17.1. INTRODUCTION

The main limitation of the 8 bit microprocessors were their low speed of execution, low memory addressing capability, limited number of General purpose registers and less powerful instruction set. All these limitations of the 8 bit microprocessors tempted the designers to go for more powerful processors in terms of advanced architecture, more processing capability, larger memory addressing capability and a more powerful instruction set. The 8086 was a result of such developmental design efforts.

In the family of 16 bit microprocessors, Intel 8086 was the first one, launched in 1978. The introduction of the 16 bit processor was a result of the increasing demand for more and more powerful and high speed computational resources.

17.2. MICROPROCESSOR 8086

The architecture of 8086 provides a number of improvement over 8085 architecture. It supports a 16 bit ALU, a set of 16 bit registers and provides segment memory addressing capability, a rich instruction set, powerful interrupt structure and a six byte instruction queue.

17.2.1. Main Features of 8086 :

(*i*) 16 bit microprocessor
(*ii*) 16 bit ALU
(*iii*) Maximum clock frequency is 5 MHz
(*iv*) Memory is divided in two banks : (*i*) even bank, (*ii*) odd bank
(*v*) Maximum memory that can be connected is 1 MB
(*vi*) Address range of memory is from 00000 H – FFFFF H.
(*vii*) It requires a single + 5 V power supply
(*viii*) It has 20 bit address bus
(*ix*) It can generate 16 bit input/output address, hence it can access 2^{16} = 65536 I/O ports
(*x*) The Intel 8086 is designed to operate in two modes, namely the minimum mode and maximum mode
(*xi*) 8086 supports multiprogramming
(*xii*) The main feature of 8086 is that it fetches upto six instruction bytes (four instruction bytes for 8088) from memory and queues them in order to speed up instruction execution.

17.3. ARCHITECTURE OF 8086

The complete architecture of 8086 as shown in Fig. 17.1 can be divided into two parts : (*a*) Bus interface unit (BIU) and (*b*) Execution unit (EU).

(*a*) **Bus Interfacing Unit (BIU).** The Bus interface unit is responsible for establishing communications with external devices and peripherals including memory via the bus. Bus interfacing unit perform following functions :

(*i*) It sends instruction from memory.
(*ii*) It reads data from port/memory.

(*iii*) It writes data into port/memory.
(*iv*) It supports instruction queuing.
(*v*) It provides the address relocation facility.

To implement these functions, the BIU contains segment registers, internal communication register, instruction pointer, instruction object code queue, address summer and bus control logic.

BIU contains a dedicated adder which is used to generate a 20 bit Physical address that is output on the address bus. This address is formed by adding 20 bit segment address called *Base address* and 16 bit offset address called *Effective address*.

BIU is also responsible for generating bus control signals such as those for memory read or write and I/O read or write. These signals are needed for control of the circuits in the memory and I/O sub-systems.

(*b*) **Execution Unit (EU).** The Execution unit contains the register set of 8086 except segment registers and Instruction pointer. It has a 16 bit ALU, able to perform the arithmetic and logic operation. The execution unit may pass the results to the Bus interface unit for storing them in memory.

The execution unit is responsible for decoding and executing all instructions. It consists of an ALU, status and control flags, eight general purpose registers, temporary registers and queue control logic.

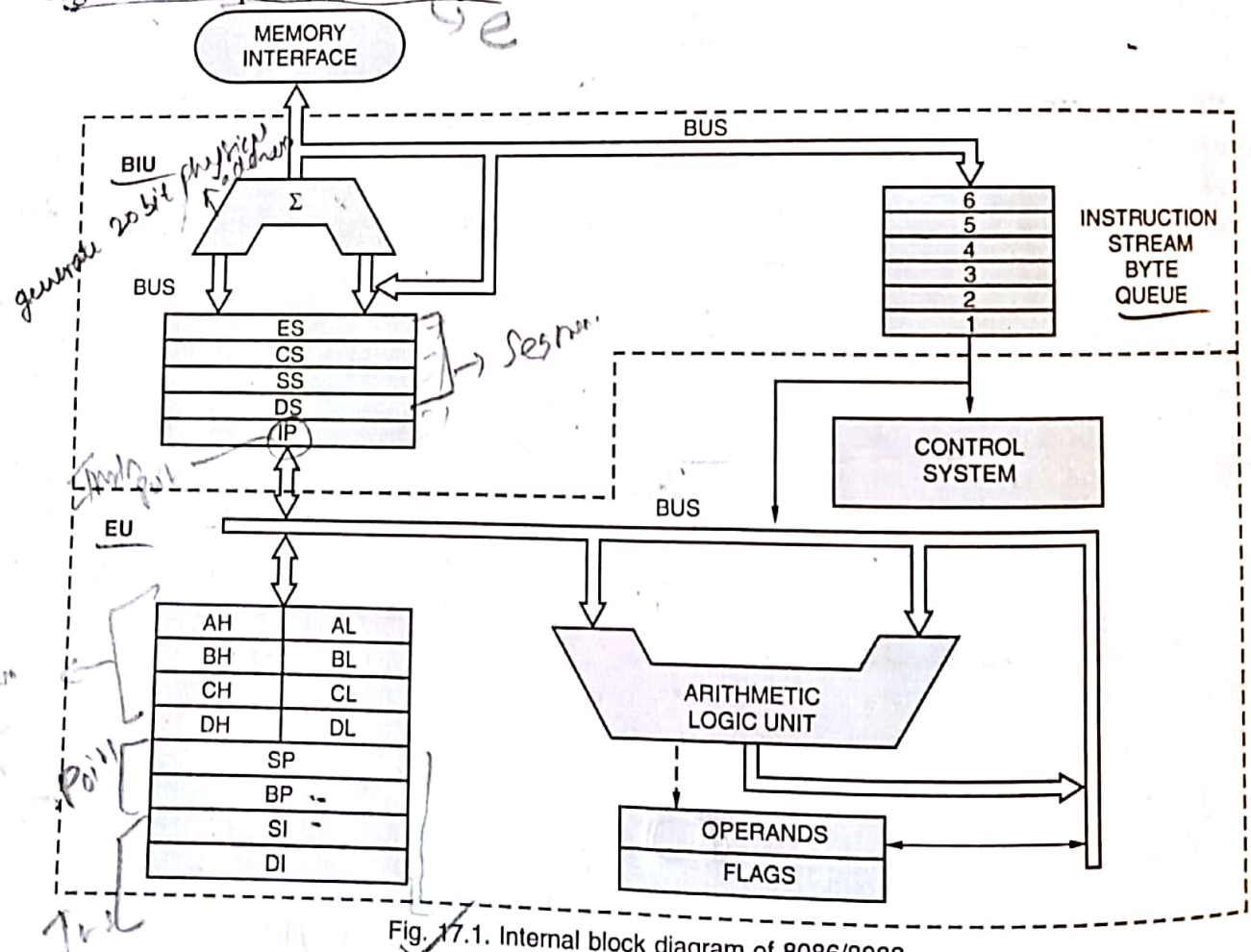


Fig. 17.1. Internal block diagram of 8086/8088.

Execution unit extracts instruction from the top of the queue in the BIU, decodes them, generates operands address if necessary, passes them to BIU and requests it to perform the read or write bus cycles to memory or I/O, and performs the operation specified by the instruction on the operands. During execution of the instruction, the EU tests the status and control flags and updates them based on the results of executing the instructions. If the queue is empty, the EU waits for the next instruction data to be fetched and shifted to the top of the queue.

Both units operates synchronously to give 8086 an overlapping instruction fetch and instruction execution mechanism. This parallel processing of BIU and EU eliminates the time needed to fetch many of the instructions. This results in efficient use of the system bus and significantly improve system performance.

Internal Block diagram of 8086 CPU is shown in Fig. 17.1.

17.4. DETAILED DESCRIPTION OF BLOCK DIAGRAM OF 8086

(1) Address pins. It has 20 address pins A_{19}–A_{16} and AD_{15} to AD_0.

(2) Data pins. It has 16 data pins AD_{15} to AD_0. So address and data pins are common or time shared.

(3) Register set of 8086. The register set of 8086 can be divided into four groups. They are

(a) General purpose registers
(b) Segment registers
(c) Index registers
(d) Pointer registers.

These four groups are shown in Fig. 17.2. All these registers are 16 bit registers. The general purpose register can be used as 8 bit or 16 bit depending on the requirement.

AX	AH	(8)	AL	(8)	GENERAL PURPOSE REGISTERS
BX	BH	(8)	BL	(8)	
CX	CH	(8)	CL	(8)	
DX	DH	(8)	DL	(8)	
	SP			(16)	POINTER REGISTERS
	BP			(16)	
	SI			(16)	INDEX REGISTERS
	DI			(16)	
	ES			(16)	SEGMENT REGISTERS
	CS			(16)	
	SS			(16)	
	DS			(16)	
	IP			(16)	

Fig. 17.2. Registers set of 8086.

General Purpose Registers. There are eight 8 bit registers AH, AL, BH, BL, CH, CL, DH, DL. Each register has 8 flip-flops so each register can store maximum 8 bit of data. There are four register pairs AX (AH and AL), BX, CX, DX. Each register pair can store maximum 16 bit of data. The importance of each register is given below :

(*a*) **AX/AL (Accumulator).** In some instructions like multiplication, division, IN, OUT, XLAT (translate) microprocessor will use AL as 8 bit accumulator and AX as 16 bit accumulator.

(*b*) **BX (Base Register).** In most of the memory related instructions, register BX is used to store 16 bit effective address (EA) of corresponding memory location.

(*c*) **CX/CL (Counter).** In string instruction like REP, LOOP microprocessor will use register CX as implicit counter. Similarly in Rotate/Shift instruction, microprocessor will use register CL as 8 bit counter.

(*d*) **DX (Data Register).** In some instructions like multiplication, division microprocessor will use register DX for storing data or result.

Similarly for IN and OUT instructions, 16 bit port address is stored in register DX and the name of register DX is given alongwith IN and OUT instruction.

Segment Registers. The 8086 uses memory segmentation. Segmentation means dividing the memory into various parts which improves the memory utilisation. For example let us consider a home which contains various rooms instead of a large Hall. This partition is necessary to live conveniently in the house. The same principle can be applied to microprocessor memory. 8086 treats the 1 Mega byte of memory as four each segments, with the maximum size of segment as 64 K bytes. Thus a location within a segment can be addressed using 16 bits. This is convenient in view of the 16 bit word size of 8086. A segment always starts at a memory location with 0 H (zero) as the least significant hex digit, for example 31240 H. But 61496 H cannot be a starting address of a segment because its least significant bit is not zero. To get the starting address of a segment it is enough to specify only the most significant bits, with the least significant 4 bits always taken as zeros. There are 4 different types of segments in memory. They are the

(*a*) Code Segment
(*b*) Data Segment
(*c*) Stack Segment
(*d*) Extra Segment.

(*a*) **CS Register (Code Segment).** The code segment holds the program code, where exactly the code segment starts in memory and is indicated by the contents of a register called CS (Code Segment) register. CS register is 16 bits wide. But we can imagine it to be a 20 bit register where the least significant hex digits is 0 H i.e., the least significant 4 bits are always zeros. This 20 bit value provides the starting address of the Code Segment in memory. If the contents of CS is 1234 H, it means that the code segment starts from memory location 12340 H, which is a 20 bit address. The code segment cannot start at locations where the LSB is not zero. The 16 bit contents of a segment register, is called the *segment offset*.

(b) DS Register (Data Segment). During program execution, the microprocessor accesses a data memory location, where data is stored for a read or a write operation. In the case of 8086 there will be a separate Data Segment, where the data will reside. The data segment holds the data, constants and work areas needed by the program. Where exactly the data segment starts in memory is indicated by the contents of a register called DS (Data Segment) register. DS register is also 16 bits wide but 20 bit address can be obtained by considering the least significant bits as zeros, i.e., 0 H. If the contents of DS is 2799, then the data segment starts from 27990 which is a 20 bit address.

(c) SS (Stack Segment) Register. A stack is a portion of memory set aside to store addresses and data, when the microprocessor control branches to a subroutine. This is always necessary while dealing with instructions like JUMP (similar to GOTO in C). The stack is a last in first out (LIFO) data structure implemented in RAM. During program execution, the microprocessor accesses the stack for a POP or a PUSH operation. In the case of 8086, there will be a seperate stack segment, where the stack will reside. Where exactly the stack segment starts in memory is indicated by the contents of register called *SS (Stack Segment) register*. SS register is also 16 bits wide and 20 bit address can be obtained by considering the least significant bits as 0 H. This 20 bit value provides the starting address of the Stack Segment in memory.

(d) ES Register (Extra Segment). A string is a series of bytes or words in Sequential memory locations. 8086 has a number of instructions for manipulation of strings. If we want to move a string from one memory area to another, the string must be in the data segment, with the offset value or effective address specified in SI register. SI stands for Source Index and is a 16 bit register. SI can also be used for holding a 16 bit data temporarily. The destination where the string will be moved must be in another segment called the *Extra Segment*. Where exactly the Extra Segment starts in memory is indicated by the contents of register called *ES register*, which is 16 bits wide. The address for the destination of the string must be in the extra segment with the offset value or effective address specified in DI register. DI stands for Destination index and is a 16 bit register. DI can also be used for holding a 16 bit data temporarily.

Index registers. 8086 contains two Index registers. They are Source index registers and Destination index register. These two registers are used while dealing with string operations. Both the registers are 16 bit registers. If a string is to be moved from one location to another location, then the offset value or effective address of starting memory location is specified in Source index register. The offset or effective memory location address to which the string is to be moved (*i.e.*, the destination) is specified in Destination index register. Both SI and DI can be used to hold 16 bit data temporarily, if it is necessary.

Pointer registers. 8086 contains 3 pointer registers. They are

1. Stack pointer
2. Base pointer
3. Instruction pointer.

All these pointers are 16 bit registers. The pointer registers will indicate the address in memory locations.

a) The *instruction pointer* always indicates the address of the next instruction which is to be executed next.

b) The *stack point* indicates the top of the stack from where the data or address can be obtained and can be stored. The PUSH and POP operations can be performed from the address location indicated by the stack.

c) The *Base pointer* also indicate the address. In some operations it may be necessary to move the control from one segment to another segment. During these operations the Base pointer (BP) is used instead of stack pointer to indicate the top.

(4) Arithmetic and Logic unit (ALU). ALU can perform various arithmetic and logical operations of 8 as well as 16 bit numbers. 8086 can perform following arithmetic operations :

Addition, Addition with carry, Subtraction, Subtraction with borrow, BCD numbers Addition and Subtraction, Comparison of two numbers, Multiplication, Division and arithmetic operations of decimal numbers respresented in ASCII Code (American standard code for information interchange codes). 8086 can perform following logical operations ANDing, ORing, XORing, Inverting, obtaining 2's compliment, Rotating and Shifting the data towards left and right.

(5) Processor status word (Flag register)

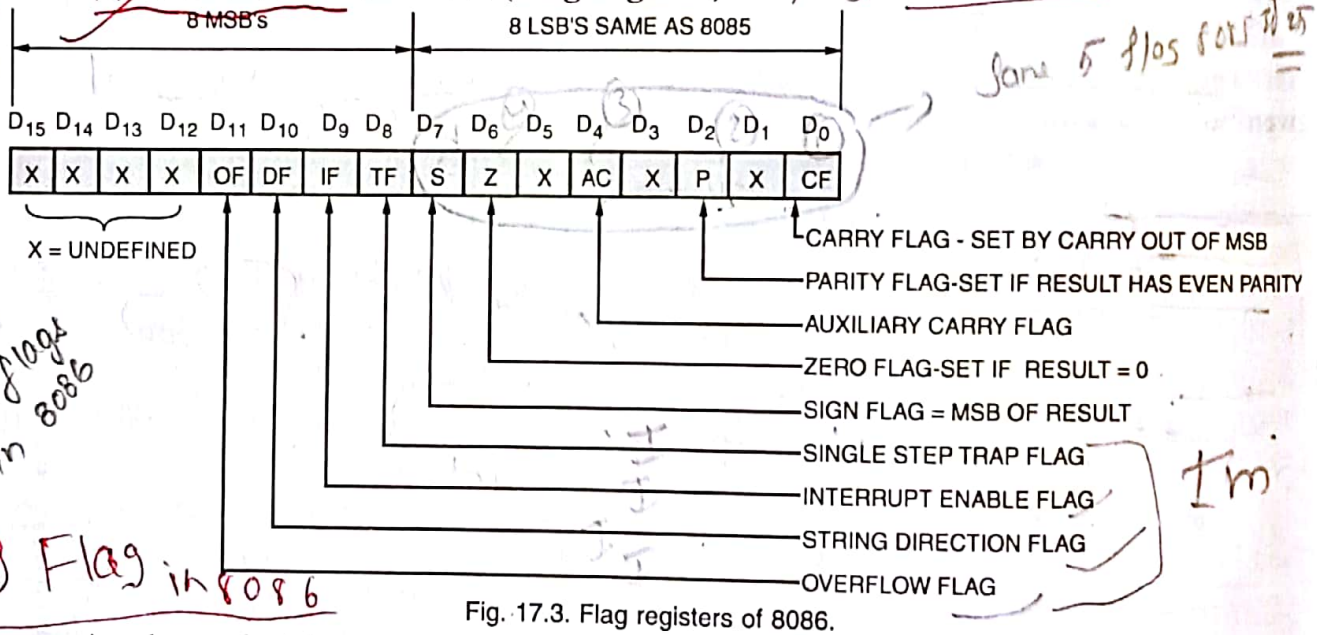


Fig. 17.3. Flag registers of 8086.

As shown in Fig. 17.3, PSW is a 16 bit register which contain 16 flip-flops but 7 flip-flop's are unused and rest of the flip-flops are called *flags*. Out of these 9 flags, there are 6 status flags (CF, P, AC, Z, S, OF) and rest of the three are controlling flags (DF, IF, TF).

Status Flags. When microprocessor performs any operation in ALU, then depending upon the status of result obtained in ALU, microprocessor will store corresponding status bit into the flip-flops of these status flags.

(a) Carry Flag. When microprocessor performs addition of 8/16 bit number, then last carry bit is copied in carry flag, when microprocessor performs subtraction of 8/16 bit

number thus additional borrow required is copied into Carry Flag.

(b) Auxiliary Carry (AC) flags. When microprocessor performs addition of 8 bit number, then carry bit generated after adding 4 LSB's is copied into AC when microprocessor perform subtraction of 8 bit numbers, then borrow required to perform subtraction of 4 LSB's is copied into AC flag.

(c) Parity (P) flag. When microprocessor performs any arithmetic and logical operation of 8/16 bit number, then the parity status of only 8 LSB's is copied into Parity flag. If the parity of LSB's of result is Even/Odd, then P = 1/0 respectively.

(d) Zero (Z) flag. When microprocessor performs arithmetic and logical operation of 8/16 bit number, then if 8 LSB's /16 LSB's of the result is zero *i.e.* 00 H/0000 H, then Z = 1. But if 8 LSB's/16 LSB's is non-zero, then Z = 0.

(e) Sign (S) flag. When microprocessor performs arithmetic and logical operation of 8/16 bit number, then the MSB sign bit of 8/16 bit LSB's of the result is copied into Sign Flag. If result is correct signed binary number, then sign bit will indicate correct sign of that result, *i.e.*, if S = 1/0, then result is –ve /+ve respectively.

(f) Overflow Flag (OF). If carry into MSB = carry out of MSB, then result will be in the range and so OF = 0.

The signed Binary number represented using 2's compliment method in 4 bit is given below in Table 17.1.

Table 17.1.

Signed Binary Number		Equivalent Decimal
Signed Bit	Magnitude	
0	111	(+7)D
0	110	(+6) D
0	101	(+5) D
0	100	(+4) D
0	011	(+3) D
0	010	(+2) D
0	001	(+1) D
0	000	(+0) D
1	111	(–1) D
1	110	(–2) D
1	101	(–3) D
1	100	(–4) D
1	011	(–5) D
1	010	(–6) D
1	001	(–7) D
1	000	(–8) D

In 4 bit we can represent signed numbers from (+7) D to (–8) D, when microprocessor performs arithmetic operations of signed binary numbers, then result will be signed binary number and if the signed binary result obtained is within the range, then OF = 0 and result will be correct signed binary number.

→ If the signed binary result is out of the range, then OF = 1 and result will be incorrect signed binary number.

Logic used for Overflow Flag (OF) :

→ (*a*) When microprocessor performs addition of 8/16 bit numbers, then if carry into the MSB (C_{in}) = carry out of the MSB (C_{out}), then result will be within the range and OF = 0, if $C_{in} \neq C_{out}$, then result is out of the range and OF = 1.

→ (*b*) When microprocessor performs subtraction of 8/16 bit numbers, then if borrow into the MSB (B_{in}) = Borrow out of the MSB (B_{out}), the result will be with in the range and OF = 0, if $B_{in} \neq B_{out}$, then result is out of range and OF = 1.

For Examples :

C_{in}

0 1

(+ 3) D $(0\ 0\ 1\ 1)_2$

+ (+ 2) D $(0\ 0\ 1\ 0)_2$

(+ 5) D 0 | 0 1 0 1

C_{out}

As $C_{in} = C_{out}$, so OF = 0

C_{in}

1 1 1

(+ 7) D (0 1 1 1)

+ (+ 3) D (0 0 1 1)

(+ 10) D 0 1 0 1 0

C_{out}

As $C_{in} \neq C_{out}$, so OF = 1

→ **Controlling flags of 8086.** For storing bit 0/1 in the Controlling Flags of microprocessor, we have to use particular instructions. Depending upon the value of binary bit stored in the Controlling flags flip-flop, microprocessor will control the corresponding operation hence these flags are called *controlling flags.*

(*a*) **Direction flag (DF).** If DF is made zero by giving instruction CLD (Clear direction flag), then after each string Byte/Word operation, microprocessor will increment SI (Source Index Register) and DI (Destination Index Register) by 1/2 respectively.

If DF = 1 by giving instruction STD (Set Direction Flag), thus after each string Byte/Word operation microprocessor will decrement SI and DI by 1/2 respectively.

(b) Interrupt Flag (IF). If Interrupt flag is made '0' (CLI - Clear Interrupt), then Hardware interrupt INTR is disabled. If Interrupt flag is made '1' (STI - Set interrupt), then INTR is enabled.

(c) TRAP Flag (TF). The method of detecting error in the program is called as *software debugging*. For software debugging, microprocessor has to execute the program in single stepping mode *i.e.* microprocessor will execute only one instruction at a time. If TF = 0, then Microprocesor will execute the complete program in a single operation, but if TF = 1, then microprocessor will execute the program in single stepping mode *i.e.* after execution of each instruction, programmer can verify the contents of different registers and flags. In this way the error in the program can be detected.

There is no instruction in 8086 to directly set the TF to 1, or reset the TF to 0. So, to set the TF to 1, the following instructions are executed. These instructions are discussed later.

```
PUSHF
MOV BP, SP
OR WORD PTR (BP + 0), 0100 H
POPF
```

If the TF is 1, the 8086 gets automatically interrupted at the end of every instruction. The interrupt request does not come to 8086 on NMI or INTR. It is internally generated by the 8086. The 8086 then branches to a Interrupt service subroutine (ISS), which should be written to display the contents of the various registers and memory variable on the CRT. At the end of the ISS, the 8086 returns to the program to be checked, and executes the next instruction. Once again, at the end of this instruction, the 8086 branches to the single step ISS and displays the register contents. This way, the entire program can be run, an instruction at a time, and the errors can be easily traced and corrected.

The instruction in the ISS should not be executed in single step. To facilitate this, the T flag should be made 0 in the beginning of this ISS, using the instruction below :

```
PUSHF
MOV BP, SP
AND WORD PTR (BP + 0), 0FEFF H
POPF
```

At the end of the ISS once again T flag is set to 1, so that actual program is interrupted at the end of every instruction.

Instruction Queue (IQ) and Pipelining. There are 6 eight bit registers in the instruction queue of 8086 as shown in Fig. 17.4. Initially the program is prepared using the instruction in form of mnemonics. Each mnemonics is then translated into Hexadecimal

codes and stored into successive memory location. Following steps took place when microprocessor executes the programme :

(1) Initially microprocessor will read, store six bytes of instructions codes in the six registers of instruction queue. This is called *prefetching*.

(2) For Executing the instruction codes, microprocessor will read these codes from the registers of instruction queue in FIFO (first-in-first-out) sequence.

(3) When two register of instruction queue becomes vacant, then in parallel with internal operation, microprocessor will fetch two bytes of instruction codes in sequence and store these codes in the vacant registers of instruction queue *i.e.* speed increases.

(4) This method of prefetching instruction codes in advance in FIFO sequence is called as *pipelining*.

(5) If branching instruction comes in between, then all the six registers of instruction queue are flushed or cleared to 0000 H and the new instruction codes are read from new branching address.

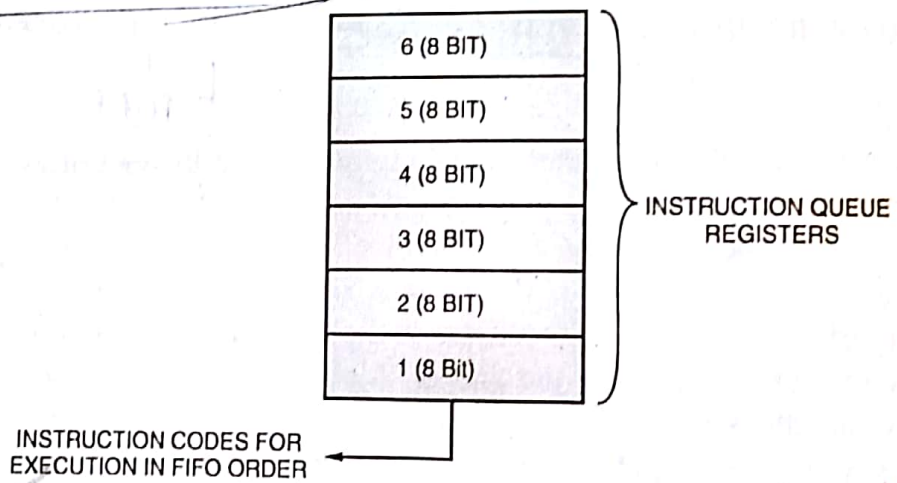


Fig. 17.4.

17.5. SEGMENTATION OF MEMORY USED WITH 8086/8088

8086/8088 has 20 address pins, so each memory location connected with microprocessor will have 20 bit or five Hexadecimal digit address from 00000 H upto FFFFF H. This actual address of 20 bit of each memory location is called as *Physical Address (PA)*. To select any one memory location, microprocessor has to transfer 20 bit physical address (PA) of corresponding location on 20 address pins.

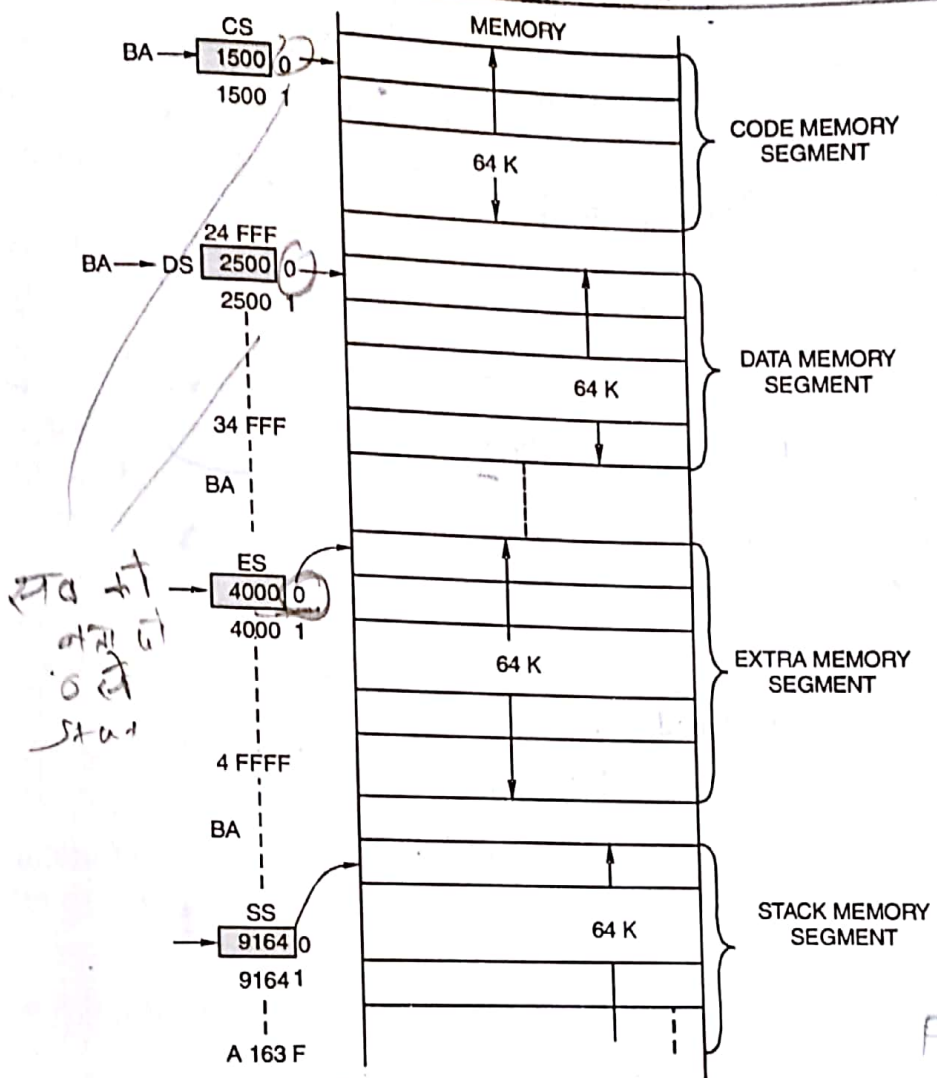


Fig. 17.5. Segmentation of memory.

The memory connected with 8086/8088 is divided into four memory segments. These memory segments will be maximum of 64 K memory location, The 20 bit starting memory location address of each memory segment is called as *Base address* (BA) of the corresponding memory segment. The 4 LSB's of each Base address should be always 0000 = 0 H and rest of the 16 MSB's of Base address can be from 0000 H to FFFF H. These 16 MSB's should be transferred to the corresponding segment register as given below :

(1) Code Memory Segment (CMS). Code memory segment is used to store only instruction codes of the program. For defining Base address of Code memory segment, we have to transfer the corresponding Base address into Code segment register. For example :

If address in CS is 1500 H, then Base address of the Code memory segment = 15000 H.

$$\text{BASE ADDRESS (B.A.)} = \boxed{\text{CS}\ 1500}\ 0 \quad (\text{IMPLICIT})$$

(2) Data Memory Segment (DMS). It is used to store data as well as result. For defining Base Address of Data Memory Segment, we have to transfer 16 MSB's of the corresponding base address into Data segment register.

For example : If 2500 H is transferred into DS, then Base address of Data Memory Segment = 25000 H.

DS

BASE ADDRESS = 2500 0

IMPLICIT

(3) Extra Memory Segment (EMS). It is used to store data as well as result. For defining Base address of Extra Memory Segment, we have to transferred 16 MSB's of the corresponding base address into Extra segment register.

For example : If 4000 H is transferred into ES, then Base address of EMS = 40000 H

ES

BASE ADDRESS = 4000 0

IMPLICIT

(4) Stack Memory Segment (SMS). It is used for storing stack of useful data using PUSH and POP instruction, for defining BA of SMS, we have to transfer 16 MSB's of the corresponding BA into Stack segment register.

For example : If 9164 H is transferred into SS, then Base address of Stack Memory Segment = 91640 H

SS

BASE ADDRESS = 9164 0

IMPLICIT

The maximum memory capacity of 8086 is 1 M Byte. But for a particular base address microprocessor can use maximum 4 blocks of 64 K.

These 4 memory segments can be completely isolated or there can be partially overlapping or there can be complete overlapping.

17.6. METHODS OF GENERATING PHYSICAL ADDRESS OF MEMORY LOCATIONS

The 20 bit actual address of a memory location is called *Physical address* (P.A.). To select any one memory location microprocessor has to transfer 20 Bit Physical address of the corresponding location at 20 address pins. For generating 20 Bit Physical address microprocessor will add Base address and Effective address (E.A.) in the summation unit (Σ).

As shown in Fig. 17.6.

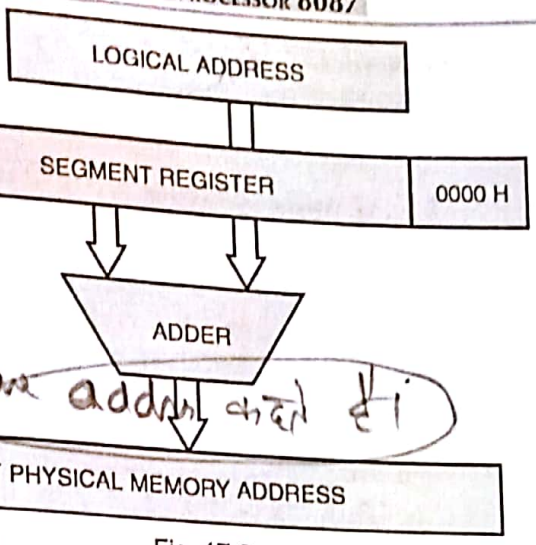
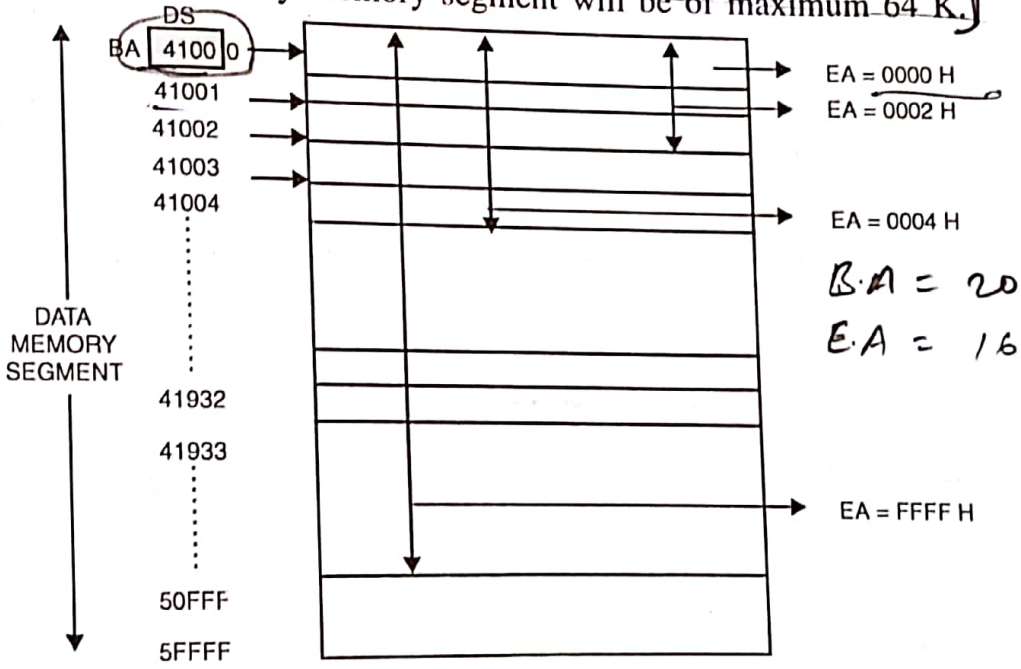


Fig. 17.6.

The 20 bit starting memory location address of memory segment is called as *base address* (*B.A.*) and the offset or displacement of any memory location from the base address is called on *Effective address (E.A.)/Offset/Logical Address*. Normally effective address is stored in 16 bit register. Hence the maximum value of effective address will be FFFF H. So the capacity of any memory segment will be of maximum 64 K.



Microprocessor will calculate Physical address of any memory location as given below :

B.A. = SEGMENT REGISTER (16) | 0(4)

E.A. = POINTERS (16)

PHYSICAL ADDRESS (20) = B.A. (20) + E.A. (16)

Example 17.1. *If content of DS = 4100 H, E.A. = 0002 H, then calculate Physical address.*

Solution.

B.A.= | 4100 | 0 |

E.A. = + 0002

So Physical address = 41002 H

Example 17.2. *If content of DS = 5602 H and EA = 1907 H. Calculate Physical address.*

Solution.

B.A.= | 5602 | 0 |

E.A = 190 7

So Physical address = 57927 H

Example 17.3. *Content of*

SS = 15 A1 H

E.A. = ABC0 H

Calculate Physical address.

Solution. BA = 15A10 H

EA = ABC0 H

So Physical address = 205D0 H

Example 17.4. *BA : EA = 90000 H : 15C0 H*

Calculate Physical address.

Solution. BA = 90000 H

EA = +15C0 H

So Physical address = 915C0 H

17.7. POINTERS AND INDEX REGISTERS

Pointers are 16 bit registers which are used to store 16 bit effective address of memory location. As pointers will be of 16 bits, so the maximum value of effective address will be of 16 bit *i.e.* FFFF H.

There are 4 pointers and 2 index registers as given below :

(*a*) BX and BP (Base registers)

(*b*) SI and DI (Index registers)

(*c*) IP and SP (Pointer register).

(1) IP (Instruction pointer). The process of transferring instruction codes from memory location to instruction queue register is called as *opcode fetch.*

For opcode fetch or for reading any other byte of the instruction from memory, microprocessor will first select corresponding memory location of code memory segment by transfering 20 bit Physical address (on address pins). For generating this Physical address microprocessor will always take Base address from Code segment (CS) and Effective address from Instruction Point (IP) respectively.

B.A. =	CS (16)	0(4)
E.A. = +	I.P. (16)	
P.A. =	(20)	

After each code fetch, the effective address of instruction pointer is auto incremented by 1/2 so microprocessor will perform fetching of codes in sequence from successive memory location till Branching instruction (JMP, CALL) comes in the program.

For example : If CS : IP = 9105 H : 1724 H, then calculate the Physical address of memory location from where microprocessor will fetch the next codes.

Solution.

B.A.	=	91050 H
E.A.	=	1724 H
PA	=	92774 H
After reading the Opcode IP	=	1726 H

So microprocessor will fetch next two bytes of instruction codes from memory location 92774 H and 92775 H. The effective address of IP is incremented by 2 after reading the opcode.

(2) SP (Stack Pointer). SP is 16 bit register which contains 16 bit effective address of the current stack top memory locations. For obtaining physical address of stack top memory location, microprocessor will always take base address and effective adddress from stack segment and SP respectively.

B.A. =	SS (16)	0(4)
E.A. =	+ SP (16)	
Stack top →	PA (20)	

Example 17.5. *In the Fig. 17.8 shown contents of SS: SP = 4500 : 0236 H are given.*

Calculate Physical address of stock top.

Solution.

B.A.	=	45000 H
E.A.	=	0236 H
Physical address of Stack top	=	45236 H

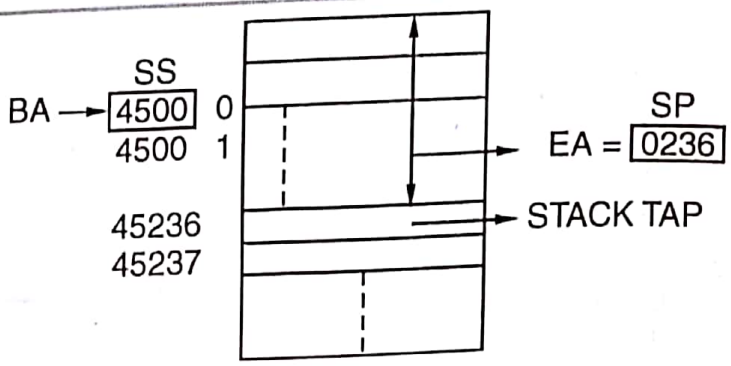


Fig. 17.7.

After each PUSH/POP instruction, the 16 bit effective address of SP is decremented/ incremented by two.

(3) BP (Base Pointer). BP is a 16 bit register. It is used to store 16 bit effective address of Stack memory segment location for data transfer instruction. If EA is present in BP and the name of segment register is not given then microprocessor will always take base address from SS register as given below :

B.A. = | SS (16) | 0(4) |

E.A. = + | BP (16) |

Physical address = PA (20)

Let us take an example where the contents of BP is 1532 H and stack segment contains 1500 H. Consider the instruction

MOV A H, [BP + 0]

[BP + 0] indicates that BP contains effective address. So the physical address is calculated as follows :

SS

B.A. = | 1500 | 0(4) |

E.A. = + | 1532 H | BP

P.A. = | 16532 H |

Mem

| 29 H | 16532 H ← PA

AH | 29 |

With this instruction the data at memory location 16532H is transfer to AH as shown above.

(4) BX (Base Register). BX is a 16 bit register which is used to store 16 bit Effective address of Data memory segment location for Data transfer instruction. If Effective address is present in register BX and the name of segment register is not given in the instruction, then microprocessor will always take B.A. from DS register.

PUSH P
MOV BP,SP
OR WORD PTR

B.A. = | DS (16) | 0(4) |

E.A. = + | BX (16) |

| PA (20) |

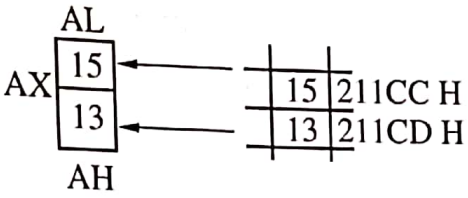
For example : Explain the instruction MOV AX, [BX] assuming DS = 1906 BX = 816C.

DS

B.A. = | 1906 | 0 |

E.A. = | 816 C | BX

P.A. = 211 CC H



(5) **Index Registers.** Source index (SI) and Destination index (DI) are 16 bit registers which are used for storing 16 bit Effective address of Memory locations for Data transfer instructions as well as string instructions.

If microprocessor executes any string instruction, then for selecting source memory location, microprocessor will generate physical address by taking Base address and Effective address from DS and SI respectively.

B.A.s = | DS (16) | 0 (4) |

E.A.s = + | SI (16) |

| P.A.s (20) |

Here s → Source

Microprocessor will read 8/16 bit number from this selected source memory location.

Similarly, if microprocessor stores 8/16 bit number into destination memory using string instruction, then to select this destination memory location, microprocessor will generate Physical address by taking BA and EA from ES and DI respectively.

B.A.d = | ES (16) | 0 (4) |

E.A.d = + | DI (16) |

| (20) |

Here d → Destination

After executing each string byte/word operation, microprocessor will increment/decrement SI and DI by one/two depending upon Direction flag (DF) = 0/1.

17.8. DEFAULT SEGMENT REGISTERS AND SPECIFIED SEGMENT REGISTERS FOR EACH POINTER

Pointers are used for storing 16 bit Effective address of memory location. For selecting memory location, microprocessor will generate 20 bit Physical address by adding Base address (B.A.) and Effective address (E.A.). If the name of segment register is not given alongwith the instruction then microprocessor will always take Base address from default segment register but if the name of segment register is given alongwith instruction, then microprocessor will take Base address from this specified segment register.

The name of default segment register and specified segment register for each pointer is given below in Table 17.2.

Table 17.2.

Type of Memory Reference	Default Segment	Alternate Segment	Offset (Logical Address)
Instruction fetch	CS	None	IP
Stack operation	SS	None	SP
General data	DS	CS, ES, SS	Effective address
String source	DS	CS, ES, SS	SI
String destination	ES	None	DI
BX used as pointer	DS	CS, ES, SS	Effective address
BP used as pointer	SS	CS, ES, DS	Effective address

17.9. ADDRESSING MODES OF 8086

For executing any instruction microprocessor will need data. The method by which address of source of data is given alongwith instruction is called as *addressing mode of source.*

→ After executing the instructions, microprocessor will store the result. The method by which address of destination of result is given alongwith the instruction is called as *addressing mode of destination.*

Addressing modes of 8086/8088 is divided into two parts.

(*a*) Addressing mode for Data related instructions.

(*b*) Addressing mode for Branching instructions.

17.9.1. Addressing Modes for Data Related Instructions. First we will discuss addressing mode for accessing data immediate, register and memory and then addressing modes for accessing I/O ports (I/O modes).

(1) Immediate Addressing Mode (IAM). If 8/16 bit data required for executing the instruction is given alongwith the instruction, then it is called *Immediate addressing mode*.

For example :

Instruction	Comments
(i) MOV AL, 75 H,	; 75 H → 75 AL
(ii) MOV BX, 7506 H,	; 7506 H → 75 (BH) 06 (BL) BX

(2) Direct Addressing Mode (DAM). If 8/16 bit data is present in memory and 16 bit Effective address of this memory location is given alongwith the instruction, then it is called *Direct Addressing Mode instructions*.

If Effective address is directly given alongwith the instruction, then microprocessor will take Base address from default segment register DS.

Example 17.5. *Explain MOV AL, [9106 H], if DS contains 1005 H.*

Solution.

	DS	
B.A.	: 10050	H
E.A.	: + 9106	H
← P.A.	: 19156 H	

19156 [93 H] → [93 H]
M AL

(3) Register Direct Addressing Mode (RDAM). If 8/16 bit data required for executing the instruction, is present in register and the name of register is given alongwith the instruction, then it is called *Register Direct Addressing Mode (RDAM) instruction.*

For example :

MOV CX, BX

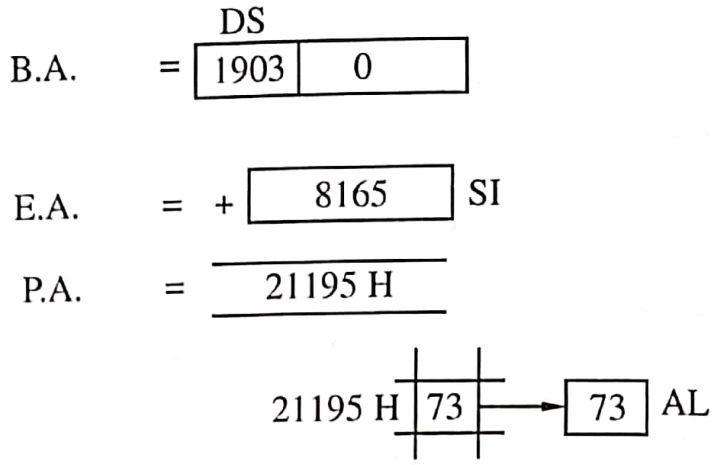
[1807] → [1807]
BX CX

(4) Register Indirect Addressing Mode (RIAM) or Indirect Addressing Mode. If 8/16 bit data required for executing the instruction is present in memory and the 16 bit Effective address of this memory location is present in a register and the name of this register is given alongwith the instruction, then it is called *Register Indirect Addressing Mode (RIAM) instruction*. In this addressing mode the Effective address can be stored either in BX or SI or DI.

EA = [BX] / [SI] / [DI]

Example 17.6. *Explain the instruction MOV AL, [SI].*

Solution. Let the contents of Data segment is 1903 H and source Index register is 8165 H, so the Physical address can be calculated as given below :



With this instruction the contents of memory location 21195 H is transferred to register AL.

(5) Register Relative Addressing Mode or Relative Addressing Mode (RRAM). If data required for executing the instruction is present in memory location and the Effective address of this memory location is obtained by using the equation.

E.A. = [BX] / [BP] / [SI] / [DI] + 8/16 bit Displacement.

If the displacement is of 8 bit then it is converted into 16 bit by extending sign bit.

Examples of this addressing mode are

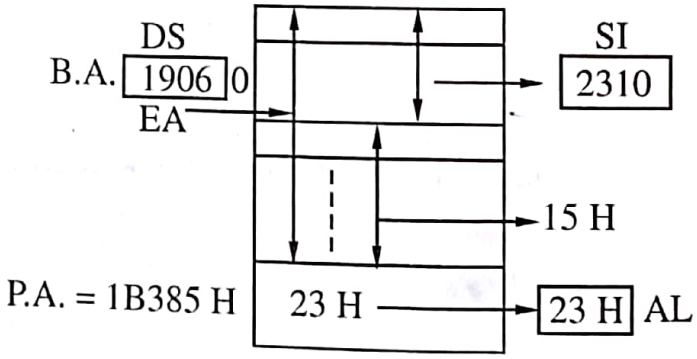
(*i*) MOV CX, 97 H [BP]

(*ii*) MOV AL, 15 H [SI]

(*iii*) MOV DL, 1537 H [DI].

Example 17.7. *Explain the instruction MOV AL, 15 H [SI] or MOV AL, [SI + 15 H].*

Solution. Let us assume DS contains 1906 and SI = 2310



To convert 8 bit displacement 15 H into 16 bit displacement we have to extend. M.S.B. sign bit = 0

15 H = | 0000 0000 | 0 001 01 01 |

15 H = 2310 → SOURCE INDEX (SI)
+ 0015 → DISPLACEMENT
2325 H

The 16 LSB of addition will be Effective address

	D.S	
B.A. =	1906	0
E.A. =	+ 232	5
P.A. =	1B38	5

Example 17.8. *Explain the instruction MOV DL, 1537 H [DI]. Assume [DI] = C15D H, [ES] = 1583 H.*

Solution. Effective address can be calculated by adding the contents of DI with 1537

= C15D H
+ 1537 H
D694 H

So Physical adddress can be calculated as follows :

B.A. = 15830 H
E.A. = D694 H

Physical address = 15830 + D694 = 22EC4 H

After execution of instruction MOV DL, 1537 H [DI], the contents of memory having Physical address 22EC4 H is transferred to DL.

(6) Base Index Addressing Mode (BIAM). If data required for executing the instruction is present in memory location, then in base index addressing mode, microprocessor will calculate Effective address by using the equation.

EA = [BX]/[BP] + [SI]/[DI],

Alongwith the instruction the name of any one base register BX/BP and the name of any one index register SI/DI will be given, microprocessor will take Base address from the Default segment register of the Base register BX/BP given in the instruction. Example of this addressing mode is given below.

Example 17.9. *Explain the instruction*

MOV CX, [BX] [SI]

OR

MOV CX, [BX + SI].

Solution. Let BX = 1573, SI = A1C2 and DS = 1723

In this instruction effective address can be calculated by adding BX with SI

1573	BX
A1C2	SI
B 755 = EA	

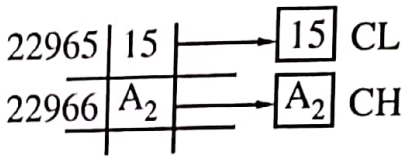
Physical address is obtained by adding Base address and Effective address

B.A. = | 1723 | 0 |

E.A. = + B 735

P.A. = 22965 H

With this instruction data at memory location 22965 H and 22966 H is transfer to CL and CH respectively.



(7) Relative Base Index Addressing Mode [RBIAM]. Data required for executing the instruction is present in memory and in the instruction the name of any one Base register [BX]/[BP], the name of any one index register [SI]/[DI] and 8/16 bit displacement is given. Microprocessor will calculate the Effective address of the corresponding memory location using the equation.

E.A. = [BX]/[BP] + [SI]/[DI] + 8/16 bit displacement.

If displacement is of 8 bit. then it is converted into 16 bit by extending the sign bit. Example of this addressing mode are

(*i*) MOV A H, 1907 H [BX] [DI]

(*ii*) MOV DX, [BP + SI + 9F H] or MOV DX, 9F H [BP] [SI]

Example 17.10. *Explain the instruction :*

MOV DX, [BP + SI + 9F H] or MOV DX, 9F H [BP] [SI].

Solution. Let BP = 1823, SI = 2910 and SS contains 8100

and sign extention of 9F = FF9F H

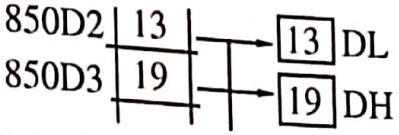
1823 H	(Contents of BP)
2910 H	(Contents of SI)
+ FF9F H	(16 Bit Displacement)
Carry generated → ① 40D2 H	

Neglect carry if generated.

Now let us calculate the Physical address

B.A. = 81000 H
E.A. = 40D2 H
P.A. = 850D2 H

Since the Physical address is even so the 16 bit number will transfer in one operation.



(8) Implicit Addressing Mode [IPAM]. If address of source of data as well as address of destination of result, are fixed then no operand is given alongwith the instruction. Hence such instructions are called *Implicit Addressing Mode instructions.*

For example : CLD, STD.

17.9.2. Addressing Modes for Accessing I/O Port (I/O Modes). Standard I/O uses port addressing modes. For memory-mapped I/O, memory addressing modes are used. There are two types of port addressing modes : Direct and Indirect.

(*a*) **Direct port mode.** In Direct port mode, the port number is an 8-bit immediate operand. This allows fixed access to ports numbered 0 to 255.

For Example :

```
OUT 05 H, AL    :   Sends the contents of AL to 8-bit port 05 H.
```

(*b*) **Indirect Port Mode.** In Indirect port mode, the port number is taken from DX allowing 64 K 8 bit ports or 32 K 16-bit ports.

For Example :

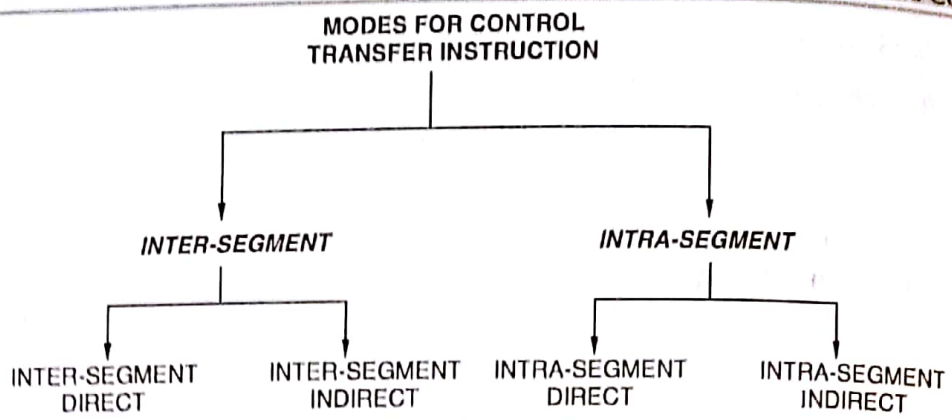
```
IN AL, DX       ;  If [DX] = 7890 H, then it copies 8-bit content of port 7890 H
                ;  into AL.

IN AX, DX       ;  Copies the 8-bit contents of ports 7890 H and 7891 H into
                   AL and AH, respectively.
```

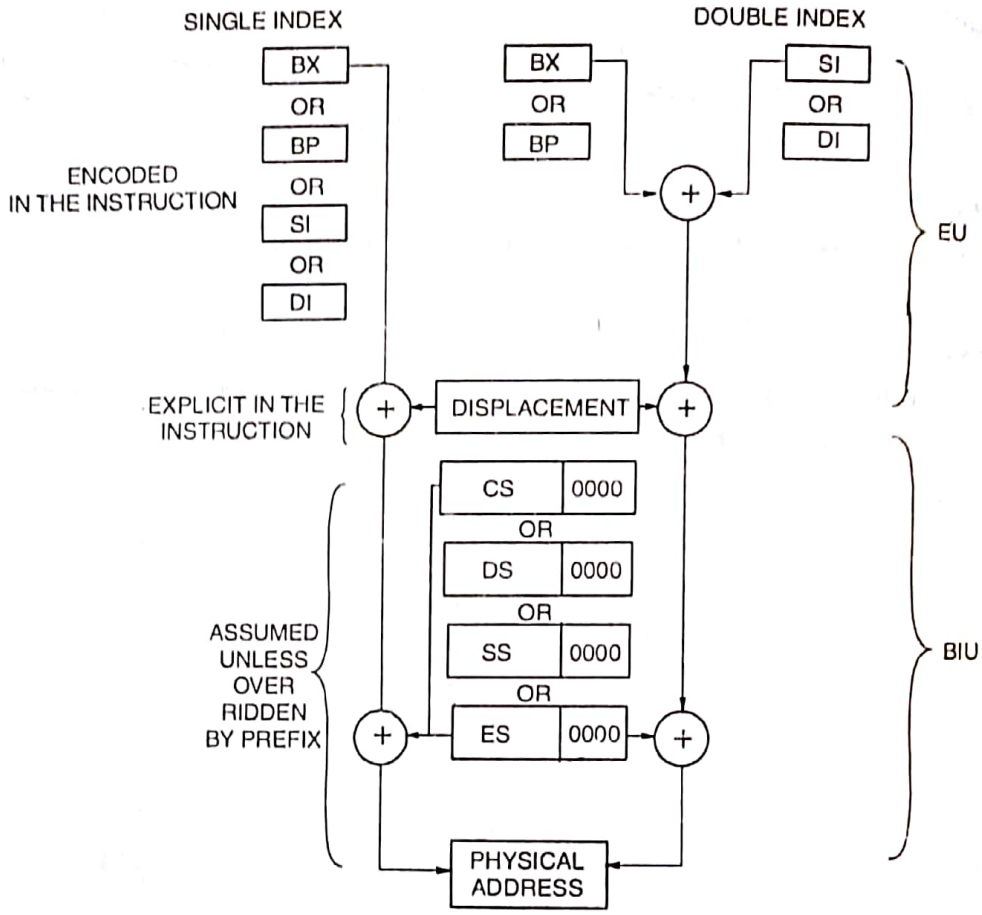
Note. The 8-bit and 16-bit I/O transfers must take place via AL and AX, respectively.

17.9.3. Addressing Mode for Branching Instructions. For the Control transfer/ Branching instructions, the addressing modes depend upon whether the destination location is within the same segment or a different one. It also depends upon the method of passing the destination address to the processor. Basically, there are two addressing modes for the control transfer instruction, *viz.* intersegment and intrasegment addressing modes.

If the location to which the control is to be transferred lies in a different segment other than the current one, the mode is called *Intersegment modes*. If the destination location lies in the same segment, the modes is called *Intrasegment mode*. Fig. 17.8 (*a*) shows the modes for control transfer instructions.



(a) Addressing modes for Branching instructions.



(b) Summary of 8086 Addressing modes.

Fig. 17.8.

Intra-segment direct mode. In this mode, the address to which the control is to be transferred lies in the same segment in which the control transfer, instruction lies and apppears directly in the instruction as an immediate displacement value. In this addressing mode, the displacement is computed relative to the content of the Instruction Pointer (IP).

The Effective address to which the control will be transferred is given by the sum of 8 or 16 bit displacement and current content of IP. In case of Jump instruction, if the signed displacement (d) is of 8 bits (*i.e.*, $-128 < d < +128$), we term it as short jump and if it is of 16 bits (*i.e.*, $-32768 < d < +32768$), it is termed as long jump.

Intra-segment indirect mode. In this mode, the displacement to which the control

is to be transferred, is in the same segment in which the Control transfer instruction lies, but it is passed to the instruction indirectly. Here, the branch address is found as the content of a register or a memory location. This addressing mode may be used in unconditional branch instructions.

(11) **Inter-segment direct mode.** In this mode, the address to which the control is to be transferred is in a different segment. This addressing mode provides a means of branching from one code segment to another code segment. Here, the CS and IP of the destination address are specified directly in the instruction.

(12) **Intersegment indirect mode.** In this mode, the address to which the control is to be transferred lies in a different segment and it is passed to the instruction indirectly, *i.e.* contents of a memory block containing four bytes, i.e. IP (LSB), IP (MSB), CS (LSB) and CS (MSB) sequentially. The starting address of the memory block may be referred using any of the addressing modes, except immediate mode.

The address can be generated by using one index register or by using two index registers. The overview of all addressing modes is shown below in Fig. 17.8 (*b*).

17.10. RELOCABILITY OF 8086/8088

The 16 bit displacement of a memory location from base address is called as *effective address*. In any instruction memory is represented by effective address.

The physical address of any memory location will be fixed but if the base address is changed, then its effective address will change as shown in Fig. 17.9. Hence the same memory location can be represented or accessed or selected by more than one effective address. This characteristic is called as *Relocability*.

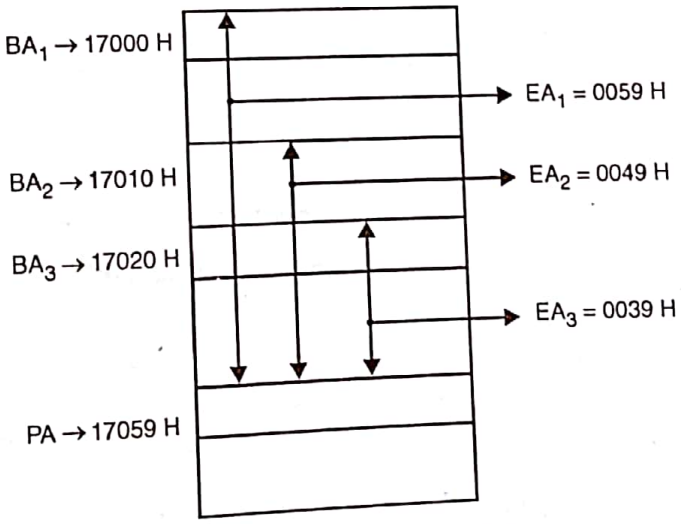


Fig. 17.9. Relocability of 8086/8088.

For example : As shown in Fig. 17.9, if Physical address = 17059 H, then 8 bit data of this memory location can be transferred by any one of the following instructions :

(1) MOV AL, [0059 H]; Considering Base address = 17000 H and effective address = 0059 H.

(2) MOV AL, [0049 H] ; Considering Base address = 17010 H and Effective address = 0049 H

(3) MOV AL, [0039 H]; Considering Base address = 17020 H and Effective address = 0039 H.

17.11. INSTRUCTION SET OF 8086/8088

Depending upon the type of operation performed, the instruction of 8086 are divided into following types :

(1) DATA TRANSFER INSTRUCTIONS :

Mnemonic	Description
Word Transfer Instruction :	
MOV	This instruction is used to Copy byte or word from specified source to specified destination.
PUSH	This instruction is used to Copy specified word to top of stack.
POP	This instruction is used to Copy word from top of stack to specified location.
PUSHA	(80186/80188 only). This instruction is used to Copy all registers to stack.
POPA	(80186/80188 only). This instruction is used to Copy words from stack to all registers.
XCHG	This instruction is used to Exchange bytes or Exchange words.
XLAT	This instruction is used to Translate a byte in AL using a look up table in memory.
Simple Input and Output Port Transfer Instruction :	
IN	It is used to Copy a byte or word from specified port to accumulator.
OUT	It is used to Copy a byte or word from accumulator to specified port.
Special Address Transfer Instructions :	
LEA	This is use to Load effective address of operand into specified register.
LDS	This is use to Load DS register and other specified register from memory.
LES	This is use to Load ES register and other specified register from memory.
Flag Transfer Instructions :	
LAHF	This instruction is use to Load (copy to) AH with the Low byte of the flag register.
SAHF	This instruction is use to Store (copy) AH register to Low byte of flag register.

PUSHF This instruction is use to Copy flag register to top of stack.

POPF This instruction is use to Copy word at top of stack to flag register.

(2) ARITHMETIC INSTRUCTIONS :

Mnemonic	Description
Addition instructions :	
ADD	Use to Add specified byte to byte or specified word to word.
ADC	Use to Add byte + Byte + Carry flag or word + Word + Carry flag.
INC	Use to Increment specified byte or specified word by 1.
AAA	Use to ASCII adjust after addition.
DAA	Use to Decimal (BCD) adjust after addition.
Subtraction instructions :	
SUB	This instruction is use to Subtract byte from byte or word from word.
SBB	This instruction is use to Subtract byte and carry flag from byte or word and carry flag from word.
DEC	This instruction is use to Decrement specified byte or specified word by 1.
NEG	This instruction is use to Negate-invert each bit of a specified byte or word and add 1 (form 2's compliment).
CMP	This instruction is use to Compare two specified bytes or two specified words.
AAS	This instruction is use to ASCII adjust after subtraction.
DAS	This instruction is use to Decimal (BCD) adjust after subtraction.
Multiplication instructions :	
MUL	It is use to Multiple unsigned byte by byte or unsigned word by word.
IMUL	It is use to Multiply signed byte by byte or signed word by word.
AAM	It is use to ASCII adjust after multiplication.
Division instructions :	
DIV	This instruction is use to Divide unsigned word by byte or unsigned double word by word.
IDIV	This instruction is use to Divide signed word by byte or signed double word by word.
AAD	This instruction is use to ASCII adjust before division
CWD	This instruction is use to convert word into double word.
CBW	This instruction is use to convert upper byte of word with copies of sign bit of lower word.

(3) LOGICAL INSTRUCTIONS :

NOT	Invert each bit of a byte or word.
AND	AND each bit in a byte or word with the corresponding bit in another byte or word.
OR	OR each bit in a byte or word with the corresponding bit in another byte or word.
XOR	Exclusive OR each bit in a byte or word with the corresponding bit in another byte or word.
TEST	AND operands to update flags, but don't change operands.

Shift instructions :

SHL/SAL	It shifts bits of word or byte left, put zero(s) in LSB(s).
SHR	This is use to shift bits of word or byte right, put zero(s) in MSB(s).
SAR	This is use to shift bit of word or byte right, copy old MSB into new MSB.

Rotate instructions :

ROL	This is use to Rotate bits of byte or word left, MSB to LSB and to CF.
ROR	This is use to Rotate bits of byte or word right, LSB to MSB and to CF.
RCL	This is use to Rotate bits of byte or word left, MSB to CF and CF to LSB,
RCR	This is use to Rotate bit of byte or word right, LSB to CF and CF to MSB.

(4) STRING INSTRUCTIONS :

REP	An instruction prefix. Repeat following instruction until CX = 0.
REPE/REPZ	An instruction prefix. Repeat instruction until CX = 0 or zero flag ZF ≠ 1.
REPNE/REPNZ	An instruction prefix. Repeat until CX = 0 or ZF = 1.
MOVS/MOVSB/MOVSW	Move byte or word from one string to another.
CMPS/CMPSB/CMPSW	Compare two string bytes or two string words.
INS/INSB/INSW	Input string byte or word from port. This instruction is only for 80186 and 80188.
SCAS/SCASB/SCASW	Scan a string. Compare a string byte with a byte in AL or a string word with a word in AX.
LODS/LODSB/LODSW	Load string byte into AL or string word into AX.
STOS/STOSB/STOSW	Store byte from AL or word from AX into string.
OUTS/OUTSB/OUTSW	Output string byte or word to port. This instruction is only for 80186 and 80188.

(5) BRANCHING INSTRUCTIONS :

Unconditional Branching instructions :

CALL	Call a subprogram, save return address on stack.
RET	Return from subprogram to calling program.
JMP	Go to specified address to get next instruction.

Conditional Branching instructions :

JA/JNBE	Jump if above/Jump if not below or equal.
JAE/JNB	Jump if above or equal /Jump if not below.
JB/JNAE	Jump if below/Jump if not above or equal.
JBE/JNA	Jump if below or equal/jump if not above.
JC	Jump if carry flag CF = 1.
JE/JZ	Jump if equal/Jump if zero flag ZF = 1.
JGE/JNL	Jump it greater than or equal/Jump if not less than.
JL/JNGE	Jump if less than/Jump if not greater than or equal.
JLE/JNG	Jump if less than/Jump if not greater than or equal.
JNC	Jump if no carry (CF = 0).
JNE/JNZ	Jump if not equal/Jump if not zero (ZF = 0).
JNP/JPO	Jump if not parity/Jump if parity odd (PF = 0).
JNS	Jump if not sign (sign flag SF = 0).
JO	Jump if overflow flag OF = 1.
JP/JPE	Jump if parity/Jump if parity even (PF = 1).
JS	Jump if sign (SF = 1).
JNO	Jump if no overflow flag (OF = 0)
LOOP	Loop through a sequence of instructions until CX = 0.
LOOPE/LOOPZ	Loop through a sequence of instruction while ZF = 1 and CX ≠ 0.
LOOPNE/LOOPNZ	Loop through a sequence of instruction while ZF = 0 and CX ≠ 0.
JCXZ	Jump to specified address if CX = 0.
INT	Interrupt program execution call service procedure.
INTO	Interrupt program execution if OF = 1.
IRET	Return from interrupt service procedure to main program.

(6) FLAG CONTROLLING INSTRUCTIONS :

STC	Set Carry flag (CF = 1).
CLC	Clear Carry flag (CF = 0).
CMC	Complement the state of the Carry flag (CF).
STD	Set Direction flag (DF = 1) (Decrement string pointers).

CLD	Clear Direction flag (DF = 0).
STI	Set interrupt enable flag to 1 (Enable INTR input).
CLI	Clear interrupt enable flag to 0 (Disable INTR input).

(7) OTHER INSTRUCTIONS :

HLT	Halt (stop) until interrupt or reset.
WAIT	Wait (stop) until signal on the TEST pin is low.
ESC	Escape to external coprocessor such as 8087 or 8089.
LOCK	An instruction prefix. Prevents another processor from taking the bus while the adjacent instruction executes.
NOP	No operation except fetch and decode.

17.12. CONSTRUCTING THE MACHINE CODES FOR 8086 INSTRUCTIONS

Here we will learn how to construct the binary codes for 8086 instructions. Normally assembler is used for this task, but it is useful to understand how the codes are constructed.

17.12.1. Instruction Templates (Format). To code the instructions for 8-bit processor (For example : 8085) we can directly look up hexadecimal code for each instruction from a one page chart but in case of 8086 the process is not so simple. There are 32 ways to specify the source of operand in an instruction.

The 8086 instruction sizes vary from one to six bytes.

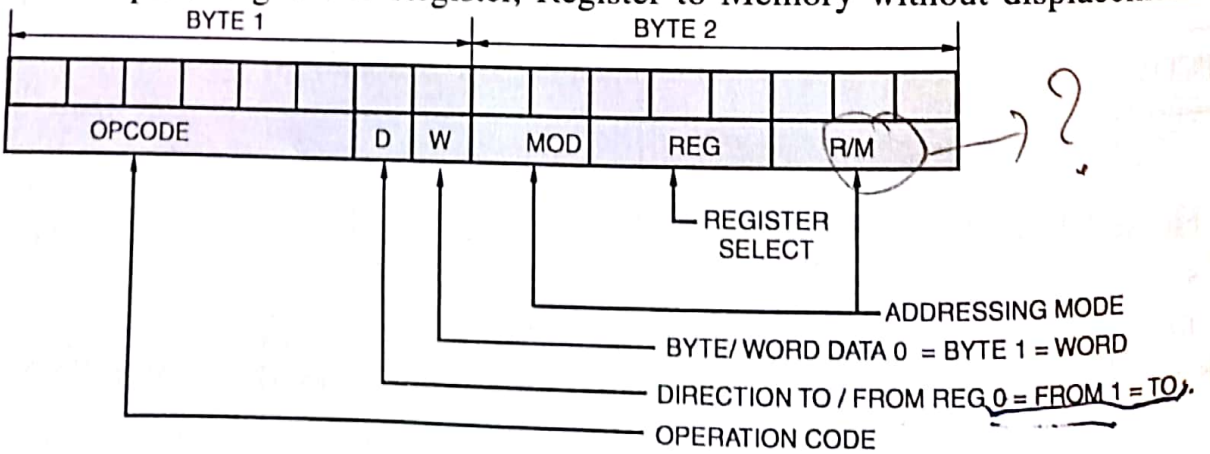
The instruction format is defined as below :

1. One Byte instruction. This type of instruction takes only one byte of memory and may have Implicit data or Register operands e.g., LAHF, SAHF, CBW, CWD.

Mnemonics	Instruction code
LAHF	10011111
SAHF	10011110
CBW	10011000
CWD	10011001

2. Two Bytes instruction. This type of instruction requires two bytes of memory. The first byte of the code specifies the Operation code and Width of Operand specified by W bit. The second byte shows the register operands and R/M field.

For example : Register to Register, Register to Memory without displacement.



Complete explanation of these bits is given in Section 17.12.2.

For example : MOV CH, BL
MOV BL, CL

S.No.	MNEMONICS	INSTRUCTION CODES
1.	MOV CH , BL	1000 1000 1101 1101 88 DD
2.	MOV BL , CL	1000 1000 11001011 88 CB
3.	MOV [SI] , DL	1000 1001 0001 0100 8 9 1 4

3. Three Byte instruction. This type of instruction requires three location of memory. Instruction format contains one additional byte for 8 bit displacement alongwith 2-bytes. For OPCODE, REG and MOD.

OPCODE	D	W	MOD	REG	R/M
Ist BYTE			2nd BYTE		

DISPLACEMENT (8 BIT)
3rd BYTE

For example : (*i*) Register to memory with displacement (8 bit).
(*ii*) Memory to Register with displacement (8 bit).
(*iii*) Immediate data to Register.

For example :

S.No.	MNEMONICS	INSTRUCTION CODES
1.	MOV 43 H [SI], DH	1000 1000 0111 0100 0100 0011 8 8 7 4 4 3

4. Four Byte instruction. These type of instructions requires four bytes of memory locations. Two additional bytes are used for storing 16 bit displacement or 16 bit. Immediate data alongwith first two bytes containing OPCODE, MOD and R/M field as shown below :

OPCODE	D	W	MOD	REG	R/M
Ist BYTE			2nd BYTE		

LOW BYTE OF DISPLACEMENT
3rd BYTE

HIGHER BYTE OF DISPLACEMENT
4th BYTE

For example : MOV 1234 [BP], DX.

5. Five Byte and Six Byte instructions. These types of instructions requires 5 or 6-bytes for memory. The first two bytes contains the information regarding OPCODE, MOD and R/M fields. The remaining bytes contains two bytes of displacement and one byte of data or 2-byte of data.

OPCODE	D	W	MOD	REG	R/M
1st BYTE			2nd BYTE		

LOWER BYTE OF DISPLACEMENT	HIGHER BYTE OF DISPLACEMENT
3rd BYTE	4th BYTE

LOWER BYTE OF DATA	HIGHER BYTE OF DATA
5th BYTE	6th BYTE

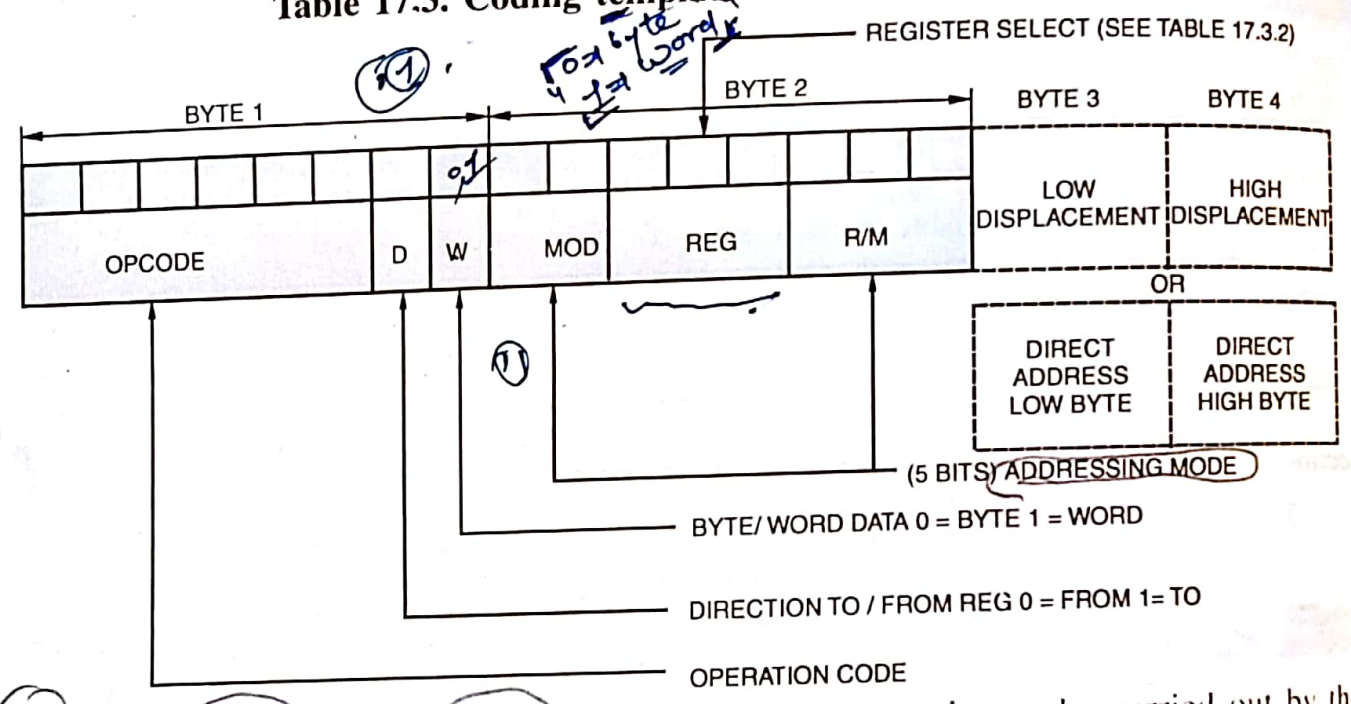
For example : MOV BP [SI + 500], 9298 H

The machine code will be

C78200 059298

17.12.2. Explanation of Instruction Template. Table 17.3 shown below is explained as follow :

Table 17.3. Coding template for 8086 instructions



(A) The opcode occupies six bits and it defines the operation to be carried out by the instruction.

(B) Register Direct bit (D) occupies one bit. It defines whether the register operand in byte second is the source or destination operand.

D = 1 Specifies that the register operand is the destination operand.

D = 0 Indicates that the register is a source operand.

(C) Data size bit (W) defines whether the operation to be performed is an 8 bit or 16 bit data.

The second byte of the instruction usually identifies whether one of the operands is in memory or whether both are registers.

This byte contains 3 fields. These are the mode (MOD) field, the register (REG) field and the Register/Memory (R/M) field.

Table 17.3.1.

MOD (2 bits)	Interpretation
00	Memory mode with no displacement follows except for 16 bit displacement when R/M=110
01	Memory mode with 8 bit displacement
10	Memory mode with 16 bit displacement
11	Register mode (no displacement)

Register field occupies 3 bits. It defines the register for the first operand which is specified as source or destination by the D bit.

Table 17.3.2.

REG	W = 0	W = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

The R/M field occupies 3 bits. The R/M field alongwith the MOD field defines the second operand as shown below :

MOD 11 Register to Register Mode :

Table 17.3.3.

R/M	W=0	W=1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Effective Address Calculation :

Table 17.3.4.

R/M	MOD = 00	MOD 01	MOD 10
000	[BX] + [SI]	[BX] + [SI] + D_8	[BX] + [SI] + D_{16}
001	[BX] + [DI]	[BX] + [DI] + D_8	[BX] + [DI] + D_{16}
010	[BP] + [SI]	[BP] + [SI] + D_8	[BP] + [SI] + D_{16}
011	[BP] + [DI]	[BP] + [DI] + D_8	[BP] + [DI] + D_{16}
100	[SI]	[SI] + D_8	[SI] + D_{16}
101	[DI]	[DI] + D_8	[DI] + D_{16}
110	Direct address [D_{16}]	[BP] + D_8	[BP] + D_{16}
111	[BX]	[BX] + D_8	[BX] + D_{16}

In this Table, D_8 = 8 bit displacement

D_{16} = 16 bit displacement

In the above, encoding of the R/M field depends on how the mode field is set. If MOD = 11 (register to register mode), this R/M identifies the second register operand.

MOD selects memory mode, then R/M indicates how the effective address of the memory operand is to be calculated. Bytes 3 through 6 of an instruction are optional fields that normally contain the displacement value of a memory operand and/or the actual value of an immediate constant operand.

Summary of all addressing modes is as shown below in Table 17.3.5 :

Table 17.3.5.

MOD/RM	Memory Mode (EA Calculation)			Register Mode	
	00	01	10	W = 0	W = 1
000	[BX] + [SI]	[BX] + [SI] + D_8	[BX] + [SI] + D_{16}	AL	AX
001	[BX] + [DI]	[BX] + [DI] + D_8	[BX] + [DI] + D_{16}	CL	CX
010	[BP] + [SI]	[BP] + [SI] + D_8	[BP] + [SI] + D_{16}	DL	DX
011	[BP] + [DI]	[BP] + [DI] + D_8	[BP] + [DI] + D_{16}	BL	BX
100	[SI]	[SI] + D_8	[SI] + D_{16}	AH	SP
101	[DI]	[DI] + D_8	[DI] + D_{16}	CH	BP
110	Dd_{16}	[BP] + D_8	[BP] + D_{16}	DH	SI
111	[BX]	[BX] + D_8	[BX] + D_{16}	BH	DI

17.12.3. Segment Over-ride Prefix. Whenever we are providing the base address from the segment register other than the default segment, then we have to provide the Segment over-ride prefix byte to start with. The Segment over-ride prefix byte is

0	0	1	S	R	1	1	0

SR	Segment Register
00	ES
01	CS
10	SS
11	DS

Example 17.11. *Give the machine code for MOV CH, BL.*

This instruction transfers 8 bit content of BL into CH.

Solution. The 6 bit op-code for this instruction is 100010_2. D bit indicates whether the register specified by the REG field of byte 2 is a source or destination operand.

D = 0 indicates BL is a source operand.

W = 0 byte operation

In byte 2, since the second operand is a register MOD field is 11_2.

The R/M field = 101 (CH)

Register (REG) field = 011 (BL)

Hence the machine code for MOV CH, BL is

10001000 11 011 101

Byte 1 Byte 2

= 88DD H

Example 17.12. *Give the machine code for SUB BX, [DI].*

Solution. This instruction subtracts the 16 bit content of memory location addressed by DI and DS from BX. The 6 bit op-code for SUB is 001010_2.

D = 1 so that REG field of byte 2 is the destination operand. W=1 indicates 16 bit operation.

MOD = 00

REG = 011

R/M = 101

So, the machine code is 0010 1011 0001 1101

2 B 1 D

= 2B1D H

Example 17.13. *Code for MOV 1234 [BP], DX.*

Solution. Here we have specify DX using REG field, the D bit must be 0, indicating the DX is the source register. The REG field must be 010 to indicate DX register. The W bit must be 1 to indicate it is a word operation. 1234 [BP] is specified using MOD value of 10 and R/M value of 110 and a displacement of 1234 H. The 4 byte code for this instruction would be 89 96 34 12 H.

Opcode	D	W	MOD	REG	R/M	LB Displacment	HB Displacement
100010	0	1	10	010	110	34 H	12 H

Example 17.14. *Code for MOV DS : 2345 [BP], DX.*

Solution. Here we have to specify DX using REG field. The D bit must be 0, indicating that DX is the source register. The REG field must be 010 to indicate DX register. The W bit must be 1 to indicate it is a word operation. 2345 [BP] is specified with MOD = 10 and R/M = 110 and displacement = 2345 H.

Whenever BP is used to generate the Effective address (EA), the default segment would be SS. In this example, we want the segment register to be DS, we have to provide the Segment over-ride prefix byte (SOP byte) to start with. The SOP byte is 001 SR 110, where SR value is provided as per table shown below :

SR	Segment register
00	ES
01	CS
10	SS
11	DS

To specify DS register, the Segment over-ride prefix byte would be 0011 1110 = 3E H. Thus the 5 byte code for this instruction would be 3E 89 96 45 23 H.

SOP	Opcode	D	W	MOD	REG	R/M	Lower byte displacement	Higher byte displacement
3E H	1000 10	0	1	10	010	110	45	23

Suppose we want to code MOV SS : 2345 [BP], DX. This generates only a 4 byte code, without SOP byte, as SS is already the default segment register in this case.

Example 17.15. *Give the instruction template and generate code for the instruction ADD FABE [BX] [DI], DX (code for ADD instruction is 000000).*

Solution. ADD FABE [BX] [DI], DX

Here we have to specify DX using REG field. The bit D is 0, indicating that DX is the source register. The REG field must be 010 to indicate DX register. The W must be 1 to indicate it is a word operation. FABE [BX + DI] is spec ed using MOD value of 10 and R/M value of 001 (from the summary table). The 4 byte code for this instruction would be

Opcode	D	W	MOD	REG	R/M	16 bit displacement	
000000	0	1	10	010	001	BE H	FA H

So code = 0191BEFA H.

Example 17.16. *Give the instruction template and generate the code for the instruction MOV AX, [BX].*

Solution. (Code for MOV instruction is 100010)

AX destination register with D = 1 and code for AX is 000 [BX] is specified using 00 Mode and R/M value 111

It is a word operation.

So code will be

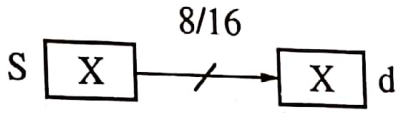
Opcode	D	W	MOD	REG	R/M
100010	1	1	00	000	111

So code for MOV AX, [BX] = 8B07 H

17.13. DETAILED DESCRIPTION OF 8086 INSTRUCTIONS

(A) DATA TRANSFER INSTRUCTIONS :

(*i*) **MOV d, S** (Move source data into destination)



S-SOURCE d-DESTINATION

The different combinations of Destination and Source is given below :

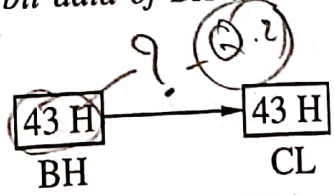
Destination	Source
Register (R)	Register
Register	Memory (M)
Memory	Register
Register/Memory (R/M)	Data
Segment Register	Register/Memory (R/M)
Register/Memory (R/M)	Segment Register.

The different 8 bit registers are AH, AL, BH, BL, CH, CL, DH, DL.

The different 16 bit registers are AX, BX, CX, DX, BP, SP, SI, DI.

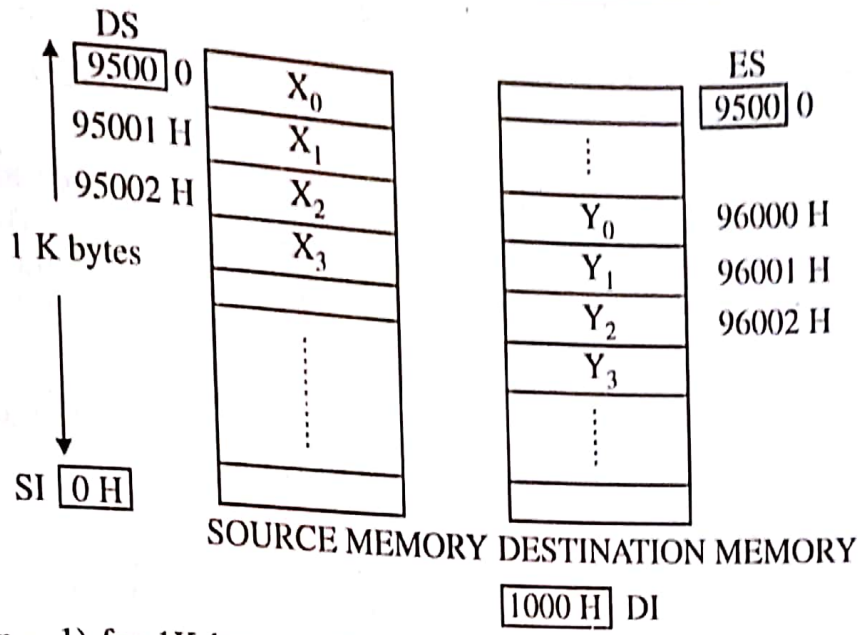
Example 17.17. *Transfer 8 bit data of BH into CL.*

Solution. MOV CL, BH



Example 17.18. *Transfer16 bit number of AX into BP.*

Solution.



Count (n – 1) for 1K bytes = 01111111111 = 3FF H

Program :

```
        MOV AX, 9500 H
        MOV DS, AX
        MOV SI, 0 H
        MOV AX, 9500 H
        MOV ES, AX
        MOV DI, 1000 H
        MOV CX, 3FF H
        CLD
REPZ : CMPSB
        HLT/INT3
```

17.14. ASSEMBLER DIRECTIVES

When we are performing the programming in machine language, its advantage is that the memory control is managed by the programmer. Its disadvantage is that programming, coding and resource management techniques are more tedious.

The assembly language programming is simpler as compared to machine language programming. The instruction mnemonics are directly written in the assembly language programs. The programs are now more readable to users than the machine language programs. The main improvement is assembly language over machine language is that the address values and constants can be identified by the labels. If the labels are suggestive then certainly the program will become more understandable and each time the programmer will not have to remember the different constants and the addresses at which they are stored throughout the programs.

An Assembler is a program used to convert an assembly language program into the equivalent machine codes. The assember decides the address of each label and substitutes the values for each of the constants and variables. It then forms the machine code for the

mnemonics and data in the assembly language program.

There are two types of Assemblers :

(1) One pass Assembler

(2) Two pass Assembler

One pass assembler. It scans the whole program only once and each instruction will be replaced by its equivalent binary code and stored in memory. The drawback of one pass assembler is that it cannot take forward jump. One pass assembler can assemble jump with backward address (label). It cannot assemble jump with forward address (label). This drawback can be eliminated in two pass assembler.

Two pass assembler. It scans the program twice. In first pass it allocates memory locations and simultaneously prepare a look up table for labels.

Label	Address
L_1	....
L_2	....
L_3	..

In second pass it will replace all labels by their equivalent address. Assembler generates two files on the hard disk. The first file called as *object file* and other is *assembler list file.*

Assembler directives. It is instruction for assembler regarding assembly of program. They are also called as *Pseudo instructions*. Various assembler directives are as given below :

(*a*) **END : End of Program.** The END directive marks the end of an assembly language program. When the assembler comes across this END directive, it ignores the source line available later on. END statement should be last statement in the file.

(*b*) **EQU : Equate :** The directive EQU is used to assign a label with a value of symbol. The use of this directive is just to reduce the recurrence of the numerical values or constants in a program code. Using the EQU directive, even an instruction mnemonic can be assigned with a label and the label can then be used in the program in place of that mnemonic.

For example :

```
LABEL  EQU 0900 H
INBUF  EQU  ADD
```

The first statement assign that the constant 900 H with the label LABEL, while the second statement assigns another label INBUF with mneomnic ADD.

(*c*) **ORG : Origin :** The ORG directive directs the assembler to start the memory allotment for particular segment block or code from the declared address in the ORG statement. If the ORG statement is not written in the Program, the location counter is initialised to 0000 H. If an ORG 2000 H statement is present at the starting of the code

segment of that module, then the code will start from 2000 H instead of 0000 H.

(*d*) **DB : Define Byte :** The DB directive is used to reserve byte or bytes of memory locations in the available memory. This directive directs the assembler to allocate the specified number of memory bytes to the data bytes.

For example :

DB 01 H, 02 H, 03 H, 04 H

This statement directs the assembler to reserve four memory locations and initialise them with the above specified four values.

(*e*) **DW : Define Word :** The DW directive serves the same purposes as the DB directive, but it now makes the assembler reserve the number of memory words (16 bit) instead of bytes.

DW 1234 H, 4567 H, 78AB H, 045C H.

(*f*) **DQ : Define Quardword :** This directive is used to direct the assember to define reserve 4 words (8 bytes) of memory for the specified variable.

(*g*) **DT : Define Ten Bytes :** The DT directive directs the assember to define specified variable requiring 10 bytes for its storage.

(*h*) **ASSUME : Assume logical segment name :** The ASSUME directive is used to inform the assembler, the names of logicals segments to be assumed for different segments used in the program.

(*i*) **PROC : Procedure :** The PROC directive marks the start of a named procedure in the statement. Also the types NEAR or FAR specify the type of the procedure.

NEAR

FAR

i.e., whether it is to be called by the main program located with in 64K of physical memory or not. The statement RESULT PROC NEAR marks the start of a routine RESULT, which is to be called by a program located in the same segment of memory. FAR directive is used to call the procedure by the programs from the different segment of memory.

(*j*) **PTR : Pointer :** The Pointer operator is used to declare the type of label, variable or memory operand. The operator PTR is defined by either BYTE or WORD. If the prefix is BYTE, then the particular label variable or memory operand is treated as an 8 bit quantity, while if WORD is prefix, then it is treated as a 16 bit quantity.

Example :

(*i*) MOV BL, BYTE PTR [SI]

Moves content of memory location addressed by SI (8 bit) to BL

(*ii*) MOV BX, WORD PTR [2000 H]

Moves 16 bit content of memory location 2000 H to BX

[2000 H] → BL

[2001 H] → BH

(k) SHORT : The SHORT operator indicates to the assembler that only one byte is required to code the displacement for a jump. The displacement is within –128 to +127 bytes from the address of byte next to jump opcode.

For Example : JMP SHORT LABEL

(l) PAGE AND TITLE : The PAGE and TITLE directives help to control the format of a listing of an assembled program.

PAGE : At the start of a program the PAGE directive specifies the maxium number of lines to list on a page and the maximum number of characters on a line.

FORMAT : PAGE [length], [width].

Example. PAGE 52, 132, ; 52 lines per page and 132 characters per line.

TITLE : TITLE directive to cause a title for a program to print on line 2 of each page of the program listing. Maximum 60 charcters are allowed as title.

FORMAT : TITLE text.

Example. TITLE program to find maximum number.

(m) SEGMENT AND ENDS : An assembly program in .EXE format consists of one or more segments. The start of these segments are defined by SEGMENT directive and the ENDS statement indicates the end of the segment.

```
Format :    Name      Segment [options];   Begin segment
            name      ENDS                 ; End segment
```

```
Example. CODE  SEGMENT
         CODE ENDS
```

(n) EVEN : EVEN tells the assembler to advance its location counter if necessary so that the next defined data item or label is aligned on an even storage boundary. This feature makes processing more efficient on processors that access 16 or 32 bits at a time.

Example :

```
EVEN L00KUP DW 10 DUP (0)      ; Declares the array of ten words
                               ; Starting from even address
```

(o) PUBLIC : Large programs are usually written as several separate modules. Each module is individually assembled, tested and debugged. When all the modules are working correctly, their object code files are linked together to form the complete program. In order for the modules to link together correctly, any variable name or label referred to in other modules must be declared public in the module where it is defined. The PUBLIC directive is used to tell the assembler that a specified name or label will be accessed from other modules.

Format : Public symbol [....]
Example : PUBLIC SETPT ; Makes SETPT available for other modules.

(p) **EXTRN :** THE EXTRN directive is used to inform assembler that the names or labels following the directive are in some other assembly module. For example, if you want to call a procedure which is in a program module assembled at a different time, you must tell the assembler that the procedure is EXTRN. The assembler will then put information in the object code file so that the linker can connect the two modules together. For a reference it is necessary to specify whether the label is near or far.

Note. Names and labels referred to as external inone module must be declared public.

Example :

```
CALLING PROGRAM                CALLED PROGRAM
DATA SEGMENT                   EXTRN VAR :  FAR
PUBLIC VAR                     DATA SEGMENT
VAR DW                         . .
. .                            MOV AX, VAL
DATA END                       . .
. .                            DATA ENDS
```

(q) **GROUP :** A program may contain several segments of the same type (code, data, or stack). The purpose of the GROUP is to collect them all under one name, so that they reside within one segment, usually a data segment.

Format : Name Group seg-name, . . . , Segn-name.

Example :

```
SEG GROUP SEG1, SEG2
SEG1 SEGMENT PARA 'DATA'
ASSUME DS : SEG
. . .
SEG1 ENDS
SEG2 SEGMENT PARA 'DATA'
ASSUME DS : SEG
. . .
SEG2 ENDS
```

(r) **LABEL :** The directive LABEL assigns a name to the current value in the location counter. The label directive must be followed by the definition of data type to which label is associated. If the label is used as a destination in the jump or call instruction, the label must be specified as a near or far type. When the label is used to reference the data item, the label must be specified as type byte, word or double word.

General Form :

LABEL Labelname Label type

(s) **MACRO :** The directive MACRO informs the assembler the beginning of a macro. It consists of the name of a macro followed by the keyword MACRO and macro arguments

if any. In between the directives MACRO and ENDM at the invocation of a macro the program must present which is to be activated.

General Form :

```
MacroName        Macro [Argument 1, ....., ArgumentN]
```

(t) **ENDM :** End of macro :

The directive ENDM informs the assembler the end of the macro. The directives MACRO and ENDM must enclose the code which have to be substituted at the invocation of a macro.

General Form :

```
        ENDM
```

(u) **ENDP : End of procedure :** The directive ENDP informs the assembler the end of a procedure. The directives ENDP and PROC must enclose the procedure code.

General Form :

```
Procedure Name          ENDP
```

Example :

```
ABC         ENDP
```

It is used with the name of a procedure as follows :

```
ABC         PROC
            :
            ; do all procedure code stuff here
            :
            RET
            ENDP
```

(v) **OFFSET :** The directive OFFSET informs the assembler to determine the displacement of the specified variable with respect to the base of the segment. It is usually used to load the offset of a variable into the register. Using this offset value, a variable can be referenced using indexed addressing modes.

General Form :

```
OFFSET      Variable Name
```

Examples :

```
... DATA    SEGMENT
            :
            MSG  DB     'Welcome$'
            DW   55 DUP (?)
            :
... DATA    ENDS
```

(w) EVEN : Align as Even memory address : The directive EVEN is used to inform the assembler to increment the location counter to the next even memory address if it is not pointing to even memory location already.

The 8086 processor reads a word from the memory in one bus cycle while accessing an even memory address word. It requires two bus cycles to access a word from the odd memory location. The even alignment with EVEN directive helps in accessing a series of consective memory words quickly. The directive can be used in both code and data segments which increment the location counter to the next even memory location if necessary. The use of EVEN directive in the code segment is actually replaced by the instruction NOP.

17.15. PROCEDURES AND MACROS

In some cases we need to use a group of instructions several times throughout a program, there are two ways we can avoid having to write the group of instructions each time we want to use them, one way is to write the group of instructions as a separate procedure. We can then just CALL the procedure whenever we need to execute that group of instructions. For calling the procedure we have to store the return address onto the stack. This process takes some time. If the group of instructions is big enough then this overhead time is negligible with respect to execution time. But if the group of instructions is too short, the overhead time and execution time are comparable. In such cases, it is not desirable to write procedures. For these cases, we can use macros. Macro is also a group of instructions. Each time we "CALL a macro in our program , the assembler will insert the defined group of instructions in place of the CALL". An important point here is that the assembler generates machine codes for the group of instructions each time macro is called. So there is not overhead time involved in calling and returning from a procedure. The disadvantage of macro is that it generates inline code each time when the macro is called which takes more memory.

17.15.1. Procedures : Procedure is a group of instructions stored as a separate program in the memory and it is called from the main program whenever required. The type of procedure depends on where the procedure is stored in the memory. If it is in the same code segment where the main program is stored then it is called as near procedure otherwise it is referred to as far procedure. For near procedure CALL instruction pushes only the IP register contents on the stack, since CS register contents remains unchanged for main program and procedure. But for far procedures CALL instruction pushes both IP and CS on the stack.

We often want a procedure to process some data or address variable from the main program. For processing it is necessary to pass these address variables or data, usually referred as passing parameters to the procedure. There are four ways to pass parameters to and from the procedure.

(*a*) Using registers
(*b*) Using general memory
(*c*) Using pointers
(*d*) Using stack

```
BACK :  MUL BL          ;  K = BL  I
        INC BL          ;  BL = BL + 1
        CMP BL, CL      ;  IS BL < = CL
        JNA BACK        ;  If yes then do another pass
DONE :  MOV 3000 H, AX  ;  Store AX at 3000 H.
```

17.18. PIN CONFIGURATION OF 8086

The INTEL 8086 microprocessor consists a total of 40 pins and some of which are multiplexed. Multiplexed pins are used for two or more different applications. The 8086 microprocessor is provided with one power supply (V_{CC} = +5V, Pin 40) and two ground (V_{SS} pins 1, 20) pins. The 8086 microprocessor can be operated in two different modes namely, minimum and maximum modes. The pins 24 to 31 represents different functions in these two modes.

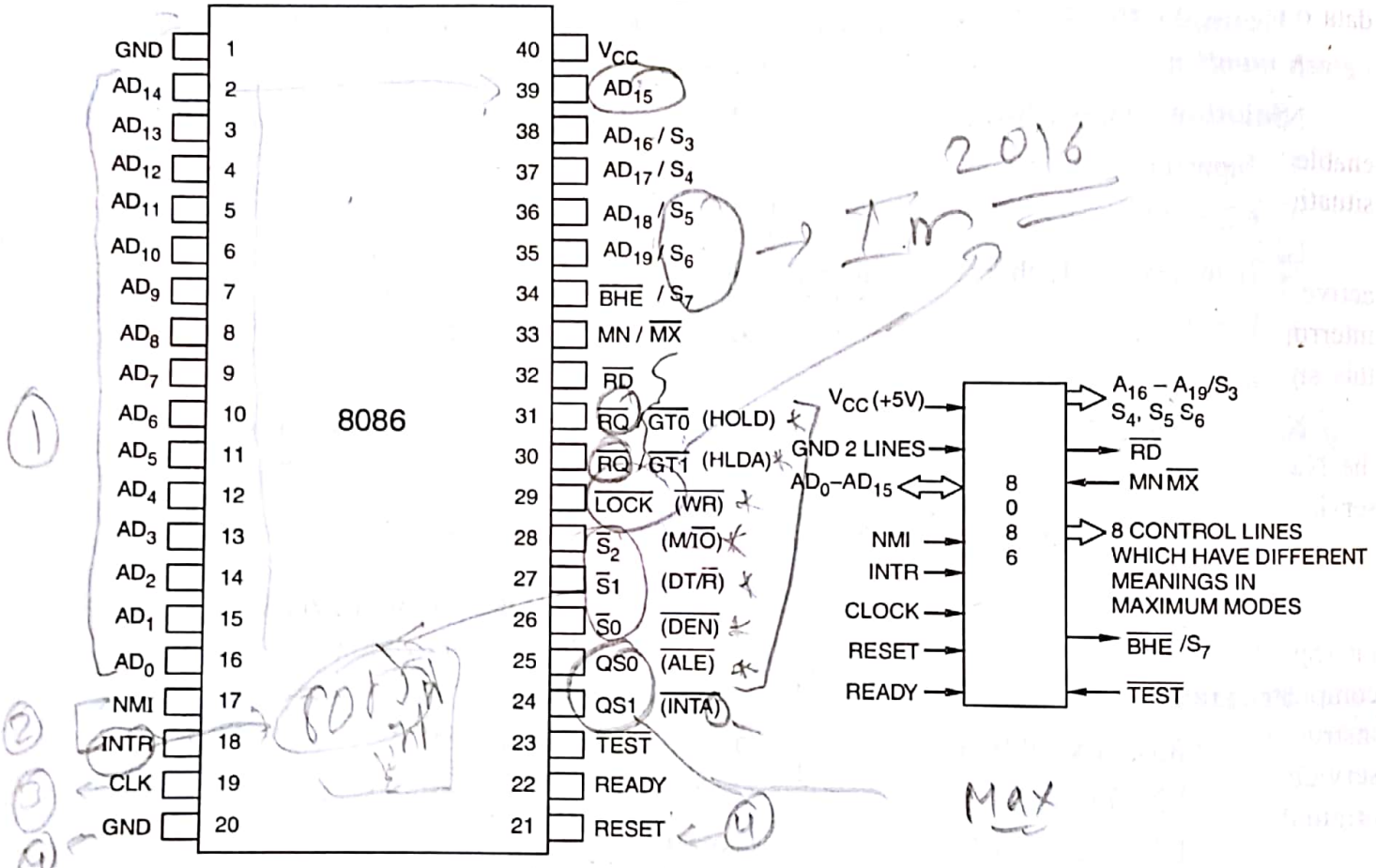


Fig. 17.21. Pin diagram of 8086 and functional diagram of 8086.

The different pins and their functions are as listed below.

AD_{15} – AD_0 (Address/Data): Pins 2 to 16 and 39.

AD_{15} – AD_0. As 8086 is a 16 bit microprocessor, it has 16 data pins, D_{15} – D_0 are bi-directional.

8086 receives 16 bits of information on these pins from memory or I/O ports, manipulates the information internally, and then sends result to memory or I/O ports on the same pins. 8086 is contains twenty address lines. The width of the address bus indicates the primary memory *i.e.*, number of address locations which can be addressed. Hence total number of memory locations that can be addressed in 8086 are 2^{20} = 1 million memory locations, i.e., the primary memory of 8086 in 1 MB. The data bus and the least significant 16 bits of address bus are shared on the same pins. This is known as multiplexing.

Important Information. The pin $AD_{15} - AD_0$ are used for data as well as the least significant 16 bits of address.

The pins are used in the following way :

1. During T_1 state of a machine cycle the microprocessor sends out address on $AD_{15} - AD_0$.

2. In the later part of a machine cycle the microprocessor sends out data or receives data on these pins.

2. NMI, INTR, (Non-maskable Interrupt, Maskable Interrupt) : Pins 17 and 18.

NMI is a positive going-edge triggered non-maskable interrupt. This can not be enabled and cannot be disabled using software instructions. It can be used in emergency situations like power supply failure and emergency shut-off etc.

NMI stands for Non maskable Interrupt. It is an input pin to the 8086, and is an active high signal. Whenever an external interrupt is given the microprocessor will be interrupted. CLI and STI instruction are used to change the interrupts. In other words, this signal cannot be masked.

NMI is a vectored interrupt. This means the 8086 knows where to branch, to service the NMI interrupt. If both NMI and INTR are activated at the same time, NMI will be serviced first. NMI has a higher priority than INTR.

INTR is a maskable interrupt request, which is level triggered.

In a microcomputer system whenever an I/O port wants to communicate with the microprocessor urgently, it interrupts the microprocessor. In such a case, the microprocessor completes the instruction it is presently executing. Then, it saves the address of the next instruction on the stack top, and branches to an Interrupt Service Subroutine (ISS), to service the interrupting I/O device. After completing the ISS, the 8086 returns to the original program, by making use of the address that was saved on the stack top.

As already mentioned in 8086 we have 2 interrupt pins. They are NMI and INTR. INTR stands for interrupt. It is an input pin to the 8086 and is an active high signal whenever an external device activates this pin, the microprocessor will be interrupted only if interrupts are enabled using a STI (set interrupt flag) instruction. If interrupts are disabled using CLI (clear interrupt flag) instruction, the microprocessor will not get interrupted even if INTR line gets activated by an external device. In other words, INTR can be masked. INTR is a non-vectored interrupt. This means the 8086 does not know where to branch, to service the interrupt. The 8086 has to be told by an external device

like a programmable interrupt controller regarding the interrupt mode.

CLK (Clock): pin 19. The clock used by 8086 is 5 MHz, where as clock used by 8086 A-2 is 8 MHz. This clock signal is generated externally by using generator IC 8284 and is applied on CLK pin.

RESET : Pin 21. When a logic 1 is sent into the 8086 on the reset pin, for atleast 4 clock cycles, the 8086 microprocessor gets reset. That is, it starts from initial state. The CS register contents will become FFFF H and IP register contents changes to 0000 H. After reset, the first instruction to be executed will be fetched from FFFF0 H.

Because BA = FFFF0

EA = 00000

Physical address = FFFF0

This address should be present in ROM.

The other segment registers *viz.* DS, ES, SS will be initialized to 0000 H. Also, the flags register contents become 0000 H.

Important Information :

When the 8086 detects the positive-going edge of a pulse on RESET, it stops all activities until the signal goes LOW. When RESET is LOW, the 8086 initializes the system as follows :

8086 Component	Content
IP	0000 H
CS	FFFF H
DS	0000 H
SS	0000 H
ES	0000 H
Queue	Empty
Flags	Clear

READY : Pin 22. This is the input pin of microprocessor, when ready input = 1, then 8086 performs normal operation. If Ready input = 0, then 8086 enters in wait state from next clock cycle. The ready pin is an active high input pin to the 8086. If it is at logic 0, the microprocessor inserts a wait state between T_3 and T_4 clock cycles.

If the ready signals is still at logic 0 at the end of the wait state, then one more wait state is introduced. The microprocessor enters T_4 state only if ready pin is active at the end of T_3, or a wait state. A wait state exists for one clock cycle, and during this time, the various signals sent out by the microprocessor remain the same as they were at the start of the wait state. The ready pin receives the input from 8284.

$\overline{\text{TEST}}$: Pin 23. 8086 has a software instruction WAIT. If WAIT instruction is given to 8086, then 8086 will check the $\overline{\text{TEST}}$ pin.

(a) If $\overline{TEST}$ = 0, then 8086 does not enter in wait state, but execute the next instruction in sequence.

(b) If $\overline{TEST}$ = 1, then 8086 enters in WAIT state and it will remain in this state till $\overline{TEST}$ become zero.

(c) This pin is used when 8086 is operating with co-processor 8087 or any other CPU.

$\overline{RD}$ (Read) : Pin 32. The $\overline{RD}$ pin is an active low output pin. Whenever the microprocessor needs to get information from a memory location or an I/O port, it activates the $\overline{RD}$ pin.

MN/$\overline{MX}$ (Minimum/Maximum mode) : Pin 33. When this pin is at +5 V, the CPU enters into minimum mode and enters into maximum mode when it is connected to ground. In simple systems with a single microprocessor, all the control signals needed in the microcomputer system can be directly generated by the 8086. In complex systems, especially those containing arithmetic coprocessor, I/O processors and multiple processors, more number of control signals are required to be generated for proper interaction among them.

This problem is solved by allowing 8086 to work in 2 different modes, called the *minimum* and *maximum modes*. This is indicated by the input pin MN/$\overline{MX}$. In simple systems, where 8086 is the only processor in the system, we connect MN/$\overline{MX}$ pin to +5 volts. In such a case, the 8086 directly generates all the control signals needed in the system. In complex systems which for example uses a co-processor, we connect MN/$\overline{MX}$ pin to ground. In this case, pins 24 to 31, a total of 8 pins will have different functions. Pins $\overline{S}_0, \overline{S}_1, \overline{S}_2$ in the maximum mode provide control signals in coded form.

Important information. From the above discussion we can come to the conclusion that the 8086 can operate in two modes. These are minimum mode (uniprocessor system with a single 8086) and maximum mode (multiprocessor, system with more than one 8086). MN/$\overline{MX}$ is an input pin used to select one of these modes. When MN/$\overline{MX}$ is HIGH, the 8086 operates in the minimum mode. In this mode, the 8086 support small, single-processor systems using a few devices that use the system bus. When MN/$\overline{MX}$ is LOW, the 8086 support multiprocessor systems. In this case, the Intel 8288 bus controller is added to the 8086 to provide bus controls and compatibility with the multibus architecture.

$\overline{BHE}/S_7$ (Bus high enable/Status line): Pin 34. During the first clock cycle, T_1 of a machine cycle, the microprocessor uses this pin to send out $\overline{BHE}$ (stands for Bus High Enable). In subsequent clock cycles the microprocessor sends out a logic 0 on S_7 as status

information. It is a spare status line. It is used by 8087 numeric coprocessor to determine whether the CPU is a 8086 or a 8088.

Important information. Physically in 8086 based microcomputer, the main memory can be viewed as two banks, each of 512K bytes maximum. The banks are called the 'Lower bank' and the 'Upper bank'. To select a location in these banks, a 19 bit address (as 2^{19}–512K) is sent by the 8086 on $A_{19} - A_1$. The lower bank contains bytes with only even addresses, like 0, 2, 4 etc. In other words, location 0 in this bank will have the address 0, location 1 will have the address 2, location 2 will have the address 4 etc. The data lines of the lower bank is connected to $D_0 - D_7$ of 8086. The lower bank is selected by A_0. (See Fig. 17.22).

The upper bank contains bytes with only odd addresses, like 1, 3, 5 etc. In other words, location 0 in this bank will have the address 1, location 1 will have the address 3, location 2 will have the address 5 etc. The data lines of the upper bank is connected to $D_8 - D_{15}$ of 8086. The upper bank is selected by $\overline{BHE}$.

A_{19}/S_6–A_{16}/S_3 : Pins 35 to 38. Pins $A_{19} - A_{16}$ provide the most significant 4 bits of memory address during the first clock cycle, T_1 of a machine cycle. In case, the 8086 is sending out address for accessing I/O ports, the information on these pins will be zeros during T_1. This is because, I/O port addresses in 8086 are only 16 bits. In subsequent clock cycles these pins send out S_6–S_3 status information. The memory in a 8086 based microcomputer can be visualized as shown below.

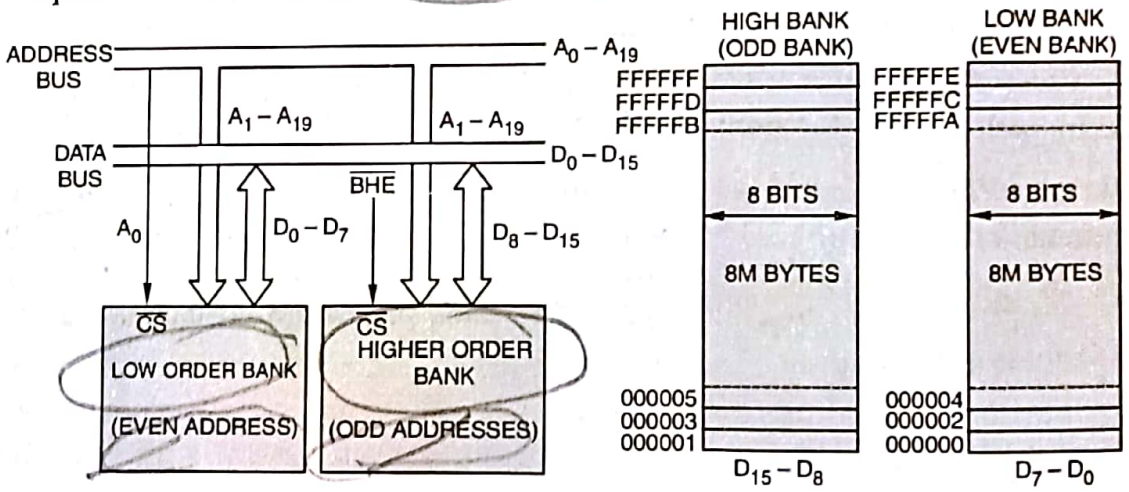


Fig. 17.22. Physical memory system of 8086 and its accessing.

As can be noticed in the figure, the number of bits stored in ever memory location is not 16 bits, but only 8 bits. In other words, every byte has a separate address. This is because, sometimes it is desired to access only a byte of information from memory. To facilitate this, the memory is having the ability to address an individual byte.

Thus two locations are needed to store a word of information. Hence, in the 1 million locations, 0.5 million words can be stored. Word 0 information is contained in memory locations 0 and 1. Next, contents of locations 2 and 3 constitute word 2, and not word 1. Word I is the contents of locations 1 and 2. As already discussed. Pins A_{16}–A_{10} provide the most significant 4 bits of memory address during the first clock cycle, T_1 of a machine

cycle. In subsequent clock cycles these pins sent out $S_3 - S_6$ status information, the detailed description is given below.

S_4 and S_3 indicate the segment register to be used for accessing data as follows :

S_4	S_3	Segment Register
0	0	ES
0	1	SS
1	0	CS
1	1	(or none in case of I/O port access)

S_5 indicates the value of the interrupt enable flag bit. S_6 will be always in the 0 state and is not used.

Minimum Mode : Pins 24 to 31 :

$\overline{\text{INTA}}$ (Interrupt acknowledge) : Pin 24 :

INTA stands for Interrupt Acknowledge.

This pin is related to the non-vectored interrupt INTR. Whenever INTR input pin is activated by an I/O port, if interrupts are enabled, and NMI is not active at the same time, the microprocessor finishes the instruction it is executing, and gives out a logic 0 on $\overline{\text{INTA}}$ pin. In response to this $\overline{\text{INTA}}$ the microprocessor receives an 8 bit interrupt type number, from an I/O port or a Programmable Interrupt Controller.

ALE (Address Latch Enable) : Pin 25 :

The 8086 sends out address information on $AD_0 - AD_{15}$ during T_1 of a machine cycle. In subsequent clock cycles of the machine cycle, the 8086 uses the same pins for transmitting or receiving data. Similarly, 8086 sends out address information on A_{19-16}/ S_{6-3} during T_1 of a machine cycle. The same pins are used later for sending out status information.

Then, how does the memory or an I/O port remember the address for the whole machine cycle, when the microprocessor sends the address for just a clock cycle ? This problem is solved by using the ALE pin.

ALE stands for Address Latch Enable. This is an output pin. The 8086 sends out a logic 1 on this pin during T_1, clock cycle of a machine cycle.

An external latch such as 74LS373 uses this signal to latch on to the information sent out by the 8086 during T_1. This latched information is available to memory or I/O ports for the whole machine cycle.

In the same way, $\overline{\text{BHE}}$ on the multiplexed signal $\overline{\text{BHE}}/S_7$ is also latched using ALE and an external latch such as 74LS373.

$\overline{DEN}$ (Data enable) : Pin 26. Whenever 8086 performs data transfer i.e. either reading or writing opereation, then 8086 gives $\overline{DEN}$ = 0, this signal is used to enable bi-directional tristate Buffer during DMA operation. 8086 data pins are to be isolated from system data bus so 8086 gives $\overline{DEN}$.

DT/$\overline{R}$ (Data Transmit or Receive) : Pin 27. If we connect more than a few devices on the data bus, the 8086 outputs are not capable of supplying enough current drive. The bidirectional buffers, also called *Transreceivers* (for Transmitter/Receiver), like Intel 8286 are used on the data bus to increase the current driving ability. The DT/$\overline{R}$ is an output pin of the 8086, which is activated in T_1 of a machine cycle.

If the 8086 outputs a logic 1 on this pin, it means the buffers are used to transmit the data from the 8086. If the 8086 outputs a logic 0 on this pin, the buffers are set up so that the 8086 receives the data from the buffers. When the microprocessor is sending out address on $AD_0 - AD_{15}$, this address should not be sent by the buffers on the data bus. Hence, there is a need to have the ability to tristate the buffer output. For this purpose the $\overline{DEN}$ output signal is used by the 8086. When 8086 outputs a logic 0 on $\overline{DEN}$, a little while after ALE is made logic 0 so that the data on the data pins are settled, the buffer output is placed on the data bus.

M/$\overline{IO}$ (Memory/Input or output) Pin 28. M/$\overline{IO}$ is an output pin. Whenever the microprocessor wants to communicate with memory, the microprocessor outputs a logic 1 on this pin. If the microprocessor wants to communicate with I/O ports it outputs a logic 0 on this pin.

$\overline{WR}$ (Write) : Pin 29. The **$\overline{WR}$** pin is an active low output pin. Whenever the microprocessor needs to write information to a memory location or to an I/O port it activates the **$\overline{WR}$** pin.

HLDA (Hold acknowledge) : Pin 30

HOLD (Hold request) : Pin 31

These two pins are used during direct memory access. The 'HOLD' is used whenever the DMA operation is required by some I/O device. The microprocessor sends acknowledge signal on the pin HLDA.

Important information. In 8086, there are no instructions to transfer data directly between memory and an I/O port. Memory and I/O port have to communicate via the processor. For example, if the data has to be moved from a memory location to an I/O port, it has to be first moved from memory to the microprocessor, and then from the microprocessor to the I/O port. This kind of data transfer involving the microprocessor is called *programmed data transfer*.

The process of an I/O port accessing memory directly, without passing data through the microprocessor, is called *Direct Memory Access (DMA)*. DMA data transfer is essential

when large amount of data has to be transferred between a fast peripheral, like disk controller and memory. A DMA controller like the Intel 8237/8257 can be used for performing DMA data transfer in a microcomputer system. The DMA controller activates the HOLD input of 8086, whenever DMA data transfer is requested by an I/O port. The microprocessor checks this input pin at the end of every machine cycle in an instruction cycle. When the 8086 finds that the HOLD pin is active, it tristates the address bus data bus and the control signals, and enters the 'hold' state. The contents of the various registers inside the microprocessor remains the same for the duration of the hold state. The microprocessor indicates to the DMA controller that it has entered the hold state, by sending out HLDA signal. Then, the DMA controller takes control over the system buses, and generates the necessary control signals, and memory addresses so that the peripheral can perform the DMA data transfer.

Maximum mode pins :

QS0, QS1, (Queue Status) : Pins 24 and 25. These signals indicate the queue status. When 8086 is in control of the local bus, these signals sent out by the 8086 allow external tracking of the instruction queue. These signals are output from the 8086. In the maximum mode, QS1 and QS0 sent out by 8086 provide the queue status as shown below :

QS1	QS0	Queue status
0	0	No instruction is taken from IQ, so all the six registers of IQ are full.
0	1	First byte (opcode) from queue.
1	0	Due to branching instruction, all six registers of IQ are cleared or flushed.
1	1	Next byte (opcode) from queue.

These signals indicate the local bus cycle status. When 8086 is in control of the local bus, the 8288 bus controller generates bus control signals using these signals sent out by the 8086. The status information sent out by S_2, S_1 and S_0 provide the bus status as shown in Fig. 17.23.

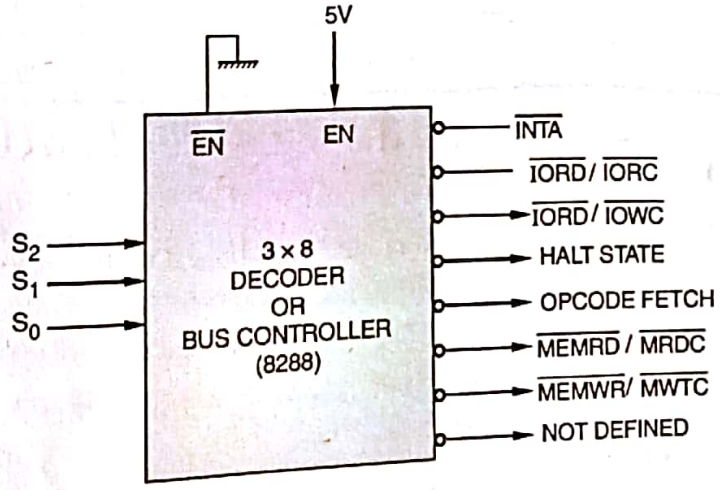


Fig. 17.23.

S_2	S_1	S_0	Bus cycle
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	Fetch
1	0	1	Read memory
1	1	0	Write memory
1	1	1	Not defined

$\overline{LOCK}$: Pin 29 :

A microcomputer system can control several microprocessors. Each microprocessor will have its own local buses for accessing local resources like local memory. The various microprocessors in the system are connected by a system bus. Thus each of the microprocessors can access global resources like disk drives or global memory. The $\overline{LOCK}$ prefix for an instruction allows a microprocessor to make sure that another microprocessor does take control of the system bus during execution of a critical instruction which uses the system bus. This is achieved by the 8086 by activating lock signal, which is one of the signals in the maximum mode of operation This signal is connected to an external bus controller, which prevents any other microprocessor from taking control of the system bus.

$\overline{RQ}/\overline{GT}0, \overline{RQ}/\overline{GT}0$ (Request/Grant) : Pin 30 and Pin 31 :

These are input as well as output pins for 8086. They are active low pins. If they are left without any connection, they will be in the logic 1 state, which is the inactive state. These pins are used for gaining control over the local bus. In 8086 if bus requests come at the same time from two different local bus masters on $\overline{RQ}0/\overline{GT}0$ and $\overline{RQ}1/\overline{GT}1$, pins, bus grant signal will be issued by the 8086 on $\overline{RQ}0/\overline{GT}0$. This is because, $\overline{RQ}0/\overline{GT}0$ has higher priority than $\overline{RQ}1/\overline{GT}1$.

17.19. DEMULTIPLEXING 8086

To use 8086 microprocessors with memory I/O interfacing devices its multiplexed buses must be demultiplexed. From the pin configuration of 8086 microprocessor we know that the multiplexed pins are $AD_0 - AD_{15}$ i.e. address data bus $A_{16}/S_3 - A_{19}/S_6$, higher order of the address bus and status signals, $\overline{BHE}/S_7$. Bus high enable and status signal S_7. All these pins must be demultiplexed for proper operation of the system.

information on the pins is passed to the outputs. After a clock cycle ALE returns to logic 0. As long as ALE is at logic 0 the latch holds the information it was holding at the time of the change to logic 0. The first latch holds A_0–A_7, D_0 –D_7. The second latch holds A_8–A_{15}, D_8–D_{15}. The third latch holds A_{16}–A_{19}, $\overline{BHE}$.

17.20. MODES OF OPERATION OF 8086

8086 can be used in two modes :

(*a*) **Minimum mode system.** If MN/$\overline{MX}$ pin of microprocessor 8086 is connected to logic '1', the processor operates in minimum mode. In this mode, there is only one microprocessor and all the control signals are given out by the microprocessor chip itself. The remaining components in the system are latches, transreceivers, memories, I/O devices and clock generator.

(*b*) **Maximum mode system.** If MN/$\overline{MX}$ pin of microprocessor 8086 is connected to GND, the processor operates in maximum mode. In the maximum mode, there may be more than one microprocessor (multiprocessor system) in the system configuration. The other component in the system Bus controller which drives the control signals using the status signals $\overline{S}_2$, $\overline{S}_1$ and $\overline{S}_0$.

17.20.1. Minimum Mode 8086 System. In a Minimum mode 8086, the microprocessor 8086 is operated in minimum mode by connecting MN/$\overline{MX}$ 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 are latches, transreceivers, clock generator, memory and I/O devices. The latches are like 74LS373 or 8282. These are used to separate the valid address from multiplexed address/data signals and are controlled by the ALE signal generated by 8086.

Transreceivers are used to separate the valid data from the time multiplexed address/data signal. They are controlled by two signals namely, $\overline{DEN}$ and DT/$\overline{R}$.

The system contains memory for the monitor and users program storage. System may contain I/O devices for communication with the processor as well as some special purpose I/O devices.

The clock generator generates the clock from 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.

Fig. 17.25 shows the typically minimum mode configuraion. As shown in Fig. 17.25, AD_0–AD_{15}, A_{16}/S_3–A_{19}/S_6, and $\overline{BHE}/S_7$ signals are multiplexed. These signals are demultiplexed by external latches and ALE signal generated by the processor. This is accomplished by using three latch ICs (Intel 8282/8283), two of them are required for a 16-bit address and three are needed if a full 20 bit address is used. The 8282 provides

non-inverting outputs while the 8283 version inverts the input data. In addition to their demultiplexing function, these chips also buffer the address lines, providing increased output drive capability. For example : low level is specified as 0.45 V maximum with a sink current of 32 mA maximum. The high level is specified as 2.4 V minimum while supplying a 5 mA maximum high load current.

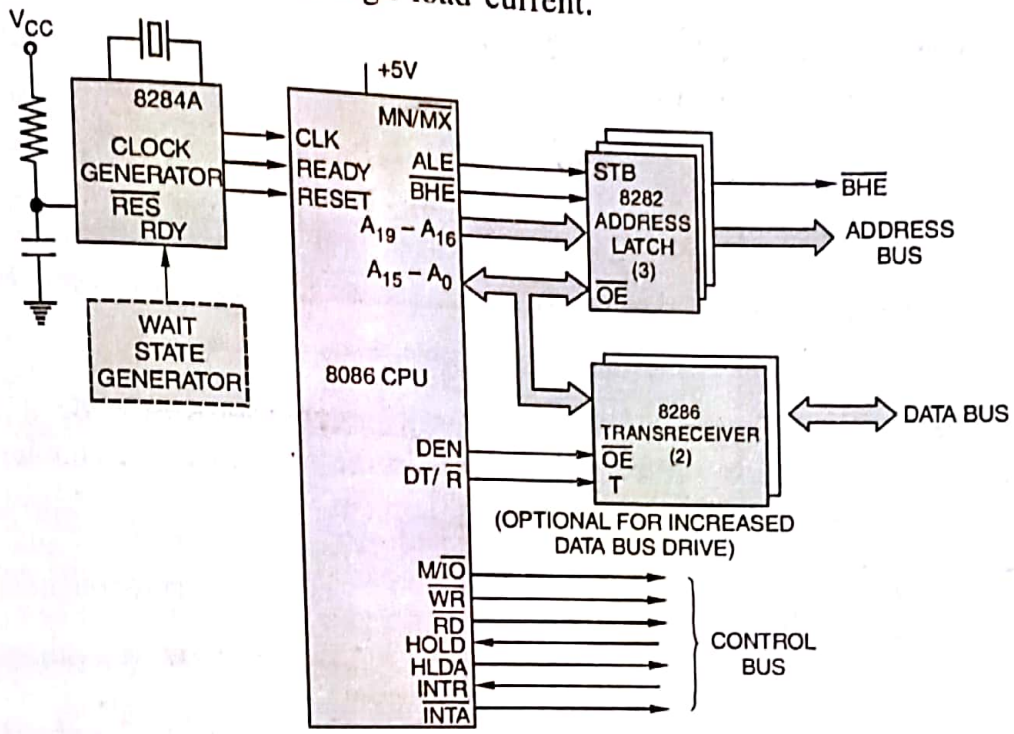


Fig. 17.25. Typical mimimum mode configuration.

17.21. CLOCK GENERATOR : 8284

The time base for the synchronization of the internal and external operations of these processors is provided by their CLK input signal, which is externally generated by IC 8284A—Clock Generator and driver IC.

The crystal frequency is divided by 3. The crystal frequency of 15 MHz is used at X_1 and X_2 pins. So the clock frequency of 5 MHz is generated.

17.21.1 Operation of 8284. The operation of clock generator is mainly divided into two sections :

(*i*) **Clock section.** The upper section of the block diagram of 8284 shows the clock and reset synchronization of 8284. When crystal is connected at X_1 and X_2 inputs of 8284, the oscillator generates the square-wave signal of same frequency which is fed to AND gate and also to an inverting buffer resulting the OSC output signal.

(*ii*) **Reset section.** Reset section consists of a Schmitt trigger buffer and a single D flip-flop circuit. The D-type flip-flop ensures the timing requirement of 8086/8088. Reset inputs are met the circuit applies the Reset signal to the microprocessor on the negative edge of each clock.

Block diagram of clock generator 8284A shown in Fig. 17.26.

RDY1 and RDY2 (Bus Ready). These inputs are provided in conjunction with the $\overline{AEN1}$ and $\overline{AEN2}$ pins to cause wait state in 8086 based system.

$\overline{AEN1}$ and $\overline{AEN2}$ (Address Enable). These pins are provided to qualify bus ready signals RDY1 and RDY2.

RESET (Reset Output). This signal is connected to RESET input pin of 8086.

CYSNC (Clock Synchronisation). This input signal is used for external synchronisation in the systems that employ multiple clocks.

F/$\overline{C}$ (Frequency/Crystal). This pin is used to select the Clocking Source for the 8284A.

EFI (External Frequency Input). The 8284 can also be driven from an external clock source.

17.22. MAXIMUM MODE CONFIGURATION

Fig. 17.28 shows the typical maximum mode configuration. In the maximum mode additional circuitry is required to translate the control signals. The additional circuitry converts the status signals ($\overline{S}_2$–$\overline{S}_0$) into the I/O and memory transfer signals. It also generates the control signals required to direct the data flow and for controlling 8282 latches and 8286 transreceiver. The Intel 8288 bus controller is used to implement this control circuitry.

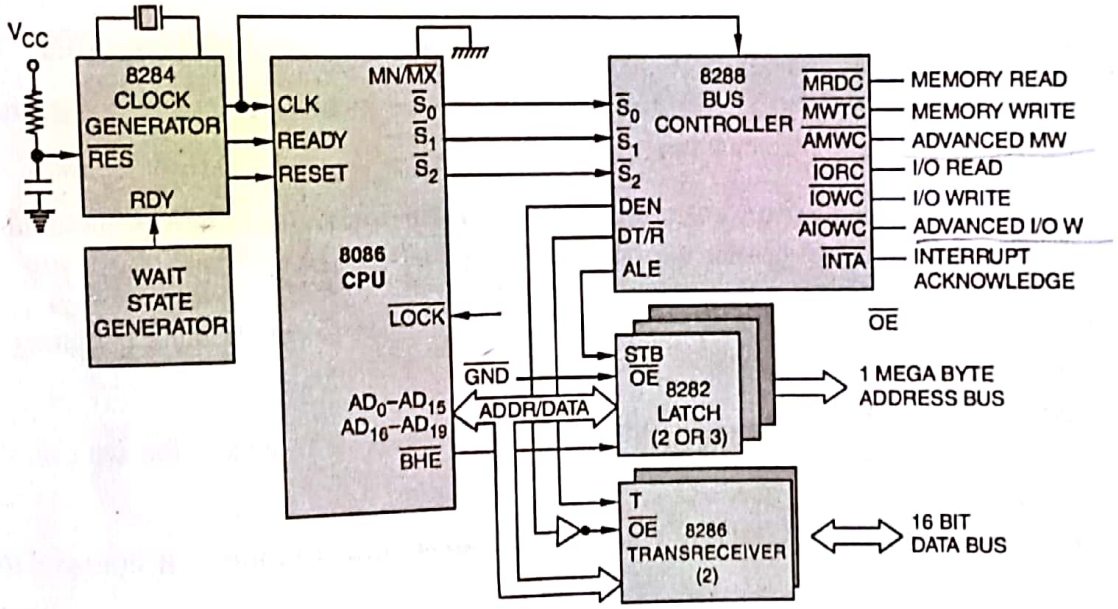


Fig. 17.28. Typical maximum mode configuration.

Fig. 17.29 shows that the 8288 is able to originate the address latch enable signal to the 8282's, the enable and direction signals to the 8286 transreceivers, and the interrupt acknowledge signal to the interrupt controller. It also decodes the $\overline{S}_2$–$\overline{S}_0$ signals to generate $\overline{MRDC}$, $\overline{MWTC}$, $\overline{IORC}$, $\overline{IOWC}$, MCE/$\overline{PDEN}$, $\overline{AEN}$, IOB, CEN, AIOWC and ANWC signals.

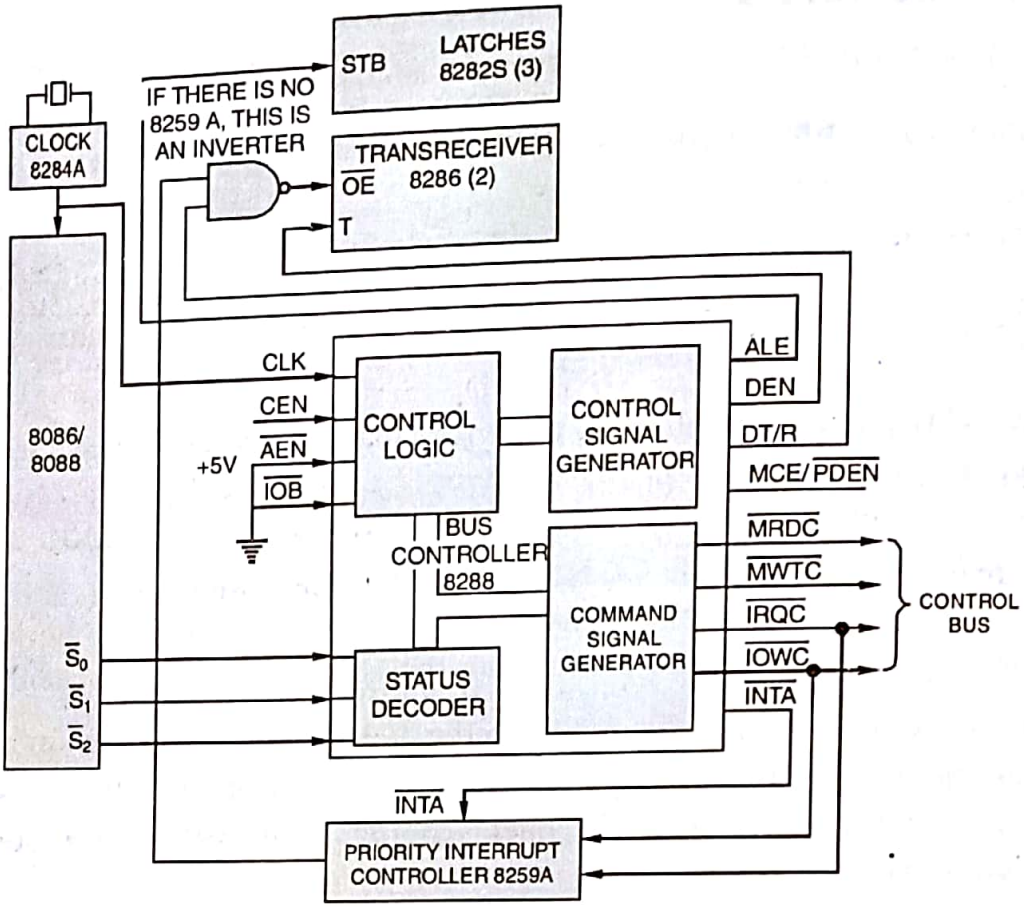


Fig. 17.29. 8288 Bus controller.

$\overline{\text{MRDC}}$ (Memory Read Command) : It instructs the memory to put the contents of the addressed location on the data bus.

$\overline{\text{MWTC}}$ (Memory Write Command) : It instructs the memory to accept the data on the data bus and load the data into the addressed memory location.

$\overline{\text{IORC}}$ (I/O Read Command) : It instructs an I/O device to put the data contained in the addressed port on the data bus.

$\overline{\text{IOWC}}$ (I/O Write Command) : It instructs an I/O device to accept the data on the data bus and load the date upto the addressed port.

MCE/$\overline{\text{PDEN}}$ (Master Cascade Enable/Peripheral Data Enable) : It controls the mode of operation of 8259.

AEN, IOB and CEN : These pins are used in multiprocessor system. With a single processor in the system, AEN and IOB are grounded and CEN is tied high.

$\overline{\text{AIOWC}}$/$\overline{\text{AMWC}}$ (Advance I/O Write Command/Advance Memory Write Command)) : These pins are used in multiprocessor system. With a single processor in the system, AEN and IOB are grounded and CEN is tied high.

17.23. 8086 BUS CYCLE

In order to communicate with external devices via the system bus for transferring data or fetching instructions, the 8086 executes a bus cycle. The 8086 basic bus cycle timing diagram is shown. The minimum bus cycle contains four CPU clock periods called T states.

1. During the first T state (T_1), the 8086 outputs the 20 bit address computed from a segment register and an offset on the multiplies address/data status bus.
2. For the second T state (T_2), the 8086 removes the address from the bus and either tristates or activates the $AD_{15} - AD_0$ lines in preparation for reading data via $A_{15} - AD_0$ lines during the T_3 cycle. In case of write bus cycle, the 8086 outputs data on $AD_{15} - AD_0$ lines. Also, during T_2, the upper four multiplexed bus lines switch from address ($A_{19} - A_{16}$) to bus cycle states (S_6, S_5, S_4, S_3). The 8086 outputs low on $\overline{RD}$ (for read cycle) or $\overline{WR}$ (for write cycle) during portions of T_2, T_3 and T_4.
3. During T_3, the 8086 continues to output staus information on the four $A_{19} - A_{16}/S_6 - S_3$ lines and will either continue to output write data or input read data to or from $AD_{15} - AD_0$ lines.
4. During T_4, the 8086 disable the command lines and the selected memory and I/O devices from the bus. Thus the bus cycle is terminated in T_4.
5. $\overline{DEN}$ and DT/$\overline{R}$ pins are use by 8286/8287 transreceiver in a minimum system.

During the read cycle, the 8086 outputs $\overline{DEN}$ low during part of T_2 and all of T_3 cycles. This signal can be used to the 8086/8287 transreceiver

17.23.1. Minimum Mode Timing Diagrams. The Timing diagram for 8086 minimum mode input and output timing diagrams are shown in Fig. 17.30 (a) and (b) respectively.

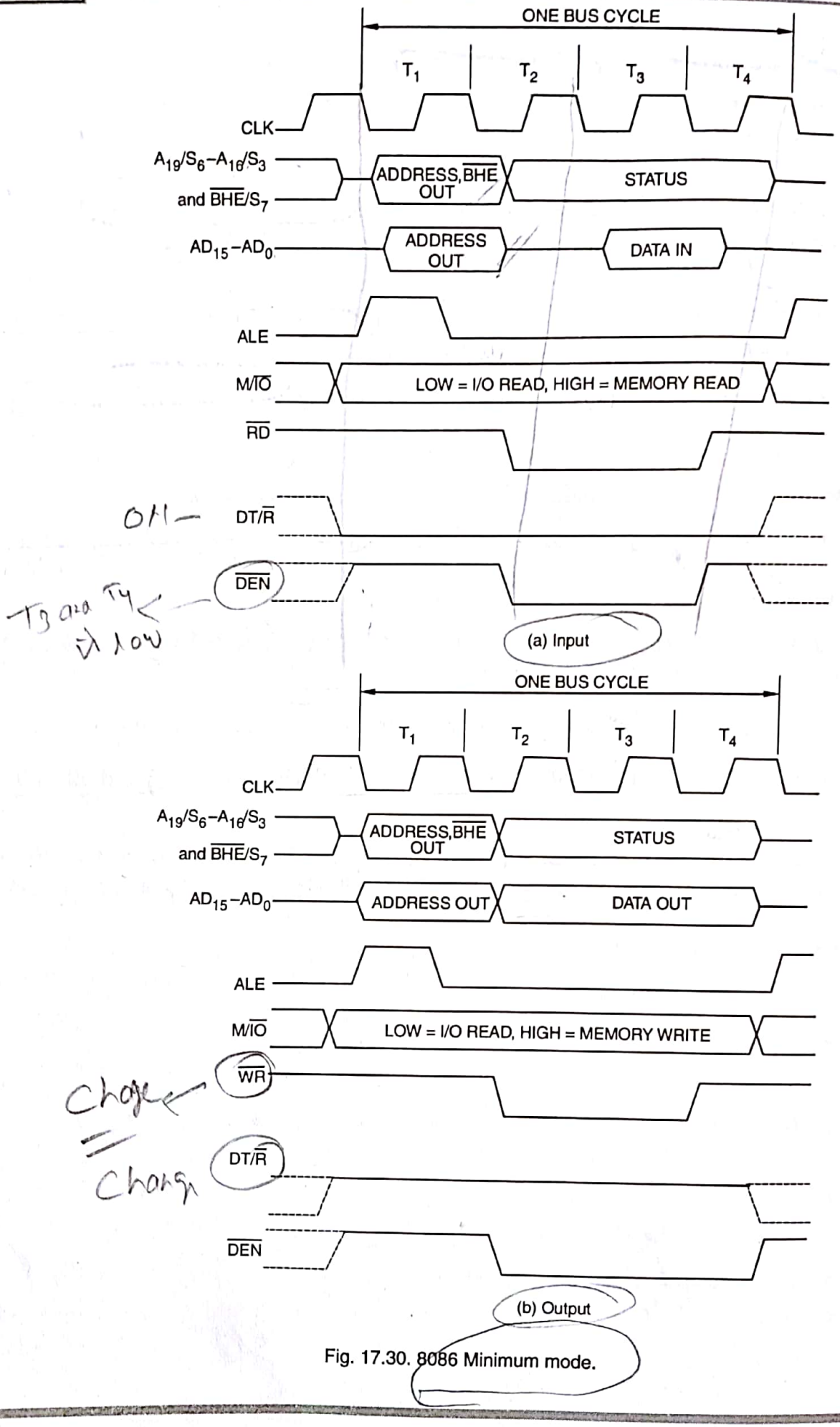
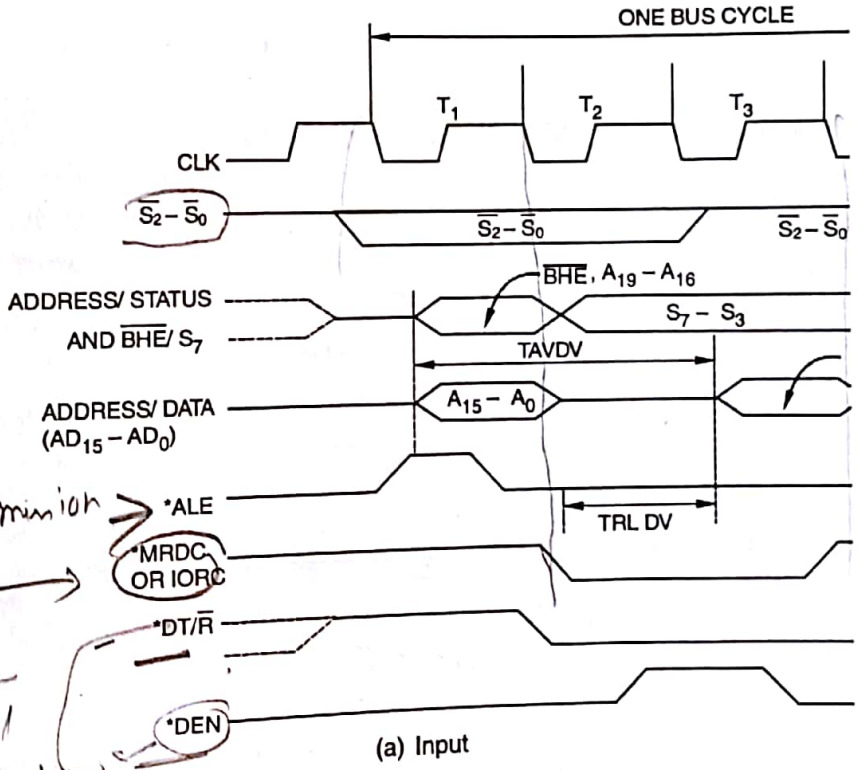


Fig. 17.30. 8086 Minimum mode.

17.23 2. Maximum Mode Timing Diagram. The Timing diagrams of input and output transfers are shown in Fig. 17.31 (a) and (b) respectively.

These are explained in steps :

1. $\overline{S}_0, \overline{S}_1, \overline{S}_2$ are set at the beginning of Bus cycle on detecting the change on passive state $\overline{S}_0 = \overline{S}_1 = \overline{S}_2 = 1$, the 8288 bus controller will output a pulse on its ALE and apply a required signal to its DT/$\overline{R}$ pin during T_1.
2. In T_2, 8288 will set DEN = 1, thus enabling transreceiver, and for an input it will activate $\overline{MRDC}$ or $\overline{IORC}$. This signals are activated until T_4. For an output, the $\overline{AMWC}$ or $\overline{AIOWC}$ is activated from T_2 to T_4 and $\overline{MWTC}$ or $\overline{IOWC}$ is activated from T_3 to T_4.
3. The status bits $\overline{S}_0$ to $\overline{S}_2$ remain active until T_3, and become passing T_3 and T_4.
4. If ready input is not activated before T_3, wait state will be inserted between T_3 and T_4.



(a) Input

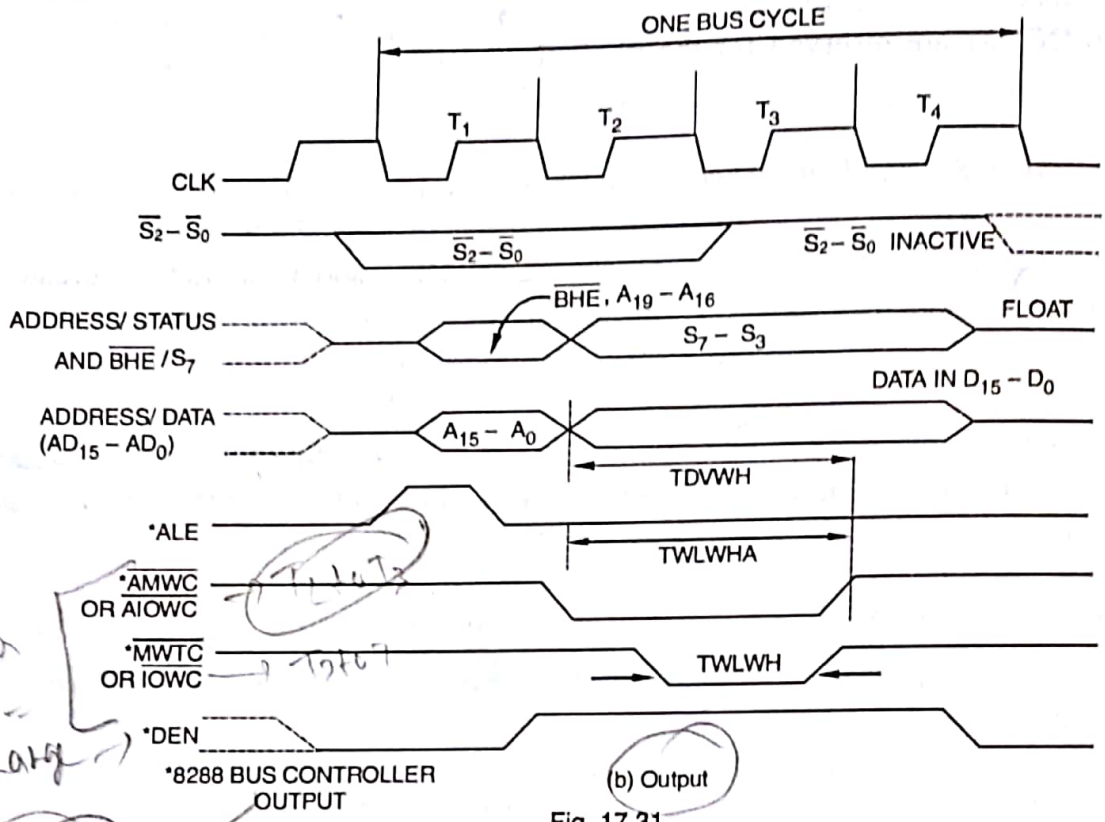


Fig. 17.31.

17.24. COMPARISON BETWEEN 8086 AND 8088

The 8088 with an 8 bit external data bus has been designed for internal 16 bit processing capability. While fetching or writing the 16 bit data, the task is performed in two consecutive bus cycles. The 8088 may take more time for execution of a particular task as compared to 8086. The changes in 8086 as compared to 8088 are as follows :

1. The instruction queue is of 4 bytes in 8088, whereas the 8086 queue contains 6 bytes.
2. The overall execution time of instructions in 8088 is affected by the 8 bit external data bus. All the 16 bit operations now require additional 4 clock cycles.
3. No use of $\overline{BHE}$ pin in 8088.
4. $\overline{SS0}$ pin is introduced in 8088 instead of $\overline{BHE}$ pin in 8086.
5. IO/$\overline{M}$ pin in 8088 whereas M/$\overline{IO}$ in 8086.

17.25. INTERRUPTS OF 8086

Interrupts of 8086 are classified mainly into two types :

(*a*) Software Interrupts

(*b*) Hardware Interrupts.

Software Interrupts :

INT n : There are 256 Software interrupt instructions *i.e.* INT 0 to INT 255, out of which

(*i*) INT 0 to INT 4 are reserved for dedicated (fixed) operation.
(*ii*) INT 5 to INT 31 are reserved for the particular operation of the system.
(*iii*) INT 32 to INT 255 are used by the user.

17.26. DEDICATED SOFTWARE INTERRUPT (INT0 TO INT4)

INT 0 : This interrupt is kept reserved for division operation. When microprocessor performs division of signed/unsigned, binary number (DIV/IDIV) and if quotient is greater than the destination (AL/AX), then 8086 executes INT 0. For INT 0 microprocessor will take the new vector EA and BA of ISR from four memory location starting from $4 \times 0000 = 0000$ H.

INT 1 : It is dedicated for single stepping mode. When Trap flag TF = 1, then after executing each instruction of program, microprocessor will execute interrupt INT 1. The ISR of INT 1 is used to verify the content of different registers used in previous instruction of main program. The new vectors EA and BA for IP and CS is obtained from location $4 \times 0001 = 0004$ H. As the location belong to RAM memory, the value of EA and BA stored at these locations can execute different ISR at different locations. Single stepping mode is used for software debugging.

INT 2 : The interrupt instruction INT 2 is dedicated for non-maskable hardware interrupt NMI. When +ve edge and level signal is obtained on NMI pin, the 8086 executes the instruction INT 2. The new value of EA and BA is taken from $4 \times 0002 = 0008$ H.

INT 3 : This instruction is dedicated for Break Point Technique (BPT). BPT is used to debug the software of 8086. In single stepping mode microprocessor will execute only one instruction at a time and then execute INT 1. So single stepping mode will take more time for software debugging. In BPT 8086 executes all the instructions before INT 3 and after executing INT 3, microprocessor is interrupt. The new EA and BA is taken from $0003 \times 4 = 0012D = 000C$ H. INT 3 is also used for HLT instruction in the main program. Normally the ISR of INT 3 is used to display all the contents of different Registers, pointers, segment registers and flags.

INT 4 : This is dedicated for software instruction INTO (interrupt on overflow). If INTO instruction is given in Program and if overflow flag OF = 1, microprocessor executes INT 4. The new data of EA and BA is taken from $4 \times 0004 = 0010$ H.

If OF = 0, then microprocessor does not executes INT 4 and go to next instruction in sequence.

2015 → TSR 2

17.27. HARDWARE INTERRUPTS OF 8086

17.27.1. NMI (Non-maskable Interrupt). NMI is a positive going-edge triggered non-maskable interrupt. This cannot be enabled and cannot be disabled using Software instructions. It can be used in emergency situations like power supply failure and emergency shut-off etc.

NMI stands for Non-maskable Interrupt. It is an input pin to the 8086, and is an active high signal. Whenever an external interrupt is given the microprocessor will be

interrupted. CLI and STI instructions are used to change the interrupts. In other words, this signal cannot be masked.

NMI is a vectored interrupt. This means, the 8086 knows where to branch, to service the NMI interrupt. If both NMI and INTR are activated at the same time, NMI will be serviced first. NMI has a higher priority than INTR.

INTR is a maskable interrupt request.

17.27.2. INTR. In a microcomputer system whenever an I/O port wants to communicate with the microprocessor urgently, it interrupts the microprocessor. In such a case, the microprocessor completes the instruction it is presently executing. Then, it saves the address of the next instruction on the stack top, and branches to an Interrupt Service Subroutine (ISS), to service the interrupting I/O device. After completing the ISS, the 8086 returns to the original program, by making use of the address that was saved on the stack top.

INTR stands for interrupt. It is an input pin to the 8086 and is an active high signal whenever an external device activates this pin, the microprocessor will be interrupted only if interrupts are enabled using a STI (Set interrupt flag) instruction. If interrupts are disabled using CLI (Clear interrupt flag) instruction, the microprocessor will not get interrupted even if INTR line gets activated by an external device. In other words, INTR can be masked. INTR is a non-vectored interrupt. This means, the 8086 does not know where to branch, to service the interrupt. The 8086 has to be told by an external device like a programmable interrupt controller regarding the interrupt mode.

17.27.3. Operation of 8259 with 8086 :

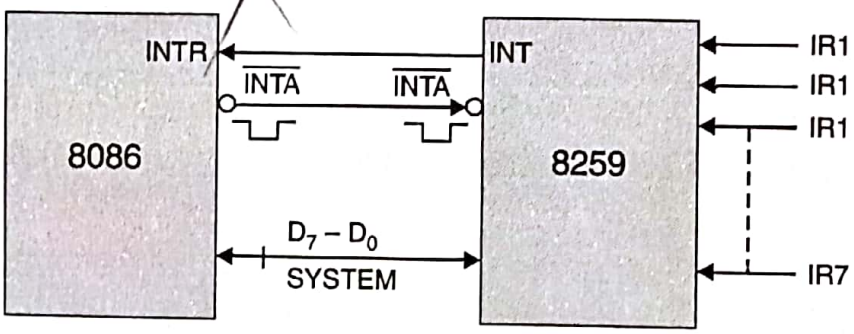


Fig. 17.32.

1. The interrupting device will give interrupt signal through interrupt levels pins.
2. 8259 will check IRR, IMR, ISR and then give INT = 1.
3. When 8086 receives INTR = 1, then 8086 gives a first pulse on $\overline{\text{INTA}}$ pin and tristate its data line. In response to first $\overline{\text{INTA}}$ pulse pin 8259 does not transfer anything on data lines. 8086 generates internally opcode of INT n instruction and decode it.
4. As INT n is double byte instruction so 8086 gives second $\overline{\text{INTA}}$ pulse. In response to second $\overline{\text{INTA}}$ pulse. 8259 gives the value of '*n*' on data lines $D_7 - D_0$.

Depending upon the value of '*n*' received microprocessor executes corresponding INT instruction *i.e.* microprocessor execute corresonding ISR.

The value of '*n*' for IR0 is defined with the help of ICW2 as given in format below :

N_7	N_6	N_5	N_4	N_3	X	X	X

$N_7 - N_3$ are the five MSB's of the value of '*n*' for IR0. Three LSB's of '*n*' for IR0 will be always zero. For successive IR level 8259 will use successive value of '*n*'.

Priorities of Interrupts :

INT0	First priority
INTn (2 to 255)	Second priority
INTO	Third priority
NMI	Forth priority
INTR	Fifth priority
INT1	Sixth priority

17.28. SEQUENCE OF INTERRUPT EXECUTION IN 8086

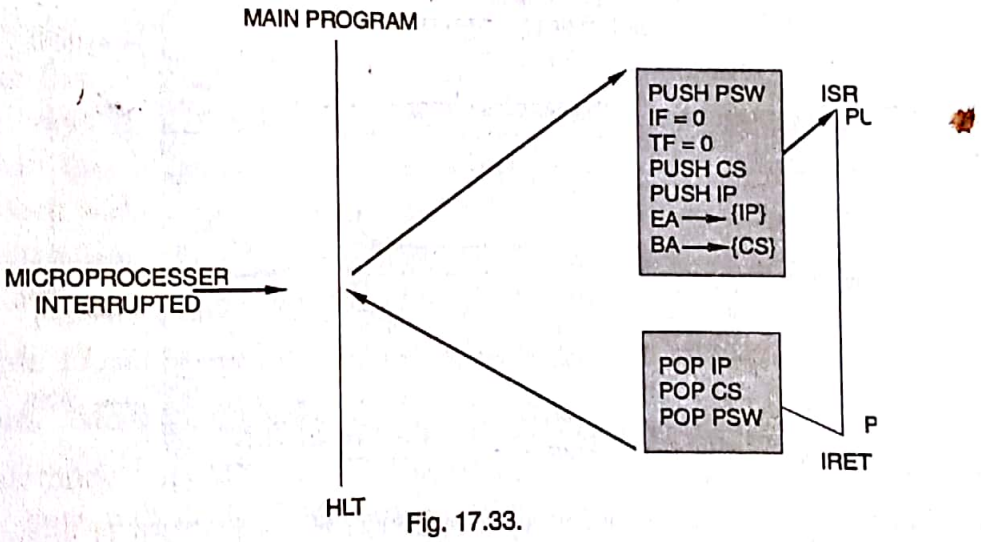


Fig. 17.33.

1. PUSH PSW (Flag saved).
2. IF = 0 (INTR disabled).
3. TF = 0 (Microprocessor will not execute ISR by single stepping).
4. PUSH CS (Address of next instruction of the main program saved in stack).
5. Depending upon value of n microprocessor will calculate Physical address of memory location using the formula PA = (4 × n) H. The 16 bit data of first two memory location is transferred to Instruction pointer and 16 bit data of next two memory location is transferred to CS. So microprocessor will execute ISR whose Effective address and BA is transferred into (IP) and (CS).