

8086 ARCHITECTURE* Introduction to Microprocessor:

- The word microprocessor means a CPU only.
- The functional blocks like memory, & other peripherals are to be connected externally to a microprocessor chip to form a complete microprocessor.
- The central processing unit (CPU) of a digital computer, built into a single IC is called microprocessor.
- The word length of a μp may be 8-bit, 16-bit, 32-bit or 64-bit.

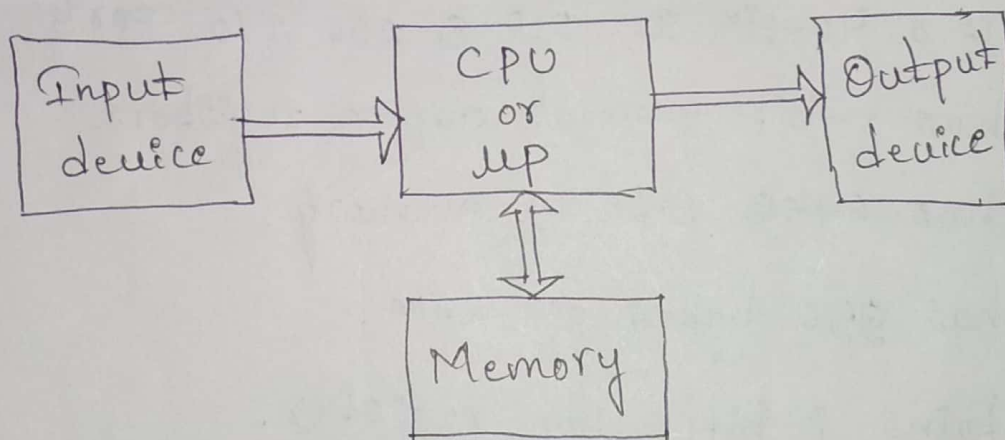


Fig: Block diagram of digital computer.

- An 8-bit μp processes 8-bit data at a time. If data consists of more than 8-bits, the μp will first take upto 8-bits of data for processing, then next group of 8-bits and so on.
- Similarly, a 16-bit μp handles 16-bits of data at a time. Similarly, 32-bit & 64-bit μp 's handle

32-bit & 64-bit operands (data) at a time.

→ A computer whose CPU contains more than one μp is called a multiprocessor system.

→ The first μp , Intel 4004, a 4-bit μp .

→ Intel introduced the first 8-bit μp , Intel 8008. Intel also introduced an improved 8-bit μp , Intel 8085.

* Features of 8085:

→ It is an 8-bit μp .

→ It has the 8-bit data bus.

→ 16-bit address bus.

→ It requires +5V d.c supply & 3MHz clock. freq.

→ It is a 40-pin IC DIP & 256 I/O ports

→ It has 8-bit general purpose registers.

→ It has 64KB size of memory.

→ It has 256 bytes of RAM.

→ Contains 8-bit flag register.

* Features of 8086 :

1. The 8086 is a 16-bit HMOS μ P. (N-channel silicon gate)
2. It is available in a 40-pin IC.
3. Operates at +5V d.c supply. (3)
4. Its clock frequency is 5 MHz for 8086.
 8 MHz " 8086-2.
 10 MHz " 8086-1.
5. It has a 16-bit data bus. & has 20-bit address bus.
6. Memory size $2^{20} = 1\text{ MB}$ (1 megabyte) (RAM)
7. Instruction byte queue is 6 bytes long.
8. It has 16-bit flag register.
9. It has 64K I/O Ports.

* 8086 Architecture: (or) functional diagram:

→ The complete architecture of 8086 can be divided into two parts (a) Bus Interface unit (BIU) & (b) Execution unit (EU).

* Bus Interface unit :

→ The BIU is the 8086's interface to the outside world. It provides all 'external bus operations'.

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

Address/data
 $AD_0 - AD_{15}$

Memory address bus &
data bus interface.

(u)

Internal data bus.

BUS
INTERFACE
UNIT

Address Conversion
mechanism (address)

CS

DS

SS

ES

IP

Internal data bus

Instruction byte
queue 6 bytes.

Decoding
Circuit.

EXECUTION
UNIT

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

ALU (16)

Flags (16)

Register bank.

Timing & control
circuit.

clock & control
signals.

Functions of Bus Interface unit:

1. It sends address of the memory or I/O.
2. It fetches instruction from memory.
3. It reads data from port/memory.
4. It writes data into port/memory.
5. It supports instruction queuing.
6. It provides address relocation facility.

→ Instruction Queue:

→ While the fetched instruction is executed internally, the external bus is used to fetch the machine code of the next instruction & arrange it in a queue known as predecoded instruction byte queue.

→ It is a 6 bytes long, first-in first-out structure.

→ The instructions from the queue are taken for decoding sequentially.

* Execution unit:

→ While the opcode is fetched by the BIU, the execution unit (EU) executes the previously decoded instruction.

→ The BIU along with the EU thus forms a pipeline.

- The EU contains the register set of 8086.
 - It has a 16-bit ALU, able to perform arithmetic & logic operations.
 - ⑥ → The 16-bit reg. flag reflects the results of execution by the ALU.
 - The decoding unit decodes the opcode bytes from the instruction byte queue.
 - Timing & control unit derives the necessary control signals to execute the instruction.
-

* Register Organization of 8086:

- 8086 has a powerful set of registers known as general purpose & special purpose registers.
- All of them are 16-bit registers.
- The general purpose registers, can be used as either 8-bit or 16-bit registers.
- They may be either used for holding data, variables, & ~~imm~~ intermediate results temporarily.
- The special purpose registers are used as segment registers, pointers, index registers or as offset storage registers for particular addressing modes.

1. General Data registers:

- The registers AX, BX, CX & DX are the general purpose 16-bit registers.

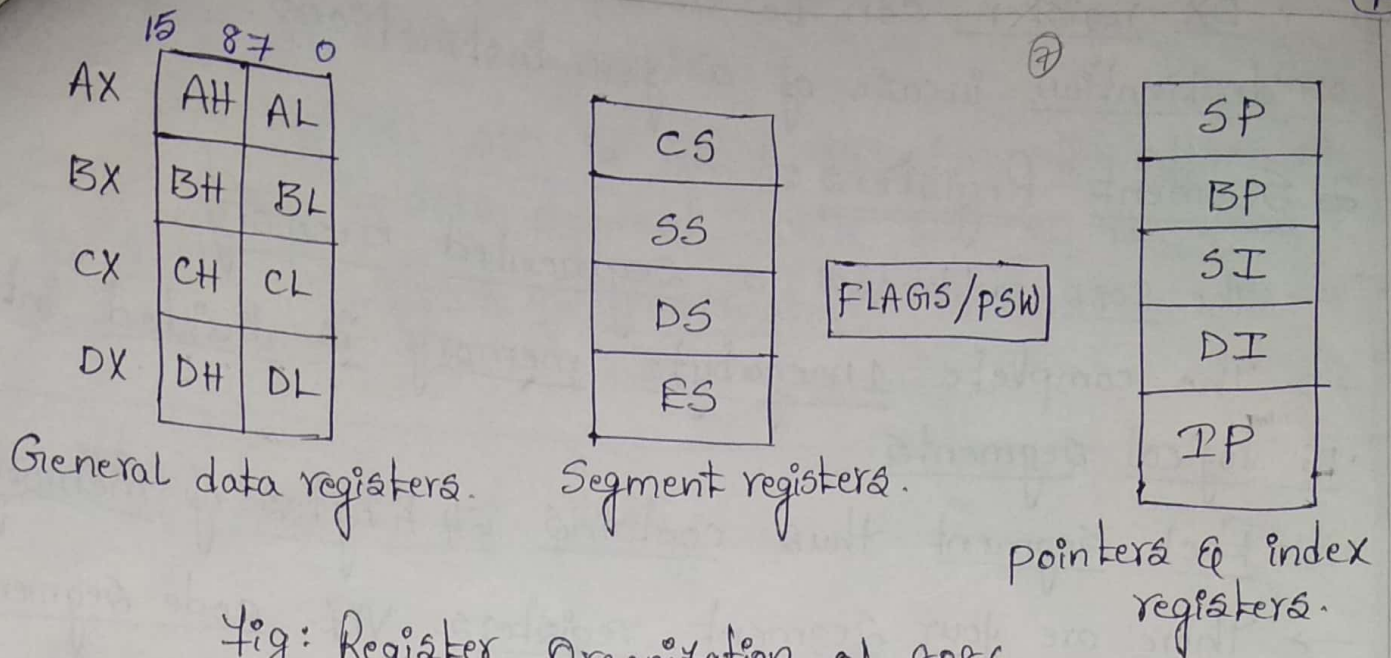


Fig: Register Organization of 8086.

- AX is used as 16-bit accumulator, with the lower 8-bits of AX designated as AL & higher 8-bits as AH.
- AL can be used as an 8-bit accumulator for 8-bit operations.
- The letters L & H specify the lower & higher bytes of a particular register.
- CH means the higher 8-bits of the CX register. & CL means the lower 8-bits of the CX register.
- The letter 'X' is used to specify the complete 16-bit register.
- The register CX is used as a default counter in case of string & loop instructions.
- The register BX is used as an offset storage for forming physical addresses in case of certain addressing modes.

→ DX register can be used as an implicit operator or destination in case of a few instructions.

2. Segment Registers:

- The 8086 addresses a segmented memory.
- ② → The complete 1 megabyte memory is divided into 16 logical segments.
- Each segment thus contains 64 Kbytes of memory.
- There are four segment registers. viz. code segment register (CS), Data segment register (DS), Extra segment register (ES), & Stack segment register (SS).
- The CS register is used for addressing a memory location in the code segment of the memory, where the executable program is stored.
- The DS register points to the data segment of the memory, where the data is resided.
- The ES also refers to another data segment of the memory. Thus, it also contains data.
- The SS register is used for addressing stack segment of memory i.e. memory which is used to store stack data. The cpu uses the stack for temporarily storing important data.

3. Pointers and Index registers:

- The pointers contain offset within the particular segments. The index registers are useful for string manipulations.

→ The pointers IP, BP & SP contain offsets within the code & stack segments. (5)

→ The register SI is used to store the offset of source data in data segment while the DI is used to store the offset of destination in the data or extra segment. (1)

* Flag Register of 8086:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	X	X	X	0	D	I	T	S	≠	X	AC	X	P	X	Cy

0 - Overflow flag

D - Direction "

I - Interrupt "

T - Trap "

S - Sign "

≠ - Zero "

AC - Auxiliary Carry flag.

P - Parity "

Cy - Carry "

X - Not used.

→ The 8086 flag register contents indicate the results of computations in the ALU. It has 16-bit flag register which is divided into two parts viz. (a) condition code or status flags & (b) machine control flags.

→ The condition code flag register is the lower byte of the 16-bit flag register along with the overflow flag.

→ The control flag register is the higher byte of the flag register of 8086. It contains three flags, viz. direction flag (D), interrupt flag (I) & trap flag (T).

S - Sign flag: It is set when the result of any comp is negative.

Z - Zero flag: It is set if the result of computation or comparison performed by the previous instruction is zero.

P - Parity flag: This is set to 1 if the lower byte of the result contains even number of 1's.

(10) C - Carry flag: This is set when there is a carry out of MSB in case of addition or a borrow in case of subtraction.

T - Trap flag: If this flag is set, the processor enters the single step execution mode. A trap interrupt is generated after execution of each instruction.

I - Interrupt flag: If it is set, the maskable interrupts are recognised by the CPU, otherwise they are ignored.

D - Direction flag: If this flag bit is '0', the string is processed & beginning from lowest address to highest address, i.e. autoincrementing mode. Otherwise, the string is processed from the highest address towards the lowest address, i.e. autodecrementing mode.

AC - Auxiliary carry flag: This is set if there is a carry from the lowest nibble, i.e. bit three, during addition or borrow for the lowest nibble, i.e. bit three, during subtraction.

O - Overflow flag: The result is more than 7-bits in size in case of 8-bit signed operations & more than 15-bits in size in case of 16-bit signed operations, then the Overflow flag will be set.

Memory Segmentation:

(11)

(6)

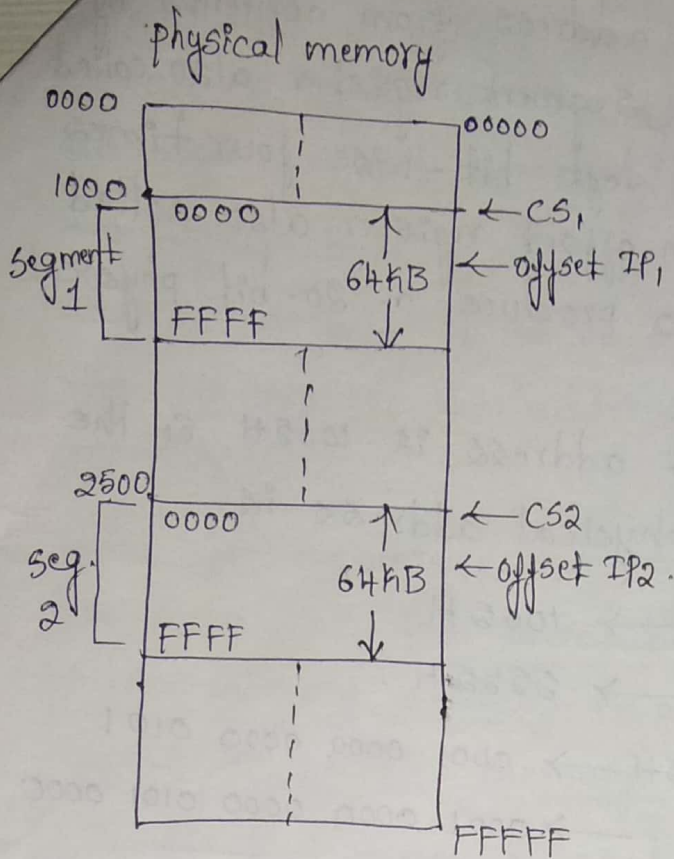


Fig: Non-Overlapping Segments.

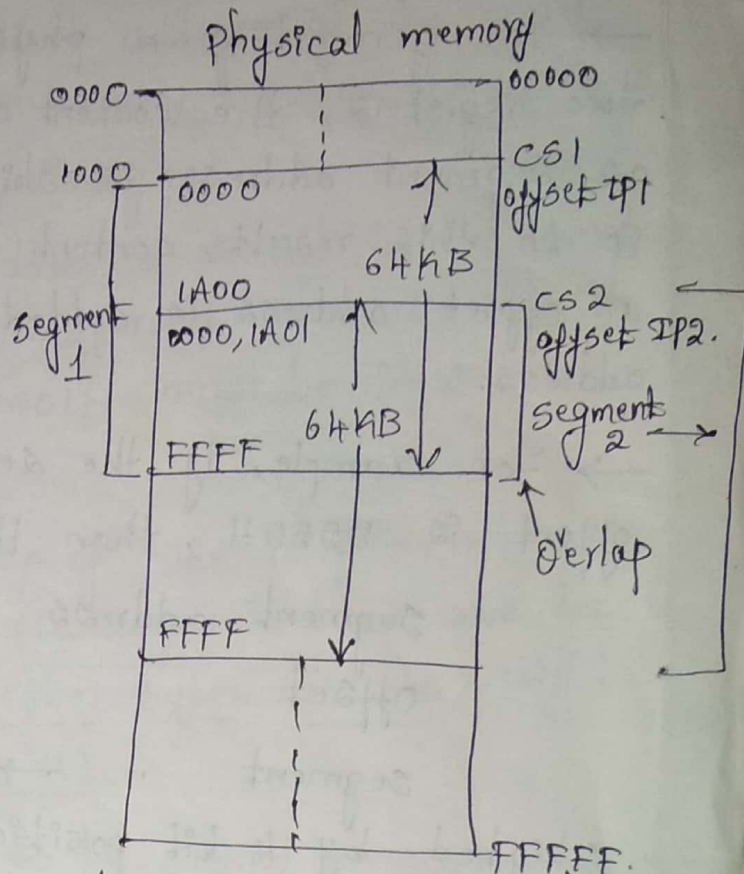


Fig: Overlapping segments.

- The 8086 is able to address 1MB of physical memory. The complete 1MB is divided into 16 segments, each of 64Kbytes size.
- The addresses of the segments may be assigned as 0000H to F000H resp.
- To address a specific memory location within a segment, we need an offset address.
- This is also 16-bit long so that the address values are from 0000H to FFFFH. So, physical address range from 00000H to FFFFFFH.
- Suppose a segment starts before at a particular segment & its maximum size can be 64KB. But, if another segment starts before this 64KB location of first segment, the two segments are said to be Overlapping segments.

* Memory addresses:

→ For generating a physical address from contents of two registers, the content of a segment register also called as segment address is shifted left bit-wise four times & to this result, content of an offset register also called as offset address is added, to produce a 20-bit physical address.

→ For example, if the segment address is 1005H & the offset is 5555H, then the physical address is,

Segment address → 1005H.

Offset " → 5555H.

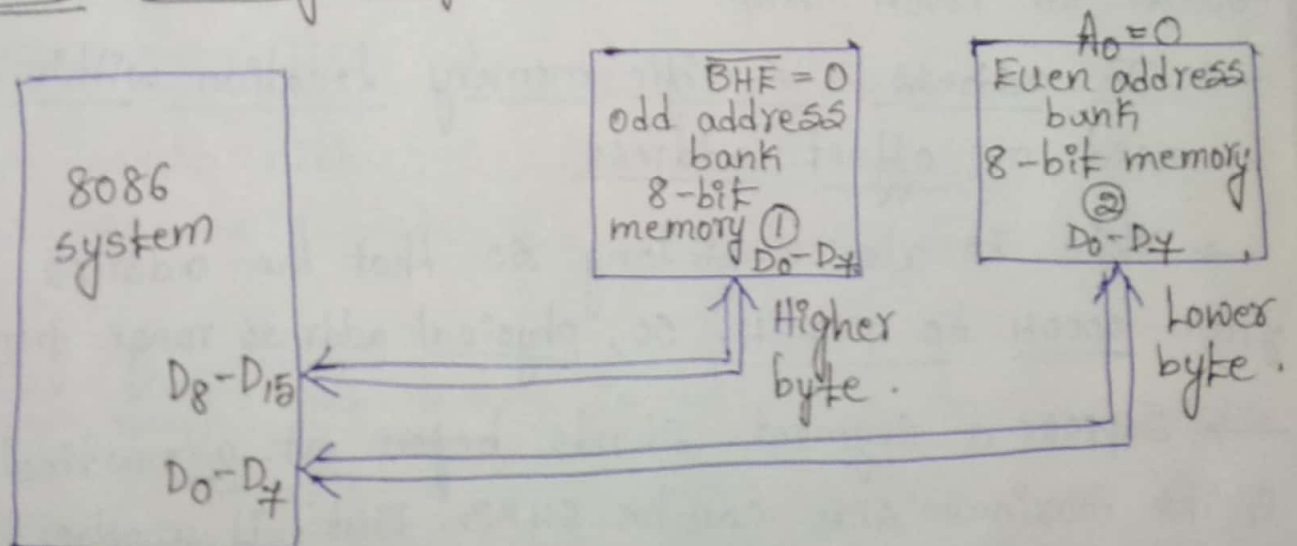
Segment " → 1005H → 0001 0000 0000 0101

Shifted by 4 bit positions → 0001 0000 0000 0101 0000

+
offset address → 0101 0101 0101 0101

Physical address → 0001 0101 0101 1010 0101
1 5 5 A 5

* Physical memory Organization:



The 8086 is a 16-bit data, the lower byte of a 16-bit data is stored at the first address of the map 00000H. It is to be transferred over D_0-D_7 of the up bus so 00000H must be in 8-bit memory (2). (7) (13)

→ Higher byte of the 16-bit data is stored in the next address 00001H, it is to be transferred over D_8-D_{15} of the up bus so the address 00001H must be in 8-bit memory (1).

→ The complete memory map of 8086 is filled with 16-bit data, all the lower bytes (D_0-D_7) will be stored in 8-bit memory bank (2) & all higher bytes (D_8-D_{15}) will be stored in 8-bit memory bank (1).

→ All the lower bytes have to be stored at even addresses & all higher bytes to be stored at odd addresses.

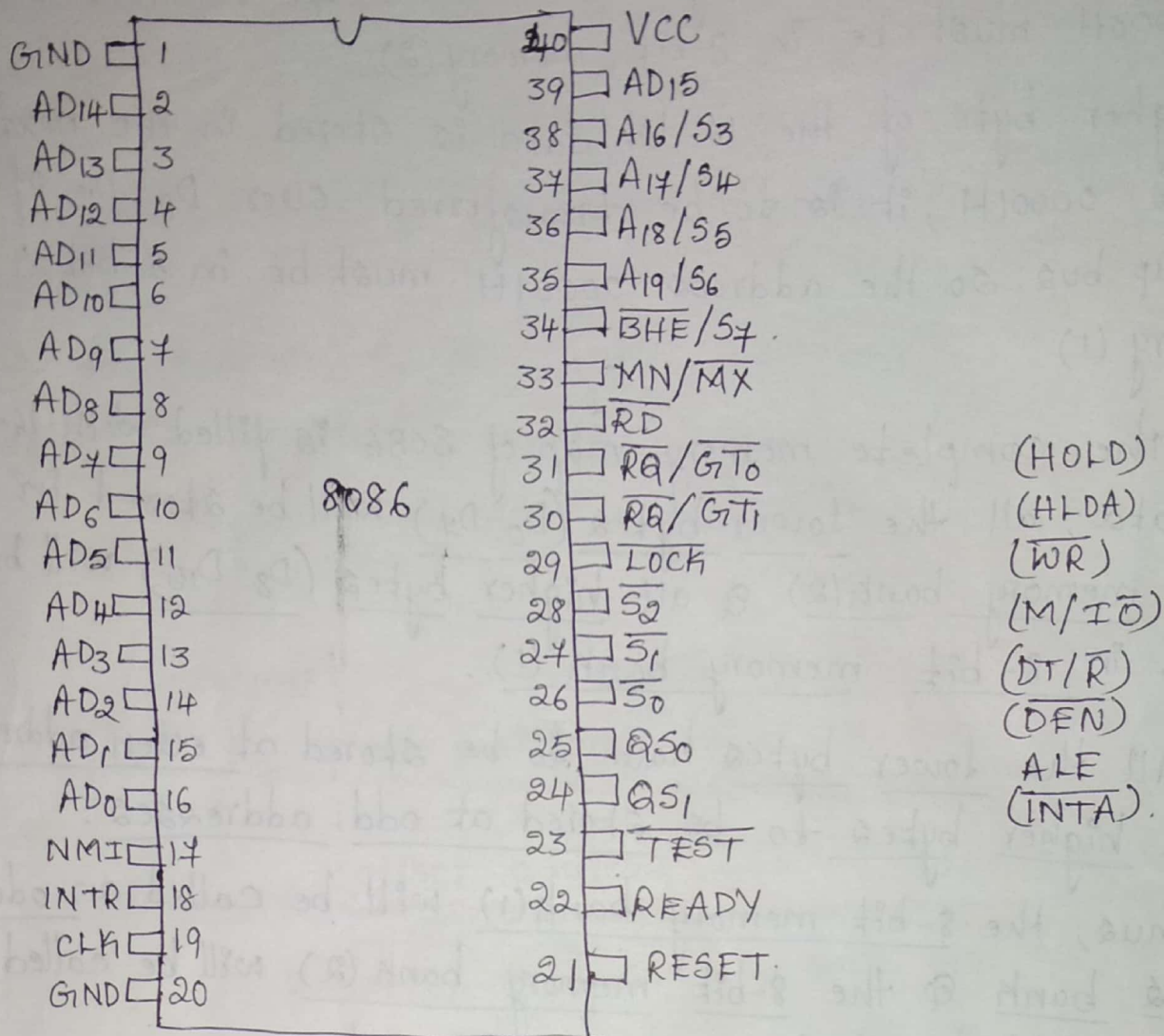
→ Thus, the 8-bit memory bank (1) will be called an odd address bank & the 8-bit memory bank (2) will be called an even address bank.

→ If the processor fetches a word, (two bytes) from memory, there are different possibilities:

1. Both the bytes may be data operands.
2. " " " " contain Opcode bits.
3. One of the bytes may be Opcode while other may be data.

* Signal descriptions of 8086 :

Maximum mode. Minimum mode.



→ The 8086 is a 40-pin IC. Some of the pins serve a particular function in minimum mode (single processor mode) & others function in maximum mode (multiprocessor mode).

1. Common function Signals :

AD₁₅ - AD₀ : These are the time multiplexed memory I/O address & data lines. Address remains on the lines during T₁ state, while the data bus during T₂, T₃, T₄ & T₅. T₁, T₂, T₃ & T₄ are clock states & T₅ is a wait state.

A₁₉/S₆, A₁₈/S₅, A₁₇/S₄, A₁₆/S₃ : These are the time multiplexed address & status lines. During T₁, these are address lines.

Status information is available on those lines for T_2 , T_3 , T_0 & T_4 . The status line indicates the operation is memory or I/O.

→ The S_4 & S_3 together indicate which segment register is being used for memory accesses.

S_4	S_3	Indications.
0	0	Alternate data.
0	1	stack
1	0	code
1	1	Data.

$\overline{BHE}/S_4$ - Bus high Enable / status :

→ The $\overline{BHE}$ signal remains low during T_1 , & it is used to enable data of high-order data lines, D_8-D_{15} .

→ The $\overline{BHE}/S_4$ is a time multiplexed line. During T_2 to T_4 the status signal S_4 is transmitted on this line. It remains always high.

→ $\overline{BHE}$ in conjunction with A_0 determines whether a byte or word will be transferred from/to memory locations.

$\overline{BHE}$	A_0	word/Byte access.
0	0	whole word.
0	1	Upper byte from/to odd add
1	0	Lower " " " even "
1	1	None

$\overline{RD}$ - Read (output) : It is a read signal issued by the processor. It is an active low signal. It

indicates that the processor is performing operation.

Ready (Input): It is an active high signal. This is the acknowledgement from the slow devices or memory that they have completed the data transfer.

INTR - Interrupt request: It is active high. If any interrupt request is pending, the processor enters the interrupt acknowledge cycle.

TEST: This input is examined by 'WAIT' instruction. If the TEST i/p goes low, execution will continue, else, the processor remains in an idle state.

NMI - Non-Maskable Interrupt: The NMI is not maskable internally by software.

RESET: It is a system reset & active high signal. When it goes high, the processor goes into reset state. This signal remain high for at least four clock cycles.

CLK - Clock Input: This provides the basic timing for processor operation & bus control activity. It is an asymmetric square wave with 33% duty cycle.

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

GND: Ground for the internal circuit.

MN/ $\overline{\text{MX}}$: The logic level at this pin decides whether the processor is to operate in either minimum or maximum mode.

Minimum Mode Signals:

M/ $\overline{IO}$: Memory/ $\overline{IO}$: When it is low, it indicates the CPU is having an I/O operation, & when it is high, it indicates that the CPU is having a memory operation.

INTA - Interrupt Acknowledge : It is active low. On receiving interrupt signal the processor issues an interrupt acknowledge signal through this line.

ALE - Address Latch Enable : It is a high pulse issued by the processor during T_1 state of a bus cycle. The processor sends this signal to latch the address into the 8282/8283 address latch.

DT/ $\overline{R}$ - Data transmit/Receive : This op is used to decide the direction of data flow through the trans-receivers. When this signal is high, data are sent out, & when it is low, data are received.

$\overline{DEN}$ - Data Enable : It is used to enable the trans-receivers. It is an active low signal. It is used to separate the data from the multiplexed address/data signal.

HOLD : It is an active high signal. When another device in the system wants to use the address & data bus, it sends a HOLD request to the processor through this line.

H $\overline{LDA}$ - Hold Acknowledge : It is active high signal.

On receiving HOLD signal, the processor is
HOLD acknowledge signal through this pin.

3. Maximum mode Signals:

$\overline{S}_2, \overline{S}_1, \overline{S}_0$ - Status lines: these lines indicate the type of operation, being carried out by the processor. These lines are encoded as,

$\overline{S}_2$	$\overline{S}_1$	$\overline{S}_0$	Indication.
0	0	0	Interrupt acknowledge
0	0	1	Read I/O port.
0	1	0	Write I/O port.
0	1	1	Halt.
1	0	0	Code access.
1	0	1	Read memory.
1	1	0	Write memory.
1	1	1	Passive.

LOCK: It is an active low signal. It remains active until the completion of next instruction. When it goes low, all interrupts are masked. & HOLD request is not granted.

QS_1, QS_0 - Queue status: These signals provide status to allow external tracking of the 8086 internal instruction queue. These are encoded as

QS_1	QS_0	Indication.
0	0	No operation.
0	1	1st byte of opcode from queue.
1	0	Empty queue.
1	1	Subsequent byte from queue

$\overline{GTO}$, $\overline{RQ}/\overline{GTT}$ - Request / Grant: An external device sends a hold request to the CPU through these lines to release the local bus at the end of CPU's current bus cycle. (10)

→ After receiving hold request, the CPU sends hold acknowledge signal through these lines.

* Read write cycles Timing diagrams:

→ Read cycle:

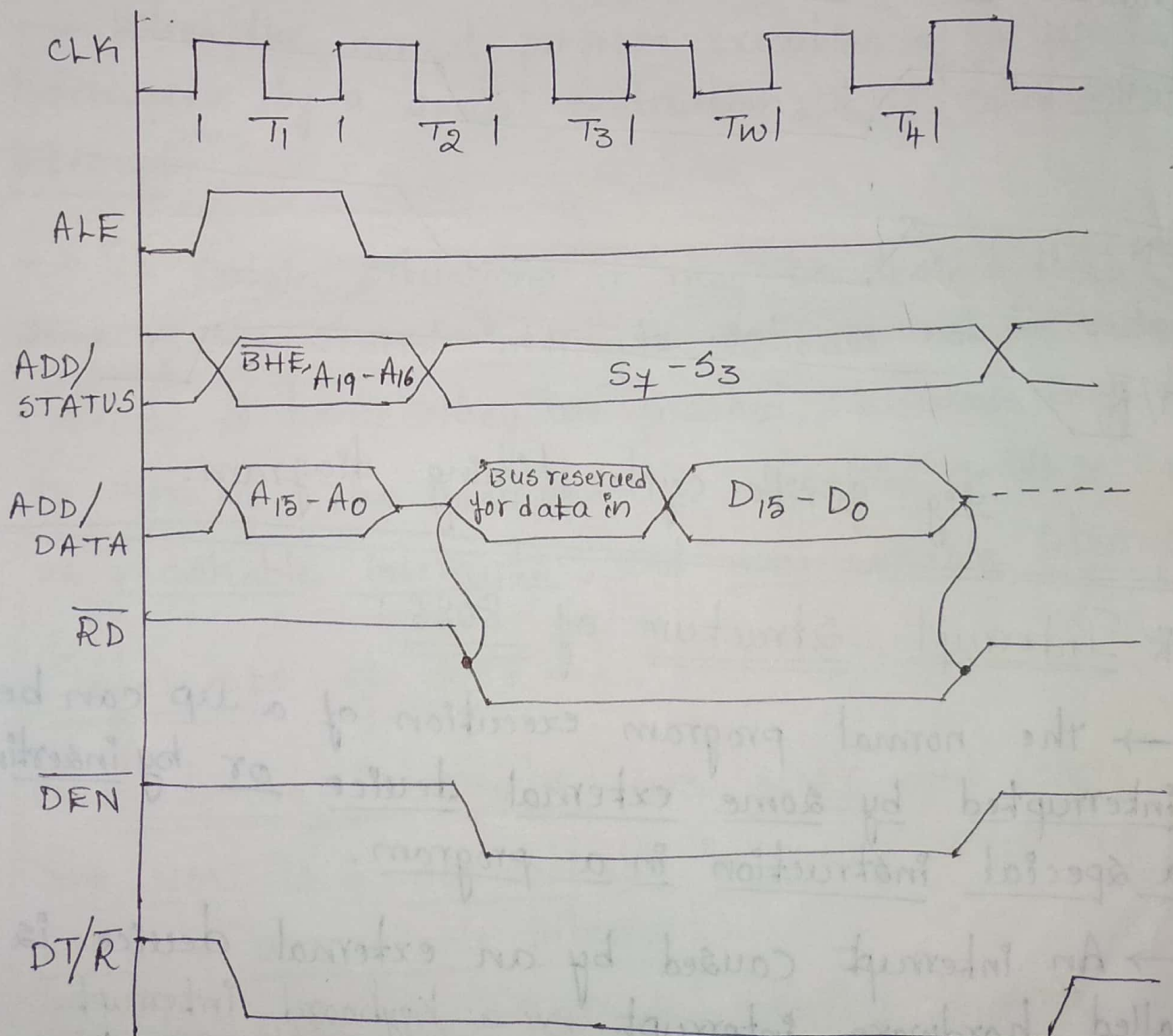


Fig: Read cycle timing diagram.

→ Write cycle:

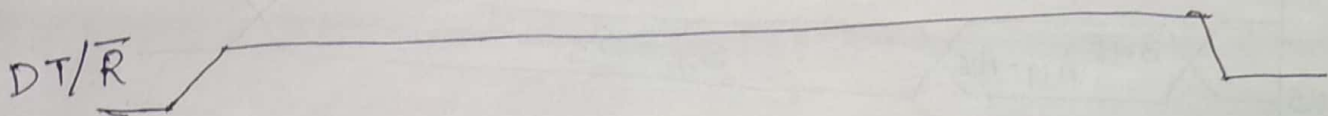
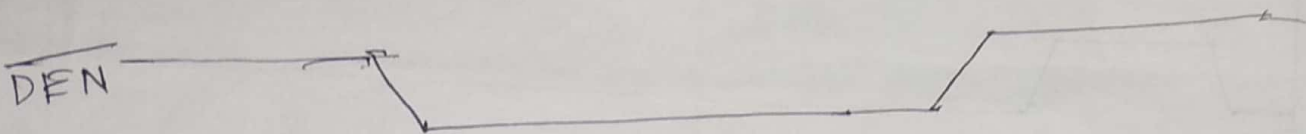
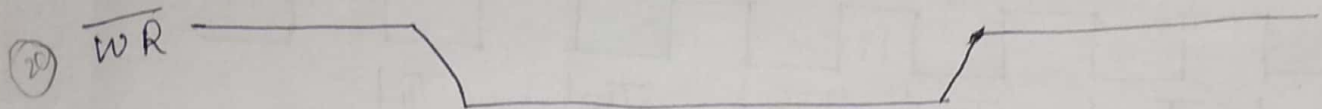
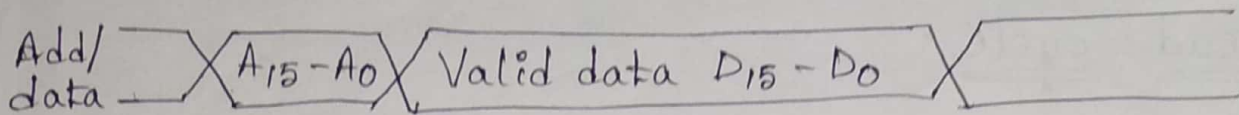
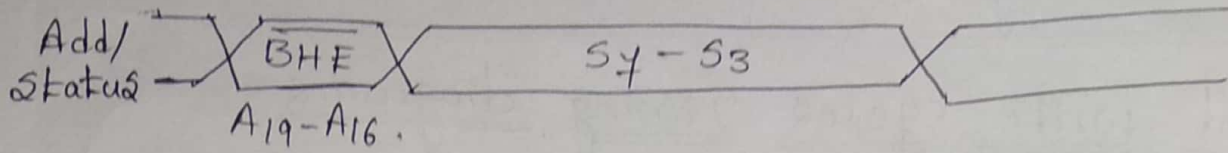
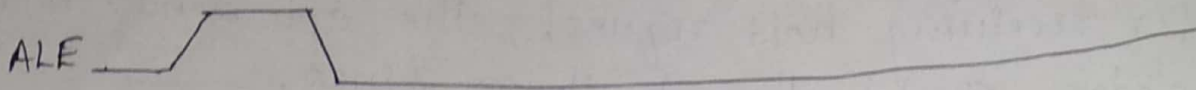
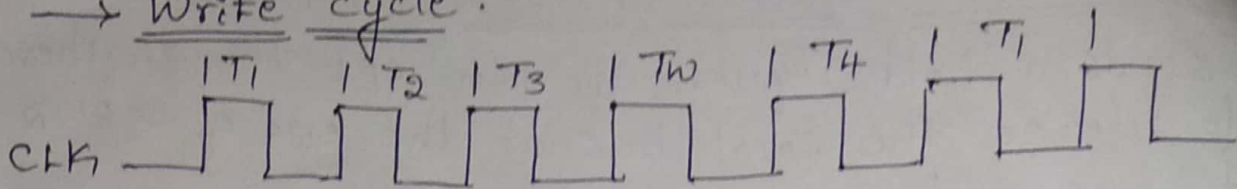


Fig: Write cycle timing diagram.

* Interrupt Structure of 8086:

→ The normal program execution of a µp can be interrupted by some external device or by inserting a special instruction in a program.

→ An interrupt caused by an external device is called hardware interrupt. Ex: a keyboard interrupt.

→ When an external device wants to interrupt

1	0	Empty queue.
1	1	

the μp , it sends an interrupt signal to the μp through the interrupt line to which it is connected (11)

→ The μp executes a subroutine to transfer data from the interrupting device. This subroutine is called interrupt service subroutine (ISS).

→ When data transfer is completed the μp returns back to the main program which it was executing before the interrupt occurred. (21)

→ When the normal program execution of a μp is interrupted by a special instruction, it is called software interrupt.

→ In certain situations it may be desired that some of the several interrupts should not be entertained, & hence they are masked, i.e. made ineffective by software. The interrupts are classified on this basis, as maskable interrupts, and nonmaskable interrupts.

Interrupts of 8086:

① Hardware interrupts: are NMI and INTR. The NMI is a nonmaskable interrupt where as INTR is a maskable interrupt.

→ The NMI has highest priority than INTR.

② Software interrupt: is generated by inserting INT instruction in a program. Any of the 256 interrupt types (0-255) can be generated specifying the interrupt type after INT instruction.

→ The instruction INTO generates only overflow interrupt.

- ①. Divide by zero interrupt (Type 0).
- ②. Single step interrupt (Type 1).
- ③. Non Maskable Interrupt (Type 2).
- ④. Breakpoint interrupt (Type 3).
- ⑤. Overflow interrupt (Type 4).

Interrupt priorities:

Interrupt	Priority
Divide Error, Int n, Int 0.	Highest.
NMI	↓
INTR	↓
Single - step	Lowest.

→ "Interrupt" is to break the sequence of operation.

→ While the CPU is executing a program, an interrupt breaks the normal sequence of execution of instructions, diverts its execution to some other program called Interrupt Service Routine (ISR).

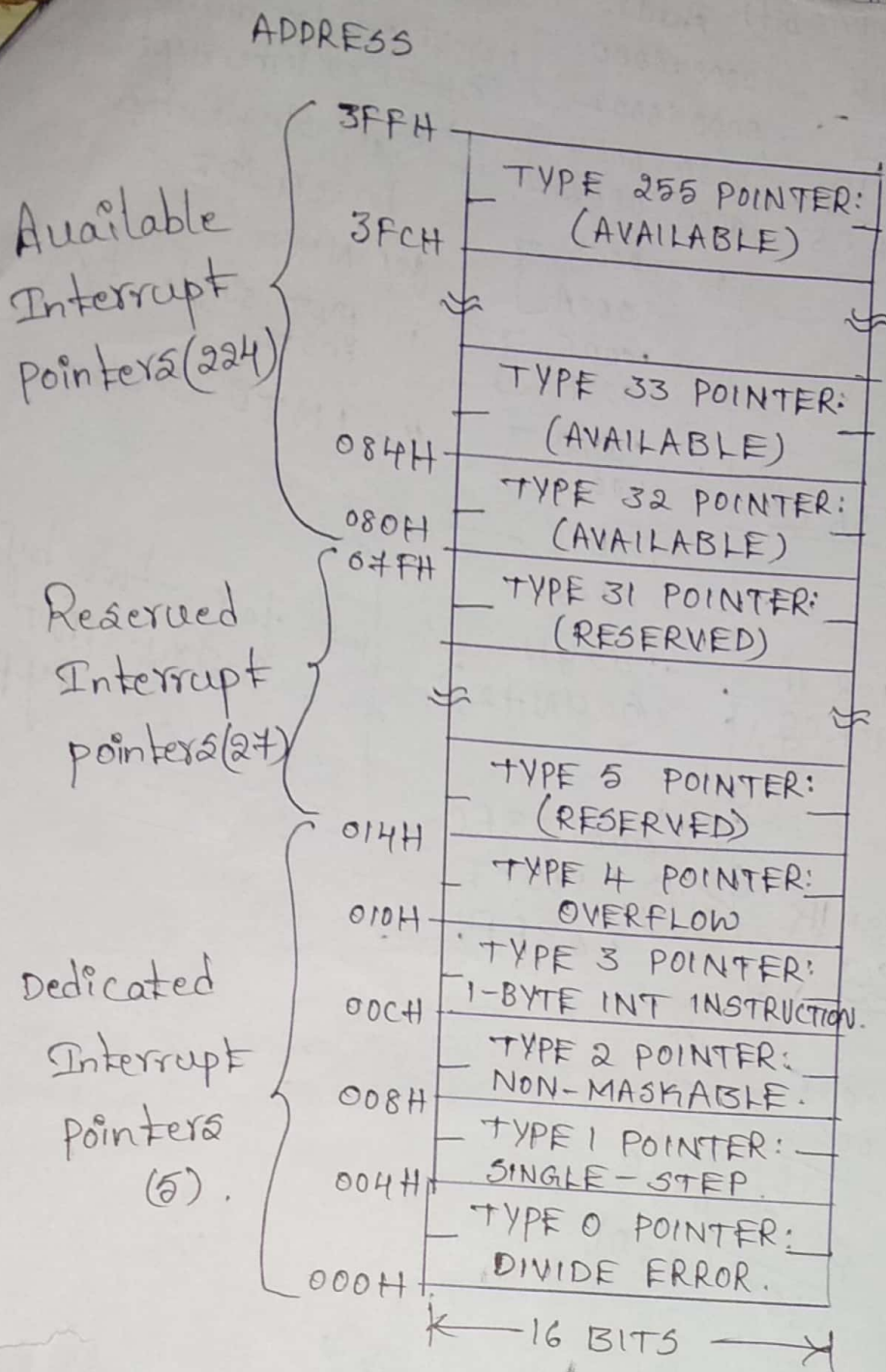


Fig:

Interrupt structure
of
8086.

(23)

→ The interrupt within the interrupt is called "nested interrupt."

→ In 8086, there are two hardware interrupt pins NMI and INTR.

→ NMI is nonmaskable which means that any interrupt request at NMI input cannot be masked or disabled by any means.

Instruction Set and Assembly Language Programming of 8086.

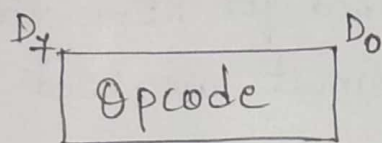
* Instruction format:

→ The instruction format for 1 to 6 bytes instruction for each instruction format first field is the Operation code field, commonly known as Opcode field.

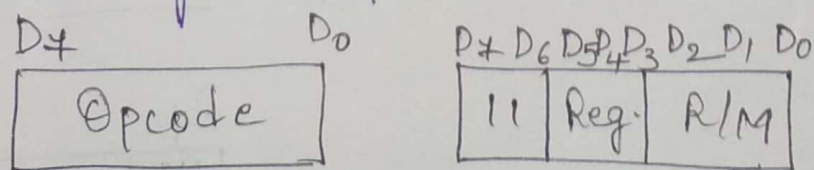
→ Opcode field indicates the type of operation to be performed by the processor.

→ The other field in the instruction format is Operand field. may consists of source/destination operand, source operand address, destination operand address or next instruction address.

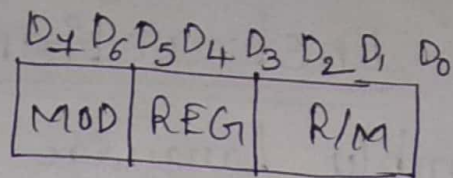
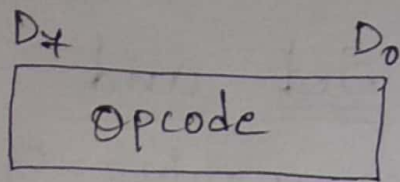
1. One byte Instruction: Implied Operands.



2. Register to Register: This format is 2 bytes long. The first byte of the code specifies the operation code. The second byte of the code shows the register operands. & R/M field.



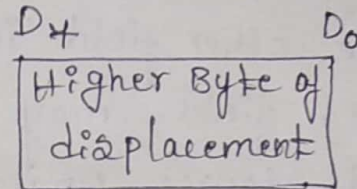
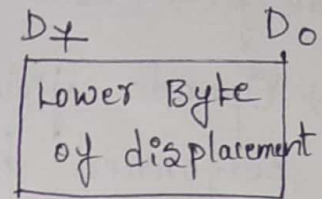
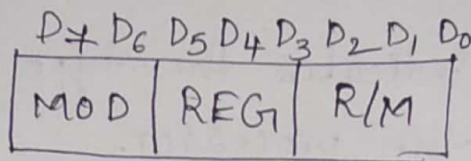
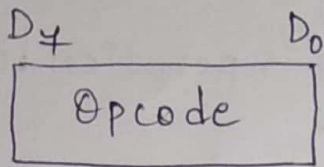
3. Register to/from memory with no displacements
It is also 2 bytes long.



→ MOD field shows the mode of addressing.

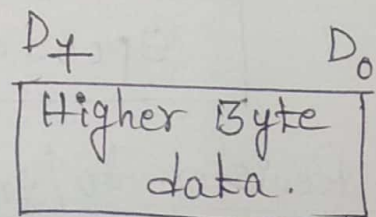
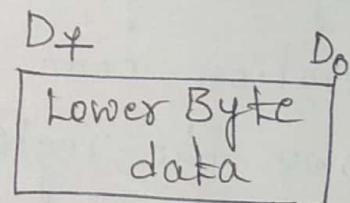
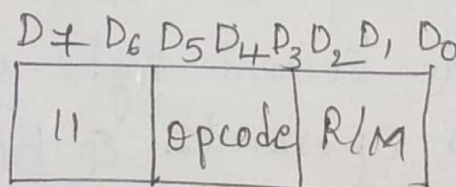
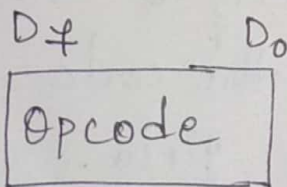
4. Register to/from memory with displacement:

→ It contains one or two additional bytes for displacement along with 2-byte the format of the register to/from memory without displacement.

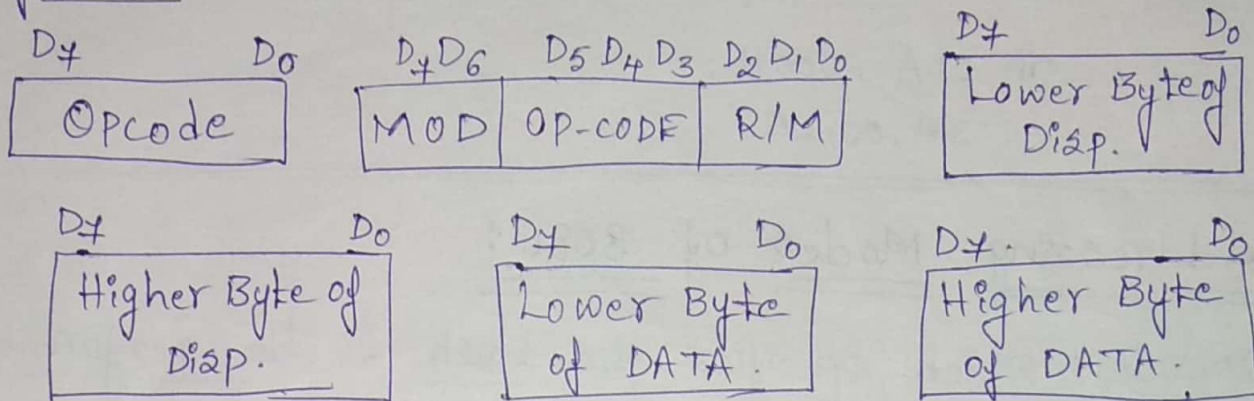


5. Immediate Operand to Register: In this, the first byte as well as the 3-bits from the second byte which are used for REG field in case of register to register format are used for opcode.

→ It also contains one or two bytes of immediate data.



6. Immediate Operand to Memory with 16-bit Displacement: It requires 5 or 6 bytes for coding. The first 2 bytes contain the information regarding opcode, MOD, & R/M fields. The remaining 4 bytes contain 2 bytes of displacement & 2 bytes of data.



* Addressing Modes of 8086:

→ One byte instruction:

- EX: 1. MOV A, B
2. ADD B.
3. RAL

→ Two-Byte Instruction:

EX: 1. MOV B, 05; Move 05 to reg. B.
06, 05; MOV B, 05 in the code form.

The 1st byte 06 is the opcode for MOV B & the 2nd byte 05 is the data.

2. IN 01; Read data at port B.
DB, 01; IN 01 in the code form.
DB is the opcode for instruction IN & 01 address for

→ Three-Byte Instruction:

(i) LXI H, 2400H ; Load H-L Pair with 2400H.
21, 00, 24 ; ~~xx~~: In code form.

The 1st byte 21 is opcode for LXI H. The 2nd byte 00 is 8LSBs of data, loaded into reg. L. The 3rd byte 24 is 8MSBs of data, loaded into reg. H.

(ii) LDA 2500H ;
3A, 00, 25.

* Addressing Modes of 8086:

Opcode: which specifies the task to be performed by the computer is called opcode.

Operand: the data to be operated on, called operand.

→ Addressing mode indicates a way of locating data or operands.

→ The 8086 has 8-addressing modes.

① Immediate Addressing: The operand is specified in the instruction itself.

Ex: 1. MOV AX, 0005H.

→ 0005H is the immediate data.

2. MOV AL, 35H.

5. ADD AX, 4836H.

3. MOV BX, 0301H.

4. MOV [0401], 3598H.

② Direct Addressing: A 16-bit memory address (offset) is directly specified in the instruction as a part of it. ③

1. MOV AX, [5000H].

2. ADD AL, [0301] → Add the content of offset address 0301 to AL. The operand or data is placed at the given offset (0301) within the DS.

3. ADD [0301], AX.

③ Register Addressing: The data is stored in a register.

Ex: 1. MOV BX, AX.

2. MOV AX, CX.

3. ADD AL, BL.

4. ADD CX, DX.

④ Register Indirect addressing: In this mode, the offset address of data is in either BX or SI or DI register.

Ex: 1. MOV AX, [BX] → Data is present in a memory location, whose offset address is in BX.

• For suppose BX contains 0301, & the content of 0301 is 53H & the content of next memory location is 95H. The 9553H will move to AX.

2. ADD AL, [SI].

⑤ Indexed Addressing: Offset of the data is stored in one of Index registers SI or DI.

Ex: 1. MOV AX, [SI] → Data is available at an offset

address stored in SI.

2. MOV AX, [SI+05]

3. MOV AX, [SI+1528H]

⑥ Based Addressing: or Register Relative addressing:

The operand's offset is the sum of an 8-bit or 16-bit displacement and the contents of the base register BX or BP.

1) MOV AX, 50H[BX].

→ Move the content 50H + BX to AX.

2. MOV AL, [BX+05]

→ Suppose, the register BX contains 0301. The offset will be $0301 + 05 = 0306$. The content of memory location 0306 will move to AL.

3. MOV AL, [BX+1346H]

→ If [BX] = 0301. The offset = $0301 + 1346 = 1647H$.

The content of 1647H will move to AL.

⑦ Based Indexed addressing: The effective address of data is formed, by adding content of base register (BX or BP) to content of index register (SI or DI).

Ex: 1. MOV AX, [BX][SI].

↓ ↓
Base reg. Index reg.
Base + Index

2. ADD AX, [BX+SI]

3. MOV CX, [BX+SI].

⑧ Relative Based Indexed addressing:

1. MOV AX, 50H[BX][SI] → $50H + [BX] + [SI]$

2. MOV AX, [BX+SI+1235H] → 16-bit displacement.

3. MOV AX, [BX+SI+05] → 8-bit displacement.

* Instruction Set of 8086:

(4)

(i) Data copy / Transfer Instructions:

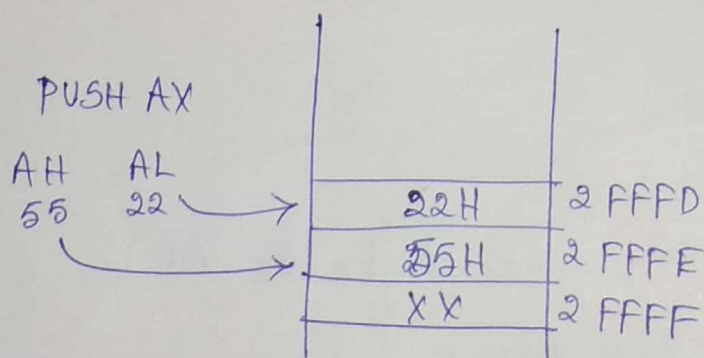
→ These type of instructions are used to transfer data from source operand to destination operand.

→ All the store, move, load, exchange, input and output instructions.

MOV: MOVE → Load DS with 5000H.

1. MOV DS, 5000H. → Invalid.
2. MOV AX, 5000H.
MOV DS, AX.
3. MOV AX, 5000H;
4. MOV AX, BX;
5. MOV AX, [SI];
6. MOV AX, [2000H];
7. MOV AX, 50H[BX];

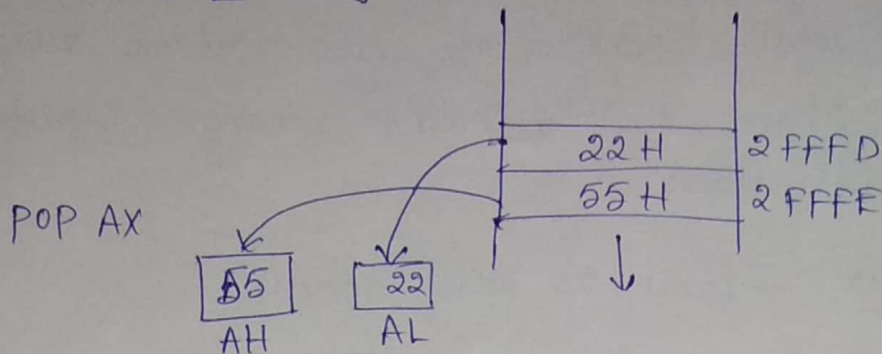
PUSH: Push to stack: This instruction pushes the contents of the specified register/memory location on to the stack. The stack pointer is decremented by 2, after execution of each instruction.



- Ex:
1. PUSH AX
 2. PUSH DS
 3. PUSH [5000H]

POP: POP from Stack: The sp is incremented by 2.

- EX:
1. POP AX
 2. POP DS
 3. POP [5000H]



XCHG: Exchange: Exchanges the contents of the specified source and destination operands.

- EX:
1. XCHG [5000H], AX
 2. XCHG BX, AX

IN: Input the port: reading an input port.

- EX:
1. IN AL, 03H
 2. IN AX, DX
 3. MOV DX, 0800H
IN AX, DX

OUT: Output to the port: Writing to an o/p port.

- EX:
1. OUT 03H, AL
 2. OUT DX, AX
 3. MOV DX, 0300H
OUT DX, AX

XLAT: Translate: (Translate byte to AL).

```
MOV AX, SEG TABLE
MOV DS, AX
MOV AL, CODE
MOV BX, OFFSET TABLE
XLAT
```

LEA: Load Effective Address.

LDS/LES: Load pointer to DS/ES.

LAHF: Load AH from Lower Byte of Flag.

SAHF: Store AH to lower Byte of Flag Reg.

PUSHF: Push Flags to stack.

POPF: POP Flags from stack.

(ii) Arithmetic Instructions:

ADD: Add:

1. ADD AX, BX

3. ADD AX, [SI]

4. ADD AX, [5000H]

5. ADD [5000H], 0100H

6. ADD 0100H.

ADC: Add with carry:

1. ADC AX, BX

2. ADC AX, [SI]

3. ADC AX, [5000H]

INC: Increment: Increased the contents of specified register or memory location by 1.

1. INC AX

2. INC [BX]

3. INC [5000H].

DEC: Decrement: Subtracts 1 from the contents of specified register or memory location.

1. DEC AX.

2. DEC [5000H]

SUB: Subtract: Subtracts source operand from destination operand

1. SUB AX, 0100H.

2. SUB AX, BX

3. SUB AX, [5000H]

SBB: subtract with Borrow:

1. SBB AX, 0100H.
2. SBB AX, BX
3. SBB AX, [5000H]
4. SBB [5000H], 0100

cmp: compare: For comparison, it subtracts the source operand from the destination operand but does not store the result anywhere.

The flags are affected depending upon the result of the subtraction. If both of the operands are equal, Zero flag is set. If the source operand is greater than the destination operand, carry flag is set or else, carry flag is reset.

1. CMP BX, 0100H.
2. CMP AX, 0100H.
3. CMP BX, CX.
4. CMP BX, [3F].

AAA: ASCII adjust after Addition: It is used in BCD addition. Only the AF & CF flags are set to 1. For lower nibble, AF is set & upper nibble CF will activate.

AAS: ASCII adjust after subtraction: "

AAM: ASCII adjust after Multiplication: PF, SF & ZF

AAD: ASCII adjust before Division: PF, SF, Zero flag are active.

DAA: Decimal Adjust Accumulator: In BCD addition,

If the lower nibble is greater than 9, it will add 06 to lower nibble. AF is set.

⑥

DAS: Decimal Adjust after subtraction

MUL: Unsigned Multiplication Byte or word. →

MUL BX
MUL CX
MUL [SI]

DIV: Unsigned Division

IDIV: signed Division

IMUL: signed multiplication:

*Logical Instructions:

AND: Logical AND:
AND AX, BX
AND AX, 0008H
AND AX, [5000H]
AND [5000H], DX.

EX: If AX content is 3F0FH.

0011	1111	0000	1111	= 3F0FH [AX]
↓↓↓↓	↓↓↓↓	↓↓↓↓	↓↓↓↓	AND
0000	0000	0000	1000	= 0008H.
0000	0000	0000	1000	= 0008H [AX]

→ The result will be in AX.

OR: Logical OR:
OR AX, 0098H.
OR AX, BX
OR AX, [5000H]
OR [5000H], 0008H.

NOT: Logical Invert:

NOT AX

NOT [5000H]

XOR: Logical Exclusive OR.

XOR AX, 0098H

XOR AX, BX

XOR AX, [5000H]

SHL/SAR: shift logical / Arithmetic left.

SHR: shift logical Right.

SAR: shift Arithmetic Right.

ROR: Rotate Right without carry

ROL: Rotate left without carry

RCR: Rotate Right through carry.

RCL: Rotate left through carry.

* Branch Instructions: a loop Instructions:

CALL: Unconditional call.

RET: Return from the procedure.

INT N: Interrupt Type N. → EX: INT 21H.

INTO: Interrupt on Overflow.

JMP: Unconditional Jump

LOOP: Loop unconditionally

EX: MOV CX, 0005.

MOV BX, 0FF7H.

Label: MOV AX, CODE1.

OR BX, AX.

AND DX, AX.

Loop Label.

4

JZ/JE: Jump on Zero/Equal.

JNZ/JNE: Jump on Not Zero/Equal

JS: Jump on Sign.

JNS: Jump on NOT Sign.

JO: Jump on Overflow.

JNO: Jump on NOT overflow.

JP/JPE: Jump on parity/parity Even.

JNP: Jump on Not parity.

JB/JNAE/JC: Jump on Below/Not above or equal carry.

JNB/JAE/JNC: Jump not Below/above equal/No carry.

JNL/JGE: Jump on not less/greater or equal.

JLE/J^{NG}~~LE~~: Jump on less or equal/Not greater

* Flag Manipulation and Process Control

Instructions:

CJC — clear carry flag.

CMC — complement carry flag.

STC — set carry flag.

CJD — clear direction flag.

STD — set " ".

CLI — clear Interrupt flag.

STI — set " ".

(a) Flag Instructions.

WAIT.

HLT - Halt.

NOP - No Operation.

ESC - Escape

LOCK.

(b) Processor control Instructions.

* String Manipulation Instructions.

→ A series of data bytes or words available in memory at consecutive locations, to be referred to collectively or individually, are called as byte strings or word strings.

CMPS: Compare string Byte or string word

SCAS: Scan string Byte or string word

LODS: Load string Byte or string word

STOS: Store string Byte or string word

* Assembler Directives:

→ The assembly language programming is simpler as compared to machine language programming.

→ The instruction mnemonics are directly written in the assembly language programs.

→ The advantage of assembly language is, the address values of the constants can be identified by labels.

→ Each time the programmer will not have to ⑧ remember the different constants & the addresses at which they are stored, throughout the programs.

→ An Assembler is a program used to convert an assembly language program into the equivalent machine code modules which may further be converted to executable codes.

→ It decides the address of each label and substitutes the values for each of the constants & variables. It then forms machine code for these.

→ While doing these things, the assembler may find out syntax errors.

→ For completing all these tasks, an assembler needs some hints from the programmer i.e. the required storage for a particular constant or a variable, logical names of the segments, end of files, etc.

These type of hints are given to the assembler using some predefined alphabetical strings called assembler directives, which help the assembler to correctly understand the assembly language

programs to prepare the codes.

→ Microsoft Macro Assembler or Turbo Assembler are commonly used assemblers.

DB: Define Byte:

It is used to reserve byte or bytes of memory locations in the available memory.

EX: RANKS DB 01H, 02H, 03H, 04H

This statement reserve four memory locations for a list named RANKS.

EX: MESSAGE DB "GOOD MORNING"

In this no. of bytes of memory equal to the no. of characters in the string named MESSAGE.

DW: Define Word:

It reserve the no. of memory words (16-bits) instead of bytes.

EX: WORDS DW 1234H, 4567H, 78ABH, 045CH.

DQ: Define Quadword:

It reserve 4 words (8 bytes) of memory.

DT: Define Ten Bytes:

It is used to define the specified variable requiring 10-bytes for its storage.

ASSUME: Assume Logical Segment Name:

It is used to inform the assembler, the names of the logical segments to be assumed for different segments used in the program.

EX: ASSUME CS: CODE

ASSUME DS : DATA.

(9)

END: END of program:-

It marks the end of an assembly language program.

ENDP: END of procedure:

The subroutines are called procedures. The ENDP directive is used to indicate the end of a procedure. A procedure is usually assigned a name i.e. label.

PROCEDURE STAR

⋮

STAR ENDP.

ENDS: END of segment:

This directive marks the end of a logical segment. The logical segments are assigned with the names using the ASSUME directive. The names appear with the ENDS directive as prefixes to mark the end of those particular segments.

DATA

SEGMENT

⋮

DATA

ENDS.

ASSUME
CODE

CS: CODE, DS: DATA
SEGMENT

⋮

CODE
END

ENDS

EVEN: Align on Even Memory Address:

It updates the location counter to the next even address, if the current location counter contents are not even, then assigns even address.

```
        EVEN
PROCEDURE  ROOT
        ;
        ROOT  ENDP.
```

EQU: Equate:

This is used to assign a label with a value or a symbol.

EX: 1. LABEL EQU 0500H.
2. ADDITION EQU ADD.

EXTRN: External and PUBLIC: Public:

→ The "EXTRN" informs the assembler that the names, procedures & labels declared after this directive have already been defined in some other assembly language modules.

→ While in other module, where the names, procedures & labels actually appear, they must be declared public, using "PUBLIC".

```
MODULE1      SEGMENT
PUBLIC
MODULE1      FACTORIAL
MODULE1      ENDS
MODULE2      SEGMENT
EXTRN
MODULE2      FACTORIAL
MODULE2      ENDS.
```

GROUP : Group the Related Segments:

It is used to form logical groups of segments with similar purpose or type.

PROGRAM GROUP CODE, DATA, STACK.

ORG : Origin:

→ This directs the assembler to start the memory allotment for the particular segment, block or code from the declared address in the ORG statement.

→ If ORG statement not written in program, the location counter is initialised to 0000. If an ORG 0200H statement is present at the starting, then code will start from 0200H address.

PROC : Procedure:

RESULT PROC NEAR

ROUTINE PROC FAR.

→ The statement RESULT PROC NEAR marks the start of a routine RESULT, which is to be called by a program located in the same segment of memory.

→ The FAR is used for the procedures to be called by the programs located in different segments of memory.

* Macros : (MACROS) :

→ Suppose, a no. of instructions are repeating through in the main program, the listings becomes lengthy.

→ So, a macro definition i.e. a label, is assigned with the repeatedly appearing string of instructions.

→ The process of assigning a label or macroname to the string is called defining a macro.

→ A macro within a macro is called a nested macro. A macro may be defined in another macro or in other words a macro may be called from inside a macro.

→ The macroname or macro definition is then used throughout the main program to refer to that string of instructions.

* Defining a MACRO :

→ A macro can be defined anywhere in a program using the directives MACRO and ENDM.

→ The ENDM directive marks the end of the instruction assigned with the macro name.

```
DISPLAY MACRO  
    MOV AX, SEG MSG  
    MOV DS, AX
```

MOV DX, OFFSET MSG

(11)

MOV AH, 09H

INT 21H

ENDM

→ While assembling, the above sequence of instructions will replace the label 'DISPLAY', whenever it appears in the program.

* Write a program to add a data byte located at offset 0500H in 2000H segment to another data byte available at 0600H in the same segment and store the result at 0700H in the same segment.

MOV AX, 2000H

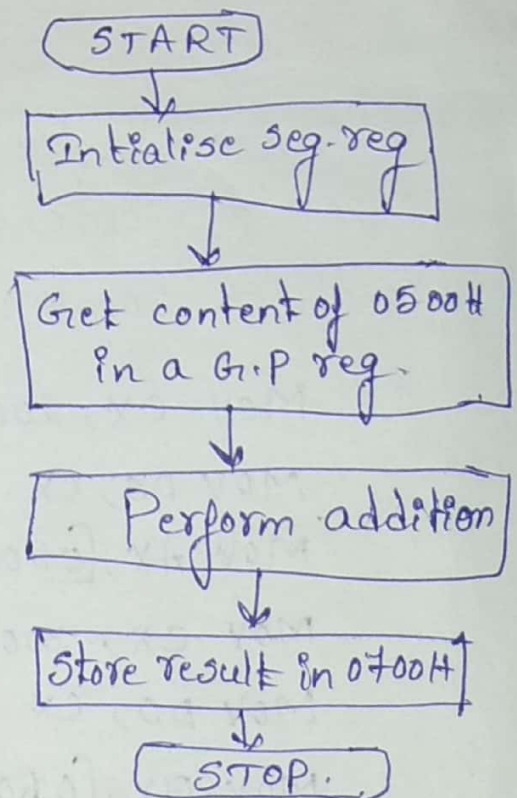
MOV DS, AX

MOV AX, [0500H]

ADD AX, [0600H]

MOV [0700H], AX

HLT



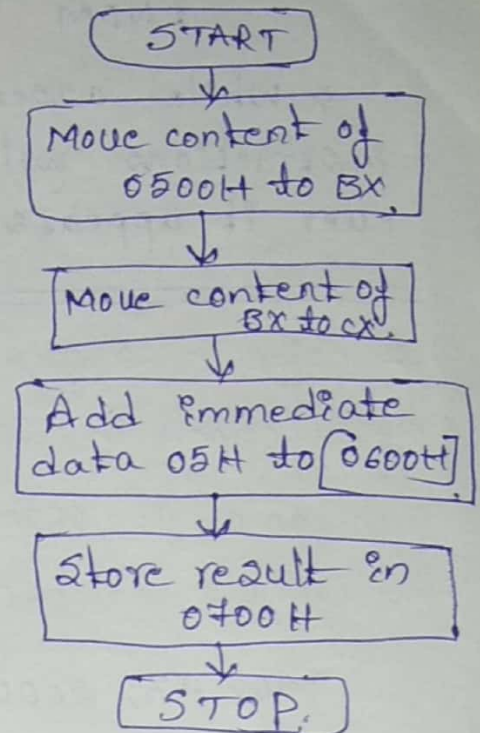
* Write a program to move the contents of the memory location 0500H to reg. BX & to CX. Add Immediate Byte 05H to the data residing in memory location, whose address is computed

using DS = 2000H & offset = 0600H. Store the result in 0700H of addition. Assume that the data is in DS which contain 2000H.

```

MOV AX, 2000H
MOV DS, AX.
MOV BX, [0500H]
MOV CX, BX
ADD [0600H], 05H.
MOV DX, [0600H]
MOV [0700H], DX
HLT.

```



* Add the contents of the memory location 2000H 0500H to contents of 3000H:0600H and store the result in 5000H:0700H.

```

MOV CX, 2000H
MOV DS, CX.
MOV AX, [0500H]
MOV CX, 3000H
MOV DS, CX
MOV BX, [0600H]
ADD AX, BX
MOV CX, 5000H.
MOV DS, CX
MOV [0700H], AX
HLT.

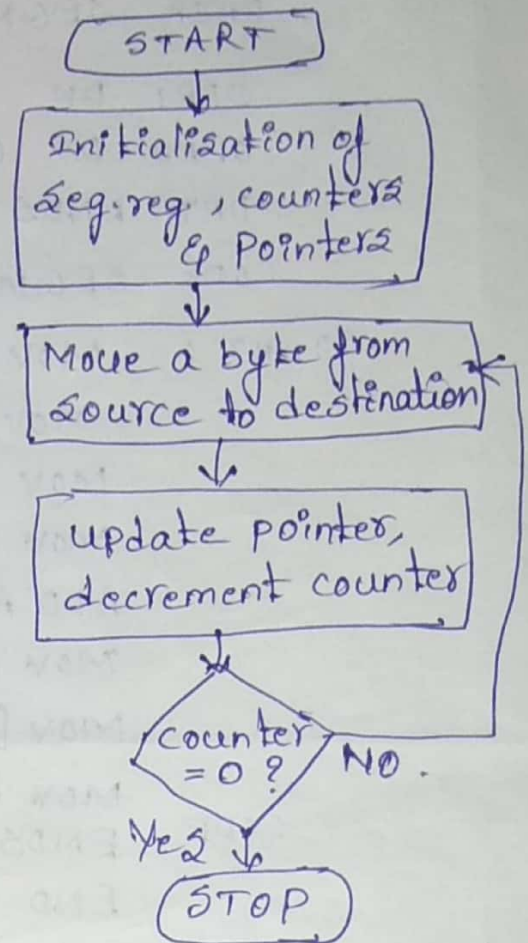
```

* Move a byte string, 16-bytes long, from the offset 0200H to 0300H in the segment 4000H. (12)

```

MOV AX, 4000H.
MOV DS, AX
MOV SI, 0200H.
MOV DI, 0300H.
MOV CX, 0010H.

BACK: MOV AL, [SI]
      MOV [DI], AL
      INC SI
      INC DI
      DEC CX
      JNZ BACK
      HLT.
  
```



* Find out the largest number from an array of sixteen 8-bit numbers in the memory locations starting at offset 0500H in the segment 2000H.

```

MOV CX, 0FH.
MOV AX, 2000H.
MOV DS, AX.
MOV SI, [0500H]
MOV AL, [SI]

BACK: INC SI
      CMP AL, [SI]
      JNL NEXT
      MOV AL, [SI]
  
```

```

NEXT: LOOP BACK
      HLT.
  
```


* Write a program for addition of two numbers.

ASSUME CS: CODE, DS: DATA.

DATA SEGMENT.

OPR1 DW 1234H

OPR2 DW 0002H

DATA ENDS.

CODE SEGMENT

START: MOV AX, DATA

MOV DS, AX

MOV AX, OPR1

MOV BX, OPR2

ADD AX, BX

MOV DI, OFFSET RESULT.

MOV [DI], AX

~~MOV~~ INT 21H

CODE ENDS

END START.

* Write a program for the addition of a series of 8-bit numbers. The series contains 100 (numbers)

ASSUME CS: CODE, DS: DATA

DATA SEGMENT

NUMLIST DB 52H, 23H, -

COUNT EQU 100D

DATA ENDS

CODE SEGMENT

START: MOV AX, DATA

MOV DS, AX.

MOV CX, COUNT.

MOV SI, NUMLIST. OFFSET

```

AGAIN:  MOV BL, [SI]
        ADD AX, BX
        INC SI
        DEC CX
        JNZ AGAIN
        MOV DI, OFFSET RESULT
        MOV [DI], AX
        INT 21H
        CODE ENDS

        END START.

```

* A program to find out the largest number from a given ~~unordered array of 8-bit numbers~~ ^{Series of words}, stored in the locations starting from a known address.

ASSUME CS: CODE, DS: DATA

DATA SEGMENT

LIST DW 1234H, 2354H, 0056H, 045AH, -

COUNT EQU OF

DATA ENDS

CODE SEGMENT

START: MOV AX, DATA

MOV DS, AX

MOV SI, OFFSET LIST

MOV CL, COUNT

MOV AX, [SI]

AGAIN: CMP AX, [SI+2]

JNL NEXT

MOV AX, [SI+2]


```

NEXT:  INC SI
        INC SI
        DEC CL
        JNZ AGAIN
        MOV SI, OFFSET LARGEST
        MOV [SI], AX
        INT 21H
CODE   ENDS
        END START.

```

* Write a program to find out the no. of positive numbers & negative numbers from a given series of signed numbers.

→ The status of carry flag, i.e. the most significant bit of the number will give the information about the sign of the number. Rotate the no. left through carry.
 → If CF is 1, the no. is -ve; otherwise, it is +ve.

```

ASSUME CS:CODE, DS:DATA

```

```

DATA SEGMENT

```

```

LIST DW 2579H, 0A500H, 0C009H, 0159H,
      0B900H.

```

```

COUNT EQU 05H

```

```

DATA ENDS

```

```

CODE SEGMENT

```

```

START:  MOV AX, DATA      → XOR BX, BX
        MOV DS, AX        XOR DX, DX.

```

```

        MOV CL, COUNT

```

```

        MOV SI, OFFSET LIST

```

```

AGAIN:  MOV AX, [SI]

```

```

SHL AX, 01
JC NEG
INC BX
JMP NEXT
NEG: INC DX
NEXT: ADD SI, 02
      DEC CL
      JNZ AGAIN
      INT 21H
      COD ENDS
      END START.

```

* A program to find out the number of even and odd numbers from a given series of 16-bit hexadecimal numbers.

→ To check the LSB of the number, If the bit is zero, the number is even, otherwise it is odd. Check the LSB by rotating the number through carry flag & increment even or odd number counter.

ASSUME CS: CODE, DS: DATA

DATA SEGMENT

LIST DW 2357H, 0A579H, 0C322H, 0C91EH,
0C000H, 0957H,

COUNT EQU 06H

DATA ENDS

CODE SEGMENT

```

START: XOR BX, BX
      XOR DX, DX
      MOV AX, DATA
      MOV DS, AX

```



```

MOV CL, COUNT
MOV SI, OFFSET LIST
AGAIN: MOV AX, [SI]
      ROR AX, 01
      JC ODD
      INC BX
      JMP NEXT
ODD:   INC DX
NEXT:  ADD SI, 02
      DEC CL
      JNZ AGAIN
      INT 21H
      CODE ENDS
      END START.

```

* Write a program to move a string of data words from offset 2000H to offset 3000H, the length of the string is 0FH.

```

MOV SI, 2000H
MOV DI, 3000H
MOV CX, 0FH.
AGAIN: MOV AX, [SI]
      MOV [DI], AX.
      ADD SI, 02H
      ADD DI, 02H
      DEC CX
      JNZ AGAIN
      HLT

```

⇒ By using string command or instruction the program below.

```

ASSUME CS: CODE, DS: DATA
DATA SEGMENT
SOURCESTR EQU 2000H
DESTSTR    EQU 3000H
COUNT EQU 0FH
DATA ENDS
CODE SEGMENT
START:  MOV AX, DATA
        MOV DS, AX
        MOV SI, SOURCESTR
        MOV DI, DESTSTR
        MOV CX, COUNT
        CLD
REP     MOVSW
        MOV INT 21H
CODE ENDS
END START

```

* Write an assembly language program to arrange a given series of hexadecimal bytes in ascending order.

```

ASSUME CS: CODE, DS: DATA
DATA SEGMENT
LIST DW 53H, 25H, 19H, 02H
COUNT EQU 04
DATA ENDS
CODE SEGMENT
START:  MOV AX, DATA
        MOV DS, AX

AGAIN0: MOV CX, COUNT-1
        MOV SI, OFFSET LIST
AGAIN1: MOV AX, [SI]

```


	CMP AX, [SI+2]	
	JL PRI	53, 25
	XCHG [SI+2], AX	SI, 53
	XCHG [SI], AX	1, 2
PRI:	ADD SI, 02	25, 19
	LOOP AGAIN1	AX, SI
	DEC SI	19, 25
	INX AGAIN0	19, 02
	INT 21H	19, 02
	ENDS	02
CODE	END START	

* Write a program that performs addition, subtraction, multiplication & division of the given operands.

* Display the message "the study of microprocessors is interesting", on the CRT screen of a microcomputer.

ASSUME CS: CODE, DS: DATA

DATA SEGMENT

MESSAGE DB "STUDY OF MICROPROCESSORS
IS INTERESTING"

DATA ENDS

CODE SEGMENT

START: MOV AX, DATA
MOV DS, AX
MOV DX, OFFSET MESSAGE.
INT 21H.

CODE ENDS

END START

UNIT-II

①

INTRODUCTION TO MICROCONTROLLERS

* Microcontroller:

- A microcontroller is a microprocessor with integrated peripherals.
- All the peripherals are integrated into a single chip.
- It contains CPU, ROM, RAM, input device, Output device, timers & counters, ADC, DAC, serial ports, interrupt logic, Oscillator circuitry & other peripherals on a single chip.
- All these functional blocks on a single Integrated circuit (IC), results into a reduced size of control board, low power consumption, more reliability & ease of integration within an application design.
- The usage of microcontrollers not only reduces the cost of automation, but also provides more flexibility.

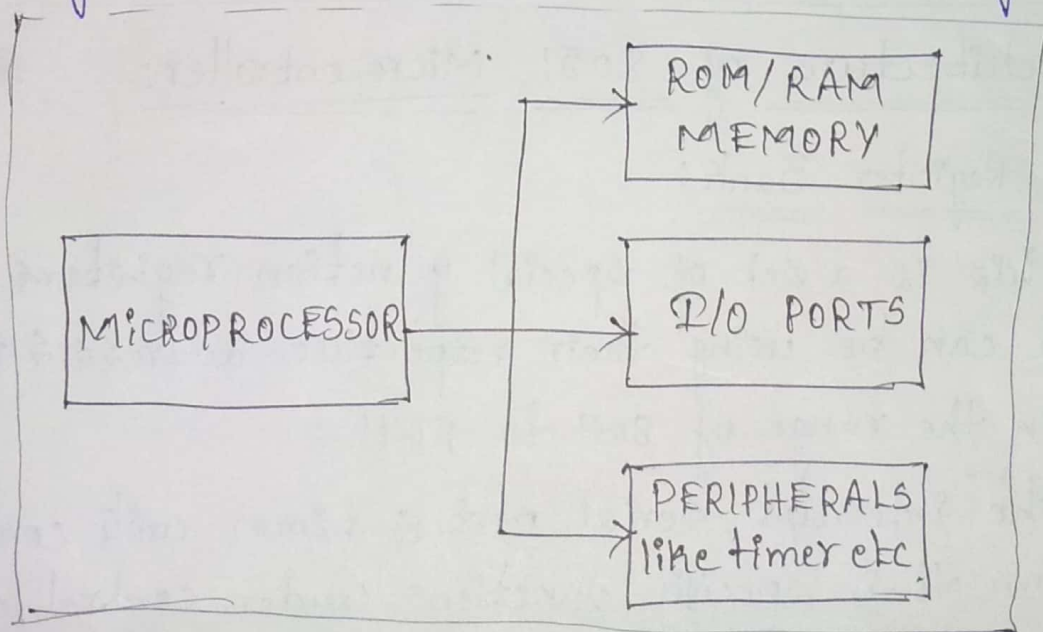


Fig: Microcontroller Internal Block diagram.

→ Many industrial & domestic products like remote controllers, telephone bill printing machines, automatic power regulators, automatic or semi-automatic washing machines, microwave ovens, automobiles, engines, indicating & measuring instruments uses microcontroller.

* 8051 Microcontroller:

* Features:

1. It is a 8-bit microcontroller.
 2. It has a 16-bit address bus.
 3. 4K⁽⁵¹²⁾ bytes of Internal ROM (Program memory) to 64K.
 4. 128 Bytes of RAM (Data memory) to 64K bytes.
 5. 4 I/O ports (8-bit)
 6. 2 16-bit timers & counters.
 7. Serial Interface.
 8. Two external Interrupts $\overline{INT0}$ & $\overline{INT1}$.
 9. Operating frequency is 12MHz.
-

* Architecture of 8051 Microcontroller:

SFR Register Bank:

→ This is a set of special function registers, which can be using their respective addresses which lie in the range of 80H to FFH.

→ The interrupt, serial port & timer unit control perform their specific functions under control of timing & control unit.

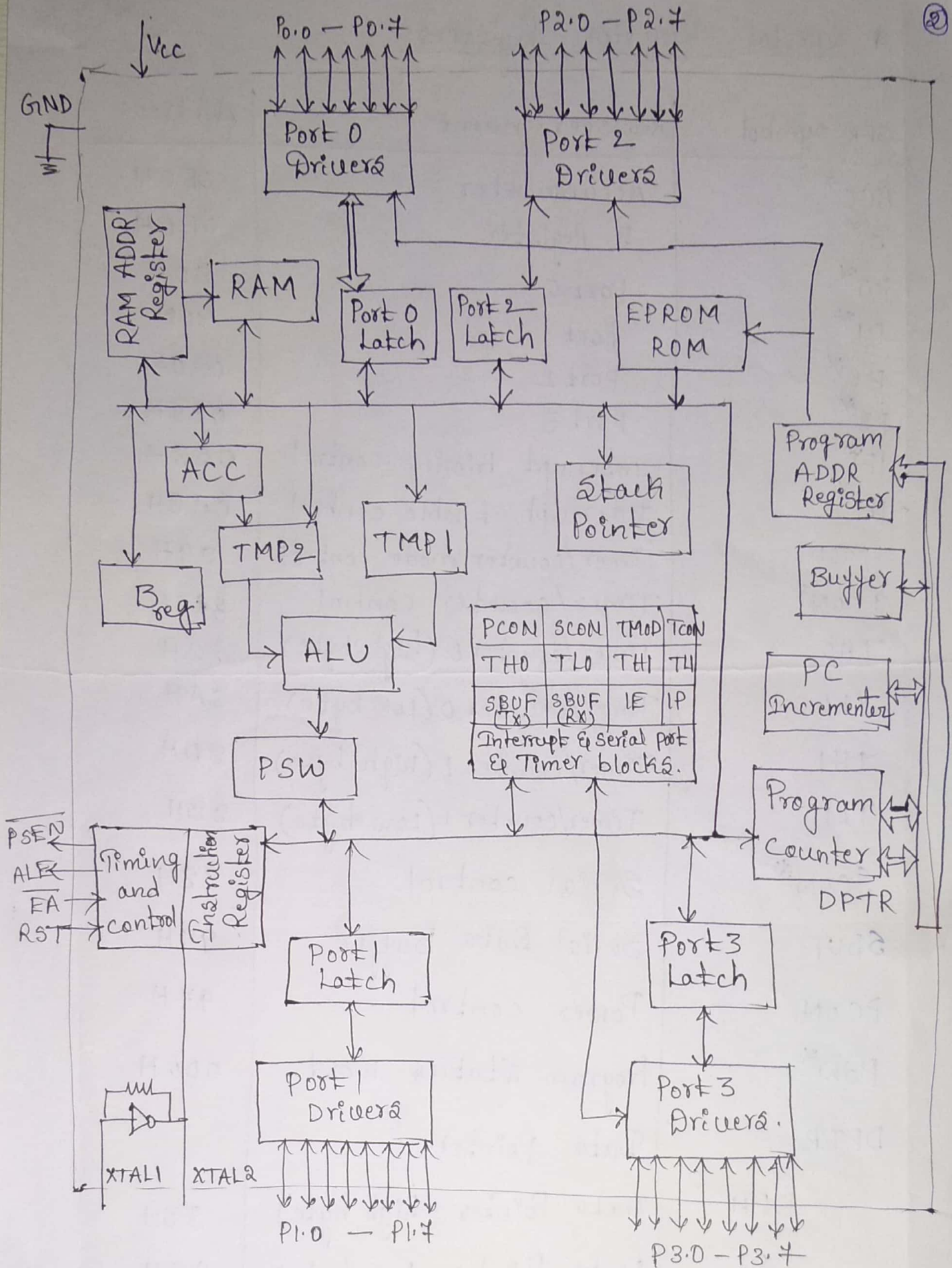


Fig: Block diagram of 8051 Microcontroller

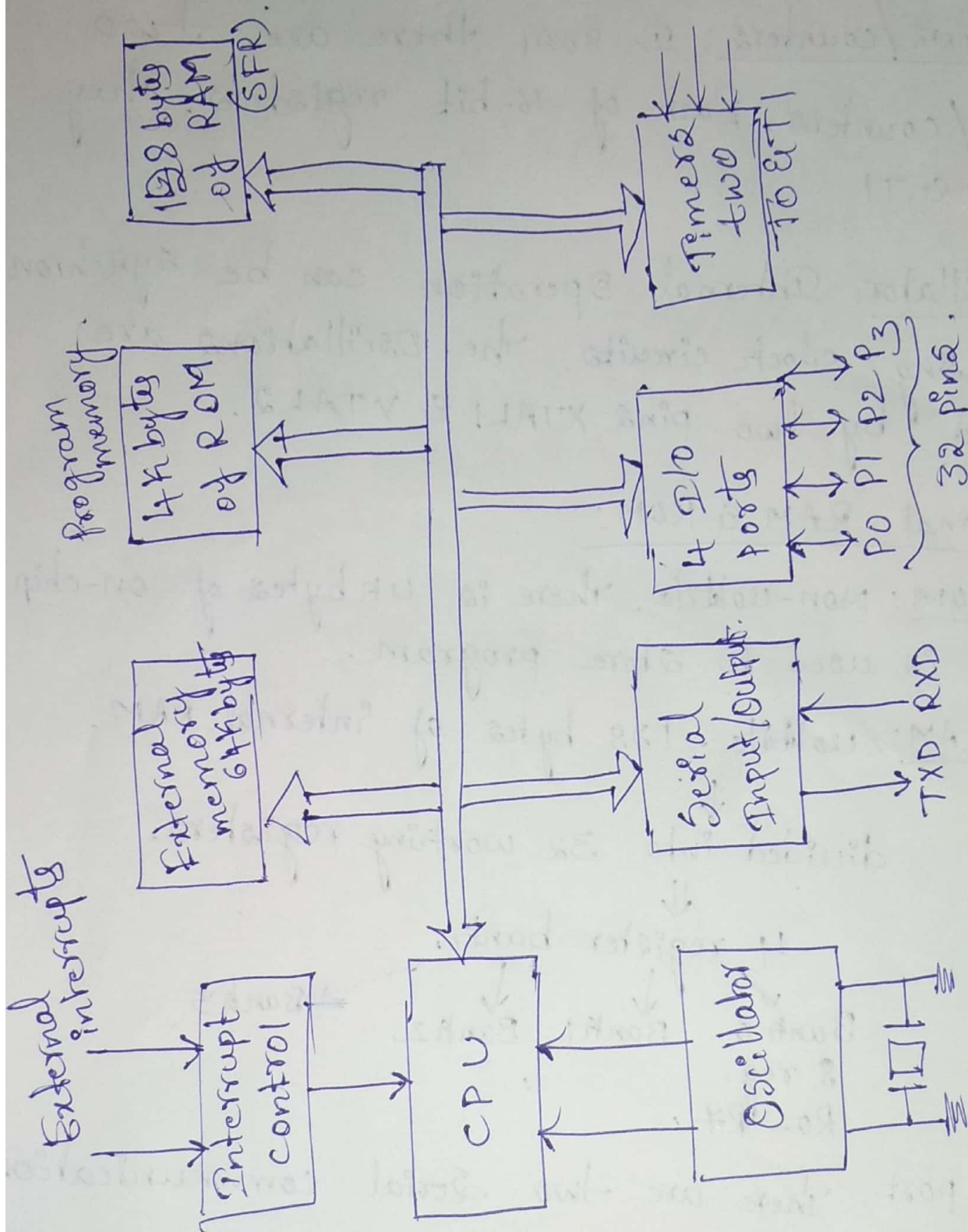
* Special function Registers:

SFR symbol	Register name	Address
ACC*	Accumulator	0E0H
B*	B-Register	0F0H
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
TMOD	Timer/counter mode control	89H
TCON*	Timer/counter control	88H
TH0	Timer/counter 0 (high byte)	8CH
TLO	Timer/counter 0 (low byte)	8AH
TH1	Timer/counter 1 (high byte)	8DH
TL1	Timer/counter 1 (low byte)	8BH
SCON*	Serial control	98H
SBUF	Serial Data Buffer	99H
PCON	Power control	97H
PSW*	Program Status Word	0D0H
DPTR	Data Pointer	
DPH	Data Pointer (high byte)	83H
DPL	Data Pointer (low byte)	82H
SP	Stack pointer	81H

* → bit & byte addressable.

no * → Byte

* Architecture of 8051:



① I/O ports: There are 4 I/O ports. Each has 8-bit bidirectional I/O lines, so total 32 I/O lines are available.

② Timers/counters: In 8051 there are two timers/counters. Each of 16-bit registers. They are T0 & T1.

③ Oscillator: Internal operation can be synchronized using clock circuits. The oscillations are provided by two pins XTAL1 & XTAL2.

④ Internal RAM & ROM:

ROM: Non-volatile. There is 4K bytes of on-chip ROM. It is used to store program.

RAM: volatile. 128 bytes of internal RAM.

↓
divided into 32 working registers.

↓
4 register banks.

Bank 0

Bank 1

Bank 2

Bank 3

8 reg.

R0 - R7.

⑤ Serial port: There are two serial communication

ports, RXD & TXD.

↓
SPI

↓
PISO.

Ports 0 to 3:

P0, P1, P2, P3 are the SFR's corresponding to four I/O ports respectively. Each of these ports is bit addressable as well as byte addressable.

Control registers:

TCON, TMOD, IE, IP, SCON, PCON & SBUF contain the control & status for interrupts, serial I/O & timer/counter.

Timer registers:

There are instructions for reading & writing these ~~instructions~~ registers byte wise.

Data pointer (DPTR):

DPTR is a 16-bit register consisting of two bytes.
→ The data pointer is used for addressing the off-chip data.

→ With 16-bit pointer DPTR, a maximum of 64K of off-chip data memory can be addressed.

Stack pointer (SP):

→ Stack pointer of 8051 is 8-bit wide. It is incremented during push & is decremented during POP operation. It may be available initialized anywhere in the available on-chip data RAM.

→ Accumulator (Acc): The accumulator acts as an ^{(2)^a} Operand register.

→ B register: Used to store one of the operands for multiply & divide instructions.

→ Program status word (PSW): Set of flags contains status information.

→ Stack pointer (SP): 8-bit wide register is incremented before the data is stored onto the stack using push or call instructions. It contains 8-bit stack top address.

→ Data pointer (DPTR): 16-bit register contains a higher byte (DPH) and lower byte (DPL) of a 16-bit external data RAM address.

→ Port 0 to 3 Latches & Drivers: Using the allotted addresses, user can communicate with these ports. These are identified as P0, P1, P2 & P3. Latches have been allotted addresses.

→ Serial data buffer (SBUF): contains two independent registers. Transmit buffer is a parallel-in-serial-out register. Receive buffer is a serial-in-parallel-out register.

→ Timer Registers: Two 16-bit register. T0 & T1. T10 represents lower byte of T0, while T1H represents higher byte of T0. Similarly for T1 also.

→ Control registers: special function registers status information is controlled by control reg.

→ Timing & control unit: derives all the necessary timing & control signals required for internal operation of ckt.

→ Oscillator: Generates basic timing clock signal for operation of ckt using crystal oscillator.

→ Instruction register: Decodes opcode of an instruction to be executed.

→ EPROM & Program Address register: Provide on-chip EPROM & a mechanism to internally address it.

→ RAM & RAM address register: Provide internal 128 bytes of RAM & a mechanism to address it.

→ ALU: Performs 8-bit arithmetic & logical operations held by the temporary reg. TMP1 & TMP2.

* Signal Description of 8051:

→ 8051 is a 40-pin plastic & ceramic DIP.

→ V_{CC}: +5V supply.

→ V_{SS}: Return pin for supply.

→ RESET: Resets 8051, when it goes high for two or more machine cycles. After reset, clock must be running.

→ ALE/PROG: ALE signal is valid only for external memory accesses. Program pulse i/p during on-chip

EPROM Programming.

$\overline{EA}/V_{PP}$: External access enable pin, indicates 8051 can address external program memory. It receives 21 volts for programming on-chip EPROM.

PSEN: Program store enable reads the external program memory.

Port 0 (P0.0-P0.7): 8-bit bidirectional I/O port. Acts as multiplexed address/data lines.

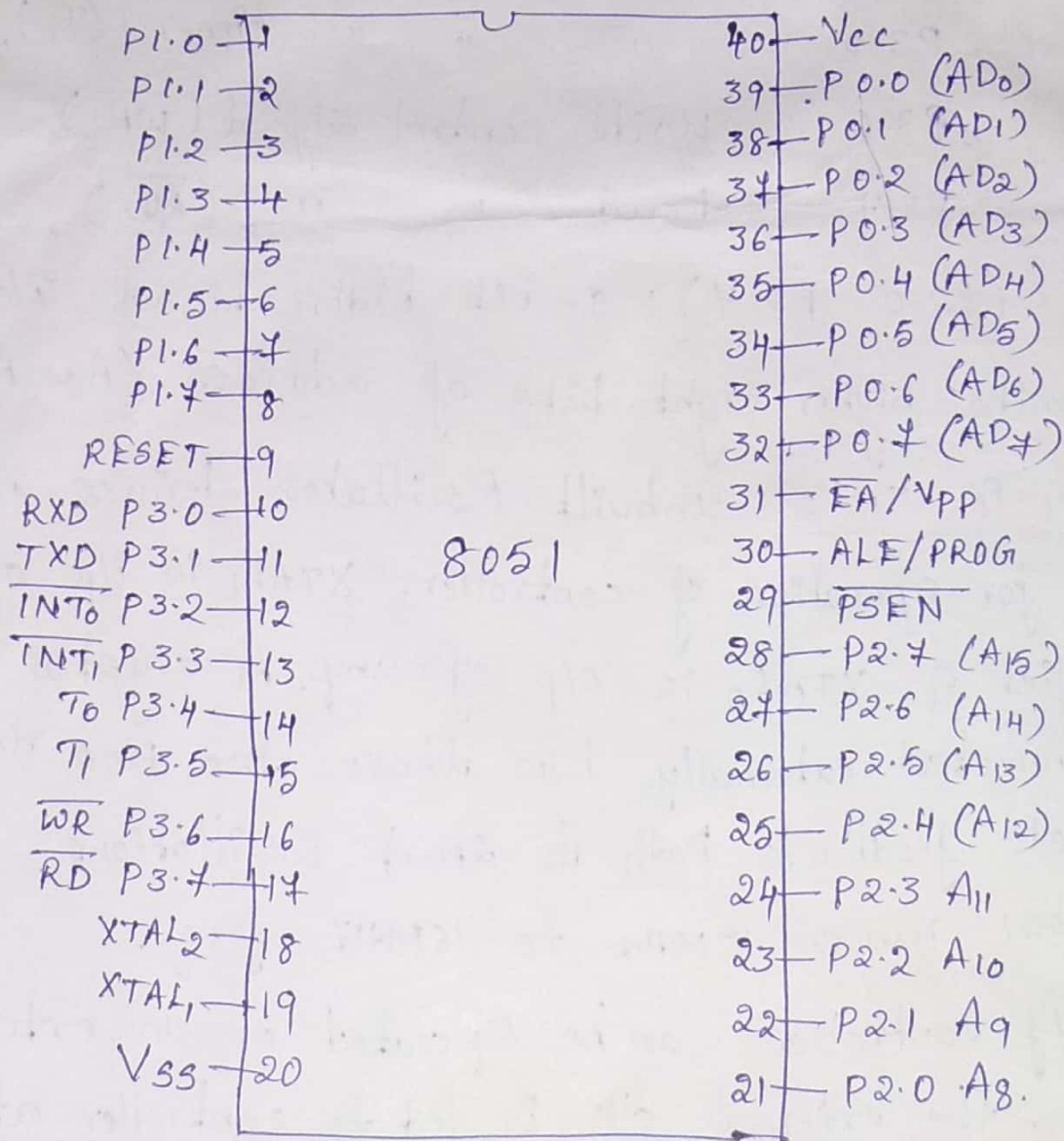


Fig: 8051 Pin configuration.

Port 1 (P1.0 - P1.7): 8-bit bidirectional port. It emits higher eight bits of address ($A_8 - A_{15}$).
Port 3 (P3.0 - P3.7): 8-bit bidirectional I/O port. It also serve alternative functions.

P3.0	Receive serial i/p data (RXD).
P3.1	Transmit " o/p " (TXD).
P3.2	External interrupt pin 0. $\overline{INT_0}$
P3.3	" " " " pin 1. $\overline{INT_1}$
P3.4	External i/p to timer 0 (T_0).
P3.5	" " " " timer 1 (T_1).
P3.6	Write control signal ($\overline{WR}$).
P3.7	Read " " ($\overline{RD}$).

Port 2 (P2.0 - P2.7): 8-bit bidirectional I/O port. It emits higher eight bits of address ($A_8 - A_{15}$).

XTAL₁ & XTAL₂: Inbuilt Oscillator derives clock freq. for operation of controller. XTAL₁ is i/p of amplifier & XTAL₂ is o/p of amp. A crystal is to be connected externally b/w these two pins to complete feedback path to start oscillations.

→ 8051 run on 12MHz to 16MHz freq.

→ If controller can be operated on an external clock, the external clk is fed to controller at pin XTAL₂ & XTAL₁ pin should be grounded.

* Memory Organisation of 8051:

- The total memory of an 8051 is logically divided into program memory & data memory.
- Program memory is invariably implemented using EPROM.
- Data memory may be read from or written to & thus it is implemented using RAM.
- Further, the program memory & data memory both may be categorized as on-chip (internal) and external memory.
- 8051 can address 4Kbytes on-chip (internal) program memory (EPROM), whose map starts from 0000H to 0FFFH.
- It can address 64Kbytes of external program memory, whose address map is from 0000H to FFFFH.

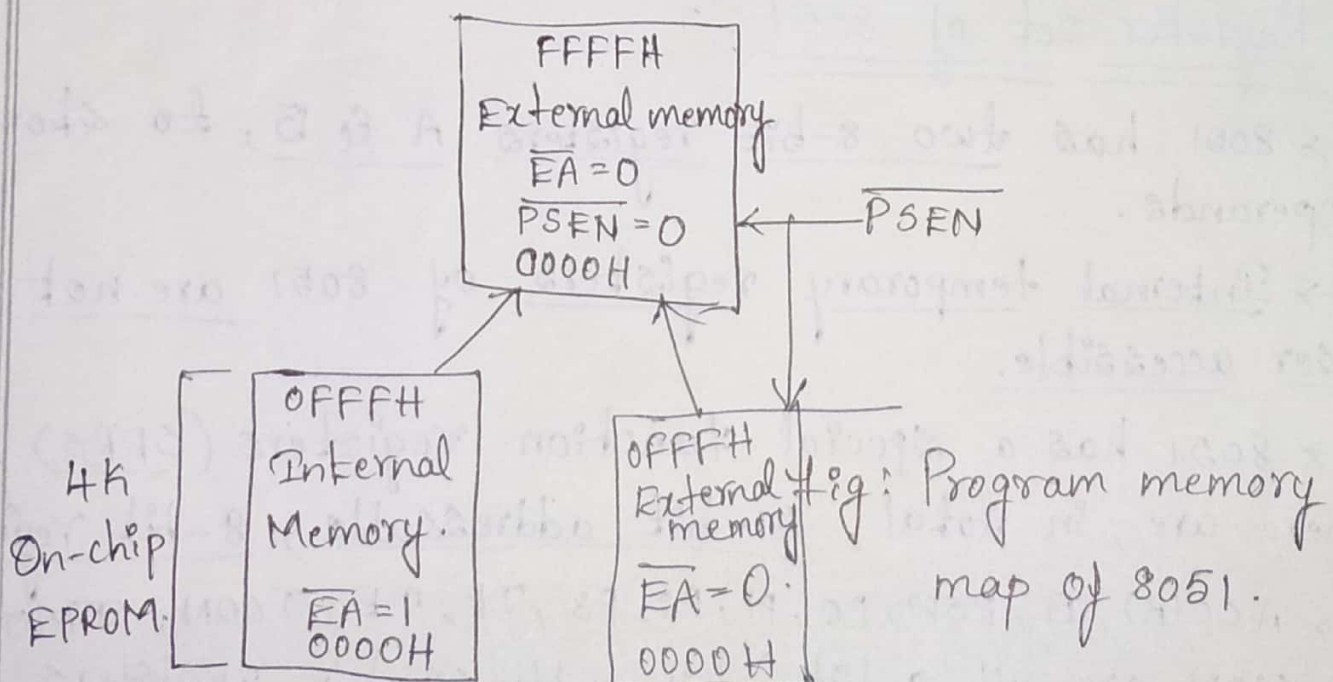


Fig: Program memory map of 8051.

→ 8051 supports 64Kbytes of external data memory whose map starts at 0000H to FFFFH.

→ Internal data memory of 8051 (RAM) is of 128 bytes. whose address map starts from 00 & ends at 7FH.

→ RAM can be addressed by using direct or indirect mode of addressing.

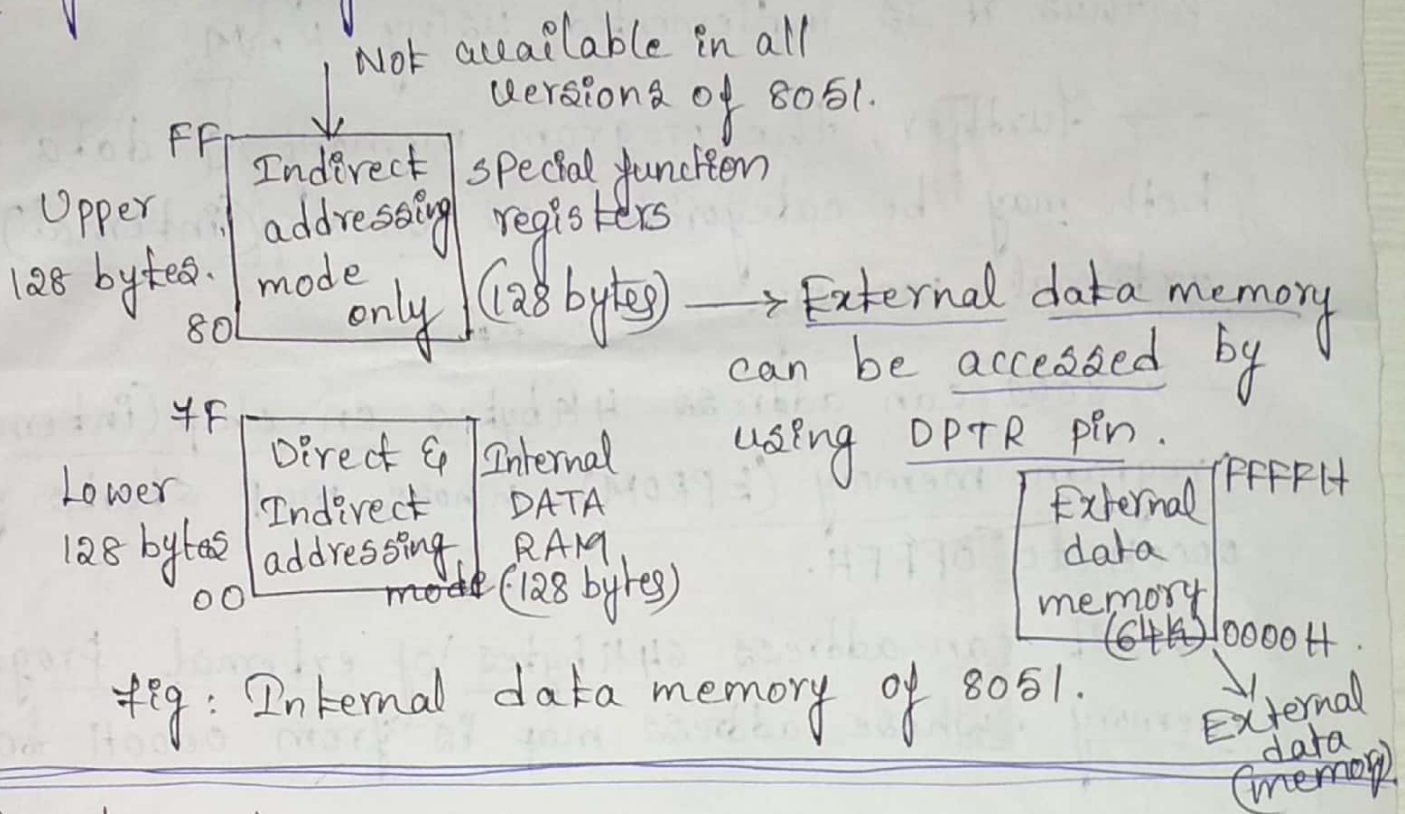


Fig: Internal data memory of 8051.

* Register set of 8051:

→ 8051 has two 8-bit registers A & B, to store operands.

→ Internal temporary registers of 8051 are not user accessible.

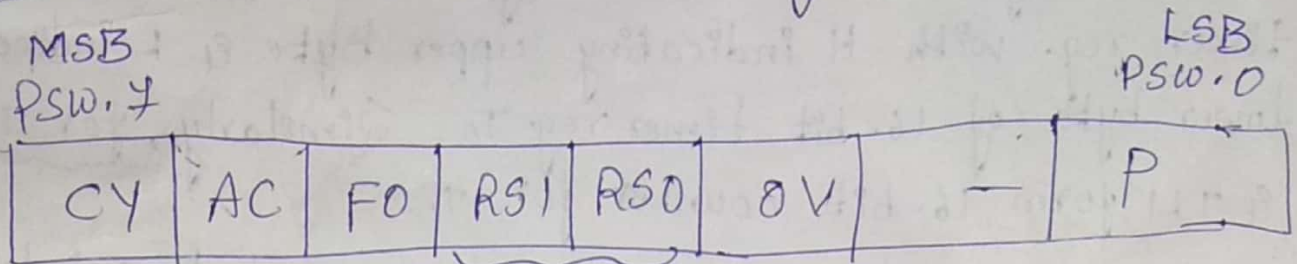
→ 8051 has a special function registers (SFRs). There are, in total, 21-bit addressable, 8-bit registers.

→ ACC(A), B, PSW, P0, P1, P2, P3, IP, IE, TCON, ~~TMOD~~ & SCON are all 8-bit, bit addressable registers.

- The remaining registers, SP, DPH, DPL, TMOD, TH0, TL0, TH1, TL1, SBUF & PCON registers are to be addressed as bytes, i.e. they are not bit-addressable.
- DPH & DPL are higher & lower bytes of 16-bit reg. DPTR.
- The registers TH0 & TL0 form a 16-bit counter/timer reg. with H indicating upper byte & L indicating lower byte of 16-bit timer reg. T₀. Similarly, for TH1 & TL1 form 16-bit counter for T₁.
- PSW is a flag reg. & contains status information.
- Register IP can be programmed to control the interrupt priority.
- Register IE, enable or disable interrupts.
- TCON is called timer/counter control register.
- TMOD: Modes of operation of timers/counters. In various modes timers can be configured.
- SCON: Control the operation of serial port.
- SBUF: Serial data buffer for transmit & receive operations.
- PCON: power control reg.
 - Power down mode
 - idle mode.
- In power down mode, on-chip osci. is stopped. All the functions of controller maintaining contents of RAM.

→ In Idle mode, Osc. continues to run, but the clk to the cpu is disabled. Interrupt, Serial port & timer blocks are active.

* PSW (Program status word) & flag register of 8051: → It is an 8-bit register.



CY - carry flag

AC - Auxiliary carry.

FO - General purpose flag reg

RSI - Register bank selector bit 1

RSO - " " " " 0

OV - Overflow flag.

P - Parity flag.

→ available to user. the user can define its role.

→ PSW is a bit addressable register.

→ Register Bank select Bits: → Each register bank consists R0-R7.

RSI	RSO	Register bank selected	Address range in on-chip RAM
0	0	Bank 0	00-07H
0	1	Bank 1	08-0FH
1	0	Bank 2	10-17H
1	1	Bank 3	18-1FH

* Addressing Modes of 8051:

→ Addressing modes define the way in which the operands are accessed by the instruction.

→ There are five addressing modes supported by 8051.

① Register addressing

② Direct addressing

③ Register Indirect addressing

④ Immediate addressing

⑤ Base Register plus Index Register addressing.

1. Register addressing:

→ In this mode of addressing, the operands are specified using 8-bit address field.

→ In this addressing, registers R0 through R7 from the selected register bank, accumulator, B-register are used.

→ The least significant bits of opcode indicate which register is to be used.

Opcode part (5 bits)					Register (3 bits)		
1 USB	1	1	0	1	1	1	1 LSB

Ex: MOV A, R7

~~MOV R0, 89H~~

ADD A, R7 → ~~Address of TMOD.~~

→ LSB three bits refer to register R0 through R7 used.

2. Direct addressing:

→ In this mode, the direct address of the operand is specified in the instruction itself.

→ It uses the lower 128 bytes of internal RAM & the special function registers.

Ex: MOV R0, 89H.

↳ address of TMOD.

MOV A, 54H.

↳ address of memory location.

MOV A, SBUF.

↳ It has direct address 99H.

opcode Byte	8-bit Direct address
-------------	----------------------

3. Register Indirect addressing:

→ In this mode, the 8-bit address of an operand is stored in a register & the register, instead of 8-bit address, is specified in the instruction.

→ The Registers R0 or R1 of selected bank of registers or stack pointer can be used as address registers for storing the 8-bit addresses.

→ Note that the SFRs are not addressed by this mode.

Ex: ADD A, @R0.

4. Immediate addressing:

→ In this mode, an immediate data, i.e. a constant is specified in the instruction, after the opcode byte.

Ex: ADD A, #100.

Immediate data 100 (decimal) is added to the contents of accumulator.

MOV A, #45H.

It will store the constant or immediate data 45H in the accumulator.

→ For specifying a hex number, it should be followed by H.

→ The symbol # Preceding the constant, indicates the immediate data type.

5. Base Register plus Indexed Addressing:

→ Only program memory can be accessed using this addressing mode.

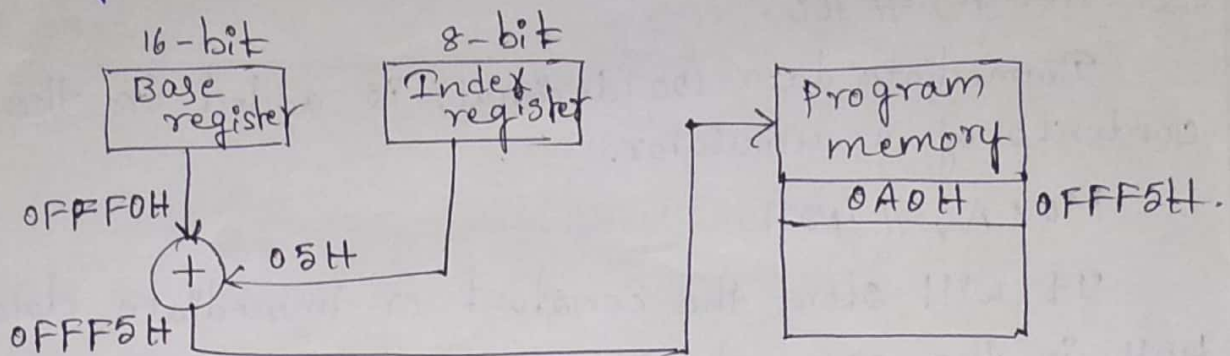
→ Whose address is calculated as the sum of a base register (DPTR or PC) and index register, accumulator.

Ex: MOV A, @A + DPTR.

↳ whose address is calculated by adding the original 8-bit contents of accumulator & the 16-bit contents of DPTR.

JMP @ A+DPTR.

↳ The contents of accumulator are added with one of the specified 16-bit registers (DPTR) to form the jump destination address.



* Instruction Set of 8051:

- There are 111 instructions supported by 8051.
- The instructions can further be classified as single byte, two-byte & three-byte instructions.
- There are 45 single-byte instructions, 45 two-byte instructions & 17 three-byte instructions.

① Data Transfer Instructions:

- There are MOV, MOVX, MOVC, PUSH, POP, and exchange XCHG, XCH instructions.

- Data transfer instructions do not affect any of the flags.

EX: MOV A, Rn
MOV A, @Ri → move indirect RAM to accumulator.
MOV A, #data
MOV Rn, A
MOV Rn, #data

MOV @R_i, A
 MOV @R_i, #data.
 MOVX A, @R_i
 MOVX A, @DPTR
 MOVX @R_i, A
 XCH A, R_n
 XCH A, @R_i
 XCHD A, @R_i

→ Immediate
 @ → Indirect

⑧

└→ Low order digit.

② Arithmetic Instructions:

→ 8-bit arithmetic unsigned operations are supported by 8051.

→ To carry out both signed & unsigned addition & subtraction by using the OV flag.

Ex: ADD A, R_n
 ADD A, @R_i
 ADD A, #data
 ADDC A, R_n
 ADDC A, @R_i
 ADDC A, #data.

Ex: SUBB A, R_n
 SUBB A, @R_i
 SUBB A, #data
 INC A
 INC R_n
 INC @R_i.

DEC A
 DEC R_n
 DEC @R_i
 MUL AB
 DIV AB.

DA A.

③ Logical Instructions:

→ Bitwise logical AND, OR, EX-OR operations are possible in 8051.

Ex: ANL A, R_n → AND operation.

ANL A, @R_i

ANL A, #data

ORL A, R_n → OR operation.

ORL A, @R_i

ORL A, #data.

XRL A, @R_i → EX-OR operation.

CLR A → clear accumulator.

CPL A → Complement accumulator.

RL A → Rotate left

RR A

RLC A → Rotate left through carry.

RRC A

④ Boolean Variable Manipulation Instructions.

Ex: CLR C

SETB C

CPL C

CPL bit

CLR bit

MOV C, bit

JC rel → Jump if carry is set.

JNC rel

JB bit, rel → Jump if direct bit is set.

JNB bit, rel.

ANL C, bit → AND direct bit to carry.

ORL C, bit

ORL C, /bit → OR compliment direct bit to carry.

⑤ Program Branching Instructions.

→ Both conditional branch & unconditional branch operations are possible in 8051.

Ex: ACALL addr11 — Absolute subroutine call

LCALL addr16 — Long subroutine call.

RET — Return from subroutine

RETI — Return from interrupt.

AJMP addr11 — Absolute Jump.

LJMP addr16 — Long Jump.

SJMP — Short Jump.

JZ — Jump if accumulator is Zero.

JNZ — Jump if accumulator is not Zero.

NOP — No operation.

CJNE A, #data — Compare Immediate to accumulator
and Jump if not equal.

DJNZ Rn, rel — Decrement register and jump if
not Zero.

8051 Real Time Control.

①

* Interrupt:

→ upon receiving an interrupt signal, the uc interrupts whatever it is doing & serves the device.

→ For every interrupt there is a fixed memory location that holds the address of its ISR.

→ The group of memory locations set aside to hold the addresses of ISRs is called the interrupt vector table.

* Interrupts in 8051:

→ There are five interrupts in 8051.

→ Two interrupts are set aside for timers one for timer 0 & timer 1.

→ Two interrupts are set aside for hardware external interrupts. INTO & INT1.

→ Serial communication has a single interrupt that belongs to both receive & transmit.

Interrupt	ROM Location.		
External hardware interrupt 0 (EX0)	0003H.		Highest
Timer 0 interrupt (TFO)	000BH	P r i o r i t y	
External hardware interrupt 1 (EX1)	0013H		
Timer 1 interrupt (TF1)	001BH		
Serial com interrupt (RI & TI)	0023H		Lowest

* Interrupt enable register (IE):

→ The IE register is responsible for enabling & disabling the interrupts.

→ Bit D7 of the IE register (EA) must set to high to allow the rest of register to take effect.

D7							D0
EA	—	ET2	ES	ET1	EX1	ET0	EX0

EA IE.7 Enable all, If EA=1
Disable all, If EA=0.

— IE.6 Not implemented.

ET2 IE.5 Enables or disables Timer 2 interrupt.

ES IE.4 Enables or disables the serial port interrupt.

ET1 IE.3 Enables or disables Timer 1.

EX1 IE.2 Enables or disables external interrupt 1.

ET0 IE.1 Enables or disables Timer 0 interrupt.

EX0 IE.0 Enables or disables external interrupt 0.

→ If EA=1, interrupts are enabled & 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.

* Program:

: Initialize the external interrupt 0.

MOV IE, #1000 0100 B. → enable external interrupt
'1' (bit EA=1 & bit EX1=1)

* Program:

: Show the instructions to

(a) enable the serial interrupt, Timer 0 interrupt, & external hardware interrupt 1 (EX1).

(b) Disable timer 0 interrupt.

(c) Show how to disable all the interrupts with a single instruction.

Sol. (a) MOV IE, #1001 0110 B: enable serial, timer 0 & EX1.

(b) CLR IE.1

(c) CLR IE.7.

* Interrupt Priority Register (IP):

→ When more than one interrupts are enabled, User can program the interrupt priority levels by setting or clearing the bits in SFR called interrupt priority (IP) register.

→ IP register is also bit addressable.

→ If the bit is set, that particular interrupt will have high priority.

→ If the request of interrupts of two different priority levels occur simultaneously, naturally the interrupt having the high priority will be served.

→ If the same priority level interrupts request simultaneously, then within each priority level there is a polling structure due to the inherent priority in the order.

IP.7	IP.6	IP.5	IP.4	IP.3	IP.2	IP.1	IP.0
X	X	PT2	PS	PT1	PX1	PT0	PX0

IP.5 PT2 Timer 2 Priority in 8032/8052

IP.4 PS Serial interrupt priority

IP.3 PT1 Timer 1 interrupt "

IP.2 PX1 External '1' interrupt "

IP.1 PT0 Timer 0 interrupt. "

IP.0 PX0 External '0' interrupt "

* Program:

: Assigning interrupt priorities.

MOV IE, #1000 1100H : Enable EX1 & ET1

SETB PT1 : Timer 1 interrupt has high priority.

* MOV IE, #1000 1100H : Enable EX1 & ET1

SETB PT1 : Timer 1 interrupt has high priority.

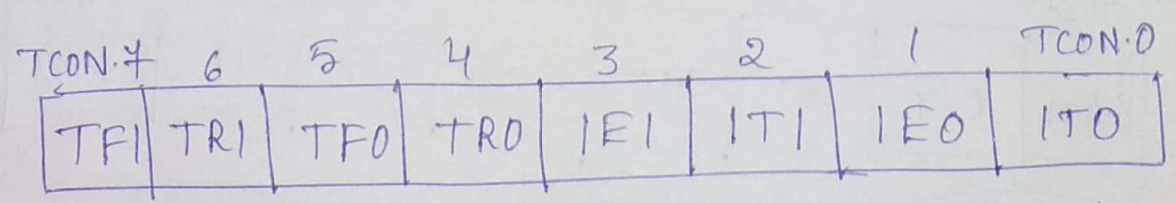
SETB PX1 : External interrupt also has high priority.

→ In this the external interrupt will be served as per the priority order. because two interrupts are occurred simultaneously.

* Timers & Counters:

- On chip timing/counting facility has proved the capabilities of the μ c's for implementing real time applications.
- These include pulse counting, frequency measurement, PWM measurement, baud rate generation etc.
- A timer provides a reference time delay or a clock. At 12MHz, the clocking period will be equal to 1 μ s.
- A counter, count inputs at regular intervals.
- A counter of 8051 is incremented in response to a transition from 1 to 0 at its corresponding pin. (T0 or T1).
- Thus, the counter o/p will be a count or a number representing the occurrence of such '1' to '0' transitions.
- When a timer or counter overflows from FFFFH to 0000H, it sets a flag & generates an interrupt.

* TCON (Timer control) register:



TF1 — Timer 1 overflow flag, set when timer/counter overflows.

TR1 — Timer 1 run control bit.

TFO — Timer 0 overflow flag, set when timer/counter 0 overflows.

TRO — Timer 0 run control bit.

IE1 — Interrupt '1' flag, when '1' it is edge triggered (1 to 0 transition), when '0' level triggered (level 0).

IT1 — Timer interrupt '1'.

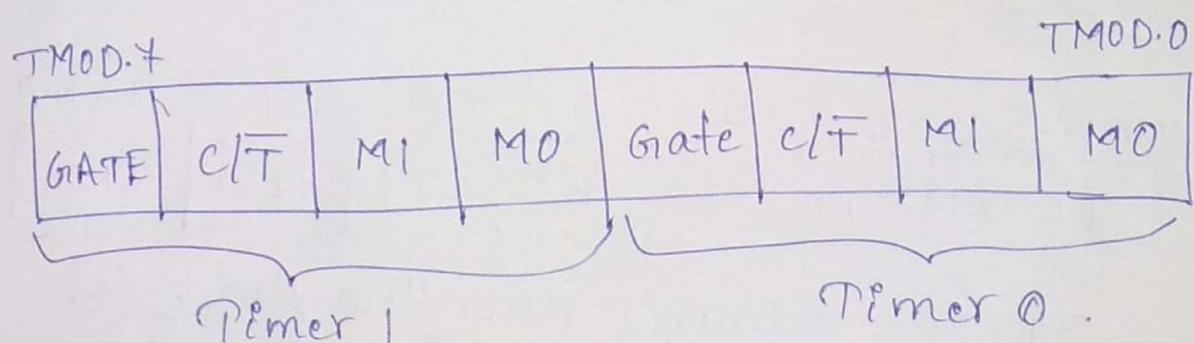
IE0 — Interrupt '0' flag. (defines whether INTO is edge triggered or level activated).

ITO — Timer 0 interrupt. ITO=0, low level trigger, ITO=1, edge trigger (falling edge).

→ The control of timer/counter operation is done through the SFR, called the timer control register (TCON).

→ Timer/counter are enabled by a TR bit.

* TMOD (Timer mode register):



GATE — If '1', timer is enabled or run when the gate pin (pin INT1) is activate. (4)
 Gate = 0 means timer runs without gate input & runs as timer TR1 or TR0 only.

C/T — 0 = timer, 1 = counter.

M1 M0	M1	M0	
Timer mode	0	0	Mode 0
is determined	0	1	Mode 1
by these bits	1	0	Mode 2
	1	1	Mode 3.

* Program to initialize timer 1 in mode 0.

```
MOV TMOD, # 1000 0000B
SETB TR1
CLR TR1
SJMP $ —> Infinite loop.
```

* Timer 1 in mode 1:

```
MOV TMOD, # 0001 0000B
SETB TR1
SJMP $.
```

* Timer 0 in mode 2:

```
MOV TMOD, # 0000 0010B
MOV TH0, # 33H
MOV TL0, # 33H
SETB TR0.
SJMP $.
```

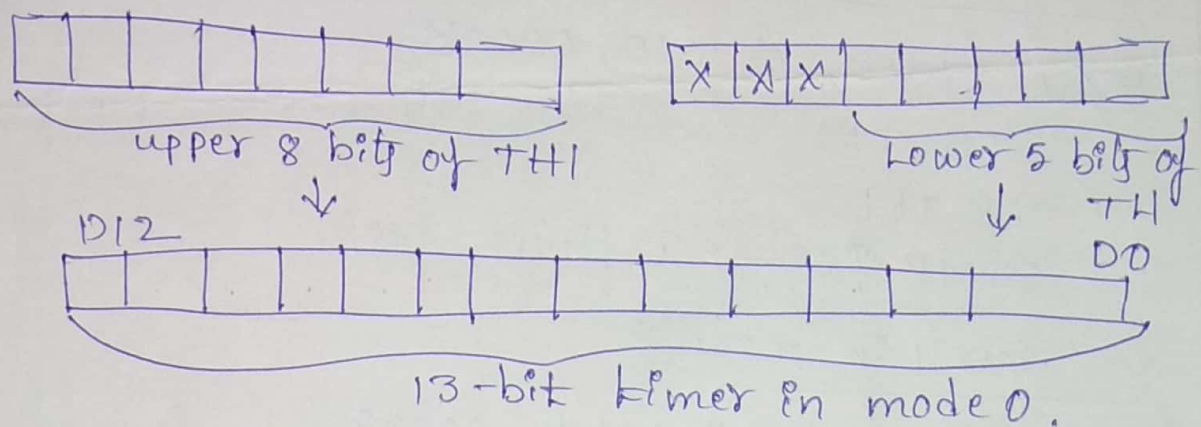

* Mode 0 of TMOD:

→ In mode 0, the timer is 13-bit wide. This mode is same for timer 0 as well as timer 1.

→ When the count overflows, it sets TFI or TFO.

→ To start timer 0, TR0 bit in TCON is required to be set.

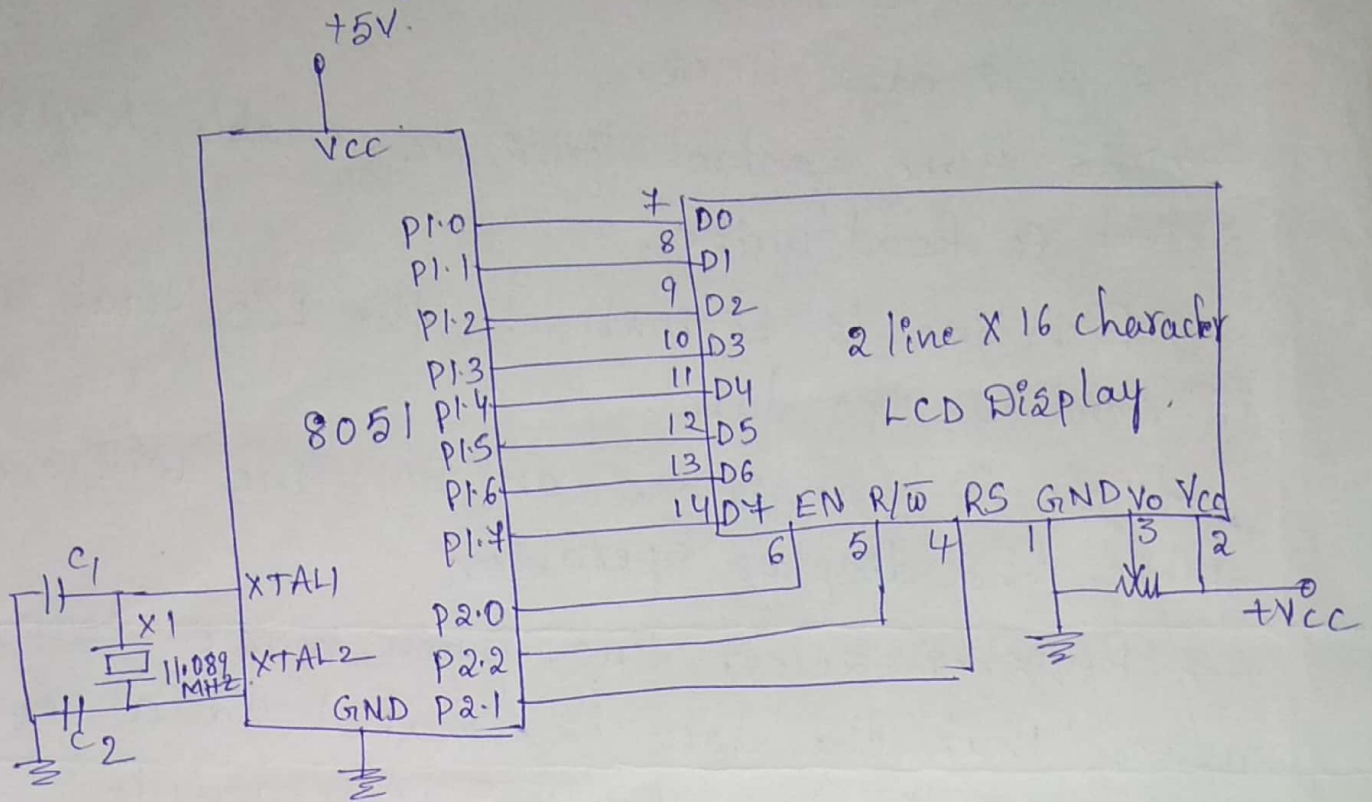
→ Using the upper byte TH0 (TH1) & the lower 5 bits of TL0 (TL1) form the 13 bits.



UNIT-III

I/O and Memory Interface.

* LCD Interfacing with 8051:



① LCD Pin description:

1. GND — Ground
2. Vcc — +5V power supply
3. VEE — Contrast control power supply.
4. RS — 0 = select command register.
1 = select data register.
5. R/W — 0 = write; 1 = Read.
6. EN — Enable.
- 7-14 D0-D7 — 8-bit data bus.

② Commands or instruction codes of LCDs

01H — clear display screen.

→ LCD ^{instructions} allow selecting the display functions, cursor positioning, blinking, selecting character font, number of lines etc.

* → In LCD module, there are 3 control lines & 8-data lines.

→ The three control lines are enable, Register select & Read/write.

→ EN line is to instruct the LCD that we are sending the data.

→ 1 to 0 transitions on EN line will complete the display operation.

→ Register select line, when made low, means that the data is treated as a command or instruction to the LCD module.

→ when RS line is high, the data is a text data to be displayed on the LCD.

→ the read/write line RW, when made low, the information on data bus is written to LCD & when RW line is high, it means that the data is being read from the LCD.

→ There is only one read command, namely Get LCD status.

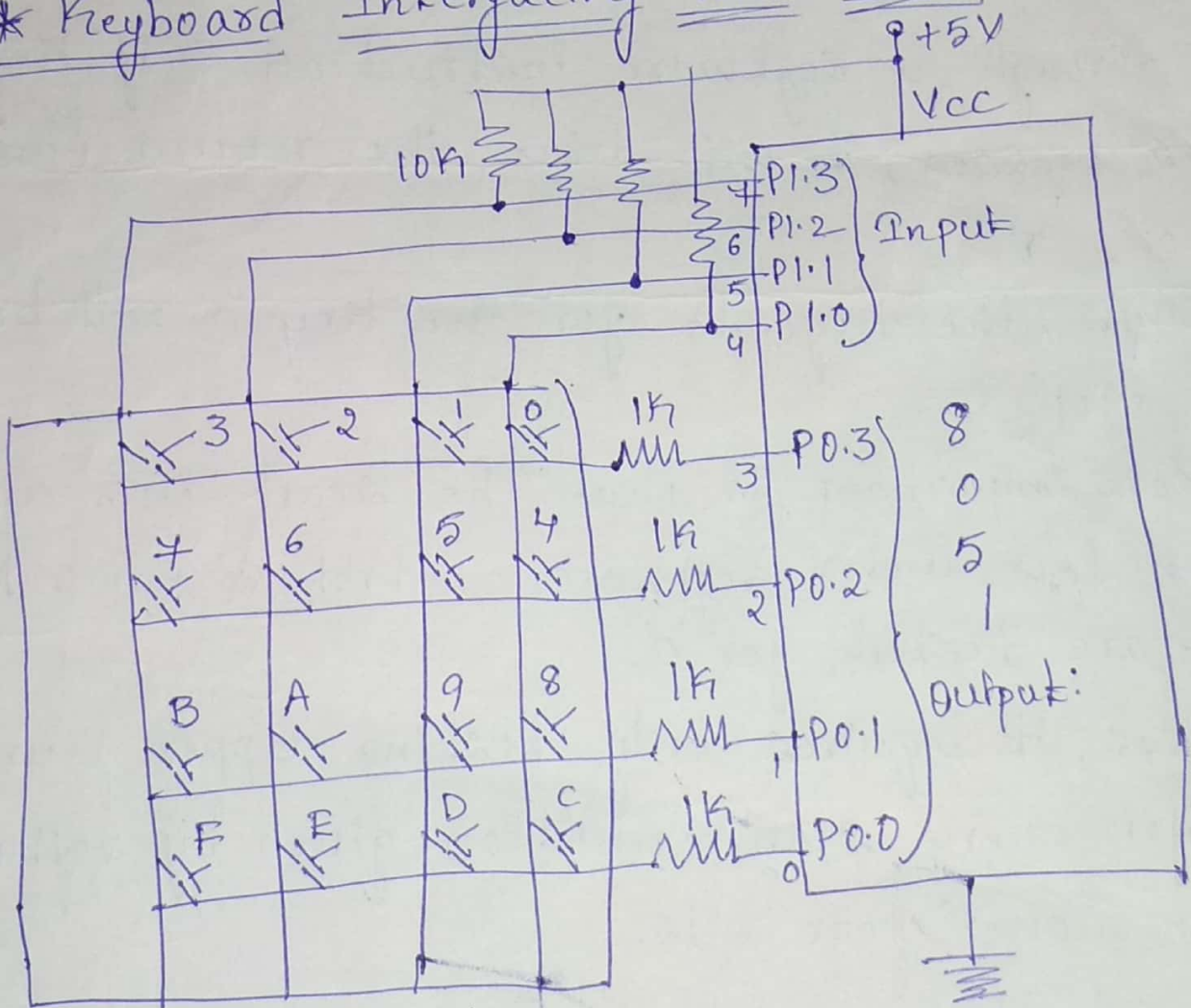
→ The EN line must go from high to low for LCD to execute the instruction.

→ Get LCD status command will make bit 4 of data bus high if it is busy in executing the command & will make it low if the LCD is not busy.

→ RS line must be low, while sending any of the commands.

→ For writing the text, set line EN & RS. EN line must go from high to low for the execution of LCD operation.

* Keyboard Interfacing with 8051:



Hex Key Pad.

0	4	8	C
1	5	9	D
2	6	A	E
3	7	B	F

→ Port '1' pins 4, 5, 6, 7 are pulled high through 10K resistors and are configured as inputs.

→ 1K resistors are connected in each of the rows to pins 3, 2, 1, 0 to avoid any short circuit between these pins in case two keys are pressed simultaneously.

→ Pins 3, 2, 1, 0 are configured as outputs & are used as the scan lines.

→ Pins 7, 6, 5, 4 are the return lines.

→ For example, if a scan line '3' is made low through a software instruction & if key '0' is pressed, it will drive the return line '4' low.

→ Thus, the keycode for the key '0' will be "0111 0111".

→ More important is that the scan lines are driven low under software control & the return lines are tested for '0'.

→ Thus, it requires only testing upper four bits for every scan condition given by software driven lower four bits.

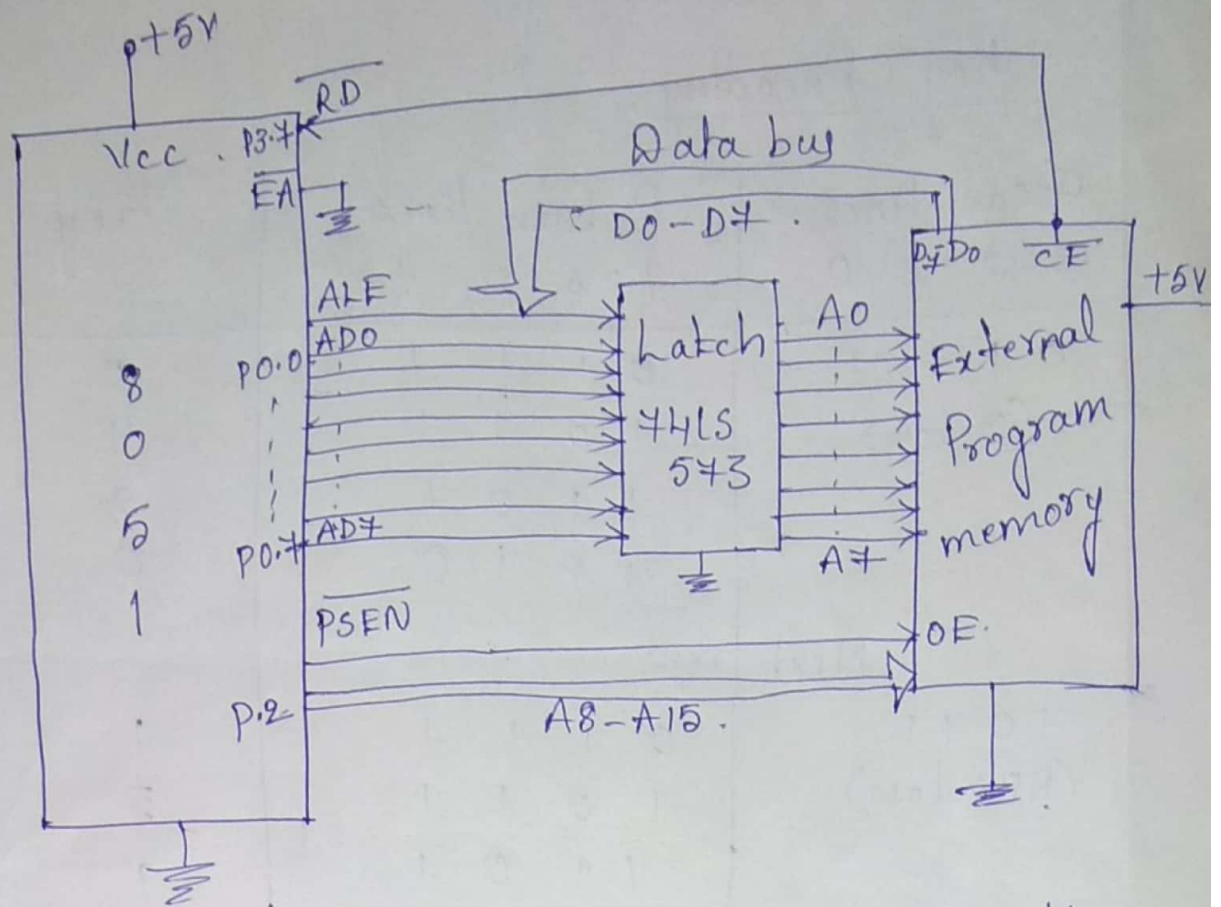
→ For testing any key closure in spite of which key is pressed all the scan lines may be driven low & test for any return line becoming low.

→ Initially, the program register must be loaded with B"1111 0000".

* Key Encoding:

Scan lines 3:2:1:0	Return lines 4:6:5:4	Key
0 1 1 1 (3 low)	0 1 1 1 1 0 1 1 1 1 0 1 1 1 1 0	0 4 8 C
Next row		
1 0 1 1 (RB2 low)	0 1 1 1 1 0 1 1 1 1 0 1 1 1 1 0	1 5 9 D
Next row		
1 1 0 1 (1 low)	0 1 1 1 1 0 1 1 1 1 0 1 1 1 1 0	2 6 A E
Next row		
1 1 1 0 (0 low)	0 1 1 1 1 0 1 1 1 1 0 1 1 1 1 0	3 7 B F

* Interfacing External ROM with 8051:



- In 8051 ports 0 & 2 provides 16-bit address to access the external memory.
- P0 is multiplexed address/data bus.
- By using ALE pin
 - ALE = 0 → Data path (Access data from P0)
 - ALE = 1 → address path. (Access address from P0)
- o/p signal of 8051 must be connected to (PSEN) the OE pin of ROM containing the program code.
- EA pin (active low):
 - 1 → pgm is on on-chip ROM
 - 0 → pgm to be access from external pgm memory.
- PSEN:
 - 0 → can access external memory.
 - 1 → Cannot access external memory.

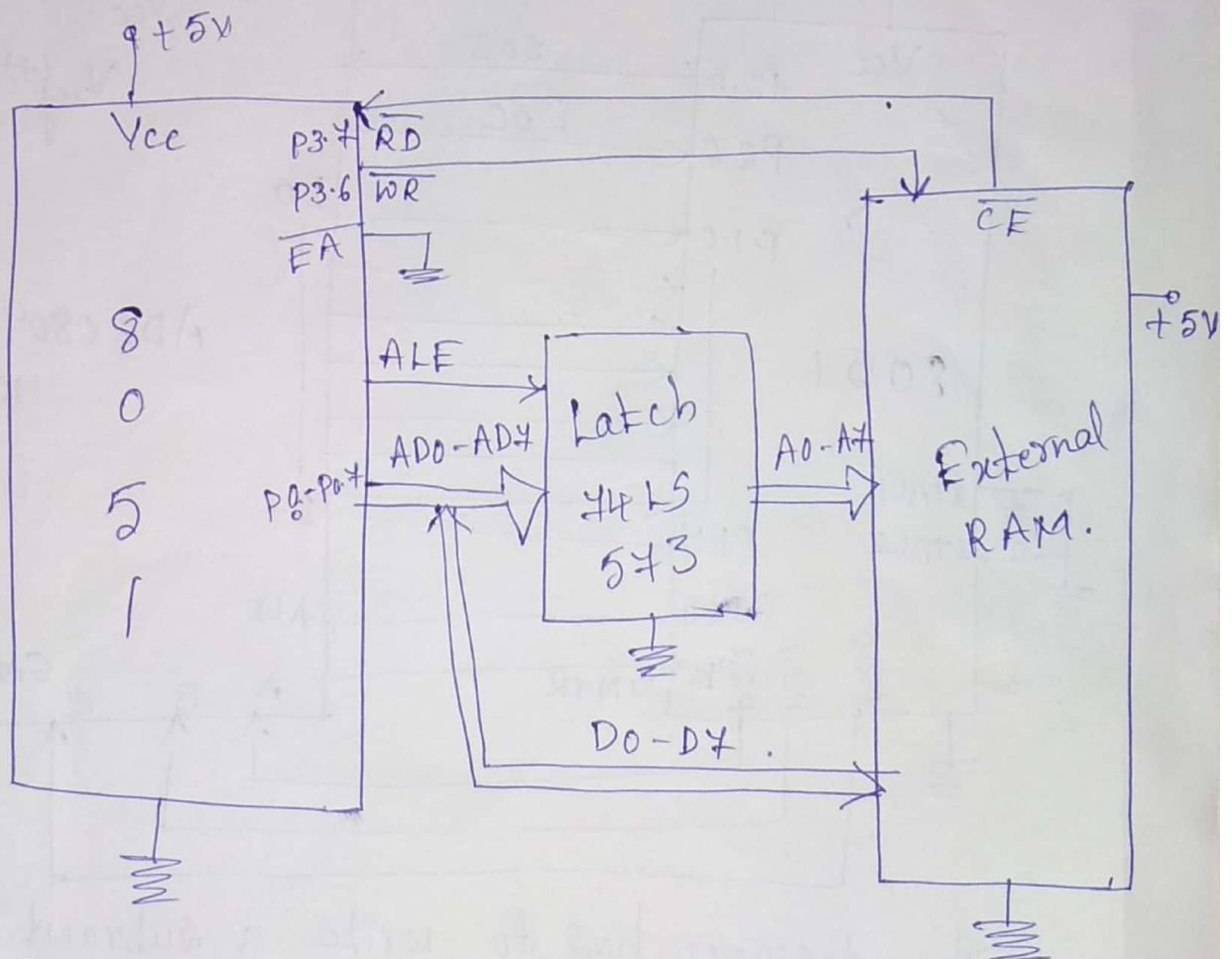
→ Latch is a 8-bit latch used for address & data multiplexing. (4)

* → External program memory access is possible in 8051 if the $\overline{EA}$ pin is low.

→ It is necessary to latch the lower address byte when the external program byte is read on the data bus.

→ The lower order address byte is latched, as the ALE pulse goes low.

* Interfacing of External RAM to 8051:



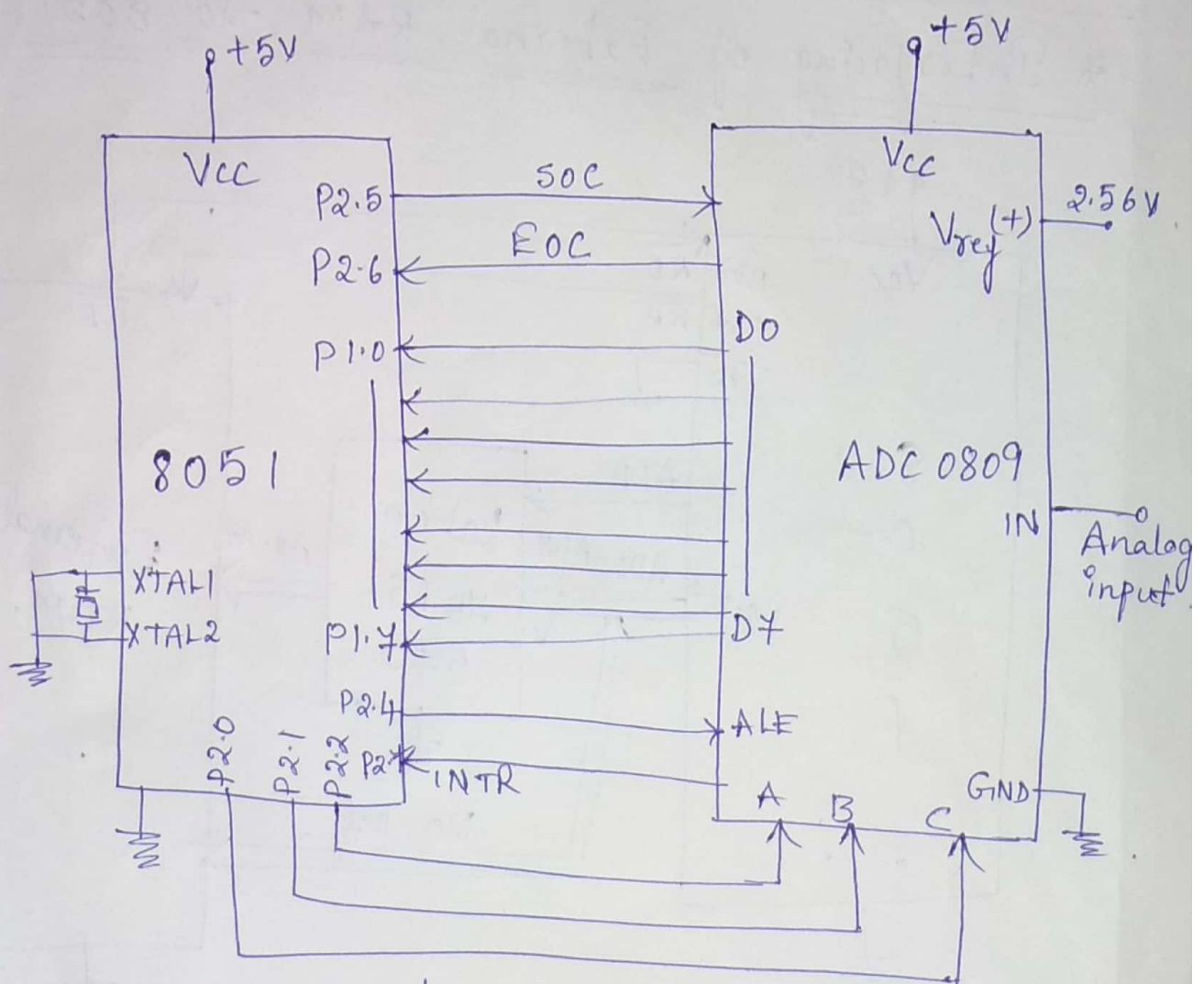
→ External data memory up to 64K can be accessed using a 16-bit data pointer (DPTR).

→ If the access is to data memory, then the signals $\overline{RD}$ & $\overline{WR}$ are generated by 8051.

→ During read cycle, $\overline{RD}$ signal goes low & a data byte from the external memory location is loaded onto the data bus.

→ $\overline{WR}$ signal is generated to write this byte into the external data memory.

* Interfacing ADC to 8051:



→ The designer has to write a subroutine for the control of ADC operation.

→ The analog to digital converter is treated (5) as an input device by the microcontroller, that sends an initialising signal to ADC to start the analog to digital data conversion process.

→ The soc signal is a pulse of a specific duration.

→ The process of ADC is a slow process, & the uc has to wait for digital data till the conversion is over.

→ After the conversion is over, the ADC sends EOC signal to inform the uc about it & the result is ready at the o/p buffer of ADC.

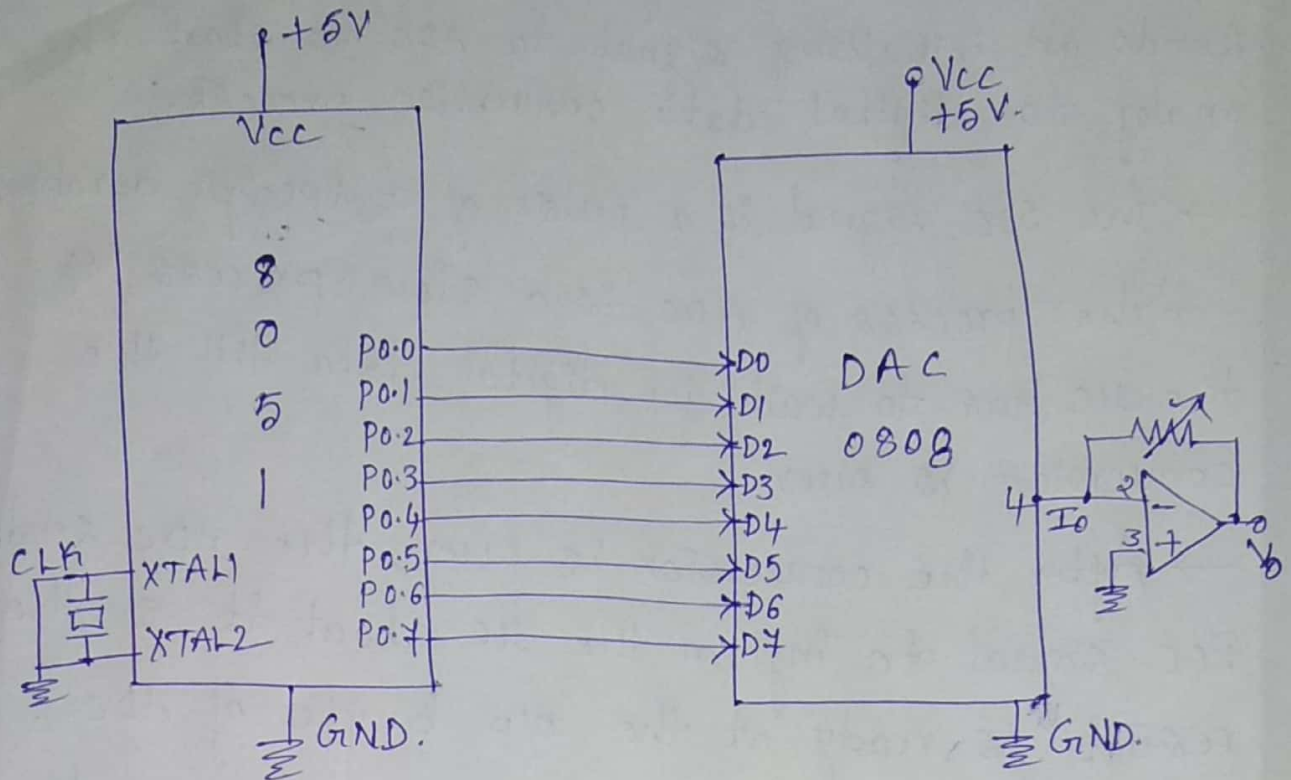
→ The time taken by the ADC from the active edge of soc pulse till the active edge of EOC signal is called the conversion delay of ADC.

→ These converters initially have a 3:8 analog multiplexer so that at a time 8 different analog inputs can be connected to the chips.

→ Out of these 8 inputs only one can be selected for conversion by using address lines A, B & C.

Analog i/p Selected	Address lines		
	C	B	A
I/P 0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

* Interfacing of DAC to 8051 *



→ The DAC convert binary numbers into their equivalent analog voltages.

→ An operational amplifier is used as a current to voltage converter at the o/p of DAC. to convert current o/p to a proportional o/p voltage.

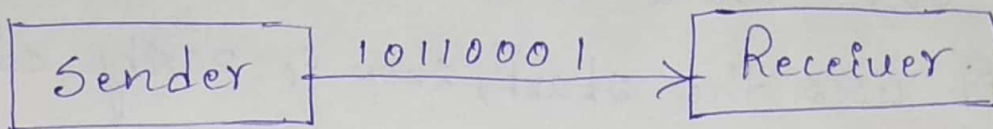
→ An external feedback resistor acts to control the gain.

Serial Communication & BUS Interface.

* Serial communication:

→ The process of sending data one bit at a time, sequentially, over a communication channel or computer bus.

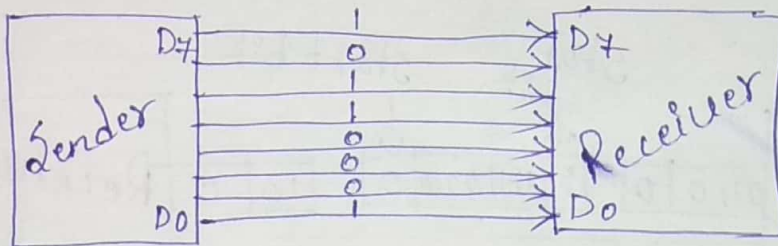
Serial transfer.



* Parallel communication:

→ The process where several bits are sent as a whole, on a link with several channels.

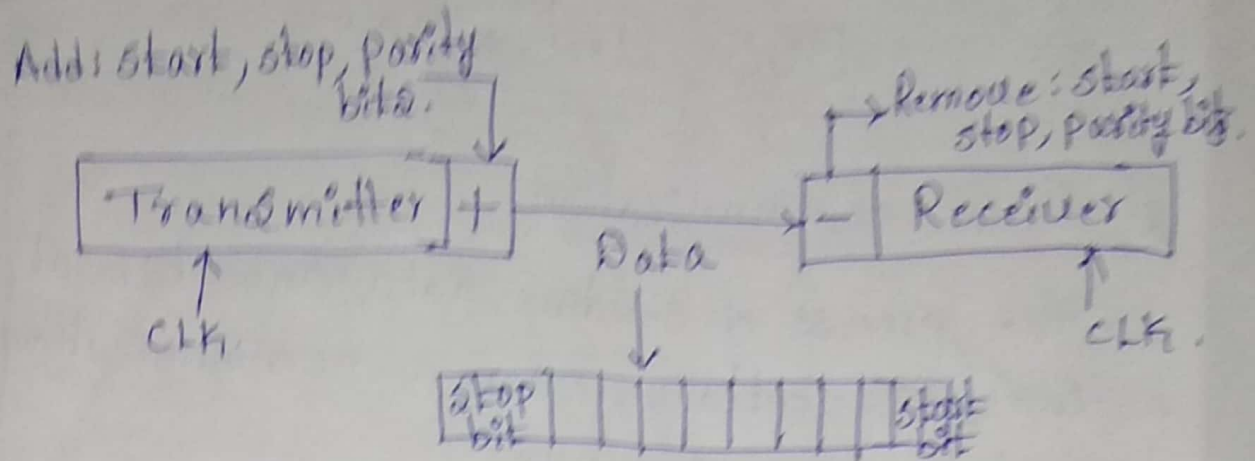
Parallel transfer.



* Types of Serial communication:

1. Asynchronous Serial communication.
2. Synchronous " "

* Asynchronous Serial communication:

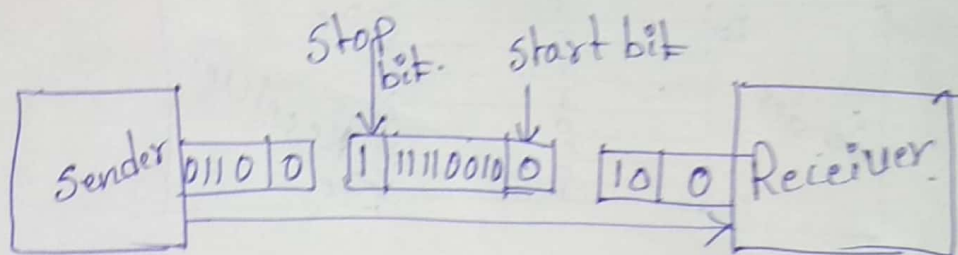


→ Both the transmitter & Receivers are have those own clocks to send & receive the data.

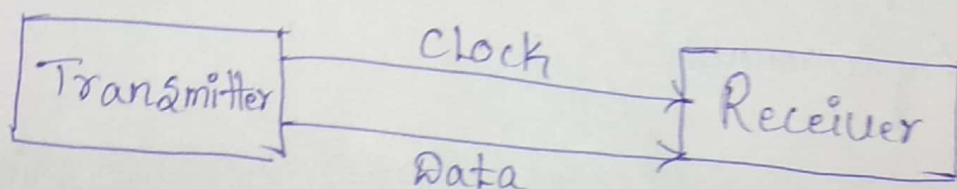
→ It uses the start, stop & parity check bits.

→ Data is sent at a regular intervals of time.

→ Only 1 byte is sent at a time. there is an ideal time b/w 2 data bytes. Ex: Emails.



* Synchronous Serial communication:



→ The tx & Rx both are using same clock, that is provided by the transmitter. ②

→ Provides the clock as a separate signal.

→ No start, stop or parity bits added to data.

→ Used for high speed data transmission.

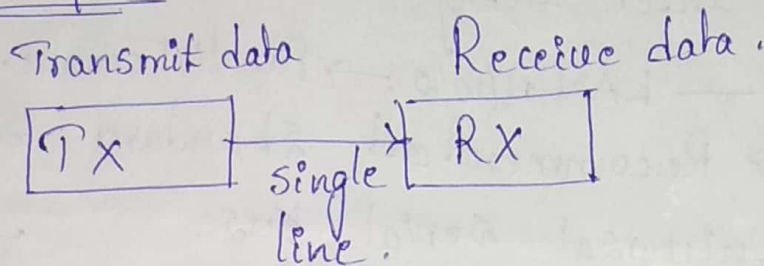
→ Expensive.

→ Ex: Video conferencing.

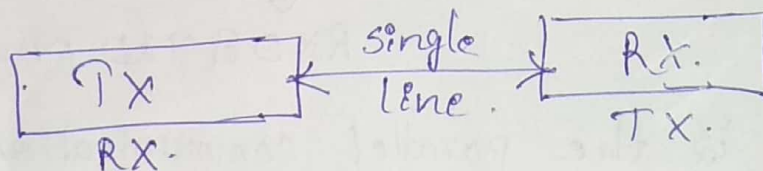
* Serial Data transfer Schemes:

① Simplex ② Half Duplex ③ Full Duplex.

① Simplex:



② Half Duplex:



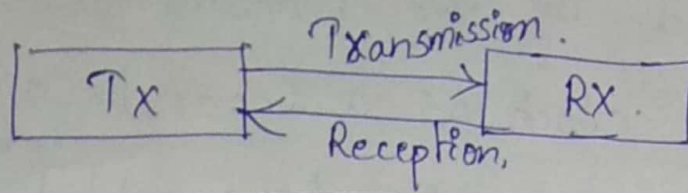
→ Can be used for either tx or Rx.

→ Only one direction at a time.

③ Full Duplex:

→ Data is transmitted in both ways at

a time.



* Serial communication standards: or protocols:

→ A protocol is a set of rules & regulations.

→ A communication protocol is a set of rules & regulations that allowed electronic devices to connect to exchange the data with one & another.

- * → SPI → Serial peripheral Interface
 - * → I2C → Inter Integrated circuit.
 - Ethernet. — LAN N/w's. → Parallel commu.
 - RS-232 → Recommended standard 232.
 - USB → Universal serial bus.
 - UART/USART. → Basic purposes, Not available in advanced devices. → 8051 supports only.
- ↓
RXD & TXD pins.

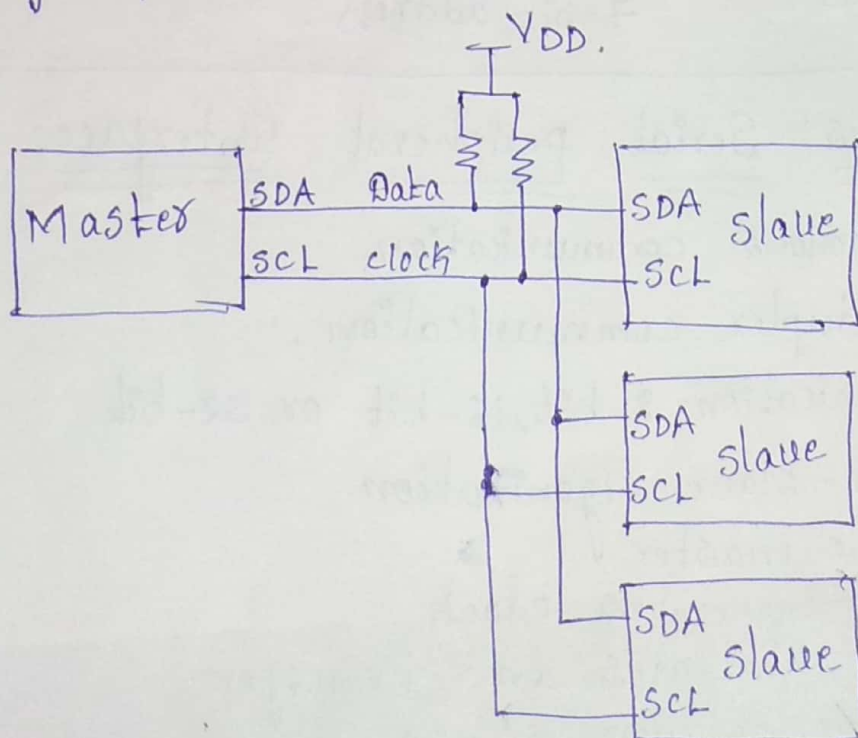
* TCP/IP is the parallel communication protocol.

* I2C communication protocol: (Developed by Philips) (3)

- Inter-Integrated circuit communication.
- Referred to as IIC or I2C or I²C.
- Also called two wire Interface (TWI).
- chip-to chip bidirectional communication standard. (not simultaneous).

Features:

1. 8-bit communication.
2. Speeds: 100kbps, 400kbps & 3.4Mbps.
3. Master & slave concept.
4. Two I/Os — SDA & SCL — (Serial data & serial clock)
5. Allows upto 127 devices to be connected.
6. 7-bit, 10-bit slave addressing.
7. Each device must have a unique address.
8. Synchronous communication.
9. Multi-master support.
10. Half Duplex communication.



→ Simple, no need for chip select signals (addressing via SDA line).

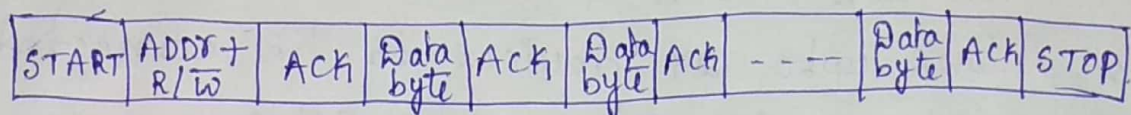
→ External pullup resistors required.

→ Master device: Initiates & terminates bus transaction & generates all clock pulses.

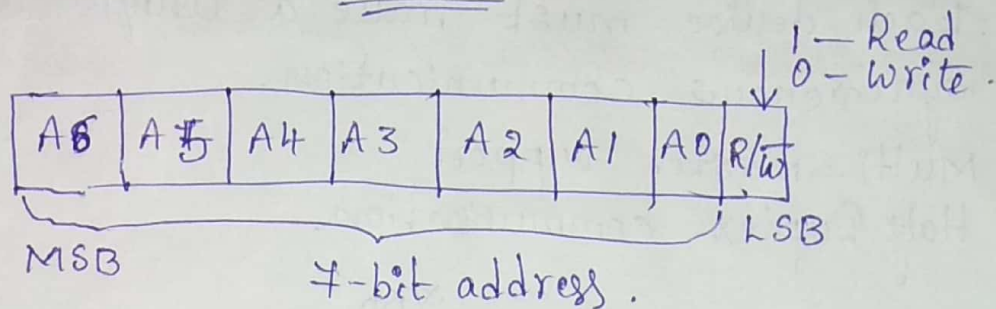
→ Slave device: Listens for address and responds as necessary.

* Signaling waveforms:

* Write operation or Read operation.



R/W operation.



* SPI Bus: Serial peripheral Interface:

→ Synchronous communication.

→ Full duplex communication.

→ Communication: 8-bit, 16-bit or 32-bit.

→ Master-slave organization.

— One master.

 └ Generates clock.

 └ Initiates data transfer.

— One or more slaves.

→ Four wire Communication.

— SCLK/SCK - Serial clock.

— MOSI - Master out, slave in } Serial data In (SDI)

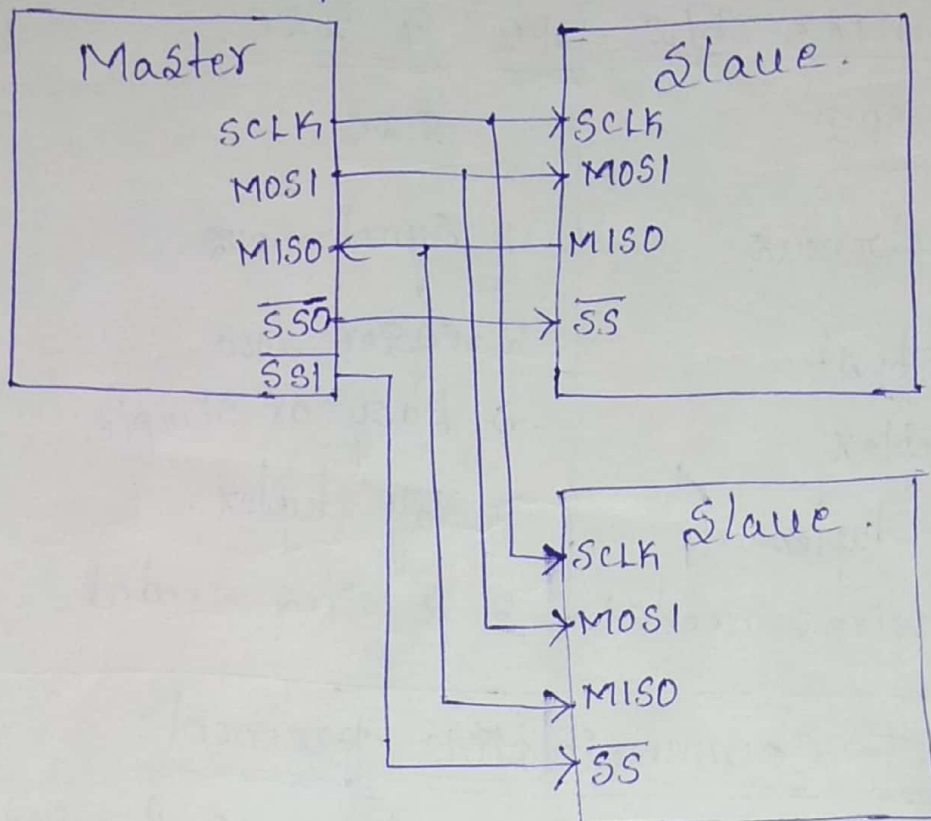
— MISO - Master in, slave out } Serial data out (SDO)

— SS - slave select (or chip select).

└ Active low signal.

└ Tells slave to prepare for data.

→ Developed by Motorola.



→ Frequency upto 4MHz.

→ Fastest speed comparing to I2C.

SCLK

$\overline{SS}$

MISO

MOSI

Applications:

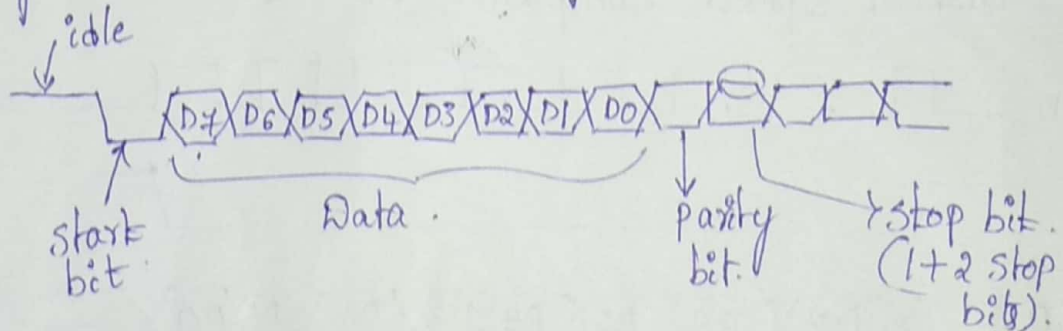
- LCD
- ADC/DAC
- Sensors.
- EEPROM.
- MP3 players.
- Cell phones
- Camera's.
- Set-top boxes.
- Desktop routers.

* Differences b/w SPI & I2C:

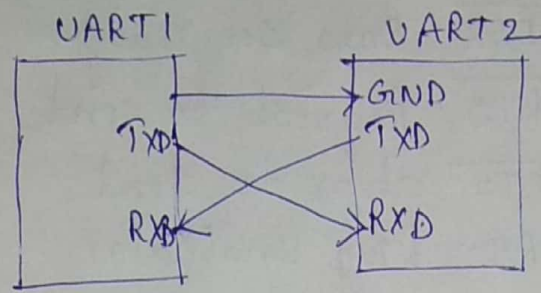
<u>SPI</u>	<u>I2C</u>
1. Synchronous	1. Synchronous.
2. Fastest	2. easy slow
3. Complex	3. Easy or simple.
4. Full duplex.	4. Half duplex.
5. 4 wires needed.	5. 2 wires needed.

* UART Communication protocol:

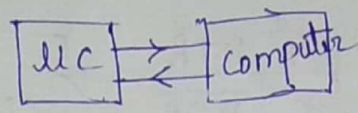
UART → Universal Asynchronous Receiver/Tx.
Asynchronous → no clock signal.



- Single wire communication.
- Start bit — Initiates the transfer.
- Speed — 9600 baud or 9600 bits/sec.



- UART offers a full-duplex data exchange.
- Used for 8051.



* External communication Interfaces

* RS232: Recommended standard 232

- Developed by Electronics Industries Association in 1960.
- To allow compatibility among data communication equipment made by various manufactures, an interfacing standard called RS-232 was developed.
- Its I/p & o/p voltages are not TTL compatible.
- RS-232
 - logic 0 (+3V to +25V) ("space") Receiver
 - logic 1 (-3V to -25V) ("MARK") Tx

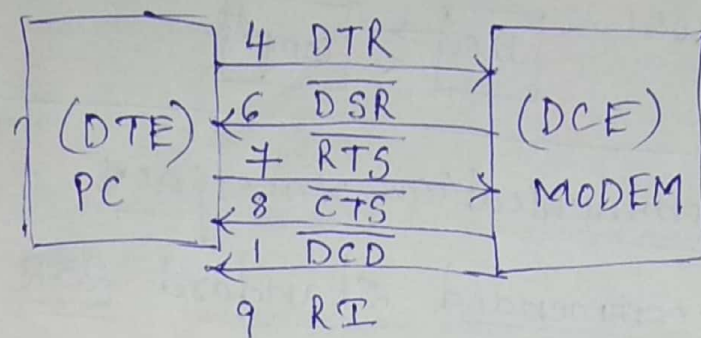
+5 to +15Vdc.
 -5 to -15Vdc.

→ DB-9 Pin connection.

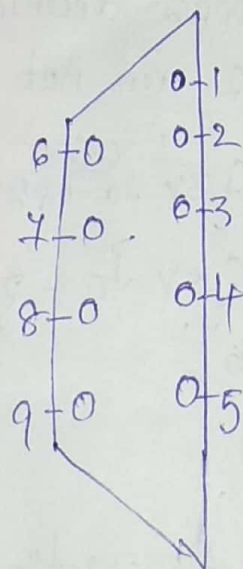
<u>Pin</u>	<u>Description.</u>
1	
2	DCD Data carrier detector
3	RXD Receive data
4	TXD Transmitted data.
5	DTR Data terminal ready.
	GND Signal ground

- 6 $\overline{DSR}$ Data Set ready.
- 7 $\overline{RTS}$ Request to send.
- 8 $\overline{CTS}$ clear to send
- 9 RI Ring Indicator.

→ It is connected using a DB9 connector.



- NOT Maximum data rate is 20k bits/sec.
- DTE should be male & DCE should be female.
- Existing communication lines (Telephone Telex).
- Used for Interfacing μ comp to terminal communication.

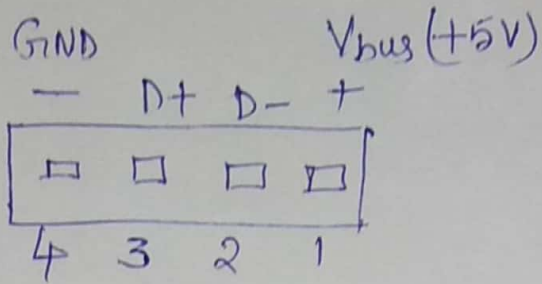


DB9 Connector.

* USB:

Universal Serial bus:

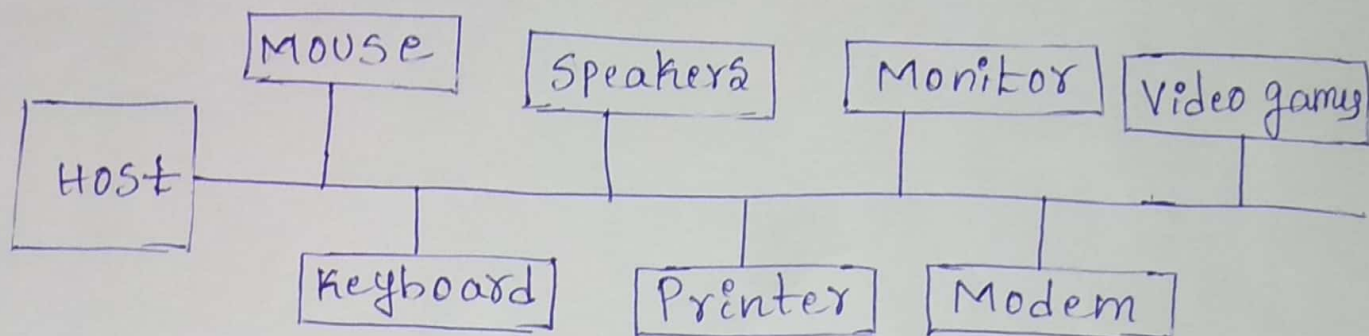
- Asynchronous serial protocol.
- Max 127 devices.



→ Serial communication & reception b/w host & serial devices.

→ USB 1.1 → low speed 1.5Mbps.

USB 2.0 → high speed 480Mbps.



UNIT-IV

①

ARM ARCHITECTURE

* ARM Processor :-

- ARM stands for Advance RISC Machines (up's)
- It is a RISC processor architecture.
- Gives high performance & low power consumption.

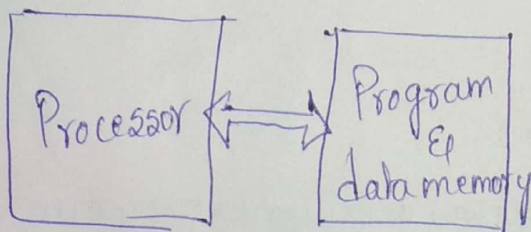
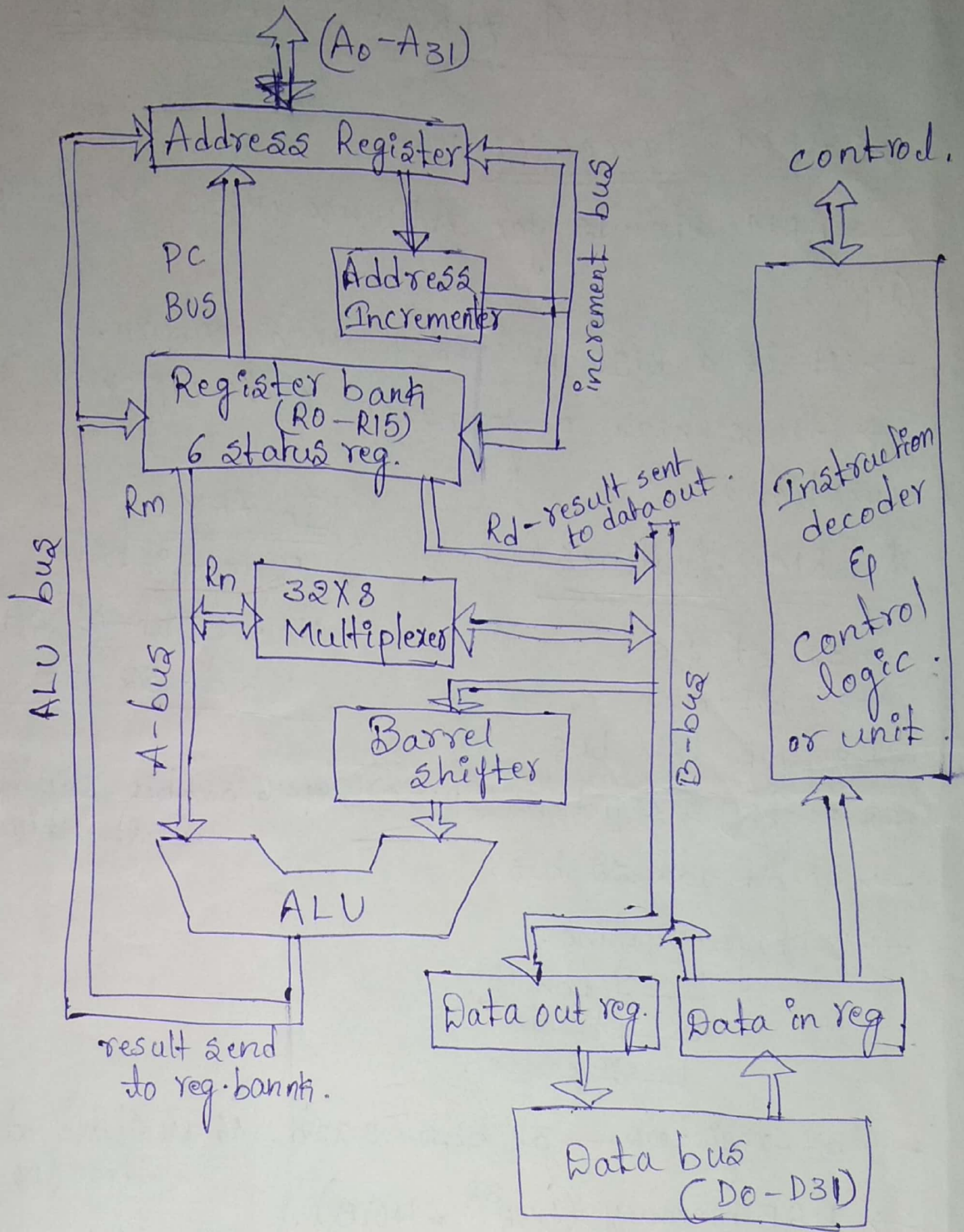
* ARM Features:

- 32-bit MC
- 32-bit ALU
- 32-bit data bus
- 32-bit instructions (RISC). & 16/32-bit Thumb Instruction set.
- 32-bit address bus.
- 3 stage pipeline.
 - └ Fetch
 - └ Decode
 - └ Execute.
- 37 registers — 32 bits each. (16 available at a time (R0-R15)).
- 4GB memory ($\because 2^{32} = 4GB$).
- Load-store model.
- 7 Operating modes.
- 7 Interrupts/Exceptions.
- Von Neuman Model (common memory-program & data).
- 4 addressing modes.
- 3 data formats (8, 16, 32-bit).

In ARM:

Byte - 8 bit.
Half word - 16 bit.
Word - 32 bit.

* ARM Architecture:



Von Neumann architecture.

→ In Arm processors the data can be manipulated only in the registers not in the data memory. (2)

Data bus:

- The data & instructions share the same (data) bus., because it is von-neuman architecture.
- The data enters into the core ARM through the data bus.

Instruction decoder:

- This unit decodes the instruction opcode read from the memory & then the instruction is executed.

Register file:

- This is the bank of 32-bit registers used for storing data items.

Sign Extend hardware:

- The instructions treat the registers as holding signed or unsigned 32-bit values.
- When processor reads 8-bit or 16-bit numbers from memory, the sign extend hardware converts these numbers into 32-bit values & then places into a register file.

ALU & MAC (Multiply - Accumulate unit):

- Most of the ARM instructions are the

two operand instructions.

→ The two source registers R_n & R_m using the internal data bus A & B.

→ The ALU reads the operands values from R_n & R_m registers via A & B buses.

→ Perform the operation & store them in to the destination register R_d & then to the register file.

→ Address Register:

→ This holds the address generated by the load & store instructions & places it in on the address bus.

Barrel shifter:

→ The address of the R_m register alternatively can be preprocessed in the barrel shifter before applying an i/p to the ALU.

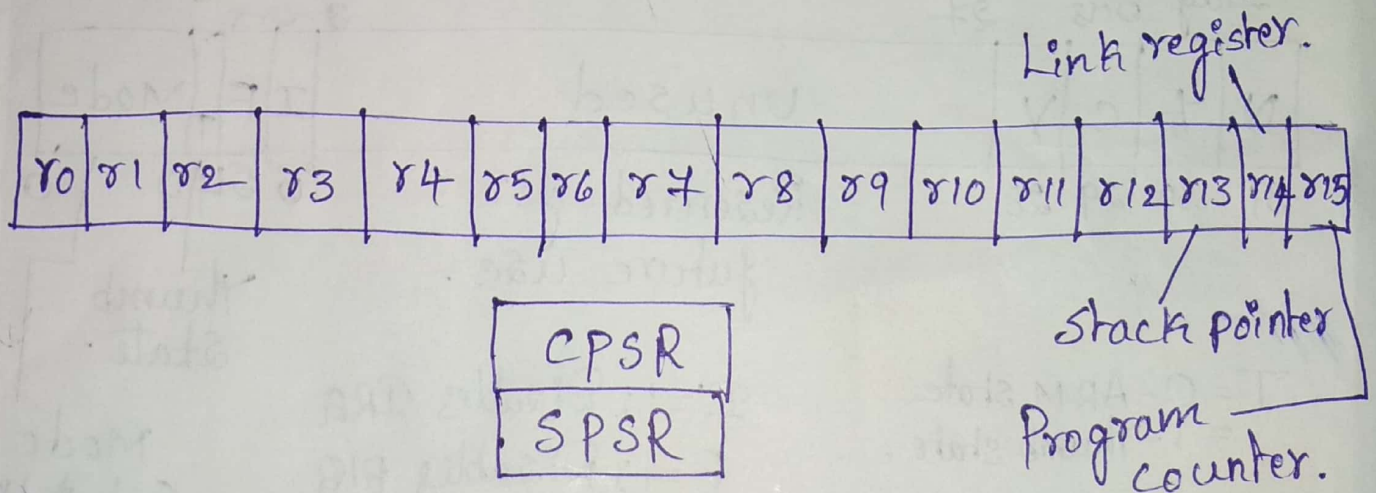
Incrementer:

→ The incrementer updates the contents of the address register before the processor reads or writes the next register value from or to memory location.

* ARM Register Set:

(3)

- 30 General purpose registers
- 1 dedicated program counter
- 1 dedicated current program status register
- 5 dedicated saved program status registers.
- There are 13 general purpose registers (r0-r12) available for the programmers to use.
- r prefix, means the RAM location & also register.
- All the registers are 32 bits in their size.
- There are some special function registers r13, r14 & r15.

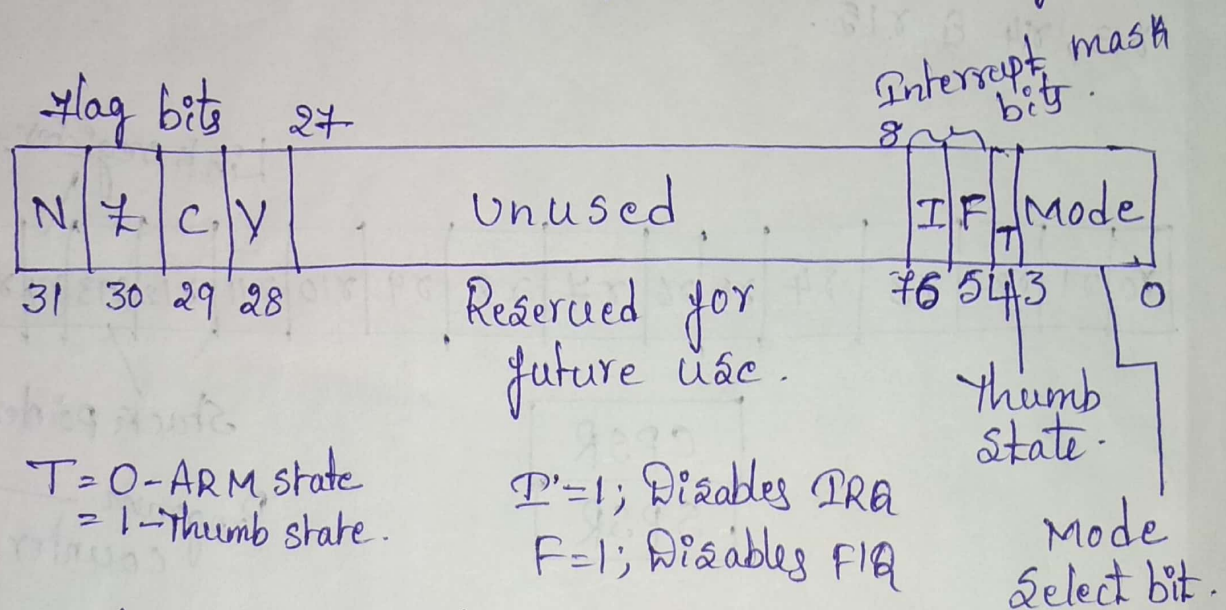


Stack pointer (r13): Every uc or a up will have a stack area for temporary storage purpose. It will hold the address of the top of the stack.

Link register (r14): When a subroutine is called, the return address will be put in here & when the subroutine execution is over, this register will be used for fetching the address back from where the control of execution has been initiated.

Program Counter (r15): It has got no address. It gives an idea about where should the control go next. The address of next instruction to be executed will be kept in PC.

* CPSR (Current Program Status Register)



→ This is referred to be a status register or a flag register.

→ CPSR is a 32 bit register.

→ There are 4 flags: Negative flag, Zero flag, Carry flag & Overflow flag.

(4)

Negative flag: will be set if result of an ALU operation is negative.

Zero flag: will be set if result of an ALU operation is zero.

Carry flag: will be set if result of an ALU is getting a carry generated.

Overflow flag: when result of a signed operation results in an overflow, this flag is set.

→ Few bits are reserved & kept for future usage.

→ There are two bits which are meant for interrupt masking purpose.

I → Can mask or unmask IRQ interrupt.

F → Can mask or unmask FIQ interrupt.

→ Thumb bit, it can help in enabling or disabling the thumb mode.

→ Mode select bit: One of the seven modes can be selected.

* ARM has seven operating modes:

User: Unprivileged mode under which most tasks run.

FIA: Entered when a high priority (Fast) interrupt is raised.

IRA: entered when a low priority (normal) interrupt is raised.

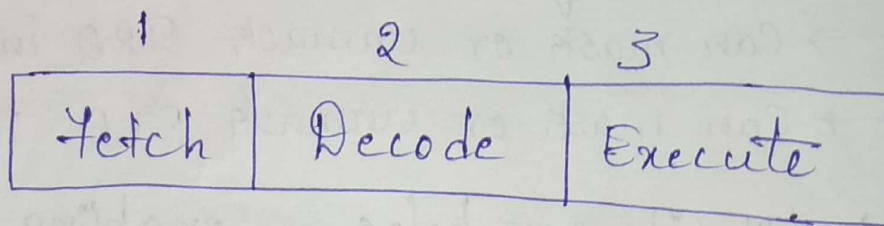
Supervisor: entered on reset & when a ϕ/w interrupt instruction is executed.

Abort: Used to handle memory access violations.

Undef: Used to handle undefined instructions.

System: Privileged mode using the same registers as user mode.

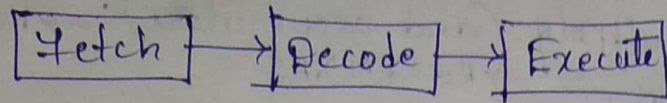
* Pipeline:



→ Pipelining is an implementation technique whereby multiple instructions are overlapped in execution. It is not visible to the programmer.

→ Pipeline machine cycle is the time required to move an instruction one step down the pipeline.

→ Throughput of a pipeline: number of instructions (5) that can leave the pipeline in each cycle.



Ex: A task takes a total time of T .

→ The task is divided into K subtasks.

* ARM Instruction Set:

→ All ARM instructions are 32 bits long, Instructions are stored word-aligned.

→ Thumb instructions are of 16 bits long. Instructions are stored half-word aligned.

① Data processing Instructions:

→ These instructions operate on the general-purpose registers.

→ They can perform operations such as addition, subtraction, or bitwise logic on the contents of two registers & place the result in a third register.

→ They can also operate on the value in a single register, or on a multiple registers.

Ex: `ADD R1, R1, #9` — $R1 = R1 + 9$

`SUB R2, R1, #6` — $R2 = R1 - 6$

`AND R3, R1, R2` — $R3 = R1 * R2$

`MOV R4, R2`

ADD, ADC : Add (w. carry)

SUB, SBC : Subtract (w. carry)

RSB, RSC : Reverse subtract (w. carry)

MUL, MLA : Multiply (and accumulate) : $\rightarrow R_d = (R_m \times R_s) + R_n$
 $\rightarrow R_d = R_m \times R_s$

AND, ORR, EOR

BIC : Bit clear

Logical.

LSL, LSR : Logical shift left/right

ASL, ASR : Arithmetic shift left/right

ROR : rotate right

RRX : Rotate right extended with C.

CMP : compare.

BIC r0, r1, r2

$\rightarrow r0 = r1 \text{ AND } (\text{NOT } r2)$

② Branch Instructions.

③ Load-store instructions.

④ Software interrupt instructions

⑤ Program status register instructions.

② Branch & control instructions:

$\rightarrow$ These instructions are used to call a routine.

or * Branch to subroutines

* Branch backwards to form loops.

* Branch forward in conditional structures.

* Change the processor between ARM state & Thumb state.

LOOP CMP R5, #3
BGE EXIT_LOOP

* B $\rightarrow$ Branch (label) $\rightarrow$ PC

BL $\rightarrow$ Branch with Link

BGE $\rightarrow$ Branch with >

③ Register load & store instructions:

$\rightarrow$ Load or store the value of a single register or multiple registers

from or to memory.

→ They can load or store a 32-bit word, a 16-bit halfword, or an 8-bit byte.

Ex: `MOV R6, #0x4` `LDR r0, [r1, #8]`
`LDR R1, [R6]` — Load single register.
`LDM R6, {R1-R5}` — Load multiple registers.

④ Program status register instructions:

→ Move the contents of a status register to or from a general purpose register.

Ex: `MRS R1, CPSR` — Read the CPSR.

→ Both CPSR & SPSR can be accessed.

⑤ Software interrupt instructions:

→ Support a general way to extend the ARM architecture.

Load & Store examples:

`LDR` `STR`: word.

`LDRB` `STRB`: Byte

`LDRH` `STRH`: Halfword.

`LDRSB` : signed byte load.

`LDRSH` : signed halfword load.

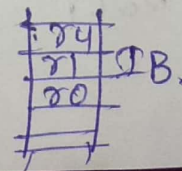
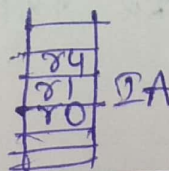
`LDMIA`/`STMIA` : Increment after.

`LDMIB`/`STMIB` : Increment before.

`LDMDA`/`STMDA` : Decrement after.

`LDMDB`/`STMDB` : Decrement before.

Ex: `LDMXX r10, {r0, r1, r4}`.



⑤ Software Interrupt Instruction:

→ It causes a s/w interrupt exception, which provides a mechanism for applications to OS routines.

Ex: SWI — S/w instruction is caused to swi exception to occur.

BRPT — Break point instruction is used for s/w break points.

* Interrupts & exceptions:

→ Interrupts & exceptions alter the normal program flow to handle external events or to report errors or exceptional conditions.

→ Interrupts are used to handle asynchronous events external to the processor.

→ IRA, FIA, Reset.

→ Exceptions handle conditions detected by the processor itself in the course of executing instructions.

→ Prefetch abort, Data abort, undefined, SWI instructions.

Interrupt vector table:

→ When an exception or interrupt occurs, the processor sets the PC to a specific memory address.

- The address is within a specific address range called the Vector Table.
- If an exception or an interrupt occurs in a program, the PC is directed to a particular memory address by the processor.
- The directed address lies in a special range of address known as the vector table.
- Vector table consists of a number of instructions (branch to specific routines) that are used to control a specific exception or interrupt.
- The address range of vector table is from 0x0000.0000 to 0x 0000 001C of 32-bit words.
- If an exception or interrupt arises, the processor stops the current execution & loads instructions from the vector table.

Exception/Interrupt	Instruction	Address
Reset	RESET	0x 0000 0000
Undefined Instruction	UNDEF	0x 0000 0004
Software Interrupt	SWI	0x 0000 0008
Prefetch abort	PABT	0x 0000 000C
Data abort	DABT	0x 0000 0010
Reserved	—	0x 0000 0014
Interrupt request	IRQ	0x 0000 0018
Last Interrupt request	FIQ	0x 0000 001C

* Reset: Points to the location of first instruction in the table.

→ It is executed if power is applied & branches to the initialization code.

* Undefined Instruction:

→ It is employed if an instruction is not decoded by the processor.

* Software Interrupt:

→ This instruction is called by the processor when SWI instruction is executed.

→ SWI is frequently used as a mechanism to invoke operating system routines.

* Prefetch abort: (Instruction fetch memory fault)

→ It occurs when the processor attempts to fetch an instruction from an address without correct access permissions or the contents of that virtual address is not available in the memory.

* Data abort: (Data access memory fault)

→ It is similar to a prefetch abort.

→ It occurs if data memory is accessed by an instruction with incorrect access permissions.

* Interrupt Request Vector: (Normal interrupt)

→ It is employed to interrupt the flow of current

execution by an external hardware. It appears ^⑧ only if IRQs are not masked in the CPSR.

* Fast Interrupt request:

→ An external hardware uses this instruction if there is a necessity of faster response time. It appears only if FIQs are not masked in the CPSR.

* → If the multiple exceptions/Interrupts can happen at the same time or while one is being handled another one might happen, the priorities of exceptions/Interrupts are given below.

Priority	Exception.
Highest	Reset
	Data abort
	FIQ
	IRQ
	Pre-fetch abort
Lowest.	Undefined & SWI

* Thumb Instruction Set:

→ The Thumb state is one of the states of ARM processor that executes the 16 bit Thumb instructions.

→ It is a subset of standard ARM 32 instructions
→ It is developed by ARM for reducing the memory requirements.

→ In ARMv7TDMI, T stands for 'Thumb'.

↳ (ARMv7 + 16-bit Thumb + JTAG Debug + fast multiplier + enhanced ICE)

→ The different operations that are supported by thumb instruction set is

1. Arithmetic & Logic Operations
2. Load/Store Data transfer
3. Conditional/Unconditional branches.

→ The register sets that are available in 16-bit thumb state are R0-R7, SP, Link reg. (LR), Program counter (PC).

→ Thumb implementation of the same code takes up around 30% less memory than the equivalent ARM implementation.

→ The ARM thumb instruction set supports an 8-bit, 16-bit & 32-bit data types & also the memory is half-word aligned in which each offset gets increased or decreased by two addresses.

* Loading constants:

→ There is no ARM instruction to move a 32-bit constant into a register.

→ Since ARM instructions are 32 bits in size & cannot specify a general 32-bit constant.

→ To support for programming there are two pseudo instructions to move a 32-bit value into a register.

Syntax: LDR Rd, =constant. Rd ← mem32
ADR Rd, label.

LDR load constant to register. Rd = 32-bit constant.

ADR load relative address to register.

↳ Rd = 32-bit relative address.

→ The first pseudoinstruction writes a 32-bit constant to a register using whatever instructions are available.

→ It defaults to a memory read if the constant cannot be encoded using other instructions.

→ The second pseudoinstruction writes a relative address into a register which will be encoded using a PC-relative expression.

EX: loading a 32-bit constant 0xff00ffff into register r0.

LDR r0, [PC, #constant-number-8-{PC}].

⋮
Constant-number.
DCD 0xff00ffff.

EX: LDR r0, =0xff		MOV r0, #0xff.
Pseudo instruction		Actual instruction.

* Conditional execution:

→ ARM instructions can be made to execute conditionally by postfixing them with the appropriate condition code field.

→ This improves the code density & performance by reducing the no. of forward branch instructions.

CMP r3, #0
BNE skip ←
ADD r0, r1, r2
skip.

CMP r3, #0
ADDNE r0, r1, r2

EX: CMP r0, #0.
BNE else
ADD r1, r1, #1
B end
else
ADD r2, r2, #1
end.

CMP r0, #0
ADDEQ r1, r1, #1
ADDNE r2, r2, #1

* unconditional.

→ 5 Instructions, 5 cycles.

* Conditional

→ 3 Instructions,
3 cycles.

* Condition mnemonics:

(10)

Suffix	Description.	Flags tested.
EQ	Equal	$Z = 1$
NE	Not equal	$Z = 0$
CS/HS	Unsigned higher or same	$C = 1$
CC/LO	Unsigned lower	$C = 0$
MI	Minus	$N = 1$
PL	Positive or zero	$N = 0$
VS	Overflow	$V = 1$
VC	No overflow	$V = 0$
HI	Unsigned higher	$C = 1$ & $Z = 0$
LS	Unsigned lower or same	$C = 0$ or $Z = 1$
GE	Greater or equal	$N = V$
LT	Less than	$N \neq V$
GT	Greater than	$Z = 0$ & $N = V$
LE	Less than or equal	$Z = 1$ or $N \neq V$
AL	Always.	

- Conditional code field. → Mnemonic in instruction
- Condition flags. → in CPSR.
- It also maximizes the code density.

UNIT-V

①

Advanced ARM Processors

* Cortex Processor:

- Cortex Processors are 32-bit RISC Processors, manufactured based on ARMv7 architecture.
- These processors are mainly categorized into three families,

① Cortex-A:

- This is manufactured for high performance open application platforms.

② Cortex-R:

- For real time embedded systems.

③ Cortex-M:

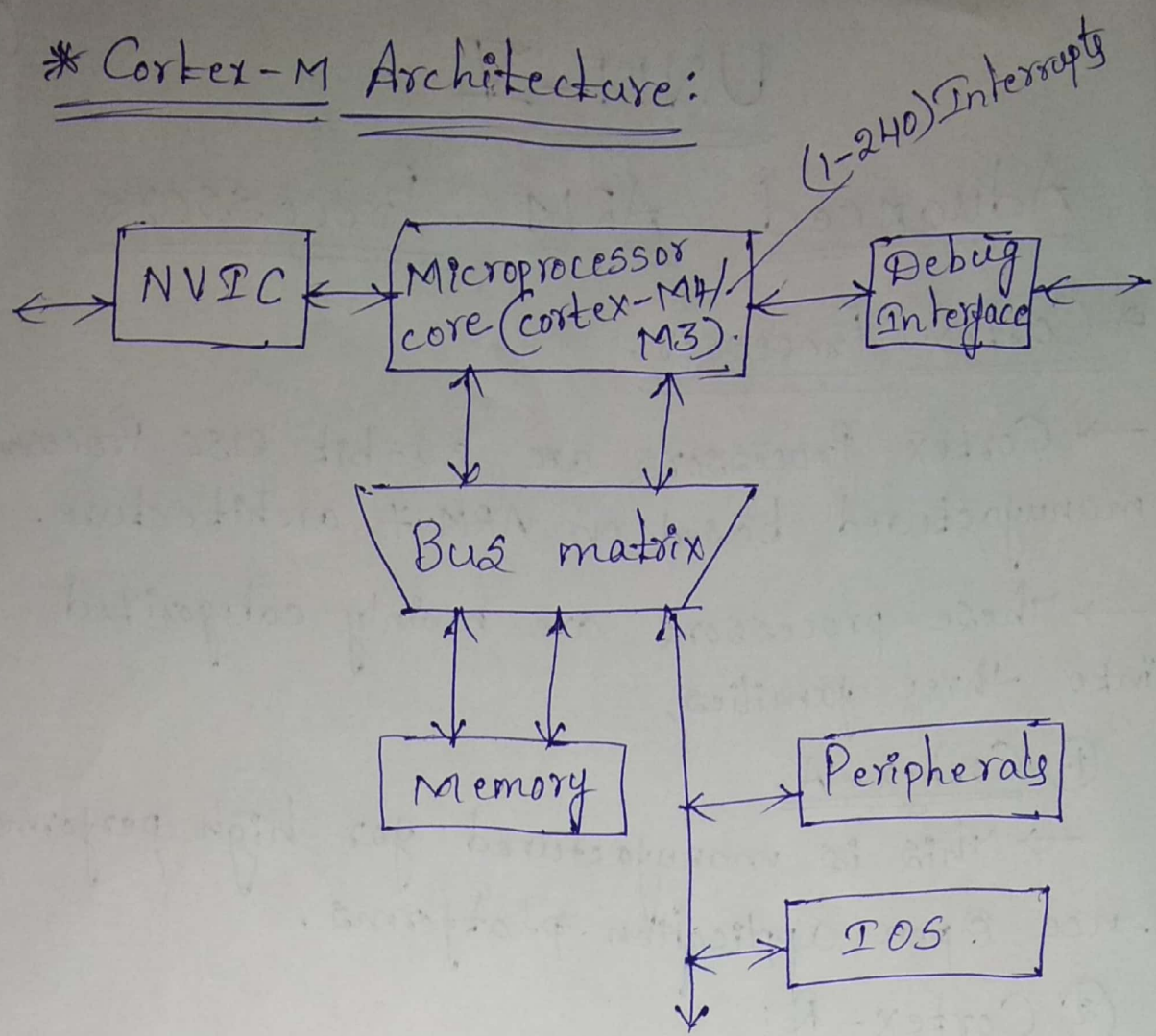
- For microcontroller based embedded systems.

→ Cortex-A processors support 8 stage pipeline & ARMv7.

→ Cortex-R family support 8 stage pipeline & ARMv7 architecture and Harvard memory architecture.

→ Cortex-M family support 3 stage pipeline, Von-neuman & Harvard memory, ARMv6-M, ..
→ 1 to 32 interrupts. ARMv7-M architecture.

* Cortex-M Architecture:



→ It is developed based on 32-bit RISC processor & Harvard architecture.

① Microprocessor core: It is the heart of CORTEX-M processor.

→ It employs two individual buses for simultaneous transmission of instruction & data without any effect.

→ Hence Improved performance can be achieved.

② NVIC (Nested Vectored Interrupt Controller):

→ NVIC provides interrupt handling capabilities

to processor.

(2)

→ The processor handles the interrupts by using relocatable vector table that contains address of the exception handler & PSR.

③ Bus matrix:

→ It associates the memory, peripherals & debug system with up core.

④ Memory & peripherals:

→ It employs a 4GB memory map. It is stored in a ROM to contribute for several system devices. (Peripheral - APB - Advanced peripheral bus)

⑤ Debug system:

→ It provides the features

- * Program Execution controls.

- * Registers & Memory accesses

- * Traces. → offers an effective trace information using a JTAG connector.

* Applications of CORTEx Processors:

* CORTEx-A Family

→ Smartphones

→ Tablets

→ E-book readers

→ Digital TV appliances

→ Electronic wallets

* Servers

* Desktop processors.

* CORTEX-R Family:

- Real time & safety applications.
- High end braking systems.
- Hard drive controllers.

* CORTEX-M Family:

- Low cost applications
 - Real time control systems.
 - Internet of things.
 - Human interface devices.
 - Automotive & industrial control systems.
 - Consumer & Medical instrumentation.
 - Domestic household appliances.
-

* OMAP Processor Features:

- OMAP support high-level OS like windows CE & Linux.
- 40% faster performance.
- It contains multiple hardware versions for motion & pixel interpolation.
- There are four versions of OMAP i.e., OMAP1, OMAP2, OMAP3 & OMAP4.
- It is dual processor, combination of DSP core & RISC CPU. (ARM cortex A8 or A9).
- The advantage of using two processors is to reduce the consumption of power to a large extent i.e. greater battery life.

* OMAP Processor Architecture:

(3)

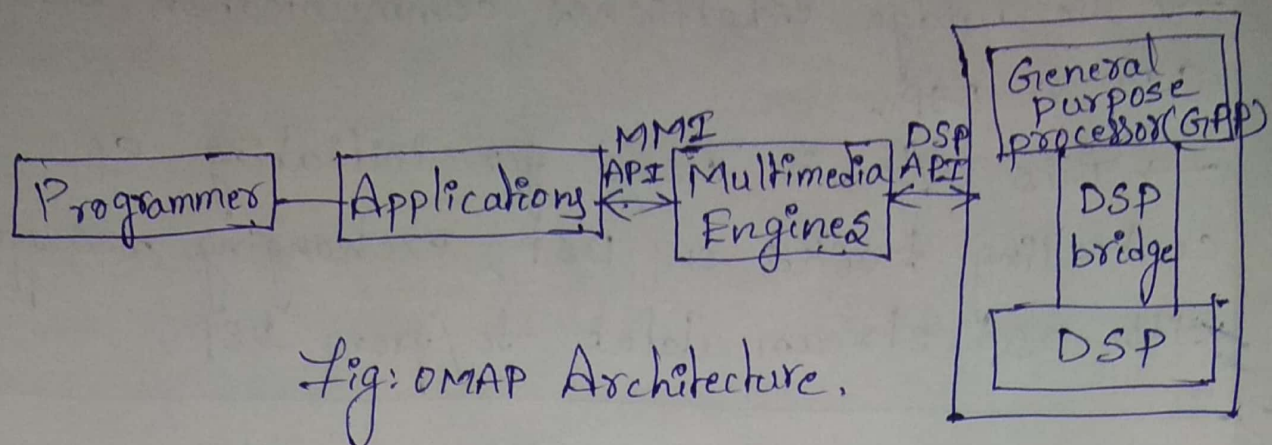


Fig: OMAP Architecture.

→ This architecture is a combination of DSP core & high performance RISC processor.

→ It is targeted for advanced speech & video processing tasks such as data compression, speech recognition & synthesis.

→ The DSP processor is more suitable for signal processing applications. such as speech.

→ recognition, video & audio playback.

→ The RISC processor is more suitable for execution of control instructions required for OS, man-machine interfaces & OS applications.

* OMAP employs standard programming interfaces such as API i.e. Application programming interfaces to user-friendly applications.

* Multimedia engines (i.e., video, web, speech) use proxies to perform DSP tasks in GPP domain.

- DSP bridge contains a set of APIs which comprises device driver interfaces.
 - The bridge establishes communication b/w GPP & DSP.
 - This bridge is used for initiating & controlling tasks on DSP, exchanging messages with DSP stream data to/from DSP.
-

* Applications of OMAP:

1. Speech Recognition
 2. Video & audio play back.
 3. Man-machine interfaces.
 4. OS applications.
 5. Video conferencing (simultaneously) applications.
- ⑥ Navigation.
-