrupts
Non-Maskable Interrupts

8086 CPU

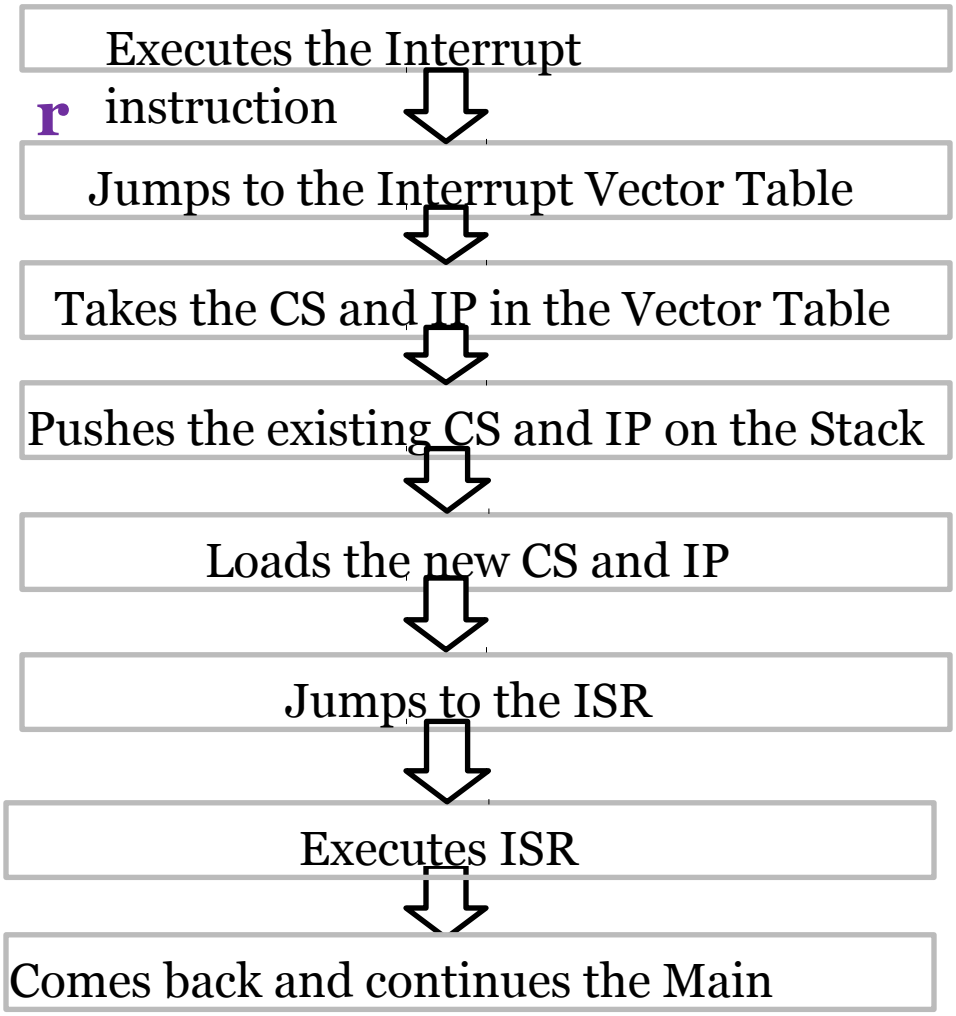

GND 1
AD14 2
AD13 3
AD12 4
AD11 5
AD10 6
AD9 7
AD8 8
AD7 9
AD6 10
AD5 11
AD4 12
AD3 13
AD2 14
AD1 15
AD0 16
NMI 17
INTR 18
CLK 19
GND 20
8086
CPU
40 VCC
39 AD15
38 A16/S3
37 A17/S4
36 A18/S5
35 A19/S6
34 BHE/S7
33 MN/MX
32 RD
31 RQ/GT0 (HOLD)
30 RQ/GT1 (HLDA)
29 LOCK (WR)
28 S2 (M/IO)
27 S1 (DT/R)
26 S0 (DEN)
25 QS0 (ALE)
24 QS1 (INTA)
23 TEST
22 READY
21 RESET


8086 CPU
Interrupts
Hardware Interrupts
Maskable Interrupts
Non-Maskable Interrupts
GND 1
AD14 2
AD13 3
AD12 4
AD11 5
AD10 6
AD9 7
AD8 8
AD7 9
AD6 10
AD5 11
AD4 12
AD3 13
AD2 14
AD1 15
AD0 16
NMI 17
INTR 18
CLK 19
GND 20
8086 CPU
40 VCC
39 AD15
38 A16/S3
37 A17/S4
36 A18/S5
35 A19/S6
34 BHE/S7
33 MN/MX
32 RD
31 RQ/GT0 (HOLD)
30 RQ/GT1 (HLDA)
29 LOCK (WR)
28 S2 (M/IO)
27 S1 (DT/R)
26 S0 (DEN)
25 QS0 (ALE)
24 QS1 (INTA)
23 TEST
22 READY
21 RESET

8086 Interrupt Processing Steps

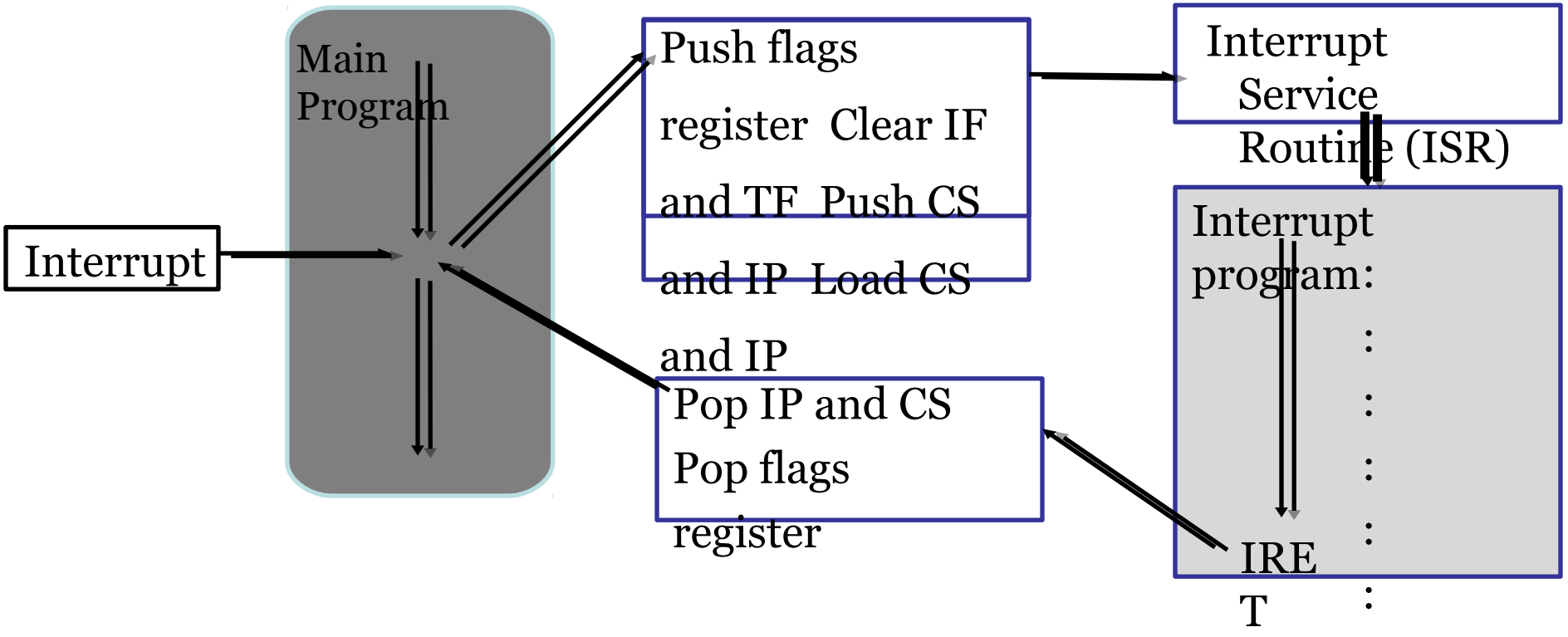
If an interrupt has been requested, the 8086 Microprocessor processes it by performing the following series of steps:

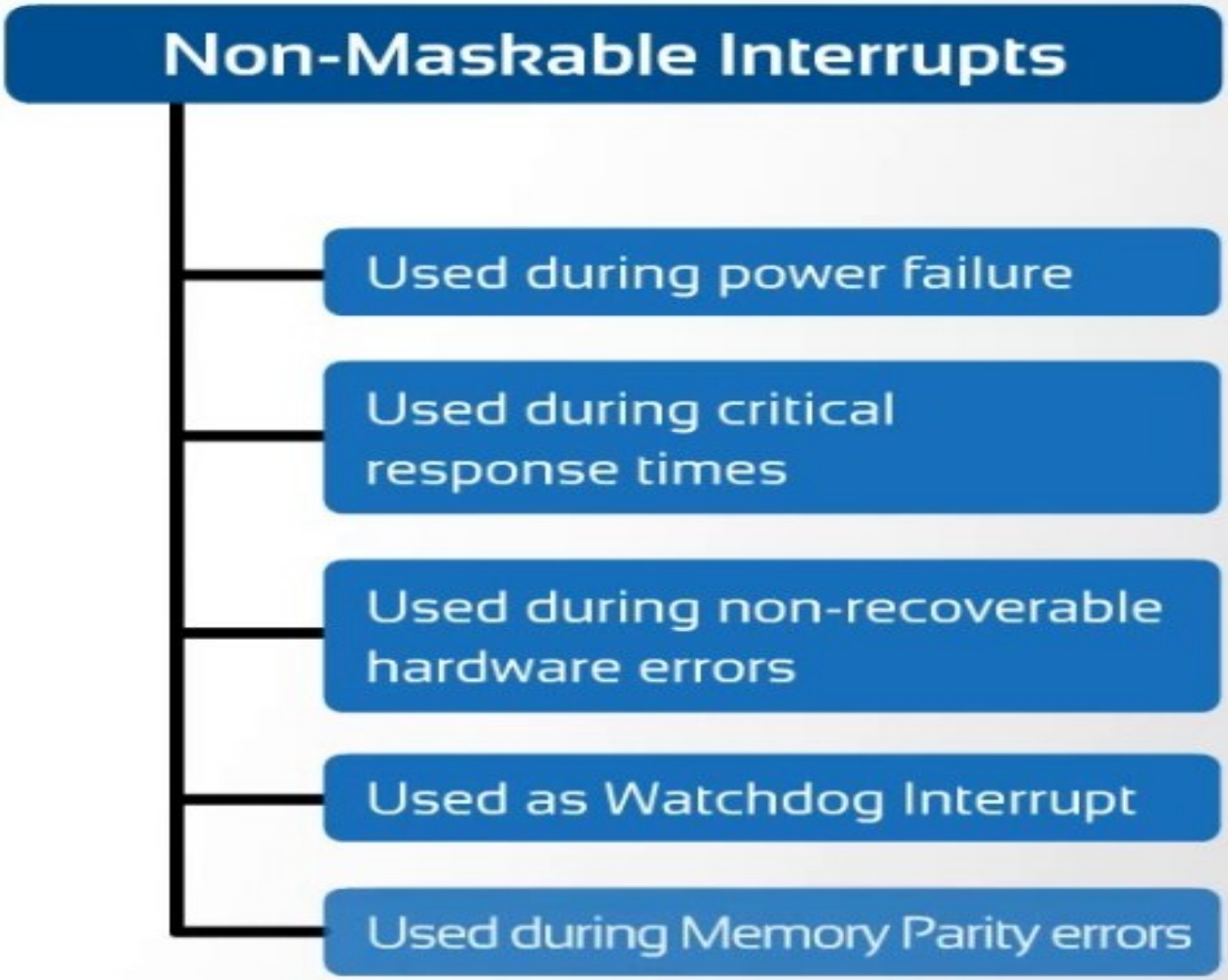
1. Pushes the content of the flag register onto the stack to preserve the status of IF and TF flags, by decrementing the stack pointer (SP) by 2
2. Disables the INTR interrupt by clearing IF in the flag register
3. Resets TF in the flag register, to disable the single step or trap interrupt
4. Pushes the content of the code segment (CS) register onto the stack by decrementing SP by 2
5. Pushes the content of the instruction pointer (IP)onto the stack by decrementing SP by 2
6. Performs an indirect far jump to the start of the interrupt service routine (ISR) corresponding to the received interrupt.



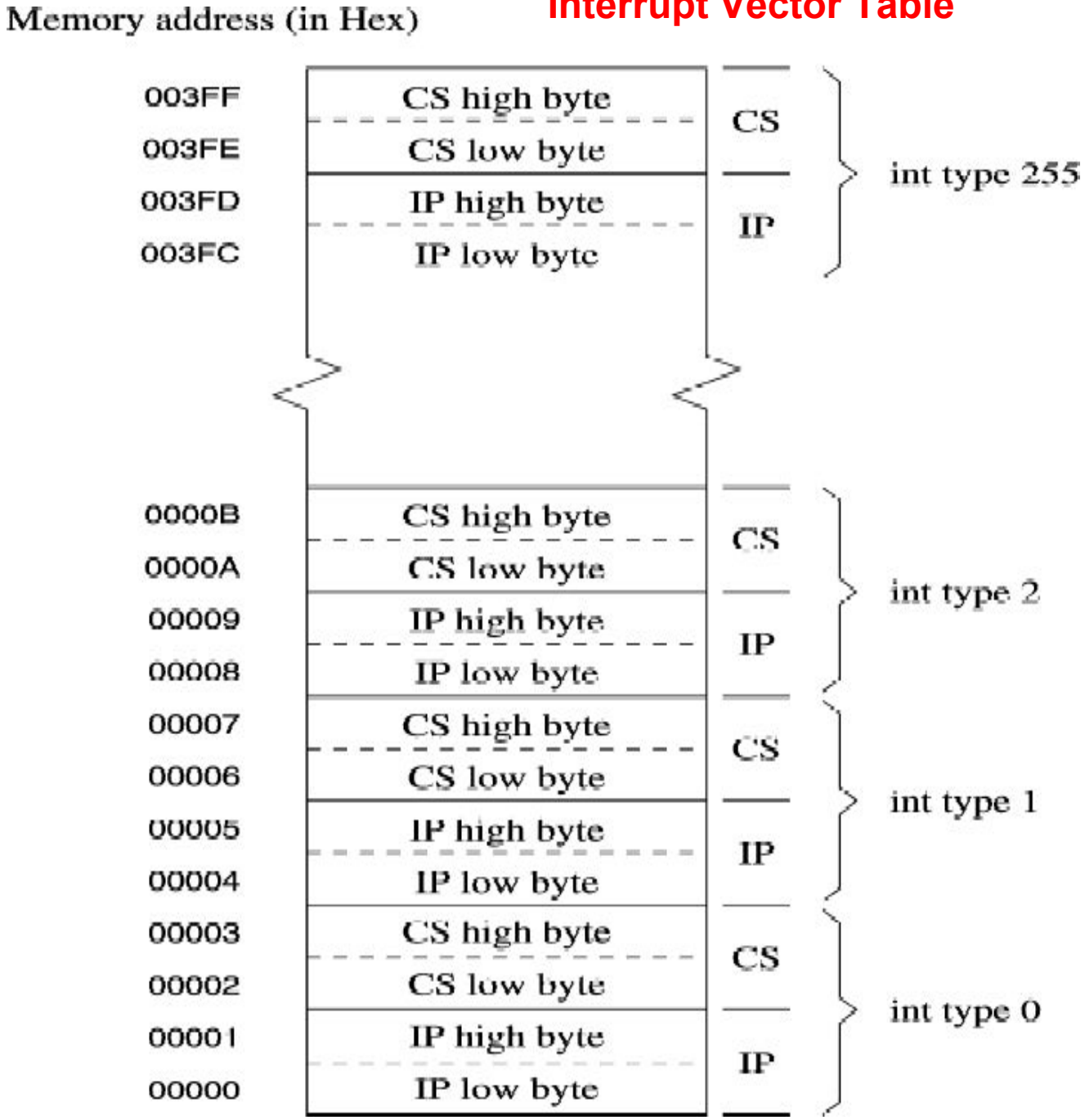
Steps involved in processing an interrupt instruction by the processor

Processing of an Interrupt by the 8086




Non-Maskable Interrupts
Used during power failure
Used during critical response times
Used during non-recoverable hardware errors
Used as Watchdog Interrupt
Used during Memory Parity errors

Interrupt Vector Table



Given a vector, where is the ISR address stored in memory ?

Offset = Type number X 4

Example:- INT 02H

Offset = 02 x 4 = 08
= 00008H

256 Interrupts of 8086 are Divided into 3 Groups

1. **Type 00 to Type 04 interrupts** -
   These are used for fixed operations and hence are called dedicated interrupts

2. **Type 05 to Type 31 interrupts**
   Not used by 8086,reserved for higher processors like 80286 80386 etc.

3. **Type 32 to Type 255 interrupts**
   Available for user, called user defined interrupts. These can be either H/W interrupts and activated through INTR line or can be S/W interrupts.

Type – 0 :- Divide by Zero Error Interrupt

Quotient is large, cant be fit in AL/AX or divide by zero

Type –1:- Single step or Trap Interrupt

Used for executing the program in single step mode by setting trap flag.

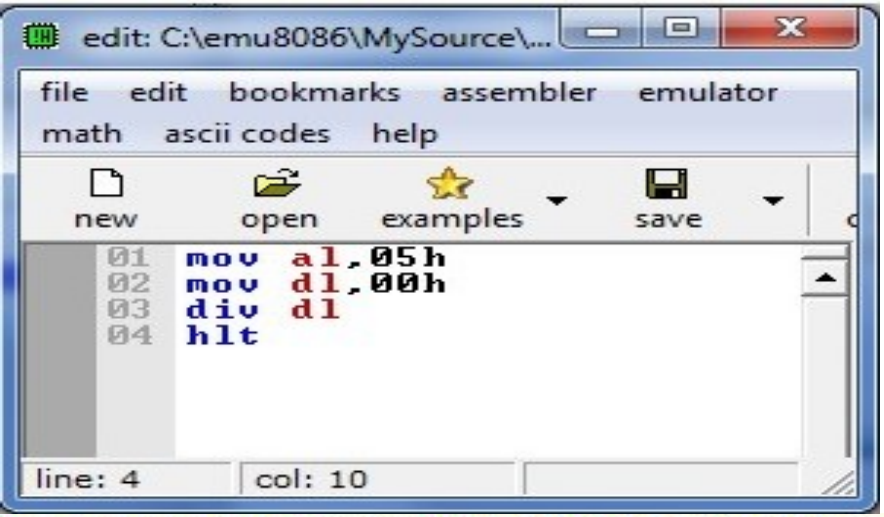
Type – 2:- Non-Maskable Interrupt

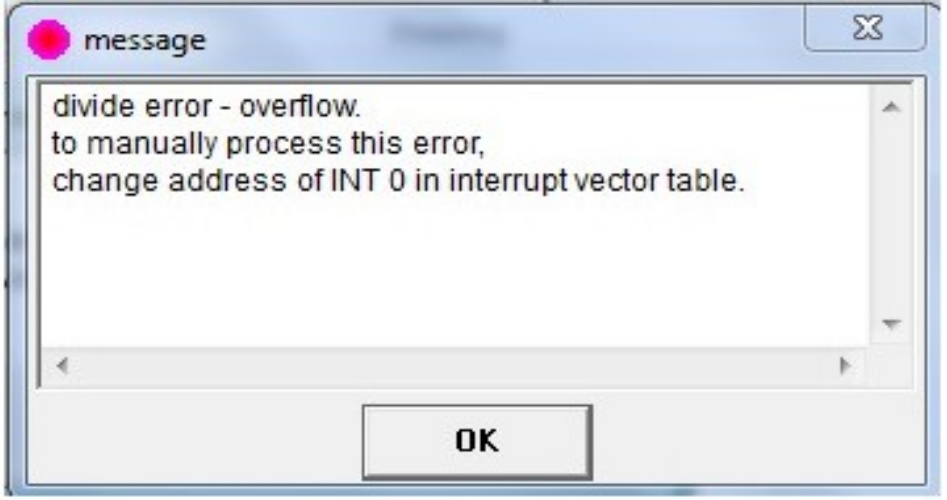
This interrupt is used for executing ISR of NMI pin (positive edge signal), NMI can't be masked by S/W.

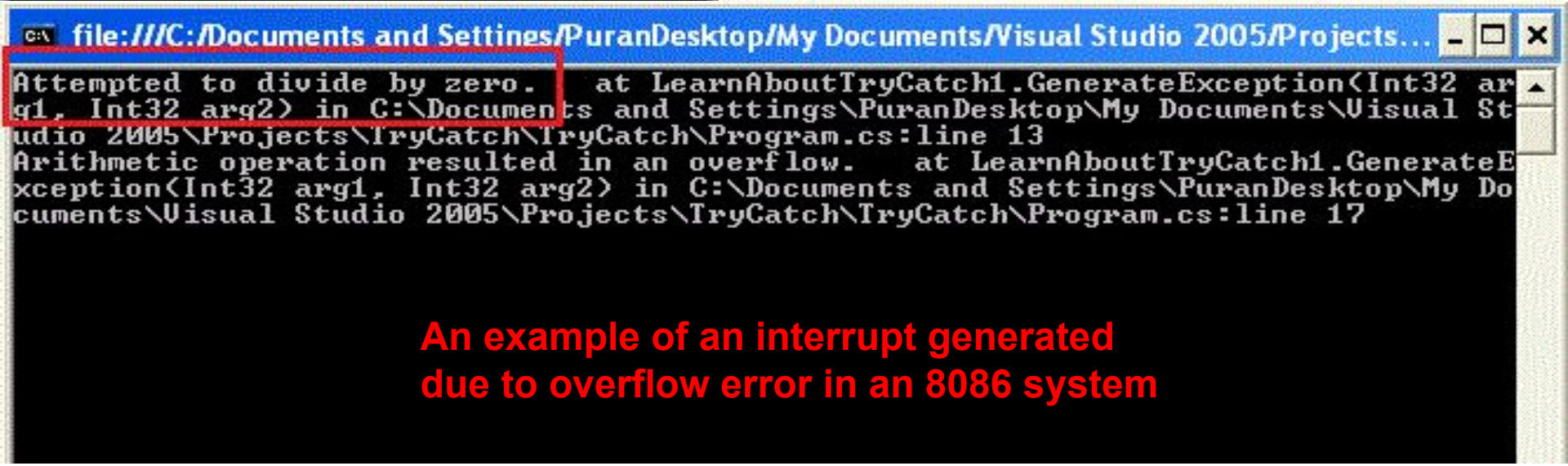
Type – 3:- One-byte INT instruction interrupt

Used for providing break points in the program

Type – 4 Over flow Interrupt







An example of an interrupt generated due to overflow error in an 8086 system

Procedure Example program:

```
ASSUME CS:CODE, DS:DATA, SS:STACK_SEG

DATA SEGMENT
NUM1 DB 50H
NUM2 DB 20H
ADD_RES DB ?
SUB_RES DB ?
DATA ENDS

STACK_SEG SEGMENT

DW 40 DUP(0) ; stack of 40 words, all initialized to zero
TOS LABEL WORD
STACK_SEG ENDS

CODE SEGMENT

START: MOV AX, DATA ; initialize data segment
MOV DS, AX
MOV AX, STACK_SEG ; initialize stack segment
MOV SS, AX
MOV SP, OFFSET TOS ; initialize stack pointer to TOS
CALL ADDITION
CALL SUBTRACTION
MOV AH, 4CH
INT 21H

ADDITION PROC NEAR
MOV AL, NUM1
MOV BL, NUM2
ADD AL, BL
MOV ADD_RES, AL
RET
ADDITION ENDP

SUBTRACTION PROC
MOV AL, NUM1
MOV BL, NUM2
SUB AL, BL
MOV SUB_RES, AL
RET
SUBTRACTION ENDP

CODE ENDS
END START
```

MACRO program Example

```
ASSUME CS:CODE, DS:DATA

DATA SEGMENT
NUM1 DW 1000H
NUM2 DW 2000H
RES DW ?
DATA ENDS

CODE SEGMENT

ADDITION MACRO NO1, NO2, RESULT
MOV AX, NO1
MOV BX, NO2
ADD AX, BX
MOV RESULT, AX
ENDM

START: MOV AX, DATA ; initialize data segment
MOV DS, AX
ADDITION NUM1, NUM2, RES
MOV AH, 4CH
INT 21H

CODE ENDS
END START
```