

# 8087 Co Processors and Architecture

## Overview

- ☐ Each processor in the 80x86 family has a corresponding coprocessor with which it is compatible.
  - ☐ Math Coprocessor is known as NPX, NDP, FUP.
- Numeric processor extension (NPX),  
Numeric data processor (NDP),  
Floating point unit (FUP).

## Compatible Processor and Coprocessor

### Processors

1. 8086 & 8088
2. 80286
3. 80386DX
4. 80386SX
5. 80486DX
6. 80486SX

### Coprocessors

1. 8087
2. 80287, 80287XL
3. 80287, 80387DX
4. 80387SX
5. It is Inbuilt
6. 80487SX

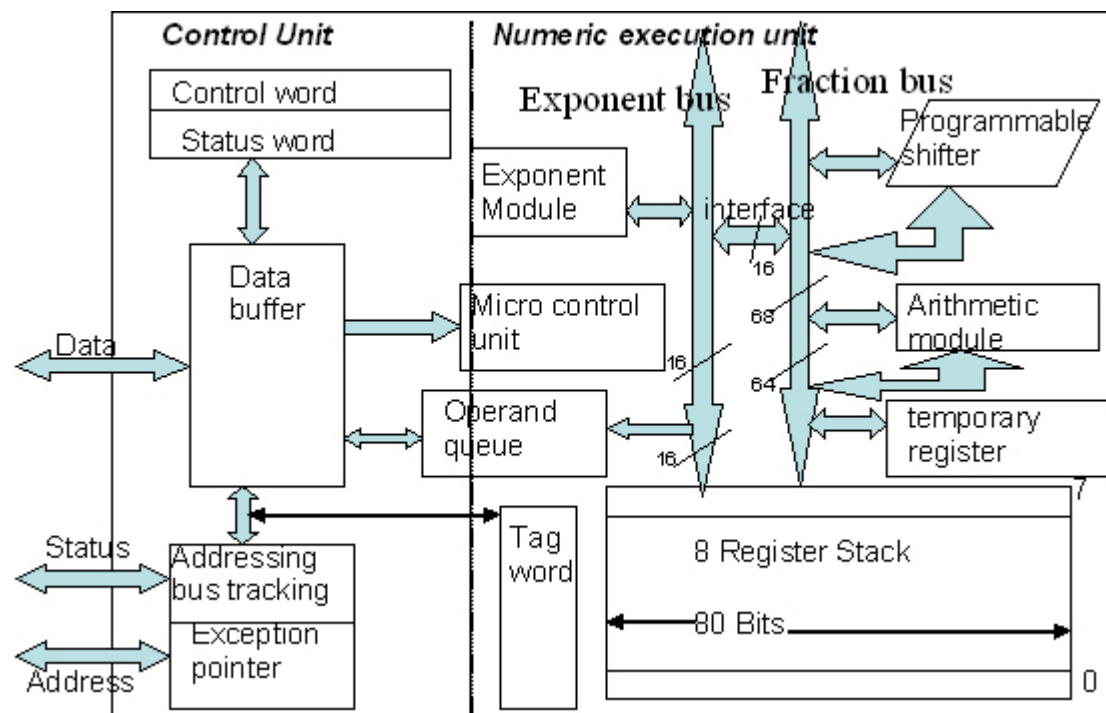
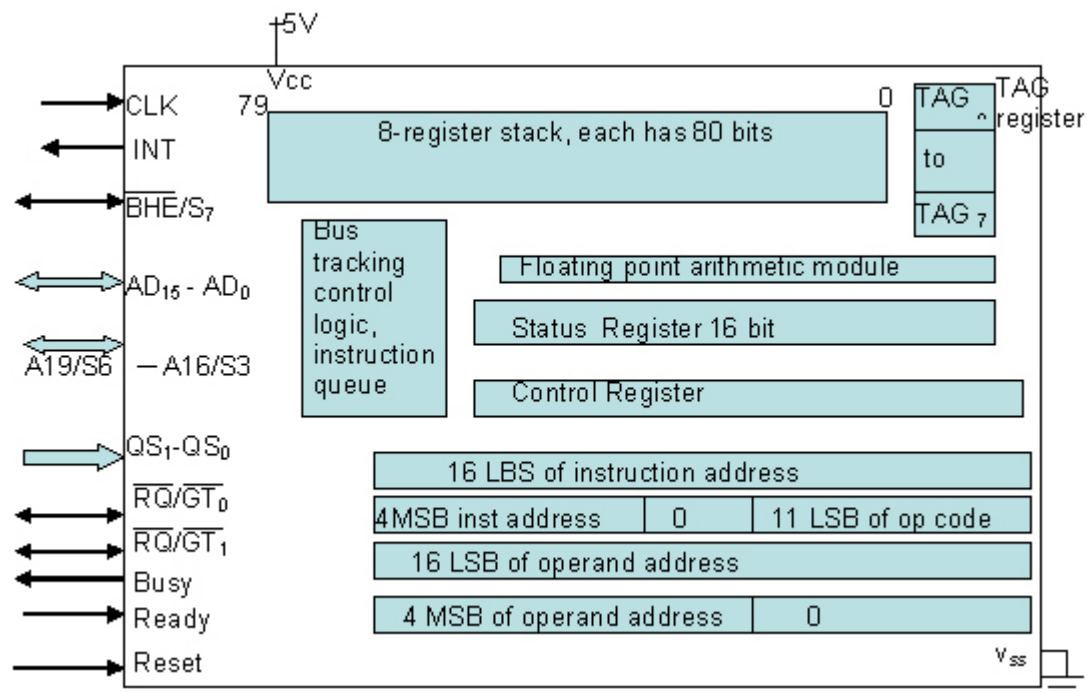
## Architecture of 8087

- ☐ Control Unit
- ☐ Execution Unit

### Control Unit

- ☐ Control unit: To synchronize the operation of the coprocessor and the processor.
- ☐ This unit has a Control word and Status word and Data Buffer
- ☐ If instruction is an *ESC*ape (coprocessor) instruction, the coprocessor executes it, if not the microprocessor executes.
- ☐ Status register reflects the over all operation of the coprocessor.

## Architecture of 8087





## Status Register

15										0					
B	C3		ST		C2	C1	C0	ES		PE	UE	OE	ZE	DE	IE

- C3-C0 Condition code bits
- TOP Top-of-stack (ST)
- ES Error summary
- PE Precision error
- UE Under flow error
- OE Overflow error
- ZE Zero error
- DE Denormalized error
- IE Invalid error
- B Busy bit

☐ B-Busy bit indicates that coprocessor is busy executing a task. Busy can be tested by examining the status or by using the FWAIT instruction. Newer coprocessor automatically synchronize with the microprocessor, so busy flag need not be tested before performing additional coprocessor tasks.

☐ C3-C0 Condition code bits indicates conditions about the coprocessor.

☐ TOP- Top of the stack (ST) bit indicates the current register address as the top of the stack.

☐ ES-Error summary bit is set if any unmasked error bit (PE, UE, OE, ZE, DE, or IE) is set. In the 8087 the error summary is also caused a coprocessor interrupt.

☐ PE- Precision error indicates that the result or operand executes selected precision.

☐ UE-Under flow error indicates the result is too large to be represent with the current precision selected by the control word.

☐ OE-Over flow error indicates a result that is too large to be represented. If this error is masked, the coprocessor generates infinity for an overflow error.

☐ ZE-A Zero error indicates the divisor was zero while the dividend is a non-infinity or non-zero number.

☐ DE-Denormalized error indicates at least one of the operand is denormalized.

☐ IE-Invalid error indicates a stack overflow or underflow, indeterminate from (0/0,0,-0, etc) or the use of a NAN as an operand. This flag indicates error such as those produced by taking the square root of a negative number.

## CONTROL REGISTER

☐ Control register selects precision, rounding control, infinity control.

☐ It also masks an unmask the exception bits that correspond to the rightmost Six bits of status register.

☐ Instruction FLDCW is used to load the value into the control register.

## Control Register

15													0	
		IC	R	C	P	C			PM	UM	OM	ZM	DM	IM

- IC Infinity control
- RC Rounding control
- PC Precision control
- PM Precision control
- UM Underflow mask
- OM Overflow mask
- ZM Division by zero mask
- DM Denormalized operand mask
- IM Invalid operand mask

☐ IC –Infinity control selects either affine or projective infinity. Affine allows positive and negative infinity, while projective assumes infinity is unsigned.

### INFINITY CONTROL

0 = Projective

1 = Affine

☐ RC –Rounding control determines the type of rounding.

### ROUNDING CONTROL

00=Round to nearest or even

01=Round down towards minus infinity

10=Round up towards plus infinity

11=Chop or truncate towards zero

☐ PC- Precision control sets the precision of the result as defined in table

### PRECISION CONTROL

00=Single precision (short)

01=Reserved

10=Double precision (long)

11=Extended precision (temporary)

☐ Exception Masks – It Determines whether the error indicated by the exception affects the error bit in the status register. If a logic1 is placed in one of the exception control bits, corresponding status register bit is masked off.

### Numeric Execution Unit

☐ This performs all operations that access and manipulate the numeric data in the coprocessor's registers.

☐ Numeric registers in NUE are 80 bits wide.

- NUE is able to perform arithmetic, logical and transcendental operations as well as supply a small number of mathematical constants from its on-chip ROM.
- Numeric data is routed into two parts ways a 64 bit mantissa bus and a 16 bit sign/exponent bus.

Source : <http://nprcet.org/e%20content/Misc/e-Learning/IT/IV%20Sem/CS%202252-Microprocessors%20and%20Microcontrollers.pdf>

## \* Difference between minimum mode & Maximum Mode

### minimum mode

### Maximum mode

1) In minimum mode only one processor is available i.e. 8086

1) In maximum mode there can be multiple processors with 8086 like 8087, 8089 etc.

2)  $\overline{MN/\overline{MX}}$  is 1 to indicate min mode.

2)  $\overline{MN/\overline{MX}}$  is 0, to indicate max mode.

3) ALE for the latch is given by 8086, as it is the only processor in the circuit.

3) ALE for the latch is given by 8288, bus controller, as there can be multiple processors in the circuit.

4)  $\overline{DEN}$  and  $\overline{DT/R}$  for the trans-receivers are given by 8086 itself

4)  $\overline{DT/R}$  for the trans-receivers are given by 8288 bus controller

5) Direct control signal  $\overline{M/\overline{IO}}$ ,  $\overline{RD}$ , &  $\overline{WR}$  given by 8086

5) Instead of control signal, each processor generate status signal called  $\overline{S_1}$ ,  $\overline{S_2}$  &  $\overline{S_0}$

6) control signals  $\overline{M/\overline{IO}}$ ,  $\overline{RD}$ , &  $\overline{WR}$  are decoded by 2:8 decoder like 74138.

6) Status signals  $\overline{S_1}$ ,  $\overline{S_2}$  &  $\overline{S_0}$  are decoded by bus controller like 8288.

7)  $\overline{INTA}$  is given by 8086 in response to an interrupt on  $\overline{INTR}$  line.

7)  $\overline{INTA}$  is given by bus controller 8288. in response to an interrupt on  $\overline{INTR}$  line.

8)  $\overline{HOLD}$  &  $\overline{HLDA}$  signals are used for bus request with DMA controller like 8237

8)  $\overline{RQ}/\overline{GT}$  lines are used for bus request by other processor like 8087 & 8089

9) The circuit is simpler.

9) circuit is more complex

10) performance is slower.

10) performance is faster.

## Module 02:-

### Instruction Set.

DR. NAVEEN I. B

Associate professor

Dept of ECE

B-G-S-I-T.

#### ✓ Addressing modes:-

\* The CPU can access data in various ways. The data could be in a register, or in memory or be provided as an immediate value.

\* The various ways of accessing data are called addressing modes.

There are 5 addressing modes in 8051

✓ 1) Immediate addressing mode.

2) Register addressing mode.

3) Direct addressing mode.

4) Register indirect addressing mode.

5) Indexed addressing mode.

#### ✓ 1) Immediate addressing mode:-

\* In this addressing mode, the source operand is a constant. The immediate data must be preceded by the pound sign "#".

\* This addressing mode can be used to load information into any of the registers, including the DPTR register & 8051 ports.

ex:-

- 1) MOV A, #FFh.
- 2) MOV R4, #0Ah.
- 3) MOV B, #10h.
- 4) MOV DPTR, #1234h.
- 5) MOV P1, #55h.



## 2) Register addressing mode :-

\* Register addressing mode involves the use of registers to hold the data to be manipulated.

\* The registers A, DPTR, & R<sub>0</sub> to R<sub>7</sub> may be used as source as well as destination.

Ex:- 1) MOV A, R<sub>0</sub>

2) MOV R<sub>2</sub>, A

3) ADD A, R<sub>5</sub>

4) ADD A, R<sub>7</sub>

5) MOV R<sub>6</sub>, A

6) MOV DPTR, #0123h.

7) MOV R<sub>7</sub>, DPL

8) MOV R<sub>6</sub>, DPH.

NOTE:-

\* We can move data between the accumulator & R<sub>n</sub> register (n=0 to 7) but movement of data b/w R<sub>n</sub> register is not allowed.

ex:- MOV R<sub>4</sub>, R<sub>7</sub> is Invalid.

## 3) Direct addressing mode :-

{

1) RAM locations 00-1Fh are assigned to the register banks & stack.

2) RAM locations 20-2Fh are set aside as bit addressable space to save single-bit data.

3) RAM locations 30-7Fh are available as a place to save byte-sized data.

}

\* The entire 128 bytes of RAM can be accessed using direct addressing mode. The RAM locations 30h to 7Fh are most often used.

\* In direct addressing mode, the data is in a RAM memory location whose address is known, & this address is given as a part of the instructions.

NOTE:- The '#' sign distinguishes b/w immediate & direct addressing.

Exs:-

- 1) MOV R0, 40h.
- 2) MOV 56h, A
- 3) MOV R7, 01h.
- 4) PUSH 0E0h.
- 5) POP 03h etc.

{ RAM locations 0 to 7 are allocated to bank 0 registers R0 - R7. These registers can be accessed in two ways.

- 1) MOV A, R4 is same as MOV A, 4
- 2) MOV A, R7 is same as MOV A, 7
- 3) MOV A, R0 is same as MOV A, 0.
- 4) MOV R2, R3 is same as MOV 2, 3 but Invalid inst

{



#### 4) Register Indirect addressing mode :-

- \* In register indirect addressing mode a register is used to hold the address of the data.
- \* The register itself is not the address, but rather the number in the register.
- \* The instruction for indirect addressing uses MOV opcode along with register R<sub>0</sub> or R<sub>1</sub>. Register R<sub>0</sub> or R<sub>1</sub> will hold the RAM addresses ranging from 00h to 7Fh.
- \* The mnemonic symbol used for indirect addressing is the "at" sign i.e., "@".

Ex:-

- 1) MOV A, @R<sub>0</sub>.
- 2) MOV @R<sub>1</sub>, B.
- 3) MOV @R<sub>1</sub>, A
- 4) MOV 20h, @R<sub>1</sub>
- 5) MOV @R<sub>0</sub>, 03h.

#### Limitations of register indirect addressing mode :-

- \* R<sub>0</sub> & R<sub>1</sub> are the only registers that can be used for pointers in register indirect addressing mode.  
Since R<sub>0</sub> & R<sub>1</sub> are 8-bit wide, their use is limited



## 5) Indexed addressing mode :-

\* Indexed addressing mode is widely used in accessing data elements of look-up table entries located in the program ROM space.

\* The instructions used for this purpose is

i) `MOVC A, @A + DPTR`

ii) `MOVC A, @A + PC`

}

i) `MOVC A, @A + DPTR` : Add the contents of the accumulator with the contents of the DPTR register to form a program code memory location address. Move the contents of this external memory address to the accumulator.

ii) `MOVC A, @A + PC` :-

Add the contents of the accumulator with the contents of the PC register to form a program code memory location address. Move the contents of this external memory address to the accumulator.

# Instruction Set - 8051

DR. NAVEEN B  
Associate prof.  
Dept of EEE  
B.G.S.I.T.

Based on the operations performed, the instruction set of 8051 are classified as

- 1) Data transfer instructions. ✓
- 2) Arithmetic instructions. ✓
- 3) Logical instructions. ✓
- 4) Boolean instructions. ✓
- 5) Program branching or Machine control instructions. ✓

\* Each instruction has two parts:  
operation code & operands.

## Data Transfer group:-

The instructions in this group are

MOV, PUSH, POP, XCH.

### MOV :-

MOV instruction copies data from one location to another location.

Syntax:- MOV operand 1, operand 2

Format:- MOV Destination, source;  
copy from source to destination.



✓ 1) MOV A, Rn. → Destination (Operator)  
→ Source (operand)  
Bytes : 1  
cycles : 1

status flags affected : None.

operation: MOV

$(A) \leftarrow (R_n)$

Description :- This instruction moves the contents of Rn register to accumulator. The Rn register is not affected.

NOTE :- Rn may be R0 to R7 register of the currently selected bank.

ex:- MOV A, R3      extmp R3 = 58h      A = 10h

Before Execution → R3 = 58h, A = Any Value say 10h.

After Execution → A = 58h. & R3 = 58h.

2) MOV A, direct (Direct address)

Bytes : 2.

cycles : 1

operation:  $(A) \leftarrow (\text{direct})$

Description : This instruction moves the contents of the address into A register.

Flags affected: None

ex:- MOV A, 40h.

Before Execution.

$A \leftarrow 'xx'$

$40h \leftarrow FF$

After Execution.

$A \leftarrow FF$

$40h \leftarrow FF$

NOTE:- Here 40h is a direct address. The direct address content 'FF' is moved into Accumulator.

3) MOV A, @R<sub>i</sub>

Bytes : 1

Cycles : 1.

operation :  $(A) \leftarrow ((R_i))$

Description :

Flags affected: None.

ex:-

MOV A, @R<sub>0</sub>.

Say  $R_0 = 40h \leftarrow \text{address.}$

$40h = FF \leftarrow \text{data.}$

\* The address 40h is in register R<sub>0</sub>.

\* The data FF is in address 40h.

Before execution.

$R_0 \leftarrow 40h.$

$40h \leftarrow FF.$

$A \leftarrow 'xx'$

After execution.

$R_0 \leftarrow 40h.$

$40h \leftarrow FF$

$A \leftarrow FF.$



4) MOV A, #data.

Bytes : 2.

cycles : 1

operation:  $(A) \leftarrow \#data$ .

Flags affected: None.

Description: 8-bit data is moved into accumulator.

ex:- MOV A, #28

Before execution

$A \leftarrow 'xx'$

After execution.

$A \leftarrow 28$ .

5) MOV Rn, A.

Bytes : 1.

cycles : 1.

operation:  $(R_n) \leftarrow (A)$ .

Description: The contents of accumulator is moved to register Rn.

Flags affected: None.

ex:- MOV R7, A

Before Execution

$R_7 = 'xx'$

$A = FF$

After execution.

$R_7 = FF$

$A = FF$ .

6) MOV R<sub>n</sub>, Direct

R<sub>n</sub> may be any register

Bytes : 2.

(R<sub>0</sub> to R<sub>7</sub>).

cycles : 2.

operation : (R<sub>n</sub>) ← (direct)

Flags affected: None.

ex:-

MOV R<sub>1</sub>, 40h

Before execution.

40h = FF

R<sub>1</sub> = 'xx'

After execution.

40h = FF

R<sub>1</sub> = FF

7) MOV R<sub>n</sub>, #data.

Bytes : 2.

R<sub>n</sub> may be any register

cycles : 1.

(R<sub>0</sub> to R<sub>7</sub>)

operation : (R<sub>n</sub>) ← #data.

Flags affected: None

ex:-

MOV R<sub>5</sub>, #00

Before Execution

R<sub>5</sub> = 'xx'

After Execution.

R<sub>5</sub> = 00.

8) MOV direct, A.

Bytes : 2.

cycles : 1.

operation : (direct) ← (A)

Flags affected: None.



ex:- MOV 50h, R<sub>3</sub>.

Before execution

50h = FF

R<sub>3</sub> = 00

After execution.

50h = 00

R<sub>3</sub> = 00

10) MOV direct, direct

Bytes : 3

cycles : 2

operation: (direct) ← (direct)

Flags affected: None.

ex:- MOV 30h, 50h.

Before execution.

30h = 11

50h = 00

After execution.

30h = 00

50h = 00.

11) MOV direct, @Ri

Bytes : 2

cycles : 2

operation: (direct) ← (Ri)

Flags affected: None.

ex:- MOV 70h, @R<sub>0</sub>

Before execution

70h = 00

R<sub>0</sub> = 40h.

40h = FF.

After execution.

70h = FF

R<sub>0</sub> = 40h.

40h = FF

12) MOV direct, #data.

Bytes: 3.

Cycles: 2.

operation: (direct)  $\leftarrow$  #data.

Flags affected: None.

ex:- MOV 70h, #FF

Before execution.

70h = 'xx'

After execution.

70h = FF

13) MOV @Ri, direct

Bytes: 2.

Cycles: 2.

operation: ((Ri))  $\leftarrow$  (direct)

Flags affected: None.

ex:- MOV @R1, 70h.

Before execution

R<sub>1</sub> = 40h

40h = 'xx'

70h = FF

After execution.

R<sub>1</sub> = 40h

40h = FF

70h = FF.

14) MOV @Ri, #data.

Bytes: 2

Cycles: 1

operation: ((Ri))  $\leftarrow$  #data.

Flags affected: None.



ex:- MOV @R0, #00

Before execution

$R_0 = 40h$ .

$40h = 'xx'$

After execution.

$R_0 = 40h$ .

$40h = 00$ .

\* MOV dest-bit, source-bit

Function: Move bit data from source bit to destination bit.

NOTE:-

One of the operands must be the carry flag; the other may be any directly addressable bit.

Flags affected: carry flag.

15) MOV C, bit

Bytes: 2

Cycles: 1.

operation:  $(C) \leftarrow (\text{bit})$

Flags affected: carry flag (CY)

ex:- i) MOV C, P1.4

Before execution

$C = 'x'$

$P1.4 = 1$

After execution.

$C = 1$

$P1.4 = 1$ .

ii) MOV C, P0.7

Before execution.

$C = 'x'$

$P0.7 = 0$ .

After execution.

$C = 0$

$P0.7 = 0$ .

16) MOV bit, C

Bytes : 2.

cycles : 2.

operation : (bit)  $\leftarrow$  (C)

Flags affected : None.

ex:- i) MOV P1.2, C.

Before execution

P1.2 = 'X'

C = 1.

After execution.

P1.2 = 1

C = 1.

ii) MOV P3.7, C.

Before execution

P3.7 = 'X'

C = 0

After execution

P3.7 = 0

C = 0.



# ✓ Arithmetic Group Op. Instructions :-

\* ADD A, source.

✓ Syntax : ADD A, operand.

✓ Function : Add

Description : Adds the contents of source with Accumulator contents & result is stored in accumulator.

Flags affected : C, AC & OV.

NOTE :- OV is set if there is a carry-out of bit 6 but no carry out from bit 7 or a carry out of bit 7 but no carry out from bit 6 ; otherwise  $OV = 0$ .

1) ADD A, R<sub>n</sub>.

Where R<sub>n</sub> = R<sub>0</sub> to R<sub>7</sub>.

Bytes : 1.

Cycles : 1

operation:  $(A) \leftarrow (A) + (\text{direct})$

Flags affected: CY, OV & AC.

ex:-

1) ADD A, R<sub>4</sub>.

Before execution.

A = '11'

R<sub>4</sub> = 11.

After execution.

A = 22

R<sub>4</sub> = 11

A = 02

R<sub>4</sub> = 02

A + R<sub>4</sub> = 04

(A = 04)

1) ADD A, R<sub>7</sub>

Before execution.

A = 00

R<sub>7</sub> = FF.

After execution

A = FF

R<sub>7</sub> = FF.

2) ADD A, direct.

Bytes : 2 ✓

cycles : 1

operation :  $(A) \leftarrow (A) + (\text{direct})$

Flags affected: CY, AC, OV.

✓ ex:-

ADD A, 40h.

Before execution.

A = 11

40h = 22.

After execution.

A = 33

40h = 22.

3) ADD A, @R<sub>i</sub>

Bytes : 1

cycles : 1 ✓

operation :  $(A) \leftarrow (A) + ((R_i))$

Flags affected: CY, AC, OV.

Where  $R_i \rightarrow$  may be R<sub>0</sub> or R<sub>1</sub>.

Program:-

MOV @R<sub>1</sub>, #70h.

MOV A, #30h

ADD A, @R<sub>1</sub>.

✓ ex:- ADD A, @R<sub>1</sub>

Before execution.

A = 30

R<sub>1</sub> = 70h.

70h = 20

After execution.

A = 50

R<sub>1</sub> = 70h.

70h = 20.



4) ADD A, # data.

Bytes : 2.

Cycles : 1.

operation :  $(A) \leftarrow (A) + \# \text{ data}$ .

Flags affected : CY, OV, AC.

ex:- ADD A, # 01h.

Before execution.

A = 01.

After execution.

A = 02.

5) ADC A, Rn.

Bytes : 1.

Cycles : 1.

operation :  $(A) \leftarrow (A) + (C) + (Rn)$

Description: Simultaneously add the contents of Rn register, the carry flag & the accumulator contents, result is stored in accumulator.

Flags affected: OV, CY, AC.

ex:- 1) ADC A, R7

Before execution

✓ A = 01

✓ C = 1

✓ R7 = 01.

After execution

A = 03

✓ C = 1

R7 = 01.

ii) ADDC A, R<sub>0</sub>

Before execution

A = 01

C = 0

R<sub>0</sub> = 01

After execution

A = 02

C = 0

R<sub>0</sub> = 01

6) ADDC A, direct

Bytes : 2

Cycles : 1

operation :  $(A) \leftarrow (A) + (C) + (\text{direct})$

Flags affected : CY, AC, OV.

✓ ex: ADDC A, 80h

Before execution

A = 02

C = 1

80h = 02

After execution

A = 05

C = 1

80h = 02

7) ADDC A, @R<sub>i</sub>

Bytes : 1

Cycles : 1

operation :  $(A) \leftarrow (A) + (C) + ((R_i))$

Flags affected : CY, AC, OV.

where  $R_i = R_0 \text{ or } R_1$



Before execution

$A = 01$

$C = 1$

$R_0 = 70h$

$70h = 01$

After execution.

$A = 03$

$C = 1$

$R_0 = 70h$

$70h = 01$

8) ADDC A, #data.

Bytes : 2

cycles : 1

operation:  $(A) \leftarrow (A) + (C) + \#data.$

Flags affected: CY, AC, OV

Ex:- ADDC A, #01h

Before execution.

$A = 01$

$C = 1$

After execution.

$A = 03$

$C = 1$

9) MUL AB.

Function : Multiply.  $A \times B$ .

Description : MUL AB multiplies the unsigned eight-bit integer i.e the contents of accumulator is multiplied with the contents of register B.

Bytes : 1

cycles : 4.

operation :  $(A)_{7-0}$  }  $\leftarrow (A) \times (B)$   
 $(B)_{15-8}$

Flags affected : C, OV

ex:- {  
 MOV A, #5  
 MOV B, #7  
 MUL AB

$$7 \times 5 = 35 = 23h$$

$A = 35 = 23h$ $B = 00$
----------------------------

Before execution.

$$50 \times A0 = 3200h$$

$$A = 50h$$

$$B = A0h$$

After execution.

$$A = 00h \leftarrow (0-7) \text{ lower byte}$$

$$B = 32h \leftarrow (15-8) \text{ higher byte}$$

$$\text{ex:- } A = 100, B = 200$$

$$\rightarrow 100 \times 200 = 20,000 \\ = 4E20h$$

$$A \times B = 4E20h$$

$A = 20h, B = 4E$
-------------------

10 DIV AB

Function : Divide.

Bytes : 1

cycles : 4.

operation :  $(A)_{15-8}$  }  $\leftarrow (A) / (B)$   
 $(B)_{7-0}$

Flags affected : CY, OV.



Description : DIV AB divides the unsigned 8-bit integer content of accumulator by the unsigned 8-bit integer content of B-register.

The accumulator receives the integer part of the quotient; register B receives the integer remainder.

the carry & overflow flags will be cleared.

ex:- Before execution

A = FBh (251 dec)

B = 12h (18 dec).

After execution.

A = 0dh (13 dec)

B = 11h (17 dec)

13 ← (A).  

$$\begin{array}{r} 18 \overline{) 251} \\ \underline{234} \\ 17 \end{array}$$
  
 17 ← (B)

\* the quotient 13 (decimal) 0dh is stored in accumulator & the remainder (17 decimal) 11h is stored in B-register.

ex:- > A = 35, B = 10

DIV AB

A = 3, B = 5

$$\begin{array}{r} 35 \\ 10 \end{array}$$

3 ← Quotient  

$$\begin{array}{r} 10 \overline{) 35} \\ \underline{30} \\ 05 \end{array}$$
  
 05 ↑  
 Remainder.

10 ) 35 (3 → A  

$$\begin{array}{r} 30 \\ \underline{05} \end{array}$$
  
 05

2) A = 97h = 151

B = 12h = 18

DIV AB

A = 8, B = 7

8  

$$\begin{array}{r} 18 \overline{) 151} \\ \underline{144} \\ 7 \end{array}$$

11) INC A

Bytes : 1.

cycles : 1.

operation :  $(A) \leftarrow (A) + 1$ .

Flags affected: None.

Before execution.

A = 01

After execution.

A = 02

12) INC Rn.

Bytes : 1.

cycles : 1.

operation :  $(Rn) \leftarrow (Rn) + 1$ .

Flags affected: None.

ex: INC R0.

Before execution.

R0 = 30h.

After execution.

R0 = 31h.

13) INC direct

Bytes : 2

cycles : 1

operation:  $(\text{direct}) \leftarrow (\text{direct}) + 1$

Flags affected: None.



ex:- INC 40h.

Before execution.

40h = 01

After execution.

40h = 02.

14) INC @R<sub>i</sub>

where, R<sub>i</sub> → R<sub>0</sub> or R<sub>1</sub>.

Bytes : 1.

cycles : 1.

operation : ((R<sub>i</sub>)) ← ((R<sub>i</sub>)) + 1.

Flags : None.

Function : Increment.

ex: INC @R<sub>0</sub>.

Before execution.

R<sub>0</sub> = 80h.

80h = 01.

After execution.

R<sub>0</sub> = 80h

80h = 02.

15) INC DPTR.

Function : Increment data pointer.

Bytes : 1.

cycles : 2.

operation : (DPTR) ← (DPTR) + 1.

Description : This instruction increments the 16-bit register content (DPTR) by 1. This is the only 16-bit register that can be incremented.

Flags : None.

ex: INC DPTR.

i) Before Execution.

DPTR = 16 FFh.

(DPH = 16, DPL = FF)

ii) After execution.

DPTR = 1700h.

(DPH = 17, DPL = 00)

iii) Before Execution.

DPTR = 00FFh.

DPH = 00h.

DPL = FFh.

00

iv) After execution.

DPTR = 0100h.

DPH = 01h

DPL = 00h.

16) DEC A.

Function : Decrement.

Bytes : 1.

Cycles : 1.

operation:  $(A) \leftarrow (A) - 1$

Flags : None.

Before execution.

A = 05h.

After execution.

A = 04h.

17) DEC Rn.

Bytes : 1.

Cycles : 1.

operation:  $(Rn) \leftarrow (Rn) - 1$ .

Flags : None.



ex:- DEC R0.

Before execution.

$R_0 = 80h$ .

$80h = 05h$ .

After execution.

$R_0 = 80h$ .

$80h = 04h$ .

18> DEC direct

Bytes : 2.

Cycles : 1.

operation :  $(direct) \leftarrow (direct) - 1$ .

Flags : None.

ex:- DEC 40h.

Before execution

$40h = 05h$ .

After execution.

$40h = 04h$ .

19> DEC @Ri

Bytes : 1.

Cycles : 1.

operation :  $((R_i)) \leftarrow ((R_i)) - 1$ .

Flags : None.

ex:- DEC @R0.

Before execution.

$R_0 = 80h$

$80h = 04h$ .

After execution

$R_0 = 80h$

$80h = 03h$ .

# DA Instruction

20/ DA A

Function : Decimal - adjust accumulator after addition.

Flag : CY.

Description : This instruction is used after addition of BCD numbers to convert the result back to BCD. The data is adjusted in the following two possible cases.

- 1) It adds 6 to the lower 4-bits of A if it is greater than 9 or if  $AC=1$ .
- 2) It also adds 6 to the upper 4-bits of A if it is greater than 9 or if  $CY=1$ .

Bytes : 1.

Cycles : 1.

operation: if  $\left[ \left[ (A_{3-0}) > 9 \right] \vee [AC=1] \right]$  then

$$(A_{3-0}) \leftarrow (A_{3-0}) + 6.$$

if  $\left[ \left[ (A_{7-4}) > 9 \right] \vee [CY=1] \right]$  then.

$$(A_{7-4}) \leftarrow (A_{7-4}) + 6.$$

ex:- MOV A, #47h.

A = 47h.

ADD A, #38h ;

A = 47h + 38h = 7Fh, invalid BCD.

DA A

;

A = 85h, valid BCD.



i.e. 47h.

+ 38h

7Fh.

Invalid BCD

6h.

After DA A

85h

Valid BCD

AC=1

\* Since the lower nibble is greater than 9, DA added 6 to A. If the lower nibble is less than 9 but AC=1, it also adds 6 to the lower nibble.

ii) MOV A, #29h.

ADD A, #18h.

DA A

AC=1

29h.

+ 18h

41h.

← (Incorrect result in BCD)

+ 6

47h.

← (correct result in BCD)

iii) MOV A, #52h.

ADD A, #91h.

DA A

CY=1

52h.

+ 91h

E3h.

Invalid BCD.

+ 6

143h.

upper nibble is > 9, so added 6.

Valid BCD

iv) MOV A, #54h.

ADD A, #87h.

DA A

AC=1, CY=1

54h.

87h

① db.

66

141.

Invalid BCD

BCD.

21) SUBB A, Source byte.

Function: Subtract with borrow.

Flags: OV, AC, CY.

Operation:  $(A) = (A) - (\text{byte}) - (C)$  or  $(A) = (A - \text{byte} - C)$

Description: SUBB subtracts the source byte & the carry flag from the accumulator & puts the result in the accumulator.

SUBB instruction sets the carry flag according to the following.

			CY	
i)	destination byte	>	source byte.	0 the result is +ve.
ii)	destination byte.	=	source byte.	0 the result is 0.
iii)	destination byte	<	source byte.	1 the result is -ve in 2's complement.

ex:- XCH A, Rn.

Bytes: 1

Cycles: 1.

Operation:  $(A) \leftrightarrow (Rn)$

Flags: None.

ex:- XCH A, R2.

Before execution.

A = FFh.

R2 = 11h.

After execution.

A = 11h.

R2 = FFh.



Ex:- MOV A, #FFh.

MOV R2, #11h.

XCH A, R2.

23> XCH A, direct

Bytes : 2

Cycles : 1

operation :  $(A) \leftarrow (\text{direct})$

Flags : None

Ex:- XCH A, 40h.

Before execution.

$A = FFh.$

$40h = 11h.$

After execution.

$A = 11h.$

$40h = FFh$

24> XCH A, @Ri

Bytes : 1

Cycles : 1

operation :  $(A) \longleftrightarrow ((Ri))$

Flags : None

Ex:- XCH A, @R1

MOV 40h, #11h

MOV R1, #40h.

MOV A, #FFh.

XCH A, @R1.

Before execution

$40h = 11h.$

$A = FFh.$

After execution.

$40h = FFh.$

$A = 11h.$

25) XCHD A, @Ri

846/6

Function : Exchange Digit

Description : The XCHD instruction exchanges only the lower nibble of accumulator (bit 3-0), with the lower nibble of the RAM location pointed to by Ri.

\* The higher-order nibbles (bits 7-4) of each registers are not affected.

Flags : None.

Bytes : 1.

Cycles : 1.

Operation :  $A_{(3-0)} \leftrightarrow ((Ri)_{(3-0)})$

Ex:- XCHD A, @R1.

MOV A, #FFh.

MOV R1, #50h.

MOV 50h, #00h.

XCHD A, @R1.

Before execution.

A = FFh.

R1 = 50h.

50h = 00h.

After execution.

A = F0h.

R1 = 50h.

50h = 0Fh.

26) MOV DPTR, #16-bit Value.

Function : Load Data pointer with a 16-bit constant.

Description : The data pointer is loaded with the 16-bit constant indicated.

The DPH register holds high-order byte, while DPL holds the low-order byte.



Flags : None.

Bytes : 3.

Cycles : 2.

operation:  $(DPTR) \leftarrow \# \text{data}(0-15)$

$DPH \leftarrow \# \text{data } 15-8$

$DPL \leftarrow \# \text{data } 7-0$

1 2 3 4

14 32

Ex:- MOV DPTR, #1234.

After execution

DPH = 12 h.

DPL = 34 h.

27> MOVX dest-byte; source-byte.

Function : Move External.

Description: \* This instruction transfers data b/w external memory & register A, hence the "x" appended to MOV.

\* The address of external memory location being accessed can be 16-bit or 8-bit.

MOVX A, @Ri

Bytes : 1.

Cycles : 2.

operation:  $(A) \leftarrow ((Ri))$

Flags : None.

Ex:- MOVX A, @R0.

MOV R0, #50h.

MOV 50h, #FFh

MOVX A, @R0.

Before execution

R0 = 50h.

50h = FFh.

A = 'xx'

After execution.

A = FFh.

R0 = 50h.

50h = FFh.

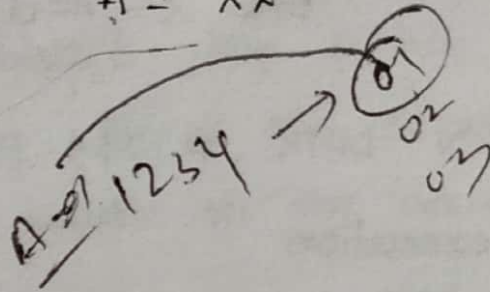
28> MOV A, @DPTR.

Bytes : 1.

Cycles : 2

operation:  $(A) \leftarrow ((DPTR))$

Flags : None.



Ex:- MOVX A, @DPTR.

MOV DPTR, #1234

MOV 1234h, #FFh.

MOVX A, @DPTR.

Before execution

1234h = FFh.

DPTR = 1234.

A = 'xx'

After execution.

A = FFh.

DPTR = 1234h.

1234h = FFh.

29> MOVX @Ri, A

Flags : None

Bytes : 1

Cycles : 2

operation:  $((Ri)) \leftarrow (A)$



Ex:- MOVX @R1, A

MOV R1, #80h.

MOV 80h, #AAh.

MOVX @R1, A.

Before Execution.

After Execution.

R1 = 80h.

A = AAh.

80h = AAh.

R1 = 80h.

A = 'xx'

80h = AAh.

R1 = 80 A = AA

80h = AA

30} MOVX @DPTR, A.

Bytes : 1.

cycles : 2.

operation : ((DPTR)) ← (A)

Flags : None.

Ex:- MOVX @DPTR, A.

MOV DPTR, #1234

MOV 1234, #FFh.

MOVX @DPTR, A.

Before execution

After execution.

DPTR ← 1234h

A = FFh.

1234h ← FFh.

DPTR = 1234h.

A ← 'xx'

1234 = FFh.

# ✓ Logical Instructions :-

\* ANL dest-byte, Src-byte.

Function : Logical - AND for byte variables.

Description : ANL performs the bitwise logical - AND operation b/w the variables indicated & stores the result in the destination variable.

Flags : None.

And Operator

A, #02

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

1) ANL A, Rn.

✓ Bytes : 1.

✓ cycles : 1.

operation:  $(A) \leftarrow (A) \wedge (Rn)$

Function: Logical AND for byte variables.

Flags : None.

ex:- ANL A, R5.

Before execution

A = 39

R5 = 09.

After execution

A = 09

R5 = 09.

0011 1001 ← 39

0000 1001 ← 09

0000 1001 09



2) ANL A, direct

Bytes : 1

Cycles : 1.

operation:  $(A) \leftarrow (A) \wedge (\text{direct})$

Flags : None.

ex:- ANL A, 40h.

Before execution

A = 39

40h = 09

After Execution.

A = 09

40h = 09

39	0011 1001
$\wedge$ 09	0000 1001
<u>09</u>	<u>0000 1001</u>

3) ANL A, @Ri

Ri  $\rightarrow$  R0 or R1

Bytes : 1.

Cycles : 1.

operation:  $(A) \leftarrow (A) \wedge ((R_i))$

Flags : None.

ex:- ANL A, @R0.

Before execution.

A = 39

R0 = 80h.

80h = 09.

After execution.

A = 09

R0 = 80h.

80h = 09.

A = 39	0011 1001
80h = 09	0000 1001
<u>09.</u>	<u>0000 1001.</u>



4) ANL A, #data.

Bytes: 2.

Cycles: 1

Operation:  $(A) \leftarrow (A) \wedge \#data.$

Flags: None.

ex:- ANL A, #09h.

Before execution.

A = 39

After execution.

A = 09

39	0011 1001
$\wedge$ 09	0000 1001
<div style="border: 1px solid black; padding: 2px;">09</div>	<div style="border: 1px solid black; padding: 2px;">0000 1001</div>

5) ANL Direct, A

Bytes: 2

Cycles: 1

Operation:  $(Direct) \leftarrow (Direct) \wedge (A)$

Flags: None

ex:- ANL 40h, A

40h = 39

A = 09

After execution.

40h =

A =

ex:-

ANL PI, #11111110 B; Mask PI.0 (i.e D0 of port 1)

ANL PI, #01111111 B; Mask PI.7 (i.e D7 of port 1)

ANL PI, #11110111 B; Mask PI.3 (i.e D3 of port 1)

ANL PI, #11111100 B; Mask PI.0 & PI.1.

\* ANL C, Source-bit

Function : Logical AND for bit variables.

Flag : CY.

Description : In this instruction carry flag bit is ANDed with a source bit & the result is placed in carry flag.

ie if source bit = 0, then CY = 0 otherwise CY = 1.

6) ANL C, bit

Bytes : 2

Cycles : 2

operation :  $(C) \leftarrow (C) \wedge (\text{bit})$

Flags : CY.

ex's:-

i) ANL C, ACC.7

Before execution.

C = 1

A = 13.

After execution

C = 0

A = 13.

A = 13 = 00010011.

C = 1 = 1

Result C = 00010011.

ii) ANL C, P2.2.

Before execution.

C = 1.

P2 = 00001011 B.

or

P2 = 0Bh.

After execution.

C = 0

P2 = 0Bh.

P2 = 00001011.

C = 1

Result C = 

0
---



\* ANL C, Source-bit

Function : Logical AND for bit variables.

Flag : CY.

Description : In this instruction carry flag bit is ANDed with a source bit & the result is placed in carry flag.

ie if source bit = 0, then CY = 0 otherwise CY = 1.

6) ANL C, bit

Bytes : 2

Cycles : 2

operation :  $(C) \leftarrow (C) \wedge (\text{bit})$

Flags : CY.

ex's:-

i) ANL C, ACC.7

Before execution.

C = 1

A = 13.

After execution

C = 0

A = 13.

A = 13 = 00010011.

C = 1 = 1

Result C = 00010011.

ii) ANL C, P2.2.

Before execution.

C = 1.

P2 = 00001011 B.

or

P2 = 0Bh.

After execution.

C = 0

P2 = 0Bh.

P2 = 00001011.

C = 1

Result C = 

0
---



7) ANL C, 1 bit

NOTE:- 1 bit  $\rightarrow$  means Invert

the bit data  
(NOT)

Bytes: 2

cycles: 2

operation:  $(C) \leftarrow (C) \wedge T(\text{bit})$

Flag: CY.

Description: AND carry bit with inverse of bit data  
 $\uparrow$   
(complement)

ex:- i) ANL C, 1 ACC.7

Before execution

C = 1

Acc = 01111111 B.

or

AC = 7Fh.

After execution

C = 1

ACC = 7Fh.

ACC.7 = 0111111

1ACC.7 = 0111111.

1ACC.7 1111111

$\wedge$  C

C Result 1  $\leftarrow$  C

ii) ANL C, 1 P2.0.

Before execution

C = 1.

P2 = 00010000 B.

or

P2 = 10h.

After execution.

C = 1.

P2 = 10h.

P2.0 = 00010000

1P2.0 = 00010001

1P2.0 . 00010001

$\wedge$  C

C Result C = 1

8) SWAP A

Bytes: 1

Cycles: 1

operation:  $(A_{3-0}) \leftrightarrow (A_{7-4})$

Function: Swap nibbles within the accumulator.

Flags: None

Description: The swap instruction interchanges the lower nibble ( $A_0-A_3$ ) with the upper nibble ( $A_4-A_7$ ) inside register A.

ex:- MOV A, #59H

SWAP A

Before execution

$A = 59H$

ie  $A = 0101\ 1001$

After execution

$A = 95$

ie  $A = 1001\ 0101$

SUM = 0011

0010 0001  
0001 0011



## Logical OR for the byte variables :-

ORL <dest-byte>, <src-byte> or

ORL dest-byte, source-byte.

Function : Logical OR for byte variables.

Description: ORL performs the bitwise logical-OR operation between the indicated variables, storing the results in the destination byte.

(The ORL instruction can be used to set certain bits of an operand to 1).

1) ORL A, Rn.

Bytes : 1

Cycles : 1

Operation:  $(A) \leftarrow (A) \vee (Rn)$

Flag: None

Description: This instruction performs a logical OR on the byte operands, bit by bit, and stores the result in the destination.

Ex: - ORL A, R5

mov A, #32h	Before execution.	After execution.	A = 00110010
mov R4, #50h	A = 32h	A = 72h	R5 = 01010000
ORL A, R4	R5 = 50h	R5 = 50h.	A ← 01110010
			72h



2) ORL A, direct

Bytes: 2

Cycles: 1

operation:  $(A) \leftarrow (A) \vee (\text{direct})$

Flags: None

ex:- ORL A, 30h.

Before execution.

A = 32h.

30h = 50h.

After execution

A = 72h.

30h = 50h.

ie.

MOV A, # 32h.

MOV 30h, # 50h.

ORL A, 30h.

3) ORL A, @Ri

Bytes: 1

Cycles: 1

operation:  $(A) \leftarrow (A) \vee ((Ri))$

Flags: None

$Ri \rightarrow R_0 \text{ or } R_1 \text{ only.}$

ex:- ORL A, @R1

MOV 30h, # 50h.

MOV R1, # 30h

MOV A, # 32h.

ORL A, @R1.

Before execution

A = 32h.

R1 = 30h

30h = 50h.

After execution

A = 72h.

R1 = 30h.

30h = 50h.

4) ORL A, # data.

Bytes: 2

Cycles: 1

operation:  $(A) \leftarrow (A) \vee \# \text{data}$

Flags : None.

ex:- ORL A, #50h.

Before execution.

A = 32h.

data = 50h.

After execution.

A = 72h.

5) ORL direct, A

Bytes : 2

cycles : 1.

operation: (direct)  $\leftarrow$  (direct) V (A)

Flags : None

ex:- ORL 30h, A.

mov A, #32h

mov 30h, #50h

ORL 30h, A

Before execution

A = 32h.

30h = 50h.

After execution.

A = 72h.

30h = 50h.

6) ORL direct, #data.

Bytes : 3.

cycles : 2.

operation: (direct)  $\leftarrow$  (direct) V #data

Flags : None

ex:- ORL 30h, #32h.

mov 30h, #50h

ORL 30h, #32h.

Before execution.

30h = 50h

data = 32h.

After execution

30h = 72h.



#> ORL C, source-bit

Function: logical OR for bit variables.

Description: The carry flag bit is ORed with a source bit and the result is placed in the carry flag.

∴ If the source bit is 1, CY is set;  
otherwise, the CY flag remains unchanged

Flags: CY

\* ORL C, bit

Bytes: 2

Cycles: 2

operation:  $(C) \leftarrow (C) \vee (\text{bit})$ .

Flags: CY

ex:- ORL C, ACC.3

i) Before execution.

$A = FF$ .

$CY = 0$ .

i.e  $A = 11110111$

After execution.

$A = FF$ .

$CY = 1$

ii) Before Execution

$A = F7$

$CY = 0$

$A = 11110111$

After execution.

$A = F7$

$CY = 0$



8) ORL C, 1bit

1bit  $\rightarrow$  NOT of bit or  $\overline{\text{bit}}$

Byte: 2

cycles: 2.

operation:  $(C) \leftarrow (C) \vee (\overline{\text{bit}})$

Flag: CY

ex:- ORL C, 1ACC.3

i) Before execution

A = 00011000 b.

ie A = 18h.

CY = 0

After execution.

CY = 0

A = 00010000  $\overline{\text{bit 3}}$   
+ CY = + 0

Result CY = 0

ii) Before execution.

A = 00010000b

ie A = 10h

CY = 0

After execution

A = 00011000  $\overline{\text{bit 3}}$   
CY = + 0.

CY = 1.

CY = 1

## LOGICAL XOR :-

XRL dest-byte, source-byte

Function: logical exclusive-OR for byte variables.

Flags: None

Description: performs the bitwise logical Exclusive-OR operation b/w the indicated variables, storing the results in the destination.

1) XRL A, Rn

Bytes: 1

Cycles: 1

operation:  $(A) \leftarrow (A) \vee (Rn)$

Flags: None

Ex:- XRL A, R3.

$\vee \rightarrow \text{XOR}$

$Rn \rightarrow R_0 \text{ to } R_7$

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

mov A, #39h

mov R3, #09h

XRL A, R3

Before execution.

A = 39h.

R3 = 09h.

After execution.

$$\begin{array}{rcl} 00111001 & = & A \\ \vee & & \\ 00001001 & = & R \\ \hline 00110000 & = & A \end{array}$$

A = 30h  
R3 = 09h.

2) XRL A, direct.

Bytes: 2

Cycles: 1

operation:  $(A) \leftarrow (A) \vee (\text{direct})$

Flags: None



mov A, #39h

mov 30h, #09h

xrl A, 30h.

Before execution.

A = 39h.

30h = 09h.

After execution.

A = 30h.

30h = 09h.

$$\begin{array}{rcl} A = 39 & = & 0011\ 1001 \\ \vee & & \\ 30h = 09 & = & 0000\ 1001 \\ \hline A = 30 & = & 0011\ 0000 \end{array}$$

3) xrl A, @Ri

$R_i \rightarrow R_1 \text{ or } R_2$

Bytes : 1

Cycles : 1

operation:  $(A) \leftarrow (A) \vee ((R_i))$

Flags : None

Ex:- xrl A, @R1

mov A, #39h.

mov R1, #50h

mov 50h, #09h.

xrl A, @R1.

Before execution.

A = 39h

R1 = 50h

50h = 09h.

After execution.

A = 30h.

4) xrl A, #data.

Bytes : 2

Cycles : 1

operation:  $(A) \leftarrow (A) \vee \#data$

Flags : None

Ex:- XRL A, #09h.

mov A, #39h  
XRL A, #09h.

Before execution.

A = 39h.

data = 09h.

After execution.

A = 30h.

5) XRL direct, A

Bytes : 2

cycles : 1

operation : (direct)  $\leftarrow$  (direct)  $\vee$  (A)

Flags : None

Ex:- XRL 50h, A

mov A, #39h  
mov 50h, #09h.  
XRL 50h, A

Before execution.

A = 39h

50h = 09h.

After execution.

A = 30h.

6) XRL direct, #data.

Bytes : 3.

cycles : 2.

operation : (direct)  $\leftarrow$  (direct)  $\vee$  #data

Flags : None.

Ex:- XRL 50h, #09h.

mov 50h, #39h.  
XRL 50h, #09h.

Before execution

50h = 39h.

data = 09h.

After execution.

50h = 30h.



IE:-

XRL is used to check whether the two register have same value. If the value is same then '0' is placed in accumulator or location (destination).

### ROTATE Instructions :-

1) RL A

Function : Rotate Accumulator left.

Description : The eight bits in the accumulator are rotated one-bit to the left.

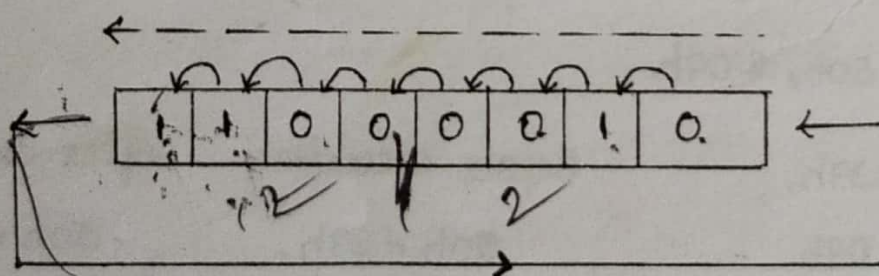
Bit-7 is rotated into the bit-0 position.

Flags : None

Bytes : 1

Cycles : 1

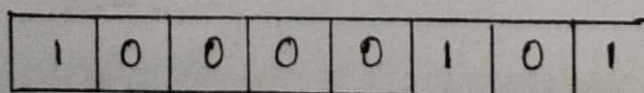
Operation :  $(A_{n+1}) \leftarrow (A_n)$  where  $n=0-6$   
 $(A_0) \leftarrow (A_7)$  } NOT clear



Before Execution.

$A = C2h$

A



After execution.

$A = 85h$

ie  $A = 10000101b$

2) RRA

Function : Rotate Accumulator Right

Bytes : 1

Cycles : 1

Flags : None

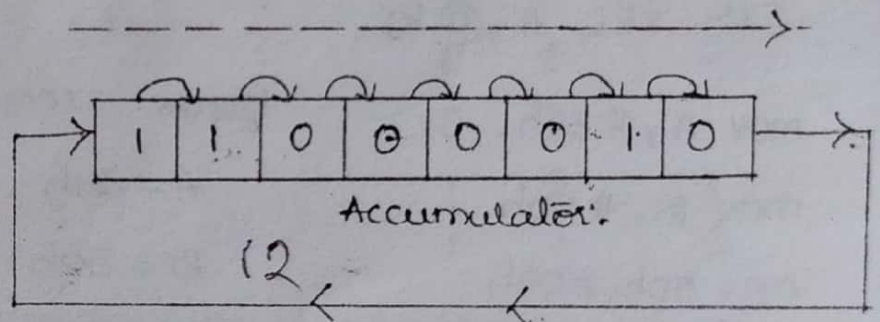
Description : The eight bits in the accumulator are rotated one bit to the right. Bit-0 is rotated into the bit-7 position.

Operation :  $(A_n) \leftarrow (A_{n+1})$ ,  $n=0-6$

$(A_7) \leftarrow (A_0)$

Before Execution :-

20  
0010 0000  
0001 0000



After execution :-

$A = 61h = 01100001$

0 1 1 0 0 0 0 1



July 2012

~~RLC A~~

$eq = 1A$

$cy = 0$

5



Function : Rotate Accumulator left through the carry flag

Flags :  $cy$

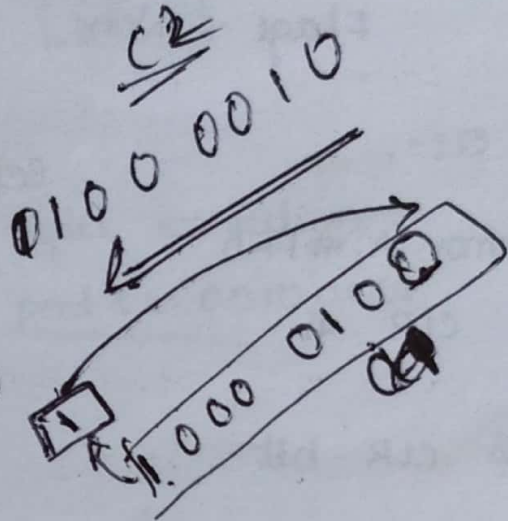
Bytes : 1

Cycles : 4

operation :  $(A_{n+1}) \leftarrow (A_n)$

$(A_0) \leftarrow (C)$

$(C) \leftarrow (A_7)$



Description: The 8-bits in the accumulator & the carry flag are together rotated one bit to the left.

Bit-7 moves into the carry flag.

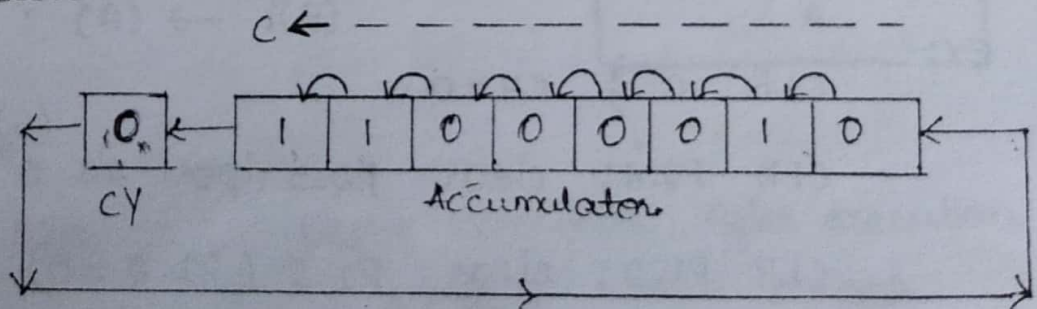
The original state of the carry flag moves into the bit-0 position.

Before execution.

$A = C2h$

$A = 11000010b$

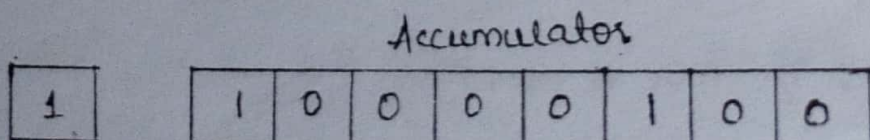
$C = 0$



After execution:-

$A = 84h$

$A = 10000100$



4) RRC A

P →

Function : Rotate accumulator Right through the carry flag.

Byte : 1.

Cycles : 1

Operation :  $(A_n) \leftarrow (A_{n+1})$

$(A_7) \leftarrow (C)$

$(C) \leftarrow (A_0)$

Flags : CY

Description : The 8-bits in the Accumulator and the carry flag are together rotated 1-bit to the right.

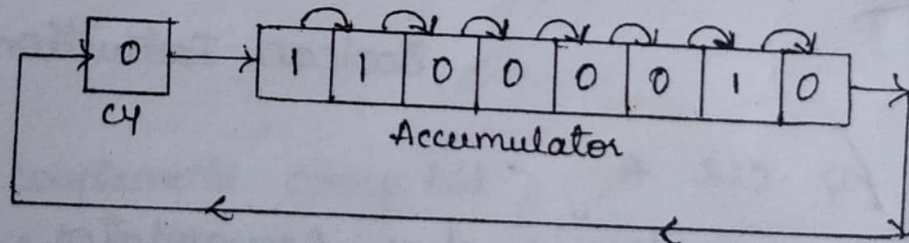
Bit-0 moves into the carry flag.

The original state of the carry flag moves into the bit-7 position.

Before execution:-

$A = C2h = 11000010b$

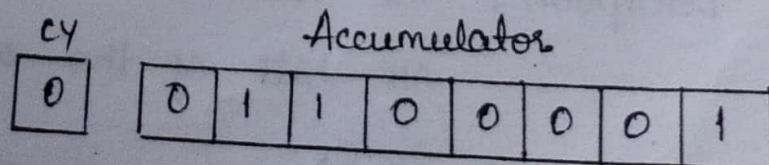
$CY = 0$



After execution:-

$A = 01100001b = 26h$

$CY = 0$





## 5) NOP

Function : No operation.

Flags : None

Description : \* This instruction performs no operation & execution continues with the next instruction.

\* It is sometimes used for timing delays to waste clock cycles.

\* This instruction only updates the program counter (PC) to point to the next instruction following NOP.

Bytes : 1

Cycles : 1

operation :  $(PC) \leftarrow (PC) + 1$

## Boolean Instructions:-

### ✓ 1) CLR A

Function : clear Accumulator

Description : The Accumulator is cleared ( $A = 00h$ )  
All bits of the accumulator are set to 0.

Bytes : 1

Cycles : 1

operation :  $(A) \leftarrow 0$

Flags : None

ex:-

mov A, #FFh

CLR A

Before execution

A = FFh.

After execution.

A = 00h.

2) CLR bit

Function : clear bit

Description : This instruction clears a indicated bit.

\* The bit can be the carry flag, or any (bit-addressable) location in 8051.

Bytes : 1.

Cycles : 1.

operation :  $(C) \leftarrow 0$

Flags : CY.

Ex:-

CLR C ; CY = 0

CLR P2.4; clear P2.5 (port 2's <sup>(6)</sup>5<sup>th</sup> pin is cleared)

CLR P1.2; clear P1.2 (P1.2 = 0)

CLR ACC.7; clear D7 of accumulator (ACC.7 = 0)

ex: 1) mov A, #FFh.

CLR ACC.7

Before execution.

ACC = 11111111

After execution

ACC = 01111111



ii) CLR C ✓

Before execution.

$CY = 1$

After execution.

$CY = 0$

iii) CLR P1.2 ✓

Before execution.

port 1 = 0010 0111

After execution.

port 1 = 0010 0011

3) CPL A ✓

Function : complement Accumulator.

Description : This instruction complements the contents of the Accumulator.

\* The result is 1's complement of the accumulator i.e 0's become 1's & 1's become zero.

Bytes : 1.

cycles : 1.

operation :  $(A) \leftarrow \neg(A)$

$\neg(A) \rightarrow \text{NOT}(A)$

$\frac{01}{A}$

Ex:- CPL A.

i) MOV A, #FFh.

CPL A

Before execution

$A = FFh$

After execution.

$A = 00h$

ii) MOV A, #00h.

CPL A

$A = 00h$

$A = FFh$

4) CPL bit ✓

Bytes : 2.

Cycles : 1

Function : Complement bit

Description : \* This instruction complements a specified single bit.

\* The bit can be any bit addressable location in 8051.

Flags : None.

Ex:- CPL P0.3. ✓

Before execution.

P0.3 = 1 (I/p pin)

After Execution.

P0.3 = 0 (O/p pin).

CPL P3.3. ✓

Before execution.

P3.3 = 0 (O/p pin)

After Execution.

P3.3 = 1 (I/p pin).

5) CPL C ✓

Function : complements carry bit

operation :  $(C) \leftarrow \neg(C)$

Bytes : 1

Cycles : 1

Flags : CY



Before execution.

i)  $CY = 0$

ii)  $CY = 1$

After execution.

i)  $CY = 1$ .

ii)  $CY = 0$ .

6) `SETB C` ✓

Bytes : 1.

Cycles : 1

operation :  $(C) \leftarrow 1$

Function : Set the carry bit

Flag :  $CY$ .

ex:-

Before execution.

i)  $CY = 0$

ii)  $CY = 1$ .

After execution.

i)  $CY = 1$

ii)  $CY = 1$ .

7) `SETB bit` ✓

Bytes : 2

Cycles : 1

operation :  $(bit) \leftarrow 1$ .

Function : Set specified bit.

EX:- `SETB P1.0`

Before execution.

$P1.0 = 0$  (O/p pin)

After execution.

$P1.0 = 1$  (I/p pin).

## JUMP & CALL Instructions

✓ \* Jump & call instructions change the flow of the program by changing the contents of program counter.

2/ ✓ \* A Jump permanently changes the program flow whereas call temporarily changes the program flow to allow another part of the program to run.

3/ ✓ \* Jump instructions are classified into  
1) conditional Jump.  
2) Unconditional Jump.

✓ \* The Jump instruction which changes the program flow if certain condition exists is called conditional Jump.

✓ \* The Jump instruction which changes the program flow irrespective of the condition (NOT depends on any condition) is called Unconditional Jump.



compare instruction:- CINE (compare & Jump if not equal)

This instruction compares the magnitude of the first two operands, & changes program flow if their values are not equal.

The instructions that change the program flow are:-

- \* Jump on bit conditions.
- \* compare byte & Jump if not equal.
- \* Decrement byte & Jump if zero.
- \* Jump unconditionally.
- \* call a subroutine.
- \* Return from a subroutine.

1) JB bit, rel.

(rel  $\rightarrow$  relative address)

Function: Jump if Bit set.

Description: If the indicated bit is a one (1), Jump to the address indicated; otherwise proceed with the next instruction.

The bit tested is not modified.

Bytes : 3

Cycles : 2

operation:  $(PC) \leftarrow (PC) + 3.$

If (bit) = 1

Then,  $(PC) \leftarrow (PC) + \text{rel}.$

Flags : None.

Ex :- Address.

Acc.0 = 1    0300    JB Acc.0, down.  
                 0301  
                 0302  
                 0304. — down : INC PC ←  
                 0305

i) Before execution.      After execution.

PC = 0300h.

ACC.0 = 1

Say ACC.0 = 1.

PC = 0304h.

ii) PC = 0300h.

ACC.0 = 0,

Say ACC.0 = 0.

PC = 0301

NOTE:

i) If condition is false then, processor executes next instruction.

ii) If condition is True. then, processor jumps to the specified address (label) & executes that address instruction.

i) SETB P1.5

iii) JB ACC.0, Next

iv) SETB P1.4

Here : JB P1.5, Here

INC A

Here : JB P1.4, Here

MOV R2, #55h

Next : DEC A

MOV R2, B.



## 2) JNB bit, sel

Function : Jump if bit NOT set.

Description : If the indicated bit is a zero, branch to the indicated address; otherwise proceed with the next instruction.

The bit tested is not modified.

Flag : None

Byte : 3.

Cycles : 2.

operation :  $(PC) \leftarrow (PC) + 3$

If  $(bit) = 0$

then.

$(PC) \leftarrow (PC) + sel.$

Ex:-

Say address

0300

JNB ACC.0, down.

0301

INC R0.

0302

⋮

0303.

down; Dec R1.

Before execution.

i)  $PC = 0300h.$

Say  $ACC.0 = 1.$

After execution.

ii)  $ACC.0 = 1$

$PC = 0301h.$

Here  $ACC.0 = 1$ , the condition is false, so processor executes next instruction ie INC R0.

ii)  $PC = 0300h$ .

$ACC.0 = 0$

say  $ACC.0 = 0$ .

$PC = 0303$ .

Here the condition is True, so processor jumps to the address specified by the label & executes that instruction.

ii)  $SETB\ ACC.0$

iii)  $JNB\ ACC.0, Next$

iv)  $CLR\ A$

here:  $JNB\ ACC.0, here$

$INC\ A$

here:  $JNB\ ACC.0, here$

$MOV\ R2, R3$ .

Next:  $DEC\ A$ .

$MOV\ R2, R3$ .

3)  $JBC\ bit, sel$ .

Function: Jump if bit is set & clear bit!

Description: If the desired bit is high it will ~~on else~~ <sup>(on else)</sup> jump to the target address (specified address i.e. label), at the same time the bit is cleared to zero.

\* If the desired bit is low, then processor proceeds with the next instruction i.e. executes next instruction.

Bytes : 3.

Cycles : 2.

Operation :  $(PC) \leftarrow (PC) + 3$ .

If  $(bit) = 1$

Then  $(bit) \leftarrow 0$ .



$$(PC) \leftarrow (PC) + \text{rel.}$$

ex:-  
0300      SETB   ACC.7  
0301      JBC    ACC.7, Next  
0302      MOV    P1, A  
0303  
 Next:    INC R0.

Before execution.

$$\text{ACC.7} = 1$$

$$PC = 0301.$$

After execution.

$$\text{ACC.7} = 0$$

$$PC = 0303.$$

4) JC target or JC rel

Function : Jump if carry is set i.e.,  $CY = 1$ .

Flags : None.

Description : If the carry flag is set, branch to the address indicated; otherwise proceed with the next instruction.

Bytes : 2.

Cycles : 2.

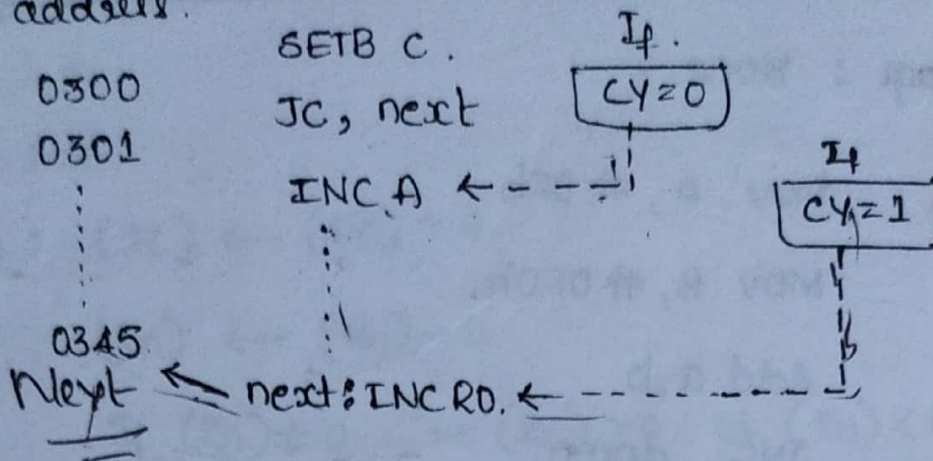
operation :  $(PC) \leftarrow (PC) + 2$

If  $(C) = 1$

then.

$$(PC) \leftarrow (PC) + \text{rel.}$$

Ex:- Say address.



\* If  $CY=1$ , then Jumps to address specified by label.

\* If  $CY=0$ , then executes next instruction.

Before execution

After execution.

$CY=1$

$CY=1$

$PC=0301$

$PC=0345$

5) JNC target or JNC rel.

Function: Jump if No carry ( $CY=0$ )

Description: This instruction checks the carry flag, & if  $CY=0$ , it will Jump to the target address.

Bytes : 2.

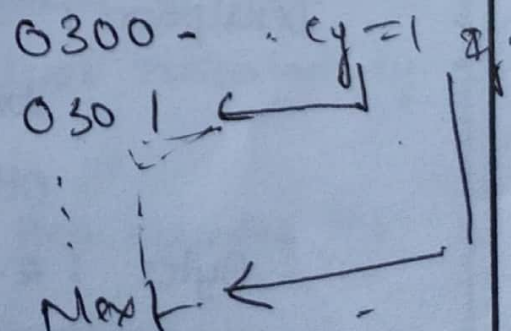
Cycles : 2.

operation:  $PC \leftarrow PC + 2$

If  $CY=0$

Then

$PC \leftarrow PC + rel$





Flags : None

Ex:-      MOV A, #0Fh.  
            MOV B, #0F0h.  
            Add a, b  
            JNC, down.  
            INC RO.  
            Addc a, #30h.  
            JNC, down  
down:    MOV 40h, a.

\* This instruction checks the carry flag, if  $cy=0$  (condition true), then jumps to the specified label. If  $cy=1$  (condition false), then executes next instruction.

6) JZ target or JZ rel

function : Jump if  $A=0$  (Jump if Accumulator zero).

Flags : None

Description : If all bits of the accumulator are zero, branch to the indicated address; otherwise proceed with the next inst.

Bytes : 2.

cycles : 2

operation :  $(PC) \leftarrow (PC) + 2$

If  $A=0$ .

then,  $(PC) \leftarrow (PC) + rel.$

Ex:-    `mov A, #01h.`  
          `mov B, #02h.`  
          `Add A, B.`  
          `JZ , down.`  
          `INC R0.`

down: `mov 40h, A`  
      `end`

\* If  $A=0$ , (condition is True) then Jumps to the address specified by the label down.

ie. `mov 40h, A` in above ex.

\* If  $A \neq 0$ , (condition is false). then proceed with the next instruction.

ie. `INC R0` in above ex.

7) JNZ target or JNZ rel.

Function : Jump if accumulator is NOT zero.

Flags : None.

Description : If any bit of the accumulator is a one, branch to the indicated address; otherwise proceed with the next instruction.

ie, If  $A \neq 0$  (True condition), then branch (Jump) to the indicated address.

If  $A=0$  (false condition), then proceed with the next instruction.



Bytes : 2

cycles : 2.

operation :  $(PC) \leftarrow (PC) + 2$ .

If  $A \neq 0$

Then  $(PC) \leftarrow (PC) + \text{rel}$ .

Ex:- MOV A, #0Fh.

MOV B, #0E0h.

ADD A, B.

JNZ, down.

INC R0

ADDC A, #01h.

JNZ, down.

INC R1.

down: MOV 40h, A.

- \* If  $A \neq 0$  (True condition), then Jump to the address specified by the label down i.e. MOV 40h, A.
- \* If  $A = 0$  (False condition), then proceeds with the next instruction i.e. INC R0 and INC R1.

8) DJNZ byte, target or DJNZ byte, rel - address.

Function : Decrement and Jump if not zero.

Flags : None.

Description : In this instruction a byte is decremented, & if the result is NOT zero it will Jump to the target address.

DJNZ Rn, target (where n=0 to 7)

Bytes : 2

Cycles : 2.

operation :  $(PC) \leftarrow (PC) + 2$

$(Rn) \leftarrow (Rn) - 1$

If  $(Rn) \neq 0$  i.e.  $(Rn) > 0$  or  $(Rn) < 0$ .

Then.

$(PC) \leftarrow (PC) + \text{rel.}$

Ex:-

mov r0, #30h

mov r1, #40h.

mov r3, #05h.

up: mov a, @r0

addc a, #01h

mov @r1, a

INC r1

INC r0.

DJNZ r3, up

end.

\* 1<sup>st</sup> decrements the contents of r3 register & then checks the condition i.e.  $r3 \neq 0$ . If r3 contents is not zero, then jumps to the specified address indicated by the label.

\* If  $r3 = 0$ , (condition is false) then executes the next instruction. i.e. end.



9) DJNZ direct, rel

Bytes : 2

Cycles : 2

Flags : None

operation:  $(PC) \leftarrow (PC) + 2$

$(direct) \leftarrow (direct) - 1$

If  $direct \neq 0$  i.e.  $(direct) > 0$  or  $(direct) < 0$ .

Then.

$(PC) \leftarrow (PC) + rel.$

Ex:- mov 20, #30h.

mov 21, #40h.

mov 40h, #05h

up: mov a, @20

add a, #01h.

mov @21, a

INC 21

INC 20.

DJNZ 40h, up

end.

\* 1<sup>st</sup> decrements the contents of 40h (address), then checks the condition i.e.  $(40) \neq 0$  if contents of 40h address is NOT zero, then jumps to the specified address indicated by the label.

\* If  $(40)h = 0$  (condition is false) then executes the next instruction i.e., end.

NOTE :-

- \* The target address can be no more than 128 bytes backward or 127 bytes forward, since it is a 2-byte instruction.

10) SJMP rel      (OR) SJMP 8-bit address

Function : short Jump

Description : program control branches unconditionally to the address indicated.

Bytes : 2

Cycles : 2.

operation :  $(PC) \leftarrow (PC) + 2$

$(PC) \leftarrow (PC) + rel$

Range : -128 bytes to +127 bytes.

Flags : None.

Ex:-

029 00h.

SJMP OVER

019 30h.

OVER : MOV A, #10h.

029 00h.

SJMP 30h.

019 30h.

11) LJMP 16-bit address.

Function : long Jump

Description : LJMP causes an unconditional branch to the indicated address, by loading the high order & low order bytes of the PC respectively.



Bytes : 3.

Cycles : 2.

operation :  $(PC) \leftarrow \text{address}_{(0-15)}$

Flags : None

Range : -32768 bytes to +32767 bytes.

NOTE :- The destination may therefore be anywhere in the full 64K bytes program address space.

12. JMP @A + DPTR

Function : Jump Indirect.

Description : The JMP instruction is an unconditional Jump to a target address. The target address is provided by the total sum of register A & the DPTR register.

Bytes : 1.

Cycles : 2.

operation :  $(PC) \leftarrow (A) + (DPTR)$

Flags : None.

NOTE : This instruction is not widely used.

\* Jump Instructions are classified into.

1) conditional Jump &

2) Unconditional Jump.

## 1) Conditional Jump:-

\* Depending upon the condition i.e. True or False, program branches (Jumps) to the specified address.  
i.e, If condition is True  $\rightarrow$  then JMP to specified label.  
If condition is false  $\rightarrow$  No Jump. Executes Next instruction.

\* All conditional Jumps are short Jumps, meaning that the target address cannot be more than -128 bytes backward to +127 bytes forward of the PC of the instruction following the Jump.

\* If the target address is beyond the -128 to +127 byte range, the assembler gives an error.

\* If the target address is beyond the -128 to +127 byte range, the assembler gives an error.

## 2) Unconditional Jump:-

Jumps to the specified address without any condition (unconditionally Jumps to the specified address).

The unconditional Jump instructions are

1) SJMP 8-bit address.

2) LJMP 16-bit address.

3) JMP @A+DPTR.



1) SJMP :-

\* This is a 2-byte instruction. The 1<sup>st</sup> byte is the opcode & the second byte is the signed number, which is added to the PC of the instruction following the SJMP to get the target address.

\*  $\therefore$  In this Jump the target address must be within -128 to +127 bytes of the PC of the instruction.

2) LJMP :-

\* This is a 3-byte instruction. The 1<sup>st</sup> byte is the opcode & the next two bytes are the target address.

\* The LJMP is used to Jump to any address location within the 64kbytes code space of 8051.

3) JMP @A + DPTR :-

\* The target address is provided by the total sum of register A & the DPTR register.

\* This instruction is not widely used.

conditional Jumps :-

### 1) PUSH direct :-

Function : Push onto stack.

Description : The stack pointer is incremented by one.

The contents of the indicated variable is then copied into the internal RAM location, addressed by the stack pointer.

Flags : None.

Bytes : 2

Cycles : 2.

Operation :  $(SP) \leftarrow (SP) + 1$

$((SP)) \leftarrow (\text{direct})$

#### NOTE :-

This instruction supports only direct addressing mode.  $\therefore$  The instruction such as "PUSH A" or "PUSH R3" are illegal instructions.

Ex:- i) PUSH 0E0h.

where E0h is the RAM address belonging to register A.

ii) PUSH 03h.

where 03h is the RAM address of R3 of Bank0.



2) POP direct.

Bytes : 2.

cycles : 2.

Function : POP from the stack.

operation : This copies the byte pointed to by SP (stack pointer) to the location where direct address is indicated & decrements SP by 1.

Flags : None.

operation : (direct)  $\leftarrow$  (SP)

(SP)  $\leftarrow$  (SP) - 1.

NOTE:-

This instruction supports only direct addressing mode.  $\therefore$  The instructions such as "POP A" or "POP R3" are illegal instructions.

Ex:- i) POP 0E0h.

where 0E0h is the RAM address belonging to register A.

ii) POP 03h.

where 03h is the RAM address of R3 of Bank 0.

1) ACALL target address.

Function : Absolute call. Transfers control to a Subroutine

Description : \* Acall unconditionally calls a subroutine located at the indicated address. The instruction increments the PC twice to obtain the address of the following instruction, then pushes 16-bit result onto the stack (low order byte 1<sup>st</sup>) & increment the stack pointer to store high-order byte.

\* Acall is a 2-byte instruction, in which 5-bits are used for the opcode & the remaining 11-bits are used for the target Subroutine address.

\* A 11-bit address limits the range to 2K-bytes.

Bytes : 2.

Cycles : 2.

Flags : None.

operation :  $(PC) \leftarrow (PC) + 2$   
 $(SP) \leftarrow (SP) + 1$   
 $(SP) \leftarrow (PC_{7-0})$   
 $(SP) \leftarrow (SP) + 1$   
 $(SP) \leftarrow (PC_{15-8})$   
 $(PC_{10-0}) \leftarrow \text{page address.}$



2) LCALL 16-bit address.

Function : Long call. Transfers control to a Subroutine.

Bytes : 3.

Cycles : 2

Flags : None.

Description : LCALL calls a subroutine located at the indicated address. The instruction adds three to the program counter to generate the address of the next instruction & then pushes the 16-bit result onto the stack (1<sup>st</sup> lower byte) incrementing the stack pointer by 2 & pushes higher byte.

\* The subroutine may  $\therefore$  begin anywhere in the full 64 K-byte program memory address space.

Operation :  $(PC) \leftarrow (PC) + 3.$

$$(SP) \leftarrow (SP) + 1$$
$$((SP)) \leftarrow (PC_{7-0})$$
$$(SP) \leftarrow (SP) + 1$$
$$((SP)) \leftarrow (SP) + 1.$$
$$((SP)) \leftarrow (PC_{15-0})$$
$$(PC) \leftarrow \text{address}_{0-15}.$$

## CALL Instructions :- (Lcall & Acall)

\* call instruction transfers control to a subroutine.

There are two types of call:

1) ACALL &

2) LCALL.

1) ACALL :- \* ACALL is a 2-byte instruction.

\* In Acall, the target address is within 2K bytes of the current program counter (PC).

\* If a subroutine is called, the PC (which has the address of the instruction after the Acall) is pushed onto the stack, & the stack pointer (SP) is incremented by 2.

\* Then the program counter (PC) is loaded with the New address & control is transferred to the subroutine.

\* At the end of the procedure (subroutine), when RET instruction is executed, PC is popped off the stack, which returns control to the instruction after the CALL.

2) LCALL :-

\* LCALL is a 3-byte instruction in which one-byte is the opcode, & the other two bytes are the 16-bit address of the target subroutine.



## ⇒ RET :-

Function : Return from Subroutine.

Description : This instruction is used to return from a subroutine previously entered by insts CALL or ACALL. The top two bytes of the stack are popped in the program counter (PC) & program execution continues at this new address.

\* After popping the top two bytes of the stack into the program counter, the stack pointer (SP) is decremented by 2.

Flag : None

Bytes : 1

Cycles : 2.

Operation :  $(PC_{15-8}) \leftarrow ((SP))$

$(SP) \leftarrow (SP) - 1$

$(PC_{7-0}) \leftarrow ((SP))$

$(SP) \leftarrow (SP) - 1$

## ⇒ RETI :-

Function : Return from interrupt.

Bytes : 1.

Cycles : 2

Flag : None

Description : This is used at the end of an interrupt service routine (Interrupt handler). The top two bytes of the stack are popped into the program counter (PC), the stack pointer (SP) is decremented by 2.

operation :  $(PC_{15-8}) \leftarrow ((SP))$   
 $(SP) \leftarrow (SP) - 1$   
 $(PC_{7-0}) \leftarrow ((SP))$   
 $(SP) \leftarrow (SP) - 1.$

NOTE:

The RET instruction is used to at the end of a subroutine associated with the ACALL & LCALL instructions, RETI must be used for the Interrupt Service Subroutine.

---



## Compare & Jump Instructions

CJNE dest - byte, source - byte, target.

Function : compare and Jump if not equal.

Description : The magnitudes of the source byte & destination byte are compared. If they are not equal, it jumps to the target.

Flag : CY.

1) CJNE A, direct, rel address.

Bytes : 3.

Cycles : 2.

operation:  $(PC) \leftarrow (PC) + 3$

If  $(A) < > (\text{direct})$

then.

$(PC) \leftarrow (PC) + \text{relative address.}$

If  $(A) < (\text{direct})$

Then.

$(C) \leftarrow 1$

Else.

$(C) \leftarrow 0$

Flag : CY.

2) CJNE A, #data, rel address.

Bytes : 3.

Cycles : 2

Flag : CY.

operation :  $(PC) \leftarrow (PC) + 3.$

If  $(A) \neq \text{data}.$

Then.

$(PC) \leftarrow (PC) + \text{relative address}.$

If  $(A) < \text{data}$

Then.

$(C) \leftarrow 1.$

Else

$(C) \leftarrow 0$

3) CJNE  $R_n, \# \text{data}, \text{rel}.$

Bytes : 3.

cycles : 2.

operation:  $(PC) \leftarrow (PC) + 3.$

IF  $(R_n) \neq \text{data}$

THEN

$(PC) \leftarrow (PC) + \text{relative address}$

IF  $(R_n) < \text{data}.$

THEN

$(C) \leftarrow 1$

ELSE

$(C) \leftarrow 0$

Flags : cy.



4) CJNE @ Ri, #data, rel.

Bytes : 3

Cycles : 2.

Flags : CY

operation :  $(PC) \leftarrow (PC) + 3$

IF  $((Ri)) < > \text{data}$

THEN

$(PC) \leftarrow (PC) + \text{relative address}$

IF  $(Ri) < \text{data}$

Then.

$(C) \leftarrow 1.$

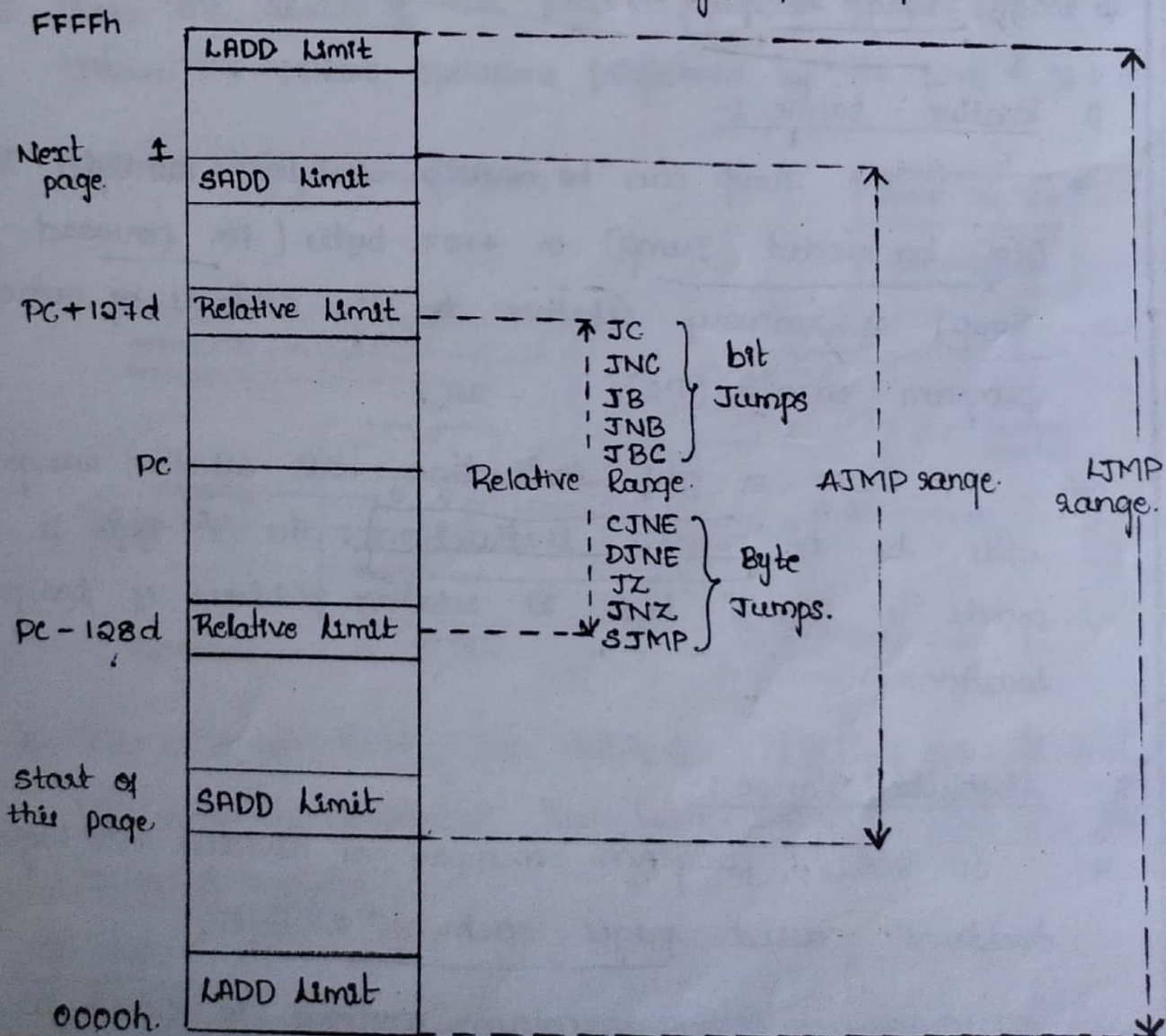
Else

$(C) \leftarrow 0.$

## JUMP & CALL INSTRUCTIONS :-

\* Jump & call instructions replaces the contents of program counter (PC) with New address & program execution to start from that new address.

\* The difference (in bytes) of this new address from address in program where Jump or call instruction is called range of Jump or call.





## fig ① Jump Instruction Range.

SADD  $\rightarrow$  short address.

LADD  $\rightarrow$  long address.

\* Jump or call instructions may have one of the

3 ranges:-

i) Relative range -  $+127d$  to  $-128d$ . ✓

ii) Absolute range - within a page (2Kbyte) ✓

iii) Long range -  $0000h$  to  $FFFFh$ .

i) Relative range :-

\* The Jump can be written within  $-128$  bytes (for backward jump) or  $+127$  bytes (for forward jump) of memory relative to the address of current program counter (PC).

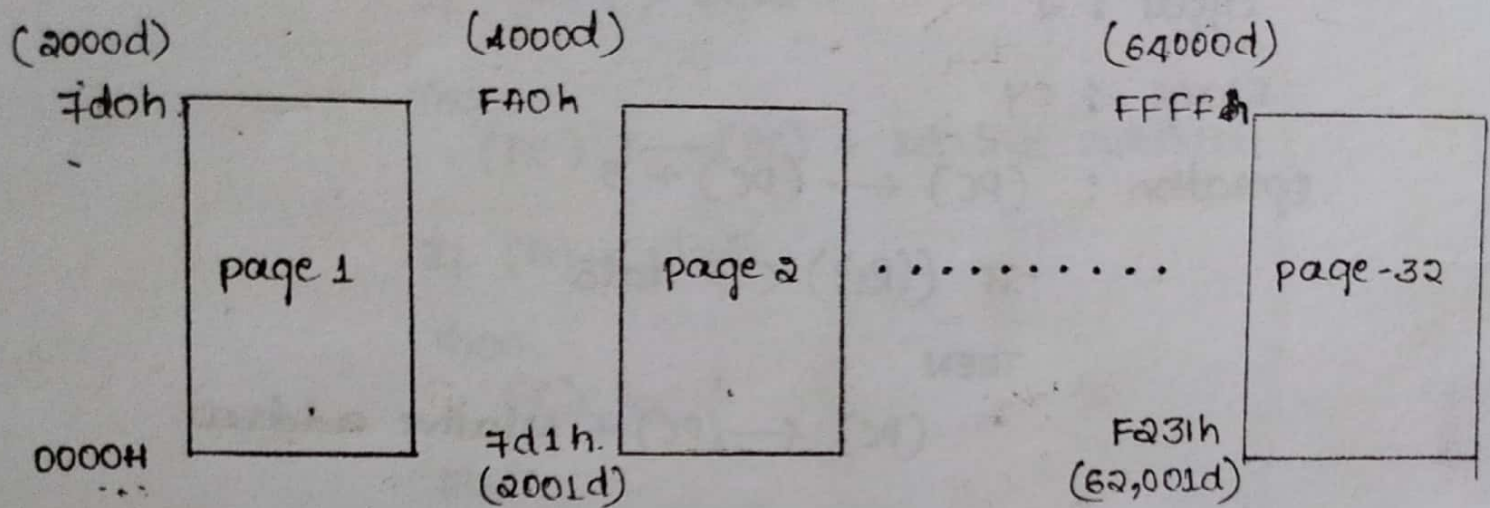
\* Jump or call instruction with relative range will be of 2-byte instructions. The 1<sup>st</sup> byte is opcode & second byte is relative address of target location.

ii) Absolute Range :-

\* In 8051, program memory is divided into logical divisions called pages each of 2Kbyte.

\* Maximum size program memory is 64K bytes.  
Size of each page is 2Kbytes.

∴ Maximum number of pages =  $\frac{64 \text{ Kb}}{2 \text{ Kb}} = 32 \text{ pages}$ .



\* In absolute range, Jump can be within a single page.

\* The upper 5-bits of PC holds the page number & lower 11-bits holds the address within that page.

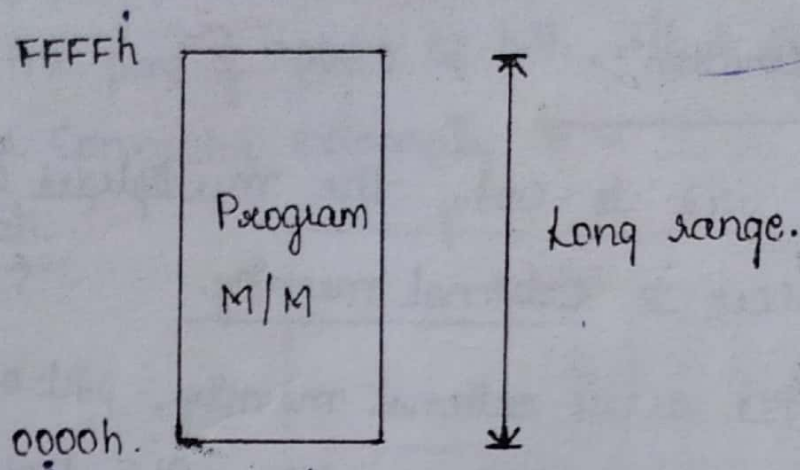
i.e.,  $2^5 \rightarrow 32 \text{ page}$ .

$2^{11} \rightarrow 2 \text{ Kb range}$

← 16-bit PC →



### iii) Long Absolute Range :-



- \* This range allows the Jump to any where in the memory location from 0000h to FFFFh.
- \* The Jump or call instructions with this range will be of 3 byte instructions in which 1<sup>st</sup> byte is opcode & 2<sup>nd</sup> & 3<sup>rd</sup> bytes represents the 16-bit address of target location.

Type of Jump or CALL	Range	No of bytes	example.
Relative Range	-128d to +127d.	2-byte instructions.	JC, JNC, JB, JNB, JBC, JZ, JNZ, DJNZ, CJNE.
Absolute range	Within a page (2K byte)	2-byte instructions.	ACALL.
Long Range.	Anywhere within program memory (0-FFFFh)	3-byte instructions.	LCALL

## Subroutine:-

Subroutine is a program that may be used many times in the execution of a larger program.

(OR)

Subroutine are the programs that are often used to perform tasks that need to be performed frequently.

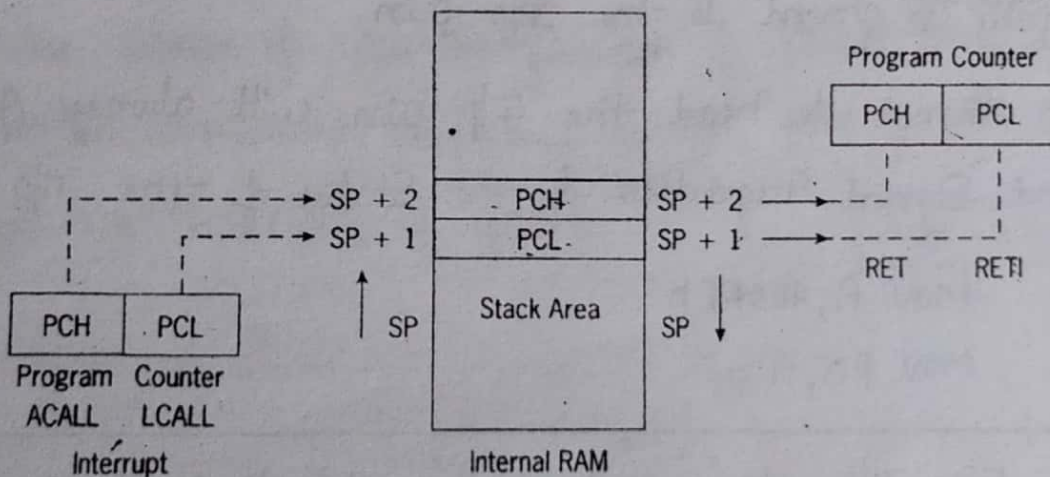
- \* With the relevant figure, write a Sequence of events that occur in 8051 microcontroller when the CALL & RET instructions are executed

Jan-10, 6M

- \* Explain with a neat diagram, the Significance of Stack memory, whenever a CALL instruction is executed by the 8051  $\mu C$ .

June-08, 5M

June-07, 5M



When call instruction is executed the following Sequence of events occurs:

- 1) The return address of the next instruction after the call instruction will be in the program counter (PC).
- 2) The return address are pushed onto the Stack i.e 1<sup>st</sup> lower



byte then higher byte.

- 3) The Stack pointer is incremented for each push on the stack (thus Sp is incremented by 2)
- 4) Then pc fetches the Subroutine address.
- 5) The Subroutine is executed.
- 6) After executing RET instruction at the end of the Subroutine, the return address is popped from the stack & restored in pc.
- 7) Then Stack pointer is decremented for each pop (thus Sp is decremented by 2).

\* Differentiate between Jump & call Instructions. July-06, 4M

Sl No	Jump Instruction	Call Instruction
1	Jump Instructions permanently changes the program flow	Call Instructions temporarily changes the program flow
2	Jump Instructions won't store return address on stack.	Call Instructions store the return address on stack.
3	Jump Instructions branches (Jump) to the target address	Call Instructions are used to call a Subroutine.
4	Conditional Jump Instructions are available	Conditional Call Instructions are NOT available.
5	Jump ranges <ol style="list-style-type: none"> <li>i) Relative <math>\rightarrow -128d</math> to <math>+127d</math></li> <li>ii) Absolute <math>\rightarrow</math> Within a page (2Kb)</li> <li>iii) Long <math>\rightarrow</math> Anywhere within 64K byte</li> </ol>	Call ranges <ol style="list-style-type: none"> <li>i) Acall <math>\rightarrow</math> within a page (2Kb)</li> <li>ii) Lcall <math>\rightarrow</math> Anywhere within 64Kb.</li> </ol>



6>	ex:- SJMP, JNC, JNZ, JZ, JC JB etc	ex:- ACall Lcall
----	--	------------------------

\* Differentiate between Conditional Jump & unconditional Jump Inst's.

Sl No	Conditional Jump	unconditional Jump
1>	All Conditional Jumps are Short Jumps i.e. -128d to +127d	Unconditional Jump may be LJMP & SJMP
2>	The Conditional Jump Inst's Changes the program flow if certain Condition exists	The unconditional Jump Inst's Changes the program flow irrespective of the Condition i.e. NOT depends on any condition
3>	ex:- JNZ, JZ, JNC, JC, JB etc	ex:- SJMP, LJMP.

\* Specify the memory area for bit level logical Instructions used in 8051 & list bit level logical Inst's. June-09, 5M

\* 20h to 2Fh is the memory area used for bit level logical Inst's

\* List of bit level logical Inst's are:

ANL C, bit      RL C, bit      CPL C

ANL C, /bit      RL C, /bit



## Important Instructions :-

### 1) SWAP & SWAP A :-

Function : Swap nibbles within the Accumulator.

Description : The Swap instruction interchanges the lower nibble ( $A_0-A_3$ ) with the upper nibble ( $A_4-A_7$ ) inside the Register A.

Bytes : 1

Cycles operation : 1  
operation :  $(A_{0-3}) \leftrightarrow (A_{4-7})$

Flags affected : None

Eg :-

MOV A, #59H

SWAP A

Before Execution	After Execution
A = 59H	A = 95H
ie. A = 01011001	ie. A = 10010101

### 2) XCHG :- XCH A, Byte

Function : Exchange Accumulator (A) with byte variable.

Description : This instruction Swaps (exchange) the contents of Register A & the Source byte. The Source byte can be any register & RAM location.

Bytes : 1  
 Cycles : 1  
 operation :  $(A) \longleftrightarrow (\text{Byte})$   
 Flags affected : None

Eg:-

i) XCH A, R<sub>n</sub>

mov A, #0FFh  
 mov R<sub>2</sub>, #11h  
 XCH A, R<sub>2</sub>

Before Execution	After Execution
A = FFh	A = 11h
R <sub>2</sub> = 11h	R <sub>2</sub> = FFh

ii) XCH A, direct

mov A, #0FFh  
 mov 40h, #11h  
 XCH A, 40h

Before Execution	After Execution
A = FFh	A = 11h
R <sub>2</sub> = 11h	R <sub>2</sub> = FFh

3) DAA :-

Function : Decimal - adjust accumulator after addition

Description : This instruction is used after addition of BCD - numbers to convert the result back to BCD.



The data is adjusted in the following two possible cases:

1) It adds 6 to the lower 4-bits of accumulator if it is greater than 9 or if  $AC=1$

2) It also adds 6 to the upper 4-bits of accumulator if it is greater than 9 or if  $CY=1$ .

Bytes : 1

$V \rightarrow 0h$

Cycles : 1

operation: IF

$(A_{3-0}) > 9 \vee AC=1$ , then  $(A_{3-0}) \leftarrow (A_{3-0}) + 6$

IF

$(A_{7-4}) > 9 \vee CY=1$ , then  $(A_{7-4}) \leftarrow (A_{7-4}) + 6$

Flags affected :  $CY, AC$ .

Eg:- 1)  $mov\ A, \#47h$   
 $ADD\ A, \#38h$   
 $DAA$

47 h
+ 38 h
-----
1
7F h
6
-----
85 h

ii)

55 h
+ 68 h
-----
Bd h

Here  $B > 9$  &  $d > 9$ , So add 6 to each digit

1
Bd h
66 h
-----
(1) 23 h

$CY=1$

4) XCHD A, @R<sub>p</sub> (OR) XCHD A, @R<sub>i</sub>

Function : Exchange digit

Description : The XCHD instruction exchanges only the lower nibble of accumulator (A<sub>3-0</sub>), with the lower nibble of the RAM location pointed by R<sub>p</sub> register.

The higher-order nibble (bits 7-4) of each registers are not affected.

Bytes : 1

Cycles : 1

operation :  $(A_{3-0}) \longleftrightarrow (R_p(3-0))$

Flags  
affected : None

Eg:-

mov A, #0FFh

mov R<sub>1</sub>, #50h

mov 50h, #00h

XCHD A, @R<sub>1</sub>

Before Execution	After Execution
A = FFh	A = F0h
R <sub>1</sub> = 50h	R <sub>1</sub> = 50h
50h = 00h	50h = 0Fh



## 5) MUL AB :-

Function : Multiply  $\rightarrow (A \times B)$

Description : The 8-bit Contents of A-register is multiplied with the 8-bit Contents of B-register.

The 8-bit multiplication results in 16-bit result.

After multiplication, the lower byte of the result is available in A-register & higher byte of the result is available in B-register.

Bytes : 1

Cycles : 4

operation :  $(A) \times (B) \rightarrow \begin{cases} \text{Higher byte} \uparrow \text{ in A-register} \\ \text{Lower byte} \uparrow \text{ in B-register} \end{cases}$   
of result

Flags affected : CY, OV.

Eg:-

mov A, #05h

mov B, #07h

MUL AB

$$\begin{array}{r} 05 \times 07h \\ \hline 00, 23h \\ \hline B \quad A \end{array}$$

Before Execution	After Execution
A = 05h B = 07h	A = 23h B = 00h

## 6) SUBB A, Src :-

Function : Subtract with borrow

Description : SUBB Subtracts the Source byte & the Carry Flag from the accumulator & puts the result in the accumulator.

Bytes : 1

Cycles : 1

operation :  $(A) = (A) - (\text{byte}) - (C)$

Flags affected : CY, AC, OV

Subb instruction Sets the Carry Flag according to the following

- i) destination byte > Source byte  $\rightarrow$  CY=0, result is +ve
- ii) destination byte = Source byte  $\rightarrow$  CY=0, result is 0
- iii) destination byte < Source byte  $\rightarrow$  CY=1, result is -ve & in 2's Complement.

Eg :-

SUBB A, #data  $\rightarrow$  SUBB A, #25h  $\rightarrow (A) = (A) - (25h) - (CY)$

SUBB A, R<sub>n</sub>  $\rightarrow$  SUBB A, R<sub>0</sub>  $\rightarrow (A) = (A) - (R_0) - (CY)$

SUBB A, direct  $\rightarrow$  SUBB A, 50h  $\rightarrow (A) = (A) - (50h) - (CY)$

SUBB A, @R<sub>1</sub>  $\rightarrow$  SUBB A, @R<sub>0</sub>  $\rightarrow (A) = (A) - ((R_0)) - (CY)$



7) MOV C, b :- 81 MOV C, bit 81 MOV dest-bit, Source-bit

Function : Move bit data from Source bit to destination bit.  
one of the operands must be the Carry Flag; the other may be any directly addressable bit.

Description: The Single bit is moved to the Carry Flag.

Bytes : 2

Cycles : 1

operations :  $(C) \leftarrow (\text{bit})$

Flags affected : CY

Eg :-

i) MOV C, P1.4

Before Execution	After Execution
C = 'X' P1.4 = 1	C = 1 P1.4 = 1

ii) MOV C, P1.4

Before Execution	After Execution
C = 'X' P1.4 = 0	C = 0 P1.4 = 0

## 8) DIV AB :-

Function : Divide  $\rightarrow (A)/(B)$

Description : Div AB divides the Contents of accumulator by the Contents of B-register.

The accumulator receives the quotient & B-register receives the remainder.

Bytes : 1

Cycles : 4

operation :  $(A)/(B) \rightarrow \begin{cases} A = \text{quotient} \\ B = \text{Remainder} \end{cases}$

Flags affected : CY, OV

Eg:-  
mov A, #0FBh  
mov B, #12h  
DIV AB

FBh  $\rightarrow$  251d

12h  $\rightarrow$  18d

$$\begin{array}{r} 13 \rightarrow (A) \\ 18 \overline{) 251} \\ \underline{234} \\ 17 \rightarrow (B) \end{array}$$

Before Execution	After Execution
A = FBh (251d)	A = 0dh (13d)
B = 12h (18d)	B = 11h (17d)

After executing the instruction, the Quotient is Stored in accumulator & the remainder is Stored in B-register.



9) MOVX :-

MOVX dest-byte, Source-byte

Function : MOVE external

Description : This instruction transfers data between external memory & register A, hence the 'X' is appended to MOV.

The address of external memory location being accessed can be 16-bit & 8-bit.

Bytes : 1

Cycles : 2

operation :  $(A) \leftarrow ((R_i))$

Flags affected : None

Eg:-  $MOVX A, @R_0$  &  $MOVX A, @R_1$  Where,  
 $R_i = R_0 \text{ & } R_1$

i)  $MOVX A, @R_0$

mov R<sub>0</sub>, #50h

mov 50h, #FFh

movx A, @R<sub>0</sub>

Before Execution	After Execution
A = 'xx' R <sub>0</sub> = 50h 50h = FFh	A = FFh R <sub>0</sub> = 50h 50h = FFh

ii)  $MOVX A, @DPTR$

mov DPTR, #1234h

mov 1234h, #0FFh

movx A, @DPTR

Before Execution	After Execution
A = 'xx' 1234h = FFh DPTR = 1234h	A = FFh 1234h = FFh DPTR = 1234h



## 10) MOVc

Function : Move Code

Description : This instruction loads the accumulator with a code byte or constant from program memory.

Bytes : 1

Cycles : 2

Flags affected : None

Eg :-  
 1) MOVc A, @A+DPTR  
 2) MOVc A, @A+PC

1) MOVc A, @A+DPTR

Before Execution	After Execution
A = 00h	A = FFh
DPTR = 1234h	DPTR = 1234h
1234h = FFh	1234h = FFh

Code memory address

Data

## 11) SETB :-

i) SETB C

Function : Sets the carry flag bit

Bytes : 1

Cycles : 1

Operation : (C) ← 1

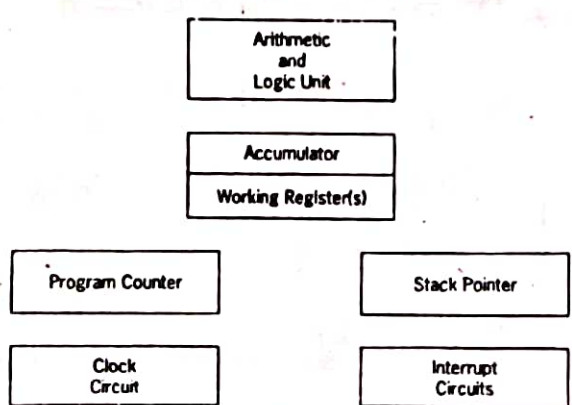
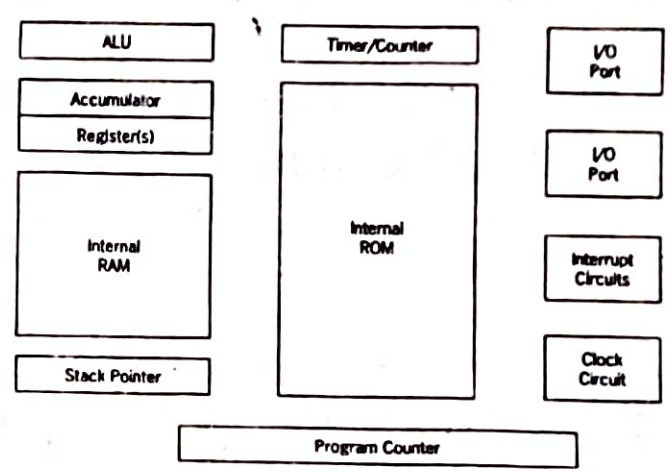
Flags affected : CY

Before Execution	After Execution
i) CY = 0	i) CY = 1
ii) CY = 1	ii) CY = 1



## Microcontroller

- 1> Differentiate between a microprocessor & a microcontroller Jan-09, 6M
- 2> Give the comparison b/w microprocessor & microcontroller Model, 6M
- 3> Bring out the architectural difference b/w a microprocessor & a microcontroller. Jan 06, 6M

Sl No	Microprocessor	Microcontroller
1>	 <p>The diagram shows the internal components of a microprocessor. At the top is the 'Arithmetic and Logic Unit'. Below it are the 'Accumulator' and 'Working Register(s)'. To the left are the 'Program Counter' and 'Clock Circuit'. To the right are the 'Stack Pointer' and 'Interrupt Circuits'.</p> <p>Fig : Block diagram of Microprocessor</p>	 <p>The diagram shows the internal components of a microcontroller. It includes the 'ALU', 'Accumulator', and 'Register(s)' at the top left. Below them is 'Internal RAM' and a 'Stack Pointer'. To the right is a large 'Internal ROM' block, with a 'Timer/Counter' above it and a 'Program Counter' below it. On the far right, there are two 'I/O Port' blocks, 'Interrupt Circuits', and a 'Clock Circuit'.</p> <p>Fig: Block diagram of Microcontroller</p>
2>	<p>Microprocessor Contains ALU, general purpose registers, Stack pointer, Program Counter, Clock timing Ckt &amp; Interrupt Ckt</p>	<p>Microcontroller Contains the circuitry of microprocessor &amp; in addition it has built in ROM, RAM, I/O devices, timers &amp; counters.</p>

3) It has many instructions to move data b/w memory & CPU.

4) It has one or two bit handling instructions.

5) Less number of pins are multifunctioned.

6) It has single memory map for data & code (program).

7) Access time for memory & I/O devices are more.

8) Microprocessor based system requires more hardware.

9) Microprocessor based system is more flexible in design point of view.

(CR)

Designer can decide the amount of ROM, RAM etc., to be connected.

Expensive applications

volatile

It has one or two instructions to move data b/w memory & CPU.

It has many bit handling instructions.

{ex:- CLRC, SETB P1.0 etc.,}

More number of pins are multifunctioned.

It has separate memory map for data & code (program).

Less access time for built-in memory & I/O devices.

Microcontroller based system requires less hardware reducing PCB size & increasing the reliability.

Less flexible in design point of view.

(OR)

Fixed amount of ROM, RAM, I/O ports on chip.

Applications in which cost, space & power are critical.

Not volatile.



12)	Large number of instructions with flexible addressing modes	Limited number of instructions with few addressing modes.
-----	---	---

## Computer architecture (OP) processor architectures :- June 2012

\* Every processor needs to store the Code (instructions) & also the data. Depending on how these are stored in memory & how the memory is accessed, the processor architectures are classified into

- 1) VON-NEUMANN & Princeton architecture
- 2) HARVARD architecture.

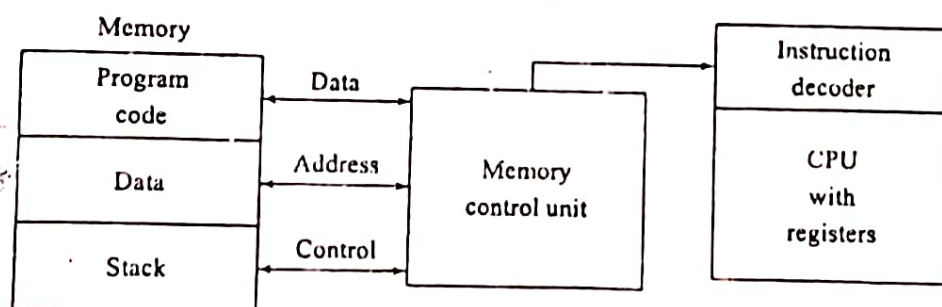


Figure Block diagram of Von Neumann Architecture.

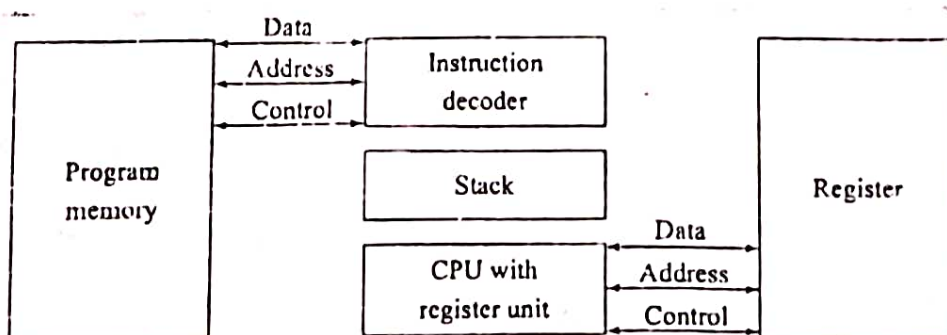


Figure Block diagram of Harvard Architecture.

1) Distinguish Harvard & Princeton architecture with diagrams.

Jan-01, 611

2) Explain the difference b/w Harvard & von-Neumann architecture.

Jan-01, 4M

Sl NO	Von-Neumann & Princeton architecture.	Harvard architecture
1)	Block diagram	Block diagram.
2)	It uses Single memory Space for both Instructions & data. It is also Called Stored program Computer.	It has Separate program - memory & data memory.
3)	It is not possible to Fetch the instruction Code & data.	Instruction Code & data can be Fetched Simultaneously
4)	Execution of Instruction takes more Instruction (machine) cycle	Execution of Instruction take less instruction (machine) cycle.
5)	uses CISC processor	uses RISC processor.
6)	Main Feature is "pre-fetching"	Main Feature is instruction "parallelism"
7)	The Computer based on Princeton architecture are also known as control flow & control driven Computers.	They are also called as data flow & data driven processors.



Sl No	RISC	CISC
4	Multiple Register Set	Single Register Set
5	Few addressing modes & most instructions have register to register addressing mode.	Many addressing modes.
6	Fixed format instructions	variable format instructions
7	Highly pipelined	Less pipelined
8	Conditional Jump can be based on a bit any where in memory.	Conditional Jump is usually based on the Status Register bit.
9	Complexity is in the <u>Compiler</u>	Complexity is in the <u>micro-program</u> .
10	Complex addressing modes are Synthesized in Software	Supports complex addressing modes.
11	eg:- <u>PIC Microcontroller Series</u>	eg:- 8085, 8086, MC6800, Z-80 & 8051 microcontroller.

### Microcontroller:-

Microcontroller contains the circuitry of microprocessor & in addition it has built in ROM, RAM, I/O devices, timers & counters.

### Microcontroller Survey:-

#### i) 4-bit microcontroller:-

- \* CPU can handle only 4-bit of data at a time.
- \* These microcontroller are introduced 1<sup>st</sup> & are still used in very small appliances.

#### Applications :- In Toys.

#### Eg :-

Sl No	Manufacturer	Model	RAM	ROM
1	Hitachi	HMCS40	32-bytes	512 bytes
2	Toshiba	TLCS47	128 bytes	2K-bytes



### ii) 8-bit Microcontroller :-

- \* CPU can handle 8-bits at a time
- \* 8-bit Size of data is proven to be very useful data size - because ASCII data is stored in 8-bit format. This makes 8-bits a good choice for data communication.
- \* Most of memory IC's are changed in 8-bit configuration, which can be interfaced easily to data buses of 8-bits.

### Applications :-

Variety of applications that involve limited calculations & simple control operations such as Washing machines, TV etc.,

### Eg :-

Manufacturer	Model	No. of pins	RAM	ROM
Intel	8051	40	128	4K-byte
Intel	8052	40	256	8K-byte

### iii) 16-bit Microcontroller :-

- \* CPU can handle only two bytes at a time.
- \* Designed for high Speed / high performance applications. These provide large program & data memory spaces for more flexible I/O capabilities, greater Speed & less cost than any previous microcontrollers

Manufacturer	Model	No. of Pins	RAM	ROM
Intel	80C196	64	1-Kbytes	32-Kbytes
Hitachi	H8/532	84	232-Kbytes	8-Kbytes

### 32-bit MicroController :-

- \* CPU can handle 32-bit data at a time
- \* Designed for applications such as Robotics Control, highly intelligent instrumentation, image processing & other high end control systems.

eg:- Intel 80960  
ARM processors.



\* What Criteria do designers consider in choosing microcontroller?

July-08, 6M

There are a wide variety of microcontrollers available in the market. Program written for one (microcontroller) will not run on others. The choice of the microcontroller is determined by three parameters:

- 1) It must perform the required task efficiently & effectively  
i.e.
  - i) Speed
  - ii) Amount of RAM & ROM on chip
  - iii) Power Consumption
  - iv) The number of I/O pins & the timer on the chip
  - v) Cost per unit
  - vi) Ease of upgrading
  - vii) Packaging - The number of pins & the packaging format.

This determines the required space & assembly layout.

2) Availability of Software development tools such as Compilers, assemblers & debuggers.

3) Availability & reliable source for the microcontroller.

Jan-09, 6M

Jan-07, 5M

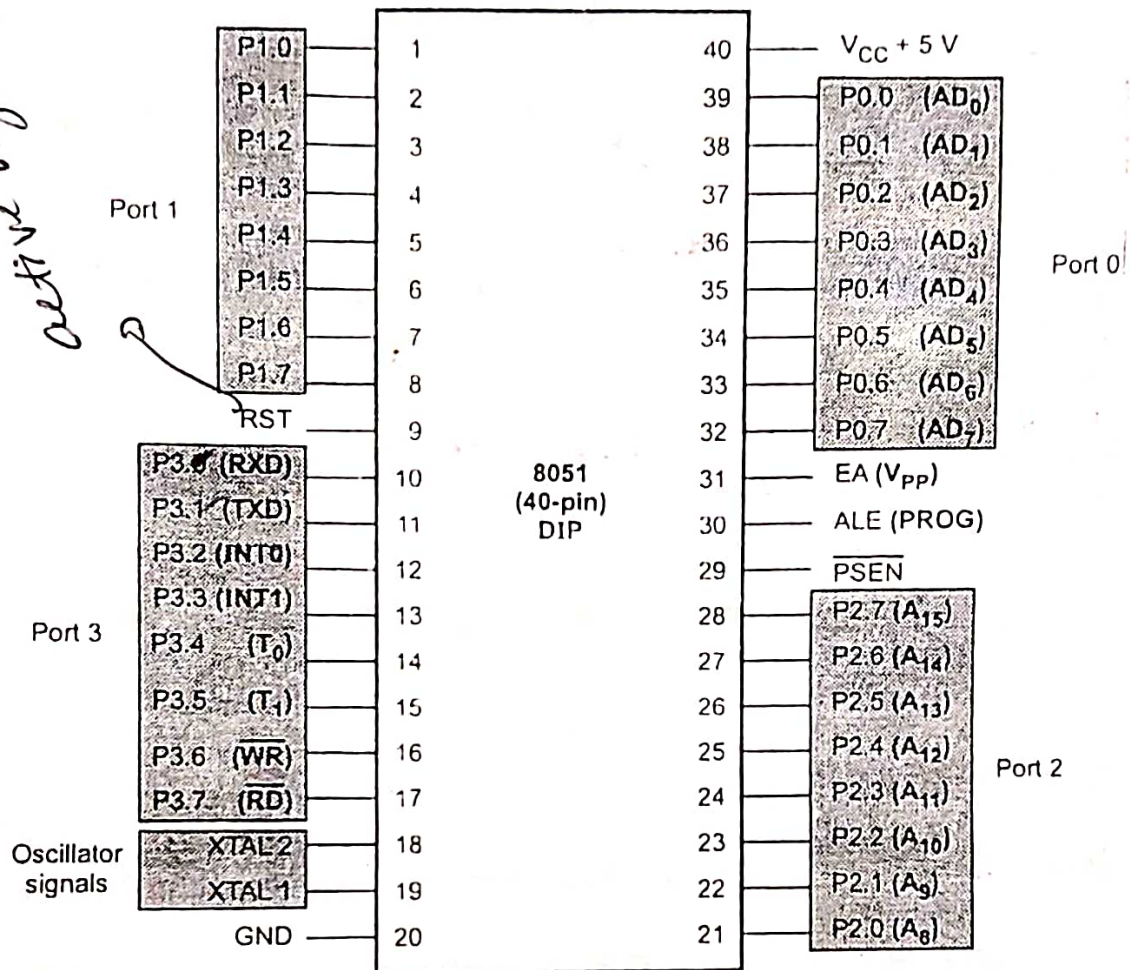
- 1) List the Salient Features of 8051 microcontroller
- 2) List the Specific Features of 8051 microcontroller

The Salient Features of 8051 microcontroller are :-

- 1) 8-bit CPU
- 2) Internal ROM of 4-Kbytes
- 3) Internal RAM of 128-bytes
- 4) 32-I/O pins
- 5) Two 16-bit Timers/Counters ( $T_0$  &  $T_1$ )
- 6) 8-bit Stack pointer (SP)
- 7) 8-bit Program Status Word (PSW)
- 8) 16-bit Program Counter (PC) & Data pointer (DPTR)
- 9) 6 Interrupt Sources with priority levels
- 10) Full duplex Serial data Transmitter/Receiver
- 11) on-chip oscillator circuits



~~18th June 2012~~



Pins 1-8 : Port 1 :-

Each of these pins can be configured as I/P or O/P pins.

Pin 9: RST (Reset) :-

It is an active high signal. When a pulse (Square wave) is applied to this pin, microcontroller will terminate all its activities & Reset.

Program Counter is loaded with 0000.

Pins 10-17 : Port 3 :-

Each of these pins can be configured as I/P or O/P pins.

Pins 10 & 11: RXD & TXD :-

8051 has Serial data Communication circuits that use 'SBUF' register to hold the data & 'SCON' register to control the data communication.

\* The data is Transmitted out of 8051 through the TXD line.

\* The data is Received by 8051 through the RXD line.

Pin 12: INT0 :-

Pin 13: INT1 :-

} Interrupt 0 & Interrupt 1 are two interrupt pins that are triggered by external circuits.

Pins 14 & 15: T<sub>0</sub> & T<sub>1</sub> :- The 8051 has two 16-bit Timers/Counters.

T<sub>0</sub> → Timer 0 register (16-bit)

T<sub>1</sub> → Timer 1 register (16-bit)



\* They can be used either as Timers to generate a time delay or as Counters to count events happening outside the microcontroller.

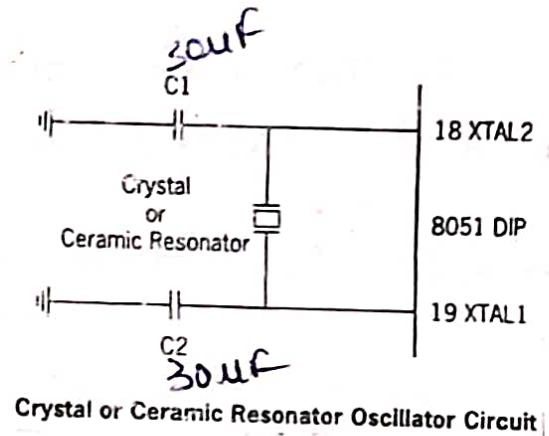
Each 16-bit registers can be accessed as two separate 8-bit registers.

Pins 16 & 17:  $\overline{RD}$  &  $\overline{WR}$  :- These are active low pins.

When  $\overline{RD} = 0$ , microcontroller reads the data from external RAM.

When  $\overline{WR} = 0$ , microcontroller writes the data into external RAM.

Pins 18 & 19: XTAL2 & XTAL1 :-



\* The 8051 has an on-chip oscillator but requires an external clock to run it. A Quartz Crystal oscillator is connected to I/p's XTAL1 & XTAL2 with two capacitors having values 30pF.

\* If an external frequency (from AFO) have to be applied, then it must be applied b/w XTAL1 & GND. XTAL2 must be left open.

\* oscillator frequency may vary from 10MHz to 40MHz.

Pin 20: VSS :- It is a ground pin.

Pins 21-28: Port 2 :-

If external memory is not used, these pins can be used as I/p's & o/p's.

\* If external memory is used then the higher address i.e.  $A_8 - A_{15}$  will appear on this port.

Pin 29:  $\overline{\text{PSEN}}$  (Program Store Enable) :-

If the memory access is for the byte of program code in the ROM, then  $\overline{\text{PSEN}}$  signal goes low & the data byte from the ROM is placed on the data bus.

Pin 30: ALE (Address Latch Enable) :-

When  $\text{ALE} = 1$ , port 0 is providing lower order address. ( $A_0 - A_7$ )  
When  $\text{ALE} = 0$ , port 0 is used as data lines.

Pin 31:  $\overline{\text{EA}}$  (External Access) :-

This pin is used whenever external memory is used.

\* When  $\overline{\text{EA}}$  is connected to VCC i.e.  $\overline{\text{EA}} = 1$ , code is stored in internal ROM. The program is fetched from address location 0000 to 0FFFh.

\* When  $\overline{\text{EA}}$  is connected to GND i.e.  $\overline{\text{EA}} = 0$ , code is stored in external ROM. The program is fetched from address location 0000 to FFFFh.

Pins 32-39: Port 0 :-

If external memory is not used, then these pins can be used as I/p's & o/p's.



\* If external memory is used then the lower address i.e.  $A_0 - A_7$  will appear on this port.

Pin 40: VCC :-

DC power Supply +5V is connected to this pin.

1) Explain the function of the following pins of 8051

i)  $\overline{EA}$  ii) ALE iii) RST iv)  $\overline{PSEN}$

June-07, 8M

8051 Microcontroller :-

It is a 8-bit microcontroller which has got RAM, ROM, 2 timers, 1 Serial port & 4 other ports on a Single Chip.

Features of 8051 :-

Features of the 8051

Feature	Quantity
ROM	4K bytes
RAM	128 bytes
Timer	2
I/O pins	32
Serial port	1
Interrupt sources	6

Table Comparison of 8051 Family Members

Feature	8051	8052	8031
ROM (on-chip program space in bytes)	4K	8K	0K
RAM (bytes)	128	256	128
Timers	2	3	2
I/O pins	32	32	32
Serial port	1	1	1
Interrupt sources	6	8	6

8051 Architecture :- June 2012

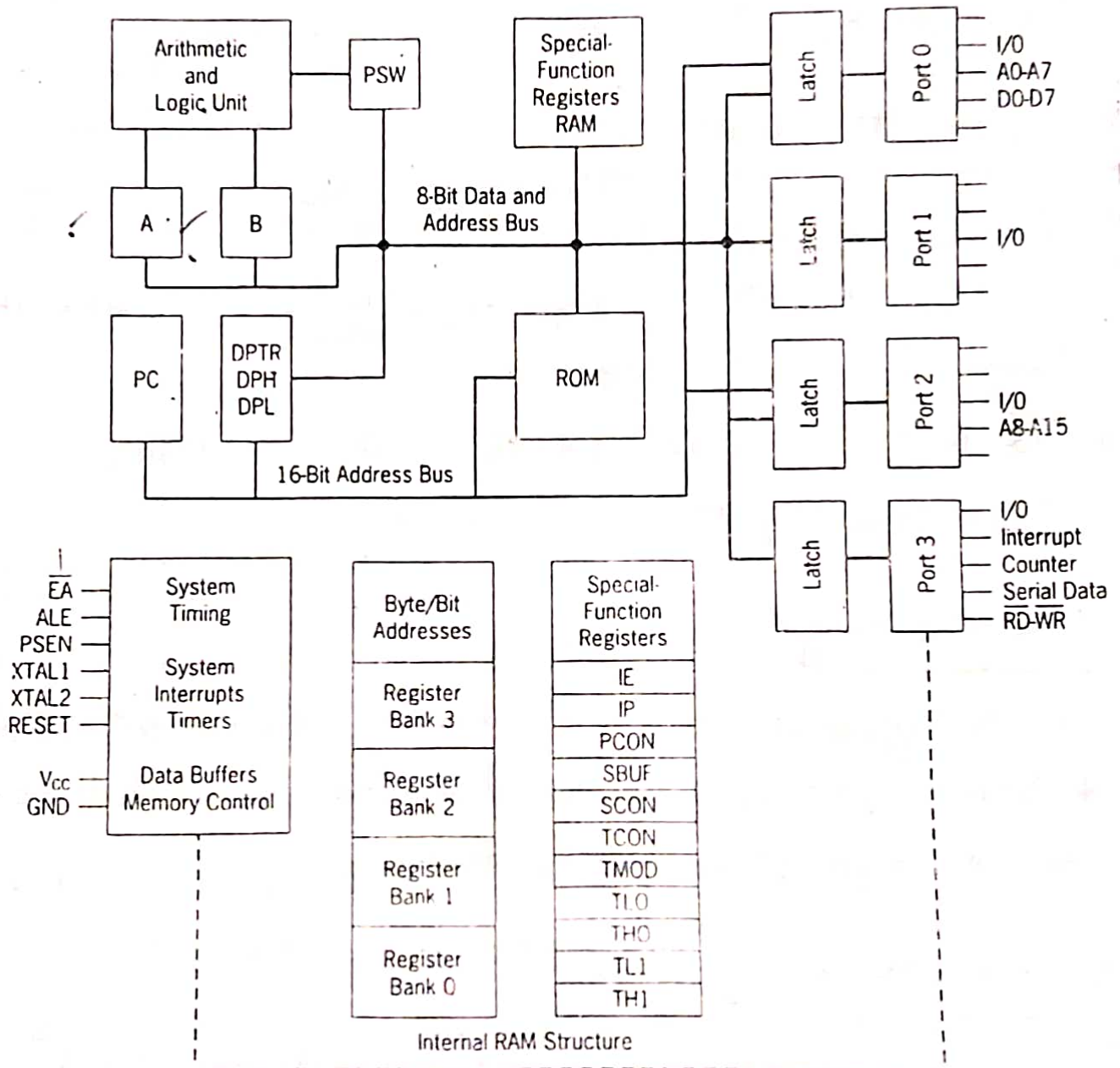
July - 09, 10M

June - 07, 12M

Model, 10M

\* With the neat block diagram, explain the architecture of 8051.

8051 Block Diagram





## Central processing unit (CPU) :-

- \* The 8051 CPU consists of 8-bit arithmetic & Logic unit with associated registers like A, B, PSW, Sp, 16-bit program Counter & "Data pointer registers" (DPTR).
- \* The 8051's ALU can perform arithmetic & Logic functions on 8-bit variables. The arithmetic unit performs addition, subtraction, multiplication & division.
- \* The Logic unit can perform logical operations such as AND, OR & EX-OR, as well as rotate, clear & complement.

## Internal RAM :-

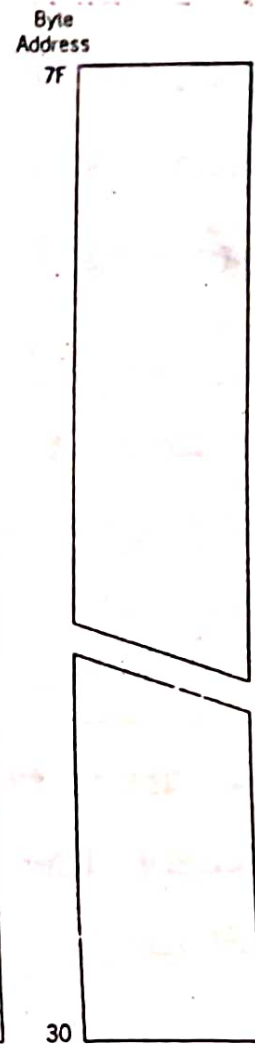
Bank 3		Byte Address	
		1F	R7
		1E	R6
		1D	R5
		1C	R4
		1B	R3
		1A	R2
		19	R1
		18	R0
		17	R7
		16	R6
		15	R5
		14	R4
		13	R3
		12	R2
		11	R1
		10	R0
		0F	R7
		0E	R6
		0D	R5
		0C	R4
		0B	R3
		0A	R2
		09	R1
		08	R0
		07	R7
		06	R6
		05	R5
		04	R4
		03	R3
		02	R2
		01	R1
		00	R0

Byte Address	Bit Address	Bit Address
2F	7F	78
2E	77	70
2D	6F	68
2C	67	60
2B	5F	58
2A	57	50
29	4F	48
28	47	40
27	3F	38
26	37	30
25	2F	28
24	27	20
23	1F	18
22	17	10
21	0F	08
20	07	00

Working Registers

Bit Addressable

General Purpose



\* The 8051 has 128-byte Internal RAM. The Internal RAM of 8051 is organized into three distinct areas

- 1) Working registers ✓
- 2) Bit addressable registers ✓
- 3) General purpose registers.

### Working register :-

The 1<sup>st</sup> 32-bytes from address 00h to 1Fh of Internal RAM constitutes 32 Working registers i.e.

Bank 0 → 8 registers ( $R_0 - R_7$ ) : 00h to 07h

Bank 1 → 8 registers ( $R_0 - R_7$ ) : 08h to 0Fh

Bank 2 → 8 registers ( $R_0 - R_7$ ) : 10h to 17h

Bank 3 → 8 registers ( $R_0 - R_7$ ) : 18h to 1Fh

\* Bits  $RS_0$  &  $RS_1$  in the PSW determine which bank of registers is currently in use.

\* When 8051 is RESET, the BANK 0 is selected.

### Bit addressable register :-

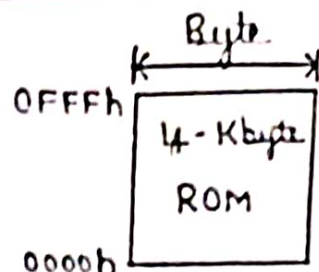
\* The 8051 provides 16-bytes of a bit addressable area. It occupies RAM area from 20h to 2Fh, forming a total of 128 addressable bits. ( $16 \text{ bytes} \times 8 \text{ bits} = 128 \text{ bits}$ )

### General purpose registers :-

The RAM area above bit addressable area from 30h to 7Fh is called general purpose RAM. It is addressable as byte.



## INTERNAL ROM :- ✓



- \* The 8051 has 4-Kbytes of Internal ROM with address space from 0000h to 0FFFh.
- \* The program address higher than 0FFFh, which exceeds the internal ROM capacity will cause the 8051 to automatically fetch code bytes from external program memory.

## A register (Accumulator) :- ✓

- \* Accumulator is a 8-bit register & is widely used for many operations like addition, subtraction, multiplication, division & boolean bit manipulations.
- \* The A-register is also used for all data transfers b/w the 8051 and any external memory.

## B-register :- ✓

The B-register is used with the A-register for multiplication & division operations & has no other function other than as a location where data may be stored.

## STACK pointer (8-bit) :-

- \* The stack refers to an area of internal RAM used by the CPU to store & retrieve (take back) data quickly.
- \* The register used to access the stack is called the Stack pointer (SP) register.
- \* The stack pointer register is used by the 8051 to hold an -

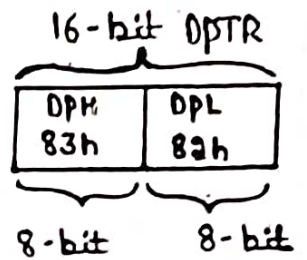
Internal RAM address - that is called the Top of the Stack.

\* When 8051 is RESET, the SP is Set to 07h.

\* The Storing of a CPU register in the Stack is called a PUSH.

\* Loading the contents of the Stack back into the CPU register is called a POP.

Data pointer (DPTR) :- ✓



\* DPTR is a 16-bit register, which holds a 16-bit address.

\* DPTR can be Split into two parts:

➤ DPH → Data pointer high byte having Internal address 83h.

➤ DPL → Data pointer low byte having Internal address 82h.

{ The DPTR does not have a Single Internal address }

Program Counter (PC) :- ✓

\* PC is a 16-bit register which holds the address of the next instruction to be executed. The PC is automatically incremented after every instruction byte is fetched.

\* The 8051 has 16-bit PC hence it can address upto  $2^{16}$  bytes  
i.e.  $2^{16} = 64 \text{ K-bytes}$  of memory.

{ PC is the only register that does not have an Internal address. }

I/O ports (I/O ports) :-

The 8051 has 32 I/O pins configured as Four 8-bit parallel ports  
i.e. Port 0, Port 1, Port 2 & Port 3.



\* All Four ports are bi-directional i.e. each pin can be configured as I/p or o/p under Software Control.

### Timers & Counters :-

\* 8051 use two 16-bit registers namely  $T_0$  &  $T_1$  either for Timers or Counter.

\* The two Timers or Counters are divided into two 8-bit registers called Timer Low ( $TL_0, TL_1$ ) and Timer High ( $TH_0$  &  $TH_1$ ).

### Program Status Word (PSW) or Flag register :-

$D_7$	$D_6$	$D_5$	$D_4$	$D_3$	$D_2$	$D_1$	$D_0$
CY	AC	FO	RS1	RS0	OV	-	P

MSB

LSB

### Carry Flag (CY) :-

After performing arithmetic & logic operation if there is a Carryout from the MSB ( $D_7$ -bit) then  $CY = 1$ , otherwise  $CY = 0$ .

### Auxiliary Carry Flag (AC) :-

After performing arithmetic & logic operation if a Carry is generated from  $D_3$  to  $D_4$  bit then  $AC = 1$ , otherwise  $AC = 0$ .

### RS1 & RS0 :- Register Bank Selection.

RS1	RS0	Register Bank	Address
0	0	0	00H - 07H
0	1	1	08H - 0FH
1	0	2	10H - 17H
1	1	3	18H - 1FH

### Overflow Flag (OV) :-

OV Flag is Set to 1 if either of the following two conditions occur:

- 1) There is a Carry from  $D_6$  to  $D_7$  but NO Carry out of  $D_7$  ( $CY=0$ )
- 2) There is a Carry out from  $D_7$  bit ( $CY=1$ ) but NO Carry from  $D_6$  to  $D_7$  - bit.

### Parity Flag (P) :-

Parity Flag indicates the number of 1's present in the accumulator.

- \* If the number of 1's in the accumulator is odd then  $P=1$ .
- \* If the number of 1's in the accumulator is Even then  $P=0$ .

### Special Function Registers (SFR) :-

The operations of 8051 are done by a group of specific internal registers, each called a Special Function Register (SFR).



1) Explain the Significance of processor Status word. Briefly discuss PSW register of 8051.

Model, 4M

July - 06, 6M

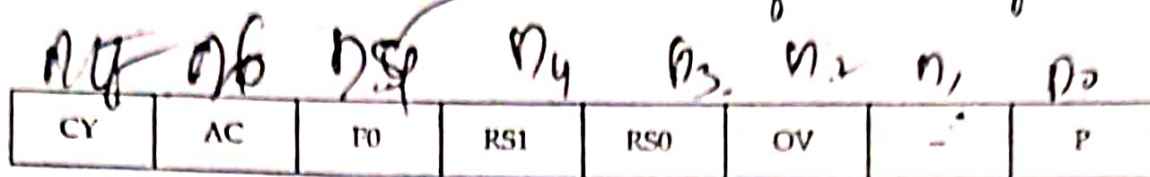
\* The PSW is an 8-bit register. It is also called as Flag register. out of these only 6-bits of PSW registers are used & 2-bits are unused.

(math flag)

\* 4 flags are called Conditional flags because these 4 flags - Indicates some conditions that results after an instruction is executed. i.e. Carry flag, Auxiliary carry flag, parity flag & overflow flag.

\* RS0 & RS1 are used to change the bank register.

\* The two unused bits are called user-definable flags



Carry Flag (CY) :-

After performing arithmetic & logic operation if there is a carryout from the MSB ( $D_7$ -bit) then  $CY=1$ , otherwise  $CY=0$ .

\* Carry Flag 'CY' can be Set to 1 or 0 directly by an instruction such as "SETB C" & "CLR C"

{ Where SETB C  $\rightarrow$  Set bit carry.  
CLR C  $\rightarrow$  Clear carry. }

Auxiliary carry flag (AC) :-

After performing arithmetic & logic operation if a carry is generated from  $D_3$  to  $D_4$  bit then  $AC=1$ , otherwise  $AC=0$ .

RS1 & RS0 :-

Register Bank Select.

RS1	RS0	Register Bank	Address
0	0	0	00H - 07H
0	1	1	08H - 0FH
1	0	2	10H - 17H
1	1	3	18H - 1FH

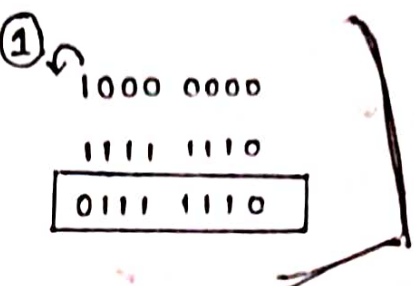


### Overflow Flag (OV):- ✓

\* In 8-bit Signed number operations, OV is Set to 1 if either of the following two conditions occur.

- 1) There is a Carry from  $D_6$  to  $D_7$  but No Carry out of  $D_7$  ( $CY=0$ )
- 2) There is a carryout from  $D_7$  ( $CY=1$ ) but No Carry from  $D_6$  to  $D_7$ .

Eg:-  $-128 \longrightarrow 80h = 1000\ 0000$   
           $-2 \longrightarrow FEh = 1111\ 1110$   
CPU Result  $\rightarrow$   $+126$        $7Eh = 0111\ 1110$



\*  $OV=1$  because carryout from  $D_7$  & No Carry from  $D_6$  to  $D_7$ .

\* Result is Wrong because CPU Shows answer as  $+126_d$  instead of  $-130_d$ .

### Parity Flag (P):- ✓

Parity Flag indicates the number of 1's present in the accumulator.

\* If the number of 1's in the accumulator is odd then  $P=1$ .

\* If the number of 1's in the accumulator is even then  $P=0$ .

Eg:-

\* If  $A = 00011000 \rightarrow P=0$   $\because$  Number of 1's are even

\* If  $A = 10001100 \rightarrow P=1$   $\because$  Number of 1's are odd

## Problems :-

1) Show the contents of the PSW register after the execution of the following instructions.

MOV A, #9CH

ADD A, #64H

Sol :-

$$\begin{array}{rcl}
 9CH & \longrightarrow & 1001\ 1100 \\
 + 64H & \longrightarrow & 0110\ 0100 \\
 \hline
 \boxed{100H} & & \textcircled{1} 0000\ 0000
 \end{array}$$

- 1) CY = 1 : Since there is a Carry beyond the D<sub>7</sub>-bit
  - 2) AC = 1 : Since there is a Carry from D<sub>3</sub> to D<sub>4</sub>-bit
  - 3) P = 0 : Since the accumulator has an even number of 1's i.e. it has Zero 1's.
  - 4) RS0 = 0 : } By default Bank 0 is Selected.
  - 5) RS1 = 0 : }
  - 6) F<sub>0</sub> = 0 : unused, hence 0.
  - 7) OV = 0 : Since there is Carry from D<sub>6</sub> to D<sub>7</sub> & a Carryout from D<sub>7</sub>-bit.
  - 8) PSW.1 = 0 : Not used, hence 0
- ∴ The Contents of PSW is 11000000 i.e. C0h.

2) Show the contents of the PSW register after the addition of BFH & 1BH in the following instruction.

MOV A, #0BFH

ADD A, #1BH

Sol :-

$$\begin{array}{rcl}
 BF & \longrightarrow & 1011\ 1111 \\
 + 1B & \longrightarrow & 0001\ 1011 \\
 \hline
 \boxed{DA} & & \boxed{1101\ 1010}
 \end{array}$$



The bits of PSW are as follows:

- 1)  $CY=0$ : Since there is no carry beyond the  $D_7$ -bit
- 2)  $AC=1$ : Since there is a carry from the  $D_3$  to the  $D_4$  bit.
- 3)  $F_0=0$ : Unused, hence 0.
- 4)  $RS1=0$ : By default, Bank 0 is Selected
- 5)  $RS0=0$ : By default, Bank 0 is Selected.
- 6)  $OV=0$ : Since there is No carry from  $D_6$  to  $D_7$
- 7)  $PSW.1=0$ : Not used, hence 0.
- 8)  $P=1$ : Since there is an odd number of 1's in the accumulator

\* The contents of the PSW is thus 01000001 i.e. 41h

### Special Function Register:- (SFR)

July - 08, 5M

July - 07, 5M

Name	Function	Internal RAM address (HEX)
A	Accumulator	0E0
B	Arithmetic	0F0
DPH	Addressing external memory	83
DPL	Addressing external memory	82
IE	Interrupt enable control	0A8
IP	Interrupt priority	0B8
P0	Input/output port latch	80
P1	Input/output port latch	90
P2	Input/output port latch	A0
P3	Input/output port latch	0B0
PCON	Power control	87
PSW	Program status word	0D0
SCON	Serial port control	98
SBUF	Serial port data buffer	99
SP	Stack pointer	81
TMOD	Timer/counter mode control	89
TCON	Timer/counter control	88
TL0	Timer 0 low byte	8A
TH0	Timer 0 high byte	8C
TL1	Timer 1 low byte	8B
TH1	Timer 1 high byte	8D

\* The 8051 operations that do not use the internal 128-byte RAM - address from 00h to 7Fh.

\* The operations of 8051 are done by a group of Specific internal registers, each called a Special Function Register "SFR".

\* The SFR's may be accessed by their names & by using addresses from 80h to FFh.

\* Not all the addresses from 80h to FFh are used for SFR's. If we attempt to use an address that is not defined & empty, results in unpredictable results.

NOTE:- The program Counter (PC) is not part of the SFR & has NO Internal RAM address.

\* List out the different bit addressable SFR's available in 8051.

Jan-06, 4M

Bit addressable SFR's available in 8051 are shown below

Symbol	Name	Address
* ACC	accumulator	0E0H
* B	B Register	0F0H
* PSW	Program Status Word	0D0H
* P0	Port 0	80H
* P1	Port 1	90H
* P2	Port 2	0A0H
* P3	Port 3	0B0H
* IP	Interrupt Priority Control	0B8H
* IE	Interrupt Enable Control	0A8H
* TCON	Timer/Counter Control	88H
* SCON	Serial Control	98H



NOTE:- { Dont remember }

Byte address	Bit address	
FF		
F0	F7 F6 F5 F4 F3 F2 F1 F0	B
E0	E7 E6 E5 E4 E3 E2 E1 E0	ACC
D0	D7 D6 D5 D4 D3 D2 D1 D0	PSW
B8	-- -- -- BC BB BA B9 B8	IP
B0	B7 B6 B5 B4 B3 B2 B1 B0	P3
A8	AF -- -- AC AB AA A9 A8	IE
A0	A7 A6 A5 A4 A3 A2 A1 A0	P2
99	not bit-addressable	SBUF
98	9F 9E 9D 9C 9B 9A 99 98	SCON
90	97 96 95 94 93 92 91 90	P1
8D	not bit-addressable	TH1
8C	not bit-addressable	TH0
8B	not bit-addressable	TL1
8A	not bit-addressable	TL0
89	not bit-addressable	TMOD
88	8F 8E 8D 8C 8B 8A 89 88	TCON
87	not bit-addressable	PCON
83	not bit-addressable	DPH
82	not bit-addressable	DPL
81	not bit-addressable	SP
80	87 86 85 84 83 82 81 80	P0

Special Function Registers

Figure SFR RAM Address (Byte and Bit)

1) Explain the memory organization in 8051 microcontroller

Jan-09, 8M

2) With neat diagrams, give the details of program memory & data memory of 8051.

Model, 8M

3) Draw & explain the memory structure of 8051

6M

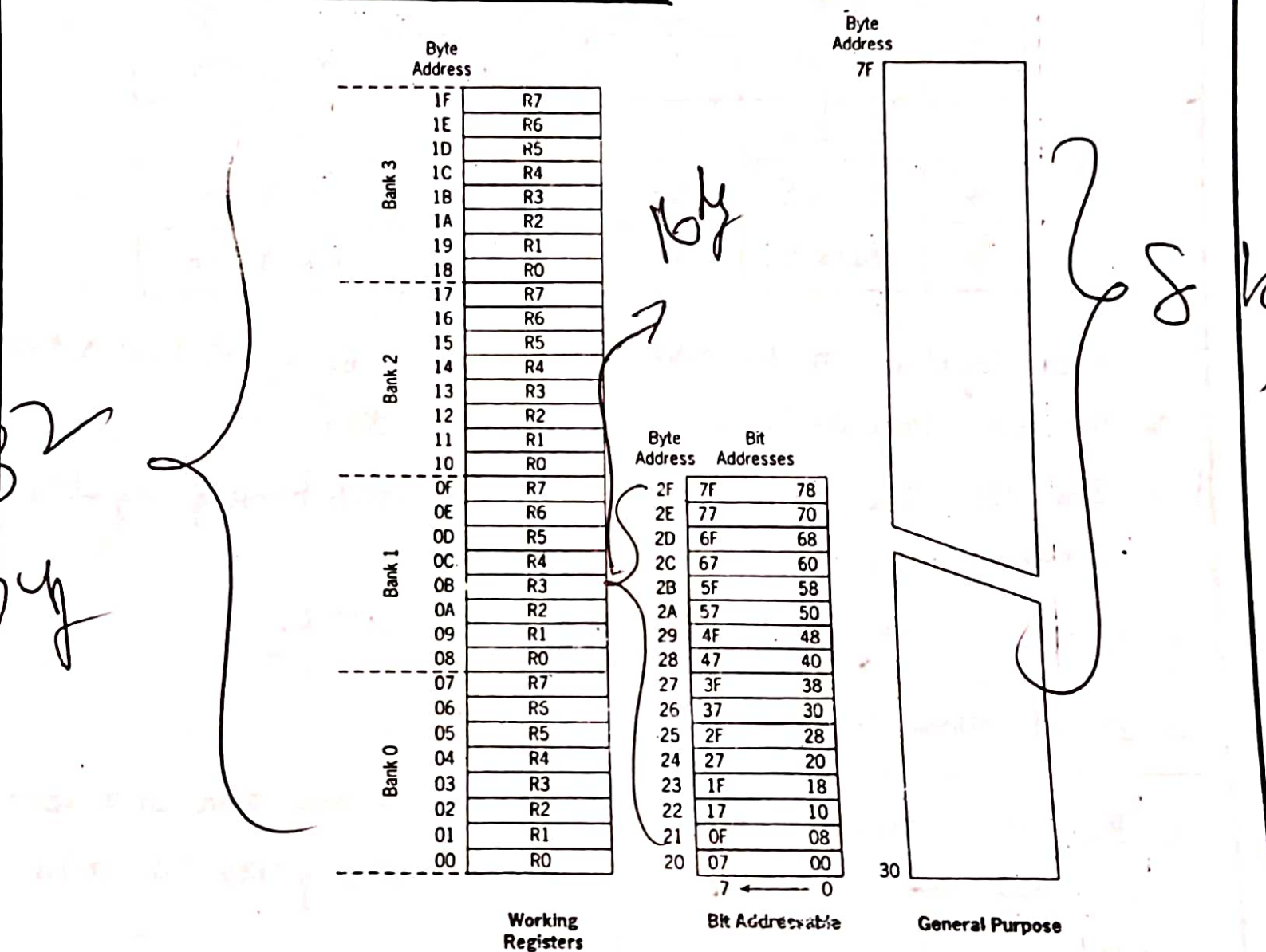
4) Explain the Internal RAM organization of 8051 microcontroller.

\* The memory organization of 8051 microcontroller is classified into two types:

1) Data Memory (RAM)

2) Program Memory (ROM)

1) Data Memory & Internal RAM :-





\* The 8051 has 128-byte Internal RAM. The internal RAM of 8051 is organized into three distinct areas.

- 1) Working registers
- 2) Bit addressable registers &
- 3) General purpose registers.

### 1) Working Register :-

\* The 1st 32 bytes from address 00h to 1Fh of Internal RAM constitutes 32 Working registers i.e.

Sl No	Bank	No of registers	Register Name	Address Range
1)	Bank 0	8	R <sub>0</sub> - R <sub>7</sub>	00h to 07h
2)	Bank 1	8	R <sub>0</sub> - R <sub>7</sub>	08h to 0Fh
3)	Bank 2	8	R <sub>0</sub> - R <sub>7</sub>	10h to 17h
4)	Bank 3	8	R <sub>0</sub> - R <sub>7</sub>	18h to 1Fh

\* Each register can be addressed by name & by its RAM address.

\* Only one register bank is in use at a time.

\* Bits RS0 & RS1 in the PSW determine which bank of registers is currently in use.

\* When 8051 is RESET, the BANK 0 is selected.

### 2) Bit addressable register :-

\* The 8051 provides 16-bytes of a bit addressable area. It occupies RAM area from 20h to 2Fh, forming a total of 128 addressable bits (i.e. 16 bytes  $\times$  8-bits = 128 bits)

\* The addressable bit may be specified by its bit address of 00h to 7Fh.

2) 8-bit may form any byte address from 20h to 2Fh.

{ eg:-

\* Bit address 4Fh refers bit 7 of the byte address 29h.

\* Bit address 4Eh refers bit 6 of the byte address 29h.

\* Mainly used for a binary event such as SWITCH ON, Light, OFF etc  
}

3) General purpose register:-

The RAM area above bit addressable area from 30h to 7Fh is called general purpose RAM. It is addressable as byte.

II) Program memory (ROM)

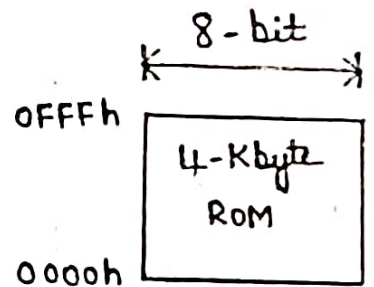
Program memory is classified into:

1) Internal ROM ✓

2) External ROM ✓

1) Internal ROM:-

\* The 8051 has 4-Kbyte of Internal ROM with address ranges from 0000h to 0FFFh.



\* The 8051 has control pins such as  $\overline{\text{PSEN}}$  (Program Store Enable) &  $\overline{\text{EA}}$  (External access) that determines whether external memory is accessed or internal memory is accessed.

\* If  $\overline{\text{EA}}$  pin is connected to VCC (usually +5V), then the program memory accessed by the chip is Internal memory.



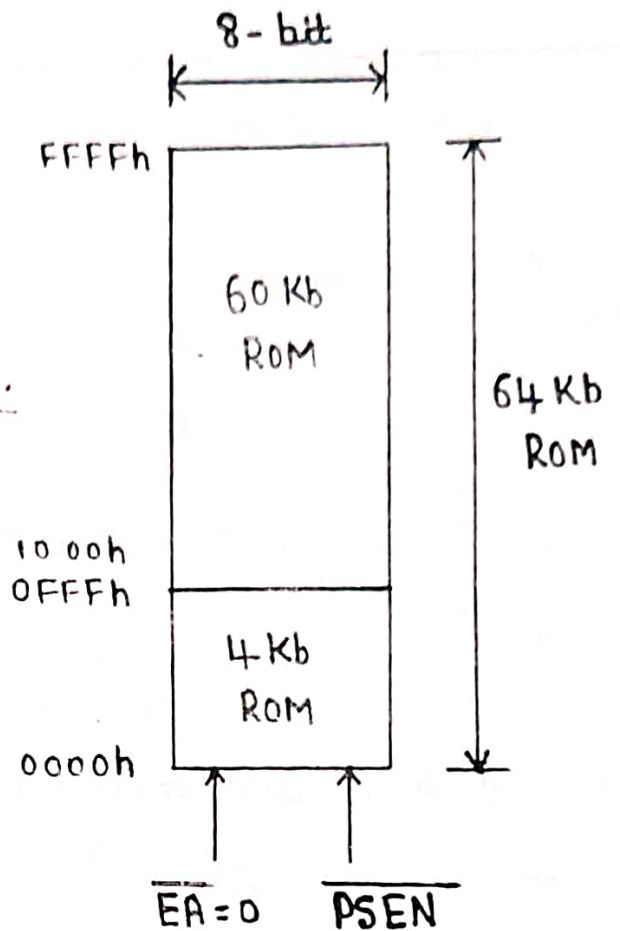
## ➤ External memory :-

\* When  $\overline{EA}$  pin is connected to VSS (0V & ground) then the 8051 can access external memory starting from address 0000h to FFFFh.

### NOTE :-

ON-Chip ROM address : 0000h to 0FFFh

OFF-Chip ROM address : 0000h to FFFFh



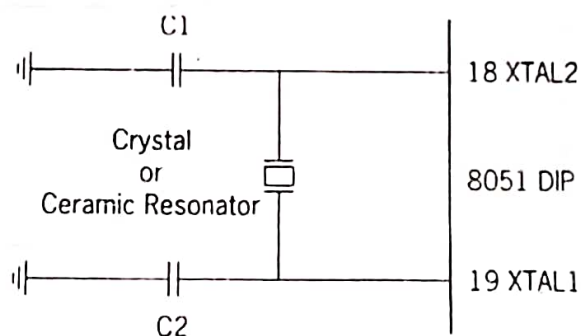
## 8051 oscillator & Clock:-

1) Explain the oscillator circuit & timing of 8051 microcontroller  
Jan - 07, 6M

2) Explain with neat CKT diagram how the clock circuit works  
Jan - 08, 14M

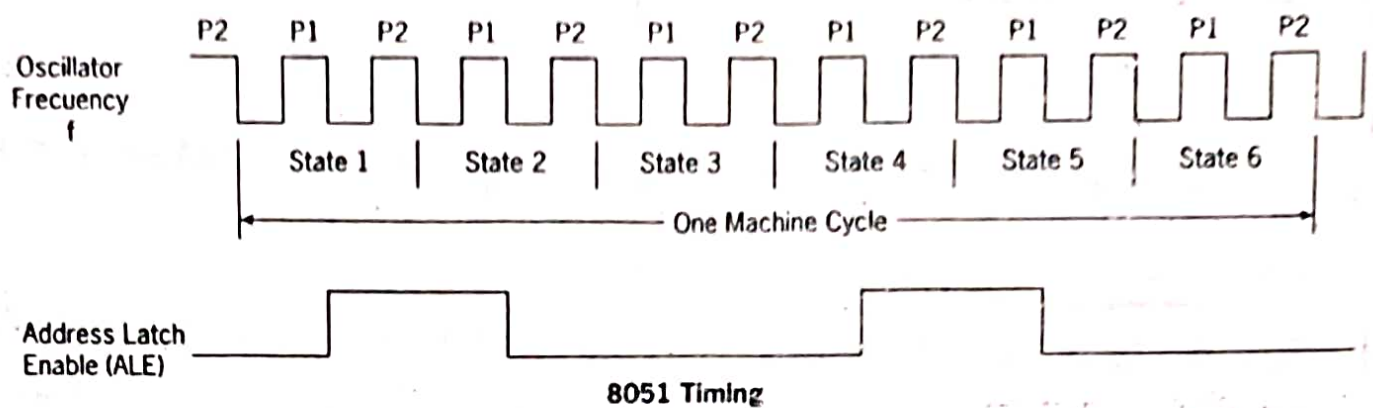
\* All internal operations of the 8051 are Synchronized by the clock pulses. These clock pulses are generated by using oscillator Ckt.

\* The 8051 provides XTAL1 & XTAL2 pins for connecting a resonant network to form an oscillator as shown in fig ①.





- \* The oscillator circuit consists of Quartz Crystal & Capacitors.
- \* The Crystal Frequency is the basic internal clock frequency of the microcontroller.
- \* The minimum & maximum operating frequencies for 8051 are typically 1MHz to 16MHz respectively.



- \* In 8051 one machine cycle consists of 6 States numbered  $S_1$  to  $S_6$   
i.e. one Machine cycle = 6 States
- \* Each State consists of two oscillator pulses.  
i.e. one State = 2 oscillator pulses
- \* The machine cycle is defined as the Smallest interval of time needed to execute (accomplish) any Simple instruction.
- \* Instructions may require one, two or four machine cycles to execute any instructions depending on the type of instructions.
- \* When microcontroller is RESET, instructions are fetched & executed by microcontroller automatically beginning with the instruction located at ROM memory address 0000h.
- \* Time needed to execute an instruction is calculated as:  
$$T_{\text{inst}} = \frac{C \times 12d}{\text{Crystal Frequency}}$$

Where

' $T_{inst}$ ' is the time for instruction to be executed

' $C$ ' is the number of machine cycles &

' $f$ ' is the crystal frequency.

- \* In fig (2), there are two ALE pulses per machine cycle. These ALE pulses are used for external memory access.
- \* Instructions which are two byte long can be fetched & executed in one machine cycle.
- \* Single byte instructions are not executed in a half machine cycle, however single byte instructions "throw-away" the 2<sup>nd</sup> byte (which is the 1<sup>st</sup> byte of the next instruction)
- \* The next instruction is then fetched in the following machine cycle. (i.e. in next machine cycle)

1) Calculate the machine cycle if the crystal frequency is  
i) 16 MHz ii) 12 MHz

Sol:- i) Time period for one pulse =  $\frac{1}{f} = \frac{1}{16 \times 10^6} = \underline{0.0625 \mu\text{sec}}$

one machine cycle = 12 pulses

$\therefore$  one machine cycle time =  $0.0625 \mu\text{sec} \times 12 = \underline{0.75 \mu\text{sec}}$

ii) Time period for one pulse =  $\frac{1}{f} = \frac{1}{12 \times 10^6} = \underline{0.0833 \mu\text{sec}}$

one machine cycle = 12 pulses

$\therefore$  one machine cycle time =  $0.0833 \mu\text{sec} \times 12 = \underline{1 \mu\text{sec}}$



Q) In an 8051 System, driven by 11.0592 MHz Clock, find the time taken for an instruction which takes 11 machine cycles.

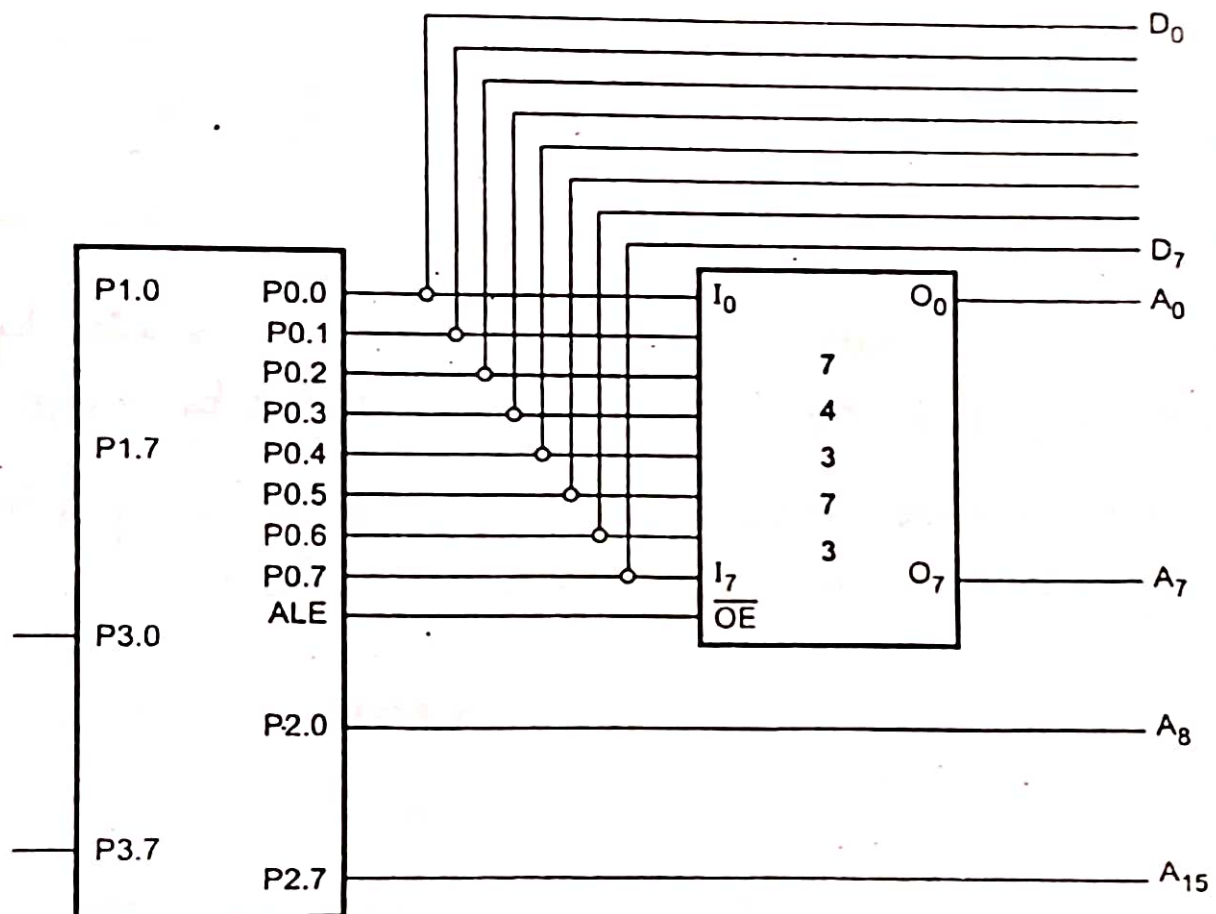
Sol:-

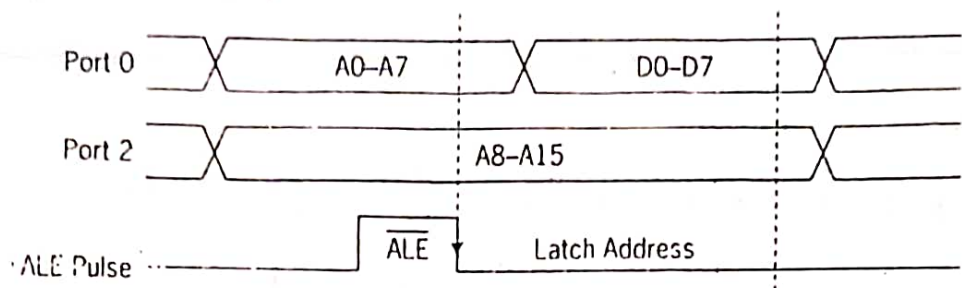
$$T_{inst} = \frac{C \times 12d}{f} = \frac{4 \times 12}{11.0592 \times 10^6}$$

$$T_{inst} = 4.34 \mu sec$$

\* Write the hardware required to demultiplex the address bus of 8051.

July-06, SM





\* In 8051 microcontroller, the port 0 has multiplexed address/data lines i.e.  $A_0-A_7$ .

\* The ALE Signal is used to demultiplex (Separate) lower order address bus ( $A_0-A_7$ ) & data bus Signal ( $D_0-D_7$ ).

\* When  $ALE = 1$ , port 0 contains address bus &  
When  $ALE = 0$ , port 0 contains data bus.

So we use 74LS373 to demultiplex address & data bus.

\* The 74LS373 will be enabled when ALE is high, So O/P of 74LS373 has lower order address  $A_0-A_7$  as Shown in fig ①.



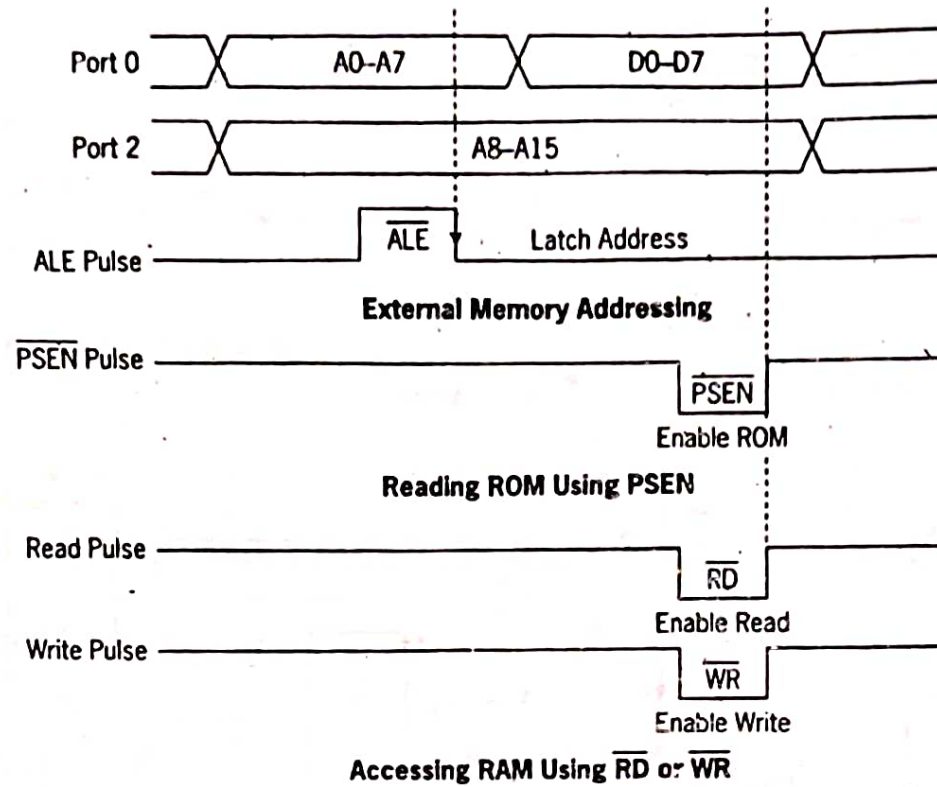
Jan-06, 8M

Jan-09, 8M

\* Draw a Schematic to interface external ROM & RAM to 8051.  
How to access them?

Jan-08, 8M

July-07, 8M



### Accessing RAM:-

\* The 8051 Can address upto 64K-bytes of external data memory.  
The "MOVX" instruction is used to access the external data memory.

\* The Internal data memory is divided into

Sl No	Internal RAM	Address range	Registers
1	Lower 128 byte	00h - 7Fh	Working register, bit-addressable register & General purpose register
2	Upper 128 byte	80h - FFh	SFR registers.

\* The SFR registers are accessed by Indirect Addressing only.

\* The Lower 128 bytes are accessed either by direct addressing or

by indirect addressing.

}

$\overline{RD}$ ,  $\overline{WR}$  :-

$\overline{RD}$  &  $\overline{WR}$  pins are used when a RAM has to be accessed. When  $\overline{RD}=0$ , a data byte can be read from a RAM location.

When  $\overline{WR}=0$ , a data byte can be written into a RAM location.

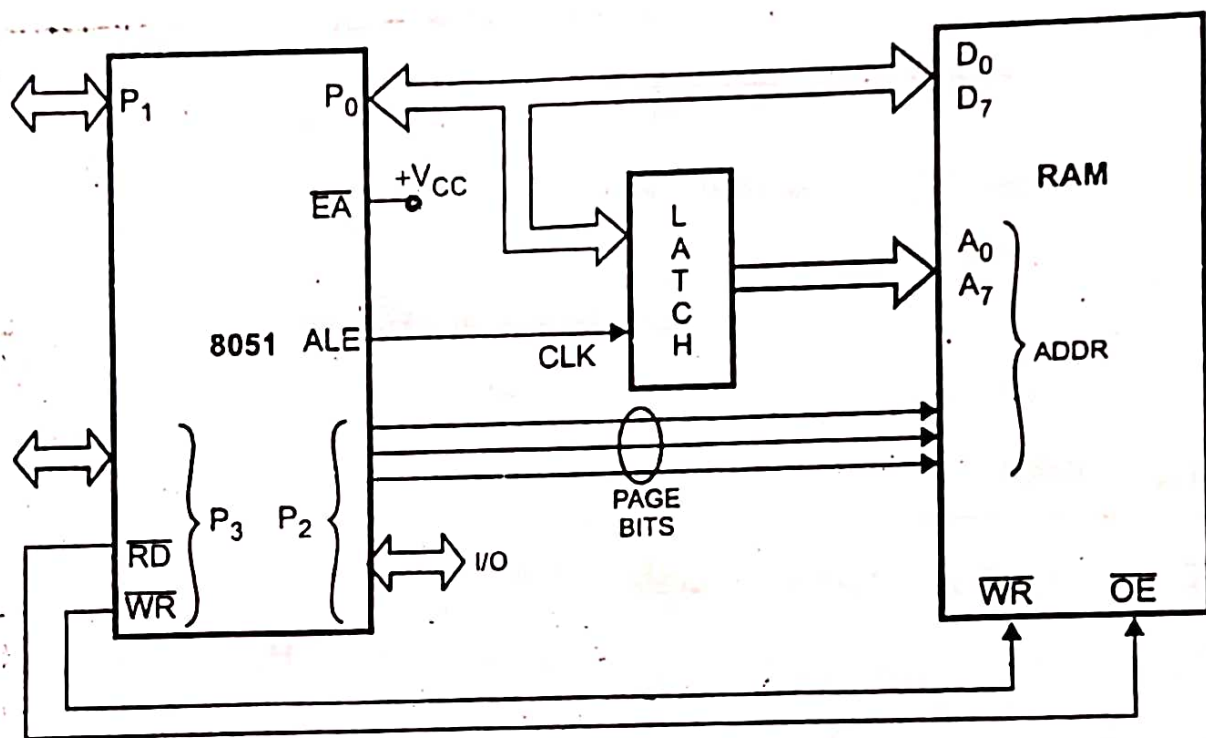


Fig. Accessing external data memory

Accessing ROM :-

\* In 8051, When the  $\overline{EA}$  pin is connected to VCC, the 8051 fetches addresses 0000h through 0FFFh & are directed to Internal ROM.

\* When  $\overline{EA}=0$  (GND), the 8051 fetches address 0000h through FFFFh & are directed to external ROM/EPROM.



\* P8t0 is a multiplexed address/data bus.

When ALE=1, P8t0 Contains address bus ( $A_0-A_7$ )

When ALE=0, P8t0 Contains data bus ( $D_0-D_7$ )

\* The 74LS373 is used to demultiplex address & data bus.

\* The 74LS373 will be enabled when ALE is high, So o/p of 74LS373 has lower order address  $A_0-A_7$  as Shown in Figure.

\* Program Store Enable ( $\overline{PSEN}$ ) pin is an active low pin, which is used to activate o/p enable Signal of the external ROM/EPROM as Shown in Figure ③.

\* When 8051 has to access program Code from an external ROM,  $\overline{PSEN}$  is connected to the enable pin ( $\overline{OE}$ ) of the ROM Chip

\* To access the program Code,  $\overline{EA}$  must be grounded then -  $\overline{PSEN}$  will go low, to enable the external ROM to place a byte of program Code on the data bus.

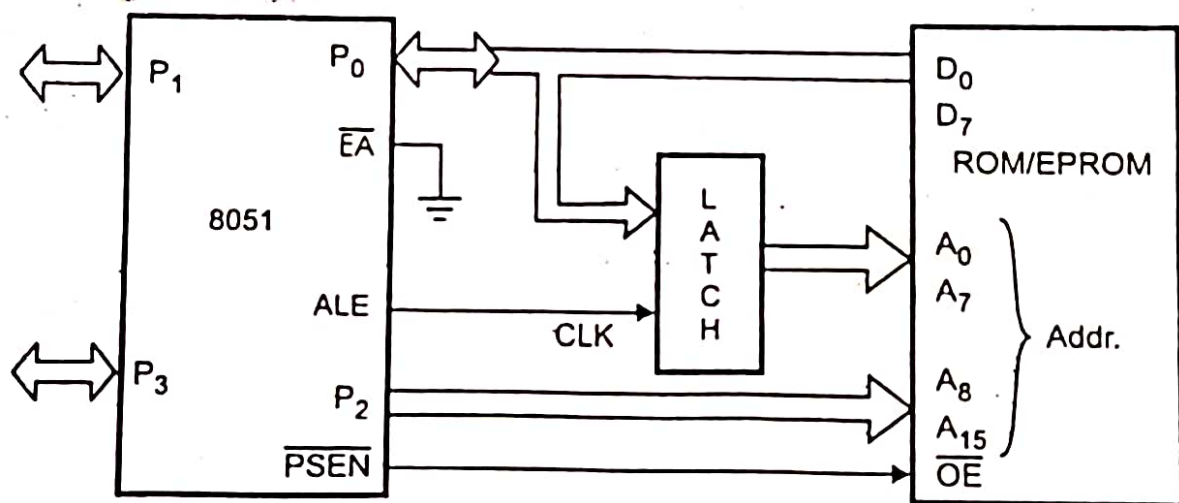
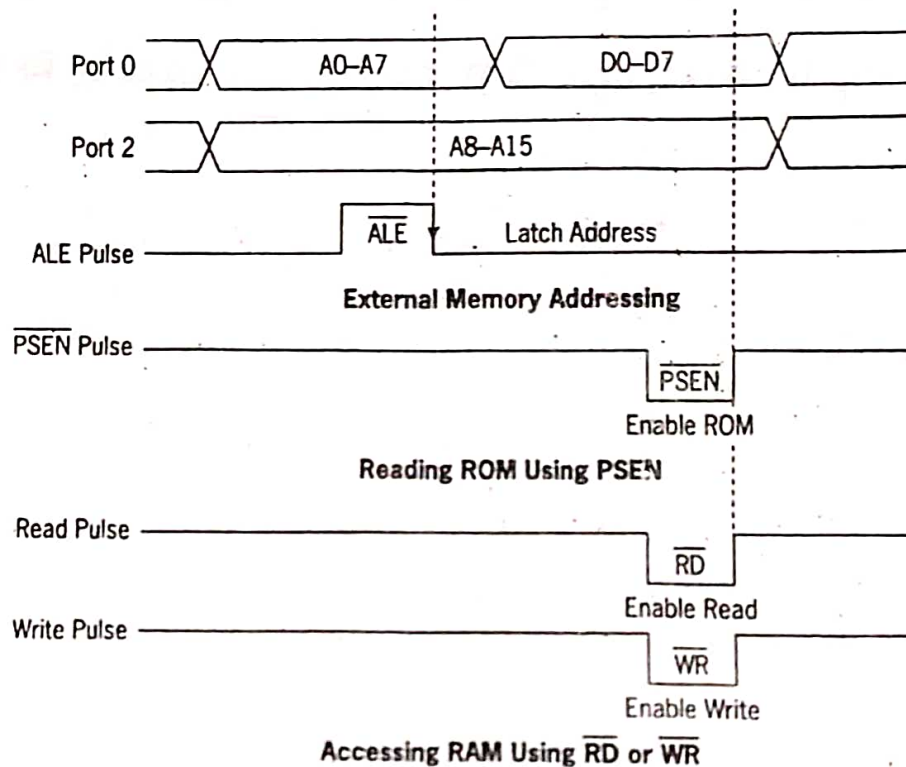


Fig. Accessing external program memory

\* Explain the timing diagram of external memory cycle of 8051 microcontroller

July-06, 7M





\* The 8051 address bus is 16-bit ( $A_0-A_{15}$ )

\*  $P_{80}$  has multiplexed address/data bus ( $AD_0-AD_7$ ), it provides lower byte address ( $A_0-A_7$ ) of 16-bit address.

\* The ALE Signal is used to demultiplex (Separate) lower order address bus ( $A_0-A_7$ ) & data bus Signal ( $D_0-D_7$ ).

\* When ALE=1,  $P_{80}$  Contains address bus &

When ALE=0,  $P_{80}$  Contains data bus.

So we use 74LS373 to demultiplex address & data bus.

\* When external ROM is to access (read),

$\overline{PSEN}=0$ , ROM is enabled & a byte of data is placed on the data bus.

\* RD & WR pins are used when a RAM has to be accessed.

When  $\overline{RD}=0$ , a data byte can be read from a RAM location.

When  $\overline{WR}=0$ , a data byte can be written into a RAM location.

I/O ports :-

- \* A port is a pin where data can be transferred between 8051 and an external device. The 8051 has four 8-bit ports P<sub>0</sub>, P<sub>1</sub>, P<sub>2</sub> & P<sub>3</sub>.
- \* Each port has a D-type flip-flop for each pin & each port pins can be either used as I/p or o/p pin under software control.

Port 0 :-

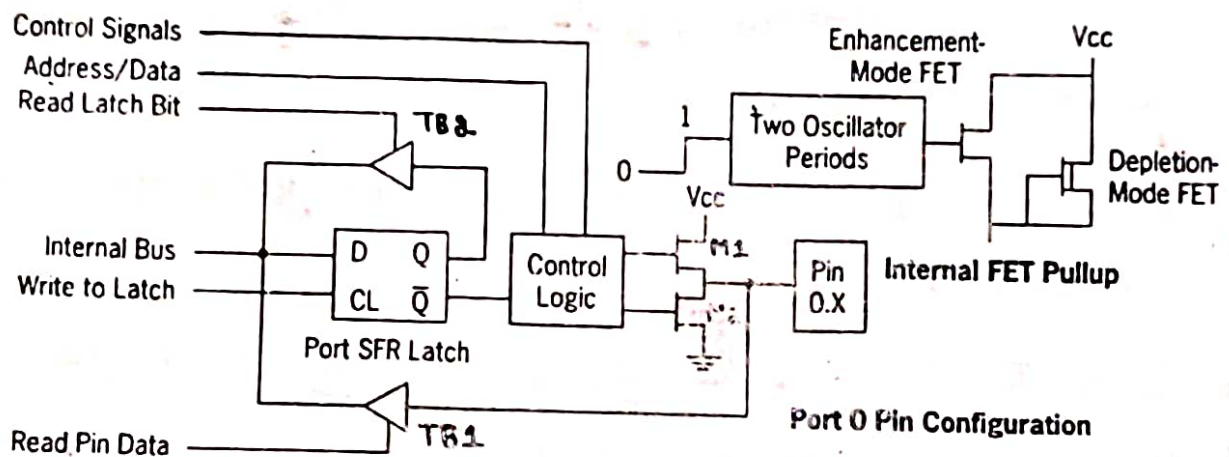
June-07, 8M

- \* Draw the sketch of the pin configuration of port 0 pins & explain the various operations performed by the pins of port 0.

- \* Hardware Configuration of port 0

June-06, 5M

Jan-07, 6M



- \* Port 0 is an 8-bit, bit addressable I/p - o/p port. It is also used as a bi-directional low-order address & data bus for external memory.

I/p port :-

- \* To use each pin as an I/p pin, 1<sup>st</sup> we must write '1' (logic



high) to that bit i.e.

- 1) Writing a 1 to the port 0 pin, the D-FF o/p is high i.e.  
 $Q=1$  &  $\bar{Q}=0$ .
- 2) Since  $\bar{Q}=0$  & is connected to the FET's gate  $M_1$  &  $M_2$ ,  
thus turning OFF the both FET's.
- 3) When  $M_1$  &  $M_2$  are OFF, it acts like open circuit & there  
will be no connection between port 0 pin & ground, thus  
I/p Signal is directed to the tristate buffer TB1.

ex:-      `Mov A, #0FFh`  
            `Mov po, A.`

o/p port :-


\* To use each pins as an o/p pin, 1<sup>st</sup> we must write a '0'  
(logic Zero) to that bit i.e.

- 1) Writing a '0' to the port 0 pin, the D-FF o/p is low i.e.  
 $Q=0$  &  $\bar{Q}=1$ .
- 2) Since  $\bar{Q}=1$  & is connected to the FET's gate  $M_1$  &  $M_2$ ,  
thus turning ON the both FET's.
- 3) When  $M_1$  &  $M_2$  are ON, it acts like short circuit, thus  
port pin is connected to the ground.

∴ Any attempt to read the I/p pin will always get the  
low ground signal regardless of the status of the I/p pin

ex:-      `Mov A, #00h`  
            `Mov po, A`

### Address / Data bus operation :-

- \* P<sub>0</sub> is also used to carry the multiplexed address / data bus during access to external memory.
- \* When the 8051 access external memory, P<sub>0</sub> pin carry the low-order address whenever the ALE has a rising edge signal i.e. 
- \* P<sub>0</sub> lines are further used to carry the bi-directional data bus, to read & write to external memory.

### NOTE :-

\* P<sub>0</sub> Address  $\rightarrow$  80h

\* P<sub>0</sub> bit address.

P <sub>0</sub> .7	P <sub>0</sub> .6	P <sub>0</sub> .5	P <sub>0</sub> .4	P <sub>0</sub> .3	P <sub>0</sub> .2	P <sub>0</sub> .1	P <sub>0</sub> .0
87	86	85	84	83	82	81	80



NOTE :-

- 1) To use the pins of port 0 as both I/p & o/p ports, each pin must be connected externally to a 10K $\Omega$  resistor, called pull up resistor.

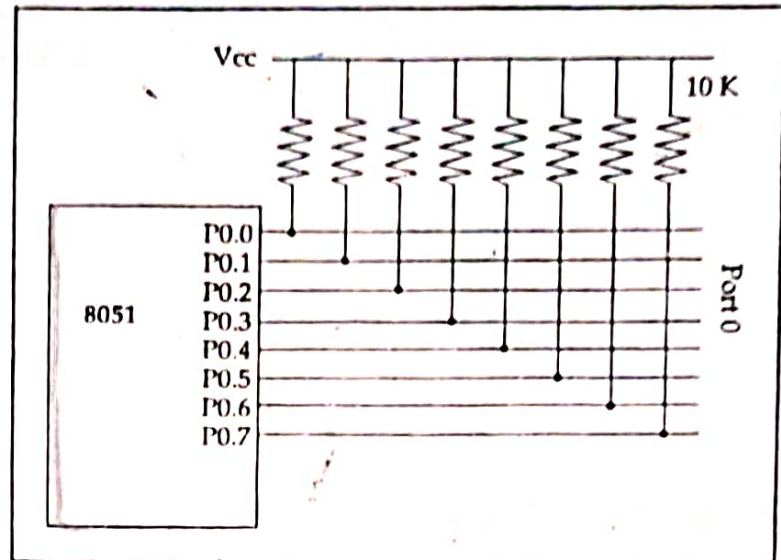


Figure Port 0 with Pull-Up Resistors

- 2) Port 0 does not need a pull up resistor when used to access external memory.

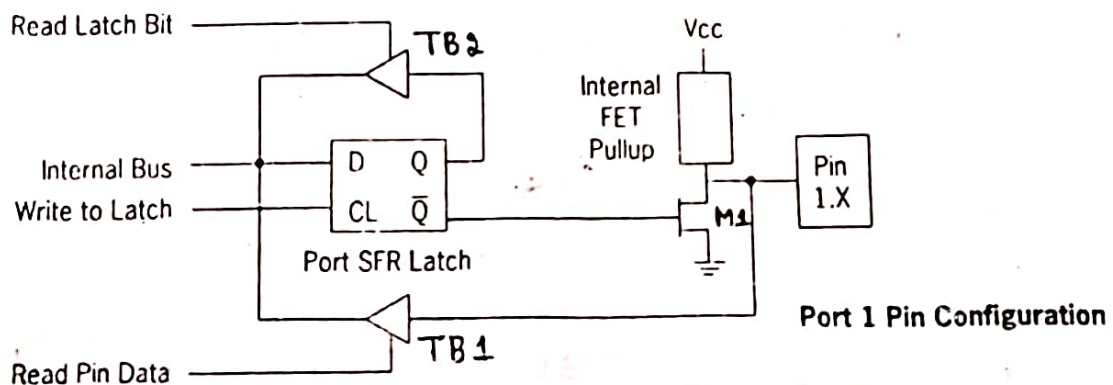
Port 1 : 5 marks

Jan-09, 8M

- \* Sketch the internal circuit diagram of port 1 of 8051 & briefly explain how to use it as I/p & o/p port. How does this port differ from other ports?

Jan-06, 6M

- \* Explain with diagram the feature & operation of port 1



### I/O port :-

\* To use each pin has an I/p pin, first we must write a '1' (logic high) to that bit i.e.

- 1> Writing a '1' to the port 1 pin, the D-FF o/p is high i.e.  $A=1$  &  $\bar{A}=0$ .
- 2> Since  $\bar{A}=0$  & is connected to the FET gate  $M_1$ , thus turning OFF the FET.
- 3> When  $M_1$  is OFF, it acts like open circuit (it blocks any path to the ground) thus I/p signal is directed to the tri-state buffer TB1.

ex:- `MOV A, #0FFh`  
`MOV P1, A`

### O/p port :-

\* To use each pin has an o/p pin, 1<sup>st</sup> we must write a '0' (logic low) to that bit i.e.

- 1> Writing a '0' to the port 1 pin, the D-FF o/p  $A=0$  &  $\bar{A}=1$
- 2> Since  $\bar{A}=1$  & is directly connected to the FET gate  $M_1$ , thus turning ON the FET.
- 3> When  $M_1$  is ON, it acts like Short Circuit, thus port pin is connected to the ground.

∴ Any attempt to read the I/p pin will always get the low ground signal regardless of the status of the I/p pin.

ex:- `MOV A, #00h`  
`MOV P1, A`



## NOTE:-

\* Port 1 address  $\rightarrow 90h$

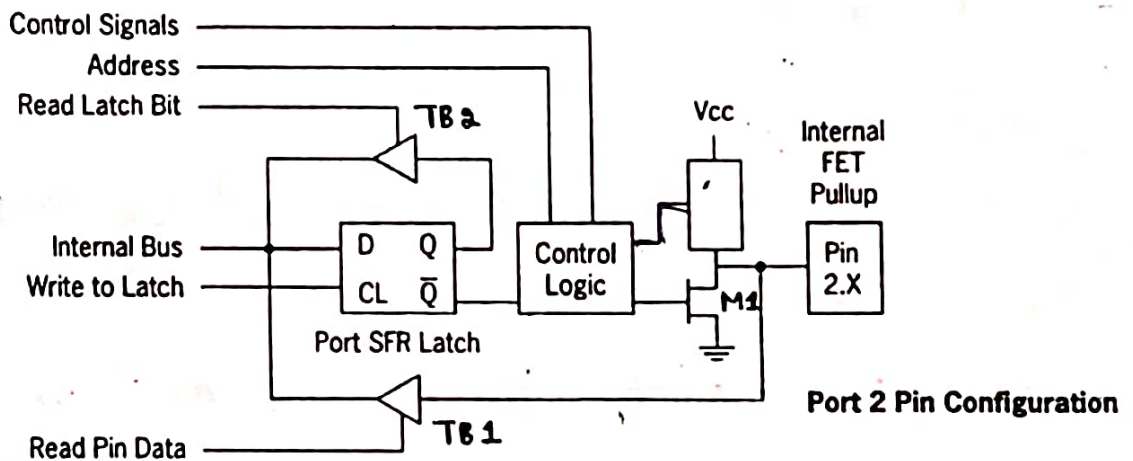
\* Port 1 bit address:

P1.7	P1.6	P1.5	P1.4	P1.3	P1.2	P1.1	P1.0
97	96	95	94	93	92	91	90

\* Port 1 pins do not have any dual functions (only I/O function)

\* Port 1 pins have Internal pull-up resistors.

## Port 2 :-



\* Port 2 pins can be used as


1) I/O port

2) Higher order address ( $A_8 - A_{15}$ )

## I/O port :-

## O/P port :-

\* Port 2 pins are used to carry the higher order address ( $A_8-A_{15}$ ) bus during external memory access.

i.e. during rising edge of ALE Signal, Port 2 provides  $A_8-A_{15}$  address lines 

### NOTE :-

\* Port 2 address is  $A0h$

\* Port 2 bit address

Pa.7	Pa.6	Pa.5	Pa.4	Pa.3	Pa.2	Pa.1	Pa.0
A7	A6	A5	A4	A3	A2	A1	A0

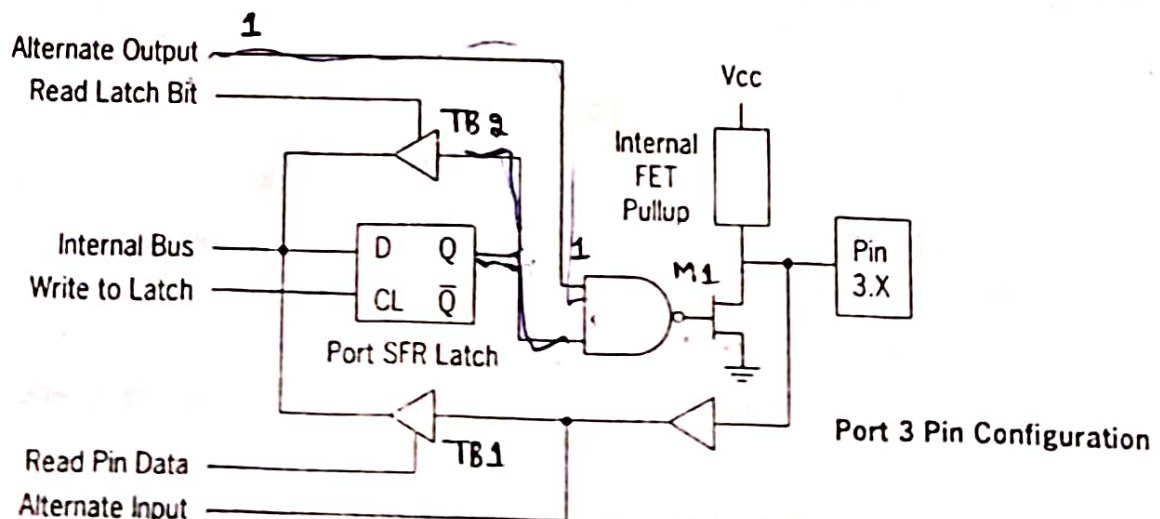
### Port 3 :-

July-08, 10M

\* With the Sketch of Pin Configuration of Port 3 pins & explain the various operations performed by the pins of port 3.

\* Explain the Functions of port 3.

Jan-07, 6M





### I/p port :-

\* To use each pin as I/p pin, 1<sup>st</sup> we must write a '1' (logic high) to that bit i.e.

- 1> Writing a '1' to the port 3 pins, the D-FF o/p is high i.e.  $A=1$
- 2> Since  $A=1$  & is connected to one I/p of NAND gate & another I/p is '1' thus o/p of NAND gate is '0' & Turns OFF the FET.
- 3> When FET is OFF, it acts like open circuit, thus I/p signal is directed to the tristate buffer, TB1.

ex:- `MOV A, #0FFh`

`MOV P3, A`

### O/p port :-

\* To use each pin as an o/p pin, 1<sup>st</sup> we must write a '0' (logic low) to that bit i.e.

- 1> Writing a '0' to the port 3 pin, the D-FF o/p  $A=0$ , thus o/p of NAND gate is '1' (high) & Turns ON the FET.
- 2> When FET is ON, it acts like Short circuit, thus it provides the path to ground to the I/p pin.

∴ Any attempt to read the I/p pin will always get the low ground signal regardless of the status of the I/p pin.

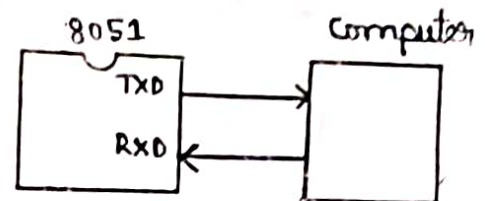
ex:- `MOV A, #0FFh`

`MOV P3, A`

<p><u>NOTE:-</u> For I/p - o/p operation, alternate o/p is always '1'</p>
---

## Alternate Function of port 3 :

Pin	Alternate Use
P3.0-RXD	Serial data input
P3.1-TXD	Serial data output
P3.2-INT0	External interrupt 0
P3.3-INT1	External interrupt 1
P3.4-T0	External timer 0 input
P3.5-T1	External timer 1 input
P3.6-WR	External memory write pulse
P3.7-RD	External memory read pulse



### RXD & TXD :-

\* In 8051  $\mu$ c RXD & TXD pins are used for Serial Communication.

TXD :- The data is Transmitted out of 8051 through TXD pin

RXD :- The data is Received by 8051 through the RXD pin.

INT0 & INT1 :- Interrupt 0 & Interrupt 1 are two Interrupt pins that are triggered by external circuits.

T<sub>0</sub> & T<sub>1</sub> :- The 8051 has two 16-bit Timers/Counters.

T<sub>0</sub>  $\rightarrow$  Timer 0 register (16-bit)

T<sub>1</sub>  $\rightarrow$  Timer 1 register (16-bit)

\* T<sub>0</sub> & T<sub>1</sub> Can be used either as Timers to generate a time delay or as Counters to count events happening outside the microcontroller.

### $\overline{RD}$ & $\overline{WR}$ :-

When  $\overline{RD} = 0$ , microcontroller reads the data from external RAM

When  $\overline{WR} = 0$ , microcontroller writes the data into external RAM.



NOTE :-

\* port 3 address is B0h

\* port 3 bit address

P3.7	P3.6	P3.5	P3.4	P3.3	P3.2	P3.1	P3.0
B7	B6	B5	B4	B3	B2	B1	B0

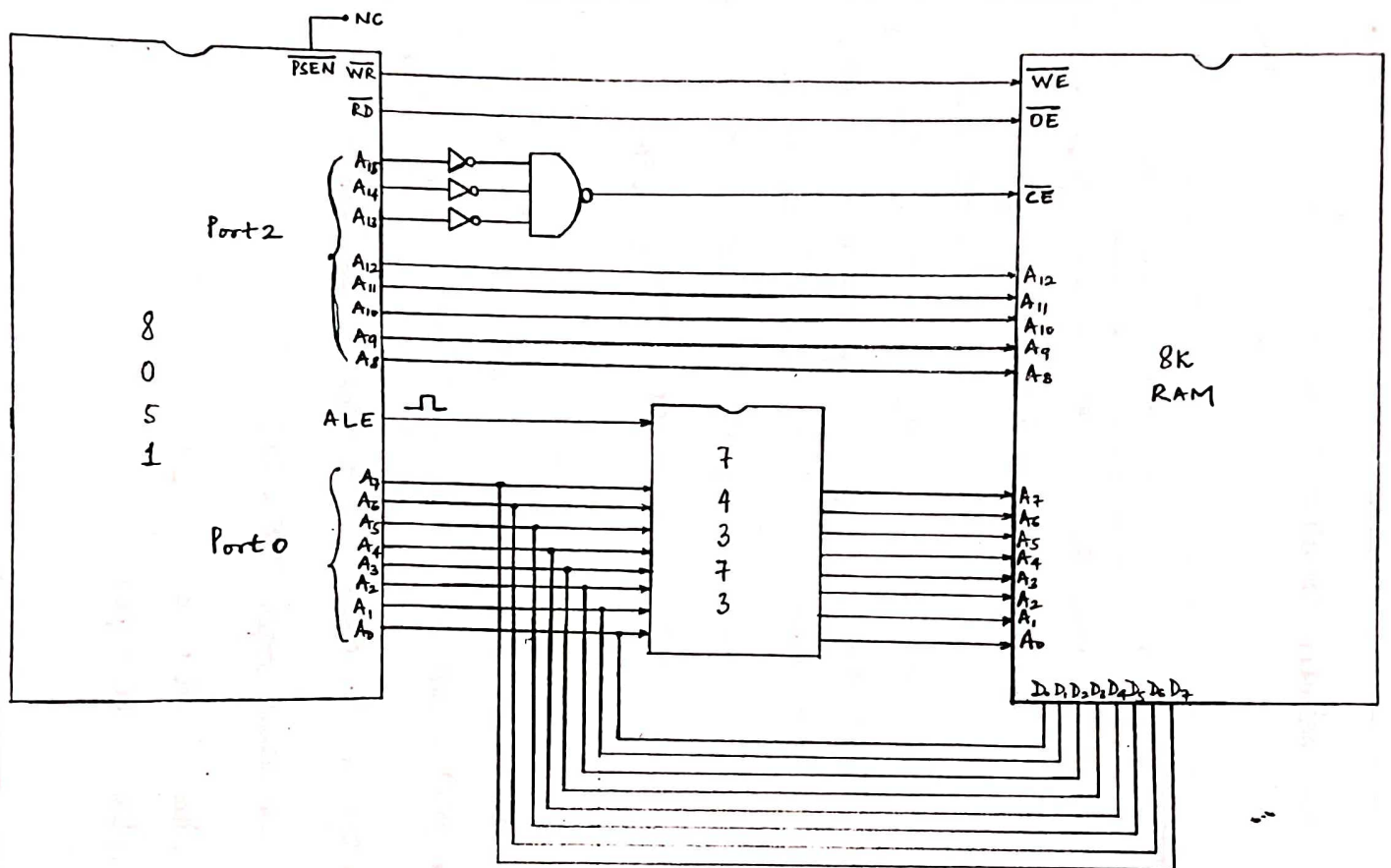


Fig: Interfacing 8K RAM to 8051 Microcontroller.



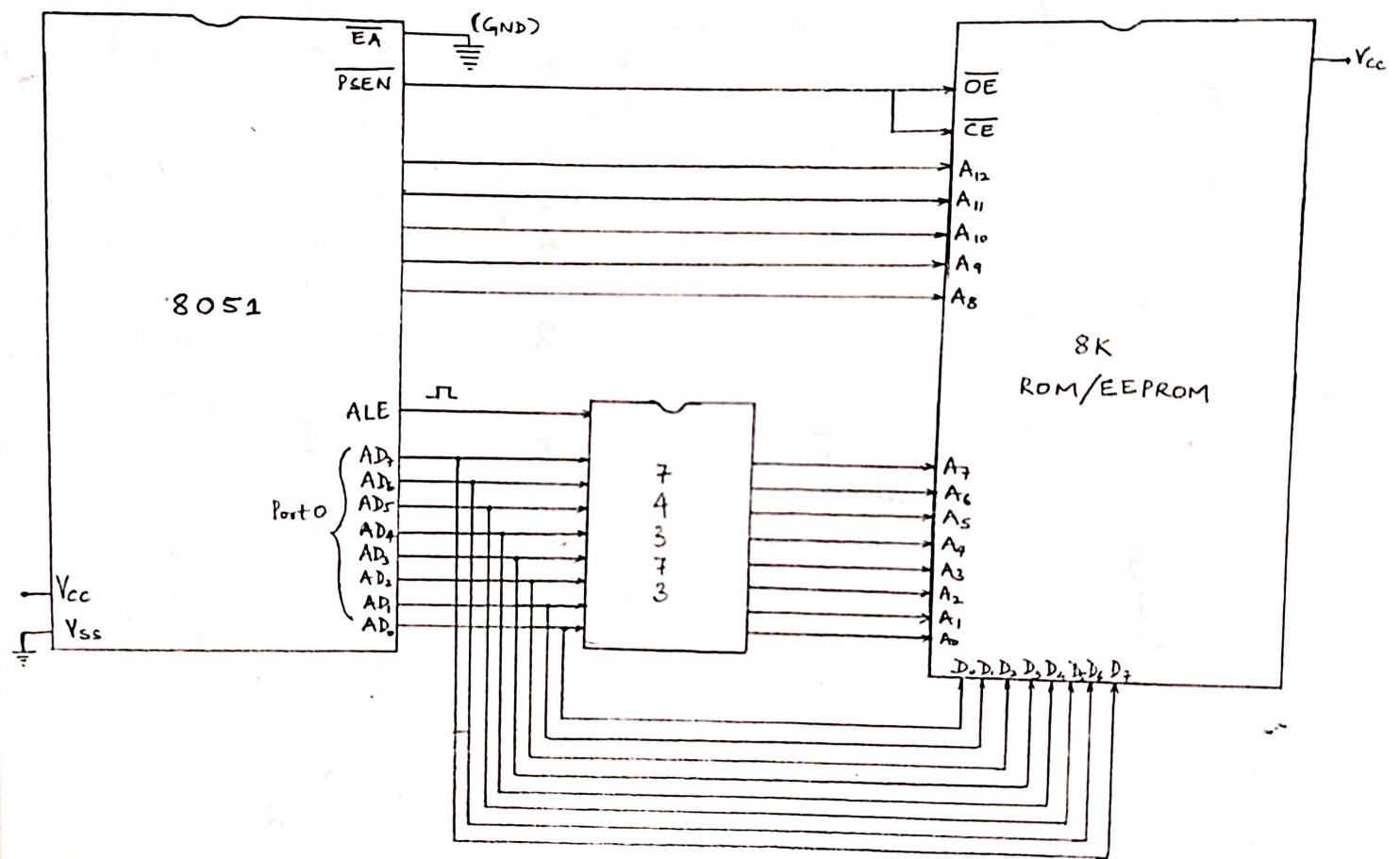


fig: Interfacing 8K ROM to 8051 Microcontroller.

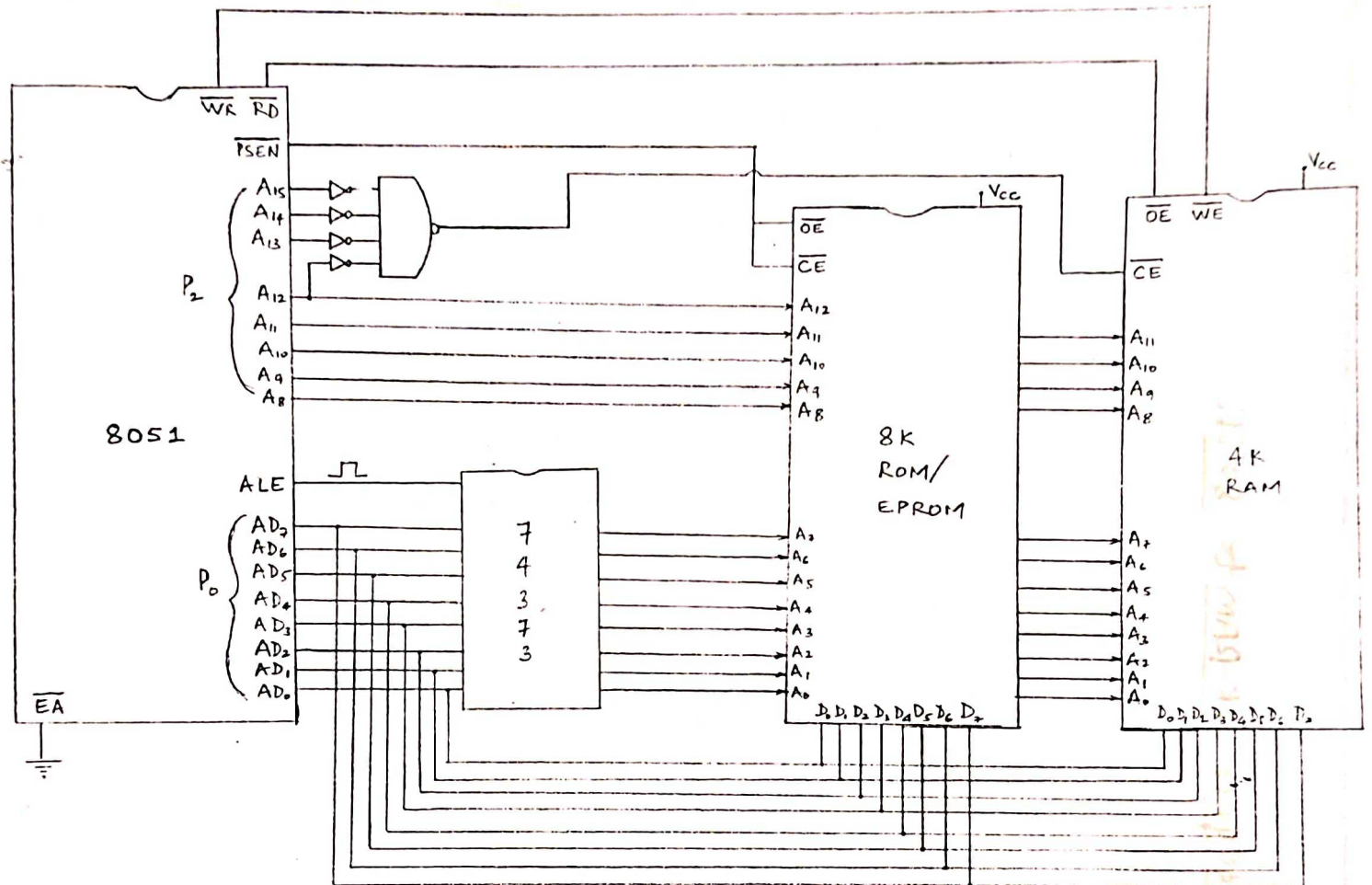


Fig: Interfacing 8K EPROM & 4K RAM to 8051 Microcontroller.



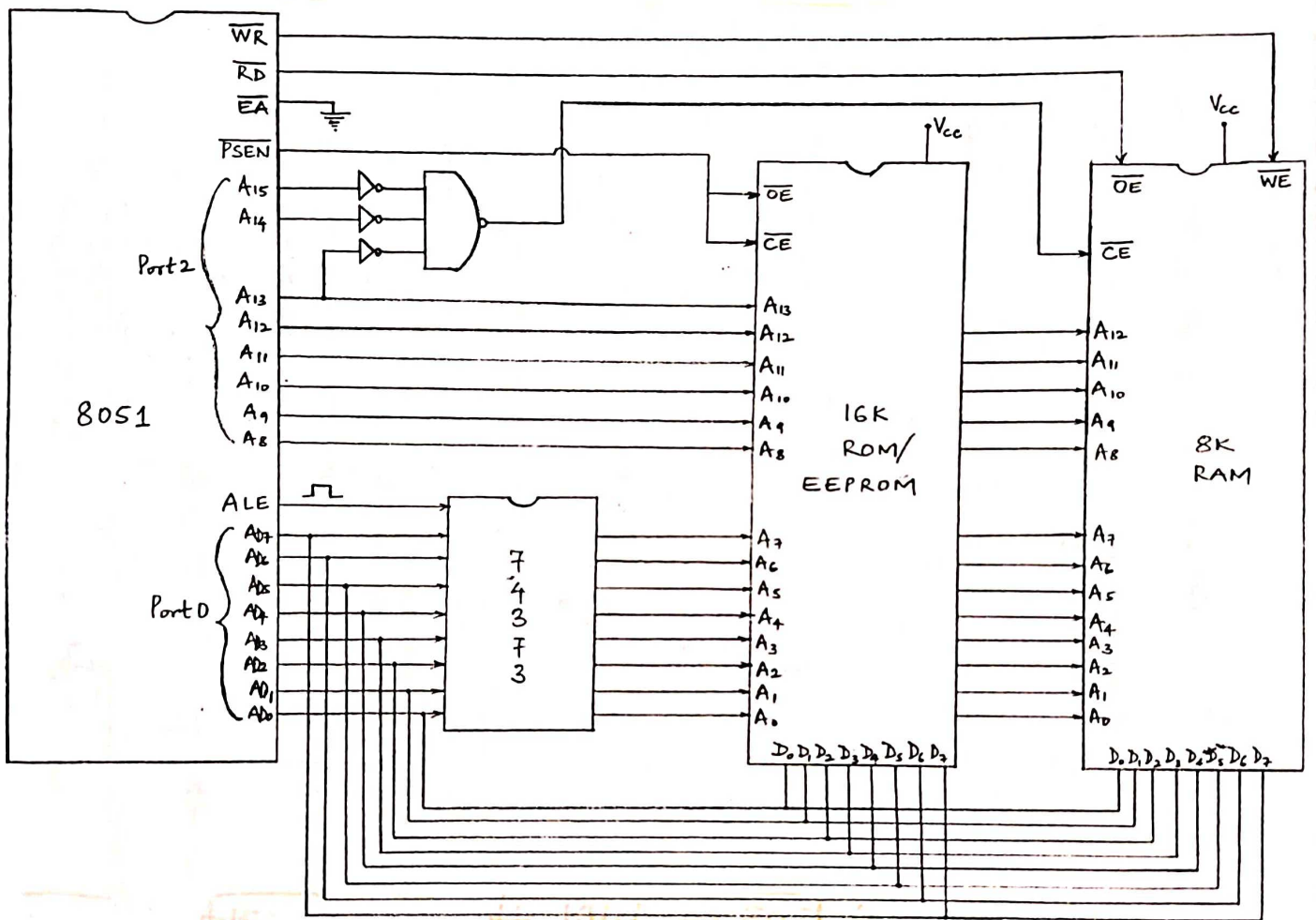
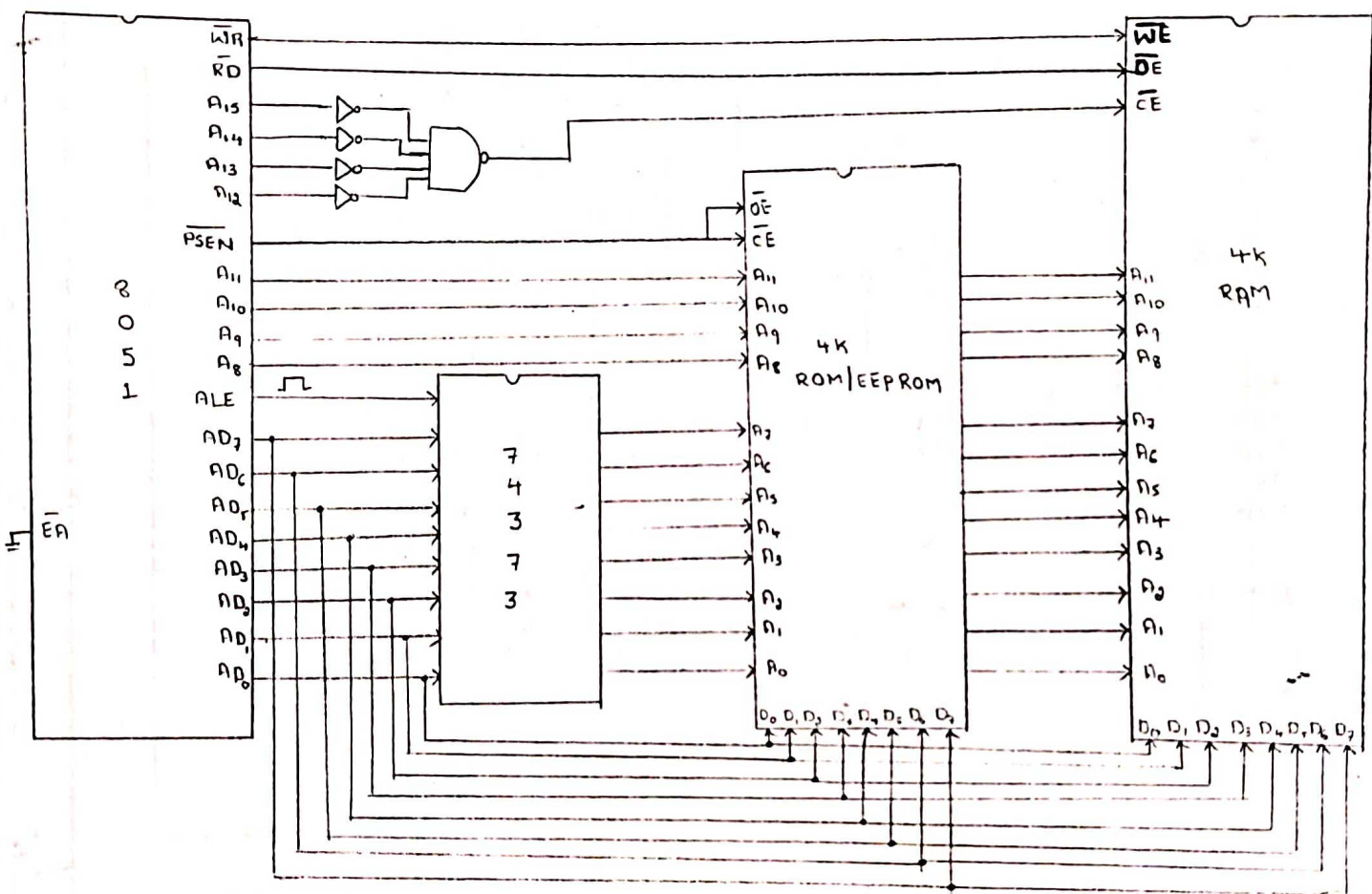


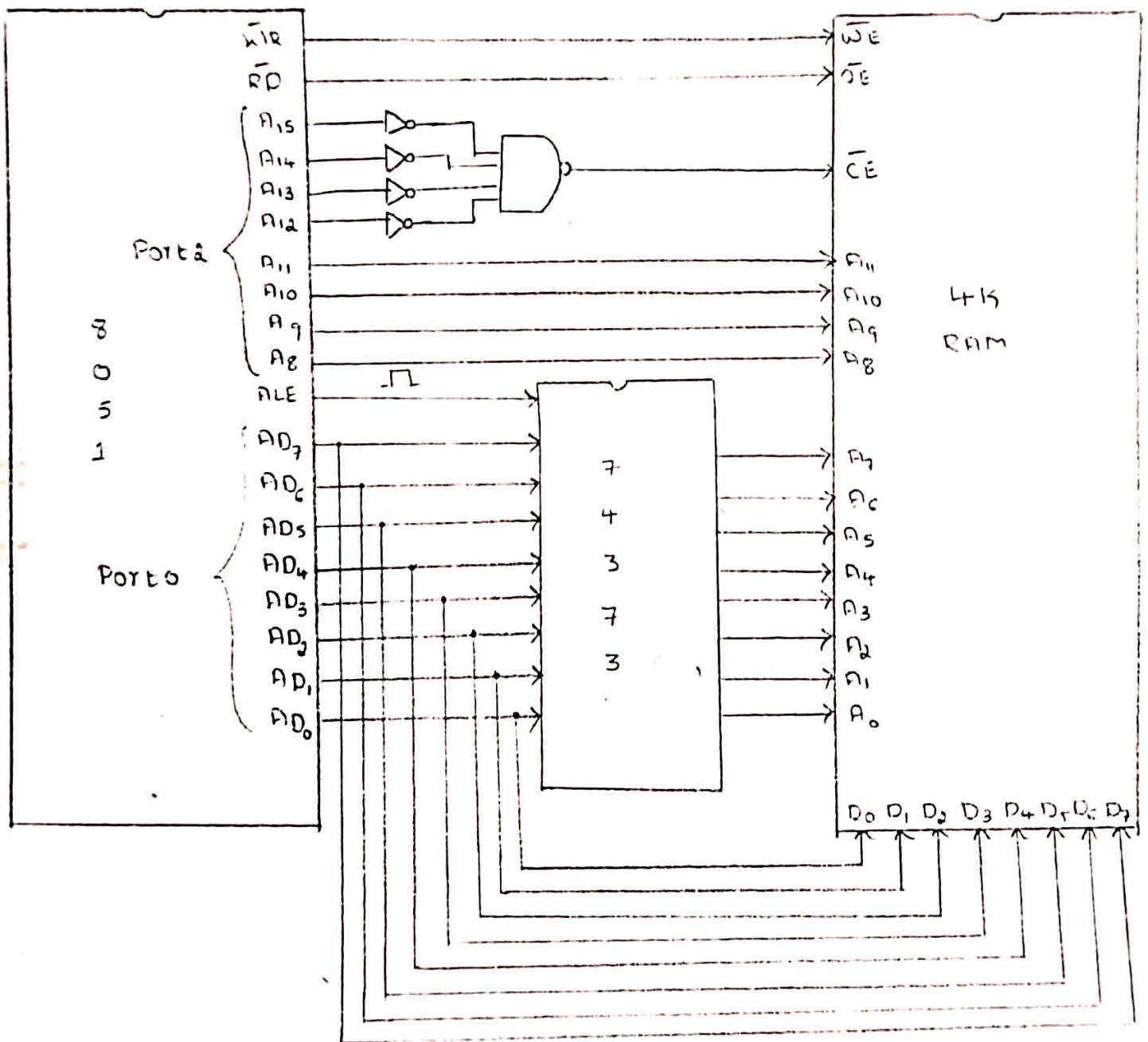
Fig: Interfacing 16K EPROM & 8K RAM to 8051 Microcontroller.

### 3. Interface 4K ROM and 4K RAM to 8051 :-

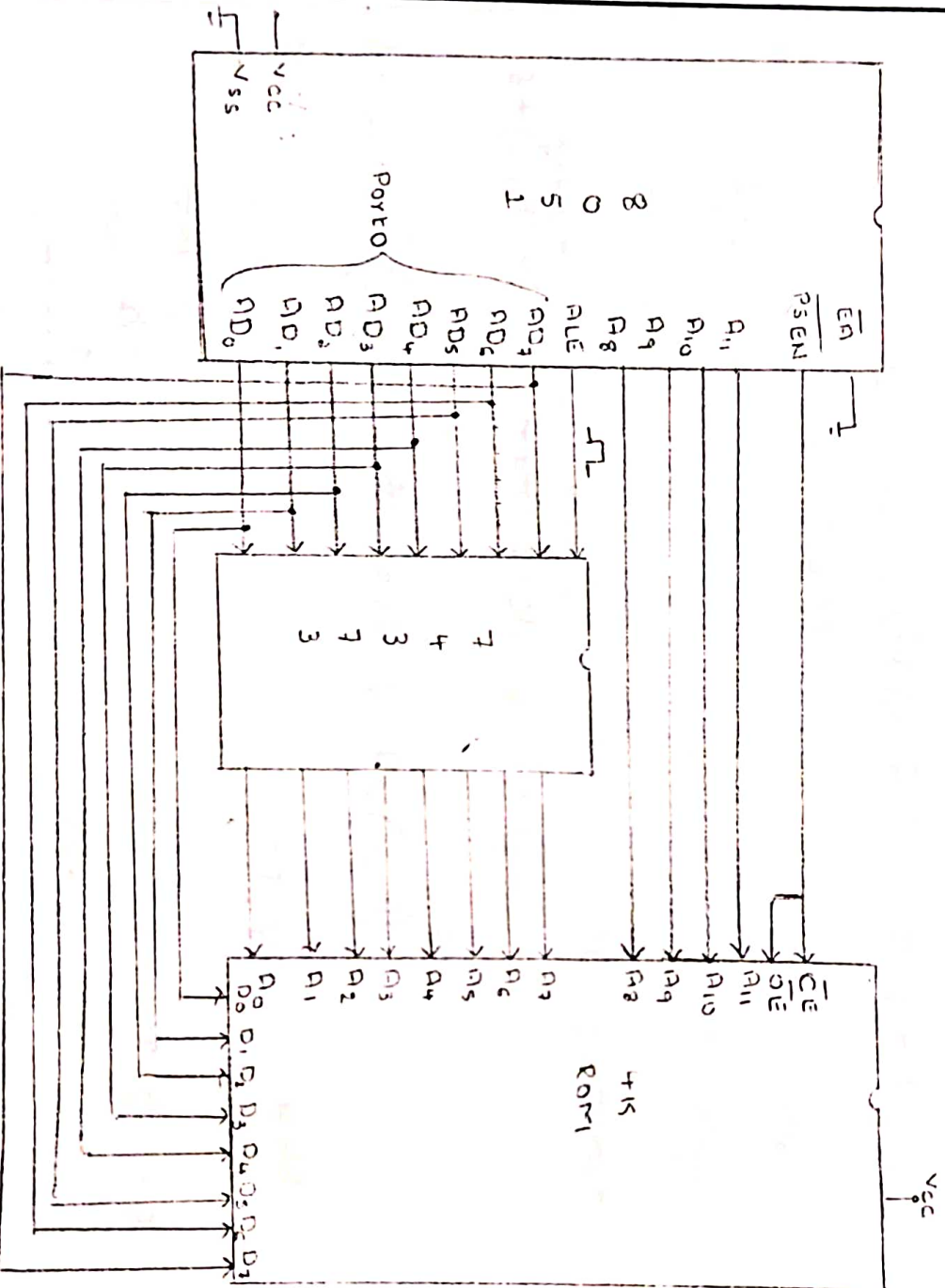




## 4.



# 5 Interfacing 4K-Rom to 8085



# **Minimum and Maximum Modes For 8086 Microprocessor**



# ***ROAD MAP***

- **General Bus Operation**
- **Minimum Mode configuration In 8086**
- **Maximum Mode Configuration In 8086**

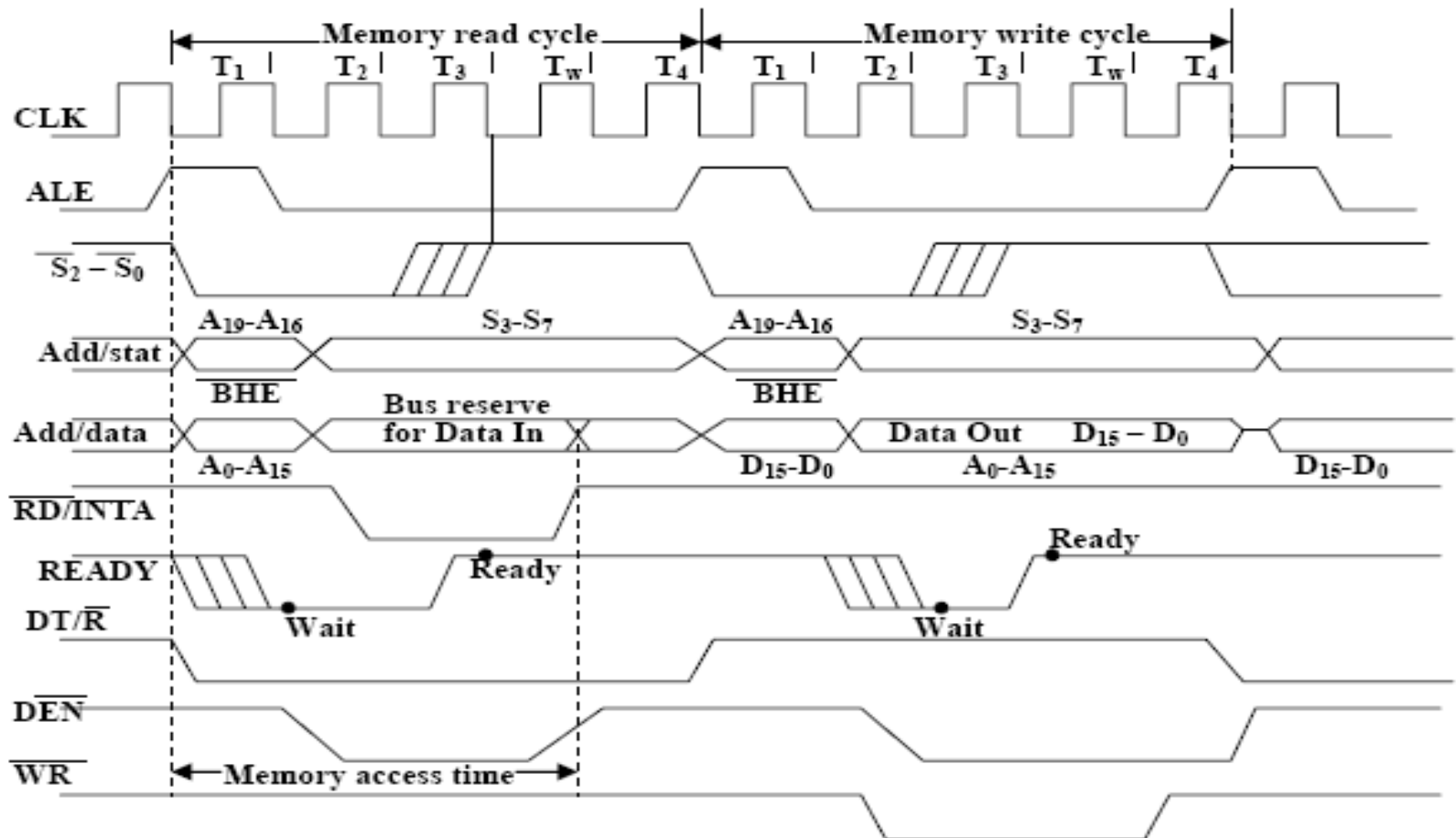
# General Bus Operation

- 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, when ever required.
- Basically, all the processor bus cycles consist of at least four clock cycles. These are referred to as T<sub>1</sub>, T<sub>2</sub>, T<sub>3</sub>, T<sub>4</sub>. The address is transmitted by the processor during T<sub>1</sub>. 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 S<sub>0</sub>, S<sub>1</sub> and S<sub>2</sub> are used to indicate the type of operation.
- Status bits S<sub>3</sub> to S<sub>7</sub> are multiplexed with higher order address bits and the BHE signal.
- Address is valid during T<sub>1</sub> while status bits S<sub>3</sub> to S<sub>7</sub> are valid during T<sub>2</sub> through T<sub>4</sub>.



# General Bus Cycle For 8086



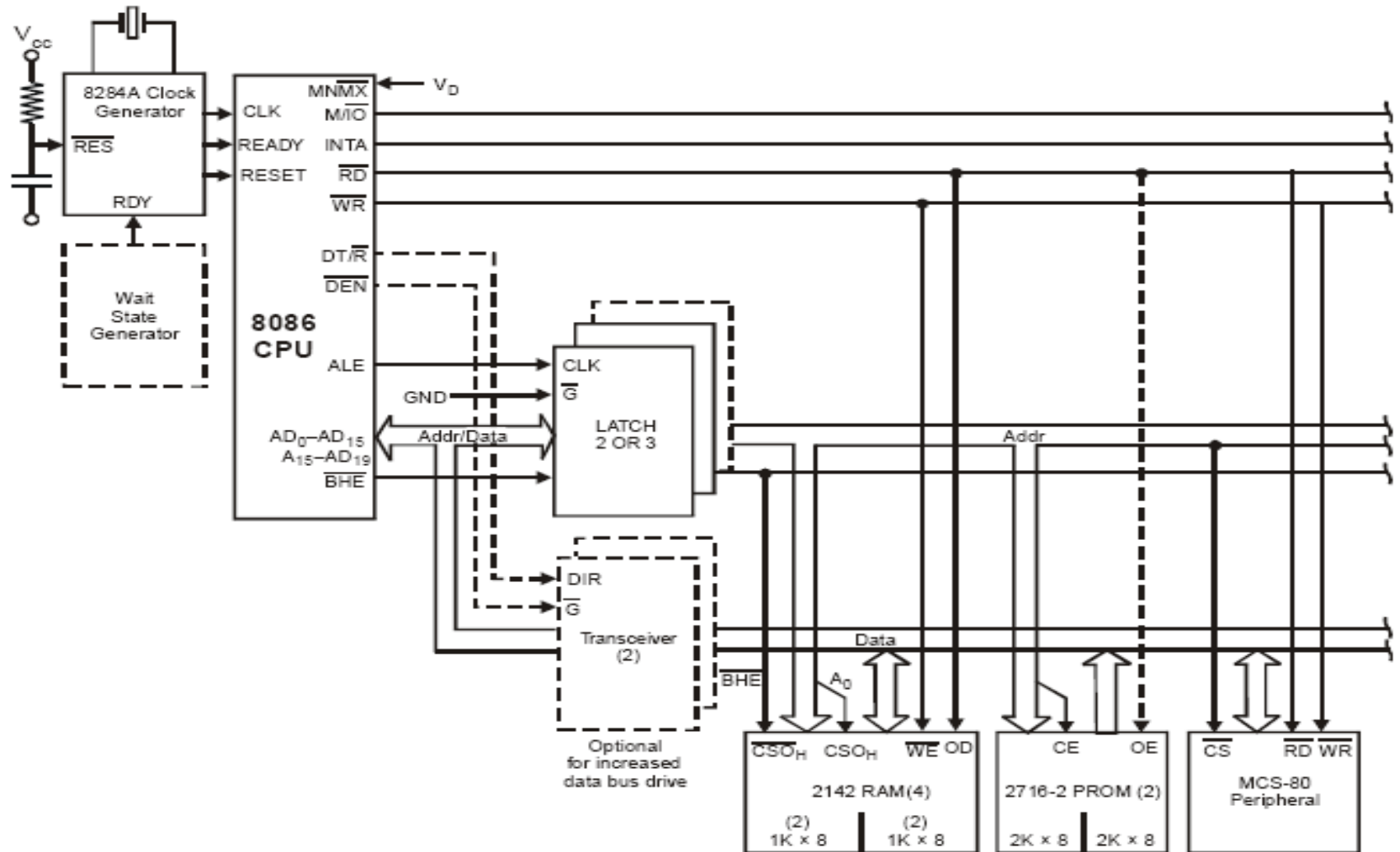
General Bus Operation Cycle in Maximum Mode

# Minimum Mode 8086 System

- The microprocessor 8086 is operated in minimum mode by strapping its MN/MX pin to logic 1.
- In this mode, all the control signals are given out by the microprocessor chip itself. There is a single microprocessor in the minimum mode system.
- The remaining components in the system are latches, transreceivers, clock generator, memory and I/O devices.
- Latches are generally buffered output D-type flip-flops like 74LS373 or 8282. They are used for separating the valid address from the multiplexed address/data signals and are controlled by the ALE signal generated by 8086.

# Minimum Mode Configuration For 8086

*Minimum Mode 8086 Configuration*



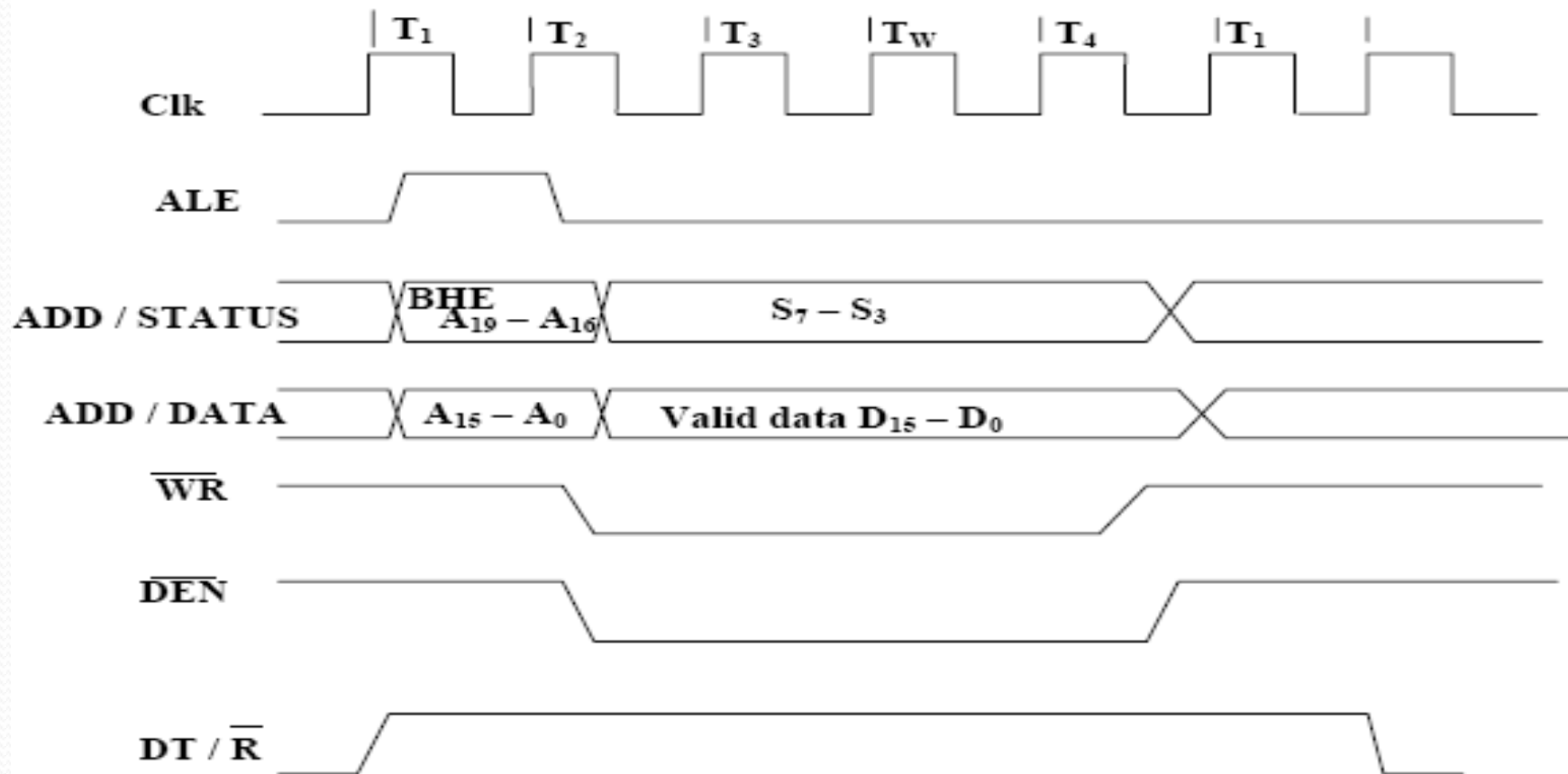


- Transreceivers are the bidirectional buffers and some times they are called as data amplifiers. They are required to separate the valid data from the time multiplexed address/data signals. They are controlled by two signals namely, DEN and DT/R.
- The DEN signal indicates the direction of data, i.e. from or to the processor.
- The system contains memory for the monitor and users program storage. Usually, EPROM are used for monitor storage, while RAM for users program storage. A system may contain 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.
- The read cycle begins in T<sub>1</sub> with the assertion of address latch enable (ALE) signal and also M / IO signal. During the negative going edge of this signal, the valid address is latched on the local bus.

- The BHE and Ao signals address low, high or both bytes. From T<sub>1</sub> to T<sub>4</sub>, the M/I $\bar{O}$  signal indicates a memory or I/O operation.
- At T<sub>2</sub>, the address is removed from the local bus and is sent to the output. The bus is then tristated. The read (RD) control signal is also activated in T<sub>2</sub>.
- The read (RD) signal causes the address device to enable its data bus drivers. After RD goes low, the valid data is available on the data bus.
- The addressed device will drive the READY line high. When the processor returns the read signal to high level, the addressed device will again tristate its bus drivers.

- 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 T<sub>2</sub>, after sending the address in T<sub>1</sub>, the processor sends the data to be written to the addressed location.
- The data remains on the bus until middle of T<sub>4</sub> state. The WR becomes active at the beginning of T<sub>2</sub> (unlike RD is somewhat delayed in T<sub>2</sub> to provide time for floating).
- The BHE and Ao signals are used to select the proper byte or bytes of memory or I/O word to be read or write.
- The M/IO, RD and WR signals indicate the type of data transfer as specified in table below.



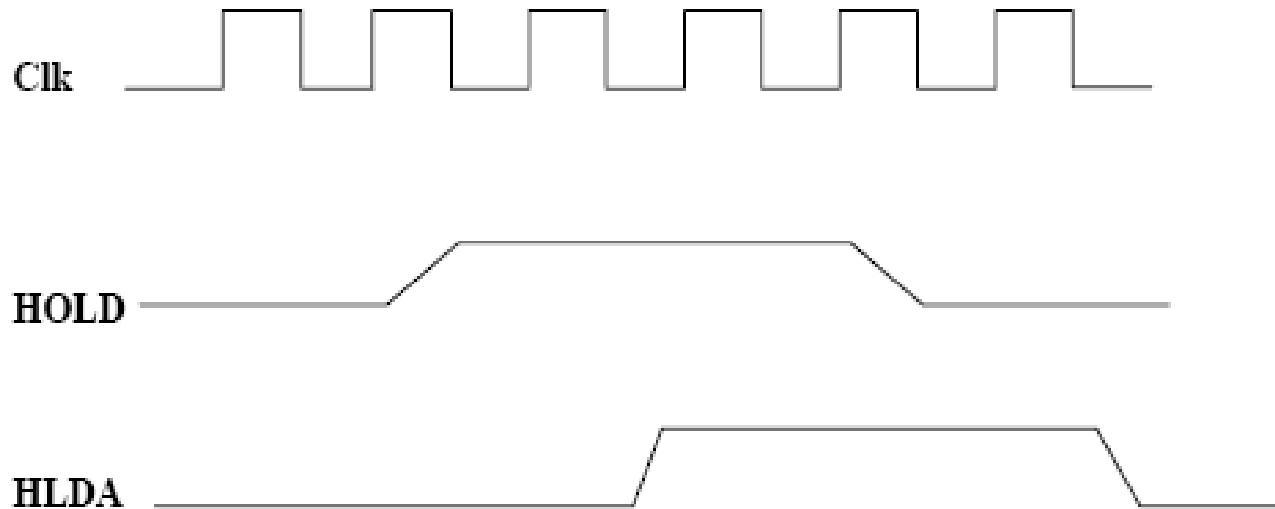


Write Cycle Timing Diagram for Minimum Mode

### ***Hold Response sequence:***

- The HOLD pin is checked at leading edge of each clock pulse. If it is received active by the processor before T<sub>4</sub> of the previous cycle or during T<sub>1</sub> state of the current cycle, the CPU activates HLDA in the next clock cycle and for succeeding bus cycles, the bus will be given to another requesting master.
- The control of the bus is not regained by the processor until the requesting master does not drop the HOLD pin low.
- When the request is dropped by the requesting master, the HLDA is dropped by the processor at the trailing edge of the next clock.

# ***Hold Response Timing Cycle***



Bus Request and Bus Grant Timings in Minimum Mode System



# Maximum Mode 8086 System

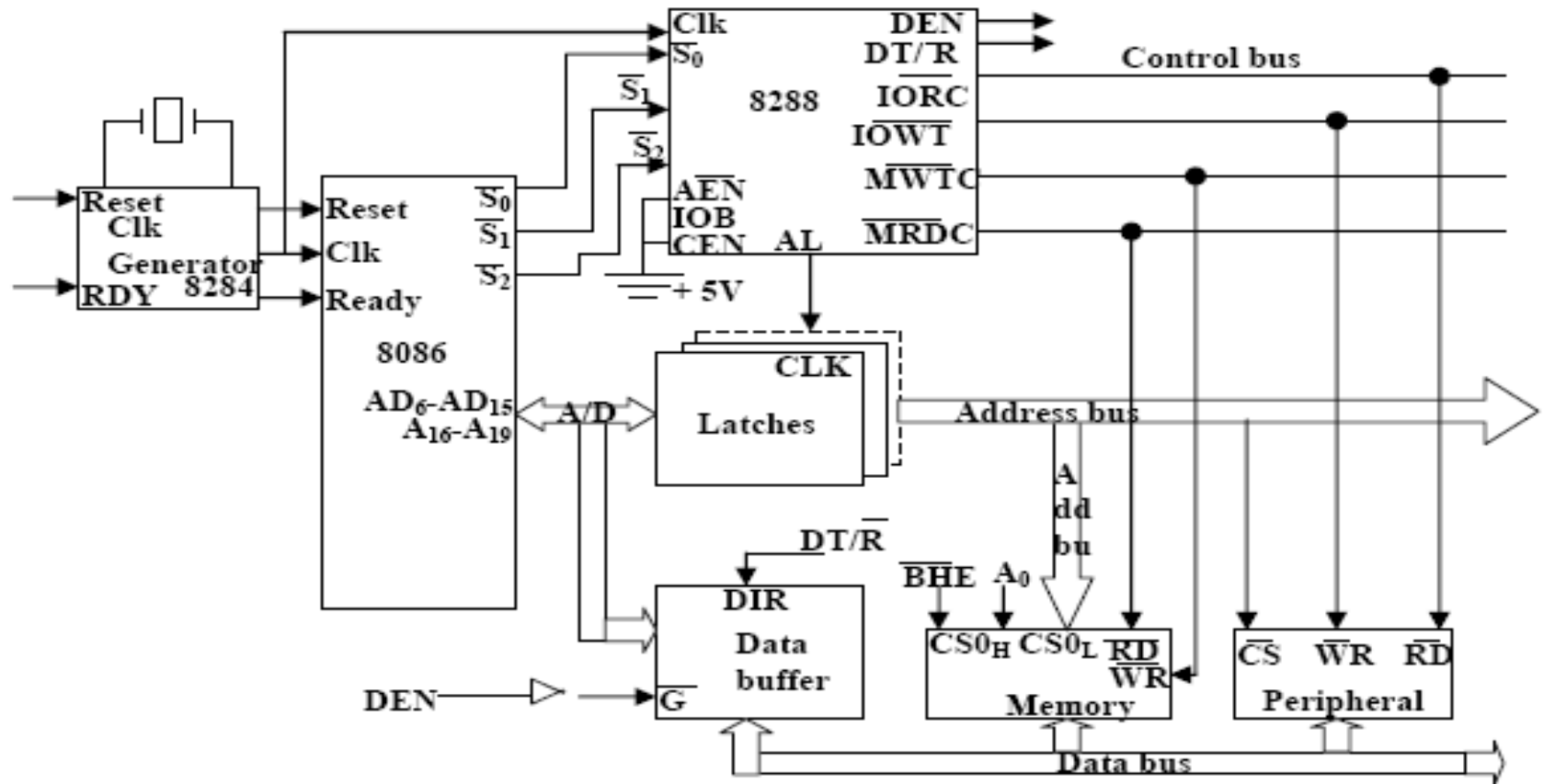
- In the maximum mode, the 8086 is operated by strapping the MN/MX pin to ground.
- In this mode, the processor derives the status signal S<sub>2</sub>, S<sub>1</sub>, S<sub>0</sub>. Another chip called bus controller derives the control signal using this status information .
- In the maximum mode, there may be more than one microprocessor in the system configuration. The components in the system are same as in the minimum mode system.
- The basic function 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 by the processor on the status lines.

- The bus controller chip has input lines S<sub>2</sub>, S<sub>1</sub>, S<sub>0</sub> and CLK. These inputs to 8288 are driven by CPU.
- It derives the outputs ALE, DEN, DT/R, MRDC, MWTC, AMWC, IORC, IOWC and AIOWC. The AEN, IOB and CEN pins are specially useful for multiprocessor systems.
- AEN and IOB are generally grounded. CEN pin is usually tied to +5V. The significance of the MCE/PDEN output depends upon the status of the IOB pin.
- INTA pin 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 address port.
- The MRDC, MWTC are memory read command and memory write command signals respectively and may be used as memory read or write signals.
- All these command signals instructs the memory to accept or send data from or to the bus.
- Here the only difference between in timing diagram between minimum mode and maximum mode is the status signals used and the available control and advanced command signals.

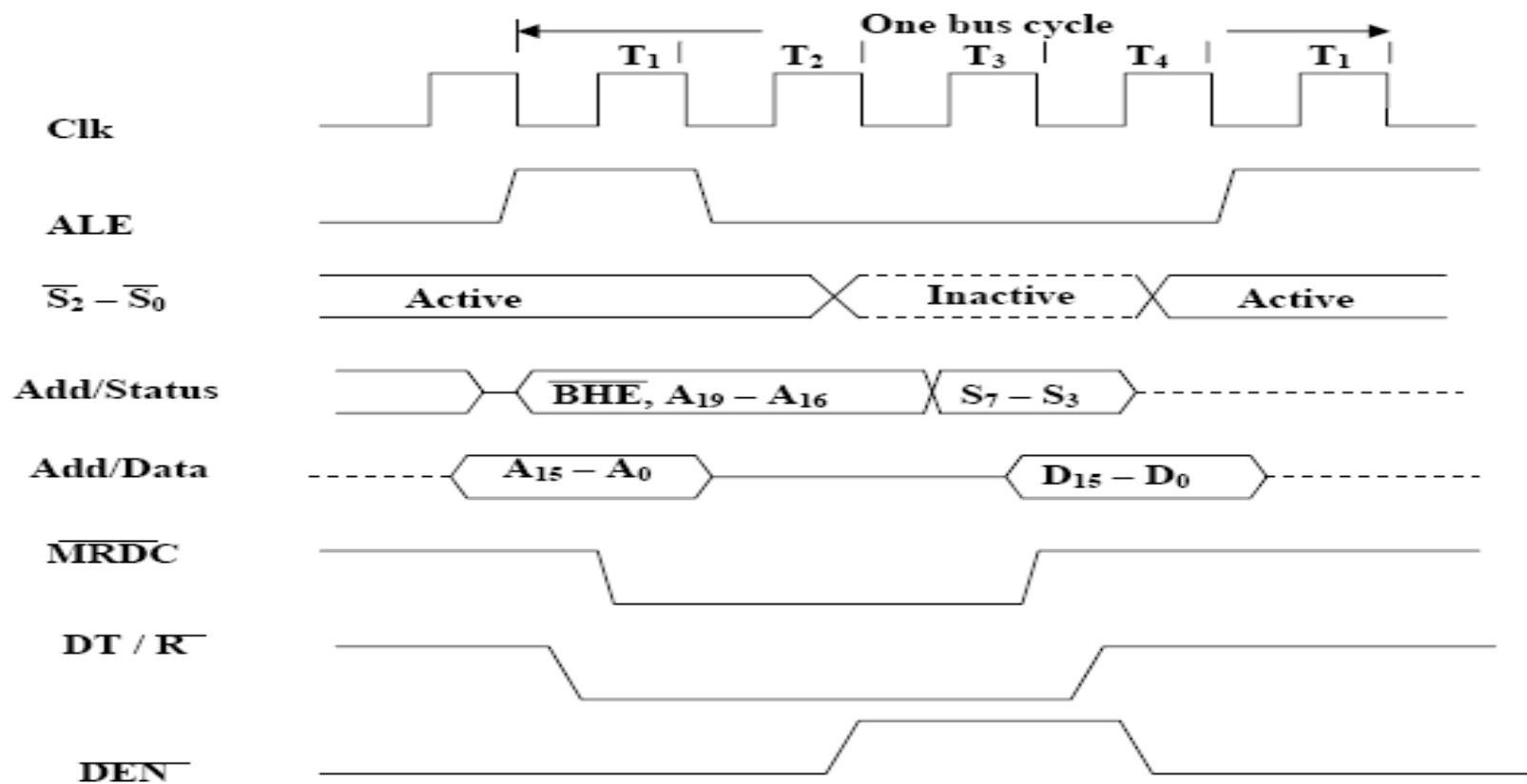


# Maximum Mode Configuration For 8086



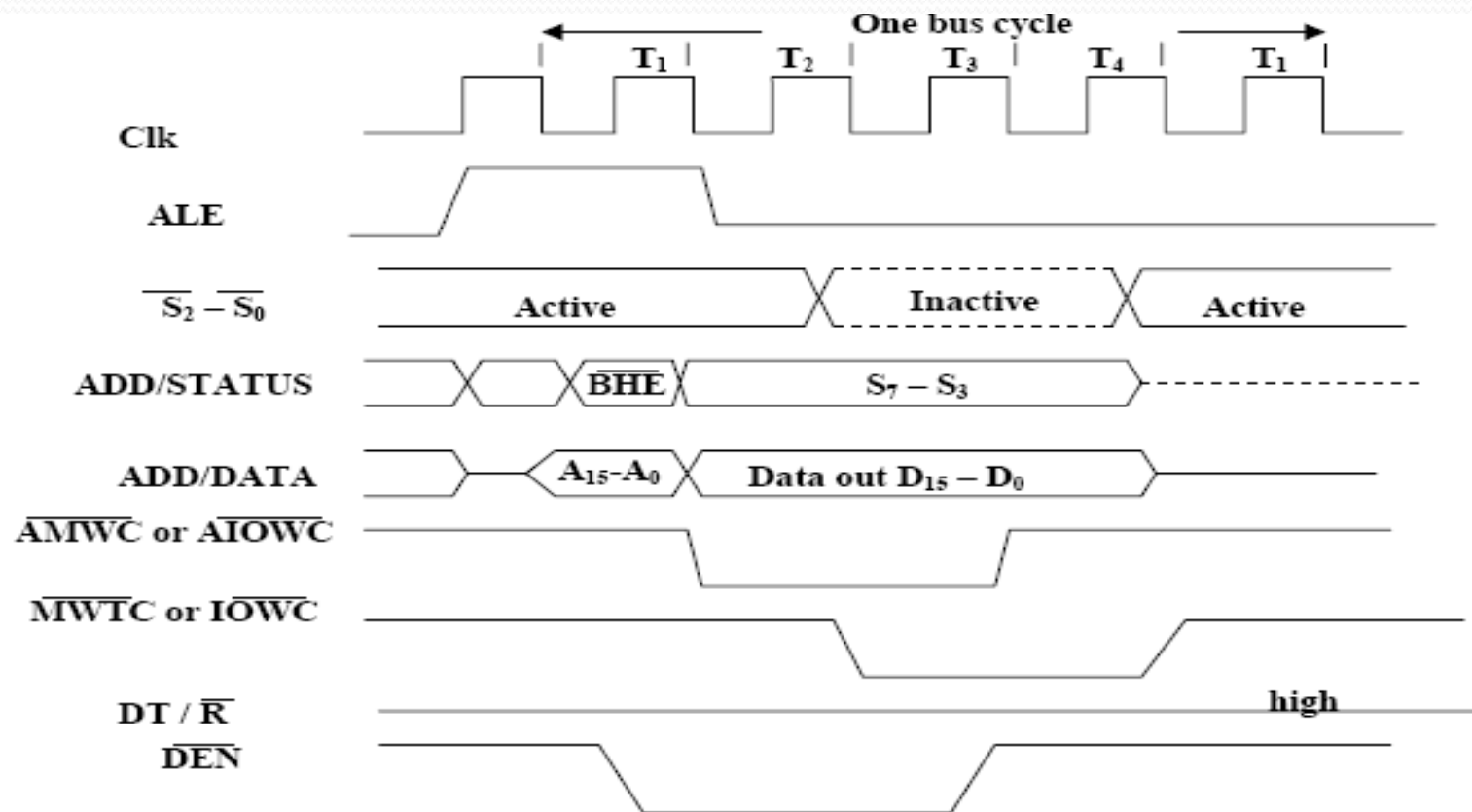
### Maximum Mode 8086 System.

- $R_0$ ,  $S_1$ ,  $S_2$  are set at the beginning of bus cycle. 8288 bus controller will output a pulse as on the ALE and apply a required signal to its DT / R pin during  $T_1$ .
- In  $T_2$ , 8288 will set  $DEN=1$  thus enabling transceivers, and for an input it will activate MRDC or IORC. These signals are activated until  $T_4$ .
- For an output, the AMWC or AIOWC is activated from  $T_2$  to  $T_4$  and MWTC or IOWC is activated from  $T_3$  to  $T_4$ .
- The status bit  $S_0$  to  $S_2$  remains active until  $T_3$  and become passive during  $T_3$  and  $T_4$ .
- If reader input is not activated before  $T_3$ , wait state will be inserted between  $T_3$  and  $T_4$ .



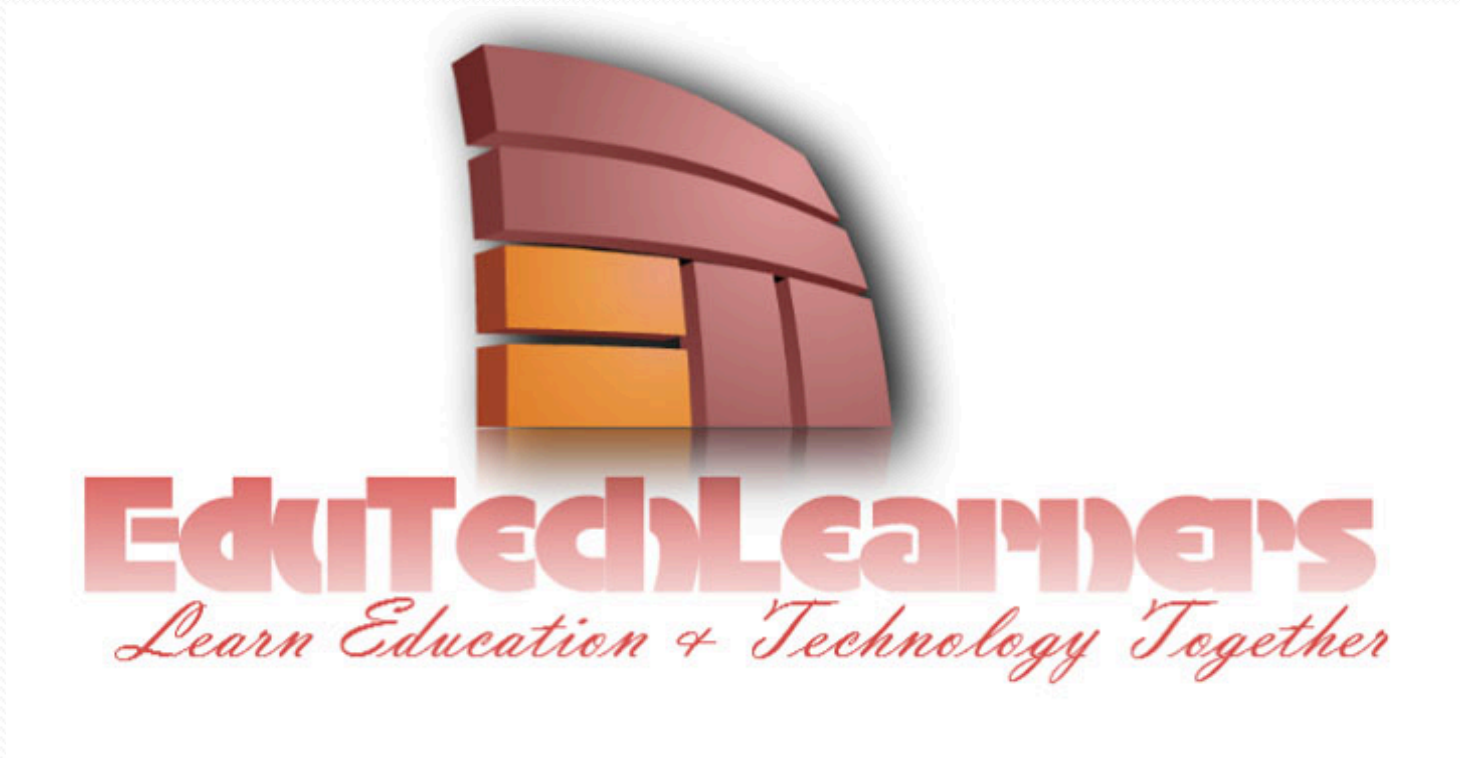
Memory Read Timing in Maximum Mode





Memory Write Timing in Maximum mode.

# THANKS!



For more Notes Follow <http://www.edutechlearners.com>

## **DOS FUNCTIONS AND INTERRUPTS** **(KEYBOARD AND VIDEO PROCESSING)**

The Intel CPU recognizes two types of interrupts namely hardware interrupt when a peripheral device needs attention from the CPU and software interrupt that is call to a subroutine located in the operating system. The common software interrupts used here are INT 10H for video services and INT 21H for DOS services.

### **INT 21H:**

It is called the DOS function call for keyboard operations follow the function number. The service functions are listed below:

#### **# 00H- It terminates the current program.**

- Generally not used, function 4CH is used instead.

#### **# 01H- Read a character with echo**

- Wait for a character if buffer is empty
- Character read is returned in AL in ASCII value

#### **# 02H- Display single character**

- Sends the characters in DL to display
- MOV AH, 02H
- MOV DL, 'A' ; move DL, 65
- INT 21H

#### **# 03H and 04H – Auxiliary input/output**

- INT 14H is preferred.

#### **# 05H – Printer service**

- Sends the character in DL to printer

#### **# 06H- Direct keyboard and display**

- Displays the character in DL.

#### **# 07H- waits for a character from standard input**





- does not echo

**# 08H- keyboard input without echo**

- Same as function 01H but not echoed.

**# 09H- string display**

- Displays string until '\$' is reached.
- DX should have the address of the string to be displayed.

**# 0AH – Read string****# 0BH- Check keyboard status**

- Returns FF in AL if input character is available in keyboard buffer.
- Returns 00 if not.

**# 0CH- Clear keyboard buffer and invoke input functions such as 01, 06, 07, 08 or 0A.**

- AL will contain the input function.

**INT 21H Detailed for Useful Functions****# 01H**

MOV, AH 01H; request keyboard input INT 21H

- Returns character in AL. IF AL= nonzero value, operation echoes on the screen. If AL= zero means that user has pressed an extended function key such as F1 OR home.

**# 02H**

MOV AH, 02H; request display character

MOV DL, CHAR; character to display

INT 21H

- Display character in D2 at current cursor position. The tab, carriage return and line feed characters act normally and the operation automatically advances the cursor.

**# 09H**

MOV Ah, 09H; request display

LEA DX, CUST\_MSG; local address of prompt

INT 21H

CUST\_MSG DB "Hello world", '\$'

- Displays string in the data area, immediately followed by a dollar sign (\$ or 24H), which uses to end the display.

**# 0AH**

MOV AH, 0AH ; request keyboard input

LEA DX, PARA\_LIST ; load address of parameter list

INT 21H

**Parameter list for keyboard input area :**

PARA\_LIST LABEL BYTE; start of parameter list

MAX\_LEN DB 20; max. no. of input character

ACT\_LEN DB ? ; actual no of input characters  
 KB-DATA DB 20 DUP (''); characters entered from keyboard

- LABEL directive tells the assembler to align on a byte boundary and gives location the name PARA\_LIST.
- PARA\_LIST & MAX\_LEN refer same memory location, MAX\_LEN defines the maximum no of defined characters.
- ACT\_LEN provides a space for the operation to insert the actual no of characters entered.
- KB\_DATA reserves spaces (here 20) for the characters.

### Example:

```
TITLE to display a string
.MODEL SMALL
.STACK 64
.DATA
    STR DB 'programming is fun', '$'
.CODE
MAIN PROC FAR
    MOV AX, @DATA
    MOV DS, AX
    MOV AH, 09H ;display string LEA
    DX, STR
    INT 21H
    MOV AX, 4C00H
    INT 21H
MAIN ENDP
END MAIN
```

### INT 10H

It is called video display control. It controls the screen format, color, text style, making windows, scrolling etc. The control functions are:

#### **# 00H – set video mode**

```
MOV AH, 00H      ; set mode
MOV AL, 03H      ; standard color text
INT 10H          ; call interrupt service
```

#### **# 01H- set cursor size**

```
MOV AH, 01H
MOV CH, 00H      ; Start scan line
MOV CL, 14H      ; End scan line
INT 10H          ; (Default size 13:14)
```

**# 02H – Set cursor position:**

```
MOV AH, 02H
MOV BH, 00H      ; page no
MOV DH, 12H      ; row/y (12)
MOV DL, 30H      ; column/x (30)
INT 10H
```

**# 03H – return cursor status**

```
MOV AH, 03H
MOV     BH,
00H; INT 10H
Returns: CH- starting scan line, CL-end scan line, DH- row, DL-column
```

**# 04H- light pen function****# 05H- select active page**

```
MOV AH, 05H
MOV AL,page-no.    ; page number
INT 10H
```

**# 06H- scroll up screen**

```
MOV AX, 060FH      ; request scroll up one line (text)
MOV BH, 61H        ; brown background, blue foreground
MOV CX, 0000H      ; from 00:00 through
MOV DX, 184F H     ; to 24:79 (full screen)
INT 10H
AL= number of rows (00 for full screen)
BH= Attribute or pixel value
CX= starting row: column
DX= ending row: column
```

**# 07H-Scroll down screen**

Same as 06H except for down scroll

**# 08H (Read character and Attribute at cursor)**

```
MOV AH, 08H
MOV BH, 00H      ; page number 0(normal)
INT 10H
AL= character
BH= Attribute
```

**# 09H -display character and attribute at cursor**

```
MOV AH, 09H
MOV AL, 01H      ; ASCII for happy face display
```



```
MOV BH, 00H      ; page number
MOV BL, 16H      ; Blue background, brown foreground
MOV CX, 60       ; No of repeated character
INT 10H
```

#### # 0AH-display character at cursor

```
MOV AH, 0AH
MOV AL, Char MOV
BH, page_no MOV
BL, value MOV CX,
repetition INT 10H
```

#### # 0BH- Set color palette

- ✓ Sets the color palette in graphics mode
  - ✓ Value in BH (00 or 01) determines purpose of BL
  - ✓ BH= 00H, select background color, BL contains 00 to 0FH (16 colors)
  - ✓ BH = 01H , select palette, BL, contains palette
- ```
MOV AH, 0BH
MOV BH, 00H; background      MOV BH, 01H ; select palette
MOV BL, 04H; red             MOV BL, 00H ; black
INT 21H                      INT 21H
```

#### #0CH- write pixel Dot

- Display a selected color
- AL=color of the pixel      CX= column  
BH=page number              DX= row

```
MOV AH, 0CH
MOV AL, 03
MOV BH, 0
MOV CX, 200
MOV DX, 50
INT 10H
```

It sets pixel at column 200, row 50

#### #0DH- Read pixel dot

- Reads a dot to determine its color value which returns in AL
- ```
MOV AH, 0DH
MOV BH, 0 ; page no
MOV CX, 80 ; column
MOV DX, 110 ; row
INT 10H
```

**#0EH- Display in teletype mode**

- Use the monitor as a terminal for simple display  
 MOV AH, 0EH  
 MOV AL, char  
 MOV BL, color; foreground  
 color INT 10H

**#0FH- Get current video mode**

Returns values from the BIOS video .

AL= current video mode      **MOV AH, 0FH**

AH= no of screen columns   **INT 10H**

BH = active video page

**TITLE To Convert letters into lower case**

.MODEL SMALL

.STACK 99H

.CODE

MAIN PROC

MOV AX, @ DATA

MOV DS, AX

MOV SI, OFFSER STR

M: MOV DL, [SI]

MOV CL, DL

CMP DL, ' \$'

JE N

CMP DL, 60H

JL L

K: MOV DL, CL

MOV AH, 02H

INT 21H

INC SI

JMP M

L: MOV DL, CL

ADD DL, 20H

MOV AH, 02H

INT 21H

INC SI

JMP M

N: MOV AX, 4C00H

INT 21H

MAIN ENDP

.DATA

STR DB 'I am MR Rahul ', '\$'

END MAIN

**TITLE to reverse the string**

```
.MODEL SMALL
.STACK 100H
.DATA
    STR1 DB " My name is Rahul" , '$'
    STR2 db 50 dup ('$')
.CODE
MAIN PROC FAR
    MOV BL,00H
    MOV AX, @ DATA
    MOV DS, AX
    MOV SI, OFFSER STR1
    MOV DI, OFFSET STR2
L2:  MOV DL, [SI]
     CMP DI, '$'
     JE L1
     INC SI
     INC BL
     JMP L2
L1:  MOV CL, BL
     MOV CH, 00H
     DEC SI
L3:  MOV AL, [SI]
     MOV [DI], AL
     DEC SI
     INC DI
     LOOP L3
     MOV AH,09H
     MOV DX, OFFSET STR2
     INT 21H
     MOV AX, 4C00H
     INT 21H
MAIN ENDP
END MAIN
```

**TITLE to input characters until 'q' and display**

```
.MODEL SMALL
.STACK 100H
.DATA
    STR db 50 DUP ('$')
.CODE
MAIN PROC FAR
```



```
        MOV AX, @ DATA
        MOV DS, AX
        MOV SI, OFFSET STR
L2:     MOV AH, 01H
        INT 21H
        CMP AL, 'q'
        JE L1
        MOV [SI], AL
        INC SI
        JMP L2
L1:     MOV AH, 09H
        MOV DX, OFFSET STR
        INT 21H
        MOV AX, 4C00H
        INT 21H
MAIN ENDP
END MAIN
```

# Serial Communication

## Communication:

Communication means exchange of meaningful information between transmitter & receiver i.e. source & destination.

Communication are mainly classified into

1. Serial communication
2. Parallel communication

## Serial communication:

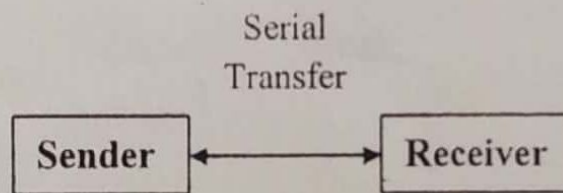


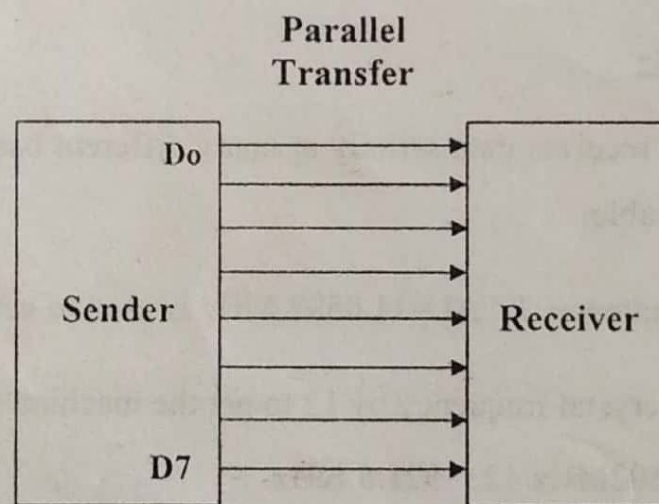
Fig 1 shows serial communication

- ❖ Serial communication uses **single data line** to transfer data.
- ❖ In serial communication **one bit** is transferred at a time over a single data line.
- ❖ Serial communication **enables two computers** located in two different cities to communicate over the **telephone**.
- ❖ Serial communication uses **single data line** instead of the **8-bit data line** of parallel communication makes it **much cheaper**.
- ❖ Serial communication is used for **long distance** communication

- ❖ In serial communication **data byte** (8-bit data) must be **converted** to **serial bits** using **parallel-in-serial out shift register**, and then it can be transmitted over a single data line.
- ❖ At the receiving end there must be a **serial-in-parallel out shift register** to receive the serial data & **pack** them into a **byte**.
- ❖ Serial communication is **slower than parallel** communication
- ❖ If the **data** is to be **transferred** on the **telephone line**, it must be converted from 1's & 0's to **AUDIO tones**, which are sinusoidal shape signals. This conversion is performed by a peripheral device called a **MODEM** i.e. "**MODULATOR / DEMODULATOR**".
- ❖ When the communication **distance is short**, the digital signal can be transferred it on a simple wire & requires **NO modulation**

Eg: - IBM keyboards transfer data to the motherboard.

### Parallel communication:



**Fig 2 Parallel communication**

- ❖ In parallel communication **number of lines required to transfer data depends on the number of bits to be transferred simultaneously**.
- ❖ The information is simply grabbed from the **8-bit data bus** of the **sender** presented (transferred) to the **8-bit data bus of the receiver**.



- ❖ Parallel communication works only for **shorter distance**.
- ❖ For **longer distance** communication long cables **diminish & even distorts signals**.
- ❖ The data transmission over a **long distance** using parallel communication is **impractical** due to **increase in cost of cabling**.
- ❖ Parallel communication is **faster than serial** communication.

Eg: - Data transmission from computer to printer.

### Serial Data transmission formats:-

The data in serial communication may be sent in two ways:

1. ASYNCHRONOUS
2. SYNCHRONOUS

### ASYNCHRONOUS Serial communication:-

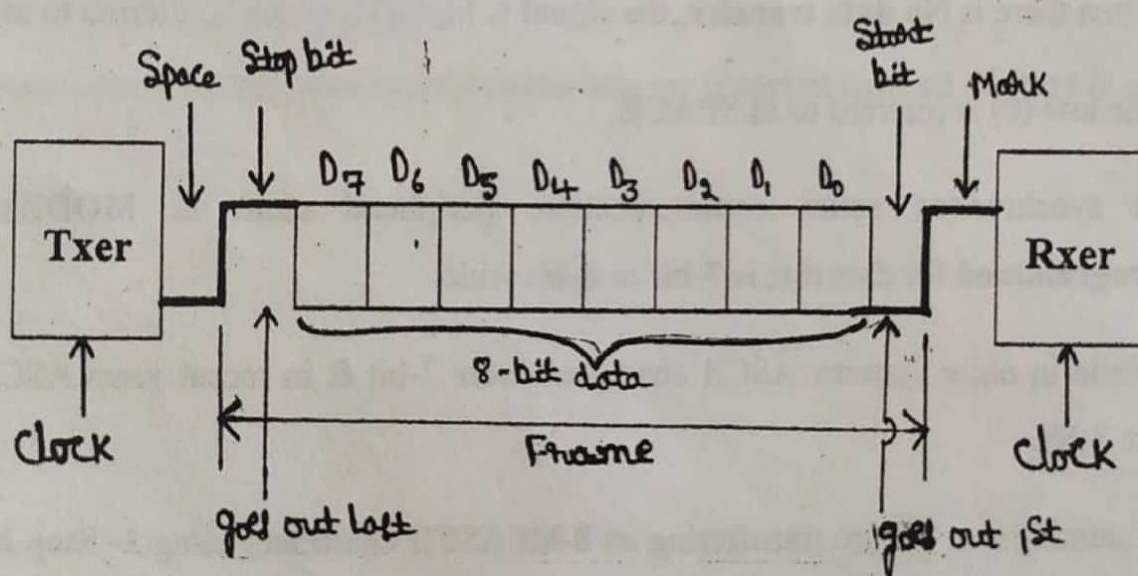
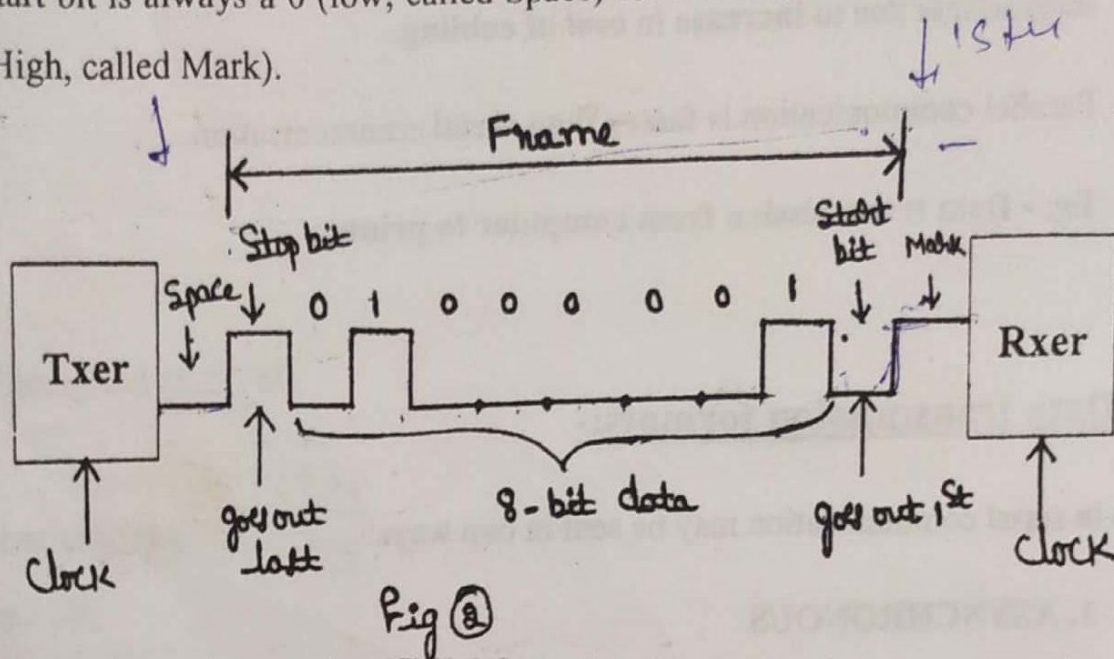


Fig 1 Asynchronous format.

- ❖ Asynchronous serial data communication is widely used for character-oriented transmission.
- ❖ Each character (data) is placed between start & stop bits as shown in figure. This is called Framing.

❖ The start bit is always one-bit, but the stop bit can be one or two-bits.

❖ The start bit is always a 0 (low, called Space) followed by a character & one or two stop bits (High, called Mark).



❖ In fig 2 the ASCII character "A" (8-bit binary 01000001 = 41h) is framed between the start bit & a single stop bit. The LSB is sent out first.

❖ When there is No data transfer, the signal is high (1), which is referred to as MARK.

❖ The low (0) is referred to as SPACE.

❖ In synchronous serial communication, peripheral chips & MODEMS can be programmed for data that is 7-bit or 8-bit wide.

❖ While in older systems ASCII characters were 7-bit & in recent years ASCII characters are 8-bit.

❖ Assuming that we are transferring an 8-bit ASCII characters using 1- Stop bit & 1- Start bit i.e. total of 10-bits for each character. Therefore for each 8-bit character there are extra 2-bits, which give 20% overhead.

❖ In some systems, the parity bit of the character byte is included in the data frame in order to maintain data Integrity.

i.e. for each 8-bit character we have a single parity bit in addition to start & stop bits.  
The parity bit is ODD or EVEN.



- ❖ In case of **ODD-parity** bit the number of **data bits**, including the **parity bit**, has an odd number of 1's.
- ❖ In case of **EVEN-parity** bit the number of **data bits**, including the **parity bit**, has an even number of 1's.

### Data Transfer Rates:

- ❖ The rate of data transfer in serial communication is stated in **bps** (bits per second). Another widely used terminology for bps is **Baud rate**.
- ❖ The **baud rate & bps** are not same. The **baud rate** is the **MODEM** terminology & is defined as the **number of signal changes per second**.
- ❖ In *modem* a **single change** of signal sometimes transfers several bits of data.
- ❖ For **conductor wire**, the **baud rate & bps** are the same. So for this reason we use the term bps & baud interchangeably.

### Baud rate in the 8051:-

- ❖ The 8051 transfers & receives data serially at many different baud rates. The baud rate in the 8051 is programmable.
- ❖ A standard crystal frequency,  $XTAL=11.0592 \text{ MHz}$  is used to generate the baud rate.
- ❖ The 8051 divides the crystal frequency by 12 to get the machine cycle frequency.  
i.e.  $XTAL/12 = 11.0592\text{MHz}/12 = 921.6 \text{ KHz}$ .

The 8051's serial communication **UART circuitry** divides the machine cycle frequency of 921.6 KHz by 32 i.e.  $921.6 \text{ KHz}/32 = 28,800 \text{ Hz}$ , then fed to the **Timer1** to set the baud rate as shown in below figure.

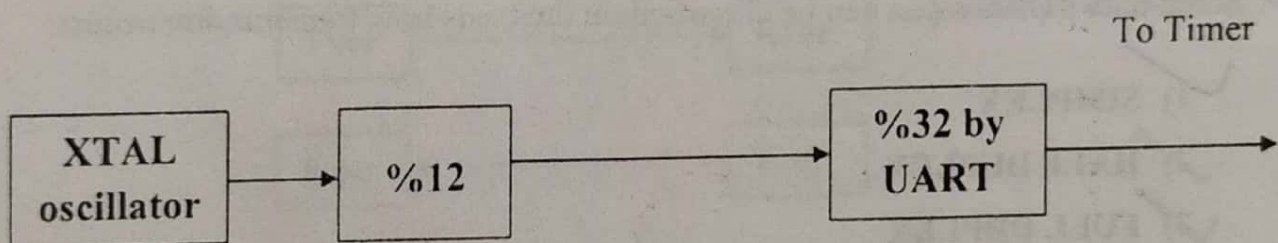
- ❖ The 8051 **baud rate** is set by **Timer1** using **Mode 2** (8-bit auto reload).
- ❖ To get the baud rates compatible with the PC, we must load **TH1** with the values shown in below table.



Table1: Timer1 TH1 register values for various Baud Rates.

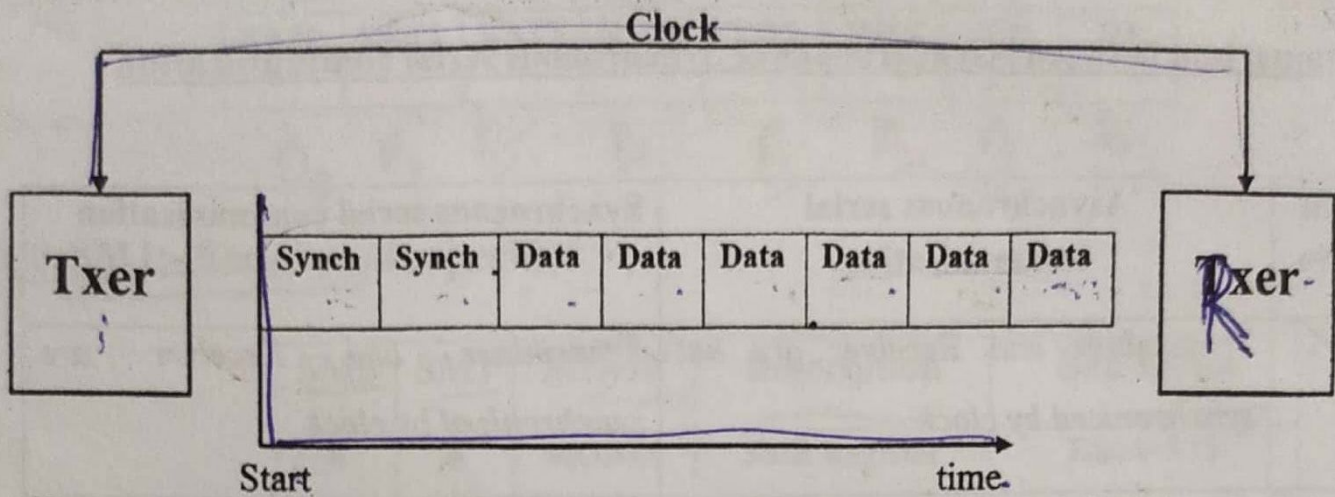
TH1		Baud Rate	
DECIMAL	HEX	SMOD = 0	SMOD = 1
-3	FD	9,600	19,200
-6	FA	4,800	9,600
-12	F4	2,400	4,800
-24	E8	1,200	2,400
<u>Note:</u> XTAL = 11.0592 MHz			

Note:



Baud rate:

## SYNCHRONOUS serial communication:-



*Fig1: Synchronous Transmission format.*

- ❖ The synchronous method **transfers a block of data (Character) at a time.**
- ❖ The start & stop bits in each frame of asynchronous format represents wasted overhead bytes that **reduces the overall character rate.**
- ❖ These start & stop bits can be eliminated by **synchronizing receiver & transmitter i.e. by having a common clock signal.**
- ❖ In synchronous transmission **synchronous bits are inserted instead of start & stop bits.**



## Comparison between Asynchronous & Synchronous serial communication

Sl No.	Asynchronous serial communication	Synchronous serial communication
1	Transmitter and Receiver are not synchronized by clock	Transmitter and Receiver are synchronized by clock
2	Bits of data are transmitted at constant rate	Data bits are transmitted with synchronization of clock
3	Character may arrive at any rate at receiver	Character is received at constant rate
4	Data transfer is character oriented	Data transfer takes place in <u>blocks</u>
5	Start & Stop bits are required to establish communication of each character	Start & Stop bits are not required to establish to communication of each character; however <u>synchronization bits</u> are required to transfer the data block.
6	Used in <b>LOW-SPEED</b> transmission at about speed less than <u>20 Kbs</u>	Used in <b>HIGH-SPEED</b> transmission

### Baud Rate & Transmission Rate:-

Baud rate is defined as the number of bits transmitted per second.

Eg: Consider a baud rate of 1200 then

**Transmission rate = 1 Sec / Baud rate.**

$$1 \text{ bit} = 1 \text{ sec} / 1200 = 0.83\text{ms.}$$

**0.83ms is the delay between two bits**



## PC Baud Rates:

### Note:-

*Some of the Baud rates supported by 486/Pentium IBM PC BIOS*

110	150	300	600	1200	2400	4800	9600	19200
-----	-----	-----	-----	------	------	------	------	-------

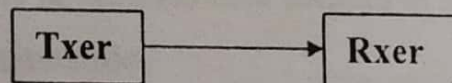
Imp.

### Serial Data Transmission classification:-

Serial data transmission can be classified on the basis how transmission occurs:

- ✓ 1) **SIMPLEX**
- ✓ 2) **HALF DUPLEX**
- ✓ 3) **FULL DUPLEX**

#### 1. Simplex:-



In simplex transmission data is transmitted in only one direction. There is no possibility of data transfer in other direction.

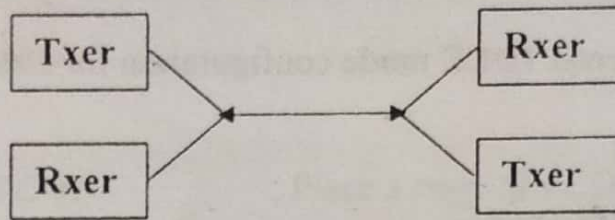
Eg: - From computer to printer.

{

**Duplex:** - In data transmission if the data can be transmitted and received, it is a duplex transmission.

}

## 2. Half Duplex:-



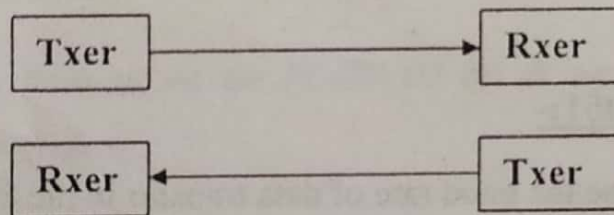
In half duplex, data transmission is possible only one way at a time.

OR

\* The half duplex transmission allows the data transfer in both direction, <sup>2)</sup> but not simultaneously. Eg: - Walkie-Talkie

\* There is no need of more than one wire (cables) <sup>here only one wire is used for</sup>  
3) Train in bridge

## 3. Full Duplex:-



Full duplex transmission allows the data transfer in both directions <sup>3)</sup> simultaneously.

Eg: - Transmission through Telephone lines.

\* There is need of two wire in the data transfer for data transmission <sup>separately</sup>.



## SCON (Serial control) Register:

SM0	SM1	SM2	REN	TB8	RB8	TI	RI
$D_7$	$D_6$	$D_5$	$D_4$	$D_3$	$D_2$	$D_1$	$D_0$

### SM0:SM1:- Serial mode specifier

<u>SM0</u>	<u>SM1</u>	<u>MODE</u>	<u>Description</u>	<u>Baud Rate</u>
0	0	MODE0	Shift Register	Baud=f/12
0	1	MODE1	8-bit UART	Baud= variable (Set by mode1,2)
1	0	MODE2	9-bit UART	Baud=f/12 of f/32
1	1	MODE3	9-bit UART	Baud= variable

Where f is the crystal frequency.

**SM2:-** Used for **multiprocessor communication**. In 8051 we are not using multiprocessor communication so SM2 is made 0 i.e. **SM2=0**

**REN: - Receive Enable bit.**

REN is set to 1 to **enable reception** (REN=1)

REN is cleared to 0 to **disable reception** (REN=0)

**TB8:- Transmit bit 8**

Set/cleared by hardware to **determine state** of the 9<sup>th</sup> data bit transmitted in 9-bit UART (In mode 2 & 3)

**RB8:- Receive bit 8.**

Set/cleared by hardware to **indicate state** of 9<sup>th</sup> data bit received. (In mode 2 & 3)

In 8051 mode is used, so these two bits are cleared i.e. **TB8=RB8=0**



## **TI: - Transmit interrupt flag**

Set by hardware at the beginning of the stop bit in Mode 1. (i.e. set by hardware whenever byte is transmitted)

TI must be cleared by software.

## **RI: - Receive interrupt flag.**

Set by hardware at the beginning of the stop bit in Mode 1. (i.e. set by hardware whenever byte is received)

RI must be cleared by software.

## **PCON (Power Mode Control) Special function register:-**

SMOD	-	-	-	GP1	GP0	PD	IDL
------	---	---	---	-----	-----	----	-----

(NOT bit addressable register)

### **SMOD:- Serial Baud Rate modify bit.**

Set to 1 by program to double the baud rate using Timer1 for mode 1, 2 & 3. (SMOD=1)

Cleared to 0 by program to use Timer1 Baud Rate (SMOD=0)

**Bit 6-4: Not implemented**

**GF1:- General purpose user flag bit 1**

Set/cleared by program

**GF0:- General purpose user flag bit 0**

Set/cleared by program.

**PD: - Power Down bit**

Set to 1 by program to enter **power down** configuration for CHMOS processors.

**IDL: - Idle mode bit.**

Set to 1 by program to enter **IDLE mode** configuration for CHMOS processors.

### Formula to compute BAUD Rate:-

$$\text{Baud Rate} = \frac{\text{Crystal Frequency}}{12 \times 32} \times \frac{1}{256 - \text{TH1}}$$

### Formula to compute Initial value for Particular Baud rate:-

$$\text{TH1} = 256 - \frac{\text{Crystal Frequency}}{12 \times 32 \times \text{Baud rate}}$$

### Doubling the Baud rate in 8051:-

There are two ways to increase the baud rate of data transfer in the 8051.

1. Use a Higher frequency crystal(Not Feasible)
2. **Change a bit in the PCON register.** (used by 8051 Microcontroller)

### Procedure for Doubling the baud rate in the 8051:-

❖ When the 8051 is powered up, **SMOD bit(D7)** of the PCON register is zero (i.e. SMOD=0)

❖ We can set it to **high** by software & thereby double the baud rate.

When

$$\text{SMOD} = 0: \quad \text{Baud Rate} = \frac{\text{Crystal Frequency}}{12 \times 32} \times \frac{1}{256 - \text{TH1}}$$

$$\text{SMOD} = 1: \quad \text{Baud Rate} = \frac{\text{Crystal Frequency}}{12 \times 16} \times \frac{1}{256 - \text{TH1}}$$



- ❖ The following sequence of instructions must be used to set high SMOD (D7) of PCON register.

MOV A, PCON ; Place a copy of PCON in ACC

SETB ACC.7 ; Make D7=1

MOV PCON, A ; Now SMOD=1, without changing any other bits

#### Note:

SETB ACC.7

MOV PCON, A

- *Now only SMOD is set & all other bits of PCON is cleared.*
- *Only we have to set the PCON D7 bit & we must not alter the other bits of PCON register.*

#### SBUF Register:-

- ❖ SBUF is an **8-bit** register used only for **serial communication** in the 8051.
- ❖ Whenever 8051 wants a **byte of data** to be transferred via **TxD** line, it must be placed in the **SBUF register**.
- ❖ Similarly **SBUF** holds the **byte of data**, when it is **received** by the 8051's **RxD** line.
- ❖ SBUF can be accessed like any other registers in the 8051.

Eg:-

1. MOV SBUF, #'D' ; load SBUF = 44h ASCII for 'D'
2. MOV SBUF, A ; copy Accumulator into SBUF
3. MOV A, SBUF ; copy SBUF into accumulator



## NOTE:-

- ❖ When a byte is written into SBUF, it is *framed* with the *start & stop bits* & transferred serially via the TxD pin
- ❖ Similarly when the bits are received serially via RxD, the 8051 *deframes* it by eliminating the *stop & start bits*, making a byte out of the data received, & then placing it in SBUF.

## Different Baud Rates:

Baud rate comparison for **SMOD=0 & SMOD=1**

TH1		Baud Rate	
DECIMAL	HEX	SMOD = 0	SMOD = 1
-3	FD	9,600	19,200
-6	FA	4,800	9,600
-12	F4	2,400	4,800
-24	E8	1,200	2,400
<u>Note:</u> XTAL = 11.0592 MHz			

## Procedure to program the 8051 to TRANSFER data serially:

In programming the 8051 to transfer character bytes serially, the following steps must be taken:

1. The **TMOD** register is loaded with the value **20H**, indicating the use of **TIMER1** in **mode 2** (8-bit auto-reload) to set the **baud rate**.
2. The **TH1** is loaded with one of the values four values to set the **baud rate** for serial data transfer (Assuming XTAL=11.0592MHz).
3. The **SCON** register is loaded with the value **50H**, indicating serial mode 1, where an **8-bit data** is framed with **start** and **stop bits**.
4. **TR1** is set to 1 to start Timer1.
5. **TI** is cleared by the "**CLR TI**" instruction.
6. The character byte to be transferred serially is written into **SBUF** register.
7. The **TI** flag bit is monitored with the use of the instruction "**JNB TI, label**" to see if the character has been transferred completely.
8. To transfer the next character, go to step 5.

### Importance of the TI Flag:

To understand the importance of the role of TI, look at the following sequence of steps that the 8051 goes through in transmitting a character via **TxD**

1. The **byte** character to be **transmitted** is written into **SBUF** register.
2. The **start bit** is **transferred**.
3. The **8-bit** character is **transferred one bit at a time**.
4. The **stop bit** is **transferred**. It is during the transfer of the stop bit that the 8051 **raises the TI flag (TI=1)**, indicating that the **last character** was **transmitted** and it is **ready to transfer the next character**.
5. By **monitoring the TI flag**, we make sure that we are **not overloading** the **SBUF** register.  
If we write another byte into SBUF register before TI is raised, the untransmitted portion



of the previous byte will be lost. In other words, when the 8051 finishes transferring a byte, it raises the TI flag to indicate it is ready for the next character.

6. After **SBUF** is loaded with a **new byte**, the **TI flag** bit must be forced to 0 by the "**CLR TI**" instruction in order for this **new byte** to be **transferred**.

### Procedure to program the 8051 to RECEIVE data serially:

In programming the 8051 to receive character bytes serially, the following steps must be taken:

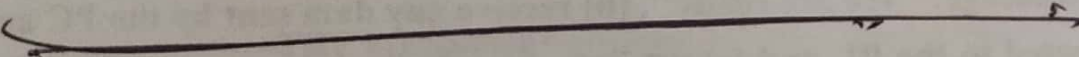
1. The **TMOD** register is loaded with the value **20H**, indicating the use of **TIMER1** in **mode 2** (8-bit auto-reload) to set the **baud rate**.
2. The **TH1** is loaded with one of the values four values to **set the baud rate** for serial data transfer (Assuming **XTAL=11.0592MHz**).
3. The **SCON** register is loaded with the value **50H**, indicating serial mode 1, where an **8-bit data** is framed with **start** and **stop** bits.
4. **TR1** is **set to 1** to start Timer1.
5. **RI** is cleared by the "**CLR RI**" instruction.
6. The **RI** flag bit is monitored with the use of the transmission "**JNB RI, label**" to see if an entire character has been received yet.
7. **When RI is raised, SBUF** has the byte. Its contents are moved into a safe place.
8. To receive the next character, go to step 5.



## Importance of the RI Flag:

In receiving bits via its **RxD pin**, the 8051 goes through the following steps:

1. It receives the **start bit** indicating that the **next bit** is **first bit of the character byte** it is about to receive.
2. The **8-bit character** is **received one bit at a time**. When the last bit is received, a byte it is about to receive.
3. The **stop bit** is received. When receiving the stop bit the 8051 makes **RI=1**, indicating that an entire character byte has been received and must be placed up before it gets overwritten by an incoming character.
4. By checking the **RI flag bit** when it is raised, we know that a character has been **received** and is **sitting** in the **SBUF register**. We copy the **SBUF** contents to a safe place in some other register or memory before it is lost.
5. After the **SBUF** contents are **copied** into a **safe place**, the RI flag bit must be forced to 0 by the "**CLR RI**" instruction in order to allow the next received character byte to be placed in **SBUF**. **Failure to do this causes loss of the received character.**

  
8255 Clp. Copy  
                      
Rs232  
Imp  
Imp

1. Write a program to transfer a letter 'Y' serially at 9600 baud continuously, and also to send a letter 'N' through port0, which is connected to a display device.

```
ORG 00h  
MOV TMOD, #20H      ; timer 1, mode 2  
MOV TH1, #-3        ; 9600 baud rate  
MOV SCON, #50H      ; 8 bits, 1stop, REN enabled  
SETB TR1            ; START TIMER 1
```

AGAIN:

```
MOV SBUF, #'Y'      ; transfer 'Y' serially
```

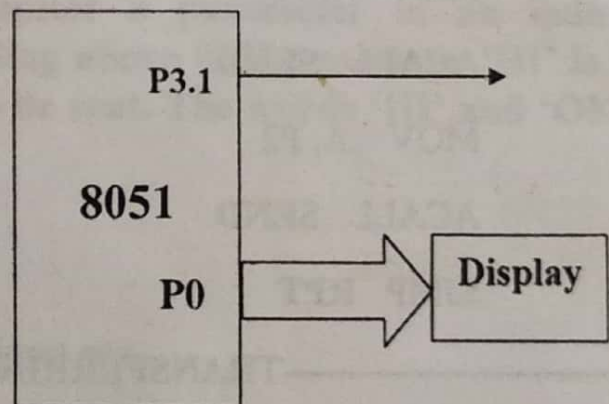
HERE: JNB TI, HERE ; WAIT FOR TRANSMISSION TO BE OVER

```
CLR TI              ; clear TI for next transmission
```

```
MOV P0, #'N'        ; move 'N' to p0 for parallel transfer
```

```
SJMP AGAIN          ; repeat
```

```
END
```



2. Take a data in through ports 0, 1 and 2, one after the other and transfer this data serial, continuously.

ORG 00h

MOV TMOD, #20H

MOV TH1, #-6

MOV SCON, #50H

MOV P0, #0FFH

MOV P1, #0FFH

MOV P2, #0FFH

SETB TR1

**RPT:** MOV A, P0

ACALL SEND

MOV A, P1

ACALL SEND

MOV A, P2

ACALL SEND

SJMP RPT

TRANSFERRING SERIALY



3. Write a program to receive the data which has been sent in serial form and send it out to port0 in parallel form. Also save the data at RAM location 60h.

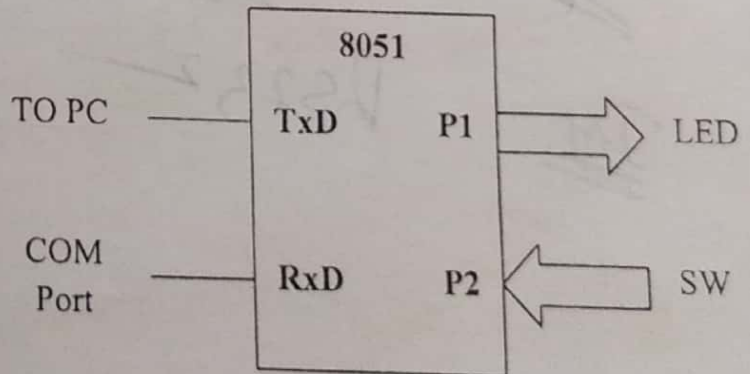
Sol:-

```
ORG 00h
MOV TMOD, #20H
MOV TH1, #-3
MOV SCON, #50H
SETB TR1
CLR RI
RPT: JNB RI, RPT
MOV A, SBUF
MOV P0, A
MOV 60H, A
END
```

4. Assume that the 8051 serial port is connected to the COM port of the IBM PC, and on the C we are using the hyperterminal program to send and receive data serially. P1 & P2 of the 8051 are connected to LEDs and switches, respectively. Write an 8051 program to (a) send to the PC the message " We are ready", (b) receive any data sent by the PC and put it on the LEDs connected to the P1, and (c) get data on switches connected to P2 and send it to the PC serially. The program should perform part (a) once, but parts (b) and (c) continuously. Use the 4800 baud rate.

Sol:-

```
ORG 00h
MOV P2, #0FFH
MOV TMOD, #20H
MOV TH1, #0FAH
MOV SCON, #50H
SETB TR1
MOV DPTR, #MYDATA
H_1: CLR A
MOVC A, @A+DPTR
```



JZ B\_1

ACALL SEND

INC DPTR

SJMP H\_1

B\_1: MOV A, P2

ACALL SEND

ACALL RECV

MOV P1, A

SJMP B\_1

; -----SERIAL DATA TRANSFER-----

SEND: MOV SBUF, A

H\_2: MOV T1, H\_2

CLR T1

RET

; -----RECIEVE DATA SERIALLY-----

RCV: JNB R1, RCV

MOV A, SBUF

CLR R1

RET

; -----THE MESSAGE-----

MYDATA: DB "WE ARE READY", 0

END

**Ex 10.10** A Square wave is being generated at pin p1.2 this square wave is to be sent to a receiver connected in serial form to this 8051. Write a program for this.

**Solution:**

Timer 0 in mode 2 is used to generate the square wave on P1.2. Whenever this pin is high, a data FFH is transmitted serially, and when this pin is low, a data 00H is transmitted. This data can be converted into parallel form at the receiver side to regenerate the square Wave there.

```
ORG    0000H

MOV    TMOD, #22H    ; timer 0 and timer 1 in mode 2

MOV    SCON, #50H

MOV    TH1, #-3

MOV    TH0, #00H    ; count value for timer 0

SETB   TR1          ; start timer 1

MOV    A, #00H      ; move A=00

CLR    P1.2

REPT: SETB   TRO          ; start timer 0

BACK: JNB   TF0, BACK; wait for timer 0 rollover

CPL    A            ; complement A

CPL    P1.2        ; complement P1.2

MOV    SBUF, A      ; move A to SBUF for transmission

CLR    TR0          ; stop timer 0

CLR    TF0          ; clear Timer 0 flag

HERE: JNB   TI, HERE    ; check for TI flag

CLR    TI           ; clear TI to enable next transmission

SJMP   REPT        ; repeat the whole process

END
```



Eg10-12) Write a program to send the text string "Hello" to serial #1. Set the baud rate at 9600, 8-bit data, and 1 stop bit.

Solution:

```
SCON1    EQU 0C0H
SBUF1    EQU 0CIH
TI1      BIT  0C1H
ORG      00H                ; starting position
MOV      TMOD, #20H
MOV      TH1, #-3           ; 9600 baud rate
MOV      SCON1, #50H
SETB     TR1
MOV      DPTR, #MESS1      ; display "Hello"
FN: CLR   A
MOV      A, @A+DPTR        ; read value
JZ       S1                 ; check for end of line
ACALL    SENDCOM2          ; send to serial port
INC      DPTR              ; move to next value
SJMP     FN
S1: SJMP  S1
```

**SENDERCOM2:**

```
MOV      SBUF1, A          ; place value in buffer
HERE1:   JNB     TI1, HERE1 ; wait until transmitted
CLR      TI1              ; clear
RET
MESS1:   DB      "Hello", 0
END
```

Eg10-13) Program the second serial port of the DS89C4x0 to receive bytes of data serially and output them on P1. Set the baud rate at 4800, 8-bit data, and 1 stop bit.

**Solution:**

```
SBUF1    EQU    0C1H    ; second serial SBUF addr
SCON1    EQU    0C0H    ; second serial SCON addr
RI1      BIT    0C0H    ; second serial RI bit addr
ORG      0H            ; starting position
MOV      TMOD, #20H    ; COM2 uses Timer 1 upon reset
MOV      TH1, #-6      ; 4800 baud rate
MOV      SCON1, #50H    ; COM2 has its own SCON1
SETB     TR1           ; start Timer 1
HERE:   JNB     RI1, HERE    ; wait for data to come in
          MOV     A, SBUF1    ; save data
          MOV     P1, A      ; display on P1
          CLR     RI1
          SJMP    HERE
          END
```

END

10-15) write a C program for the 8051 to transfer the letter "A" serially at 4800 baud continuously. Use 8-bit data and 1 stop bit.

```
#include<reg51.h>

void main (void)
{
    TMOD=0x20;           //use Timer 1, 8-BIT auto-reload
    TH1=0XFA;            //4800 baud rate
    SCON=0x50;
    TR1=1;
    while (1)
    {
        SBUF='A';        //place value in buffer
        while (T1==0);
        TI=0;
    }
}
```



10-16) write an 8051 C program to transfer the message "YES" serially at 9600 baud, 8-bit data, 1 stop bit. Do this continuously.

```
#include<reg51.h>
```

```
void SerTx (unsigned char);
```

```
void main (void)
```

```
{
```

```
    TMOD=0x20;
```

```
    TH1=0XFD;
```

```
//9600 baud rate
```

```
    SCON=0x50;
```

```
    TR1=1;
```

```
//start timer
```

```
    while (1)
```

```
    {
```

```
        SerTx ('Y');
```

```
        SerTx ('E');
```

```
        SerTx ('S');
```

```
    }
```

```
}
```

```
void SerTx (unsigned char x)
```

```
{
```

```
    SBUF=x;
```

```
//place value in buffer
```

```
    while (TI==0);
```

```
//wait until transmitted
```

```
    TI=0;
```

```
}
```

10-17) Program the 8051 in C to receive bytes of data serially and put them in P1.

Set the baud rate at 4800, 8-bit data, and 1 stop bit.

```
#include<reg51.h>
```

```
void main (void)
```

```
{
```

```
    unsigned char mybyte;
```

```
    TMOD=0x20;           //use Timer 1, 8-BIT auto-reload
```

```
    TH1=0xFA;           //4800 baud rate
```

```
    SCON=0x50;
```

```
    TR1=1;              //start timer
```

```
    while (1)           //repeat forever
```

```
    {
```

```
        while (RI==0);  //wait to receive
```

```
        mybyte=SBUF;    //save value
```

```
        P1=mybyte;      //write value to port
```

```
        RI=0;
```

```
    }
```

```
}
```

1) Write an 8051 assembly language program to transfer letter "G" serially at 9600 baud rate, continuously.

```
ORG 00H
MOV TMOD, #20H           ; timer 1, mode 2
MOV TH1, #-3             ; 9600 baud rate or FDh
MOV SCON, #50H           ; 8 bits, 1 stop, REN enabled
SETB TR1                 ; START TIMER 1
AGAIN:
MOV SBUF, #'G'           ; transfer 'Y' serially
HERE: JNB TI, HERE       ; WAIT FOR TRANSMISSION TO BE OVER
CLR TI                   ; clear TI for next transmission
SJMP AGAIN               ; repeat
END
```

### C- Program:

```
#include<reg51.h>
Void main ()
{
    TMOD=0x20;
    TH1=0xFD;
    SCON=0x50;
    TR1=1;
    while (1)
    {
        SBUF = 'G';
        while (TI==0);
        TI=0;
    }
}
```



2) Write an 8051 assembly language program to transfer the message "HELLO" serially at 9600 baud rate, 8-bit data, 1 stop bit continuously.

```
ORG 00H
MOV TMOD, #20H           ; timer 1, mode 2
MOV TH1, #-3             ; 9600 baud rate
MOV SCON, #50H           ; 8 bits, 1stop, REN enabled
SETB TR1                 ; START TIMER 1
```

**START:**

```
MOV A, #'H'
ACALL TRANS
MOV A, #'E'
ACALL TRANS
MOV A, #'L'
ACALL TRANS
MOV A, #'L'
ACALL TRANS
MOV A, #'O'
ACALL TRANS
SJMP START                ; repeat
```

---

**TRANS:**

```
MOV SBUF, A
```

**HERE:**

```
JNB TI, HERE              ; WAIT FOR TRANSMISSION TO BE
                           ; OVER
CLR TI                     ; clear TI for next transmission
RET
END
```

**HERE:**

JNB TI, HERE

; WAIT FOR TRANSMISSION TO BE over

CLR TI

; clear TI for next transmission

RET

END

6. Write an 8051 assembly language program to transfer letter "H" serially at 9600 baud rate, continuously.

ORG 00H

MOV TMOD, #20H ; timer 1, mode 2

MOV TH1, #-3 ; 9600 baud rate or FDh

MOV SCON, #50H ; 8 bits, 1 stop, REN enabled

SETB TR1 ; START TIMER 1

**AGAIN:**

MOV SBUF, #'H' ; transfer 'Y' serially

WAIT: JNB TI, WAIT ; WAIT FOR TRANSMISSION TO BE OVER

CLR TI ; clear TI for next transmission

SJMP AGAIN ; repeat

END

### C- Program:

```
#include<reg51.h>
```

```
Void main ()
```

```
{
```

```
    TMOD=0x20;
```

```
    TH1=0xFD;
```

```
    SCON=0x50;
```

```
    TR1=1;
```

```
    while (1)
```

```
    {
```

```
        SBUF = 'H';
```

```
        while (TI==0);
```

```
        TI=0;
```

```
    }
```

```
}
```

7. Read port P1 data and transfer this data serially continuously at a baud rate of 4800.

```
ORG 00H
MOV TMOD, #20H
MOV TH1, #-6           ; 4800 baud rate or FA
MOV SCON, #50H
MOV P1, #0FFH
SETB TR1
```

**START:**

```
MOV A, P1
MOV SBUF, A
```

**WAIT:**

```
JNB TI, WAIT
CLR TI
SJMP START
END
```

**C- Program:**

```
#include<reg51.h>
void main ()
{
    unsigned char x;
    P1=0xFF;
    TMOD=0x20;
    TH1=0xFA;           ; 4800 baud rate or -6
    SCON=0x50;
    TR1=1;
    while (1)
    {
        x=P1;           // read port P1
        SBUF = x;       // place in SBUF
        while (TI==0);
        TI=0;
    }
}
```



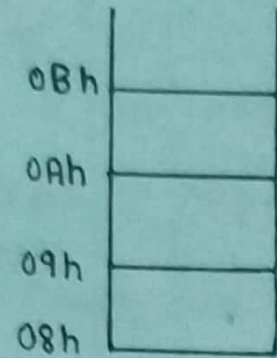
## Stack:-

- \* Stack refers to an area of internal RAM used by the CPU to store & retrieve data (take back) quickly.
- \* The register used to access the Stack is called the Stack pointer (Sp) register.
- \* The Stack pointer is a 8-bit register used by the 8051 to hold an internal RAM address that is called the Top of the Stack.
- \* When 8051 is RESET, the Sp is Set to 07h
- \* When data is to be placed on the Stack, the Sp increments - before storing data on the Stack i.e.  $Sp = Sp + 1$ , so that the Stack grows up as data is stored.
- \* As the data is retrieved from the Stack, the byte is read from Stack & then Sp decrements i.e.  $Sp = Sp - 1$  to point to the next available byte of stored data.
- \* Storing the data onto the Stack is called a PUSH.
- \* Retrieving the contents of the Stack is called a POP.
- \* RAM location 08h is the 1<sup>st</sup> location used by the Stack to store the data.

Eg:- 1) `move R2, #30h`  
`PUSH 2`

Assume that initially Bank0 is Selected &  $Sp = 07h$

Before Execution

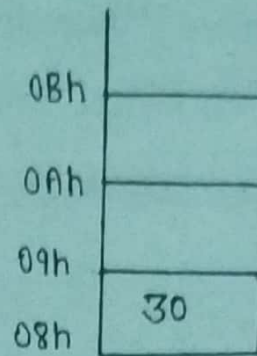


Sp = 07h

Sp = Sp + 1

Sp = 08h

After Execution



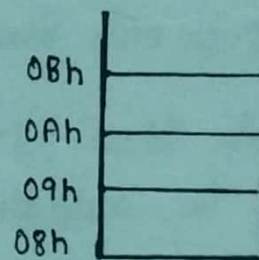
Sp = 08h

\* Sp is 1<sup>st</sup> incremented by one i.e.  $Sp = Sp + 1$ , then the contents of R<sub>2</sub> is stored in top of stack i.e. 08h address.

2) move R<sub>2</sub>, #30h  
 move R<sub>3</sub>, #40h  
 move R<sub>4</sub>, #41h  
 PUSH 2  
 PUSH 3  
 PUSH 4

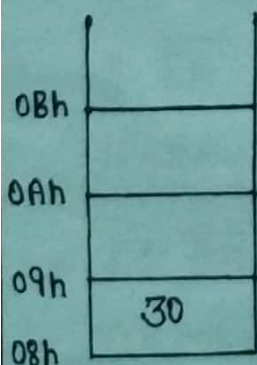
\* Assume Bank 0 is Selected & Sp has initially 07h

Before Execution



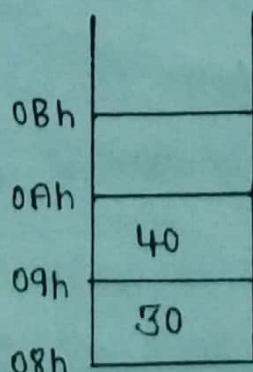
After Execution

After PUSH 2



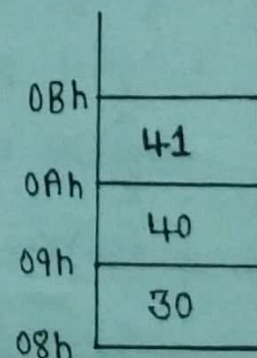
Sp = 08h

After PUSH 3



Sp = 09h

After PUSH 4



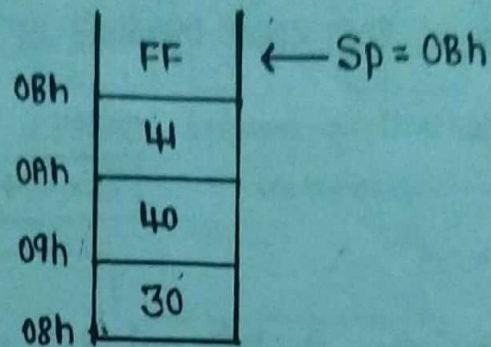
Sp = 0Ah



## Popping from Stack:-

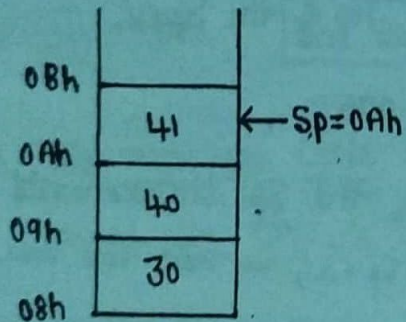
Eg:-  
pop 4 ; pop Stack into R<sub>4</sub> of Bank0  
pop 3 ; pop Stack into R<sub>3</sub> of Bank0  
pop 2 ; pop Stack into R<sub>2</sub> of Bank0

### Before Execution



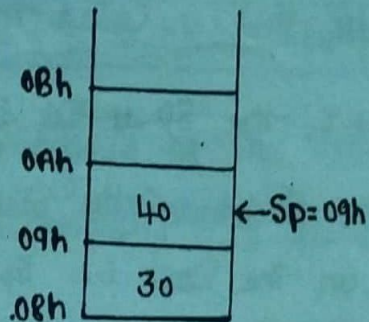
### After Execution:-

After pop 4 → Sp = Sp - 1



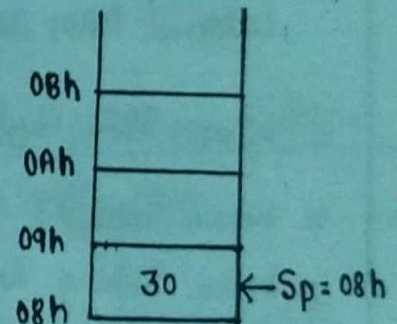
(R<sub>4</sub>) = FFh

After pop 3 → Sp = Sp - 1



(R<sub>3</sub>) = 41h

After pop 2 → Sp = Sp - 1



(R<sub>2</sub>) = 40h

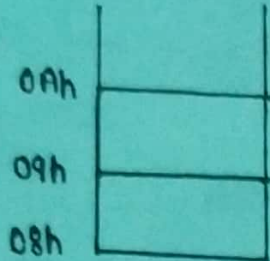


Eg:-  $\rightarrow$  move  $R_2, \#30h$

PUSH 2

Assume that initially Bank0 is Selected &  $Sp = 07h$

Before Execution

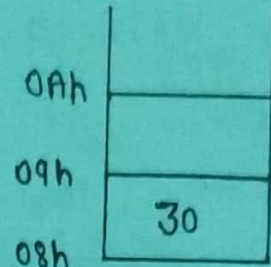


$Sp = 07h$

$Sp = Sp + 1$

$Sp = 08h$

After Execution

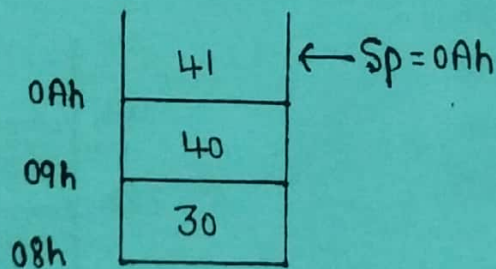


$Sp = 08h$

\*  $Sp$  is 1<sup>st</sup> incremented by one i.e.  $Sp = Sp + 1$ , then the Contents of  $R_2$  is Stored in top of Stack i.e. 08h address.

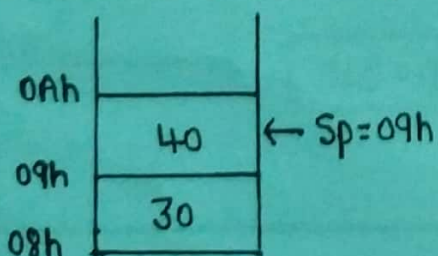
$\rightarrow$  pop 3 ; Pop Stack into  $R_3$  of Bank0  
pop 2 ; Pop Stack into  $R_2$  of Bank0

Before Execution



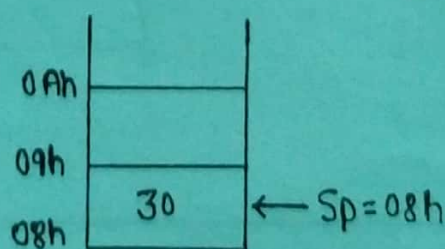
After Execution :

After pop 3  $\rightarrow Sp = Sp - 1$



$(R_3) = 41h$

After pop 2  $\rightarrow Sp = Sp - 1$



$(R_2) = 40h$

## Instructions:-

### 1) PUSH direct address:-

Function : push onto Stack

Description : The Stack pointer is incremented by one. The contents of the indicated variable is then copied into the internal RAM location addressed by the Stack pointer.

Flags  
affected : None

Bytes : 2

Cycles : 2

operation :  $(SP) \leftarrow (SP) + 1$



NOTE:- This instruction Supports only direct addressing mode.

∴ The instructions Such as

"PUSH A" & "PUSH R<sub>n</sub>" are illegal instructions.

Eg:-

1) PUSH 0E0h

Where E0h is the RAM address belonging to register A.

2) PUSH 03h

Where 03h is the RAM address of R<sub>3</sub> of Bank 0.

2) pop direct address:-

Function : pop from the stack.

Description : This Copies the byte pointed to by Stack pointer (Sp) to the location where direct address is indicated & decrements Sp by 1.

Flags affected : None

Bytes : 2

Cycles : 2

operation :  $(Sp) \leftarrow (Sp) - 1$

NOTE:- This instruction Supports only direct addressing mode.

∴ The instructions Such as

"POP A" & "POP R<sub>n</sub>" are illegal instructions.

Eg:-

1) POP 0E0h

Where E0h is the RAM address belonging to register A.



### Interrupt:-

Interrupt is a hardware or software signal which interrupt the microcontroller, to ~~service~~ request for its service.

### Interrupt Service Routine: (ISR)

The program associated with the interrupt is called Interrupt Service Routine (ISR) or Interrupt handler.

### Difference between polling & Interrupt:-

#### Polling

- 1) In polling, microcontroller continuously monitor status of given device, to perform the service.
- 2) The polling method cannot assign priority, since it check all device in round robin fashion.
- 3) In polling masking of interrupt is not possible.
- 4) Waste of Time in polling

#### Interrupt

- 1) In interrupt, whenever the device need its service, the device notifies the microcontroller by sending a signal.
- 2) In interrupt, priority is assign.
- 3) In interrupt, masking of interrupt is possible.
- 4) no waste of Time.

### Note:-

- \* For every interrupt, there must be an Interrupt Service Routine (ISR), or Interrupt handler.
- \* When interrupt is invoked, the microcontroller runs the Interrupt Service Routine.
- \* For every interrupt, there is a fixed memory location that holds the address of ISR, called Interrupt vector table.

## Steps in executing an interrupt

Upon activation of an interrupt, the microcontroller goes through the following steps.

1. It finishes the instruction it is executing and saves the address of the next instruction (PC) on the stack.
2. It also saves the current status of all the interrupts internally (i.e., not on the stack).
3. It jumps to a fixed location in memory called the interrupt vector table that holds the address of the interrupt service routine.
4. The microcontroller gets the address of the ISR from the interrupt vector table and jumps to it. It starts to execute the interrupt service subroutine until it reaches the last instruction of the subroutine, which is RETI (return from interrupt).
5. Upon executing the RETI instruction, the microcontroller returns to the place where it was interrupted. First, it gets the program counter (PC) address from the stack by popping the top two bytes of the stack into the PC. Then it starts to execute from that address.

### \* Interrupt vector table of 8051:-

There are six interrupts in 8051.

\* First the Reset address location starts at 0000h.

\* Two interrupts Timer 0 & Timer 1 address location at ~~000~~000B and 001B.

### Interrupt of 8051

	<u>RAM location Address</u>
Reset	→ 0000h
External hardware interrupt 0 (INT0)	→ <del>0000h</del> 0003h
Timer 0 interrupt (TF0)	→ 000Bh
External hardware interrupt 1 (INT1)	→ <del>0003h</del> 0013h
Timer 1 interrupt (TF1)	→ 001Bh
Serial <del>COM</del> COM interrupt (RI & TI)	→ 0023h

Table 1 :- Interrupt vector table



\* Two interrupt for hardware extend interrupt address location at 0003h. (INT0) & 0013 (INT1)

\* one serial communication interrupt address location at 0023h.

\* Enabling & Disabling an Interrupt:-

\* Steps in Enabling Interrupt:-

### Steps in enabling an interrupt

To enable an interrupt, we take the following steps:

1. Bit D7 of the IE register (EA) must be set to high to allow the rest of register to take effect.
2. If EA = 1, interrupts are enabled and will be responded to if their corresponding bits in IE are high. If EA = 0, no interrupt will be responded to, even if the associated bit in the IE register is high.

To understand this important point look at Example 11-1.

D7				D0			
EA	--	ET2	ES	ET1	EX1	ET0	EX0

<b>EA</b>	IE.7	Disables all interrupts. If EA = 0, no interrupt is acknowledged. If EA = 1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit.
--	IE.6	Not implemented, reserved for future use.*
<b>ET2</b>	IE.5	Enables or disables Timer 2 overflow or capture interrupt (8052 only).
<b>ES</b>	IE.4	Enables or disables the serial port interrupt.
<b>ET1</b>	IE.3	Enables or disables Timer 1 overflow interrupt.
<b>EX1</b>	IE.2	Enables or disables external interrupt 1.
<b>ET0</b>	IE.1	Enables or disables Timer 0 overflow interrupt.
<b>EX0</b>	IE.0	Enables or disables external interrupt 0.

\*User software should not write 1s to reserved bits. These bits may be used in future flash microcontrollers to invoke new features.

Figure 11-2. IE (Interrupt Enable) Register



## \* Level-triggered interrupt

- In level triggered mode, INTO & INT1 pins are normally high.
- If low level signal is applied, it triggers the interrupt.
- Then microcontroller stops whatever it's doing and jump to ~~ISR~~ interrupt vector table to serve the interrupt.
- This is called level triggered or level activated interrupt.
- The low level signal at INT pin must be removed before the execution of last instruction of ISR, otherwise other interrupt will be generated.

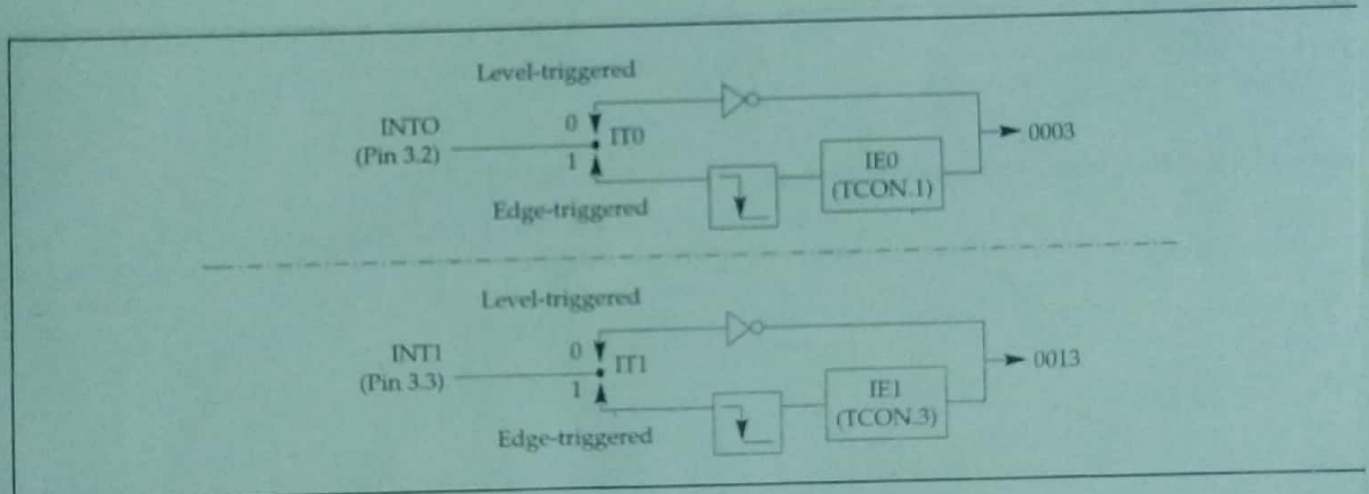


Figure 11-4. Activation of INT0 and INT1

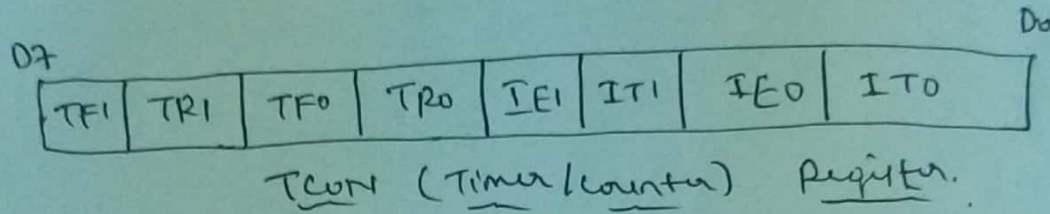
## \* Edge triggered interrupt:-

- To make edge triggered in 8051, we must program the bits of TCON Register.
- IT0 & IT1 in TCON determine level triggered if these bits are zero.
- To make them edge triggered we must enable IT0 & IT1 to one.

P.T.O

→ TCON.0 & TCON.2 represent ITO & ITI.

→ By the instruction SETB TCON.0 & SETB TCON.2 we can the external hardware interrupt INTO & INTI become edge triggered.



\* Interrupt priority in 8051:-

After 8051 powered up, the following interrupt priority is assigned as per the table.

\* External interrupt INTO is having highest priority followed by, Timer interrupt (TF0), (INTI), (TF1) & Serial communication interrupt.

**Table 11-3: 8051/52 Interrupt Priority Upon Reset**

**Highest to Lowest Priority**

External Interrupt 0	(INT0)	<b>Highest</b>
Timer Interrupt 0	(TF0)	
External Interrupt 1	(INT1)	
Timer Interrupt 1	(TF1)	
Serial Communication	(RI + TI)	<b>Lowest</b>
<del>Timer Interrupt 2</del>	<del>(TF2)</del>	

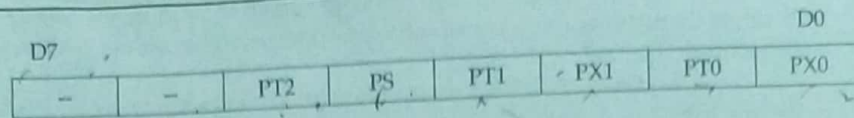
\* Setting interrupt priority with Ip register:-

→ we can alter priority table by using Ip register.

→ we can assign a higher priority any register by altering & setting Ip register.

P.T.O





Priority bit = 1 assigns high priority. Priority bit = 0 assigns low priority.

-	IP.7	Reserved	
-	IP.6	Reserved	
PT2	IP.5	Timer 2 interrupt priority bit (8052 only)	X
PS	IP.4	Serial port interrupt priority bit	→ Serial Communication
PT1	IP.3	Timer 1 interrupt priority bit	→ TF1
PX1	IP.2	External interrupt 1 priority bit	→ INT1
PT0	IP.1	Timer 0 interrupt priority bit	→ TF0
PX0	IP.0	External interrupt 0 priority bit	→ INT0

User software should never write 1s to unimplemented bits, since they may be used in future products.

Figure 11-8. Interrupt Priority Register (Bit-addressable)

Problem: - on interrupt priority setting

#### Example 11-1

Show the instructions to (a) enable the serial interrupt, Timer 0 interrupt, and external hardware interrupt 1 (EX1), and (b) disable (mask) the Timer 0 interrupt, then (c) show how to disable all the interrupts with a single instruction.

#### Solution:

(a) `MOV IE, #10010110B ; enable serial, Timer 0, EX1`

Since IE is a bit-addressable register, we can use the following instructions to access individual bits of the register.

#### Example 11-13

Assume that after reset, the interrupt priority is set by the instruction "MOV IP, #00001100B". Discuss the sequence in which the interrupts are serviced.

#### Solution:

The instruction "MOV IP, #00001100B" (B is for binary) sets the external interrupt 1 (INT1) and Timer 1 (TF1) to a higher priority level compared with the rest of the interrupts. However, since they are polled according to Table 11-3, they will have the following priority.

Highest Priority	External Interrupt 1	(INT1)
	Timer Interrupt 1	(TF1)
	External Interrupt 0	(INT0)
	Timer Interrupt 0	(TF0)
Lowest Priority	Serial Communication	(RI + TI)