(ii) The switches are connected to buffer and LED's to latch. When a switch is pressed that input to buffer will contain logic 1 other wise it will contain logic 0. Similarly for LED's connected. When output of latch is logic 1, the LED will be OFF and when output is logic 0 the LED will be ON.

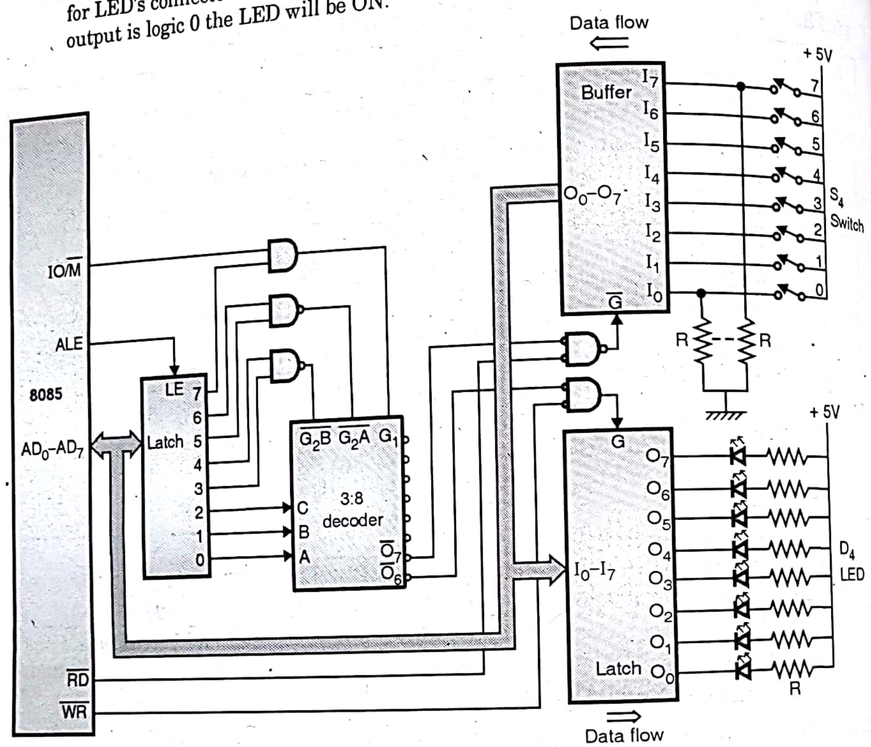


Fig. P. 10.18.1 : Buffer and latch interfacing in I/O mapped I/O

Program

Label	Instruction	Comments
UP :	IN F9 H	Take data from switches to accumulator
	CMA	Complement A.
	OUT F8 H	Send data to LED's from accumulator
	JMP UP.	Go to up.

IN F9 will take data from input device which will contain the information of switches. But the same data we can't use for LED's as LED's are operated in opposite voltage levels. So we have to complement the switch data and then send it to output device. Program control will remain in same loop so that whenever a key is pressed it will be displayed on LED's.

❑❑❑

CHAPTER

11

8086 Architecture

11.1 Introduction

- A microprocessor is a semiconductor chip that implements the central processor of a computer. The microprocessor works as a brain of a computer. It consists of an arithmetic and logic unit (ALU) and a control unit.
- The microprocessors are usually characterised by speed, word length, architecture and instruction set.
- This central processing unit built into single chip is called as ***Microprocessor***.

In this chapter we will study features of 8086, detailed architecture of INTEL'S 8086 microprocessor and pin configuration.

11.1.1 Comparison of 8085 and 8086

Q. Give Comparison between 8085 and 8086.

Sr. No.	Parameter	8085	8086
1.	Size	It is an 8 bit Microprocessor	It is 16 bit Microprocessor
2.	Address Bus	Its address bus is of 16 bits	Its address bus is of 20 bits
3.	Memory	It can access upto 2^{16} = 65,536 bytes i.e. 64 Kbytes of memory	It can access upto 2^{20} = 1,048,576 bytes i.e. 1 MB of memory
4.	Instruction Queue	It does not have an instruction queue.	It has a 6 byte instruction queue
5.	Pipelining	It does not support pipelined architecture	It supports 2 stage pipelined architecture.
6.	Multiprocessor Support	It does not have multiprocessing support.	It supports multiprocessing. It has compatibility with further processors like Intel 80386, 80486 and Pentium.
7.	I/Os	It can address 2^8 = 256 I/Os.	It can address 2^{16} = 65,536 I/Os.
8.	Coprocessor Interface	It does not have coprocessor interface.	It has coprocessor interface. Coprocessor 8087 can be interfaced with 8086.
9.	Arithmetic support	It only supports integer and decimal arithmetic.	It supports integer, decimal and ASCII arithmetic.

Sr. No.	Parameter	8085	8086
10.	Multiplication and Division	It does not have instruction that computes the multiplication and division operation.	It supports instructions that compute the multiplication and division operation.
11.	Check speed	It operates on 3 MHz, 5 MHz or 6 MHz. i.e. it operates on low clock speed.	It operates on 5 MHz, 8 MHz or 10 MHz. i.e. it operates on high clock speed.
12.	External Hardware	It requires less external hardware.	It requires more external hardware.
13.	Operating modes	It supports single operating mode.	It supports two different operating modes : maximum mode and minimum mode.
14.	Addressing modes	It supports 5 addressing modes.	It supports the addressing modes supported by microprocessor 8085. It also supports some additional addressing modes.
15.	Cost	Its cost is low.	Its cost is high.
16.	Multiplexed pins	It has very few multiplexed pins.	It has number of multiplexed pins than 8085.
17.	Memory Segmentation	The memory spaces not segmented.	The memory space is segmented.
18.	Interrupts	8085 provides 8 software vectored interrupts and five hardware interrupts.	8086 has 256 software vectored interrupts and 2 hardware interrupts.

11.2 Features of 8086

The features of a processor can be divided into three broad groups viz. basic features, special features and miscellaneous features.

11.2.1 Basic Features of 8086

- Processor size
- Speed of processor
- Address bus size for memory
- Address bus size for I/O

(1) It is a 16-bit processor. This implies that
(a) It has a 16-bit ALU that can perform 16-bit operation simultaneously.
(b) It has 16-bit registers and internal data bus.
(c) It has 16-bit external data bus.

(2) It has three versions based on the basis of frequency of operation.
(i) 8086 → 5 MHz.
(ii) 8086-2 → 8 MHz.
(iii) 8086-1 → 10 MHz.

Q. What is memory addressing capacity of 8086 ?

(3) 8086 has 20-bit address lines to access memory, hence it can access
2^{20} = 1 MB memory locations

(4) It has 16-bit address lines to access I/O devices, hence it can access
$2^{16} = 2^6 \times 2^{10} = 64 \times 1$ K = 64 K I/O locations

11.2.2 Special Features

Q. Explain how throughput of 8086 microprocessor is increased.

- 8086 is a pipelined **processor**
- 8086 can operate in 2 modes
- 8086 uses **memory banks**
- 8086 uses memory segmentation

1) **8086 is a pipelined processor**
- It uses a two stage pipelining i.e. **Fetch stage** that pre-fetches up to 6 bytes of instructions stores them in the queue and **Execute stage** that executes these instructions.
- Pipelining improves the performance of the processor i.e. the operations are faster.

2) **8086 can operate in 2 modes**
a. **Minimum mode** → A system with only 1 processor i.e. 8086.
b. **Maximum mode** →. A system with 8086 and other processors like 8087-(Math Co-processor), 8089-(IO processor) or multiple 8086 processors

3) **8086 uses memory banks**
- The 8086 uses a memory banking system i.e. the entire data is not stored sequentially in a single memory of 1 MB but the memory is divided into two banks of 512KB each.
- The banks are called Lower bank (or even bank, because it stores the data bytes at even locations i.e. 0, 2, 4....) & Higher Bank (or odd bank, because it stores the data bytes at odd locations i.e. 1, 3, 5...).
- The benefit of this is that 16-bit data can be accessed in a single access even though the memory chip can store only 8-bit at a location

4) **8086 uses memory segmentation**
- A 16-bit address in an instruction or a 16-bit address in a register can access a memory location, although 8086 has 20 address lines. This is made possible using the concept of Segmentation that divides the memory into logical components.
- Here the memory is divided into 16 segments of a capacity of 2^{16} (= 65536 B = 64 KB) each and is used as: Code, Stack, Data and Extra Segment.

11.2.3 Miscellaneous Features

- Interrupts
- Instruction set
- Registers
- Data size for ALU

(1) **It has 256 vectored interrupts :** There are also non-vectored interrupts in 8086, but they are routed to one of these interrupts.

(2) It has 14, 16-bit registers.

(3) It has a powerful instruction set, that supports multiply and divide operations also. (These operations were not possible in the processors earlier to 8086).

(4) 8086 can perform operations on bit, byte (8-bit), word (16-bit) or a string (block of data) types of data.

11.3 8086 Internal Architecture

Q. Draw and explain the architecture of 8086.

- Before talking about programming, we need to discuss the special features and internal architecture of Intel 8086. Fig. 11.3.1 shows the block schematic of the internal structure of Intel 8086.

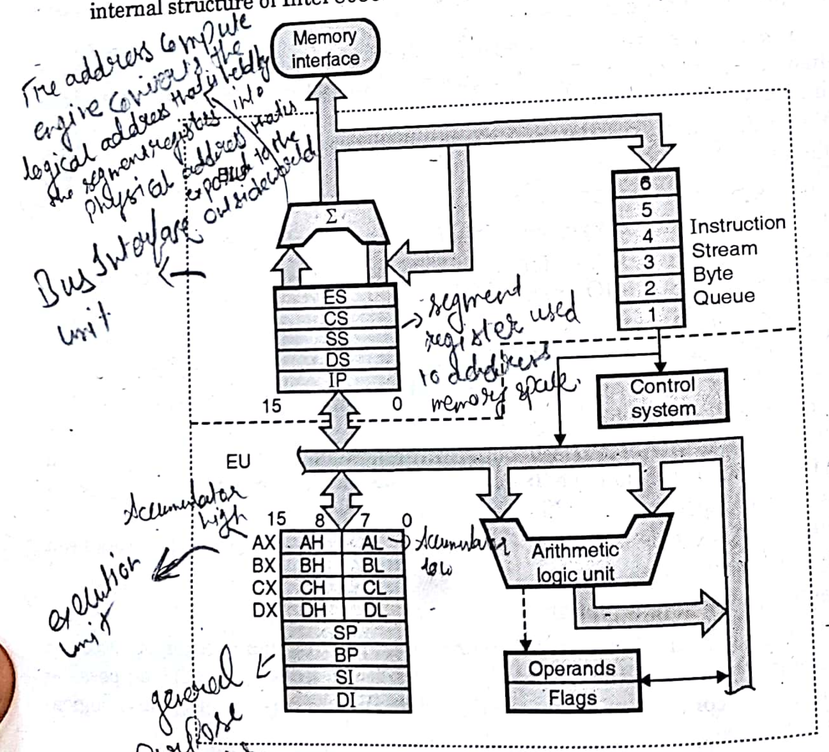


Fig. 11.3.1 : 8086 internal architecture

- As shown in Fig. 11.3.1, the 8086 CPU is divided into two units viz :
  - The bus interfacing unit (BIU)
  - The execution unit (EU).

These two units are completely independent of each other and they share the work of CPU. Such a work division speeds up the processing and reduces the processing time. This is called as pipelining.

The work of 8086 CPU is divided between BIU and EU in the following manner.

8086 CPU

Bus Interfacing Unit (BIU)

Tasks of BIU are :

- It sends out addresses.
- It fetches instructions from memory
- It reads data from memory and ports.
- It also writes data from memory and ports.
- So BIU takes care of all the address and data transfers on the buses.
- Hence it is called the external world interface of the processor
- It works in synchronous with machine cycles

Execution Unit (EU)

Tasks of EU are :

- It tells the BIU, from where to fetch the instruction or data.
- It decodes the fetched instructions.
- It executes the decoded instructions.
- EU takes care of performing operations on the data
- EU is called as the execution heart of the processor.
- It works in synchronous with T-states

11.4 The Execution Unit (EU)

Main function of EU is **decoding** and **execution** of the instructions. In order to carry out its tasks it has the following units :

- Arithmetic Logic Unit (ALU)
- General Purpose Registers.
- Decoder
- Flag Register.
- Control Unit
- Pointer and Index Registers

11.4.1 Arithmetic Logic Unit (ALU)

- The ALU in the EU is a 16 bit unit i.e. it can perform 16-bit operation simultaneously.
- It is capable of performing a variety of arithmetic and logic operations such as add, subtract, AND, OR, NOT, EX-OR, increment, decrement, shift etc.

11.4.2 Flag Register

Q. Explain the flag register of 8086.

- Flag register is a part of EU. It is a 16-bit register with each bit corresponding to a flipflop.
- A flag is a flipflop. It indicates some condition produced by the execution of an instruction. For example the zero flag (ZF) will set if the result of execution of an instruction is zero.
- A flag can control certain operations of EU.
- Fig. 11.4.1 shows the details of the 16 bit flag register of 8086 CPU. As shown, it consists of nine active flags out of sixteen
- The remaining seven flags marked "U" are undefined flags.

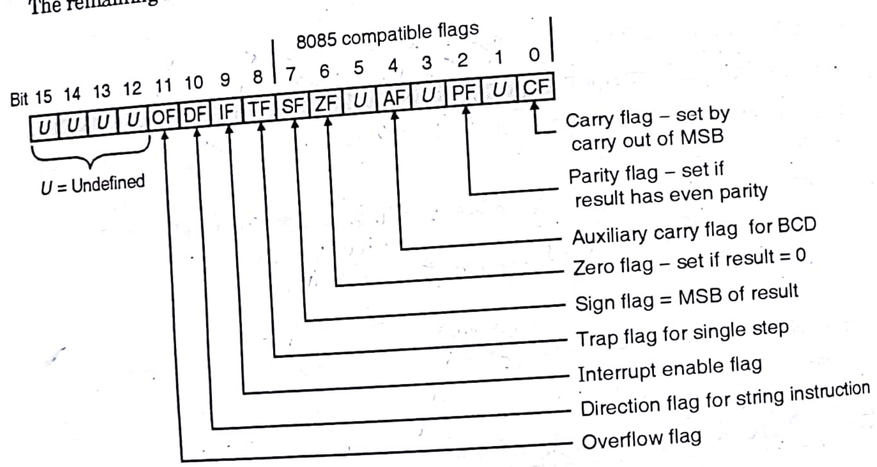


Fig. 11.4.1 : 8086 flag register format

Active flags

There are nine active flags out of 16, in the 8086 flag register. The remaining are undefined flags.

Control flags

- Out of the nine active flags, six are conditional (status) flags and the remaining three are called as the control flags, because they are used to control certain operations of the processor.
- The three control flags are :
  - The trap flag (TF)
  - The interrupt flag (IF)
  - The direction flag (DF)

1. **The trap flag (TF)**
   (i) Setting TF puts the processor into single step mode for debugging. In single stepping, microprocessor executes a instruction and enters into single step ISR.
   (ii) After that user can check registers or memory contents, if found ok, he/she will proceed further, else necessary action will be taken. **This utility is, to debug the program.** If TF = 1, the CPU automatically generates an internal interrupt after each instruction, allowing a program to be inspected as it executes instruction by instruction.
   Used by debuggers for single step operation.
   TF = '1' - Trap on, TF = '0' - Trap off

2. **The interrupt flag (IF)**
   (i) If user sets IF flag, the CPU will recognize external (maskable) interrupt requests.
   (ii) Clearing IF disables these interrupts.
   (iii) IF has no effect on either non-maskable external or internally generated interrupt. The interrupt flag (IF) is used for allowing or prohibiting the interruption of a program.
   IF = '1' - Interrupt enabled, IF = '0' - Interrupt disabled

3. **The direction flag (DF)**
   (i) This bit is specifically for string instructions. In string instruction we use SI (source index) and DI (destination index) registers as offset registers to point source area and destination area respectively. DF flag controls direction of SI and DI pointers.
   (ii) If DF = 1, the string instruction will automatically decrement the pointer(s), i.e. process string from high addresses to low addresses, or from right to left.
   (iii) If DF = 0, the string instruction will automatically increment the pointer(s), i.e. process string from low addresses to high addresses or from left to right.
   It is used with string instructions
   DF = '1' - Up , DF = '0' - Down

Conditional (status) flags

- Six flags out of these nine active flags indicate status of the result produced after the execution of an instruction. Such flags are called as the conditional flags.
- The names of the conditional flags indicate what conditions affect them. For example the carry flag (CF) is set to 1 if addition of two numbers produces a carry output.
- Some of the 8086 instructions check the status of the conditional flags, before execution.
- The six conditional flags are :
  - The parity flag (PF)
  - The sign flag (SF)
  - The carry flag (CF)
  - The zero flag (ZF)
  - The auxiliary carry flag (AF)
  - The overflow flag (OF)

1. **The parity flag (PF)**
   (i) This flag is set when the result has even parity, an even number of 1 bits.
   (ii) If parity is odd, PF is resetted.
   (iii) This flag is normally used to check for data transmission errors.
   PF = '1' - low byte has an even number of 1 bits, *PF = '0' - low byte has odd parity*

2. **The zero flag (ZF)**
   This flag is set, when the result of operation is zero, else it is reset.
   ZF = '1' - zero result , *ZF = '0' - non-zero result*

3. **The sign flag (SF)**
   (i) This flag is set, when MSB (most significant bit) of the result is 1.
   (ii) In other words it copies the MSB of the result
   (iii) Since negative binary numbers are represented (in the 8086 CPU) using standard two's complement notation, the MSB (copied in sign flag) indicates the number is positive or negative.
   (iv) SF indicates sign of the result only in case of signed operation
   (v) In case of unsigned operation, sign bit has no significance.
   SF = '1' - msb is 1 (negative) , *SF = '0' - msb is 0 (positive)*

4. **The auxiliary carry flag (AF)**
   (i) It is a carry from lower digit to upper digit
   (ii) This flag is set, whenever there has been a carry out of the low nibble into the high nibble or a borrow from high nibble into the low nibble of an 8 bit quantity else AF is reset.
   (iii) This flag is used by decimal arithmetic instructions.
   AF = '1' - carry out from bit 3 on addition or borrow into bit 3 on addition
   AF = '0' - no carry out from bit 3 on addition nor borrow into bit 3 on addition

5. **The carry flag (CF)**
   (i) It can also be called as a final carry
   (ii) This flag is set whenever there has been a carry out of, or a borrow into, the MSB of the result (8 or 16 bit).
   (iii) The flag is used by the instructions that add and subtract multibyte numbers.
   (iv) Rotate instructions can also isolate a bit in memory or a register by placing it in the carry flag.
   CF = '1' - there is a carry out from the most significant bit (MSB),
   CF = '0' - no carry out from msb

6. **The overflow flag (OF)**
   (i) It indicates an overflow from the magnitude to the sign bit of result
   (ii) If OF is set, an arithmetic overflow has occurred; that is, a significant bit has been lost because the size of the result exceeded the capacity of its destination location.
   (iii) In 8086 interrupt on overflow instruction is available that will generate an interrupt in this situation. If result is not out of range, OF remains reset.
   OF = '1' - signed overflow occurred, *OF = '0' - no overflow*

Hence we can say for the following bit structure of data, the flags will be affected as indicated in the table

D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0

Flags name	8-bit operation	16-bit operation
Auxiliary carry	Carry from D_3 to D_4	Carry from D_3 to D_4
Carry	Carry from D_7 to D_8	Carry from D_{15} to D_{16}
Overflow	Carry from D_6 to D_7	Carry from D_{14} to D_{15}
Sign	Copy of D_7	Copy of D_{15}

11.4.3 General Purpose Registers

Q. Explain the dedicate use of GPR.

- As shown in Fig. 11.4.1, the execution unit (EU) has four general purpose 16-bit registers.
- Each one of them can be used for temporary storage of 8 bit data, 16-bit data or 32-bit data.
- 8-bit Registers - AH, AL, BH, BL, CH, CL, DH, DL. Any of these registers can be used as an 8-bit operand.
- 16-bit Registers - AX, BX, CX, DX, SI, DI, SP, BP. Any of these registers can be used as a 16-bit operand.
- 32-bit Registers - DX : AX together can be used for 32-bit operand.
- These registers can be used for general purpose computing when their other specialized functions do not interfere.

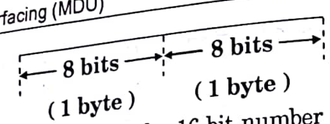
Can we store 16 bit data words in the general purpose registers ?

- The answer is yes. We can use certain pairs of the general purpose registers to store 16 bit data words.
- Such register pairs are AH and AL, BH and BL, CH and CL, DH and DL.
- The above mentioned register pairs are referred to as follows.

Pair of general purpose registers	Referred to as
1. Pair of AH and AL	AX register.
2. Pair of BH and BL	BX register
3. Pair of CH and CL	CX register
4. Pair of DH and DL	DX register

- In short, the four general-purpose 16-bit data registers : AX, BX, CX, and DX, each of these is a combination of two 8-bit registers which are separately accessible as AL, BL, CL, DL (the "low" bytes) and AH, BH, CH, and DH (the "high" bytes).

AX:	AH	AL	Accumulator
BX:	BH	BL	Base Register
CX:	CH	CL	Count Register
DX:	DH	DL	Data Register



- For example, if AX contains the 16-bit number 1234H, then AL contains 34H and AH contains 12H.
- The upper and lower halves of the data registers are separately addressable. This means that each data register can be used as a single 16-bit register or as two 8-bit registers.

Special functions of general purpose registers

Although the first four registers AX, BX, CX, and DX are called as "general-purpose", each of them is designed to play a particular role in common use:

1. **Register AX**

AX is the "16-bit accumulator" while *AL is "8-bit accumulator"*

Accumulator has the following special functions :

(i) Some of the operations, such as Multiplication and Division, require that one of the operands be in the accumulator and also the result is stored in accumulator.
Some other operations, such as Addition and Subtraction, may be applied to any of the registers (that is, any of the eight general- and special-purpose registers) but are more efficient when working with the accumulator.

(ii) It works as a via register for I/O accesses i.e. a data is routed through accumulator for the communication of the processor and I/O devices.
For OUT instruction the data in accumulator (AL for 8-bit data and AX for 16-bit data) can only be given to the output device
For IN instruction the data taken from the input device can be taken only in accumulator (AL for 8-bit data and AX for 16-bit data)

(iii) It also works as a via register for string instructions. Whenever a data is to be brought from memory or given to memory in case of string operations it is routed through accumulator only.

2. **Register BX**

BX is the "base" register;

(i) It is the only general-purpose register which may be used for indirect addressing. (various addressing modes are discussed in chapter 4.)

(ii) For example, the instruction MOV [BX], AX causes the contents of AX to be stored in the memory location whose address is given in BX.

3. **Register CX**

CX is the "count" register. It works as a default counter register for three instructions viz :

(i) The looping instructions (LOOP, LOOPE, and LOOPNE), to indicate the number of iterations

(ii) The shift and rotate instructions (RCL, RCR, ROL, ROR, SHL, SHR, and SAR), to indicate number of shifts or rotations (Here only CL is used and not entire CX)

(iii) The string instructions (with the prefixes REP, REPE, and REPNE) to indicate the size of the string block.

4. **Register DX**

DX is the "data" register

(i) It is used together with AX for the word-size MUL and DIV operations, when the operand size is greater than the register AX i.e. operand is 32-bit

(ii) It also holds the port number for the IN and OUT instructions. For 16-bit address accesses of I/O ports only DX can be used as a pointer.

Summary of Implicit use of General Purpose Registers

Registers	Operations
AX	Word multiply, Word divide, Word I/O and Word string
AL	Byte multiply, byte divide, byte I/O, byte string and decimal / ASCII arithmetic.
BX	Store address information
CX	Counter for String operations and loops
CL	Counter for Variable shift and rotate
DX	Word multiply, word divide, Indirect I/O

11.4.4 Control Circuitry

The control circuit is a part of EU. It is used for directing the internal operations.

11.4.5 A Decoder

- "The process of translation from instructions into action is known as decoding"
- A decoder in the execution unit (EU) is used for translating the instructions fetched from the memory into a series of actions.
- The EU will actually carry out these actions.

11.4.6 Pointer and Index Register

- The execution unit also contains the following 16 bit registers.

• Base Pointer (BP) register	• Stack Pointer (SP) register
• Source Index (SI) register	• Destination Index (DI) register

These registers can be used as general purpose 16-bit registers. But mainly they are used to hold the 16 bit offset of data word in one of the segments as given below :

Base Pointer (BP) register

- This is a 16 bit register in the E.U. It holds the 16 bit offset relative to the stack segment (SS) register.
- But BP has a specific use. BP is used whenever we pass a parameter by way of stack.
- The BP register can also be used as an offset register in the addressing mode called **base addressing mode**.

Stack Pointer (SP) register

- This is also a 16-bit register in the E.U. It holds the 16-bit offset address relative to stack segment (SS) register.
- SP is used for sequential access of stack segment.
- It always points to the top of stack.
- It is mainly used during instructions like PUSH, POP, CALL, RETURN etc.

Source Index (SI) register

- This register is used for holding the offset of a data word in the data segment (DS).
- The physical address of a location in the data segment can be generated by adding a hardwired zero to the segment base (16 bit) held by the data segment (DS) register and then by adding the offset in the SI register to it.

Data segment (DS) Register →		2	0	0	0	0	← Hard wired 0
Source index (SI) register (offset) →	+		1	F	2	3	
20 bit physical address in D.S. →		2	1	F	2	3	

Destination Index (DI) register

- This register is used for holding the 16 bit offset of a data word in the extra segment (ES).
- The source index (SI) register and destination index (DI) register are used for the string related instructions.
- For example if we want to move a block of data from memory to memory, then source index (SI) register can be used to point to the source memory address and destination index (DI) register is used to point to the destination memory address.

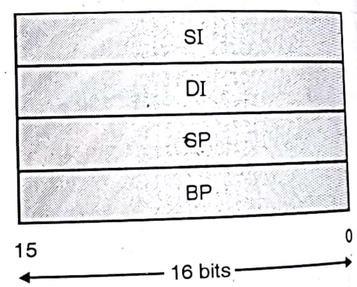


Fig. 11.4.2 : Pointer register of EU

11.5 The Bus Interfacing Unit (BIU)

The bus interface unit performs all the activities related to Bus. Specifically BIU has the following **five functions**

1. Instruction **fetching** (reading) from primary memory. (performed over the system bus)
2. R/W of data operand from/to primary memory. (performed over the system bus)
3. I/O of data from/to peripheral ports. (performed over the system bus)
4. Address generation for memory reference
5. Instruction queuing in an instruction queue.

In order to carry out its functions BIU has the following modules :

- Instruction queue.
- An instruction pointer register (IP).
- Segment registers
- Address generation and bus control.

11.5.1 The Instruction Queue

- The execution unit is supposed to decode or execute an instruction. Decoding does not require the use of buses.
- When EU is busy in decoding and executing an instruction, the BIU fetches upto six instruction bytes for the next instructions.
- These bytes are called as the prefetched bytes and they are stored in a first-in-first-out (FIFO) register set, which is called as a "**queue**."

11.5.1(A) Significance of Queue

- To understand the significance of the queue, refer Fig. 11.5.1.

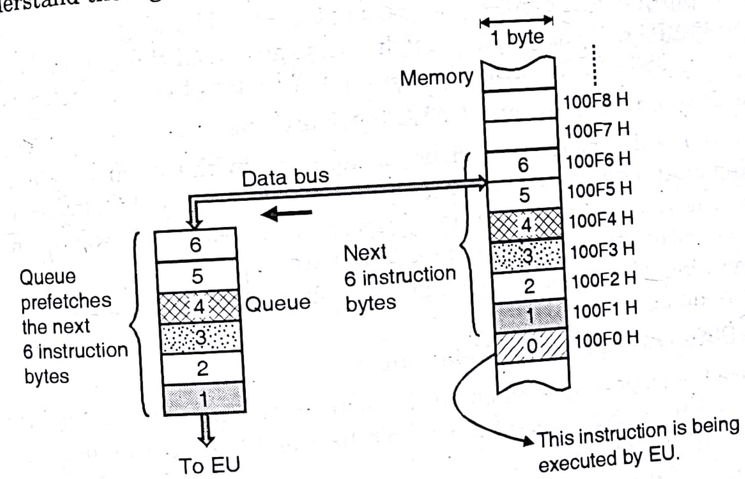


Fig. 11.5.1 : Significance of queue

- As shown in Fig. 11.5.1, while the EU is busy in decoding the instruction corresponding to memory location 100F0, the BIU fetches the next six instruction bytes from locations 100F1 to 100F6 numbered as 1 to 6.
- These instruction bytes are stored in the 6 byte Queue on the first-in-first-out (FIFO) basis.
- When EU completes the execution of the existing instruction, and becomes ready for the next instruction, it simply reads the instruction bytes in the sequence 1, 2, ... from the Queue.
- Thus the Queue will always hold the instruction bytes of the next instructions to be executed by the EU.

11.5.1(B) Pipelining

Q. Explain the concept of pipelining in 8086 architecture.

- The process of fetching the next instruction, when the present instruction is being executed is called as pipelining.
- Pipelining has become possible due to the use of queue.
- BIU fills in the queue, until the entire queue is full.
- BIU restarts filling in the queue when atleast two locations of queue are vaccant.

11.5.1(C) Advantage of Pipelining

Q. What are the advantages of pipelining ?

- The EU always reads the next instruction byte from the queue in BIU. This is much faster than sending out an address to the memory and waiting for the next instruction byte to come.
- In short pipelining eliminates the waiting time of EU and speeds up the processing.
- The 8086 BIU will not initiate a fetch unless and until there are two empty bytes in its queue. 8086 BIU normally obtains two instruction bytes per fetch.

Behaviour of queue in JMP and CALL instructions

- If a Jump (JMP) or CALL instruction appears in the main program, then all the existing instruction bytes in the Queue are flushed out and Queue is made empty.
- Then it is reloaded with the new instruction bytes which correspond to the new locations mentioned in the JMP or CALL instructions.
- The refilling of the queue then continues from the new locations corresponding to the in main program.

Why is the 8086 Queue only six byte long ?

- Because the longest instruction, in the instruction set of 8086 is six byte long. Hence with a six byte long queue it is possible to prefetch even the longest instruction in the instruction set.
- The queue of 8088 is however 4 byte long.

11.5.2 Segment Registers

Q. Describe the concept of segmentation in 8086 microprocessor.

- The BIU contains four special purpose registers called as segment registers. They are :
  - The code segment (CS) register
  - The stack segment (SS) register
  - The extra segment (ES) register and
  - The data segment (DS) register
- The purpose of using these segment registers and segmentation can be explained as follows :-
- All these are 16 bit registers.

- The number of address lines in 8086 is 20. So the 8086 BIU will send out a 20 bit address in order to access one of the 1,048,576 or 1 Mb memory locations.
- But it is interesting to note that the 8086 does not work the whole 1,048,576 byte (1M.byte) memory at any given time. However it works with only four 65,536 (64k-byte) segments within the whole 1 M-byte memory.
- The four segment registers actually hold (contain) the upper 16 bits of the starting addresses of the four memory segments of 64 k byte each with which the 8086 is working at that instant of time.
- A word is any two consecutive bytes in memory. Word are stored in memory with the most significant byte at the higher memory address. They bytes are stored sequentially from byte 0000H to byte FFFFH.
- Programs view memory space as a group of segments defined by the application.
- A segment is a logical unit of memory that may be upto 64 K bytes long.
- Each segment is made up of contiguous memory locations. It is independent, separately addressable unit.
- Note that these starting addresses will always be changing. They are not fix.
- This concept can be clearly understood by referring to Fig. 11.5.2.
- Fig. 11.5.2 shows one of the possible ways to position the four 64 k byte segments within the 1-M byte memory space of 8086. There is no restriction on the locations of these segments in the memory.

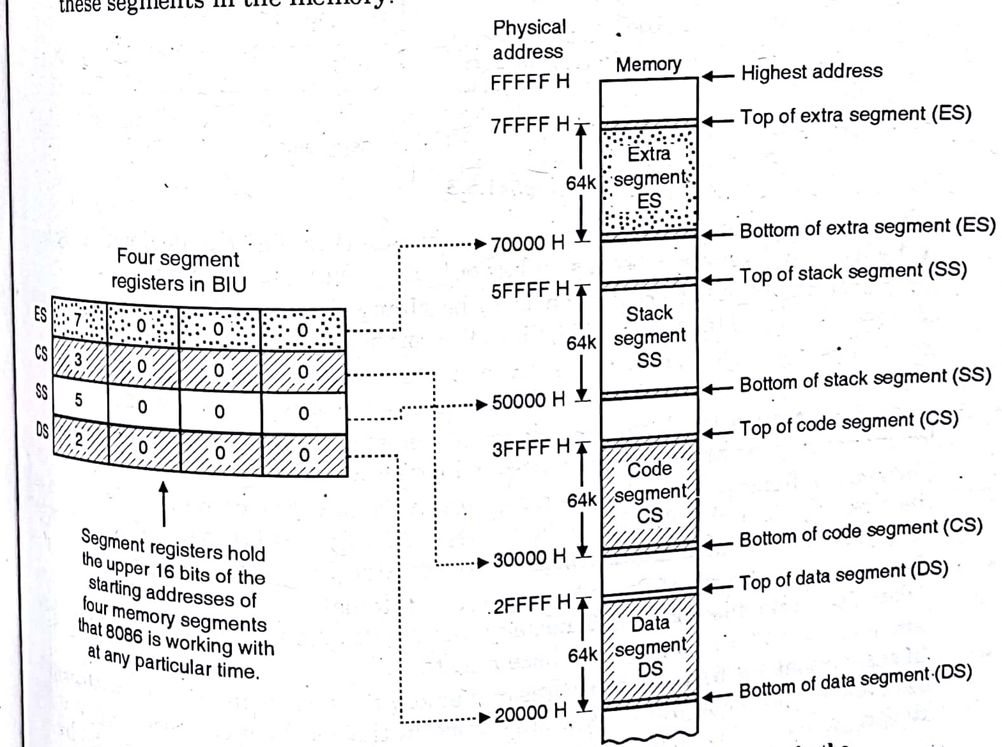


Fig. 11.5.2 : One way of positioning four 64 k byte segments within the 1 M byte memory space of an 8086

- Note that these segments can be separate from each other as shown in Fig. 11.5.2 or they can overlap.
- Note that the starting address or base address of the data segment is 20000H. The upper 16-bits of this i.e. 2000 are loaded into the data segment register (DS).
- Similarly upper 16 bits of the starting addresses of the other segments are stored into the corresponding segment registers.
- The segment overlapping usually takes place for small programs which do not need all the 64 k bytes in each segment.
- The segments can adjacent, disjoint, partially overlapping or fully overlapping.
- Fig. 11.5.3 shows another way of positioning segments within 1 MB memory space of 8086.

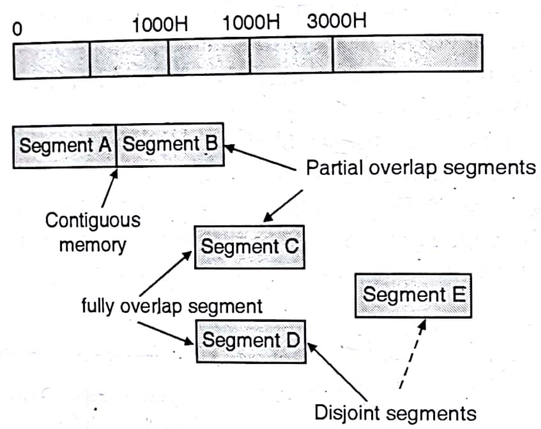


Fig. 11.5.3

- In the users program there can be many segments. But 8086 can deal with only four of them at any given time, because it has only four segment registers.
- Whenever the segment orientation is to be changed, we have to change the base addresses and load the upper 16 bits into the corresponding segment registers.
- Each time the CPU needs to generate a memory address, one of the segment registers is automatically chosen and its contents added to a logical address.
- For an instruction fetch, the code segment register is automatically added to the logical address to compute the value of the instruction address.
- For stack refrencing operations, either data or extra segment register is chosen as the base. The logical address may be made up of different types of values: it can be simply the immediate data value and a base register plus an index register.
- Generally the selection of DS or ES register is automatically done, though provisions exist to override this selection. Thus, any memory location can be addressed without changing the value of the segment base register.
- In system that use 64K or fewer bytes of memory for each memory area (code, stack, data and extra) the segment registers can be initialized to zero at the beginning of the program and then ignored, since zero plus a 16 bit offset yields a 16 bit address.

- In a system where the total amount of memory is 64 KB or less, it is possible to set all segments equal and have fully overlapping segments.
- Segment registers are very useful for large programming tasks that require isolation of program code from the data code or isolation of module data from the stack information etc.
- Segmentation makes it easily to build relocatable and re-entrant programs. In many cases the task of relocating a program (relocation means to ability for an the same program in several different areas of memory without changing addresses in the program itself. Simply requires moving the program code and then adjusting the code segment register to point to the base of the new code area.
- Since programs can be written for the 8086/8088 in which all branches and jumps are relative to the instruction pointer, it branches and jumps are relative to the instruction pointer, it does not matter what value is kept in the code segment register.
- Every application will define and use segments differently. The currently addressable segment override prefix provides a generous workspace; 64KB bytes for code, 64 Kbytes stack and 128 Kbytes of data storage.

11.5.3 Memory Segmentation

- Memory can be thought of as a vast collection of bytes. These bytes need to be organized in some efficient manner in order to be of any use.

11.5.4 Advantages of Segmentation

Q. Explain the advantages of segmentation.

The principle of segmentation discussed now has certain benefits. Some of them are as follows :

1. Segmentation provides a powerful memory management mechanism.
2. The segmented structure of 8086 memory space supports modular software design. It discourages huge monolithic programs. i.e. Among other things, It allows programmers to partition their programs into modules that operate independently of one another.
3. Segments provide a way to easily implement object-oriented programs.
4. Segments allow two processes to easily share data.
5. It allows you to extend the address ability of a processor i.e. segmentation allows the use of 16-bit registers to give an addressing capability of 1 M byte. Without segmentation, we would require 20-bit registers. In the case of the 8086, segmentation let Intel's designers extend the maximum addressable memory from 64K to one megabyte.
6. Segmentation makes it possible to separate the memory areas for stack, code and data.
7. It is possible to increase the memory size of code data or stack segments beyond 64 k bytes by allotting more than one segment for each area.
8. Segmentation makes it possible to write programs which are **position independent or dynamically relocatable**.

11.5.5 Code Segment (CS)

- Code segment (CS) is the part of memory from which the BIU fetches the instruction code bytes.
- The upper 16 bits of the starting address of code segment are held by the code segment (CS) register.
- But the memory address for the base of code segment is 20 bit long. So what about the lower 4-bits ? The answer is that the BIU always inserts zeros for the lowest 4-bits.
- So if the CS register contents are 3428 H then after adding zeros, the physical starting address of code segment becomes 34280 H.
- Therefore a segment will always start at an address with zeros in the lowest 4 bits.

Segment Base : The upper 16-bits of the starting address of a segment, stored in the segment register is called as the segment base.

11.5.6 Stack

- In Fig. 11.5.3 we have defined the stack segment from the memory location 50000 H to 5FFFF H.
- The stack is a section of memory which is reserved to store the addresses and data while executing a subroutine program.
- The stack segment (SS) register holds the upper 16 bits of the starting address of the stack area.

11.5.7 Instruction Pointer (IP) Register

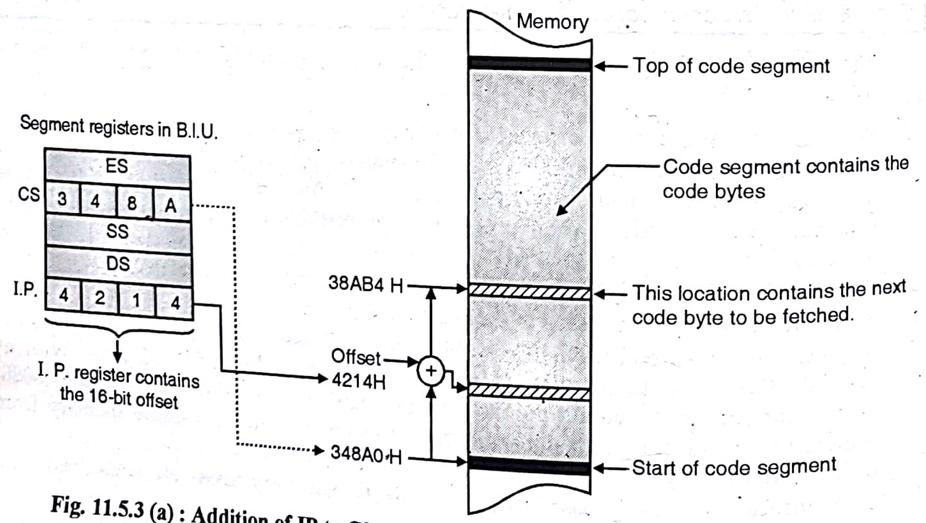


Fig. 11.5.3 (a) : Addition of IP to CS to produce the physical address of the code byte

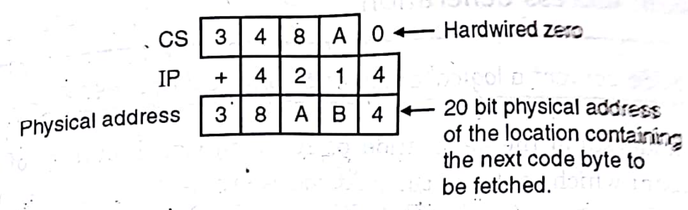


Fig. 11.5.3 (b) : Computation

- Another special purpose register in the BIU is the instruction pointer (IP) register.
- As discussed earlier the code segment (CS) register holds the upper 16 bits of the 20 bit starting address of the code segment. And code segment is the segment from which the BIU is currently fetching the instruction code bytes.
- The instruction pointer (IP) register holds the 16 bit address or **offset** of the next code byte within the code segment. This is illustrated in Fig. 11.5.3(a).

Rules of segmentation

1. If non-overlapping, there can be 16 segment in all, each of 64KB (since, 1MB / 64KB = 16)
2. Segments can overlap each other
3. A segment can begin at any location that is a multiple of 10H
4. At any given time a maximum of 4 segments and hence 256KB can be accessed
5. The memory of 8086 is a wrap around memory. This means that once the address generated crosses the 1MB limit, it accesses the location at the top 00000H. For e.g. if CS = FFFF and IP = 0010. Physical address = CS*10H + IP = 100000H. The address lines in 8086 are only 20, so the MSB '1' is discarded and the location being accessed is 00000H.
6. In an instruction only the offset address in mentioned. The segment register is fixed for each of the pointer registers. Default segment register assignments are as follows :

Pointer Register	Default Segment Register
BX	DS
SI	DS
DI	DS (ES in case of string operations)
SP	SS
BP	SS
IP	CS

7. Using segment override prefix, one can change the above default segment register assignments

11.5.8 Physical Address Generation

Q. How does 8086 convert a logical address to physical address ?

- Let us now understand the generation of 20 bit physical address of the location in the code segment which contains the next code byte.
- The sequence of operation is as follows.

Physical Address Generation

Step 1 : The CS register contains the upper 16 bits of the starting address of the code segment.

∴ CS Register : | 3 | 4 | 8 | A | Segment base

⇩

Step 2 : The BIU will automatically insert zeros for the lowest four bits of the segment base address to get the 20 bit physical address for the starting of code segment.

∴ Starting address of code segment : | 3 | 4 | 8 | A | 0 ← BIU adds this zero

⇩

Step 3 : The I.P. register contains the offset or distance from this address. The offset here is 4214H.

∴ I.P. Register : | 4 | 2 | 1 | 4 |

⇩

Step 4 : Add the starting address of code segment (20 bit) to the offset to get the physical address of the location containing the next code byte as follows :

Starting address of code segment →		3	4	8	A	0	← Hard wired 0
Offset in the I.P. Register →	+		4	2	1	4	
Physical address of the location containing the next code byte →		3	8	A	B	4	

- This 20 bit physical address is then sent out by the BIU to fetch the next code byte stored at this location.

Alternative way to represent the physical address

- An alternative way of representing a 20 bit physical address is as follows :

  Segment base : Offset

- For example the 20 bit physical address in the above example is 348A : 4214.

What does an offset tell us ?

- The offset is a 16 bit hex number contained by the instruction pointer (I.P.) register and it tells BIU the location in the code segment, from which the next instruction byte is to be fetched.

11.6 Pin Diagram of 8086

Sr. No.	Pins	Name of the pins
1.	Supply pins (3 pins)	V_{CC} GND GND
2.	Clock related pins (3 pins)	CLK RESET READY
3.	Address and Data pins (21 pins)	$AD_0 - AD_{15}$ $A_{16}/S_3 - A_{19}/S_6$ $\overline{BHE}$ / S_7
4.	Interrupt pins (2 pins)	NMI INTR
5.	Other control (3 pins)	$\overline{TEST}$ MN / $\overline{MX}$ $\overline{RD}$
6.	Mode multiplexed signals (8 pins) (MIN mode – MAX mode signals)	HOLD – $\overline{RQ_0}$ / $\overline{GT_0}$ HLDA – $\overline{RQ_1}$ / $\overline{GT_1}$ $\overline{WR}$ – $\overline{LOCK}$ $\overline{DEN}$ – $\overline{S_0}$ DT / $\overline{R}$ – $\overline{S_1}$ M / $\overline{IO}$ – $\overline{S_2}$ ALE – QS_0 $\overline{INTA}$ – QS_1

I. Supply pins (3 pins)

V_{CC}	GND	GND

- Used for power supply i.e. +5V on V_{CC} w.r.t. GND.
- Two separate GND pins for two layers of 8086 chip, improves the noise rejection.

II. Clock related pins (3 pins)

CLK	RESET	READY

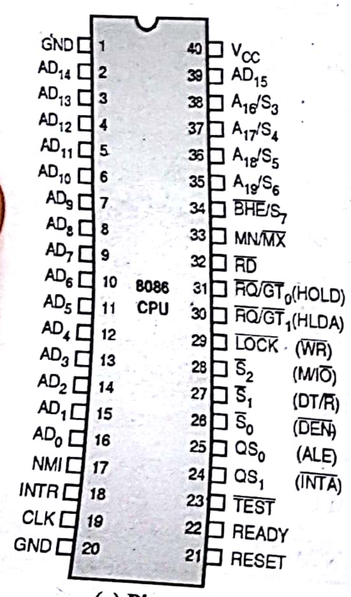
CLK

- This pin provides the basic timing for the processor.
- 8086 does not has an on-chip clock generator hence an external clock generator like 8284 is used to provide the clock signal.
- It is asymmetric with 33% duty cycle, TTL clock signal.

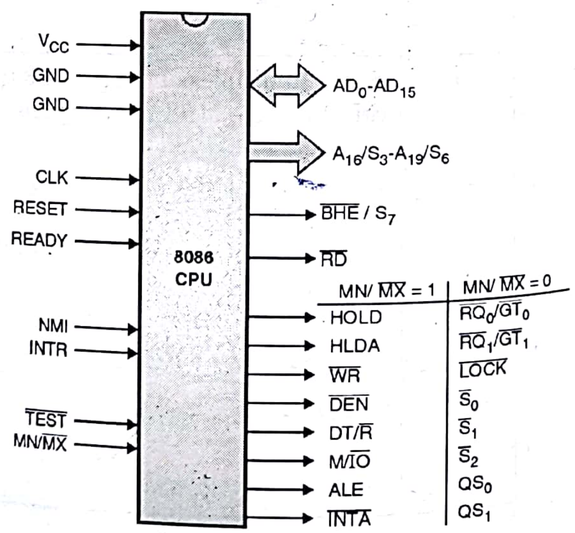
RESET

- It causes the processor to immediately terminate its present activity. The 8284 clock generator provides this signal.
- This signal must be active high for atleast 4 clock cycles
- It clears all the flag register, the Instruction Queue, the DS, SS, ES and IP registers and sets the bits of CS register.
- Hence the reset vector address of 8086 is FFFF0H (as CS = FFFFH and IP = 0000H).

READY



(a) Pin configuration



(b) Functional pin diagram

Fig. 11.6.1

- It is an acknowledgement from the addressed memory or I/O that it will complete the data transfer specially meant for slow devices.
- μP samples the READY input between T2 and T3 of a M/C cycle.
- If READY pin is LOW, μP inserts wait-states between T2 and T3, until READY becomes HIGH.

III. Address and data pins (21 pins)

$AD_{15} - AD_0$	$A_{16}/S_3 \ldots A_{19}/S_6$	$\overline{BHE}/S_7$

$AD_{15} - AD_0$

- These are time multiplexed data address lines i.e. for some time they have address and for some time data
- It gives the address $A_{15} - A_0$ during T1 of an Machine Cycle. (When ALE = 1)
- It gives the data $D_{15} - D_0$ after T1 of an M/C Cycle (Machine cycle).

$A_{16}/S_3 \ldots A_{19}/S_6$

- These lines work as Address bus ($A_{16} \ldots A_{19}$) during T1 of every M/C Cycle.
- T2 onwards these lines work as Status signals $S_3 \ldots S_6$.
- S_3 and S_4 gives the status of the memory segment currently accessed. S_5 gives the status of the Interrupt Enable Flag updated every clock cycle. S_6 goes low when 8086 controls the shared system bus.

S_4	S_3	Segment accessed
0	0	Extra Segment
0	1	Stack Segment
1	0	Code Segment or None
1	1	Data Segment

$\overline{BHE}/S_7$

- This line carries the $\overline{BHE}$ signal during T1.
- $\overline{BHE}$ and A_0 are used together to access a word/byte from the memory as shown in the banking in section 11.8.
- Status line S_7 is reserved for "further development".

IV. Interrupt pins (2 pins)

NMI	INTR

NMI

- This is a non-maskable, edge triggered that causes type 2 interrupt i.e. on receiving an interrupt on NMI line, the μP executes INT 2 and transfers the control to location 2 * 4 =00008H in the Interrupt Vector Table (IVT). It reads 4

locations starting from this address to get values for IP and CS of the ISR address.

- It is not maskable internally by software.
- A transition from LOW to HIGH on this pin, causes the interrupt at the end of the current instruction

INTR

- This is a non-vectored, maskable, level triggered interrupt sampled during last clock cycle of each instruction.
- To get the vector number for the interrupt the following procedure is followed
- On receiving an interrupt on INTR line, the µP executes 2 $\overline{INTA}$ pulses.
- First $\overline{INTA}$ pulse → the interrupting device is indicated for its interrupt being accepted while the device calculates the vector number.
- Second $\overline{INTA}$ pulse → the interrupting device sends the vector number to the microprocessor on the data lines.
- Control shifts to location pointed by IP and CS which are loaded from IVT at Vector No * 4.

V. Other control (3 pins)

$\overline{TEST}$	MN/$\overline{MX}$	$\overline{RD}$

$\overline{TEST}$

- It is an active low input line dedicated for 8087 Co-processor.
- In minimum mode it is connected to GND. In Maximum Mode whenever the Co-processor is busy it makes this pin HIGH.
- $\overline{TEST}$ input is examined by the WAIT instruction.
- If the TEST pin is high, the µP enters idle state; till TEST pin becomes low i.e. 8087 is free.

MN/$\overline{MX}$

- This is an input signal to 8086 that indicates the processor has to work in which mode.
- If this signal is HIGH, 8086 is in Minimum mode i.e. Single-processor system.
- If this signal is LOW, 8086 is in Maximum mode i.e. Multiprocessor system.

$\overline{RD}$

- It is an active low output signal. When it is low 8086 reads from memory or an I/O device.

VI. Mode multiplexed signals (8 pins) (MIN Mode --- Max Mode Signals)

HOLD --- $\overline{RQ_0}/\overline{GT_0}$	HLDA --- $\overline{RQ_1}/\overline{GT_1}$	$\overline{WR}$ --- $\overline{LOCK}$	$\overline{DEN}$ --- $\overline{S_0}$
DT/$\overline{R}$ --- $\overline{S_1}$	M/$\overline{IO}$ --- $\overline{S_2}$	$\overline{ALE}$ --- QS_0	$\overline{INTA}$ --- QS_1

HOLD --- $\overline{RQ_0}/\overline{GT_0}$

- In Minimum Mode this line carries the HOLD input signal from another master requesting a local bus.
- The DMA Controller issues the HOLD signal to request for the system bus.
- In response 8086 completes the current bus cycle and releases the system bus.
- In Maximum Mode it carries the bi-directional $\overline{RQ_0}/\overline{GT_0}$ (Request/Grant) signal.
- The external bus master (8089 or 8087) sends an active low pulse to request for the control over the system bus.
- In response the 8086 completes the current bus cycle, releases the system bus and sends an active low Grant pulse on the same line to the external bus controller.
- 8086 gets back the system bus only after external bus master sends an active low release pulse on the same line.

HLDA --- $\overline{RQ_1}/\overline{GT_1}$

- In Minimum Mode, this line carries the HLDA signal.
- This signal is issued by 8086 after releasing the system bus.
- In Maximum Mode it functions as $\overline{RQ_1}/\overline{GT_1}$ which is the same as $\overline{RQ_0}/\overline{GT_0}$; but is of lower priority

$\overline{WR}$ --- $\overline{LOCK}$

- In Minimum Mode this line carries the $\overline{WR}$ signal indicates a write operation when this pin is Low.
- It is used with M/$\overline{IO}$ to write to Memory or IO Device.
- In Maximum mode it functions as the $\overline{LOCK}$ output line.
- When 8086 executes an instruction with the LOCK prefix this signal is active (i.e. low) remains active till next instruction, indicating the external bus master cannot take control of the system bus.

$\overline{DEN}$ --- $\overline{S_0}$

- In Minimum Mode it carries the $\overline{DEN}$ signal and is used to enable the Data transceivers (bidirectional buffer IC 8286).
- In Maximum Mode it carries the $\overline{S_0}$ signal. $\overline{S_0}$ is a status signal given to 8288.
- In Maximum Mode, Bus Controller (IC 8288) generates the DEN signal for 8286.

DT/$\overline{R}$ --- $\overline{S_1}$

- In Minimum Mode it carries the DT/$\overline{R}$ signal indicating Data Transmit or Receive.
- This signal goes low for a read operation and high for a write operation.
- In Maximum Mode it carries the $\overline{S_1}$ signal. $\overline{S_1}$ is a status signal given to 8288.
- In Maximum Mode, Bus controller issues the DT/$\overline{R}$ signal to 8286.

M/$\overline{IO}$ --- $\overline{S_2}$

- In Minimum Mode it carries the M/$\overline{IO}$ signal, to distinguish between Memory and IO access.
- In Maximum Mode it carries the $\overline{S_2}$ signal. $\overline{S_2}$ is a status signal given to 8288.
- In Maximum Mode $\overline{S_2}$, $\overline{S_1}$ and $\overline{S_0}$ are used to generate the appropriate control signal.

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	Machine 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	Code Access
1	0	1	Read Memory
1	1	0	Write Memory
1	1	1	Passive

ALE --- QS_0

- In Minimum Mode it carries the ALE signal, which is used to demultiplex the address data line by latching the address.
- In Maximum Mode it carries the QS_0 signal.
- It is used with QS_1 to indicate the Instruction Queue Status.
- In Maximum Mode, Bus Controller 8288 issues the ALE signal. (to Latches IC - 8282)

$\overline{INTA}$ --- QS_1

- In Minimum Mode it carries the $\overline{INTA}$ signal.
- It is issued in response to an interrupt on the INTR line.
- It is used to read the vector number form the interrupting device.
- In Maximum Mode it carries the QS_1 signal.
- In Maximum Mode, Bus Controller issues the $\overline{INTA}$ signal. (to 8086)

QS_1	QS_0	Queue Status
0	0	No Operation
0	1	First byte of Opcode from queue
1	0	Empty the Queue
1	1	Subsequent byte from queue

11.7 Intel 8088 Microprocessor

- The Intel 8088 processor is very similar to 8086 in many aspects. In this section we will discuss the pin diagram internal architecture and features of 8088.
- The internal architecture of 8086 is similar to that of 8086 but the external data bus of 8088 is only 8 bit ($D_7 - D_0$).

11.7.1 Internal Architecture of 8088

- The internal architecture of 8088 is as shown in Fig. 11.7.1. It shows that except the Queue every thing else is exactly same as that in 8086.
- The queue in 8088 is of 4 byte length instead of being 6-byte long like 8086.

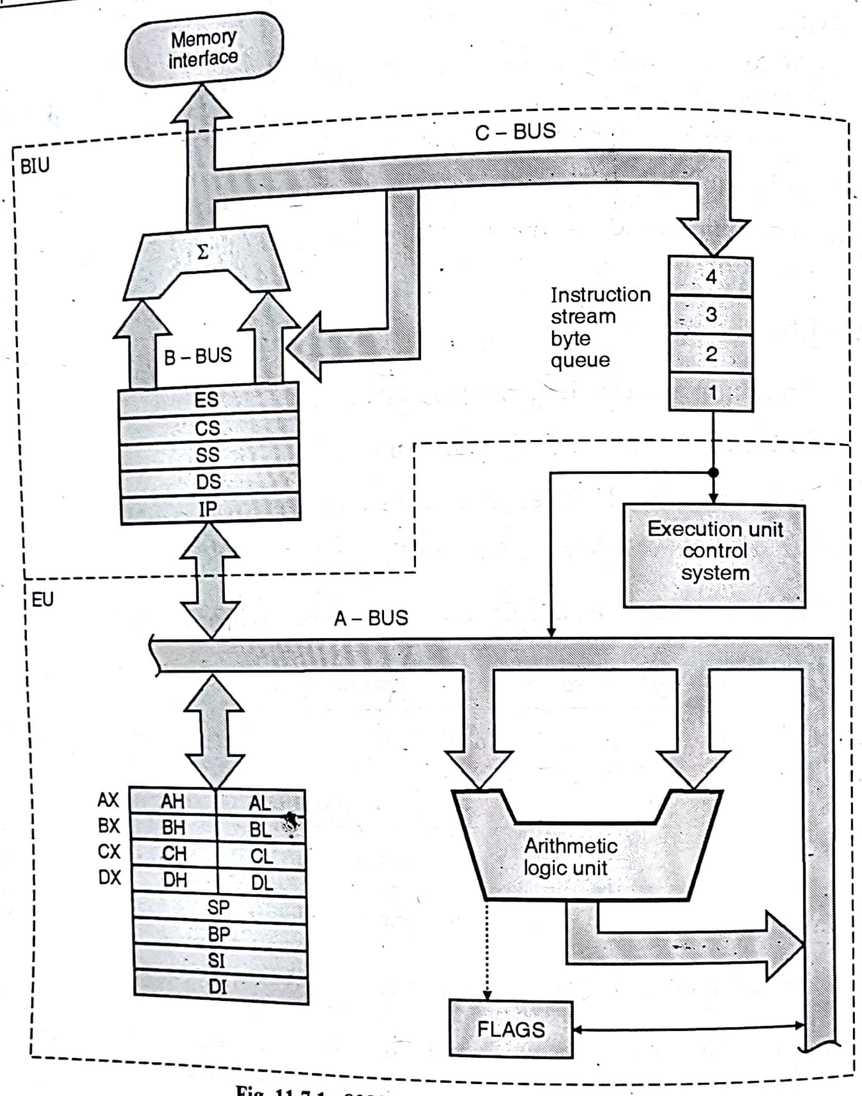


Fig. 11.7.1 : 8088 internal architecture

11.7.2 Difference between 8086 and 8088

Following are some of the points on which the 8088 is different from 8086 :

1. The 8088 has an 8-bit external data bus. ($D_7 - D_0$). So no even and odd memory banks.
2. The internal queue of 8088 is 4 byte long. The reason for having a smaller queue is that the 8088 can only fetch one byte at a time so a 6 byte queue would not be fully utilized.
3. The 8088 BIU fetches an instruction byte when a byte in its queue becomes empty.

11.7.3 Important Features of 8088

- 6 bit data bus
- 16 bit internal architecture.
- Capable of addressing 1M byte memory directly.
- 20 address lines for accessing 1M byte memory.
- Capable of operating in single processor (minimum mode) or multiprocessor (maximum mode) systems.
- Direct software compatibility with 8086.
- Capable of byte, word and block operations.
- It can perform 8 bit and 16 bit signed and unsigned arithmetic in binary or decimal including the multiply and divide operations.
- Compatible will all the Intel peripheral units.
- Multiplexed address/data bus.

11.7.4 Pin Diagram of 8088

- The pin diagram of 8088 is shown in Fig. 11.7.2.

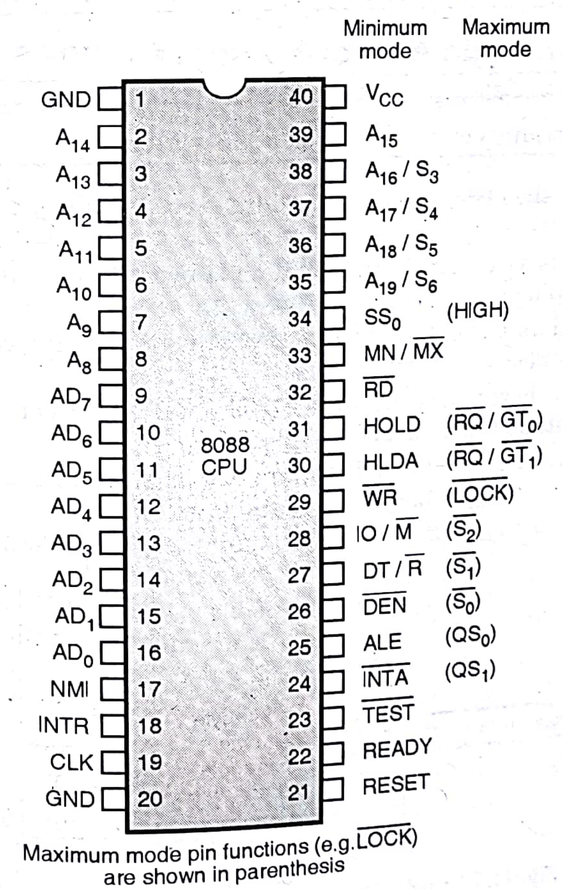


Fig. 11.7.2 : 8088 pin diagram

Observations :

- The pins 8 to 2 and 39 for 8086 were AD_8 to AD_{15} whereas for 8088 these pins are A_8 to A_{15}. This is because in 8088 there are only 8 data bus lines.
- Secondly pin 34 for 8086 was $\overline{BHE}$ / S_7 and for 8088 it is SS_0.
- Except for these two changes all the other pins are exactly same as those of 8086 and their operations are also the same.

11.7.5 Comparison of 8086 and 8088

- The points of differences between 8086 and 8088 are as follows :

Table 11.7.1 : Comparison of 8086 and 8088

Parameter	8086	8088
Dimensions of Queue	6 byte	4 byte
External data bus	16 bit	8 bit
Function of pin 34	$\overline{BHE}$ / S_7	SS_0
Function of pins 8 to 2 and 39	AD_8 to AD_{15}	A_8 to A_{15}

11.8 Even and Odd Memory Banks for 8086

Q. Explain organisation of memory in an 8086 system.

- We know that the 8086 has a 20-bit address bus, so it can address 2^{20} or 1,048, 576 addresses.
- At each address we can store an 8-bit data (1 byte). Hence the total memory capacity of 8086 is 1 M byte.
- However the data bus of 8086 is 16 bit and the processor is capable of processing 16 bit data i.e. words.
- The question is how to write a word (16 bit data) into a memory which is segmented to store the data in the byte form.
- The answer is that a word is written into two consecutive memory addresses.
- That means the lower byte is written into the specified memory address say 0437A H and the higher byte is written into the next higher address as illustrated in Fig. 11.8.1.

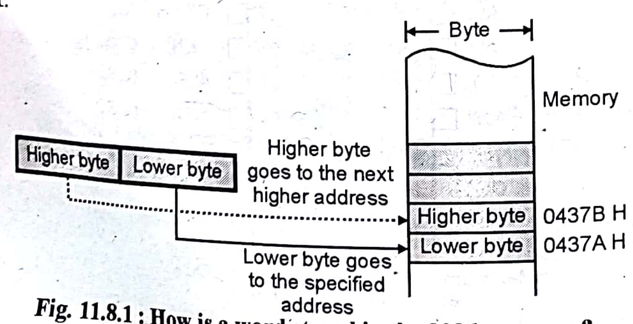


Fig. 11.8.1 : How is a word stored in the 8086 memory ?

- In order to make it possible to read or write a word with one machine cycle, the memory of 8086 is divided into two "banks" of upto 524, 288 bytes i.e. 500 K bytes each, as shown in Fig. 11.8.2.

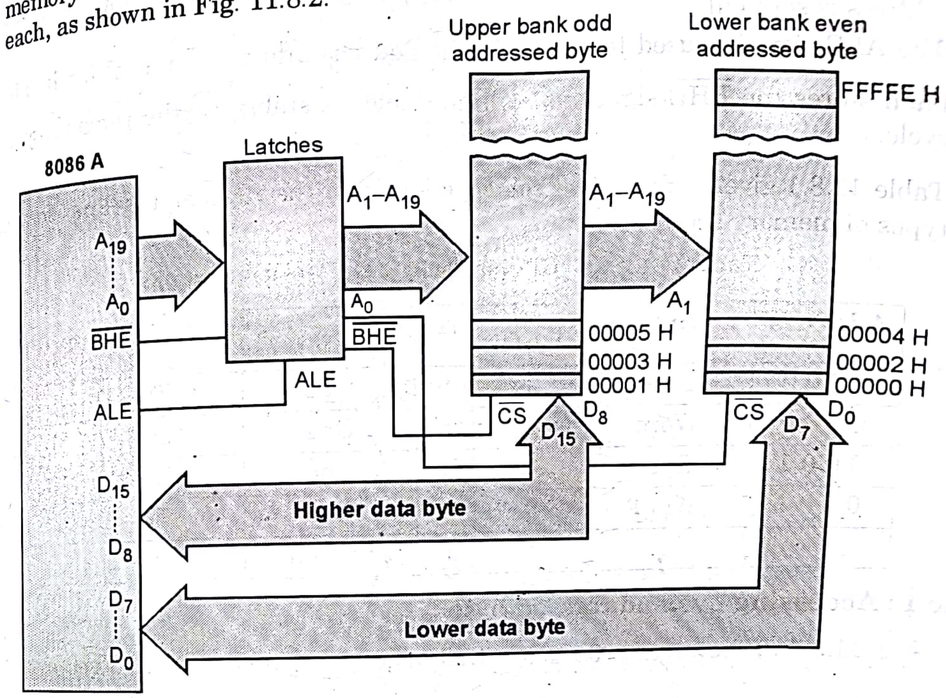


Fig. 11.8.2 : 8086 memory banks block diagram

- The two memory banks are known as :
  1. The even addressed memory bank
  2. The odd addressed memory bank
- The even addressed memory bank contains all the bytes which have even addresses such as 00000, 00002, 00004, ... etc.
- The odd addressed memory bank contains all the bytes which have odd addresses such as 00001. 00003, 00005 etc.
- The even addressed bank is also called as lower bank and the data lines of this bank are connected to the lower eight data lines i.e. D_0 to D_7 of 8086 as shown in Fig. 11.8.2.
- The odd addressed bank is also called as the upper bank and the data lines of this bank are connected to the upper eight data lines i.e. D_8 to D_{15} of 8086 as shown in Fig. 11.8.2.
- The address line A_0 is used to enable the lower memory bank because A_0 is connected to the $\overline{CS}$ input of the lower bank.
- Address lines A_1 through A_{19} are used for selecting the desired memory device in the bank and to address the desired byte in that device.
- The address line A_1 through A_{19} are also used to select a desired memory device in the upper bank and also to address the desired byte in the selected device.

- The bus high enable ($\overline{BHE}$) is used for enabling the memory in the upper bank.
- $\overline{BHE}$ is multiplexed out on a single line from 8086 at the same time when an address is sent out.
- The ALE signal is used to strobe in the address into the external latch. The same latch stores the $\overline{BHE}$ signal as well and holds it stable for the remaining machine cycle.
- Table 11.8.1 given below gives the logic levels on the $\overline{BHE}$ and A_0 lines for different types of memory accesses.

Table 11.8.1 : Signals for the byte and word operations

Address	Data type	$\overline{BHE}$	A_0	Bus cycles	Data lines used
0000	Byte	1	0	One	$D_0 - D_7$
0000	Word	0	0	One	$D_0 - D_{15}$
0001	Byte	0	1	One	$D_7 - D_{15}$
0001	Word	0	1	First	$D_0 - D_7$
		1	0	Second	$D_7 - D_{15}$

Case I : Accessing even addressed byte :

- Fig. 11.8.3(a) shows the process involved in accessing an even addressed byte.

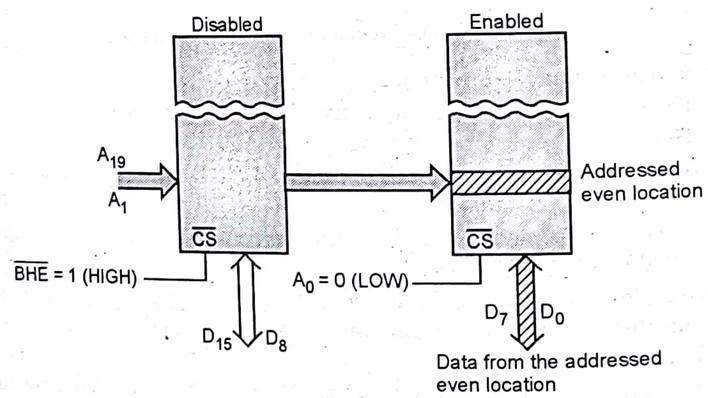


Fig. 11.8.3(a) : Accessing an even addressed byte

- The 8086 forces A_0 line LOW and $\overline{BHE}$ HIGH.
- This will enable the lower memory bank and disable the higher memory bank.
- The 8086 outputs the address of the desired even memory location on the address lines A_1 to A_{19}.
- The data stored (byte) at the addressed memory location appears on the data lines $D_0 - D_7$ as shown by the hatched lines in Fig. 11.8.3(a).

Case II : Accessing the odd addressed byte :

- Fig. 11.8.3(b) shows the process involved in accessing an odd addressed byte.

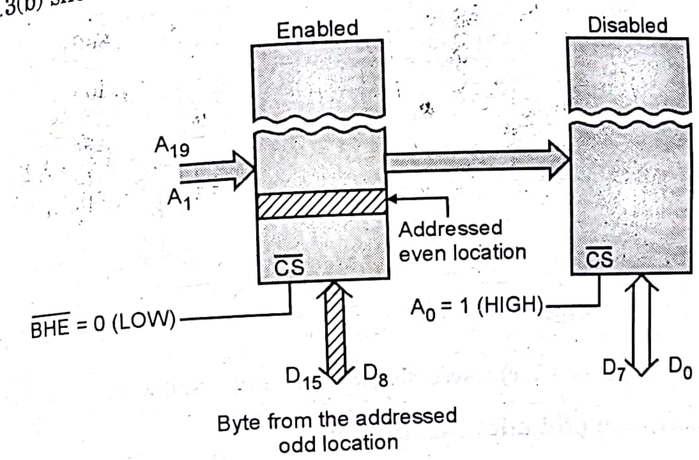


Fig. 11.8.3(b) : Accessing an odd byte

- The 8086 forces $A_0 = 1$ (HIGH) and $\overline{BHE} = 0$ (LOW).
- This will disable the lower (even) memory bank and enable the upper memory bank.
- The 8086 will output the address of desired odd memory location on A_1 to A_{19} address lines.
- The data byte on the addressed odd location appears on the data lines $D_8 - D_{15}$, as shown in Fig. 11.8.3(b).

Case III : Accessing an even addressed word :

- Here the 8086 wants to read a 16 bit word.
- As stated earlier, the lower byte of this word is stored first and the higher byte is stored at the next higher.
- This is illustrated in Fig. 11.8.3(c). The lower byte of the word has been stored at the even memory location A and the higher byte of the same word has been stored at the next location (odd memory location A + 1).
- Both the memory banks are enabled because 8086 will force A_0 as well as $\overline{BHE}$ low.
- The address on the lines $A_{19} - A_1$ will point towards the locations A and A + 1 simultaneously.
- The lower byte stored at location A will now appear on the data lines $D_0 - D_7$ and the higher byte stored at location (A + 1) will appear on the data lines $D_8 - D_{15}$ simultaneously.
- Thus to read a word from even address the 8086 needs only one bus cycle.

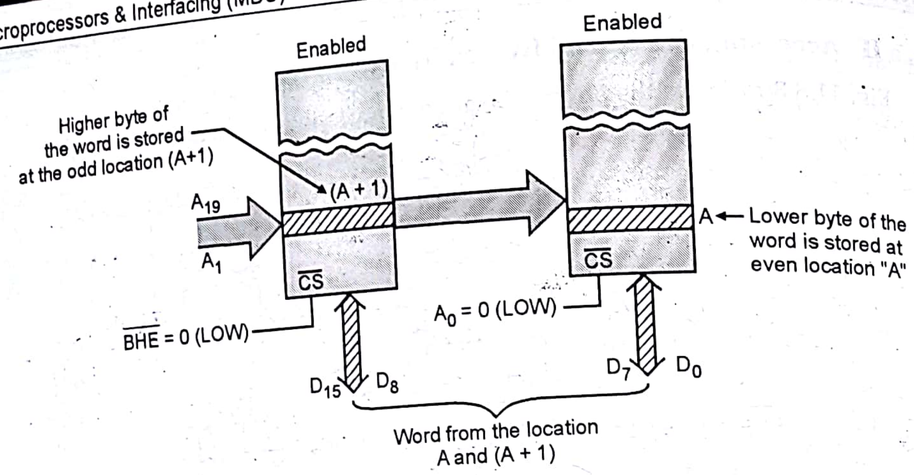


Fig. 11.8.3(c) : Accessing an even addressed word

Case IV : Accessing an odd addressed word :

- In this case the 8086 wants to read a word, the lower byte is stored at an odd address. (say A + 1) and the upper byte of the word is at even address (i.e. at B immediately next to A + 1).
- Now refer Fig. 11.8.3(d) and notice that the locations (A + 1) and B are not at the same level.
- Therefore the 8086 cannot use the same address for both these locations, (A + 1) and B.
- Therefore it is not possible for 8086 to output a single address and read all the 16 bits of the word simultaneously.

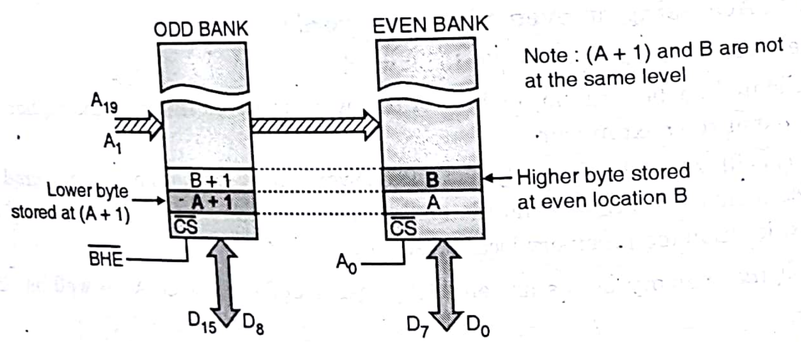


Fig. 11.8.3(d) : Accessing an odd addressed word

- Hence the 8086 needs two bus cycles, to read the 16 bit word. One bus cycle is required to read the contents of location (A + 1) and other to read the contents of location B. The sequence of operations in these two bus cycles is as explained below.

First bus cycle :

- In the first bus cycle, the 8086 forces A_0 = 1 (HIGH) and $\overline{BHE}$ = 0 (LOW) so as to enable the odd bank.
- It then outputs the address on A_1 – A_{19} lines to point at the location (A + 1).
- The contents of location (A + 1) will appear on the data lines D_8 to D_{15}, as shown in Fig. 11.8.3(e).

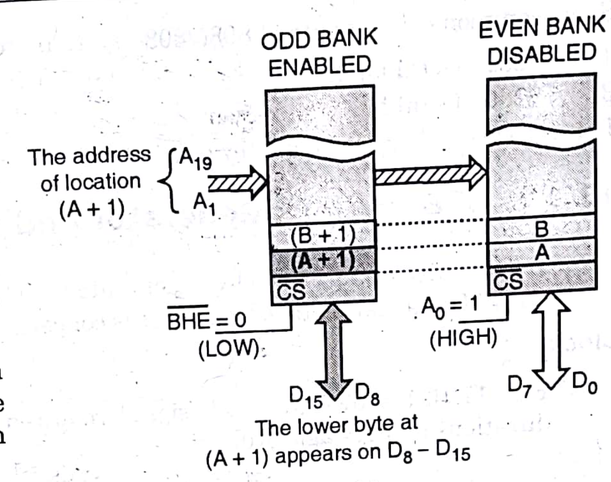


Fig. 11.8.3(e) : First bus cycle

Second bus cycle :

- In the second bus cycle the 8086 forces A_0 = 0 (LOW) and $\overline{BHE}$ = 1 (HIGH) so as to enable the lower (even) bank.
- It then outputs the new address on the A_1 – A_{19} lines to point at location B.
- The contents of location B will appear on D_0 – D_7 as shown in Fig. 11.8.3(f).

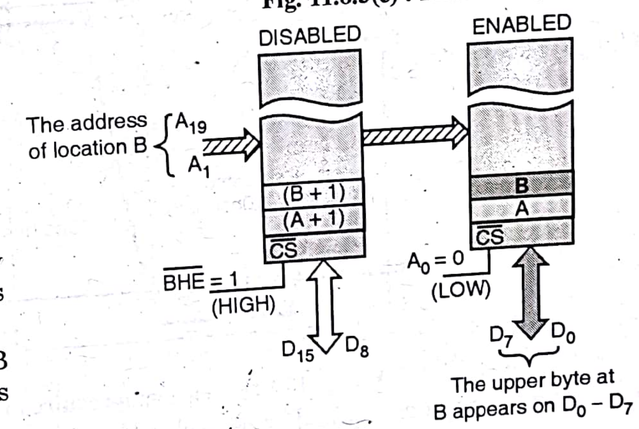


Fig. 11.8.3(f) : Second bus cycle

Conclusion :

Accessing a word which starts at the odd address takes two machine cycles i.e. longer time. Therefore generally it is preferred that the word operand should have lower byte in the even bank so that the microprocessor can read 16 bits at a time in single machine cycle.

- This can be achieved by using the assembler directive EVEN.
- The directive EVEN stands for align on even memory address.
- EVEN : EVEN is an assembler directive which informs the assembler to **align** the word variables at the even address boundaries

11.9 8086 Configurations

8086 can work in either of the two modes :

1. **Minimum mode :** In this mode 8086 is the only processor in the system. It is also called as **uniprocessor mode.**
2. **Maximum mode :** In this mode, multiple processors can be present in the system.

Some common components of 8086/8088 system are :

1. 8282 : Octal latch
2. 8284 : Clock generator and driver
3. 8286 : Octal bus transceiver
4. 8288 : Bus controller

Let us discuss these chips one by one.

11.10 8284 : Clock Generator and Driver

- Before discussing the clock generator let us understand the requirements of 8086/8088 as far as the clock input is concerned.

Clock

- Fig. 11.10.1 shows the clock signal required by 8086 with all the standard time durations and voltage levels,

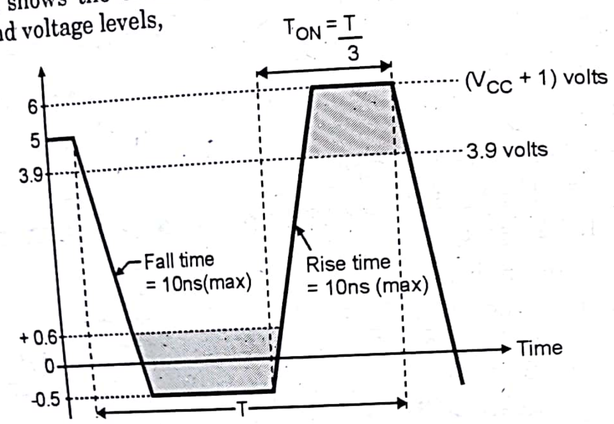


Fig. 11.10.1 : Clock signal required by 8086/8088

- The clock signal required by 8086 should satisfy the following specifications.

Specifications of clock signal

1.	Rise and fall times	:	10 nsec. (maximum)
2.	Maximum clock frequency	:	8 MHz for 8086 – 2
		:	5 MHz for 8086
		:	10 MHz for 8086 – 1
3.	Minimum clock frequency	:	2 MHz
4.	Duty cycle = (Ton / T) × 100	:	33%
5.	Low voltage range	:	– 0.5 V to + 0.6 V
6.	High voltage range	:	3.9 V to (V_{CC} + 1) V.

- The minimum clock frequency of 2 MHz has been decided by keeping the refreshing cycle of the dynamic RAM cells of the memory in mind.
- The task of designing an external clock oscillator is greatly simplified due to the intel IC 8284 which is a clock generator for 8086.

11.10.1 8284 Pin Configuration

- The internal block diagram and pin configuration of the clock generator 8284 are shown in Figs. 11.10.2(a) and 11.10.2(b) respectively.
- 8284 is a bipolar clock generator / driver which has been designed to provide the clock signals for 8086/8088 and peripherals.
- This chip mainly produces the following three signals for microprocessor

1. Clock 2. Ready 3. Reset.

- It also provides the READY 1 and READY 2 signals along with their corresponding address enable signals $\overline{AEN1}$ and $\overline{AEN2}$ for the multibus system.

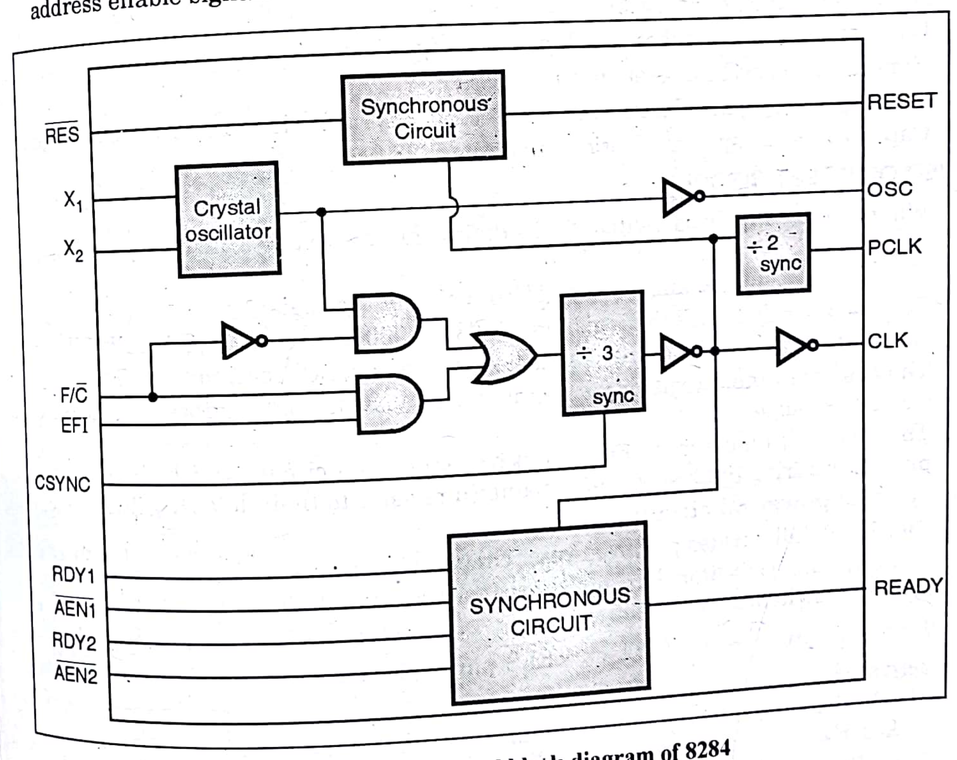


Fig. 11.10.2(a) : Internal block diagram of 8284

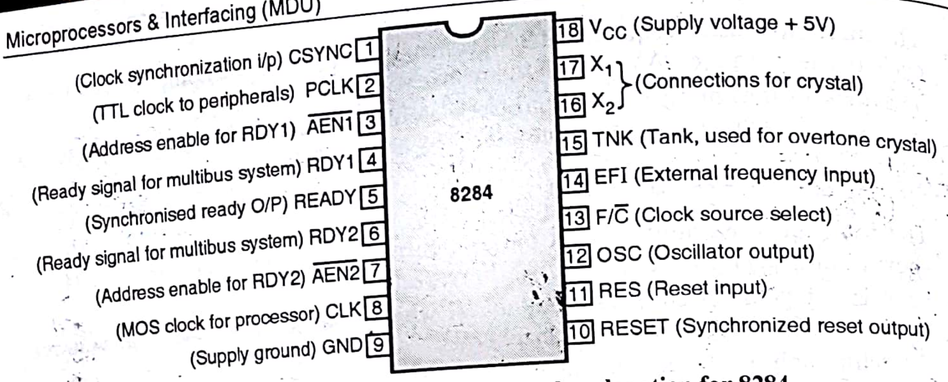


Fig. 11.10.2(b) : Pin configuration and explanation for 8284

11.10.2 Features of 8284

- Generates system clock signal for 8086, 8088 processors and the peripherals.
- Operates on a single + 5V supply.
- Available in the 18 pin package.
- Can use either a crystal or a TTL external signal as frequency source.
- Generates the system reset output.
- Provides the synchronication between local ready and microprocessor ready signals.
- Capable of clock synchronization with other 8284 chips.

POWER ON RESET CIRCUIT

- When power supply is switched on, there are some spikes in its output for some time.
- These spikes cause change in the default value of the registers.
- The default value of all the registers of 8086 is 0000H except for CS is FFFFH.
- Hence the address of the first instruction fetched by the processor is FFFF0H.
- The change in these register values alter the behaviour of the processor, especially if CS or IP change.
- This makes it necessary to implement power on reset circuit, which should reset the processor during these spikes and maintain register to their default values.

Operation of power on circuit

- Fig. 11.10.3 illustrates power-on-reset circuit.
- Initially the capacitor has no charge across it, when system is switched on.
- Hence it provides logic '0' on $\overline{RES}$ pin and reset is activated.
- Slowly the capacitor is charged through the resistor and makes the reset pin to logic '1'. The charging time of capacitor is such that by this time the spikes of power supply are vanished.
- This disables the reset, and the processor switches on.
- The diode is connected across the resistor to discharge the capacitor rapidly when power supply is switched off.
- The switch is for manual reset to discharge the capacitor, by connecting it to ground.

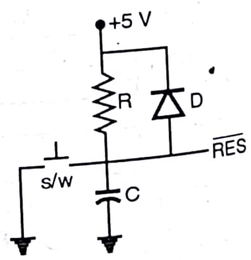


Fig. 11.10.3

11.10.3 Interfacing 8284 to 8086

The typical connections for 8284 and its interfacing with the 8086 processor is shown in Fig. 11.10.4.

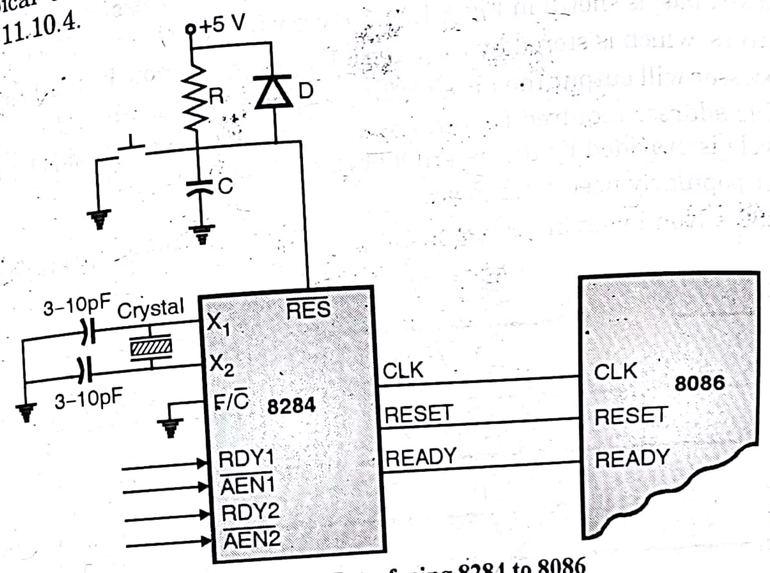


Fig. 11.10.4 : Interfacing 8284 to 8086

11.11 8282 /8283 : An Octal Latch

"A latch is a register with a specific purpose. This group of D flip-flops is used to hold values to be output to other devices."

For example, if you wanted to control a number of LEDs say for a level meter, a latch would hold the appropriate 1's and 0's to drive those LEDs while the processor went off to do other things.

Before going into the details of 8282/8283 octal latch let us see its utility and why do we need to use the latch.

Address data bus of 8086

- The address data bus of 8086 is a **multiplexed** bus. That means the same pins (AD_0 to AD_{15}) are being used as address lines as well as data lines.
- The same lines are being used to carry the address information as well as data, on the time sharing basis, and hence are to be demultiplxed.
- The remaining four address lines A_{16} to A_{19} also are multiplexed with the S_3 to S_6 signals. (A_{16} / S_3 to A_{19} / S_6),and hence are to be demultiplexed.
- Also the $\overline{BHE}$ / S_7 signal is to be demultiplexed.
- The 8086 first outputs the 20 bit address on the multiplexed lines. The presence of address on the multiplexed bus is indicated by forcing the ALE (address latch enable) pin of 8086 to HIGH.

- This address should be latched into external latches. The ALE signal is used to enable the external latches, to store the address.
- The ALE signal then goes low (0), before the address disappears from the multiplexed bus as shown in Fig. 11.11.1. This will disable the external latches.
- The address which is stored by the external latches can now be used as address bus.
- The processor will output the data on the multiplexed bus, after ALE goes low.
- The stable address required for the system memory and peripherals for the duration of bus cycle is provided by the external latches at their outputs.
- The most popularly used latches are :

1. 8282 : Non inverting octal latch.
2. 8283 : Inverting octal latch.

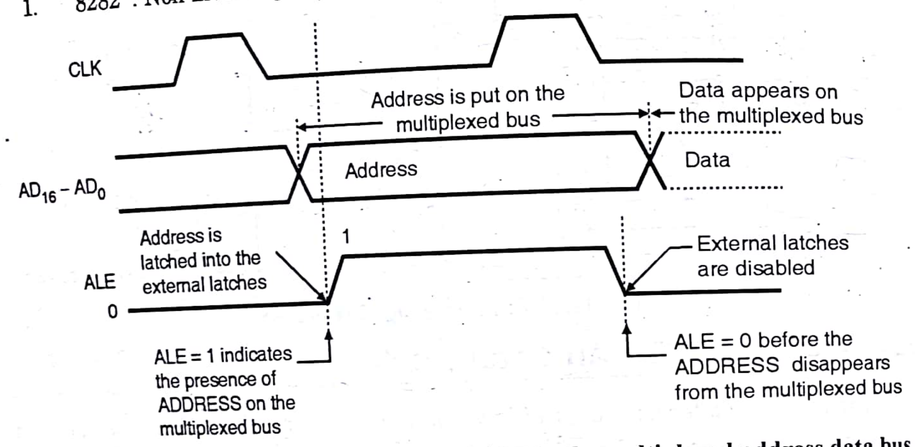


Fig. 11.11.1 : Timing diagram showing the role of ALE in the multiplexed address data bus

11.11.1 Use of External Latches for Address Latching

- Fig. 11.11.2 shows diagrammatically the use of external latches to latch the address.

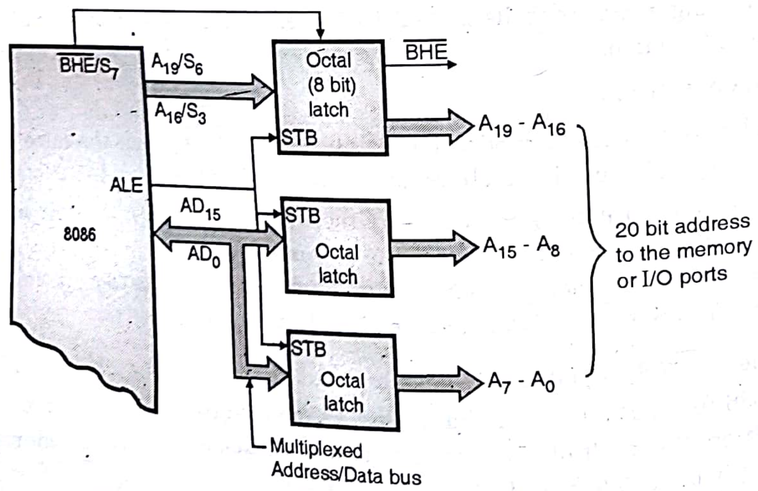
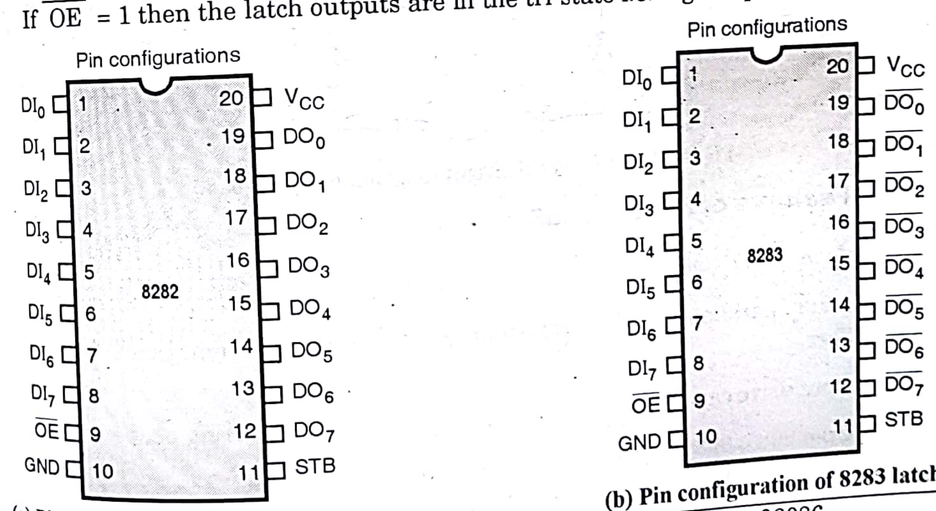


Fig. 11.11.2 : Use of external latches for address latching

- The ALE signal is being used as the strobe (STB) input for the latches, which enables the latches and loads the address into the latches.
- Since each latch is an octal i.e. 8-bit latch, we have to use three such latches in order to latch a 20 bit address.
- On receiving the HIGH (1) ALE signal from the processor, the latches will be enabled and the A_0 to A_{19} address is latched.
- Before the address disappears from the multiplexed bus, the ALE reduces to 0 and the latches are disabled. So they do not respond to the data appearing on the multiplexed bus.

11.11.2 Pin Diagram of Octal Latch 8282 and 8283

- Fig. 11.11.3(a) and (b) show the pin diagram for the octal latch ICs 8282 and 8283 respectively.
- The 8282/8283 are 8 bit bipolar latches with tristate output buffers.
- The 8283 is an inverting octal latch. So it inverts the input while 8282 is a non-inverting latch so it does not invert the input.
- The strobe input (STB) is an active high input which latches the input appearing on the 8 input lines DI_0 to DI_7 into the latch.
- Whereas the output enable ($\overline{OE}$) is an active low input which enables the output buffers and connects the latched information to the output lines DO_0 to DO_7.
- If $\overline{OE}$ = 1 then the latch outputs are in the tri-state i.e. high impedance T-state.



(a) Pin configuration of 8282 latch **(b) Pin configuration of 8283 latch**

$DI_0 - DI_7$	8 input lines	Connected to the multiplexed bus of 8086
$DO_0 - DO_7$	8 output lines	Produce the latched address
STB	Strobe input	If STB = 1, input is latched. STB is connected to ALE.
$\overline{OE}$	Output enable input	$\overline{OE}$ = 0 , outputs are enabled $\overline{OE}$ = 1, outputs in tristate

(c) Pin definition of octal latch IC 8282 and 8283

Fig. 11.11.3

Fig. 11.11.4 shows the internal circuit diagram of 8282 and 8283.

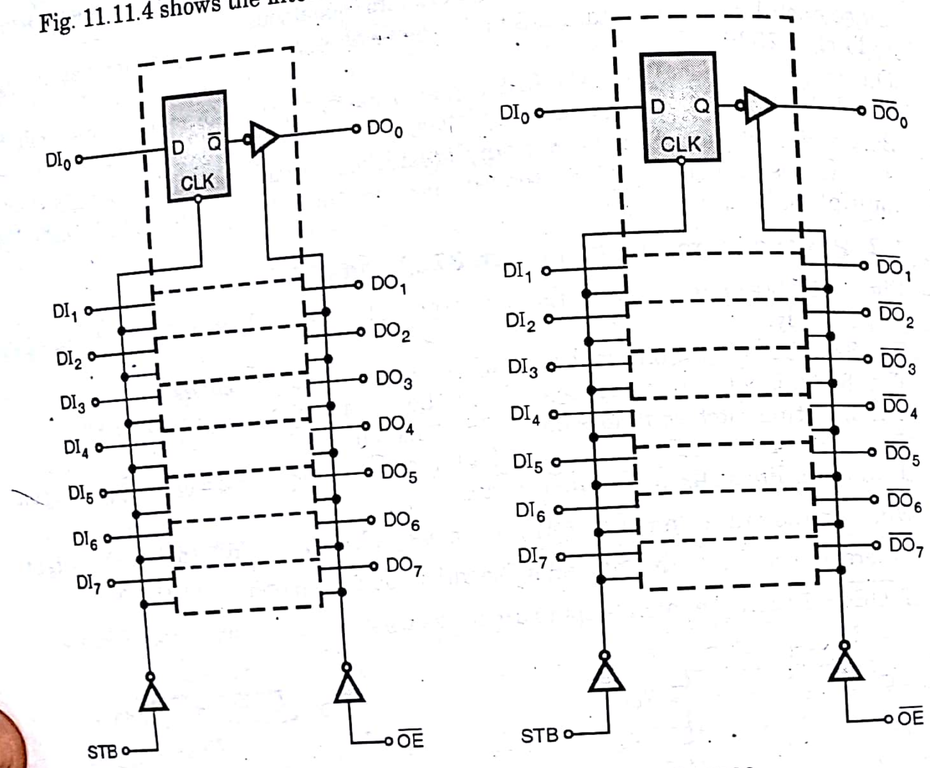


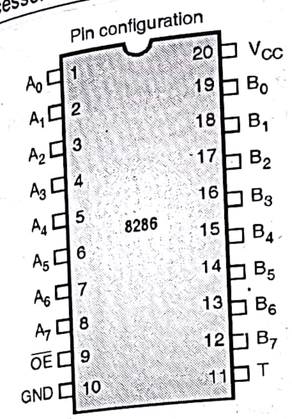
Fig. 11.11.4 : Internal circuit diagram of 8282 / 8283

11.11.3 Features of 8282 / 8283

- Available in 20 pin package.
- Contain fully parallel 8 bit data registers and buffers.
- High output drive capability.
- Support the 8085, 8086, 8088 microprocessor systems.
- Operate on single polarity supply.
- Tri-state (high impedance) outputs.
- Transparent during active probe.

11.12 8286 : Octal Bus Transceiver

- The 8286 is an octal bus transceiver which is used for obtaining the 16 bit data bus from the multiplexed address data bus of 8086 processor.
- The pin diagram of 8286 is shown in Fig. 11.12.1.



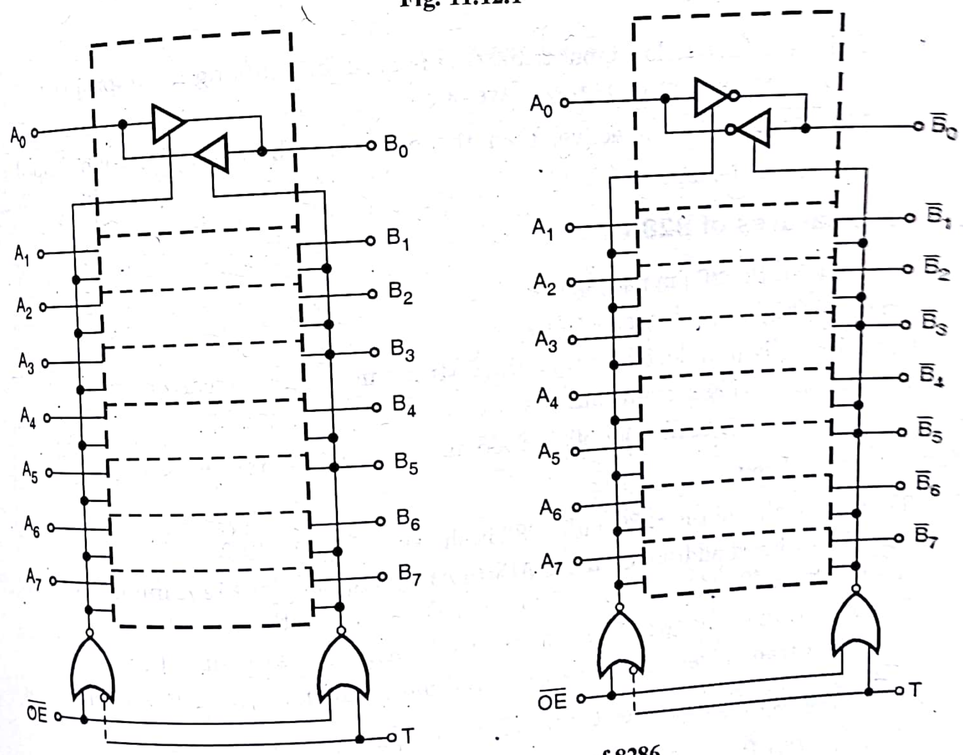
(a) 8286 pin configuration

Pin names

$A_0 - A_7$	Local bus data
$B_0 - B_7$	System bus data
$\overline{OE}$	Output enable
T	Transmit

(b) Pin names

Fig. 11.12.1



(c) Internal circuit diagram of 8286

Fig. 11.12.1

Description

1. $A_0 - A_7$

- These 8-lines are called as the local bus data lines. They can act as input lines or as output lines depending on the status of the transmit (T) pin.
- If T = 1, then $A_0 - A_7$ act as inputs, whereas if T = 0, then $A_0 - A_7$ will act as output lines.

2. **$B_0 - B_7$**
   - These 8-lines are called as system bus data lines. They too can act as input or output lines depending on the status of the transmit (T) pin.
   - If T = 1, then $B_0 - B_7$ will act as output lines whereas with T = 0, they will act as the input lines.

3. **T (Transmit)**
   - Pin 11 i.e. T or transmit pin is an input pin of 8286.
   - It controls the direction of the transceivers, as follows :
     - If T = 1; $A_0 - A_7$ act as inputs and $B_0 - B_7$ act as outputs.
     - If T = 0; $A_0 - A_7$ act as outputs with $B_0 - B_7$ acting as inputs.

4. **Output Enable ($\overline{OE}$)**
   - This is an active low input to 8286. It is used for enabling the appropriate driver to put his data onto the respective bus.
   - If $\overline{OE}$ = 1 i.e. in active, then the outputs will be in the high impedance (tri-state) T-state.

11.12.1 Features of 8286

- Available in the 20 pin package.
- Tri state (high impedance) outputs.
- Contains fully parallel 8 bit transceivers (transmitters and receivers).
- Capable of driving system data bus.
- Used as data bus buffers for 8085, 8086 and 8088 based systems.

Interfacing with 8086

- The interfacing of the 8286 with 8086 is shown in Fig. 11.12.2.
- The multiplexed address data bus AD_0 to AD_{15} is applied at the A inputs (A_0 to A_7) of the two 8286 ICs ($AD_0 - AD_7$ to first and $AD_8 - AD_{15}$ to the second).
- The B outputs of the two 8286 chips act as the demultiplexed data bus.
- The 8286 transceivers allow bi-directional data transfer. The T (transmit) input of both 8286 chips is connected to the DT/$\overline{R}$ output of 8086.
- So when DT/$\overline{R}$ = 0, T = 0, so B lines act as input lines, a lines act as the output lines and the data will flow from memory or ports to 8086, that means the read operation will take place.
- On the other hand if DT / $\overline{R}$ = 1, then T = 1 and the data will get transferred from the processor to the memory or ports.
- The $\overline{OE}$ lines of both the 8286 chips are connected to the $\overline{DEN}$ output of 8086.

This signal is used to enable or disable the transceivers. With $\overline{DEN}$ = 0, the transceivers are enabled and with $\overline{DEN}$ = 1 they are disabled and go into tri-state.

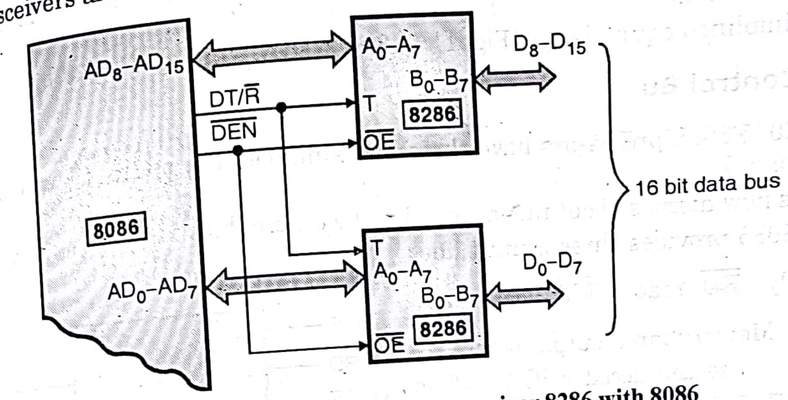
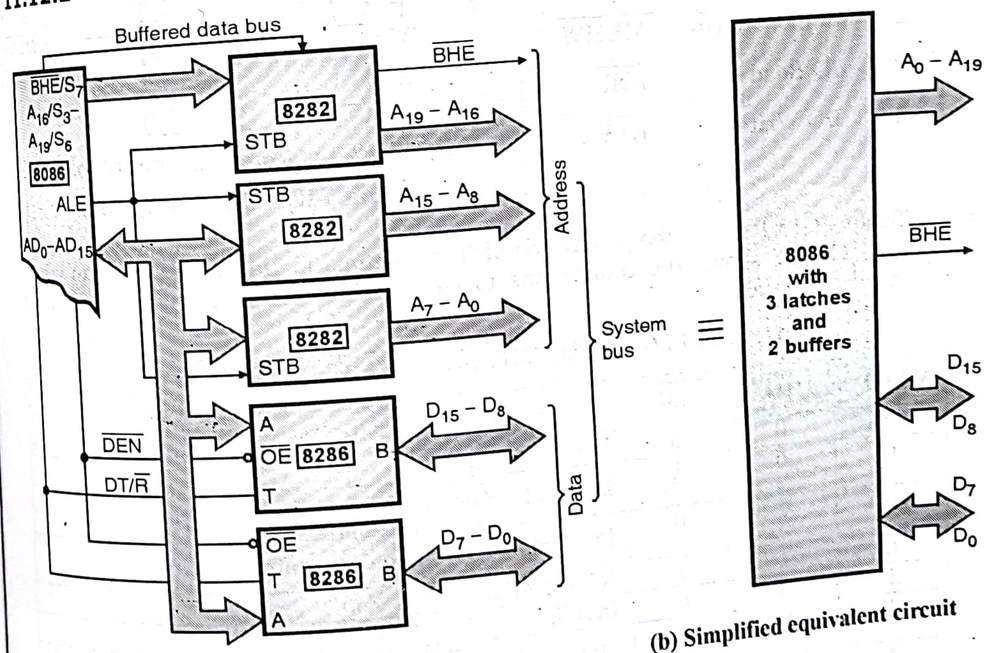


Fig. 11.12.2 : Interfacing of transceiver 8286 with 8086

11.12.2 Demultiplexing Address and Data Bus



(a) Demultiplexing the address and data buses (b) Simplified equivalent circuit

Fig. 11.12.3

- We have discussed the use of 8282 latches in address latching and 8286 as data bus buffer.
- If we use them simultaneously then we can demultiplex the address data bus as shown in Fig. 11.12.3(a).

- Thus we require three octal latch chips (8282) and two octal transceiver chips (8286) in order to completely demultiplex the address and data bus.
- The basic diagram of Fig. 11.12.3 is often required in the memory interfacing and I/O interfacing.
- The simplified equivalent of Fig. 11.12.3(a) is shown in Fig. 11.12.3(b).

11.12.3 Control Bus

- The 8086/8088 processors have three buses namely the address bus, data bus and control bus.
- Let us now discuss about the last one i.e. the control bus.
- The 8086 provides three control lines namely $\overline{RD}$ (read), $\overline{WR}$ (write) and $M/\overline{IO}$ (Memory/input output port).
- We will use the decoder IC 74138 to convert these three lines into the following four signals.

1. Memory Read $\overline{MEMR}$
2. Memory Write $\overline{MEMW}$
3. I/O Read $\overline{IOR}$
4. I/O Write $\overline{IOW}$

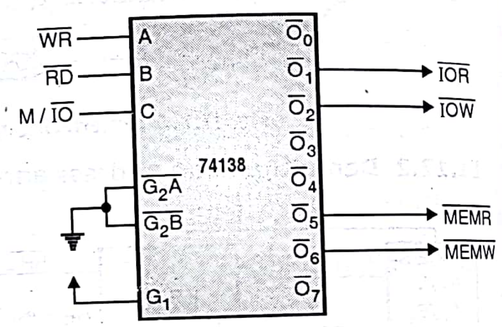


Fig. 11.12.4 : Generating control signals using 74LS138 decoder chip

- Table 11.12.1 shows the generation of these signals in the tabulated form whereas Fig. 11.12.4 shows the connections to be made to 74138 in order to obtain the required signals.

Table 11.12.1 : Generating $\overline{MEMR}$, $\overline{MEMW}$, $\overline{IOR}$, $\overline{IOW}$

$M/\overline{IO}$	$\overline{RD}$	$\overline{WR}$	Operation
0	0	0	Invalid
0	0	1	I/O read ($\overline{IOR}$ to be generated)
0	1	0	I/O write ($\overline{IOW}$ to be generated)
0	1	1	No operation
1	0	0	Invalid
1	0	1	Memory read ($\overline{MEMR}$ to be generated)
1	1	0	Memory write ($\overline{MEMW}$ to be generated)
1	1	1	No operation

We will use decoder 74138 to generate control signals.

11.13 8288 Bus Controller

- The 8288 bus controller is a 20 pin bipolar IC. It is used in the maximum mode configuration of 8086. The 8288 is used in the medium to large 8086 processing systems.
- The 8288 bus controller accepts the CLK signal along with $\overline{S}_0$, $\overline{S}_1$ and $\overline{S}_2$ outputs of 8086 and generates the command, control and timing signals at its output.
- It also provides the bipolar bus drive capability and optimizes the system performance.

11.13.1 Internal Block Diagram and Pin Diagram

- Fig. 11.13.1(a) shows the simplified internal block diagram of 8288 bus controller and Fig. 11.13.1(b) shows the pin diagram of 8288.

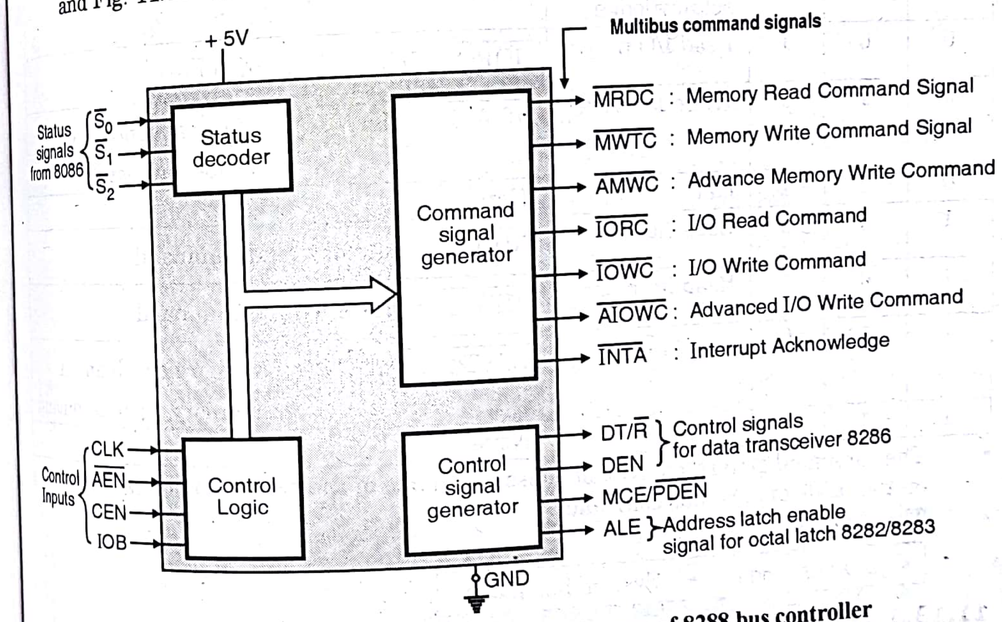


Fig. 11.13.1(a) : Simplified internal block diagram of 8288 bus controller

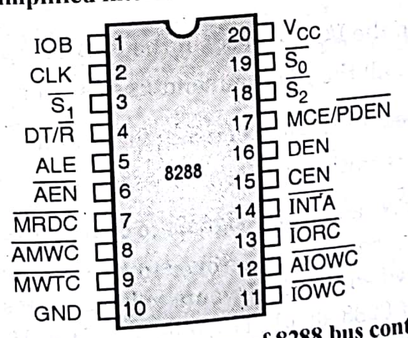


Fig. 11.13.1(b) : Pin diagram of 8288 bus controller

11.13.2 Command and Control Logic

- It is possible to use the 8288 bus controller with the 8086, 8088 and 8089 processors.
- The CPU status lines $\overline{S_0}$, $\overline{S_1}$ and $\overline{S_2}$ are applied as the inputs to the 8288 bus controller.
- The status decoder and command logic inside the 8288 will generate a command depending on the status of the $\overline{S_0}$, $\overline{S_1}$ and $\overline{S_2}$ lines, as given in the Table 11.13.1.

Table 11.13.1 : Relation between the command words and status words

Status of CPU			Status word (8086)	8288 command
$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$		
0	0	0	Interrupt acknowledge	$\overline{INTA}$
0	0	1	Read I/O port	$\overline{IORC}$: I/O read command
0	1	0	Write I/O port	$\overline{IOWC}$, $\overline{AIOWC}$: I/O or advanced I/O write command
0	1	1	Halt	None
1	0	0	Code Access	$\overline{MRDC}$: Memory read command
1	0	1	Read Memory	$\overline{MRDC}$: Memory read command
1	1	0	Write Memory	$\overline{MWTC}$, $\overline{AMWC}$: Memory or advanced memory write command
1	1	1	Passive	None

- The command generated by 8288 is issued in one of the two possible ways depending on the mode of 8288 bus controller.
- The modes of 8288 are :

  - I/O bus mode
  - System bus mode

11.13.3 I/O Bus Mode

- The 8288 operates in the I/O bus mode if the IOB pin (pin 1) is strapped HIGH (1).
- In the I/O bus mode, all the I/O command lines such as $\overline{IORC}$, $\overline{IOWC}$, $\overline{AIOWC}$, $\overline{INTA}$ are always enabled, independent of the $\overline{AEN}$ status.
- If an I/O command is initiated by the processor, then 8288 activates the command lines using the $\overline{PDEN}$ and DT/$\overline{R}$ signals to control the I/O bus transceivers.
- This mode allows the 8288 bus controller to handle two external buses and there is no waiting involved when the CPU wants to have access to the I/O bus.
- The I/O bus mode of 8288 should be preferred if the I/O or peripherals dedicated to single processor happen to exist in the multiprocessor system.

11.13.4 System Bus Mode

- The 8288 bus controller is in the system bus mode if the IOB input (pin 1) is strapped LOW (0).
- In this mode first the $\overline{AEN}$ line is activated (forced LOW), then there is a waiting period of 105 n sec during which no command is issued.
- After 105 n sec the required commands are issued.
- This mode assumes that the arbitration logic will inform the bus controller on its $\overline{AEN}$ line, about when the bus is free for use.
- Both the I/O and memory commands will wait for the $\overline{AEN}$ signal to get activated by the bus arbitration logic.
- The system bus mode is to be used when only one bus exists on which, both I/O and memory are shared by more than one processor.

Command outputs of 8288

- As shown in the internal block diagram of 8288 in Fig. 11.13.1(a), the 8288 bus controller produces the following seven command outputs.

  1. $\overline{MRDC}$: Memory read command
  2. $\overline{MWTC}$: Memory write command
  3. $\overline{IORC}$: I/O read command
  4. $\overline{IOWC}$: I/O write command
  5. $\overline{AMWC}$: Advanced memory write command
  6. $\overline{AIOWC}$: Advanced I/O write command
  7. $\overline{INTA}$: Interrupt acknowledge.

- The first four command outputs correspond to the read and write operations or the memory and I/O ports. They are simple straight forward commands.
- The fifth and sixth command outputs correspond to the advanced memory or I/O writing. The advanced write commands are made available by 8288 so as to initiate the write procedures early in the machine cycles.
- The advanced write command signals are used to avoid the processor entering into unnecessary wait T-state.
- The last command output is $\overline{INTA}$ i.e. interrupt acknowledge. This output command actually acts as the I/O read, during the interrupt cycle.
- The $\overline{INTA}$ signal tells the interrupting device that its interrupt has been acknowledged by 8086 and now it should place the required information on the data bus.

11.13.5 Control Outputs

- As shown in the internal block diagram of 8288 in Fig. 11.13.1(a) the 8288 bus controller produces the following four control outputs.

1. DEN	:	Data enable
2. DT/$\overline{R}$	:	Data transmit / Receive
3. MCE / $\overline{PDEN}$	:	Master cascade enable / Peripheral data enable
4. ALE	:	Address latch enable

DEN (Data enable)

- The DEN control output is applied to the $\overline{OE}$ input of the 8286 transceivers.
- This output determines when the external data bus is to be enabled and connected onto the local bus.

DT/$\overline{R}$ (Data transmit/receive)

- The DT/$\overline{R}$ decides the direction of data transfer. This control output is connected to transmit (T) input of the 8286 transceivers.
- The MCE / $\overline{PDEN}$ control output changes its T-state depending on the operating mode of 8288 bus controller.
- If 8288 is in the I/O bus mode (IOB), then the $\overline{PDEN}$ signal works as a dedicated data enable signal for the I/O or peripheral system bus.
- MCE (Master control enable) is used during the interrupt acknowledge cycle if 8288 is in the system Bus Mode (IOB pin LOW).
- During every interrupt cycle there are two interrupt acknowledge cycles which occur back to back.
- During the first interrupt cycle no data or address transfers take place. Therefore logic '0' should be provided to mask the MCE signal during the first cycle.
- Just before the beginning of the second cycle, the MCE signal gates the cascade address of master priority interrupt controller (PIC) onto the processors local bus. The ALE then strobes it into the address latches.
- On the leading edge of the second interrupt cycle, the addressed slave PIC will gate an interrupt vector onto the system data bus where it is read by the processor.
- But if a system contains only one PIC, then the MCE signal is not used. Then the second interrupt acknowledge signal simply gates the interrupt vector onto the processor bus.

ALE and halt

- The ALE (address latch enable) control output is connected to the strobe (STB) input of the 8282 latches.
- The ALE signal is activated in each machine cycle in order to strobe the current address appearing on the address bus into the address latches (8282).
- There is one more application of the ALE signal. It strobes the 8086 status ($\overline{S_0}$, $\overline{S_1}$, $\overline{S_2}$) into a latch for the status decoding.

Command enable (CEN) control input

- This is one of the four control inputs to the 8288 bus controller. If CEN = 1 then 8288 functions normally.
- But if CEN = 0, then all the seven command outputs are forced into their inactive T-state (and not the tristate).

11.13.6 Interfacing of 8288 to 8086

- The interfacing of the bus controller 8288 with the 8086 processor is shown in Fig. 11.13.2.
- Note that the MN/ $\overline{MX}$ pin of 8086 is connected to ground, which indicates that it is going to operate in the maximum mode.

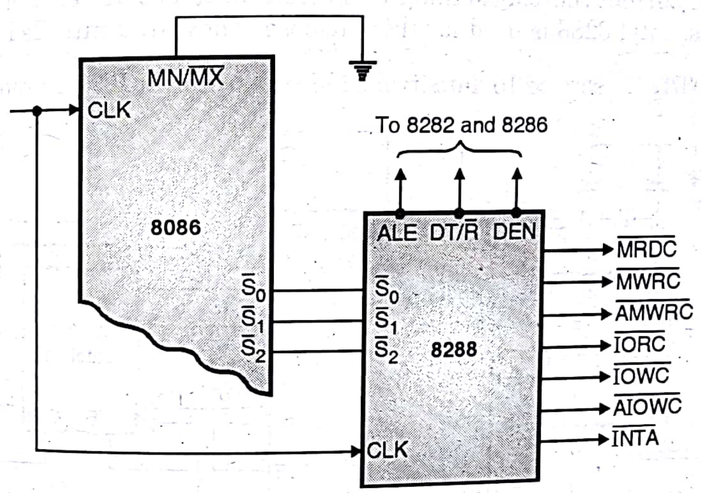


Fig. 11.13.2 : Interfacing of 8288 with 8086

11.13.7 Features of 8288 Bus Controller

- Available in 20 pin package.
- Operates on a single polarity supply.
- Generates the control and command signals useful for the operation in maximum mode.
- Bipolar drive capability.
- 3 T-state command output drivers.
- Configurable for use with the I/O bus.
- Facilitates the interface to one or two multimaster buses.
- Provides a wide flexibility in the system configurations.

11.14 Minimum Mode System Block Diagram

- The minimum mode system block diagram is as shown in Fig. 11.14.1.
- As shown in Fig. 11.14.1, the system in the minimum mode contains the support chips such as 8282, 8284 and if necessary, the 8286 data buffers.
- As shown in Fig. 11.14.1 the signals $AD_0 - AD_{15}$, $A_{16} / S_3 - A_{19} / S_6$ and $\overline{BHE} / S_7$ are multiplexed. These signals are demultiplexed by external latches and the ALE signal. The latches are generally buffered output D-type flipflops like 74LS373 or 8282. If 16 bit address is used then two latches are required and if full 20 bit address is used then three latches are required. These latches provide increased output drive capacity.
- If the microprocessor system has several devices that are interfaced with it, then to increase the current sourcing/sinking it is essential to use drives and transreceivers for data bus. Intel 8286 is used for this purpose. They are controlled by two signals $\overline{DEN}$ and $DT/\overline{R}$. To service 16 data, two 8286 transreceiver ICs are required.

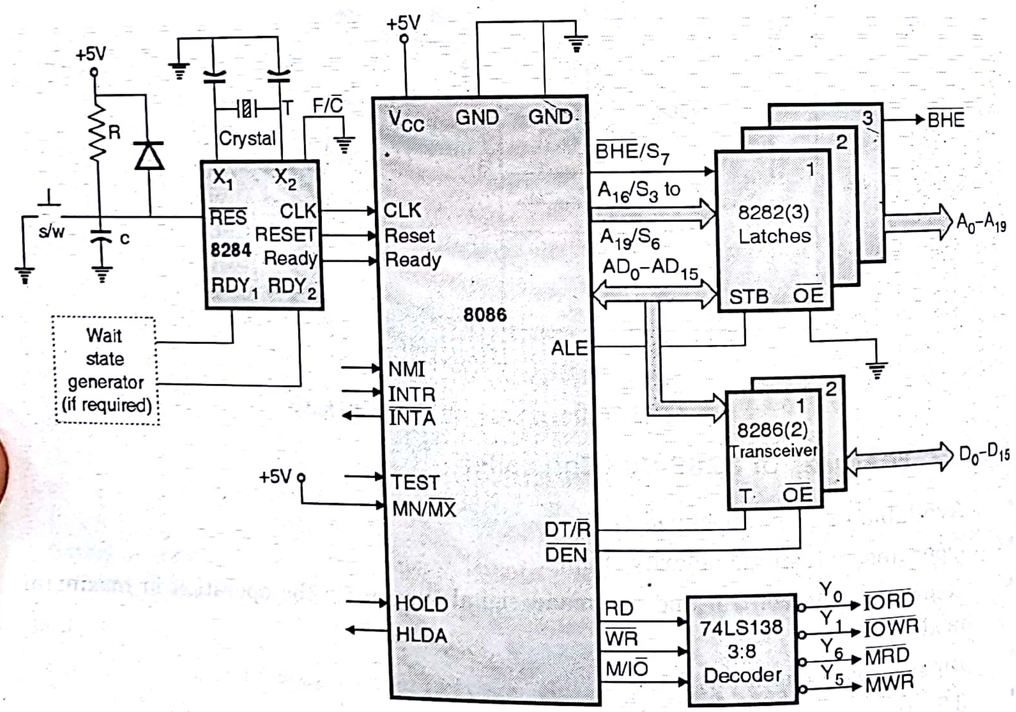


Fig. 11.14.1 : Block diagram of minimum system in minimum mode

- The $\overline{DEN}$ signal indicates that valid data is available on the data bus, while the $DT/\overline{R}$ is responsible for indicating the direction of data to or from the processor. At the time of data transfer the $\overline{OE}$ pin should be active low.

- The 8284 clock generator provides a clock pulses at constant frequency. The clock generator is synchronized with the READY signal and some external signals. The $\overline{RES}$ signal, initializes the system with clock pulses.
- The status on the lines $M/\overline{IO}$, $\overline{RD}$ and $\overline{WR}$ will decide the type of operation I/O read, I/O write, memory read or memory write. The HOLD and HLDA signals are used to interface with other bus masters. The INTR and $\overline{INTA}$ signals are used to increase the interrupt handling capacity of the 8086.

11.15 Maximum Mode System Block Diagram

- Fig. 11.15.1 shows the block diagram of the 8086 system in maximum mode.

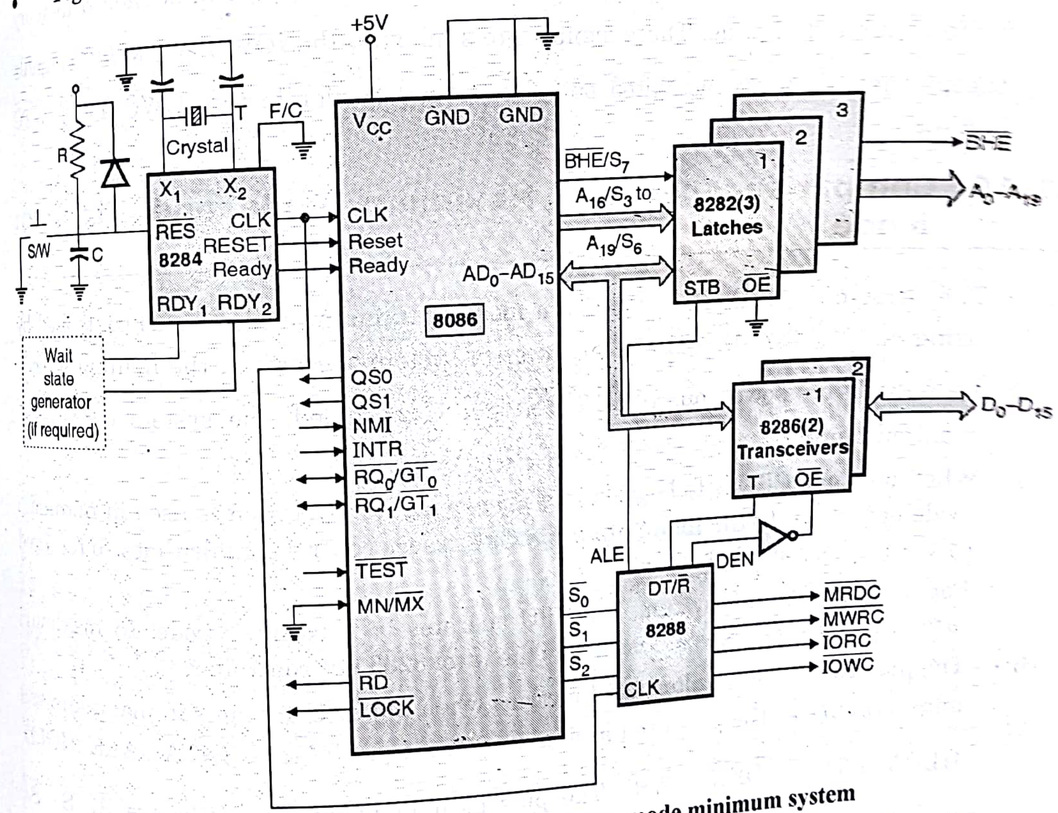


Fig. 11.15.1 : Block diagram of maximum mode minimum system

- Note the use of the bus controller IC 8288 alongwith the other support chips. Additional circuitry is required to translate the control signals. The additional circuitry converts the status signals ($S_2 - S_0$) to I/O and memory transfer signals. The Intel 8288 bus controller is used for this purpose. It generates the control

signals required to direct the data flow and for controlling the latches 8282 and transreceivers 8286.

- It generates the signals $\overline{MRDC}$, $\overline{MWTC}$, $\overline{AMWC}$, $\overline{AIOWC}$, $\overline{IORC}$, $\overline{IOWC}$ signals.
- The $\overline{MRDC}$ and $\overline{MWTC}$ are memory read command and memory write command signals. They instruct the memory to accept or send data on the data bus.
- The $\overline{IORC}$ and $\overline{IOWC}$ are I/O read command and I/O write command signals. They instruct the I/O device to read or write data to and from addressed port on the data bus.
- The $\overline{AIOWC}$ and $\overline{AMWTC}$ are advanced I/O write command and advanced memory write command signals. These signals are similar to the $\overline{IOWC}$ and $\overline{MWTC}$ signals except that, they are activated one clock signal earlier to the $\overline{IOWC}$ and $\overline{MWTC}$ signals.

11.16 Comparison between Maximum Mode and Minimum Mode

(i) The mode of operation i.e. maximum mode / minimum mode can be controlled by using pin 33 MN/ $\overline{MX}$. The microprocessor system operates in the minimum mode by default on power on. If MN/ $\overline{MX}$ pin is grounded, the system operates in maximum mode.

(ii) When only one CPU is to be used in a microcomputer system it is used in minimum mode operation. In the minimum mode CPU issues control signals that are required by the memory and I/O devices.

For a multiprocessor system, the CPU operates in maximum mode. In maximum mode the control signals are issued by the Intel 8288 bus controller.

(iii) The pins 24 to 31 have alternate functions when the CPU operates in maximum and minimum mode. The pins used in minimum mode are M/ $\overline{IO}$, $\overline{INTA}$, ALE, HOLD, HLDA, DT/ $\overline{R}$, $\overline{DEN}$, $\overline{WR}$. The pins used in maximum mode are $\overline{S}_2$, $\overline{S}_1$, $\overline{S}_0$, $\overline{LOCK}$, QS_1, QS_0, $\overline{RQ}_0/\overline{GT}_0$, $\overline{RQ}_1/\overline{GT}_1$.

(iv) Minimum mode operation is less expensive because all the memory and I/O control signals are generated by the microprocessor itself. In maximum mode we require an external bus controller like 8288 that generates the control signals. Hence, maximum mode operation is expensive.

11.17 8086 Timing Diagrams

- All the operations in 8086 are carried out with a *particular sequence* in a synchronized manner.
- So to understand every operation it is necessary to understand the *timing diagrams*.
- Before going into the details of the timing diagrams for the read and write operations, we will first define some of the important terms.

11.17.1 Clock Waveform

The clock waveform represents the crystal controlled signal sent to 8086 from an external clock generator using 8284.

11.17.2 T-State

- One complete cycle of clock is called as T- state, as shown in Fig. 11.17.1. The time intervals T_1 or T_2 are the examples of T-state.
- A T-state is measured from the falling edge of one clock pulse to the falling edge of the next clock pulse.

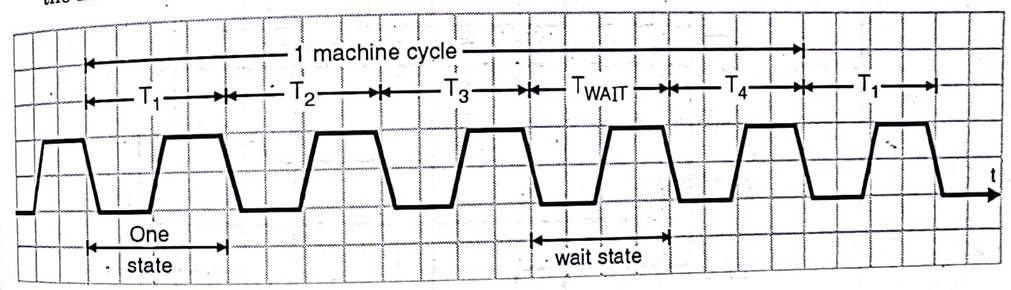


Fig. 11.17.1 : Clock, T-state, machine cycle etc.

- Various versions of 8086 have maximum clock frequency from 5 MHz to 10 MHz. Hence the minimum time for one T- state is between 100 to 200 n sec.

11.17.3 A Machine Cycle

- The basic microprocessor operation such as reading a byte from I/O port or writing a byte to memory is called as machine cycle.
- The time T_{cy} in Fig. 11.17.1 is called as the machine cycle. Thus a machine cycle consists of several T- states.

11.17.4 Instruction Cycle

- The time a microprocessor needs to fetch and execute one entire instruction is known as an instruction cycle.

Summary

1. A machine cycle is made up of many T- states.
2. A T- state corresponds to the duration of one clock cycle.

11.18 Read Machine Cycle of 8086 (Minimum Mode)

- Fig. 11.18.1(a) shows the timing diagram which shows the activities of various signals during the read operation.

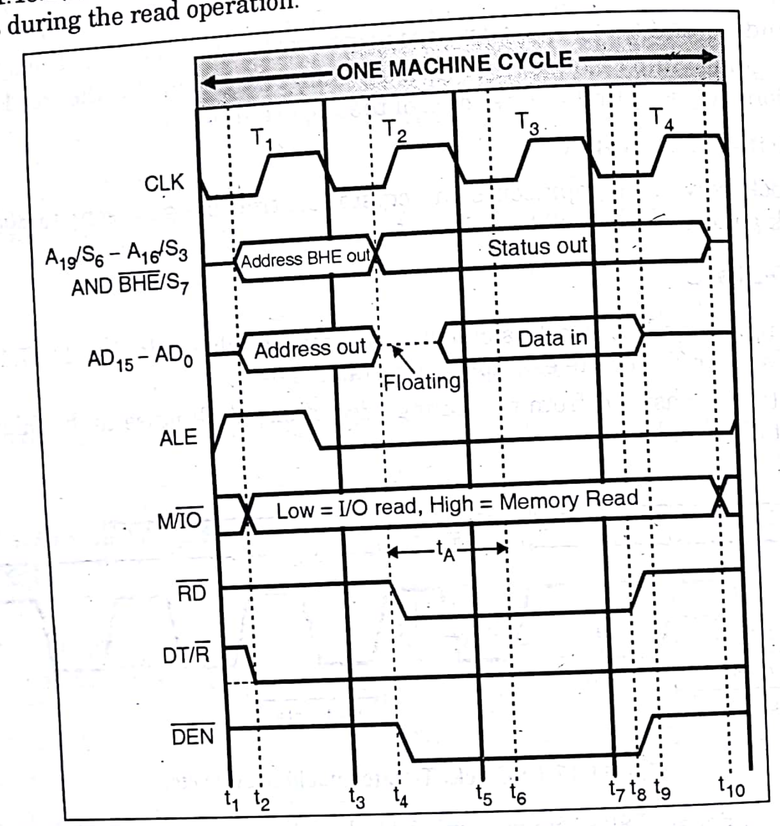


Fig. 11.18.1(a) : Timing diagrams for the read machine cycle of 8086

- The sequence of operations during the read machine cycle are as follows :

Step 1 : The 8086 will make $M/\overline{IO}$ = 1 if the read is from memory and $M/\overline{IO}$ = 0 if the read is from the I/O device.

Step 2 : At about the same time the ALE output is asserted to 1.

Step 3 : Make $\overline{BHE}$ low/high and send out the desired address on AD_0 to AD_{15} and A_{16} to A_{19} lines.

Step 4 : Pull down ALE (make it 0). The address is latched into external latch.

Step 5 : Remove the address from AD_0 to AD_{15} lines and put them in the input mode (float them).

Step 6 : Assert the $\overline{RD}$ (read) signal low. This will put the data from the addressed memory location or I/O port on to the data bus.

Step 7 : Insert the "wait" T- states if the 8086 READY input is made low before or during the T_2 state of a machine cycle.

Step 8 : As soon as READY input goes high, 8086 comes out of the wait T-states and completes the machine cycle.

Step 9 : Complete the "Read" cycle by making the $\overline{RD}$ line high (inactive).

Step 10 : For larger systems we need to use the data buffers. (8286 transceivers). Then the $DT/\overline{R}$ and $\overline{DEN}$ signals of 8086 are connected to 8086 and enabled at the appropriate time.

Description of read machine cycle

Step 1 : Assert $M/\overline{IO}$

- Refer Fig. 11.17.1 (a). During T_1 state of the read machine cycle, 8086 first asserts the $M/\overline{IO}$ signal as follows :

  $M/\overline{IO}$ = 1 → If reading is to be performed on memory.

  $M/\overline{IO}$ = 0 → If reading is to be performed on I/O port.

- The point of cross section of the $M/\overline{IO}$ signals in Fig. 11.17.1(a), indicates the time instant at which the $M/\overline{IO}$ signal becomes valid in this machine cycle.

Step 2 : Force ALE high :

- At about the same instant, 8086 asserts ALE high. This signal acts as strobe (STB) input for the external latches (8282).

Step 3 : Send out the desired address :

- After making ALE high, the 8086 sends out the valid address of the memory location or I/O port to be read, on the address lines AD_0 – AD_{15} and A_{16} – A_{19}.

Step 4 : Pull ALE down :

- Before the address on the multiplexed bus disappears, the ALE line is pulled down i.e. made inactive.
- The address gets latched into the external latches on the low going edge of ALE signal.
- The latched address will continue to appear at the output of the external latches (8282).

Step 5 : Float the multiplexed address / data bus :

- The 8086 then floats the AD_0 – AD_{15} lines so that they can be used to input (read) the data from the memory or I/O ports.

- At about the same time, the $\overline{BHE}$ signal is removed, and the $A_{16} - A_{19}$ lines will be used for sending the status information.

Step 6 : Assert $\overline{RD}$ signal low :

- The 8086 is now ready to read data from the addressed memory location or port. Therefore near the end of state T_2, the 8086 will assert $\overline{RD}$ signal low.
- The $\overline{RD}$ signal is used to enable the addressed memory device or port device. So, as $\overline{RD} = 0$, the addressed memory or port device is enabled.
- When enabled, this device will put its byte of data on the data bus.

Memory access time (t_A)

- Refer to the waveforms of Fig. 11.18.1(a), in which the memory access time has been marked. It is defined as the time taken by the memory to output valid data after it receives an address and a $\overline{RD}$ signal.
- The memory access time should be as short as possible. But practically it is not so short.
- If the memory access time is not short enough, then the memory cannot put a valid data on the data line, immediately after receiving the address and $\overline{RD}$ signal.
- The 8086 will then treat whatever garbage (invalid data) present on the data bus as valid data and continue with the next machine cycle.
- Such a garbage is likely to cause the entire program to crash down. To avoid this the 8086 should wait for the memory to put its valid data on data bus. This can be implemented by the use of READY pin.

Steps 7, 8 : Concept of wait T-states :

- In order to make the 8086 wait for the memory to put the valid data, we have to make use of the "READY" input of the 8086.
- When the "READY" input is high, 8086 operates normally.
- But if the READY input is made low at the appropriate instant in a machine cycle, then the 8086 will insert one or more "WAIT" T-states between the T-states T_3 and T_4 in the machine cycle as shown in Fig. 11.18.1(b).

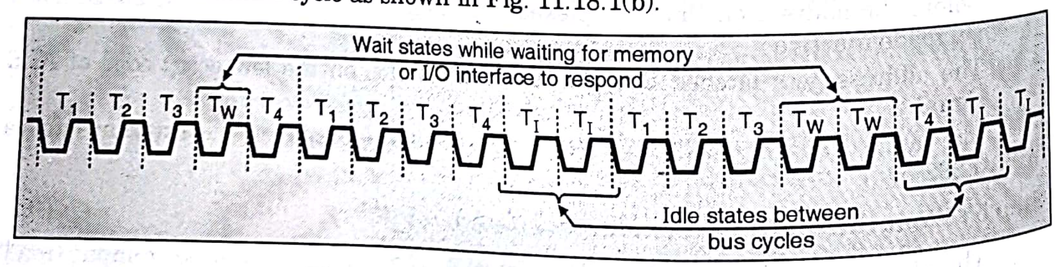


Fig. 11.18.1(b) : Concept of WAIT T-states

- An external device may be used to produce the low ready pulse, before the rising edge of clock in T-state T_2.
- Then 8086 finishes the T-state T_3 of the machine cycle and enters into a WAIT T-state.
- During the WAIT T-state all the signals on various buses will remain as they were before the beginning of WAIT T-state.
- In other words all the buses will freeze during the WAIT T-state. T_{WAIT}.
- The memory or port devices will have atleast one more clock cycle (T_{WAIT}) to put their valid data on the data bus.
- If READY input is made high during the T_3 or during the wait T-state, then after one WAIT T-state, the 8086 will move to the regular T_4 state of the machine cycle as shown in Fig. 11.18.2.
- But if the READY input is still low at the end of the first WAIT T-state, then 8086 will insert one more wait T-state.

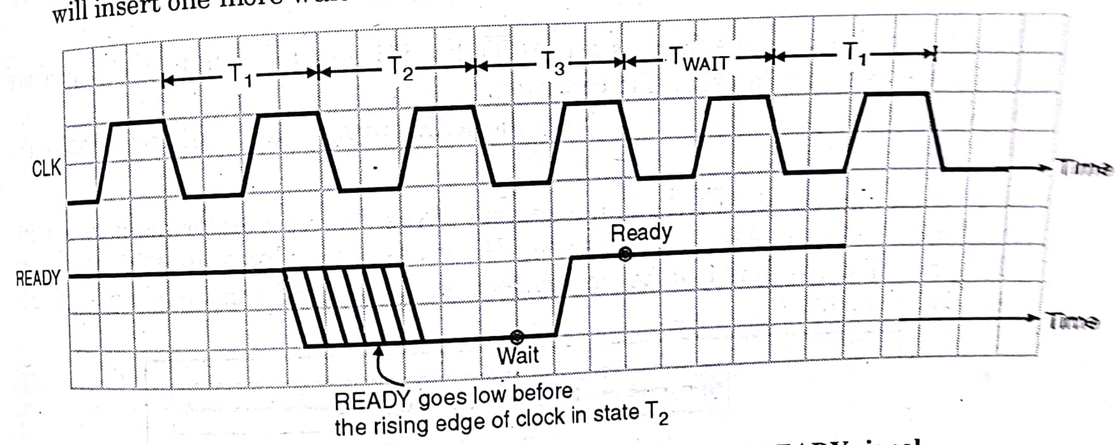


Fig. 11.18.2 : Relation between the WAIT T-states and READY signal

Note : The 8086 will continue to insert the WAIT T-states as long as the READY signal is LOW.

Advantage of using the WAIT T-states

- Due to insertion of wait T-states, the slow peripheral devices will get some time to put their valid data on the data bus.
- Hence the fast processor such as 8086 will be adequately slowed down to work successfully with the slow peripherals.
- The READY input signal is generally passed to 8086 through the 8284 A clock generator. This is done in order to synchronize the READY signal with the system clock.

Step 9 : Function of $\overline{DEN}$ and DT/$\overline{R}$ signals :

- If the external data buffers are used then the $\overline{DEN}$ and DT/$\overline{R}$ signals will come into play. Refer to the waveforms shown in Fig. 11.18.1 (a).

- During the T-state T_1 of the machine cycle, the DT/$\overline{R}$ signal is pushed LOW. This will put the data buffers into the receiver mode.
- Then as soon as sending of address on the multiplexed bus is complete, the $\overline{DEN}$ signal is asserted low to enable the data buffers.
- The data put by the addressed memory or I/O port will then be passed through the external buffers to 8086.

11.19 8086 Write Machine Cycle in the Minimum Mode

- Let us now see the activities taking place in the write machine cycle of 8086 in the minimum mode.
- The timing diagram for the write machine cycle is shown in Fig. 11.19.1. This timing diagram is very much similar to the timing diagram of the read machine cycle discussed earlier.

1. During the T_1 state, the 8086 asserts M/$\overline{IO}$ low if it has to write to an I/O port and high if the memory location is to be written.
2. At about the same time, it will push the ALE output high. This will enable the address latches.
3. 8086 then outputs the $\overline{BHE}$ and the address of the desired port or memory location on the lines AD_0 to AD_{15} and A_{16} to A_{19}. Note that the address lines A_{16} to A_{19} will always be low if a port is to be written because the port address is always going to be a 16 bit address.

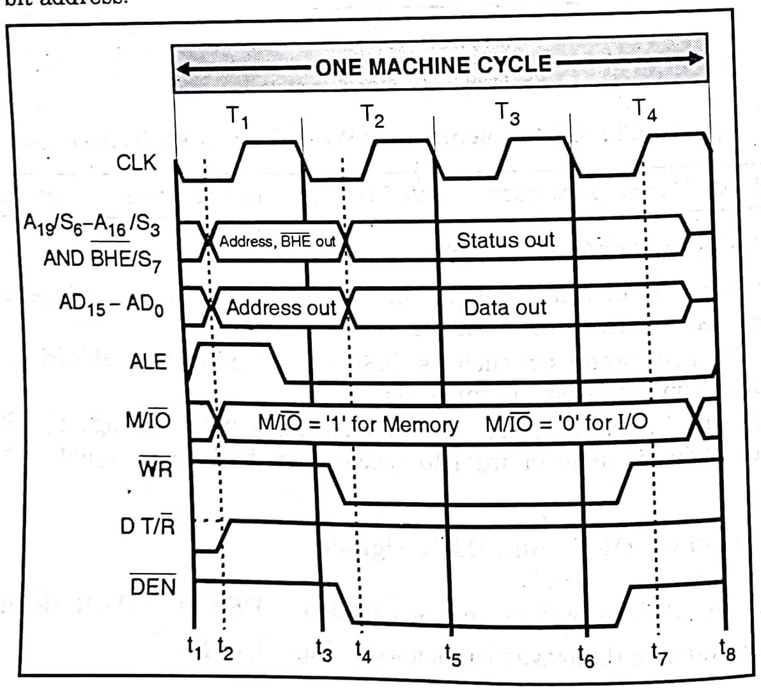


Fig. 11.19.1 : Write machine cycle in the 8086 minimum mode with wait T-state

4. After some time the ALE output is pulled down. The address gets latched into the external latches.
5. The 8086 removes the address information from $AD_0 - AD_{15}$ and outputs the desired data on the data bus.
6. It then asserts the $\overline{WR}$ signal low. The low $\overline{WR}$ signal will turn on the memory or port where the data is to be written.
7. 8086 then gives some time for the addressed I/O port or memory to accept the data from the data bus and then raises the $\overline{WR}$ output high and floats the data bus.
8. If the addressed memory or port devices cannot accept the data within the normal machine cycle, then the external hardware can be set up to produce a low READY input.
9. If the READY input is pulsed low before or during the T-state T_2 of the write machine cycle, then the 8086 will introduce WAIT T-states after T_3 as long as the READY input is held low.
10. During the WAIT T-states, the signals on the data bus, address bus and control bus do not change.
11. If the READY input is made high before the end of a WAIT T-state, then 8086 will continue with the T_4 state as soon as it finishes the WAIT T-state.
12. If the system is large enough then it will need the external data buffers (IC 8286). Then the DT/$\overline{R}$ is used to decide the direction of data transfer.
13. During the write cycle, the 8086 pushes the DT/$\overline{R}$ high to configure the external data buffers in the "transmit" mode. Then the $\overline{DEN}$ signal is asserted low so as to enable the buffers.
14. The data output from 8086 is then passed through the data buffers to the addressed memory or I/O port.

11.20 Maximum Mode Read/Write Cycle

Read Cycle

Fig. 11.20.1 shows read cycle in a maximum mode of 8086. The cycle is identical to read cycle of 8086 in minimum mode of operation. The few differences we have those are as follows :

(1) Status lines $\overline{S}_2$-$\overline{S}_0$ lines are taken into account. These lines are active for T_1 and T_2 cycle. After that they are inactive.

(2) ALE, Memory Read, I/O Read, DT/$\overline{R}$, DEN are generated by 8288 bus controller. They are not generated by microprocessor directly.

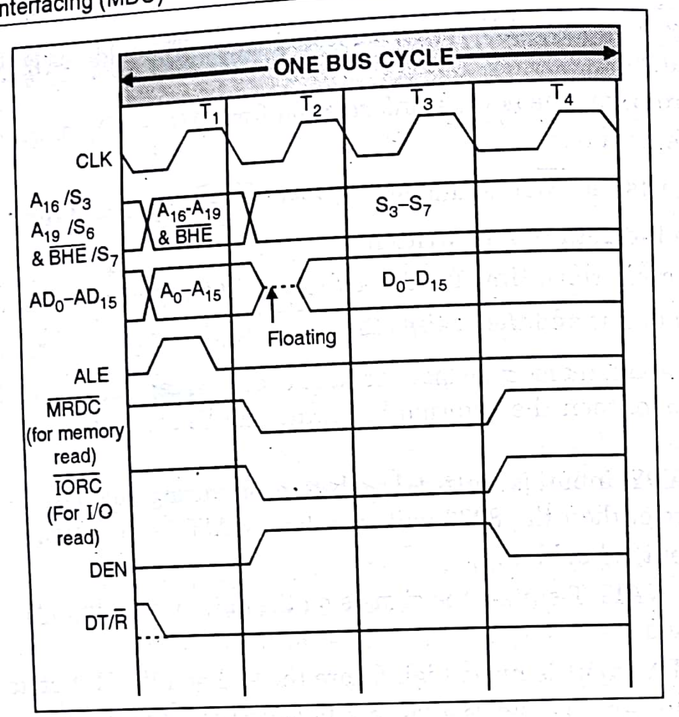


Fig. 11.20.1 : Maximum mode read bus cycle

Write Cycle

Write cycle of 8086 in maximum mode is again similar to write cycle in minimum mode. Differences are already mentioned in points, in read cycle.

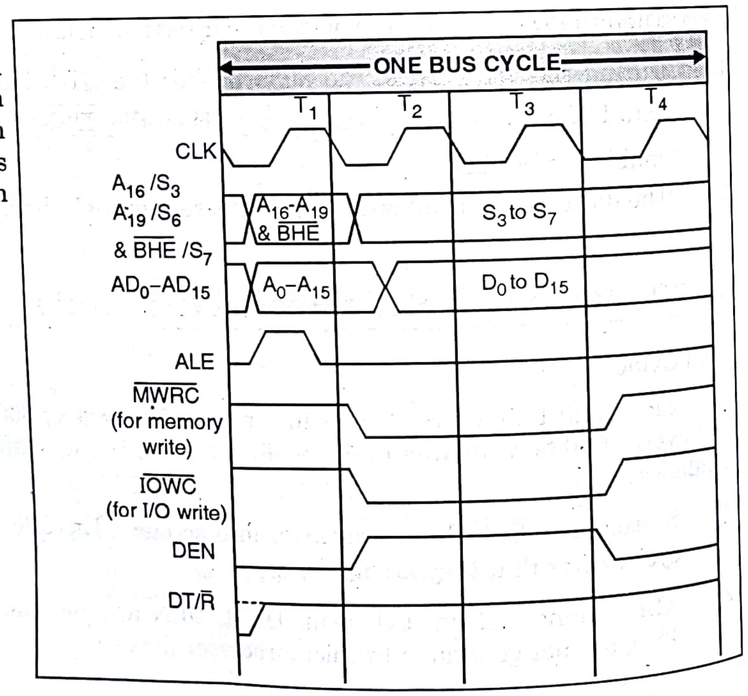


Fig. 11.20.2 : Maximum mode write cycle

CHAPTER

12

Addressing Modes and Instruction Set

12.1 Programmer's Model of 8086

Q. Explain in detail the programmer's model of 8086 microprocessor.

The 8086 programmer's model is a picture of the processor as available to the programmer. These are the registers used to hold numbers and addresses, as well as indicate status and act as controls.

Data Registers (16 bits each for AX, BX, CX and DX)

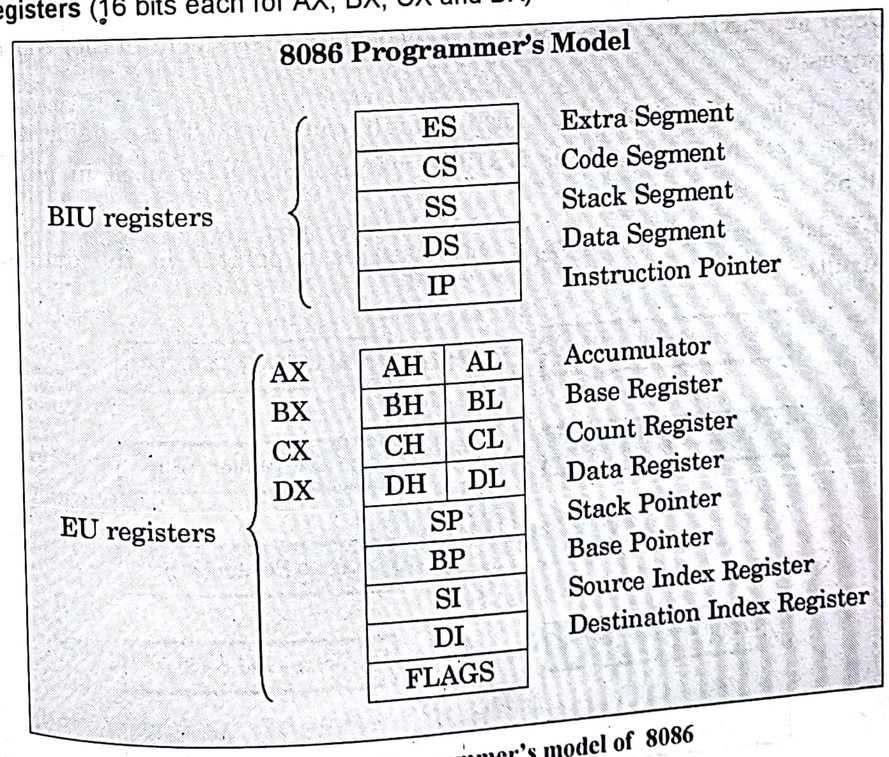


Fig. 12.1.1 : Programmer's model of 8086

- For 16-bit data registers can be accessed by names AX, BX, CX and DX,
- Individual bytes of these registers can be accessed by name, AH, AL, BH, BL, CH, CL, DH and DL.

Address Registers (note : you need both a segment and an offset to specify an address)

- The segment register are CS, DS, ES and SS and offset is stored in instruction pointer IP or stack pointer SP or base registers BX or BP, or one of the index registers SI or DI.
- The CS:IP pair gives the address of the next instruction to be executed in the program sequence.
- The SS:SP pair gives the address of the top of the stack, a temporary storage area often automatically used by the computer. SP is used for sequential access of the stack.
- The SS:BP pair is used as a pointer into the stack. BP is used for random access of the stack.
- The DS:SI pair is used as a source pointer and ES : DI pair is used as destination pointer for string instructions. For all other instructions DI register is used with DS segment register.
- The DS:SI and ES:DI registers are used as general purpose pointers for copying and data processing.
- The dotted lines below show common default couplings. These defaults can be overridden by a notation such as DS:[BP], where DS will be used in place of the default SS.
- The BX data register not shown is also used as a pointer in the data segment (by default).

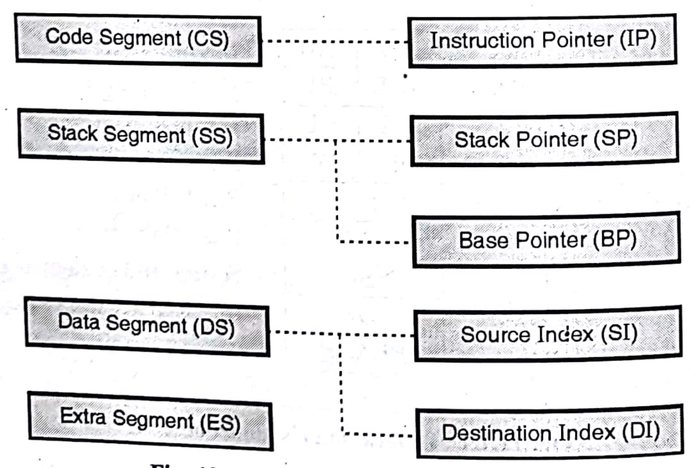


Fig. 12.1.2 : Address registers of 8086

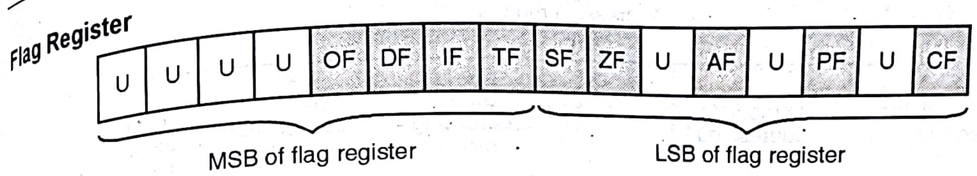


Fig. 12.1.3 : Flag register of 8086

CF = Carry Out
PF = Parity of last operation
ZF = Zero result
TF = Trap Flag – do single instruction
DF = Direction flag
U = It can be 1 or 0. It is undefined
AF = Auxiliary carry (used in BCD arithmetic)
SF = Sign bit from last operation
IF = Interrupt Enable Flag
OF = Overflow (error for signed numbers)

12.2 Addressing Modes

Q. What do you understand by addressing mode ? Explain different addressing modes with suitable examples for 8086 microprocessor.

This will be our first step towards programming. Before we study the instruction set, it is necessary to know how 8086 accesses data operands in different ways.

- When 8086 executes an instruction, it performs specific function on data. The data is normally referred as ***operands***. Operands may be contained in registers, within the instruction itself, in memory or in I/O ports.
- To access these different types of operands, 8086 has to ***address*** memory or I/O. The address of memory and I/Os can be calculated in several different ways, normally referred as ***Addressing modes***. These addressing modes greatly extend the flexibility and convenience of the instruction set.
- Addressing modes (methods) refer to the different methods of addressing (selecting) the operands.
- We can categorize addressing modes of 8086 as follows :

  - Register addressing mode.
  - Memory addressing modes.
  - I/O addressing modes
  - Immediate addressing mode.
  - String addressing mode
  - Implied addressing mode

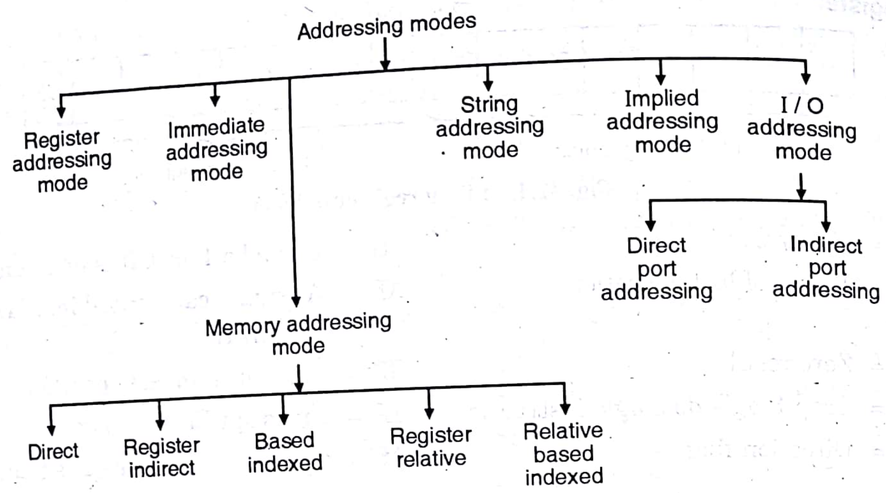


Fig. 12.2.1 : Addressing modes of 8086

12.2.1 Register Addressing Mode

- In this mode of addressing, data is in the register, and instruction specifies the particular register as shown in Fig. 12.2.2.
- This addressing mode is normally preferred because the instructions are compact and fastest executing of all instruction forms. The reason why it is fastest executing is just because, all the registers reside on chip, therefore data transfer is **within** the chip and external bus is not at all required.

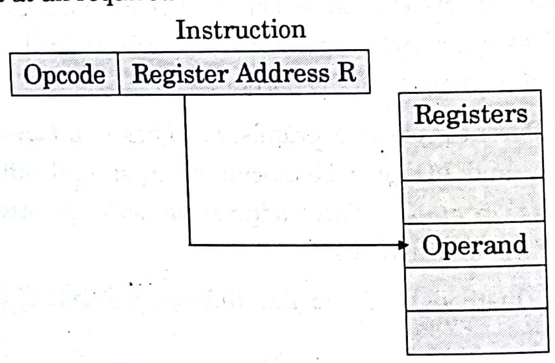


Fig. 12.2.2 : Register addressing

- Registers may be used as source operands, destination operands or both.
- The registers may be 8/16 bit.

  e.g. MOV AX , BX

  This instruction copies the contents of BX register to AX register.

12.2.2 Immediate Operand Addressing Mode

- Immediate operand is nothing but ***constant*** data contained in an ***instruction***. Thus if a source operand is part of instruction instead of register or memory, it is referred as

immediate addressing mode. Fig. 12.2.3 shows format of instruction encoded with immediate operand.

OPCODE	Immediate operand

Fig. 12.2.3 : Instruction encoded with an immediate operand

- The immediate data may be 8 bits (byte) or 16 bits (word) in length.
- Immediate operand can be accessed quickly because they are available directly in instruction queue, like a register operand. Hence, there is no need of external bus and bus cycles to obtain the data.
- Limitation with immediate operands are, they can only be used as source operands and that they are constant values.

  Let's take one example :

  e.g. MOV CL, 02 H MOV CL, 2

  This instruction copies the immediate number 2H in the CL register.

12.2.3 Memory Addressing Modes

- In previous addressing modes, Execution Unit (EU) was having direct access to register and immediate data (operands). But in this addressing mode, memory operands ***must be*** transferred to or from the CPU ***over the BUS***.
- Whenever EU needs to read or write a memory operand, it must pass an offset value to the BIU.
- The BIU adds offset to the shifted contents of segment register, which produces 20 bit physical address and after that it executes the bus cycle(s) needed to access the operand.

Effective Address

- ***The offset for a memory operand is called the operand's effective address or EA.***
- It is an unsigned 16 bit number that expresses the operand's distance in bytes from the beginning of the segment in which it resides.
- In 8086 we have base registers and Index registers. So EU calculates the EA by summing a displacement, the content of base register and the content of index register.

$$EA = \{\text{Base register}\} + \{\text{Index register}\} + \{\text{8 or 16 bit displacement}\}$$

$$= \begin{Bmatrix} BX \\ BP \end{Bmatrix} + \begin{Bmatrix} SI \\ DI \end{Bmatrix} + \begin{Bmatrix} \text{8 or 16 bit} \\ \text{displacement} \end{Bmatrix}$$

- The fact is that, ***any combination of these three components may be present in a given instruction gives rise to the variety of 8086/8088 memory addressing modes.***
- The displacement is an 8 or 16 bit number that is contained in the instruction.
- The displacement generally is derived from the position of the operand name (a variable or label) in the program. It is also possible for a programmer to modify this value or to specify the displacement explicitly.
- A programmer may specify that either BX or BP is to be used as a base register whose content is to be used in the EA computation.
- Similarly either SI or DI may be specified as an index register whereas displacement value is constant.
- The contents of base and index registers may change during execution. This makes it possible for one instruction to access different memory locations as determined by the current values in the base and/or index registers.
- It takes time for EU to calculate a memory operand's EA. In general the more elements in the calculations, the longer it takes.

Finally, once we get EA we can calculate PA (physical address) as,

$$
\begin{aligned}
PA = \ & \text{Segment} && : \ \text{Offset} \\
& \Downarrow && \ \ \Downarrow \\
= \ & \text{Segment register} && : \ \text{EA} \\
= \ & \text{Segment register} && : \ \text{BASE + INDEX + DISPLACEMENT} \\
& \begin{Bmatrix} CS, SS \\ DS, ES \end{Bmatrix} && : \ \begin{Bmatrix} BX \\ BP \end{Bmatrix} + \begin{Bmatrix} SI \\ DI \end{Bmatrix} + \begin{Bmatrix} \text{8 or 16 bit} \\ \text{displacement} \end{Bmatrix}
\end{aligned}
$$

The different memory addressing modes available are :

- Direct memory addressing mode.
- Register indirect addressing mode.
- Register relative addressing mode.
- Based indexed addressing mode.
- Relative based indexed addressing mode.

12.2.3(A) Direct Memory Addressing Mode

- In this mode, the 16-bit effective address EA is taken from the displacement field of the instruction. The physical address is generated by adding this to segment register * 10 H as shown in the Fig. 12.2.4.
- The displacement may be 8/16 bit it is stored in the location which follows the instruction opcode.

EA = 8/16 bit displacement

PA = segment : EA

$$PA = \begin{Bmatrix} CS \\ DS \\ ES \\ SS \end{Bmatrix} : \begin{Bmatrix} \text{Direct Address} \end{Bmatrix}$$

e.g. MOV [1023], AL

The contents of AL are copied to memory location whose offset is [1023].

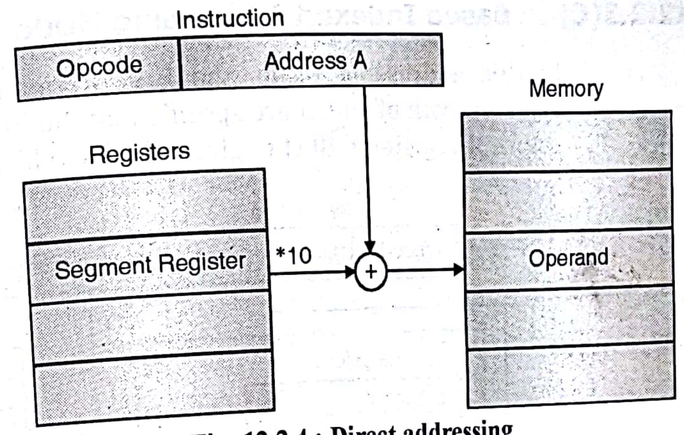


Fig. 12.2.4 : Direct addressing

Note : By default, it is DS segment register for direct addressing mode.

12.2.3(B) Register Indirect Addressing Mode

- In this addressing mode the effective address of the memory may be taken directly from one of the base register or index register, specified by instruction. This address is added with the segment register * 10 H to generate the physical address as shown in Fig. 12.2.5.

$$\therefore \ EA = \begin{Bmatrix} (BX) \\ (BP) \\ (SI) \\ (DI) \end{Bmatrix} \ PA$$

$$= \text{Segment : EA}$$

$$\therefore \ PA = \begin{Bmatrix} CS \\ DS \\ SS \\ ES \end{Bmatrix} : \begin{Bmatrix} BX \\ BP \\ SI \\ DI \end{Bmatrix}$$

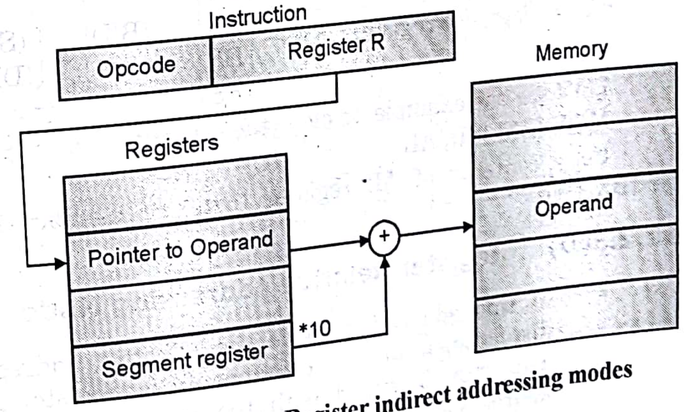


Fig. 12.2.5 : Register indirect addressing modes

- If register is SI, DI, BX then DS is by default segment register.
- If BP is used, then SS is as by default segment register.
- In this addressing mode, again programmer can use segment override prefix so that he can get CS, or ES (as discussed in section 12.8).
  e.g. MOV [SI], AL
  The contents of AL register are copied to memory location whose offset address in SI.
- The advantage of this type of addressing mode is, ***one instruction can operate on many different memory locations if the value in the base or index register is updated appropriately.***

12.2.3(C) Based Indexed Addressing Mode

- In this addressing mode, the EA is sum of a base register and an index register, both of which are specified by the instruction. The sum is added to the segment register * 10 H to give effective address as shown in Fig. 12.2.6.

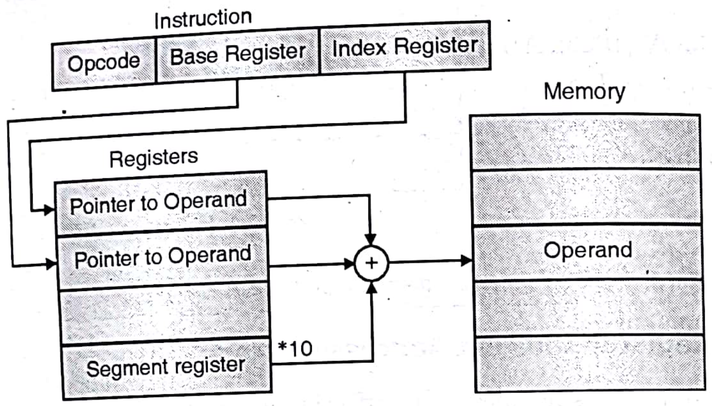


Fig. 12.2.6 : Based indexed addressing mode

$$\therefore \text{ EA} = \{\text{Base register}\} + \{\text{Index register}\} = \begin{Bmatrix}(BX)\\(BP)\end{Bmatrix} + \begin{Bmatrix}(SI)\\(DI)\end{Bmatrix}$$

$$\text{PA} = \text{Segment register : EA} = \begin{Bmatrix}CS\\SS\\DS\\ES\end{Bmatrix} : \begin{Bmatrix}(BX)\\(BP)\end{Bmatrix} + \begin{Bmatrix}(SI)\\(DI)\end{Bmatrix}$$

Let's see one example, to clear the concept.

MOV [BX] [SI], AL

Copy contents of AL register (byte) to memory location offset is in [BX] [SI] i.e. [BX + SI].

12.2.3(D) Register Relative Addressing Mode

- In this addressing mode the effective address is sum of an 8 or 16 bit displacement and the contents of base register or an index register are added. This sum is added with the segment register *10H to generate effective addresss as shown in Fig. 12.2.7.

$$\therefore \quad \text{EA} = \begin{Bmatrix}(BX)\\(BP)\\(SI)\\(DI)\end{Bmatrix} + \begin{Bmatrix}\text{8 bit displacement}\\\text{(sign extended)}\\\text{16 bit displacement}\end{Bmatrix}$$

$$\therefore \quad \text{PA} = \text{Segment : EA}$$

$$= \begin{Bmatrix}CS\\ES\\DS\\SS\end{Bmatrix} : \begin{Bmatrix}(BX)\\(BP)\\(SI)\\(DI)\end{Bmatrix} + \begin{Bmatrix}\text{8/16 bit}\\\text{offset}\end{Bmatrix}$$

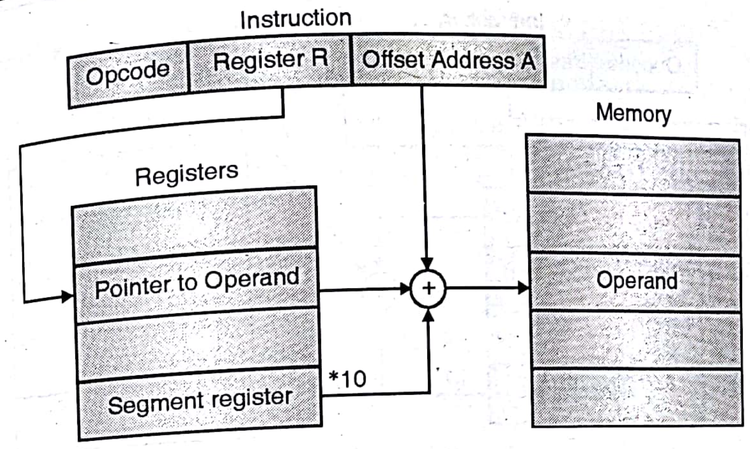


Fig. 12.2.7 : Register relative addressing mode

e.g. MOV [BX + 1100], AL

Copy contents of AL register to a memory location, whose offset is given by BX plus displacement, from the beginning of the current segment selected by the user.

	23140 H	→	Contents of DS
+	1023 H	→	Offset of BX
+	1100 H	→	Offset
	25263 H	→	Physical address

The contents of Reg. AL will be transferred at the location whose address is given by 25263 H.

12.2.3(E) Relative Based Indexed Addressing

- This addressing mode generates an effective address that is the sum of a base register, an index register and a displacement. This is sum is added to segment register *10H to generate effective address as shown in Fig. 12.2.8.

$$\therefore \text{ EA} = \{\text{Base register}\} + \{\text{Index register}\} + \begin{Bmatrix}\text{8 bit displacement}\\\text{(sign extended)}\\\text{16 bit displacement}\end{Bmatrix}$$

$$= \begin{Bmatrix}(BX)\\(BP)\end{Bmatrix} + \begin{Bmatrix}(SI)\\(DI)\end{Bmatrix} + \begin{Bmatrix}\text{8/16 bit}\\\text{displacement}\end{Bmatrix}$$

$$\text{PA} = \text{Segment register : EA} = \begin{Bmatrix}CS\\SS\\DS\\ES\end{Bmatrix} : \begin{Bmatrix}(BX)\\(BP)\end{Bmatrix} + \begin{Bmatrix}(SI)\\(DI)\end{Bmatrix} + \begin{Bmatrix}\text{8/16 bit}\\\text{displacement}\end{Bmatrix}$$

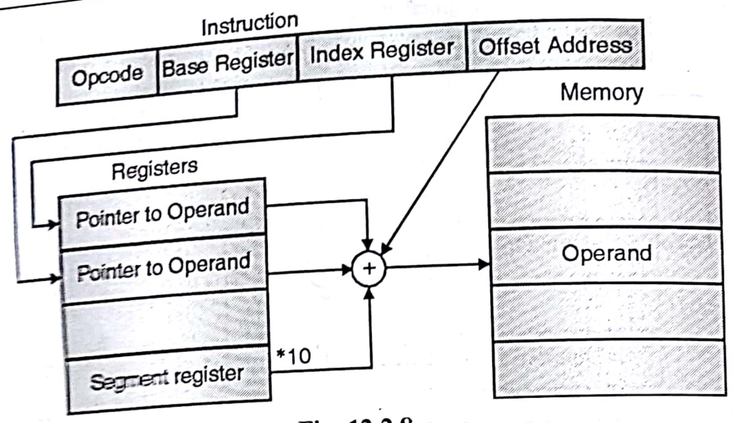


Fig. 12.2.8

Let's take one example :

MOV CX, [BX + SI + 0400]

Move the contents of the memory location offset by the value of EA, in SI plus BX plus displacement from the beginning of the current segment selected by user, to the CX register.

- In this case, word is moved not only a byte.

Generation of EA	Generation of PA
1000H → [BX]	23140H → [DS]
+ 1100H → [SI]	+ 2500H → EA
+ 0400H → **Displacement**	**25640H** → PA
2500H → EA	

The contents of location 25640H will be transferred to the CL register and the contents of location 25641H will be transferred to the CH register.

Ex. 12.2.1 : IF (BX) = 0158H Displacement = 1B57H
(DI) = 10A5H (DS) = 2100H
and DS is used as segment register, then calculate EA and PA produced for all various addressing modes.

Soln. :

1) **Register addressing mode :** No EA (data specified in register).
2) **Immediate addressing mode :** No EA (data specified in instruction).
3) **Direct addressing mode**

EA = Displacement = 1B57H
∴ PA = Segment : offset = DA : EA = DS : displacement
= 2100H : 1B57H = 21000H + 1B57H = 22B57H

4) **Register indirect, assuming BX**

EA = (BX) = 0158H
∴ PA = Segment : offset = PA : EA = DS : (BX)
= 2100H : 0158H = 21000H + 0158H = 21158H

5) **Register indirect, assuming DI**

EA = (DI) = 10A5H
PA = Segment : offset = PA : EA = DS : (DI)
= 2100H : 10A5H = 21000H + 10A5H = 220A5H

6) **Register relative, assuming BX**

EA = (BX) + displacement = 0158H + 1B57H = 1CAFH
∴ PA = Segment : offset = PA : EA = DS : ((BX) + displacement)
= 2100H : 1CAFH = 21000H + 1CAFH = 22CAFH

7) **Register relative, assuming DI**

EA = (DI) + displacement = 10A5H + 1B57H = 2BFCH.
∴ PA = Segment : offset = PA : EA = ES : ((DI) + displacement)
= 2100H : 2BFCH = 21000H + 2BFCH = 23BFCH

8) **Based indexed, assuming register BX and DI**

EA = (DI) + (BX) = 10A5H + 0158H = 11FDH
∴ PA = Segment : offset = PA : EA = DS : [(DI) + (BX)]
= 2100H : 11FDH = 21000H + 11FDH = 221FDH.

9) **Relative based indexed addressing, assuming BX and DI**

EA = (DI) + (BX) + displacement
= 10A5H + 0158H + 1B57H = 2D54H
∴ PA = Segment : offset = PA : EA
= DS : [(DI) + (BX) + displacement] = 2100H : 2D54H
= 21000H + 2D54H = 23D54H

12.2.4 String Addressing Mode

- String instructions do not use the normal memory addressing modes to access their operands.
- When a string instruction is executed, SI is assumed to point to the first byte or word of the source string and by default DS is assumed segment register.
- DI will point to the first byte or word of the destination string and ES is assumed to be by default segment register.
- In a repeated string operation, the CPU automatically adjust the SI and DI to obtain subsequent bytes or word.
- The automatic adjustment is done with the help of DF (Direction Flag) in flag register, and achieved by MOD and R/M bits in instruction format.

12.2.5 I/O Addressing Mode in 8086

This addressing mode is basically used for IOs. Under IO mapping we have :

- Memory mapped I/O.
- I/O mapped I/O.

1) **Memory mapped I/O**

- If an I/O port is memory mapped, any of the memory operand addressing modes may be used to access the port.
- For example, a group of terminals can be accessed as an ***array***. String instructions also can be used to transfer data to memory mapped I/O ports with in appropriate hardware interface.

2) **I/O mapped I/O**

- If I/Os are mapped in ***I/O map I/O,*** then 8086 supports two different addressing modes :
  1) Direct port addressing. 2) Indirect port addressing.
- In direct port addressing, the port number is an 8 bit ***immediate*** operand. This allows fixed access to ports numbered 0-255.
- Indirect port addressing is similar to register indirect addressing of memory operands. The port number is taken forms register DX and can range from 0000H to FFFFH (0 to 65535 decimal). By previously adjusting the contents of register DX, one instruction can access any port in the IO space.
- A group of adjacent ports can be accessed using a simple software loop that adjusts the value in DX.

12.2.6 Implied Addressing Mode

The instructions which do not have operands come under implied addressing mode.

e.g. **XLAT**
CMA
STC
STD

These instructions do not have operands.

12.3 Instruction Encoding Format

- In 8086 we have variety in instructions, therefore all instructions will not be of same size.
- The instructions vary from 1 to 6 bytes in length.
- The obvious question is, ***what these bytes will contain and on what parameter the length of the instruction bytes is decided ?*** The length of instruction bytes is dependent upon addressing mode used by programmer i.e. immediate, register, register relative, based indexed, relative based indexed and so on.

Basically instruction bytes will contain information of :

1) OPCODE

2) Addressing mode designations :
   a) 2 byte Effective Address.
   b) 1 or 2 byte displacement.
   c) 1 or 2 byte immediate operand.

To understand more clearly, let's observe Fig. 12.3.1 carefully.

1) Normally first byte is OPCODE byte.
2) 2nd byte normally specifies addressing mode. (Remember MOD and R/M). Sometime it may also contain OPCODE part.
3) After OPCODE and addressing mode bytes, we have following different cases :
   a) No additional bytes (Figs. 12.3.1(a), (b), (c) and (d)).
   b) A 2 byte EA (for direct addressing mode (Fig. 12.3.1(e)).
   c) A 1 or 2 byte immediate operand (Fig. 12.3.1(f)).
   d) A 1 or 2 byte displacement followed by 1 or 2 byte immediate operand (Fig. 12.3.1(g)).

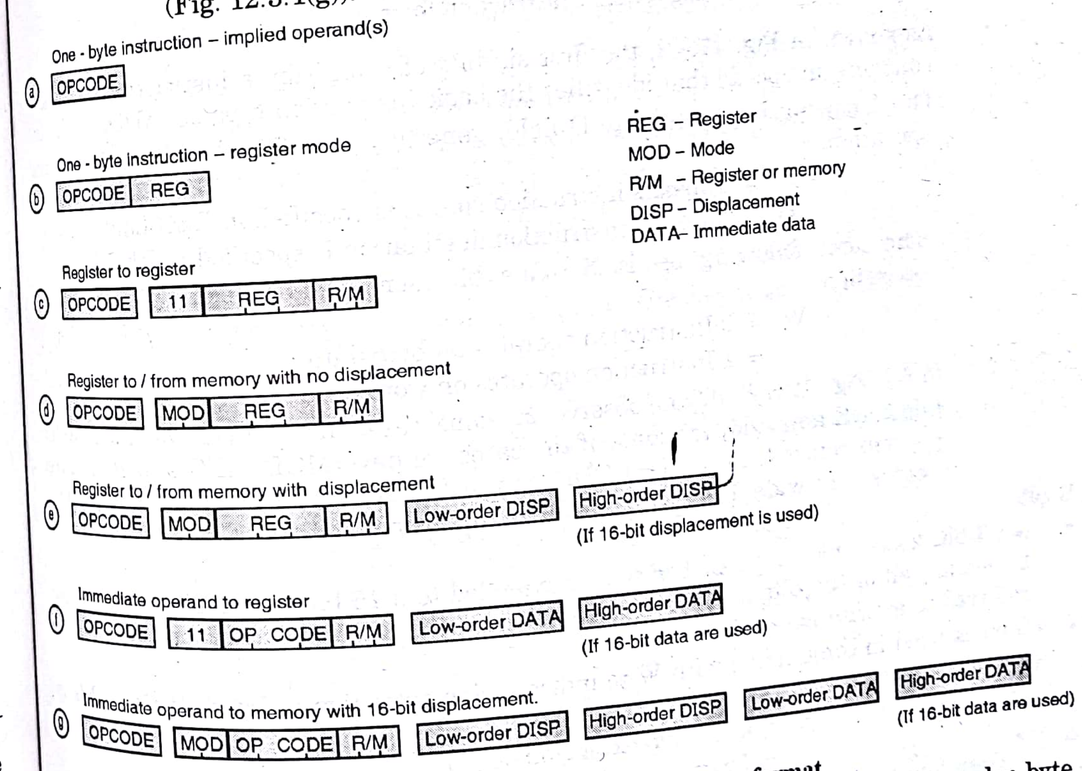


Fig. 12.3.1: Summary of 8086 instruction format

4) If a displacement or immediate operand is 2 bytes long, the low order byte always appears first, this is Intel standard (same was followed by 8085 also).

- To remember these formats, I will give you only a single format, from that we get these different formats refer Fig. 12.3.2.

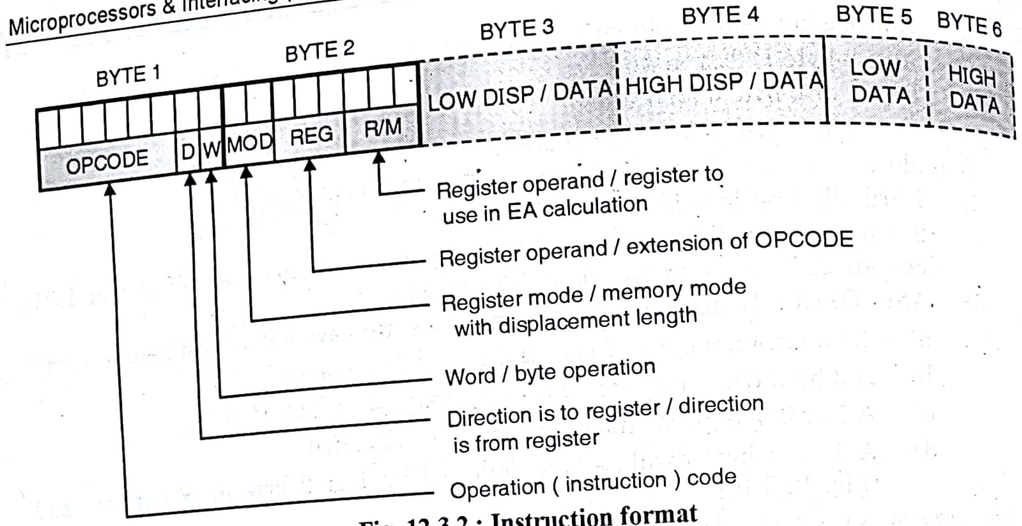


Fig. 12.3.2 : Instruction format

- As shown in Fig. 12.3.2, the first six bits of a multibyte instruction generally contains an opcode that identifies the basic instruction type i.e. ADD, XOR etc.
- The following bit, called the D field, generally specifies the ***direction*** of the operation.

  D = 1 means instruction source is specified in REG field.
  D = 0 means instruction destination is specified in REG field.

  The next following bit is *W*. This bit identifies between byte and word operation.

  W = 0 Instruction operates on byte data.
  = 1 Instruction operates on word data.
- Refer Fig. 12.3.2, if you observe in some case, in 2nd byte we have MOD, OPCODE and R/M, for some of the cases we have MOD, REG and R/M. First we will concentrate on OPCODE bits in 2nd byte of instruction format. This field is 3 bit wide. Under that we have three single bit fields, S, V and Z.

S bit

- An 8 bit, 2's complement number can be extended to a 16 bit 2's complement number by letting all of the bits in high order byte equal the MSB in low order byte. This is referred to as ***sign extension***.
- S bit is used in conjunction with W to indicate sign extension of ***Immediate fields*** in arithmetic instructions.

  S = 0 No sign extension
  = 1 Sign extended 8 bit immediate data to 16 bits if W = 1.

  Therefore for 8 bit operation : S = W = 0
  16 bit operation with a 16 bit immediate operand : S = 0, W = 1
  16 bit operation with a sign extended 8 bit immediate operand : S = W = 1

V bit

- Used by shift and rotate, to determine single and variable - bit shifts and rotate.

  V = 0 shift/rotate count is one
  = 1 shift/rotate count is specified in CL register.

Z bit

- This bit is used as a compare bit with zero flag in conditional repeat (REP) and loop instructions.

  Z = 0 Repeat/loop while zero flag is clear.
  = 1 Repeat/loop while zero flag is set.

Refer Table 12.3.1, it summarizes all 5 bits used in OPCODE field.

Table 12.3.1 : 5 bits used in OPCODE field

Field	Value	Function
S	0	No sign extension
	1	Sign extend 8-bit immediate data to 16 bits if W = 1
W	0	Instruction operates on byte data
	1	Instruction operates on word data
D	0	Instruction source is specified in REG field
	1	Instruction destination is specified in REG field.
V	0	Shift/rotate count is one
	1	Shift/rotate count is specified in CL register
Z	0	Repeat/loop while zero flag is clear
	1	Repeat/loop while zero flag is set

- Now concentrate on MOD, R/M and REG field in 2nd byte of instruction format. The second byte of the instruction usually identifies the instruction's operands.

MOD

- The mode (MOD) field indicates whether one of the operands is in memory or whether both operands are register. Table 12.3.2 shows MOD field encoding, this field is of size 2 bits.

Table 12.3.2 : MOD field ENCODING

CODE	EXPLANATION
0 0	Memory mode, no displacement follows *
0 1	Memory mode, 8 bit displacement follows
1 0	Memory mode, 16 bit displacement follows
1 1	Register mode (No displacement)

- * Except when R/M = 110, then 16 bit displacement follows. As seen MOD is basically concerned with displacement i.e. 8 bit or 16 bit or no displacement.

REG

- The Register (REG) field identifies a register that is one of the instruction operands. REG field depends upon W bit. Table 12.3.3 shows the selection of register(s) depending upon W bit.

Table 12.3.3 : REG (Register) field encoding

REG	W = 0	W = 1
0 0 0	AL	AX
0 0 1	CL	CX
0 1 0	DL	DX
0 1 1	BL	BX
1 0 0	AH	SP
1 0 1	CH	BP
1 1 0	DH	SI
1 1 1	BH	DI

- When W = 0, all 8 bit registers are selected, whereas for W = 1 all 16 bit registers are selected. Thus in a number of instructions and mainly in immediate to memory variety, REG is used as an extension of the OPCODE to identify the type of operation i.e. 8 bit or 16 bit.
- **R/M : Register or memory :** This field is of 3 bits. The meaning of R/M bits changes depending upon mode (MOD) field.

At this stage we have general clear idea about these three fields, now, we will take some cases.

Case I : Register to register transfer :

- In this operation, data movement is within the register either 8 bit or 16 bit. As mentioned in this operation REG field identifies ONE of the instruction operands. ***What about another instruction operand ?*** It is specified by R/M, W and MOD bits. Refer Table 12.3.4.

Table 12.3.4 : R/M field encoding when MOD = 11 (binary)

MOD = 11 (binary)		
R/M	W = 0	W = 1
0 0 0	AL	AX
0 0 1	CL	CX
0 1 0	DL	DX
0 1 1	BL	BX
1 0 0	AH	SP
1 0 1	CH	BP
1 1 0	DH	SI
1 1 1	BH	DI

- You will find that Table 12.3.3 matches with Table 12.3.4. Secondly when W = 0, you can select ONLY 8 bit source and destination operand. When W = 1, you can select ONLY 16 bit source and destination operand.

- Following are the example instructions those are not correct or valid :

MOV AH, CX ; Moving 16 bit to 8 bit (wrong)

MOV DX, BL ; Moving 8 bit to 16 bit (wrong)

Thus any such combination with other register(s) is INVALID :

Case II : Memory MODE (8 bit/16 bit or No displacement).

- When MOD selects memory mode (MOD = 00 or 01 or 10), then data transfer is register to/from memory. In that case R/M field indicates how the effective address of the memory operand is to be calculated. Now refer Table 12.3.5, it depicts EA calculation.

Table 12.3.5 : R/M field encoding when MOD = 00/01/10

MOD = 11			EFFECTIVE ADDRESS CALCULATION			
R/M	W = 0	W = 1	R/M	MOD = 00	MOD = 01	MOD = 10
000	AL	AX	000	(BX) + (SI)	(BX) + (SI) + D8	(BX) + (SI) + D16
001	CL	CX	001	(BX) + (DI),	(BX) + (DI) + D8	(BX) + (DI) + D16
010	DL	DX	010	(BP) + (SI)	(BP) + (SI) + D8	(BP) + (SI) + D16
011	BL	BX	011	(BP) + (DI)	(BP) + (DI) + D8	(BP) + (DI) + D16
100	AH	SP	100	(SI)	(SI) + D8	(SI) + D16
101	CH	BP	101	(DI)	(DI) + D8	(DI) + D16
110	DH	SI	110	DIRECT ADDRESS	(BP) + D8	(BP) + D16
111	BH	DI	111	(BX)	(BX) + D8	(BX) + D16

D8 = 8 bit displacement D16 = 16 bit displacement

REG field in this case, as usual identify the register that is one of the instruction operand.

12.4 Segment Override Prefix

Q. What do you mean by segment override prefix?

- Normally for each offset, segment is fixed. But using ***segment override prefix*** one can change segment registers.

- For addressing modes DS may be overridden by CS, SS, or ES; and when BP is used, SS may be overridden by CS, DS, or ES. Specific cases that cannot involve overrides are as follows :
  1) The CS register is always used as the segment register when computing the address of the next instruction to be executed.
  2) For stack pointer SP, SS is the segment register.
  3) For string operation ES is by default segment register for destination operand.

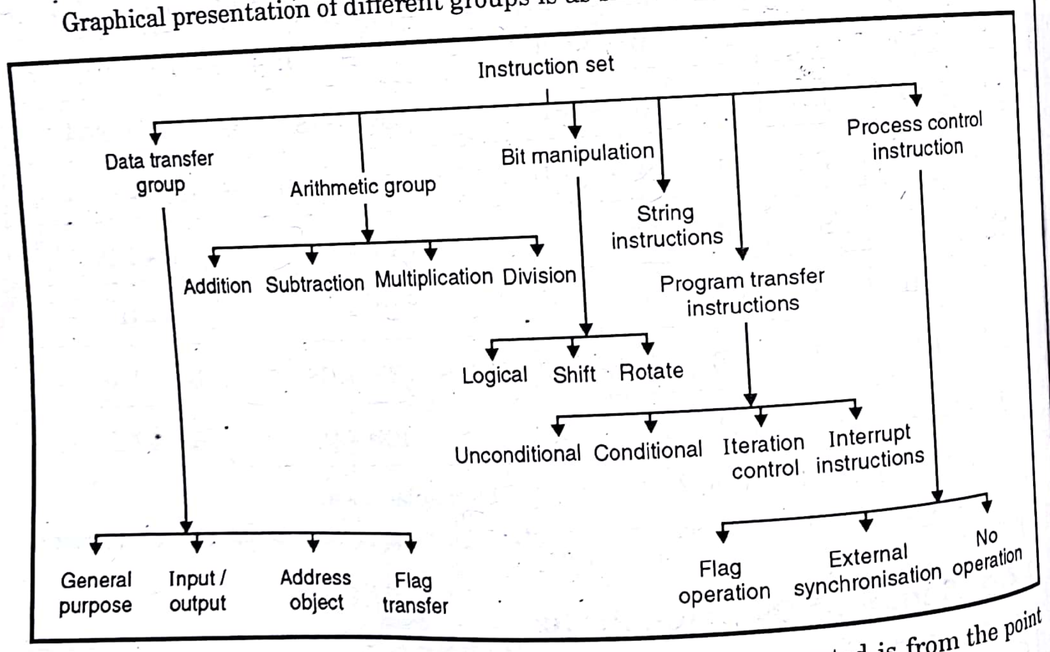
12.5 Instruction Set of 8086

The instruction set of 8086/8088 is divided into number of groups, of functionally related instructions.

Different groups are :

- Data transfer group.
- Bit manipulation group.
- Program transfer instruction group.
- Arithmetic group.
- String instruction group.
- Process control instruction group.

Graphical presentation of different groups is as shown.



Now we will start with instruction set. The information presented is from the point of view of utility to the assembly language programmer. The information given is :

1) *Mnemonic (Syntax of the instruction)*
2) *Algorithm*
3) *Operation of the instruction*
4) *Examples..*

- While giving you above information some typical symbols/labels are used. I feel that you should know the significance and meaning of those labels.

12.6 Data Transfer Group

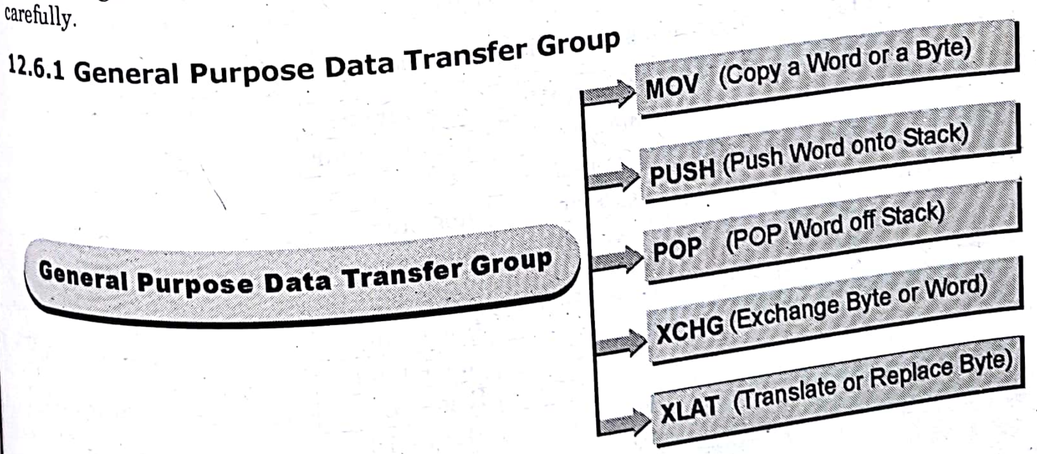
The 14 data transfer instructions are listed as follows :

Table 12.6.1 : Data transfer instructions

General Purpose		Address Object	
MOV	Move byte or word	LEA	Load effective address
PUSH	Push word onto stack	LDS	Load pointer using DS
POP	Pop word off stack	LES	Load pointer using ES
XCHG	Exchange byte or word		
XLAT	Translate byte		
Input/Output		**Flag Transfer**	
IN	Input byte or word	LAHF	Load AH register from flags
OUT	Output byte or word	SAHF	Store AH register in flags
		PUSHF	Push flags onto stack
		POPF	Pop flags off stack

These instructions move single bytes and words between memory and register as well as between register AL or AX and I/O ports. The stack manipulation instructions are included in the group as are instructions for transferring flag contents and for loading segment registers. Now let's start with one by one subgroup and study each instruction carefully.

12.6.1 General Purpose Data Transfer Group



1. MOV – Copy a Word or a Byte

Mnemonic MOV destination, source
MOV operand1, operand2

Destination ← Source

Operand 1 ← Operand 2

Flags No flags are affected.

Algorithm Destination = Source or Operand 1 = Operand 2

Addr. Mode Register addressing mode

Operation

- The MOV instruction copies a word or a byte of data from a fixed/specified source to a fixed/specified destination.

Examples

1. MOV [SI], AL
   i.e. MOV [SI] , AL — This instruction copies the contents of the AL register to memory location whose offset is stored in SI register.
2. MOV AX, Temp_Result — The contents of memory location Temp_result will be transferred/copied to the AL register. Then the IP will increment by 1 and contents of location after Temp_Result will be copied to the AH register.
3. MOV AX, BX — This instruction copies the contents of BX register to AX register. The LSB of BX i.e. BL is copied to AL and MSB of BX i.e. BH is copied to AH.
4. MOV COUNT [DI],2DH
   i.e. MOV COUNT [DI],2DH — This instruction copies immediate number 2DH to the required memory location. EA of the memory location is the sum of displacement COUNT and the contents of DI (EA = COUNT + DI)

Following table contains valid source and destination operands.

Sr. No.	Destination	Source
1.	Memory	Accumulator
2.	Accumulator	Memory
3.	Register	Register
4.	Register	Memory
5.	Memory	Register.
6.	Register	Immediate
7.	Memory	Immediate
8.	Seg – Reg	Reg – 16
9.	Seg – Reg	Mem – 16
10.	Reg – 16	Seg – Reg
11.	Memory	Seg – Reg

Following rules are observed while executing the instruction

The Source & Destination in an Instruction both **CANNOT** be Memory Locations	Incorrect ⇨	MOV [1100], [1200] (Memory Location 1, Memory Location 2)
The Destination in an Instruction **CANNOT** be Immediate Number	Incorrect ⇨	MOV 592F H, BX (Immediate number)
The Destination in an Instruction **CANNOT** be Code Segment Register CS incorrect	Incorrect ⇨	MOV 592F H, CS (Immediate number)
The Source & Destination must both be of a type **BYTE**, or they must be of a type **WORD** Such a data transfer is not possible because BL is 8 Bit & AX is 16 Bit	Incorrect ⇨	MOV AX, BL (16 bit (WORD), 8 bit (BYTE))
It **CANNOT** copy value of one segment Register to another segment Register (One should copy to general register first)	Incorrect ⇨	MOV DS, CS (Data Seg., Code Seg.)
It **CANNOT** copy immediate value to segment Register.	Incorrect ⇨	MOV CS, 5487 H (Code Seg., Immediate number)
It **CANNOT** set the value of CS & IP Registers		

Fig. 12.6.1

2. PUSH - Push Word onto Stack

Mnemonic PUSH Source

Flags No flags are affected.

Algorithm SP = SP – 2
SS : [SP] (Top of the Stack) = Operand

Addr. Mode Register addressing mode

Operation SP → SP – 2 SS → data from specified source

- This instruction decrements the stack pointer by 2 and copies a word from a specified source to the location in the stack segment where stack pointer points.
- The source of operand (16 bit data to be stored on stack) can be a general purpose register, flag register, segment register or memory.

- The stack segment register and stack pointer must be initialised before using this instruction.
- PUSH can be used to save data on the stack so that it will not be destroyed by a execution of successive instructions.

Example PUSH AX

Now we will see detailed analysis of what exactly happens when instruction PUSH AX executes.

PUSH AX

SP → SP – 2

SS → Copy word from AX register to location in Stack segment (SS) where SP points.

3. POP – Pop Word off Stack

Mnemonic POP Destination OR POP Operand **Flags** No flags are affected

Algorithm Operand = SS : [SP] (top of stack) SP = SP + 2

Addr. Mode Register Addressing mode

Operation SS → Data copied to destination or operand SS → SS : [SP + 2]

- This instruction copies a word (two successive memory location contents) from stack segment in memory to a destination specified in the instruction.
- The destination can be a general purpose register, flag register, a segment register or a memory location.
- The data in the stack pointer is not changed.
- After the word is copied to specified destination the stack pointer is automatically incremented by 2 to point to next word on stack.

Example POP AX

This instruction copies a word (two successive memory location contents) from stack segment in memory to the AX register.

4. XCHG – Exchange byte or word

Mnemonic XCHG destination, source. **Flags** No flags are affected

Algorithm destination = source

Addr. Mode Implied addressing mode

Operation destination ↔ source

- This instruction exchanges the contents of a register with contents of another register or the contents of a register with contents of memory location.
- The register can be 8/16 bit.
- XCHG cannot directly exchange the contents of two memory locations. e.g. XCHG [1234],[5068] ; This transfer is not possible
- The source and destination must both be words or they both must be bytes.

Example XCHG AX, BX i.e. AX ↔ BX

- This instruction will exchange the word in AX register with the word in BX register.
- Contents of AL ↔ Contents of BL, Contents of AH ↔ Contents of BH

5. XLAT – Translate or Replace Byte

Mnemonic XLAT/XLATB [B indicates byte operation] (meaning of XLAT and XLATB is same, either XLAT or XLATB can be written) **Flags** No flags are affected

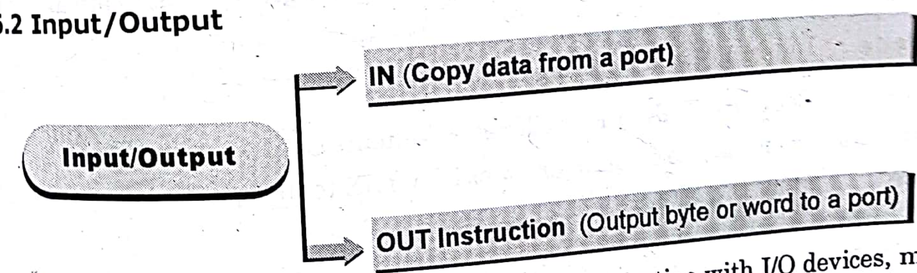
Algorithm AL = DS : [BX + unsigned AL]

Addr. Mode Implied Addressing mode

Operation AL ← DS : [BX + AL]

- This instruction replaces a byte in AL register with a byte from a look up table in the memory. i.e. it copies the value of memory byte at location DS : [BX + unsigned AL] to the AL register.
- Here contents of AL before execution acts as index to the desired location in look up table.
- This instruction is used to translate a byte from one code to another code.
- Mostly this concept is used to convert BCD to seven segment code or ASCII to EDCBIC code conversion

12.6.2 Input/Output



These instructions are basically related to communication with I/O devices, mapped in I/O map.

1. IN – Copy data from a port

Mnemonic IN accumulator, port address. **Flags** No flags are affected

Algorithm —

Addr. Mode Direct port addressing mode

Operation AX ← Contents of port **or** AL ← contents of port.

This instruction will copy data from a port whose address is given in the instruction to AX register or AL register. The data can be either 8 bit or 16 bit.

Example IN AL, C8H.

This instruction will copy the contents of port whose address is C8H to the AL register.

- The IN instruction has two possible formats
  - Fixed port
  - Variable port.
- For the **Fixed port**, the port address is specified directly in the instruction. The port numbers are from 00 to FF i.e. 8 bit address is directly specified. Thus addressing mode is Direct port addressing The above example is an example of direct port addressing.
- For the **Variable port** IN instruction, the port address is loaded into the DX register before the IN instruction. Since DX is a 16 bit register, the port address can be any number between 0000H and FFFFH. Therefore it is possible to address up to 65, 536 ports in this mode.

  e.g. MOV DX, 0FF0H ; Initialize DX to point to port.

  IN AL, DX; Input an 8 bit data from port 0FF0 to AL i.e. contents of port 0FF0 are copied to the AL register. The addressing mode is Indirect port addressing mode .

Note : This instruction is basically related to communication with I/O devices which are mapped in the I/O mapped I/O .

2. OUT Instruction – Output byte or word to a port

Mnemonic OUT port address, Accumulator. **Flags** No flags are affected

Algorithm —

Addr. Mode Indirect port addressing mode

Operation Contents of AX → Port address / Contents of AL → port address.

This Instruction copies a byte from AL or a word from AX to the specified port.

Example MOV DX, FFF8H
OUT DX, AL

Load desired port address in DX.

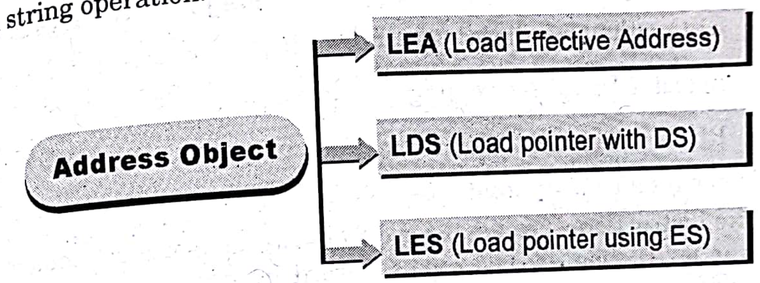
Copies contents of AL to given port address in register DX .

- It has two possible forms.
  - Fixed port
  - Variable port.
- In the **fixed port** form, the 8 bit port address is specified directly in the instruction with this form, any one of 256 possible ports can be addressed.

  e.g. OUT 3B H, AL ; copies the contents of AL to port 3B H.
- The addressing mode is Direct port addressing mode.
- In the **variable port** form, the contents of AL or AX will be copied to the port at an address contained in DX. The DX register should be loaded with the desired port address. The above example in table is an example of indirect port addressing.

12.6.3 Address Object

These instructions manipulate the addresses of the variables, rather than the contents or values of the variables. These are mainly used for list processing, based variables and string operation.



1. LEA – Load Effective Address

Mnemonic LEA register , source. **Flags** No flags are affected

Algorithm REG = Address of memory (offset)

Addr. Mode Register Direct Addressing

Operation REG ← source.

- This instruction determines the offset of variables or memory location named as source and puts the offset in the indicated 16 bit register.
- Generally this instruction is replaced by MOV when assembling is possible.
- Normally the offset is loaded into index register or base pointer registers such as SI, DI, BX, BP.

Example LEA AX, COUNT

This instruction loads AX with the offset of COUNT in DS.

2. LDS – Load pointer with DS

(Load Register and DS with words from memory)

Mnemonic LDS register , source. **Flags** No flags are affected

Algorithm REG = First word, DS = Second word

Addr. Mode Register Direct Addressing.

Operation REG ← Source, DS ← (Source + 2)

- The source is always a memory location. DS is used as a segment register for memory.
- This instruction is a 2 byte instruction. It copies a word from two memory locations into register specified in the instruction. It then copies a word from the next two memory locations into the DS register.

3. LES – Load pointer using ES

(Load register and ES with words from the memory)

Mnemonic LES register , source. **Flags** No flags are affected

Algorithm REG = First word, ES = Second word

Addr. Mode Register Direct Addressing

Operation REG ← Source, ES ← (source + 2)

- The source is always a memory location.
- This is a 2 byte instruction. It copies a word from two memory locations into register specified in the instruction. It then copies a word from next two memory locations into the ES register.

12.6.4 Flag Instructions (Flag Transfer)

These instructions are related to movement of flag register to/from a register and memory.

Flag Instructions (Flag Transfer)
- LAHF (Load AH register from flags)
- SAHF (Store AH register in flags)
- PUSHF (Push flags onto stack)
- POPF (Pop flags off stack)

1. LAHF – Load AH register from flags

(Copy lower byte of flag register to AH)

Mnemonic LAHF **Flags** No flags are affected.

Algorithm AH = flag register's lower byte

Addr. Mode Implied Addressing mode

Operation AH ← Lower byte of flag register
The lower byte of 8086 flag register is copied to the AH register

2. SAHF – Store AH register in flags

(Copy contents of AH to lower byte of flag register)

Mnemonic SAHF **Flags** All the flags are changed.

Algorithm AH = flag register

Addr. Mode Implied Addressing mode

Operation AH → Lower byte of flag register

- This instruction copies the contents of AH register to the lower byte of flag register.
- It is included for 8085 compatibility.
- The OF, DF, IF and TF are not affected.

3. PUSHF – Push flags onto stack

(PUSH flag register on the stack)

Mnemonic PUSHF **Flags** No flags are changed.

Algorithm SP = SP – 2
SS : [SP] (top of stack) = operand

Addr. Mode Register Addressing mode

Operation SP → SP – 2 SS → data from flag register.

- This instruction decrements the stack pointer by 2 and copies word in the flag register to the memory location pointed by stack pointer.
- The stack segment register is not affected.

4. POPF – Pop flags off stack

Mnemonic POPF **Flags** All flags are affected.

Algorithm SS = data to flag register SP = SP + 2

Addr. Mode Register Addressing mode

Operation SS : [SP] → Copy data to flag register SP = SP + 2

- This instruction copies a word from the two memory locations at the top of the stack to flag register and increments the stack pointer by 2.
- The stack segment register and word on the stack are not affected.

12.7 Arithmetic Instructions

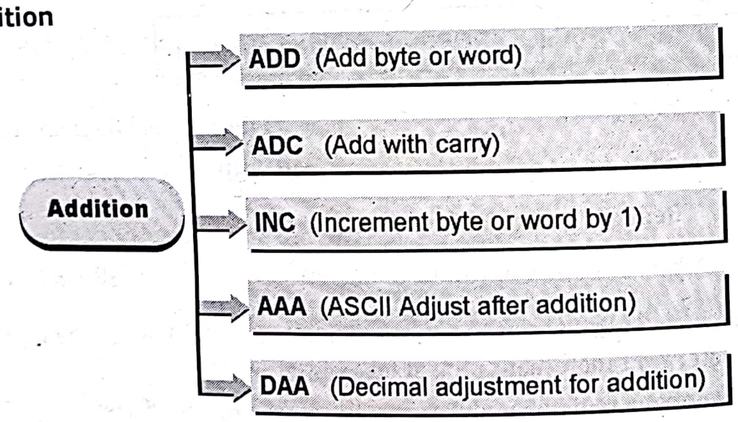
- Table 12.7.1 lists out arithmetic instructions available in 8086 microprocessor, under these subgroups.

Table 12.7.1 : Arithmetic instructions

Addition		Multiplication	
ADD	Add byte or word	MUL	Multiply byte or word unsigned
ADC	Add byte or word with carry	IMUL	Multiply byte or word signed
INC	Increment byte or word by 1	AAM	Integer multiply byte or word
AAA	ASCII adjust for addition		ASCII adjust for multiply
DAA	Decimal adjustment for addition		
Subtraction		**Division**	
SUB	Subtract byte or word	DIV	Divide byte or word unsigned
SBB	Subtract byte or word with borrow	IDIV	Integer divide byte or word
DEC	Decrement byte or word by 1	AAD	ASCII adjust for division
NEG	Negate byte or word (2's complement)	CBW	Convert byte to word
CMP	Compare byte or word	CWD	Convert word to doubleword
AAS	ASCII adjust for subtraction		
DAS	Decimal adjust for subtraction		

We will study the subgroups one by one.

12.7.1 Addition



1. ADD – Add byte or word

Mnemonic : ADD destination, source

Algorithm : destination = destination + source

Operation : (Destination) ← (destination) + (source)

This instruction adds a number from source to number from destination and puts the result to specified destination.

Sr. No.	Destination	Source
1.	Register	Register
2.	Register	Memory
3.	Memory	Register
4.	Register	Immediate
5.	Memory	Immediate
6.	Accumulator	Immediate

Note : (i) Both the operands i.e. source and destination, cannot be memory locations.
e.g. ADD [1234], [5678] ⇒ Incorrect
(ii) Source and destination both have to be of same type i.e. byte or word.

2. ADC – Add with carry

Mnemonic ADC destination, source.

Flags All flags are affected.

Algorithm destination = destination + source + CY

Addr. Mode Register Immediate addressing mode.

Operation destination ← destination + source + CY

- This instructions adds destination operand contents, source operand contents and carry flag and answer is stored back to destination operand.
- The source and destination can be 8/16 bit register or memory location . The source and destination can also be a 8/16 bit register or memory location and immediate data .
- The segment registers cannot be used . The memory uses DS as segment register .
- The addition of two memory locations along with carry is not possible .
- It is easy to perform multiple- precision arithmetic by using ADC instruction.

Examples

1. ADC AL, BL
   i.e. AL ← AL + BL + CY
   - This instruction adds the contents of AL register with BL register and contents of CY.
2. ADD BL, CL
   BL ← BL + CL
   - This instruction adds the data in register CL and BL. The result is stored in BL register. It can be 8 bit/ 16 bit instruction
3. ADD AX , [2048]
   i.e. AX ← AX + contents of memory location
   - This instruction adds the data at memory locations whose offset in DS are [2048] and [2049] with data in AX register. The result is stored in the AX register.

4. ADD [2048], AX
   - This instruction adds the data at memory locations whose offset in DS are [2048] and [2049] with data in AL register and AH register.

5. ADD AL, 74 H
   i.e. AL ← AL + 74
   - This instruction adds the immediate number 74 H with the contents of AL register . The result is stored in AL.

3. INC – Increment byte or word by 1

Mnemonic INC Destination **Flags** All the flags except carry flag are affected.

Algorithm destination = destination + 1

Addr. Mode Implied addressing mode

Operation destination ← destination + 1 .

- This instruction adds 1 to the destination operand.
- The operand may be a byte or word and is treated as an unsigned binary number.
- The destination operand may be a register or memory location.

Example INC CX Add 1 to contents of CX
INC AL Add 1 to contents of AL register.

4. AAA – ASCII Adjust after addition

Mnemonic AAA **Flags** AF and CF flags are changed, while OF, PF, SF, ZF are left unchanged.

Algorithm If lower nibble of AL > 9 or AF = 1 then : AL = AL + 6
AH = AH + 1 AF = 1 CF = 1
else : AF = 0 , CF = 0
In both cases, clear the higher nibble of AL.

Addr. Mode Implied addressing mode

Operation
- Numerical data coming into a computer from a terminal through keyboard is usually in ASCII code. The numbers 0 to 9 are represented by ASCII codes 30 H to 39 H. The 8086 allows to add the ASCII codes for two decimal digits without masking off "3" in the upper nibble of each. After addition, AAA instruction is used to make sure that the result is the correct unpacked BCD.
- The AAA instruction works only on AL register.
- If the lower nibble of AL register after addition is greater than 9, add 6 to it and increment AH by 1. The auxiliary carry flag and carry flag are set.

Example Assume
AL = 0 0 1 1 0 1 0 1 ASCII 5
BL = 0 0 1 1 1 0 0 1 ASCII 9
ADD AL, BL

```
          0 0 1 1   0 1 0 1
    +     0 0 1 1   1 0 0 1
          ---------------------
          0 1 1 0   1 1 1 0   → Result of
    +                 0 1 1 0   addition
                                Invalid BCD
          ---------------------
AAA       0 0 0 0   0 1 0 0
Carry → [1]  Clear higher
             nibble
```

⇓

(04 H) → Result in AL valid BCD.
(01 H) → Result in AH. The carry is stored in AH register

5. DAA – Decimal adjustment for addition

Mnemonic DAA **Flags** It changes AF, CF, PF, ZF and SF.

Algorithm If lower nibble of AL > 9 or AF = 1 then, AL = AL + 06 H, AF = 1
If AL > 9F H or CF = 1 then, AL = AL + 60 H, CF = 1

Addr. Mode Implied addressing mode

Operation AL ← Sum in adjusted to packed BCD format

- This instruction is used to make sure that the result of adding two packed BCD numbers is adjusted to be a valid BCD number.
- It operates only on AL register.
- If number in the lower nibble of AL register after addition is greater than 9 or if the auxiliary carry flag is set add 6.
- If the lower nibble of AL is greater than 9 or if the carry flag is set then add 60 H.

Example If AL = 59 H valid BCD, BL = 34 H valid BCD

ADD AL, BL

```
    0 1 0 1   1 0 0 1
  + 0 0 1 1   0 1 0 0
  ---------------------
    1 0 0 0   1 1 0 1
    \_____/   \_____/
       8         D
```

AL = 8DH invalid BCD after addition of AL and BL

```
DAA →  1 0 0 0   1 1 0 1
     + 0 0 0 0   0 1 1 0
     ---------------------
       1 0 0 1   0 0 1 1
       \_____/   \_____/
          9         3
```

∴ AL = 93 H BCD

12.7.2 Subtraction

Subtraction

- **SUB** (Subtract byte or word)
- **SBB** (Subtract byte or word with borrow)
- **DEC** (Decrement byte or word by 1)
- **AAS** (ASCII adjust for subtraction)
- **DAS** (Decimal adjust for subtraction)
- **NEG** (Negate byte or word)
- **CMP** (Compare byte or word)

1. SUB – Subtract byte or word

Mnemonic SUB destination, source

Flags The flags affected are AF, CF, OF, PF, SF and ZF.

Algorithm destination = destination – source

Addr. Mode –

Operation destination ← destination – source

- This instruction subtracts a number from source with number from destination and puts result in destination location.
- Both the operands i.e. source and destination, cannot be memory locations.

Examples

1. SUB BL, CL
   BL ← BL – CL
   This instruction subtracts the contents of CL register from BL and result is stored in BL.
2. SUB AX , [2048]
   AX ← AX – contents of memory location
   - This instruction subtracts the data at memory locations whose offset in DS are [2048] and [2049] with data in AX register.
   - The result is stored in the AX register.
3. SUB [2048], AX
   [2048] ← [2048] – AL, [2049] ← [2049] – AH
   - This instruction subtracts the data at memory locations whose offset in DS are [2048] and [2049] with data in AL register and AH register.
   - The result is stored at memory locations whose offset in DS are [2048] and [2049] i.e. (25188 H and 25189 H).
4. SUB COST, 14 H.
   COST ← COST–14 H
   [Cost : is a memory location]
   - This instruction will subtract the immediate Data 14 H with contents of memory location COST whose offset is given in the instruction [i.e. memory location 28354 H]
   - Result of subtraction is stored at COST.
5. SUB AL, 24 H
   AL ← AL – 24
   - This instruction subtracts the immediate number 24 H with the contents of AL register. The result is stored in AL.

The type of both the operands should match i.e. byte or word.

Destination	Source
Register	Register
Register	Memory
Memory	Register
Accumulator	Immediate
Register	Immediate
Memory	Immediate

2. SBB – Subtract byte or word with borrow

Mnemonic SBB destination, source

Flags SF flag is not affected.

Algorithm Destination = destination – source – CY

Addr. Mode Register addressing mode.

Operation Destination – source – CY

- This instruction subtracts source and carry flag (i.e. Borrow) from destination.
- The source and destination can be 8/16 bit register or memory location . The source and destination can also be a 8/16 bit register or memory location and immediate data .
- The segment registers cannot be used . The memory uses DS as segment register .
- The result is stored in destination.
- Both the source and destination may be words or bytes.
- They can be signed or unsigned binary numbers.
- It can be used to write routines that subtract the numbers longer than 16 bits i.e. double words ,quad words etc.

Example SBB AL, BL AL ← AL – BL – CY

- This instruction subtracts contents of BL and CY from AL.
- The result is stored in AL.

3. DEC – Decrement byte or word by 1

Mnemonic DEC destination

Flags CF is not affected

Algorithm destination = destination – 1

Addr. Mode Register Addressing mode

Operation destination ← destination – 1

- This instruction subtracts 1 from the destination word or byte.
- The destination can be a register or a memory location.
- This instruction cannot be used to decrement the contents of segment registers.

Example DEC AL AL = AL – 1

- This instruction subtracts 1 from the contents of CL and result is stored in CL.

4. AAS – ASCII adjust for subtraction

Mnemonic AAS

Flags It updates AF, CF but OF, PF, SF and ZF are left undefined.

Algorithm If lower nibble of AL > 9 or AF = 1 then :
AL = AL – 06 H AH = AH – 1 AF = 1 CF = 1
else AF = 0 CF = 0
In both cases clear the high nibble of AL

Addr. Mode Implied addressing mode

Operation

- The number 0 to 9 are represented as 30 – 39 in ASCII code
- The 8086 allows us to subtract the ASCII codes for two decimal digits without masking "3" in upper nibble of each.
- The AAS instruction is used to make sure that the result is in unpacked BCD form.
- It works only on the AL register.

Example Let AL = 0011 1001 = 39 H = ASCII 9
Let BL = 0011 0101 = 35 H = ASCII 5

SUB AL, BH → Result AAS	=	0 0 1 1 1 0 0 1	
	–	0 0 1 1 0 1 0 1	
	0	0 0 0 0 0 1 0 0	= BCD = 04
CF = 0 no borrow required			

5. DAS – Decimal adjust for subtraction

Mnemonic DAS

Flags The OF is undefined.

Algorithm If lower nibble of AL > 9 or AF = 1 then
AL = AL – 06 H AF = 1
If AL > 9F H or CF = 1 then AL = AL – 60 H
CF = 1

Addr. Mode Implied Addressing mode

Operation AL ← difference in AL adjusted to packed BCD format

- This instruction is used after subtracting two packed BCD numbers to make sure that the result is correct packed BCD. The result of subtraction must be in AL for DAS to work correctly.
- This instruction works only on AL register.
- If the lower nibble in AL after a subtraction is greater than 9 or if the auxiliary carry flag is set then the DAS instruction will subtract 6 from the lower nibble of AL.
- If the result in the upper nibble is greater than 9 or the carry flag is set the DAS instruction will subtract 60 H from the AL register

Example

AL = 1000 0110 = 86 BCD	AL = 0010 1111 = 2F H, CF = 0
BH = 0101 0111 = 57 BCD	Lower nibble of result is F H, i.e. greater than 9 so DAS subtracts 0000 0110 to give
SUB AL, BH	AL = 0010 1001 = 29 H BCD
DAS	

6. NEG – Negate byte or word

Mnemonic NEG destination **Flags** All flags are affected.

Algorithm Invert all bits of operand and add 1 to inverted operand

Addr. Mode Register Addressing mode

Operation
- This instruction replaces the number is a destination with 2's complement of that number
- The destination can be a register or memory location.
- This instruction forms the 2's complement by subtracting the original word or byte in the indicated destination from zero.
- It is useful for changing the sign of a signed word or byte.
- Attempt to negate a byte containing – 128 or a word containing – 32, 768 causes no change to the operand and sets Overflow flag.

Example NEG AX

AX ← 2's complement of number in AX.

Suppose AX =00A3 H = 0000 0000 1010 0011

Then it's 2's complement will be,

0000 0000 1010 0011 = 1111 1111 0101 1100 ← 1's complement of 00A3 H

\+ 1

1111 1111 0101 1101 ← 2's complement of number

= FF5D H ← contents of AX after execution

7. CMP – Compare byte or word

Mnemonic CMP destination, source. **Flags** AF, OF, SF, ZF and PF, CF are updated according to the result

Algorithm Destination – source

Addr. Mode Register addressing mode

Operation
- This instruction compares a word/byte from source with byte/word from destination. The comparison is done by subtracting the source byte or word from the destination byte or word.
- The result is not stored in either of the destination or source. The destination and source remain unchanged, only flags are updated.
- The source may be register, memory location or an immediate number.
- The destination may be a register or memory location.

Compare	CF	ZF	SF	
Source > destination	1	0	1	Subtraction required borrow, so CF = 1
Source < destination	0	0	0	No borrow required CF = 0
Source = destination	0	1	0	Result of subtraction is zero

Example CMP BH, CL

Compares a byte in CL with byte in BH.

Let CL → 05 H and BH → 04 H, CF → 0, ZF → 0, SF → 0:

= 0000 0100 → BH

– 0000 0101 → CL

Carry flag → [0] 0000 0100 → Result of subtraction

Carry flag → [1] 1111 1011 ← 2's complement of the result i.e. result of comparison of BH and CL registers

> **Note :** (i) The source and destination both cannot be memory locations.
> (ii) The compare instructions are often used with conditional Jump instructions.
> (iii) One of the operand must be in register.

12.7.3 Multiplication

Multiplication
- **MUL** (Multiply byte or word unsigned)
- **IMUL** (Multiply byte or word signed)
- **AAM** (Integer multiply byte or word ASCII adjust for multiply)

1. MUL – Multiply byte or word unsigned

Mnemonic MUL source **Flags** (i) AF, PF, SF, ZF are undefined. (ii) CF and OF will both be 0.

Algorithm When operand is a byte AX = AL * operand

When operand is a word (DX : AX) = AX * operand

Addr. Mode Register addressing mode

Operation
- This instruction multiplies an unsigned byte from source with an unsigned byte in the AL register or an unsigned word from source with an unsigned word in AX.
- When a byte is multiplied by contents of AL, the result is stored in AX. The MSB of result is stored in AH register and the LSB of result is stored in the AL register.
- When a word is multiplied by contents of AX, the product can be double word . The MSB of multiplication is stored in DX and LSB in AX register.
- The source can be a register or memory location.

- This instruction cannot be used to multiply immediate data. If we want to multiply immediate data. Then that immediate data has to be stored into some valid register and then multiplied with byte or word in AL or AX

Example
MUL CX

DX:AX =CX * AX : Result of multiplication MSB will be stored in DX register and the LSB will be stored in AX register.

2. IMUL – Multiply byte or word signed

Mnemonic IMUL source **Flags** AF, PF, SF and ZF are undefined.

Algorithm When operand is a byte → AX = AL * operand
When operand is a word → (DS : AX) = AX * operand

Addr. Mode Register Addressing mode

Operation Byte operands → AX ← AL * source
Word operands → (DS : AX) ← AX * source

- This instruction multiplies a signed byte from some source and a signed byte in AL, or a signed word from some source and a signed word in AX.
- The source can be register or memory location.
- When a signed byte is multiplied by AL a signed result will be put in AX.
- When a signed word is multiplied by AX, the MSB 16-bits are put in DX and LSB 16-bits are put in AX.
- If the magnitude of product does not require all bits of the destination, the unused bits are filled with copies of the sign bit.
- To multiply a signed byte by a signed word it is necessary to move signed byte into lower byte of word and fill the upper byte of word with copies of sign bit. This can be done by CBW instruction.
- If the upper byte of 16 bit result or upper word of 32 bit result contains only copies of sign bit (all 0's or all 1's) then the CF and OF will both be zeros.
- If the upper byte of 16 bit result or upper word of 32 bit result contains part of the product then the CF and OF will both be zeros.

Example
IMUL BL

Let AL = 69 decimal = 0100 0101 = 45H
BL = 14 decimal = 00001110 = 0EH

```
IMUL BL      45 H   ← Contents of AL register
           ×  0E H   ← Contents of BL register
           ---------
             03CE H  ← Result of multiplication  AX = 03CE H
```

MSB = 0, Positive result magnitude in true form. ∴ SF = 0, CF = OF = 1

3. AAM – Integer multiply byte or word ASCII adjust for multiply

Mnemonic AAM **Flags** It updates PF, SF and ZF. The AF, CF and OF are left undefined.

Algorithm AH = Quotient of AL/10 AL = remainder of AL/10

Addr. Mode Implied addressing mode

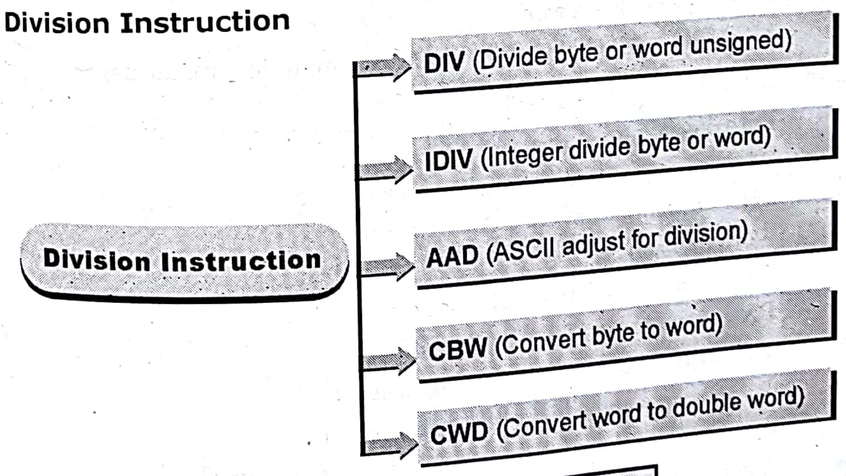
Operation

- Numerical data coming into a computer from a terminal through keyboard is usually in ASCII code. The numbers 0 to 9 are represented by ASCII codes 30 H to 39 H.
- Before multiplying two ASCII digits, the upper nibble bits of each need to be masked. This leaves unpacked BCD in each byte. After the two unpacked BCD digits are multiplied, the AAM instruction is used to adjust the product of two unpacked BCD digits in AX.
- It works only on register AL.
- It is used after multiplying the two unpacked BCD numbers

Example

Let AL → 0000 0101 = unpacked BCD 5
BH → 0000 1001 = unpacked BCD 9
MUL BH : AL * BH → Result in AX register
AX = 0000 0000 00101101 = 002D H
AAM AX : 0000 0100 0000 0101 = 0405 H
Which is unpacked BCD for 45.

12.7.4 Division Instruction



1. DIV – Divide byte or word unsigned

Mnemonic DIV source **Flags** All flags are undefined.

Let AX = 37D7 H = 14,295 decimal,
BH = 97 H = 151 decimal

Algorithm When operand is a byte :
AL = AX/operand (Quotient) AH = remainder (Modulus)
When operand is a word : AX = (DS : AX) / operand (Quotient)
DX = remainder (modulus)

Addr. Mode Register Addressing mode

Operation This instruction basically performs two operations :

1. Divide an unsigned word by a byte.
2. Divide unsigned double word by a word.

For 1st operation, the word must be in AX register. After division,
AL → 8 bit quotient, AH → 8 bit remainder

- If an attempt is made to divide by a or quotient is too large to fit in AL. 8086 will execute a type 0 interrupt.
- For 2nd operation, MSB of word of the double word must be in DX and LSB must be in AX.
- AX → 16 bit quotient, DX → 16 bit remainder

2. IDIV – Integer divide byte or word

Mnemonic IDIV source **Flags** All flags are undefined

Algorithm When operand is byte :
AL = AX/operand AH = remainder (modulus)
When operand is a word :
AX = (DX : AX) / operand DX = remainder (modulus)

Addr. Mode Register Addressing mode

Operation Byte division :
AL ← quotient of (DS : AX) / source
AH ← remainder of (DS : AX) / source
Word division :
AX ← quotient of (DS : AX) / source
DX ← remainder of (DS : AX) / source
This instruction performs two operation :

1. Divide a signed word by a signed byte
2. Divide a signed double word by a signed word.

Example A signed word divided by a single byte
Let AX = 03 AB H BL = 00 D3 H
IDIV BL
Quotient in AL = EC H Remainder in AH = 27 H

3. AAD – ASCII adjust for division

Mnemonic AAD **Flags** (i) The PF, SF and ZF are affected.
(ii) AF, CF and OF are undefined after AAD.

Algorithm AL = (AH * 10) + AL AH = 0
AL ← 10* (AH) + AL AH ← 0

Addr. Mode Implied Addressing mode

Operation

- It converts two unpacked BCD digits in AH and AL to the equivalent binary number in AL. This adjustment must be made before dividing the two unpacked BCD digits in AX, by an unpacked BCD byte. After the division,
  AL → unpacked BCD quotient
  AH → unpacked BCD remainder

Example AX = 0607H unpacked BCD for 67 decimal
CL = 09 H, now adjust to binary using AAD.
AAD
DIV CL

This instruction will perform following operation.
(AH)* 10 = 06 * 10 = $(60)_{decimal}$ = 3CH
(AH) * 10 + (AL) = $(60)_{10}$ + $(7)_{10}$
= 3 C H + 7 H = 43 H
AL = 43 H and AH = 00 H
∴ AX (New) = 0043 H
Divide AX by unpacked BCD in CL
Quotient AL = 07 H unpacked BCD
Remainder AH = 04 H unpacked BCD

4. CBW – Convert byte to word

Mnemonic CBW **Flags** No flags are affected.

Algorithm If MSB bit of AL = 1 then AH = 255 (FF H)
else AH = 0

Addr. Mode Implied Addressing mode

Operation

- This instruction copies the sign bit in AL to all the bits in AH. AH is then said to be a sign extension of AL.
- This operation must be done before a signed byte in AL can be divided by another signed byte with IDIV instruction.

Example AX = 00AC H
= – 163 decimal

CBW
Convert signed byte in AL to signed word in AX.
Result : 1 1 1 1 1 1 1 1 1 0 1 0 0 0 1 1
= – 163 decimal

5. CWD – Convert word to double word

Mnemonic CWD **Flags** No flags are affected.

Algorithm If MSB bit of AX = 1 then, DX = 655 35 (FFFF H)
else DX = 0

Addr. Mode Implied addressing mode

Operation
- This instruction copies the sign bit of word in AX to all the bits of DX register. DX is sign extension of AX then.
- It must be done before signed word in AX can be divided by another signed word with IDIV instruction.

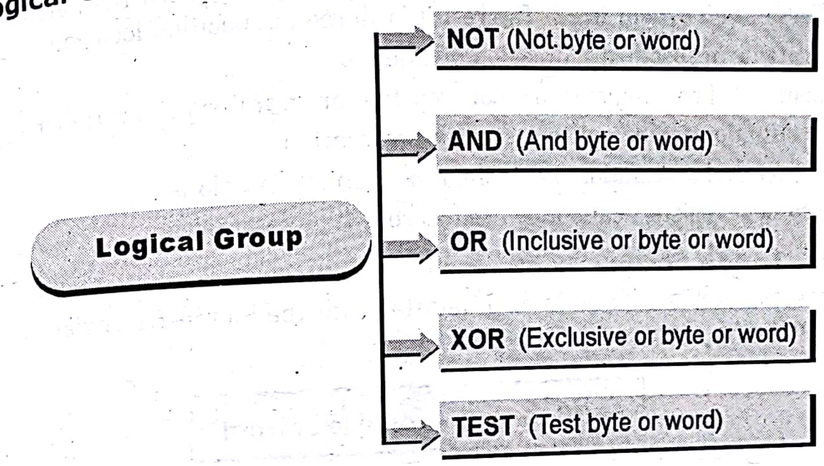
12.8 Bit Manipulation Instructions

The 8086/8088 provide three groups of instructions, for manipulating bits within both bytes and words. Three groups are listed in Table 12.8.1.

Table 12.8.1 : Bit manipulation instructions

LOGICALS	
NOT	Not byte or word
AND	And byte or word
OR	Inclusive or byte or word
XOR	Exclusive or byte or word
TEST	Test byte or word
SHIFTS	
SHL/SAL	Shift logical/arithmetic left byte or word
SHR	Shift logical right byte or word
SAR	Shift arithmetic right byte or word
ROTATES	
ROL	Rotate left byte or word
ROR	Rotate right byte or word
RCL	Rotate through carry left byte or word
RCR	Rotate through carry right byte or word

12.8.1 Logical Group



1. NOT – Not byte or word

Mnemonic NOT destination. **Flags** No flags are affected.

Algorithm If bit is 1 turn it to 0 If bit is 0 turn it to 1

Addr. Mode Register Addressing mode

Operation Destination ← $\overline{\text{Destination}}$

- This instruction inverts each bit of byte or word at the specified destination i.e. it finds 1's complement of the number.
- The destination can be a register or a memory location.
- The destination cannot be immediate data.
- The destination cannot be a segment register.

Example NOT AX AX ← $\overline{\text{AX}}$

This instruction complements the contents of AX register.

2. AND – And byte or word

Mnemonic AND destination, source

Flags CF and OF are both 0 after the execution of AND instruction.
PF, SF and ZF are updated by AND instruction.
AF is undefined.

Algorithm Destination = destination ∧ source

Addr. Mode Register addressing mode

Operation 1 AND 1 = 1 1 AND 0 = 0
0 AND 1 = 0 0 AND 0 = 0

- This instruction ANDs each bit in a source byte or word with the same number bit in a destination byte or word. The result is stored at specified location.
- The contents of specified source do not change.
- The source can be a register, memory location or an immediate number.
- The destination can be register, or memory location.
- The source and destination both cannot be memory locations .
- The segment registers cannot be used as source or destination .

Example AND AL, BL

This instruction ANDs each bit in a AL register with the bits in BL register. The result is stored in the AL register.

3. OR – Inclusive or byte or word

Mnemonic OR destination, source.

Flags CF = 0, OF = 0, after OR instruction PF, SF and ZF are affected. AF is undefined.

Algorithm destination ← destination ∨ source.

Addr. Mode Register addressing mode

Operation 1 OR 1 = 1 1 OR 0 = 1 0 OR 1 = 1 0 OR 0 = 0

- This instruction ORs each bit in a source with the corresponding bits in a destination.
- The result is stored in the specified destination.
- Contents of source will not change.
- Source can be register, memory location or immediate number.
- The source and destination both cannot be memory locations.
- The segment registers cannot be used as source or destination .
- Destination can be a register or Memory location.

Example OR AL , BL

This instruction ORs each bit in a AL register with the bits in BL register. The result is stored in the AL register.

4. XOR – Exclusive or byte or word

Mnemonic XOR destination, source.

Flags CF = 0, OF = 0 after XOR instruction. PF, SF and ZF are affected. AF will be undefined.

Algorithm 1 XOR 1 = 0 1 XOR 0 = 1 0 XOR 1 = 1 0 XOR 0 = 0

Addr. Mode Register addressing mode

Operation

- This instruction logically XORs each bit of the source byte or word the corresponding bit in the destination and stores the result in the destination.
- The source may be register, memory location or immediate number.
- The destination may be register or memory location.
- The source and destination both cannot be memory locations.
- The segment registers cannot be used as source or destination.

Example XOR AL, BL

This instruction XORs each bit in a AL register with the bits in BL register. The result is stored in the AL register.

5. TEST – Test byte or word

Mnemonic TEST destination, source.

Flags CF = 0, OF = 0
PF, SF, ZF will be affected to show result of ANDing.

Algorithm destination ← destination ∧ source.

Addr. Mode Immediate addressing mode

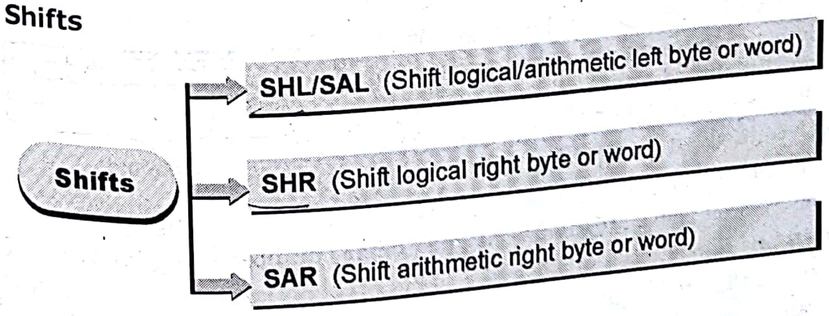
Operation 1 AND 1 = 1 1 AND 0 = 0 0 AND 1 = 0 0 AND 0 = 0

- This instruction ANDs contents of a source byte or word with contents of specified destination word.
- The source can be register, memory location, immediate data.
- The source and destination both cannot be memory locations.
- Segment registers are not allowed to be used as source or destination.
- The destination can be a register or memory location.
- Flags are affected, but neither operand is changed.
- It is used to set flags before conditional JUMP.

Example TEST AL, 75 H

Let AL = 0 1 1 1 0 1 0 1 AND Immediate number 75H with AL.
PF = 0, SF = 0, ZF = 0, CF = 0, OF = 0

12.8.2 Shifts



1. SHL/SAL – Shift logical/arithmetic left byte or word

Mnemonic SAL/SHL destination, count. **Flags** All flags are affected.

Algorithm Shift all bits left, the bit that goes off is set to CF. Zero bit is inserted in the right most position.

Addr. Mode Immediate addressing mode

Operation CF ← MSB ← LSB ← 0

- SAL/SHL are two mnemonics for the same instruction. This instruction shifts each bit of the specifies destination, some number of bit positions to the left. As left bit is shifted out of the LSB position, 0 is put in the LSB position. The MSB will be shifted into CF.

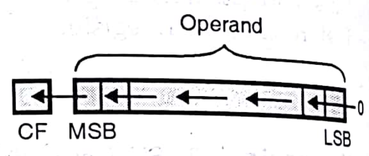


Fig. 12.8.1

- The count can be any immediate number.
- In case of multiple bit shifts, CF will contain the bit most recently shifted in from the MSB.
- Bits shifted into CF previously will be lost.
- This instruction can be used to multiply an unsigned binary number by a power of 2.
- The destination can be a memory location or register.
- The count is specified in the CX register.
- Negative shifts are illegal.

Example SHL AX , 01 This instruction shift AX by 1 bit to the left.

2. SHR – Shift logical right byte or word

Mnemonic SHR destination, count. **Flags** All the flags are affected.

Algorithm Shift all bits right, the bit that goes off is set to CF. Zero bit is increased to the right most position.

Addr. Mode Register addressing mode

Operation 0 → MSB → LSB → CF

- This instruction shifts each bit in specified destination some number of bit position to the right Bit shifted from LSB, goes to CF.

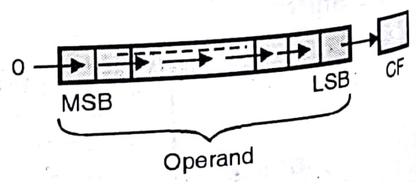


Fig. 12.8.2

- For Multi bit shift CF will contain bit most recently shifted out from LSB position.
- Bits shifted into CF previously will be lost.
- This instruction is used to divide unsigned binary number by power of 2.
- Count is any immediate number.
- Negative shifts are illegal
- The count is specified in CL = 02H the CX register.

Example SHR AX, CL

This instruction shifts each bit in the accumulator by two times to the right.

3. SAR – Shift arithmetic right byte or word

Mnemonic SAR destination, count **Flags** All flags are affected

Algorithm Shift all bits right, the bit that goes off is set to CF.
The sign bit that is inserted to the leftmost position has the same value as before shift.

Addr. Mode Immediate addressing mode

Operation MSB → MSB → LSB → CF

- This instruction shifts each bit is specified destination, same number of bit position of the right. Number of bits to be shifted depends upon count.
- As bit is shifted out of the MSB position, a copy of old MSB is put in the New MSB position. i.e. the sign bit is copied into the MSB. LSB is moved into CF.

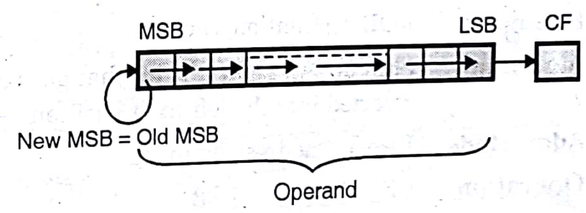
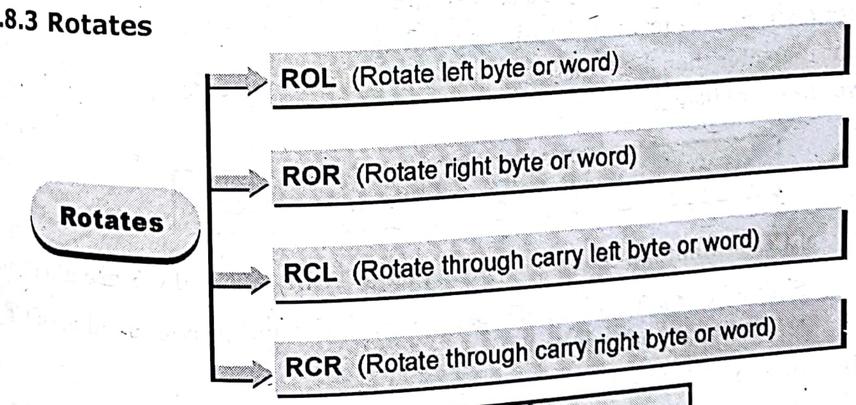


Fig. 12.8.3

- Bits shifted into CF previously will be lost.
- The count can be an any immediate number or specified in the CX register
- Negative shifts are illegal.

12.8.3 Rotates



1. ROL – Rotate left byte or word

Mnemonic ROL destination, source **Flags** Only CF and OF are affected.

Algorithm Shift all bits left, the bit goes off is set to CF and the same bit is inserted to the right most position.

Addr. Mode Immediate addressing mode.

Operation CF ← MSB ← LSB

- This instruction rotates all the bits in a specified word or byte to the left, by some bit positions.
- The data bit rotated out of MSB is circled back into the LSB. The data bit rotated out of MSB is also copied to CF.
- CL is default register used for rotate instruction when count is greater than 1.

Example ROL AX, 01 H

This instruction rotates the contents of AX register by 1 bit to left.

2. ROR – Rotate right byte or word

Mnemonic ROR destination, count. **Flags** –

Algorithm Shift all bits right, the bit that goes off is set to CF and the same bit is inserted into the left most position

Addr. Mode Register addressing mode.

Operation CF MSB → LSB

- This instruction rotates all the bits of specified destination operand to the right. The bit moved out of the LSB is rotated around into the MSB.
- The data bit moved out of LSB is also copied into CF.
- The destination may be memory location or register.
- The count if greater than 1 should be specified in the CX register.
- Negative shifts are illegal.

3. RCL – Rotate through carry left byte or word

Mnemonic RCL destination, count. **Flags** Only CF and OF are affected.

Algorithm Shift all bits left, the bit that goes is set to CF and previous value of CF is inserted to the rightmost position.

Addr. Mode Immediate addressing mode

Operation CF ← MSB ← LSB

- This instruction rotates all the bits in specified destination by some number of bit positions to the left. The destination can be register or memory location.

- Negative shifts are illegal.
- CL is default register used for rotate instruction when count is greater than 1.

4. RCR – Rotate through carry right byte or word

Mnemonic RCR destination, count. **Flags** It affects only CF and OF

Algorithm Shift all bits right, the bit that goes off is set to CF and previous value of CF is inserted to the left most position

Addr. Mode Immediate addressing mode

Operation CF → MSB → LSB

- This instruction rotates all the bits is specified word or byte same number of bit positions to right. The destination can be register or memory location.
- Negative shifts are illegal.
- CL is default register used for rotate instruction when count is greater than 1.

12.9 Processor Control Instruction

Q. Explain control transfer and branching instruction of 8086.

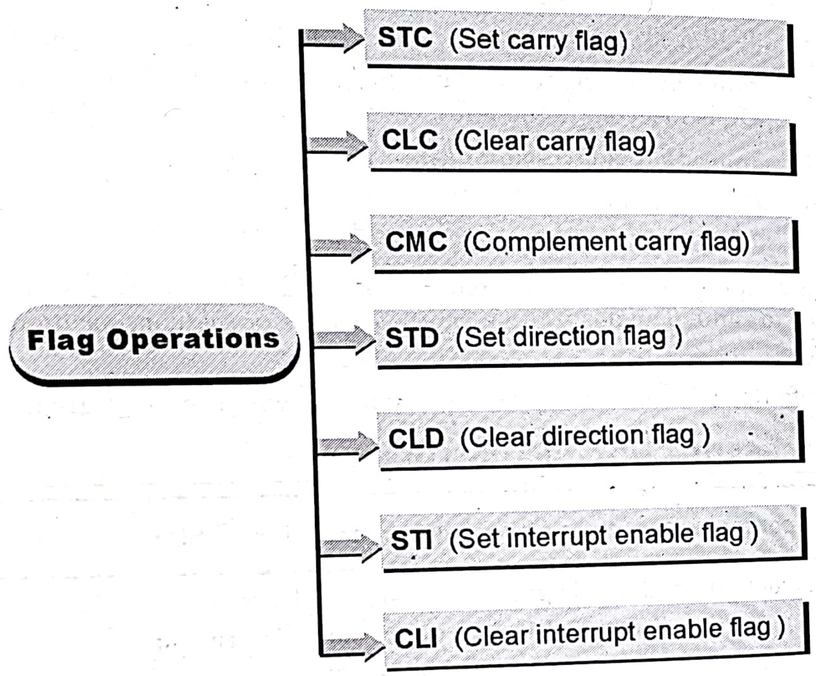
The instructions under this group are listed as follows :

Table 12.9.1 : Processor control instructions

FLAG OPERATIONS	
STC	Set carry flag
CLC	Clear carry flag
CMC	Complement carry flag
STD	Set direction flag
CLD	Clear direction flag
STI	Set interrupt enable flag
CLI	Clear interrupt enable flag
NO OPERATION	
NOP	No operation
EXTERNAL SYNCHRONIZATION	
HLT	Halt until interrupt or reset
WAIT	Wait for $\overline{TEST}$ pin active
ESC	Escape to external co-processor
LOCK	Lock bus during next instruction

These instructions allow programs to control various CPU functions. One group of instructions updates flags and another group is used primarily for synchronizing the 8086 or 8088 with external events. A final instruction causes the CPU to do nothing. Except for the flag operations, none of the processor control instructions affect the flags.

12.9.1 Flag Operations



1. CLC – Clear carry flag

Mnemonic CLC **Flags** Except carry, no other flags are affected.
Algorithm CF = 0
Addr. Mode Implied addressing mode
Operation CF ← 0 This instructions clears (resets) the carry flag to zero.

2. STC – Set carry flag

Mnemonic STC **Flags** Except carry no other flags are affected.
Algorithm CF = 1
Addr. Mode Implied addressing mode
Operation This instruction sets the carry flag.

3. CMC – Complement carry flag

Mnemonic CMC **Flags** Except carry no other flags are affected.
Algorithm if CF = 0 then CF = 1 if CF = 1 then CF = 0

Addr. Mode Implied addressing mode
Operation $CF = \overline{CF}$ This instruction inverts the value of carry flag

4. CLD – Clear direction flag

Mnemonic CLD **Flags** Except direction flag, no other flags are affected.
Algorithm DF = 0
Addr. Mode Implied addressing mode
Operation
- It resets (DF) direction flag to zero. If the direction flag is reset, SI and / or DI will be automatically incremented during string operations.
- When one of the string instructions like MOVS, CMPS or SCAS executes it automatically increments the SI and / or DI registers.

5. STD – Set direction flag

Mnemonic STD **Flags** Except Direction no other flags are affected.
Algorithm DF = 1
Addr. Mode Implied addressing mode
Operation
- It is used to set the direction flag, so that SI and / or DI will be decremented by string instructions. i.e. CMPSB, MOVSB, STOSB, STOSW etc.

6. CLI – Clear interrupt enable flag

Mnemonic CLI **Flags** Except interrupt flag, it does not affect any other flag.
Algorithm IF = 0
Addr. Mode Implied addressing mode .
Operation
- This instruction resets the interrupt flag to zero. No other flags are affected. If the interrupt flag is reset, the 8086 will not respond to an interrupt signal on its INTR input.
- This instruction has no effect on the NMI.

7. STI – Set interrupt enable flag

Mnemonic STI **Flags** Except interrupt flag, it does not affect any other flag.
Algorithm IF = 1
Addr. Mode Implied addressing mode
Operation
- This interrupt sets the interrupt flag to 1. This enables INTR interrupt of the 8086.

12.9.2 No Operation

Mnemonic NOP **Flags** It does not affect any flag.

NOP : No operation

Algorithm Do nothing

Addr. Mode Implied addressing mode

Operation The execution of this instruction causes the CPU to do nothing.

- This instruction causes the CPU to do nothing. This instruction uses three clock cycles and increments the instruction pointer to point to the next instruction.
- It can be used to increase the delay of a delay loop.

12.9.3 External Synchronisation

External Synchronisation
- HLT (Halt until interrupt or reset)
- WAIT (Wait for TEST pin active)
- ESC (Escape to external processor)
- LOCK (Lock bus during next instruction)

1. HLT – Halt until interrupt or reset

Mnemonic Halt processing **Flags** No flags are affected.

Operation

- The HLT instruction will cause the 8086 to stop fetching and executing instructions. The 8086 enters into a halt state. To come out of the halt state, there are 3 ways given below.
  (i) Interrupt signal on INTR pin (ii) Interrupt signal on NMI pin
  (iii) Reset signal on reset pin.
- It may be used as an alternative to an endless software loop in situations where a program must wait for an interrupt.

2. WAIT – Wait for TEST pin active

Mnemonic WAIT **Flags** No flags are affected.

Operation When this instruction executes, the 8086 enters on idle condition in which it is doing no processing.

- The 8086 will stay in this idle state until 8086 $\overline{TEST}$ input pin is made low or on interrupt signal is received on the INTR or NMI interrupt pins.

- If a valid interrupt occurs while the 8086 is in the idle state, the 8086 will return to idle state after the interrupt service procedure executes.
- It is used to synchronize the 8086 with external hardware. Such as 8087 math processor.

3. ESC – Escape to external processor

Mnemonic ESC external – opcode, source.

Operation This instruction is used to pass instruction to a coprocessor, such as 8087 math co-processor which shares the address and data bus with on 8086.

- The instruction for the Co-processor are represented by a 6 bit code embedded in the escape instruction.
- When the 8086 fetches on ESC instruction, the coprocessor decodes the instruction and carries out the action specified by the 6 bit code specified in the instruction.
- In most cases 8086 treats the ESC instruction as a NOP in some cases 8086 will access a data item in memory for co-processor.

4. LOCK – Lock bus during next instruction

Mnemonic LOCK

Operation Many multiprocessor systems contain several microprocessors. Each microprocessor has its own local buses and memory. The individual microprocessors are connected together by a shared system bus so that each can access system resources such as disk drives or memory.

- Each microprocessor takes control of the system bus. Only when it needs to access some resource.
- Lock prefix allows a microprocessor to make sure that another processor does not take control of the system bus.
- While it is in the middle of a critical instruction which uses the system bus when an instruction with lock prefix executes the 8086 will assert its bus lock signal output. This signal is connected to an external bus controller, which then prevents any other processor from taking over the system bus.

Example LOCK XCHG SEMAPHORE, AL : The XCHG instruction requires two bus accesses. The lock prefix prevents another processor from taking control of system bus between two accesses.

12.10 Program Transfer Group

8086/8088 provides you :

1) Unconditional CALL
2) JMP → Conditional / Unconditional
3) INT (Software interrupt) and many more.

- These instructions are also referred to as Branch instructions.
  We have four subgroups under this :

• Unconditional transfers.	• Conditional transfers.
• Iteration control.	• Interrupt relate instructions.

Instructions under these subgroups are listed as follows in Table 12.10.1.

Table 12.10.1 : Program transfer instructions

UNCONDITIONAL TRANSFERS		ITERATION CONTROLS	
CALL	Call procedure	LOOP	Loop
RET	Return from procedure	LOOPE / LOOPZ	Loop if equal/zero
JMP	Jump	LOOPNE/LOOPNZ	Loop if not equal/not zero
CONDITIONAL TRANSFERS		**INTERRUPTS**	
JA/JNBE	Jump if above / not below or equal	INT	Interrupt
JAE/JNB	Jump if above or equal / not below	INTO	Interrupt if overflow
JB/JNAE	Jump if below / not above nor equal	IRET	Interrupt return
JBE/JNA	Jump if below or equal / not above		
JC	Jump if carry		
JE/JZ	Jump if equal / zero		
JG/JNLE	Jump if greater / not less nor equal		
JGE/JNL	Jump if greater or equal / not less		
JL/JNGE	Jump if less / not greater nor equal		
JLE/JNG	Jump if less or equal / not greater		
JNC	Jump if not carry		
JNE/JNZ	Jump if not equal / not zero		
JNO	Jump if not overflow		
JNP/JPO	Jump if not parity / parity odd		
JNS	Jump if not sign		
JO	Jump if overflow		
JP/JPE	Jump if parity / parity even		
JS	Jump if sign		

12.10.1 Unconditional Transfers

The unconditional transfer instructions may transfer control to a target instruction within the current code segment (intrasegment transfer) or to a different code segment (intersegment transfer). The transfer is made unconditionally any time the instruction is executed.

Unconditional Transfers
- CALL (Call a procedure)
- RET (Return from Procedure)
- JMP (Jump)

1. CALL – Call a procedure

Q. Describe execution of CALL instruction.

Mnemonic : CALL procedure

Operation : This instruction is used to transfer program control to a subroutine or a procedure. There are two basic types of CALLS : NEAR Call and FAR Call.

- **Near Call :** A near call is a call to a procedure which is in the same segment, which has the CALL instruction. It is also called as Intra segment call.

(i) If the call to the subroutine or procedure with a 16-bit signed displacement the 8086 will decrement the SP by 2 and push the IP contents onto the stack. Then adds the signed 16 bit value of DISP of IP. The contents of CS are unchanged. Such a call is an intra segment direct call.

e.g. DISP PROC NEAR : It indicates that DISP is the name of procedure which is in the same code segment. DISP is 16 bit signed displacement.

Addressing mode : Relative addressing mode

(ii) If the CALL is to a subroutine is in the same segment and is addressed by the contents of a 16-bit general register/memory. Then the 8086 decrements the SP by 2 and pushes the contents of IP onto the stack and then the contents of specified 16 bit register/memory location.

The registers can be BX, SI or DI to provide the new value of IP.

The memory location is addressed by contents of 16 bit register such as BX, SI, DI into IP.

The contents of CS remain unchanged. Such a call is called is an intra segment indirect call.

e.g. : CALL Reg 16.

- **Far Call :** A far is a call to a procedure which is in a different segment from that which contains the CALL instruction. Far calls are also called as intersegment calls.

(i) If the far call to a subroutine or procedure is with a signed displacement, the 8086 decrements SP by 2 and pushes the contents of CS onto the stack and moves the lower 16 bit value of the number like DISP in CALL DISP into CS. The SP is again decremented by 2. The contents of IP are pushed onto the stack. The IP is then loaded with the higher 16 bits i.e. MSB value of DISP. Thus, as this instruction CALLS a subroutine in another code segment it is an intersegment direct call.

(ii) If the CALL to a subroutine in another segment is addressed by the contents of a 16 bit register then the 8086 decrements the SP by 2 and pushes CS contents onto the stack.

The CS is then loaded with the contents of memory locations addressed by [reg 16 + 2] and [reg 16 + 3] in DS.

The SP is then again decremented by 2, IP is pushed onto the stack. The IP is then loaded with contents of memory locations addressed by [reg 16] and [reg 16 + 1] in DS.

The registers used as reg 16 are BX, SI, DI.

Such a CALL is called as intersegment indirect call.

e.g. : CALL DWORD, PTR [reg16]

2. RET – Return from Procedure

Mnemonic RET optional – pop – value

Algorithm

- Return from near procedure

  POP from stack : IP

  If immediate operand is present :

  SP = SP + operand
- Return from far procedure

  POP from stack

  IP

  CS

  If immediate operand is present

  SP = SP + operand

Operation

- The RET instruction will return execution from a procedure to the next instruction after CALL instruction.
- If the procedure is a near procedure (i.e. in same code segment as CALL instruction), then the return will be done by replacing the IP with a word from the top of stack. This word from the top of stack is the offset of the next instruction after CALL. The SP will be incremented by 2. After return address is popped off the stack.
- If the procedure is a far procedure (in a different code segment), the instruction pointer will be replaced by the word at the top of stack. The SP will be incremented by 2. The CS is then replaced with a word from new top of stack. After CS word is popped the SP will again incremented by two.
- A RET instruction can be followed by a number.

Example RET 2.

- In this case the SP will be incremented by additional 2 addresses after the IP or IP and CS are popped off the stack. This form is used to increment the stack pointer UP over parameters passed to the procedure on the stack.

3. JMP – Jump

(Unconditional jump to specified destination)

Mnemonic JMP **Flags** No flags are affected.

Algorithm Always jump.

Operation

- This instruction will cause the 8086 to fetch its next instruction from the location specified in the instruction rather than from next location after JMP instruction.
- There are two basic types of JMPs, near and far.
- Near JMP is a jump where destination location is in the same code segment only IP is changed to get destination location. It is known as intrasegment JMP.
- If the destination is in a segment with a different name from the segment containing the JMP instruction, then both the IP and CS contents will be changed to get the destination location. Such a JMP is far JMP. A far JMP is an Inter segment JMP.
- The near and far JMPs are further described as either direct or indirect. If the destination address is specified within instruction. It is a direct JMP, if the destination address is contained in register or memory location, the JMP is indirect, because 8086 has to access the specified register or memory to get the destination address.

Example JMP WORD PTR [BX]

- This instruction will replace IP with a word from memory location pointed by BX in DS.
- This is an indirect near JMP.

12.10.2 Conditional Transfers

Conditional transfer instructions are also referred as conditional jump instructions.

There are total 18 instructions. Refer Table, each test a different combinations of flags, for a condition. If condition is true, then control is transferred to the target specified in the instruction. If the condition is ***false***, then control passes to the instruction that follows the conditional jump. Very important point regarding these instructions is, all conditional jumps are ***SHORT***, that is, the target must be within the current code segment and within – 128 to + 127 bytes of the first byte of the next instruction. Since the jump is made by adding the relative displacement of the target to the instruction pointer, all conditional jumps are self relative and are appropriate for position independent routines. These instructions does not affect any flag.

Common Operations		
JC	JUMP if carry	CF = 1
JNC	JUMP if not carry	CF = 0
JZ/JE	JUMP if zero (or equal)	ZF = 1
JNZ/JNE	JUMP if not zero (or not equal)	ZF = 1
JP/JPE	JUMP if parity (or even parity)	PF = 1
JNP/JPO	JUMP if not parity (or odd parity)	PF = 0
JCXZ	JUMP if CX is zero	CX = 0000H
Signed Operations		
JO	JUMP if overflow	OF = 1
JNO	JUMP if not overflow	OF = 0
JS	JUMP if sign (– ve)	S = 1
JNS	JUMP if not sign (+ ve)	S = 0
JL/JNGE	JUMP if less (i.e. either greater nor equal)	SF ⊕ OF = 1
JNL/JGE	JUMP if not less (i.e. either greater or equal)	SF ⊕ OF = 0
JLE/JNG	JUMP if less or equal (i.e. not greater)	(SF ⊕ OF) + ZF = 1
JNLE/JG	JUMP if neither less nor equal (i.e. greater)	(SF ⊕ OF) + ZF = 0
Unsigned Operations		
JB/JNAE	JUMP if below (i.e. neither above nor equal)	CF = 1
JNB/JAE	JUMP if not below (i.e. either above or equal)	CF = 0
JBE/JNA	JUMP if below or equal (i.e. not above)	CF ⊕ ZF = 1
JNBE/JA	JUMP if neither below nor equal (i.e. above)	CF ⊕ ZF = 0

12.10.3 Iteration Control Instructions

The Instructions in this group are used to regulate the repetition of software loops. Like conditional transfers the iteration control instructions are self relative and may only transfer to targets that are within –128 to +127 bytes of themselves, i.e. they are short transfers.

Iteration Control Instructions
- LOOP
- LOOPZ/LOOPE (Loop if zero / equal)
- LOOPNZ/LOOPNE (Loop if not zero / not equal)

1. LOOP

Mnemonic LOOP short_label

LOOP : Jump to specified lable if CX ≠ 0 after auto decrement

Flags No flags are affected.

Algorithm
CX = CX – 1
If CX < > 0 then jump
else no jump, continue

Operation This instruction is used to repeat a series of instructions some number of times. The number of times the instruction sequence is to be repeated is loaded into CX. Each time loop executes CX is decremented by 1.
- If CX ≠ 0 execution will jump to destination specified by label.
- If CX = 0 execution will go to the next instruction after loop.

2. LOOPZ / LOOPE – Loop if zero / equal

Mnemonic Loop while CX ≠ 0 and ZF = 1
LOOPE short-label/LOOPZ short-label.

Flags No flags are affected

Algorithm
CX = CX – 1
If (CX < > 0) and ZF = 1 then jump
else
no jump, continue.

Operation
- This instruction is used to repeat a group of instructions some number of times or until zero flag becomes zero.
- Number of times of repetition is loaded in CX.

3. LOOPNZ / LOOPNE – Loop if not zero / not equal

Mnemonic Loop while CX ≠ 0 and ZF = 0
LOOPNZ short-label or
LOOPNE short-label

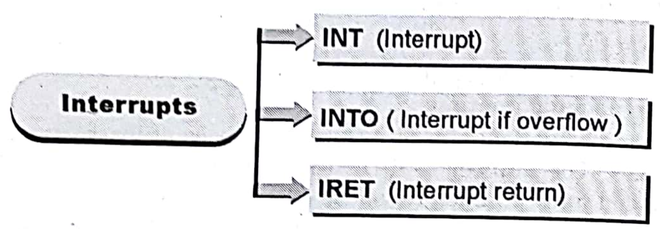
Flags No flags are affected.

Algorithm
- CX = CX – 1
- if (CX < > 0) and
  ZF = 0 then jump
  else
  no jump, continue

Operation
- This instruction is used to repeat a group of instructions some number of times or until zero flag becomes 1.
- Number of times of repetition is loaded in CX

12.10.4 Interrupts

The interrupt instructions allow interrupt service routines to be activated by programs as well as by external hardware device. The effect of software interrupts is similar to hardware-initiated interrupts.



1. INT – Interrupt

Mnemonic INT interrupt – type **Flags** IF = 0 and TF = 0
No other flags are affected

Algorithm
- Push to stack
- flag register
- CS
- IP
- IF = 0, TF = 0
- Transfer control to interrupt procedure.

Operation This instruction causes 8086 to call a far procedure. The term 'type' refers number between 0 to 255 which identifies the interrupt.

When an 8086 executes an INT instruction, it will

(i) Decrement SP by 2 and push flag register on stack.

(ii) Decrement SP by 2 and push CS contents on stack.

(iii) Decrement SP by 2 and push the IP after INT on stack.

(iv) Get a new value for IP from a memory address of 4 times the type specified in instruction.
e.g. For INT 8, the new IP will be read from 00020H.

(v) Get new CS from memory address of 4 times the type specified in instruction plus 2. e.g. For INT 8, new value of CS will be read from 00022H.
e.g. INT 35 : New IP from 008C H,
New CS from : 008EH

(vi) Reset IF and TF

2. INTO – Interrupt if overflow

Mnemonic INTO **Flags** IF = 0 and TF = 0
No other flag is affected.

Algorithm If OF =1 then INT

Operation
- If the overflow flag is set, this instruction will cause the 8086 to do an indirect far call to a procedure you write to handle overflow condition. To do this call the 8086 will read a new value for IP from address 0010 H and CS from 0012 H.

3. IRET – Interrupt return

Mnemonic IRET **Flags** No flags are affected.

Algorithm
- POP from stack
- IP
- CS
- Flag register

Operation The IRET instruction is used at the end of interrupt service routine to return execution to the interrupted program.

The 8086 copies return address from stack into IP, and CS registers and stored value of flags back to flag register.

Note : The RET instruction does not copy flags from the stack back to the flag register

12.11 String Instructions Group

Q. Explain with example string instructions of 8086 microprocessor.

Table 12.11.1 : String instructions

REP	Repeat
REPE/REPZ	Repeat while equal/zero
REPNE/REPNZ	Repeat while not equal/not zero
MOVSB/MOVSW	Move byte or word string
CMPSB/CMPSW	Compare byte or word string
SCASB/CMPSW	Scan byte or word string
LODSB/LODSW	Load byte or word string
STOSB/STOSW	Store byte or word string

1. MOVS/MOVSB/MOVSW : Move string byte or string word

Mnemonic MOVSB/MOVSW **Flags** No flags are affected.

Algorithm In case of byte :

- ES : [DI] = DS : [SI]
- If DF = 0 then

  SI = SI + 1 DI = DI + 1

  else

  SI = SI – 1 DI = DI – 1

In case of word :

- ES : [DI] = DS : [SI]
- If DF = 0 then,

  SI = SI + 2 DI = DI + 2

  Else

  SI = SI – 2 DI = DI – 2

Addr. Mode String addressing mode

Operation

- This instruction copies a byte or a word from a location in the data segment to a location in the extra segment.
- The offset of the source byte/word in the DS must be SI register.
- The offset of the destination in ES must be in DI register.
- After byte or word is moved, SI and DI are automatically adjusted to point to the next source and next destination.

Example

```
LEA SI, SOURCE_STRING   ; SI points to start of source string.
LEA DI, DEST_STRING     ; DI points to start of destination string.
CLD                     ; Clear direction flag to use incrementing
MOVSB                   ; MOV (DI) ← (SI), byte transfer
```

2. CMPS : Compare string byte or string words

Mnemonic CMPSB/CMPSW **Flags** All flags are affected.

Two alternate mnemonics for the same instruction

CMPB, (compare string byte)

CMPW, (compare string word)

Algorithm **For byte operation**

DS : [SI] – ES : [DI]

Set flags according to result → OF, SF, ZF, AF, PF, CF.

If DF = 0 then SI = SI + 1 DI = DI + 1

else SI = SI – 1 DI = DI – 1

For word operation

DS : [SI] – ES : [DI] Set flags, according to result

→ OF, SF, ZF, AF, PF, CF.

If DF = 0 then SI = SI + 2 DI = DI + 2

else SI = SI – 2 DI = DI – 2

Addr. Mode String addressing mode

Operation

- This instruction is used to compare a byte in one string with a byte in another string or to compare a word in one string with another string.
- SI is used to hold the offset of a byte or word in the source string and DI is used to hold the offset of byte or word is another string.
- Comparison is done by subtracting the byte or word pointed by DI from byte or word pointed by SI.
- After comparison SI and DI will be automatically incremented or decremented according to direction flag to point to next element in the string.

Example

```
LEA SI, SRC        ; SI points to start of source string labeled SRC.
LEA DI, DEST       ; DI points to start of destination string labeled DEST
CLD                ; Clear DF (Incrementing is used).
CMPS SRC, DEST     ; Compare.
```

3. SCAS/SCASB/SCASW

(Scan a string Byte or Word)

Mnemonic SCASB/SCASW **Flags** No flags are affected.

SCASB (for byte operation)

Or SCASW (for word operation)

Algorithm For SCASB

- AL – ES : [DI]
- Set flags according to result
- If DF = 0 then DI = DI + 1

For SCASW

- AX – ES : [DI]
- Set flags according to result
- If DF = 0 then DI = DI + 2

  Else DI = DI – 2

Addr. Mode String addressing mode

Operation

- This instruction compares a byte in AL or word in AX with a byte or word pointed to by DI in ES. Hence the string to be scanned must be in ES, and offset in DI.
- After comparison DI will be automatically incremented or decremented according to direction flag DF.

Example

```
Let BUFF ← 0EH
LEA DI, BUFF      ; Load offset Buffer in DI
MOV AL, 0DH       ; AL is loaded with the Byte to be compared. i.e. 0DH
                  ; it is ASCII for carriage return it indicates line
                  ; termination.
CLD               ; Clear direction flag DF to use incrementing
SCASB             ; Scan for byte in the string.
```

4. LODS

(Load string byte in AL or string word into AX.)

Mnemonic LODSB/LODSW **Flags** No flags are affected.

Or LODSB (For byte operation)

Or LODSW (For word operation)

Algorithm LODSB

- AL = DS : [SI]

If DF = 0 then SI = SI + 1 else SI = SI – 1

LODSW

- AX = DS : [SI]

If DF = 0 then SI = SI + 2 else SI = SI – 2

Addr. Mode String addressing mode

Operation This instruction copies a byte from a string location pointed to by SI to AX.

Example

```
CLD
MOV SI, OFFSET A1 : SI points to start of A1.
LODSB : Load byte in AL from the string
```

5. STOS/STOSB/STOSW

(Store byte or word in string)

Mnemonic STOSB/STOSW **Flags** No flags are affected

STOSB (For byte operation)

or STOSW (for word operation)

Algorithm For STOSB

- ES : [DI] = AL

If DF = 0 then DI = DI + 1 else DI = DI – 1

For STOSW

- ES : [DI] = AX

If DF = 0 then DI = DI + 2 else DI = DI – 2

Addr. Mode String addressing mode

Operation It transfers (copies) a byte or word from register AL or AX to the string element addressed in ES by DI. Depending on DF flag, DI is automatically incremented or decremented.

Example

```
MOV DI, OFFSET STR 1      ; DI points to top of string STR 1.
CLD                       ; DF = 0
MOV AX, 00H
STOSW                     ; Store string by AX value
```

6. REP/REPE/REPNE/REPNZ : (Prefix)

(Repeat string instruction until specified conditions exist).

Mnemonic REP/REPE/REPNE/REPNZ **Flags** –

Algorithm Check_CX For REP

- CX = CX – 1
- Repeat the instruction to which it is a prefix
- go back to check _ CX

else

- exit from REP cycle

Operation

- REP is a prefix written before one of the string instructions. These instructions repeat until specified condition exists.

Instruction code	Condition for exit
REP	CX = 0
REPE/REPZ	CX = 0 or ZF = 0
REPNE/REPNZ	CX = 0 or ZF = 1

A. REPE/REPZ (Repeat while equal / Repeat while zero).

Mnemonic REPE/REPZ

Algorithm Check CX

if CX < > 0 then

CX = CX – 1

if ZF = 1 then

repeat the instruction to which it is prefix

go back to check CX

else

exit from REPE cycle

else

exit from REPE cycle.

Operation

- These are two mnemonics for the same instruction.
- This prefix will cause the string instructions to be repeated as long as ZF = 1 and CX ≠ 0.

If ZF = 0 or CX = 0, string instructions will not be repeated.

Example

```
LEA DI, STR 1     ; DI = offset of STR 1
MOV AL, 05 H      ; AL = 05 H
CLD               ; DF = 0
MOV CX, 35 H      ; CX = 35 H i.e. counter = 35 H
REPE SCASB        ; Repeat, scan string byte operation till
                  ; CX ≠ 0 and ZF = 1.
```

B. REPNE/REPNZ (Repeat while not equal / Repeat while not zero)

Mnemonic REPNE/REPNZ

Algorithm

Check CX
if CX < > 0 then CX = CX – 1
If ZF = 0 then
repeat the instruction to which it is prefix
go back to check CX
else exit from REPNE cycle
else exit from REPNE cycle.

Operation

- These are two mnemonics for the same instruction.
- This will cause string instructions to be repeated as long as ZF = 0 and CX ≠ 0.
- If ZF = 1 or CX = 0, string instructions will not be repeated.

□□□

CHAPTER

13

Assembly Language Programming

13.1 Introduction to Assembly Language

Microprocessor chip designers create a basic set of instructions for every processor they design. These are really simplistic instructions that take baby steps as compared with high-level languages such as C++.

- Each instruction has a code so that the instruction decoder can decode them in order to energize the proper circuits to execute the function using the registers of the processor. These are referred to as ***operation code*** (OPCODE). *"Machine code is the instruction set that machine understands".*
- Machine language is the only language understood directly by the CPU in our computers.
- Humans, however, understand words, so each machine code is given an "English" equivalent. *These instructions in text form are called* ***Mnemonic***.
- Assembly language is a mnemonic representation of machine code.
- Assembly language is not just a simple mapping of numbers to words. It also contains many high-level-language type constructs to make data definition and program structuring easier.
- There is a one-to-one correlation between assembly language instructions and the machine code.

Reasons for learning the assembly language

The assembly language is learnt for the following reasons :

(1) It helps to learn about the internal architecture of the computer. The more you use the assembly language the more knowledge you gain about the architecture.

(2) So as to make use of the utilities of a computer e.g. to directly communicate with the microprocessor of the computer. This is useful under the circumstances where programming in high level language is difficult or even impossible.

(3) Most of the times a high level language program written for the above applications does not yield the required performance. So use of assembly level programming is always a better solution.

(4) The high level language programs may need a large memory space. Instead the memory requirement will reduce if we use the assembly language programming.