CONTENTS

UNIT – I

THE 8086 MICROPROCESSOR

1.1 INTRODUCTION

· It is a semiconductor device consisting of electronic logic circuits manufactured by using either a Large scale (LSI) or Very Large Scale (VLSI) Integration Technique.

· It includes the ALU, register arrays and control circuits on a single chip. The microprocessor has a set of instructions, designed internally, to manipulate data and communicate with peripherals.

· The era microprocessors in the year 1971, the Intel introduced the first 4-bit microprocessor is 4004. Using this the first portable calculator is designed.

· The 16-bit Microprocessor families are designed primarily to complete with microcomputers and are oriented towards high-level languages. They have powerful instruction sets and capable of addressing mega bytes of memory.

· The era of 16-bit Microprocessors began in 1974 with the introduction of PACE chip by National Semiconductor. The Texas Instruments TMS9900 was introduced in the year 1976. The Intel 8086 commercially available in the year 1978, Zilog Z800 in the year 1979, The Motorola MC68000 in the year 1980.

· The 16-bit Microprocessors are available in different pin packages. Ex: Intel 8086/8088 40 pin package Zilog Z8001 40 pin package, Digital equipment LSI-II 40 pin package, Motorola MC68000 64 pin package National Semiconductor NS16000 48 pin package.

· The primary objectives of this 16-bit Microprocessor can be summarized as follows.
  1. Increase memory addressing capability
  2. Increase execution speed
  3. Provide a powerful instruction set
  4. Facilitate programming in high-level languages.

1.2 Microprocessor Architecture:

· The 8086 CPU is divided into two independent functional parts, the Bus interface unit (BIU) and execution unit (EU).

  The Bus Interface Unit contains Bus Interface Logic, Segment registers, Memory addressing logic and a Six byte instruction object code queue. The BIU sends out address, fetches the instructions from memory, read data from ports and memory, and writes the data to ports and memory.

· **The execution unit:** contains the Data and Address registers, the Arithmetic and Logic Unit, the Control Unit and flags. tells the BIU where to fetch instructions or data from, decodes instructions and executes instruction. The EU contains control circuitry which directs internal operations. A decoder in the EU translates instructions fetched from memory into a series of actions which the EU carries out. The EU is has a 16-bit ALU which can add, subtract, AND, OR, XOR, increment, decrement, complement or shift binary numbers. The EU is decoding an instruction or executing an instruction which does not require use of the buses.

  In other words the BIU handles all transfers of data and addresses on the buses for the execution unit.

· **The Queue**: The BIU fetches up to 6 instruction bytes for the following instructions. The BIU stores these prefetched bytes in first-in-first-out register set called a queue. When the EU is ready for its next instruction it simply reads the instruction byte(s) for the instruction from the queue in the BIU. This is much faster than sending out an address to the system memory and waiting for memory to send back the next instruction byte or bytes.
  .

1

Address/status
$A_{16}/S_3 - A_{19}/S_6$

Address/data
$AD_0 - AD_{15}$

Memory address and data
bus interface

Internal data bus

BUS INTERFACE UNIT

Address conversion
mechanism (adder)

CS
DS
SS
ES

IP

Instruction byte
queue 6 bytes

Internal data bus

EXECUTION UNIT

15   0

AX   AH   AL
BX   BH   BL
CX   CH   CL
DX   DH   DL
        SP
        BP
        SI
        DI

Register bank

ALU (16)

Flags (16)

Decoding
circuit

Timing and control
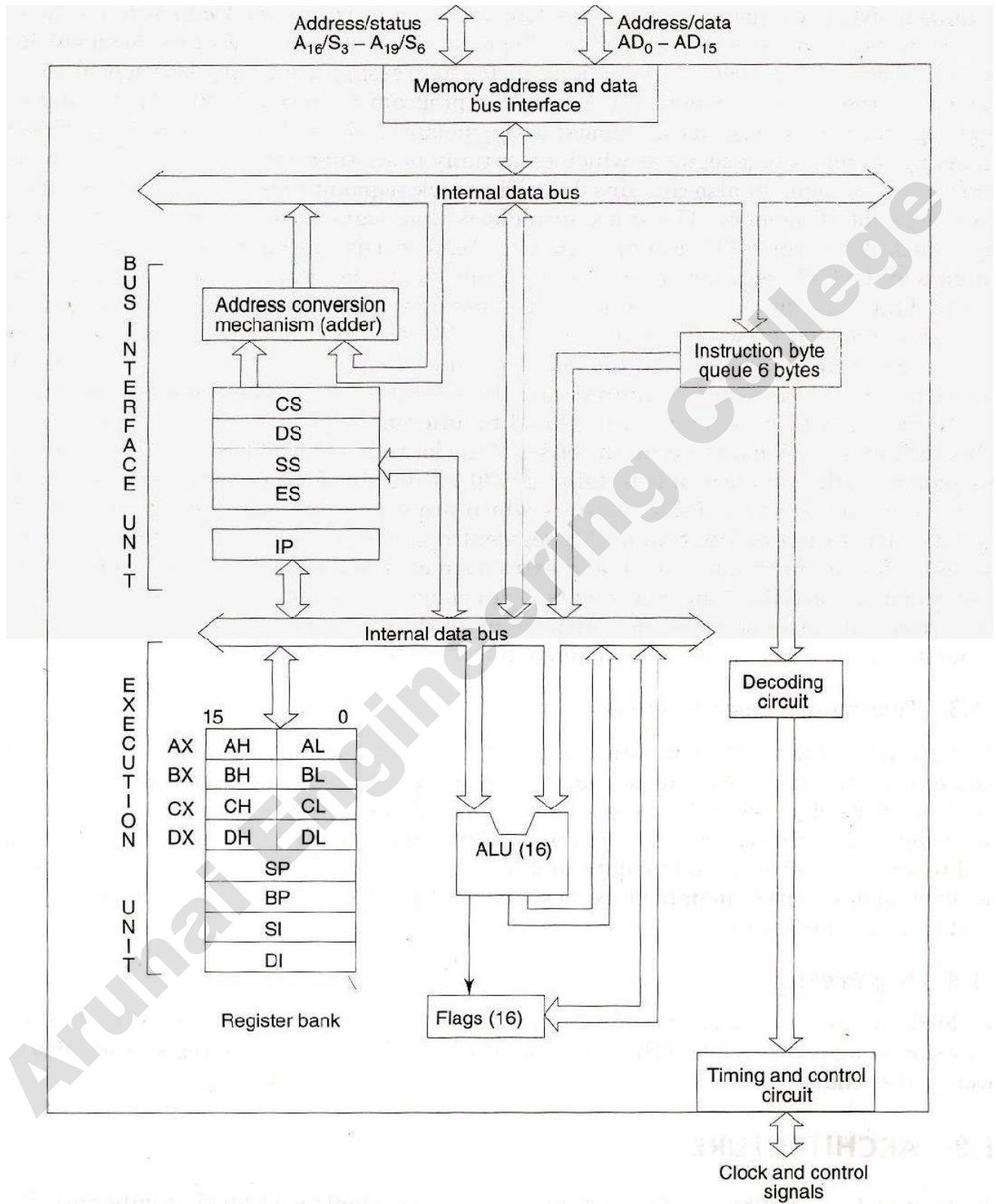circuit

Clock and control
signals

Fig.1.1 8086 Architecture

2

Except in the case of JMP and CALL instructions, where the queue must be dumped and then reloaded starting from a new address, this prefetch-and-queue scheme greatly speeds up processing. Fetching the next instruction while the current instruction executes is called pipelining.

· **Word Read:** Each of 1 MB memory address of 8086 represents a byte wide location.16-bit

words will be stored in two consecutive memory locations. If first byte of the data is stored at an even address, 8086 can read the entire word in one operation.

For example if the 16 bit data is stored at even address 00520H is 9634H

    MOV BX, [00520H]

8086 reads the first byte and stores the data in BL and reads the 2nd byte and stores the data in BH

    BL= (00520H) i.e. BL=34H

    BH= (00521H) BH=96H

If the first byte of the data is stored at an odd address, 8086 needs two operations to read the 16 bit data.

For example if the 16 bit data is stored at even address 00521H is 3897H

    MOV BX, [00521H]

In first operation, 8086 reads the 16 bit data from the 00520H location and stores the data of 00521H location in register BL and discards the data of 00520H location In 2nd operation, 8086 reads the 16 bit data from the 00522H location and stores the data of 00522H location in register BH and discards the data of 00523H location.

    BL= (00521H) i.e. BL=97H

    BH= (00522H) BH=38H

· **Byte Read**:

MOV BH, [Addr]

For Even Address:

Ex: MOV BH, [00520H]

8086 reads the first byte from 00520 location and stores the data in BH and reads the 2nd byte from the 00521H location and ignores it

BH =[00520H]

For Odd Address

MOV BH, [Addr]

Ex: MOV BH, [00521H]

8086 reads the first byte from 00520H location and ignores it and reads the 2nd byte from the 00521 location and stores the data in BH

BH = [00521H]

· **Physical address formation**:

The 8086 addresses a segmented memory. The complete physical address which is 20-bits long is generated using segment and offset registers each of the size 16-bit.The content of a segment register also called as segment address, and content of an offset register also called as offset address. To get total physical address, put the lower nibble 0H to segment address and add offset address. The fig 1.3 shows formation of 20-bit physical address.
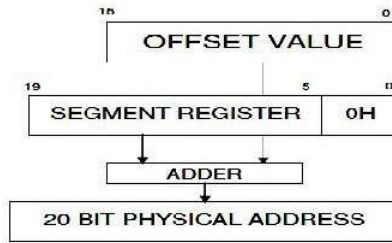
Fig 1.2 Physical Address formation

· **Register organization of 8086:**

All the registers of 8086 are 16-bit registers. The general purpose registers, can be used either 8-bit registers or 16-bit registers used for holding the data, variables and intermediate results temporarily or for other purpose like counter or for storing offset address for some particular addressing modes etc. The special purpose registers are used as segment registers, pointers, index registers or as offset storage registers for particular addressing modes. Fig 1.3
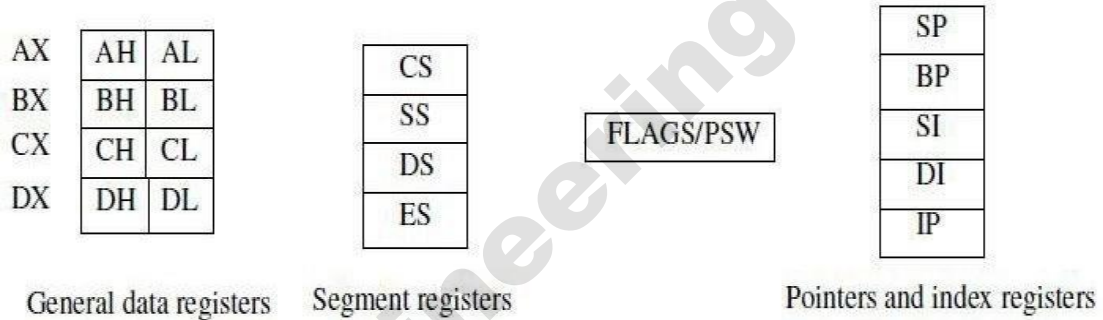

Fig 1.3 Register organization of 8086

✓ **AX Register: Accumulator** register consists of two 8-bit registers AL and AH, which can be combined together and used as a 16- bit register AX. AL in this case contains the low-order byte of the word, and AH contains the high-order byte. Accumulator can be used for I/O operations, rotate and string manipulation.

✓ **BX Register:** This register is mainly used as a **base register**. It holds the starting base location of a memory region within a data segment. It is used as offset storage for forming physical address in case of certain addressing mode.

✓ **CX Register:** It is used as default counter - **count register** in case of string and loop instructions.

✓ **DX Register: Data register** can be used as a port number in I/O operations and implicit operand or destination in case of few instructions. In integer 32-bit multiply and divide instruction the DX register contains high-order word of the initial or resulting number.

Segment registers:

1Mbyte memory is divided into 16 logical segments. The complete 1Mbyte memory segmentation is as shown in fig 1.4. Each segment contains 64Kbyte of memory. There are four segment registers.

✓ **Code segment** (CS) is a 16-bit register containing address of 64 KB segment with processor instructions. The processor uses CS segment for all accesses to instructions referenced by instruction pointer (IP) register. CS register cannot be changed directly.

The CS register is automatically updated during far jump, far call and far return instructions. It is used for addressing a memory location in the code segment of the memory, where the executable program is stored.

- ✓ **Stack segment** (SS) is a 16-bit register containing address of 64KB segment with program stack. By default, the processor assumes that all data referenced by the stack pointer (SP) and base pointer (BP) registers is located in the stack segment. SS register can be changed directly using POP instruction. It is used for addressing stack segment of memory. The stack segment is that segment of memory, which is used to store stack data.
- ✓ **Data segment** (DS) is a 16-bit register containing address of 64KB segment with program data. By default, the processor assumes that all data referenced by general registers (AX, BX, CX, DX) and index register (SI, DI) is located in the data segment. DS register can be changed directly using POP and LDS instructions. It points to the data segment memory where the data is resided.
- ✓ **Extra segment** (ES) is a 16-bit register containing address of 64KB segment, usually with program data. By default, the processor assumes that the DI register references the ES segment in string manipulation instructions. ES register can be changed directly using POP and LES instructions. It also refers to segment which essentially is another data segment of the memory.
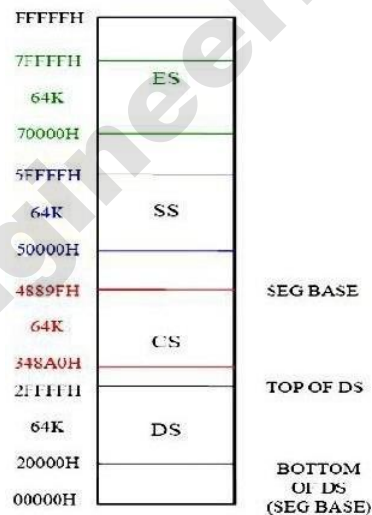
- ✓ It also contains data.

Fig1.4. Memory segmentation

- ✓ **Pointers and index registers.**
  The pointers contain within the particular segments. The pointers IP, BP, SP usually contain offsets within the code, data and stack segments respectively

  Stack Pointer (SP) is a 16-bit register pointing to program stack in stack segment.

  Base Pointer (BP) is a 16-bit register pointing to data in stack segment. BP register is usually used for based, based indexed or register indirect addressing.

  Source Index (SI) is a 16-bit register. SI is used for indexed, based indexed and register indirect addressing, as well as a source data addresses in string manipulation instructions.

5

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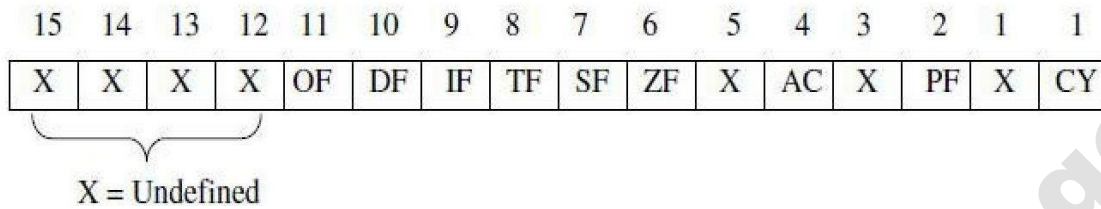
✓ **Flag Register:**

Fig. 1.5 Flag Register

Flags Register determines the current state of the processor. They are modified automatically by CPU after mathematical operations, this allows to determine the type of the result, and to determine conditions to transfer control to other parts of the program. The 8086 flag register as shown in the fig 1.5. 8086 has 9 active flags and they are divided into two categories:

1. Conditional Flags

2. Control Flags

✓ **Conditional Flags**

Carry Flag (CY): This flag indicates an overflow condition for unsigned integer arithmetic. It is also used in multiple-precision arithmetic.

Auxiliary Flag (AC): If an operation performed in ALU generates a carry/barrow from lower nibble (i.e. $D_0 – D_3$) to upper nibble (i.e. $D_4 – D_7$), the AC flag is set i.e. carry given by D_3 bit to D_4 is AC flag. This is not a general-purpose flag, it is used internally by the Processor to perform Binary to BCD conversion.

Parity Flag (PF): This flag is used to indicate the parity of result. If lower order 8-bits of the result contains even number of 1's, the Parity Flag is set and for odd number of 1's, the Parity flag is reset.

Zero Flag (ZF): It is set; if the result of arithmetic or logical operation is zero else it is reset.

Sign Flag (SF): In sign magnitude format the sign of number is indicated by MSB bit. If the result of operation is negative, sign flag is set.

✓ **Control Flags**

Control flags are set or reset deliberately to control the operations of the execution unit.

Control flags are as follows:

Trap Flag (TF): It is used for single step control. It allows user to execute one instruction of a program at a time for debugging. When trap flag is set, program can be run in single step mode.

Interrupt Flag (IF): It is an interrupt enable/disable flag. If it is set, the maskable interrupt of 8086 is enabled and if it is reset, the interrupt is disabled. It can be set by executing instruction sit and can be cleared by executing CLI instruction.

Direction Flag (DF): It is used in string operation. If it is set, string bytes are accessed from higher memory address to lower memory address. When it is reset, the string bytes are accessed from lower memory address to higher memory address.

1.3 Addressing Modes

The 8086 has 12 addressing modes can be classified into five groups.

· Addressing modes for accessing immediate and register data (register and immediate modes).

· Addressing modes for accessing data in memory (memory modes)

6

· Addressing modes for accessing I/O ports (I/O modes)
· Relative addressing mode
· Implied addressing mode

✓ **Immediate addressing mode:**
In this mode, 8 or 16 bit data can be specified as part of the instruction - OP Code Immediate Operand
Example 1: MOV CL, 03 H:Moves the 8 bit data 03 H into CL
Example 2: MOV DX, 0525 H: Moves the 16 bit data 0525 H into DX

In the above two examples, the source operand is in immediate mode and the destination operand is in register mode.
A constant such as "VALUE" can be defined by the assembler EQUATE directive such as VALUE EQU 35H
　　　Example: MOV BH, VALUE   Used to load 35 H into BH

✓ **Register addressing mode:**
The operand to be accessed is specified as residing in an internal register of 8086. Table 1.1 below shows internal registers, anyone can be used as a source or destination operand, however only the data registers can be accessed as either a byte or word.

Table 1.1 Internal registers of 8086

Register	Operand sizes	
	Byte (Reg 8)	Word (Reg 16)
Accumulator	AL, AH	Ax
Base	BL, BH	Bx
Count	CL, CH	Cx
Data	DL, DH	Dx
Stack pointer	-	SP
Base pointer	-	BP
Source index	-	SI
Destination index	-	DI
Code Segment	-	CS
Data Segment	-	DS
Stack Segment	-	SS
Extra Segment	-	ES

Example 1: MOV DX (Destination Register) , CX (Source Register)
Which moves 16 bit content of CS into DX.
Example 2: MOV CL, DL
Moves 8 bit contents of DL into CL
MOV BX, CH is an illegal instruction.
* The register sizes must be the same.

✓ **Direct addressing mode:**
The instruction Opcode is followed by an affective address, this effective address is directly used as the 16 bit offset of the storage location of the operand from the location specified by the current value in the selected segment register. The default segment is always DS.The 20 bit physical address of the operand in memory is normally obtained as PA = DS: EA

But by using a segment override prefix (SOP) in the instruction, any of the four segment registers can be referenced,

$$PA = \left\{ \begin{array}{l} CS \\ DS \\ SS \\ ES \end{array} \right\} : \left\{ Direct\ Address \right\}$$

Fig 1.6 Physical address generation of 8086

The Execution Unit (EU) has direct access to all registers and data for register and immediate operands. However the EU cannot directly access the memory operands. It must use the BIU, in order to access memory operands.

In the direct addressing mode, the 16 bit effective address (EA) is taken directly from the displacement field of the instruction.

Example 1: MOV CX, START

If the 16 bit value assigned to the offset START by the programmer using an assembler pseudo instruction such as DW is 0040 and [DS] = 3050. Then BIU generates the 20 bit physical address 30540 H.

The content of 30540 is moved to CL

The content of 30541 is moved to CH

Example 2: MOV CH, START

If [DS] = 3050 and START = 0040

8 bit content of memory location 30540 is moved to CH.

Example 3: MOV START, BX

With [DS] = 3050, the value of START is 0040.

Physical address: 30540

MOV instruction moves (BL) and (BH) to locations 30540 and 30541 respectively.

✓ **Register indirect addressing mode:**

The EA is specified in either pointer (BX) register or an index (SI or DI) register. The 20 bit physical address is computed using DS and EA.

Example: MOV [DI], BX register indirect

If [DS] = 5004, [DI] = 0020, [Bx] = 2456 PA=50060.

The content of BX(2456) is moved to memory locations 50060 H and 50061 H.

$$PA = \left\{ \begin{array}{l} CS \\ DS \\ SS \\ ES \end{array} \right\} = \left\{ \begin{array}{l} BX \\ SI \\ DI \end{array} \right\}$$

Based addressing mode:

$$PA = \left\{ \begin{array}{l} CS \\ DS \\ SS \\ ES \end{array} \right\} : \left\{ \begin{array}{l} BX \\ or \\ BP \end{array} \right\} + displacement$$

when memory is accessed PA is computed from BX and DS when the stack is accessed PA is computed from BP and SS.

Example: MOV AL, START [BX]

or

MOV AL, [START + BX]

based mode

8

EA: [START] + [BX]

PA: [DS] + [EA]

The 8 bit content of this memory location is moved to AL.

✓ **String addressing mode:**

The string instructions automatically assume SI to point to the first byte or word of the source operand and DI to point to the first byte or word of the destination operand. The contents of SI and DI are automatically incremented (by clearing DF to 0 by CLD instruction) to point to the next byte or word.

Example: MOV S BYTE

    If [DF] = 0, [DS] = 2000 H, [SI] = 0500,

    [ES] = 4000, [DI] = 0300

    Source address: 20500, assume it contains 38

    PA: [DS] + [SI]

    Destination address: [ES] + [DI] = 40300, assume it contains 45

Indexed addressing mode:

$$PA = \begin{Bmatrix} CS \\ DS \\ SS \\ ES \end{Bmatrix} : \begin{Bmatrix} SI \\ or \\ DI \end{Bmatrix} + 8 \text{ or } 16\text{bit displacement}$$

Example : MOV BH, START [SI]

    PA : [SART] + [SI] + [DS]

The content of this memory is moved into BH.

Based Indexed addressing mode:

$$PA = \begin{Bmatrix} CS \\ DS \\ SS \\ ES \end{Bmatrix} : \begin{Bmatrix} BX \\ or \\ BP \end{Bmatrix} + \begin{Bmatrix} SI \\ or \\ DI \end{Bmatrix} + 8 \text{ or } 16\text{bit displacement}$$

Example : MOV ALPHA [SI] [BX], CL

    If [BX] = 0200, ALPHA – 08, [SI] = 1000 H and [DS] = 3000

Physical address (PA) = 31208

8 bit content of CL is moved to 31208 memory address.

After executing MOV S BYTE,

    [40300] = 38

    [SI] = 0501   incremented

    [DI] = 0301

✓ **I/O mode (direct):**

Port number is an 8 bit immediate operand.

Example: OUT 05 H, AL

Outputs [AL] to 8 bit port 05 H

I/O mode (indirect):

The port number is taken from DX.

Example 1: IN AL, DX

If [DX] = 5040

8 bit content by port 5040 is moved into AL.

Example 2: IN AX, DX

Inputs 8 bit content of ports 5040 and 5041 into AL and AH respectively.

✓ **Relative addressing mode:**

Example: JNC START

If CY=O, then PC is loaded with current PC contents plus 8 bit signed value of START,otherwise the next instruction is executed.

✓ **Implied addressing mode:**

Instruction using this mode have no operands.

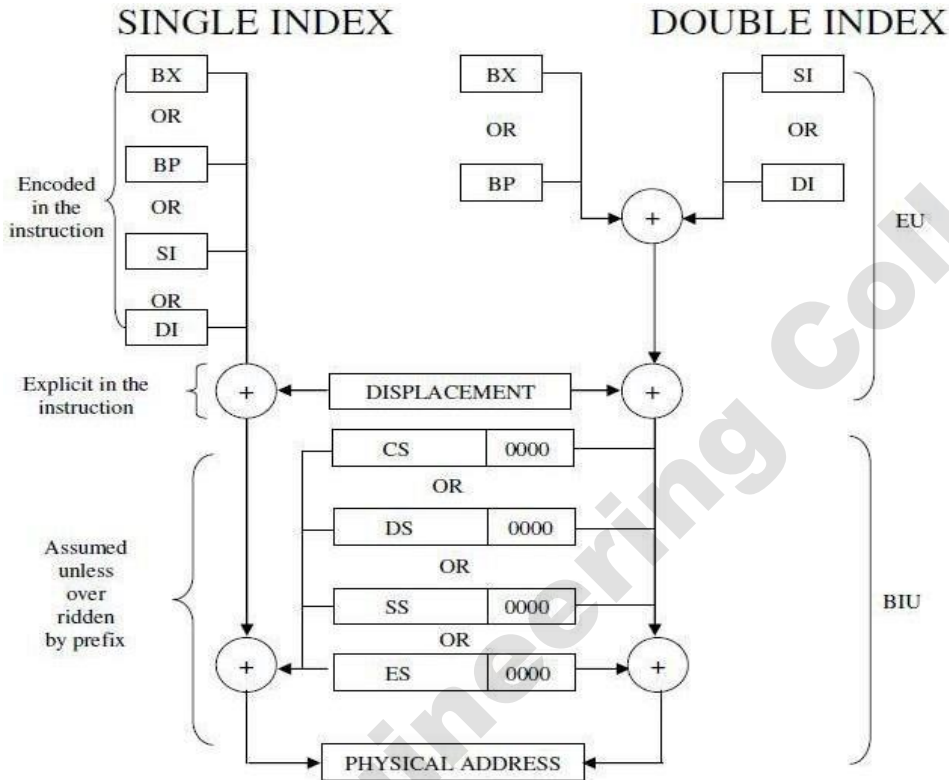Example: CLC which clears carry flag to zero.

Fig 1.7 Summary of 8086 addressing modes

INSTRUCTION SET OF 8086

The 8086 instructions are categorized into the following main types.

1. Data Copy / Transfer Instructions
2. Arithmetic and Logical Instructions
3. Shift and Rotate Instructions
4. Loop Instructions
5. Branch Instructions
6. String Instructions
7. Flag Manipulation Instructions
8. Machine Control Instructions

Data Copy / Transfer Instructions:MOV:

This instruction copies a word or a byte of data from some source to a destination. The destination can be a register or a memory location. The source can be a register, amemory location, or an immediate number.

MOV AX, BX
MOV AX, 5000H
MOV AX, [SI]
MOV AX, [2000H]
MOV AX, 50H[BX]
MOV [734AH], BX
MOV DS, CX
MOV CL, [357AH]

Direct loading of the segment registers with immediate data is not permitted.

PUSH: Push to Stack

This instruction pushes the contents of the specified register/memory location on to the stack. The stack pointer is decremented by 2, after each execution of the instruction.

     E.g. PUSH AX
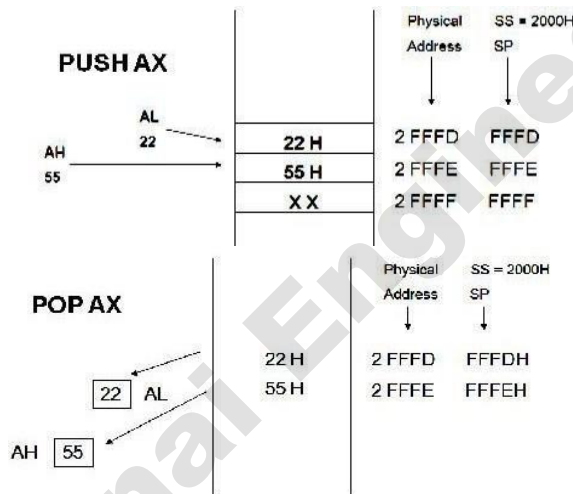   • PUSH DS
     • PUSH [5000H]

POP: Pop from Stack

This instruction when executed, loads the specified register/memory location withthe contents of the memory location of which the address is formed using the current stack segment and stack pointer.The stack pointer is incremented by 2

Eg. POP AX
   POP DS
   POP [5000H]

Fig 1.8 Push into and Popping Register Content from Stack Memory

XCHG: Exchange byte or word
This instruction exchange the contents of the specified source and destination operands
Eg. XCHG [5000H], AX

XCHG BX, AX

XLAT:

Translate byte using look-up table

Eg. LEA BX, TABLE1

   MOV AL, 04H

    XLAT

Input and output port transfer instructions:

IN:

Copy a byte or word from specified port to accumulator.

Eg. IN AL,03H

IN AX,DX

OUT:

Copy a byte or word from accumulator specified port.

Eg. OUT 03H, AL

OUT DX, AX

LEA:

Load effective address of operand in specified register.

[reg] offset portion of address in DS

Eg. LEA reg, offset

LDS:

Load DS register and other specified register from memory.

[reg] [mem]

[DS] [mem + 2]

Eg. LDS reg, mem

LES:

Load ES register and other specified register from memory.

[reg] [mem]

[ES] [mem + 2]

Eg. LES reg, mem

Flag transfer instructions:

LAHF:

Load (copy to) AH with the low byte the flag register.

[AH] [Flags low byte]

Eg. LAHF

SAHF:

Store (copy) AH register to low byte of flag register.

[Flags low byte] [AH]

Eg. SAHF

PUSHF:

Copy flag register to top of stack.

[SP] [SP] – 2

[[SP]] [Flags]

Eg. PUSHF

POPF:

Copy word at top of stack to flag register.

[Flags] [[SP]]

[SP] [SP] + 2

Arithmetic Instructions:

The 8086 provides many arithmetic operations: addition, subtraction, negation, multiplication and comparing two values.

ADD:

The add instruction adds the contents of the source operand to the destination operand.

Eg. ADD AX, 0100H

ADD AX, BX

ADD AX, [SI]

ADD AX, [5000H]

ADD [5000H], 0100H

ADD 0100H

ADC: Add with Carry

This instruction performs the same operation as ADD instruction, but adds the carry flag to the result.

Eg. ADC 0100H

ADC AX, BX

ADC AX, [SI]

ADC AX, [5000]

ADC [5000], 0100H

SUB: Subtract

The subtract instruction subtracts the source operand from the destination operand and the result is left in the destination operand.

Eg. SUB AX, 0100H

SUB AX, BX

SUB AX, [5000H]

SUB [5000H], 0100H

SBB: Subtract with Borrow

The subtract with borrow instruction subtracts the source operand and the borrow flag (CF) which may reflect the result of the previous calculations, from the destination operand

Eg. SBB AX, 0100H

SBB AX, BX

SBB AX, [5000H]

SBB [5000H], 0100H

INC: Increment

This instruction increases the contents of the specified Register or memory location by 1. Immediate data cannot be operand of this instruction.

Eg. INC AX

INC [BX]

INC [5000H]

DEC: Decrement

The decrement instruction subtracts 1 from the contents of the specified register or memory location.

Eg. DEC AX

DEC [5000H]

NEG: Negate

The negate instruction forms 2's complement of the specified destination in the instruction. The destination can be a register or a memory location. This instruction can be implemented by inverting each bit and adding 1 to it.

Eg. NEG AL

AL = 0011 0101 35H Replace number in AL with its 2's complement

AL = 1100 1011 = CBH

13

CMP: Compare

This instruction compares the source operand, which may be a register or an immediate data or a memory location, with a destination operand that may be a register or a memory location

Eg. CMP BX, 0100H

CMP AX, 0100H

CMP [5000H], 0100H

CMP BX, [SI]

CMP BX, CX

MUL:Unsigned Multiplication Byte or Word

This instruction multiplies an unsigned byte or word by the contents of AL.

Eg. MUL BH; (AX) (AL) x (BH)

MUL CX; (DX)(AX) (AX) x (CX)

MUL WORD PTR [SI]; (DX)(AX) (AX) x ([SI])

IMUL:Signed Multiplication

This instruction multiplies a signed byte in source operand by a signed byte in AL or a signed word in source operand by a signed word in AX.

Eg. IMUL BH

IMUL CX

IMUL [SI]

CBW: Convert Signed Byte to Word

This instruction copies the sign of a byte in AL to all the bits in AH. AH is then said to be sign extension of AL.

Eg. CBW

AX= 0000 0000 1001 1000 Convert signed byte in AL signed word in AX. Result in AX = 1111 1111 1001 1000

CWD: Convert Signed Word to Double Word

This instruction copies the sign of a byte in AL to all the bits in AH. AH is then said to be sign extension of AL.

Eg. CWD

Convert signed word in AX to signed double word in DX: AX

DX= 1111 1111 1111 1111

Result in AX = 1111 0000 1100 0001

DIV: Unsigned division

This instruction is used to divide an unsigned word by a byte or to divide an unsigned double word by a word.

Eg. DIV CL; Word in AX / byte in CL; Quotient in AL, remainder in AH

DIV CX; Double word in DX and AX / word; in CX, and Quotient in AX; remainder in DX

AAA: ASCII Adjust After Addition

The AAA instruction is executed after an ADD instruction that adds two ASCII coded operand to give a byte of result in AL. The AAA instruction converts the resulting contents of Al to a unpacked decimal digits.

Eg. ADD CL, DL; [CL] = 32H = ASCII for 2; [DL] = 35H = ASCII for 5; Result [CL] = 67H

MOV AL, CL; Move ASCII result into AL since; AAA adjust only [AL]

AAA; [AL]=07, unpacked BCD for 7

AAS: ASCII Adjust AL after Subtraction

This instruction corrects the result in AL register after subtracting two unpacked ASCII operands. The result is in unpacked decimal format. The procedure is similar to AAA instruction except for the subtraction of 06 from AL.

AAM: ASCII Adjust after Multiplication

This instruction, after execution, converts the product available In AL into unpacked BCD format.

Eg. MOV AL, 04; AL = 04

MOV BL ,09; BL = 09

MUL BL; AX = AL*BL; AX=24H

AAM; AH = 03, AL=06

AAD: ASCII Adjust before Division

This instruction 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. In the instruction sequence, this instruction appears Before DIV instruction.

Eg. AX 05 08

AAD result in AL 00 3A 58D = 3A H in AL

The result of AAD execution will give the hexadecimal number 3A in AL and 00 in AH where 3A is the hexadecimal Equivalent of 58 (decimal).

DAA: Decimal Adjust Accumulator

This instruction is used to convert the result of the addition of two packed BCD numbers to a valid BCD number. The result has to be only in AL.

Eg. AL = 53 CL = 29

ADD AL, CL; AL (AL) + (CL); AL 53 + 29;

AL 7C

DAA; AL 7C + 06 (as C>9); AL 82

DAS: Decimal Adjust after Subtraction

This instruction converts the result of the subtraction of two packed BCD numbers to a valid BCD number. The subtraction has to be in AL only.

Eg. AL = 75, BH = 46

SUB AL, BH; AL 2 F = (AL) - (BH)

; AF = 1

DAS; AL 2 9 (as F>9, F - 6 = 9)

Logical instructions

AND: Logical AND

This instruction bit by bit ANDs the source operand that may be an immediate register or a memory location to the destination operand that may a register or a memory location. The result is stored in the destination operand.

Eg. AND AX, 0008H

AND AX, BX

OR: Logical OR

This instruction bit by bit ORs the source operand that may be an immediate, register or a memory location to the destination operand that may a register or a memory location. The result is stored in the destination operand.

Eg. OR AX, 0008H

OR AX, BX

NOT: Logical Invert

This instruction complements the contents of an operand register or a memory location, bit by bit.

Eg. NOT AX

NOT [5000H]

OR: Logical Exclusive OR

This instruction bit by bit XORs the source operand that may be an immediate, register or a memory location to the destination operand that may a register or a memory location. The result is stored in the destination operand.

Eg. XOR AX, 0098H

XOR AX, BX

TEST: Logical Compare Instruction

The TEST instruction performs a bit by bit logical AND operation on the two operands. The result of this ANDing operation is not available for further use, but flags are affected.

Eg. TEST AX, BX

TEST [0500], 06H

1.4. 3 Shift and Rotate Instructions

SAL/SHL: SAL / SHL destination, count.

SAL and SHL are two mnemonics for the same instruction. This instruction shifts each bit in the specified destination to the left and 0 is stored at LSB position. The MSB is shifted into the carry flag. The destination can be a byte or a word. It can be in a register or in a memory location. The number of shifts is indicated by count.

Eg. SAL CX, 1

SAL AX, CL

SHR: SHR destination, count

This instruction shifts each bit in the specified destination to the right and 0 is stored at MSB position. The LSB is shifted into the carry flag. The destination can be a byte or a word.

It can be a register or in a memory location. The number of shifts is indicated by count.

Eg. SHR CX, 1

MOV CL, 05H

SHR AX, CL

SAR: SAR destination, count

This instruction shifts each bit in the specified destination some number of bit positions to the right. As a bit is shifted out of the MSB position, a copy of the old MSB is put in the MSB position. The LSB will be shifted into CF.

Eg. SAR BL, 1

MOV CL, 04H

SAR DX, CL

ROL Instruction: ROL destination, count

This instruction rotates all bits in a specified byte or word to the left some number of bit positions. MSB is placed as a new LSB and a new CF.

Eg. ROL CX, 1

MOV CL, 03H

ROL BL, CL

ROR Instruction: ROR destination, count

This instruction rotates all bits in a specified byte or word to the right some number of bit positions. LSB is placed as a new MSB and a new CF.

Eg. ROR CX, 1

MOV CL, 03H

ROR BL, CL

RCL Instruction: RCL destination, count

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

Eg. RCL CX, 1

MOV CL, 04H

RCL AL, CL

RCR Instruction: RCR destination, count

This instruction rotates all bits in a specified byte or word some number of bit positions to the right *along with the carry flag*. LSB is placed as a new carry and previous carry is place as new MSB.

Eg. RCR CX, 1

MOV CL, 04H

RCR AL, CL

ROR Instruction: ROR destination, count

This instruction rotates all bits in a specified byte or word to the *right* some number of bit positions. LSB is placed as a new MSB and a new CF.

Eg. ROR CX, 1

MOV CL, 03H

ROR BL, CL

RCL Instruction: RCL destination, count

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

Eg. RCL CX, 1

MOV CL, 04H

RCL AL, CL

RCR Instruction: RCR destination, count

This instruction rotates all bits in a specified byte or word some number of bit positions to the right *along with the carry flag*. LSB is placed as a new carry and previous carry is place as new MSB.

Eg. RCR CX, 1

MOV CL, 04H

RCR AL, CL

Loop Instructions:

Unconditional LOOP Instructions

LOOP: LOOP Unconditionally

This instruction executes the part of the program from the Label or address specified in the instruction upto the LOOP instruction CX number of times. At each iteration, CX is decremented automatically and JUMP IF NOT ZERO structure.

Example: MOV CX, 0004H

Conditional LOOP Instructions

LOOPZ / LOOPE Label

Loop through a sequence of instructions from label while ZF=1 and CX=0.

LOOPNZ / LOOPENE Label

Loop through a sequence of instructions from label while ZF=1 and CX=0.

Branch Instructions:

Branch Instructions transfers the flow of execution of the program to a new address specified in the instruction directly or indirectly. When this type of instruction is executed, the CS and IP registers get loaded with new values of CS and IP corresponding to the location to be transferred.

The Branch Instructions are classified into two types

1. Unconditional Branch Instructions.
2. Conditional Branch Instructions.

Unconditional Branch Instructions:

In Unconditional control transfer instructions, the execution control is transferred to the specified location independent of any status or condition. The CS and IP are unconditionally modified to the new CS and IP.

CALL: Unconditional Call

This instruction is used to call a Subroutine (Procedure) from a main program. Address of procedure may be specified directly or indirectly. There are two types of procedure depending upon whether it is available in the same segment or in another segment.

i. Near CALL i.e., ±32K displacement.

ii. For CALL i.e., anywhere outside the segment.

On execution this instruction stores the incremented IP & CS onto the stack and loads the CS & IP registers with segment and offset addresses of the procedure to be called.

RET: Return from the Procedure.

At the end of the procedure, the RET instruction must be executed. When it is executed, the previously stored content of IP and CS along with Flags are retrieved into the CS, IP and Flag registers from the stack and execution of the main program continues further.

INT N: Interrupt Type N.

In the interrupt structure of 8086, 256 interrupts are defined corresponding to the types from 00H to FFH. When INT N instruction is executed, the type byte N is multiplied by 4 and the contents of IP and CS of the interrupt service routine will be taken from memory block in 0000 segment.

INTO: Interrupt on Overflow

This instruction is executed, when the overflow flag OF is set. This is equivalent to a Type 4 Interrupt instruction.

JMP: Unconditional Jump

This instruction unconditionally transfers the control of execution to the specified address using an 8-bit or 16-bit displacement. No Flags are affected by this instruction.

IRET: Return from ISR

When it is executed, the values of IP, CS and Flags are retrieved from the stack to continue the execution of the main program.

MOV BX, 7526H

Label 1 MOV AX, CODE

OR BX, AX

LOOP Label 1

Conditional Branch Instructions

When this instruction is executed, execution control is transferred to the address specified relatively in the instruction, provided the condition implicit in the Opcode is satisfied. Otherwise execution continues sequentially.

JZ/JE Label

Transfer execution control to address 'Label', if ZF=1.

JNZ/JNE Label

Transfer execution control to address 'Label', if ZF=0

JS Label

Transfer execution control to address 'Label', if SF=1.

JNS Label

Transfer execution control to address 'Label', if SF=0.

JO Label

Transfer execution control to address 'Label', if OF=1.

14

JNO Label

Transfer execution control to address 'Label', if OF=0.

JNP Label

Transfer execution control to address 'Label', if PF=0.

JP Label

Transfer execution control to address 'Label', if PF=1.

JB Label

Transfer execution control to address 'Label', if CF=1.

JNB Label

Transfer execution control to address 'Label', if CF=0.

JCXZ Label

Transfer execution control to address 'Label', if CX=0

String Manipulation Instructions

A series of data byte or word available in memory at consecutive locations, to be referred as Byte String or Word String. A String of characters may be located in consecutive memory locations, where each character may be represented by its ASCII equivalent. The 8086 supports a set of more powerful instructions for string manipulations for referring to a string, two parameters are required.

       I. Starting and End Address of the String.

       II. Length of the String.

The length of the string is usually stored as count in the CX register. The incrementing or decrementing of the pointer, in string instructions, depends upon the Direction Flag (DF) Status. If it is a Byte string operation, the index registers are updated by one. On the other hand, if it is a word string operation, the index registers are updated by two.

REP: Repeat Instruction Prefix

This instruction is used as a prefix to other instructions, the instruction to which the REP prefix is provided, is executed repeatedly until the CX register becomes zero (at each iteration CX is automatically decremented by one).

       i. REPE / REPZ - repeat operation while equal / zero.

       ii. REPNE / REPNZ - repeat operation while not equal / not zero.

These are used for CMPS, SCAS instructions only, as instruction prefixes.

MOVSB / MOVSW: Move String Byte or String Word

Suppose a string of bytes stored in a set of consecutive memory locations is to be moved to another set of destination locations. The starting byte of source string is located in the memory location whose address may be computed using SI (Source Index) and DS (Data Segment) contents. The starting address of the destination locations where this string has to be relocated is given by DI (Destination Index) and ES (Extra Segment) contents.

CMPS: Compare String Byte or String Word

The CMPS instruction can be used to compare two strings of byte or words. The length of the string must be stored in the register CX. If both the byte or word strings are equal, zero Flag is set.

The REP instruction Prefix is used to repeat the operation till CX (counter) becomes zero or the condition specified by the REP Prefix is False.

SCAN: Scan String Byte or String Word

This instruction scans a string of bytes or words for an operand byte or word specified in the register AL or AX. The String is pointed to by ES: DI register pair. The length of the string s stored in CX. The DF controls the mode for scanning of the string. Whenever a match to the specified operand is found in the string, execution stops and the zero Flag is set. If no match is found, the zero flag is reset.

LODS: Load String Byte or String Word

The LODS instruction loads the AL / AX register by the content of a string pointed to by DS: SI register pair. The SI is modified automatically depending upon DF, If it is a

19

byte transfer (LODSB), the SI is modified by one and if it is a word transfer (LODSW), the SI is modified by two. No other Flags are affected by this instruction.

STOS: Store String Byte or String Word

The STOS instruction Stores the AL / AX register contents to a location in the string pointer by ES: DI register pair. The DI is modified accordingly, No Flags are affected by this instruction.

The direction Flag controls the String instruction execution, The source index SI and Destination Index DI are modified after each iteration automatically. If DF=1, then the execution follows auto decrement mode, SI and DI are decremented automatically after each iteration. If DF=0, then the execution follows auto increment mode. In this mode, SI and DI are incremented automatically after each iteration.

Flag Manipulation and a Processor Control Instructions

These instructions control the functioning of the available hardware inside the processor chip. These instructions are categorized into two types:

1. Flag Manipulation instructions.
2. Machine Control instructions.

Flag Manipulation instructions

The Flag manipulation instructions directly modify some of the Flags of 8086.

    i. CLC – Clear Carry Flag.
    ii. CMC – Complement Carry Flag.
    iii. STC – Set Carry Flag.
    iv. CLD – Clear Direction Flag.
    v. STD – Set Direction Flag.
    vi. CLI – Clear Interrupt Flag.
    vii. STI – Set Interrupt Flag.

Machine Control instructions

The Machine control instructions control the bus usage and execution

    i. WAIT – Wait for Test input pin to go low.
    ii. HLT – Halt the process.
    iii. NOP – No operation.
    iv. ESC – Escape to external device like NDP
    v. LOCK – Bus lock instruction prefix.

Assembler directives:

       Assembler directives help the assembler to correctly understand the assembly language programs to prepare the codes. Another type of hint which helps the assembler to assign a particular constant with a label or initialize particular memory locations or labels with constants is called an operator. Rather, the operators perform the arithmetic and logical tasks unlike directives that just direct the assembler to correctly interpret the program to code it appropriately. The following directives are commonly used in the assembly language programming practice using Microsoft Macro Assembler

(MASM) or Turbo Assembler (TASM).

✓ **DB: Define Byte** The DB directive is used to reserve byte or bytes of memory locations in the available memory. While preparing the EXE file, this directive directs the assembler to allocate the specified number of memory bytes to the said data type that may be a constant, variable, string, etc. Another option of this directive also initializes the reserved memory bytes with the ASCII codes of the characters specified as a string. The following examples show how the DB directive is used for different purposes.

Example:

LIST DB 0lH, 02H, 03H, 04H

This statement directs the assembler to reserve four memory locations for a list named LIST and initialize them with the above specified four values.

MESSAGE DB 'GOOD MORNING'

This makes the assembler reserve the number of bytes of memory equal to the number of characters in the string named MESSAGE and initialize those locations by the ASCII equivalent of these characters.

✓ **DW: Define Word**. The DW directive serves the same purposes as the DB directive, but it now makes the assembler reserve the number of memory words (16-bit) instead of bytes. Some examples are given to explain this directive.

Examples

WORDS DW 1234H, 4567H, 78ABH, 045CH

This makes the assembler reserve four words in memory (8 bytes), and initialize the words with the specified values in the statements. During initialisation, the lower bytes are stored at the lower memory addresses, while the upper bytes are stored at the higher addresses. Another option of the DW directive is explained with the DUP operator.

WDATA DW 5 DUP (6666H)

This statement reserves five words, i.e. 10-bytes of memory for a word label WDATA and initializes all the word locations with 6666H.

✓ **DQ: Define Quad word** This directive is used to direct the assembler to reserve 4words (8 bytes) of memory for the specified variable and may initialize it with the specified values.

✓ **DT: Define Ten Bytes.** The DT directive directs the assembler to define the specified variable requiring la-bytes for its storage and initialize the 10bytes with the specified values. The directive may be used in case of variables facing heavy numerical calculations, generally processed by numerical processors.

✓ **ASSUME**: **Assume Logical Segment Name** The ASSUME directive is used to inform the assembler, the names of the logical segments to be assumed for different segments used in the program. In the assembly language program, each segment is given a name. For example, the code segment may be given the name CODE, data segment may be given the name DATA etc. The statement ASSUME CS: CODE directs the assembler that the machine codes are available in a segment named CODE, and hence the CS register is to be loaded with the address (segment) allotted by the operating system for the label CODE, while loading. Similarly, ASSUME DS: DATA indicates to the assembler that the data items related to the program, are available in a logical segment named DATA, and the DS register is to be initialized by the segment address value decided by the operating system for the data segment, while loading. It then considers the segment DATA as a default data segment for each memory operation, related to the data and the segment CODE as a source segment for the machine codes of the program. The ASSUME statement is a must at the starting of each assembly language program,

✓ **END: END of Program** The END directive marks the end of an assembly language program. When the assembler comes across this END directive, it ignores the source lines available later on. Hence, it should be ensured that the END statement should be the last statement in the file and should not appear in between. No useful program statement should lie in the file, after the END statement

✓ **ENDP: END of Procedure.** In assembly language programming, the subroutines are called procedures. Thus, procedures may be independent program modules which return particular results or values to the calling programs. The ENDP directive is used to indicate the end of a procedure. A procedure is usually assigned a name, i.e. label. To mark the end of a particular procedure, the name of the procedure, i.e. label may appear as a prefix with the directive ENDP. The statements, appearing in the same module but after the ENDP directive, are neglected from that procedure. The structure given below explains the use of ENDP.

PROCEDURE STAR

.

.

.

STAR ENDP

✓ **ENDS: END of Segment** This directive marks the end of a logical segment. The logical segments are assigned with the names using the ASSUME directive. The names appear with the ENDS directive as prefixes to mark the end of those particular segments. Whatever are the contents of the segments, they should appear in the program before ENDS. Any statement appearing after ENDS will be neglected from the segment. The structure shown below explains the fact more clearly.

          DATA SEGMENT
          .
          .
          .
          DATA ENDS
          ASSUME CS: CODE, DS:DATA
          CODE SEGMENT.
          .
          .
          .
          CODE ENDS
          END

The above structure represents a simple program containing two segments named DATA and CODE. The data related to the program must lie between the DATA SEGMENT and DATA ENDS statements. Similarly, all the executable instructions must lie between CODE SEGMENT and CODE ENDS statements.

✓ **EVEN: Align on Even Memory Address** The assembler, while starting the assembling procedure of any program, initializes a location counter and goes on updating it, as the assembly proceeds. It goes on assigning the available addresses, i.e. the contents of the location counter, sequentially to the program variables, constants and modules as per their requirements, in the sequence in which they appear in the program. The EVEN directive updates the location counter to the next even address if the current location counter contents are not even, and assigns the following routine or variable or constant to that address. The structure given below explains the directive.

          EVEN
          PROCEDURE ROOT
          .
          .
          .
          ROOT ENDP

The above structure shows a procedure ROOT that is to be aligned at an even address. The assembler will start assembling the main program calling ROOT. When the assembler comes across the directive EVEN, it checks the contents of the location counter. If it is odd, it is updated to the next even value and then the ROOT procedure is assigned to that address, i.e. the updated contents of the location counter. If the content of the location counter is already even, then the ROOT procedure will be assigned with the same address.

✓ **EQU: Equate** The directive EQU is used to assign a label with a value or a symbol. The use of this directive is just to reduce the recurrence of the numerical values or constants in a program code. The recurring value is assigned with a label, and that label is used in place of that numerical value, throughout the program. While assembling, whenever the assembler comes across the label, it substitutes the numerical value for that label and finds out the equivalent code. Using the EQU directive, even an instruction mnemonic can be assigned with a label, and the label can then be used in the program in place of th at mnemonic. Suppose, a numerical constant appears 'in a program ten times. If that constant is to be changed at a later time, one will have to make all these ten corrections. This may lead to human errors, because it

is possible that a human programmer may miss one of those corrections. This will result in the generation of wrong codes. If the EQU directive is used to assign the value with a label that can be used in place of each recurrence of that constant, only one change in the EQU statement will give the correct and modified code. The examples given below show the syntax.

Example
LABEL EQU 0500H
ADDITION EQU ADD

The first statement assigns the constant 500H with the label LABEL, while the second statement assigns another label ADDITION with mnemonic ADD.

✓ **EXTRN: External and PUBLIC: Public** The directive EXTRN informs the assembler that the names, procedures and labels declared after this directive have already been defined in some other assembly language modules. While in the other module, where the names, procedures and labels actually appear, they must be declared public, using the PUBLIC directive. If one wants to call a procedure FACTORIAL appearing in MODULE 1 from MODULE 2; in MODULE1, it must be declared PUBLIC using the statement PUBLIC FACTORIAL and in module 2, it must be declared external using the declaration EXTRN FACTORIAL. The statement of declaration EXTRN must be accompanied by the SEGMENT and ENDS directives of the MODULE 1, before it is called in MOBULE 2. Thus the MODULE 1 and MODULE 2 must have the following declarations.

MODULEl SEGMENT
PUBLIC FACTORIAL FAR
MODULEl ENDS
MODULE2 SEGMENT
EXTRN FACTORIAL FAR
MODULE2 ENDS

✓ **GROUP: Group the Related segment** The directive is used to form logical groups of segments with similar purpose or type. This directive is used to inform the assembler to form a logical group of the following segment names. The assembler passes information to the linker/loader to form the code such that the group declared segments or operands must lie within a 64Kbyte memory segment. Thus all such segments and labels can be addressed using the same segment base.

PROGRAM GROUP CODE, DATA, STACK

The above statement directs the loader/linker to prepare an EXE file such that CODE, DATA and STACK segment must lie within a 64kbyte memory segment that is named as PROGRAM. Now, for the ASSUME statement, one can use the label PROGRAM rather than CODE, DATA and STACK as shown.

ASSUME CS: PROGRAM, DS: PROGRAM, SS: PROGRAM.

✓ **LABEL: Label** The Label directive is used to assign a name to the current content of the location counter. At the start of the assembly process, the assembler initializes a location counter to keep track of memory locations assigned to the program. As the program assembly proceeds, the contents of the location counter are updated. During the assembly process, whenever the assembler comes across the LABEL directive, it assigns the declared label with the current contents of the location counter. The type of the label must be specified, i.e. whether it is a NEAR or a FAR label, BYTE or WORD label, etc. A LABEL directive may be used to make a FAR jump as shown below. A FAR jump cannot be made at a normal label with a colon. The label CONTINUE can be used for a FAR jump, if the program contains the following statement.

CONTINUE LABEL FAR

The LABEL directive can be used to refer to the data segment along with the data type, byte or word as shown.

DATA SEGMENT

DATAS DB 50H DUP (?)

DATA-LAST LABEL BYTE FAR

DATA ENDS

After reserving 50H locations for DATAS, the next location will be assigned a label DATALAST and its type will be byte and far.

✓ **LENGTH: Byte Length of a Label** This directive is not available in MASM. This is used to refer to the length of a data array or a string.

MOV CX, LENGTH ARRAY

This statement, when assembled, will substitute the length of the array ARRAY in bytes, in the instruction.

✓ **LOCAL** The labels, variables, constants or procedures declared LOCAL in a module are to be used only by that module. At a later time, some other module may declare a particular data type LOCAL, which is previously declared LOCAL by another module or modules. Thus the same label may serve different purposes for different modules of a program. With a single declaration statement, a number of variables can be declared local, as shown.

LOCAL a, b, DATA, ARRAY, ROUTINE

✓ **NAME: Logical Name of a Module** the NAME directive is used to assign a name to an assembly language program module. The module may now be referred to by its declared name. The names, if selected to be suggestive, may point out the functions of the different modules and hence may help in the documentation.

✓ **OFFSET: Offset of a Label** When the assembler comes across the OFFSET operator along with a label, it first computes the 16-bit displacement (also called as offset interchangeably) of the particular label, and replaces the string 'OFFSET LABEL' by the computed displacement. This operator is used with arrays, strings, labels and procedures to decide their offsets in their default segments. The

segment may also be decided by another operator of similar type, viz., SEG. Its most common use is in the case of the indirect, indexed, based indexed or other addressing techniques of similar types, used to refer to the memory indirectly. The examples of this operator are as follows:

Example:

CODE SEGMENT

MOV SI, OFFSET LIST

CODE ENDS

DATA SEGMENT

LIST DB 10H

DATA ENDS

✓ **ORG: Origin** The ORG directive directs the assembler to start the memory allotment for the particular segment, block or code from the declared address in the ORG statement while starting the assembly process for a module, the assembler initializes a location counter to keep track of the allotted addresses for the module. If the ORG statement is not written in the program, the location counter is initialized to 0000. If an ORG 200H statement is present at the starting of the code segment of that module, then the code will start from 200H address in code segment) In other words, the location counter will get initialized to the address 0200H instead of 0000H. Thus, the code for different modules and segments can be located in the available memory as required by the programmer. The ORG directive can even be used with data segments similarly.

✓ **PROC: Procedure** The PROC directive marks the start of a named procedure in the statement. Also, the types NEAR or FAR specify the type of the procedure, i.e whether it is to be called by the main program located within 64K of physical memory or not. For example, the statement RESULT PROC NEAR marks the start of a routine RESULT, which is to be called by a program located in the

Same segment of memory. The FAR directive is used for the procedures to be called by the programs located in different segments of memory. The example statements are as follows:

Example

RESULT PROC NEAR

ROUTINE PROC FAR

✓ **PTR: Pointer** The pointer operator is used to declare the type of a label, variable or memory operand. The operator PTR is prefixed by either BYTE or WORD. If the prefix is BYTE, then the particular label, variable or memory operand is treated as an 8-bit quantity, while if WORD is the prefix, then it is treated as a 16- bit quantity. In other words, the PTR operator is used to specify the data type -byte or word. The examples of the PTR operator are as follows:

Example:

MOV AL, BYTE PTR [SI]; Moves content of memory location addressed by SI (8-bit) to AL

INC BYTE PTR [BX]; Increments byte contents of memory location addressed by BX

MOV BX, WORD PTR [2000H]; Moves 16-bit content of memory location 2000H to BX, i.e. [2000H] to BL [2001 H] to BH

INC WORD PTR [3000H] - Increments word contents of memory location 3000H considering contents of 3000H (lower byte) and 3001 H (higher byte) as a 16-bit number

In case of JMP instructions, the PTR operator is used to specify the type of the jump, i.e. near or far, as explained in the examples given below.

JMP WORD PTR [BX] -NEAR Jump

JMP WORD PTR [BX] -FAR Jump

✓ **PUBLIC** As already discussed, the PUBLIC directive is used along with the EXTRN directive. This informs the assembler that the labels, variables, constants, or procedures declared PUBLIC may be accessed by other assembly modules to form their codes, but while using the PUBLIC declared labels, variables, constants or procedures the user must declare them externals using the EXTRN directive. On the other hand, the data types declared EXTRN in a module of the program, may be declared PUBLIC in at least anyone of the other modules of the same program.

✓ **SEG: Segment of a Label** The SEG operator is used to decide the segment address of the label, variable, or procedure and substitutes the segment base address in place of 'SEG label'. The example given below explain the use of SEG operator.

Example

MOV AX, SEG ARRAY; This statement moves the segment address

MOV DS, AX; of ARRAY in which it is appearing, to register AX and then to DS.

✓ **SEGMENT: Logical Segment** The SEGMENT directive marks the starting of a logical segment. The started segment is also assigned a name, i.e. label, by this statement. The SEGMENT and ENDS directive must bracket each logical segment of a program. In some cases, the segment may be assigned a type like PUBLIC (i.e. can be used by other modules of the program while linking) or GLOBAL (can be accessed by any other modules). The program structure given below explains the use of the SEGMENT directive.

EXE . CODE SEGMENT GLOBAL; Start of segment named

EXE.CODE, that can be accessed by any other module.

EXE . CODE ENDS; END of EXE.CODE logical segment.

✓ **SHORT** The SHORT operator indicates to the assembler that only one byte is required to code the displacement for a jump (i.e. displacement is within -128 to +127 bytes from the address of the byte next to the jump opcode). This method of specifying the jump address saves the memory. Otherwise, the assembler may reserve two bytes for the displacement. The syntax of the statement is as given below.

JMP SHORT LABEL

✓ **TYPE** The TYPE operator directs the assembler to decide the data type of the specified label and replaces the 'TYPE label' by the decided data type. For the word type variable, the data type is 2, for double word type, it is 4, and for byte type, it is 1. Suppose, the STRING is a word array. The instruction

MOV AX, TYPE STRING moves the value 0002H in AX.

✓ **GLOBAL** The labels, variables, constants or procedures declared GLOBAL may be used by other modules of the program. Once a variable is declared GLOBAL, it can be used by any module in the program. The following statement declares the procedure ROUTINE as a global label.

ROUTINE PROC GLOBAL

1.5 ASSEMBLY LANGUAGE PROGRAMMING

ALP for addition of two 8-bit numbers
```
DATA SEGMENT
VAR1 DB 85H
VAR2 DB 32H
RES DB?
DATA ENDS
ASSUME CS:CODE, DS:DATA
CODE SEGMENT
START: MOV AX, DATA
MOV DS, AX
MOV AL, VAR1
MOV BL, VAR2
ADD AL, BL
MOV RES, AL
MOV AH, 4CH
INT 21H
CODE ENDS
END START
```

ALP for Subtraction of two 8-bit numbers
```
DATA SEGMENT
VAR1 DB 53H
VAR2 DB 2AH
RES DB?
DATA ENDS
ASSUME CS:CODE,DS:DATA
CODE SEGMENT
START: MOV AX,DATA
MOV DS,AX
MOV AL,VAR1
MOV BL,VAR2
SUB AL,BL
MOV RES,AL
MOV AH,4CH
INT 21H
CODE ENDS
END START
```

ALP for Multiplication of two 8-bit numbers
```
DATA SEGMENT
VAR1 DB 0EDH
VAR2 DB 99H
RES DW?
DATA ENDS
ASSUME CS: CODE, DS:DATA
CODE SEGMENT
START: MOV AX, DATA
MOV DS, AX
MOV AL, VAR1
MOV BL, VAR2
MUL BL
MOV RES, AX
MOV AH, 4CH
INT 21H
CODE ENDS
END START
```

ALP for division of 16-bit number with 8-bit number
```
DATA SEGMENT
VAR1 DW 6827H
VAR2 DB 0FEH
QUO DB?
REM DB?
DATA ENDS
ASSUME CS:CODE,DS:DATA
CODE SEGMENT
START: MOV AX, DATA
MOV DS, AX
MOV AX, VAR1
DIV   VAR2
MOV QUO, AL
MOV REM, AH
MOV AH, 4CH
INT 21H
CODE ENDS
END START
```

ALP for Subtraction of two 16-bit numbers
```
DATA SEGMENT
VAR1 DW 8560H
VAR2 DW 3297H
RES DW?
DATA ENDS
ASSUME CS: CODE,DS:DATA
CODE SEGMENT
START: MOV AX, DATA
MOV DS, AX
MOV AX, VAR1
CLC
SUB AX, VAR2
MOV RES, AX
```

```
MOV AH, 4CH
INT 21H
CODE ENDS
END START
```

1.6   Modular programming

ALP for Multiplication of two 32-bit numbers

```
DATA SEGMENT
MULD DW 0FFFFH, 0FFFFH
MULR DW 0FFFFH, 0FFFFH
RES DW 6 DUP (0)
DATA ENDS
ASSUME CS: CODE,DS: DATA
CODE SEGMENT
START: MOV AX, DATA
MOV DS, AX
MOV AX, MULD
MUL MULR
MOV RES, AX
MOV RES+2, DX
MOV AX, MULD+2
MUL MULR
ADD RES+2, AX
ADC RES+4, DX
MOV AX, MULD
MUL MULR+2
ADD RES+2, AX
ADC RES+4, DX
JNC K
INC RES+6
K: MOV AX, MULD+2
MUL MULR+2
ADD RES+4, AX
ADC RES+6, DX
MOV AH, 4CH
INT 21H
CODE ENDS
END START
```

ALP to Sort a set of unsigned integer numbers in a nding/ de nding order using Bubble sort algorithm.

```
DATA SEGMENT
A DW 0005H, 0ABCDH, 5678H, 1234H, 0EFCDH, 45EFH
DATA ENDS
ASSUME CS: CODE, DS: DATA
CODE SEGMENT
START: MOV AX, DATA
MOV DS, AX
MOV SI, 0000H
MOV BX, A[SI]
DEC BX
X2: MOV CX, BX
MOV SI, 02H
```

```
X1: MOV AX, A[SI]
INC SI
INC SI
CMP AX, A[SI]
XCHG AX, A[SI]
MOV A[SI-2], AX
X3: LOOP X1
DEC BX
JNZ X2
MOV AH, 4CH
INT 21H
CODE ENDS
END START
```

1.7 Linking And Relocation

The DOS linking program links the different object modules of a source program and function library routines to generate an integrated executable code of the source program. The main input to the linker is the .OBJ file that contains the object modules of the source programs. Other supporting information may be obtained from the files generated by the MASM. The linker program is invoked using the following options.
C> LINK
or
C>LINK MS.OBJ

The .OBJ extension is a must for a file to be accepted by the LINK as a valid object file. The first object may generate a display asking for the object file, list file and libraries as inputs and an expected name of the .EXE file to be generated. The output of the link program is an executable file with the entered filename and .EXE extension. This executable filename can further be entered at the DOS prompt to execute the file.

In the advanced version of the MASM, the complete procedure of assembling and linking is combined under a single menu invokable compile function. The recent versions of MASM have much more sophisticated and user-friendly facilities and options. A linker links the machine codes with the other required assembled codes. Linking is necessary because of the number of codes to be linked for the final binary file.

The linked file in binary for *run* on a computer is commonly known as executable file or simply '.exe.' file. After linking, there has to be re-allocation of the sequences of placing the codes before actually placement of the codes in the memory.

The loader program performs the task of reallocating the codes after finding the physical RAM addresses available at a given instant. The DOS linking program links the different object modules of a source program and function library routines to generate an integrated executable code of the source program. The main input to the linker is the .OBJ file that contains the object modules of the source programs. Other supporting information may be obtained from the files generated by the MASM. The linked file in binary for *run* on a computer is commonly known as executable file or simply '.exe.' file. After linking, there has to be re-allocation of the sequences of placing the codes before actually placement of the codes in the memory.

The loader program performs the task of reallocating the codes after finding the physical RAM addresses available at a given instant. The *loader* is a part of the operating system and places codes into the memory after reading the '.exe' file. This step is necessary because the available memory addresses may not start from 0x0000, and binary codes have to be loaded at the different addresses during the run. The loader finds the appropriate start address. In a computer, the loader is used and it loads into a section of RAM the program

that is ready to run. A program called *locator* reallocates the linked file and creates a file for permanent location of codes in a standard format.

Segment combination

In addition to the linker commands, the assembler provides a means of regulating the way segments in

different object modules are organized by the linker.

Segments with same name are joined together by using the modifiers attached to the SEGMENT directives. SEGMENT directive may have the form

Segment name SEGMENT Combination-type

where the combine-type indicates how the segment is to be located within the load module. Segments that have different names cannot be combined and segments with the same name but no combine-type will cause a linker error. The possible combine-types are:

✓ **PUBLIC** – If the segments in different modules have the same name and combine-type PUBLIC, then they are concatenated into a single element in the load module. The ordering in the concatenation is specified by the linker command.

✓ **COMMON** – If the segments in different object modules have the same name and the combine-type is COMMON, then they are overlaid so that they have the same starting address. The length of the common segment is that of the longest segment being overlaid.

✓ **STACK** – If segments in different object modules have the same name and the combine type

STACK, then they become one segment whose length is the sum of the lengths of the individually specified segments. In effect, they are combined to form one large stack

✓ **AT** – The AT combine-type is followed by an expression that evaluates to a constant which is to be the segment address. It allows the user to specify the exact location of the segment in memory.

MEMORY – This combine-type causes the segment to be placed at the last of the load module. If more than one segment with the MEMORY combine-type is being linked, only the first one will be treated as having the MEMORY combine type; the others will be overlaid as if they had COMMON combine-type.

```
Source module 1
  DATA      SEGMENT   COMMON
  DATA              ENDS
  CODE          SEGMENT    PUBLIC
  CODE          ENDS


Source module 2
  DATA      SEGMENT   COMMON
              .
              .
  DATA      ENDS
  CODE      SEGMENT    PUBLIC
              .
              .
  CODE      ENDS
```
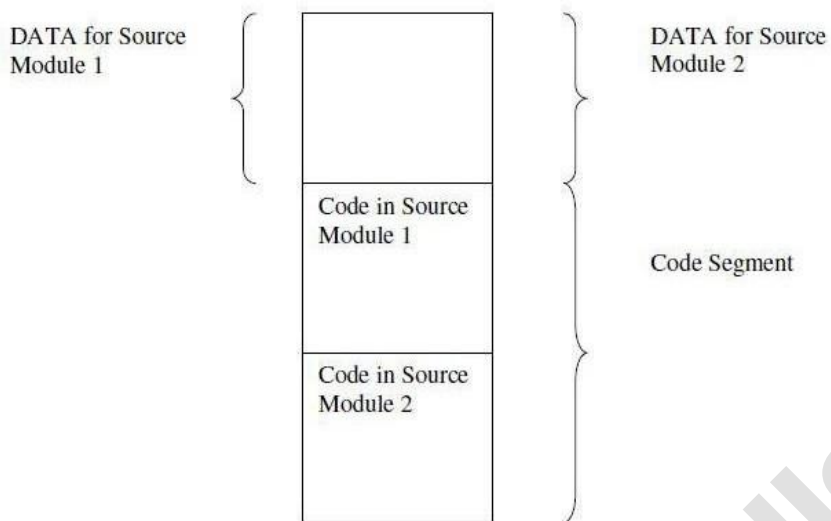
Fig. 1.9 Segment combinations resulting from the PUBLIC and Common Combination types

```
Source module 1
STACK_SEG  SEGMENT STACK
      DW    20 DUP (?)
      TOP-OF_STACK LABEL WORD
STACK_SEG ENDS
      END


Source module 2
      STACK_SEG        SEGMENT   STACK
      DW   30 DUP (?)
STACK_SEG        ENDS
            .
            .
            .
END
```
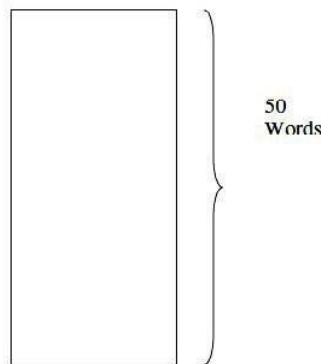

Fig.1.10 Formation of a stack from two segments

Access to External Identifiers

If an identifier is defined in an object module, then it is said to be a *local* (or *internal) identifier* relative to the module. If it is not defined in the module but is defined in one of the other modules being linked, then it is referred to as an *external* (or *global*) *identifier* relative to the module. In order to permit other object modules to reference some of the identifiers in a given module, the given module must include a list of the identifiers to

31

which it will allow access. Therefore, each module in multi-module programs may contain two lists, one containing the external identifiers that can be referred to by other modules. Two lists are implemented by the EXTRN and PUBLIC directives, which have the forms:

EXTRN Identifier: Type..., Identifier: Type
   and
PUBLIC   Identifier,..., Identifier

where the identifiers are the variables and labels being declared or as being available to other modules.

The assembler must know the type of all external identifiers before it can generate the proper machine code, a type specifier must be associated with each identifier in an EXTRN statement. For a variable the type may be BYTE, WORD, or DWORD and for a label it may be NEAR or FAR.

✓ One of the primary tasks of the linker is to verify that every identifier appearing in an EXTRN statement is matched by one in a PUBLIC statement. If this is not the case, then there will be an undefined reference and a linker error will occur. The offsets for the local identifier will be inserted by the assembler, but the offsets for the external identifiers and all segment addresses must be inserted by the linking process. The offsets associated with all external references can be assigned once all of the object modules have been found and their external symbol tables have been examined. The assignment of the segment addresses is called *relocation* and is done after the linking process has determined exactly where each segment is to be put in memory.

1.8 Stacks

The stack is a block of memory that may be used for temporarily storing the contents of the registers inside the CPU. It is a top-down data structure whose elements are accessed using the stack pointer (SP) which gets decremented by two as we store a data word into the stack and gets incremented by two as we retrieve a data word from the stack back to the CPU register.

The process of storing the data in the stack is called 'pushing into' the stack and the reverse process of transferring the data back from the stack to the CPU register is known as 'popping off' the stack. The stack is essentially *Last-In-First-Out* (LIFO) data segment. This means that the data which is pushed into the stack last will be on top of stack and will be popped off the stack first.

The stack pointer is a 16-bit register that contains the offset address of the memory location in the stack segment. The stack segment, like any other segment, may have a memory block of a maximum of 64 Kbytes locations, and thus may overlap with any other segments. Stack Segment register (SS) contains the base address of the stack segment in the memory.

The Stack Segment register (SS) and Stack pointer register (SP) together address the stack-top as explained below:

SS ⟹ 5000H

SP ⟹ 2050H

If the stack top points to a memory location 52050H, it means that the location 52050H is already occupied with the previously pushed data. The next 16 bit push operation will decrement the stack pointer by two, so that it will point to the new stack-top 5204EH and the decremented contents of SP will be 204EH. This location will now be occupied by the recently pushed data.

Thus for a selected value of SS, the maximum value of SP=FFFFH and the segment can have maximum of 64K locations. If the SP starts with an initial value of FFFFH, it will be decremented by two whenever a 16-bit data is pushed onto the stack. After successive push operations, when the stack pointer contains 0000H, any attempt to further push the data to the stack will result in stack overflow.

After a procedure is called using the CALL instruction, the IP is incremented to the next instruction. Then the contents of IP, CS and flag register are pushed automatically to the stack. The control is then transferred to the specified address in the CALL instruction i.e. starting address of the procedure. Then the procedure is executed.
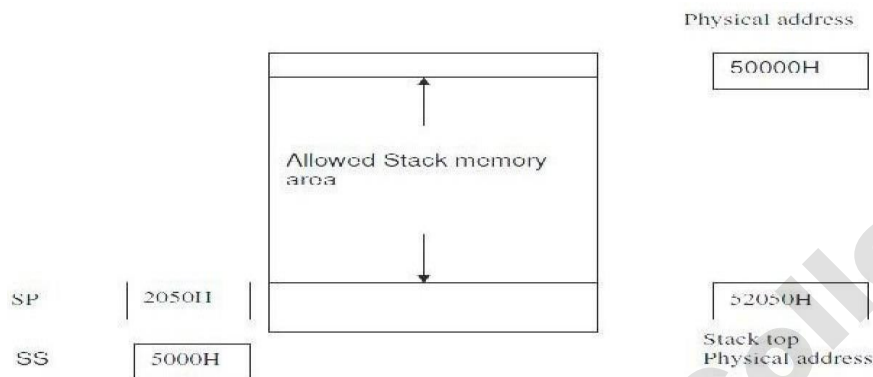
Fig. 1.11 Stack –top address calculation

1.9 Procedures

A procedure is a set of code that can be branched to and returned from in such a way that the code is as if it were inserted at the point from which it is branched to. The branch to procedure is referred to as the *call*, and the corresponding branch back is known as the *return*. The return is always made to the instruction immediately following the call regardless of where the call is located.

Calls, Returns, and Procedure Definitions

The CALL instruction not only branches to the indicated address, but also pushes the return address onto the stack. The RET instruction simply pops the return address from the stack. The registers used by the procedure need to be stored before their contents are changed, and then restored just before their contents are changed, and then restored just before the procedure is excited.

A CALL may be direct or indirect and intrasegment or intersegment. If the CALL is intersegment, the return must be intersegment. Intersegment call must push both (IP) and (CS) onto the stack. The return must correspondingly pop two words from the stack. In the case of intrasegment call, only the contents of IP will be saved and retrieved when call and return instructions are used.

Procedures are used in the source code by placing a statement of the form at the beginning of the procedure

Procedure name PROC Attribute

and by terminating the procedure with a statement

Procedure name ENDP

The attribute that can be used will be either NEAR or FAR. If the attribute is NEAR, the RET instruction will only pop a word into the IP register, but if it is FAR, it will also pop a word into the CS register.

A procedure may be in:

1. The same code segment as the statement that calls it.

2. A code segment that is different from the one containing the statement that calls it, but in the same source module as the calling statement.

3. A different source module and segment from the calling statement.

In the first case, the attribute could be NEAR provided that all calls are in the same code segment as the procedure. For the latter two cases the attribute must be FAR. If the

procedure is given a FAR attribute, then all calls to it must be intersegment calls even if the call is from the same code segment. For the third case, the procedure name must be declared in EXTRN and PUBLIC statements.

Saving and Restoring Registers

When both the calling program and procedure share the same set of registers, it is necessary to save the registers when entering a procedure, and restore them before returning to the calling program.

```
MSK PROC NEAR
PUSH AX
PUSH BX
PUSH CX
POP CX
POP BX
POP AX
RET
MSK ENDP
```

Procedure Communication

There are two general types of procedures, those operate on the same set of data and those that may process a different set of data each time they are called. If a procedure is in the same source module as the calling program, then the procedure can refer to the variables directly.

When the procedure is in a separate source module it can still refer to the source module directly provided that the calling program contains the directive

```
PUBLIC ARY, COUNT, SUM
EXTRN ARY: WORD, COUNT: WORD, SUM: WORD
```

Recursive Procedures

When a procedure is called within another procedure it called recursive procedure. To make sure that the procedure does not modify itself, each call must store its set of parameters, registers, and all temporary results in a different place in memory

Eg. Recursive procedure to compute the factorial

1.10 Macros

Disadvantages of Procedure

1. Linkage associated with them.

2. It sometimes requires more code to program the linkage than is needed to perform the task. If this is the case, a procedure may not save memory and execution time is considerably increased.

3. **Macros** is needed for providing the programming ease of a procedure while avoiding the linkage. Macro is a segment of code that needs to be written only once but whose basic structure can be caused to be repeated several times within a source module by placing a single statement at the point of each reference.

A macro is unlike a procedure in that the machine instructions are repeated each time the macro is referenced. Therefore, no memory is saved, but programming time is conserved (no linkage is required) and some degree of modularity is achieved. The code that is to be repeated is called the prototype code. The prototype code along with the statements for referencing and terminating is called the macro definition.

Once a macro is defined, it can be inserted at various points in the program by using macro calls. When a macro call is encountered by the assembler, the assembler replaces the call with the macro code.   Insertion of the macro code by the assembler for a macro call is referred to as a macro expansion. In order to allow the prototype code to be used in a variety of situations, macro definition and the prototype code can use dummy parameters which can be replaced by the actual parameters

when the macro is expanded. During a macro expansion, the first actual parameter replaces the first dummy parameter in the prototype code, the second actual parameter replaces the second dummy parameter, and so on.

A macro call has the form

%Macro name (Actual parameter list) with the actual parameters being separated by commas.

%MULTIPLY (CX, VAR, XYZ[BX]

1.10.2 Local Labels

Consider a macro called ABSOL which makes use of labels. This macro is used to replace the operand by its absolute value.

%*DEFINE (ABSOL(OPER))
(CMP %OPER, 0
JGE NEXT
NEG %OPER
%NEXT: NOP)

When the macro ABSOL is called for the first time, the label NEXT will appear in the program and, therefore it becomes defined. Any subsequent call will cause NEXT to be redefined. This will result in an error during assembly process because NEXT has been associated with more than one location. One solution to this problem would be to have NEXT replaced by a dummy parameter for the label. This would require the programmer to keep track of dummy parameters used.

One solution to this problem is the use of **Local Labels.** Local labels are special labels that will have suffixes that get incremented each time the macros are called. These suffixes are two digit numbers that gets incremented by one starting from zero. Labels can be declared as local label by attaching a prefix **Local. Local List of Local labels** at the end of first statement in the macro definition.

1.11 Interrupts And Interrupt Routines

Interrupt and its Need:

The microprocessors allow normal program execution to be interrupted in order to carry out a specific task/work. The processor can be interrupted in the following ways

i) by an external signal generated by a peripheral,

ii) by an internal signal generated by a special instruction in the program,

iii) by an internal signal generated due to an exceptional condition which occurs while executing an instruction. (For example, in 8086 processor, divide by zero is an exceptional condition which initiates type 0 interrupt and such an interrupt is also called execution).

The process of interrupting the normal program execution to carry out a specific task/work is referred to as interrupt. The interrupt is initiated by a signal generated by an external device or by a signal generated internal to the processor.

When a microprocessor receives an interrupt signal it stops executing current normal program, save the status (or content) of various registers (IP, CS and flag registers in case of 8086) in stack and then the processor executes a subroutine/procedure in order to perform the specific task/work requested by the interrupt. The subroutine/procedure that is executed in response to an interrupt is also called Interrupt Service Subroutine (ISR). At the end of ISR, the stored status of registers in stack is restored to respective registers, and the processor resumes the normal program execution from the point {instruction) where it was interrupted.

The external interrupts are used to implement interrupt driven data transfer scheme. The interrupts generated by special instructions are called software interrupts and they are used to implement system services/calls (or monitor services/calls). The system/monitor services are procedures developed by system designer for various operations and stored in

35

memory. The user can call these services through software interrupts. The interrupts generated by exceptional conditions are used to implement error conditions in the system.

Interrupt Driven Data Transfer Scheme

The interrupts are useful for efficient data transfer between processor and peripheral. When a peripheral is ready for data transfer, it interrupts the processor by sending an appropriate signal. Upon receiving an interrupt signal, the processor suspends the current program execution, save the status in stack and executes an ISR to perform the data transfer between the peripheral and processor.

At the end of ISR the processor status is restored from stack and processor resume its normal program execution. This type of data transfer scheme is called interrupt driven data transfer scheme.

The data transfer between the processor and peripheral devices can be implemented either by polling technique or by interrupt method. In polling technique, the processor has to periodically poll or check the status/readiness of the device and can perform data transfer only when the device 'is ready. In polling technique the processor time is wasted, because the processor has to suspend its work and check the status of the device in predefined intervals.

If the device interrupts the processor to initiate a data transfer whenever it is ready then the processor time is effectively utilized because the processor need not suspend its work and check the status of the device in predefined intervals.

For an example, consider the data transfer from a keyboard to the processor. Normally a keyboard has to be checked by the processor once in every 10 milliseconds for a key press. Therefore once in every 10 milliseconds the processor has to suspend its work and then check the keyboard for a valid key code. Alternatively, the keyboard can interrupt the processor, whenever a key is pressed and a valid key code is generated. In this way the processor need not waste its time to check the keyboard once in every 10 milliseconds.

Classification of Interrupts

In general the interrupts can be classified in the following three ways:

1. Hardware and software interrupts
2. Vectored and Non Vectored interrupt:
3. Maskable and Non Maskable interrupts.

The interrupts initiated by external hardware by sending an appropriate signal to the interrupt pin of the processor is called hardware interrupt. The 8086 processor has two interrupt pins INTR and NMI. The interrupts initiated by applying appropriate signal to these pins are called hardware interrupts of 8086.

The software interrupts are program instructions. These instructions are inserted at desired locations in a program. While running a program, if software interrupt instruction is encountered then the processor initiates an interrupt. The 8086 processor has 256 types of software interrupts. The software interrupt instruction is INT n, where n is the type number in the range 0 to 255.

When an interrupt signal is accepted by the processor, if the program control automatically branches to a specific address (called vector address) then the interrupt is called vectored interrupt. The automatic branching to vector address is predefined by the manufacturer of processors. (In these vector addresses the interrupt service subroutines (ISR) are stored). In non-vectored interrupts the interrupting device should supply the address of the ISR to be executed in response to the interrupt. All the 8086 interrupts are vectored interrupts. The vector address for an 8086 interrupt is obtained from a vector table implemented in the first 1kb memory space (00000h to 03FFFh).

The processor has the facility for accepting or rejecting hardware interrupts. Programming the processor to reject an interrupt is referred to as masking or disabling and programming the processor to accept an interrupt is referred to as unmasking or enabling. In

8086 the interrupt flag (IF) can be set to one to unmask or enable all hardware interrupts and IF is cleared to zero to mask or disable a hardware interrupts except NMI.

The interrupts whose request can be either accepted or rejected by the processor are called maskable interrupts. The interrupts whose request has to be definitely accepted (or cannot be rejected) by the processor are called non-maskable interrupts. Whenever a request is made by non-maskable interrupt, the processor has to definitely accept that request and service that interrupt by suspending its current program and executing an ISR. In 8086 processor all the hardware interrupts initiated through INTR pin are maskable by clearing interrupt flag (IF). The interrupt initiated through NMI pin and all software interrupts are non-maskable.

Sources of Interrupts in 8086

An interrupt in 8086 can come from one of the following three sources.

1. One source is from an external signal applied to NMI or INTR input pin of the processor. The interrupts initiated by applying appropriate signals to these input pins are called hardware interrupts.

2. A second source of an interrupt is execution of the interrupt instruction "INT n", where n is the type number. The interrupts initiated by "INT n" instructions are called software interrupts.

3. The third source of an interrupt is from some condition produced in the 8086 by the execution of an instruction. An example of this type of interrupt is divide by zero interrupt. Program execution will be automatically interrupted if you attempt to divide an operand by zero. Such conditional interrupts are also known as exceptions.

Interrupts of 8086

The 8086 microprocessor has 256 types of interrupts. INTEL has assigned a type number to each interrupt. The type numbers are in the range of 0 to 255. The 8086 processor has dual facility of initiating these 256 interrupts. The interrupts can be initiated either by executing "INT n" instruction where n is the type number or the interrupt can be initiated by sending an appropriate signal to INTR input pin of the processor.

For the interrupts initiated by software instruction" INT n ", the type number is specified by the instruction itself. When the interrupt is initiated through INTR pin, then the processor runs an interrupt acknowledge cycle to get the type number. (i.e., the interrupting device should supply the type number through D_0- D_7 lines when the processor requests for the same through interrupt acknowledge cycle).
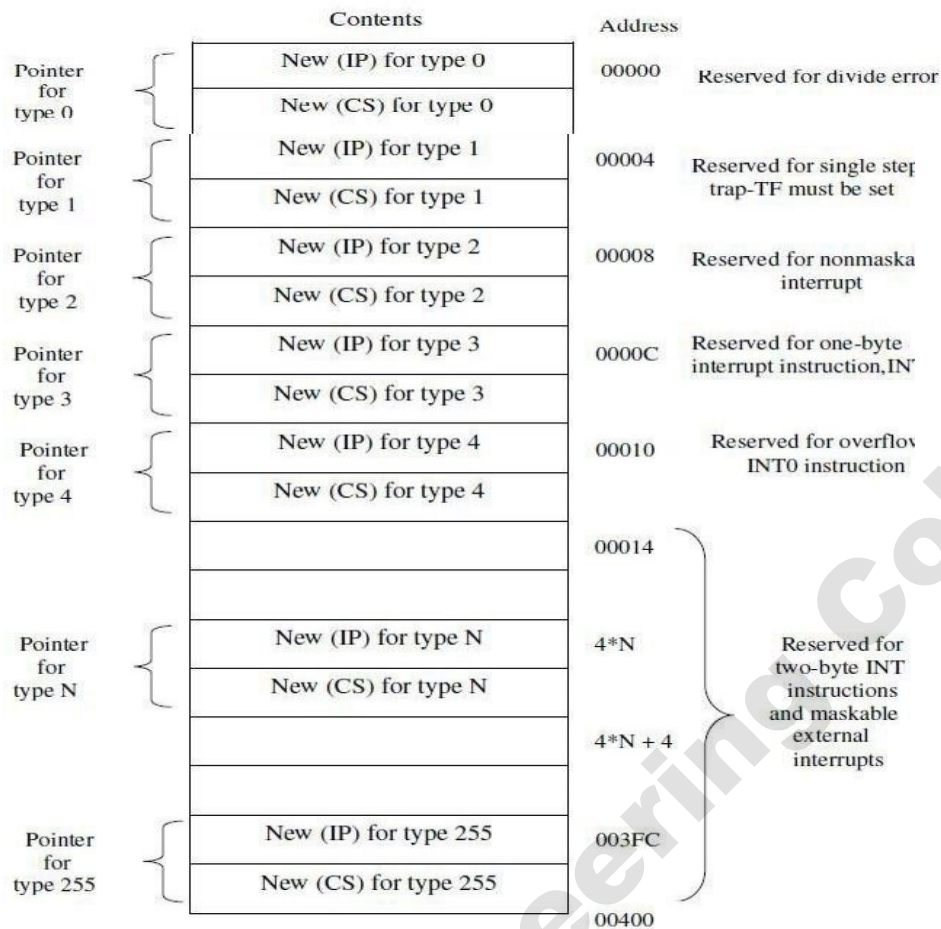
Fig. 1.12 Organization of Interrupt vector table in 8086

Only the first five types have explicit definitions; the other types may be used by interrupt instructions or external interrupts. From the figure it is seen that the type associated with a division error interrupt

is 0. Therefore, if a division by 0 is attempted, the processor will push the current contents of the PSW, CS and IP into the stack, fill the IP and CS registers from the addresses 00000 to 00003, and continue executing at the address indicated by the new contents of IP and CS. A division error interrupt occurs any time a DIV or IDIV instruction is executed with the quotient exceeding the range, regardless of the IF (Interrupt flag) and TF (Trap flag) status.

The type 1 interrupt is the single-step interrupt (Trap interrupt) and is the only interrupt controlled by the TF flag. If the TF flag is enabled, then an interrupt will occur at the end of the next instruction that will cause a branch to the location indicated by the contents of 00004H to 00007H. The single step interrupt is used primarily for debugging which gives the programmer a snapshot of his program after each instruction is executed

Name	Mnemonics	Description
Interrupt with Type	INT TYPE	$(SP) \leftarrow (SP) - 2$ $((SP)+1:(SP)) \leftarrow (PSW)$ $(SP) \leftarrow (SP) - 2$ $((SP)+1:(SP)) \leftarrow (CS)$ $((SP) \leftarrow (SP) - 2$ $((SP)+1:(SP)) \leftarrow (IP)$ $(IP) \leftarrow (TYPE * 4)$ $(CS) \leftarrow (TYPE * 4) + 2$
One byte interrupt	INT	$(SP) \leftarrow (SP) - 2$ $((SP)+1:(SP)) \leftarrow (PSW)$ $(SP) \leftarrow (SP) - 2$ $((SP)+1:(SP)) \leftarrow (CS)$ $((SP) \leftarrow (SP)\ 2$ $((SP)+1:(SP)) \leftarrow (IP)$ $(IP) \leftarrow (0CH)$ $(CS) \leftarrow (0EH)$
Interrupt on Overflow	INTO	If $(OF) = 1$, then $(SP) \leftarrow (SP) - 2$ $((SP)+1:(SP)) \leftarrow (PSW)$ $(SP) \leftarrow (SP) - 2$ $((SP)+1:(SP)) \leftarrow (CS)$ $((SP) \leftarrow (SP) - 2$ $((SP)+1:(SP)) \leftarrow (IP)$ $(IP) \leftarrow (10H)$ $(CS) \leftarrow (12H)$
Return from Interrupt	IRET	$(IP) \leftarrow ((SP)+1:(SP))$ $(SP) \leftarrow (SP) + 2$ $(CS) \leftarrow ((SP)+1:(SP))$ $(SP) \leftarrow (SP) + 2$ $(PSW) \leftarrow ((SP)+1:(SP))$ $(SP) \leftarrow (SP) + 2$

IRET is used to return from an interrupt service routine. It is similar to the RET instruction except that it pops the original contents of the PSW from the stack as well as the return address. The INT instruction has one of the forms

INT or INT Type

The INT instruction is also often used as a debugging aid in cases where single stepping provides more detail than is wanted.

By inserting INT instructions at key points, called *breakpoints*. Within a program a programmer can use an interrupt routine to provide messages and other information at these points. Hence the 1 byte INT instruction (Type 3 interrupt) is also referred to as *breakpoint interrupt*.

The INTO instruction has type 4 and causes an interrupt if and only if the OF flag is set to 1. It is often placed just after an arithmetic instruction so that special processing will be done if the instruction causes an overflow. Unlike a divide-by-zero fault, an overflow condition does not cause an interrupt automatically; the interrupt must be explicitly specified by the INTO instruction. The remaining interrupt types correspond to interrupts instructions imbedded in the interrupt program or to external interrupts.

1.12 Strings and String Handling Instructions

The 8086 microprocessor is equipped with special instructions to handle string operations. By string we mean a series of data words or bytes that reside in consecutive memory locations. The string instructions of the 8086 permit a programmer to implement operations such as to move data from one block of memory to a block elsewhere in memory. A second type of operation that is easily performed is to scan a string and data elements stored in memory looking for a specific value. Other examples are to compare the elements and two strings together in order to determine whether they are the same or different.

✓ **Move String:** MOV SB, MOV SW:

An element of the string specified by the source index (SI) register with respect to the current data segment (DS) register is moved to the location specified by the destination index (DI) register with respect to the current extra segment (ES) register. The move can be performed on a byte (MOV SB) or a word (MOV SW) of data. After the move is complete, the contents of both SI & DI are automatically incremented or decremented by 1 for a byte move and by 2 for a word move. Address pointers SI and DI increment or decrement depends on how the direction flag DF is set.

Example: Block move program using the move string instruction

```
MOV AX, DATA SEG ADDR
MOV DS, AX
MOV ES, AX
MOV SI, BLK 1 ADDR
MOV DI, BLK 2 ADDR
MOV CK, N
CDF; DF=0
NEXT: MOV SB
LOOP NEXT
HLT
```

✓ **Load and store strings:** (LOD SB/LOD SW and STO SB/STO SW)

LOD SB: Loads a byte from a string in memory into AL. The address in SI is used relative to DS to determine the address of the memory location of the string element.

$(AL) \neg [(DS) + (SI)]$

$(SI) \neg (SI) + 1$

LOD SW: The word string element at the physical address derived from DS and SI is to be loaded into AX. SI is automatically incremented by 2.

$(AX) \neg [(DS) + (SI)]$

$(SI) \neg (SI) + 2$

STO SB: Stores a byte from AL into a string location in memory. This time the contents of ES and DI are used to form the address of the storage location in memory.

$[(ES) + (DI)] \neg (AL)$

$(DI) \neg (DI) + 1$

STO SW: $[(ES) + (DI)] \neg (AX)$

$(DI) \neg (DI) + 2$

Repeat String: REP

The basic string operations must be repeated to process arrays of data. This is done by inserting a repeat prefix before the instruction that is to be repeated. Prefix REP causes the basic string operation to be repeated until the contents of register CX become equal to zero. Each time the instruction is executed, it causes CX to be tested for zero, if CX is found to be nonzero it is decremented by 1 and the basic string operation is repeated.

     Example: Clearing a block of memory by repeating STOSB

     MOV AX, 0
     MOV ES, AX
     MOV DI, A000
     MOV CX, OF
     CDF
     REP STOSB
     NEXT:

The prefixes REPE and REPZ stand for same function. They are meant for use with the CMPS and SCAS instructions. With REPE/REPZ the basic compare or scan operation can be repeated as long as both the contents of CX are not equal to zero and zero flag is 1.

     REPNE and REPNZ works similarly to REPE/REPZ except that now the operation is repeated as long as $CX \neq 0$ and ZF=0. Comparison or scanning is to be performed as long as the string elements are unequal (ZF=0) and the end of the string is not yet found ($CX \neq 0$).

✓ **Auto Indexing for String Instructions:**

SI & DI addresses are either automatically incremented or decremented based on the setting of the direction flag DF.

When CLD (Clear Direction Flag) is executed DF=0 permits auto increment by 1.

When STD (Set Direction Flag) is executed DF=1 permits auto decrement by 1.

<div align="center">

UNIT-II
8086 SYSTEM BUS STRUCTURE

</div>

The 8086 Microprocessor is a 16-bit CPU available in 3 clock rates, i.e. 5, 8 and 10MHz, packaged in a 40 pin CERDIP or plastic package. The 8086 Microprocessor operates in single processor or multiprocessor configurations to achieve high performance. The pin configuration is as shown in fig1. Some of the pins serve a particular function in minimum mode (single processor mode) and others function in maximum mode (multiprocessor mode) configuration.

The 8086 signals can be categorized in three groups. The first are the signals having common functions in minimum as well as maximum mode, the second are the signals which have special functions in minimum mode and third are the signals having special functions for maximum mode

2.1 8086 signals

The following signal description is common for both the minimum and maximum modes
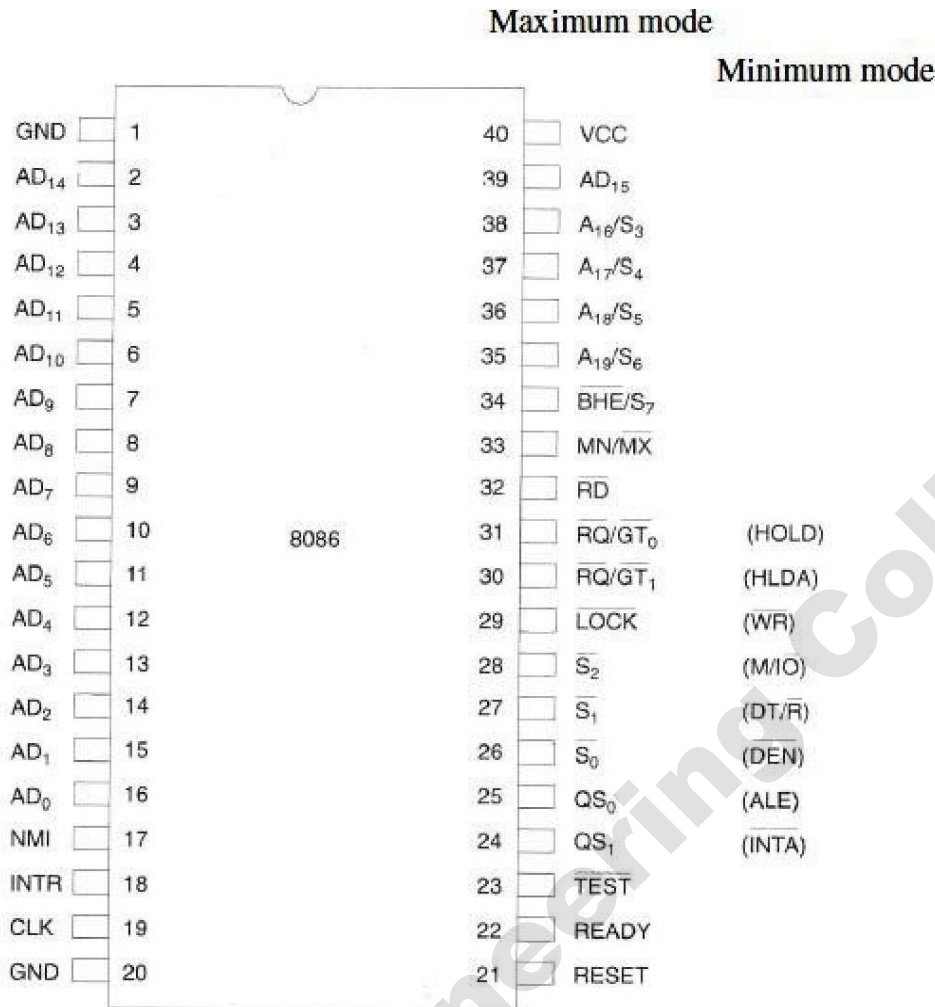
Fig.2.1 Bus signals

The 8086 signals can be categorized in three groups. The first are the signals having common functions in minimum as well as maximum mode, the second are the signals which have special functions in minimum mode and third are the signals having special functions for maximum mode.

The following signal description are common for both the minimum and maximum modes.

AD15-AD0: These are the time multiplexed memory I/O address and data lines. Address remains on the lines during T1 state, while the data is available on the data bus during T2, T3, TW and T4. Here T1, T2, T3, T4 and TW are the clock states of a machine cycle. TW is a wait state. These lines are active high and float to a tristate during interrupt acknowledge and local bus hold acknowledge cycles.

A19/S6, A18/S5, A17/S4, A16/S3: These are the time multiplexed address and status lines. During T1, these are the most significant address lines or memory operations. During I/O operations, these lines are low. During memory or I/O operations, status information is available on those lines for T2, T3, TW and T4 .The status of the interrupt enable flag bit(displayed on S5) is updated at the beginning of each clock cycle. The S4 and S3 combinedly indicate which segment register is presently being used for memory accesses as shown in Table 2.1.

These lines float to tri-state off (tristated) during the local bus hold acknowledge. The status line S6 is always low (logical). The address bits are separated from the status bits using latches controlled by the ALE signal.

Table 2.1 Bus High Enable / status

S4	S3	Indication
0	0	Alternate Data
0	1	Stack
1	0	Code or none
1	1	Data

BHE/S7-Bus High Enable/Status: The bus high enable signal is used to indicate the transfer of data over the higher order (D15-D8) data bus as shown in Table 2.1. It goes low for the data transfers over D15-D8 and is used to derive chip selects of odd address memory bank or peripherals. BHE is low during T1 for read, write and interrupt acknowledge cycles, when- ever a byte is to be transferred on the higher byte of the data bus. The status information is available during T2, T3 and T4. The signal is active low and is tristated during 'hold'. It is low during T1 for the first pulse of the interrupt acknowledge cycle.

Table 2.2 Bus high enable status

$\overline{BHE}$	A_0	Indication
0	0	Whole Word
0	1	Upper byte from or to odd address
1	0	Upper byte from or to even address
1	1	None

RD-Read: Read signal, when low, indicates the peripherals that the processor is performing a memory or I/O read operation. RD is active low and shows the state for T2, T3, TW of any read cycle. The signal remains tristated during the 'hold acknowledge'.

READY: This is the acknowledgement from the slow devices or memory that they have completed the data transfer. The signal made available by the devices is synchronized by the 8284A clock generator to provide ready input to the 8086. The signal is active high. **INTR-lnterrupt Request**: This is a level triggered input. This is sampled during the last clock cycle of each instruction to determine the availability of the request. If any interrupt request is pending, the processor enters the interrupt acknowledge cycle. This can be internally masked by resetting the interrupt enable flag. This signal is active high and internally synchronized.

TEST: This input is examined by a 'WAIT' instruction. If the TEST input goes low, execution will continue, else, the processor remains in an idle state. The input is synchronized internally during each clock cycle on leading edge of clock.

NMI-Non-maskable Interrupt: This is an edge-triggered input which causes a Type2 interrupt. The NMI is not maskable internally by software. A transition from low to high initiates the interrupt response at the end of the current instruction. This input is internally synchronized.

RESET: This input causes the processor to terminate the current activity and start execution from FFFF0H. The signal is active high and must be active for at least four clock cycles. It restarts execution when the RESET returns low. RESET is also internally synchronized.

CLK-Clock Input: The clock input provides the basic timing for processor operation and bus control activity. Its an asymmetric square wave with 33% duty cycle. The range of frequency for different 8086 versions is from 5MHz to 10MHz.

VCC: +5V power supply for the operation of the internal circuit. GND ground for the internal circuit.

MN/MX: The logic level at this pin decides whether the processor is to operate in either minimum (single processor) or maximum (multiprocessor) mode.

The following pin functions are for the minimum mode operation of 8086.

M/IO -Memory/IO: This is a status line logically equivalent to S2 in maximum mode. When it is low, it indicates the CPU is having an I/O operation, and when it is high, it indicates that the CPU is having a memory operation. This line becomes active in the previous T4 and remains active till final T4 of the current cycle. It is tristated during local bus "hold acknowledge".

INTA -Interrupt Acknowledge: This signal is used as a read strobe for interrupt acknowledge cycles. In other words, when it goes low, it means that the processor has accepted the interrupt. It is active low during T2, T3 and TW of each interrupt acknowledge cycle.

ALE-Address latch Enable: This output signal indicates the availability of the valid address on the address/data lines, and is connected to latch enable input of latches. This signal is active high and is never tristated.

DT /R -Data Transmit/Receive: This output is used to decide the direction of data flow through the transreceivers (bidirectional buffers). When the processor sends out data, this signal is high and when the processor is receiving data, this signal is low. Logically, this is equivalent to S1 in maximum mode. Its timing is the same as M/I/O. This is tristated during 'hold acknowledge'.

DEN-Data Enable This signal indicates the availability of valid data over the address/data lines. It is used to enable the transreceivers (bidirectional buffers) to separate the data from the multiplexed address/data signal. It is active from the middle ofT2 until the middle of T4 DEN is tristated during 'hold acknowledge' cycle.

HOLD, HLDA-Hold/Hold Acknowledge: When the HOLD line goes high, it indicates to the processor that another master is requesting the bus access. The processor, after receiving the HOLD request, issues the hold acknowledge signal on HLDA pin, in the middle of the next clock cycle after completing the current bus (instruction) cycle. At the same time, the processor floats the local bus and control lines. When the processor detects the HOLD line low, it lowers the HLDA signal. HOLD is an asynchronous input and it should be externally synchronized.

If the DMA request is made while the CPU is performing a memory or I/O cycle, it will release the local bus during T 4 provided:

1. The request occurs on or before T 2 state of the current cycle.

2. The current cycle is not operating over the lower byte of a word (or operating on an odd address).

3. The current cycle is not the first acknowledge of an interrupt acknowledge sequence.

4. A Lock instruction is not being executed.

So far we have presented the pin descriptions of 8086 in minimum mode.

The following pin functions are applicable for maximum mode operation of 8086.

S2, S1, S0 -Status Lines: These are the status lines which reflect the type of operation, being carried out by the processor. These become active during T4 of the previous cycle and remain active during T1 and T2 of the current bus cycle. The status lines return to passive state during T3 of the current bus cycle so that they may again become active for the next bus cycle during T4. Any change in these lines during T3 indicates the starting of a new cycle, and return to passive state indicates end of the bus cycle. These status lines are encoded in table 1.3.

Table 2.3. Status lines

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	Indication
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

LOCK: This output pin indicates that other system bus masters will be prevented from gaining the system bus, while the LOCK signal is low. The LOCK signal is activated by the 'LOCK' prefix instruction and remains active until the completion of the next instruction. This floats to tri-state off during "hold acknowledge". When the CPU is executing a critical instruction which requires the system bus, the LOCK prefix instruction ensures that other processors connected in the system will not gain the control of the bus. The 8086, while executing the prefixed instruction, asserts the bus lock signal output, which may be connected to an external bus controller.

QS1, QS0-Queue Status: These lines give information about the status of the code prefetch queue. These are active during the CLK cycle after which the queue operation is performed. These are encoded as shown in Table 1.4.

Table 2.4. Queue Status

QS_1,	QS_0	Indication
0	0	No operation
0	1	First byte of opcode from the queue
1	0	Empty queue
1	1	Subsequent byte from the queue

This modification in a simple fetch and execute architecture of a conventional microprocessor offers an added advantage of *pipelined processing* of the instructions. The 8086 architecture has a 6-byte instruction prefetch queue. Thus even the largest (6- bytes) instruction can be prefetched from the memory and stored in the prefetch queue. This results in a faster execution of the instructions. In 8085, an instruction (opcode and operand) is fetched, decoded and executed and only after the execution of this instruction, the next one is fetched. By prefetching the instruction, there is a considerable speeding up in instruction execution in 8086. This scheme is known as *instruction pipelining.* At the starting the CS:IP is loaded with the required address from which the execution is to be started. Initially, the queue will be empty and the microprocessor starts a fetch operation to bring one byte (the first byte) of instruction code, if the CS:IP address is odd or two bytes at a time, if the CS:IP address is even. The first byte is a complete opcode in case of some instructions (one byte opcode instruction) and it is a part of opcode, in case of other instructions (two byte long

opcode instructions), the remaining part of opcode may lie in the second byte. But invariably the first byte of an instruction is an opcode. These opcodes along with data are fetched and arranged in the queue. When the first byte from the queue goes for decoding and interpretation, one byte in the queue becomes empty and subsequently the queue is updated. The microprocessor does not perform the next fetch operation till at least two bytes of the instruction queue are emptied. The instruction execution cycle is never broken for fetch operation. After decoding the first byte, the decoding circuit decides whether the instruction is of single opcode byte or double opcode byte. If it is single opcode byte, the next bytes are treated as data bytes depending upon the decoded instruction length, other wise, the next byte in the queue is treated as the second byte of the instruction opcode. The second byte is then decoded in continuation with the first byte to decide the instruction length and the number of subsequent bytes to be treated as instruction data. The queue is updated after every byte is read from the queue but the fetch cycle is initiated by BIU only if at least, two bytes of the queue are empty and the EU may be concurrently executing the fetched instructions. The next byte after the instruction is completed is again the first opcode byte of the next instruction. A similar procedure is repeated till the complete execution of the program. The main point to be noted here is, that the fetch operation of the next instruction is overlapped with the execution of the current instruction. As shown in the architecture, there are two separate units, namely, execution unit and bus interface unit. While the execution unit is busy in executing an instruction, after it is completely decoded, the bus interface unit may be fetching the bytes o(the next instruction from memory, depending upon the queue status. Figure 1.6 explains the queue operation.

RQ/GT0, RQ/GT1-ReQuest/Grant: These pins are used by other local bus masters, in maximum mode, to force the processor to release the local bus at the end of the processor's current bus cycle. Each of the pins is bidirectional with RQ/GT0 having higher priority than RQ/ GT1, RQ/GT pins have internal pull-up resistors and may be left unconnected. The request! grant sequence is as follows:

1. A pulse one clock wide from another bus master requests the bus access to 8086.

 2. During T4 (current) or T1 (next) clock cycle, a pulse one clock wide from 8086 to the requesting master, indicates that the 8086 has allowed the local bus to float and that it will enter the "hold acknowledge" state at next clock cycle. The CPU's bus interface unit is likely to be disconnected from the local bus of the system.

3. A one clock wide pulse from another master indicates to 8086 that the 'hold' request is about to end and the 8086 may regain control of the local bus at the next clock cycle. Thus each master to master exchange of the local bus is a sequence of 3 pulses. There must be at least one dead clock cycle after each bus exchange. The request and grant pulses are active low. For the bus requests those are received while 8086 is performing memory or I/O cycle, the granting of the bus is governed by the rules as discussed in case of HOLD, and HLDA in minimum mode.

2.2 Basic configurations : Read Write Timing Diagram
✓ **General Bus Operation**

The 8086 has a combined address and data bus commonly referred as a time multiplexed address and data bus. The main reason behind multiplexing address and data over the same pins is the maximum utilization of processor pins and it facilitates the use of 40 pin standard DIP package. The bus can be demultiplexed using a few latches and transreceivers, whenever required.

Basically, all the processor bus cycles consist of at least four clock cycles. These are referred to as T1, T2, T3, T4. The address is transmitted by the processor during T1, It is present on the bus only for one cycle. The negative edge of this ALE pulse is used to separate the address and the data or status information.

In maximum mode, the status lines S0, S1 and S2 are used to indicate the type of operation. Status bits S3 to S7 are multiplexed with higher order address bits and the BHE signal. Address is valid during T1 while status bits S3 to S7 are valid during T2 through T4.
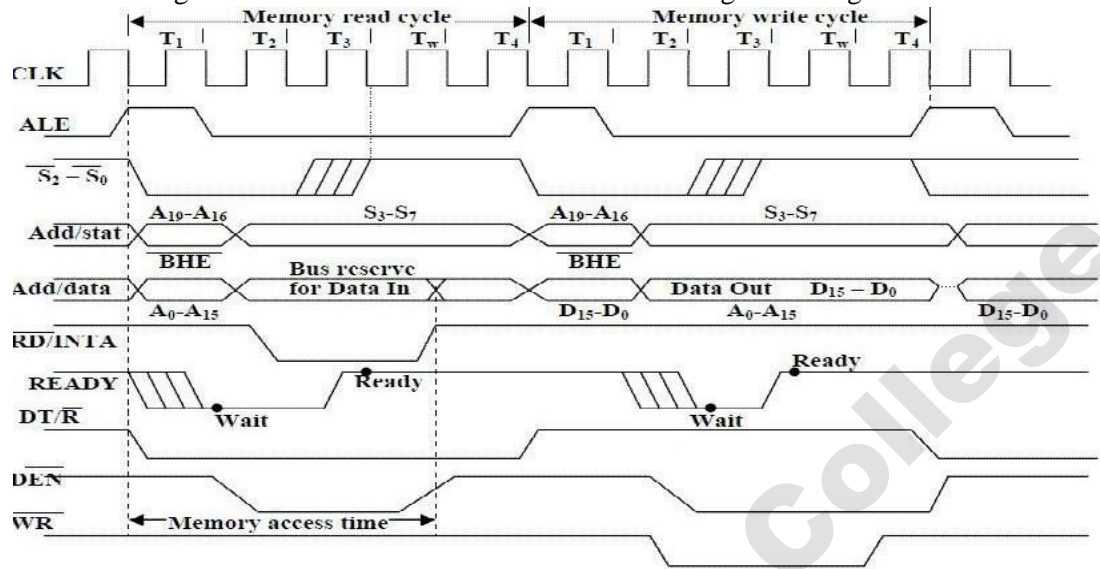
Fig.2.2. General Bus operation cycle

2.3 System Bus timings: Minimum mode 8086 system and timings

In a minimum mode 8086 system, the microprocessor 8086 is operated in minimum mode by strapping its MN/MX* pin to logic1. In this mode, all the control signals are given out by the microprocessor chip itself. There is a single microprocessor in the minimum mode system. The remaining components in the system are latches, transreceivers, clock generator, memory and I/O devices.

The opcode fetch and read cycles are similar. Hence the timing diagram can be categorized in two parts, the first is the timing diagram for read cycle and the second is the timing diagram for write cycle.

Fig 1.2 shows the read cycle timing diagram. The read cycle begins in T1 with the assertion of the address latch enable (ALE) signal and also M/IO* signal. During the negative going edge of this signal, the valid address is latched on the local bus. The BHE* and A0 signals address low, high or both bytes. From Tl to T4, the M/IO* signal indicates a memory or I/O operation. At T2 the address is removed from the local bus and is sent to the output. The bus is then tristated. The read (RD*) control signal is also activated in T2 .The read (RD) signal causes the addressed device to enable its data bus drivers. After RD* goes low, the valid data is available on the data bus. The addressed device will drive the READY line high, when the processor returns the read signal to high level, the addressed device will again tristate its bus drivers.

47
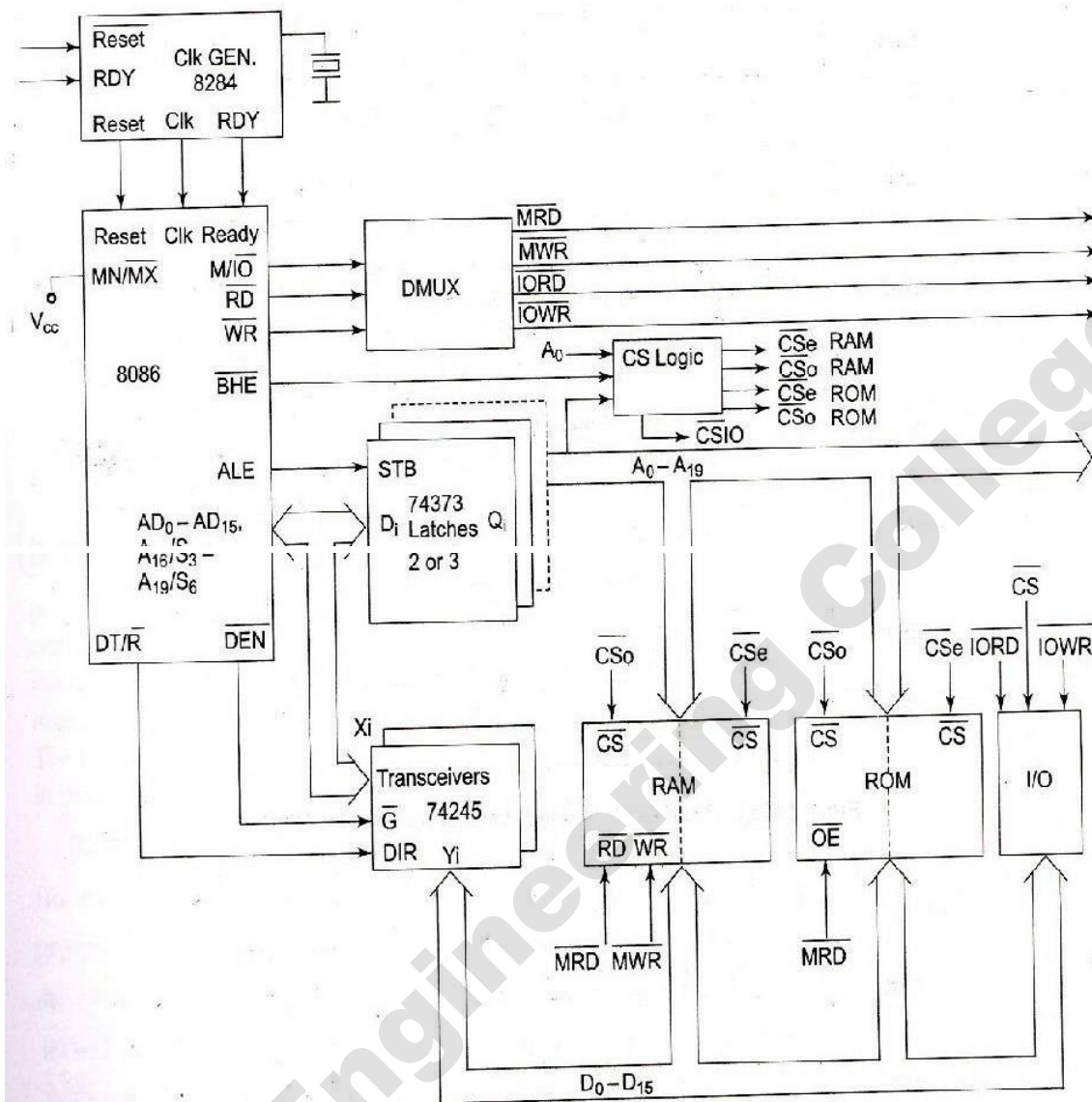
Fig 2.3. Minimum Mode 8086 System

A write cycle also begins with the assertion of ALE and the emission of the address. The M/IO* signal is again asserted to indicate a memory or I/O operation. In T2 after sending the address in Tl the processor sends the data to be written to the addressed location. The data remains on the bus until middle of T4 state. The WR* becomes active at the beginning ofT2 (unlike RD* is somewhat delayed in T2 to provide time for floating). The BHE* and A0 signals are used to select the proper byte or bytes of memory or I/O word to be read or written. The M/IO*, RD* and WR* signals indicate the types of data transfer as specified in Table

Table 2.5 Read write cycle

M/IO	RD	WR	Transfer Type
0	0	1	I/O read
0	1	0	I/O write
1	0	1	Memory read
1	1	0	Memory write

48
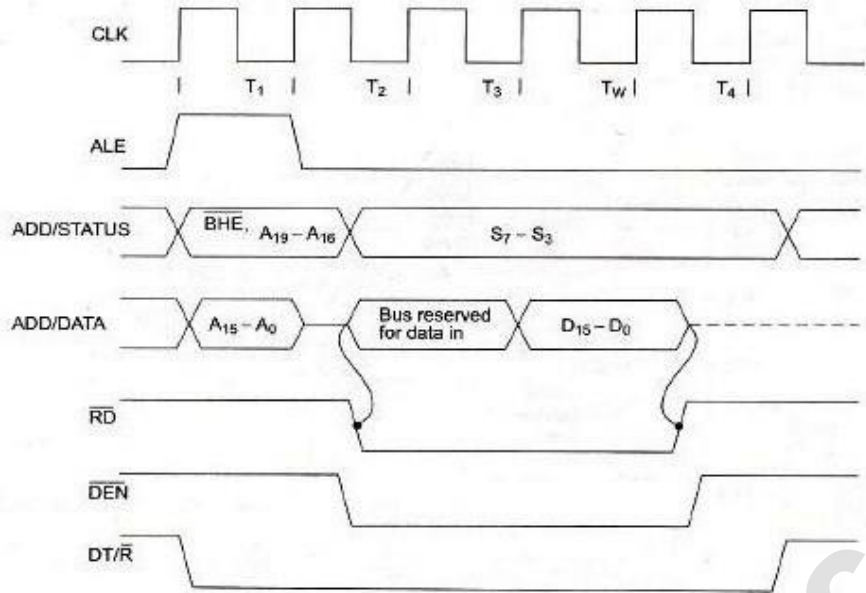
Fig. 2.4 Read cycle timing diagram for minimum mode

HOLD Response Sequence
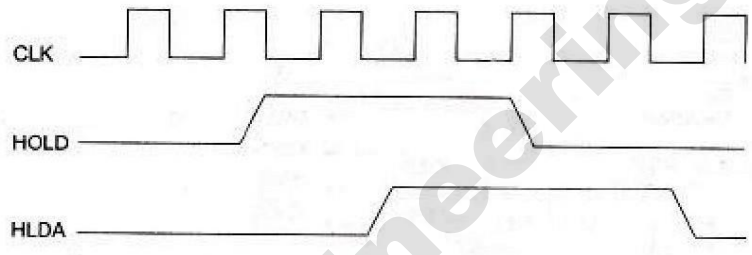
Fig 2.5 Bus request and busgrant timings in minimum mode system
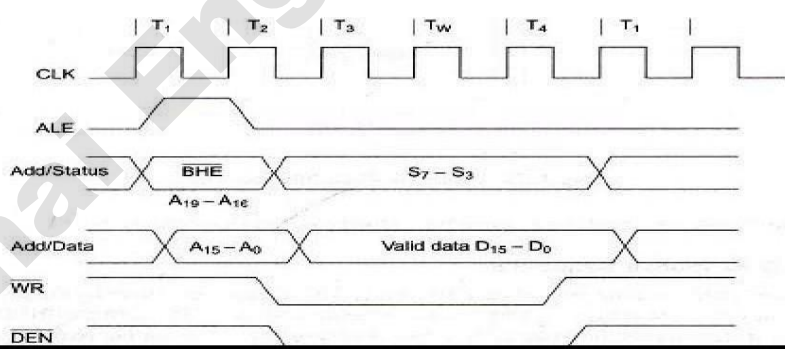
2.4 System Design using 8086: Maximum mode 8086 system and timings

In the maximum mode, the 8086 is operated by strapping the MN/MX* pin to ground. In this mode, the processor derives the status signals S2*, S1* and S0*. Another chip called bus controller derives the control signals using this status information. In the maximum mode, there may be more than one microprocessor in the system configuration.

The basic functions of the bus controller chip IC8288, is to derive control signals like RD* and WR* (for memory and I/O devices), DEN*, DT/R*, ALE, etc. using the information made available by the processor on the status lines. The bus controller chip has

input lines S2*, S1* and S0* and CLK. These inputs to 8288 are driven by the CPU. It derives the outputs ALE, DEN*, DT/R*, MWTC*, AMWC*, IORC*, IOWC* and AIOWC*. The AEN*, IOB and CEN pins are especially useful for multiprocessor systems. AEN* and IOB are generally grounded. CEN pin is usually tied to +5V.
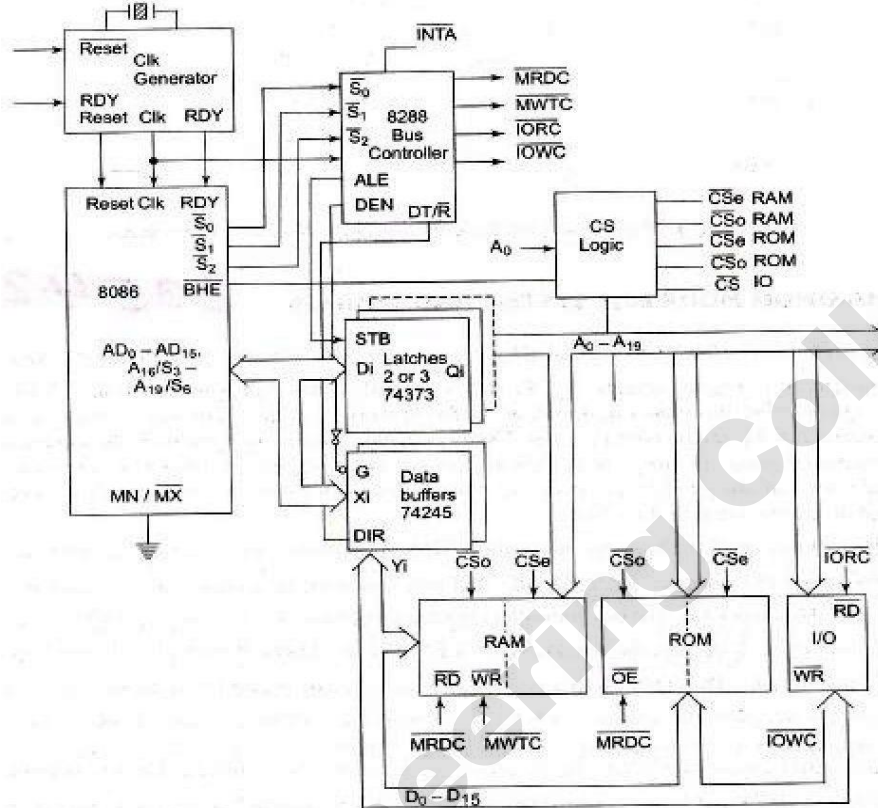
Fig. 2.6 Maximum mode configuiration

The significance of the MCE/PDEN* output depends upon the status of the IOB pin. If IOB is grounded, it acts as master cascade enable to control cascaded 8259A; else it acts as peripheral data enable used in the multiple bus configurations.

INTA* pin is used to issue two interrupt acknowledge pulses to the interrupt controller or to an interrupting device.

IORC*, IOWC* are I/O read command and I/O write command signals respectively.

These signals enable an IO interface to read or write the data from or to the addressed port. The MRDC*, MWTC* are memory read command and memory write command signals respectively and may be used as memory read and write signals. All these command signals instruct the memory to accept or send data from or to the bus.

For both of these write command signals, the advanced signals namely AIOWC* and AMWTC* are available. They also serve the same purpose, but are activated one clock cycle earlier than the IOWC* and MWTC* signals, respectively. The maximum mode system is shown in fig. 2.1.

The maximum mode system timing diagrams are also divided in two portions as read (input) and write (output) timing diagrams. The address/data and address/status timings are similar to the minimum mode. ALE is asserted in T1, just like minimum mode. The only difference lies in the status signals used and the available control and advanced command signals. The fig. 1.2 shows the maximum mode timings for the read operation while the fig. 1.3 shows the same for the write operation.
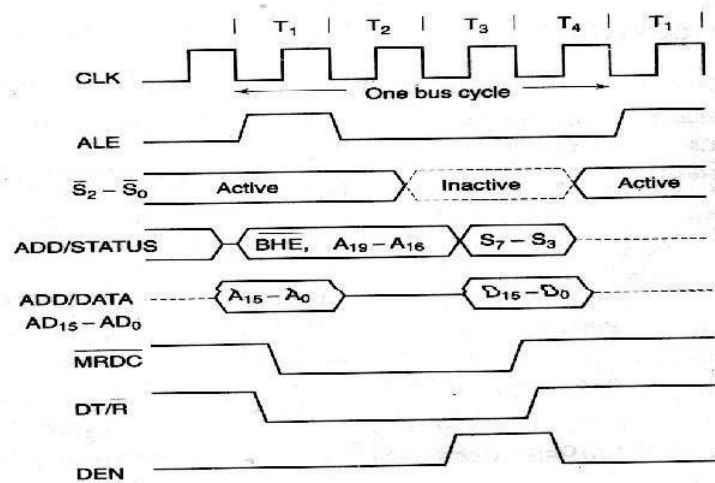
Fig.2.7 Memory read cycle
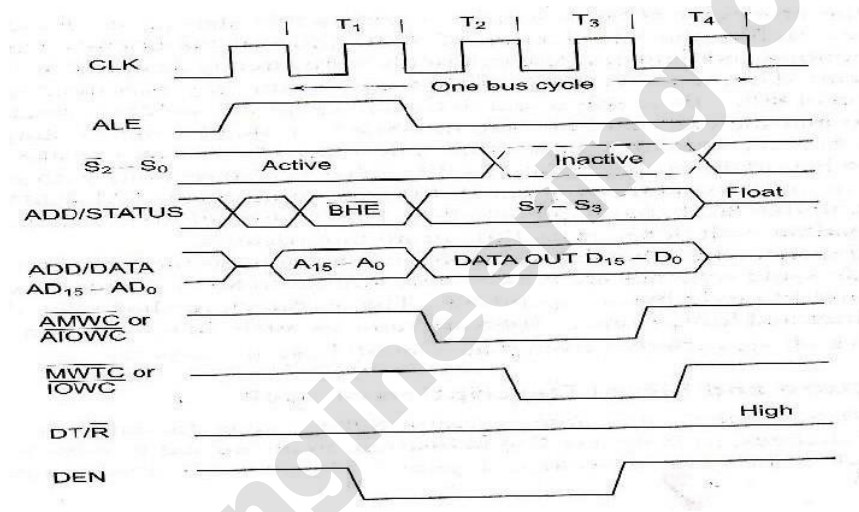
Fig.2.8 Memory write cycle
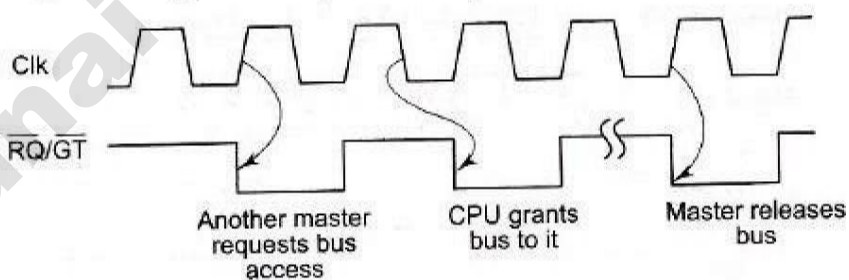
Fig 2.9 RG*/GT* Timings in maximum mode

2.5 IO programming

On the 8086, all programmed communications with the I/O ports is done by the IN and OUT instructions defined in Fig. 6-2.

✓ IN and OUT instructions

Name	Mnemonic and Format	Description
Input		
Long form, byte	IN AL, PORT	(AL) <- (PORT)
Long form, word	IN AX, PORT	(AX) <- (PORT+1: PORT)
Short form, byte	IN AL, DX	(AL) <- ((DX))
Short form, word	IN AX, DX	(AX) <- ((DX) + 1: (DX))
Output		
Long form, byte	OUT PORT, AL	(PORT) <- (AL)
Long form, word	OUT PORT, AX	(PORT+1: PORT) <- (AX)
Short form, byte	OUT DX, AL	((DX)) <- (AL)
Short form, word	OUT DX, AX	((DX)+1: (DX)) <- (AX)

Note: PORT is a constant ranging from 0 to 255
Flags: No flags are affected
Addressing modes: Operands are limited as indicated above.

If the second operand is DX, then there is only one byte in the instruction and the contents of DX are used as the port address. Unlike memory addressing, the contents of DX are not modified by any segment register. This allows variable access to I/O ports in the range 0 to 64K. The machine language code for the IN instruction is:

```
0-Byte transfer                                    ┌─Present only
1-Word transfer  ┐                                 │ in long form
                 ├────────┐
  ┌───┬───┬───┬─┬─┬───┬───┬───┐   ┌───────────────┐
  │ 1 │ 1 │ 1 │ 0 │ 1 │ 0 │ W │   │  Port Address │
  └───┴───┴───┴─┴─┴───┴───┴───┘   └───────────────┘
              └─ {0—Long form
                 {1—Short form
```
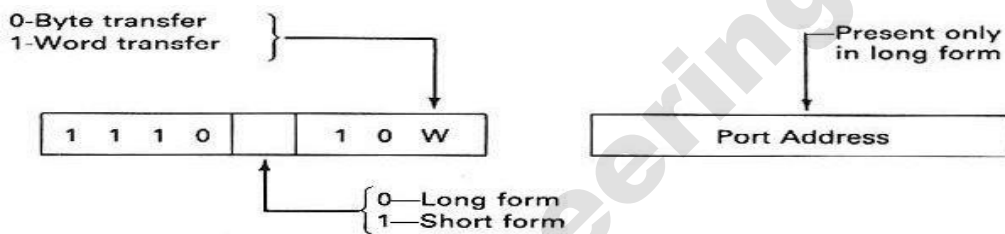
Although AL or AX is implied as the first operand in an IN instruction, either AL or AX must be specified so that the assembler can determine the W-bit.

Similar comments apply to the OUT instruction except that for it the port address is the destination and is therefore indicated by the first operand, and the second operand is either AL or AX. Its machine code is:

```
  ┌───┬───┬───┬─┬─┬───┬───┬───┐   ┌───────────────┐
  │ 1 │ 1 │ 1 │ 0 │ 1 │ 1 │ W │   │  Port Address │
  └───┴───┴───┴─┴─┴───┴───┴───┘   └───────────────┘
              └─ {0—Long form           └─ {Present only
                 {1—Short form             {in long form
```
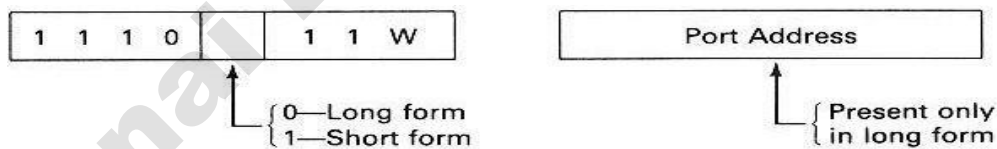
Note that if the long form of the IN or OUT instruction is used the port address must be in the range 0000 to 00FF, but for the short form it can be any address in the range 0000 to FFFF (i.e. any address in the I/O address space). Neither IN nor OUT affects the flags.

The IN instruction may be used to input data from a data buffer register or the status from a status register. The instructions

IN AX, 28H
MOV DATA_WORD, AX

would move the word in the ports whose address are 0028 and 0029 to the memory location DATA_WORD.

52

2.6 Introduction to Multiprogramming

In order to adapt to as many situations as possible both the 8086 and 8088 have been given two modes of operation, the minimum mode and the maximum mode. The minimum mode is used for a small system with a single processor, a system in which the 8086/8088 generates all the necessary bus control signals directly (thereby minimizing the required bus control logic). The maximum mode is for medium-size to large systems, which often include two or more processors.

2.7 System Bus structure
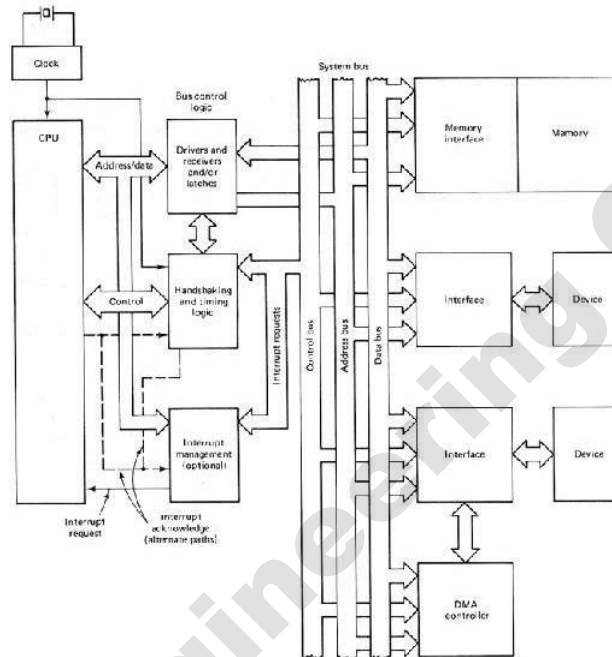
Fig.2.10 Typical system Bus architecture

Table 2.6 Pins for read/ write operation

Operation	/BHE	A0	Data pins used
Write/read a word at an even address	0	0	AD15-AD0
Write/read a byte at an even address	1	0	AD7-AD0
Write/read a byte at an odd address	0	1	AD15-AD8
Write/read a word at an odd address	0	1	AD15-AD8 (first bus cycle: puts the least significant data byte on AD15-AD8)
	1	0	AD7-AD0 (Next bus cycle: puts the most significant data byte on AD7-AD0)

2.8 Multiprocessor Systems

Multiprocessor Systems refer to the use of multiple processors that execute instructions simultaneously and communicate using mailboxes and semaphores

Maximum mode of 8086 is designed to implement 3 basic multiprocessor configurations:

1. Coprocessor (8087)
2. Closely coupled (dedicated I/O processor: 8089)

3. Loosely coupled (Multi bus)

Coprocessors and closely coupled configurations are similar - both the CPU and the external processor share:

    Memory
    I/O system
    Bus & bus control logic
    Clock generator

2.9 Coprocessor configurations

Coprocessor Configuration:

    WAIT instruction allows the processor to synchronize itself with external hardware, eg., waiting for 8087 math co-processor.

    When the CPU executes WAIT

    waiting state.

TEST input is asserted (low), the waiting state is completed and execution will resume.
ESC instruction:
ESC opcode, operand, opcode: immediate value recognizable to a coprocessor as an instruction opcode
Operand: name of a register or a memory address (in any mode)

    When the CPU executes the ESC instruction, the processor accesses the memory operand by placing the address on the address bus.

    If a coprocessor is configured to share the system bus, it will recognize the ESC instruction and therefore will get the opcode and the operand
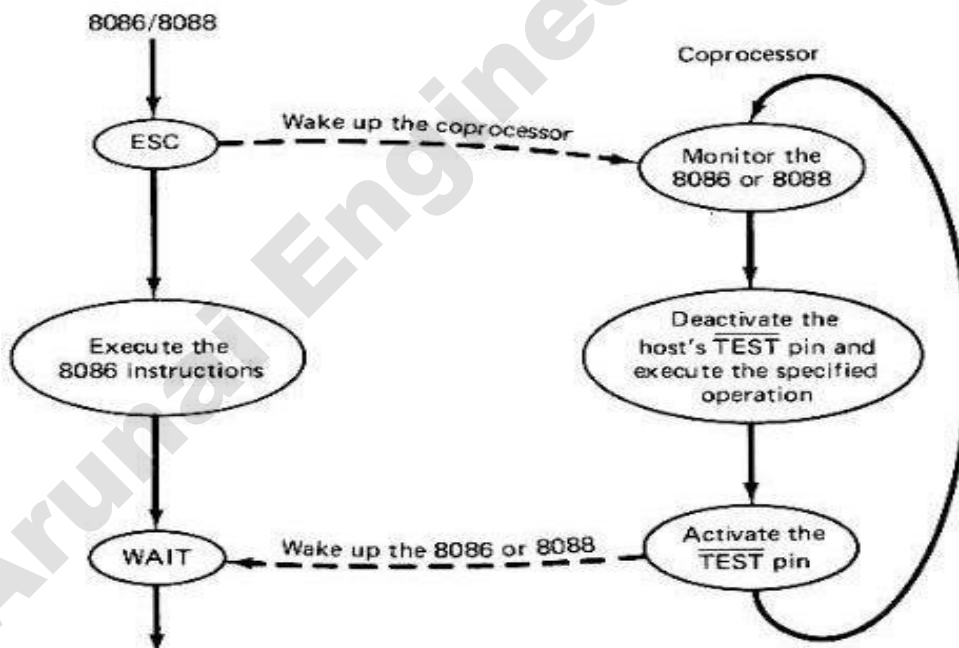
Fig. 2.11 Synchronisation between the 8086 and its coprocesssor

54

(a) Memory operand is used

(b) No memory operand is used
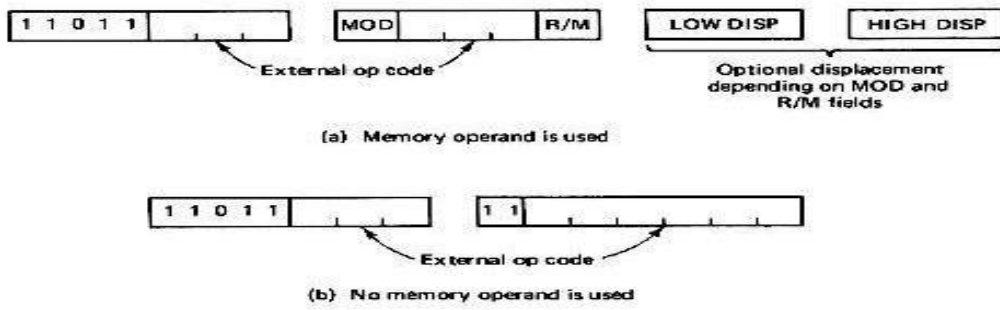
Fig 2.12 Machine code formats for the ESC instruction
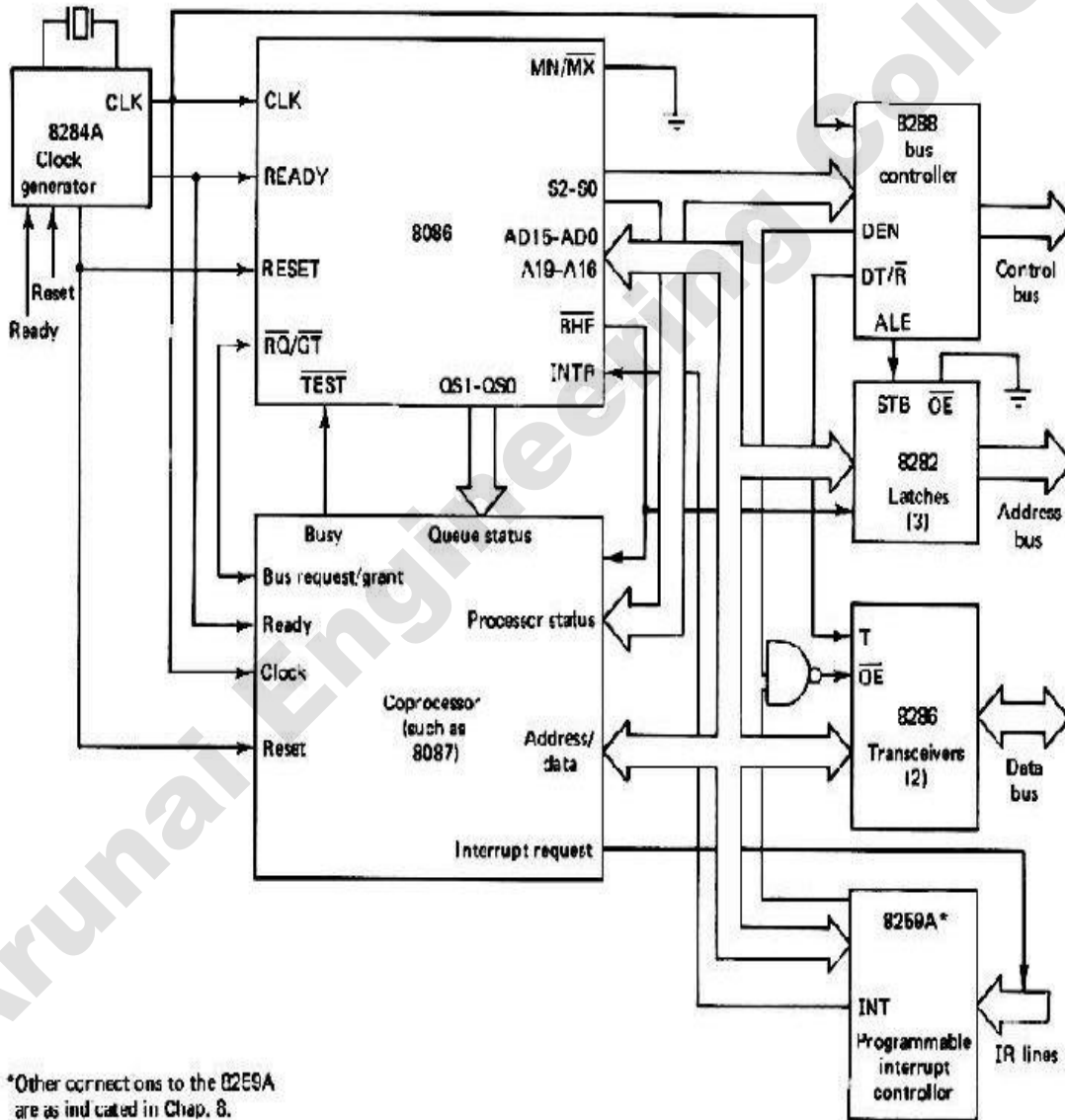
Fig.2.13   Coprocessor configuration
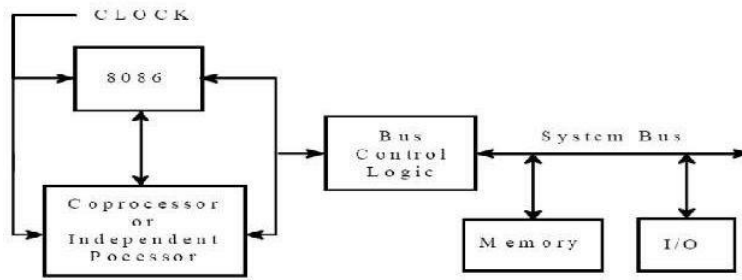
Fig.2.14 closely coupled configuration

✓ Coprocessor cannot take control of the bus, it does everything through the CPU
✓ Closely Coupled processor may take control of the bus independently - 8089 shares CPU's clock and bus control logic.
✓ communication with host CPU is by way of shared memory
✓ host sets up a message (command) in memory
✓ independent processor interrupts host on completion
✓ Two 8086's cannot be closely coupled

2.10 Closely Coupled Configuration
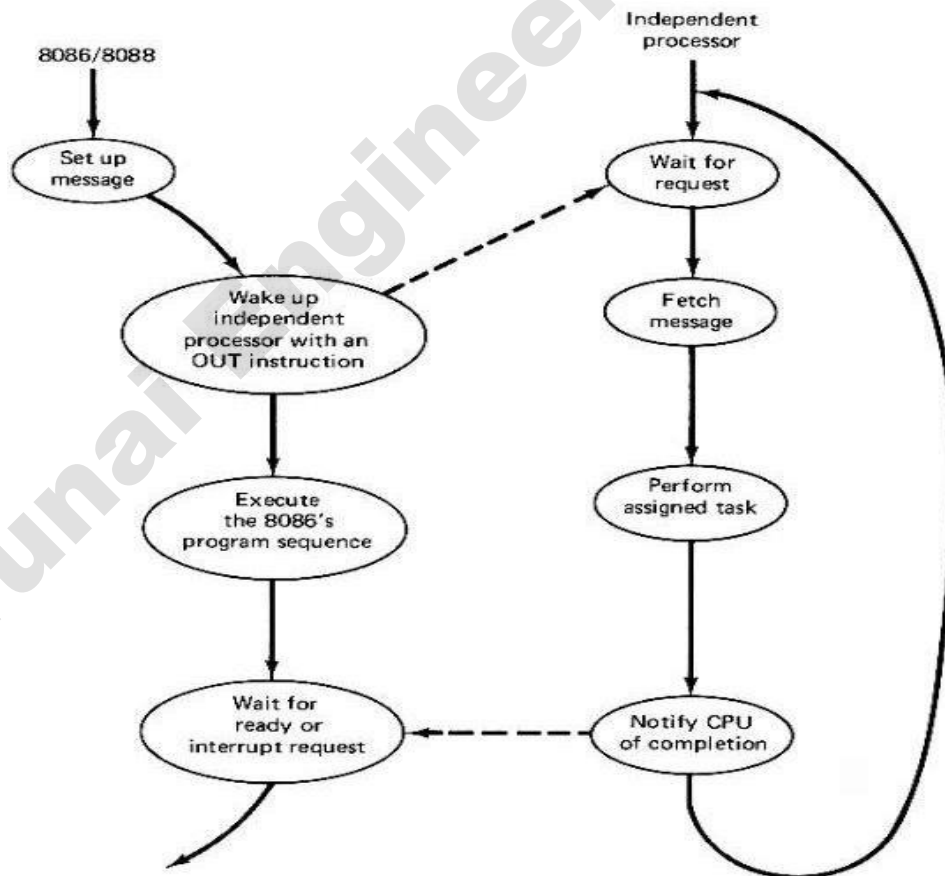
Fig.2.15 Interprocessor communication through shared memory
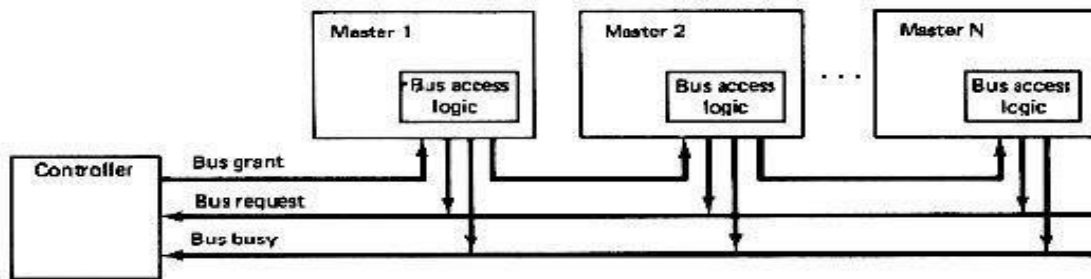
(a) Daisy chain method
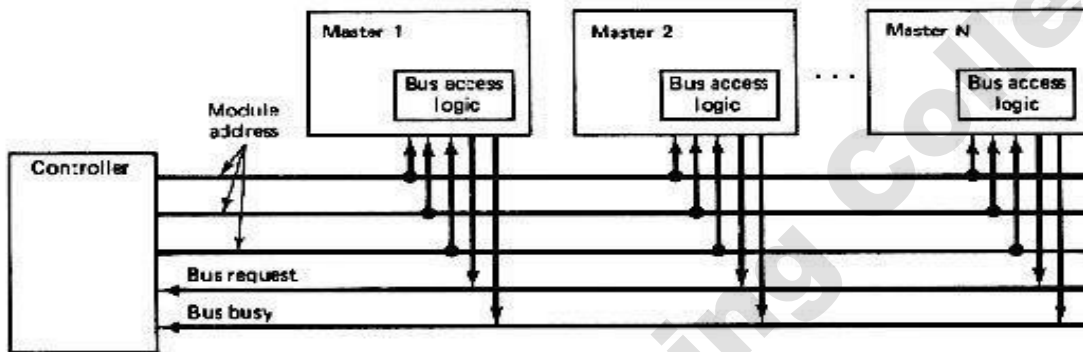
(b) Polling method
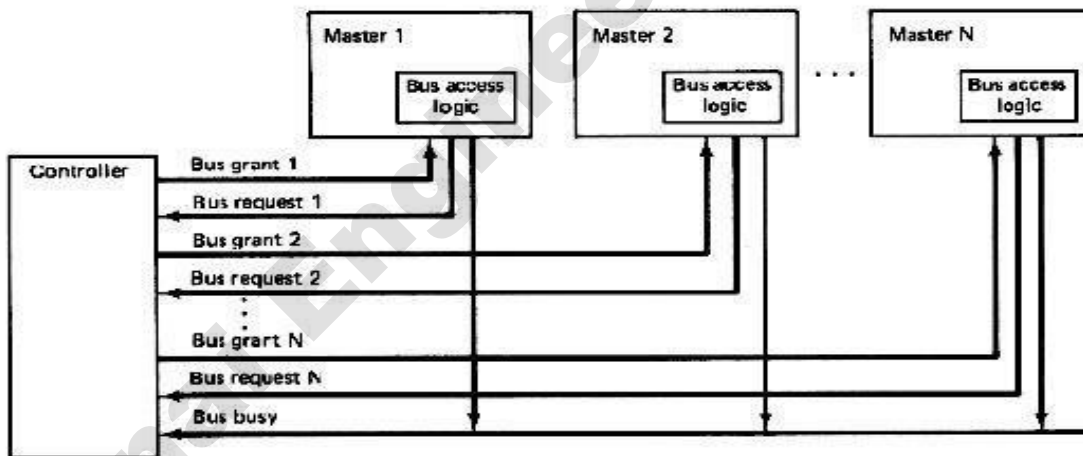
(c) Independent requests method

Fig.2.16 Bus allocation schemes

Loosely Coupled Configurations:

A loosely coupled configuration provides the following advantages:

1. High system throughput can be achieved by having more than one CPU.

2. The system can be expanded in a modular form. Each bus master module is an independent unit and normally resides on a separate PC board. Therefore, a bus master module can be added or removed without affecting the other modules in the system.

3. A failure in one module normally does not cause a breakdown of the entire system and the faulty module can be easily detected and replaced.

57

4. Each bus master may have a local bus to access dedicated memory or I/O devices so that a greater degree of parallel processing can be achieved. More than one bus master module may have access to the shared system bus

Extra bus control logic must be provided to resolve the bus arbitration problem. The extra logic is called bus access logic and it is its responsibility to make sure that only one bus master at a time has control of the bus.

Simultaneous bus requests are resolved on a **priority basis:** There are three schemes for establishing priority:

      1. Daisy chaining.
      2. Polling.
      3. Independent requesting

2.11 Introduction to Advanced processors: 80286 Microprocessor

2.11.1 Salient Features of 80286

✓ The 80286 is the first member of the family of advanced microprocessors with memory management and protection abilities. The 80286 CPU, with its 24-bit address bus is able to address 16 Mbytes of physical memory. Various versions of 80286 are available that runs on 12.5 MHz, 10 MHz and 8 MHz clock frequencies. 80286 is upwardly compatible with 8086 in terms of instruction set.

    ✓ 80286 has two operating modes namely real address mode and virtual address mode. In real address mode, the 80286 can address upto 1Mb of physical memory address like 8086. In virtual address mode, it can address up to 16 Mb of physical memory address space and 1 GB of virtual memory address space.

    ✓ The instruction set of 80286 includes the instructions of 8086 and 80186. 80286 has some extra instructions to support operating system and memory management. In real address mode, the 80286 is object code compatible with 8086. In protected virtual address mode, it is source code compatible with 8086. The performance of 80286 is five times faster than the standard 8086.

Need for Memory Management

The part of main memory in which the operating system and other system programs are stored is not accessible to the users. It is required to ensure the smooth execution of the running process and also to ensure their protection. The memory management which is an important task of the operating system is supported by a hardware unit called memory management unit.

Swapping in of the Program

Fetching of the application program from the secondary memory and placing it in the physical memory for execution by the CPU.

Swapping out of the executable Program

Saving a portion of the program or important results required for further execution back to the secondary memory to make the program memory free for further execution of another required portion of the program.

Concept of Virtual Memory

Large application programs requiring memory much more than the physically available 16 Mbytes of memory, may be executed by diving it into smaller segments. Thus for the user, there exists a very large logical memory space which is not actually available. Thus there exists a virtual memory which does not exist physically in a system. This complete process of virtual memory management is taken care of
by the 80286 CPU and the supporting operating system.

Internal Architecture of 80286

Register Organization of 80286

The 80286 CPU contains almost the same set of registers, as in 8086, namely

      1. Eight 16-bit general purpose registers
      2. Four 16-bit segment registers

Register Set of 80286
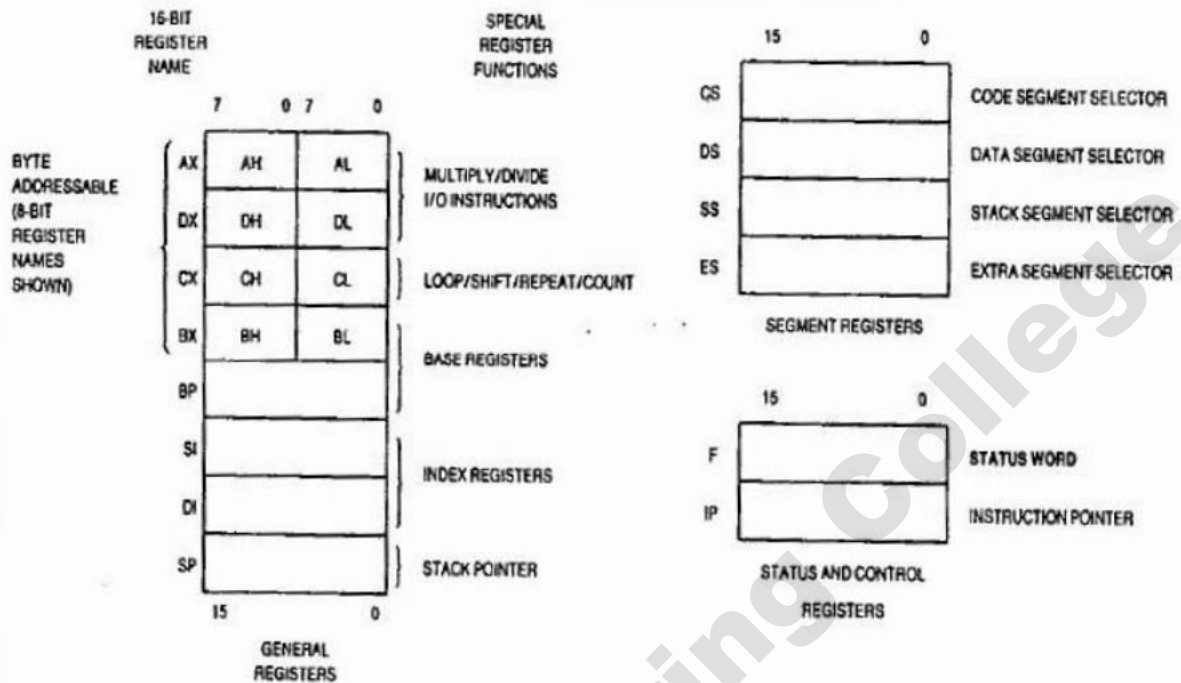
Fig.2.17 Register set of 80286

Fig. 8.1 Register Set of 80286

Fig 2.18 Flag registers

3. Status and control registers

4. Instruction Pointer

D_2, D_4, D_6, D_7 and D_{11} are called as status flag bits. The bits D_8 (TF) and D_9 (IF) are used for controlling machine operation and thus they are called control flags. The additional fields available in 80286 flag registers are:

1. IOPL - I/O Privilege Field (bits D12 and D13)

2. NT - Nested Task flag (bit D14)

3. PE - Protection Enable (bit D16)

4. MP - Monitor Processor Extension (bit D17)

5. EM - Processor Extension Emulator (bit D18)

6. TS – Task Switch (bit D19)

Protection Enable flag places the 80286 in protected mode, if set. This can only be cleared by resetting the CPU. If the Monitor Processor Extension flag is set, allows WAIT instruction to generate a processor extension not present exception.

Processor Extension Emulator flag if set, causes a processor extension absent exception and permits the emulation of processor extension by the CPU.

Task Switch flag if set, indicates the next instruction using extension will generate exception 7, permitting the CPU to test whether the current processor extension is for the current task.

Machine Status Word (MSW)

The machine status word consists of four flags – PE, MO, EM and TS of the four lower

59

order bits D19 to D16 of the upper word of the flag register. The LMSW and SMSW instructions are available in the instruction set of 80286 to write and read the MSW in real address mode.

8.2.2 Internal Block Diagram of 80286

2.11.4 Internal Block Diagram of 80286
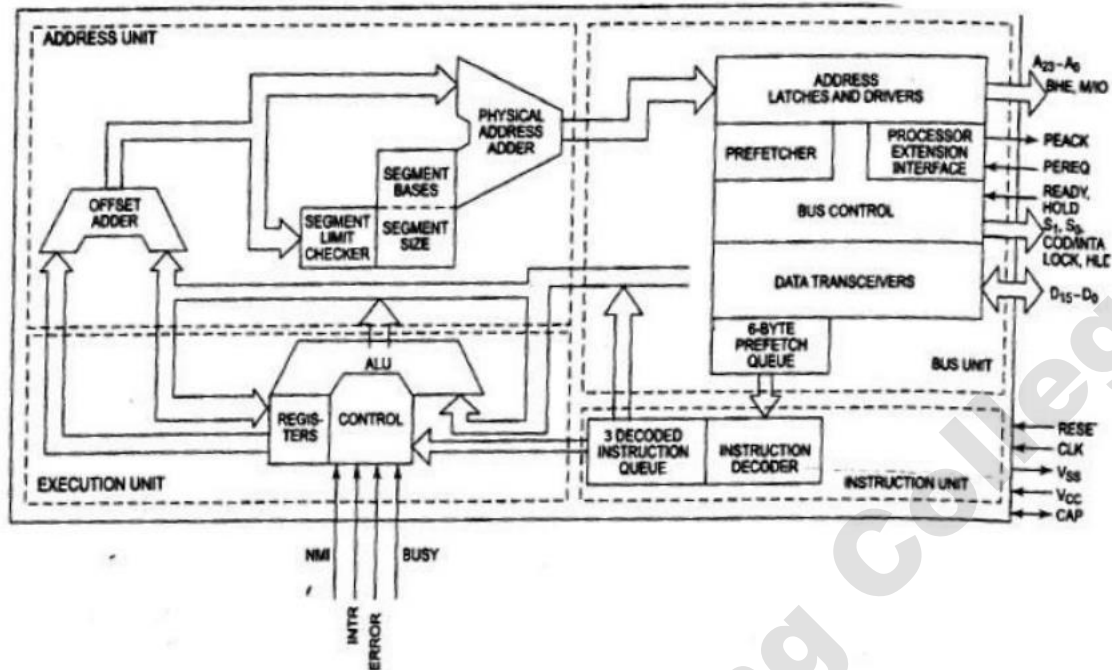
Fig. 2.19 Internal Block diagram of 80286

The CPU contain four functional blocks

      1. Address Unit (AU)
      2. Bus Init (BU)
      3. Instruction Unit (IU)
      4. Execution Unit (EU)

The address unit is responsible for calculating the physical address of instructions and data that the CPU wants to access. Also the address lines derived by this unit may be used to address different peripherals. The physical address computed by the address unit is handed over to the bus unit (BU) of the CPU. Major function of the bus unit is to fetch instruction bytes from the memory. Instructions are fetched in advance and stored in a queue to enable faster execution of the instructions. The bus unit also contains a bus control module that controls the prefetcher module. These prefetched instructions are arranged in a 6-byte instructions queue. The 6-byte prefetch queue forwards the instructions arranged in it to the **instruction unit** (IU). The instruction unit accepts instructions from the prefetch queue and an instruction decoder decodes

them one by one. The decoded instructions are latched onto a decoded instruction queue. The output of the decoding circuit drives a control circuit in the **execution unit,** which is responsible for executing the instructions received from decoded instruction queue. The decoded instruction queue sends the data part of the instruction over the data bus. The EU contains the register bank used for storing the data as scratch pad, or used as special purpose registers. The ALU, the heart of the EU, carries out all the arithmetic and logical operations and sends the results over the data bus or back to the register bank.

Interrupts of 80286

The Interrupts of 80286 may be divided into three categories,

1. External or hardware interrupts
2. INT instruction or software interrupts
3. Interrupts generated internally by exceptions

While executing an instruction, the CPU may sometimes be confronted with a special situation because of which further execution is not permitted. While trying to execute a

divide by zero instruction, the CPU detects a major error and stops further execution. In this case, we say that an exception has been generated. In other words, an instruction exception is an unusual situation encountered during execution of an instruction that stops further execution. The return address from an exception, in most of the cases, points to the instruction that caused the exception.

As in the case of 8086, the interrupt vector table of 80286 requires 1Kbytes of space for storing 256, four-byte pointers to point to the corresponding 256 interrupt service routines (lSR). Each pointer contains a 16-bit offset followed by a 16-bit segment selector to point to a particular ISR. The calculation of vector pointer address in the interrupt vector table from the (8-bit) INT type is exactly similar to 8086.

Like 8086, the 80286 supports the software interrupts of type 0 (INT 00) to type FFH (INT FFH).

Maskable Interrupt INTR: This is a maskable interrupt input pin of which the INT type is to be provided by an external circuit like an interrupt controller. The other functional details of this interrupt pin are exactly similar to the INTR input of 8086.

Non-Maskable Interrupt NMI: It has higher priority than the INTR interrupt.

Whenever this interrupt is received, a vector value of 02 is supplied internally to calculate the pointer to the interrupt vector table. Once the CPU responds to a NMI request, it does not serve any other interrupt request (including NMI). Further it does not serve the processor extension (coprocessor) segment overrun interrupt, till either it executes IRET or it is reset. To start with, this clears the IF flag which is set again with the execution of IRET, i.e. return from interrupt.

Single Step Interrupt

As in 8086, this is an internal interrupt that comes into action, if *trap* flag (TF) of 80286 is set. The CPU stops the execution after each instruction cycle so that the register contents (including flag register), the program status word and memory, etc. may be examined at the end of each instruction execution. This interrupt is useful for troubleshooting the software. An interrupt vector type 01 is reserved for this interrupt.

Interrupt Priorities:

If more than one interrupt signals occur simultaneously, they are processed according to their priorities as shown below:

Order	Interrupt
1	Interrupt exception
2	Single step
3	NMI
4	Processor extension segment overrun
5	INTR
6	INT instruction

FUNCTION	Interrupt Number
Divide error exception	0
Single step interrupt	1
NMI interrupt	2
Breakpoint interrupt	3
INTO detected overflow exception	4
BOUND range exceeded exception	5
Invalid opcode exception	6
Processor extension not available exception	7
Intel reserved, do not use	8-15
Processor extension error interrupt	16
Intel reserved, do not use	17-31
User defined	32-255

Signal Description of 80286

CLK: This is the system clock input pin. The clock frequency applied at this pin is divided by two internally and is used for deriving fundamental timings for basic operations of the circuit. The clock is generated using 8284 clock generator.

D_{15}-D_0: These are sixteen bidirectional data bus lines.

A_{23}-A_0: These are the physical address output lines used to address memory or I/O devices. The address lines A23 - A16 are zero during I/O transfers

BHE: This output signal, as in 8086, indicates that there is a transfer on the higher byte of the data bus (D15 – D8) .

S1 , S0: These are the active-low status output signals which indicate initiation of a bus cycle and with M/IO and COD/INTA, they define the type of the bus cycle.

M/ IO: This output line differentiates memory operations from I/O operations. If this signal is it "0" indicates that an I/O cycle or INTA cycle is in process and if it is "1" it indicates that a memory or a HALT cycle is in progress.

COD/ INTA: This output signal, in combination with M/ IO signal and S1 , S0 distinguishes different memory, I/O and INTA cycles.

LOCK: This active-low output pin is used to prevent the other masters from gaining the control of the bus for the current and the following bus cycles. This pin is activated by a "LOCK" instruction prefix, or automatically by hardware during XCHG, interrupt acknowledge or descriptor table access

READY This active-low input pin is used to insert wait states in a bus cycle, for interfacing low speed peripherals. This signal is neglected during HLDA cycle.

HOLD and HLDA This pair of pins is used by external bus masters to request for the control of the system bus (HOLD) and to check whether the main processor has granted the control (HLDA) or not, in the same way as it was in 8086.

INTR: Through this active high input, an external device requests 80286 to suspend the current instruction execution and serve the interrupt request. Its function is exactly similar to that of INTR pin of 8086.

NMI: The Non-Maskable Interrupt request is an active-high, edge-triggered input that is equivalent to an INTR signal of type 2. No acknowledge cycles are needed to be carried out.

PEREG and **PEACK** (**Processor Extension Request and Acknowledgement**)

Processor extension refers to coprocessor (80287 in case of 80286 CPU). This pair of pins extends the memory management and protection capabilities of 80286 to the processor extension 80287. The PEREQ input requests the 80286 to perform a data operand transfer

for a processor extension. The PEACK active-low output indicates to the processor extension that the requested operand is being transferred.

BUSY and **ERROR:** Processor extension BUSY and ERROR active-low input signals indicate the operating conditions of a processor extension to 80286. The BUSY goes low, indicating 80286 to suspend the execution and wait until the BUSY become inactive. In this duration, the processor extension is busy with its allotted job. Once the job is completed the processor extension drives the BUSY input high indicating 80286 to continue with the program execution. An active ERROR signal causes the 80286 to perform the processor extension interrupt while executing the WAIT and ESC instructions. The active ERROR signal indicates to 80286 that the processor extension has committed a mistake and hence it is reactivating the processor extension interrupt.

CAP: A 0.047 μf, 12V capacitor must be connected between this input pin and ground to filter the output of the internal substrate bias generator. For correct operation of 80286 the capacitor must be charged to its operating voltage. Till this capacitor charges to its full capacity, the 80286 may be kept stuck to reset to avoid any spurious activity.

V$_{ss}$: This pin is a system ground pin of 80286.

V$_{cc}$: This pin is used to apply +5V power supply voltage to the internal circuit of 80286.

RESET The active-high RESET input clears the internal logic of 80286, and reinitializes it

RESET The active-high reset input pulse width should be at least 16 clock cycles. The 80286 requires at least 38 clock cycles after the trailing edge of the RESET input signal, before it makes the first opcode fetch cycle.

Real Address Mode

- Act as a fast 8086
- Instruction set is upwardly compatible
- It address only 1 M byte of physical memory using A$_0$-A$_{19}$.
- In real addressing mode of operation of 80286, it just acts as a fast 8086. The instruction set is upward compatible with that of 8086.

The 80286 addresses only 1Mbytes of physical memory using A$_0$- A$_{19}$. The lines A$_{20}$-A$_{23}$ are not used by the internal circuit of 80286 in this mode. In real address mode, while addressing the physical memory, the 80286 uses BHE along with A$_0$- A$_{19}$. The 20-bit physical address is again formed in the same way as that in 8086.

The contents of segment registers are used as segment base addresses. The other registers, depending upon the addressing mode, contain the offset addresses. Because of extra pipelining and other circuit level improvements, in real address mode also, the 80286 operates at a much faster rate than 8086, although functionally they work in an identical fashion. As in 8086, the physical memory is organized in terms of segments of 64Kbyte maximum size.

An exception is generated, if the segment size limit is exceeded by the instruction or the data. The overlapping of physical memory segments is allowed to minimize the memory requirements for a task. The 80286 reserves two fixed areas of physical memory for system initialization and interrupt vector table. In the real mode the first 1Kbyte of memory starting from address 0000H to 003FFH is reserved for interrupt vector table. Also the addresses from FFFF0H to FFFFFH are reserved for system initialization.

The program execution starts from FFFFH after reset and initialization. The interrupt vector table of 80286 is organized in the same way as that of 8086. Some of the interrupt types are reserved for exceptions, single-stepping and processor extension segment overrun, etc

When the 80286 is reset, it always starts the execution in real address mode. In real address mode, it performs the following functions: it initializes the IP and other registers of 80286, it prepares for entering the protected virtual address mode.
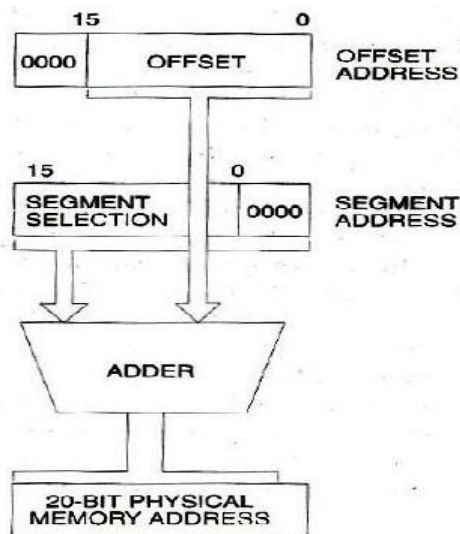
Fig 2.20 Real Address calculation

Protected Virtual Address Mode (PVAM)

80286 is the first processor to support the concepts of virtual memory and memory management. The virtual memory does not exist physically it still appears to be available within the system. The concept of VM is implemented using Physical memory that the CPU can directly access and secondary memory that is used as a storage for data and program, which are stored in secondary memory initially.

The Segment of the program or data required for actual execution at that instant is fetched from the secondary memory into physical memory. After the execution of this fetched segment, the next segment required for further execution is again fetched from the secondary memory, while the results of the executed segment are stored back into the secondary memory for further references. This continues till the complete program is executed

During the execution the partial results of the previously executed portions are again fetched into the physical memory, if required for further execution. The procedure of fetching the chosen program segments or data from the secondary storage into physical memory is called *swapping*. The procedure of storing back the partial results or data back on the secondary storage is called *unswapping*. The virtual memory is allotted per task.

The 80286 is able to address 1 G byte (2_{30} bytes) of virtual memory per task. The complete virtual memory is mapped on to the 16Mbyte physical memory. If a program larger than 16Mbyte is stored on the hard disk and is to be executed, if it is fetched in terms of data or program segments of less than 16Mbyte in size into the program memory by swapping sequentially as per sequence of execution.

Whenever the portion of a program is required for execution by the CPU, it is fetched from the secondary memory and placed in the physical memory is called *swapping in* of the program. A portion of the program or important partial results required for further execution, may be saved back on secondary storage to make the PM free for further execution of another required portion of the program is called *swapping out* of the executable program.

80286 uses the 16-bit content of a segment register as a selector to address a descriptor stored in the physical memory. The descriptor is a block of contiguous memory locations containing information of a segment, like segment base address, segment limit, segment type, privilege level, segment availability in physical memory, descriptor type and segment use another task.

UNIT –III
I/O INTERFACING

3.1 Memory Devices and Interfacing

Any application of a microprocessor based system requires the transfer of data between external circuitry to the microprocessor and microprocessor to the external circuitry. Most of the peripheral devices are designed and interfaced with a CPU either to enable it to communicate with the user or an external process and to ease the circuit operations so that the microprocessor works more efficiently.

The use of peripheral integrated devices simplifies both the hardware circuits and software considerable. The following are the devices used in interfacing of Memory and General I/O devices
• 74LS138 (Decoder / Demultiplexer).
• 74LS373 / 74LS374 3-STATE Octal D-Type Transparent Latches.
• 74LS245 Octal Bus Tran iver: 3-State.

74LS138 (Decoder / Demultiplexer)

The LS138 is a high speed 1-of-8 Decoder/ Demultiplexer fabricated with the low power Schottky barrier diode process. The decoder accepts three binary weighted inputs (A0, A1, A2) and when enabled provides eight mutually exclusive active LOW Outputs (O0–O7).

The LS138 can be used as an 8-output demultiplexer by using one of the active LOW Enable inputs as the data input and the other Enable inputs as strobes. The Enable inputs which are not used must be permanently tied to their appropriate active HIGH or active LOW state.

Fig. 3.1 Pin diagram of 74138

74LS373 / 74LS374 3-STATE Octal D-Type Transparent Latches and Edge-Triggered Flip-Flops

These 8-bit registers feature totem-pole 3-STATE outputs designed specifically for implementing buffer registers, I/O ports, bidirectional bus drivers, and working registers. The eight latches of the 74LS373 are transparent D type latches meaning that while the enable (G) is HIGH the Q outputs will follow the data (D) inputs.

When the enable is taken LOW the output will be latched at the level of the data that was

set up. The eight flip-flops of the 74LS374 are edge-triggered D-type flip flops. On the positive transition of the clock, the Q outputs will be set to the logic states that were set up at the D inputs.

Main Features
- Choice of 8 latches or 8 D-type flip-flops in a single package
- 3-STATE bus-driving outputs
- Full parallel-access for loading
- Buffered control inputs
- P-N-P inputs reduce D-C loading on data lines
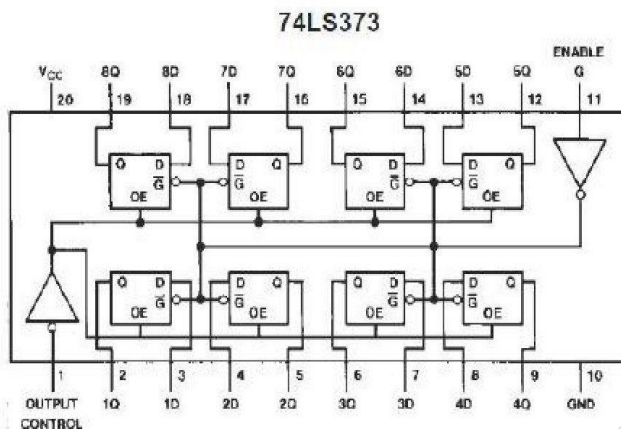
Fig. 3.2 Connection diagram of 74LS373   **Fig 3.3 Pin of 74LS245**
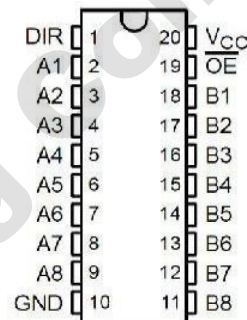
74LS245 Octal Bus Tran iver: 3-State

The 74LS245 is a high-speed Si-gate CMOS device. The 74LS245 is an octal tran iver featuring non- inverting 3-state bus compatible outputs in both send and receive directions. The 74LS245 features an Output Enable (OE) input for easy cascading and a send/receive (DIR) input for direction control. OE controls the outputs so that the buses are effectively isolated. All inputs have a Schmitt-trigger action.

These octal bus tran ivers are designed for asynchronous two-way communication between data buses. The 74LS245 is a high-speed Si-gate CMOS device. The 74LS245 is an octal tran iver featuring non-inverting 3-state bus compatible outputs in both send and receive directions.

The 74LS245 features an Output Enable (OE) input for easy cascading and a send/receive (DIR) input for direction control. OE controls the outputs so that the buses are effectively isolated. All inputs have a Schmitt-trigger action. These octal bus tran ivers are designed for asynchronous two-way communication between data buses.

Memory Devices And Interfacing

The memory interfacing circuit is used to access memory quit frequently to read instruction codes and data stored in the memory. The read / write operations are monitored by control signals. Semiconductor memories are of two types. Viz. RAM (Random Access Memory) and ROM (Read Only Memory) The Semiconductor RAM's are broadly two types-
                    static Ram and dynamic RAM

Memory structure and its requirements

The read / write memories consist of an array of registers in which each register has unique address. The size of memory is N * M as shown in figure.

67

(a) Logic diagram for RAM   (b) Logic diagram for EPROM
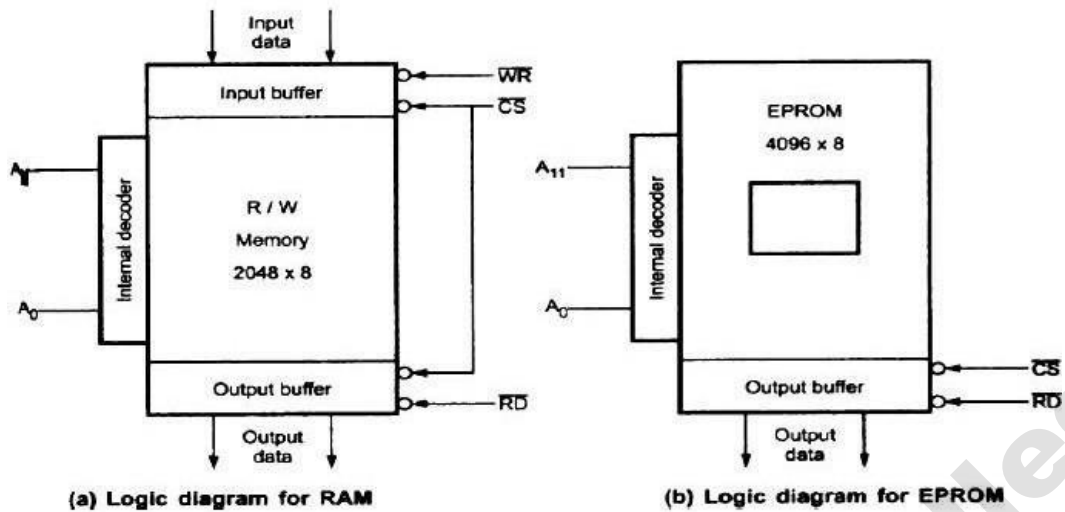
Where N is number of register and M is the word length, in number of bits. As shown in figure(a) memory chip has 12 address lines Ao–A11, one chip select (CS), and two control lines, Read (RD) to enable output buffer and Write (WR) to enable the input buffer.

The internal decoder is used to decoder the address lines. Figure(b) shows the logic diagram of a typical EPROM (Erasable Programmable Read-Only Memory) with 4096 (4K) register. It has 12 address lines Ao – A11, one chip select (CS), one read control signal. Since EPROM does not require the (WR) signal.

EPROM (or EPROMs) is used as a program memory and RAM (or RAMs) as a data memory. When both, EPROM and RAM are used, the total address space 1 Mbytes is shared by them.

Address Decoding Techniques
   • Absolute decoding
   • Linear decoding
   • Block decoding

Absolute Decoding:

In the absolute decoding technique the memory chip is selected only for the specified logic level on the address lines: no other logic levels can select the chip. Below figure the memory interface with absolute decoding. Two 8K EPROMs (2764) are used to provide even and odd memory banks. Control signals BHE and Ao are use to enable output of odd and even memory banks respectively. As each memory chip has 8K memory locations, thirteen address lines are required to address each locations, independently. All remaining address lines are used to generate an unique chip select signal. This address technique is normally used in large memory systems.
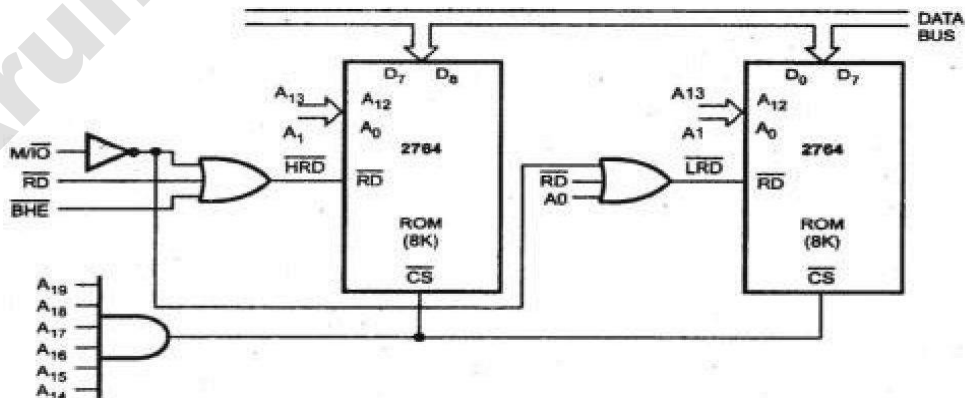
Fig 3.4 Linear decoding

68

Linear Decoding:

In small system hardware for the decoding logic can be eliminated by using only required number of addressing lines (not all). Other lines are simple ignored. This technique is referred as linear decoding or partial decoding. Control signals BHE and Ao are used to enable odd and even memory banks, respectively. Figure shows the addressing of 16K RAM (6264) with linear decoding. The address line A19 is used to select the RAM chips. When A19 is low, chip is selected, otherwise it is disabled. The status of A14 to A18 does not affect the chip selection logic. This gives you multiple addresses (shadow addresses). This technique reduces the cost of decoding circuit, but it gas drawback of multiple addresses
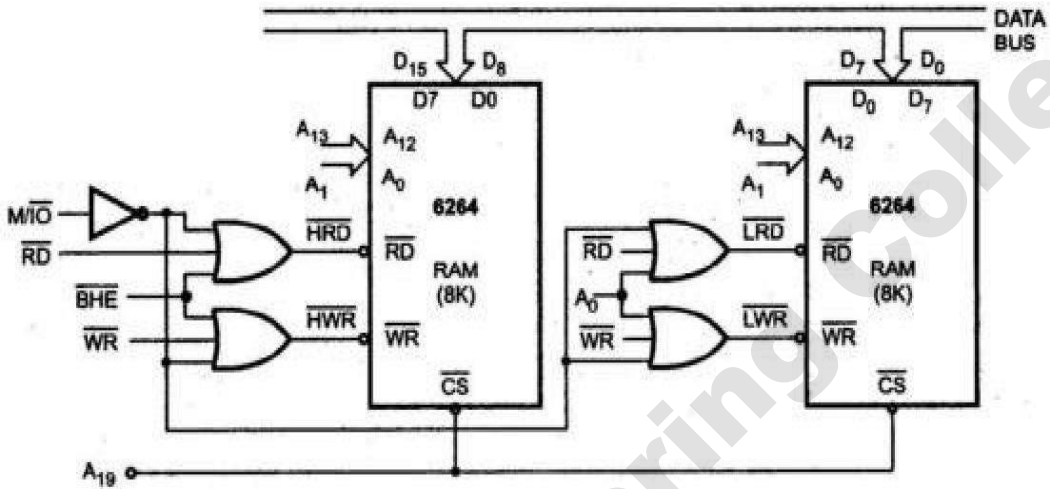
Fig 3.5 Block decoding

Block Decoding:

In a microcomputer system the memory array is often consists of several blocks of memory chips. Each block of memory requires decoding circuit. To avoid separate decoding for each memory block special decoder IC is used to generate chip select signal for each block.
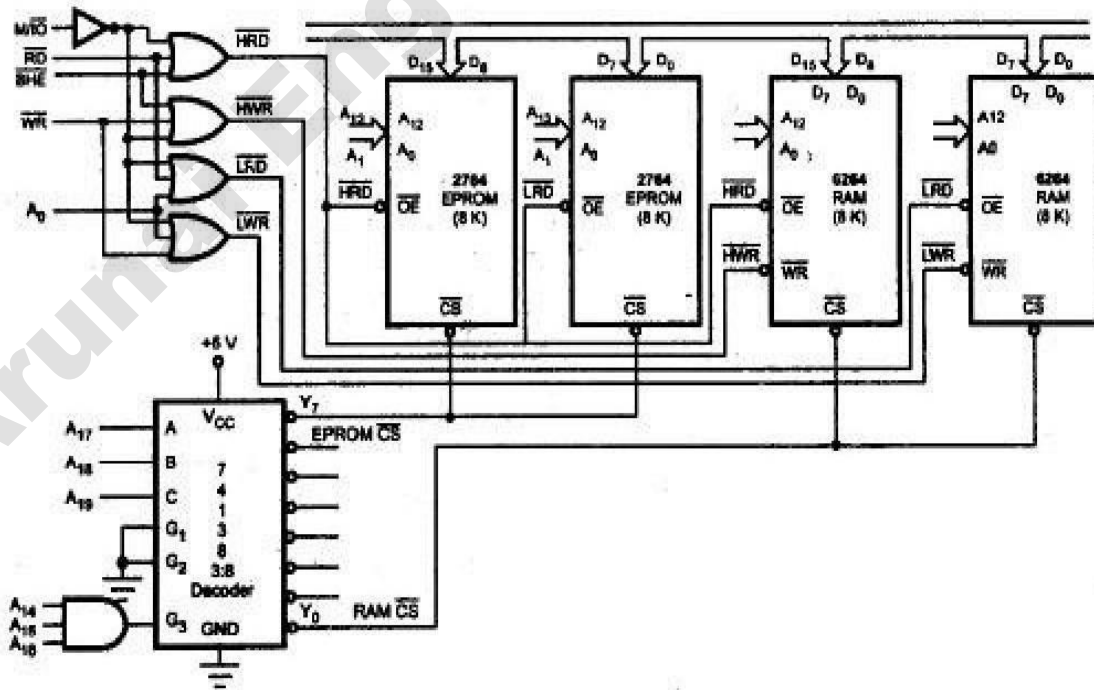
Fig. 3.6 Static Memory interfacing

Static Memory Interfacing

The general procedure of static memory interfacing with 8086 as follows:

1. Arrange the available memory chips so as to obtain 16-bit data bus width. The upper 8-bit bank is called 'odd address memory bank' and the lower 8-bit bank is called 'even address memory bank'.

2. Connect available memory address lines of memory chips with those of the microprocessor and also connect the memory RD and WR inputs to the corresponding processor control signals. Connect the 16-bit data bus of the memory bank with that of the microprocessor 8086.

3. The remaining address lines of the microprocessor, BHE and Ao are used for decoding the required chip select signals for the odd and even memory banks. The CS of memory is derived from the output of the decoding circuit.

4. As a good and efficient interfacing practice, the address map of the system should be continuous as far as possible

Dynamic RAM Interfacing

The basic Dynamic RAM cell uses a capacitor to store the charge as a representation of data. This capacitor is manufactured as a diode that is reverse-biased so that the storage capacitance comes into the picture. This storage capacitance is utilized for storing the charge representation of data but the reverse-biased diode has a leakage current that tends to discharge the capacitor giving rise to the possibility of data loss.

To avoid this possible data loss, the data stored in a dynamic RAM cell must be refreshed after a fixed time interval regularly. The process of refreshing the data in the RAM is known as refresh cycle. This activity is similar to reading the data from each cell of the memory, independent of the requirement of microprocessor, regularly. During this refresh period all other operations (accesses) related to the memory subsystem are suspended.

The advantages of dynamic RAM. Like low power consumption, higher packaging density and low cost, most of the advanced computer systems are designed using dynamic RAMs. Also the refresh mechanism and the additional hardware required makes the interfacing hardware, in case of dynamic RAM, more complicated, as compared to static RAM interfacing circuit.

Interfacing I/O Ports

I/O ports or input/output ports are the devices through which the microprocessor communicates with other devices or external data sources/destinations. Input activity, as one may expect, is the activity that enables the microprocessor to read data from external devices, for example keyboard, joysticks, mouser etc. the devices are known as input devices as they feed data into a microprocessor system.

Output activity transfers data from the microprocessor top the external devices, for example CRT display, 7-segment displays, printer, etc, the devices that accept the data from a microprocessor system are called output devices.

Steps in Interfacing an I/O Device

The following steps are performed to interface a general I/O device with a CPU:

1. Connect the data bus of the microprocessor system with the data bus of the I/O port.

2. Derive a device address pulse by decoding the required address of the device and use it as the chip select of the device.

3. Use a suitable control signal, i.e. IORD and /or IOWR to carry out device operations, i.e. connect IORD to RD input of the device if it is an input devise, otherwise connect IOWR to WR input of the device. In some cases the RD or WR control signals are combined with the device address pulse to generate the device select pulse.

Input Port

The input device is connected to the microprocessor through buffer. The simplest form of a input port is a buffer as shown in the figure. This buffer is a tri-state buffer and its output is available only when enable signal is active. When microprocessor wants to read data from the input device (keyboard), the control signals from the microprocessor activates the buffer by asserting enable input of the buffer. Once the buffer is enabled, data from the device is available on the data bus. Microprocessor reads this data by initiating read command.

Output Port

It is used to send the data to the output device such as display from the microprocessor. The simplest form of the output port is a latch.
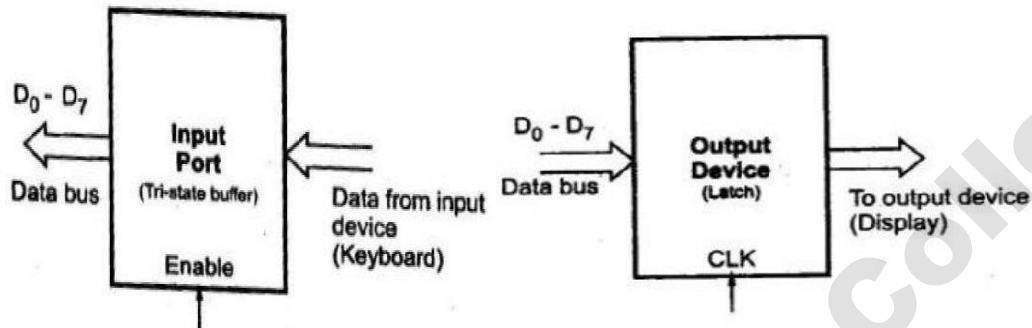
Fig.3.7 I/O interfacing

The output device is connected to the microprocessor through latch as shown in the figure. When microprocessor wants to send data to the output device it puts the data on the data bus and activates the clock signal of the latch, latching the data from the data bus at the output of latch. It is then available at the output of latch for the output device.

I/O Interfacing Techniques

Input/output devices can be interfaced with microprocessor systems in two ways:

      1. I/O mapped I/O

      2. Memory mapped I/O

1. I/O mapped I/O:

8086 has special instructions IN and OUT to transfer data through the input/output ports in I/O mapped I/O system. The IN instruction copies data from a port to the Accumulator. If an 8-bit port is read data will go to AL and if 16-bit port is read the data will go to AX. The OUT instruction copies a byte from AL or a word from AX to the specified port. The M/IO signal is always low when 8086 is executing these instructions. In this address of I/O device is 8-bit or 16-bit. It is 8-bit for Direct addressing and 16-bit for Indirect addressing.

2. Memory mapped I/O

In this type of I/O interfacing, the 8086 uses 20 address lines to identify an I/O device. The I/O device is connected as if it is a memory device. The 8086 uses same control signals and instructions to access I/O as those of memory, here RD and WR signals are activated indicating memory bus cycle.

3.2 Parallel Communication Interface: 8255 Programmable Peripheral Interface and Interfacing

      The 8255 is a widely used, programmable parallel I/O device. It can be programmed to transfer data under data under various conditions, from simple I/O to interrupt I/O. It is flexible, versatile and economical (when multiple I/O ports are required). It is an important general purpose I/O device that can be used with almost any microprocessor.

      The 8255 has 24 I/O pins that can be grouped primarily into two 8 bit parallel ports: A and B, with the remaining 8 bits as Port C. The 8 bits of port C can be used as individual bits or be grouped into two 4 bit ports: CUpper (CU) and CLower (CL). The functions of these ports are defined by writing a control word in the control register.

8255 can be used in two modes: Bit set/Reset (BSR) mode and I/O mode. The BSR mode is used to set or reset the bits in port C. The I/O mode is further divided into 3 modes: mode 0, mode 1 and mode 2. In mode 0, all ports function as simple I/O ports.

Mode 1 is a handshake mode whereby Port A and/or Port B use bits from Port C as handshake signals. In the handshake mode, two types of I/O data transfer can be implemented: status check and interrupt. In mode 2, Port A can be set up for bidirectional data transfer using handshake signals from Port C, and Port B can be set up either in mode 0 or mode 1.

Fig. 3.8 Pins of 8255   **Fig.3.9 Block diagram of 8255**

RD: (Read): This signal enables the Read operation. When the signal is low, microprocessor reads data from a selected I/O port of 8255.
WR: (Write): This control signal enables the write operation.
RESET (**Reset**): It clears the control registers and sets all ports in input mode.
CS,A0, A1: These are device select signals. is connected to a decoded address and A0, A1 are connected to A0, A1 of microprocessor.

$\overline{CS}$	A_1	A_0	Selected
0	0	0	Port A
0	0	1	Port B
0	1	0	Port C
0	1	1	Control Register
1	x	x	8255 is not selected
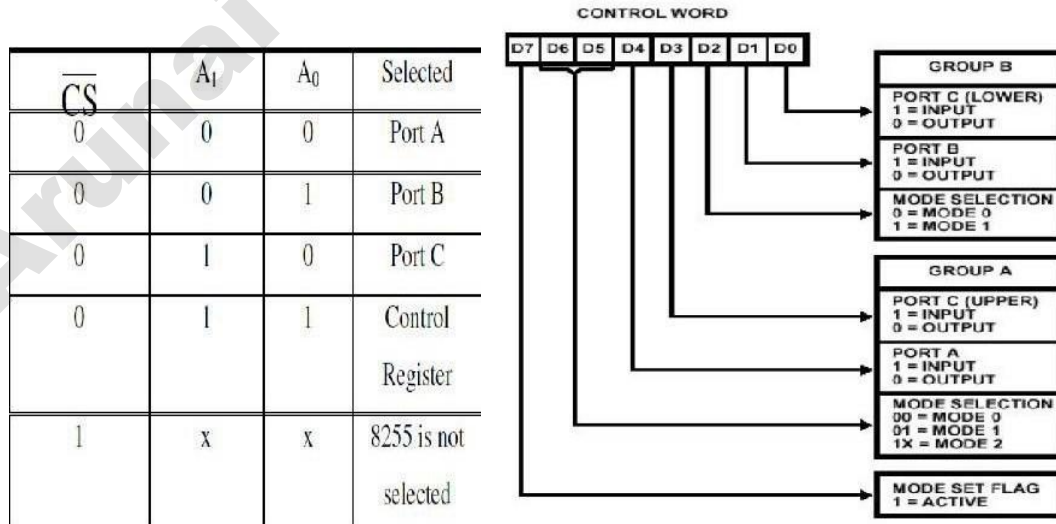
Fig 3.10 Control wordformat of8255
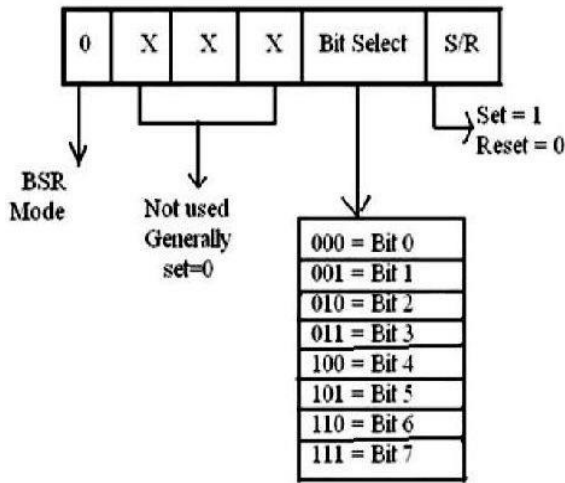
Fig. 3.11 BSR mode of 8255

✓ **I/O Modes of 8255**
Mode 0: Simple Input or Output
In this mode, Port A and Port B are used as two simple 8-bit I/O ports and Port C as two4-bit I/O ports. Each port (or half-port, in case of Port C) can be programmed to function as simply an input port or an output port. The input/output features in mode 0 are: Outputs are latched, Inputs are not latched. Ports do not have handshake or interrupt capability.
Mode 1: Input or Output with handshake
In mode 1, handshake signals are exchanged between the microprocessor and peripherals prior to data transfer. The ports (A and B) function as 8-bit I/O ports. They can be configured either as input or output ports. Each port (Port A and Port B) uses 3 lines from port C as handshake signals. The remaining two lines of port C can be used for simple I/O functions. Input and output data are latched and Interrupt logic is supported.

Mode 1: Input control signals
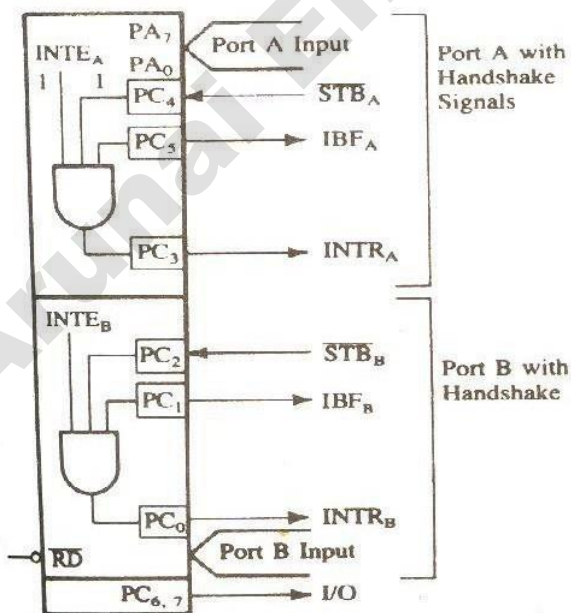
Fig.3.12 mode 1 input control signals

73

STB **Strobe Input**): This signal (active low) is generated by a peripheral device that it has transmitted a byte of data. The 8255, in response to, generates IBF and INTR.

IBF (**Input buffer full):** This signal is an acknowledgement by the 8255 to indicate that the input latch has received the data byte. This is reset when the microprocessor reads the data.

INTR (Interrupt Request): This is an output signal that may be used to interrupt the microprocessor. This signal is generated if , IBF and INTE are all at logic 1.

INTE (Interrupt Enable): This is an internal flip-flop to a port and needs to be set to generate the INTR signal. The two flip-flops INTEA and INTEB are set /reset using the BSR mode. The INTEA is enabled or disabled through PC4, and INTEB is enabled or disabled through PC2.

(a)   input operation   (b) output operation
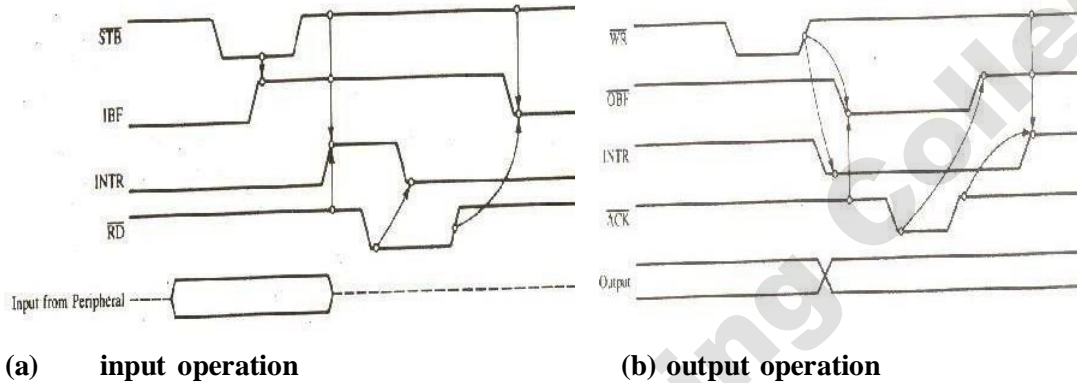
Fig.3.13 Timing waveform for mode1 operation

(Output Buffer Full): This is an output signal that goes low when the microprocessor writes data into the output latch of the 8255. This signal indicates to an output peripheral that new data is ready to be read. It goes high again after the 8255 receives a signal from the peripheral.

(**Acknowledge):** This is an input signal from a peripheral that must output a low when the peripheral receives the data from the 8255 ports.

INTR (Interrupt Request): This is an output signal, and it is set by the rising edge of the signal. This signal can be used to interrupt the microprocessor to request the next data byte for output. The INTR is set and INTE are all one and reset by the rising edge of . .

INTE (Interrupt Enable): This is an internal flip-flop to a port and needs to be set to generate the INTR signal. The two flip-flops INTEA and INTEB are set /reset using the BSR mode. The INTEA signal can be enabled or disabled through PC6, and INTEB is enabled or disabled through PC2.

✓ **Mode 2: Bidirectional Data Transfer**

OBF This mode is used primarily in applications such as data transfer between the two computers or floppy disk controller interface. Port A can be configured as the bidirectional port and Port B either in mode 0 or mode 1. Port A uses five signals from Port C as handshake signals for data transfer. The remaining three lines from Port C can be used either as simple I/O or as handshake signals for Port B.
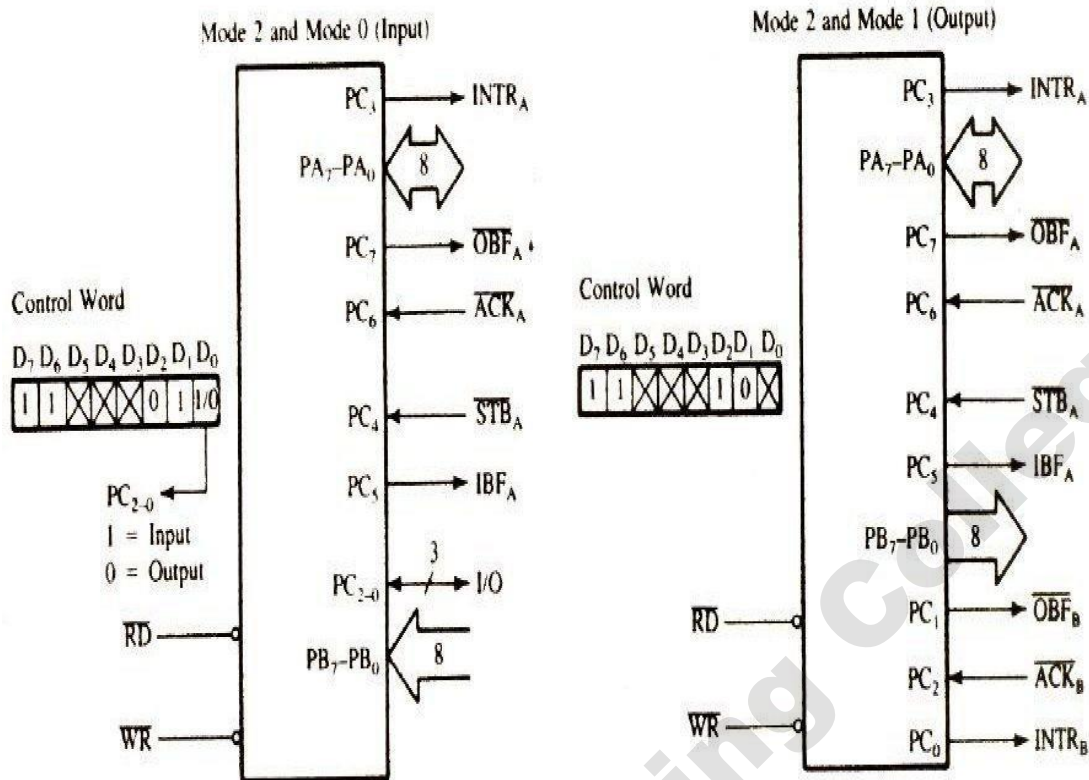
74

Fig. 3.14 Mode 2 Control Signals

3.3 Serial Communication: Using 8251

8251 is a Universal Synchronous and Asynchronous Receiver and Transmitter compatible with Intel's processors. This chip converts the parallel data into a serial stream of bits suitable for serial transmission. It is also able to receive a serial stream of bits and convert it into parallel data bytes to be read by a microprocessor.

Basic Modes of data transmission

           a) Simplex

           b) Duplex

           c) Half Duplex

a) Simplex mode

Data is transmitted only in one direction over a single communication channel. For example, the processor may transmit data for a CRT display unit in this mode.

b) Duplex Mode

In duplex mode, data may be transferred between two transreceivers in both directions simultaneously.

c) Half Duplex mode

In this mode, data transmission may take place in either direction, but at a time data may be transmitted only in one direction. A computer may communicate with a terminal in this mode. It is not possible to transmit data from the computer to the terminal and terminal to computer simultaneously.
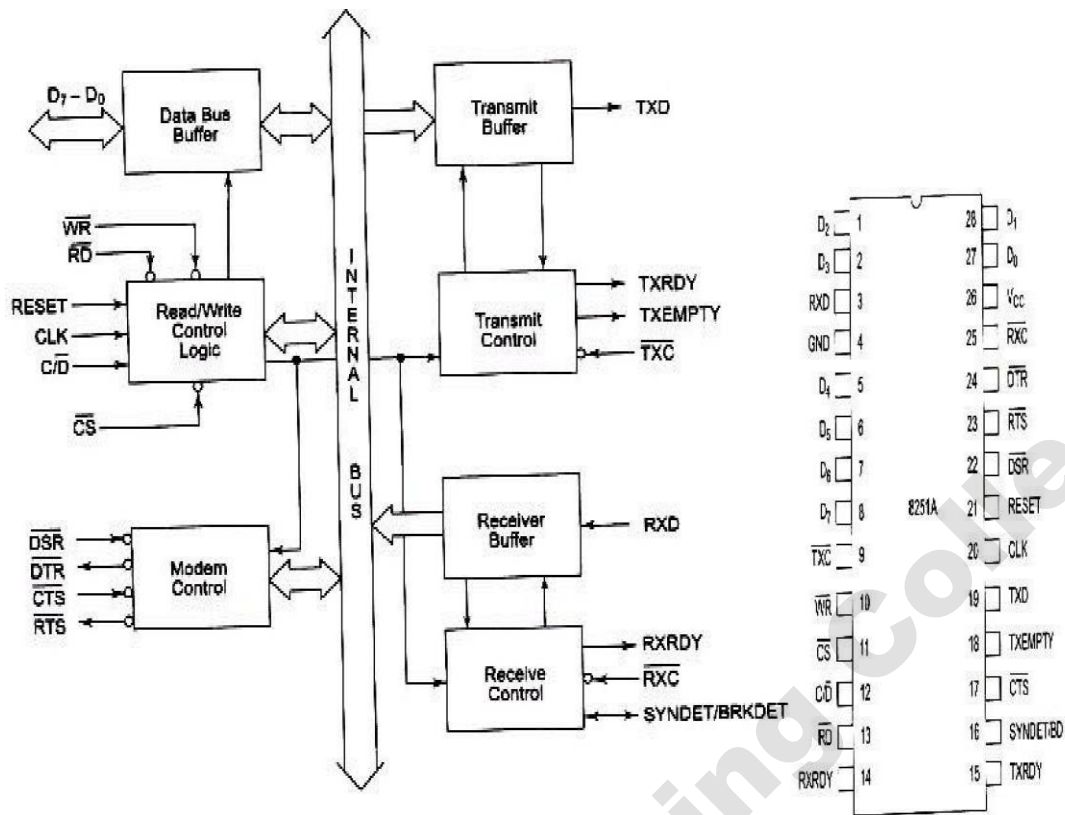
Fig. 3.15 Serial communication interface 8251

The data buffer interfaces the internal bus of the circuit with the system bus. The read / write control logic controls the operation of the peripheral depending upon the operations initiated by the CPU decides whether the address on internal data bus is control address / data address. The modem control unit handles the modem handshake signals to coordinate the communication between modem and USART.

The transmit control unit transmits the data byte received by the data buffer from the CPU for serial communication. The transmission rate is controlled by the input frequency. Transmit control unit also derives two transmitter status signals namely TXRDY and TXEMPTY which may be used by the CPU for handshaking.

The transmit buffer is a parallel to serial converter that receives a parallel byte for conversion into a serial signal for further transmission. The receive control unit decides the receiver frequency as controlled by the RXC input frequency. The receive control unit generates a receiver ready (RXRDY) signal that may be used by the CPU for handshaking. This unit also detects a break in the data string while the 8251 is in asynchronous mode. In synchronous mode, the 8251 detects SYNC characters using SYNDET/BD pin.

Signal Description of 8251

$D_0 - D_7$: This is an 8-bit data bus used to read or write status, command word or data from or to the 8251A.

C / D: (Control Word/Data): This input pin, together with RD and WR inputs, informs the 8251A that the word on the data bus is either a data or control word/status information. If this pin is 1, control / status is on the bus, otherwise data is on the bus.

RD: This active-low input to 8251A is used to inform it that the CPU is reading either data or status information from its internal registers. This active-low input to 8251A is used to inform it that the CPU is writing data or control word to 8251A.

WR: This is an active-low chip select input of 825lA. If it is high, no read or write operation can be carried out on 8251. The data bus is tristated if this pin is high.

CLK: This input is used to generate internal device timings and is normally connected to clock generator output. This input frequency should be at least 30 times greater than the receiver or transmitter data bit transfer rate.

RESET: A high on this input forces the 8251A into an idle state. The device will remain idle till this input signal again goes low and a new set of control word is written into it. The minimum required reset pulse width is 6 clock states, for the proper reset operation.

TXC (Transmitter Clock Input): This transmitter clock input controls the rate at which the character is to be transmitted. The serial data is shifted out on the successive negative edge of the TXC.

TXD (Transmitted Data Output): This output pin carries serial stream of the transmitted data bits along with other information like start bit, stop bits and parity bit, etc.

RXC **(Receiver Clock Input):** This receiver clock input pin controls the rate at which the character is to be received.

RXD (Receive Data Input): This input pin of 8251A receives a composite stream of the data to be received by 8251 A.

RXRDY (Receiver Ready Output): This output indicates that the 8251A contains a character to be read by the CPU.

TXRDY - Transmitter Ready: This output signal indicates to the CPU that the internal circuit of the transmitter is ready to accept a new character for transmission from the CPU.

DSR - Data Set Ready: This is normally used to check if data set is ready when communicating with a modem.

DTR - Data Terminal Ready: This is used to indicate that the device is ready to accept data when the 8251 is communicating with a modem.

RTS - Request to Send Data: This signal is used to communicate with a modem.

TXE- Transmitter Empty: The TXE signal can be used to indicate the end of a transmission mode.

Operating Modes of 8251
1. Asynchronous mode
2. Synchronous mode

Asynchronous Mode (Transmission)
When a data character is sent to 8251A by the CPU, it adds start bits prior to the serial data bits, followed by optional parity bit and stop bits using the asynchronous mode instruction control word format. This sequence is then transmitted using TXD output pin on the falling edge of TXC.

Asynchronous Mode (Receive)
A falling edge on RXD input line marks a start bit. The receiver requires only one stop bit to mark end of the data bit string, regardless of the stop bit programmed at the transmitting end. The 8-bit character is then loaded into the into parallel I/O buffer of 8251.

RXRDY pin is raised high to indicate to the CPU that a character is ready for it. If the previous character has not been read by the CPU, the new character replaces it, and the overrun flag is set indicating that the previous character is lost.
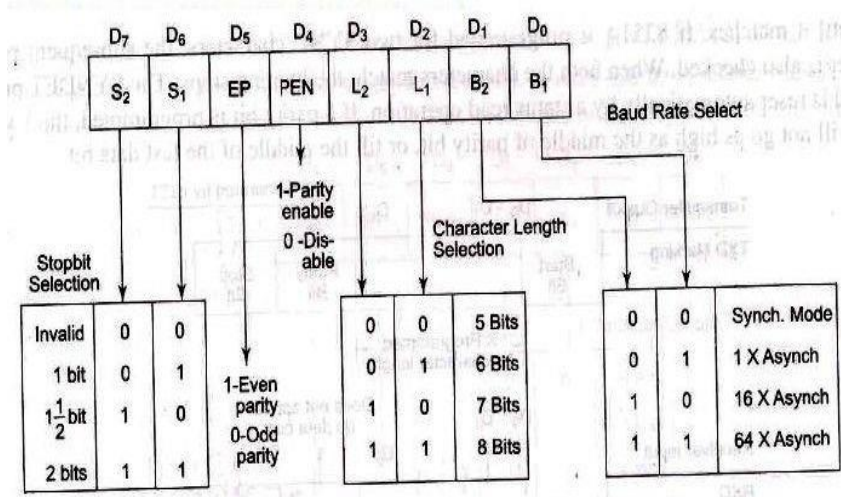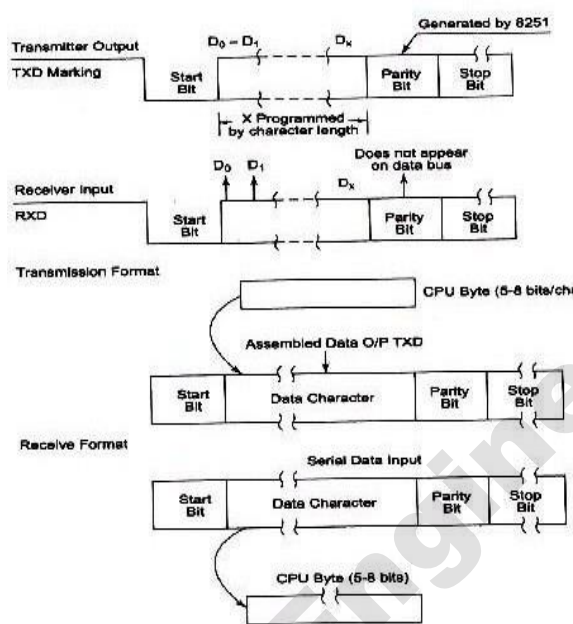
Fig.3.16 Mode instruction format-Async.

Fig. 3.17 communication format

Synchronous mode

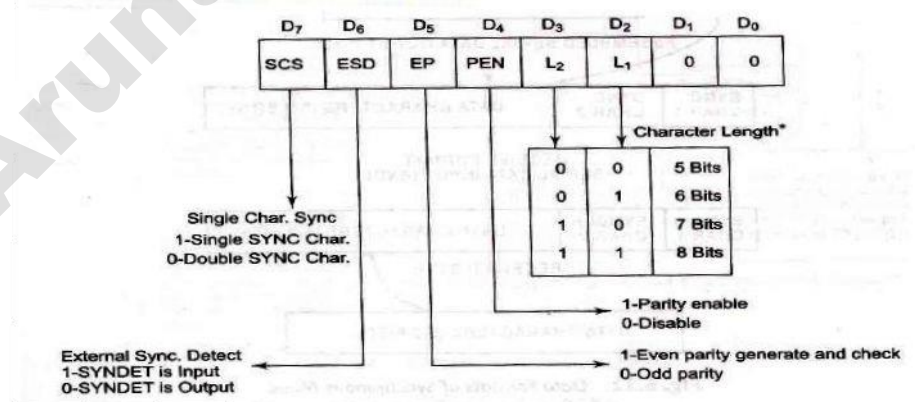## Synchronous Mode Instruction Format

Fig. 3.18 Synchronous mode Instruction format

Synchronous Mode (Transmission)

The TXD output is high until the CPU sends a character to 8251 which usually is a SYNC character. When CTS line goes low, the first character is serially transmitted out. Characters are shifted out on the falling edge of TXC .Data is shifted out at the same rate as TXC , over TXD output line. If the CPU buffer becomes empty, the SYNC character or characters are inserted in the data stream over TXD output.

Synchronous Mode (Receiver)

In this mode, the character synchronization can be achieved internally or externally. The data on RXD pin is sampled on rising edge of the RXC. The content of the receiver buffer is compared with the first SYNC character at every edge until it matches. If 8251 is programmed for two SYNC characters, the subsequent received character is also checked. When the characters match, the hunting stops. The SYNDET pin set high and is reset automatically by a status read operation. In the external SYNC mode, the synchronization is achieved by applying a high level on the SYNDET input pin that forces 8251 out of HUNT mode. The high level can be removed after one RXC cycle. The parity and overrun error both are checked in the same way as in asynchronous mode.

Synchronous mode Transmit and Receive data format
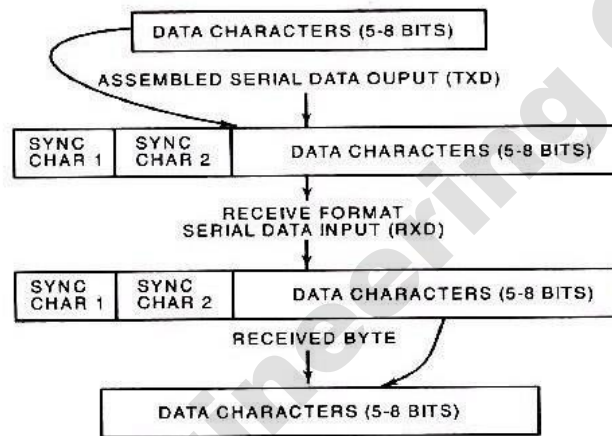
Fig.3.19 Data Formats of Synchronous Mode

Command Instruction Definition

The command instruction controls the actual operations of the selected format like enable transmit/receive, error reset and modem control. A reset operation returns 8251 back to mode instruction format.

Command Instruction format
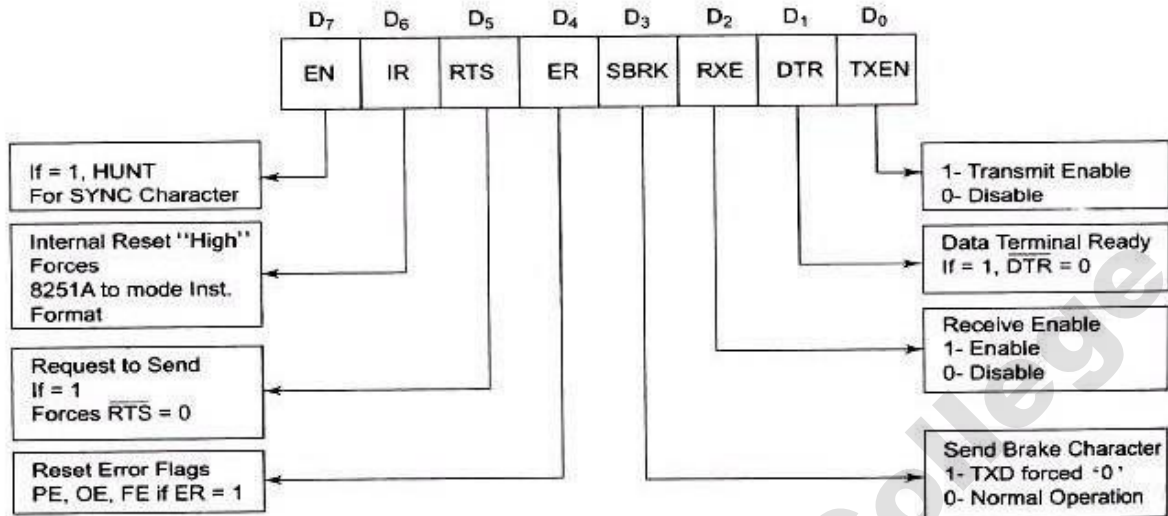
Status Read Definition

This definition is used by the CPU to read the status of the active 8251 to confirm if any error condition or other conditions like the requirement of processor service has been detected during the operation.
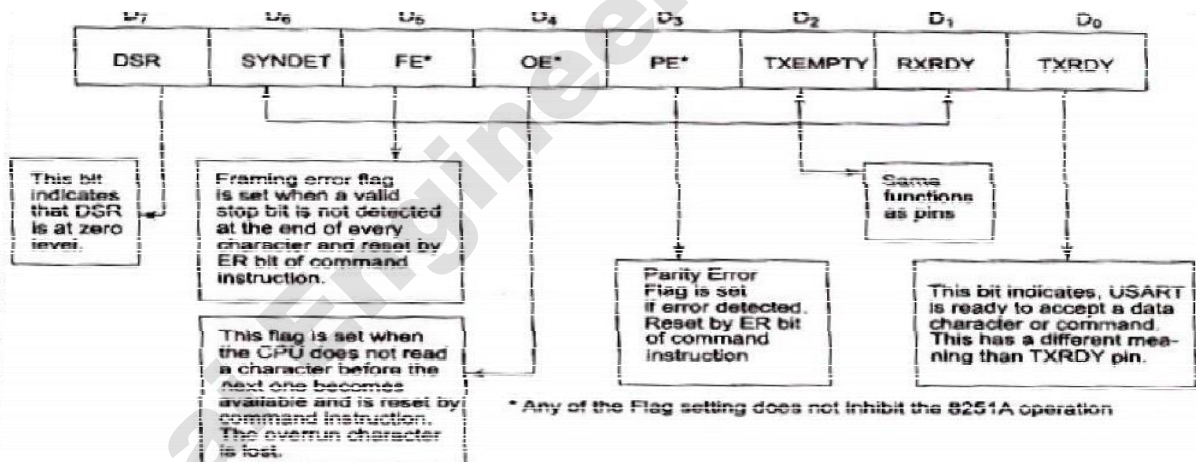

Fig. 3.20 Status information

3.4 D/A And A/D Interface:

The function of an A/D converter is to produce a digital word which represents the magnitude of some analog voltage or current.

The specifications for an A/D converter are very similar to those for D/A converter:

· The resolution of an A/D converter refers to the number of bits in the output binary word. An 8-bit converter for example has a resolution of 1 part in 256.

· Accuracy and linearity specifications have the same meaning for an A/D converter as they do for a D/A converter.

· Another important specification for an ADC is its conversion time. - the time it takes the converter to produce a valid output binary code for an applied input voltage. When we refer to a converter as high speed, it has a short conversion time.

The analog to digital converter is treated as an input device by the microprocessor that sends an initialising signal to the ADC to start the analog to digital data conversation process.

The start of conversion signal is a pulse of a specific duration. The process of analog to digital conversion is a slow process, and the microprocessor has to wait for the digital data till the conversion is over.

After the conversion is over, the ADC sends end of conversion (EOC) signal to inform the microprocessor that the conversion is over and the result is ready at the output buffer of the ADC.

These tasks of issuing an SOC pulse to ADC, reading EOC signal from the ADC and reading the digital output of the ADC are carried out by the CPU using 8255 I/O ports. The time taken by the ADC from the active edge of SOC pulse (the edge at which the conversion process actually starts) till the active edge of

EOC signal is called as the conversion delay of the ADC- the time taken by the converter to calculate the equivalent digital data output from the instant of the start of conversion is called conversion delay. It may range anywhere from a few microseconds in case of fast ADCs to even a few hundred milliseconds in case of slow ADCs.

A number of ADCs are available in the market, the selection of ADC for a particular application is done, keeping in mind the required speed, resolution range of operation, power supply requirements, sample and hold device requirements and the cost factors are considered.

The available ADCs in the market use different conversion techniques for the conversion of analog signals to digital signals.

Parallel converter or flash converter,

Successive approximation and

dual slope integration

A general algorithm for ADC interfacing contains the following steps.

1. Ensure the stability of analog input, applied to the ADC.
2. Issue start of conversion (SOC) pulse to ADC.
3. Read end of conversion (EOC) signal to mark the end of conversion process.
4. Read digital data output of the ADC as equivalent digital output.

It may be noted that analog input voltage must be constant at the input of the ADC right from the start of conversion till the end of conversion to get correct results. This may be ensured by a sample and hold

circuit which samples the analog signal and holds it constant for a specified time duration.

The microprocessor may issue a hold signal to the sample and Hold circuit. If the applied input changes before the complete conversion process is over, the digital equivalent of the analog input calculated by the ADC may not be correct. If the applied input changes before the complete conversion process is over, the digital equivalent of the analog input calculated by the ADC may not be correct.

· **ADC 0808/0809**

The analog to digital converter chips 0808 and 0809 are 8-bit CMOS, *successive approximation converters.* Successive approximation technique is one of the fast techniques for analog to digital conversion. The conversion delay is 100 µs at a clock frequency of 640 kHz, which is quite low as

compared to other converters.

These converters do not need any external zero or full scale adjustments as they are already taken care of by internal circuits. These converters internally have a 3:8 analog multiplexer so that at a time eight different analog inputs can be connected to the chips. Out of these eight inputs only one can be selected for conversion by using address lines ADD A, ADD B and ADD C, as shown. Using these address inputs, multichannel data acquisition systems can be designed using a single ADC.

The CPU may drive these lines using output port lines in case of multichannel applications. In case of single input applications, these may be hard wired to select the proper input.
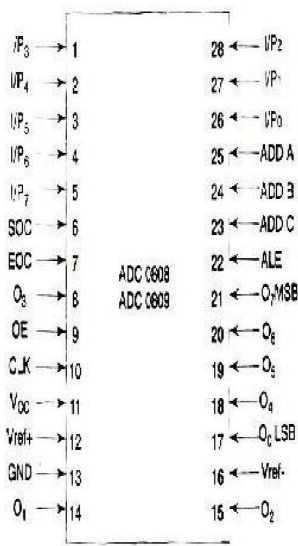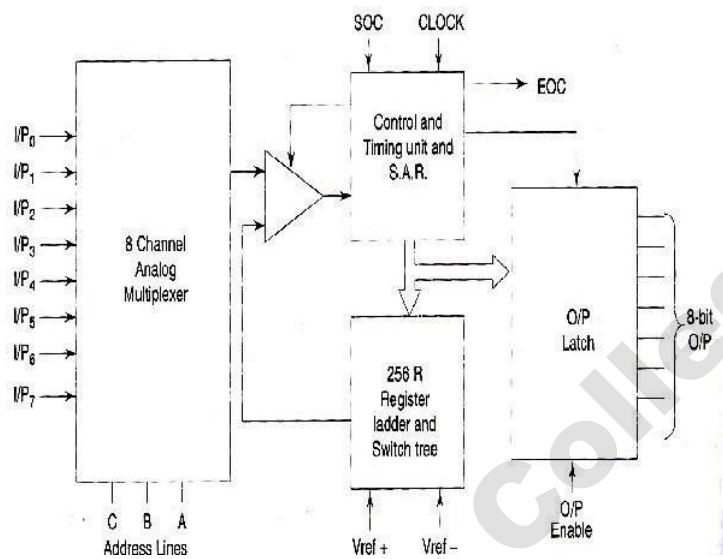

Fig. 3.21 Pins of ADC0808/09   Fig.3.22 Block Diagram of ADC 0808/0809

only positive analog input voltages to their digital equivalents. These chips do not contain any internal sample and hold circuit. If one needs a sample and hold circuit for the conversion of fast, signals into equivalent digital quantities, it has to be externally connected at each of the analog inputs.
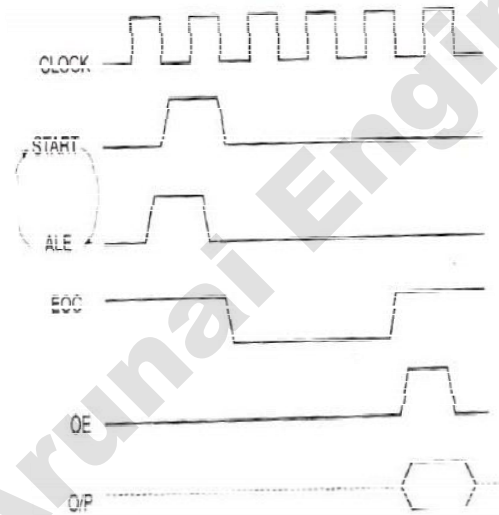
Fig.3.23 Timing Diagram of ADC 0808/0809   Fig. 3.24 Pins of DAC 0800
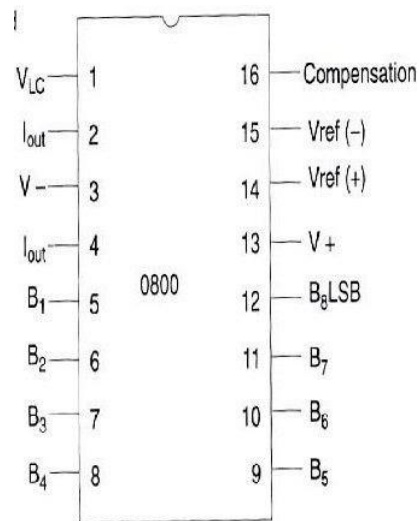
INTERFACING DIGITAL TO ANALOG ONVERTERS:

The digital to analog converters convert binary numbers into their analog equivalent voltages or currents. Several techniques are employed for digital to analog conversion.

    i. Weighted resistor network
    ii. R-2R ladder network
    iii. Current output D/A converter

- ✓ Applications in areas like
  - · digitally controlled gains, motor speed control, programmable gain amplifiers, digital voltmeters, panel meters, etc.
  - · In a compact disk audio player for example a 14-or16-bit D/A converter is used to convert the binary data read off the disk by a laser to an analog audio signal.
  - · Most speech synthesizer integrated circuits contain a D/A converter to convert stored binary data words into analog audio signals.
  - ✓ Characteristics:

    1. Resolution: It is a change in analog output for one LSB change in digital input.

    It is given by$(1/2n)$*Vref.

    If n=8 (i.e.8-bit DAC)   1/256*5V=39.06mV

    2. Settling time: It is the time required for the DAC to settle for a full scale code

    change.

DAC 0800 8-bit Digital to Analog converter

Features:

i. DAC0800 is a monolithic 8-bit DAC manufactured by National semiconductor.

ii. It has settling time around 100ms

iii. It can operate on a range of power supply voltage i.e. from   +18V.

t

o

Usually the supply V+ is 5V or +12V. The V- pin can be kept at a minimum of -12V.

iv. Resolution of the DAC is 39.06mV

3.5 Programmable timer device 8253

Intel's programmable counter/timer device (8253) facilitates the generation of accurate time delays. When 8253 is used as timing and delay generation peripheral, the microprocessor becomes free from the

tasks related to the counting process and execute the programs in memory, while the timer device may perform the counting tasks. This minimizes the software overhead on the microprocessor.

✓ Architecture and Signal Descriptions

The programmable timer device 8253 contains three independent 16-bit counters, each with a maximum count rate of 2.6 MHz to generate three totally independent delays or maintain three independent counters simultaneously. All the three counters may be independently controlled by programming the three internal command word registers.

✓   The 8-bit bidirectional data buffer interfaces internal circuit of 8253 to

,

microprocessor systems bus. Data is transmitted or received by the buffer upon the execution of IN or OUT instruction. The read/write logic controls the direction of the data buffer depending upon whether it is a read or a write operation. It may be noted that IN instruction reads data while OUT instruction writes data to a peripheral.
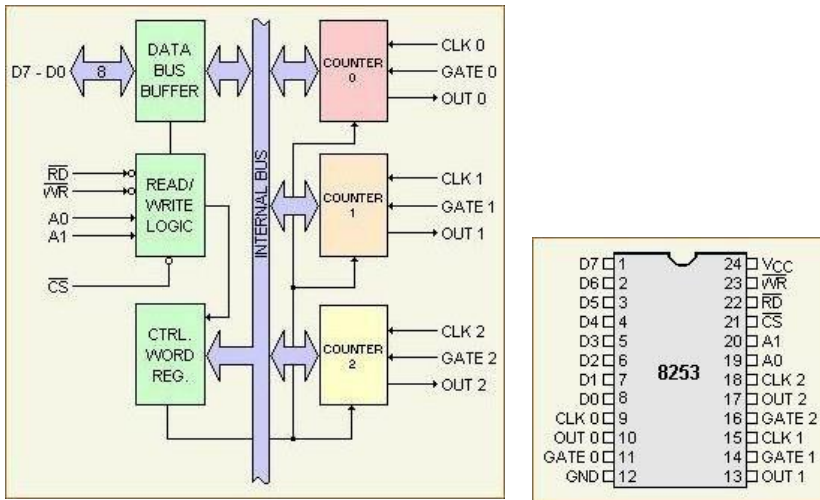
.

Fig 3.25   Internal blocks of 8253 and pin diagram

✓   The three counters all 16-bit presettable, down counters, able to operate either in BCD or in hexadecimal mode. The mode control word register contains the information that can be used for writing or reading the count value into or from the respective count register using the OUT and IN instructions. The specialty of the 8253 counters is that they can be easily read on line without disturbing the clock input to the counter. This facility is called as "on the fly" reading of counters, and is invoked using a mode control word.

✓   A0, Al pins are the address input pins and are required internally for addressing the mode control word registers and the three counter registers. A low on CS line enables the 8253. No operation will be performed by 8253 till it is enabled.

Table 3.1 selected operations for various Control

CS	RD	WR	A_1	A_0	Selected Operations
0	1	0	0	0	Write Counter 0
0	1	0	0	1	Write Counter 1
0	1	0	1	0	Write Counter 2
0	1	0	1	1	Write control Word
0	0	1	0	0	Read Counter 0
0	0	1	0	1	Read Counter 1
0	0	1	1	0	Read Counter 2
0	0	1	1	1	No Operation
0	1	1	X	X	No Operation
1	X	X	X	X	Disabled

A control word register accepts the 8-bit control word written by the microprocessor and stores it for controlling the complete operation of the specific counter. It may be noted that, the control word register can only be written and cannot be read as it is obvious from Table .The CLK, GATE and OUT pins are available for each of the three timer channels. Their functions will be clear when we study the different operating modes of 8253.

✓ **Control Word Register**

The 8253 can operate in anyone of the six different modes. A control word must be written in the respective control word register by the microprocessor to initialize each of the counters of 8253 to decide its operating mode. All the counters can operate in anyone of the modes or they may be even in different modes of operation, at a time.

The control word format is presented, along with the definition of each bit, while writing a count in the counter, it should be noted that, the count is written in the counter only after the

data is put on the data bus and a falling edge appears at the clock pin of the peripheral thereafter. Any reading operation of the counter, before the falling edge appears may result in garbage data.

CONTROL BYTE D7 - D0							
D7	D6	D5	D4	D3	D2	D1	D0
SC1	SC0	RL1	RL0	M2	M1	M0	BCP

D5 RL1	D4 RL0	R / L Definition
0	0	Counter value is latched. This means that the selected counter has its contents transferred into a temporary latch, which can then be read by the CPU.
0	1	Read / load least-significant byte only.
1	0	Read / load most-significant byte only.
1	1	Read / load least-significant byte first, then most-significant byte.

D7 SC1	D6 SC0	Counter Select
0	0	counter 0
0	1	counter 1
1	0	counter 2
1	1	illegal value

D0 BCD	Operation
0	Hexadecimal Count
1	BCD Count

D3 M2	D2 M1	D1 M0	Mode value
0	0	0	mode 0: interrupt on terminal count
0	0	1	mode 1: programmable one-shot
x	1	0	mode 2: rate generator
x	1	1	mode 3: square wave generator
1	0	0	mode 4: software triggered strobe
1	0	1	mode 5: hardware triggered strobe

Fig. 3.26 Control word format and bit definition

MODE 0 This mode of operation is called as interrupt on terminal count. In this mode, the output is initially low after the mode is set. The output remains low even after the count value is loaded in the counter. The counter starts decrementing the count value after the falling edge of the clock, if the GATE input is high. The process of decrementing the counter continues at each falling edge of the clock till the terminal count is reached, i.e. the count becomes zero. When the terminal count is reached, the output goes high and remains high till the selected control word register or the corresponding count register is reloaded with a new mode of operation or a new count, respectively.

This high output may be used to interrupt the processor whenever required, by setting suitable terminal count. Writing a count register while the previous counting is in process, generates the following sequence of response.

The first byte of the new count when loaded in the count register, stops the previous count. The second byte when written, starts the new count, terminating the previous count

then and there. The GATE signal is active high and should be high for normal counting. When GATE goes low counting is terminated and the current count is latched till the GATE again goes high.
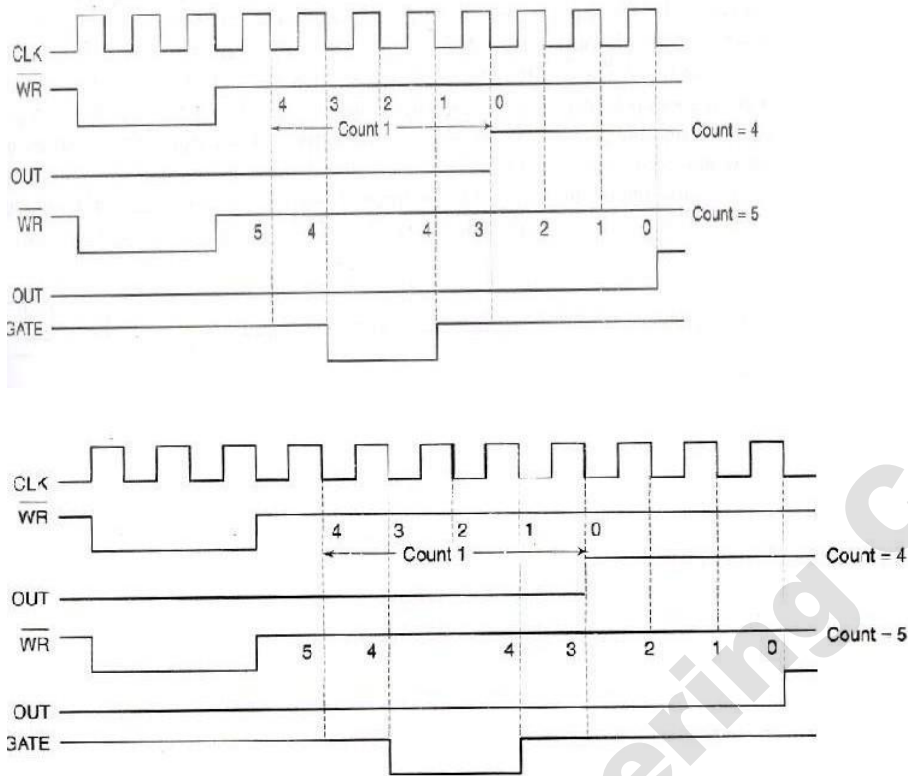

Fig. 3. 27 Waveforms WR, OUT and GATE in Mode 0

MODE 1 This mode of operation of 8253 is called as programmable one-shot mode. the 8253 can be used as a monostable multivibrator. The duration of the quasistable state of the monstable multivibrator is decided by the count loaded in the count register.

The gate input is used as trigger input in this mode of operation. Normally the output remains high till the suitable count is loaded in the count register and a trigger is applied. After the application of a trigger (on the positive edge), the output goes low and remains low till the count becomes zero. If another count is loaded when the output is already low, it does not disturb the previous count till a new trigger pulse is applied at the GATE input. The new counting starts after the new trigger pulse.
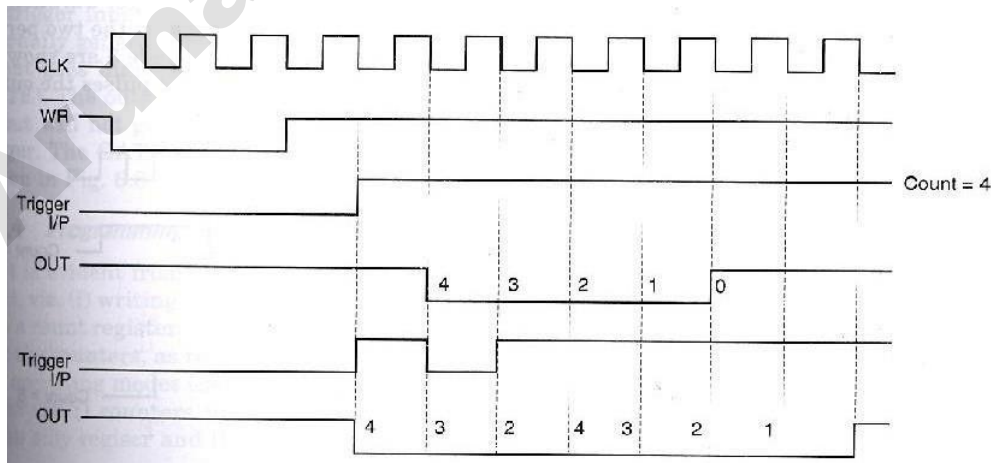

Fig.3.28. WR, GATE and OUT Waveforms in Mode 1

MODE 2 This mode is called either rate generator or divide by N counter. In this mode, if N is loaded as the count value, then, after N pulses, the output becomes low only for one clock cycle. The count *N* is reloaded and again the output becomes high and remains high for *N* clock pulses.

The output is normally high after initialisation or even a low signal on GATE input can force the output high. If GATE goes high, the counter starts counting down from the initial value. The counter generates an active low pulse at the output initially, after the count register is loaded with a count value. Then count down starts and whenever the count becomes zero another active low pulse is generated at the output.

The duration of these active low pulses are equal to one clock cycle. The number of input clock pulses between the two low pulses at the output is equal to the count loaded. Figure shows the related waveforms for mode 2. Interestingly, the counting is inhibited when GATE becomes low.
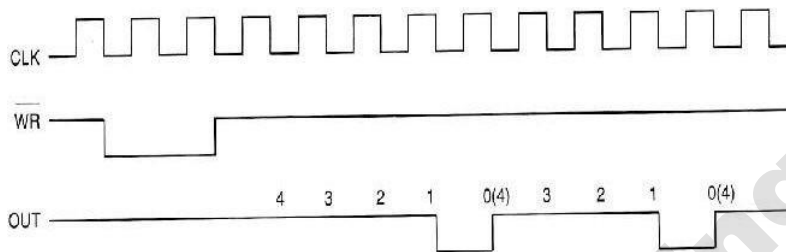
Fig. 3.29 Waveforms at pin WR and OUT in Mode 2

MODE 3 In this mode, the 8253 can be used as a square wave rate generator. In terms of operation this mode is somewhat similar to mode 2. When, the count *N* loaded is even, then for half of the count, the output remains high and for the remaining half it remains low.

If the count loaded is odd, the first clock pulse decrements it by 1 resulting in an even count value (holding the output high). Then the output remains high for half of the new count and goes low for the remaining half. This procedure is repeated continuously resulting in the generation of a square wave.

In case of odd count, the output is high for longer duration and low for shorter duration. The difference of one clock cycle duration between the two periods is due to the initial decrementing of the odd count. The waveforms for mode 3 are shown in Fig. if the loaded count value *'N* is odd, then for $(N+1)/2$ pulses the output remains high and for $(N-1)/2$ pulses it remains low.

MODE 4 This mode of operation of 8253 is named as software triggered strobe. After the mode is set, the output goes high. When a count is loaded, counting down starts. On terminal count, the output goes low for one clock cycle, and then it again goes high. This low pulse can be used as a strobe, while interfacing the microprocessor with other peripherals.

The count is inhibited and the count value is latched, when the GATE signal goes low. If a new count is loaded in the count register while the previous counting is in the next clock cycle. The counting then proceeds according to the new count.

MODE 5 This mode of operation also generates a strobe in response to the rising edge at the trigger input. This mode may be used to generate a delayed strobe in response to an externally generated signal. Once this mode is programmed and the counter is loaded, the output goes high.

The counter starts counting after the rising edge of the trigger input (GATE). The output goes low for one clock period, when the terminal count is reached. The output will not go low until the counter content becomes zero after the rising edge of any trigger. The

GATE input in this mode is used as trigger input. The related waveforms are shown in Fig. 1.8.

✓ **Programming and Interfacing 8253**

There may be two types of write operations in 8253, viz.

     (i) writing a control word into a control word register and

     (ii) writing a count value into a count register.

The control word register accepts data from the data buffer and initializes the counters, as required. The control word register contents are used for (a) initialising the operating modes (mode0-mode4) (b) selection of counters (counter0-counter2) (c) choosing binary BCD counters (d) loading of the counter registers.

The mode control register is a write only register and the CPU cannot read its contents. One can directly write the mode control word for counter 2 or counter 1 prior to writing the control word for counter0. Mode control word register has a separate address, so that it can be written independently. A count register must be loaded with the count value with same byte sequence that was programmed in the mode control word of that counter, using the bits RL0 and RL1.

The loading of the count registers of different counters is again sequence independent. One can directly write the 16-bit count register for count 2 before writing count 0 and count 1, but the two bytes in a count must be written in the byte sequence programmed using RL0 and RL1 bits of the mode control word of the counter. All the counters in 8253 are down counters, hence their count values go on decrementing if the CLK input pin is applied with a valid clock signal. A maximum count is obtained by loading all zeros into a count register, i.e. 2^{16} for binary counting and 10^{4} for BCD counting. The 8253 responds to the negative clock edge of the clock input.

The maximum operating clock frequency of 8253 is 2.6 MHz. For higher frequencies one can use timer 8254, which operates up to 10 MHz, maintaining pin compatibility with 8253. The following Table 6.2 shows the selection of different mode control words and counter register bytes depending upon address lines Ao and A1

In 8253, the 16-bit contents of the counter can simply be read using successive 8-bit IN operations. As stated earlier, the mode control register cannot be read for any of the counters. There are two methods for reading 8253 counter registers.

In the first method, either the clock or the counting procedure (using GATE) is inhibited to ensure a stable count. Then the contents are read by selecting the suitable counter using A0, Al and executing using IN instructions. The first IN instruction reads the least significant byte and the second IN instruction reads the most significant byte. Internal logic of 8253 is designed in such a way that the programmer has to complete the reading operation as programmed by him, using RL0 and RLl bits of control word.

In the second method of reading a counter, the counter can be read while counting is in progress. This method, as already mentioned is called as reading on fly. In this method, neither clock nor the counting needs to be inhibited to read the counter. The content of a counter can be read 'on fly' using a newly defined control word register format for online reading of the count register. Writing a suitable control word, in the mode control register internally latches the contents of the counter. The control word format for 'read on fly' mode is given in Fig. 1.9 along with its bit definitions. After latching the content of a counter using this method, the programmer can read it using IN instructions, as discussed before.

3.6 8279 Programmable Keyboard/Display Controller

Intel's 8279 is a general purpose Keyboard Display controller that simultaneously drives the display of a system and interfaces a Keyboard with the CPU. The Keyboard Display interface scans the Keyboard to identify if any key has been pressed and sends the code of the pressed key to the CPU. It also transmits the data received from the CPU, to the display device.

Both of these functions are performed by the controller in repetitive fashion without involving the CPU. The Keyboard is interfaced either in the interrupt or the polled mode. In the interrupt mode, the processor is requested service only if any key is pressed, otherwise the CPU can proceed with its main task.

In the polled mode, the CPU periodically reads an internal flag of 8279 to check for a key pressure. The Keyboard section can interface an array of a maximum of 64 keys with the CPU. The Keyboard entries (key codes) are debounced and stored in an 8-byte FIFO RAM, that is further accessed by the CPU to read the key codes. If more than eight characters are entered in the FIFO (i.e. more that eight keys are pressed), before any FIFO read operation, the overrun status is set. If a FIFO contains a valid key entry, the CPU is interrupted (in interrupt mode) or the CPU checks the status (in polling) to read the entry. Once the CPU reads a key entry, the FIFO is updated, i.e. the key entry is pushed out of the FIFO to generate space for new entries. The 8279 normally provides a maximum of sixteen 7-seg display interface with CPU It contains a 16-byte display RAM that can be used either as an integrated block of 16x8-bits or two 16x4-bit block of RAM. The data entry to RAM block is controlled by CPU using the command words of the 8279.

✓ **Architecture and Signal Descriptions of 8279**

The Keyboard display controller chip 8279 provides

1. A set of four scan lines and eight return lines for interfacing keyboards.
2. A set of eight output lines for interfacing display.

✓ **I/O Control and Data Buffer**

The I/O control section controls the flow of data to/from the 8279. The data buffer interface the external bus of the system with internal bus of 8279 the I/O section is enabled only if D is low.
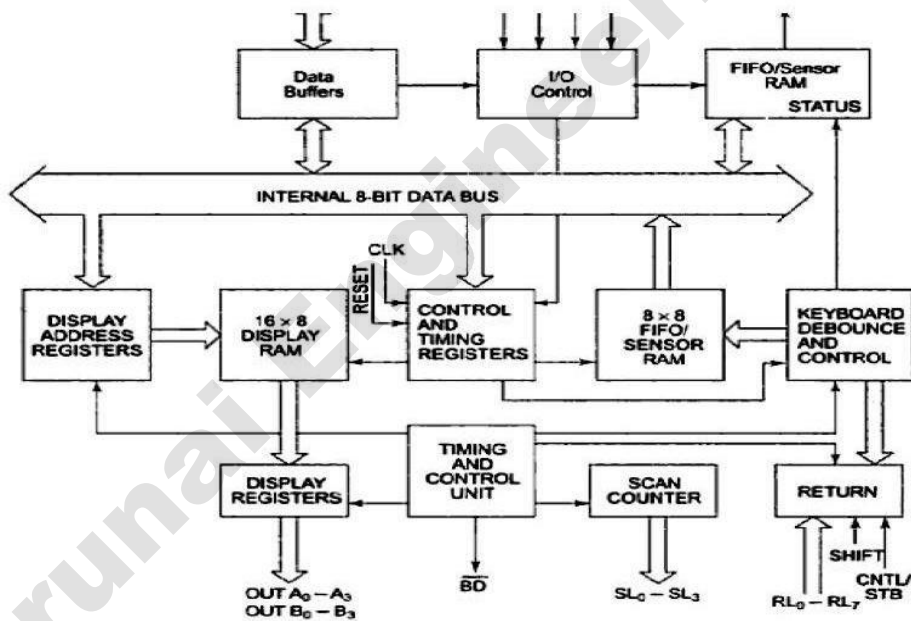
Fig. 3.30 Internal blocks of Keyboard display controller

The pin **Ao**, RD and WR select the command, status or data read/write operations carried out by the CPU with 8279.

✓ **Control and Timing Register and Timing Control**

These registers store the keyboard and display modes and other operating conditions programmed by CPU. The registers are written with Ao=1 and WR =0. The timing and control unit controls the basic timings for the operation of the circuit. Scan Counter divide down the operating frequency of 8279 to derive scan keyboard and scan display frequencies.

✓ **Scan Counter**

The Scan Counter has two modes to scan the key matrix and refresh the display. In the Encoded mode, the counter provides a binary count that is to be externally decoded to provide the scan lines for keyboard and display (four externally decoded scan lines may drive up to 16 displays). In the decoded scan mode, the counter internally decodes the least significant 2 bits and provides a decoded 1 out of 4 scan on SL**0**-SL**3** (four internally decoded scan lines may drive up to 4 Displays). The Keyboard and Display both are in the same mode at a time.

Return Buffers and Keyboard Debounce and Control

This section scans for a Key closure row-wise. If it is detected, the Keyboard debounce unit debounces the key entry (i.e. wait for 10 ms). After the debounce period, if the key continues to be detected. The code of the Key is directly transferred to the sensor RAM along with SHIFT and CONTROL key status.

FIFO/Sensor RAM and Status Logic

In Keyboard or strobed input mode, this block acts as 8-byte first-in-first-out (FIFO) RAM. Each key code of the pressed key is entered in the order of the entry, and in the meantime, read by the CPU, till the RAM becomes empty. The status logic generates an interrupt request after each FIFO read operation till the FIFO is empty.

In scanned sensor matrix mode, this unit acts as sensor RAM. Each row of the sensor RAM is loaded with the status of the corresponding row of sensors in the matrix. If a sensor changes its state, the IRQ line goes high to interrupt the CPU.

Display Address Registers and Display RAM.

The Display address registers hold the addresses of the word currently being written or read by the CPU to or from the display RAM. The contents of the registers are automatically updated by 8279 to accept the next data entry by CPU. The 16-byte display RAM contains the 16-byte of data to be displayed on the sixteen 7-seg displays in the encoded scan mode.

✓ **Pin Diagram of 8279**

DB0 - DB7:

These are bidirectional data bus lines. The data and command words to and from the CPU are transferred on these lines.

CLK:

This is a clock input used to generate internal timings required by 8279.
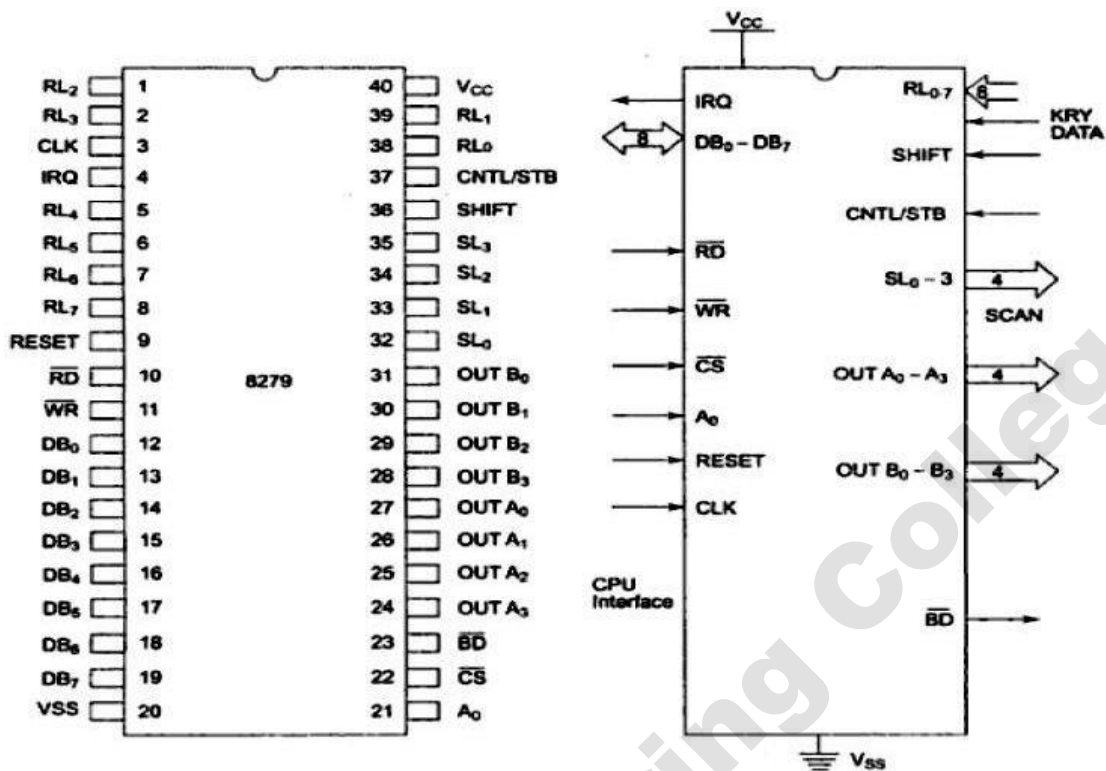
Fig. 3.31 pins and signals of keyboard controller

RESET:

This pin is used to reset 8279. A high on this line resets 8279. After resetting 8279, its in sixteen 8-bit display, left entry encoded scan, 2-key lock out mode. The clock prescaler is set to 31.

CS chip select:

A low on this line enables 8279 for normal read or write operations. Otherwise this pin should be high.

Ao:

A high on the **Ao** line indicates the transfer of a command or status information. A low on this line indicates the transfer of data. This is used to select one of the internal registers of 8279.

RD, WR:

(Input/Output) READ/WRITE input pins enable the data buffer to receive or send data over the data bus.

IRQ:

This interrupt output line goes high when there is data in the FIFO sensor RAM. The interrupt line goes low with each FIFO RAM read operation. However, if the FIFO RAM further contains any Key-code entry to be read by the CPU, this pin again goes high to generate an interrupt to the CPU.

Vss, Vcc:

These are the ground and power supply lines for the circuit.

SL0-SL3 – Scan Lines:

These lines are used to scan the keyboard matrix and display digits. These lines can be programmed as encoded or decoded, using the mode control register.

RL0-RL7 – Return Lines:

These are the input lines which are connected to one terminal of keys, while the other terminal of the keys are connected to the decoded scan lines. These are normally high, but pulled low when a key is pressed.

91

SHIFT:

The status of the Shift input line is stored along with each key code in FIFO in the scanned keyboard mode. Till it is pulled low with a key closure it is pulled up internally to keep it high.

CNTL/STB-CONTROL/STROBED I/P Mode:

In the Keyboard mode, this line is used as a control input and stored in FIFO on a key closure. The line is a strobe line that enters the data into FIFO RAM, in the strobed input mode. It has an internal pull up. The line is pulled down with a Key closure.

BD – Blank Display:

This output pin is used to blank the display during digit switching or by a blanking command.

OUTA0 – OUTA3 and OUTB0 – OUTB3:

These are the output ports for two 16x4 (or one 16 x 8) internal display refresh registers. The data from these lines is synchronized with the scan lines to scan the display and keyboard. The two 4-bit ports may also be used as one 8-bit port.

Modes of Operation of 8279

The Modes of operation of 8279 are

       i. Input (Keyboard) modes
       ii. Output (Display) modes

Input (Keyboard) modes:

8279 provides three input modes, they are:

1. Scanned Keyboard Mode:

This mode allows a key matrix to be interfaced using either encoded or decoded scans. In the encoded scan, an 8 x 8 keyboard or in decoded scan , a 4 x 8 Keyboard can be interfaced. The code of key pressed with SHIFT and CONTROL status is stored into the FIFO RAM.

2. Scanned Sensor Matrix:

In this mode, a sensor array can be interfaced with 8279 using either encoder or decoder scans. With encoder scan 8 x 8 sensor matrix or with decoder scan 4 x 8 sensor matrix can be interfaced. The sensor codes are stored in the CPU addressable sensor RAM.

3. Strobed Input:

In this mode, if the control line goes low, the data on return lines, is stored in the FIFO byte by byte.

Output (Display) Modes:

8279 provides two output modes for selecting the display options.

1. Display Scan:

In this mode, 8279 provides 8 or 16 character multiplexed displays those can be organized as dual 4-bit or single 8-bit display units.

2. Display Entry:

The Display data is entered for display either from the right side or from the left side.

Details of Modes of Operation

Keyboard Modes

1. Scanned Keyboard Mode with 2 Key Lockout

In this mode of operation, when a key is pressed, a debounce logic comes into operation. The Key code of the identified key is entered into the FIFO with SHIFT and CNTL status, provided the FIFO is not full.

2. Scanned Keyboard with N-key Rollover

In this mode, each key depression is treated independently. When a key is pressed, the debounce circuit waits for 2 keyboard scans and then checks whether the key is still depressed. If it is still depressed, the code is entered in FIFO RAM. Any number of keys can be pressed simultaneously and recognized in the order, the Keyboard scan record them.

3. Scanned Keyboard Special Error Mode

This mode is valid only under the N-Key rollover mode. This mode is programmed using *end interrupt/error mode set* command. If during a single debounce period (two Keyboard scan) two keys are found pressed, this is considered a simultaneous depression and an error flag is set. This flag, if set, prevents further writing in FIFO but allows generation of further interrupts to the CPU for FIFO read.

3. Sensor Matrix Mode

In the Sensor Matrix mode, the debounce logic is inhibited the 8-byte memory matrix. The status of the sensor switch matrix is fed directly to sensor RAM matrix Thus the sensor RAM bits contains the row-wise and column-wise status of the sensors in the sensor matrix.

8

Display Modes

There are various options of data display The first one is known as left entry mode or type writer mode. Since in a type writer the first character typed appears at the left-most position, while the subsequent characters appears successively to the right of the first one. The other display format is known as right entry mode, or calculator mode, since the calculator the first character entered appears to the right-most position and this character is shifted one position left when the next character is entered.

1. Left Entry Mode

In the Left entry mode, the data is entered from the left side of the display unit. Address 0 of the display RAM contains the leftmost display character and address 15 of the RAM contains the rightmost display character.

2. Right Entry Mode

In the right entry mode, the first entry to be displayed is entered on the rightmost display. The next entry is also placed in the right most display but after the previous display is shifted left by one display position.

✓ **Command Words of 8279**

All the Command words or status words are written or read with Ao = 1 and CS = 0 to or from 8279.

a. Keyboard Display mode set

The format of the command word to select different modes of operation of 8279 is given below with its bit definitions.

b. Programmable Clock

The clock for operation of 8279 is obtained by dividing the external clock input signal by a programmable constant called prescaler.

PPPPP is a 5-bit binary constant. The input frequency is divided by a decimal constant ranging from 2 to 31, decided by the bits of an internal prescalar, PPPPP.

c. Read FIFO/Sensor RAM

The format of this command is given as shown below

X - don't care

AI - Auto increment flag

AAA - Address pointer to 8 bit FIFO RAM

This word is written to set up 8279 for reading FIFO/Sensor RAM. In scanned keyboard mode, AI and AAA bits are of no use. The 8279 will automatically drive data bus for each subsequent read, in the same sequence, in which the data was entered.

d. Read Display RAM

This command enables a programmer to read the display RAM data The CPU writes this command word to 8279 to prepare it for display RAM read operation. AI is auto incremented flag and AAAA, the 4-bit address, points to the 16-byte display RAM that is to be read. If AI = 1, the address will be automatically, incremented after each read or write to the display RAM.

e. Write Display RAM

The format of this command is given as shown below

AI - Auto increment flag

AAAA - 4-bit address for 16-bit display RAM to be written

Other details of this command are similar to the 'Read Display RAM Command.

f. Display Write Inhibit/Blanking

The IW (Inhibit write flag) bits are used to mask the individual nibble Here Do and D2 corresponds to OUTBo – OUTB3 while D**1** and D**3** corresponds to OUTAo-OUTA3 for blanking and masking respectively.

g. Clear Display RAM

The CD**2**, CD**1**, CDo is a selectable blanking code to clear all the rows of the display RAM as given below. The characters A and B represents the output nibbles.

CD CD**1** CDo

1 0 x All Zeros (x don't care) AB = 00

1 1 0 A**3**-Ao = 2(0010) and B**3**-Bo = 00(0000)

1 1 1 All ones (AB = FF), i.e. clear RAM

Here, CA represents clear All and CF represents Clear FIFO RAM

End Interrupt/Error Mode Set

For the sensor matrix mode, this command lowers the IRQ line and enables further writing into the RAM. Otherwise, if a charge in sensor value is detected, IRQ goes high that inhibits writing in the sensor RAM.

Key-code and status Data Formats

This briefly describes the formats of the Key-code/Sensor data in their respective modes of operation and the FIFO Status Word formats of 8279.

Key-code Data Formats:

After a valid Key closure, the key code is entered as a byte code into the FIFO RAM, in the following format, in scanned keyboard mode. The Keycode format contains 3-bit contents of the internal row counter, 3-bit contents of the column counter and status of the SHIFT and CNTL Keys The data format of the Keycode in scanned keyboard mode is given below. In the sensor matrix mode, the data from the return lines is directly entered into an appropriate row of sensor RAM, that identifies the row of the sensor that changes its status. The SHIFT and CNTL Keys are ignored in this mode. RL bits represent the return lines.

Rn represents the sensor RAM row number that is equal to the row number of the sensor array in which the status change was detected. Data Format of the sensor code in sensor matrix mode

FIFO Status Word:

The FIFO status word is used in keyboard and strobed input mode to indicate the error. Overrun error occurs, when an already full FIFO is attempted an entry, Under run error occurs when an empty FIFO read is attempted. FIFO status word also has a bit to show the unavailability of FIFO RAM because of the ongoing clearing operation.

In sensor matrix mode, a bit is reserved to show that at least one sensor closure indication is stored in the RAM, The S/E bit shows the simultaneous multiple closure error in special error mode. In sensor matrix mode, a bit is reserved to show that at least one sensor closure indication is stored in the RAM, The S/E bit shows the simultaneous multiple closure error in special error mode.

Interfacing and Programming 8279

Problem:

Interface keyboard and display controller 8279 with 8086 at address 0080H. Write an ALP to set up 8279 in scanned keyboard mode with encoded scan, N-Key rollover mode. Use a 16 character display in right entry display format. Then clear the display RAM with zeros. Read the FIFO for key closure. If any key is closed, store it's code to register CL. Then write the byte 55 to all the displays, and return to DOS. The clock input to 8279 is 2MHz, operate it at 100KHz.

Solution:

_ The 8279 is interfaced with lower byte of the data bus, i.e. Do-D**7** . Hence the Ao input of 8279 is connected with address lineA**1**.
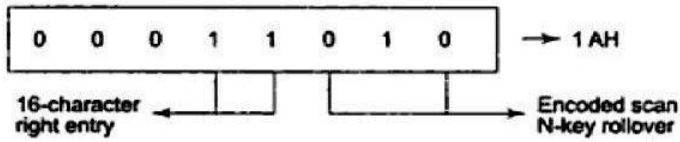
_ The data register of 8279 is to be addressed as 0080H, i.e.Ao=0.

_ For addressing the command or status word Ao input of 8279 should be 1.

_ The next step is to write all the required command words for this problem.

Keyboard/Display Mode Set CW:

This command byte sets the 8279 in 16-character right entry and encoded scan N-Key rollover mode.

Program clock selection:

The clock input to 8279 is 2MHz, but the operating frequency is to be 100KHz, i.e. the clock input is to be divided by 20 (10100). Thus the prescalar value is 10100 and trhe command byte is set as given.
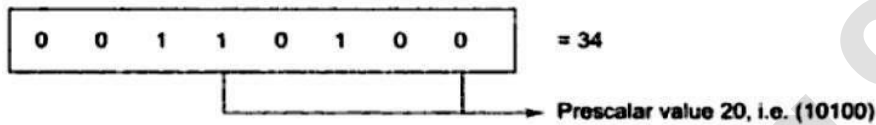
Clear Display RAM:

This command clears the display RAM with the programmable blanking code.
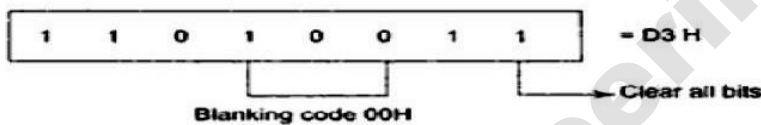
Read FIFO:

This command byte enables the programmer to read a key code from the FIFO RAM
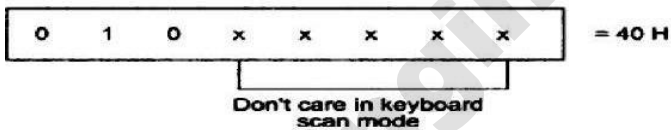
Write Display RAM:

This command enables the programmer to write the addressed display locations of the RAM as presented below.
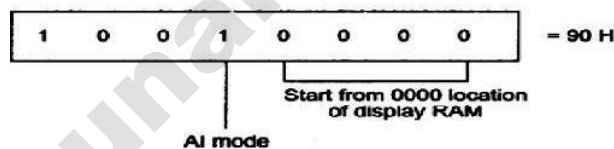
3.7 Interrupt Controller

The Intel 8259A Programmable Interrupt Controller handles up to eight vectored priority interrupts for the CPU. It is cascadable for up to 64 vectored priority interrupts without additional circuitry. It is packaged in a 28-pin DIP, uses NMOS technology and requires a single a5V supply. Circuitry is static, requiring no clock input.

The 8259A is designed to minimize the software and real time overhead in handling multi-level priority interrupts.

It has several modes, permitting optimization for a variety of system requirements. The 8259A is fully upward compatible with the Intel 8259. Software originally written for the

8259 will operate the 8259A in all 8259 equivalent modes (MCS-80/85, Non-Buffered and Edge Triggered).
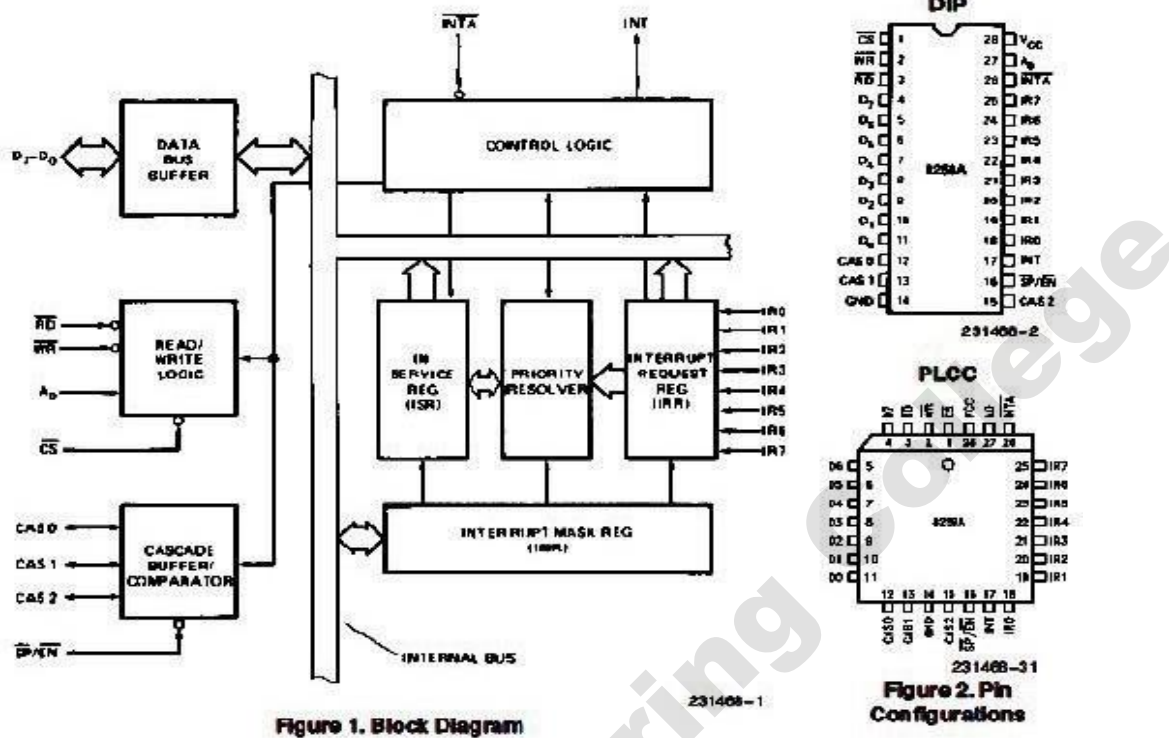

Figure 1. Block Diagram

Figure 2. Pin Configurations

Fig.3 Functional block diagram of 8259

The microprocessor will be executing its main program and only stop to service peripheral devices when it is told to do so by the device itself. In effect, the method would provide an external asynchronous input that would inform the processor that it should complete whatever instruction that is currently being executed and fetch a new routine that will service the requesting device. Once this servicing is complete, however, the processor would resume exactly where it left off.This method is called Interrupt.

System throughput would drastically increase, and thus more tasks could be assumed by the microcomputer to further enhance its cost effectiveness. The Programmable Interrupt Controller (PIC) functions as an overall manager in an Interrupt-Driven system environment. It accepts requests from the peripheral equipment, determines which of the incoming requests is of the highest importance (priority), a rtains whether the incoming request has a higher priority value than the level currently being serviced, and issues an interrupt to the CPU based on this determination.

Each peripheral device or structure usually has a special program or ``routine'' that is associated with its specific functional or operational requirements; this is referred to as a ``service routine''. The PIC, after issuing an Interrupt to the CPU, must somehow input information into the CPU that can ``point'' the Program Counter to the service routine associated with the requesting device. This ``pointer'' is an address in a vectoring table and will often be referred to, in this document, as vectoring data.

- ✓ **Interrupt request register (IRR) AND in-service register (ISR):**

The interrupts at the IR input lines are handled by two registers in cascade, the Interrupt Request Register (IRR) and the In-Service (ISR). The IRR is used to store all the interrupt levels which are requesting service; and the ISR is used to store all the interrupt levels which are being serviced.

- ✓ **Priority resolver**

This logic block determines the priorites of the bits set in the IRR. The highest priority is selected and strobed into the corresponding bit of the ISR during INTA pulse.

- ✓ **Interrupt mask register (IMR)**

The IMR stores the bits which mask the interrupt lines to be masked. The IMR operates on the IRR.Masking of a higher priority input will not affect the interrupt request lines of lower quality.

- ✓ **INT (INTERRUPT)**

This output goes directly to the CPU interrupt input. The VOH level on this line is designed to be fully compatible with the 8080A, 8085A and 8086 input levels.

- ✓ **INTA (INTERRUPT ACKNOWLEDGE)**

INTA pulses will cause the 8259A to release vectoring information onto the data bus. The format of this data depends on the system mode (mPM) of the 8259A.

- ✓ **Data bus buffer**

This 3-state, bidirectional 8-bit buffer is used to interface the 8259A to the system Data Bus. Control words and status information are transferred through the Data Bus Buffer.

- ✓ **Read/write control logic**

The function of this block is to accept OUTput commands from the CPU. It contains the Initialization Command Word (ICW) registers and Operation Command Word (OCW) registers
which store the various control formats for device operation. This function block also allows the status of the 8259A to be transferred onto the Data Bus.

- ✓ **CS (CHIP SELECT)**

A LOW on this input enables the 8259A. No reading or writing of the chip will occur unless thedevice is selected.

- ✓ **WR (WRITE)**

A LOW on this input enables the CPU to write control words (ICWs and OCWs) to the 8259A.

- ✓ **RD (READ)**

A LOW on this input enables the 8259A to send the status of the Interrupt Request Register (IRR),In Service Register (ISR), the Interrupt Mask Register (IMR), or the Interrupt level onto the Data Bus.

- ✓ **A0**

This input signal is used in conjunction with WR and RD signals to write commands into the various command registers, as well as reading the various status registers of the chip. This line can be tied directly to one of the address lines.

3.7 Interrupt sequence

The powerful features of the 8259A in a microcomputer system are its programmability and the interrupt routine addressing capability. The latter allows direct or indirect jumping to the specific interrupt routine requested without any polling of the interrupting devices. The normal sequence of events during an interrupt depends on the type of CPU being used.

The events occur as follows:

1. One or more of the INTERRUPT REQUEST lines (IR7±0) are raised high, setting the corresponding IRR bit(s).

2. The 8259A evaluates these requests, and sends an INT to the CPU, if appropriate.

3. The CPU acknowledges the INT and responds with an INTA pulse.

4. Upon receiving an INTA from the CPU group, the highest priority ISR bit is set, and the corresponding IRR bit is reset. The 8259A will also release a CALL instruction code (11001101) onto the 8-bit Data Bus through its D7±0 pins.

5. This CALL instruction will initiate two more INTA pulses to be sent to the 8259A from the CPU group.

6. These two INTA pulses allow the 8259A to release its preprogrammed subroutine address onto the Data Bus. The lower 8-bit address is released at the first INTA pulse and the higher 8-bit address is released at the second INTA pulse.

7. This completes the 3-byte CALL instruction released by the 8259A. In the AEOI mode the ISR bit is reset at the end of the third INTA pulse. Otherwise, the ISR bit remains set until an appropriate EOI command is issued at the end of the interrupt sequence. The events occurring in an 8086 system are the same until step 4.

8. Upon receiving an INTA from the CPU group, the highest priority ISR bit is set and the corresponding IRR bit is reset. The 8259A does not drive the Data Bus during this cycle.

9. The 8086 will initiate a second INTA pulse. During this pulse, the 8259A releases an 8-bit pointer onto the Data Bus where it is read by the CPU.

10. This completes the interrupt cycle. In the AEOI mode the ISR bit is reset at the end of the second INTA pulse. Otherwise, the ISR bit remains set until an appropriate EOI command is issued at the end of the interrupt subroutine. If no interrupt request is present at step 4 of either sequence (i.e., the request was too short in duration) the 8259A will issue an

interrupt level 7. Both the vectoring bytes and the CAS lines will look like an interrupt level 7 was requested. When the 8259A PIC receives an interrupt, INT becomes active and an interrupt acknowledge cycle is started. If a higher priority interrupt occurs between the two INTA pulses, the INT line goes inactive immediately after the second INTA pulse. After an unspecified amount of time the INT line is activated again to signify the higher priority interrupt waiting for service.

3.8 DMA Controller -DMA Controller 8257

The *Direct Memory Access* or DMA mode of data transfer is the fastest amongst all the modes of data transfer. In this mode, the device may transfer data directly to/from memory without any interference from the CPU. The device requests the CPU (through a DMA controller) to hold its data, address and control bus, so that the device may transfer data directly to/from memory.

The DMA data transfer is initiated only after receiving HLDA signal from the CPU. Intel's 8257 is a four channel DMA controller designed to be interfaced with their family of microprocessors. The 8257, on behalf of the devices, requests the CPU for bus access using local bus request input i.e. HOLD in minimum mode.

In maximum mode of the microprocessor RQ/GT pin is used as bus request input. On receiving the HLDA signal (in minimum mode) or RQ/GT signal (in maximum mode) from the CPU, the requesting devices gets the access of the bus, and it completes the required number of DMA cycles for the data transfer and then hands over the control of the bus back to the CPU.

Internal Architecture of 8257

The internal architecture of 8257 is shown in figure. The chip support four DMA channels, i.e. four peripheral devices can independently request for DMA data transfer through these channels at a time. The DMA controller has 8-bit internal data buffer, a read/write unit, a control unit, a priority resolving unit along with a set of registers. The 8257 performs the DMA operation over four independent DMA channels. Each of four channels of 8257 has a pair of two 16-bit registers, viz. DMA *address register* and *terminal count register*.

There are two common registers for all the channels, namely, *mode set register* and *status register*. Thus there are a total of ten registers. The CPU selects one of these ten registers using address lines A_0-A_3. Table shows how the A_0-A_3 bits may be used for selecting one of these registers.

DMA Address Register

Each DMA channel has one DMA address register. The function of this register is to store the address of the starting memory location, which will be accessed by the DMA channel. Thus the starting address of the memory block which will be accessed by the device is first loaded in the DMA address register of the channel. The device that wants to transfer data over a DMA channel, will access the block of the memory with the starting address stored in the DMA Address Register.

Terminal Count Register

Each of the four DMA channels of 8257 has one terminal count register (TC). This 16-bit register issued for a rtaining that the data transfer through a DMA channel ceases or stops after the required number of DMA cycles. The low order 14-bits of the terminal count register are initialized with the binary equivalent of the number of required DMA cycles minus one.

After each DMA cycle, the terminal count register content will be decremented by one and finally it becomes zero after the required number of DMA cycles are over. The bits 14 and 15 of this register indicate the type of the DMA operation (transfer). If the device wants to write data into the memory, the DMA operation is called DMA write operation. Bit 14 of the register in this case will be set to one and bit 15 will be set to zero.

99

Table gives detail of DMA operation selection and corresponding bit configuration of bits 14 and 15 of the TC register.

Table 3. DMA operation selection using A_{15}/RD and A_{15}/WR

Bit 15	Bit 14	Type of DMA Operation
0	0	Verify DMA Cycle
0	1	Write DMA Cycle
1	0	Read DMA Cycle
1	1	(Illegal)

Mode Set Register

The mode set register is used for programming the 8257 as per the requirements of the system. The function of the mode set register is to enable the DMA channels individually and also to set the various modes of operation.

The DMA channel should not be enabled till the DMA address register and the terminal count register contain valid information, otherwise, an unwanted DMA request may initiate a DMA cycle, probably destroying the valid memory data. The bits D_0-D_3 enable one of the four DMA channels of 8257. for example, if D_0 is '1', channel 0 is enabled. If bit 4 is set, rotating priority is enabled, otherwise, the normal, i.e. fixed priority is enabled.
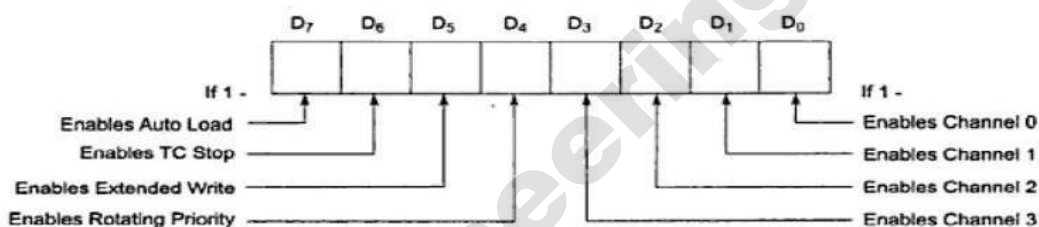
Fig 3.32 Bit definition of the mode set register

If the TC STOP bit is set, the selected channel is disabled after the *terminal count* condition is reached, and it further prevents any DMA cycle on the channel. To enable the channel again, this bit must be reprogrammed. If the TC STOP bit is programmed to be zero, the channel is not disabled, even after the count reaches zero and further request are allowed on the same channel.

The auto load bit, if set, enables channel 2 for the repeat block chaining operations, without immediate software intervention between the two successive blocks. The channel 2 registers are used as usual, while the channel 3 registers are used to store the block reinitialisation parameters, i.e. the DMA starting address and terminal count.

After the first block is transferred using DMA, the channel 2 registers are reloaded with the corresponding channel 3 registers for the next block transfer, if the *update* flag is set. The extended write bit, if set to '1', extends the duration of MEMW and IOW signals by activating them earlier, this is useful in interfacing the peripherals with different access times.

If the peripheral is not accessed within the stipulated time, it is expected to give the 'NOT READY' indication to 8257, to request it to add one or more wait states in the DMA CYCLE. The mode set register can only be written into.

Status Register

The status register of 8257 is shown in figure. The lower order 4-bits of this register contain the terminal count status for the four individual channels. If any of these bits is set, it indicates that the specific channel has reached the terminal count condition.
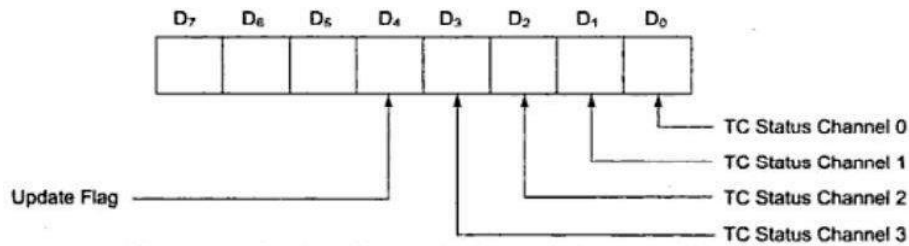
Fig 3.33 Status Register

These bits remain set till either the status is read by the CPU or the 8257 is reset.

The update flag is not affected by the read operation. This flag can only be cleared by resetting 8257 or by resetting the auto load bit of the mode set register. If the update flag is set, the contents of the channel 3 registers are reloaded to the corresponding registers of channel 2 whenever the channel 2 reaches a terminal count condition, after transferring one block and the next block is to be transferred using the auto load feature of 8257. The update flag is set every time; the channel 2 registers are loaded with contents of the channel 3 registers. It is cleared by the completion of the first DMA cycle of the new block. This register can only read.

Data Bus Buffer, Read/Write Logic, Control Unit and Priority Resolver

The 8-bit. Tristate, bidirectional buffer interfaces the internal bus of 8257 with the external system bus under the control of various control signals.

In the slave mode, the read/write logic accepts the I/O Read or I/O Write signals, decodes the Ao-A3 lines and either writes the contents of the data bus to the addressed internal register or reads the contents of the selected register depending upon whether IOW or IOR signal is activated.

In master mode, the read/write logic generates the IOR and IOW signals to control the data flow to or from the selected peripheral. The control logic controls the sequences of operations and generates the required control signals like AEN, ADSTB, MEMR, MEMW, TC and MARK along with the address lines A4-A7, in master mode. The priority resolver resolves the priority of the four DMA channels depending upon whether normal priority or rotating priority is programmed.

Signal Description of 8257

DRQo-DRQ3:

These are the four individual channel DMA request inputs, used by the peripheral devices for requesting the DMA services. The DRQo has the highest priority while DRQ**3** has the lowest one, if the fixed priority mode is selected.

DACKo-DACK3:

These are the active-low DMA acknowledge output lines which inform the requesting peripheral that the request has been honoured and the bus is relinquished by the CPU. These lines may act as strobe lines for the requesting devices.
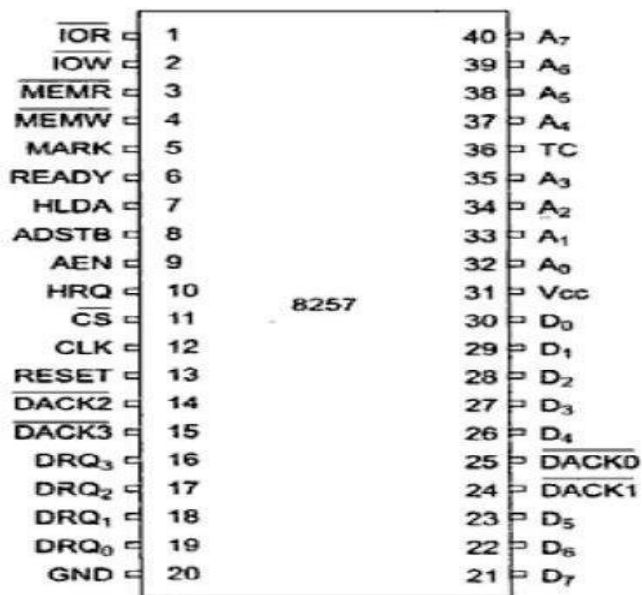
Fig 3.34 Pin Diagram of 8257

Do-D7:

These are bidirectional, data lines used to interface the system bus with the internal data bus of 8257. These lines carry command words to 8257 and status word from 8257, in slave mode, i.e. under the control of CPU. The data over these lines may be transferred in both the directions. When the 8257 is the bus master (master mode, i.e. not under CPU control), it uses Do-D7 lines to send higher byte of the generated address to the latch. This address is further latched using ADSTB signal. the address is transferred over Do-D7 during the first clock cycle of the DMA cycle. During the rest of the period, data is available on the data bus.

IOR:

This is an active-low bidirectional tristate input line that acts as an input in the slave mode. In slave mode, this input signal is used by the CPU to read internal registers of 8257.this line acts output in master mode. In master mode, this signal is used to read data from a peripheral during a memory write cycle.

IOW:

This is an active low bidirection tristate line that acts as input in slave mode to load the contents of the data bus to the 8-bit mode register or upper/lower byte of a 16-bit DMA address register or terminal count register. In the master mode, it is a control output that loads the data to a peripheral during DMA memory read cycle (write to peripheral).

CLK:

This is a clock frequency input required to derive basic system timings for the internal operation of 8257.

RESET:

This active-high asynchronous input disables all the DMA channels by clearing the mode register and tristates all the control lines.

Ao-A3:

These are the four least significant address lines. In slave mode, they act as input which select one of the registers to be read or written. In the master mode, they are the four least significant memory address output lines generated by 8257.

CS:

This is an active-low chip select line that enables the read/write operations from/to 8257, in slave mode. In the master mode, it is automatically disabled to prevent he chip from getting selected (by CPU) while performing the DMA operation.

A4-A7:

This is the higher nibble of the lower byte address generated by 8257 during the master mode of DMA operation.

READY:

This is an active-high asynchronous input used to stretch memory read and write cycles of 8257 by inserting wait states. This is used while interfacing slower peripherals..

HRQ:

The hold request output requests the access of the system bus. In the noncascaded 8257 systems, this is connected with HOLD pin of CPU. In the cascade mode, this pin of a slave is connected with a DRQ input line of the master 8257, while that of the master is connected with HOLD input of the CPU.

HLDA:

The CPU drives this input to the DMA controller high, while granting the bus to the device. This pin is connected to the HLDA output of the CPU. This input, if high, indicates to the DMA controller that the bus has been granted to the requesting peripheral by the CPU.

MEMR:

This active –low memory read output is used to read data from the addressed memory locations during DMA read cycles.

MEMW:

This active-low three state output is used to write data to the addressed memory location during DMA write operation.

ADST:

This output from 8257 strobes the higher byte of the memory address generated by the DMA controller into the latches.

AEN:

This output is used to disable the system data bus and the control the bus driven by the CPU, this may be used to disable the system address and data bus by using the enable input of the bus drivers to inhibit the non-DMA devices from responding during DMA operations. If the 8257 is I/O mapped, this should be used to disable the other I/O devices, when the DMA controller addresses is on the address bus.

TC:

Terminal count output indicates to the currently selected peripherals that the present DMA cycle is the last for the previously programmed data block. If the TC STOP bit in the mode set register is set, the selected channel will be disabled at the end of the DMA cycle. The TC pin is activated when the 14-bit content of the terminal count register of the selected channel becomes equal to zero. The lower order 14 bits of the terminal count register are to be programmed with a 14-bit equivalent of (n-1), if n is the desired number of DMA cycles.

MARK:

The modulo 128 mark output indicates to the selected peripheral that the current DMA cycle is the 128th cycle since the previous MARK output. The mark will be activated after each 128 cycles or integral multiples of it from the beginning if the data block (the first DMA cycle), if the total number of the required DMA cycles (n) is completely divisible by 128.

Vcc:

This is a +5v supply pin required for operation of the circuit.

GND:

This is a return line for the supply (ground pin of the IC).

Interfacing 8257 with 8086

Once a DMA controller is initialized by a CPU property, it is ready to take control of the system bus on a DMA request, either from a peripheral or itself (in case of memory-to memory transfer). The DMA controller sends a HOLD request to the CPU and waits for the CPU to assert the HLDA signal. The CPU relinquishes the control of the bus before asserting the HLDA signal
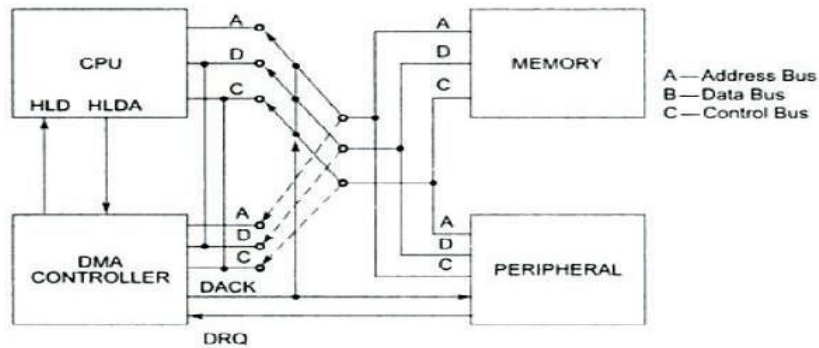
Fig 3.35 Interfacing 8257 with CPU

Once the HLDA signal goes high, the DMA controller activates the DACK signal to the requesting peripheral and gains the control of the system bus. The DMA controller is the sole master of the bus, till the DMA operation is over. The CPU remains in the HOLD status (all of its signals are tristate except HOLD and HLDA), till the DMA controller is the master of the bus. In other words, the DMA controller interfacing circuit implements a switching arrangement for the address, data and control busses of the memory and peripheral subsystem from/to the CPU to/from the DMA controller.

3.9 Traffic Light Control: Traffic Light Controller Using 8086

Traffic light controller interface module is designed to simulate the function of four way traffic light controller.

Combinations of red, amber and green LED's are provided to indicate Halt, Wait and Go signals for vehicles.

Combination of red and green LED's are provided for pedestrian crossing.

36 LED's are arranged in the form of an intersection.

A typical junction is represented on the PCB with comprehensive legend printing.

At the left corner of each road, a group of five LED's (red, amber and 3 green) are arranged in the form of a T-section to control the traffic of that road.

Each road is named North (N), South(S), East (E) and West (W). LED's L1, L10, L19 & L28 (Red) are for the stop signal for the vehicles on the road N, S, W, & E respectively.

L2, L11, L20 & L29 (Amber) indicates wait state for vehicles on the road N, S, W, & E respectively. L3, L4 & L5 (Green) are for left, strait and right turn for the vehicles on road S. similarly L12-L13-L14, L23-L22-L21 & L32-L31-L30 simulates same function for the roads E, N, W respectively.

A total of 16 LED's (2 Red & 2 Green at each road) are provided for pedestrian crossing. L7-L9.L16-L18, L25-L27 & L34-L36 (Green) when on allows pedestrians to cross and L6-L8, L15-L17, L24-L26 & L33-L35 (Red) when on alarms the pedestrians to wait.

To minimize the hardware pedestrian's indicator LED's (both red and green are connected to same port lines (PC4 to PC7) with red inverted.

Red LED's L10 & L28 are connected to port lines PC2 & PC3 while L1 & L19 are connected to lines PC0 & PC1 after inversion. All other LED's (amber and green) are connected to port A & B.

· **Working:-** 8255 is interfaced with 8086 in I/O mapped I/O and all ports are output ports. The basic operation of the interface is explained with the help of the enclosed program. The enclosed program assumes no entry of vehicles from North to West, from road East to South.

At the beginning of the program all red LED's are switch ON, and all other LED's are switched OFF. Amber LED is switched ON before switching over to proceed state from Halt state.

The sequence of traffic followed in the program is given below.

104

a) From road north to East
   From road east to north
   From road south to west
   From road west to south
   From road west to north
b) From road north to East
   From road south to west
   From road south to north
   From road south to east
   From road north to south
   From road south to north
   Pedestrian crossing at roads west & east
d) From road east to west
   From road west to east
   Pedestrian crossing at roads north & south

UNIT-IV
MICROCONTROLLER

DEVICE	ON-CHIP DATA MEMORY (bytes)	ON-CHIP PROGRAM MEMORY (bytes)	16-BIT TIMER/COUNTER	NO. OF VECTORED INTERUPTS	FULL DUPLEX I/O
8031	128	None	2	5	1
8032	256	none	2	6	1
8051	128	4k ROM	2	5	1
8052	256	8k ROM	3	6	1
8751	128	4k EPROM	2	5	1
8752	256	8k EPROM	3	6	1
AT89C51	128	4k Flash Memory	2	5	1
AT89C52	256	8k Flash memory	3	6	1

4.1 Architecture of 8051:

• It is a single chip
• Consists of CPU, Memory
• I/O ports, timers and other peripherals
• It is a CPU
• Memory, I/O Ports to be connected externally.

- Small size, low power, low cost;

- Harvard architecture with separate program and data memory;

- No data corruption or loss of data; but with complex circuit

- The 8051 has three very general types of memory.

- On-Chip Memory refers to any memory (Code, RAM, or other) that physically exists on the microcontroller itself. On-chip memory can be of several types.

·   External Code Memory is code (or program) memory that resides off-chip. This is often in the form of an external EPROM.

·   External RAM is RAM memory that resides off-chip. This is often in the form of standard static RAM or flash RAM.

The 8051 is a flexible microcontroller with a relatively large number of modes of operations.

Your program may inspect and/or change the operating mode of the 8051 by manipulating the values of the 8051's Special Function Registers(SFRs).

SFRs are accessed as if they were normal Internal RAM. The only difference is that Internal RAM is from address 00h through 7Fh whereas SFR registers exist in the address range of 80h through FFh
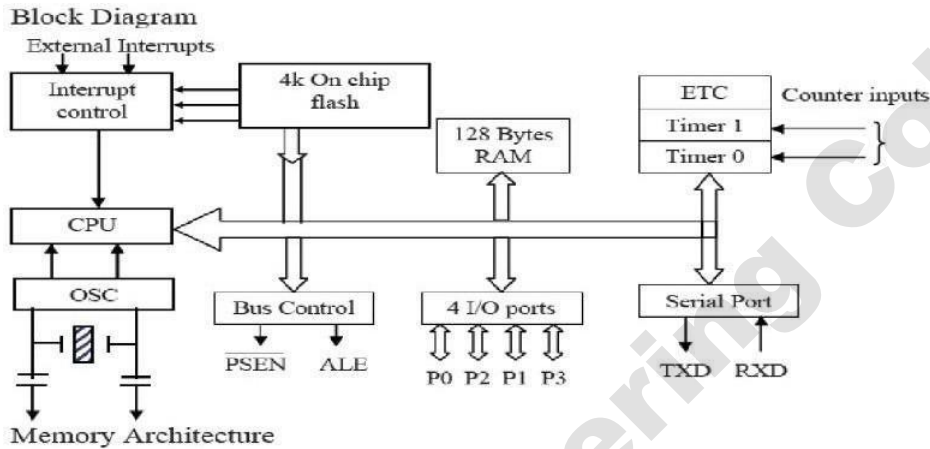
Fig. 4.1 8051 architecture

8051 Clock and Instruction Cycle
In 8051, one instruction cycle consists of twelve (12) clock cycles. Instruction cycle is sometimes called as Machine cycle by some authors.
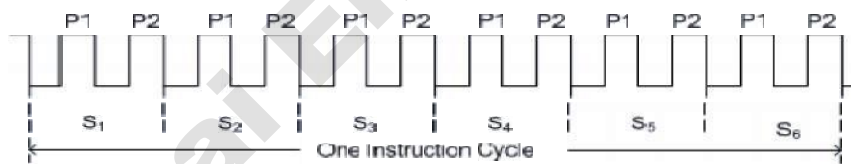
Fig 4.2 Instruction cycle of 8051

In 8051, each instruction cycle has six states (S$_1$- S$_6$). Each state has two pulses (P1 and P2)
128 bytes of Internal RAM Structure (lower address space)
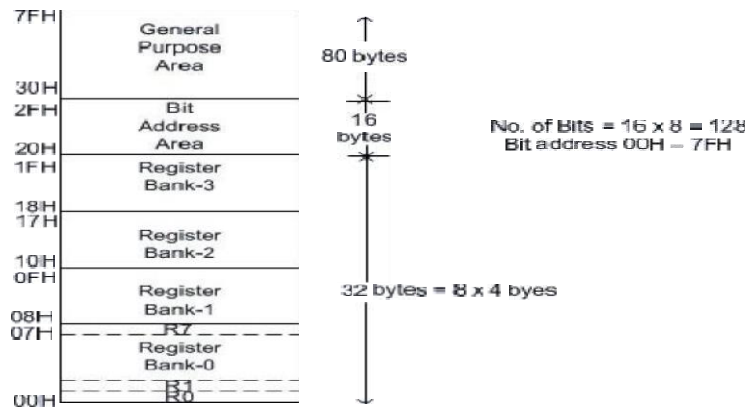
Fig 4.3: Internal RAM Structure

The lower 32 bytes are divided into 4 separate banks. Each register bank has 8 registers of one byte each. A register bank is selected depending upon two bank select bits in the PSW register. Next 16bytes are bit addressable. In total, 128bits (16X8) are available in addressable area. Each bit can be accessed and modified by suitable instructions. The bit addresses are from 00H (LSB of the first byte in 20H) to 7FH (MSB of the last byte in 2FH). Remaining 80bytes of RAM are available for general purpose.

Internal Data Memory and Special Function Register (SFR) Map
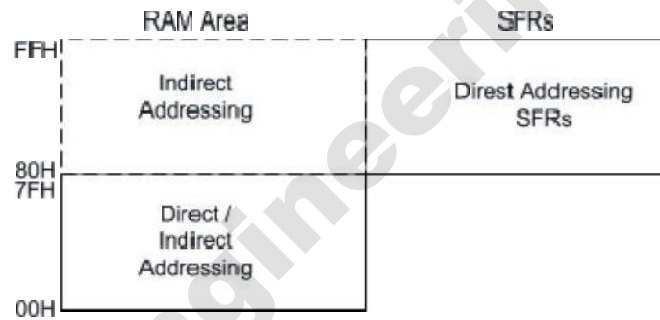
Fig. 4.4 Internal Data Memory Map

The special function registers (SFRs) are mapped in the upper 128 bytes of internal data memory address. Hence there is an address overlap between the upper 128 bytes of data RAM and SFRs. Please note that the upper 128 bytes of data RAM are present only in the 8052 family. The lower128 bytes of RAM (00H - 7FH) can be accessed both by direct or indirect addressing while the upper 128 bytes of RAM (80H - FFH) are accessed by indirect addressing.The SFRs (80H - FFH) are accessed by direct addressing only. This feature distinguishes the upper 128 bytes of memory from the SFRs, as shown in fig 5.

Processor Status Word (PSW)   Address=D0H

CY	AC	F0	RS1	RS0	OV	-	P
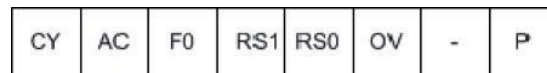
Fi g 4.5: Processor Status Word

PSW register stores the important status conditions of the microcontroller. It also stores the bank select bits (RS1 & RS0) for register bank selection.

4.2 Special Function Registers (SFRs):

The 8051 is a flexible microcontroller with a relatively large number of modes of operations. Your program may inspect and/or change the operating mode of the 8051 by manipulating the values of the 8051's Special Function Registers (SFRs).

SFRs are accessed as if they were normal Internal RAM. The only difference is that Internal RAM is from address 00h through 7Fh whereas SFR registers exist in the address range of 80h through FFh. Each SFR has an address (80h through FFh) and a name.

The following chart provides a graphical presentation of the 8051's SFRs, their names, and their address. As you can see, although the address range of 80h through FFh offer 128 possible addresses, there are only 21 SFRs in a standard 8051. All other addresses in the SFR range (80h through FFh) are considered invalid. Writing to or reading from these registers may produce undefined values or behavior.

SFR Types

SFRs related to the I/O ports: The 8051 has four I/O ports of 8 bits, for a total of 32 I/O lines. Whether a given I/O line is high or low and the value read from the line are controlled by the SFRs.

The SFRs control the operation or the configuration of some aspect of the 8051. For example, **TCON** controls the timers, SCON controls the serial port, the remaining SFRs, are auxillary SFRs in the sense that they don't directly configure the 8051 but obviously the 8051 cannot operate without them. For example, once the serial port has been configured using **SCON**, the program may read or write to the serial port using the **SBUF** register.

SFR Descriptions

P0 (Port 0, Address 80h, Bit-Addressable): This is input/output port 0. Each bit of this SFR corresponds to one of the pins on the microcontroller. For example, bit 0 of port 0 is pin P0.0, bit 7 is in P0.7. Writing a value of 1 to a bit of this SFR will send a high level on the corresponding I/O pin whereas a value of 0 will bring it to a low level.

SP (Stack Pointer, Address 81h): This is the stack pointer of the microcontroller. This SFR indicates where the next value to be taken from the stack will be read from in Internal RAM. If you push a value onto the stack, the value will be written to the address of SP + 1. This SFR is modified by all instructions which modify the stack, such as PUSH, POP, LCALL, RET, RETI, and whenever interrupts are provoked by the microcontroller. The Stack Pointer, like all registers except DPTR and PC, may hold an 8-bit (1-byte) value.

When you pop a value off the stack, the 8051 returns the value from the memory location indicated by SP, and then decrements the value of SP.

This order of operation is important. When the 8051 is initialized SP will be initialized to 07h. If you immediately push a value onto the stack, the value will be stored in Internal RAM address 08h.

First the 8051 will increment the value of SP (from 07h to 08h) and then will store the pushed value at that memory address (08h). It is also used intrinsically whenever an interrupt is triggered .

DPL/DPH (Data Pointer Low/High, Addresses 82h/83h): The SFRs DPL and DPH work together to represent a 16-bit value called the *Data Pointer*. The data pointer is used in operations regarding external RAM and some instructions involving code memory. Since it is an unsigned two-byte integer value, it can represent values from 0000h to FFFFh (0 through 65,535 decimal).

PCON (Power Control, Addresses 87h): The Power Control SFR is used to control the 8051's power control modes. Certain operation modes of the 8051 allow the 8051 to go into a type of "sleep" mode which requires much less power. These modes of operation are controlled through PCON. Additionally, one of the bits in PCON is used to double the effective baud rate of the 8051's serial port.

TCON (Timer Control, Addresses 88h, Bit-Addressable): The Timer Control SFR is used to configure and modify the way in which the 8051's two timers operate. This SFR controls whether each of the two timers is running or stopped and contains a flag to indicate

that each timer has overflowed. Additionally, some non-timer related bits are located in the TCON SFR. These bits are used to configure the way in which the external interrupts are activated and also contain the external interrupt flags which are set when an external interrupt has occurred.

TMOD (Timer Mode, Addresses 89h): The Timer Mode SFR is used to configure the mode of operation of each of the two timers. Using this SFR your program may configure each timer to be a 16-bit timer, an 8-bit autoreload timer, a 13-bit timer, or two separate timers. Additionally, you may configure the timers to only count when an external pin is activated or to count "events" that are indicated on an external pin.

TL0/TH0 (Timer 0 Low/High, Addresses 8Ah/8Bh): These two SFRs, taken together, represent timer 0. Their exact behavior depends on how the timer is configured in the TMOD SFR; however, these timers always count up. What is configurable is how and when they increment in value.

TL1/TH1 (Timer 1 Low/High, Addresses 8Ch/8Dh): These two SFRs, taken together, represent timer 1. Their exact behavior depends on how the timer is configured in the TMOD SFR; however, these timers always count up. What is configurable is how and when they increment in value.

P1 (Port 1, Address 90h, Bit-Addressable): This is input/output port 1. Each bit of this SFR corresponds to one of the pins on the microcontroller. For example, bit 0 of port 1 is pin P1.0, bit 7 is pin P1.7. Writing a value of 1 to a bit of this SFR will send a high level on the corresponding I/O pin whereas a value of 0 will bring it to a low level.

SCON (Serial Control, Addresses 98h, Bit-Addressable): The Serial Control SFR is used to configure the behavior of the 8051's on-board serial port. This SFR controls the baud rate of the serial port, whether the serial port is activated to receive data, and also contains flags that are set when a byte is successfully sent or received.

SBUF (Serial Control, Addresses 99h): The Serial Buffer SFR is used to send and receive data via the on-board serial port. Any value written to SBUF will be sent out the serial port's TXD pin. Any value which the 8051 receives via the serial port's RXD pin will be delivered to the user program via SBUF. In other words, SBUF serves as the output port when written to and as an input port when read from.

P2 (Port 2, Address A0h, Bit-Addressable): This is input/output port 2. Each bit of this SFR corresponds to one of the pins on the microcontroller. For example, bit 0 of port 2 is pin P2.0, bit 7 is pin P2.7. Writing a value of 1 to a bit of this SFR will send a high level on the corresponding I/O pin whereas a value of 0 will bring it to a low level.

IE (Interrupt Enable, Addresses A8h): The Interrupt Enable SFR is used to enable and disable

specific interrupts. The low 7 bits of the SFR are used to enable/disable the specific interrupts, where as

the highest bit is used to enable or disable ALL interrupts. Thus, if the high bit of IE is 0 all interrupts are

disabled regardless of whether an individual interrupt is enabled by setting a lower bit.

P3 (Port 3, Address B0h, Bit-Addressable): This is input/output port 3. Each bit of this SFR corresponds to one of the pins on the microcontroller. For example, bit 0 of port 3 is pin P3.0, bit 7 is pin P3.7. Writing a value of 1 to a bit of this SFR will send a high level on the corresponding I/O pin whereas a value of 0 will bring it to a low level.

IP (Interrupt Priority, Addresses B8h, Bit-Addressable): The Interrupt Priority SFR is used to specify the relative priority of each interrupt. On the 8051, an interrupt may either be of low (0) priority or high (1) priority. An interrupt may only interrupt interrupts of lower priority. For example, if we configure the 8051 so that all interrupts are of low priority except the serial interrupt, the serial interrupt will always be able to interrupt the system, even if another interrupt is currently executing. However, if a serial interrupt is executing no other interrupt will be able to interrupt the serial interrupt routine since the serial interrupt routine has the highest priority.

PSW (Program Status Word, Addresses D0h, Bit-Addressable): The Program Status Word is used to store a number of important bits that are set and cleared by 8051 instructions. The PSW SFR contains the carry flag, the auxiliary carry flag, the overflow flag, and the parity flag. Additionally, the PSW register contains the register bank select flags which are used to select which of the "R" register banks
are currently selected.

ACC (Accumulator, Addresses E0h, Bit-Addressable): The Accumulator is one of the most used SFRs on the 8051 since it is involved in so many instructions. The Accumulator resides as an SFR at E0h, which means the instruction **MOV A,#20h** is really the same as **MOV E0h,#20h**. first method requires two bytes whereas the second option requires three bytes.

It can hold an 8-bit (1-byte) value and More than half of the 8051's 255 instructions manipulate or use the accumulator in some way.

For example, if you want to add the number 10 and 20, the resulting 30 will be store in the Accumulator. Once you have a value in the Accumulator you may continue processing the value or you may store it in another register or in memory.

B (B Register, Addresses F0h, Bit-Addressable): The "B" register is used in two instructions:

the multiply and divide operations. The B register is also commonly used by programmers as an auxiliary register to temporarily store values. Thus, if you want to quickly and easily multiply or divide A by another number, you may store the other number in "B" and make use of these two instruction Aside from the MUL and DIV instructions, the "B" register is often used as yet another temporary storage register much like a ninth "R" register.

8051 Basic Registers

The "R" registers: The "R" registers are a set of eight registers named R0 to R7. These registers are used as auxiliary registers in many operations. To continue with the above example, perhaps you are adding 10 and 20. The original number 10 may be stored in the Accumulator whereas the value 20 may be stored in, say, register R4. To process the addition you would execute the command: ADD A, R4 After executing this instruction the Accumulator will contain the value 30. The "R" registers are also used to temporarily store values. For example, let's say you want to add the values in R1 and R2 together and then subtract the values of R3 and R4. One way to do this would be:

     MOV A, R3; Move the value of R3 into the accumulator
     ADD A, R4; add the value of R4
     MOV R5, A; Store the resulting value temporarily in R5
     MOV A, R1; Move the value of R1 into the accumulator
     ADD A, R2; Add the value of R2
     SUBB A, R5; Subtract the value of R5 (which now contains R3 + R4)

As you can see, we used R5 to temporarily hold the sum of R3 and R4. Of course, this isn't the most efficient way to calculate (R1+R2) - (R3 +R4) but it does illustrate the use of the "R" registers as a way to store values temporarily.

The Program Counter (PC)

The Program Counter (PC) is a 2-byte address which tells the 8051 where the next instruction to execute is found in memory. When the 8051 is initialized PC always starts at 0000h and is incremented each time an instruction is executed.

PC isn't always incremented by one. some instructions require 2 or 3 bytes the PC will be incremented by 2 or 3 in these cases.

The Program Counter has no way to directly modify its value. But if you execute LJMP 2340h you can make PC=2340h.

You may change the value of PC (by executing a jump instruction, etc.) there is no way to read the value of PC.

4.3 I/O ports and circuits

Each port of 8051 has bidirectional capability. Port 0 is called 'true bidirectional port' as it floats (tristated) when configured as input. Port-1, 2, 3 are called 'quasi bidirectional port'. Port-0 Pin Structure Port -0 has 8 pins (P0.0-P0.7).
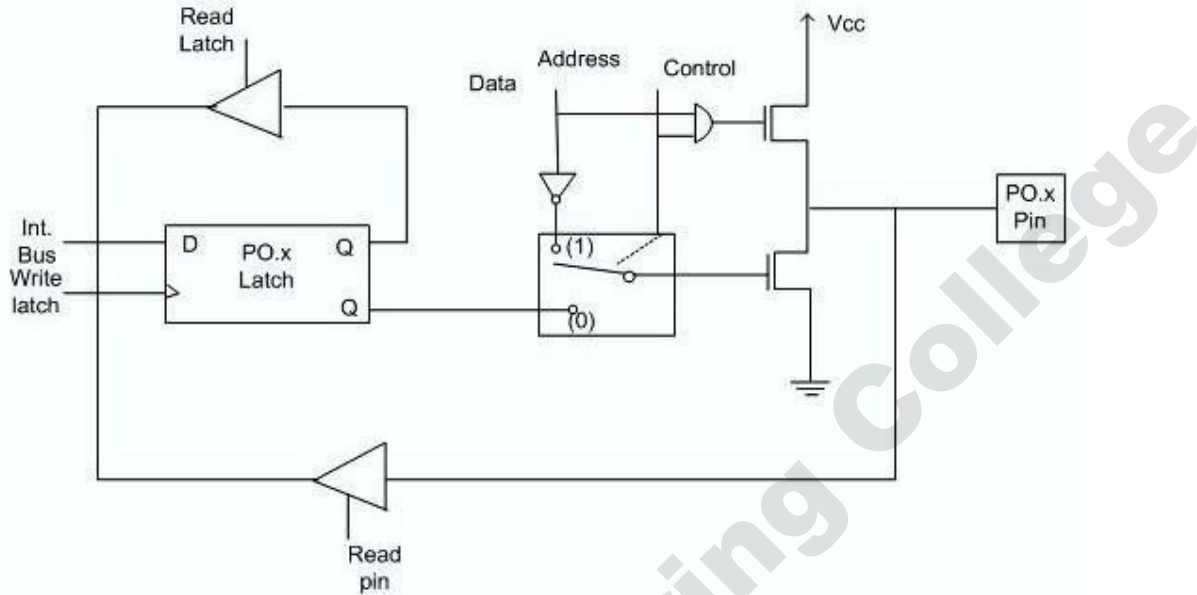

Fig 4.6 Port-0 Structure

Port-0 can be configured as a normal bidirectional I/O port or it can be used for address/data interfacing for accessing external memory. When control is '1', the port is used for address/data interfacing. When the control is '0', the port can be used as a normal bidirectional I/O port.

Let us assume that control is '0'. When the port is used as an input port, '1' is written to the latch. In this situation both the output MOSFETs are 'off'. Hence the output pin floats. This high impedance pin can be pulled up or low by an external source. When the port is used as an output port, a '1' written to the latch again turns 'off' both the output MOSFETs and causes the output pin to float. An external pull-up is required to output a '1'. But when '0' is written to the latch, the pin is pulled down by the lower MOSFET. Hence the output becomes zero.

When the control is '1', address/data bus controls the output driver MOSFETs. If the address/data bus (internal) is '0', the upper MOSFET is 'off' and the lower MOSFET is 'on'. The output becomes '0'. If the address/data bus is '1', the upper transistor is 'on' and the lower transistor is 'off'. Hence the output is '1'. Hence for normal address/data interfacing (for external memory access) no pull-up resistors are required.

Port-0 latch is written to with 1's when used for external memory access.

Port-1 Pin Structure Port-1 has 8 pins (P1.1-P1.7) .The structure of a port-1 pin is shown in fig below.
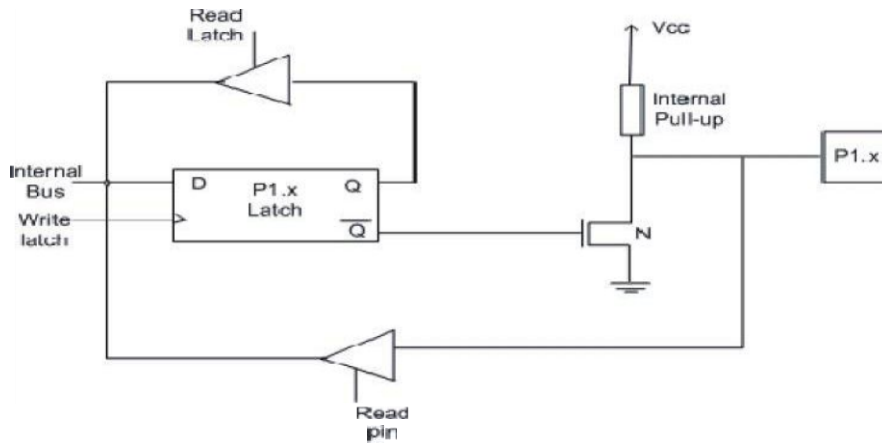
Fig 4.7 Port 1 Structure

Port-1 does not have any alternate function i.e. it is dedicated solely for I/O interfacing. When used as output port, the pin is pulled up or down through internal pull-up. To use port-1 as input port, '1' has to be written to the latch. In this input mode when '1' is written to the pin by the external device then it read fine. But when '0' is written to the pin by the external device then the external source must sink current due to internal pull-up. If the external device is not able to sink the current the pin voltage may rise, leading to a possible wrong reading.

PORT 2 Pin Structure Port-2 has 8-pins (P2.0-P2.7) . The structure of a port-2 pin is shown in figure below:
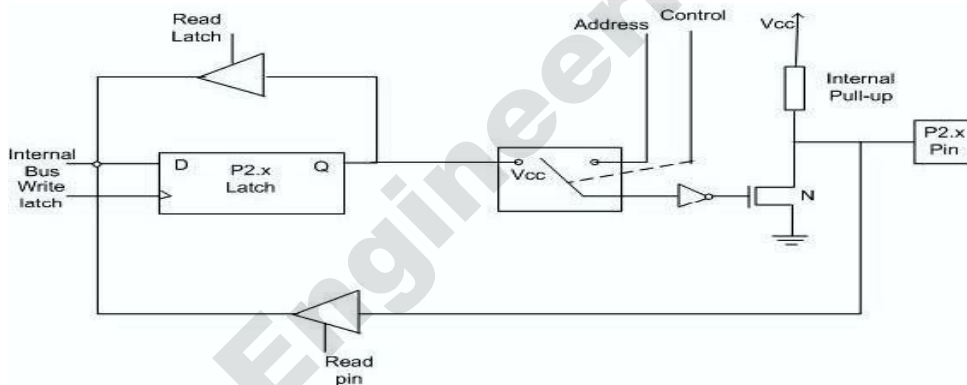
Fig. 4.8 Port 2 structure

Port-2 is used for higher external address byte or a normal input/output port. The I/O operation is similar to Port-1. Port-2 latch remains stable when Port-2 pin are used for external memory access. Here again due to internal pull-up there is limited current driving capability.

✓ PORT 3 Pin Structure

Port-3 has 8 pin (P3.0-P3.7). Port-3 pins have alternate functions. The structure of a port-3 pin is shown in figure

112

Fig. 4.9 Port 3 structure

Each pin of Port-3 can be individually programmed for I/O operation or for alternate function. The alternate function can be activated only if the corresponding latch has been written to '1'. To use the port as input port, '1' should be written to the latch. This port also has internal pull-up and limited current driving capability.

Interfacing External Memory

If external program/data memory are to be interfaced, they are interfaced in the following way.

Fig 4.10 Circuit Diagram for Interfacing of External Memory

External program memory is fetched if either of the following two conditions are satisfied.

1. (Enable Address) is low. The microcontroller by default starts searching for program from external program memory.
2. PC is higher than FFFH for 8051 or 1FFFH for 8052.

PSEN tells the outside world whether the external memory fetched is program memory or data memory.

4.4 8051 Instructions

8051 has about 111 instructions. These can be grouped into the following categories

1. Arithmetic Instructions
2. Logical Instructions
3. Data Transfer instructions
4. Boolean Variable Instructions
5. Program Branching Instructions

The following nomenclatures for register, data, address and variables are used while write instructions.

A: Accumulator

B: "B" register

C: Carry bit

Rn: Register R0 - R7 of the currently selected register bank

Direct: 8-bit internal direct address for data. The data could be in lower 128bytes of RAM (00 - 7FH) or it could be in the special function register (80 - FFH).

@Ri: 8-bit external or internal RAM address available in register R0 or R1. This is used for indirect addressing mode.

#data8: Immediate 8-bit data available in the instruction.

#data16: Immediate 16-bit data available in the instruction.

Addr11: 11-bit destination address for short absolute jump. Used by instructions AJMP & ACALL. Jump range is 2 kbyte (one page).

Addr16: 16-bit destination address for long call or long jump.

Rel: 2's complement 8-bit offset (one - byte) used for short jump (SJMP) and all conditional jumps.

bit: Directly addressed bit in internal RAM or SFR

✓ **Arithmetic Instructions**

Mnemonics	Description	Bytes	Instruction Cycles
ADD A, Rn	A ← A + Rn	1	1
ADD A, direct	A ← A + (direct)	2	1
ADD A, @Ri	A ← A + @Ri	1	1
ADD A, #data	A ← A + data	2	1
ADDC A, Rn	A ← A + Rn + C	1	1
ADDC A, direct	A ← A + (direct) + C	2	1
ADDC A, @Ri	A ← A + @Ri + C	1	1
ADDC A, #data	A ← A + data + C	2	1
DA A	Decimal adjust accumulator	1	1
DIV AB	Divide A by B A ← quotient B ← remainder	1	4
DEC A	A ← A -1	1	1
DEC Rn	Rn ← Rn - 1	1	1
DEC direct	(direct) ← (direct) - 1	2	1
DEC @Ri	@Ri ← @Ri - 1	1	1
INC A	A ← A+1	1	1
INC Rn	Rn ← Rn + 1	1	1
INC direct	(direct) ← (direct) + 1	2	1
INC @Ri	@Ri ← @Ri +1	1	1
INC DPTR	DPTR ← DPTR +1	1	2

114

| MUL AB | Multiply A by B | | |
| | A   low byte (A*B) | 1 | 4 |

	B high byte (A* B)		
SUBB A, Rn	A A - Rn - C	1	1
SUBB A, direct	A A - (direct) - C	2	1
SUBB A, @Ri	A A - @Ri - C	1	1
SUBB A, #data	A A - data - C	2	1

✓ Logical Instructions

Mnemonics	Description	Bytes	Instruction Cycles
ANL A, Rn	A A AND Rn	1	1
ANL A, direct	A A AND (direct)	2	1
ANL A, @Ri	A A AND @Ri	1	1
ANL A, #data	A A AND data	2	1
ANL direct, A	(direct) (direct) AND A	2	1
ANL direct, #data	(direct) (direct) AND data	3	2
CLR A	A 00H	1	1
CPL A	A A	1	1
ORL A, Rn	A A OR Rn	1	1
ORL A, direct	A A OR (direct)	1	1
ORL A, @Ri	A A OR @Ri	2	1
ORL A, #data	A A OR data	1	1
ORL direct, A	(direct) (direct) OR A	2	1
ORL direct, #data	(direct) (direct) OR data	3	2
RL A	Rotate accumulator left	1	1
RLC A	Rotate accumulator left through carry	1	1
RR A	Rotate accumulator right	1	1
RRC A	Rotate accumulator right through carry	1	1
SWAP A	Swap nibbles within Acumulator	1	1
XRL A, Rn	A A EXOR Rn	1	1
XRL A, direct	A A EXOR (direct)	1	1
XRL A, @Ri	A A EXOR @Ri	2	1
XRL A, #data	A A EXOR data	1	1
XRL direct, A	(direct) (direct) EXOR A	2	1
XRL direct, #data	(direct) (direct) EXOR data	3	2

Data Transfer Instructions

Mnemonics	Description	Bytes	Instruction Cycles
MOV A, Rn	A Rn	1	1
MOV A, direct	A (direct)	2	1
MOV A, @Ri	A @Ri	1	1
MOV A, #data	A data	2	1
MOV Rn, A	Rn A	1	1
MOV Rn, direct	Rn (direct)	2	2
MOV Rn, #data	Rn data	2	1
MOV direct, A	(direct) A	2	1
MOV direct, Rn	(direct) Rn	2	2
MOV direct1, direct2	(direct1) (direct2)	3	2

MOV direct, @Ri	(direct) @Ri	2	2
MOV direct, #data	(direct) #data	3	2
MOV @Ri, A	@Ri A	1	1
MOV @Ri, direct	@Ri (direct)	2	2
MOV @Ri, #data	@Ri data	2	1
MOV DPTR, #data16	DPTR data16	3	2
MOVC A, @A+DPTR	A Code byte pointed by A + DPTR	1	2
MOVC A, @A+PC	A Code byte pointed by A + PC	1	2
MOVC A, @Ri	A Code byte pointed by Ri 8-bit address)	1	2
MOVX A, @DPTR	A External data pointed by DPTR	1	2
MOVX @Ri, A	@Ri A (External data - 8bit address)	1	2
MOVX @DPTR, A PUSH direct	(SP) (direct)	2	2
@DPTR	A(External data - 16bit address)	1	2

POP direct	(direct) ← (SP)	2	2
XCH Rn	Exchange A with Rn	1	1
XCH direct	Exchange A with direct byte	2	1
XCH @Ri	Exchange A with indirect RAM	1	1
XCHD A, @Ri	Exchange least significant nibble of A with that of indirect RAM	1	1

Boolean Variable Instructions

Mnemonics	Description	Bytes	Instruction Cycles
CLR C	C-bit 0	1	1
CLR bit	bit 0	2	1
SET C	C 1	1	1
SET bit	bit 1	2	1
CPL C	C	1	1
CPL bit	bit	2	1
ANL C, /bit	C C .	2	1
ANL C, bit	C C. bit	2	1
ORL C, /bit	C C +	2	1
ORL C, bit	C C + bit	2	1
MOV C, bit	C bit	2	1
MOV bit, C	bit C	2	2

Program Branching Instructions

Mnemonics	Description	Bytes	Instruction Cycles
ACALL addr11	PC + 2 (SP); addr 11 PC	2	2
AJMP addr11	Addr11 PC	2	2
CJNE A, direct, rel	Compare with A, jump (PC + rel) if not equal	3	2
CJNE A, #data, rel	Compare with A, jump (PC + rel) if not equal	3	2
CJNE Rn, #data, rel	Compare with Rn, jump (PC + rel) if not equal	3	2
CJNE @Ri, #data, rel	Compare with @Ri A, jump (PC + rel) if not equal	3	2
DJNZ Rn, rel	Decrement Rn, jump if not zero	2	2
DJNZ direct, rel	Decrement (direct), jump if not zero	3	2
JC rel	Jump (PC + rel) if C bit = 1	2	2
JNC rel	Jump (PC + rel) if C bit = 0	2	2
JB bit, rel	Jump (PC + rel) if bit = 1	3	2
JNB bit, rel	Jump (PC + rel) if bit = 0	3	2
JBC bit, rel	Jump (PC + rel) if bit = 1	3	2
JMP @A+DPTR	A+DPTR PC	1	2
JZ rel	If A=0, jump to PC + rel	2	2
JNZ rel	If $A \neq 0$, jump to PC + rel	2	2
LCALL addr16	PC + 3 (SP), addr16 PC	3	2
LJMP addr 16	Addr16 PC	3	2
NOP	No operation	1	1
RET	(SP) PC	1	2
RETI	(SP) PC, Enable Interrupt	1	2
SJMP rel	PC + 2 + rel PC	2	2
JMP @A+DPTR	A+DPTR PC	1	2
JZ rel	If A = 0. jump PC+ rel	2	2
JNZ rel	If $A \neq 0$, jump PC + rel	2	2
NOP	No operation	1	1

4.5 8051 Addressing Modes

8051 has four addressing modes.

1. Immediate Addressing:

Data is immediately available in the
instruction. For example -
ADD A, #77; Adds 77 (decimal) to A and stores in A
ADD A, #4DH; Adds 4D (hexadecimal) to A and stores in A MOV DPTR, #1000H;

Moves 1000 (hexadecimal) to data pointer

2. Bank Addressing or Register Addressing:

This way of addressing accesses the bytes in the current register bank. Data is available in the register specified in the instruction. The register bank is decided by 2 bits of Processor Status

116

Word (PSW). For example-

ADD A, R0; Adds content of R0 to A and stores in A

3. Direct Addressing:

The address of the data is available in the instruction. For example -

MOV A, 088H; Moves content of SFR TCON (address 088H)to A

4. Register Indirect Addressing:

The address of data is available in the R0 or R1 registers as specified in the instruction.

For example - MOV A, @R0 moves content of address pointed by R0 to A .

5. External Data Addressing:

Pointer used for external data addressing can be either R0/R1 (256 byte access) or DPTR (64kbyte access).

For example -

MOVX A, @R0; Moves content of 8-bit address pointed by R0 to A

MOVX A, @DPTR; Moves content of 16-bit address pointed by DPTR to A

6. External Code Addressing:

Sometimes we may want to store non-volatile data into the ROM e.g. look-up tables. Such data may require reading the code memory. This may be done as follows -

MOVC A, @A+DPTR; Moves content of address pointed by A+DPTR to A

MOVC A, @A+PC; Moves content of address pointed by A+PC to A

4.6 Assembly language Programming

Character transmission using a time delay:

A program shown below takes the character in 'A' register, transmits it, delays for transmission time, and returns to the calling program. Timer-1 is used to set the baud rate, which is 1200 baud in this program

The delay for one character transmission (in Mode 1 i.e.10 bits) is

$10/2400 = 0.00833$ seconds

Or, 8.33 milliseconds

Hence software delay of 10ms is used.

Timer-1 generates a baud rate close to 1200. Using a 12MHz crystal, the reload value is

$$256 - \frac{12 \times 10^6}{32 \times 12 \times 2400} = 229.958$$

Or, 230 i.e. E6H .

This gives rise to an actual baud rate of 1202. SMOD is programmed to be 0.

Assembly language Program is as follows

Assembly language Program is as follows

```
RELOAD      EQU 0E6H    ; defining constant for reload value for baudrate generation
DELAY       EQU 0A6H    ; defining constant for 1 millisecond
DLYLSB      EQU 0AH     ; defining constant for 10 millisecond
DLYMSB      EQU 00H     ; defining constant

ORG 0C00H               ; Org directive
ANL PCON, #7FH          ; SMOD = 0
ANL TMOD, #0FH          ; Alter only Timer-1
ORL TMOD, #20H          ; program Timer-1 in mode-2
MOV TH1,  #RELOAD       ; program reload value to TH1
SETB TR1                ; enable Timer-1 run bit (start timer)
MOV SCON, #40H          ; Serial port in mode-1 (receive not enabled)


TRMIT: MOV SBUF, #'A'   ; send the ASCII value of 'A'
       ACALL TRMITTIME  ; wait for DELAY
       SJMP TRMIT       ; again transmit
```

         ; Code to wait for the transmission to complete

The subroutine TRMITTIME generates a delay of about 10ms. With a clock of 12MHz, one instruction cycle time is

$$\frac{1}{12 \times 10^6} \times 12 = 1 \times 10^{-6}$$

The loop "MILSEC" generates a delay of about 1×10^{-3} sec. This gets executed 10 times for a total delay of 10×10^{-3} sec or 10ms

```
TRITTIME:       MOV A, #DLYLSB
                MOV B, #DLYMSB
SOFTIME:        ACALL SOFTIME
                RET
                PUSH 07H
                PUSH A
                ORL A, B
                CJNE A, #C0H, OK
                POP A
                SJMP DONE

TIMER:          MOV R7, #DELAY   ; Generate delay for 1 millisecond
MILSEC:         NOP
                NOP
                NOP
                NOP
                DJNZ R7, MILSEC
                NOP
                NOP
                DEC A
                CJNE A, #0FFH, NO ROLL
                DEC B


NO ROLL:        CJNE A, #C0H, TIMER
DONE:           POP 07H
END             RET
```

UNIT V
INTERFACING MICROCONTROLLER

5.1 Programming 8051 Timers: Using Timers to Measure Time
One of the primary uses of timers is to measure time.
When a timer is in interval timer mode (as opposed to event counter mode) and correctly configured, it will increment by 1 every machine cycle. A single machine cycle consists of 12 crystal pulses. Thus a running timer will be incremented:11,059,000 / 12 = 921,583 times per second.

Unlike instructions which require 1 machine cycle, others 2, and others 4--the timers are consistent: They will always be incremented once per machine cycle. Thus if a timer has counted from 0 to 50,000 you may calculate:

50,000 / 921,583 = .0542.0542 seconds have passed. To execute an event once per second you'd have to wait for the timer to count from 0 to 50,000 18.45times.

To calculate how many times the timer will be incremented in .05 seconds, a simple multiplication can be done: 0 .05 * 921,583 = 46,079.15.

This tells us that it will take .05 seconds (1/20th of a second) to count from 0 to 46.0.

To work with timers is to control the timers and initialize them.

The TMOD SFR

TMOD (Timer Mode): The TMOD SFR is used to control the mode of operation of both timers. Each bit of the SFR gives the microcontroller specific information concerning how to run a timer. The high four bits (bits 4 through 7) relate to Timer 1whereas the low four bits (bits 0 through 3) perform the exact same functions, but for timer 0. The modes of operation are:

Table 5.1 modes of Timer

TxM1	TxM0	Timer Mode	Description of Mode
0	0	0	13-bit Timer.
0	1	1	16-bit Timer
1	0	2	8-bit auto-reload
1	1	3	13-bit Time Mode (mode 0)

Timer mode "0" is a 13-bit timer. When the timer is in 13-bit mode, TLx will count from 0 to 31. When TLx is incremented from 31, it will "reset" to 0 and increment THx. Thus, effectively, only 13 bits of the two timer bytes are being used: bits 0-4 of TLx and bits 0-7 of THx. The timer can only contain 8192 values. If you set a 13-bit timer to 0, it will overflow back to zero 8192 machine cycles later.

16-bit Time Mode (mode 1)

Timer mode "1" is a 16-bit timer. TLx is incremented from 0 to 255. When TLx is incremented from 255, it resets to 0 and causes THx to be incremented by 1. Since this is a full 16-bit timer, the timer may contain up to 65536 distinct values. If you set a 16-bit timer to 0, it will overflow back to 0 after 65,536 machine cycles.

8-bit Time Mode (mode 2)

Timer mode "2" is an 8-bit auto-reload mode.

When a timer is in mode 2, THx holds the "reload value" and TLx is the timer itself. Thus, TLx starts counting up. When TLx reaches 255 and is subsequently incremented, instead of resetting to 0 (as in the case of modes 0 and 1), it will be reset to the value stored in THx. For example, if TH0 holds the value FDh and TL0 holds the value FEh values of TH0 and TL0 for a few machine cycles:

The value of TH0 never changed. When we use mode 2 you almost always set THx to a known value and TLxis the SFR that is constantly incremented. The benefit of auto-reload mode is the timer always have a value from 200 to 255. If you use mode 0 or 1, you'd have to check in code to see if the timer had overflowed and, if so, reset the timer to 200. This takes precious instructions of execution time to check the value and/or to reload it. When

Table 5.2 mode 2 operation

Machine Cycle	TH0 Value	TL0 Value
1	FDh	FEh
2	FDh	FFh
3	FDh	FDh
4	FDh	FEh
5	FDh	FFh
6	FDh	FDh
7	FDh	FEh

you use mode 2 the microcontroller takes care of this. Auto-reload mode is very commonly used for establishing a baud rate in Serial Communications.

Split Timer Mode (mode 3)

Timer mode "3" is a split-timer mode. When Timer 0 is placed in mode 3, it essentially becomes two separate 8-bit timers. Timer 0 is TL0 and Timer 1 is TH0. Both timers count from 0 to 255 and overflow back to 0. All the bits that are related to Timer 1 will now be tied to TH0. While Timer 0 is in split mode, the real Timer 1 (i.e. TH1 and TL1) can be put into modes 0, 1 or 2 normally--however, you may not start or stop the real timer 1 since the bits that do that are now linked to TH0. The real timer 1,e, will be incremented every machine cycle always. The only real use in split timer mode is if you need to have two separate timers and, additionally, a baud rate generator you can use the real Timer 1 as a baud rate generator and use TH0/TL0 as two separate timers.

Reading the Timer

There are two common ways of reading the value of a 16-bit timer; which you use depends on your specific application. You may either read the actual value of the timer as a 16-bit number, or you may simply detect when the timer has overflowed.

Reading the value of a Timer

If timer is in an 8-bit mode either 8-bit Auto Reload mode or in split timer mode, you simply read the 1-byte value of the timer. With a 13-bit or16-bit timer the timer value was 14/255 (High byte 14, low byte 255) but you read 15/255.Because you read the low byte as 255. But when you executed the next instruction a small amount of time passed--but enough for the timer to increment again at which time the value rolled over from 14/255 to 15/0. But in the process you've read the timer as being 15/255.

You read the high byte of the timer, then read the low byte, then read the high byte again. If the high byte read the second time is not the same as the high byte read the first time you repeat the cycle. In code, this would appear as:

REPEAT: MOV A, TH0

MOV R0, TL0

CJNE A, TH0, REPEAT

In this case, we load the accumulator with the high byte of Timer 0. We then load R0with the low byte of Timer 0. Finally, we check to see if the high byte we read out of Timer0-- which is now stored in the Accumulator--is the same as the current Timer 0 high byte. We do by going back to REPEAT. When the loop exits we will have the low byte of the timer in R0 and the high byte in the Accumulator.Another much simpler alternative is to simply turn off the timer run bit (i.e. CLR TR0), read the timer value, and then turn on the timer run bit (i.e. SETB TR0).

Detecting Timer Overflow

Whenever a timer *overflows* from its highest value back to 0, the microcontroller automatically sets the TFx bit in the TCON register. if TF1 is set it means that timer 1 has overflowed.

We can use this approach to cause the program to execute a fixed delay. it takes the 8051 1/20thof a second to count from 0 to 46,079. However, the TFx flag is set when the timer overflows back to 0. Thus, if we want to use the TFx flag to indicate when 1/20th of a second has passed we must set the timer initially to 65536 less 46079, or 19,457. If we set the timer to 19,457, 1/20th of a second later the timer will overflow.

The following code to execute a pause of 1/20th of a second:

MOV TH0,#76;High byte of 19,457 (76 * 256 = 19,456)
MOV TL0,#01;Low byte of 19,457 (19,456 + 1 = 19,457)
MOV TMOD,#01;Put Timer 0 in 16-bit mode
SETB TR0; Make Timer 0 start counting
JNB TF0,$;If TF0 is not set, jump back to this same instruction

In the above code the first two lines initialize the Timer 0 starting value to 19,457. The next two instructions configure timer 0 and turn it on. Finally, the last instruction **JNB TF0, $**, reads "Jump, if TF0 is not set, back to this same instruction." The "$" operand means, in most assemblers, the address of the current instruction. Thus as long as the timer has not overflowed and the TF0 bit has not been set the program will keep executing this same instruction. After 1/20th of a second timer 0 will overflow, set the TF0 bit, and program execution will then break out of the loop.

5.2 Serial Port Programming: 8051 Serial Communication

One of the 8051's many powerful features -integrated *UART*, known as a serial port to easily read and write values to the serial port instead of turning on and off one of the I/O lines in rapid succession to properly "clock out" each individual bit, including start bits, stop bits and parity bits.

✓ **Setting the Serial Port Mode** configures it by specifying 8051 how many data bits we want, the baud rate we will be using and how the baud rate will be determined. First, let's present the "Serial Control" (SCON) SFR and define what each bit of the SFR represents:

Table 5.3 Definition of SCON SFR

Bit	Name	Bit Address	Explanation of Function
7	SM0	9Fh	Serial port mode bit 0
6	SM1	9Eh	Serial port mode bit 1.
5	SM2	9Dh	Mutli processor Communications Enable
4	REN	9Ch	Receiver Enable. This bit must be set in order to receive Characters.
3	TB8	9Bh	Transmit bit 8. The 9th bit to transmit in mode 2 and 3.
2	RB8	9AH	Receive bit 8. The 9th bit received in mode 2 and 3.
1	T1	99h	Transmit Flag. Set when a byte has been completely Transmitted.
0	RI	98h	Receive Flag. Set when a byte has been completely Received.

Additionally, it is necessary to define the function of SM0 and SM1 by an additional table:
Table 5.4 SCON as serial Port

Table 5.4 Modes of SCON

SM0	SM1	Serial Mode	Explanation Baud Rate
0	0	0	0 8-bit Shift Register Oscillator / 12
0	1	1	8-bit UART Set by Timer 1 (*)
1	0	2	9-bit UART Oscillator / 32 (*)
1	1	3	9-bit UART Set by Timer 1 (*)

The SCON SFR allows us to configure the Serial Port. The first four bits (bits 4 through 7) are configuration bits:

Bits **SM0** and **SM1** is to set the *serial mode* to a value between 0 and 3, inclusive as in table above selecting the Serial Mode selects the mode of operation (8-bit/9-bit, UART or Shift Register) and also determines how the baud rate will be calculated. In modes 0 and 2 the baud rate is fixed based on the oscillator's frequency. In modes 1 and 3 the baud rate is variable based on how often Timer 1 overflows.

The next bit, **SM2**, is a flag for " Multiprocessor communication whenever a byte has been received the 8051 will set the "RI" (Receive Interrupt) flag to let the program know that a byte has been received and that it needs to be processed.

However, when SM2 is set the "RI" flag will only be triggered if the 9th bit received was a "1". if SM2 is set and a byte is received whose 9th bit is clear, the RI flag will never be set .You will almost always want to clear this bit so that the flag is set upon reception of *any* character.

The next bit, **REN**, is "Receiver Enable." is set indicate to data received via the serial port.

The last four bits (bits 0 through 3) are operational bits. They are used when actually sending and receiving data--they are not used to configure the serial port.

The **TB8** bit is used in modes 2 and 3. In modes 2 and 3, a total of nine data bits are transmitted. The first 8 data bits are the 8 bits of the main value, and the ninth bit is taken from TB8. If TB8 is set and a value is written to the serial port, the data's bits will be written to the serial line followed by a "set" ninth bit. If TB8 is clear the ninth bit will be "clear."

The **RB8** also operates in modes 2 and 3and functions essentially the same way as TB8, but on the reception side. When a byte is received in modes 2 or 3, a total of nine bits are received. In this case, the first eight bits received are the data of the serial byte received and the value of the nineth bit received will be placed in RB8.**TI** means "Transmit Interrupt."

When a program writes a value to the serial port, a certain amount of time will pass before the individual bits of the byte are "clocked out" the serial port. If the program were to write another byte to the serial port before the first byte was completely output, the data being sent would be garbled. Thus, the8051 lets the program know that it has "clocked out" the last byte by setting the TI bit.

When the TI bit is set, the program may assume that the serial port is "free" and ready to send the next byte. Finally, the **RI** bit means "Receive Interrupt." It functions similarly to the "TI" bit, but it indicates that a byte has been received. Whenever the 8051 has received a complete byte it will trigger the RI bit to let the program know that it needs to read the value quickly, before another byte is read.

✓ **Setting the Serial Port Baud Rate**

Once the Serial Port Mode has been configured, the program must configure the serial port's baud rate. This only applies to Serial Port modes 1 and 3. The Baud Rate is determined based on the oscillator's frequency when in mode 0 and 2. In mode 0, the baud rate is

always the oscillator frequency divided by 12. This means if you're crystal is 1.059 Mhz, mode 0 baud rate will always be 921,583 baud. In mode 2 the baud rate is always the oscillator frequency divided by 64, so a 11.059Mhz crystal speed will yield a baud rate of172,797.

In modes 1 and 3, the baud rate is determined by how frequently timer 1 overflows. The more frequently timer 1 overflows, the higher the baud rate. There are many ways one can cause timer 1 to overflow at a rate that determines a baud rate, but the most common method is to put timer 1 in 8-bit auto-reload mode (timer mode2) and set a reload value (TH1) that causes Timer 1 to overflow at a frequency appropriate to generate a baud rate.

To determine the value that must be placed in TH1 to generate a given baud rate, (assuming PCON.7 is clear).

$$TH1 = 256 - ((Crystal / 384) / Baud)$$

If PCON.7 is set then the baud rate is effectively doubled, thus the equation becomes:

$$TH1 = 256 - ((Crystal / 192) / Baud)$$

For example, if we have an 11.059 Mhz crystal and we want to configure the serial port to

19,200 baud we try plugging it in the first equation:

$$TH1 = 256 - ((Crystal / 384) / Baud)$$
$$TH1 = 256 - ((11059000 / 384) / 19200)$$
$$TH1 = 256 - ((28,799) / 19200)$$
$$TH1 = 256 - 1.5 = 254.5$$

To obtain 19,200 baud on a 11.059Mhz crystal we'd have to set TH1 to 254.5. If we set it to 254 we will have achieved 14,400 baud and if we set it to 255 we will have achieved 28,800 baud.

To achieve 19,200 baud we simply need to set PCON.7 (SMOD). When we do this we double the baud rate and utilize the second equation mentioned above. Thus we have:

$$TH1 = 256 - ((Crystal / 192) / Baud)$$
$$TH1 = 256 - ((11059000 / 192) / 19200)$$
$$TH1 = 256 - ((57699) / 19200)$$
$$TH1 = 256 - 3 = 253$$

Therefore, to obtain 19,200 baud with an 11.059MHz crystal we must:

1) Configure Serial Port mode 1 or 3.
2) Configure Timer 1 to timer mode 2 (8-bit auto reload).
3) Set TH1 to 253 to reflect the correct frequency for 19,200 baud.
4) Set PCON.7 (SMOD) to double the baud rate.

✓ **Writing to the Serial Port**

Once the Serial Port has been properly configured as explained above, the serial port is ready to be used to send data and receive data.

To write a byte to the serial write the value to the **SBUF** (99h) SFR. For example, if you wanted to send the letter "A" to the serial port, it could be accomplished as easily as:

MOV SBUF, #'A'

Upon execution of the above instruction the 8051 will begin transmitting the character via the serial port. Obviously transmission is not instantaneous--it takes a measureable amount of time to transmit. And since the 8051 does not have a serial output buffer we need to be sure that a character is completely transmitted before we try to transmit the next character.

The 8051 lets us know when it is done transmitting a character by setting the **TI** bit in SCON. When this bit is set the last character has been transmitted and that send the next character, if any. Consider the following code segment:

CLR TI; Be sure the bit is initially clear

MOV SBUF, #'A'; Send the letter 'A' to the serial port

JNB TI,$;Pause until the RI bit is set.

The above three instructions will successfully transmit a character and wait for the TI bit to be set before continuing. The last instruction says "Jump if the TI bit is not set to $"—

$, in most assemblers, means "the same address of the current instruction." Thus the 8051 will pause on the JNB instruction until the TI bit is set by the 8051 upon successful transmission of the character.

✓ **Reading the Serial Port**

Reading data received by the serial port is equally easy. To read a byte from the serial port one just needs to read the value stored in the **SBUF** (99h) SFR after the 8051 has automatically set the **RI** flag in SCON.

For example, if your program wants to wait for a character to be received and subsequently read it into the Accumulator, the following code segment may be used:

        JNB RI,$;Wait for the 8051 to set the RI flag

        MOV A,SBUF; Read the character from the serial port

The first line of the above code segment waits for the 8051 to set the RI flag; again, the8051 sets the RI flag automatically when it receives a character via the serial port. So as long as the bit is not set the program repeats the "JNB" instruction continuously. Once the RI bit is set upon character reception the above condition automatically fails and program flow falls through to the "MOV" instruction which reads the value.

5.3 Interrupt Programming:

What Events Can Trigger interrupts, and where do they go?

The following events will cause an interrupt:

- · Timer 0 Overflow.
- · Timer 1 Overflow.
- · Reception/Transmission of Serial Character.
- · External Event 0.
- · External Event 1.

To distinguish between various interrupts and executing different code depending on what interrupt was triggered 8051 may be jumping to a fixed address when a given interrupt occurs.

Table 5.5 Interrupt handling

Interrupt	Flag	Interrupt Handler Address
External 0	IE0	0003h
Timer 0	TF0	000Bh
External 1	IE1	0013h
Timer 1	TF1	001Bh
Serial	RI/TI	0023h

If Timer 0 overflows (i.e., the TF0 bit is set), the main program will be temporarily suspended and control will jump to 000BH if we have code at address 0003H that handles the situation of Timer 0 overflowing.

✓ **Setting Up Interrupts**

By default at power up, all interrupts are disabled. Even if, for example, the TF0 bit is set, the 8051 will not execute the interrupt. Your program must specifically tell the

8051 that it wishes to enable interrupts and specifically which interrupts it wishes to enable. Your program may enable and disable interrupts by modifying the IE SFR (A8h):

Table 5.6 Interrrupt and address

Bit	Name Bit	Address	Explanation of Function
7	EA	AFh	Global Interrupt Enable/Disable
6		AEh	Undefined
5		ADh	Undefined
4	ES	ACh	Enable Serial Interrupt
3	ET1	ABh	Enable Timer 1 Interrupt
2	EX1	AAh	Enable External 1 Interrupt
1	ET0	A9h	Enable Timer 0 Interrupt
0	EX0	A8h	Enable External 0 Interrupt

Each of the 8051'sinterrupts has its own bit in the IE SFR. You enable a given interrupt by setting the corresponding bit. For example, if you wish to enable Timer 1 Interrupt, you would execute either:

        MOV IE,#08h || SETB ET1

Both of the above instructions set bit 3 of IE, thus enabling Timer 1 Interrupt. Once Timer 1 Interrupt is enabled, whenever the TF1 bit is set, the 8051 will automatically put "on hold" the main program and execute the Timer 1 Interrupt Handler at address 001Bh. However, before Timer 1 Interrupt (or any other interrupt) is truly enabled, you must also set bit 7 of IE.

      Bit 7, the Global Interrupt Enable/Disable, enables or disables all interrupts simultaneously. That is to say, if bit 7 is cleared then no interrupts will occur, even if all the other bits of IE are set. Setting bit 7 will enable all the interrupts that have been selected by setting other bits in IE. This is useful in program execution if you have time-critical code that needs to execute. In this case, you may need the code to execute from start to finish without any interrupt getting in the way. To accomplish this you can simply clear bit 7 of IE (CLR EA) and then set it after your time critical code is done.

To enable the Timer 1 Interrupt execute the following two instructions:

          SETB ET1

          SETB EA

Thereafter, the Timer 1 Interrupt Handler at 01Bh will automatically be called whenever the TF1 bit is set (upon Timer 1 overflow).

✓ Polling Sequence

The 8051 automatically evaluates whether an interrupt should occur after every instruction. When checking for interrupt conditions, it checks them in the following order:

        1) External 0 Interrupt

        2) Timer 0 Interrupt

        3) External 1 Interrupt

        4) Timer 1 Interrupt

        5) Serial Interrupt

✓ Interrupt Priorities

The 8051 offers two levels of interrupt priority: high and low. By using interrupt priorities you may assign higher priority to certain interrupt conditions. For example, you may have enabled Timer 1 Interrupt which is automatically called every time Timer 1 overflows. Additionally, you may have enabled the Serial Interrupt which is called every time a character is received via the serial port. However, you may consider that receiving a character is much more important than the timer interrupt. In this case, if Timer 1 Interrupt is already executing you may wish that the serial interrupt itself interrupts the Timer 1 Interrupt. When the serial interrupt is complete, control passes back to Timer 1 Interrupt and

finally back to the main program. You may accomplish this by assigning a high priority to the Serial Interrupt and a low priority to the Timer 1 Interrupt.

Interrupt priorities are controlled by the **IP**SFR (B8h). The IP SFR has the following format:

Bit Name Bit Address Explanation of Function

Bit	Name	Bit Address	Explanation of Function
7			Undefined
6			Undefined
5			Undefined
4	PS	BCh	Serial Interrupt Priority
3	PT1	BBh	Timer 1 Interrupt Priority
2	PX1	BAh	External 1 Interrupt Priority
1	PT0	B9h	Timer 0 Interrupt Priority
0	PX0	B8h	External 0 Interrupt Priority

When considering interrupt priorities, the following rules apply:

✓ Nothing can interrupt a high-priority interrupt--not even another high priority interrupt.

✓ A high-priority interrupt may interrupt a low priority interrupt.

✓ A low-priority interrupt may only occur if no other interrupt is already executing.

✓ If two interrupts occur at the same time, the interrupt with higher priority will execute first. If both interrupts are of the same priority the interrupt which is serviced first by polling sequence will be executed first.

What Happens When an Interrupt Occurs?

When an interrupt is triggered, the following actions are taken automatically by the microcontroller:

· The current Program Counter is saved on the stack, low-byte first.

· Interrupts of the same and lower priority are blocked.

· In the case of Timer and External interrupts, the corresponding interrupt flag is set.

· Program execution transfers to the corresponding interrupt handler vector address.

· The Interrupt Handler Routine executes. Take special note of the third step: If theinterrupt being handled is a Timer or External interrupt, the microcontroller automatically clearsthe interrupt flag before passing control to your interrupt handler routine.

✓ **What Happens When an Interrupt Ends?**

An interrupt ends when your program executes the RETI instruction. When the RETI instruction is executed the following actions are taken by the microcontroller:

· Two bytes are popped off the stack into the Program Counter to restore normal program execution.

· Interrupt status is restored to its pre-interrupt status.

· **Serial Interrupts**

Serial Interrupts are slightly different than the rest of the interrupts. This is due to the fact that there are two interrupt flags: RI and TI. If either flag is set, a serial interrupt is triggered. As you will recall from the section on the serial port, the RI bit is set when a byte is received by the serial port and the TI bit is set when a byte has been sent. This means that when your serial interrupt is executed, it may have been triggered because the RI flag was set or because the TI flag was set--or because both flags were set. Thus, your routine must check the status of these flags to determine what action is appropriate. Also, since the 8051does not automatically clear the RI and TI flags you must clear these bits in your interrupt handler.

```
INT_SERIAL: JNB RI, CHECK_TI; If the RI flag is not set, we jump to check TI
MOV A, SBUF; If we got to this line, it's because the RI bit *was* set
CLR RI; Clear the RI bit after we've processed it
CHECK_TI: JNB TI, EXIT_INT; If the TI flag is not set, we jump to the exit point
CLR TI; Clear the TI bit before we send another character
```

126

MOV SBUF, #'A'; Send another character to the serial port
EXIT_INT: RETI

As you can see, our code checks the status of both interrupts flags. If both flags were set, both sections of code will be executed. Also note that each section of code clears its corresponding interrupt flag. If you forget to clear the interrupt bits, the serial interrupt will be executed over and over until you clear the bit. Thus it is very important that you always clear the interrupt flags in a serial interrupt.

✓ **Important Interrupt Consideration: Register Protection**

One very important rule applies to all interrupt handlers: Interrupts must leave the processor in the same state as it was in when the interrupt initiated. Remember, the idea behind interrupts is that the main program isn't aware that they are executing in the "background." However, consider the following code:

CLR C; Clear carry

MOV A, #25h; Load the accumulator with 25h

ADDC A, #10h; Add 10h, with carry

After the above three instructions are executed, the accumulator will contain a value of35h.

But what would happen if right after the MOV instruction an interrupt occurred. During this interrupt, the carry bit was set and the value of the accumulator was changed to 40h. When the interrupt finished and control was passed back to the main program, the ADDC would add 10h to40h, and additionally add an additional 1h because the carry bit is set. In this case, the accumulator will contain the value 51h at the end of execution. In this case, the main program has seemingly calculated the wrong answer. How can25h + 10h yield 51h as a result? It doesn't make sense. A programmer that was unfamiliar with interrupts would be convinced that the microcontroller was damaged in some way, provoking problems with mathematical calculations.

What has happened, in reality, is the interrupt did not protect the registers it used.

To insure that the value of the accumulator is the same at the end of the interrupt as it was at the beginning. This is generally accomplished with a PUSH and POP sequence. For example:

PUSH ACC
PUSH PSW
MOV A,#0FFh
ADD A,#02h
POP PSW
POP ACC

The *guts* of the interrupt is the MOV instruction and the ADD instruction. However, these two instructions modify the Accumulator (the MOV instruction) and also modify the value of the carry bit (the ADD instruction will cause the carry bit to be set). Since an interrupt routine must guarantee that the registers remain unchanged by the routine, the routine pushes the original values onto the stack using the PUSH instruction. It is then free to use the registers it protected to its heart's content. Once the interrupt has finished its task, it pops the original values back into the registers. When the interrupt exits, the main program will never know the difference because the registers are exactly the same as they were before the interrupt executed.

In general, your interrupt routine must protect the following registers:

·   PSW
·   DPTR (DPH/DPL)
·   PSW
·   ACC
·   B
·   Registers R0-R7

PSW consists of many individual bits that are set by various 8051instructions. A*lways* protect PSW by pushing and popping it off the stack at the beginning and end of your interrupts. It will not be allow to execute the instruction: PUSH R0

Because depending on which register bank is selected, R0 may refer to either internal ram address 00h, 08h, 10h, or 18h.R0, in and of itself, is not a valid memory address that the PUSH and POP instructions can use. Thus, if you are using any "R" register in your interrupt routine, you will have to push that register's absolute address onto the stack instead of just saying **PUSH R0**. For example, instead of PUSH R0 you would execute: PUSH 00h

5.4 Interfacing a Microprocessor to Keyboard

When you press a key on your computer, you are activating a switch. There are many different ways of making these switches. An overview of the construction and operation of some of the most common types.

✓ *Mechanical key switches:* In mechanical-switch keys, two pieces of metal are pushed together when you press the key. The actual switch elements are often made of a phosphor-bronze alloy with gold platting on the contact areas. The key switch usually contains a spring to return the key to the nonpressed position and perhaps a small piece of foam to help damp out bouncing.

Some mechanical key switches now consist of a molded silicon dome with a small piece of conductive rubber foam short two trace on the printed-circuit board to produce the key pressed signal.

Mechanical switches are relatively inexpensive but they have several disadvantages. First, they suffer from contact bounce. A pressed key may make and break contact several times before it makes solid contact.

Second, the contacts may become oxidized or dirty with age so they no longer make a dependable connection.

Higher- quality mechanical switches typically have a rated life time of about 1 illion keystrokes. The silicone dome type typically last 25 million keystrokes.

✓ *Membrane key switches:* These switches are really a special type of mechanical switches. They consist of a three-layer plastic or rubber sandwich.

The top layer has a conductive line of silver ink running under each key position. The bottom layer has a conductive line of silver ink running under each column of keys.
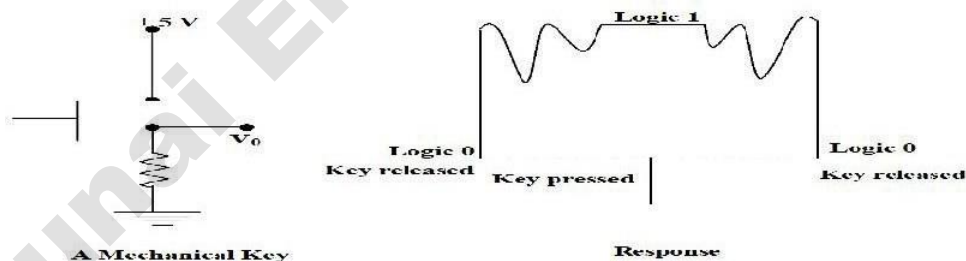
Fig .5.1 Mechanical key and its response to key press

The key board interfaced is a matrix keyboard. This key board is designed with a particular rows and columns. These rows and columns are connected to the microcontroller through its ports of the micro controller 8051. We normally use 8*8 matrix key board. So only two ports of 8051 can be easily connected to the rows and columns of the key board.

Whenever a key is pressed, a row and a column gets shorted through that pressed key and all the other keys are left open. When a key is pressed only a bit in the port goes high which indicates microcontroller that the key is pressed. By this high on the bit key in the corresponding column is identified.

Once we are sure that one of key in the key board is pressed next our aim is to identify that key. To do this we firstly check for particular row and then we check the corresponding column the key board.

To check the row of the pressed key in the keyboard, one of the row is made high by making one of bit in the output port of 8051 high . This is done until the row is found out.

Once we get the row next out job is to find out the column of the pressed key. The column is detected by contents in the input ports with the help of a counter. The content of the input port is rotated with carry until the carry bit is set.

The contents of the counter is then compared and displayed in the display. This display is designed using a seven segment display and a BCD to seven segment decoder IC 7447. The BCD equivalent number of counter is sent through output part of 8051 displays the number of pressed key.
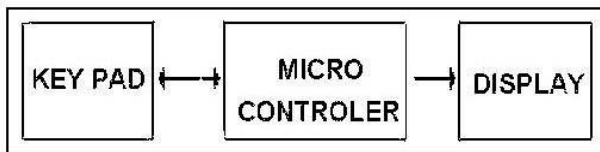
Fig 5.2 Interfacing Keyboard to 8051 Microcontroller
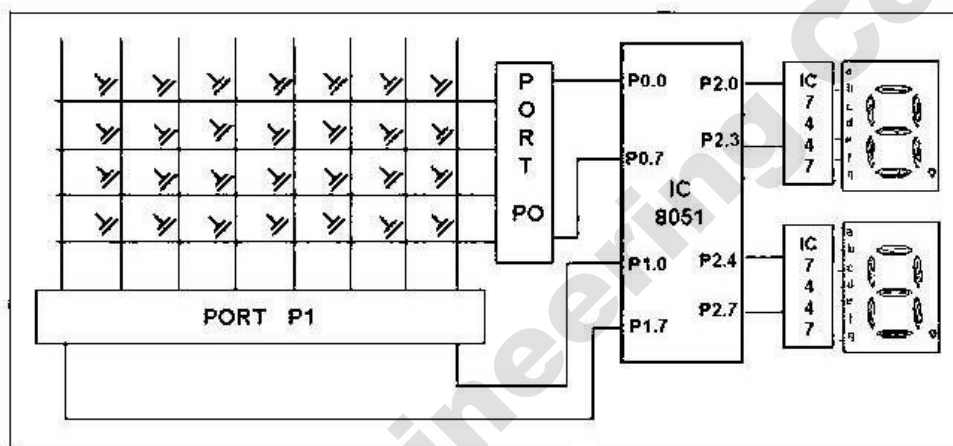
Fig. 5.3 Interfacing To Alphanumeric Displays

• To give directions or data values to users, many microprocessor-controlled instruments and machines need to display letters of the alphabet and numbers. In systems where a large amount of data needs to be displayed a CRT is used to display the data. In system where only a small amount of data needs to be displayed, simple digit-type displays are often used.

• There are several technologies used to make these digit-oriented displays but we are discussing only the two major types.

• These are *light emitting diodes* (LED) and *liquid-crystal displays* (LCD).

• LCD displays use very low power, so they are often used in portable, battery-powered instruments. They do not emit their own light, they simply change the reflection of available light. Therefore, for an instrument that is to be used in low-light conditions, you have to include a light source for LCDs or use LEDs which emit their own light.

## 5.5 Interfacing Analog to Digital Data Converters

• In most of the cases, the PPI 8255 is used for interfacing the analog to digital converters with microprocessor.

• The analog to digital converters is treaded as an input device by the microprocessor, that sends an initialising signal to the ADC to start the analogy to digital data conversation process. The start of conversation signal is a pulse of a specific duration.

• The process of analog to digital conversion is a slow process, and the microprocessor has to wait for the digital data till the conversion is over. After the conversion is over, the ADC sends end of conversion EOC signal to inform the microprocessor that the conversion is

over and the result is ready at the output buffer of the ADC. These tasks of issuing an SOC pulse to ADC, reading EOC signal from the ADC and reading the digital output of the ADC are carried out by the CPU using 8255 I/O ports.

• The time taken by the ADC from the active edge of SOC pulse till the active edge of EOC signal is called as the conversion delay of the ADC.

• It may range anywhere from a few microseconds in case of fast ADC to even a few hundred milliseconds in case of slow ADCs.

• The available ADC in the market use different conversion techniques for conversion of analog signal to digitals. Successive approximation techniques and dual slope integration techniques are the most popular techniques used in the integrated ADC chip.

• General algorithm for ADC interfacing contains the following steps:

    1. Ensure the stability of analog input, applied to the ADC.

    2. Issue start of conversion pulse to ADC

    3. Read end of conversion signal to mark the end of conversion processes.

    4. Read digital data output of the ADC as equivalent digital output.

    5. Analog input voltage must be constant at the input of the ADC right from the start of conversion till the end of the conversion to get correct results. This may be ensured by a sample and hold circuit which samples the analog signal and holds it constant for a specific time duration. The microprocessor may issue a hold signal to the sample and hold circuit.

    6. If the applied input changes before the complete conversion process is over, the digital equivalent of the analog input calculated by the ADC may not be correct.

ADC 0808/0809:

• The analog to digital converter chips 0808 and 0809 are 8-bit CMOS, successive approximation converters. This technique is one of the fast techniques for analog to digital conversion. The conversion delay is 100µs at a clock frequency of 640 KHz, which is quite low as compared to other converters. These converters do not need any external zero or full scale adjustments as they are already taken care of by internal circuits.
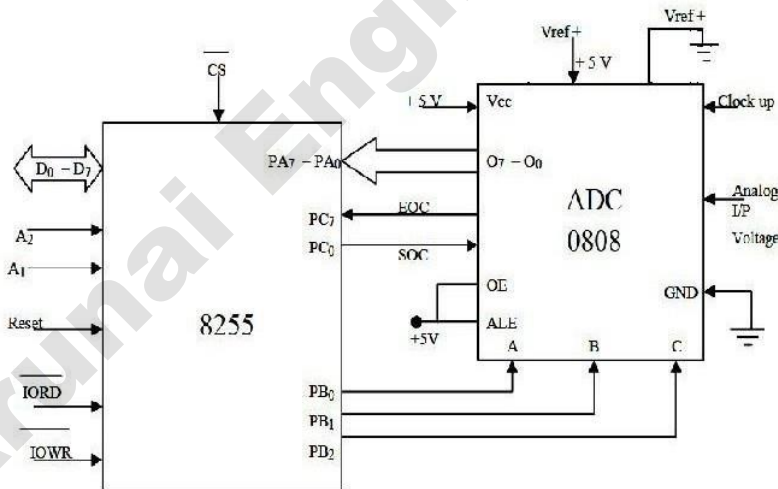
Fig. 5.4 Interfacing ADC with 8255 of microcontroller

·   These converters internally have a 3:8 analog multiplexer so that at a time eight different analog conversion by using address lines - ADD A, ADD B, ADD C. Using these address inputs, multichannel data acquisition system can be designed using a single ADC. The CPU may drive these lines using output port lines in case of multichannel applications. In case of single input applications, these may be hardwired to select the proper input.

• There are unipolar analog to digital converters, i.e. they are able to convert only positive analog input voltage to their digital equivalent. These chips do no contain any internal

sample and hold circuit. If one needs a sample and hold circuit for the conversion of fast signal into equivalent digital quantities, it has to be externally connected at each of the analog inputs.

5.5.b Interfacing Digital to Analog Converters:
The digital to analog converters convert binary number into their equivalent voltages. The DAC find applications in areas like digitally controlled gains, motors speed controls, programmable gain amplifiers etc.

AD 7523 8-bit Multiplying DAC: This is a 16 pin DIP, multiplying digital to analog converter, containing R- 2R ladder for D-A conversion along with single pole double thrown NMOS switches to connect the digital inputs to the ladder.
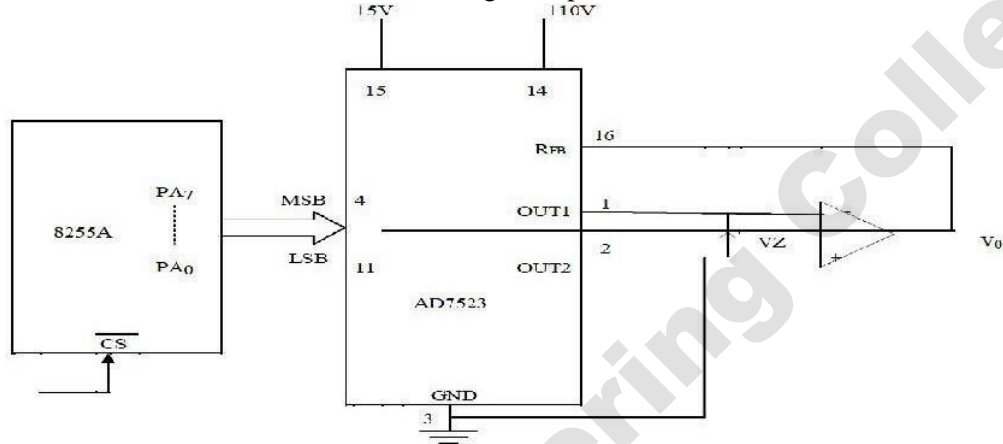
Fig. 5.5 Interfacing ADC with 8255 of microcontroller

5.6 External Memory Interface:
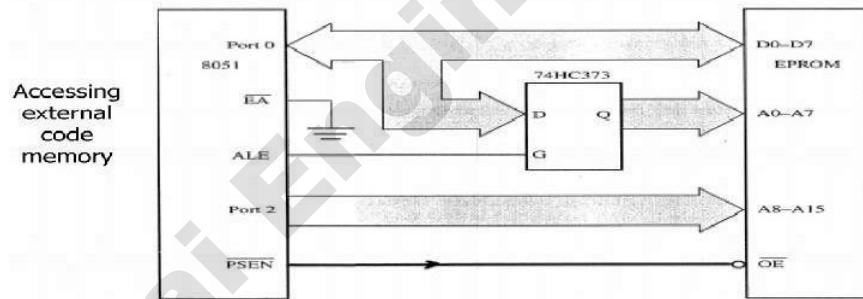
Fig. 5.6 External Memory interfacing

131
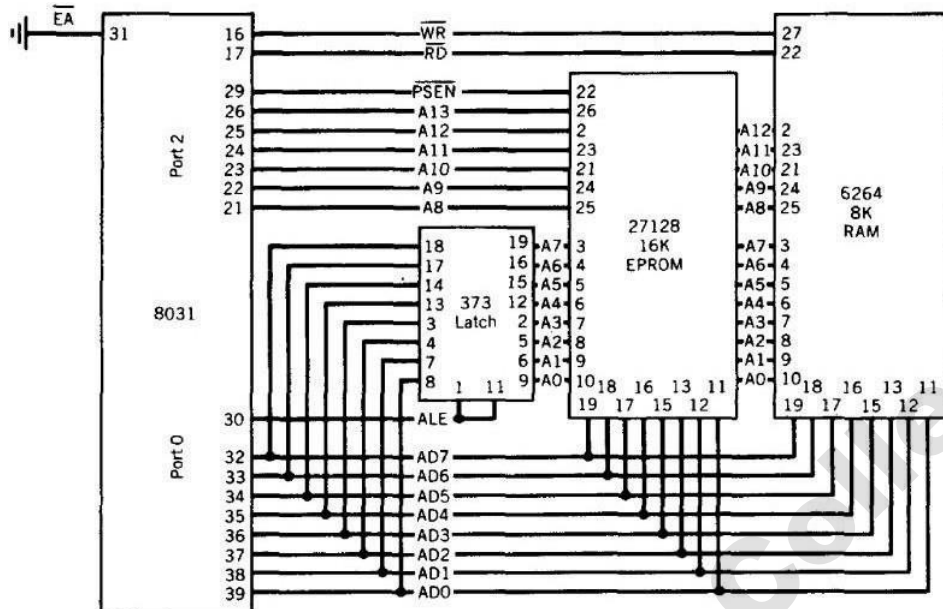
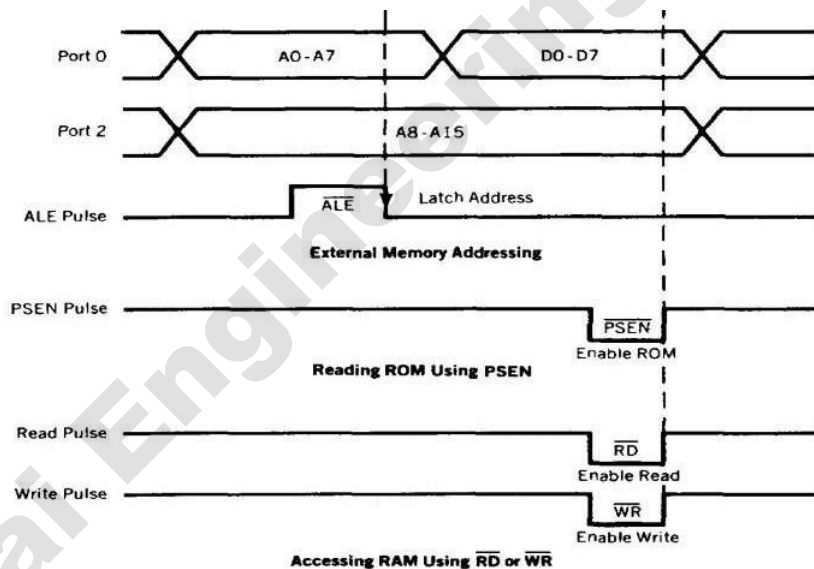**Fig 5.7 Interfacing external memories with microcontroller**

Fig. 5.8 External Memory Timing

5.7 Stepper Motor Interface

The complete board consists of transformer, control circuit, keypad and stepper motor as shown in snap.

The circuit has inbuilt 5 V power supply so when it is connected with transformer it will give the supply to circuit and motor both. The 8 Key keypad is connected with circuit through which user can give the command to control stepper motor. The control circuit includes micro controller 89C51, indicating LEDs, and current driver chip ULN2003A. One can program the controller to control the operation of stepper motor. He can give different commands through keypad like, run clockwise, run anticlockwise, increase/decrease RPM, increase/decrease revolutions, stop motor, change the mode, etc. **Unipolar stepper motor:-** unipolar stepper motor has four coils. One end of each coil is tied together and it gives common terminal which is always connected with positive terminal of supply. The other ends of each coil are given for interface. Specific color code may also be given. Like in my

motor orange is first coil (L1), brown is second (L2), yellow is third (L3), black is fourth (L4) and red for common terminal.

By means of controlling a stepper motor operation we can

1. Increase or decrease the RPM (speed) of it
2. Increase or decrease number of revolutions of it
3. Change its direction means rotate it clockwise or anticlockwise

To vary the RPM of motor we have to vary the PRF (Pulse Repetition Frequency). Number of applied pulses will vary number of rotations and last to change direction we have to change pulse sequence.

So all these three things just depends on applied pulses. Now there are three different modes to rotate this motor

1. Single coil excitation
2. Double coil excitation
3. Half step excitation

The table given below will give you the complete idea that how to give pulses in each mode

Table 5.7 Pulses for stepper motor module

Single coil excitation		Double coil excitation		Half step excitation	
Clockwise	Anticlockwise	Clockwise	Anticlockwise	Clockwise	Anticlockwise
L4 L3 L2 L1	L4 L3 L2 L1	L4 L3 L2 L1	L4 L3 L2 L1	L4 L3 L2 L1	L4 L3 L2 L1
0 0 0 1	0 0 0 1	0 0 1 1	0 0 1 1	0001	0001
0 0 1 0	1 0 0 0	0 1 1 0	1 0 0 1	0011	0011
0 1 0 0	0 1 0 0	1 1 0 0	1 1 0 0	0010	1000
1 0 0 0	0 0 1 0	1 0 0 1	0 1 1 0	0110	1001
				0100	0100
				1100	1100
				1000	0010
				1001	0110

The circuit consists of very few components. The major components are 7805, 89C51 and ULN2003A.

Connections:-

1. The transformer terminals are given to bridge rectifier to generate rectified DC.

2. It is filtered and given to regulator IC 7805 to generate 5 V pure DC. LED indicates supply is ON.

3. All the push button micro switches J1 to J8 are connected with port P1 as shown to form serial keyboard.

4. 12 MHz crystal is connected to oscillator terminals of 89C51 with two biasing capacitors.

5. All the LEDs are connected to port P0 as shown

6. Port P2 drives stepper motor through current driver chip ULN2003A.

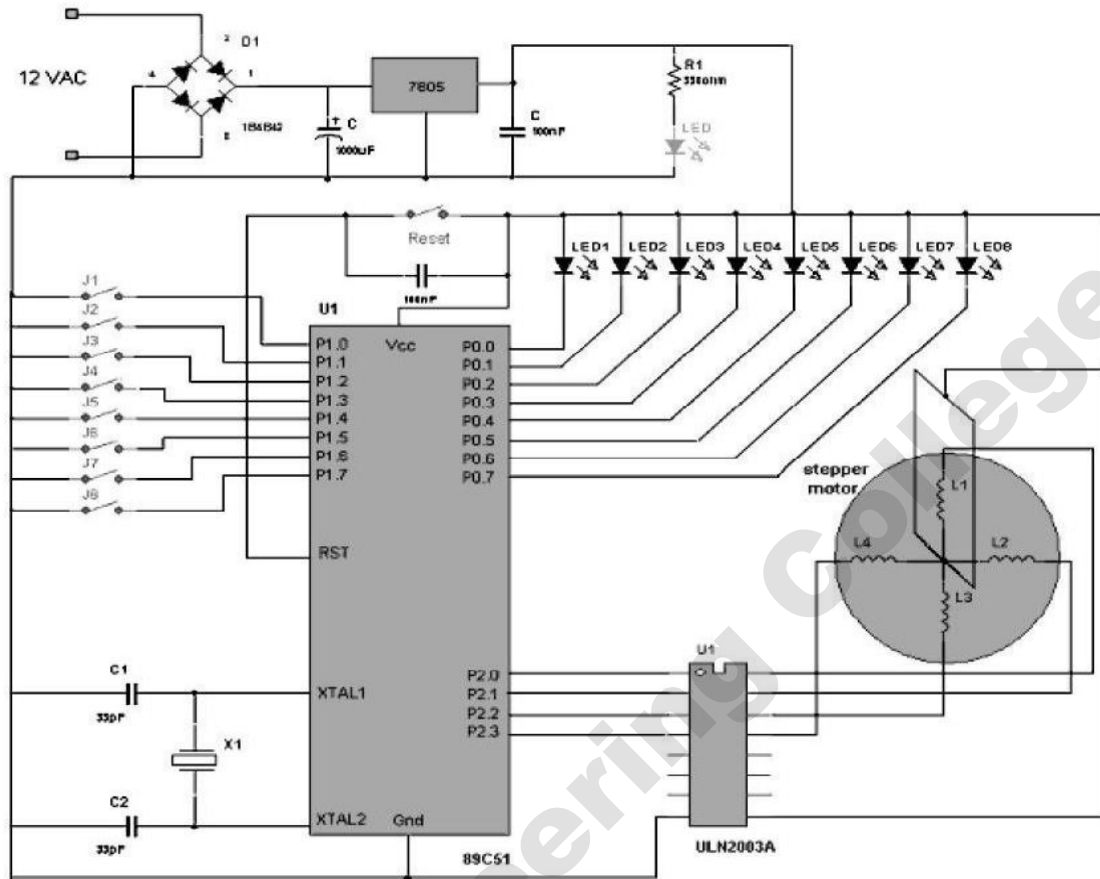7. The common terminal of motor is connected to Vcc and rest all four terminals are connected to port P2 pins in sequence through ULN chip.

Fig.5.9 Stepper motor control board circuit