INTEL 8086 Microprocessor

16/06/20

Basic features :

(i) 8086 is 16 bit, N-channel, HMOS microprocessor

(ii) Its clock frequency is 5 MHz. Other versions have 8MHz & 10MHz clock frequency.

(iii) This is 40 PIN IC Package.

(iv) It has 20 address lines & 16 data lines

(v) Its memory space size is 1 MB (2^{20} = 1 M.Byte)

(vi) The 16 lower order address lines are multiplexed with 16 data lines. and 4 higher order address lines are multiplexed with status signals.

(vii) It has two mode of operations : MINIMUM MODE
MAXIMUM MODE
(MN/$\overline{MX}$)

PIN configuration :

8086 microprocessor has 40 pins. Among them below pins are very essential :

(a) NMI : Non-Maskable Interrupt Request.

(b) INTR : Interrupt request.

(c) $\overline{BHE}/S_7$: Bus high Enable / status 7. During T_1 (of execution cycle) it is low. It is used to enable data onto MSB half of data bus, $D_{15}-D_8$. S_7 signal is available during T_3-T_4.

(d) $\overline{TEST}$: Wait for test control. When it is low the cpu continues execution else waits.

Operating Modes : 8086 microprocessor has two modes. minimum & maximum mode. When only one 8086 CPU is to be used in a micro-computer system the 8086 is used in minimum mode. In multi-processor system it is used in maximum mode. There is MN/$\overline{MX}$ pin at 33 no. exists in 8086 chip. which is used to select-any of the two modes. Pin from 24 to 31 will be use differ-ently for minimum or maximum mode operations.

When 8086 up is operating in Minimum Mode then 24-31 Pins are acting as follows -

Pin 24 → $\overline{INTA}$ – Interrupt Acknowledgement

Pin 25 → ALE – High at T_1. Latching purpose

Pin 26 → $\overline{DEN}$ – Data Enable. When communicating with octal bus transreceiver

Pin 27 → DT/$\overline{R}$ – Data Transmit / $\overline{Receive}$ from 8086 up to other device.

Pin 28 → M/$\overline{IO}$ – Memory related or Input output related operations.

Pin 29 → $\overline{WR}$ – Writing to memory

Pin 30 → HLDA – Hold acknowledgement;

Pin 31 → HOLD – Hold (DMA operations)

When 8086 is operating in Maximum mode then 24-31 Pins are acting as follows →

Pin 24 → Pin 25 →

Instruction Queue Status.

QS_1	QS_0	Ins. Queue Status
0	0	– No operation
0	1	– 1st byte of opcode from queue
1	0	Empty the queue
1	1	Subsequent byte of Queue.

Pin 28-27-~~26~~26 →

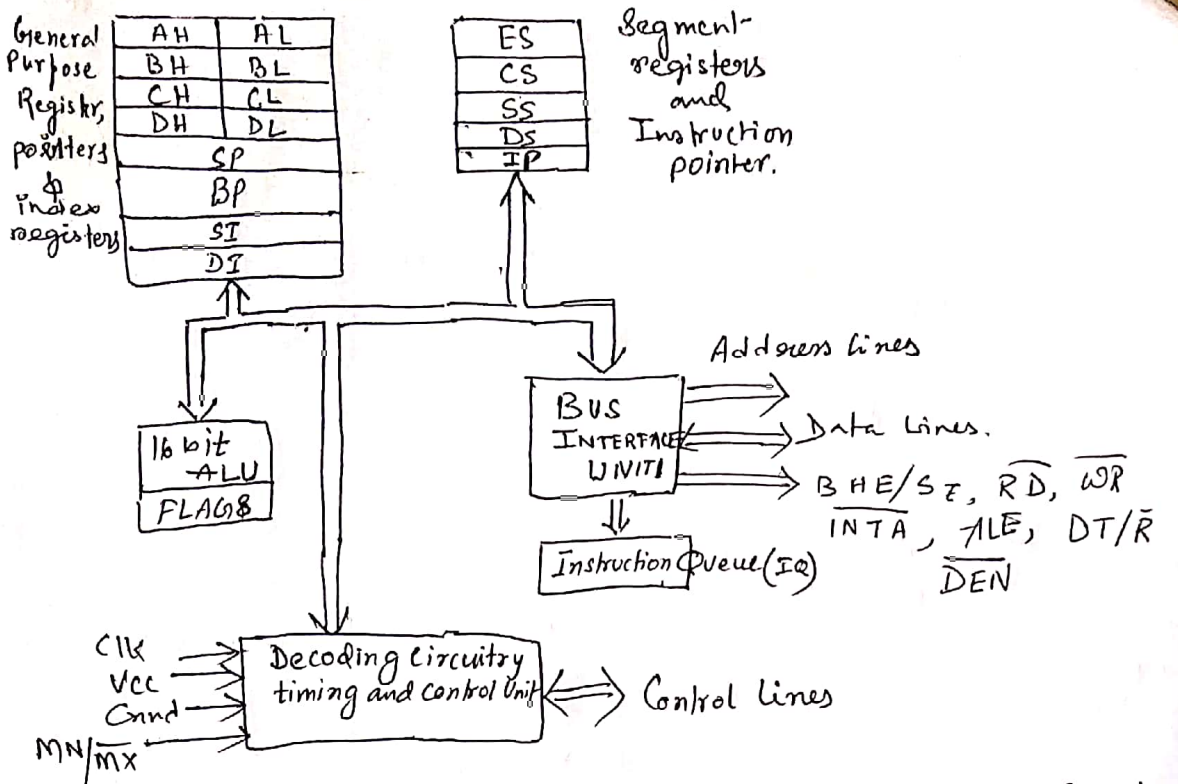
- These signals are connected with bus controller (Intel 8288)
- Bus Controller generate Memory or I/O related control signals.

S_2	S_1	S_0	
0	0	0	→ Interrupt Acknowledge
0	0	1	→ Read from I/O port
0	1	0	→ Write into I/O Port
0	1	1	→ Halt.
1	0	0	→ opcode Fetch
1	0	1	→ Memory Read
1	1	0	→ Memory write
1	1	1	→ Passive state

Pin 29 → $\overline{LOCK}$ → To Lock all operation of 8086 up.

Pin 30-31 → $\overline{RQ}/\overline{GT_1}$ - $\overline{RQ}/\overline{GT_0}$ → used for Controlling Local Bus

Functional Block Diagram of 8086 µp →



8086 contains two independant functional units : Bus Interface Unit (BIU) and an Execution Unit (EU).

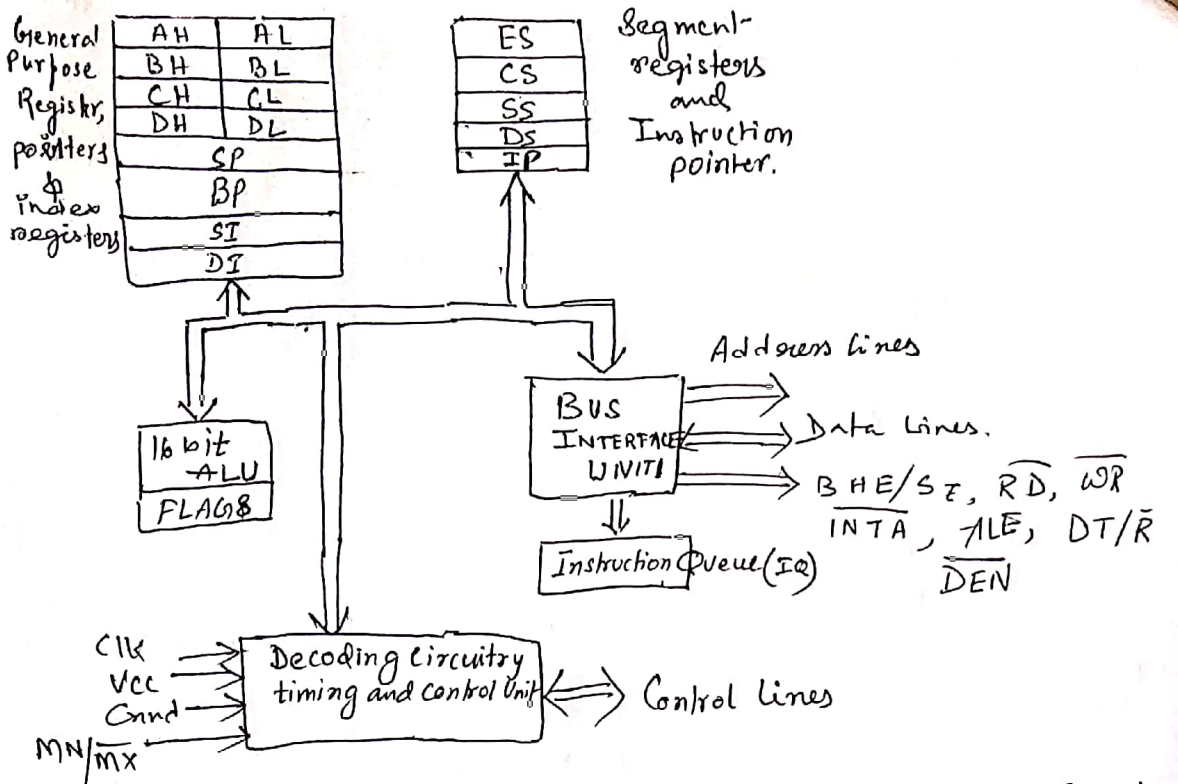
Parts of BIU Unit : The segment registers (ES, CS, DS, SS), instruction pointer (IP) & 6 byte instruction Queue.

Parts of Execution Unit : The general purpose registers, stack pointer, bas pointer & index registers, ALU, flags, timing & control unit, Instruction decoder are the parts of EU.

Function of BIU Unit : It controls transfer of data between microprocessor & I/O Devices or memory. It computes & sends out addresses, fetches instruction codes, stores fetched instruction code into queue (IQ) [FIFO], reads from / writes to memory or I/O Devices.
EU unit tells BIU unit from where to fetch instruction or data.

Function of EU Unit :- EU receives opcode of instruction from instruction queue (IQ), decodes it & executes. While EU is busy with decoding/executing the instruction, the BIU fetches the opcode of instruction from memory & put it into queue. BIU & EU operate parallely & independently. This makes processing faster. This technique is known as PIPELINING TECHNIQUE.

Functional Block Diagram of 8086 µp →



8086 contains two independant functional units : Bus Interface Unit (BIU) and an Execution Unit (EU).

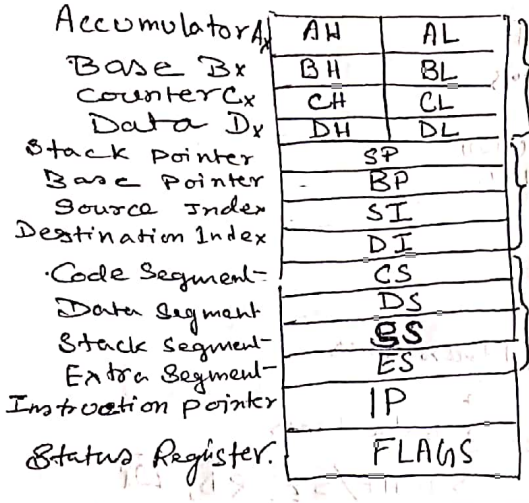
Parts of BIU unit : The segment registers (ES, CS, DS, SS), instruction pointer (IP) & 6 byte instruction Queue.

Parts of Execution Unit : The general purpose registers, stack pointer, bas pointer & index registers, ALU, flags, timing & control unit, Instruction decoder are the parts of EU.

Function of BIU Unit : It controls transfer of data between microprocessor & I/O Devices or memory. It computes & sends out addresses, fetches instruction codes, stores fetched instruction code into queue (IQ) [FIFO], reads/writes data from/to memory or I/O Devices.
EU unit tells BIU unit from where to fetch instruction or data.

Function of EU Unit :- EU receives opcode of instruction from instruction queue (IQ), decodes it & executes. While EU is busy with decoding/executing the instruction, the BIU fetches the opcode of instruction from memory & put it into queue.
BIU & EU operate parallely & independently. This makes processing faster. This technique is known as PIPELINING TECHNIQUE.

Registers of Intel 8086 µp



Register Organization of 8086 µp

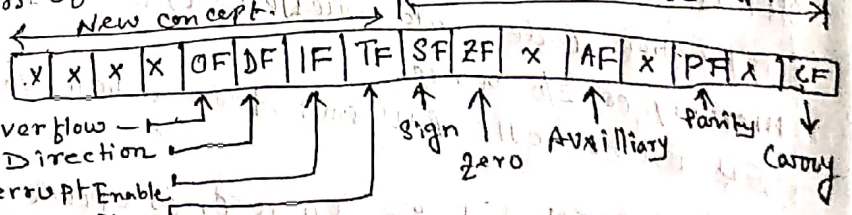
General Purpose register: 4, 16bit general purpose registers are there. (Ax, Bx, Cx, Dx). They are further divided into two parts (Ax = AH + AL). Now, Ax known as – Accumulator, Bx – Base register for computation of memory address. Cx – used as counter in multiiteration instruction. Dx is used to hold data while transfering from I/O port to memory.

Pointer, Index Register: As shown in figure SP, BP, SI, DI are in this group. The function of SP is same as 8085. But other registers are used to calculate memory address.

Segment Register: There are four segment registers CS, DS, SS, ES. In 8086 µp based system memory is divided into following four — Segments. 1. Code segment, 2. Data segment 3. Stack segment 4. Extra Segment. The code segment holds instruction code of program. The data, variables, Constants given in the program are stored in Data segment. The Stack segment is used for stack handling purpose. The extra segment holds the destination address of some data of certain string ~ instructions. Now segment registers hold the starting add. of each segment memory locations.

Instruction (IP) Pointer: It is acting as Program Counter. The function of this register is similar to PC of 8085. The Content of IP and content of CS are used to calculate memory address of instruction to be fetched.

Status Register:



Total 9 Status registers. Among them 6 - Status register
3 - Control register

Control Registers → DF, IF, TF
- DF → used in string operation.
- IF → Enable INTR
- TF → Single step mode

8086 Microprocessor

- It is a 16 bit µp.
- 8086 has a 20 bit address bus can access upto 2^{20} memory locations (1 MB) .
- It can support upto 64K I/O ports.
- It provides 14, 16-bit registers.
- It has multiplexed address and data bus AD_0-AD_{15} and $A_{16} – A_{19}$.

8086 Microprocessor (cont..)

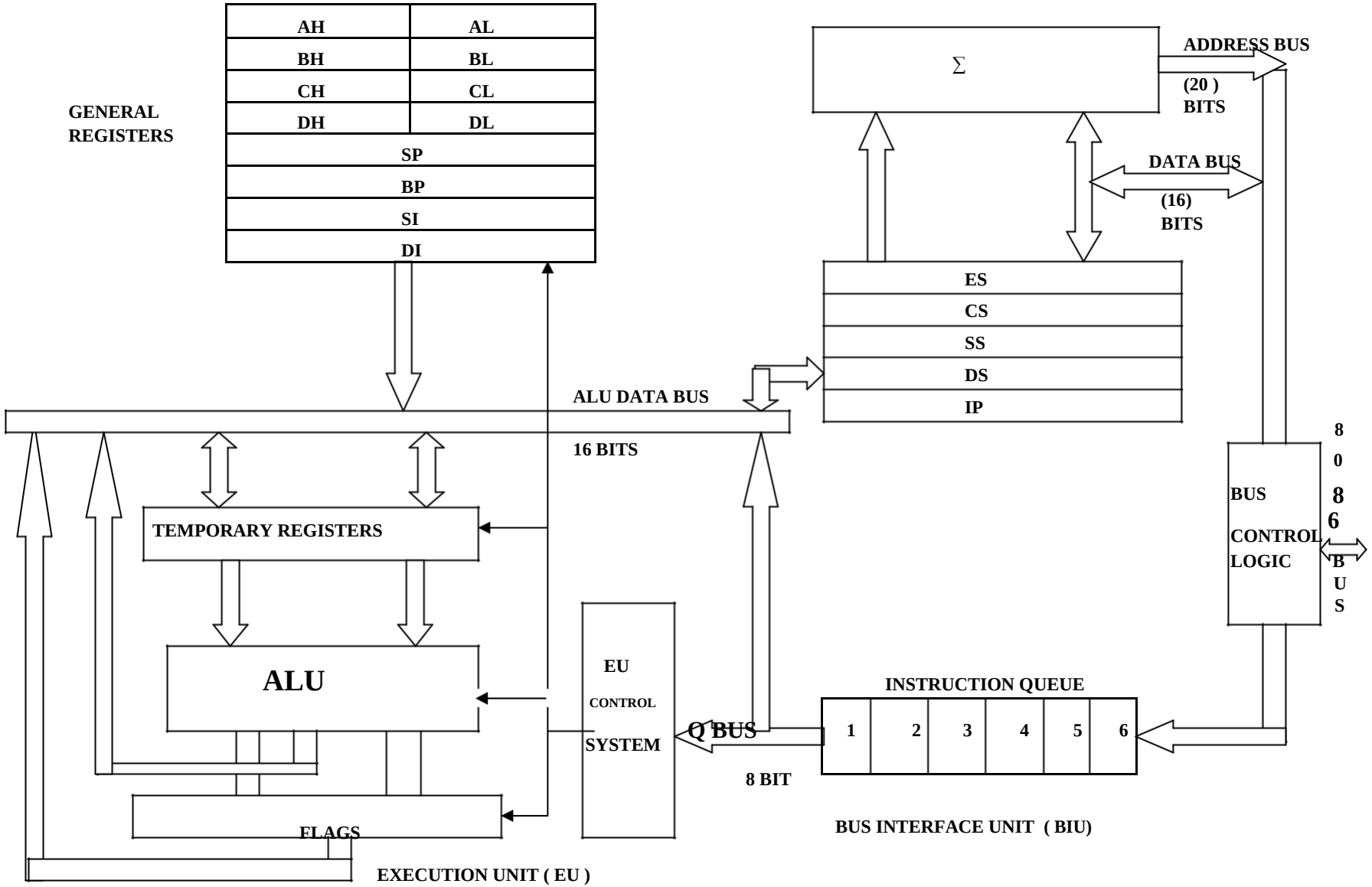
- 8086 is designed to operate in two modes, Minimum and Maximum.
- It can pre-fetches up to 6 instruction bytes from memory and queues them in order to speed up instruction execution.
- It requires +5V power supply.
- A 40 pin dual in line package.

8086 Microprocessor (cont..)

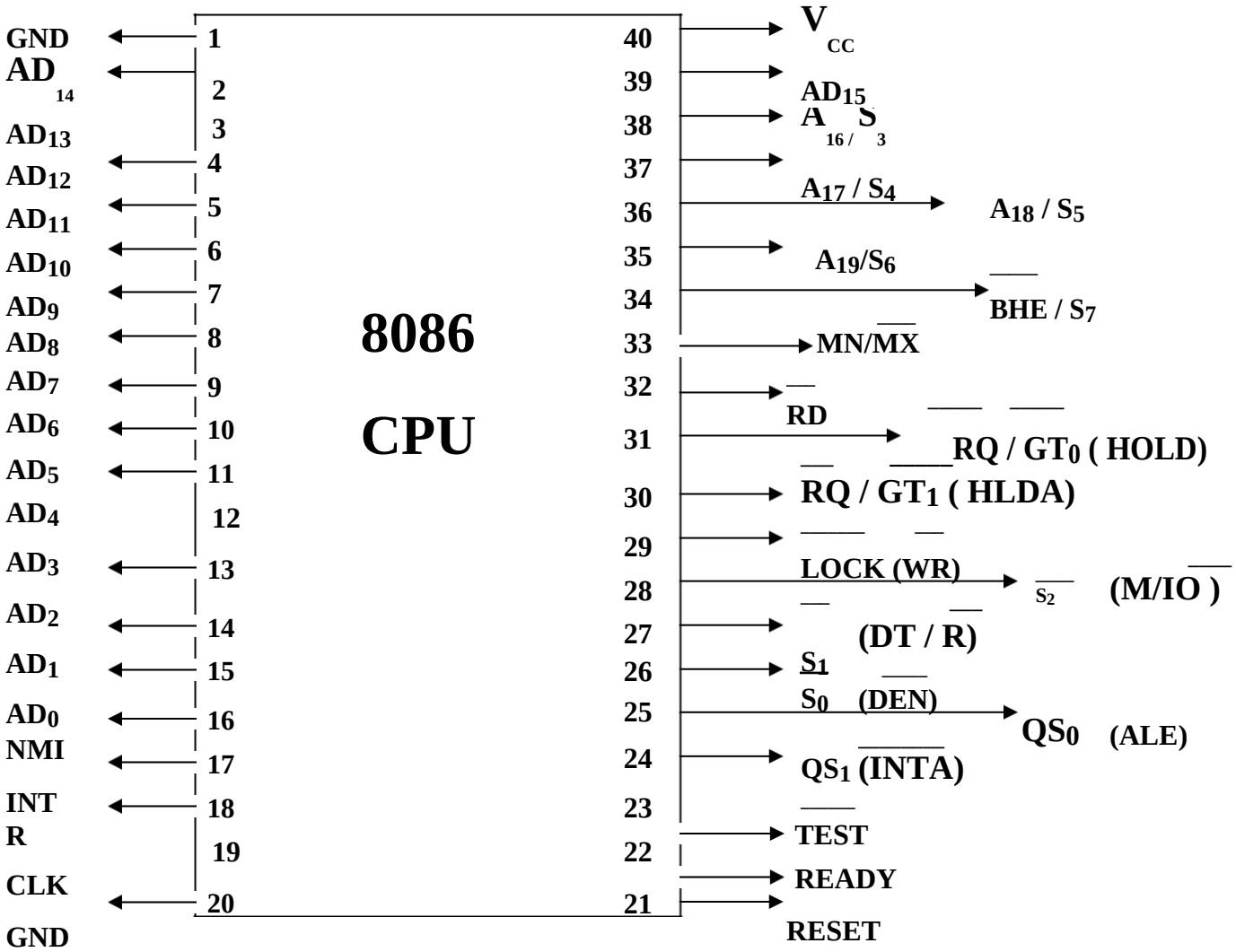
Minimum and Maximum Modes:

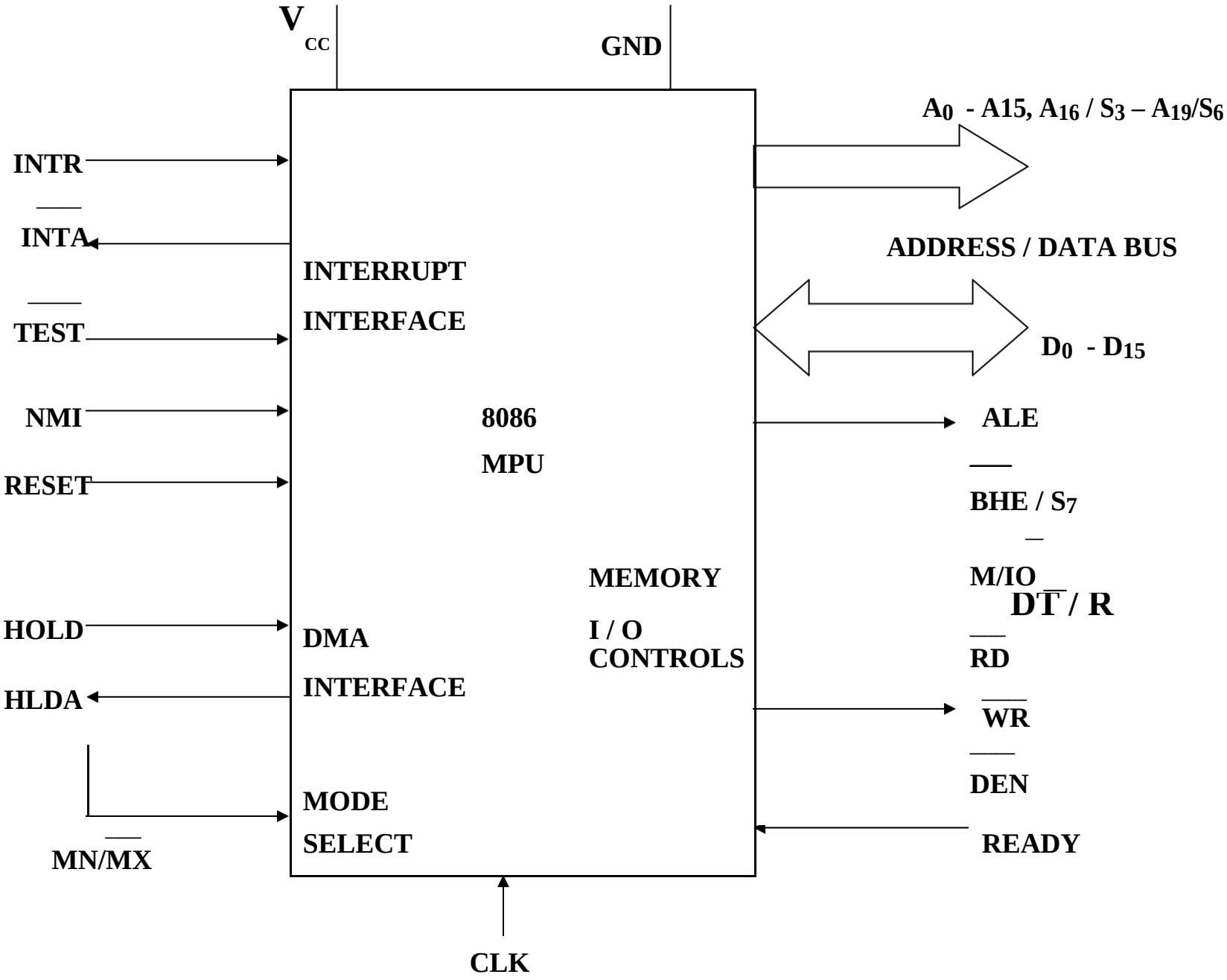
- The minimum mode is selected by applying logic 1 to the MN / MX# input pin. This is a single microprocessor configuration.
- The maximum mode is selected by applying logic 0 to the MN / MX# input pin. This is a multi-microprocessors configuration.

8086 Microprocessor (cont..)



Pin Diagram of 8086




V CC
GND
A0 - A15, A16 / S3 – A19/S6
INTR
INTA
ADDRESS / DATA BUS
INTERRUPT
INTERFACE
TEST
D0 - D15
NMI
8086
MPU
ALE
RESET
BHE / S7
M/IO
MEMORY
I / O
CONTROLS
DT / R
HOLD
DMA
INTERFACE
RD
HLDA
WR
DEN
MODE
SELECT
READY
MN/MX
CLK

Internal Architecture of 8086

- 8086 has two blocks BIU and EU.
- The BIU performs all bus operations such as instruction fetching, reading and writing operands for memory and calculating the addresses of the memory operands.
- The instruction bytes are transferred to the instruction queue.
- EU executes instructions from the instruction system byte queue.

Internal Architecture of 8086 (cont..)

- Both units operate asynchronously to give the 8086 an overlapping instruction fetch and execution mechanism which is called as Pipelining. This results in efficient use of the system bus and system performance.
- BIU contains Instruction queue, Segment registers, Instruction pointer, Address adder.
- EU contains Control circuitry, Instruction decoder, ALU, Pointer and Index register, Flag register.

Internal Architecture of 8086 (cont..)

- Bus Interfacr Unit**:**
- It provides a full 16 bit bidirectional data bus and 20 bit address bus.
- The bus interface unit is responsible for performing all external bus operations.

Specifically it has the following functions:

- Instruction fetch, Instruction queuing, Operand fetch and storage, Address relocation and Bus control.
- The BIU uses a mechanism known as an instruction stream queue to implement a ***pipeline architecture.***

Internal Architecture of 8086 (cont..)

- This queue permits prefetch of up to six bytes of instruction code. When ever the queue of the BIU is not full, it has room for at least two more bytes and at the same time the EU is not requesting it to read or write operands from memory, the BIU is free to look ahead in the program by prefetching the next sequential instruction.
- These prefetching instructions are held in its FIFO queue. With its 16 bit data bus, the BIU fetches two instruction bytes in a single memory cycle.
- After a byte is loaded at the input end of the queue, it automatically shifts up through the FIFO to the empty location nearest the output.

Internal Architecture of 8086 (cont..)

- The EU accesses the queue from the output end. It reads one instruction byte after the other from the output of the queue.
- If the queue is full and the EU is not requesting access to operand in memory, These intervals of no bus activity, which may occur between bus cycles are known as ***Idle state***.
- If the BIU is already in the process of fetching an instruction when the EU request it to read or write operands from memory or I/O, the BIU first completes the instruction fetch bus cycle before initiating the operand read / write cycle.

Internal Architecture of 8086 (cont..)

- The BIU also contains a dedicated adder which is used to generate the 20 bit physical address that is output on the address bus. This address is formed by adding an appended 16 bit segment address and a 16 bit offset address.
- For example, the physical address of the next instruction to be fetched is formed by combining the current contents of the code segment CS register and the current contents of the instruction pointer IP register.
- The BIU is also responsible for generating bus control signals such as those for memory read or write and I/O read or write.

Internal Architecture of 8086 (cont..)

- **EXECUTION UNIT** : The Execution unit is responsible for decoding and executing all instructions.
- The EU extracts instructions from the top of the queue in the BIU, decodes them, generates operands if necessary, passes them to the BIU and requests it to perform the read or write cycles to memory or I/O and perform the operation specified by the instruction on the operands.
- During the execution of the instruction, the EU tests the status and control flags and updates them based on the results of executing the instruction.

Internal Architecture of 8086 (cont..)

- If the queue is empty, the EU waits for the next instruction byte to be fetched and shifted to top of the queue.
- When the EU executes a branch or jump instruction, it transfers control to a location corresponding to another set of sequential instructions.
- Whenever this happens, the BIU automatically resets the queue and then begins to fetch instructions from this new location to refill the queue.

Internal Architecture of 8086 (cont..)

COMMON SIGNALS		
Name	**Function**	**Type**
AD 15– AD 0	**Address/ Data Bus**	**Bidirectional 3- state**
A19 / S6– A16/S3	**Address / Status**	**Output 3- State**
$\overline{BHE}$ / S 7	**Bus High Enable / Status**	**Output 3- State**
MN/MX	**Minimum / Maximum Mode Control**	**Input**
$\overline{RD}$	**Read Control**	**Output 3- State**
TEST	**Wait On Test Control**	**Input**
READY	**Wait State Controls**	**Input**
RESET	**System Reset**	**Input**
NMI	**Non - Maskable Interrupt Request**	**Input**
INTR	**Interrupt Request**	**Input**
CLK	**System Clock**	**Input**
Vcc	**+ 5 V**	**Input**
GND	**Ground**	

Internal Architecture of 8086 (cont..)

Minimum Mode Signals (MN / $\overline{MX}$ = Vcc)		
Name	**Function**	**Type**
HOLD	**Hold Request**	**Input**
HLDA	**Hold Acknowledge**	**Output**
$\overline{WR}$	**Write Control**	**Output, 3- state**
M/$\overline{IO}$	**Memory or IO Control**	**Output, 3-State**
DT/$\overline{R}$	**Data Transmit Receiver /**	**Output, 3- State**
$\overline{DEN}$	**Date Enable**	**Output, 3-State**
ALE	**Address Latch Enable**	**Output**
$\overline{INTA}$	**Interrupt Acknowledge**	**Output**

Internal Architecture of 8086 (cont..)

Maximum mode signals (MN / $\overline{MX}$ = GND)		
Name	**Function**	**Type**
RQ / $\overline{GT1, 0}$	**Request / Grant Bus Access Control**	**Bidirectional**
$\overline{LOCK}$	**Bus Priority Lock Control**	**Output, 3- State**
$\overline{S_2}$– $\overline{S_0}$	**Bus Cycle Status**	**Output, 3- State**
QS1, QS0	**Instruction Queue Status**	**Output**

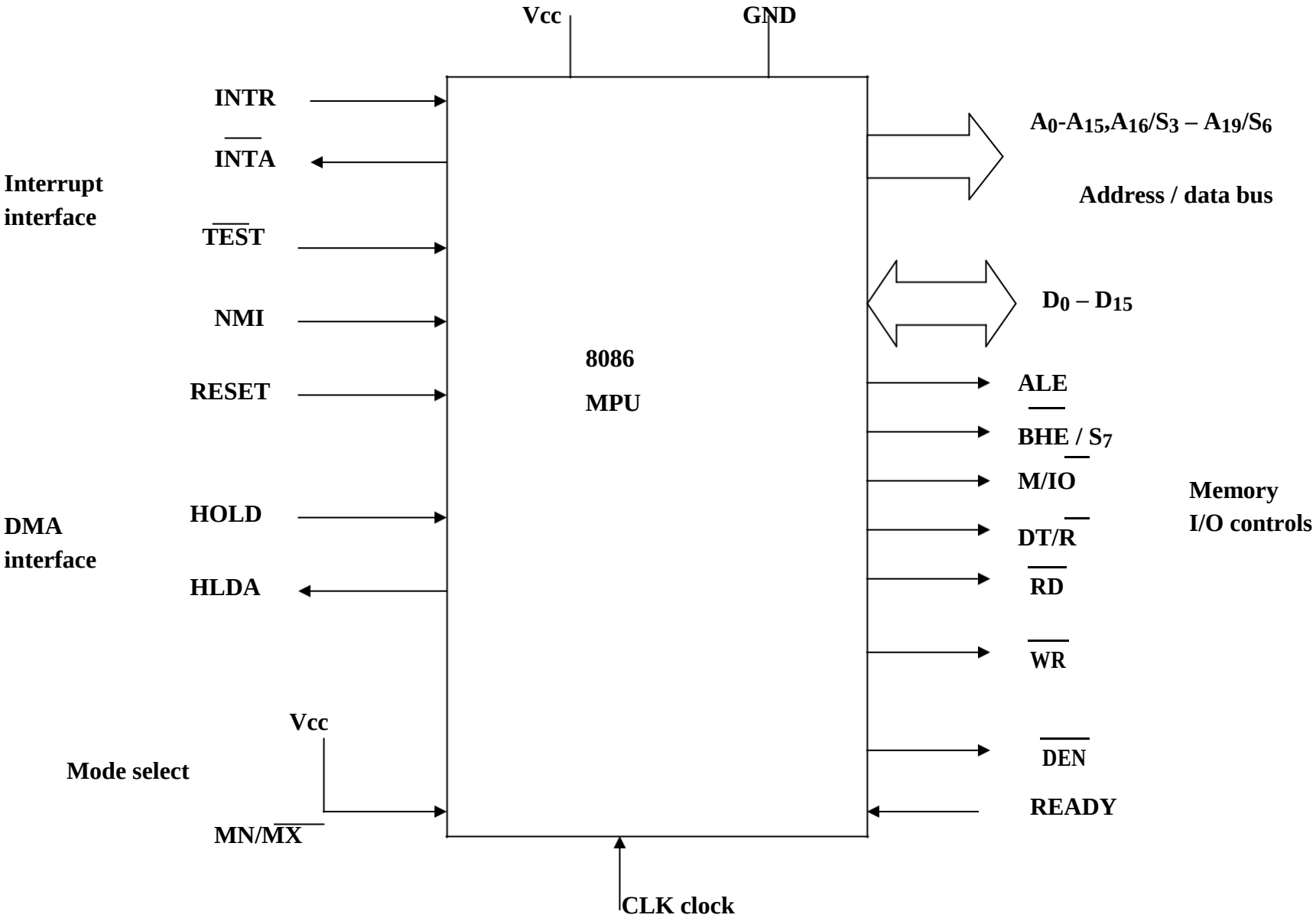
Minimum Mode Interface

- When the Minimum mode operation is selected, the 8086 provides all control signals needed to implement the memory and I/O interface.
- The minimum mode signal can be divided into the following basic groups : address/data bus, status, control, interrupt and DMA.
- **Address/Data Bus** : these lines serve two functions. As an address bus is 20 bits long and consists of signal lines A_0 through A_{19}. A_{19} represents the MSB and A_0 LSB. A 20bit address gives the 8086 a 1Mbyte memory address space.

Minimum Mode Interface (cont..)

- The 16 data bus lines D_0 through D_{15} are actually multiplexed with address lines A_0 through A_{15} respectively. By multiplexed we mean that the bus work as an address bus during first machine cycle and as a data bus during next machine cycles. D_{15} is the MSB and D_0 LSB.
- When acting as a data bus, they carry read/write data for memory, input/output data for I/O devices, and interrupt type codes from an interrupt controller.

Minimum Mode Interface (cont..)



Block Diagram of the Minimum Mode 8086 MPU

Minimum Mode Interface (cont..)

- **Status signal** : The four most significant address lines A_{19} through A_{16} are also multiplexed but in this case with status signals S_6 through S_3. These status bits are output on the bus at the same time that data are transferred over the other bus lines.
- Bit S_4 and S_3 together from a 2 bit binary code that identifies which of the 8086 internal segment registers are used to generate the physical address that was output on the address bus during the current bus cycle.
- Code $S_4S_3 = 00$ identifies a register known as ***extra segment register*** as the source of the segment address.

Minimum Mode Interface (cont..)

S_4	S_3	Segment Register
0	0	Extra
0	1	Stack
1	0	Code
1	1	Data

Memory segment status codes.

Minimum Mode Interface (cont..)

- Status line S_5 reflects the status of another internal characteristic of the 8086. It is the logic level of the internal enable flag. The last status bit S_6 is always at the logic 0 level.
- **Control Signals** : The control signals are provided to support the 8086 memory, I/O interfaces. They control functions such as when the bus is to carry a valid address in which direction data are to be transferred over the bus, when valid write data are on the bus and when to put read data on the system bus.

Minimum Mode Interface (cont..)

- ALE is a pulse to logic 1 that signals external circuitry when a valid address word is on the bus. This address must be latched in external circuitry.
- Another control signal that is produced during the bus cycle is BHE bus high enable. These lines also serves a second function, which is as the S_7 status line.
- Using the M/IO and DT/R lines, the 8086 signals which type of bus cycle is in progress and in which direction data are to be transferred over the bus.

Minimum Mode Interface (cont..)

- The logic level of $\overline{M}$ / IO tells external circuitry whether a memory or I/O transfer is taking place over the bus. Logic 0 at this output signals a memory operation and logic 1 an I/O operation.
- The direction of data transfer over the bus is signaled by the logic level output at DT/R. When this line is logic 1 during the data transfer part of a bus cycle, the bus is in the transmit mode. Therefore, data are either written into memory or output to an I/O device.
- On the other hand, logic 0 at DT/R signals that the bus is in the receive mode. This corresponds to reading data from memory or input of data from an input port.

Minimum Mode Interface (cont..)

- The signal read $\overline{RD}$ and write $\overline{WR}$ indicates that a read bus cycle or a write bus cycle is in progress. The 8086 switches WR to logic 0 to signal external device that valid write or output data are on the bus.
- On the other hand, $\overline{RD}$ indicates that the 8086 is performing a read of data of the bus. During read operations, one other control signal is also supplied. This is DEN (data enable) and it signals external devices when they should put data on the bus.
- There is one other control signal that is involved with the memory and I/O interface. This is the READY signal.

Minimum Mode Interface (cont..)

- READY signal is used to insert wait states into the bus cycle such that it is extended by a number of clock periods. This signal is provided by an external clock generator device and can be supplied by the memory or I/O sub-system to signal the 8086 when they are ready to permit the data transfer to be completed.
- **Interrupt signals** : The key interrupt interface signals are interrupt request (INTR) and interrupt acknowledge (INTA).
- INTR is an input to the 8086 that can be used by an external device to signal that it need to be serviced.

Minimum Mode Interface (cont..)

- Logic 1 at INTR represents an active interrupt request. When an interrupt request has been recognized by the 8086, it indicates this fact to external circuit with pulse to logic 0 at the INTA output.
-

Minimum Mode Interface (cont..)

- There are two more inputs in the interrupt interface: the nonmaskable interrupt NMI and the reset interrupt RESET.
- On the 0-to-1 transition of NMI control is passed to a nonmaskable interrupt service routine. The RESET input is used to provide a hardware reset for the 8086. Switching RESET to logic 0 initializes the internal register of the 8086 and initiates a reset service routine.

Minimum Mode Interface.

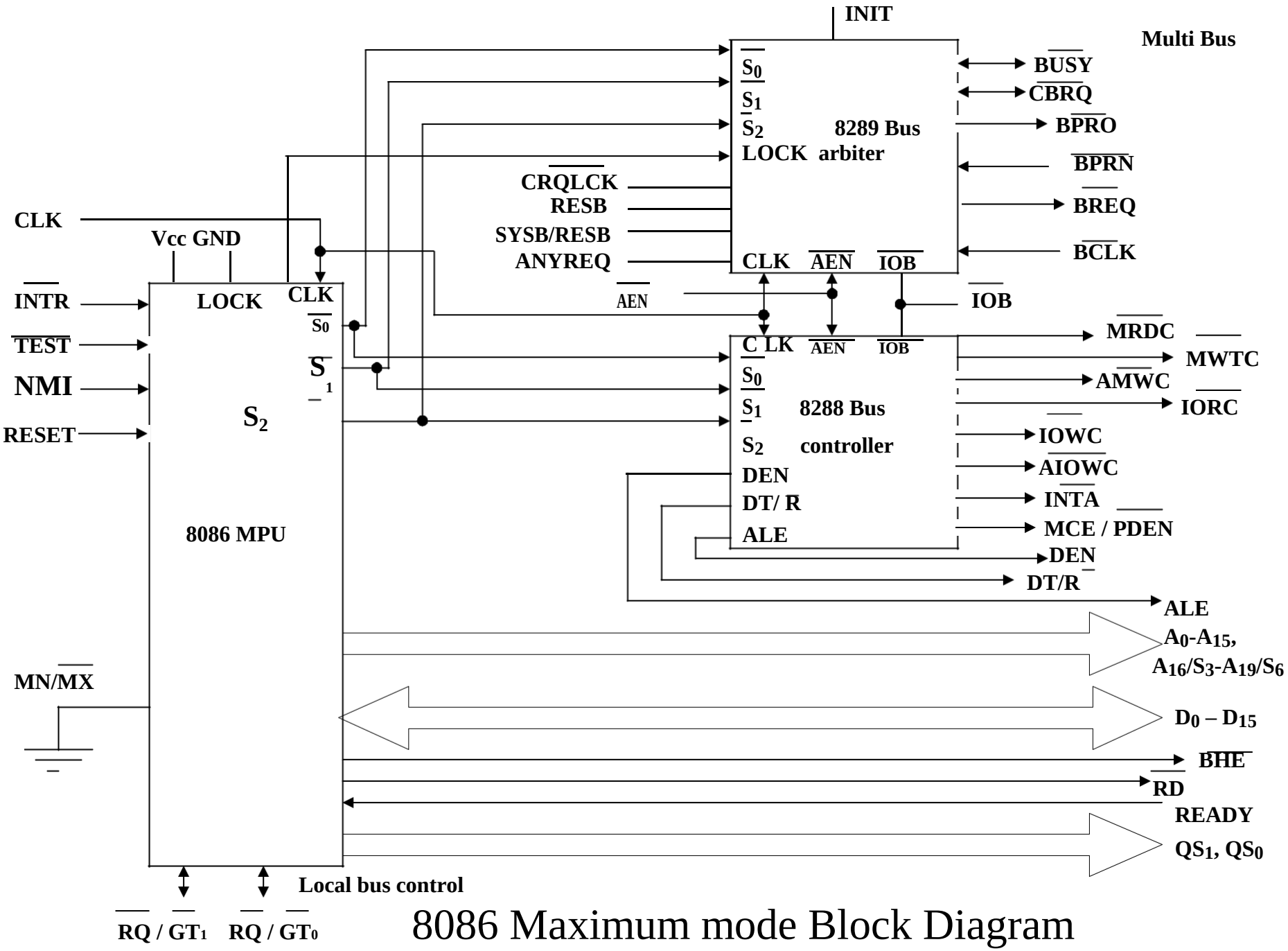
- **DMA Interface signals** :The direct memory access DMA interface of the 8086 minimum mode consist of the HOLD and HLDA signals.
- When an external device wants to take control of the system bus, it signals to the 8086 by switching HOLD to the logic 1 level. The 8086 signals external device that it is in this state by switching its HLDA output to logic 1 level.

Maximum Mode Interface

- When the 8086 is set for the maximum-mode configuration, it provides signals for implementing a multiprocessor / coprocessor system environment.
- Usually in this type of system environment, there are some system resources that are common to all processors.
- They are called as ***global resources***.

Maximum Mode Interface (cont..)

- Coprocessor means that there is a second processor in the system. In this two processor does not access the bus at the same time.
- One passes the control of the system bus to the other and then may suspend its operation.
- In the maximum-mode 8086 system, facilities are provided for implementing allocation of global resources and passing bus control to other microprocessor or coprocessor.



8086 Maximum mode Block Diagram

Maximum Mode Interface (cont..)

- **8288 Bus Controller – Bus Command and Control Signals**: 8086 does not directly provide all the signals that are required to control the memory, I/O and interrupt interfaces.
- Specially the $\overline{WR}$, M/$\overline{IO}$, DT/$\overline{R}$, $\overline{DEN}$, $\overline{ALE}$ and $\overline{INTA}$, signals are no longer produced by the 8086. Instead it outputs three status signals S_0, $\overline{S_1}$, S_2 prior to the initiation of each bus cycle. This 3- bit bus status code identifies which type of bus cycle is to follow.
- $S_2S_1S_0$ are input to the external bus controller device, the bus controller generates the appropriately timed command and control signals.

Maximum Mode Interface (cont..)

Status Inputs			CPU Cycles	8288 Command
$\overline{S}_2$	$\overline{S}_1$	$\overline{S}_0$		
0	0	0	Interrupt Acknowledge	$\overline{INTA}$
0	0	1	Read I/O Port	$\overline{IORC}$
0	1	0	Write I/O Port	$\overline{IOWC}$, $\overline{AIOWC}$
0	1	1	Halt	None
1	0	0	Instruction Fetch	$\overline{MRDC}$
1	0	1	Read Memory	$\overline{MRDC}$
1	1	0	Write Memory	$\overline{MWTC}$, $\overline{AMWC}$
1	1	1	Passive	None

Bus Status Codes

Maximum Mode Interface (cont..)

- The 8288 produces one or two of these eight command signals for each bus cycles. For instance, when the 8086 outputs the code $S_2\overline{S}_1S_0$ equals 001, it indicates that an ***I/O read cycle*** is to be performed.
- In the code 111 is output by the 8086, it is signaling that no bus activity is to take place.
- The control outputs produced by the 8288 are DEN, DT/$\overline{R}$ and ALE. These 3 signals provide the same functions as those described for the minimum system mode. This set of bus commands and control signals is compatible with the Multibus and industry standard for interfacing microprocessor systems.

Maximum Mode Interface (cont..)

- **8289 Bus Arbiter – Bus Arbitration and Lock Signals** : This device permits processors to reside on the system bus. It does this by implementing the Multibus arbitration protocol in an 8086-based system.
- Addition of the 8288 bus controller and 8289 bus arbiter frees a number of the 8086 pins for use to produce control signals that are needed to support multiple processors.
- Bus priority lock ($\overline{LOCK}$) is one of these signals. It is input to the bus arbiter together with status signals $\overline{S}$ S_0 through S_2.

Maximum Mode Interface (cont..)

- ***The output of 8289 are bus arbitration signals***: *bus busy* (BUSY), *common bus request* (CBRQ), *bus priority out* (BPRO), *bus priority in* (BPRN), *bus request* (BREQ) and *bus clock* (BCLK).
- They correspond to the bus exchange signals of the Multibus and are used to lock other processor off the system bus during the execution of an instruction by the 8086.
- In this way the processor can be assured of uninterrupted access to common system resources such as ***global memory.***

Maximum Mode Interface (cont..)

- **Queue Status Signals** : Two new signals that are produced by the 8086 in the maximum-mode system are queue status outputs QS_0 and QS_1. Together they form a 2-bit queue status code, QS_1QS_0.
- Following table shows the four different queue status.

Maximum Mode Interface (cont..)

QS_1	QS_0	Queue Status
0 (low)	0	No Operation. During the last clock cycle, nothing was taken from the queue.
0	1	First Byte. The byte taken from the queue was the first byte of the instruction.
1 (high)	0	Queue Empty. The queue has been reinitialized as a result of the execution of a transfer instruction.
1	1	Subsequent Byte. The byte taken from the queue was a subsequent byte of the instruction.

Queue status codes

Maximum Mode Interface (cont..)

- **Local Bus Control Signal – Request / Grant Signals**: In a maximum mode configuration, the minimum mode HOLD, HLDA interface is also changed. These two are replaced by request/grant lines RQ/ GT_0 and RQ/ GT_1, respectively. They provide a prioritized bus access mechanism for accessing the local bus.

Minimum Mode 8086 System

- In a minimum mode 8086 system, the microprocessor 8086 is operated in minimum mode by strapping its MN/MX pin to logic 1.
- In this mode, all the control signals are given out by the microprocessor chip itself. There is a single microprocessor in the minimum mode system.
- The remaining components in the system are latches, transreceivers, clock generator, memory and I/O devices.

Minimum Mode 8086 System (cont..)

- Latches are generally buffered output D-type flip-flops like 74LS373 or 8282. They are used for separating the valid address from the multiplexed address/data signals and are controlled by the ALE signal generated by 8086.

Minimum Mode 8086 System (cont..)

- The DEN signal indicates the direction of data, i.e. from or to the processor.

Minimum Mode 8086 System (cont..)

- The clock generator generates the clock from the crystal oscillator and then shapes it and divides to make it more precise so that it can be used as an accurate timing reference for the system.
- The clock generator also synchronizes some external signal with the system clock. The general system organisation is as shown in below fig.
- It has 20 address lines and 16 data lines, the 8086 CPU requires three octal address latches and two octal data buffers for the complete address and data separation.

Minimum Mode 8086 System (cont..)

- The working of the minimum mode configuration system can be better described in terms of the timing diagrams rather than qualitatively describing the operations.
- The opcode fetch and read cycles are similar. Hence the timing diagram can be categorized in two parts, the first is the timing diagram for read cycle and the second is the timing diagram for write cycle.
- The read cycle begins in T_1 with the assertion of address latch enable (ALE) signal and also M / IO signal. During the negative going edge of this signal, the valid address is latched on the local bus.

Minimum Mode 8086 System (cont..)

- The $\overline{BHE}$ and $\overline{A_0}$ signals address low, high or both bytes. From T_1 to T_4 , the M/IO signal indicates a memory or I/O operation.
- At T_2, the address is removed from the local bus and is sent to the output. The bus is then tristated. The read (RD) control signal is also activated in T_2.
- The read (RD) signal causes the address device to enable its data bus drivers. After RD goes low, the valid data is available on the data bus.
- The addressed device will drive the READY line high. When the processor returns the read signal to high level, the addressed device will again tristate its bus drivers.

Minimum Mode 8086 System (cont..)

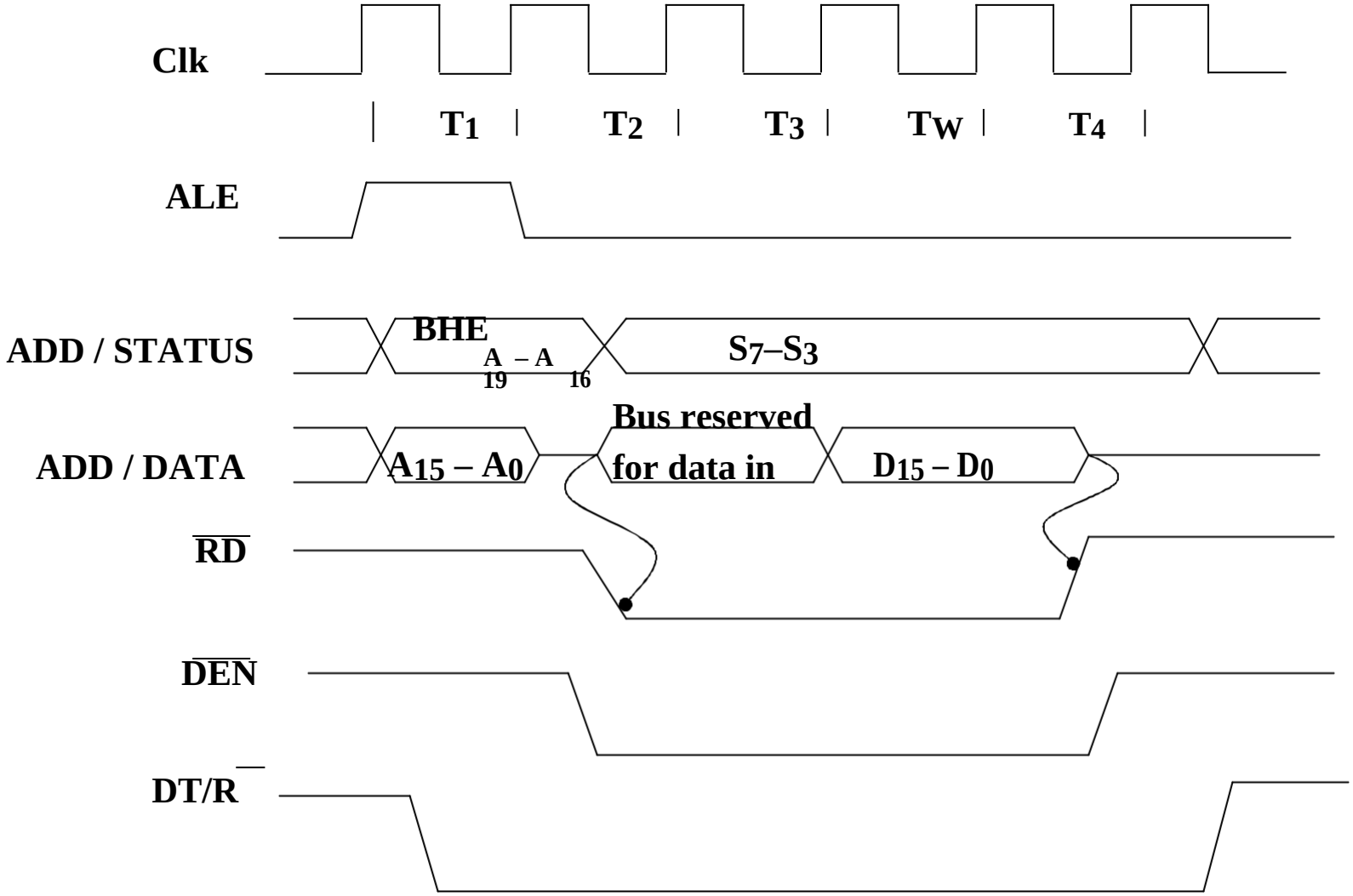
- A write cycle also begins with the assertion of ALE and the emission of the address. The M/$\overline{IO}$ signal is again asserted to indicate a memory or I/O operation. In T_2, after sending the address in T_1, the processor sends the data to be written to the addressed location.
- The data remains on the bus until middle of T_4 state. The $\overline{WR}$ becomes active at the beginning of T_2 (unlike $\overline{RD}$ is somewhat delayed in T_2 to provide time for floating).
- The BHE and A_0 signals are used to select the proper byte or bytes of memory or I/O word to be read or write.
- The M/$\overline{IO}$, $\overline{RD}$ and $\overline{WR}$ signals indicate the type of data transfer as specified in table below.

Minimum Mode 8086 System (cont..)

M /$\overline{IO}$	$\overline{RD}$	$\overline{WR}$	Transfer Type
0	0	1	I / O read
0	1	0	I/O write
1	0	1	Memory read
1	1	0	Memory write

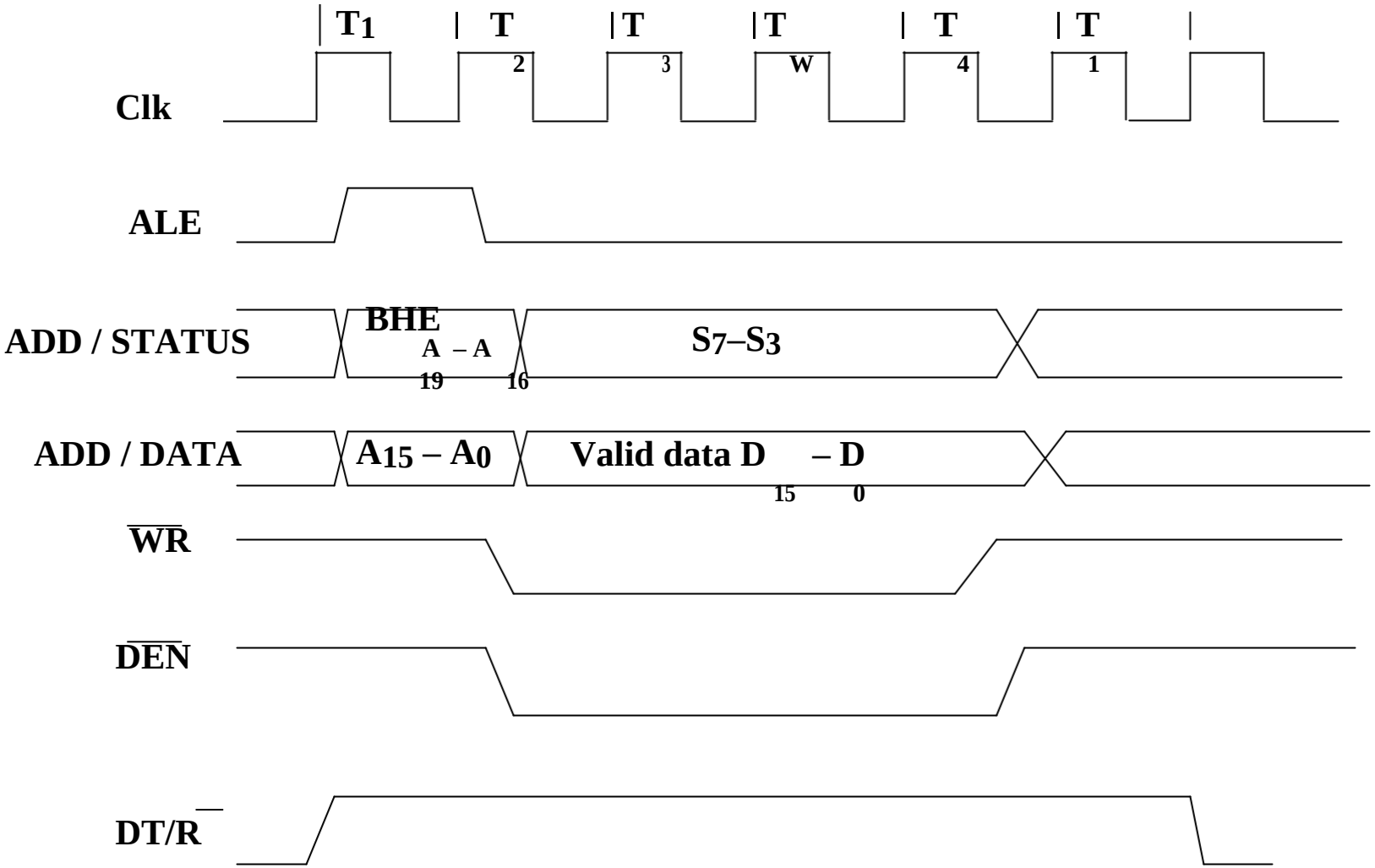
Data Transfer table

Minimum Mode 8086 System (cont..)



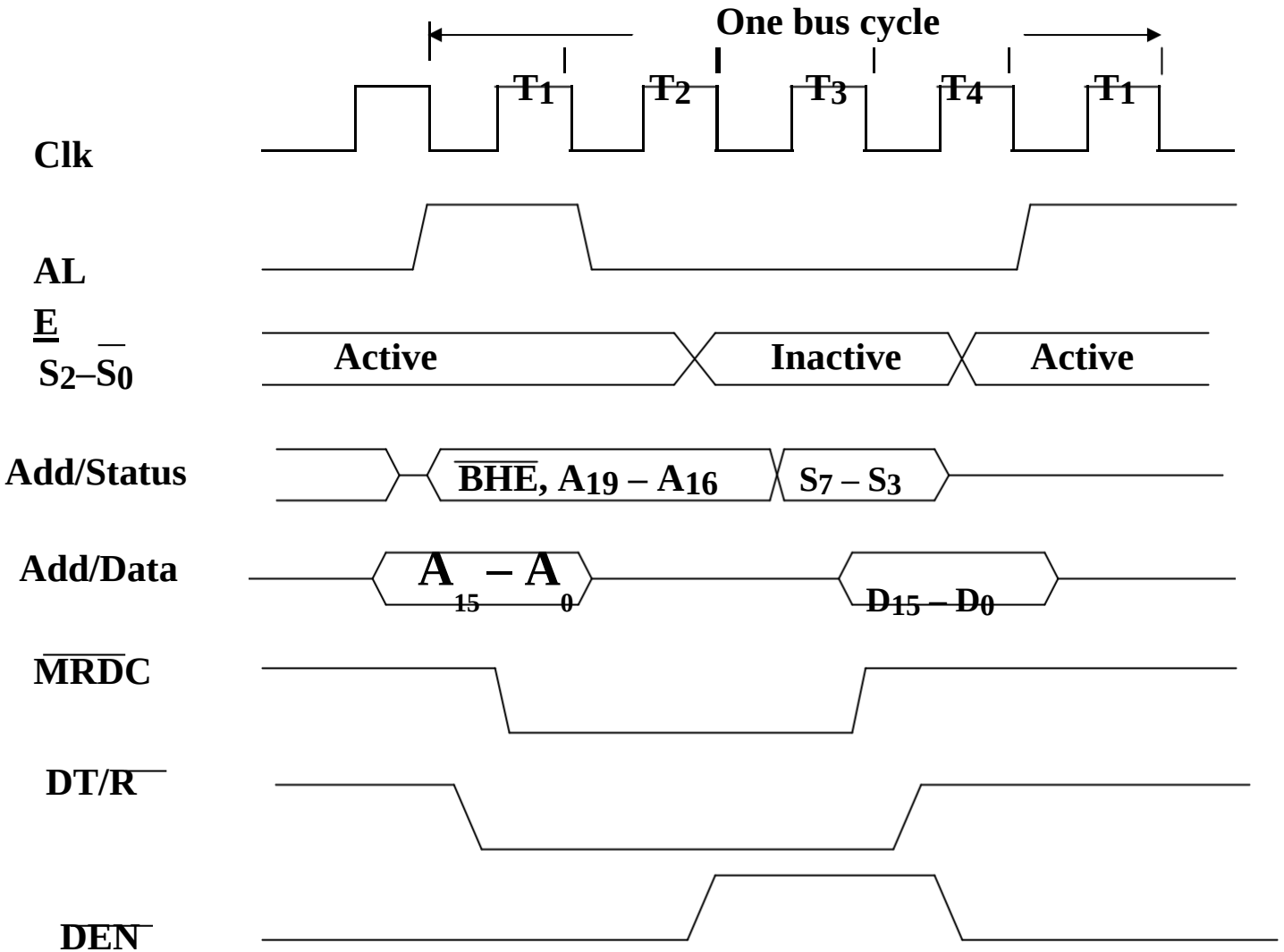
Read Cycle Timing Diagram for Minimum Mode

Minimum Mode 8086 System (cont..)



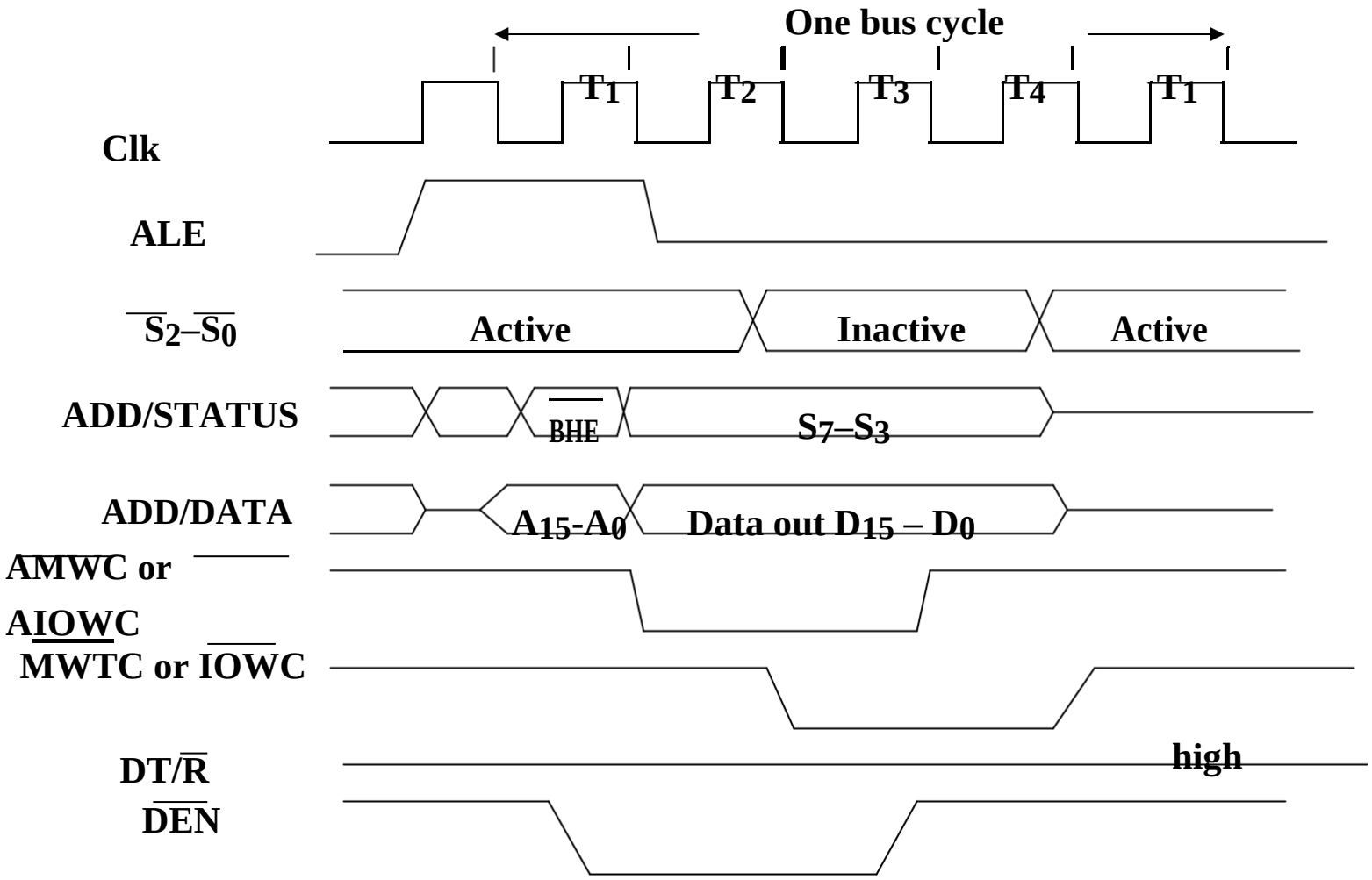
Write Cycle Timing Diagram for Minimum Mode

Maximum Mode 8086 System (cont..)



Memory Read Timing in Maximum Mode

Maximum Mode 8086 System (cont..)



Memory Write Timing in Maximum mode.

Internal Registers of 8086 (cont..)

- The 8086 has four groups of the user accessible internal registers. They are the instruction pointer, four data registers, four pointer and index register, four segment registers.
- The 8086 has a total of fourteen 16-bit registers including a 16 bit register called the ***status register***, with 9 of bits implemented for status and control flags.

- Most of the registers contain data/instruction offsets within 64 KB memory segment. There are four different 64 KB segments for instructions, stack, data and extra data. To specify where in 1 MB of processor memory these 4 segments are located the processor uses four segment registers:
- **Code segment** (CS) is a 16-bit register containing address of 64 KB segment with processor instructions. The processor uses CS segment for all accesses to instructions referenced by instruction pointer (IP) register. CS register cannot be changed directly. The CS register is automatically updated during far jump, far call and far return instructions.

- **Stack segment** (SS) is a 16-bit register containing address of 64KB segment with program stack. By default, the processor assumes that all data referenced by the stack pointer (SP) and base pointer (BP) registers is located in the stack segment. SS register can be changed directly using POP instruction.
- **Data segment** (DS) is a 16-bit register containing address of 64KB segment with program data. By default, the processor assumes that all data referenced by general registers (AX, BX, CX, DX) and index register (SI, DI) is located in the data segment. DS register can be changed directly using POP and LDS instructions.

- **Extra segment** (ES) is a 16-bit register containing address of 64KB segment, usually with program data. By default, the processor assumes that the DI register references the ES segment in string manipulation instructions. ES register can be changed directly using POP and LES instructions.
- It is possible to change default segments used by general and index registers by prefixing instructions with a CS, SS, DS or ES prefix.
- All general registers of the 8086 microprocessor can be used for arithmetic and logic operations. The general registers are:

- **Accumulator** register 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. Accumulator can be used for I/O operations and string manipulation.
- **Base** register 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. BX register usually contains a data pointer used for based, based indexed or register indirect addressing.

- **Count** register 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. Count register can be used in Loop, shift/rotate instructions and as a counter in string manipulation,.
- **Data** register 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. Data register can be used as a port number in I/O operations. In integer 32-bit multiply and divide instruction the DX register contains high-order word of the initial or resulting number.

- The following registers are both general and index registers:
- **Stack Pointer** (SP) is a 16-bit register pointing to program stack.
- **Base Pointer** (BP) is a 16-bit register pointing to data in stack segment. BP register is usually used for based, based indexed or register indirect addressing.
- **Source Index** (SI) is a 16-bit register. SI is used for indexed, based indexed and register indirect addressing, as well as a source data address in string manipulation instructions.

- **Destination Index** (DI) is a 16-bit register. DI is used for indexed, based indexed and register indirect addressing, as well as a destination data address in string manipulation instructions.

Other registers:

- **Instruction Pointer** (IP) is a 16-bit register.
- **Flags** is a 16-bit register containing 9 one bit flags.
- **Overflow Flag** (OF) - set if the result is too large positive number, or is too small negative number to fit into destination operand.

- **Direction Flag** (DF) - if set then string manipulation instructions will auto-decrement index registers. If cleared then the index registers will be auto-incremented.
- **Interrupt-enable Flag** (IF) - setting this bit enables maskable interrupts.
- **Single-step Flag** (TF) - if set then single-step interrupt will occur after the next instruction.
- **Sign Flag** (SF) - set if the most significant bit of the result is set.
- **Zero Flag** (ZF) - set if the result is zero.

8086 Microprocessor

Architecture

Registers and Special Functions

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

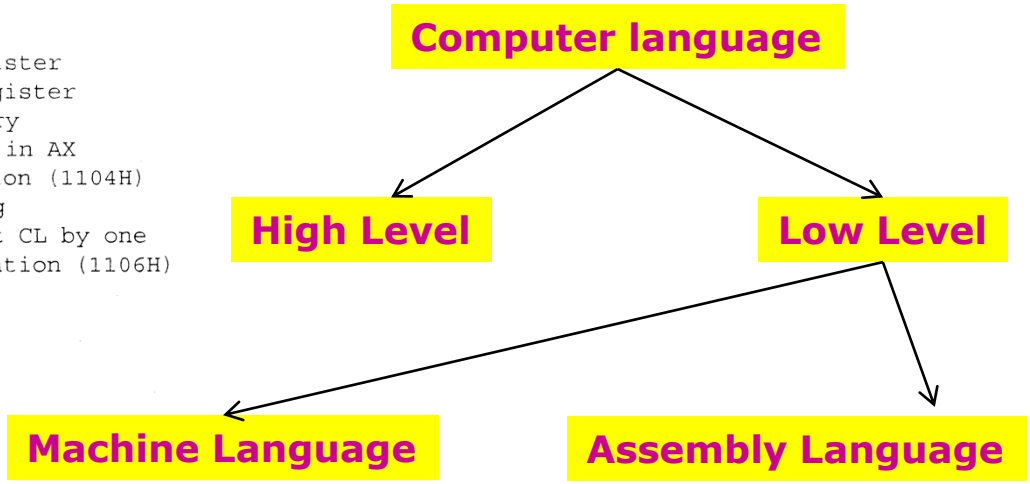
ADDRESSING MODES
&
Instruction set

Introduction

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

DATA SEGMENT              ;Assembler directive

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

DATA ENDS                 ;Assembler directive

CODE SEGMENT              ;Assembler directive

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

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

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

Program
A set of instructions written to solve a problem.

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



- **Binary bits**

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

ADDRESSING MODES

Addressing Modes

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

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

Addressing Modes

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

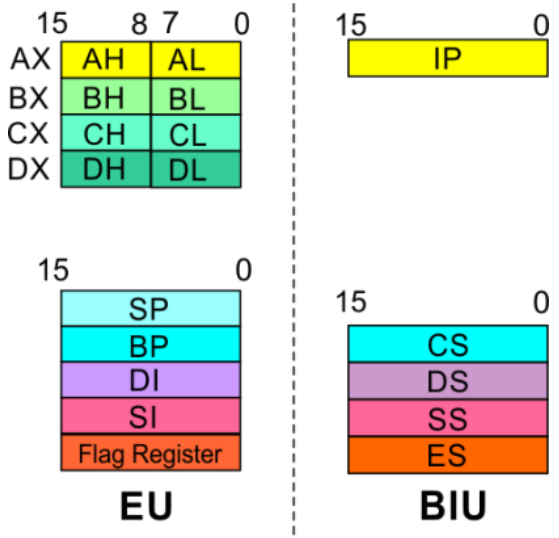
The instruction will specify the name of the register which holds the data to be operated by the instruction.

Example:

MOV CL, DH

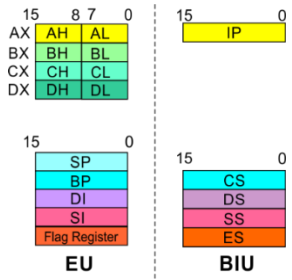
The content of 8-bit register DH is moved to another 8-bit register CL

(CL) ← (DH)



Addressing Modes

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



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

Example:

MOV DL, 08H

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

(DL) ← 08_H

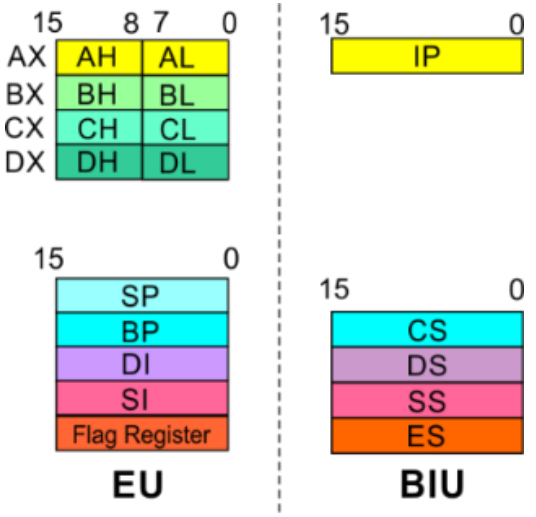
MOV AX, 0A9FH

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

(AX) ← $0A9F_H$

Addressing Modes : Memory Access

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

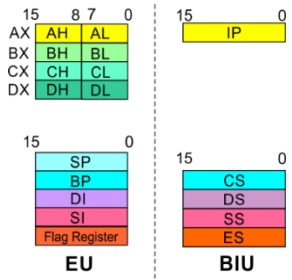


16 bytes of contiguous memory

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

Addressing Modes

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



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

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

Example:

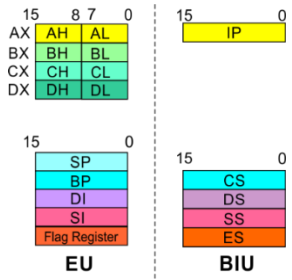
```
MOV  BX, [1354H]
MOV  BL, [0400H]
```

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

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

Addressing Modes

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



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

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

BX, BP, DI and SI.

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

Example:

MOV CX, [BX]

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

Operations:

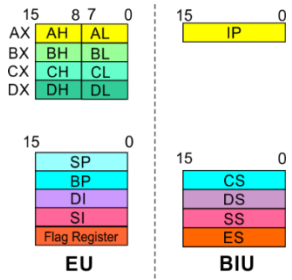
EA = (BX)
BA = (DS) x 16_{10}
MA = BA + EA

(CX) ← (MA) or,

(CL) ← (MA)
(CH) ← (MA +1)

Addressing Modes

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



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

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

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

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

Example:

MOV AX, [BX + 08H]

Operations:

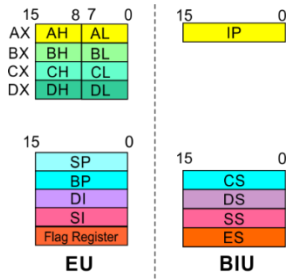
$0008_H \leftarrow 08_H$ (Sign extended)
EA = (BX) + 0008_H
BA = (DS) x 16_{10}
MA = BA + EA

(AX) ← (MA) or,

(AL) ← (MA)
(AH) ← (MA + 1)

Addressing Modes

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



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

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

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

Example:

MOV CX, [SI + 0A2H]

Operations:

$FFA2_H \leftarrow A2_H$ (Sign extended)

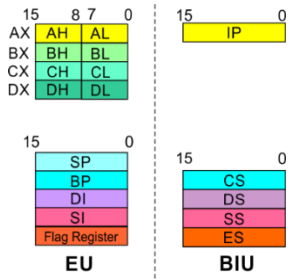
$EA = (SI) + FFA2_H$
$BA = (DS) \times 16_{10}$
$MA = BA + EA$

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

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

Addressing Modes

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



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

Example:

MOV DX, [BX + SI + 0AH]

Operations:

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

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

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

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

Addressing Modes

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

Note : Effective address of the Extra segment register

Employed in string operations to operate on string data.

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

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

Example: MOVS BYTE

Operations:

Calculation of source memory location:

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

Calculation of destination memory location:

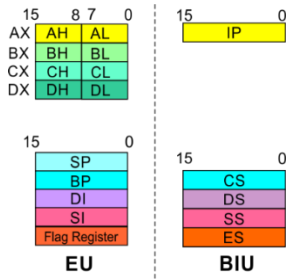
EA_E = (DI) BA_E = (ES) x 16_{10} $MA_E = BA_E + EA_E$

(MAE) ← (MA)

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

Addressing Modes

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



These addressing modes are used to access data from standard I/O mapped devices or ports.

In direct port addressing mode, an 8-bit port address is directly specified in the instruction.

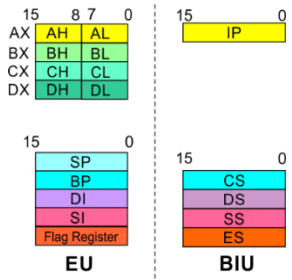
Example: **IN AL, [09H]**

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

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

Addressing Modes

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



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

Example: JZ 0AH

Operations:

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

If ZF = 1, then

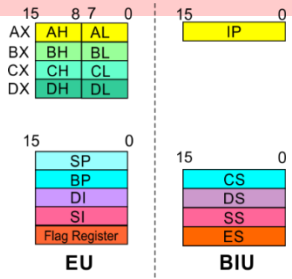
$EA = (IP) + 000A_H$
$BA = (CS) \times 16_{10}$
$MA = BA + EA$

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

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

Addressing Modes

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



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

Example: **CLC**

This clears the carry flag to zero.

INSTRUCTION SET

Instruction Set

8086 supports 6 types of instructions.

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

Instruction Set

1. Data Transfer Instructions

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

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

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

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

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

Instruction Set

1. Data Transfer Instructions

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

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

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

Instruction Set

1. Data Transfer Instructions

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

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

Instruction Set

1. Data Transfer Instructions

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

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

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

Instruction Set

2. Arithmetic Instructions

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

ADD reg2/ mem, reg1/mem **ADC reg2, reg1** **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**

Instruction Set

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**

Instruction Set

2. Arithmetic Instructions

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

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

Instruction Set

2. Arithmetic Instructions

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

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

Instruction Set

2. Arithmetic Instructions

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

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

Instruction Set

2. Arithmetic Instructions

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

CMP reg2/mem, reg1/ mem	

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

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

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

Instruction Set

2. Arithmetic Instructions

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

CMP reg/mem, data	

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

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

Instruction Set

2. Arithmetic Instructions

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

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

Instruction Set

3. Logical Instructions

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

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

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

Instruction Set

3. Logical Instructions

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

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

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

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

Instruction Set

3. Logical Instructions

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

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

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

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

Instruction Set

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

Instruction Set

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

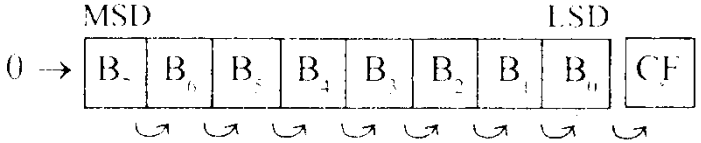
SHR mem

i) SHR mem, 1

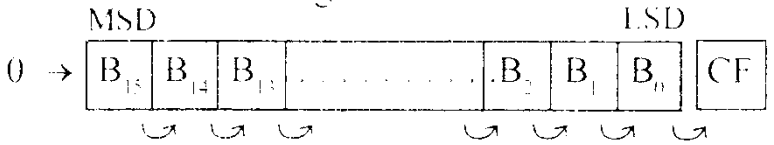
ii) SHR mem, CL

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

reg 8 / mem 8



reg 16 / mem 16



Instruction Set

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

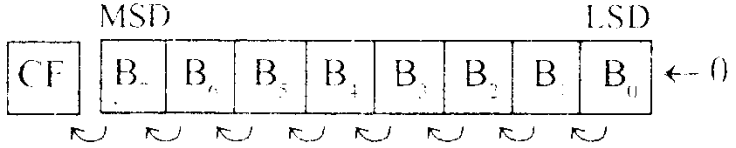
SHL mem or SAL mem

i) SHL mem, 1 or SAL mem, 1

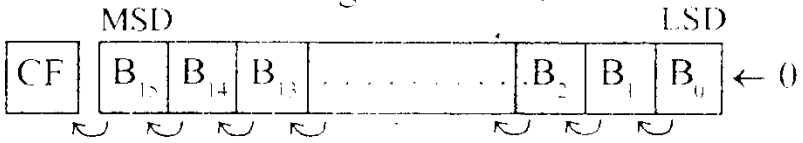
ii) SHL mem, CL or SAL mem, CL

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

reg 8 / mem 8



reg 16 / mem 16



Instruction Set

3. Logical Instructions

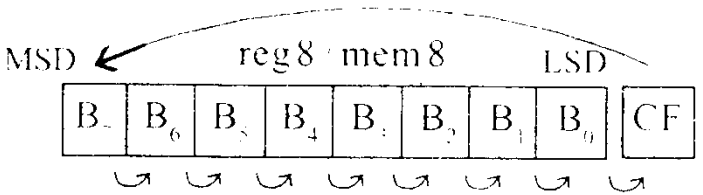
Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL …**

RCR reg/mem

RCR reg

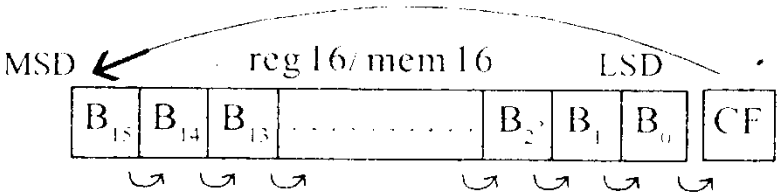
i) RCR reg, 1
ii) RCR reg, CL

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



RCR mem

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



Instruction Set

3. Logical Instructions

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

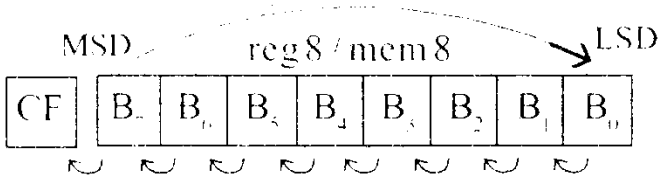
ROL reg/mem

ROL reg

i) ROL reg, 1

ii) ROL reg, CL

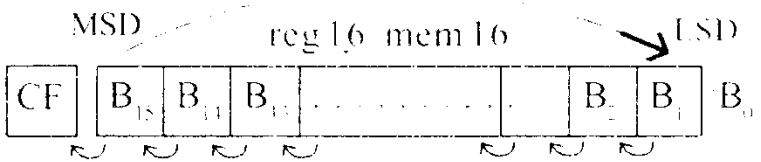
$B_{n+1} \leftarrow B_n : CF \leftarrow B_{MSD} ; B_{LSD} \leftarrow B_{MSD}$



ROL mem

i) ROL mem, 1

ii) ROL mem, CL



Instruction Set

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 **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**

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**
CMPSW	**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**

Instruction Set

4. String Manipulation Instructions

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

Scan (compare) a string byte or word with accumulator

SCAS	

SCASB	$MA_E = (ES) \times 16_{10} + (DI)$
	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	$MA_E = (ES) \times 16_{10} + (DI)$
	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) x 16_{10} + (DI)** **(MA_E) ← (AL)** **If DF = 0, then (DI) ← (DI) + 1** **If DF = 1, then (DI) ← (DI) – 1**
STOSW	**MA_E = (ES) x 16_{10} + (DI)** **(MA_E ; MA_E + 1) ← (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

- 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

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	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