# LECTURE NOTES

# ON

# MICROCONTROLLER & DIGITAL SIGNAL PROCESSING-R16

**B.Tech VI semester**

**Ms. J.SRAVANA**

**(Assistant professor)**

ELECRTONICS AND COMMUNICATION ENGINEERING

**INSTITUTE OF AERONAUTICAL ENGINEERING**

**(Autonomous)**

(Autonomous)DUNDIGAL, HYDERABAD - 50004

# MODULE – I

# MICROPROCESSORS AND MICROCONTROLLER

# UNIT-1

## Microprocessor and Microcontroller

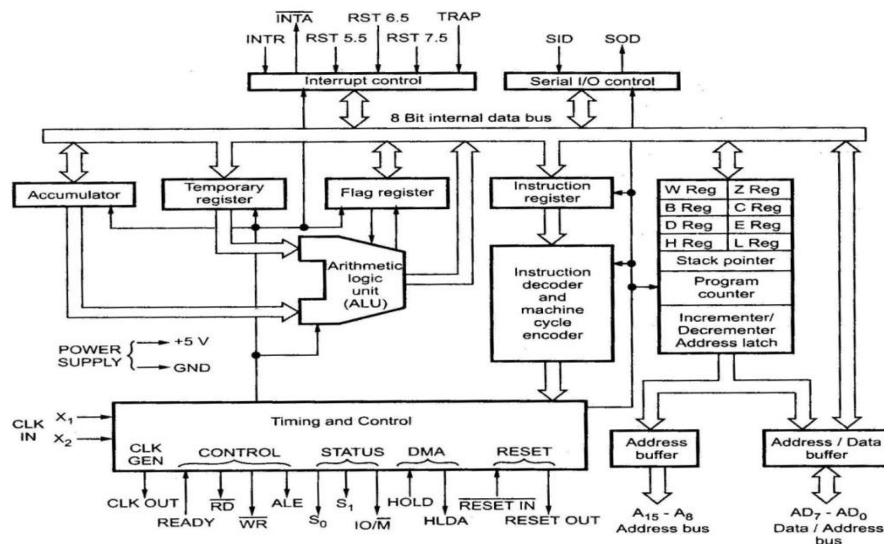
## 8086 Microprocessor

**Introduction to 8085 Microprocessor:**

**The Salient Features of 8085 Microprocessor:**

- 8085 is an 8 bit microprocessor, manufactured with N-MOS technology.
- It has 16-bit address bus and hence can address up to $2^{16}$ = 65536 bytes (64KB) memory locations through $A_0$-$A_{15}$.
- The first 8 lines of address bus and 8 lines of data bus are multiplexed $AD_0$ - $AD_7$. Data bus is a group of 8 lines $D_0$ - $D_7$.
- It supports external interrupt request.8085 consists of 16 bit program counter (PC) and stack pointer (SP).
- Six 8-bit general purpose register arranged in pairs: BC, DE, HL.
- It requires a signal +5V power supply and can operate at 3 MHz, 5 MHz and 6 MHz Serial in/Serial out Port.
- It is enclosed with 40 pins DIP (Dual in line package).

**Internal Architecture of 8085:**

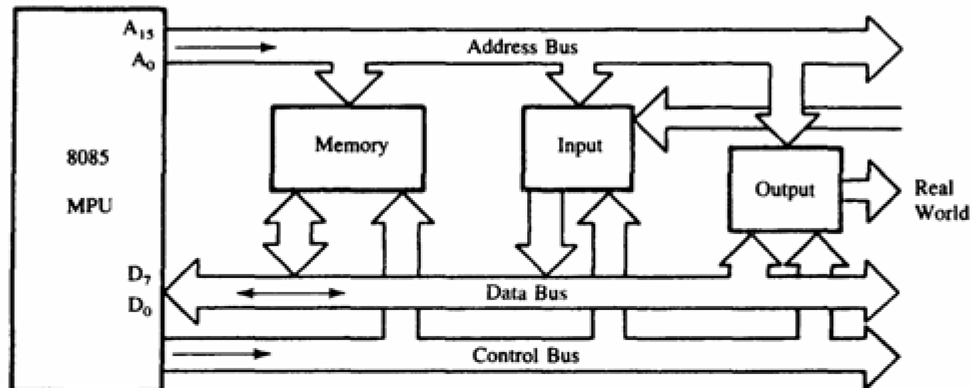



Architecture of 8085

**8085 Bus Structure:**

**Address Bus:**

- The address bus is a group of 16 lines generally identified as A0 to A15.
- The address bus is unidirectional: bits flow in one direction-from the MPU to peripheral devices.
- The MPU uses the address bus to perform the first function: identifying a peripheral or a memory location.




**Data Bus:**

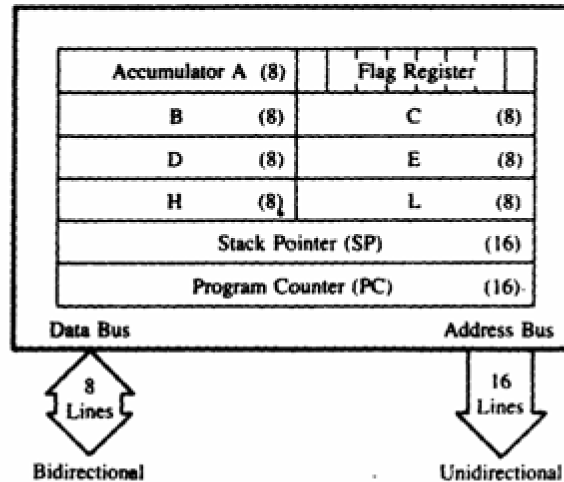
- The data bus is a group of eight lines used for data flow.
- These lines are bi-directional - data flow in both directions between the MPU and memory and peripheral devices.
- The MPU uses the data bus to perform the second function: transferring binary information.
- The eight data lines enable the MPU to manipulate 8-bit data ranging from 00 to FF (28 = 256 numbers).
- The largest number that can appear on the data bus is 11111111.

**Control Bus:**

- The control bus carries synchronization signals and providing timing signals.
- The MPU generates specific control signals for every operation it performs. These signals are used to identify a device type with which the MPU wants to communicate.

**Registers of 8085:**

- The 8085 have six general-purpose registers to store 8-bit data during program execution.
- These registers are identified as B, C, D, E, H, and L.
- They can be combined as register pairs-BC, DE, and HL-to perform some 16-bit operations.




**Accumulator (A):**

- The accumulator is an 8-bit register that is part of the arithmetic/logic unit (ALU).
- This register is used to store 8-bit data and to perform arithmetic and logical operations.
- The result of an operation is stored in the accumulator.

**Flags:**

- The ALU includes five flip-flops that are set or reset according to the result of an operation.
- The microprocessor uses the flags for testing the data conditions.
- They are Zero (Z), Carry (CY), Sign (S), Parity (P), and Auxiliary Carry (AC) flags. The most commonly used flags are Sign, Zero, and Carry.

The bit position for the flags in flag register is,

| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|
| S | Z | | AC | | P | | CY |

**1. Sign Flag (S):**

After execution of any arithmetic and logical operation, if D7 of the result is 1, the sign flag is set. Otherwise it is reset.

D7 is reserved for indicating the sign; the remaining is the magnitude of number.

If D7 is 1, the number will be viewed as negative number. If D7 is 0, the number will be viewed as positive number.

**2. Zero Flag (z):**

If the result of arithmetic and logical operation is zero, then zero flag is set otherwise it is reset.

**3. Auxiliary Carry Flag (AC):**

If D3 generates any carry when doing any arithmetic and logical operation, this flag is set. Otherwise it is reset.

**4. Parity Flag (P):**

If the result of arithmetic and logical operation contains even number of 1's then this flag will be set and if it is odd number of 1's it will be reset.

**5. Carry Flag (CY):**

If any arithmetic and logical operation result any carry then carry flag is set otherwise it is reset.

**Arithmetic and Logic Unit (ALU):**

- It is used to perform the arithmetic operations like addition, subtraction, multiplication, division, increment and decrement and logical operations like AND, OR and EX-OR.
- It receives the data from accumulator and registers.
- According to the result it set or reset the flags.

**Program Counter (PC):**

- This 16-bit register sequencing the execution of instructions.
- It is a memory pointer. Memory locations have 16-bit addresses, and that is why this is a 16-bit register.
- The function of the program counter is to point to the memory address of the next instruction to be executed.

- When an opcode is being fetched, the program counter is incremented by one to point to the next memory location.

**Stack Pointer (SP):**

- The stack pointer is also a 16-bit register used as a memory pointer.
- It points to a memory location in R/W memory, called the stack.
- The beginning of the stack is defined by loading a 16-bit address in the stack pointer (register).

**Temporary Register:** It is used to hold the data during the arithmetic and logical operations.

**Instruction Register:** When an instruction is fetched from the memory, it is loaded in the instruction register.

**Instruction Decoder:** It gets the instruction from the instruction register and decodes the instruction. It identifies the instruction to be performed.

**Serial I/O Control:** It has two control signals named SID and SOD for serial data transmission.

**Timing and Control unit:**

- It has three control signals ALE, RD (Active low) and WR (Active low) and three status signals IO/M(Active low), S0 and S1.
- ALE is used for provide control signal to synchronize the components of microprocessor and timing for instruction to perform the operation.
- RD (Active low) and WR (Active low) are used to indicate whether the operation is reading the data from memory or writing the data into memory respectively.
- IO/M(Active low) is used to indicate whether the operation is belongs to the memory or peripherals.
- If,

| IO/M(Active Low) | S1 | S2 | Data Bus Status(Output) |
|---|---|---|---|
| 0 | 0 | 0 | Halt |
| 0 | 0 | 1 | Memory WRITE |
| 0 | 1 | 0 | Memory READ |
| 1 | 0 | 1 | IO WRITE |
| 1 | 1 | 0 | IO READ |
| 0 | 1 | 1 | Opcode fetch |
| 1 | 1 | 1 | Interrupt acknowledge |

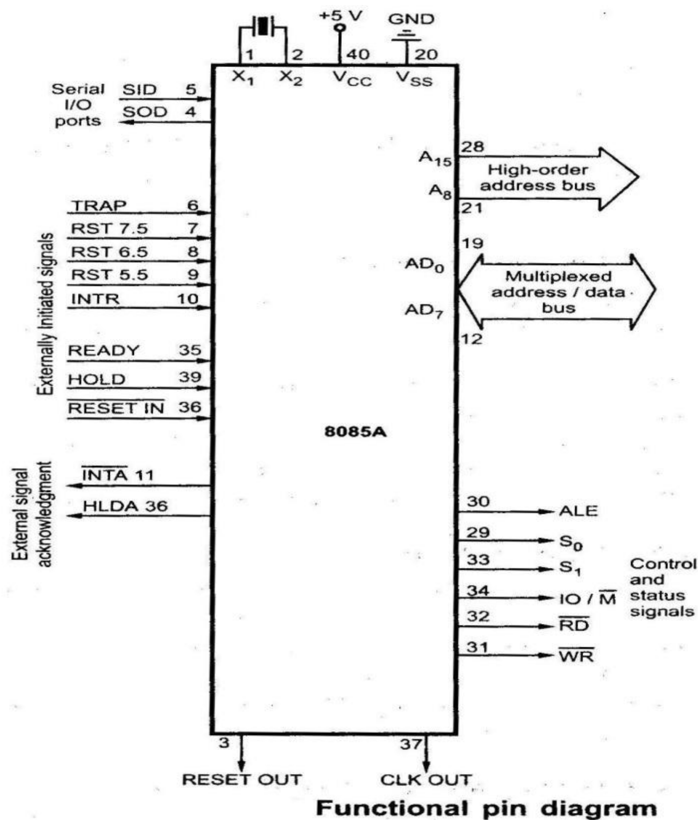
**Interrupt Control Unit:**

- It receives hardware interrupt signals and sends an acknowledgement for receiving the interrupt signal.

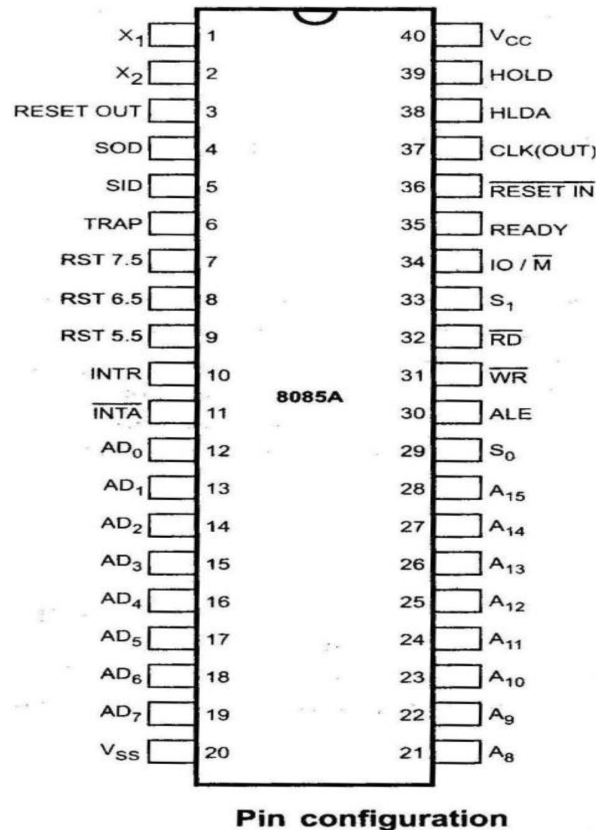
**Pin Diagram and Pin Description Of 8085**

8085 is a 40 pin IC, DIP package. The signals from the pins can be grouped as follows

1. Power supply and clock signals
2. Address bus
3. Data bus
4. Control and status signals
5. Interrupts and externally initiated signals
6. 
7. Serial I/O ports



Functional pin diagram



Pin configuration

**1. Power supply and clock frequency signals**

- Vcc + 5 volt power supply
- Vss Ground
- X1, X2: Crystal or R/C network or LC network connections to set the frequency of internal clock generator.
- The frequency is internally divided by two. Since the basic operating timing frequency is 3 MHz, a 6 MHz crystal is connected externally.
- CLK (output)-Clock Output is used as the system clock for peripheral and devices interfaced with the microprocessor.

**2. Address Bus:**

- A8 - A15 (output; 3-state)
- It carries the most significant 8 bits of the memory address or the 8 bits of the I/O address;

**3. Multiplexed Address / Data Bus:**

- AD0 - AD7 (input/output; 3-state)
- These multiplexed set of lines used to carry the lower order 8 bit address as well as data bus.
- During the opcode fetch operation, in the first clock cycle, the lines deliver the lower order address A0 - A7.
- In the subsequent IO / memory, read / write clock cycle the lines are used as data bus.
- The CPU may read or write out data through these lines.

**4. Control and Status signals:**

- ALE (output) - Address Latch Enable.
- This signal helps to capture the lower order address presented on the multiplexed address / data bus.
- RD (output 3-state, active low) - Read memory or IO device.
- This indicates that the selected memory location or I/O device is to be read and that the data bus is ready for accepting data from the memory or I/O device.
- WR (output 3-state, active low) - Write memory or IO device.
- This indicates that the data on the data bus is to be written into the selected memory location or I/O device.
- IO/M (output) - Select memory or an IO device.
- This status signal indicates that the read / write operation relates to whether the memory or I/O device.

- It goes high to indicate an I/O operation.
- It goes low for memory operations.

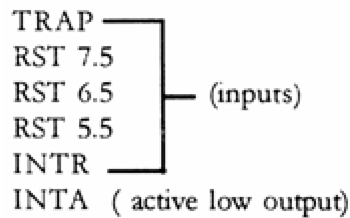
**5. Status Signals:**

- It is used to know the type of current operation of the microprocessor.

| IO/M (Active Low) | S1 | S2 | Data Bus Status (Output) |
|---|---|---|---|
| 0 | 0 | 0 | Halt |
| 0 | 0 | 1 | Memory WRITE |
| 0 | 1 | 0 | Memory READ |
| 1 | 0 | 1 | IO WRITE |
| 1 | 1 | 0 | IO READ |
| 0 | 1 | 1 | Opcode fetch |
| 1 | 1 | 1 | Interrupt acknowledge |

**6. Interrupts and externally initiated operations:**

- They are the signals initiated by an external device to request the microprocessor to do a particular task or work.
- There are five hardware interrupts called,




- On receipt of an interrupt, the microprocessor acknowledges the interrupt by the active low INTA (Interrupt Acknowledge) signal.

**Reset In (input, active low)**

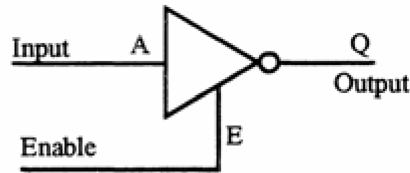
- This signal is used to reset the microprocessor.
- The program counter inside the microprocessor is set to zero.
- The buses are tri-stated.

**Reset Out (Output)**

- It indicates CPU is being reset.
- Used to reset all the connected devices when the microprocessor is reset

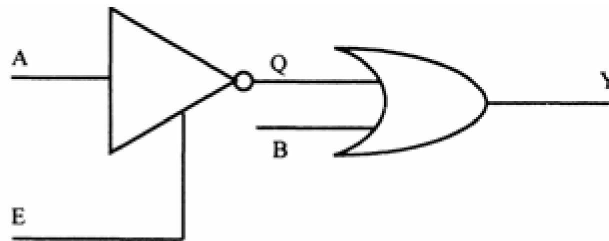
**7. Direct Memory Access (DMA):**

Tri state devices:




- 3 output states are high & low states and additionally a high impedance state.
- When enable E is high the gate is enabled and the output Q can be 1 or 0 (if A is 0, Q is 1, otherwise Q is 0). However, when E is low the gate is disabled and the output Q enters into a high impedance state.

| E | A | Q | State |
|---|---|---|---|
| 1(high) | 0 | 1 | High |
| 1 | 1 | 0 | Low |
| 0(low) | 0 | 0 | High impedance |
| 0 | 1 | 0 | High impedance |




- For both high and low states, the output Q draws a current from the input of the OR gate.
- When E is low, Q enters a high impedance state; high impedance means it is electrically isolated from the OR gate's input, though it is physically connected. Therefore, it does not draw any current from the OR gate's input.
- When 2 or more devices are connected to a common bus, to prevent the devices from interfering with each other, the tristate gates are used to disconnect all devices except the one that is communicating at a given instant.
- The CPU controls the data transfer operation between memory and I/O device. Direct Memory Access operation is used for large volume data transfer between memory and an I/O device directly.
- The CPU is disabled by tri-stating its buses and the transfer is effected directly by external control circuits.
- HOLD signal is generated by the DMA controller circuit. On receipt of this signal, the microprocessor acknowledges the request by sending out HLDA signal and leaves out the

control of the buses. After the HLDA signal the DMA controller starts the direct transfer of data.

**READY (input)**

- Memory and I/O devices will have slower response compared to microprocessors.
- Before completing the present job such a slow peripheral may not be able to handle further data or control signal from CPU.
- The processor sets the READY signal after completing the present job to access the data.
- The microprocessor enters into WAIT state while the READY pin is disabled.

**8. Single Bit Serial I/O ports:**

- SID (input) - Serial input data line
- SOD (output) - Serial output data line
- These signals are used for serial communication

## Overview or Features of 8086

- It is a 16-bit Microprocessor (µp).It's ALU, internal registers works with 16bit binary word.
- 8086 has a 20 bit address bus can access up to $2^{20}$= 1 MB memory locations.
- 8086 has a 16bit data bus. It can read or write data to a memory/port either 16bits or 8 bit at a time.
- It can support up to 64K I/O ports.
- It provides 14, 16 -bit registers.
- Frequency range of 8086 is 6-10 MHz
- It has multiplexed address and data bus AD0- AD15 and A16 – A19.
- It requires single phase clock with 33% duty cycle to provide internal timing.
- It can prefetch upto 6 instruction bytes from memory and queues them in order to speed up instruction execution.
- It requires +5V power supply.

- A 40 pin dual in line package.
- 8086 is designed to operate in two modes, Minimum mode and Maximum mode.
    - The minimum mode is selected by applying logic 1 to the MN / MX# input pin. This is a single microprocessor configuration.
    - The maximum mode is selected by applying logic 0 to the MN / MX# input pin. This is a multi micro processors configuration.

**Register Organization of 8086**

**General purpose registers**

The 8086 microprocessor has a total of fourteen registers that are accessible to the programmer. It is divided into four groups. They are:

- Four General purpose registers
- Four Index/Pointer registers
- Four Segment registers
- Two  Other registers

**G**eneral purpose registers:

**General Purpose Registers**

| | | 15 0 | |
|---|---|---|---|
| Accumulator | AX | | Multiply, divide, I/O |
| Base | BX | | Pointer to base address (data) |
| Count | CX | | Count for loops, shifts |
| Data | DX | | Multiply, divide, I/O |

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

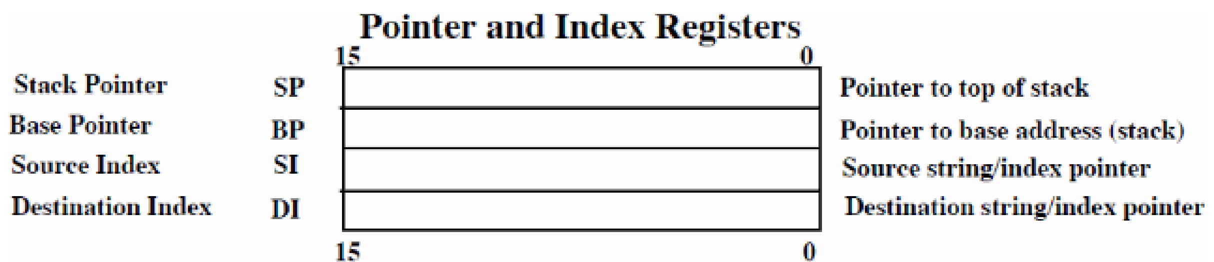
Base register consists of two 8-bit registers BL and BH, which can be combined together and used as a 16-bit register BX. BL in this case contains the low-order byte of the word, and BH contains the high-order byte. BX register usually contains a data pointer used for based, based indexed or register indirect addressing.

Count register consists of two 8-bit registers CL and CH, which can be combined together and used as a 16-bit register CX. When combined, CL register contains the low order byte of the word, and CH contains the high-order byte. Count register can be used in Loop, shift/rotate instructions and as a counter in string manipulation

Data register consists of two 8-bit registers DL and DH, which can be combined together and used as a 16-bit register DX. When combined, DL register contains the low order byte of the word, and DH contains the high-order byte. Data register can be used as a port number in I/O operations. In integer 32-bit multiply and divide instruction the DX register contains high-order word of the initial or resulting number.

**Index or Pointer Registers**

These registers can also be called as Special Purpose registers.




Stack Pointer (SP) is a 16-bit register pointing to program stack, i.e. it is used to hold the address of the top of stack. The stack is maintained as a LIFO with its bottom at the start of the stack segment (specified by the SS segment register).Unlike the SP register, the BP can be used to specify the offset of other program segments.

Base Pointer (BP) is a 16-bit register pointing to data in stack segment. It is usually used by subroutines to locate variables that were passed on the stack by a calling program. BP register is usually used for based, based indexed or register indirect addressing.

Source Index (SI) is a 16-bit register. SI is used for indexed, based indexed and register indirect addressing, as well as a source data address in string manipulation instructions. Used in conjunction with the DS register to point to data locations in the data segment.

Destination Index (DI) is a 16-bit register. Used in conjunction with the ES register in string operations. DI is used for indexed, based indexed and register indirect addressing, as well as a destination data address in string manipulation instructions. In short, Destination Index and SI Source Index registers are used to hold address.

## Segment Registers

Most of the registers contain data/instruction offsets within 64 KB memory segment. There are four different 64 KB segments for instructions, stack, data and extra data. To specify where in 1 MB of processor memory these 4 segments are located the processor uses four segment registers.

**Segment Registers**

| | | |
|---|---|---|
| Code Segment | CS | |
| Data Segment | DS | |
| Stack Segment | SS | |
| Extra Segment | ES | |

Code segment (CS) is a 16-bit register containing address of 64 KB segment with processor instructions. The processor uses CS segment for all accesses to instructions referenced by instruction pointer (IP) register. CS register cannot be changed directly. The CS register is automatically updated during far jump, far call and far return instructions.

Stack segment (SS) is a 16-bit register containing address of 64KB segment with program stack. By default, the processor assumes that all data referenced by the stack pointer (SP) and base pointer (BP) registers is located in the stack segment. SS register can be changed directly using POP instruction.

Data segment (DS) is a 16-bit register containing address of 64KB segment with program data. By default, the processor assumes that all data referenced by general registers (AX, BX, CX, DX) and index register (SI, DI) is located in the data segment. DS register can be changed directly using POP and LDS instructions.

Extra segment (ES) used to hold the starting address of Extra segment. Extra segment is provided for programs that need to access a second data segment. Segment registers cannot be used in arithmetic operations.

## Other registers of 8086

**Other Registers**

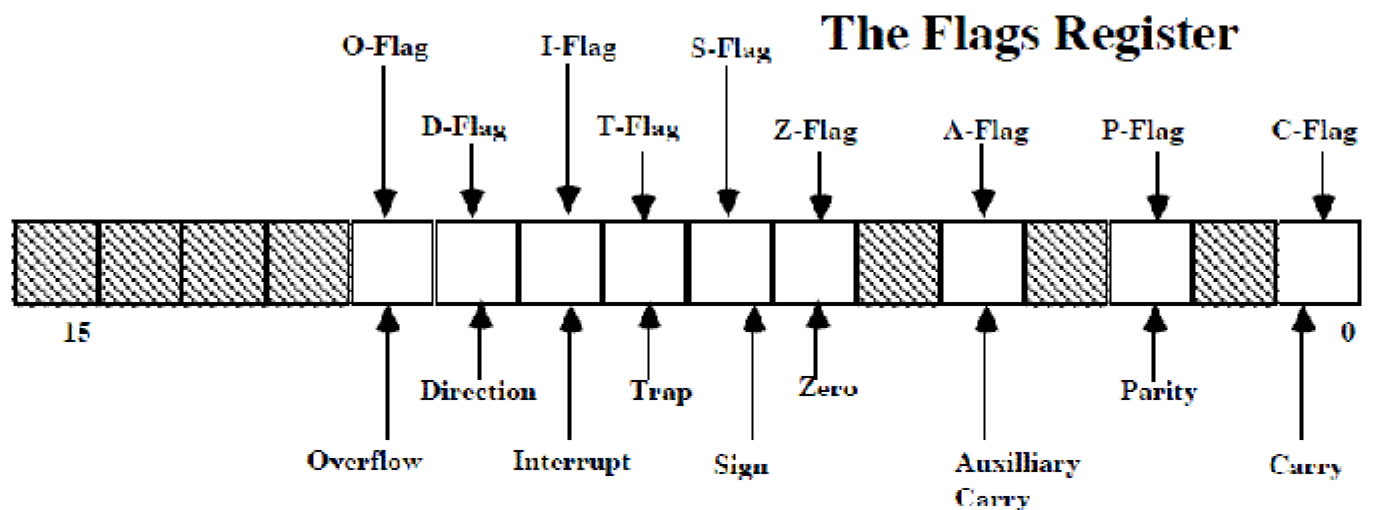
| | |
|---|---|
| Flags | Flags |
| Instruction Pointer | IP |

**Instruction Pointer (IP)** is a 16-bit register. This is a crucially important register which is used to control which instruction the CPU executes. The IP, or program counter, is used to store the memory location of the next instruction to be executed. The CPU checks the program counter to ascertain which instruction to carry out next. It then updates the program counter to point to the next instruction. Thus the program counter will always point to the next instruction to be executed.

**Flag Register** contains a group of status bits called flags that indicate the status of the CPU or the result of arithmetic operations. There are two types of flags:

1. The **status flags** which reflect the result of executing an instruction. The programmer cannot set/reset these flags directly.
2. The **control flags** enable or disable certain CPU operations. The programmer can set/reset these bits to control the CPU's operation.

Nine individual bits of the status register are used as control flags (3 of them) and status flags (6 of them).The remaining 7 are not used.
A flag can only take on the values 0 and 1. We say a flag is set if it has the value 1.The status flags are used to record specific characteristics of arithmetic and of logical instructions.




**Control Flags:** There are three control flags

1. **The Direction Flag (D):** Affects the direction of moving data blocks by such instructions as MOVS, CMPS and SCAS. The flag values are 0 = up and 1 = down and can be set/reset by the STD (set D) and CLD (clear D) instructions.

2. **The Interrupt Flag (I):** Dictates whether or not system interrupts can occur. Interrupts are actions initiated by hardware block such as input devices that will interrupt the normal execution

of programs. The flag values are 0 = disable interrupts or 1 = enable interrupts and can be manipulated by the CLI (clear I) and STI (set I) instructions.

3. **The Trap Flag (T):** Determines whether or not the CPU is halted after the execution of each instruction. When this flag is set (i.e. = 1), the programmer can single step through his program to debug any errors. When this flag = 0 this feature is off. This flag can be set by the INT 3 instruction.

**Status Flags:** There are six status flags

1. **The Carry Flag (C):** This flag is set when the result of an unsigned arithmetic operation is too large to fit in the destination register. This happens when there is an end carry in an addition operation or there an end borrows in a subtraction operation. A value of 1 = carry and 0 = no carry.

2. **The Overflow Flag (O):** This flag is set when the result of a signed arithmetic operation is too large to fit in the destination register (i.e. when an overflow occurs). Overflow can occur when adding two numbers with the same sign (i.e. both positive or both negative). A value of 1 = overflow and 0 = no overflow.

3. **The Sign Flag (S):** This flag is set when the result of an arithmetic or logic operation is negative. This flag is a copy of the MSB of the result (i.e. the sign bit). A value of 1 means negative and 0 = positive.

4. **The Zero Flag (Z):** This flag is set when the result of an arithmetic or logic operation is equal to zero. A value of 1 means the result is zero and a value of 0 means the result is not zero.

5. **The Auxiliary Carry Flag (A):** This flag is set when an operation causes a carry from bit 3 to bit 4 (or a borrow from bit 4 to bit 3) of an operand. A value of 1 = carry and 0 = no carry.

6. **The Parity Flag (P):** This flags reflects the number of 1s in the result of an operation. If the number of 1s is even its value = 1 and if the number of 1s is odd then its value = 0.

**<u>Architecture of 8086 or Functional Block diagram of 8086</u>**

- 8086 has two blocks Bus Interface Unit (BIU) and Execution Unit (EU).
- The BIU performs all bus operations such as instruction fetching, reading and writing operands for memory and calculating the addresses of the memory operands. The instruction bytes are transferred to the instruction queue.

- EU executes instructions from the instruction system byte queue.
- Both units operate asynchronously to give the 8086 an overlapping instruction fetch and execution mechanism which is called as Pipelining. This results in efficient use of the system bus and system performance.
- BIU contains Instruction queue, Segment registers, Instruction pointer, Address adder.
- EU contains Control circuitry, Instruction decoder, ALU, Pointer and Index register, Flag register.

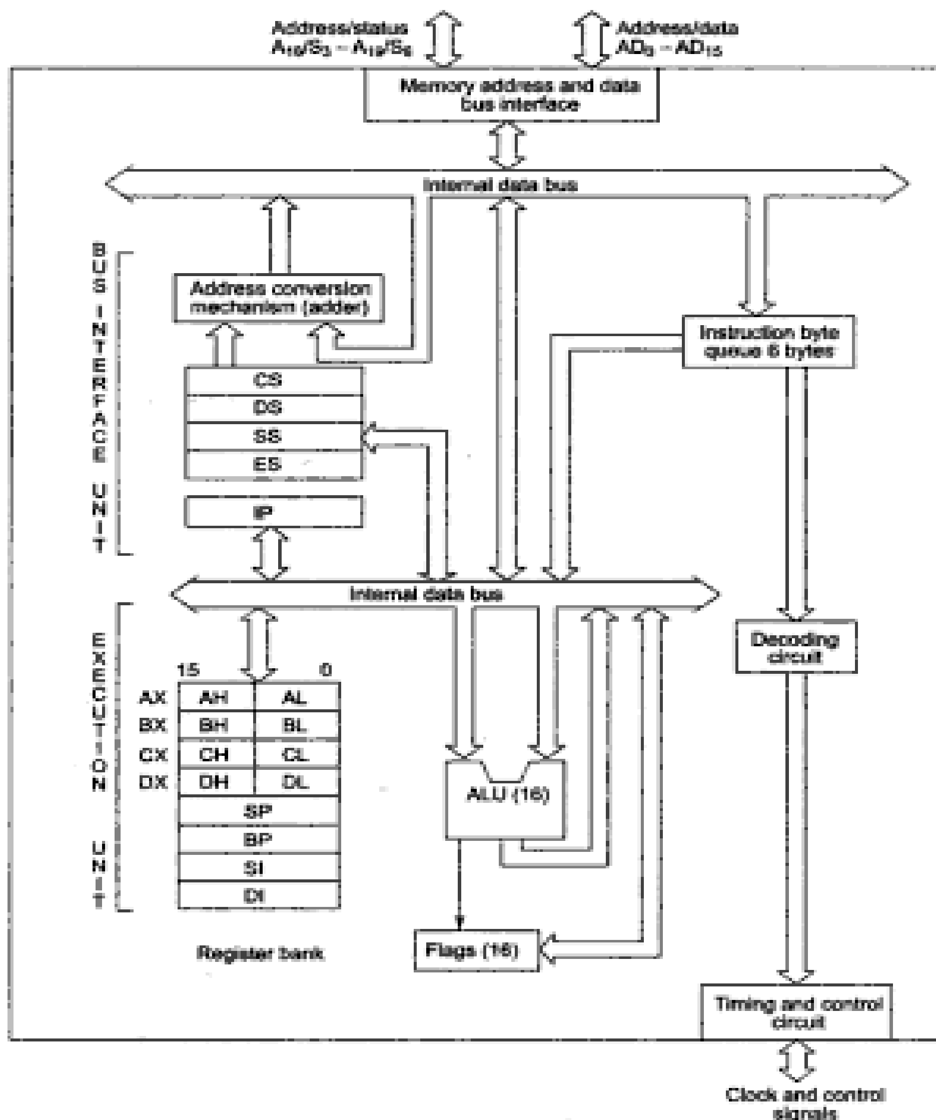



**Figure: 8086 Architecture**

## Explanation of Architecture of 8086

**Bus Interface Unit:**

- It provides a full 16 bit bidirectional data bus and 20 bit address bus.
- The bus interface unit is responsible for performing all external bus operations.
- Specifically it has the following functions:
- Instruction fetch Instruction queuing, Operand fetch and storage, Address relocation and Bus control.
- The BIU uses a mechanism known as an instruction stream queue to implement pipeline architecture.
- This queue permits prefetch of up to six bytes of instruction code. When ever the queue of the BIU is not full, it has room for at least two more bytes and at the same time the EU is not requesting it to read or write operands from memory, the BIU is free to look ahead in the program by prefetching the next sequential instruction.
- These prefetching instructions are held in its FIFO queue. With its 16 bit data bus, the BIU fetches two instruction bytes in a single memory cycle.
- After a byte is loaded at the input end of the queue, it automatically shifts up through the FIFO to the empty location nearest the output.
- The EU accesses the queue from the output end. It reads one instruction byte after the other from the output of the queue. If the queue is full and the EU is not requesting access to operand in memory.
- These intervals of no bus activity, which may occur between bus cycles are known as Idle state.
- If the BIU is already in the process of fetching an instruction when the EU request it to read or write operands from memory or I/O, the BIU first completes the instruction fetch bus cycle before initiating the operand read / write cycle.
- The BIU also contains a dedicated adder which is used to generate the 20bit physical address that is output on the address bus. This address is formed by adding an appended 16 bit segment address and a 16 bit offset address.

- For example: The physical address of the next instruction to be fetched is formed by combining the current contents of the code segment CS register and the current contents of the instruction pointer IP register.
- The BIU is also responsible for generating bus control signals such as those for memory read or write and I/O read or write.

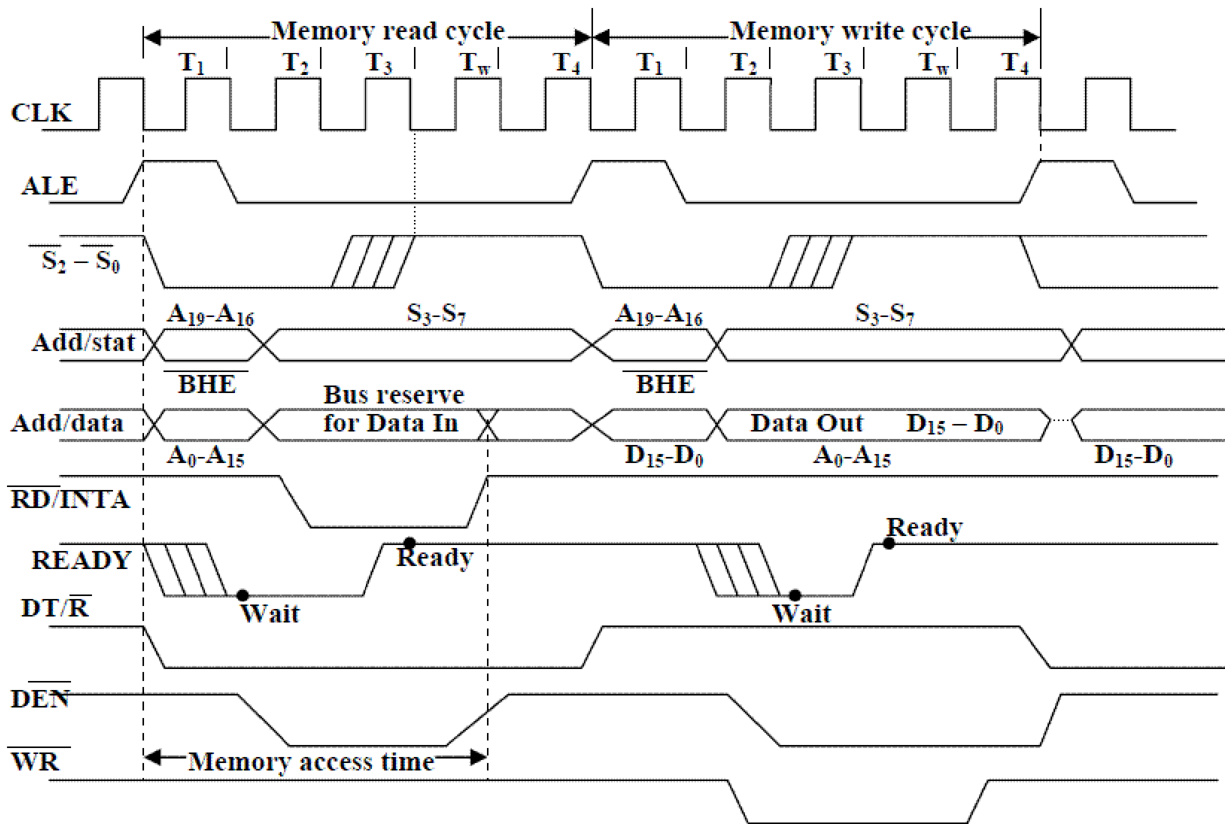
**Execution Unit**

- The Execution unit is responsible for decoding and executing all instructions.
- The EU extracts instructions from the top of the queue in the BIU, decodes them, generates operands if necessary, passes them to the BIU and requests it to perform the read or write bus cycles to memory or I/O and perform the operation specified by the instruction on the operands.
- During the execution of the instruction, the EU tests the status and control flags and updates them based on the results of executing the instruction.
- If the queue is empty, the EU waits for the next instruction byte to be fetched and shifted to top of the queue.
- When the EU executes a branch or jump instruction, it transfers control to a location corresponding to another set of sequential instructions.
- Whenever this happens, the BIU automatically resets the queue and then begins to fetch instructions from this new location to refill the queue.

**General Bus Operation**

- The 8086 has a combined address and data bus commonly referred as a time multiplexed address and data bus.
- The main reason behind multiplexing address and data over the same pins is the maximum utilization of processor pins and it facilitates the use of 40 pin standard DIP package.
- The bus can be demultiplexed using a few latches and transceivers, when ever required.
- Basically, all the processor bus cycles consist of at least four clock cycles. These are referred to as T1, T2, T3, and T4. The address is transmitted by the processor during T1. It is present on the bus only for one cycle.

- The negative edge of this ALE pulse is used to separate the address and the data or status information. In maximum mode, the status lines S0, S1 and S2 are used to indicate the type of operation.
- Status bits S3 to S7 are multiplexed with higher order address bits and the BHE signal. Address is valid during T1 while status bits S3 to S7 are valid during T2 through T4.




**Maximum mode**

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

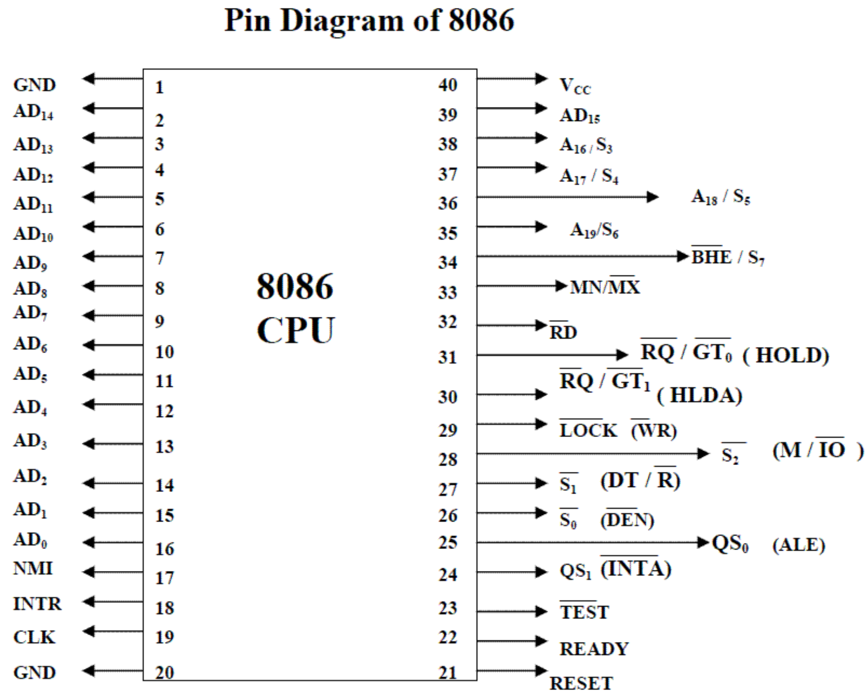
**Minimum mode**

- In a minimum mode 8086 system, the microprocessor 8086 is operated in minimum mode by strapping its MN/MX pin to logic 1.
- In this mode, all the control signals are given out by the microprocessor chip itself.

- There is a single microprocessor in the minimum mode system.

**<u>Pin Diagram of 8086 and Pin description of 8086</u>**

Figure shows the Pin diagram of 8086. The description follows it.




- The Microprocessor 8086 is a 16-bit CPU available in different clock rates and packaged in a 40 pin CERDIP or plastic package.

- The 8086 operates in single processor or multiprocessor configuration to achieve high performance. The pins serve a particular function in minimum mode (single processor mode) and other function in maximum mode configuration (multiprocessor mode).

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

- The following signal descriptions are common for both modes.

- **AD15-AD0:** These are the time multiplexed memory I/O address and data lines.

- Address remains on the lines during T1 state, while the data is available on the data bus during T2, T3, Tw and T4. These lines are active high and float to a tristate during interrupt acknowledge and local bus hold acknowledge cycles.

- **A19/S6, A18/S5, A17/S4, and A16/S3:** These are the time multiplexed address and status lines.
  - During T1 these are the most significant address lines for memory operations.
  - During I/O operations, these lines are low.
  - During memory or I/O operations, status information is available on those lines for T2, T3, Tw and T4.
  - The status of the interrupt enable flag bit is updated at the beginning of each clock cycle.
  - The S4 and S3 combine indicate which segment registers is presently being used for memory accesses as in below fig.
  - These lines float to tri-state off during the local bus hold acknowledge. The status line S6 is always low.
  - The address bit is separated from the status bit using latches controlled by the ALE signal.

| $S_4$ | $S_3$ | Indication |
|---|---|---|
| 0 | 0 | Alternate Data |
| 0 | 1 | Stack |
| 1 | 0 | Code or None |
| 1 | 1 | Data |
| 0 | 0 | Whole word |
| 0 | 1 | Upper byte from or to even address |
| 1 | 0 | Lower byte from or to even address |

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

- **RD – Read:** This signal on low indicates the peripheral that the processor is performing memory or I/O read operation. RD is active low and shows the state for T2, T3, Tw of any read cycle. The signal remains tristated during the hold acknowledge.

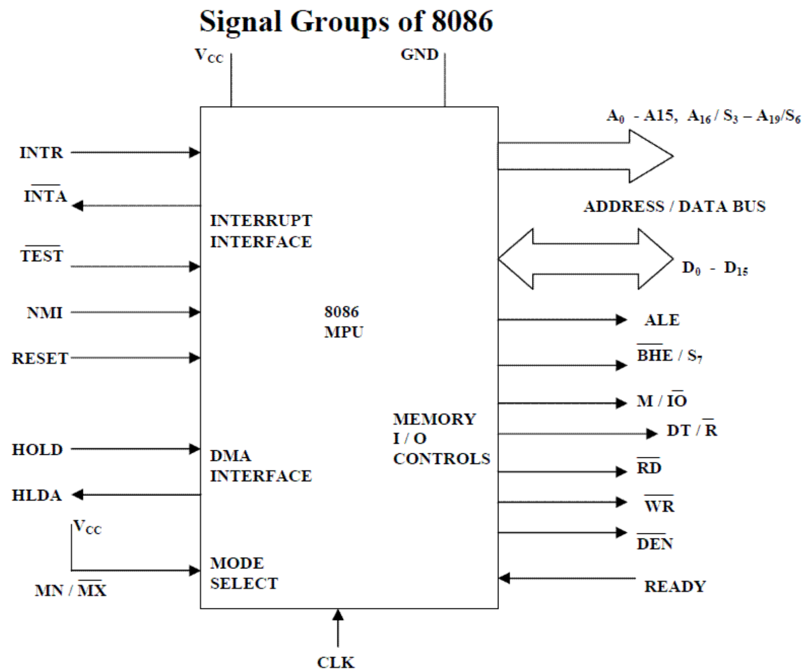
- **READY:** This is the acknowledgement from the slow device or memory that they have completed the data transfer. The signal made available by the devices is synchronized by the 8284A clock generator to provide ready input to the 8086. the signal is active high.

- **INTR-Interrupt Request:** This is a triggered input. This is sampled during the last clock cycles of each instruction to determine the availability of the request. If any interrupt request is pending, the processor enters the interrupt acknowledge cycle. This can be internally masked by resulting the interrupt enable flag. This signal is active high and internally synchronized.

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

- **CLK- Clock Input:** The clock input provides the basic timing for processor operation and bus control activity. It's an asymmetric square wave with 33% duty cycle.

Figure shows the Pin functions of 8086.




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

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

- **INTA – Interrupt Acknowledge:** This signal is used as a read strobe for interrupt acknowledge cycles. i.e. when it goes low, the processor has accepted the interrupt.

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

- **DT/R – Data Transmit/Receive:** This output is used to decide the direction of data flow through the transceivers (bidirectional buffers). When the processor sends out data, this signal is high and when the processor is receiving data, this signal is low.

- **DEN – Data Enable:** This signal indicates the availability of valid data over the address/data lines. It is used to enable the transceivers (bidirectional buffers) to separate the data from the multiplexed address/data signal. It is active from the middle of T2 until the middle of T4. This is tristated during 'hold acknowledge' cycle.

- **HOLD, HLDA- Acknowledge:** When the HOLD line goes high; it indicates to the processor that another master is requesting the bus access. The processor, after receiving

the HOLD request, issues the hold acknowledge signal on HLDA pin, in the middle of the next clock cycle after completing the current bus cycle.

- At the same time, the processor floats the local bus and control lines. When the processor detects the HOLD line low, it lowers the HLDA signal. HOLD is an asynchronous input, and is should be externally synchronized. If the DMA request is made while the CPU is performing a memory or I/O cycle, it will release the local bus during T4 provided :

1. The request occurs on or before T2 state of the current cycle.
2. The current cycle is not operating over the lower byte of a word.
3. The current cycle is not the first acknowledge of an interrupt acknowledge sequence.
4. A Lock instruction is not being executed.

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

- **S2, S1, and S0 – Status Lines:** These are the status lines which reflect the type of operation, being carried out by the processor. These become activity during T4 of the previous cycle and active during T1 and T2 of the current bus cycles.

- **LOCK:** This output pin indicates that other system bus master will be prevented from gaining the system bus, while the LOCK signal is low. The LOCK signal is activated by the 'LOCK' prefix instruction and remains active until the completion of the next instruction. When the CPU is executing a critical instruction which requires the system bus, the LOCK prefix instruction ensures that other processors connected in the system will not gain the control of the bus.

The 8086, while executing the prefixed instruction, asserts the bus lock signal output, which may be connected to an external bus controller. By prefetching the instruction, there is a considerable speeding up in instruction execution in 8086. This is known as **instruction pipelining**.

| S2 | S1 | S0 | 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 |

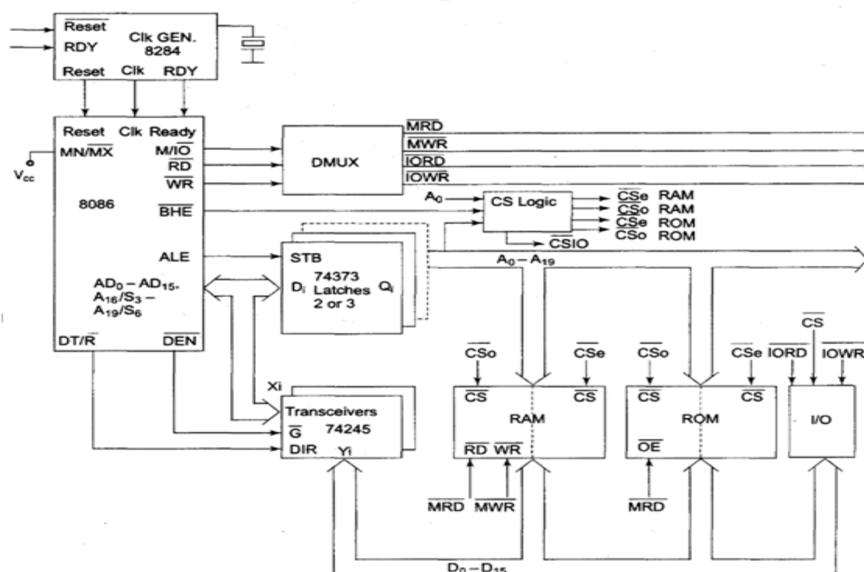
- At the starting the CS: IP is loaded with the required address from which the execution is to be started. Initially, the queue will be empty and the microprocessor starts a fetch operation to bring one byte (the first byte) of instruction code, if the CS: IP address is odd or two bytes at a time, if the CS: IP address is even.
- The first byte is a complete opcode in case of some instruction (one byte opcode instruction) and is a part of opcode, in case of some instructions (two byte opcode instructions), the remaining part of code lie in second byte.
- The second byte is then decoded in continuation with the first byte to decide the instruction length and the number of subsequent bytes to be treated as instruction data. The queue is updated after every byte is read from the queue but the fetch cycle is initiated by BIU only if at least two bytes of the queue are empty and the EU may be concurrently executing the fetched instructions.
- The next byte after the instruction is completed is again the first opcode byte of the next instruction. A similar procedure is repeated till the complete execution of the program. The fetch operation of the next instruction is overlapped with the execution of the current instruction. As in the architecture, there are two separate units, namely Execution unit and Bus interface unit.
- While the execution unit is busy in executing an instruction, after it is completely decoded, the bus interface unit may be fetching the bytes of the next instruction from memory, depending upon the queue status.

| QS1 | QS0 | Indication |
|---|---|---|
| 0 | 0 | No Operation |
| 0 | 1 | First Byte of the opcode from the queue |
| 1 | 0 | Empty Queue |
| 1 | 1 | Subsequent Byte from the Queue |

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

    1. A pulse of one clock wide from another bus master requests the bus access to 8086.
    2. During T4(current) or T1(next) clock cycle, a pulse one clock wide from 8086 to the requesting master, indicates that the 8086 has allowed the local bus to float and that it will enter the 'hold acknowledge' state at next cycle. The CPU bus interface unit is likely to be disconnected from the local bus of the system.
    3. A one clock wide pulse from another master indicates to the 8086 that the hold request is about to end and the 8086 may regain control of the local bus at the next clock cycle. Thus each master to master exchange of the local bus is a sequence of 3 pulses. There must be at least one dead clock cycle after each bus exchange. The request and grant pulses are active low. For the bus request those are received while 8086 is performing memory or I/O cycle, the granting of the bus is governed by the rules as in case of HOLD and HLDA in minimum mode.

## Minimum Mode 8086 System



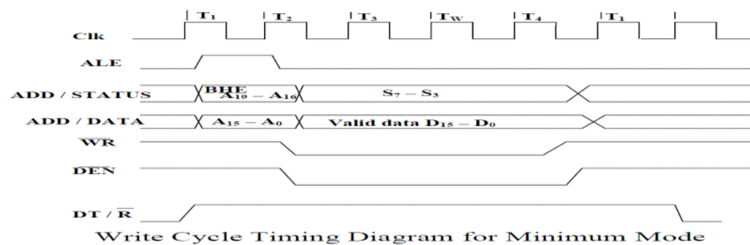

**Minimum mode 8086 system**

- In a minimum mode 8086 system, the microprocessor 8086 is operated in minimum mode by strapping its MN/MX pin to logic 1.
- In this mode, all the control signals are given out by the microprocessor chip itself. There is a single microprocessor in the minimum mode system.
- The remaining components in the system are latches, transceivers, clock generator, memory and I/O devices. Some type of chip selection logic may be required for selecting memory or I/O devices, depending upon the address map of the system.
- Latches are generally buffered output D-type flip-flops like 74LS373 or 8282. They are used for separating the valid address from the multiplexed address/data signals and are controlled by the ALE signal generated by 8086.
- Transceivers are the bidirectional buffers and some times they are called as data amplifiers. They are required to separate the valid data from the time multiplexed address/data signals.
- They are controlled by two signals namely, DEN and DT/R.
- The DEN signal indicates the direction of data, i.e. from or to the processor. The system contains memory for the monitor and users program storage.

- Usually, EPROM is used for monitor storage, while RAM for users program storage. A system may contain I/O devices.

**Write Cycle Timing Diagram for Minimum Mode**

- The working of the minimum mode configuration system can be better described in terms of the timing diagrams rather than qualitatively describing the operations.
- The opcode fetch and read cycles are similar. Hence the timing diagram can be categorized in two parts, the first is the timing diagram for read cycle and the second is the timing diagram for write cycle.
- The read cycle begins in T1 with the assertion of address latch enable (ALE) signal and also M / IO signal. During the negative going edge of this signal, the valid address is latched on the local bus.
- The BHE and A0 signals address low, high or both bytes. From T1 to T4 , the M/IO signal indicates a memory or I/O operation.
- At T2, the address is removed from the local bus and is sent to the output. The bus is then tristated. The read (RD) control signal is also activated in T2.
- The read (RD) signal causes the address device to enable its data bus drivers. After RD goes low, the valid data is available on the data bus.

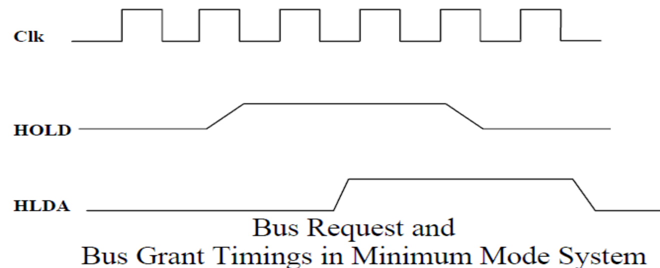


Write Cycle Timing Diagram for Minimum Mode

- The addressed device will drive the READY line high. When the processor returns the read signal to high level, the addressed device will again tristate its bus drivers.
- A write cycle also begins with the assertion of ALE and the emission of the address. The M/IO signal is again asserted to indicate a memory or I/O operation. In T2, after sending the address in T1, the processor sends the data to be written to the addressed location.
- The data remains on the bus until middle of T4 state. The WR becomes active at the beginning of T2 (unlike RD is somewhat delayed in T2 to provide time for floating).

- The BHE and A0 signals are used to select the proper byte or bytes of memory or I/O word to be read or write.
- The M/IO, RD and WR signals indicate the type of data transfer as specified in table below.

**Bus Request and Bus Grant Timings in Minimum Mode System of 8086**

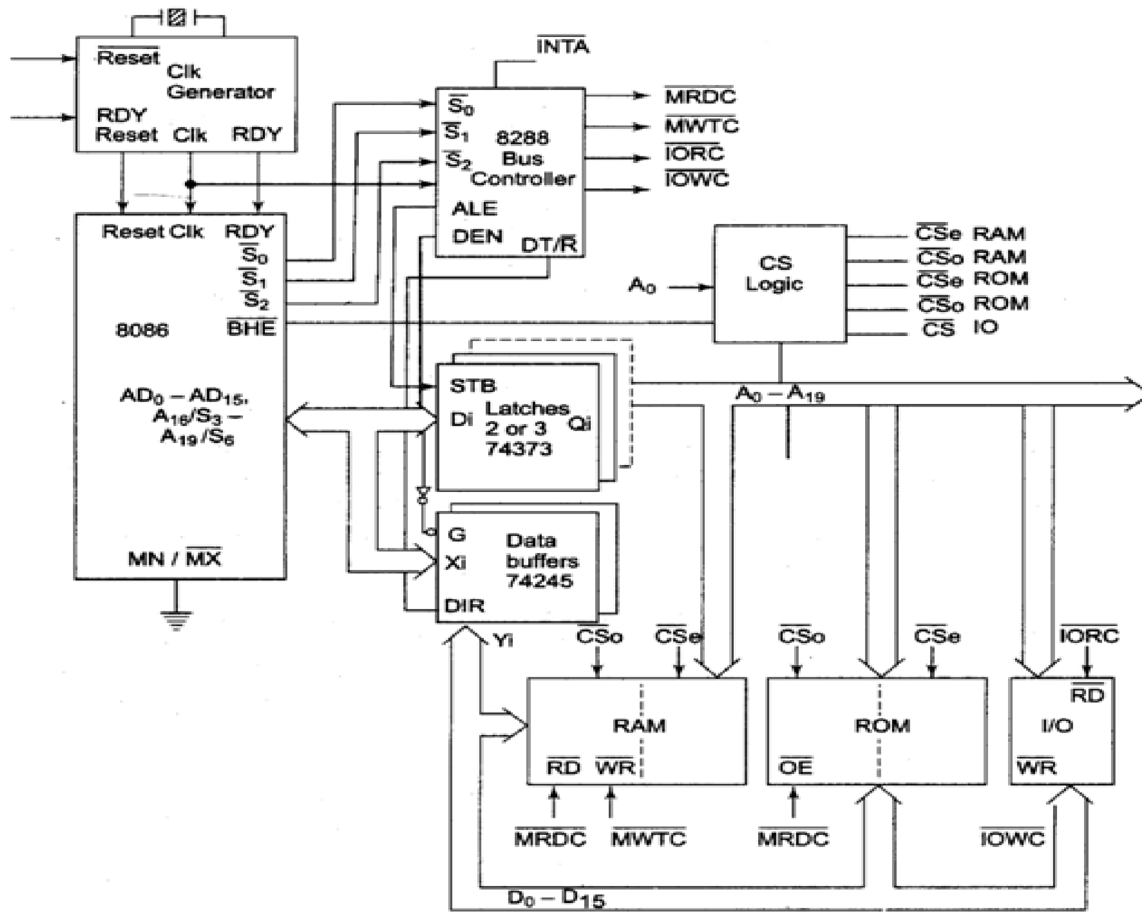


Bus Request and
Bus Grant Timings in Minimum Mode System

- Hold Response sequence: The HOLD pin is checked at leading edge of each clock pulse. If it is received active by the processor before T4 of the previous cycle or during T1 state of the current cycle, the CPU activates HLDA in the next clock cycle and for succeeding bus cycles, the bus will be given to another requesting master.
- The control of the bus is not regained by the processor until the requesting master does not drop the HOLD pin low. When the request is dropped by the requesting master, the HLDA is dropped by the processor at the trailing edge of the next clock.

**Maximum Mode 8086 System**

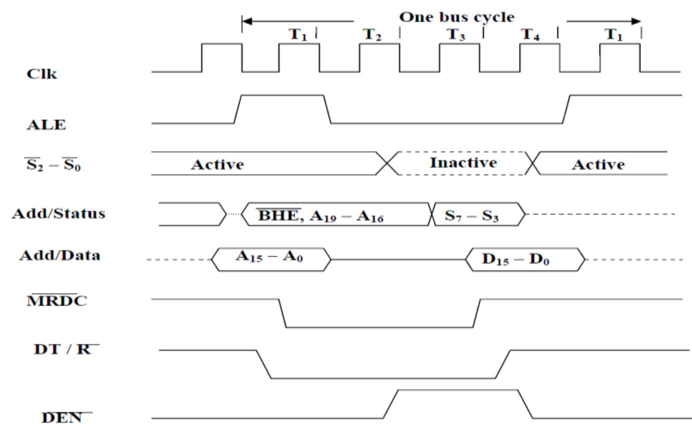
- In the maximum mode, the 8086 is operated by strapping the MN/MX pin to ground.
- In this mode, the processor derives the status signal S2, S1, S0. Another chip called bus controller derives the control signal using this status information.
- In the maximum mode, there may be more than one microprocessor in the system configuration.
- The components in the system are same as in the minimum mode system.
- The basic function of the bus controller chip IC8288 is to derive control signals like RD and WR (for memory and I/O devices), DEN, DT/R, ALE etc. using the information by the processor on the status lines.
- The bus controller chip has input lines S2, S1, S0 and CLK. These inputs to 8288 are driven by CPU.




- It derives the outputs ALE, DEN, DT/R, MRDC, MWTC, AMWC, IORC, IOWC and AIOWC. The AEN, IOB and CEN pins are especially useful for multiprocessor systems.
- AEN and IOB are generally grounded. CEN pin is usually tied to +5V. The significance of the MCE/PDEN output depends upon the status of the IOB pin.
- If IOB is grounded, it acts as master cascade enable to control cascade 8259A, else it acts as peripheral data enable used in the multiple bus configurations.
- INTA pin used to issue two interrupt acknowledge pulses to the interrupt controller or to an interrupting device.
- IORC, IOWC are I/O read command and I/O write command signals respectively.
- These signals enable an IO interface to read or write the data from or to the address port.
- The MRDC, MWTC are memory read command and memory write command signals respectively and may be used as memory read or write signals.
- All these command signals instructs the memory to accept or send data from or to the bus.

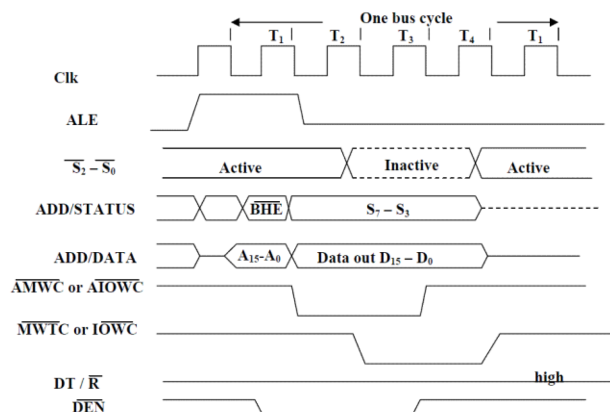
- For both of these write command signals, the advanced signals namely AIOWC and AMWTC are available.
- Here the only difference between in timing diagram between minimum mode and maximum mode is the status signals used and the available control and advanced command signals.
- R0, S1, S2 are set at the beginning of bus cycle.8288 bus controller will output a pulse as on the ALE and apply a required signal to its DT / R pin during T1.
- In T2, 8288 will set DEN=1 thus enabling transceivers, and for an input it will activate MRDC or IORC. These signals are activated until T4. For an output, the AMWC or AIOWC is activated from T2 to T4 and MWTC or IOWC is activated from T3 to T4.
- The status bit S0 to S2 remains active until T3 and become passive during T3 and T4.
- If reader input is not activated before T3, wait state will be inserted between T3 and T4.

## Memory Read Timing Diagram in Maximum Mode of 8086



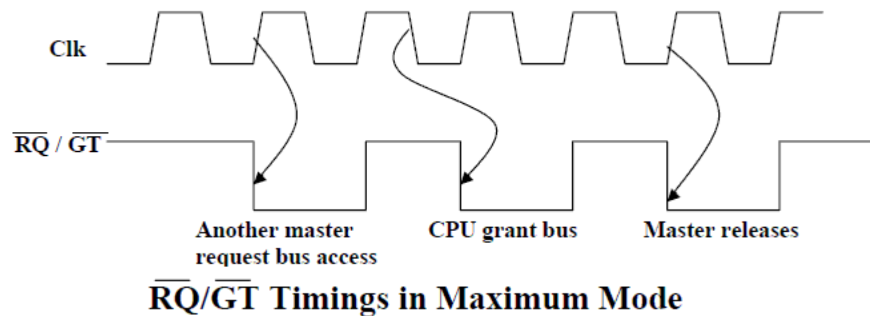
Memory Read Timing in Maximum Mode

## Memory Write Timing in Maximum mode of 8086



Memory Write Timing in Maximum mode.

## RQ/GT Timings in Maximum Mode




RQ/GT Timings in Maximum Mode

- The request/grant response sequence contains a series of three pulses. The request/grant pins are checked at each rising pulse of clock input.
- When a request is detected and if the condition for HOLD request is satisfied, the processor issues a grant pulse over the RQ/GT pin immediately during T4 (current) or T1 (next) state.
- When the requesting master receives this pulse, it accepts the control of the bus, it sends a release pulse to the processor using RQ/GT pin.

## Minimum Mode Interface

- When the Minimum mode operation is selected, the 8086 provides all control signals needed to implement the memory and I/O interface.
- The minimum mode signal can be divided into the following basic groups :

1. **Address/data bus**
2. **Status**
3. **Control**
4. **Interrupt and**
5. **DMA**.

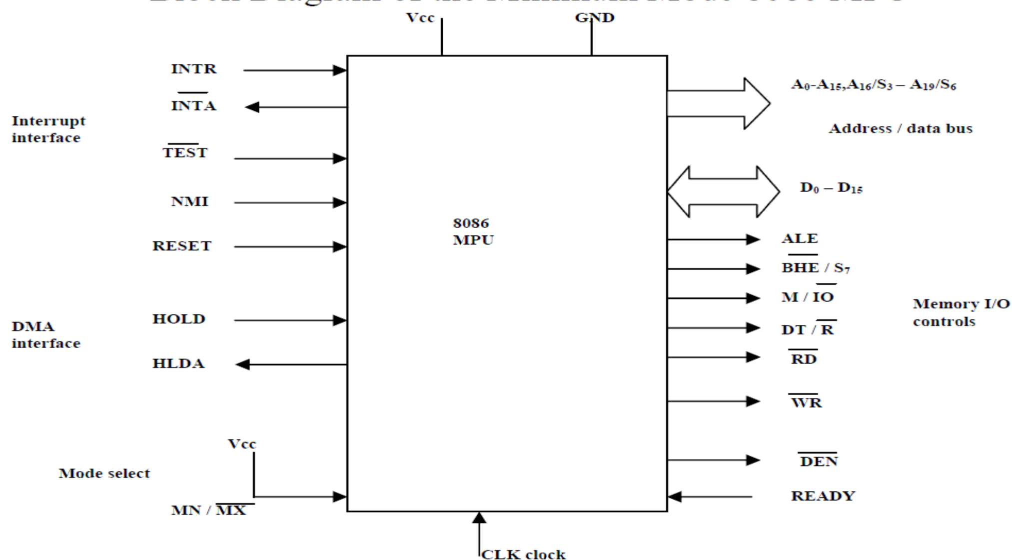
Each and every group is explained clearly.

**Address/Data Bus:**

- These lines serve two functions. As an address bus is 20 bits long and consists of signal lines A0 through A19. A19 represents the MSB and A0 LSB. A 20bit address gives the 8086 a 1Mbyte memory address space. More over it has an independent I/O address space which is 64K bytes in length.
- The 16 data bus lines D0 through D15 are actually multiplexed with address lines A0 through A15 respectively. By multiplexed we mean that the bus work as an address bus during first machine cycle and as a data bus during next machine cycles.
- D15 is the MSB and D0 LSB. When acting as a data bus, they carry read/write data for memory, input/output data for I/O devices, and interrupt type codes from an interrupt controller.

**Status signal:**

- The four most significant address lines A19 through A16 are also multiplexed but in this case with status signals S6 through S3.
- These status bits are output on the bus at the same time that data are transferred over the other bus lines.




- Bit S4 and S3 together from a 2 bit binary code that identifies which of the 8086 internal segment registers is used to generate the physical address that was output on the address bus during the current bus cycle. Code S4S3 = 00 identifies a register known as extra segment register as the source of the segment address.

- Status line S5 reflects the status of another internal characteristic of the 8086. It is the logic level of the internal enable flag. The last status bit S6 is always at the logic 0 level.

| S4 | S3 | Segment Register |
|---|---|---|
| 0 | 0 | Extra |
| 0 | 1 | Stack |
| 1 | 0 | Code / none |
| 1 | 1 | Data |

Memory segment status codes

**Control Signals:**

- The control signals are provided to support the 8086 memory I/O interfaces. They control functions such as when the bus is to carry a valid address in which direction data are to be transferred over the bus, when valid write data are on the bus and when to put read data on the system bus.

- ALE is a pulse to logic 1 that signals external circuitry when a valid address word is on the bus. This address must be latched in external circuitry on the 1-to-0 edge of the pulse at ALE.

- Another control signal that is produced during the bus cycle is BHE bank high enable. Logic 0 on this used as a memory enable signal for the most significant byte half of the data bus D8 through D1. These lines also serve a second function, which is as the S7 status line.

- Using the M/IO and DT/R lines, the 8086 signals which type of bus cycle is in progress and in which direction data are to be transferred over the bus. The logic level of M/IO tells external circuitry whether a memory or I/O transfer is taking place over the bus. Logic 1 at this output signals a memory operation and logic 0 an I/O operation.

- The direction of data transfer over the bus is signaled by the logic level output at DT/R. When this line is logic 1 during the data transfer part of a bus cycle, the bus is in the transmit mode. Therefore, data are either written into memory or output to an I/O device. On the other hand, logic 0 at DT/R signals that the bus is in the receive mode. This corresponds to reading data from memory or input of data from an input port.

- The signals read RD and write WR indicates that a read bus cycle or a write bus cycle is in progress. The 8086 switches WR to logic 0 to signal external device that valid write or output data are on the bus.

- On the other hand, RD indicates that the 8086 is performing a read of data of the bus. During read operations, one other control signal is also supplied. This is DEN (data enable) and it signals external devices when they should put data on the bus. There is one other control signal that is involved with the memory and I/O interface. This is the READY signal.
- READY signal is used to insert wait states into the bus cycle such that it is extended by a number of clock periods. This signal is provided by an external clock generator device and can be supplied by the memory or I/O sub-system to signal the 8086 when they are ready to permit the data transfer to be completed.

**Interrupt signals:**

- The key interrupt interface signals are interrupt request (INTR) and interrupt acknowledge (INTA).
- INTR is an input to the 8086 that can be used by an external device to signal that it need to be serviced.
- Logic 1 at INTR represents an active interrupt request. When an interrupt request has been recognized by the 8086, it indicates this fact to external circuit with pulse to logic 0 at the INTA output.
- The TEST input is also related to the external interrupt interface. Execution of a WAIT instruction causes the 8086 to check the logic level at the TEST input.
- If the logic 1 is found, the MPU suspend operation and goes into the idle state. The 8086 no longer executes instructions; instead it repeatedly checks the logic level of the TEST input waiting for its transition back to logic 0.
- As TEST switches to 0, execution resume with the next instruction in the program. This feature can be used to synchronize the operation of the 8086 to an event in external hardware.
- There are two more inputs in the interrupt interface: the nonmaskable interrupt NMI and the reset interrupt RESET.
- On the 0-to-1 transition of NMI control is passed to a nonmaskable interrupt service routine. The RESET input is used to provide a hardware reset for the 8086. Switching RESET to logic 0 initializes the internal register of the 8086 and initiates a reset service routine.

**DMA Interface signals:**

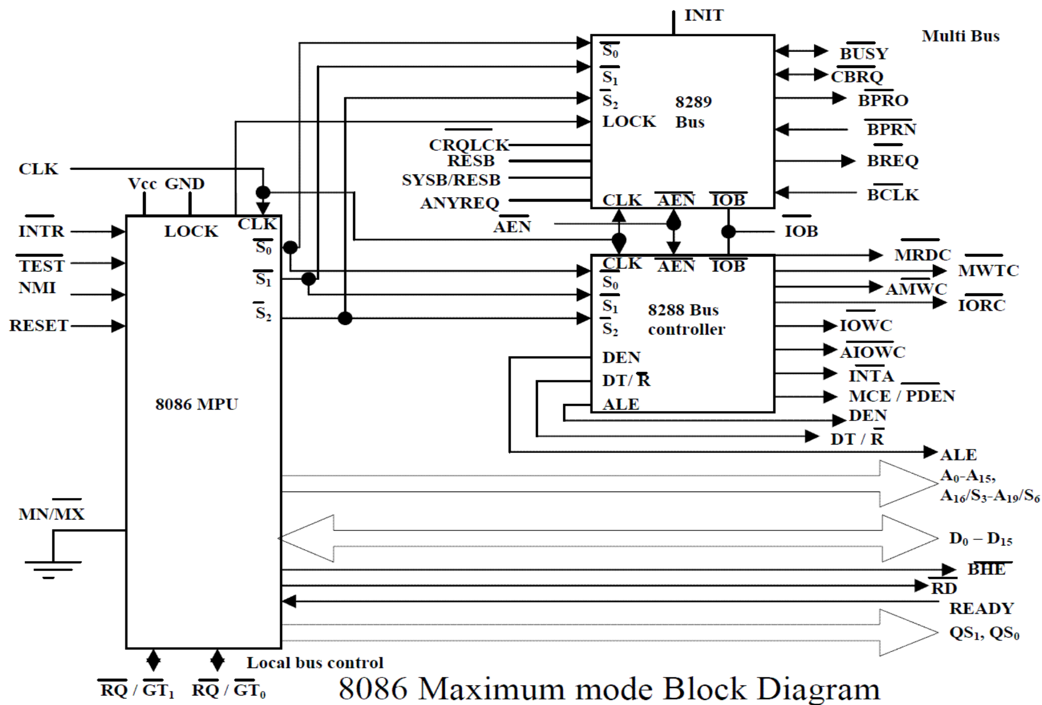
- The direct memory access DMA interface of the 8086 minimum mode consist of the HOLD and HLDA signals.
- When an external device wants to take control of the system bus, it signals to the 8086 by switching HOLD to the logic 1 level. At the completion of the current bus cycle, the 8086 enters the hold state. In the hold state, signal lines AD0 through AD15, A16/S3 through A19/S6, BHE, M/IO, DT/R, RD, WR, DEN and INTR are all in the high Z state.
- The 8086 signals external device that it is in this state by switching its HLDA output to logic 1 level.

**Maximum Mode Interface**

- When the 8086 is set for the maximum-mode configuration, it provides signals for implementing a multiprocessor / coprocessor system environment.
- By multiprocessor environment we mean that one microprocessor exists in the system and that each processor is executing its own program.
- Usually in this type of system environment, there are some system resources that are common to all processors. They are called as global resources. There are also other resources that are assigned to specific processors. These are known as local or private resources.
- Coprocessor also means that there is a second processor in the system. In these two processors does not access the bus at the same time. One passes the control of the system bus to the other and then may suspend its operation.
- In the maximum-mode 8086 system, facilities are provided for implementing allocation of global resources and passing bus control to other microprocessor or coprocessor.

**8288 Bus Controller – Bus Command and Control Signals:**

- 8086 does not directly provide all the signals that are required to control the memory, I/O and interrupt interfaces.



8086 Maximum mode Block Diagram

- Specially the WR, M/IO, DT/R, DEN, ALE and INTA, signals are no longer produced by the 8086. Instead it outputs three status signals S0, S1, S2 prior to the initiation of each bus cycle. This 3- bit bus status code identifies which type of bus cycle is to follow.
- S2S1S0 are input to the external bus controller device, the bus controller generates the appropriately timed command and control signals.
- The 8288 produces one or two of these eight command signals for each bus cycles. For instance, when the 8086 outputs the code S2S1S0 equals 001; it indicates that an I/O read cycle is to be performed.
- In the code 111 is output by the 8086, it is signaling that no bus activity is to take place.
- The control outputs produced by the 8288 are DEN, DT/R and ALE. These 3 signals provide the same functions as those described for the minimum system mode.

| $\overline{S2}$ | $\overline{S1}$ | $\overline{S0}$ | Indication | 8288 Command |
|---|---|---|---|---|
| 0 | 0 | 0 | Interrupt Acknowledge | $\overline{INTA}$ |
| 0 | 0 | 1 | Read I/O port | $\overline{IORC}$ |
| 0 | 1 | 0 | Write I/O port | $\overline{IOWC, AIOWC}$ |
| 0 | 1 | 1 | Halt | None |
| 1 | 0 | 0 | Instruction Fetch | $\overline{MRDC}$ |
| 1 | 0 | 1 | Read Memory | $\overline{MRDC}$ |
| 1 | 1 | 0 | Write Memory | $\overline{MWTC, AMWC}$ |
| 1 | 1 | 1 | Passive | None |

- This set of bus commands and control signals is compatible with the Multibus and industry standard for interfacing microprocessor systems.
- The output of 8289 are bus arbitration signals:

Bus busy (BUSY), common bus request (CBRQ), bus priority out (BPRO), bus priority in (BPRN), bus request (BREQ) and bus clock (BCLK).

- They correspond to the bus exchange signals of the Multibus and are used to lock other processor off the system bus during the execution of an instruction by the 8086.
- In this way the processor can be assured of uninterrupted access to common system resources such as global memory.
- Queue Status Signals: Two new signals that are produced by the 8086 in the maximum-mode system are queue status outputs QS0 and QS1. Together they form a 2-bit queue status code, QS1QS0.
- Following table shows the four different queue status.
- Local Bus Control Signal – Request / Grant Signals: In a maximum mode configuration, the minimum mode HOLD, HLDA interface is also changed

| QS1 | QS0 | Queue Status |
|---|---|---|
| 0 (low) | 0 | Queue Empty. The queue has been reinitialized as a result of the execution of a transfer instruction. |
| 0 | 1 | First Byte. The byte taken from the queue was the first byte of the instruction. |
| 1 | 0 | Queue Empty. The queue has been reinitialized as a result of the execution of a transfer instruction. |
| 1 | 1 | Subsequent Byte. The byte taken from the queue was a subsequent byte of the instruction. |

Table - Queue status codes

- . These two are replaced by request/grant lines RQ/ GT0 and RQ/ GT1, respectively. They provide a prioritized bus access mechanism for accessing the local bus.

**Interrupts**

**Definition:** The meaning of 'interrupts' is to break the sequence of operation. While the CPU is executing a program, on 'interrupt' breaks the normal sequence of execution of instructions, diverts its execution to some other program called Interrupt Service Routine (ISR).After executing ISR , the control is transferred back again to the main program. Interrupt processing is an alternative to polling.

**Need for Interrupt**: Interrupts are particularly useful when interfacing I/O devices that provide or require data at relatively low data transfer rate.

**Types of Interrupts:** There are two types of Interrupts in 8086. They are:

(i)Hardware Interrupts and

(ii)Software Interrupts

(i) **Hardware Interrupts** (External Interrupts). The Intel microprocessors support hardware interrupts through:

- Two pins that allow interrupt requests, INTR and NMI
- One pin that acknowledges, INTA, the interrupt requested on INTR.

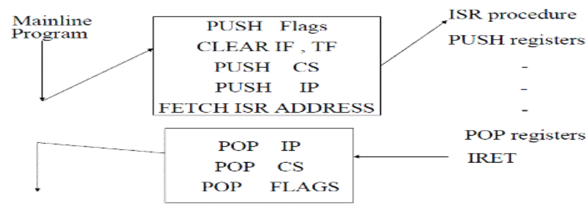
INTR and NMI

- INTR is a maskable hardware interrupt. The interrupt can be enabled/disabled using STI/CLI instructions or using more complicated method of updating the FLAGS register with the help of the POPF instruction.
- When an interrupt occurs, the processor stores FLAGS register into stack, disables further interrupts, fetches from the bus one byte representing interrupt type, and jumps to interrupt processing routine address of which is stored in location 4 * <interrupt type>. Interrupt processing routine should return with the IRET instruction.
- NMI is a non-maskable interrupt. Interrupt is processed in the same way as the INTR interrupt. Interrupt type of the NMI is 2, i.e. the address of the NMI processing routine is stored in location 0008h. This interrupt has higher priority than the maskable interrupt.
- – Ex: NMI, INTR.

(ii) **Software Interrupts** (Internal Interrupts and Instructions) .Software interrupts can be caused by:

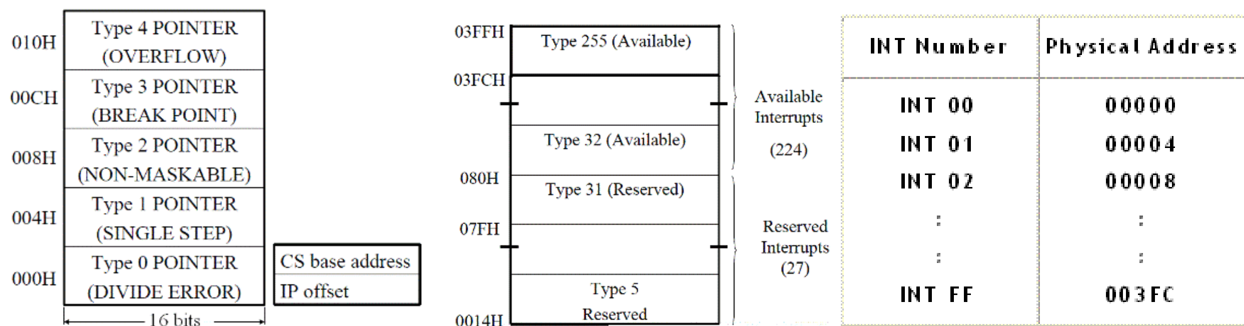
- INT instruction - breakpoint interrupt. This is a type 3 interrupt.
- INT <interrupt number> instruction - any one interrupt from available 256 interrupts.
- INTO instruction - interrupt on overflow
- Single-step interrupt - generated if the TF flag is set. This is a type 1 interrupt. When the CPU processes this interrupt it clears TF flag before calling the interrupt processing routine.
- Processor exceptions: Divide Error (Type 0), Unused Opcode (type 6) and Escape opcode (type 7).
- Software interrupt processing is the same as for the hardware interrupts.
- - Ex: INT n (Software Instructions)
- Control is provided through:
    - IF and TF flag bits
    - IRET and IRETD

**Performance of Software Interrupts**




1. It decrements SP by 2 and pushes the flag register on the stack.
2. Disables INTR by clearing the IF.
3. It resets the TF in the flag Register.
5. It decrements SP by 2 and pushes CS on the stack.
6. It decrements SP by 2 and pushes IP on the stack.
6. Fetch the ISR address from the interrupt vector table.

**Interrupt Vector Table**




| INT Number | Physical Address |
| --- | --- |
| INT 00 | 00000 |
| INT 01 | 00004 |
| INT 02 | 00008 |
| : | : |
| : | : |
| INT FF | 003FC |

**Functions associated with INT00 to INT04**

**INT 00 (divide error)**

- INT00 is invoked by the microprocessor whenever there is an attempt to divide a number by zero.
- ISR is responsible for displaying the message "Divide Error" on the screen

**INT 01**

- For single stepping the trap flag must be 1
- After execution of each instruction, 8086 automatically jumps to 00004H to fetch 4 bytes for CS: IP of the ISR.
- The job of ISR is to dump the registers on to the screen

**INT 02 (Non maskable Interrupt)**

- When ever NMI pin of the 8086 is activated by a high signal (5v), the CPU Jumps to physical memory location 00008 to fetch CS:IP of the ISR associated with NMI.

**INT 03 (break point)**

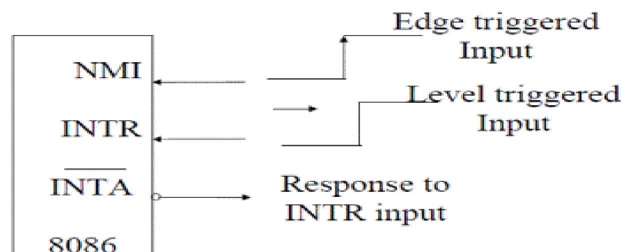
- A break point is used to examine the CPU and memory after the execution of a group of Instructions.
- It is one byte instruction whereas other instructions of the form "INT nn" are 2 byte instructions.

**INT 04 (Signed number overflow)**

- There is an instruction associated with this INT 0 (interrupt on overflow).
- If INT 0 is placed after a signed number arithmetic as IMUL or ADD the CPU will activate INT 04 if 0F = 1.
- In case where 0F = 0, the INT 0 is not executed but is bypassed and acts as a NOP.

**Performance of Hardware Interrupts**

- NMI : Non maskable interrupts - TYPE 2 Interrupt
- INTR : Interrupt request - Between 20H and FFH




**Interrupt Priority Structure**

| Interrupt | Priority |
|---|---|
| Divide Error, INT(n),INTO | Highest |
| NMI | |
| INTR | |
| Single Step | Lowest |

# INTRODUCTION TO MICRO CONTROLLERS

## 6.0 INTRODUCTION:

We have noticed that Microprocessor is just not self-sufficient, and it requires other components like memory and input/output devices to form a minimum workable system configuration. To have all these components in a discrete form and to assemble them on a PCB is usually not an affordable solution for the following reasons:

1) The overall system cost of a microprocessor based system built around a CPU, memory and other peripherals is high as compared to a microcontroller based system.
2) A large sized PCB is required for assembling all these components, resulting in an enhanced cost of the system.
3) Design of such PCBs requires a lot of effort and time and thus the overall product design requires more time.
4) Due to the large size of the PCB and the discrete components used, physical size of the product is big and hence it is not handy.
5) As discrete components are used, the system is not reliable nor is it easy to trouble- shoot such a system.

Considering all these problems, Intel decided to integrate a microprocessor along with I/O ports and minimum memory into a single package. Another frequently used peripheral, a programmable timer, was also integrated to make this device a self-sufficient one. This device which contains a microprocessor and the above mentioned components has been named a microcontroller. A microcontroller is a microprocessor with integrated peripherals. Design with microcontrollers has the following advantages:

1. As the peripherals are integrated into a single chip, the overall system cost is very low.
2. The size of the product is small as compared to the microprocessor based systems thus very handy.
3. The system design requires very little efforts and is easy to troubleshoot and maintain.
4. As the peripherals are integrated with a microprocessor, the system is more reliable.
5. Though a microcontroller may have on-chip RAM, ROM and I/O ports, additional RAM, ROM and I/O ports may be interfaced externally, if required.
6. The microcontrollers with on-chip ROM provide a software security feature which is not available with microprocessor based systems using ROM/EPROM.

However, in case of a larger system design, which requires more number of I/O ports and more memory capacity, the system designer may interface external I/O ports and memory with

the system. In such cases, the microcontroller based systems are not so attractive as they are in case of the small dedicated systems. Figure 17.1 shows a typical microcontroller internal block diagram.

As a microcontroller contains most of the components required to form a microprocessor system, it is sometimes called a single chip microcomputer, since it also has the ability to easily implement simple control functions.

## 6.1 OVERVIEW OF 8051 MICRO CONTROLLER

Let us look at Intel's 8-bit microcontroller family, popularly known as MCS-51 family. The earlier versions of Intel's microcontrollers do not have on-chip EPROM. 8031 was one such microcontroller from Intel, followed by the 8051 family. 8751 was the first microcontroller version with on-chip EPROM, followed by a number of 8751 versions with slight modifications. Recently, an electrically programmable and erasable version of 8051, named as 8951, has been introduced. Table shows the comparison between different versions of 8051. All these members of the 8051 family have identical instruction set and similar architecture with slight variations as shown in Table.

## 6.2 ARCHITECTURE OF 8051

The internal architecture of 8051 is presented in Fig.

The functional description of each block is presented briefly below.

**Accumulator (ACC):** The accumulator register (ACC or A) acts as an operand register, in case of some instructions. This may either be implicit or specified in the instruction.

**B Register:** This register is used to store one of the operands for multiply and divide instructions. In other instructions, it may just be used as a scratch pad.

**Program Status Word (PSW):** This set of flags contains the status information.

**Stack Pointer (SP):** This 8-bit wide register is incremented before the data is stored onto the stack using push or call instructions. This register contains 8-bit stack top address. The stack may be defined anywhere in the on-chip 128-byte RAM. After reset, the SP register is initialised to 07. After each write to stack operation, the 8-bit contents of the operand are stored onto the stack, after incrementing the SP register by one. Thus if SP contains 07 H, the forthcoming PUSH operation will store the data at address 08H in the internal RAM. The SP content will be incremented to 08.

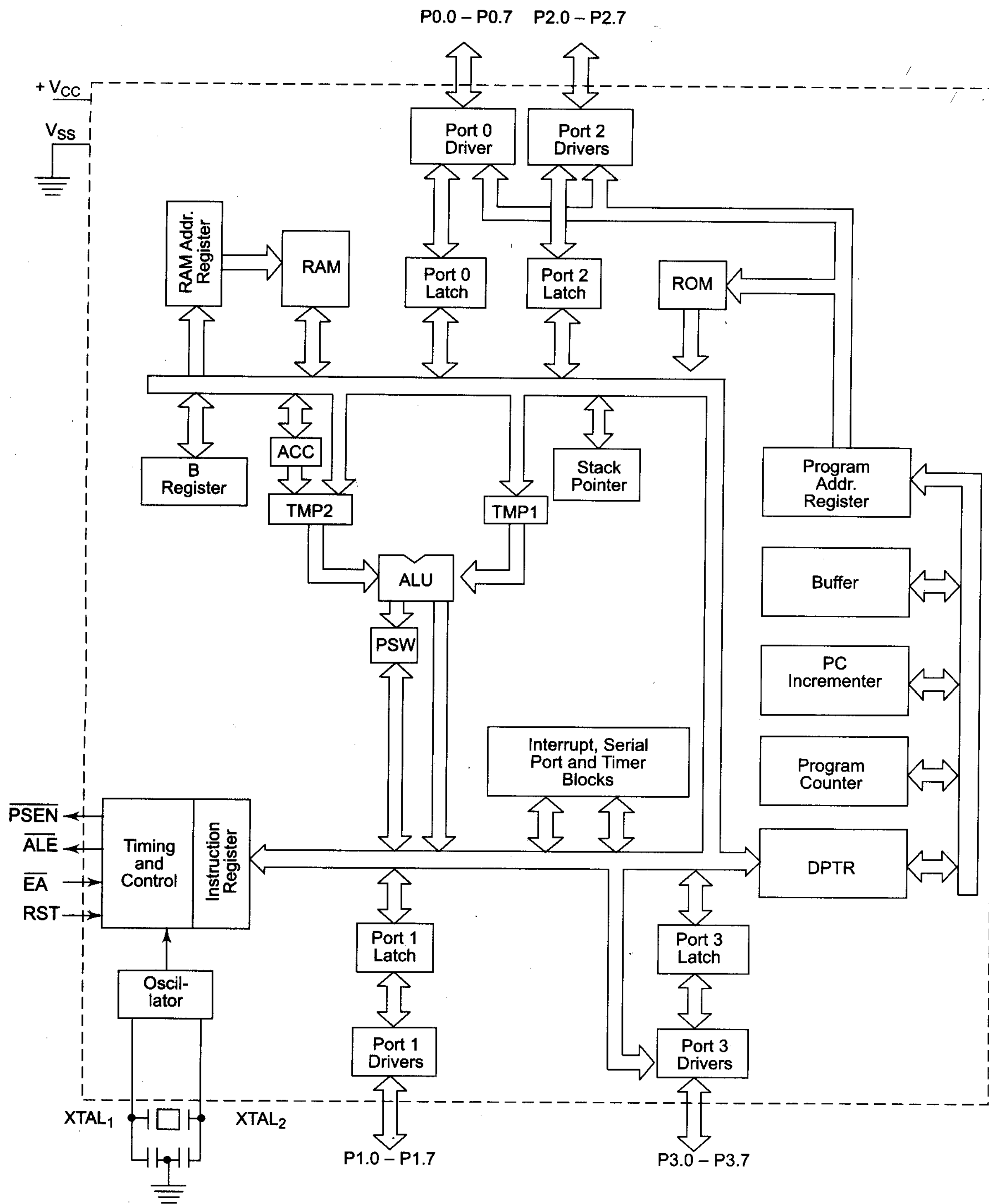



**Fig. 17.2** *8051 Block Diagram (Intel Corp.)*

**Data Pointer (DTPR):** This 16-bit register contains a higher byte (DPH) and the lower byte (DPL) of a 16-bit external data RAM address. It is accessed as a 16-bit register or two 8-bit registers as specified above.

**Port 0 to 3 Latches and Drivers:** These four latches and driver pairs are allotted to each of the four on-chip I/O ports. Using the allotted addresses, the user can communicate with these ports. These are identified as P0, PI, P2 and P3.

**Serial Data Buffer:** The serial data buffer internally contains two independent registers. One of them is a transmit buffer which is necessarily a parallel-in serial-out register. The other is called receive buffer which is a serial-in parallel-out register. The serial data buffer is identified as SBUF.

**Timer Registers:** These two 16-bit registers can be accessed as their lower and upper bytes. For example, TL0 represents the lower byte of the timing register 0, while TH0 represents higher bytes of the timing register 0. Similarly, TL1 and TH1 represent lower and higher bytes of timing register 1.

**Control Registers:** The special function registers IP, IE, TMOD, TCON, SCON and PCON contain control and status information for interrupts, timers/counters and serial port.

**Timing and Control Unit:** This unit derives all the necessary timing and control signals required for the internal operation of the circuit. It also derives control signals required for controlling the external system bus.

**Oscillator:** This circuit generates the basic timing clock signal for the operation of the circuit using crystal oscillator.

**Instruction Register:** This register decodes the opcode of an instruction to be executed and gives information to the timing and control unit to generate necessary signals for the execution of the instruction.

**EPROM and Program Address Register:** These blocks provide an on-chip EPROM/PROM and a mechanism to internally address it. Note that EPROM is not available in all 8051 versions.

**RAM and RAM Address Register:** These blocks provide internal 128 bytes of RAM and a mechanism to address it internally.

**ALU:** The arithmetic and logic unit performs 8-bit arithmetic and logical operations over the operands held by the temporary registers TMP1 and TMP2. Users cannot access these temporary registers.

**SFR Register Bank:** This is a set of special function registers, which can be addressed using their respective addresses which lie in the range 80H to FFH.

Finally, the interrupt, serial port and timer units control and perform their specific functions under the control of the timing and control unit.

## 6.3 PIN DESCRIPTIONS OF 8051

8051 is available in a 40-pin plastic and ceramic DIP packages. The pin diagram of 8051 is shown in Fig. 17.3 followed by description of each pin.

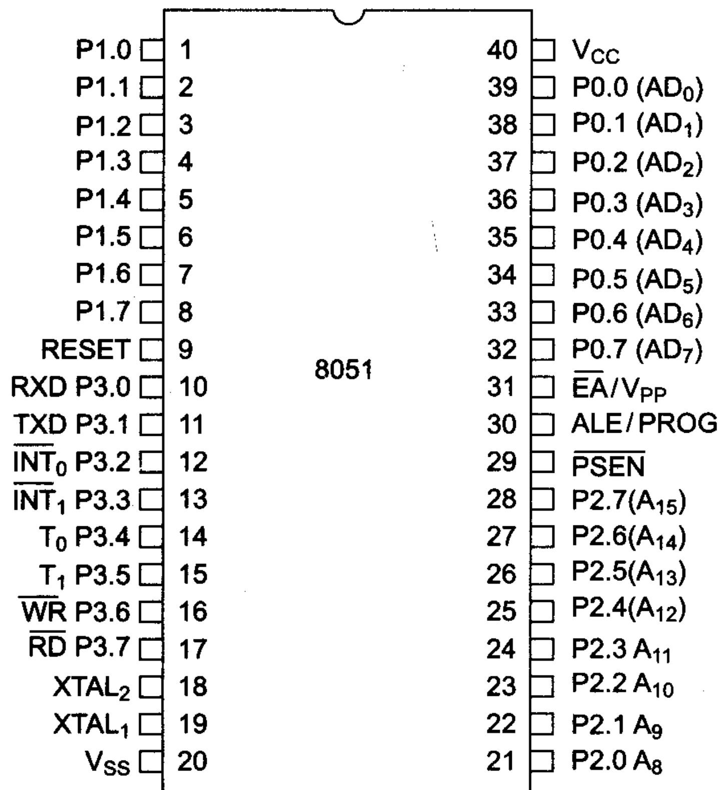



**Fig. 17.3** *8051 Pin Configuration (Intel Corp.)*

**$V_{cc}$** This is a +5 V supply voltage pin

**$V_{ss}$** This is a return pin for the supply.

**RESET** The reset input pin resets the 8051, only when it goes high for two or more machine cycles. For a proper reinitialization after reset, the clock must be running.

**ALE/PROG** The address latch enable output pulse indicates that the valid address bits are available on their respective pins. This ALE signal is valid only for external memory accesses. Normally, the ALE pulses are emitted at a rate of one-sixth of the oscillator frequency. This pin acts as program pulse input during on-chip EPROM programming. ALE may be used for external timing or clocking purpose. One ALE pulse is skipped during each access to external data memory.

**$\overline{EA}$ /$V_{pp}$** External access enable pin, if tied low, indicates that the 8051 can address external program memory. In other words, the 8051 can execute a program in external memory, only if $\overline{EA}$ is tied low.

For execution of programs in internal memory, the $\overline{EA}$ must be tied high. This pin also receives 21 volts for programming of the on-chip EPROM.

**$\overline{\textbf{PSEN}}$** Program store enable is an active-low output signal that acts as a strobe to read the external program memory. This goes low during external program memory accesses.

**Port 0 (P0.0-P0.7)** Port 0 is an 8-bit bidirectional bit addressable I/O port. This has been allotted an address in the SFR address range. Port 0 acts as multiplexed address/data lines during external memory access, i.e. when $\overline{EA}$ is low and ALE emits a valid signal. In case of controllers with on-chip EPROM, Port 0 receives code bytes during programming of the internal EPROM.

**Port 1 (P1.0-P1.7)** Port 1 acts as an 8-bit bidirectional bit addressable port. This has been allotted an address in the SFR address range.

**Port 2 (P2.0-P2.7)** Port 2 acts as 8-bit bidirectional bit addressable I/O port. It has been allotted an address in the SFR address range of 8051. During external memory accesses, port 2 emits higher eight bits of adress ($A_8$-$A_{15}$) which are valid, if ALE goes high and EA is low. P2 also receives higher order address bits during programming of the on-chip EPROM.

**Port 3 (P3.0-P3.7)** Port 3 is an 8-bit bidirectional bit addressable I/O port which has been allotted an address in the SFR address range of 8051. The port 3 pins also serve the alternative functions as listed in the Table 17.2.

**$XTAL_1$ and $XTAL_2$** There is an inbuilt oscillator which derives the necessary clock frequency for the operation of the controller. $XTAL_1$ is the input of amplifier and $XTAL_2$ is the output of the amplifier. A crystal is to be connected externally between these two pins to complete the feedback path to start oscillations. The controller can be operated on an external clock. In this case the external clock is fed to the controller at pin $XTAL_2$ and $XTAL_1$ pin should be grounded. Commercially available versions of 8051 run on 12 MHz to 16 MHz frequency.

## 6.4 REGISTER SET OF 8051

8051 has two 8-bit registers, registers A and B, which can be used to store operands, as allowed by the instruction set. Internal temporary registers of 8051 are not user accessible. Including these A and B registers, 8051 has a family of special purpose registers known as, Special Function Registers (SFRs). There are, in total, 21-bit addressable, 8-bit registers. ACC (A), B, PSW, PO, PI, P2, P3, IP, IE, TCON and SCON are all 8-bit, bit-addressable registers. The remaining registers, namely, SP, DPH, DPL, TMOD, TH0, TL0, TH1, TL1, SBUF and PCON registers are to be addressed as bytes, i.e. they are not bit-addressable. The registers DPH and DPL are the higher and lower bytes of a 16-bit register DPTR, i.e. data pointer, which is used for accessing external data memory. Starting 32-bytes of on-chip RAM may be used as general purpose registers. They have been allotted addresses in the range from 0000H to 001FH. These 32, 8-bit registers are divided into four groups of 8 registers each, called register banks.

At a time only one of these four groups, i.e. banks can be accessed. The register bank to be accessed can be selected using the RS1 and RS0 bits of an internal register called program status word.

The registers TH0 and TL0 form a 16-bit counter/timer register with H indicating the upper byte and L indicating the lower byte of the 16-bit timer register T0. Similarly, TH1 and TLl form the 16-bit count for the timer Tl. The four port latches are represented by P0, P1, P2 and P3. Any communication with these ports is established using the SFR addresses to these registers. Register SP is a stack pointer register. Register PSW is a flag register and contains status information. Register IP can be programmed to control the interrupt priority. Register IE can be programmed to control interrupts, i.e. enable or disable the interrupts. TCON is called timer/counter control register. Some of the bits of this register are used to turn the timers on or off. This register also contains interrupt control flags for external interrupts $\overline{INT_0}$ and $\overline{INT_1}$. The register TMOD is used for programming the modes of operation of the timers/counters. The SCON register is a serial port mode control register and is used to control the operation of the serial port. The SBUF register acts as a serial data buffer for transmit and receive operations. The PCON register is called power control register. This register contains power down bit and idle bit which activate the power down mode and idle mode in 80C51BH. There are two power saving modes of operation provided in the CHMOS version, namely, **idle mode and power down mode.**

In the **idle mode,** the oscillator continues to run and the interrupt, serial port and timer blocks are active but the clock to the CPU is disabled. The CPU status is preserved. This mode can be terminated with a hardware interrupt or hardware reset signal. After this, the CPU resumes program execution from where it left off.

In **power down mode,** the on-chip oscillator is stopped. All the functions of the controller are held maintaining the contents of RAM. The only way to terminate this mode is hardware reset. The reset redefines all the SFRs but the RAM contents are left unchanged. Both of these modes can be entered by setting the respective bit in an internal register called PCON register using software.

All these registers are listed in Table 17.3 along with their SFR addresses and contents after reset.

**IMPORTANT OPERATIONAL FEATURES OF 8051**

This section describes the critical special function register formats of 8051.

1. **Program Status Word (PSW)**

This bit-addressable register has the following format as shown in Fig. 17.4. The bit descriptions are presented along with the format.

2. **Timer Mode Control Register (TMOD)**

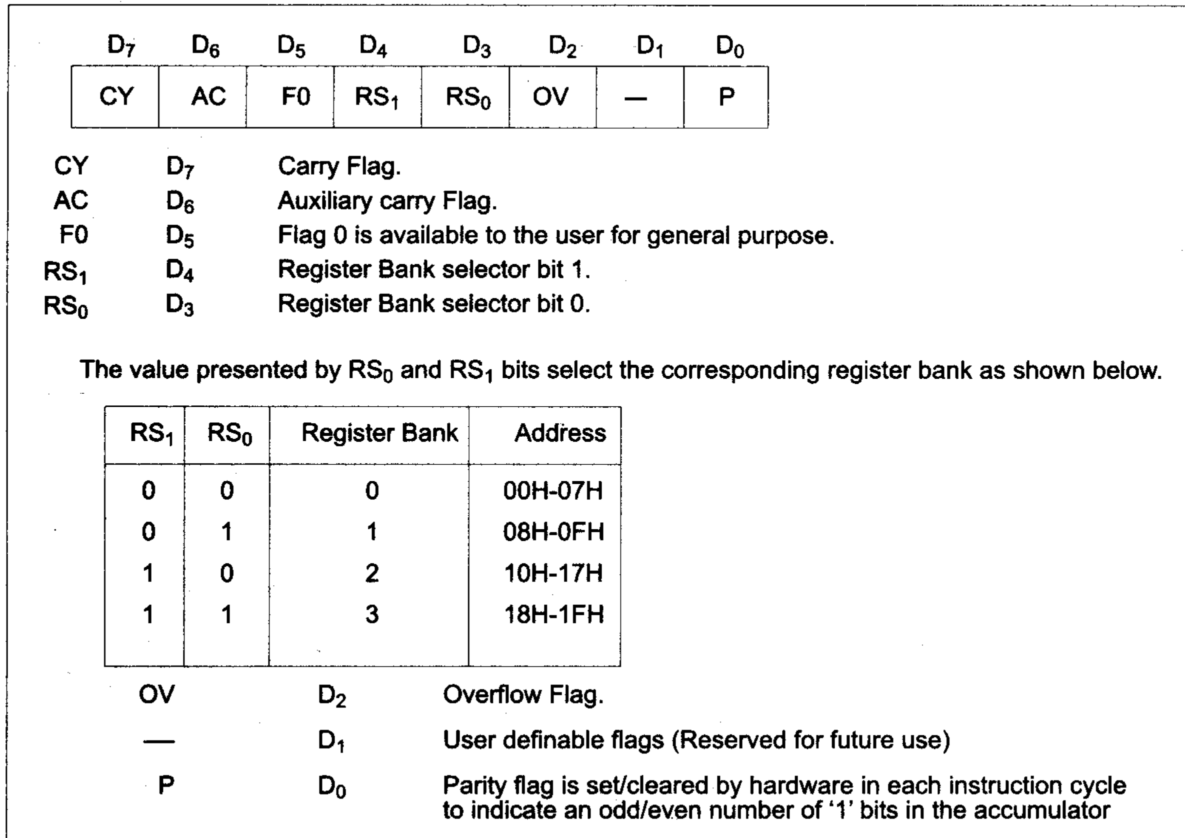
Format of this 8-bit non-bit-addressable register is shown along with its bit descriptions in Fig. 5.

3. **Timer Control Register (TCON)**

This bit-addressable register format along with its bit definitions is shown in Fig..

4. **Serial Ports Control Register (SCON)**

This 8-bit, bit-addressable register format is shown in Fig.



| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|
| CY | AC | F0 | $RS_1$ | $RS_0$ | OV | — | P |

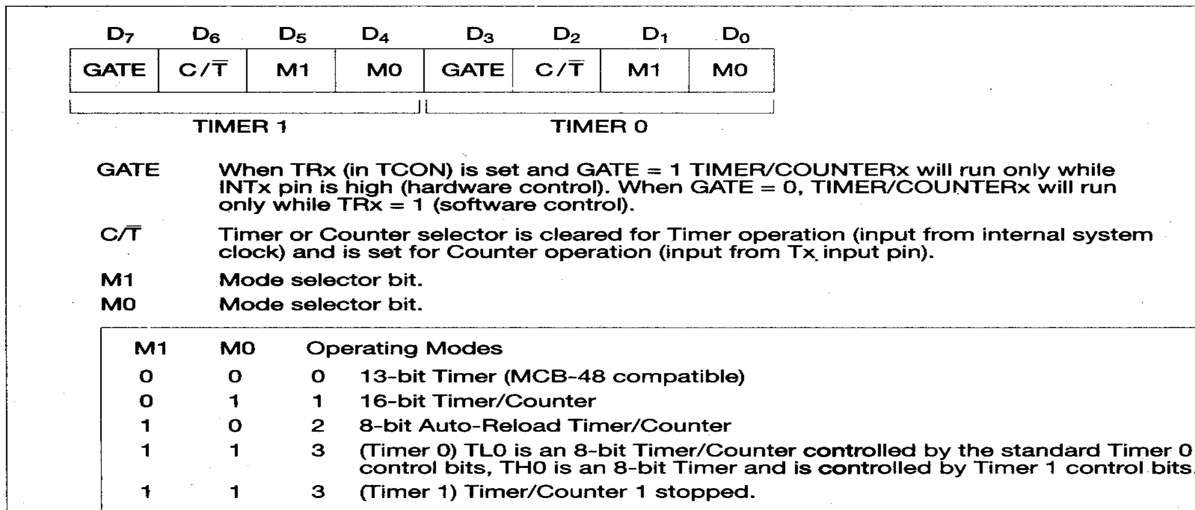
CY $D_7$ Carry Flag.
AC $D_6$ Auxiliary carry Flag.
F0 $D_5$ Flag 0 is available to the user for general purpose.
$RS_1$ $D_4$ Register Bank selector bit 1.
$RS_0$ $D_3$ Register Bank selector bit 0.

The value presented by $RS_0$ and $RS_1$ bits select the corresponding register bank as shown below.

| $RS_1$ | $RS_0$ | Register Bank | Address |
|---|---|---|---|
| 0 | 0 | 0 | 00H-07H |
| 0 | 1 | 1 | 08H-0FH |
| 1 | 0 | 2 | 10H-17H |
| 1 | 1 | 3 | 18H-1FH |

OV $D_2$ Overflow Flag.
— $D_1$ User definable flags (Reserved for future use)
P $D_0$ Parity flag is set/cleared by hardware in each instruction cycle to indicate an odd/even number of '1' bits in the accumulator

**Fig. 17.4** *Format of PSW (Intel Corp.)*

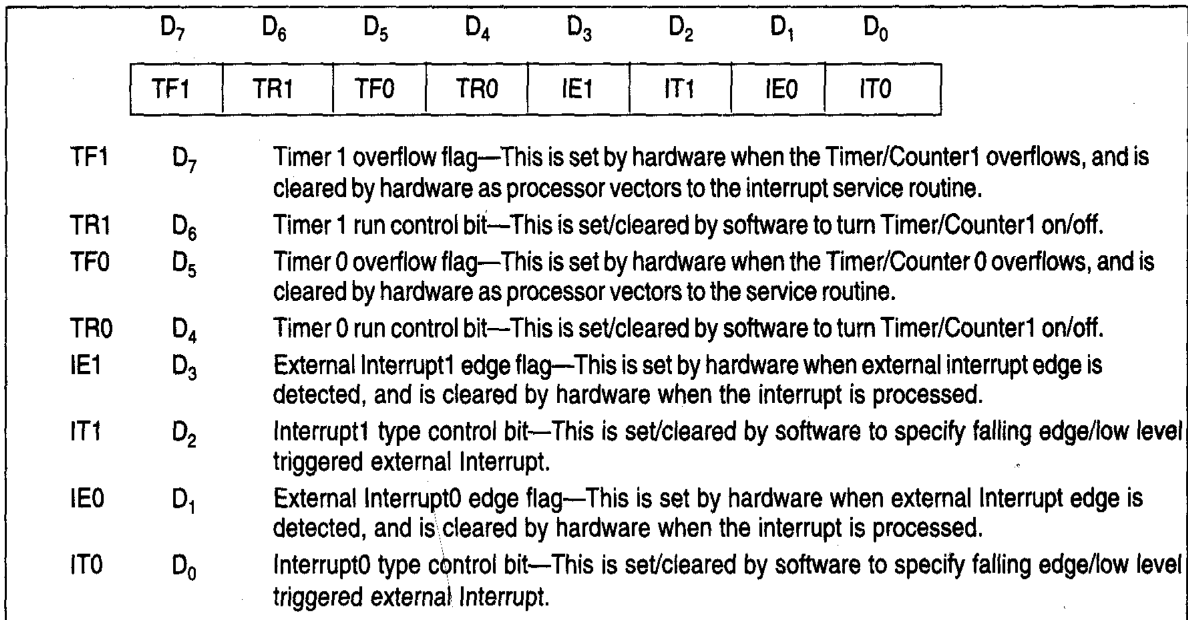


| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|
| GATE | C/$\overline{T}$ | M1 | M0 | GATE | C/$\overline{T}$ | M1 | M0 |
| TIMER 1 | | | | TIMER 0 | | | |

| | |
|---|---|
| GATE | When TRx (in TCON) is set and GATE = 1 TIMER/COUNTERx will run only while INTx pin is high (hardware control). When GATE = 0, TIMER/COUNTERx will run only while TRx = 1 (software control). |
| C/$\overline{T}$ | Timer or Counter selector is cleared for Timer operation (input from internal system clock) and is set for Counter operation (input from Tx input pin). |
| M1 | Mode selector bit. |
| M0 | Mode selector bit. |

| M1 | M0 | Operating Modes | |
|---|---|---|---|
| 0 | 0 | 0 | 13-bit Timer (MCB-48 compatible) |
| 0 | 1 | 1 | 16-bit Timer/Counter |
| 1 | 0 | 2 | 8-bit Auto-Reload Timer/Counter |
| 1 | 1 | 3 | (Timer 0) TL0 is an 8-bit Timer/Counter controlled by the standard Timer 0 control bits, TH0 is an 8-bit Timer and is controlled by Timer 1 control bits. |
| 1 | 1 | 3 | (Timer 1) Timer/Counter 1 stopped. |

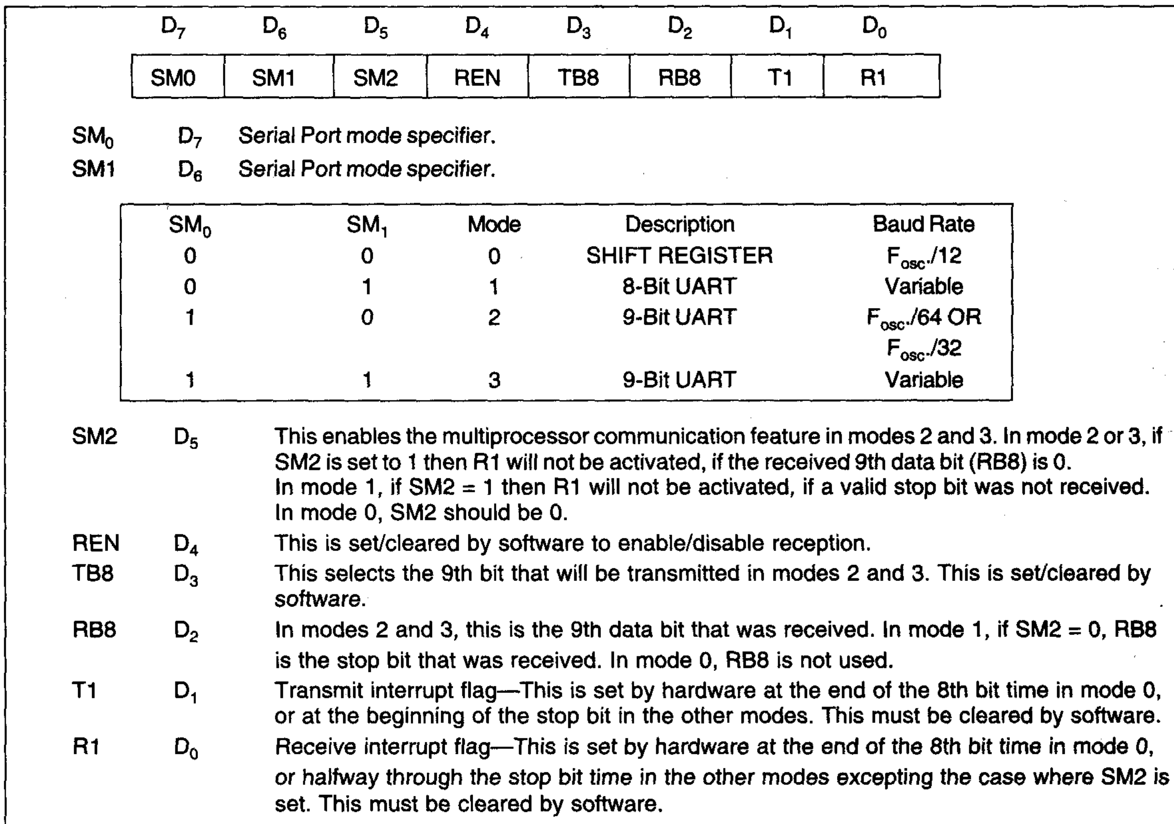
**Fig. 17.5** *Format of TMOD Register (Intel Corp.)*



| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|
| TF1 | TR1 | TF0 | TR0 | IE1 | IT1 | IE0 | IT0 |

| | | |
|---|---|---|
| TF1 | $D_7$ | Timer 1 overflow flag—This is set by hardware when the Timer/Counter1 overflows, and is cleared by hardware as processor vectors to the interrupt service routine. |
| TR1 | $D_6$ | Timer 1 run control bit—This is set/cleared by software to turn Timer/Counter1 on/off. |
| TF0 | $D_5$ | Timer 0 overflow flag—This is set by hardware when the Timer/Counter 0 overflows, and is cleared by hardware as processor vectors to the service routine. |
| TR0 | $D_4$ | Timer 0 run control bit—This is set/cleared by software to turn Timer/Counter1 on/off. |
| IE1 | $D_3$ | External Interrupt1 edge flag—This is set by hardware when external interrupt edge is detected, and is cleared by hardware when the interrupt is processed. |
| IT1 | $D_2$ | Interrupt1 type control bit—This is set/cleared by software to specify falling edge/low level triggered external Interrupt. |
| IE0 | $D_1$ | External Interrupt0 edge flag—This is set by hardware when external Interrupt edge is detected, and is cleared by hardware when the interrupt is processed. |
| IT0 | $D_0$ | Interrupt0 type control bit—This is set/cleared by software to specify falling edge/low level triggered external Interrupt. |

**Fig. 17.6** *Format of TCON Register (Intel Corp.)*



| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|
| SM0 | SM1 | SM2 | REN | TB8 | RB8 | T1 | R1 |

$SM_0$ $D_7$ Serial Port mode specifier.
SM1 $D_6$ Serial Port mode specifier.

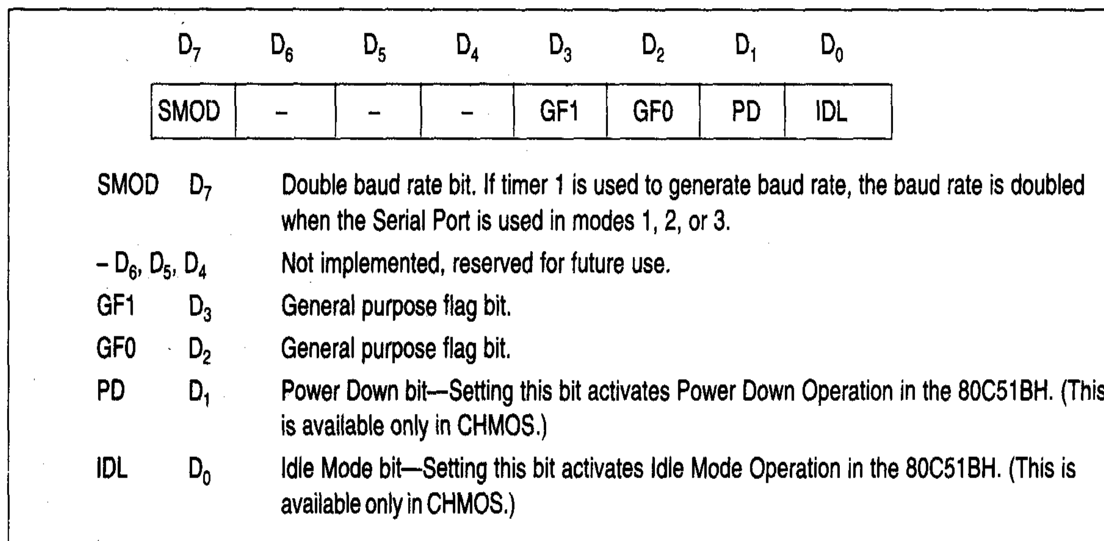
| $SM_0$ | $SM_1$ | Mode | Description | Baud Rate |
|---|---|---|---|---|
| 0 | 0 | 0 | SHIFT REGISTER | $F_{osc.}/12$ |
| 0 | 1 | 1 | 8-Bit UART | Variable |
| 1 | 0 | 2 | 9-Bit UART | $F_{osc.}/64$ OR $F_{osc.}/32$ |
| 1 | 1 | 3 | 9-Bit UART | Variable |

SM2 $D_5$ This enables the multiprocessor communication feature in modes 2 and 3. In mode 2 or 3, if SM2 is set to 1 then R1 will not be activated, if the received 9th data bit (RB8) is 0. In mode 1, if SM2 = 1 then R1 will not be activated, if a valid stop bit was not received. In mode 0, SM2 should be 0.
REN $D_4$ This is set/cleared by software to enable/disable reception.
TB8 $D_3$ This selects the 9th bit that will be transmitted in modes 2 and 3. This is set/cleared by software.
RB8 $D_2$ In modes 2 and 3, this is the 9th data bit that was received. In mode 1, if SM2 = 0, RB8 is the stop bit that was received. In mode 0, RB8 is not used.
T1 $D_1$ Transmit interrupt flag—This is set by hardware at the end of the 8th bit time in mode 0, or at the beginning of the stop bit in the other modes. This must be cleared by software.
R1 $D_0$ Receive interrupt flag—This is set by hardware at the end of the 8th bit time in mode 0, or halfway through the stop bit time in the other modes excepting the case where SM2 is set. This must be cleared by software.

**Fig. 17.7** *Format of SCON Register (Intel Corp.)*

### 6.5.5 Power Control Register (PCON)

The format of this non-bit-addressable register is shown in Fig. 17.8.



| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|
| SMOD | – | – | – | GF1 | GF0 | PD | IDL |

SMOD $D_7$ Double baud rate bit. If timer 1 is used to generate baud rate, the baud rate is doubled when the Serial Port is used in modes 1, 2, or 3.
– $D_6$, $D_5$, $D_4$ Not implemented, reserved for future use.
GF1 $D_3$ General purpose flag bit.
GF0 $D_2$ General purpose flag bit.
PD $D_1$ Power Down bit—Setting this bit activates Power Down Operation in the 80C51BH. (This is available only in CHMOS.)
IDL $D_0$ Idle Mode bit—Setting this bit activates Idle Mode Operation in the 80C51BH. (This is available only in CHMOS.)

**Fig. 17.8** *Format of PCON Register (Intel Corp.)*

## INTERRUPTS OF 8051

8051 provides five sources of interrupts. $\overline{INT_0}$ and $\overline{INT_1}$ are the two external interrupt inputs. These can either be edge-sensitive or level-sensitive, as programmed with bits $IT_0$ and $IT_1$

register TCON. These interrupts are processed internally by the flags $IE_0$ and $IE_1$. If the interrupts are programmed as edge-sensitive, these flags are automatically cleared after the control is transferred to the respective vector. On the other hand, if the interrupts are programmed level-sensitive, these flags are controlled by the external interrupts sources themselves. Both timers can be used in timer or counter mode. In counter mode, it counts the pulses at $T_0$ or $T_1$ pin. In timer mode, oscillator clock is divided by a pre-scalar (1/32) and then given to the timer. So clock frequency for timer is 1/32th of the controller operating frequency. The timer is an up-counter and generates an interrupt when the count has reached FFFFH. It can be operated in four different modes that can be set by TMOD register.

The timer 0 and timer 1 interrupt sources are generated by $TF_0$ and $TF_1$ bits of the register TCON, which are set, if a rollover takes place in their respective timer registers, except timer 0 in mode 3. When these interrupts are generated, the respective flags are automatically cleared after the control is transferred to the respective interrupt service routines.

The serial port interrupt is generated, if at least one of the two bits RI and TI is set. Neither of the flags is cleared, after the control is transferred to the interrupt service routine. The RI and TI flags need to be cleared using software, after deciding, which one of these two caused the interrupt? This is accomplished in the interrupt service routine.

In addition to these five interrupts, 8051 also allows single step interrupts to be generated with help of software. The external interrupts, if programmed level-sensitive, should remain high for at least two machine cycles for being sensed. If the external interrupts are programmed edge-sensitive, they should remain high for at least one machine cycle and low for at least one machine cycle, for being sensed.

The interrupt structure of 8051 provides two levels of the interrupt priorities for its sources of interrupt. Each interrupt source can be programmed to have one of these two levels using the interrupt priority register IP. The different sources of interrupts programmed to have the same level of priority, further follow a sequence of priority under that level as shown:

| Interrupt Source | Priority within a level |
|---|---|
| IE0 (External INT0) | Highest |
| TF0 (Timer 0) | ⋮ |
| IE1 (External INT1) | ⋮ |
| TF1 (Timer 1) | ⋮ |
| RI = TI (Serial Port) | Lowest |

All these interrupts are enabled using a special function register called **interrupt enable register** (IE) and their priorities are programmed using another special function register called **interrupt priority register** (IP). Formats of both of these registers are shown in Fig. 17.13 and Fig. 17.14.

| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|
| EA | – | ET2 | ES | ET1 | EX1 | ET0 | EX0 |

| | | |
|---|---|---|
| EA | $D_7$ | This disables all interrupts. If EA = 0, no interrupt will be acknowledged. If EA = 1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit. |
| – | $D_6$ | Not implemented, reserved for future use. User software should not write 1s to reserved bits. These bits may be used in future MCS-51 products to invoke new features. In that case, the reset or inactive value of the new bit will be 0, and its active value will be 1. |
| ET2 | $D_5$ | This enables or disables Timer 2 overflow or capture interrupt (8052 only). |
| ES | $D_4$ | This enables or disables the serial port interrupt. |
| ET1 | $D_3$ | This enables or disables the Timer 1 overflow interrupt. |
| EX1 | $D_2$ | This enables or disables external Interrupt 1. |
| ET0 | $D_1$ | This enables or disables the Timer 0 overflow interrupt. |
| EX0 | $D_0$ | This enables or disables external Interrupt 0. |

**Fig. 17.13** *Format of IE Register*

| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|
| – | – | PT2 | PS | PT1 | PX1 | PT0 | PX0 |

If the bit is 0, the corresponding interrupt is disabled. If the bit is 1 the corresponding interrupt is enabled.

| | | |
|---|---|---|
| – | $D_7$ | Not implemented, reserved for future use.* |
| – | $D_6$ | Not implemented, reserved for future use.* |
| PT2 | $D_5$ | This defines the Timer 2 interrupt priority level (8052 only). |
| PS | $D_4$ | This defines the Serial Port interrupt priority level. |
| PT1/PT0 | $D_3/D_1$ | This defines the Timer 1/Timer 0 interrupt priority level. |
| PX1/PX0 | $D_2/D_0$ | This defines External $\overline{INT1}/\overline{INT0}$ priority level. |

* The software should not write 1s to reserved bits. These bits may be used in future MCS-51 products to invoke new features. In that case, the reset or inactive value of the new bit will be 0, and its active value will be 1.

**Fig. 17.14** *Format of IP Register*

## INTERRUPTS

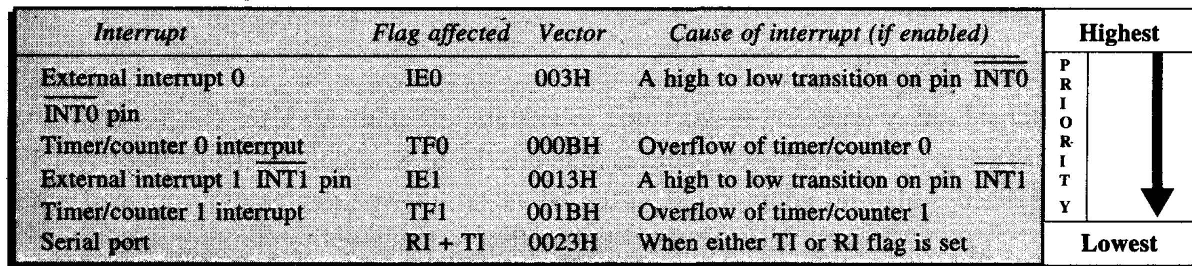
Interrupt is an input to a processor that indicates the occurrence of an event. In case of external events, the status of a microprocessor pin is altered. Interrupts are also generated due to the events occurring inside the machine like timer overflow or transmission/reception of a byte through the serial port, etc. The processor responds to an interrupt by saving the current machine status and branching to execute a subprogram called 'interrupt service subroutine'. When an interrupt occurs, the CPU jumps to the location associated with that interrupt, in the program memory and starts executing from there. This location is called 'vector' and the interrupt is called vectored interrupt. After serving the interrupt, the processor restores the original machine status and continues with the original program.

### INTERRUPTS IN MCS-51

MCS-51 supports five vectored interrupt sources. These are external interrupt 0, external interrupt 1, timer/counter 0 interrupt, timer/counter 1 interrupt and serial port interrupts. When an interrupt is generated, the program counter (PC) is pushed onto a stack. Vectored address is loaded in the program counter. As the vectoring takes place, that particular interrupt flag corresponding to the interrupt source (e.g. external interrupt 1) is cleared by the hardware. In MCS-51, these flags are bits IE0, IE1, TF0, TF1, RI and TI.

The program now starts executing from the vectored location. This subroutine is called as the interrupt service subroutine (ISS). The ISS ends with RETI instruction. The interrupt vector locations in 8051 are spaced out at every 8 bytes, so technically it is possible to put ISS there if it were no longer than 8 bytes, including RETI instruction. Otherwise and in almost all the cases, a jump instruction is written at the vectored address (3 bytes maximum), and the remaining part ISS is located somewhere else. The vector addresses are listed in the following Table 6.1 in the order of priority. Consider the external interrupt 1. Assume that this interrupt is initialized properly in program. While the CPU is busy with the main program, if a '1' to '0' transition occurs at pin number 12, ($\overline{INT0}$ pin), the program counter (PC) current contents are stored onto the stack and the PC is then loaded with the vectored address 0003H. Thus, the next instruction at 0003H would be fetched and executed. Now there are only 8 bytes available to write the interrupt service subroutine, as seen above. Therefore, normally a JMP instruction is written at this vectored location 0003H. The interrupt service subroutine lying somewhere else in the program memory, ends with RETI instruction. This RETI instruction will get the program counter contents from the stack and the CPU will again start executing from where the main program was interrupted. Thus, any external event, which causes a change in the status of the interrupt pin, can be taken care of by the interrupt service subroutine. The external interrupts may be configured as either level-triggered or edge-triggered. If the interrupt is level-triggered, the signal must stay low until the interrupt is generated. In case of an edge-triggered interrupt, a transition from high to low at the interrupt pin is sufficient. It is further necessary that proper settings in the SFR called interrupt enable (IE) register is made to initialize the MCS-51 interrupts.

**Table 6.1** Interrupts in 8051



| Interrupt | Flag affected | Vector | Cause of interrupt (if enabled) |
|---|---|---|---|
| External interrupt 0 $\overline{INT0}$ pin | IE0 | 003H | A high to low transition on pin $\overline{INT0}$ |
| Timer/counter 0 interrput | TF0 | 000BH | Overflow of timer/counter 0 |
| External interrupt 1 $\overline{INT1}$ pin | IE1 | 0013H | A high to low transition on pin $\overline{INT1}$ |
| Timer/counter 1 interrupt | TF1 | 001BH | Overflow of timer/counter 1 |
| Serial port | RI + TI | 0023H | When either TI or RI flag is set |



## Initializing 8051 Interrupts

The interrupt enable (IE) register allows the programmer to enable interrupts as needed. This register IE is bit addressable and is shown in Fig. 6.1. Enable All (EA) bit allows disabling the whole interrupt operation, if cleared. Thus, it acts as a master control bit for any of the interrupts. For any particular interrupt to occur, bit EA and the corresponding bit must be set. For example, in case of serial interrupt, bit EA and bit ES must be set. ES is the serial port interrupt, useful in serial transmission, if set, enables the serial interrupts TI or RI Similarly, bits ET1, ET0 are for timer 1 and timer 0 interrupts, respectively. EX1 and EX0 are external interrupt enable bits for external interrupts 1 and 0, respectively. Programming Example #6.1 shows initialization of external interrupt 1.

| IE.7 | IE.6 | IE.5 | IE.4 | IE.3 | IE.2 | IE.1 | IE.0 |
|---|---|---|---|---|---|---|---|
| EA | .... | ... | ES | ET1 | EX1 | ET0 | EX0 |

| Bit | Description |
|---|---|
| EA (Enable All) | 0 = Disable all interrupts, 1= Allows each of the individual interrupts to be enabled |
| ES | Enable/Disable serial port interrupt, 0 = Disable, 1 = Enable (Provided EA = 1) |
| ET1 | Enable/Disable timer interrupt 1; 0 = Disable, 1 = Enable (Provided EA = 1) |
| EX1 | Enable/Disable external interrupt 1; 0 = Disable, 1= Enable (Provided EA = 1) |
| ET0 | Enable/Disable timer interrupt 0; 0 = Disable, 1 = Enable (Provided EA = 1) |
| EX0 | Enable/Disable external interrupt 0; 0 = Disable, 1= Enable (Provided EA = 1) |

**Fig. 6.1** Interrupt Enable Register (Bit Addressable)

```
; Programming Example #6.1
; Initialize the external interrupt 0
MOV IE, # 1000 0100 B ;enable external interrupt 1 (bit EA = 1 and bit EX1=1)
```

This instruction will enable the external interrupt 1. If now this is followed by CLR EA instruction, whole interrupt operation is disabled. To initialize the serial interrupt, one may load the IE register with 10010000B.

**Interrupt Priorities**

Let us consider the case, 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.

A high-priority interrupt can interrupt the low-priority interrupt, but a high-priority interrupt will not be interrupted by the interrupt having low priority. Now, if the request of interrupts of two different priority levels occur simultaneously, naturally the interrupt having the high priority will be served. However, 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 shown in Table 6.1 by the arrow. Note that the priority within level structure is used only to distinguish the requests of the same priority levels. Programming Example #6.2 shows the assignment of interrupt priority to timer 1 interrupt.

```
; Programming Example #6.2
; Assigning Interrupt Priorities
     MOV IE, #1000 1100H        ;Enable EX1 and ET1
     SETB PT1                   ;Timer 1 interrupt has high priority.
```

The first instruction enables both interrupts, namely, the external interrupt 1 and the timer 1 interrupt. The instruction SETB PT1 assigns high priority to the timer interrupt. So, if both of them request simultaneously, then the timer interrupt will be served. However, let us see what happens when one more instruction is added to this program. This is shown in Programming Example #6.3. Both external interrupt 1 and timer 1 interrupt have the same priorities. Now, if either interrupt requests occur simultaneously, the external interrupt will be served as per the priority order mentioned in the Table 6.1 and Fig. 6.2.

| 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 case of 8032/8052 only |
| IP.4 | PS | Serial interrupt priority |
| IP.3 | PT1 | Timer 1 interrupt |
| IP.2 | PX1 | External interrupt 1 |
| IP.1 | PT0 | Timer 0 interrupt |
| IP.0 | PX0 | External interrupt 0 |

**Fig. 6.2** Interrupt Priority Register (Bit Addressable)

```
; Programming Example #6.3
; Assigning Interrupt Priorities
      MOV IE, #1000 1100H          ;Enable EX1 and ET1
      SETB PT1                     ;Timer 1 interrupt has high priority
      SETB PX1                     ;External interrupt also has high priority
```

# MODULE – II

# INSTRUCTION SET AND PROGRAMMING OF 8051

## MEMORY AND I/O ADDRESSING BY 8051

### 1. Memory Addressing

The total memory of an 8051 system is logically divided into program memory and data memory. Program memory stores the programs to be executed, while data memory stores the data like intermediate results, variables and constants required for the execution of the program. Program memory is invariably implemented using EPROM, because it stores only program code which is to be executed and thus it need not be written into. However, the data memory may be read from or written to and thus it is implemented using RAM.

Further, the program memory and data memory both may be categorized as on-chip (internal) and external memory, depending upon whether the memory physically exists on the chip or it is externally interfaced. The 8051 can address 4 Kbytes on-chip program memory whose map starts from 0000H and ends at 0FFFH. It can address 64 Kbytes of external program memory under the control of $\overline{PSEN}$ signal, whose address map is from 0000H to FFFFH. Here, one may note that the map of internal program memory overlaps with that of the external program memory. However, these two memory spaces can be distinguished using the PSEN signal. In case of ROM-less versions of 8051, the $\overline{PSEN}$ signal is used to access the external program memory. Conceptually this is shown in Fig. 17.9.

8051 supports 64 Kbytes of external data memory whose map starts at 0000H and ends at FFFFH. This external data memory can be accessed under the control of register DPTR, which stores the addresses for external data memory accesses. 8051 generates $\overline{RD}$ and $\overline{WR}$ signals during external data memory accesses. The chip select line of the external data memory may be derived from the address lines as in the case of other microprocessors. Internal data memory of 8051 consists of two parts; the first is the RAM block of 128 bytes (256 bytes in case of some versions of 8051) and the second is the set of addresses from 80H to FFH, which includes the addresses allotted to the special function registers.

The address map of the 8051 internal RAM (128 bytes) starts from 00 and ends at 7FH. This RAM can be addressed by using direct or indirect mode of addressing. However, the special function register address map, i.e. from 80H to FFH is accessible only with direct addressing mode.

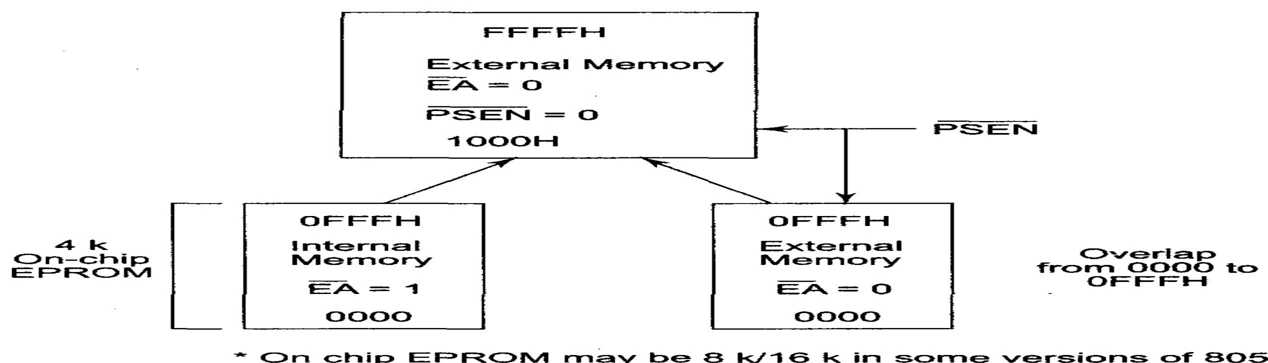



Fig. 17.9 Program Memory Map of an 8051 System

In case of 8051 versions with 256 bytes on-chip RAM, the map starts from 00H and ends at FFH. In this case, it may be noted that the address map of special function registers, i.e. 80H to FFH overlaps with the upper 128 bytes of RAM. However, the way of addressing, i.e. addressing mode, differentiates between these two memory spaces. The upper 128 bytes of the 256 byte on-chip RAM can be accessed only using indirect addressing, while the lower 128 bytes can be accessed using direct or indirect mode of addressing. The special function register address space can only be accessed using direct addressing. The address map of the internal RAM and SFR is shown in Fig. 17.10.

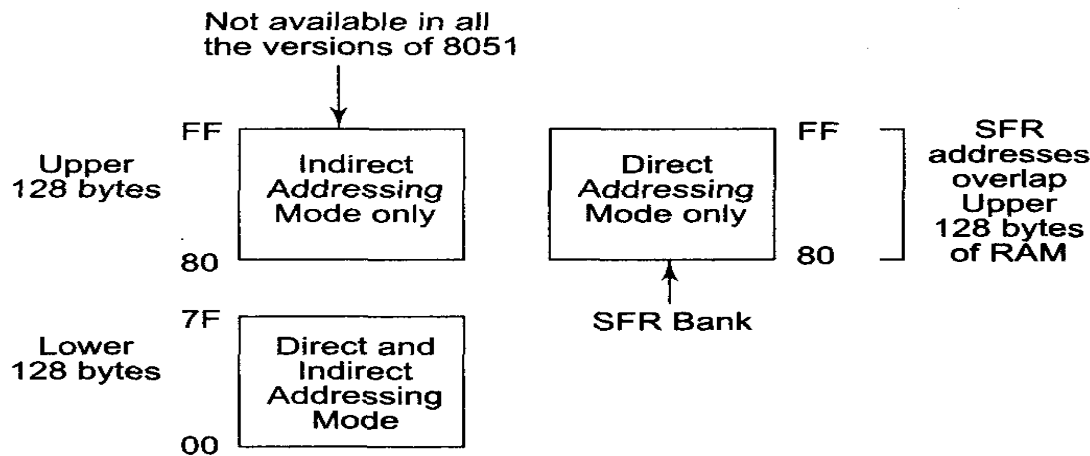



**Fig. 17.10** *Internal Data Memory of 8051*

The lower 128 bytes of RAM whose address map is from 00 to 7FH is functionally organized in three sections. The address block from 00 to 1FH, i.e. the lowest 32 bytes which form the first section, is divided into four banks of 8-bit registers, denoted as bank 00, 01, 10 and 11. Each of these banks contains eight 8-bit registers. The stack pointer gets initialized at address 07H, i.e. the last address of the bank 00, after reset operation. After reset bank 0 is selected by default but the actual stack data is stored from 08H onwards, i.e. bank 01, 10 and 11. These bank addressing bits of the register banks are present in PSW, to select one of these banks at a time. The second section extends from 20H to 2FH, i.e. 16 bytes, which is a bit-addressable block of memory, containing 16 x 8 = 128 bits. Each of these bits can be addressed using the addresses 00 to 7FH. Any of these bits can be accessed in two ways. In the first, its bit number is directly mentioned in the instruction while in the second the bit is mentioned with its position in the respective register byte. For example, the bits 0 to 7 can be referred directly by their numbers, i.e. 0 to 7 or using the notations 20.0 to 20.7 respectively. Note that 20 is the address of the first byte

of the on-chip RAM. The third block of internal memory occupies addresses from 30H to 7FH. This block of memory is a byte addressable memory space. In general, this third block of memory is used as stack memory. All the internal data memory locations are accessed using 8-bit addresses under appropriate modes of addressing. Figure 17.11 shows the categorization of 128 bytes of internal RAM into the different sections.

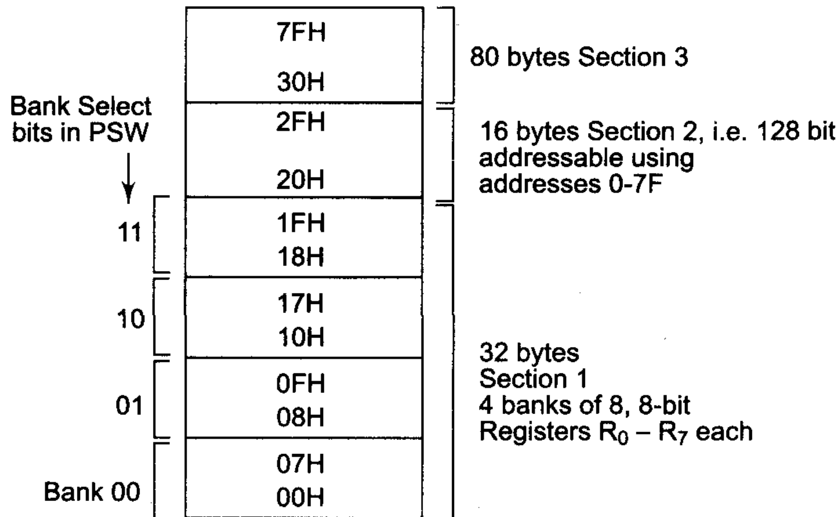



**Fig. 17.11** *Functional Description of Internal Lower 128 Bytes of RAM*

## 2. I/O Addressing

Internally, 8051 has two timers, one serial input/output port and four 8-bit, bit-addressable ports. Some complex applications may require additional I/O devices to be interfaced with 8051. Such external I/O devices are interfaced with 8051 as external memory-mapped devices. In other words, the devices are treated as external memory locations, and they consume external memory addresses. Figure 17.12 shows a system that has external RAM memory of 16 Kbytes, ROM of 16 Kbytes and one chip of 8255 interfaced externally to an 8051 family microcontroller.

Note that, the maps of external program and data memory may overlap, as the memory spaces are logically separated in an 8051 system. As the 8255 is interfaced in external data memory space its addresses are of 16-bits.

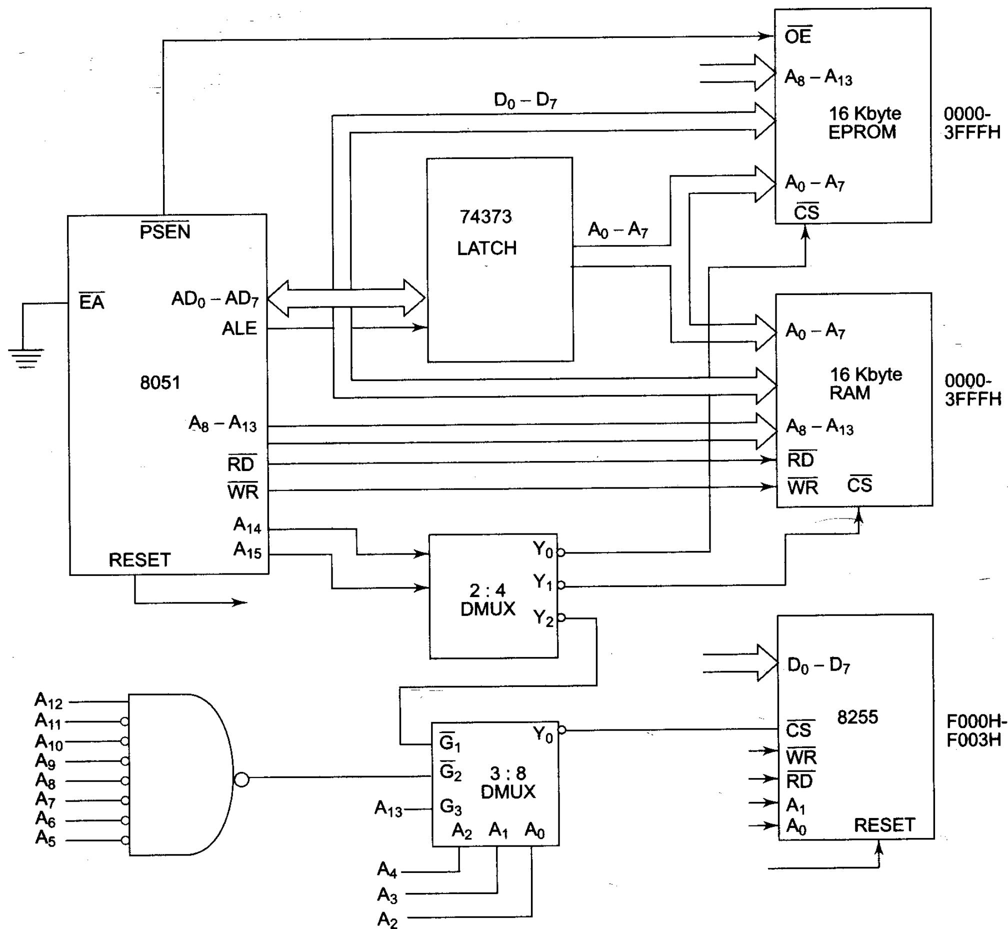



**Fig. 17.12** *Interfacing External Memory and I/O with 8051*

# UNIT – 3

# ADRESSING MODES OF 8051

## ACCESSING MEMORY USING VARIOUS ADDRESSING MODES

We can use direct or register indirect addressing modes to access data stored either in RAM or registers of the 8051. This topic will be discussed thoroughly in this section. We will also show how to access on-chip ROM containing data using indexed addressing mode.

### Direct addressing mode

As mentioned in Chapter 2, there are 128 bytes of RAM in the 8051. The RAM has been assigned addresses 00 to 7FH. The following is a summary of the allocation of these 128 bytes.

1. RAM locations 00 - 1FH are assigned to the register banks and stack.
1. RAM locations 20 - 2FH are set aside as bit-addressable space to save single-bit data. This is discussed in Section 5.3.

2. RAM locations 30 - 7FH are available as a place to save byte-sized data.

Although the entire 128 bytes of RAM can be accessed using direct addressing mode, it is most often used to access RAM locations 30 - 7FH. This is due to the fact that register bank locations are accessed by the register names of RO - R7, but there is no such name for other RAM locations. In the direct addressing mode, the data is in a RAM memory location whose address is known, and this address is given as a part of the instruction. Contrast this with immediate addressing mode, in which the operand itself is provided with the instruction. The "#" sign distinguishes between the two modes. See the examples below, and note the absence of the "#" sign.

```
MOV R0,40H    ;save content of RAM location 40H in R0
MOV 56H,A     ;save content of A in RAM location 56H
MOV R4,7FH    ;move contents of RAM location 7FH to R4
```

As discussed earlier, RAM locations 0 to 7 are allocated to bank 0 registers R0 - R7. These registers can be accessed in two ways, as shown below.

```
MOV A,4       ;is same as
MOV A,R4      ;which means copy R4 into A

MOV A,7       ;is same as
MOV A,R7      ;which means copy R7 into A
```

```
MOV A,2       ;is the same as
MOV A,R2      ;which means copy R2 into A

MOV A,0       ;is the same as
MOV A,R0      ;which means copy R0 into A
```

The above examples should reinforce the importance of the "#" sign in 8051 instructions. See the following code.

```
MOV R2,#5   ;R2 with value 5
MOV A,2     ;copy R2 to A (A=R2=05)
MOV B,2     ;copy R2 to B (B=R2=05)
MOV 7,2     ;copy R2 to R7
            ;since "MOV R7,R2" is invalid
```

Although it is easier to use the names RQ - R7 than their memory addresses, RAM locations 3 OH to 7FH cannot be accessed in any way other than by their addresses since they have no names.

**SFR registers and their addresses**

Among the registers we have discussed so far, we have seen that RO - R7 are part of the 128 bytes of RAM memory. What about registers A, B, PSW, and DPTR? Do they also have addresses? The answer is yes. In the 8051, registers A, B, PSW, and DPTR are part of the group of registers commonly referred to as SFR (special function registers). There are many special function registers and they are widely used, as we will discuss in future chapters. The SFR can be accessed by their names (which is much easier) or by their addresses. For example, register A has address EOH, and register B has been designated the address FOH, as shown in Table 5-1. Notice how the following pairs of instructions mean the same thing.

```
MOV 0E0H,#55H  ;is the same as
MOV A,#55H     ;which means load 55H into A (A=55H)

MOV 0F0H,#25H  ;is the same as
MOV B,#25H     ;which means load 25H into B (B=25H)

MOV 0E0H,R2    ;is the same as
MOV A,R2       ;which means copy R2 into A

MOV 0F0H,R0    ;is the same as
MOV B,R0       ;which means copy R0 into B

MOV P1, A      ;is the same as
MOV 90H,A      ;which means copy reg A to P1
```

Table  lists the 8051 special function registers (SFR) and their addresses. The following two points should be noted about the SFR addresses.

1. The special function registers have addresses between 80H and FFH. These addresses are above 80H, since the addresses 00 to 7FH are addresses of RAM memory inside the 8051.
2. Not all the address space of 80 to FF is used by the SFR. The unused locations 80H to FFH are reserved and must not be used by the 8051 programmer.

Regarding direct addressing mode, notice the following two points: (a) the address value is limited to one byte, 00 - FFH, which means this addressing mode is limited to accessing RAM locations and registers located inside the 8051. (b) if you examine the 1st file for an Assembly language program, you will see that the SFR registers' names are replaced with their addresses as listed in Table 5-1.

Write code to send 55H to ports P1 and P2, using (a) their names, (b) their addresses.

**Solution:**

(a)
```
MOV A,#55H      ;A=55H
MOV P1,A        ;P1=55H
MOV P2,A        ;P2=55H
```

(b) From Table 5-1, P1 address = 90H; P2 address = A0H
```
MOV A,#55H      ;A=55H
MOV 90H,A       ;P1=55H
MOV 0A0H,A      ;P2=55H
```

**Table : 8051 Special Function Register (SFR) Addresses**

| Symbol | Name | Address |
|---|---|---|
| ACC* | Accumulator | 0E0H |
| B* | B register | 0F0H |
| PSW* | Program status word | 0D0H |
| SP | Stack pointer | 81H |
| DPTR | Data pointer 2 bytes | |
| DPL | Low byte | 82H |
| DPH | High byte | 83H |
| 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 |
| T2CON* | Timer/counter 2 control | 0C8H |
| T2MOD | Timer/counter mode control | 0C9H |
| TH0 | Timer/counter 0 high byte | 8CH |
| TL0 | Timer/counter 0 low byte | 8AH |
| TH1 | Timer/counter 1 high byte | 8DH |
| TL1 | Timer/counter 1 low byte | 8BH |
| TH2 | Timer/counter 2 high byte | 0CDH |
| TL2 | Timer/counter 2 low byte | 0CCH |
| RCAP2H | T/C 2 capture register high byte | 0CBH |
| RCAP2L | T/C 2 capture register low byte | 0CAH |
| SCON* | Serial control | 98H |
| SBUF | Serial data buffer | 99H |
| PCON | Power control | 87H |

* Bit-addressable

**Example 1**

**Stack and direct addressing mode**

Another major use of direct addressing mode is the stack. In the 8051 family, only direct addressing mode is allowed for pushing onto the stack. Therefore, an instruction such as "PUSH A" is invalid. Pushing the accumulator onto the stack must be coded as "PUSH OEOH" where OEOH is the address of register A. Similarly, pushing R3 of bank 0 is coded as "PUSH 03". Direct addressing mode must be used for the POP instruction as well. For example, "POP 04" will pop the top of the stack into R4 of bank 0.

**Example** 2

Show the code to push R5, R6, and A onto the stack and then pop them back them into R2, R3, and B, where register B = register A, R2 = R6, and R3 = R5.

**Solution:**

```
PUSH 05         ;push R5 onto stack
PUSH 06         ;push R6 onto stack
PUSH 0E0H       ;push register A onto stack
POP  0F0H       ;pop top of stack into register B
                ;now register B = register A
POP  02         ;pop top of stack into R2
                ;now R2 = R6
POP  03         ;pop top of stack into R3
                ;now R3 = R5
```

**Register indirect addressing mode**

In the register indirect addressing mode, a register is used as a pointer to the data. If the data is inside the CPU, only registers RO and Rl are used for this purpose. In other words, R2 - R7 cannot be used to hold the address of an operand located in RAM when using this addressing mode. When RO and Rl are used as **pointers, that is, when they hold the addresses of RAM locations, they must be preceded by the "@" sign, as shown below.**

```
MOV A,@R0 ;move contents of RAM location whose
          ;address is held by R0 into A
MOV @R1,B ;move contents of B into RAM location
          ;whose address is held by R1
```

**Notice that RO (as well as Rl) is preceded by the "@" sign. In the absence of the "@" sign, MOV will be interpreted as an instruction moving the contents of register RO to A, instead of the contents of the memory location pointed to by RO.**

Example 3

Write a program to copy the value 55H into RAM memory locations 40H to 45H using
(a) direct addressing mode,
(b) register indirect addressing mode without a loop, and
(c) with a loop.

**Solution:**

(a)

```
      MOV A,#55H       ;load A with value 55H
      MOV 40H,A        ;copy A to RAM location 40H
      MOV 41H,A        ;copy A to RAM location 41H
      MOV 42H,A        ;copy A to RAM location 42H
      MOV 43H,A        ;copy A to RAM location 43H
      MOV 44H,A        ;copy A to RAM location 44H
```

(b)

```
      MOV A,#55H       ;load A with value 55H
      MOV R0,#40H      ;load the pointer. R0=40H
      MOV @R0,A        ;copy A to RAM location R0 points to
      INC R0           ;increment pointer. Now R0=41H
      MOV @R0,A        ;copy A to RAM location  R0 points to
      INC R0           ;increment pointer. Now R0=42H
      MOV @R0,A        ;copy A to RAM location  R0 points to
      INC R0           ;increment pointer. Now R0=43H
      MOV @R0,A        ;copy A to RAM location R0 points to
      INC R0           ;increment pointer. Now R0=44H
      MOV @R0,A
```

(c)

```
        MOV   A,#55    ;A=55H
        MOV   R0,#40H ;load pointer. R0=40H, RAM address
        MOV   R2,#05   ;load counter, R2=5
AGAIN:  MOV   @R0,A    ;copy 55H to RAM location R0 points to
        INC   R0       ;increment R0 pointer
        DJNZ R2,AGAIN      ;loop until counter = zero
```

**Advantage of register indirect addressing mode**

One of the advantages of register indirect addressing mode is that it makes accessing data dynamic rather than static as in the case of direct addressing mode. Example 5-3 shows two cases of copying 55H into RAM locations 40H to 45H. Notice in solution (b) that there are two instructions that are repeated numerous times. We can create a loop with those two instructions as shown in solution (c). Solution (c) is the most efficient and is possible only because of register indirect addressing mode. Looping is not possible in direct addressing mode. This is the main difference between the direct and register indirect addressing modes.

**Example 5-4**

Write a program to clear 16 RAM locations starting at RAM address 60H.

**Solution:**

```
        CLR     A          ;A=0
        MOV     R1,#60H    ;load pointer. R1=60H
        MOV     R7,#16     ;load counter, R7=16 (10 in hex)
AGAIN:  MOV     @R1,A      ;clear RAM location R1 points to
        INC     R1         ;increment R1 pointer
        DJNZ    R7,AGAIN   ;loop until counter = zero
```

An example of how to use both RO and Rl in the register indirect addressing mode in a block transfer is given in Example 5.

**Example 5-5**

Write a program to copy a block of 10 bytes of data from RAM locations starting at 35H to RAM locations starting at 60H.

**Solution:**

```
        MOV  R0,#35H    ;source pointer
        MOV  R1,#60H    ;destination pointer
        MOV  R3,#10     ;counter
BACK:   MOV  A,@R0      ;get a byte from source
        MOV  @R1,A      ;copy it to destination
        INC  R0         ;increment source pointer
        INC  R1         ;increment destination pointer
        DJNZ R3,BACK    ;keep doing it for all ten bytes
```

**Limitation of register indirect addressing mode in the 8051**

As stated earlier, RO and Rl are the only registers that can be used for pointers in register indirect addressing mode. Since RO and Rl are 8 bits wide, their use is limited to accessing any information in the internal RAM (scratch pad memory of 30H - 7FH, or SFR). However, there are times when we need to access data stored in external RAM or in the code space of on-chip ROM. Whether accessing externally connected RAM or on-chip ROM, we need a 16-bit pointer. In such cases, the DPTR register is used, as shown next.

**Indexed addressing mode and on-chip ROM access**

Indexed addressing mode is widely used in accessing data elements of look-up table entries located in the program ROM space of the 8051. The instruction used for this purpose is "MOVC A, @A+DPTR". The 16-bit register DPTR and register A are used to form the address of the data element stored in on-chip ROM. Because the data elements are stored in the program (code) space ROM of the 8051, the instruction MOVC is used instead of MOV. The "C" means code. In this instruction the contents of A are added to the 16-bit register DPTR to form the 16-bit address of the needed data. See Example 5-6.

**Example 6**

In this program, assume that the word "USA" is burned into ROM locations starting at 200H, and that the program is burned into ROM locations starting at 0. Analyze how the program works and state where "USA" is stored after this program is run.

**Solution:**

```
      ORG   0000H           ;burn into ROM starting at 0
      MOV   DPTR,#200H      ;DPTR=200H look-up table address
      CLR   A               ;clear A(A=0)
      MOVC  A,@A+DPTR       ;get the char from code space
      MOV   R0,A            ;save it in R0
      INC   DPTR            ;DPTR=201 pointing to next char
      CLR   A               ;clear A(A=0)
      MOVC  A,@A+DPTR       ;get the next char
      MOV   R1,A            ;save it in R1
      INC   DPTR            ;DPTR=202 pointing to next char
      CLR   A               ;clear A(A=0)
      MOVC  A,@A+DPTR       ;get the next char
      MOV   R2,A            ;save it in R2
HERE:SJMP HERE              ;stay here

;Data is burned into code space starting at 200H
          ORG 200H
MYDATA:   DB "USA"
          END               ;end of program
```

In the above program ROM locations 200H - 202H have the following contents. 200=('U') 201=('S') 202=('A")

We start with DPTR = 200H, and A = 0. The instruction "MOVC A, @A+DPTR" moves the contents of ROM location 200H (200H + 0 = 200H) to register A. Register A contains 55H, the ASCII value for "U". This is moved to RO. Next, DPTR is incremented to make DPTR = 201H. A is set to 0 again to get the contents of the next ROM location 201H, which holds character "S". After this program is run, we have RO = 55H, Rl = 53H, and R2 = 41H, the ASCII values for the characters "U", "S" and "A".

**Example 7**

Assuming that ROM space starting at 250H contains "America", write a program to transfer the bytes into RAM locations starting at 40H.

**Solution:**

```
;(a) This method uses a counter
          ORG   0000
          MOV   DPTR,#MYDATA    ;load ROM pointer
          MOV   R0,#40H         ;load RAM pointer
          MOV   R2,#7           ;load counter
  BACK:   CLR   A               ;A = 0
          MOVC  A,@A+DPTR       ;move data from code space
          MOV   @R0,A           ;save it in RAM
          INC   DPTR            ;increment ROM pointer
          INC   R0              ;increment RAM pointer
          DJNZ  R2,BACK         ;loop until counter=0
  HERE:   SJMP  HERE

 ;----------On-chip code space used for storing data
          ORG   250H
  MYDATA: DB    "AMERICA"
          END

;(b) This method uses null char for end of string
          ORG   0000
          MOV   DPTR,#MYDATA    ;load ROM pointer
          MOV   R0,#40H         ;load RAM pointer
  BACK:   CLR   A               ;A=0
          MOVC  A,@A+DPTR       ;move data from code space
          JZ    HERE            ;exit if null character
          MOV   @R0,A           ;save it in RAM
          INC   DPTR            ;increment ROM pointer
          INC   R0              ;increment RAM pointer
          SJMP  BACK            ;loop
  HERE:   SJMP  HERE

 ;----------On-chip code space used for storing data
          ORG   250H
  MYDATA: DB    "AMERICA",0     ;notice null char for
                                ;end of string
          END
```

Notice the null character, 0, indicating the end of the string, and how we use the JZ instruction to detect that.

## Look-up table and the MOVC instruction

The look-up table is a widely used concept in microprocessor programming. It allows access to elements of a frequently used table with minimum operations.

### Example 8

Write a program to get the $x$ value from PI and send $x^2$ to P2, continuously.

**Solution:**

```
          ORG  0
          MOV  DPTR,#300H       ;load look-up table address
          MOV  A,#0FFH          ;A=FF
          MOV  P1,A             ;configure P1 as input port
BACK:     MOV  A,P1             ;get X
          MOVC A,@A+DPTR        ;get X squared from table
          MOV  P2,A             ;issue it to P2
          SJMP BACK             ;keep doing it

          ORG  300H
XSQR_TABLE:
          DB   0,1,4,9,16,25,36,49,64,81
          END
```

Notice that the first instruction could be replaced with "MOV DPTR,#XSQR_TABLE"

**Example 5-9**

Answer the following questions for Example 5-8.
(a) Indicate the content of ROM locations 300 - 309H.
(b) At what ROM location is the square of 6, and what value should be there?
(c) Assume that P1 has a value of 9: What value is at P2 (in binary)?

**Solution:**

(a) All values are in hex.

```
300 = (00)     301 = (01)    302 = (04)    303 = (09)
304 = (10)     4 x 4 = 16 = 10 in hex
305 = (19)     5 x 5 = 25 = 19 in hex
306 = (24)     6 x 6 = 36 = 24H
307 = (31)     308 = (40)    309 = (51)
```

(b) 306H; it is 24H

(c) 01010001B, which is 51H and 81 in decimal ($9^2 = 81$).

In addition to being used to access program ROM, DPTR can be used to access memory externally connected to the 8051. Another register used in indexed addressing mode is the program counter.

In many of the examples above, the MOV instruction was used for the sake of clarity, even though one can use any instruction as long as that instruction supports the addressing mode. For example, the instruction "ADD A, @RO" would add the contents of the memory location pointed to by RO to the contents of register A. We will see more examples of using addressing modes with various instructions in the next few chapters.

**Indexed addressing mode and MOVX instruction**

As we have stated earlier, the 8051 has 64K bytes of code space under the direct control of the Program Counter register. We just showed how to use the MOVC instruction to access a portion of this 64K-byte code space as data memory space. In many applications the size of program code does not leave any room to share the 64K-byte code space with data. For this reason the 8051 has another 64K bytes of memory space set aside exclusively for data storage. This data memory space is referred to as *external memory* and it is accessed only by the MOVX instruction. In other words, the 8051 has a total of 128K bytes of memory space since 64K bytes of code added to 64K bytes of data space gives us 128K bytes. One major difference between the code space and data space is that, unlike code space, the data space cannot be shared between code and data. This is such an important topic that we have dedicated an entire chapter to it: Chapter 14.

**Accessing RAM Locations 30 - 7FH as scratch pad**

As we have seen so far, in accessing registers RO - R7 of various banks, it is much easier to refer to them by their RO - R7 names than by their RAM locations. The only problem is that we have only 4 banks and very often the task of bank switching and keeping track of register bank usage is tedious and prone to errors. For this reason in many applications we use RAM locations 30 - 7FH as scratch pad and leave addresses 8 - 1FH for stack usage. That means that we use RO - R7 of bank 0, and if we need more registers we simply use RAM locations 30-7FH. Look at Example 5-10.

**Example 10**

Write a program to toggle P1 a total of 200 times. Use RAM location 32H to hold your counter value instead of registers R0 - R7.

**Solution:**

```
      MOV   P1,#55H  ;P1=55H
      MOV   32H,#200 ;load counter value into RAM loc 32h
LOP1:CPL    P1       ;toggle P1
      ACALL DELAY
      DJNZ  32H,LOP1 ;repeat 200 times
```

**TIMERS AND COUNTERS**

On-chip timing/counting facility has proved the capabilities of the micrcontrollers for implementing the real time applications. These include pulse counting, frequency measurement, pulse width measurement, baud rate generation, etc. Having sufficient number of timer/counters may be a need in a certain design application. As seen in the first chapter, 8051 has two 16-bit timer/counters. Before discussing 8051 timer/counters, it is necessary to see the exact difference between a timer and a counter. A timer counts machine cycles and provides a reference time delay or a clock. A machine cycle of 8051 consists of 12 oscillator periods or the counting rate is 1/12 of the oscillator frequency. At 12 MHz, the clocking period will be equal to 1 μs. Let us now see, the counting function. A counter of 8051 is incremented in response to a transition from '1' to '0' at its corresponding external pin (either T0 or T1). Thus, the counter output will be a count or a number representing the occurrence of such '1' to '0' transitions at the external pin. For counting function, 8051 takes 2 machine cycles or 24 oscillator periods to detect a '1' to '0' transition at Pin T0 or T1. When a timer or counter overflows from FFFFH to 0000H, it sets a flag and generates an interrupt. The 16 bits of timer are referred as higher byte THx and the lower byte TLx. Thus, TH1 is the higher byte of timer 1 and TL1 is the lower byte of timer 1. 'x' can be 0 or1 (or 2 in case of 8032/52).

**Timer/Counter Modes**

There are four timer modes in 8051. A timer or counter function and modes are selected by writing appropriate bits in the SFR, called the timer mode register (TMOD), whereas the control of timer/counter operation is done through the SFR, called the timer control register (TCON). These SFRs are shown in Figs. 6.3 and 6.4. Now, the question is how exactly to

| TCON.7 | TCON.6 | TCON.5 | TCON.4 | TCON.3 | TCON.2 | TCON.1 | TCON.0 |
|---|---|---|---|---|---|---|---|
| TF1 | TR1 | TF0 | TR0 | IE1 | IT1 | IE0 | IT0 |

| | | |
|---|---|---|
| TCON.7 | TF1 | Timer 1 overflow flag, set when timer/counter overflows |
| TCON.6 | TR1 | Timer 1 run control bit |
| TCON.5 | TF0 | Timer 0 overflow flag, set when timer/counter 0 overflows |
| TCON.4 | TR0 | Timer 0 run control bit |
| TCON.3 | IE1 | Interrupt 1 |
| TCON.2 | IT1 | Timer interrupt 1 |
| TCON.1 | IE0 | Interrupt 0 flag |
| TCON.0 | IT0 | Timer 0 interrupt, IT0 = 0, low level trigger, IT0 = 1,edge trigger (falling edge) |

**Fig. 6.3** Timer Control Register (TCON) (Bit Addressable)

Configure timer/counters as a timer or counter. As seen from Fig. 6.4, the TMOD bit $C/\overline{T}$, defines

this operation.

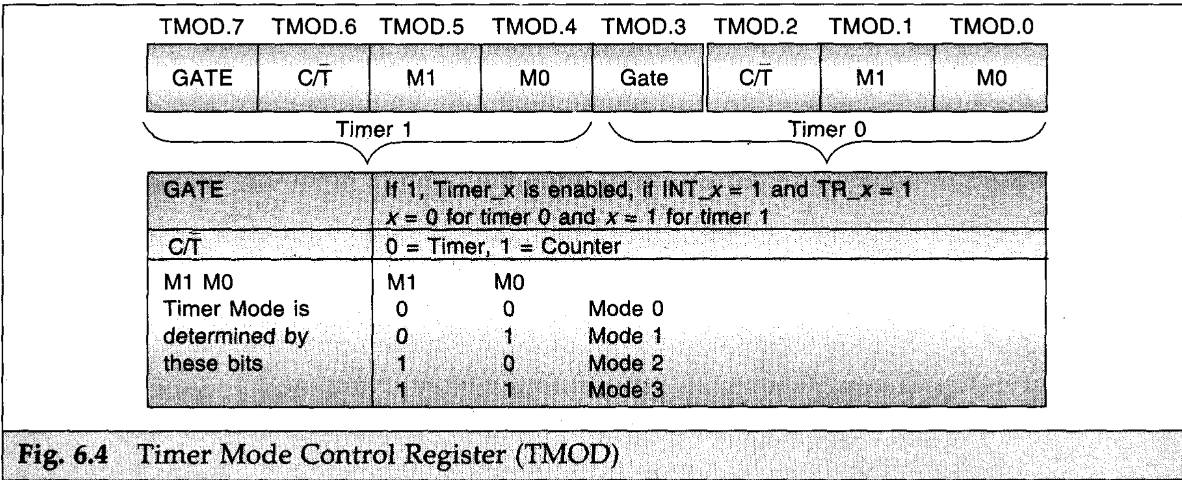


**Fig. 6.4** Timer Mode Control Register (TMOD)

## A). Mode 0

In mode 0, the timer is 13-bit wide. This mode is same for timer 0 and timer 1. When the count overflows, it sets the timer interrupt flag (TF1 for timer 1 and TF0 for timer 0). To start timer 0, TR0 bit in TCON is required to be set. Using the upper byte TH1 (or TH0) and the lower 5 bits of TL1 (or TL0) forms the 13 bits. This is shown in Fig. 6.5 and Fig. 6.6.

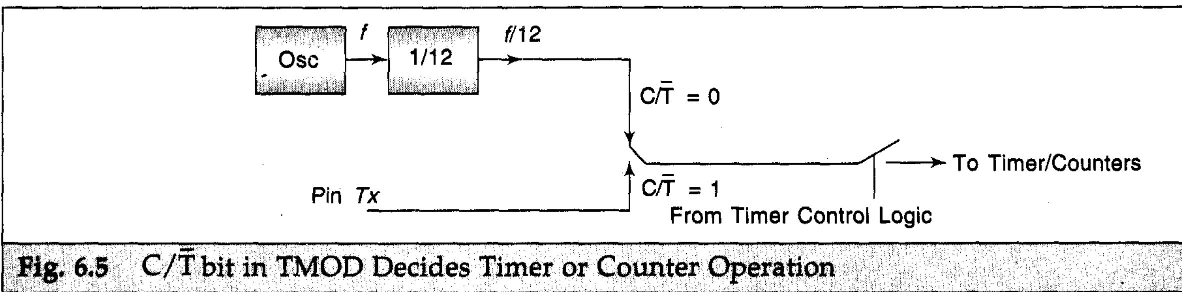


**Fig. 6.5** C/T bit in TMOD Decides Timer or Counter Operation

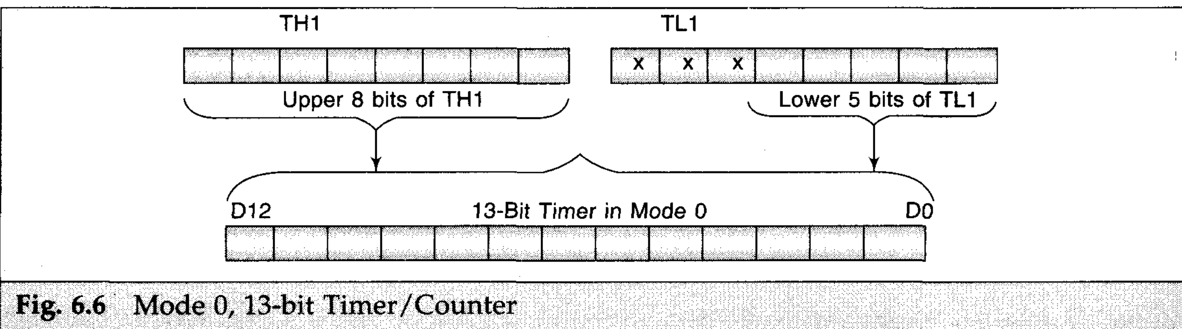


**Fig. 6.6** Mode 0, 13-bit Timer/Counter

```
; Programming Example #6.4
; Initializing timer 1 in mode 0
MOV TMOD, # 1000 0000 B
; Timer 1 in mode 0, Timer 0 in mode 0, both are configured as timers,
; Timer 1 is controlled by the external Pin 13 INT1 (Note GATE=1),
whereas the system clock clocks timer 0;
SETB TR1            ; Start timer 1
SETB TR0            ; Start timer 0
CLR TR1             ; Stop timer 1
SJMP $              ; Infinite loop
```

Programming Example #6.4 initializes timer 1 in mode 0. In the above program, timer 1 is configured as a timer in mode 0. Observe that bit TMOD.7 in TMOD is set to 1. This is the GATE bit. If this is set to 1 and TR1 is 1, then the timer 1 is controlled by the external input at Pin 13 ($\overline{\text{INT1}}$). This can be seen from Fig. 6.7. When GATE is 0, then, it is only TR1 which enables the timer.

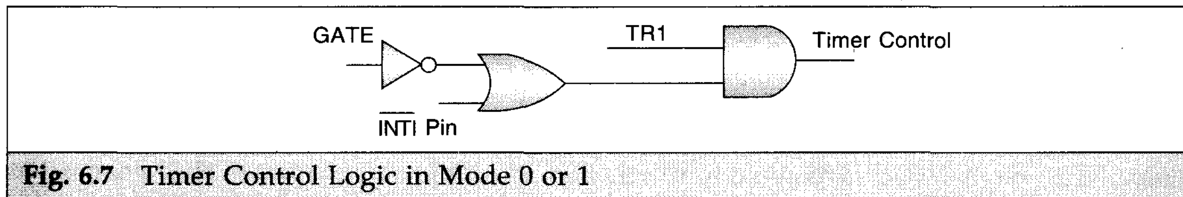



**Fig. 6.7** Timer Control Logic in Mode 0 or 1

**B). Mode 1**

Mode 1 is same as mode 0, except the timers are 16 bits wide. Mode 1 is again the same for timer 0 and timer 1. The maximum count in this mode is FFFFH. To initialize timer 1 in mode 1, see Programming Example #6.5.

```
; Programming Example #6.5
; Initializing timer1 in mode 1
MOV TMOD, # 0001 0000 B        ; Timer 1 in mode 1
SETB TR1                       ; As the GATE bit is zero, TR1 can fully;
                               ; control the timer operation
                               ; Hence to start timer 1, TR1 is set to 1.
SJMP $                         ; Infinite loop
```

If initialized, the timer overflow can generate an interrupt. Consider one such program segment (Programming Example #6.6) to initialize the timer 1 interrupt. Note that it is always

advisable to initialize the stack pointer before going for a main program, because the default value of SP 07H may not be suitable in general. This is also the address of register R7, and if any register bank switching is done, it can overwrite some useful register contents.

```
; Programming Example #6.6
; Program to initialize timer 1 in mode 1
MOV SP, #54H                    ; Initialize the stack pointer
MOV TMOD, # 0001 0000 B         ; Timer1 in mode 1
SETB ET1                        ; Enable timer 1 interrupt
SETB TR1                        ; Start timer 1
SETB EA                         ; Enable all
SJMP $                          ; Infinite loop (Jump here)
```

Note that simply setting only ET1 bit in IE register will not enable the timer interrupt. In addition, it is necessary to set the EA bit in IE. This program will start timer 1, and when it overflows, timer 1 interrupt is generated, which will cause the program counter to jump to the vector location 001B H.

**C). Mode 2**

This operation is again the same for timer 0 and timer 1. Consider timer 1 in mode 2. Timer register is configured as an 8-bit counter TL1. Overflow from TL1 sets the flag TF1, and it loads TL1 with the contents of TH1. The software can preload TH1. This mode of timer 1 or timer 0 thus supports the automatic reload operation. Mode 2 auto-reload mechanism is shown in Fig. 6.8. Timer control logic is again the same as that of mode 0 or 1.

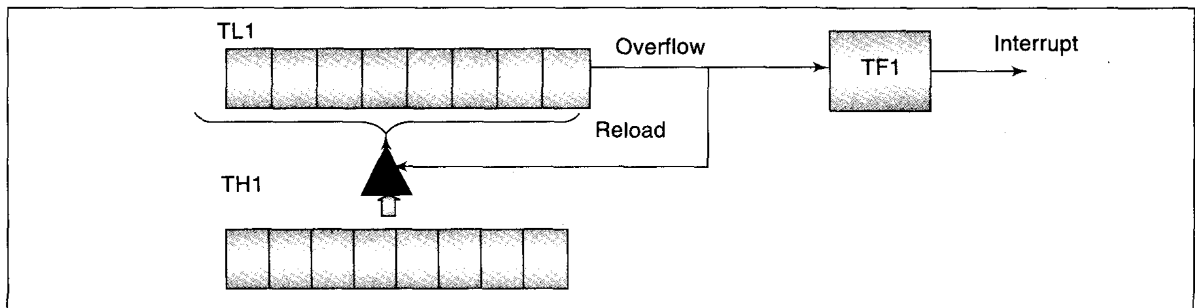



**Fig. 6.8** Mode 2 Auto-reload Mechanism of Timer

Let us now write the initialization program for timer 0 in mode 2 as shown in the Programming Example #6.7. The program must load TMOD and then the auto-reload value must be written in the timer high byte. Further, the starting count will also be the same as that of the reload value in general, but it is not so strict since this is applicable for the very first overflow.

However, it is very essential to load the timer high byte with the auto-reload value, otherwise the timer after each overflow will start from 00H.

```
; Programming Example #6.7
; Initializing timer 0 in mode 2
MOV TMOD, # 0000 0010 B      ; Load TMOD for timer 0 in mode 2
MOV THO, # 33H               ; Load THO with preset value to be reloaded
MOV TLO, # 33 H              ; Starting count = preset value
SETB TRO                     ; Start timer 0
SJMP $
```

Mode 2 is very commonly used for baud rate generation for serial port operation, or where a constant frequency square wave output is needed. The frequency or baud rate can be controlled using the preloaded value in THx register. The maximum delay generated using mode 2 will be corresponding to the auto-reload value of 00H. Thus, at 12 MHz clock, this would generate the maximum delay of 256 μs. If one can write an instruction to toggle any of the port pins, a square wave output on that pin can be seen on the oscilloscope. Consider this program to generate a 2 kHz (0.5 ms period) square waveform on pin Pl.0 (Pin 1), as shown in Programming Example #6.8. The reload count will be corresponding to 0.25mS. At 12 MHz, this will be (256-250) equal to 06H.

```
;Programming Example #6.8
; Program to generate 2 kHz square waves on pin P1.0 of port 1
            MOV SP, # 54H; Initialize the stack pointer
            MOV TMOD, 0000 0010 B; Timer 0 in mode 2 (Auto-reload mode)
            MOV THO, # 06H; Preload value for 2 kHz square waves
            MOV TLO, # 06H; Starting value in timer register TLO
            SETB TRO; Start timer 0
      LOOP: JB TFO, COMPLMNT
            SJMP LOOP
  COMPLMNT: CPL P1.0; Toggle bit P1.0
            SJMP LOOP
```

The same program could be written using timer interrupt also. Let us see how to do it! Note that timer 0 interrupt has been enabled at the time of starting the timer. The timer 0 is initialized in mode 2 or auto-reload mode. TH0 and TL0 both are initialized to the count 06H corresponding to the 2 kHz frequency of square waves. The main program is over once timer 0 is started. But notice the instruction SJMP $. This instruction is to jump at the same address and generate an infinite loop. The effect is same as the instruction "LABEL: SJMP LABEL". This program is shown in the Programming Example #6.9.

```
;Programming Example #6.9
; Program to generate 2 kHz square waves on pin P1.0 of port 1 (Use of Interrupt)
            ORG 0000H
            AJMP STRT                   ; Main program is at STRT
            ORG 000B H
            AJMP INT_TF0                ; ISR is at INT_TF0
      STRT: MOV SP, # 54H               ; Initialize the stack pointer
            SETB ET0
            SETB EA
            MOV TMOD, 0000 0010 B       ; Timer 0 in mode 2(Auto-reload mode)
            MOV TH0, # 06H              ; Preload value for 2 kHz square waves
            MOV TL0, # 06H              ; Starting value in timer register TL0
            SETB TR0                    ; Start timer 0
            SJMP $                      ; Infinite loop here
   INT_TF0: CPL P1.0                    ;
            RETI
            END
```

Further, we have written an interrupt service routine (ISR) in which we just complement the bit Pl.0. The ISR ends with RETI instruction. After the execution of this instruction, the CPU will again be in the infinite loop, from where it was interrupted. Note that timer 0, once started, is not made off or on afterwards, which is not needed also due to auto-reload feature.

**D). Mode 3**

In this mode, timer 1 has a passive role of holding its count. In effect, it looks like as the one who keeps TR1 = 0. Now, timer 0 bytes TH0 and TL0 are used as two separate timers. Because of this, mode 3 is also called as split timer mode. TH0 is locked into timer operation and simply counts the machine cycles. After overflowing, it sets the flag TF1.

TL0 can be configured and controlled by using C/$\overline{\mathrm{T}}$, GATE, TR0, INT0, and TF0. Note that TR1 controls the operation of TH0 timer, now the question remains how to control timer 1? Timer 1 can be used in a different manner, for any application that does not require the interrupt operation; like for generating the baud rate for serial port operation. When timer 0 is in mode 3, one can just control its operation by switching it out or into its mode 3 using TMOD settings. Thus, in mode 3 it resembles like 8051 having 3 timer/counters.

Note that in timer mode 3, timer 1 is a 16-bit timer and TH0, TL0 two 8-bit timers.

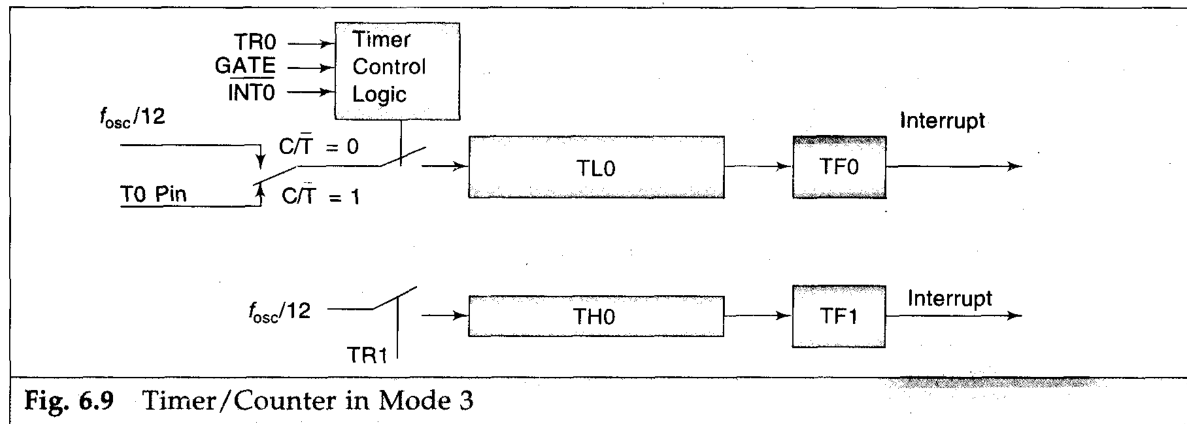



**Fig. 6.9** Timer/Counter in Mode 3

## SERIAL COMMUNICATION

Serial data transmission is very commonly used for digital data communication. Its main advantage is that the number of wires needed is reduced as compared to that in parallel communication. 8051 supports a full duplex serial port. Full duplex means, it can transmit and receive a byte simultaneously. 8051 has TXD and RXD pins for transmission and reception of serial data respectively. The 8051 serial communication is supported by RS232 standard. The term "RS" stands for Recommended Standard. Communication between two microcontrollers and multiprocessor communication is also possible. The start and stop bits are used to synchronize the serial receivers. The data byte is always transmitted with least-significant-bit first. For error checking purpose, it is possible to include a parity bit as well, just prior to the stop bit. Thus, the bits are transmitted at specific time intervals determined by the baud rate. For error-free serial communication, it is necessary that the baud rate, the number of data bits, the number of stop bits, and the presence or absence of a parity bit along with its status be the same at the transmitter and receiver ends.

The basic mechanism of serial transmission is that a data byte in parallel form is converted into serial data stream. Along with some more bits like start, stop and parity bits, a serial data frame is sent over a line. There are four modes of serial data transmission in 8051. In each of these modes, it is important to decide the baud rate, the way in which serial data frame is sent and any other information, etc.

| SCON.7 | SCON.6 | SCON.5 | SCON.4 | SCON.3 | SCON.2 | SCON.1 | SCON.0 |
|---|---|---|---|---|---|---|---|
| SM0 | SM1 | SM2 | REN | TB8 | RB8 | TI | RI |

| **Bit address** | **SCON bit** | **Description** |
|---|---|---|
| 9FH | SM0 | Serial Communications Mode |
| 9EH | SM1 | |
| 9DH | SM2 | In modes 2 and 3, if set, this will enable multiprocessor communication |
| 9CH | REN | (Receive enable) Enables serial reception |
| 9BH | TB8 | This is the 9th data bit that is transmitted in modes 2 and 3 |
| 9AH | RB8 | 9th data bit that is received in modes 2 and 3. It is not used in mode 0. In mode 1, if SM2 = 0, then RB8 is the stop bit that is received. |
| 99H | TI | Transmit interrupt flag, set by hardware, must be cleared by software |
| 98H | RI | Receive interrupt flag, set by hardware, must be cleared by software |

| SM0 | SM1 | MODE | **Description** | Baud Rate |
|---|---|---|---|---|
| 0 | 0 | 0 | 8-bit Shift register mode | $f_{osc}/12$ |
| 0 | 1 | 1 | 8-bit UART | variable (set by timer 1) |
| 1 | 0 | 2 | 9-bit UART | $f_{osc}/164$ or $f_{osc}/32$ |
| 1 | 1 | 3 | 9-bit UART | variable (set by timer 1) |

**Fig. 6.10** Serial Control Register (SCON)

What is common in all these modes is the use of the SFR called "SBUF", for transmission as well as reception. The data to be transmitted must be transferred to SBUF. One more SFR that controls the serial communication operation is the serial control register SCON. Details of SCON are shown in Fig. 6.10. Bits SM0 and SM1 in SCON define serial port mode. Bit SM2 enables the multiprocessor communication in modes 2 and 3. Transmission is initiated by the execution of any instruction that uses SBUF as the destination.

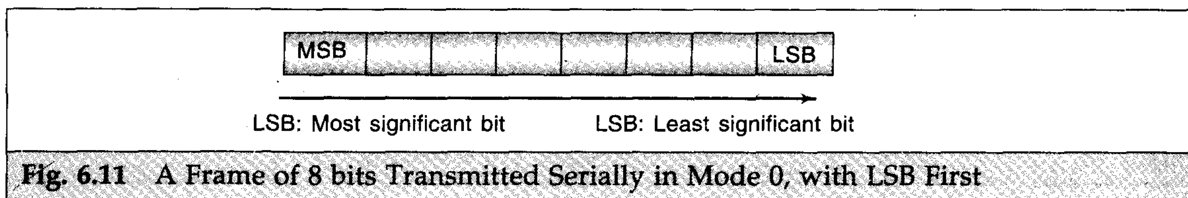



**Fig. 6.11** A Frame of 8 bits Transmitted Serially in Mode 0, with LSB First

**Serial Communication Modes**

There are four modes in which 8051 serial port can be configured.

**A. Mode 0**

This is also called as shift register mode. Only RXD is the pin through which data enter or exits. TXD pin outputs the shift clock only. Eight data bits are transmitted or received. The baud rate is fixed and is totally determined by the system clock frequency. If $f_{osc}$ is the clock frequency, then $f_{osc}/12$ will be the baud rate.

To see exactly how the operation of serial data transfer takes place in mode 0, see Programming Example #6.11.

```
; Programming Example #6.11
; Serial transmission mode 0
        ORG 0000H               ; Program starts at 0000H
        MOV SCON, #0000 0000B   ; Mode 0
; Now write the data byte to be transmitted in SBUF
        MOV SBUF, #44H          ; Transmit 0100 0100 binary
; After transmission, TI flag in SCON will be set by hardware, this can be
; tested for assuring the transmission operation
        Here: JNB TI, Here
                                ; Wait till all 8 bits are transmitted
                                ; Remember TI flag must be cleared
        CLR TI; TI flag is reset
```

**B. Mode 1**

In mode 1, 10 bits are transmitted through TXD pin or received through RXD pin. There is a start bit (0), then 8 data bits (LSB first) and a stop bit (1). This is shown in Fig. 6.12. On receiving, the stop bit goes into RB8 in SCON. The baud rate is variable and is determined by the timer 1 overflow rate. Therefore, before using this mode, one has to initialize timer 1. A simple program to initialize serial port in mode 1 is given in Programming Example #6.12. The baud rate is calculated using the formula:

Baud rate = 2SMOD/32 x (Timer 1 overflow rate) (6.1)

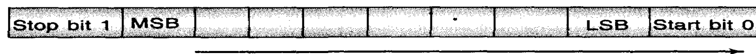



**Fig. 6.12** Ten Bits Transmitted in Mode 1 of Serial Transmission

```
;Programming Example #6.12
; Initializing the serial port in mode 1
ORG 0000H
MOV SP, #51H
; Note that SMOD is '0' after RESET
MOV SCON, #0100 0000B          ; Serial port in mode 1
MOV TMOD, #0010 0000B          ; Timer 1 in auto-reload mode
MOV TH1, #230 D                ; Baud rate =1200 at 12 MHz
SETB TR1                       ; Start timer
MOV SBUF, #56H
JNB TI, $                      ; Wait till the transmission is over
CLR TI                         ; Reset bit TI after transmission
```

If timer 1 is configured in auto-reload mode (or mode 2), with reload value in TH1, after each overflow, contents of TH1 will be loaded into TL1. This is convenient for generating baud rate. In this mode, TMOD high nibble will be 0010B. At 12 MHz oscillator frequency, the timer clocking time is 1μs. Now, the baud rate formula is simplified to

Baud rate = [2SMOD/32] x (oscillator frequency) / [12 x (256 - (TH1)] (6.2)

For example, if TH1 contents are 230D, and SMOD bit in PCON is 0, then the baud rate at 12 MHz is 1201 baud or 1.2K approximately. To get exactly 1200 baud, the oscillator frequency must be 11.059 MHz This shows the degree of dependency of the baud rate on the operating frequency. Thus, to be precise, the actual oscillator frequency must be measured on the oscilloscope.

To receive a byte in mode 1, the RI bit in SCON is tested for 1. Similarly, the REN bit in SCON must be'1'.

The following Programming Example #6.13 will receive a byte through pin RXD.

```
;Programming Example #6.13
; Receiving a serial byte through RXD
MOV SCON, #0101 0000B     ; Serial port mode 1 and REN bit is set
                          ;SMOD is 0 after RESET
MOV TMOD, #0010 0000B     ; Timer 1 in mode 2
MOV TH1, #230D            ; Baud rate 1.2K at 12 MHz
SETB TR1                  ; Start timer 1
CLR R1                    ; Ready to receive
JNB R1, $                 ; Wait till a byte is received in SBUF
MOV A, SBUF               ; Get the received byte in accumulator.89
```

**C. Mode 2**

In mode 2, 11 bits are transmitted, with a low start bit, then 8 data bits, a 9th bit and a stop bit T. This is shown in Fig. 6.13.

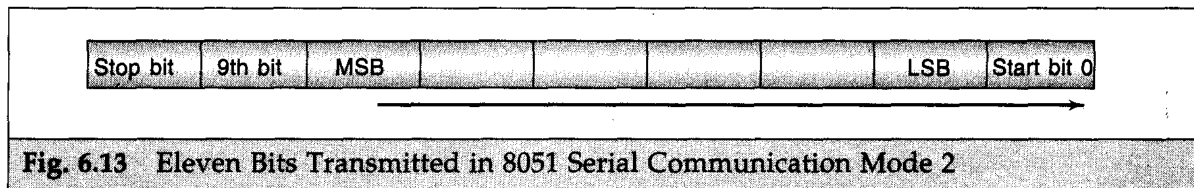



**Fig. 6.13 Eleven Bits Transmitted in 8051 Serial Communication Mode 2**

The 9th bit is programmable. User program can define 9th bit as TB8 in SCON. It may be the parity of data byte. On reception, this 9th data bit goes into RB8 in SCON. In mode 2, the bit SMOD in PCON and the oscillator frequency defines the baud rate and is given by

Baud rate = [2SMOD/64] x (oscillator frequency) (6.3)

Now consider Programming Example #6.14 to initialize the serial port in mode 2. At 12 MHz oscillator frequency, if SMD bit is 1, then the baud rate will be 375,000 or 375K.

```
; Programming Example #6.14
; Initializing the serial port in mode 2
CLR TI
MOV SCON, #1000 0000B ; Serial port mode 2
SETB SMOD; SMOD=1 and baud rate=375K at 12 MHz
MOV SBUF,# 42H
JNB TI,$ ;Wait till transmission is over
CLR TI
```

### D. Mode3

Again 11 bits are transmitted as shown in Fig. 6.13, this is almost same as mode 2, except that the baud rate is defined by the timer 1 overflow rate. The baud rate calculations are exactly same as that of mode 1.

# MODULE – III

# 8051 MICRO CONTROLLER DESIGN

## TIMERS AND COUNTERS

On-chip timing/counting facility has proved the capabilities of the micrcontrollers for implementing the real time applications. These include pulse counting, frequency measurement, pulse width measurement, baud rate generation, etc. Having sufficient number of timer/counters may be a need in a certain design application. As seen in the first chapter, 8051 has two 16-bit timer/counters. Before discussing 8051 timer/counters, it is necessary to see the exact difference between a timer and a counter. A timer counts machine cycles and provides a reference time delay or a clock. A machine cycle of 8051 consists of 12 oscillator periods or the counting rate is 1/12 of the oscillator frequency. At 12 MHz, the clocking period will be equal to 1 μs. Let us now see, the counting function. A counter of 8051 is incremented in response to a transition from '1' to '0' at its corresponding external pin (either T0 or T1). Thus, the counter output will be a count or a number representing the occurrence of such '1' to '0' transitions at the external pin. For counting function, 8051 takes 2 machine cycles or 24 oscillator periods to detect a '1' to '0' transition at Pin T0 or T1. When a timer or counter overflows from FFFFH to 0000H, it sets a flag and generates an interrupt. The 16 bits of timer are referred as higher byte THx and the lower byte TLx. Thus, TH1 is the higher byte of timer 1 and TL1 is the lower byte of timer 1. 'x' can be 0 or1 (or 2 in case of 8032/52).

### Timer/Counter Modes

There are four timer modes in 8051. A timer or counter function and modes are selected by writing appropriate bits in the SFR, called the timer mode register (TMOD), whereas the control of timer/counter operation is done through the SFR, called the timer control register (TCON). These SFRs are shown in Figs. 6.3 and 6.4. Now, the question is how exactly to

| TCON.7 | TCON.6 | TCON.5 | TCON.4 | TCON.3 | TCON.2 | TCON.1 | TCON.0 |
|---|---|---|---|---|---|---|---|
| TF1 | TR1 | TF0 | TR0 | IE1 | IT1 | IE0 | IT0 |

| | | |
|---|---|---|
| TCON.7 | TF1 | Timer 1 overflow flag, set when timer/counter overflows |
| TCON.6 | TR1 | Timer 1 run control bit |
| TCON.5 | TF0 | Timer 0 overflow flag, set when timer/counter 0 overflows |
| TCON.4 | TR0 | Timer 0 run control bit |
| TCON.3 | IE1 | Interrupt 1 |
| TCON.2 | IT1 | Timer interrupt 1 |
| TCON.1 | IE0 | Interrupt 0 flag |
| TCON.0 | IT0 | Timer 0 interrupt, IT0 = 0, low level trigger, IT0 = 1,edge trigger (falling edge) |

**Fig. 6.3** Timer Control Register (TCON) (Bit Addressable)

Configure timer/counters as a timer or counter. As seen from Fig. 6.4, the TMOD bit $C/\overline{T}$, defines this operation.

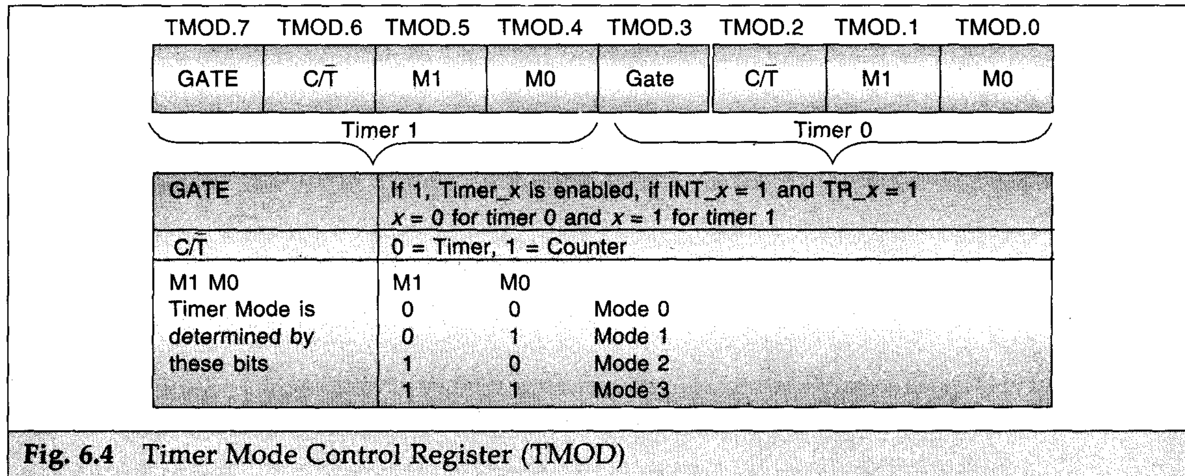



**Fig. 6.4** Timer Mode Control Register (TMOD)

### A). Mode 0

In mode 0, the timer is 13-bit wide. This mode is same for timer 0 and timer 1. When the count overflows, it sets the timer interrupt flag (TF1 for timer 1 and TF0 for timer 0). To start timer 0, TR0 bit in TCON is required to be set. Using the upper byte TH1 (or TH0) and the lower 5 bits of TL1 (or TL0) forms the 13 bits. This is shown in Fig. 6.5 and Fig. 6.6.

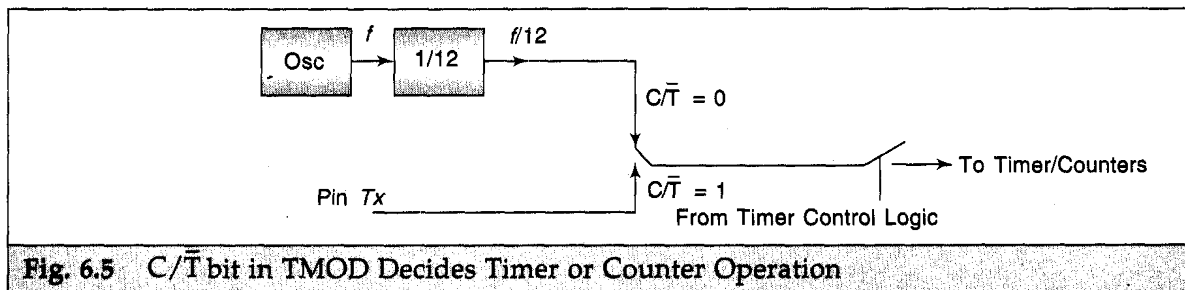



**Fig. 6.5** $C/\overline{T}$ bit in TMOD Decides Timer or Counter Operation

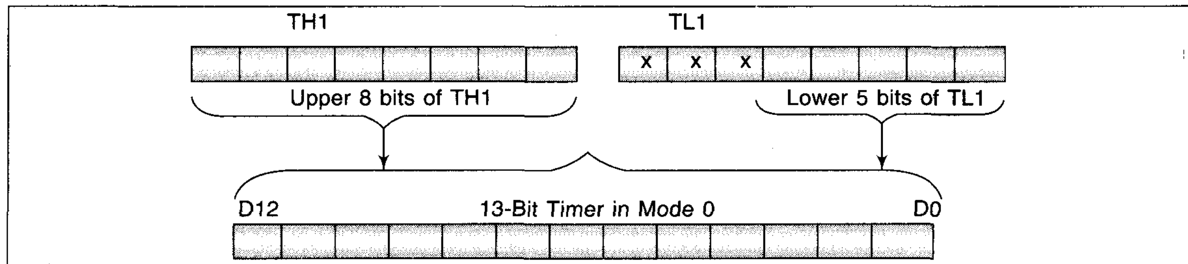



**Fig. 6.6** Mode 0, 13-bit Timer/Counter

```
; Programming Example #6.4
; Initializing timer 1 in mode 0
MOV TMOD, # 1000 0000 B
; Timer 1 in mode 0, Timer 0 in mode 0, both are configured as timers
; Timer 1 is controlled by the external Pin 13 INT1 (Note GATE=1),
whereas the system clock clocks timer 0:
SETB TR1          ; Start timer 1
SETB TR0          ; Start timer 0
CLR TR1           ; Stop timer 1
SJMP $            ; Infinite loop
```

Programming Example #6.4 initializes timer 1 in mode 0. In the above program, timer 1 is configured as a timer in mode 0. Observe that bit TMOD.7 in TMOD is set to 1. This is the GATE bit. If this is set to 1 and TR1 is 1, then the timer 1 is controlled by the external input at Pin 13 ($\overline{\text{INT1}}$). This can be seen from Fig. 6.7. When GATE is 0, then, it is only TR1 which enables the timer.

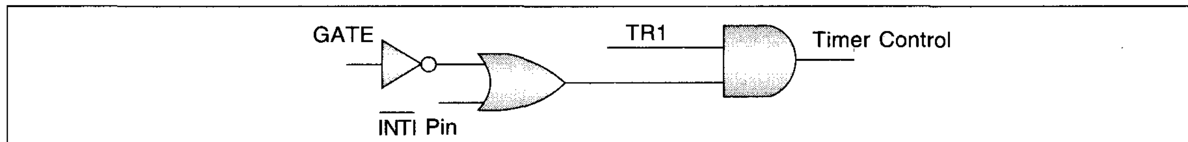



**Fig. 6.7** Timer Control Logic in Mode 0 or 1

**B). Mode 1**

Mode 1 is same as mode 0, except the timers are 16 bits wide. Mode 1 is again the same for timer 0 and timer 1. The maximum count in this mode is FFFFH. To initialize timer 1 in mode 1, see Programming Example #6.5.

```
; Programming Example #6.5
; Initializing timer1 in mode 1
MOV TMOD, # 0001 0000 B          ; Timer 1 in mode 1
SETB TR1                         ; As the GATE bit is zero, TR1 can fully;
                                 ; control the timer operation
                                 ; Hence to start timer 1, TR1 is set to 1.
SJMP $                           ; Infinite loop
```

If initialized, the timer overflow can generate an interrupt. Consider one such program segment (Programming Example #6.6) to initialize the timer 1 interrupt. Note that it is always advisable to initialize the stack pointer before going for a main program, because the default value of SP 07H may not be suitable in general. This is also the address of register R7, and if any register bank switching is done, it can overwrite some useful register contents.

```
; Programming Example #6.6
; Program to initialize timer 1 in mode 1
MOV SP, #54H                     ; Initialize the stack pointer
MOV TMOD, # 0001 0000 B          ; Timer1 in mode 1
SETB ET1                         ; Enable timer 1 interrupt
SETB TR1                         ; Start timer 1
SETB EA                          ; Enable all
SJMP $                           ; Infinite loop (Jump here)
```

Note that simply setting only ET1 bit in IE register will not enable the timer interrupt. In addition, it is necessary to set the EA bit in IE. This program will start timer 1, and when it overflows, timer 1 interrupt is generated, which will cause the program counter to jump to the vector location 001B H.

**C). Mode 2**

This operation is again the same for timer 0 and timer 1. Consider timer 1 in mode 2. Timer register is configured as an 8-bit counter TL1. Overflow from TL1 sets the flag TF1, and it loads TL1 with the contents of TH1. The software can preload TH1. This mode of timer 1 or timer 0 thus supports the automatic reload operation. Mode 2 auto-reload mechanism is shown in Fig. 6.8. Timer control logic is again the same as that of mode 0 or 1.

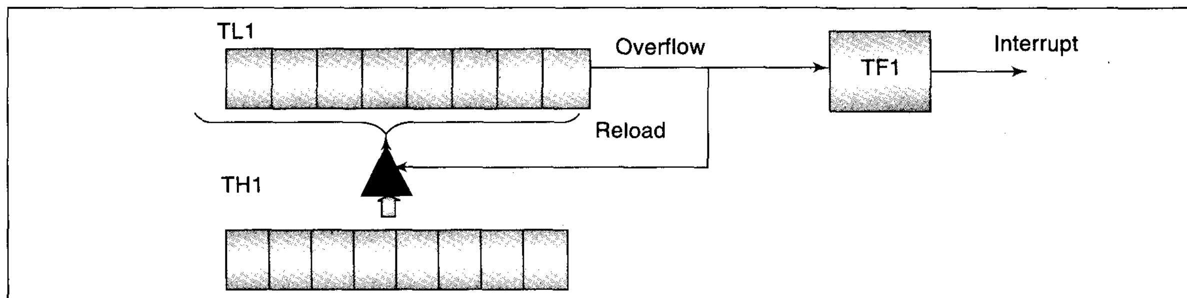



**Fig. 6.8** Mode 2 Auto-reload Mechanism of Timer

Let us now write the initialization program for timer 0 in mode 2 as shown in the Programming Example #6.7. The program must load TMOD and then the auto-reload value must be written in the timer high byte. Further, the starting count will also be the same as that of the reload value in general, but it is not so strict since this is applicable for the very first overflow.

However, it is very essential to load the timer high byte with the auto-reload value, otherwise the timer after each overflow will start from 00H.

```
; Programming Example #6.7
; Initializing timer 0 in mode 2
MOV TMOD, # 0000 0010 B        ; Load TMOD for timer 0 in mode 2
MOV TH0, # 33H                 ; Load TH0 with preset value to be reloaded
MOV TL0, # 33 H                ; Starting count = preset value
SETB TR0                       ; Start timer 0
SJMP $
```

Mode 2 is very commonly used for baud rate generation for serial port operation, or where a constant frequency square wave output is needed. The frequency or baud rate can be controlled using the preloaded value in THx register. The maximum delay generated using mode 2 will be corresponding to the auto-reload value of 00H. Thus, at 12 MHz clock, this would generate the maximum delay of 256 μs. If one can write an instruction to toggle any of the port pins, a square wave output on that pin can be seen on the oscilloscope. Consider this program to generate a 2 kHz (0.5 ms period) square waveform on pin Pl.0 (Pin 1), as shown in Programming Example #6.8. The reload count will be corresponding to 0.25mS. At 12 MHz, this will be (256-250) equal to 06H.

```
;Programming Example #6.8
; Program to generate 2 kHz square waves on pin P1.0 of port 1
            MOV SP, # 54H; Initialize the stack pointer
            MOV TMOD, 0000 0010 B; Timer 0 in mode 2 (Auto-reload mode)
            MOV TH0, # 06H; Preload value for 2 kHz square waves
            MOV TL0, # 06H; Starting value in timer register TL0
            SETB TR0; Start timer 0
      LOOP: JB TF0, COMPLMNT
            SJMP LOOP
  COMPLMNT: CPL P1.0; Toggle bit P1.0
            SJMP LOOP
```

The same program could be written using timer interrupt also. Let us see how to do it! Note that timer 0 interrupt has been enabled at the time of starting the timer. The timer 0 is initialized in mode 2 or auto-reload mode. TH0 and TL0 both are initialized to the count 06H corresponding to the 2 kHz frequency of square waves. The main program is over once timer 0 is started. But notice the instruction SJMP $. This instruction is to jump at the same address and generate an infinite loop. The effect is same as the instruction "LABEL: SJMP LABEL". This program is shown in the Programming Example #6.9.

```
;Programming Example #6.9
; Program to generate 2 kHz square waves on pin P1.0 of port 1 (Use of Interrupt)
           ORG 0000H
           AJMP STRT                   ; Main program is at STRT
           ORG 000B H
           AJMP INT_TF0                ; ISR is at INT_TF0
     STRT: MOV SP, # 54H               ; Initialize the stack pointer
           SETB ET0
           SETB EA
           MOV TMOD, 0000 0010 B       ; Timer 0 in mode 2 (Auto-reload mode)
           MOV TH0, # 06H              ; Preload value for 2 kHz square waves
           MOV TL0, # 06H              ; Starting value in timer register TL0
           SETB TR0                    ; Start timer 0
           SJMP $                      ; Infinite loop here
  INT_TF0: CPL P1.0                    ;
           RETI
           END
```

Further, we have written an interrupt service routine (ISR) in which we just complement the bit Pl.0. The ISR ends with RETI instruction. After the execution of this instruction, the CPU will again be in the infinite loop, from where it was interrupted. Note that timer 0, once started, is not made off or on afterwards, which is not needed also due to auto-reload feature.

**D). Mode 3**

In this mode, timer 1 has a passive role of holding its count. In effect, it looks like as the one who keeps TR1 = 0. Now, timer 0 bytes TH0 and TL0 are used as two separate timers.

Because of this, mode 3 is also called as split timer mode. TH0 is locked into timer operation and simply counts the machine cycles. After overflowing, it sets the flag TF1.

TL0 can be configured and controlled by using C/$\overline{T}$, GATE, TR0, INT0, and TF0. Note that TR1 controls the operation of TH0 timer, now the question remains how to control timer 1? Timer 1 can be used in a different manner, for any application that does not require the interrupt operation; like for generating the baud rate for serial port operation. When timer 0 is in mode 3, one can just control its operation by switching it out or into its mode 3 using TMOD settings. Thus, in mode 3 it resembles like 8051 having 3 timer/counters.

Note that in timer mode 3, timer 1 is a 16-bit timer and TH0, TL0 two 8-bit timers.

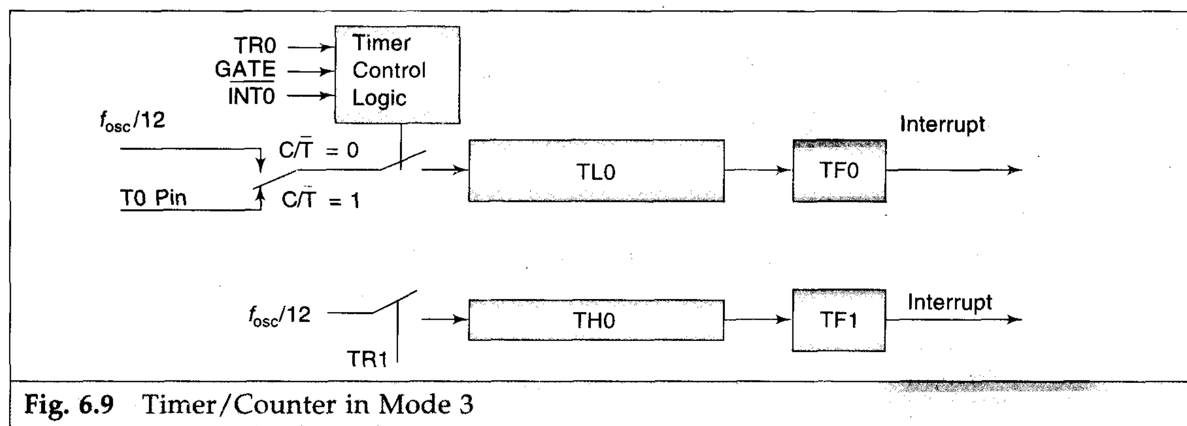



**Fig. 6.9** Timer/Counter in Mode 3

## SERIAL COMMUNICATION

Serial data transmission is very commonly used for digital data communication. Its main advantage is that the number of wires needed is reduced as compared to that in parallel communication. 8051 supports a full duplex serial port. Full duplex means, it can transmit and receive a byte simultaneously. 8051 has TXD and RXD pins for transmission and reception of serial data respectively. The 8051 serial communication is supported by RS232 standard. The term "RS" stands for Recommended Standard. Communication between two microcontrollers and multiprocessor communication is also possible. The start and stop bits are used to synchronize the

serial receivers. The data byte is always transmitted with least-significant-bit first. For error checking purpose, it is possible to include a parity bit as well, just prior to the stop bit. Thus, the bits are transmitted at specific time intervals determined by the baud rate. For error-free serial communication, it is necessary that the baud rate, the number of data bits, the number of stop bits, and the presence or absence of a parity bit along with its status be the same at the transmitter and receiver ends.

The basic mechanism of serial transmission is that a data byte in parallel form is converted into serial data stream. Along with some more bits like start, stop and parity bits, a serial data frame is sent over a line. There are four modes of serial data transmission in 8051. In each of these modes, it is important to decide the baud rate, the way in which serial data frame is sent and any other information, etc.

| SCON.7 | SCON.6 | SCON.5 | SCON.4 | SCON.3 | SCON.2 | SCON.1 | SCON.0 |
|---|---|---|---|---|---|---|---|
| SM0 | SM1 | SM2 | REN | TB8 | RB8 | TI | RI |

| Bit address | SCON bit | Description |
|---|---|---|
| 9FH | SM0 | Serial Communications Mode |
| 9EH | SM1 | |
| 9DH | SM2 | In modes 2 and 3, if set, this will enable multiprocessor communication |
| 9CH | REN | (Receive enable) Enables serial reception |
| 9BH | TB8 | This is the 9th data bit that is transmitted in modes 2 and 3 |
| 9AH | RB8 | 9th data bit that is received in modes 2 and 3. It is not used in mode 0. In mode 1, if SM2 = 0, then RB8 is the stop bit that is received. |
| 99H | TI | Transmit interrupt flag, set by hardware, must be cleared by software |
| 98H | RI | Receive interrupt flag, set by hardware, must be cleared by software |

| SM0 | SM1 | MODE | Description | Baud Rate |
|---|---|---|---|---|
| 0 | 0 | 0 | 8-bit Shift register mode | $f_{osc}/12$ |
| 0 | 1 | 1 | 8-bit UART | variable (set by timer 1) |
| 1 | 0 | 2 | 9-bit UART | $f_{osc}/164$ or $f_{osc}/32$ |
| 1 | 1 | 3 | 9-bit UART | variable (set by timer 1) |

**Fig. 6.10** Serial Control Register (SCON)

What is common in all these modes is the use of the SFR called "SBUF", for transmission as well as reception. The data to be transmitted must be transferred to SBUF. One more SFR that controls the serial communication operation is the serial control register SCON. Details of SCON

are shown in Fig. 6.10. Bits SM0 and SM1 in SCON define serial port mode. Bit SM2 enables the multiprocessor communication in modes 2 and 3. Transmission is initiated by the execution of any instruction that uses SBUF as the destination.

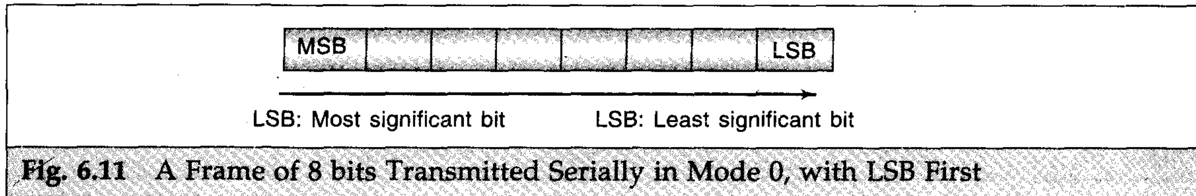



**Fig. 6.11** A Frame of 8 bits Transmitted Serially in Mode 0, with LSB First

### Serial Communication Modes

There are four modes in which 8051 serial port can be configured.

#### E. Mode 0

This is also called as shift register mode. Only RXD is the pin through which data enter or exits. TXD pin outputs the shift clock only. Eight data bits are transmitted or received. The baud rate is fixed and is totally determined by the system clock frequency. If $f_{osc}$ is the clock frequency, then $f_{osc}/12$ will be the baud rate.

To see exactly how the operation of serial data transfer takes place in mode 0, see Programming Example #6.11.

```
; Programming Example #6.11
; Serial transmission mode 0
        ORG 0000H              ; Program starts at 0000H
        MOV SCON, #0000 0000B  ; Mode 0
; Now write the data byte to be transmitted in SBUF
        MOV SBUF, #44H         ; Transmit 0100 0100 binary
; After transmission, TI flag in SCON will be set by hardware, this can be
; tested for assuring the transmission operation
        Here: JNB TI, Here
                               ; Wait till all 8 bits are transmitted
                               ; Remember TI flag must be cleared.
        CLR TI; TI flag is reset
```

#### F. Mode 1

In mode 1, 10 bits are transmitted through TXD pin or received through RXD pin. There is a start bit (0), then 8 data bits (LSB first) and a stop bit (1). This is shown in Fig. 6.12. On

receiving, the stop bit goes into RB8 in SCON. The baud rate is variable and is determined by the timer 1 overflow rate. Therefore, before using this mode, one has to initialize timer 1. A simple program to initialize serial port in mode 1 is given in Programming Example #6.12. The baud rate is calculated using the formula:

Baud rate = 2SMOD/32 x (Timer 1 overflow rate) (6.1)

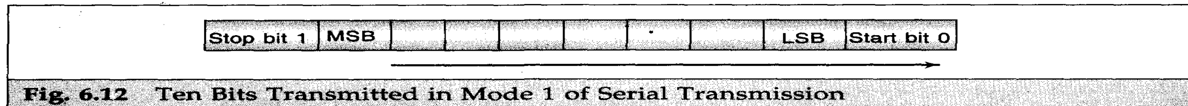



**Fig. 6.12** Ten Bits Transmitted in Mode 1 of Serial Transmission

```
;Programming Example #6.12
; Initializing the serial port in mode 1
ORG 0000H
MOV SP, #51H
; Note that SMOD is '0' after RESET
MOV SCON, #0100 0000B          ; Serial port in mode 1
MOV TMOD, #0010 0000B          ; Timer 1 in auto-reload mode
MOV TH1, #230 D                ; Baud rate =1200 at 12 MHz
SETB TR1                       ; Start timer
MOV SBUF, #56H
JNB TI, $                      ; Wait till the transmission is over
CLR TI                         ; Reset bit TI after transmission
```

If timer 1 is configured in auto-reload mode (or mode 2), with reload value in TH1, after each overflow, contents of TH1 will be loaded into TL1. This is convenient for generating baud rate. In this mode, TMOD high nibble will be 0010B. At 12 MHz oscillator frequency, the timer clocking time is 1μs. Now, the baud rate formula is simplified to

Baud rate = [2SMOD/32] x (oscillator frequency) / [12 x (256 - (TH1)] (6.2)

For example, if TH1 contents are 230D, and SMOD bit in PCON is 0, then the baud rate at 12 MHz is 1201 baud or 1.2K approximately. To get exactly 1200 baud, the oscillator frequency must be 11.059 MHz This shows the degree of dependency of the baud rate on the operating frequency. Thus, to be precise, the actual oscillator frequency must be measured on the oscilloscope.

To receive a byte in mode 1, the RI bit in SCON is tested for 1. Similarly, the REN bit in SCON must be'1'.

The following Programming Example #6.13 will receive a byte through pin RXD.

```
;Programming Example #6.13
; Receiving a serial byte through RXD
MOV SCON, #0101 0000B       ; Serial port mode 1 and REN bit is set
                            ;SMOD is 0 after RESET
MOV TMOD, #0010 0000B       ; Timer 1 in mode 2
MOV TH1, #230D              ; Baud rate 1.2K at 12 MHz
SETB TR1                    ; Start timer 1
CLR R1                      ; Ready to receive
JNB R1, $                   ; Wait till a byte is received in SBUF
MOV A, SBUF                 ; Get the received byte in accumulator.89
```

**G. Mode 2**

In mode 2, 11 bits are transmitted, with a low start bit, then 8 data bits, a 9th bit and a stop bit T. This is shown in Fig. 6.13.

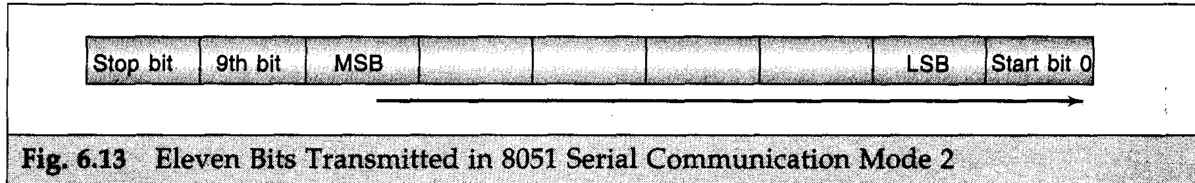



**Fig. 6.13** Eleven Bits Transmitted in 8051 Serial Communication Mode 2

The 9th bit is programmable. User program can define 9th bit as TB8 in SCON. It may be the parity of data byte. On reception, this 9th data bit goes into RB8 in SCON. In mode 2, the bit SMOD in PCON and the oscillator frequency defines the baud rate and is given by

Baud rate = [2SMOD/64] x (oscillator frequency) (6.3)

Now consider Programming Example #6.14 to initialize the serial port in mode 2. At 12 MHz oscillator frequency, if SMD bit is 1, then the baud rate will be 375,000 or 375K.

```
: Programming Example #6.14
; Initializing the serial port in mode 2
CLR TI
MOV SCON, #1000 0000B ; Serial port mode 2
SETB SMOD; SMOD=1 and baud rate=375K at 12 MHz
MOV SBUF,# 42H
JNB TI,$ ;Wait till transmission is over
CLR TI
```

**H. Mode3**

Again 11 bits are transmitted as shown in Fig. 6.13, this is almost same as mode 2, except that the baud rate is defined by the timer 1 overflow rate. The baud rate calculations are exactly same as that of mode 1.

**Keyboard interfacing**

**KEYBOARD INTERFACING**

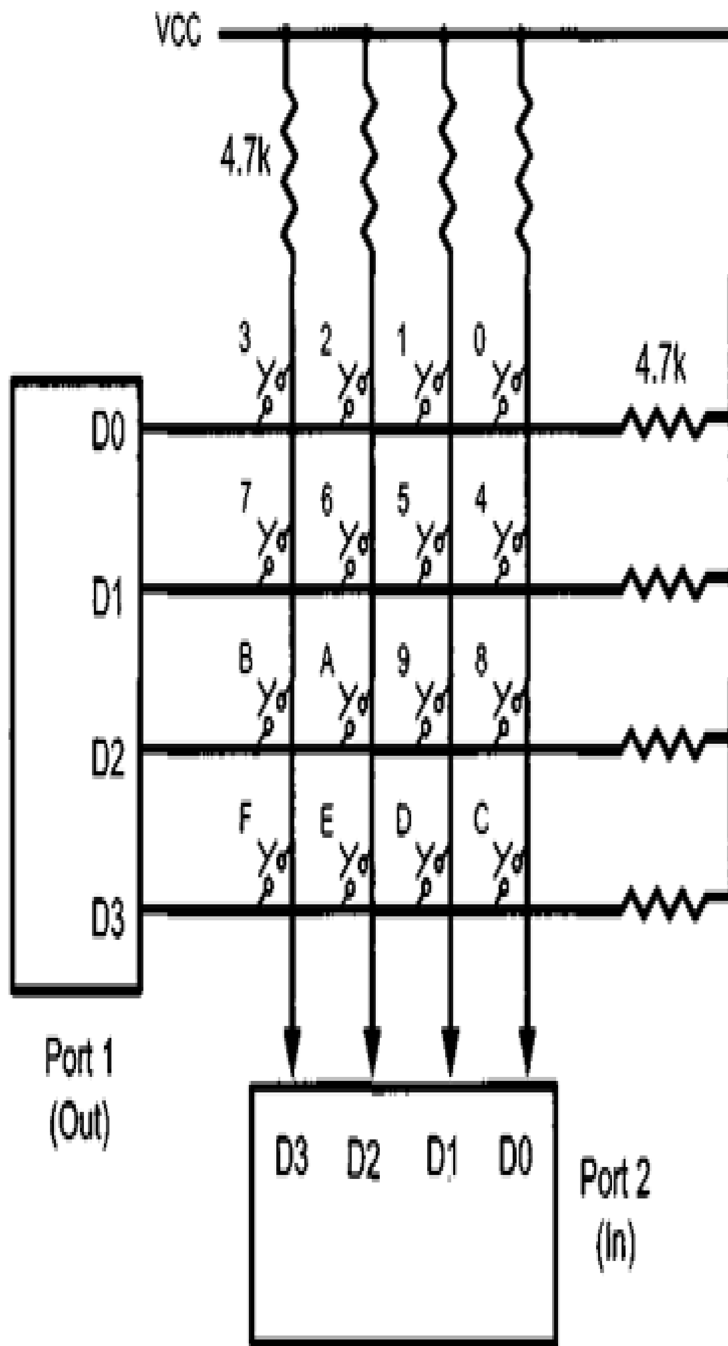
Keyboards and LCDs are the most widely used input/output devices of the 8051, and a basic understanding of them is essential. In this section, we first discuss keyboard fundamentals, along with key press and key detection mechanisms. Then we show how a keyboard is interfaced to an 8051.

**Interfacing the keyboard to the 8051**

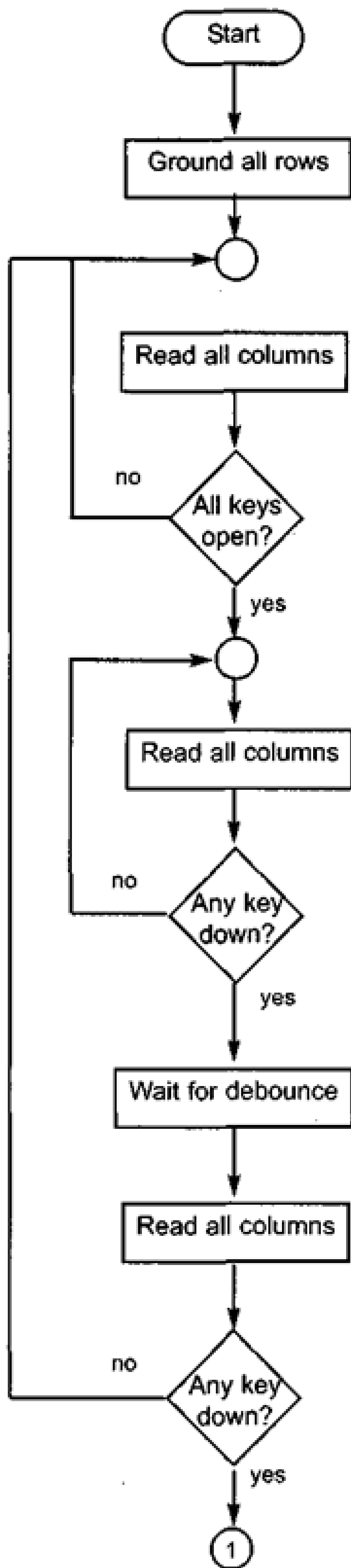
At the lowest level, keyboards are organized in a matrix of rows and columns. The CPU accesses both rows and columns through ports; therefore, with two 8-bit ports, an 8 x 8 matrix of keys can be connected to a microprocessor. When a key is pressed, a row and a column make a contact; otherwise, there is no connection between rows and columns. In IBM PC keyboards, a single microcontroller (consisting of a microprocessor, RAM and EPROM, and several ports all on a single chip) takes care of hardware and software interfacing of the keyboard. In such systems, it is the function of programs stored in the EPROM of the microcontroller to scan the keys continuously, identify which one has been activated, and present it to the motherboard. In this section we look at the mechanism by which the 8051 scans and identifies the key.

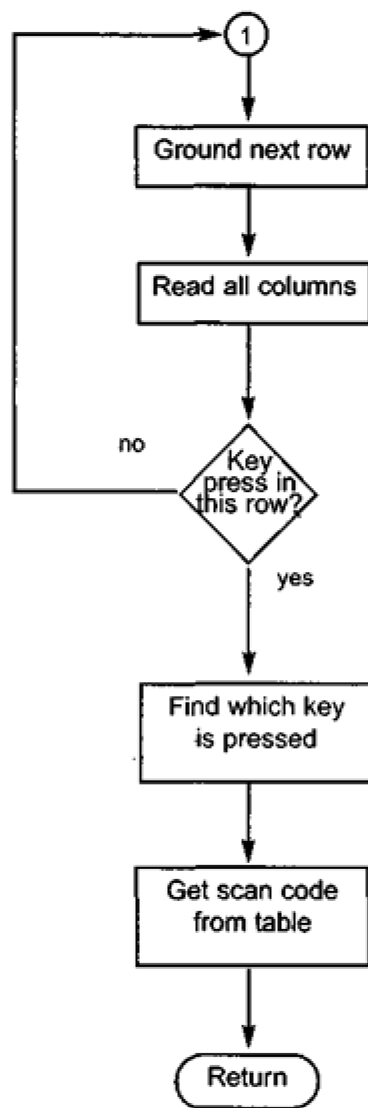
**Scanning and identifying the key**

Figure 12-6 shows a 4 x 4 matrix connected to two ports. The rows are connected to an output port and the columns are connected to an input port. If no key has been pressed, reading the input port will yield 1 s for all columns since they are all connected to high ($V_{cc}$). If all the rows are grounded and a key is pressed, one of the columns will have 0 since the key pressed provides the path to ground. It is the function of the microcontroller to scan the keyboard continuously to detect and identify the key pressed. How it is done is explained next.



VCC
4.7k
3
2
1
0
4.7k
D0
7
6
5
4
D1
B
A
9
8
D2
F
E
D
C
D3
Port 1
(Out)
D3
D2
D1
D0
Port 2
(In)


Start
Ground all rows
Read all columns
no
All keys open?
yes
Read all columns
no
Any key down?
yes
Wait for debounce
Read all columns
no
Any key down?
yes
1

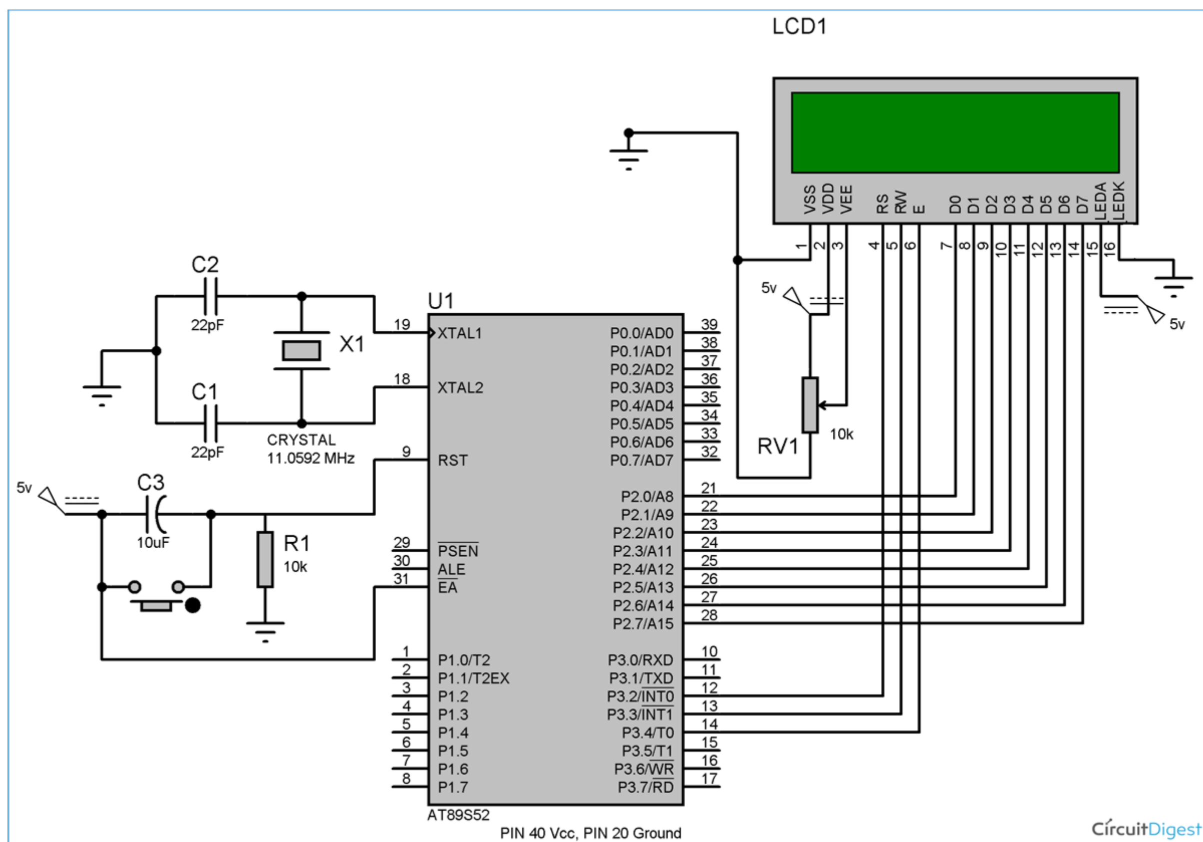

1
Ground next row
Read all columns
no
Key press in this row?
yes
Find which key is pressed
Get scan code from table
Return

```
;Keyboard subroutine. This program sends the ASCII code
;for pressed key to P0.1
;P1.0-P1.3 connected to rows P2.0-P2.3 connected to columns
        MOV   P2,#0FFH            ;make P2 an input port
K1:     MOV   P1,#0               ;ground all rows at once
        MOV   A,P2        ;read all col. ensure all keys open
        ANL   A,#00001111B        ;masked unused bits
        CJNE  A,#00001111B,K1     ;check til all keys released
  :     ACALL DELAY               ;call 20 ms delay
        MOV   A,P2                ;see if any key is pressed
        ANL   A,#00001111B        ;mask unused bits
        CJNE  A,#00001111B,OVER   ;key pressed, await closure
        SJMP  K2                  ;check if key pressed
OVER:   ACALL DELAY               ;wait 20 ms debounce time
        MOV   A,P2                ;check key closure
        ANL   A,#00001111B        ;mask unused bits
        CJNE  A,#00001111B,OVER1  ;key pressed, find row
        SJMP  K2                  ;if none, keep polling
OVER1:  MOV   P1,#11111110B       ;ground row 0
        MOV   A,P2                ;read all columns
        ANL   A,#00001111B        ;mask unused bits
        CJNE  A,#00001111B,ROW_0  ;key row 0, find the col.
        MOV   P1,#11111101B       ;ground row 1
        MOV   A,P2                ;read all columns
        ANL   A,#00001111B        ;mask unused bits
        CJNE  A,#00001111B,ROW_1  ;key row 1, find the col.
        MOV   P1,#11111011B       ;ground row 2
        MOV   A,P2                ;read all columns
        ANL   A,#00001111B        ;mask unused bits
        CJNE  A,#00001111B,ROW_2  ;key row 2, find the col.
        MOV   P1,#11110111B       ;ground row 3
        MOV   A,P2                ;read all columns
        ANL   A,#00001111B        ;mask unused bits
        CJNE  A,#00001111B,ROW_3  ;key row 3, find the col.
        LJMP  K2                  ;if none, false input, repeat

ROW_0:  MOV   DPTR,#KCODE0        ;set DPTR=start of row 0
        SJMP  FIND                ;find col. key belongs to
ROW_1:  MOV   DPTR,#KCODE1        ;set DPTR=start of row 1
        SJMP  FIND                ;find col. key belongs to
ROW_2:  MOV   DPTR,#KCODE2        ;set DPTR=start of row 2
        SJMP  FIND                ;find col. key belongs to
ROW_3:  MOV   DPTR,#KCODE3        ;set DPTR=start of row 3
FIND:   RRC   A                   ;see if any CY bit is low
        JNC   MATCH               ;if zero, get the ASCII code
        INC   DPTR                ;point to next col. address
```

```
        SJMP   FIND                 ;keep searching
MATCH:  CLR    A                    ;set A=0 (match is found)
        MOVC   A,@A+DPTR            ;get ASCII code from table
        MOV    P0,A                 ;display pressed key
        LJMP   K1
;ASCII LOOK-UP TABLE FOR EACH ROW
        ORG    300H
KCODE0: DB     '0','1','2','3'              ;ROW 0
KCODE1: DB     '4','5','6','7'              ;ROW 1
KCODE2: DB     '8','9','A','B'              ;ROW 2
KCODE3: DB     'C','D','E','F'              ;ROW 3
       END
```

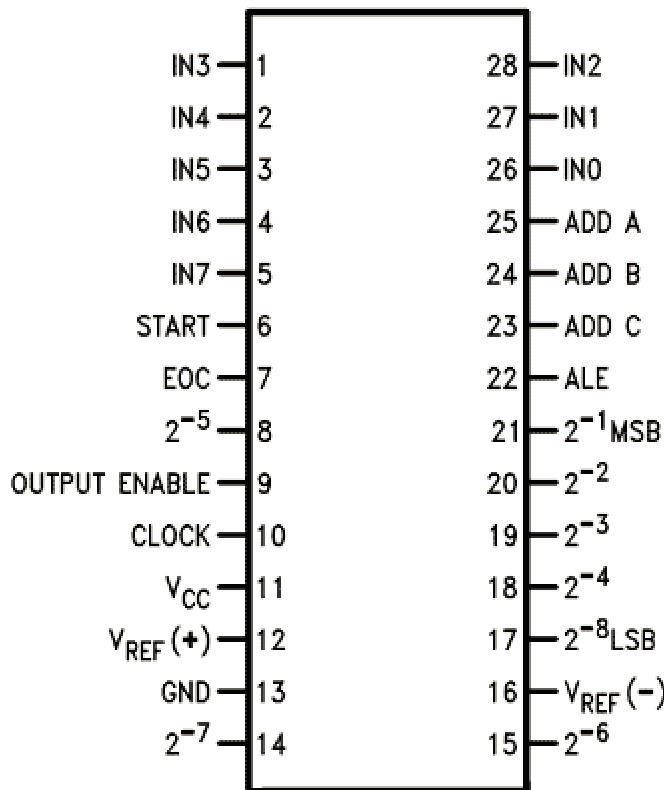
LCD INTERFACING USING 8051

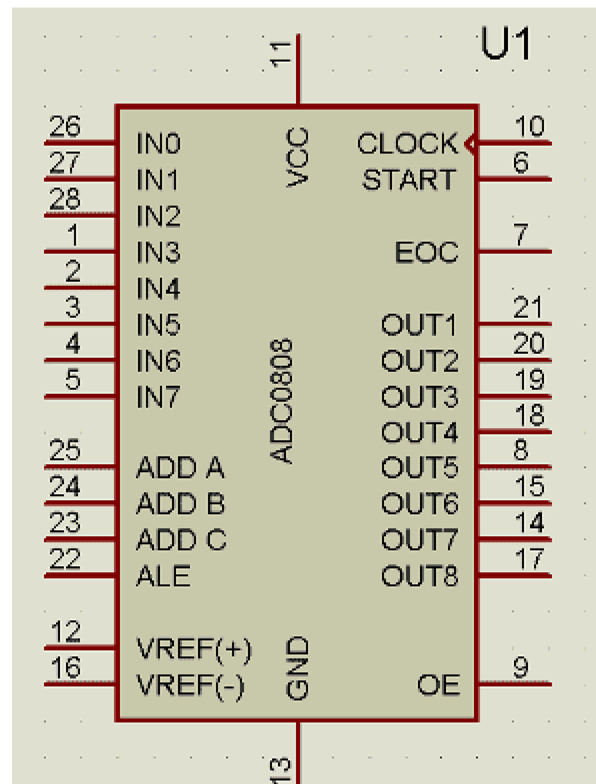



**ADC is the Analog to Digital converter**, which converts analog data into digital format; usually it is used to convert **analog voltage** into digital format. Analog signal has infinite no of values like a sine wave or our speech, ADC converts them into particular levels or states, which can be measured in numbers as a physical quantity. Instead of continuous conversion, ADC converts

data periodically, which is usually known as sampling rate. **Telephone modem** is one of the examples of ADC, which is used for internet, it converts analog data into digital data, so that computer can understand, because computer can only understand Digital data. The major advantage, of using ADC is that, we noise can be efficiently eliminated from the original signal and digital signal can travel more efficiently than analog one. That's the reason that digital audio is very clear, while listening.

ADC0808/0809 is a monolithic CMOS device and microprocessor compatible control logic and has 28 pin which gives 8-bit value in output and 8- channel ADC input pins (IN0-IN7). Its resolution is 8 so it can encode the analog data into one of the **256 levels** ($2^8$). This device has three channel address line namely: ADDA, ADDB and ADDC for selecting channel. Below is the Pin Diagram for ADC0808:




ADC0808/0809 **requires a clock pulse** for conversion. We can provide it by using oscillator or by using microcontroller. In this project we have applied frequency by using microcontroller.

We can select the any input channel by using the Address lines, like we can select the input line IN0 by keeping all three address lines (ADDA, ADDB and ADDC) Low. If we want to select input channel IN2 then we need to keep ADDA, ADDB low and ADDC high. For selecting all the other input channels, have a look on the given table:

| ADC Channel Name | ADDC PIN | ADDB PIN | ADDA PIN |
|---|---|---|---|
| IN0 | LOW | LOW | LOW |

| IN1 | LOW | LOW | HIGH |
|---|---|---|---|
| IN2 | LOW | HIGH | LOW |
| IN3 | LOW | HIGH | HIGH |
| IN4 | HIGH | LOW | LOW |
| IN5 | HIGH | LOW | HIGH |
| IN6 | HIGH | HIGH | LOW |
| IN7 | HIGH | HIGH | HIGH |

```
# include<reg51.h>

#include<stdio.h>

sbit ale=P3^3;

sbit oe=P3^6;

sbit sc=P3^4;

sbit eoc=P3^5;

sbit clk=P3^7;

sbit ADDA=P3^0;  //Address pins for selecting input channels.

sbit ADDB=P3^1;

sbit ADDC=P3^2;

#define lcdport P2  //lcd

sbit rs=P2^0;

sbit rw=P2^2;

sbit en=P2^1;

#define input_port P1  //ADC

int result[3],number;
```


```
# include<reg51.h>
#include<stdio.h>
sbit ale=P3^3;
sbit oe=P3^6;
sbit sc=P3^4;
sbit eoc=P3^5;
sbit clk=P3^7;
sbit ADDA=P3^0;  //Address pins for selecting input channels.
```

```
sbit ADDB=P3^1;
sbit ADDC=P3^2;
#define lcdport P2  //lcd
sbit rs=P2^0;
sbit rw=P2^2;
sbit en=P2^1;
#define input_port P1  //ADC
int result[3],number;
void timer0() interrupt 1  // Function to generate clock of frequency 500KHZ using Timer 0 interrupt.
{
clk=~clk;
}
void delay(unsigned int count)
{
int i,j;
for(i=0;i<count;i++)
  for(j=0;j<100;j++);
}
void daten()
{
   rs=1;
   rw=0;
   en=1;
   delay(1);
   en=0;
}
void lcd_data(unsigned char ch)
{
   lcdport=ch & 0xF0;
   daten();
   lcdport=ch<<4 & 0xF0;
   daten();
}
void cmden(void)
{
   rs=0;
   en=1;
   delay(1);
```

```
    en=0;
}
void lcdcmd(unsigned char ch)
{
    lcdport=ch & 0xf0;
    cmden();
    lcdport=ch<<4 & 0xF0;
    cmden();
}
lcdprint(unsigned char *str)  //Function to send string data to LCD.
{
    while(*str)
    {
        lcd_data(*str);
        str++;
    }
}
void lcd_ini()  //Function to inisialize the LCD
{
    lcdcmd(0x02);
    lcdcmd(0x28);
    lcdcmd(0x0e);
    lcdcmd(0x01);
}
void show()
{
   sprintf(result,"%d",number);
   lcdprint(result);
   lcdprint("  ");
}
void read_adc()
{
   number=0;
   ale=1;
   sc=1;
   delay(1);
   ale=0;
   sc=0;
```

```
  while(eoc==1);
  while(eoc==0);
  oe=1;
  number=input_port;
  delay(1);
  oe=0;
}
void adc(int i)  //Function to drive ADC
{
switch(i)
 {
  case 0:
   ADDC=0;  // Selecting input channel IN0 using address lines
   ADDB=0;
   ADDA=0;
   lcdcmd(0xc0);
   read_adc();
   show();
   break;
  case 1:
   ADDC=0;  // Selecting input channel IN1 using address lines
   ADDB=0;
   ADDA=1;
   lcdcmd(0xc6);
   read_adc();
   show();
  break;
  case 2:
   ADDC=0;  // Selecting input channel IN2 using address lines
   ADDB=1;
   ADDA=0;
   lcdcmd(0xcc);
   read_adc();
   show();
   break;
 }
}
```

```
void main()
{
 int i=0;
 eoc=1;
 ale=0;
 oe=0;
 sc=0;
 TMOD=0x02;
 TH0=0xFD;
lcd_ini();
lcdprint(" ADC 0808/0809  ");
lcdcmd(192);
lcdprint("  Interfacing   ");
delay(500);
lcdcmd(1);
lcdprint("Circuit Digest ");
lcdcmd(192);
lcdprint("System Ready...   ");
delay(500);
lcdcmd(1);
lcdprint("Ch1   Ch2   Ch3 ");
 IE=0x82;
 TR0=1;
while(1)
{
   for(i=0;i<3;i++)
   {
     adc(i);
     number=0;
   }
}
}
```

## Digital-to-analog (DAC) converter

The digital-to-analog converter (DAC) is a device widely used to convert digital pulses to analog signals. In this section we discuss the basics of interfacing a DAC to the 8051.

Recall from your digital electronics book the two methods of creating a DAC: binary weighted and R/2R ladder. The vast majority of integrated circuit DACs, including the MC1408 (DAC0808) used in this section, use the R/2R method since it can achieve a much higher degree of precision. The first criterion for judging a DAC is its resolution, which is a function of the number of binary inputs. The common ones are 8, 10, and 12 bits. The number of data bit inputs decides the resolution of the DAC since the number of analog output levels is equal to 2″, where *n* is the number of data bit inputs. Therefore, an 8-input DAC.

such as the DAC0808 provides 256 discrete voltage (or current) levels of output. Similarly, the 12-bit DAC provides 4096 discrete voltage levels. There are also 16-bit DACs, but they are more expensive.

## MC1408 DAC (or DAC0808)

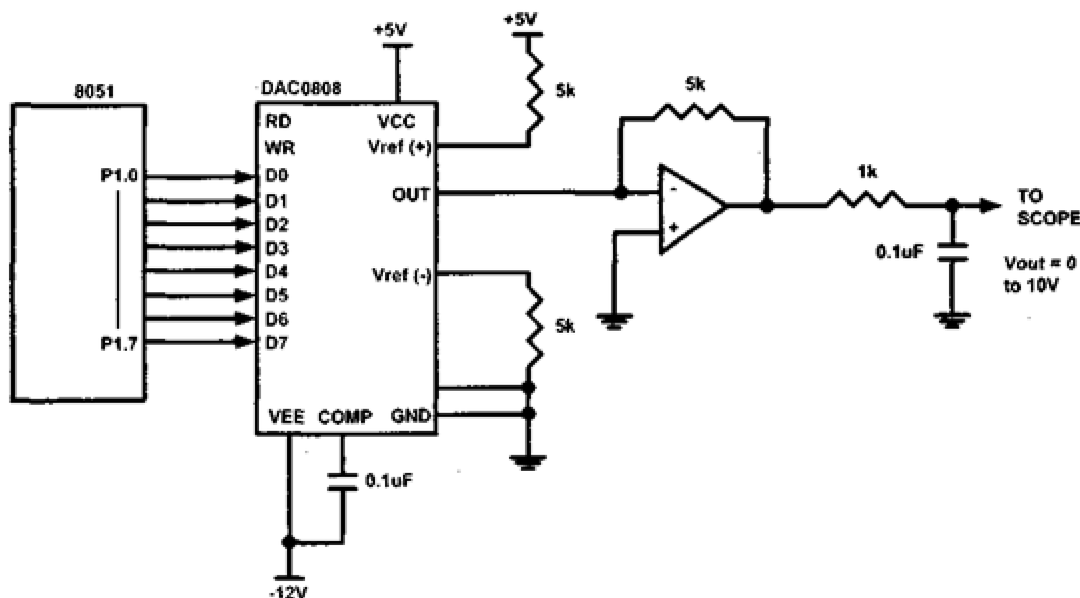
In the MC1408 (DAC0808), the digital inputs are converted to current ($I_{out}$), and by connecting a resistor to the $I_{out}$pin, we convert the result to voltage.

The total current provided by the $I_{out}$ pin is a function of the binary numbers at the DO – D7 inputs of the DAC0808 and the reference current ($I_{re}$f), and is as follows:

$$I_{out} = I_{ref}\left(\frac{D7}{2} + \frac{D6}{4} + \frac{D5}{8} + \frac{D4}{16} + \frac{D3}{32} + \frac{D2}{64} + \frac{D1}{128} + \frac{D0}{256}\right)$$

where DO is the LSB, D7 is the MSB for the inputs, and $I_{re}$f is the input current that must be applied to pin 14. The $I_{re}$f current is generally set to 2.0 mA. Figure 13-18 shows the generation of current reference (setting $I_{re}$f = 2 mA) by using the

standard 5-V power supply and IK and 1.5K-ohm standard resistors. Some DACs also use the zener diode (LM336), which overcomes any fluctuation associated



To find the value sent to the DAC for various angles, we simply multiply the $V_{out}$ voltage by 25.60 because there are 256 steps and full-scale $V_{out}$ is 10 volts. Therefore, 256 steps /10 V = 25.6 steps per volt. To further clarify this, look at the following code. This program sends the values to the DAC continuously (in an infinite loop) to produce a crude sine wave. See Figure 13-19.

```
AGAIN:      MOV DPTR,#TABLE
            MOV R2,#COUNT
BACK:       CLR A
            MOVC A,@A+DPTR
            MOV P1,A
            INC DPTR
            DJNZ R2,BACK
            SJMP AGAIN
            ORG 300
TABLE:      DB 128,192,238,255,238,192 ;see Table 13-7
            DB 128,64,17,0,17,64,128
```

# MODULE –IV

# INTRODUCTION TO DSP AND FFT

# REVIEW OF DISCRETE TIME SIGNALS AND SYSTEMS

**Signals-Definition**

Anything that carries information can be called as signal. It can also be defined as a physical quantity that varies with time, temperature, pressure or with any independent variables such as speech signal or video signal.

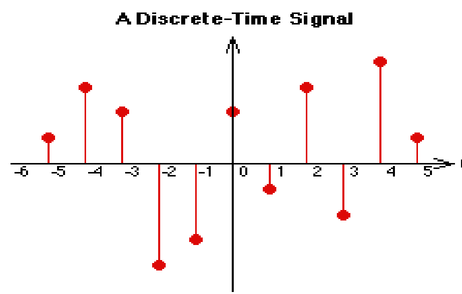
The process of operation in which the characteristics of a signal (Amplitude, shape, phase, frequency, etc.) undergoes a change is known as signal processing.

Note − Any unwanted signal interfering with the main signal is termed as noise. So, noise is also a signal but unwanted.

**Discrete Time signals**

The signals, which are defined at discrete times are known as discrete signals. Therefore, every independent variable has distinct value. Thus, they are represented as sequence of numbers.

Although speech and video signals have the privilege to be represented in both continuous and discrete time format; under certain circumstances, they are identical. Amplitudes also show discrete characteristics. Perfect example of this is a digital signal; whose amplitude and time both are discrete.




The figure above depicts a discrete signal's discrete amplitude characteristic over a period of time. Mathematically, these types of signals can be formularized as;

$x=\{x[n]\}, -\infty<n<\infty$

Where, n is an integer.

It is a sequence of numbers x, where nth number in the sequence is represented as x[n].

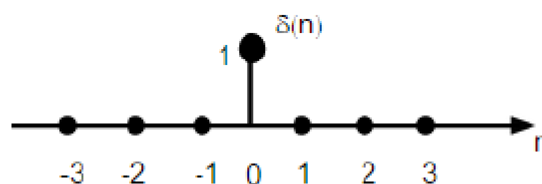
**Basic DT Signals**

Let us see how the basic signals can be represented in Discrete Time Domain.

**Unit Impulse Sequence**

It is denoted as δ(n) in discrete time domain and can be defined as;

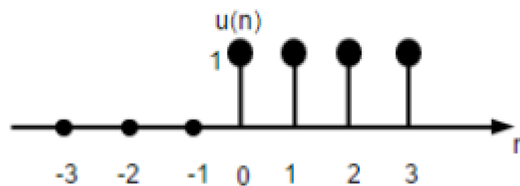
$$\delta(n) = \begin{cases} 1, & for \quad n=0 \\ 0, & Otherwise \end{cases}$$



**Unit Step Signal**

Discrete time unit step signal is defined as;

$$U(n) = \begin{cases} 1, & for \quad n \geq 0 \\ 0, & for \quad n < 0 \end{cases}$$

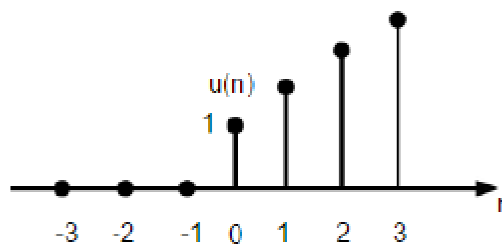



The figure above shows the graphical representation of a discrete step function.

**Unit Ramp Function**

A discrete unit ramp function can be defined as −

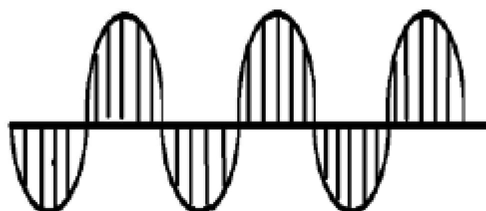
$$r(n) = \begin{cases} n, & for \quad n \geq 0 \\ 0, & for \quad n < 0 \end{cases}$$




The figure given above shows the graphical representation of a discrete ramp signal.

**Sinusoidal Signal**

All continuous-time signals are periodic. The discrete-time sinusoidal sequences may or may not be periodic. They depend on the value of ω. For a discrete time signal to be periodic, the angular frequency ω must be a rational multiple of 2π.



Discrete siinusoidal signal

A discrete sinusoidal signal is shown in the figure above.

Discrete form of a sinusoidal signal can be represented in the format −

$$x(n)=A\sin(\omega n+\phi)$$

Here A,ω and φ have their usual meaning and n is the integer. Time period of the discrete sinusoidal signal is given by −

$$N = \frac{2\pi m}{\omega}$$

Where, N and m are integers.

**Classification of DT Signals**

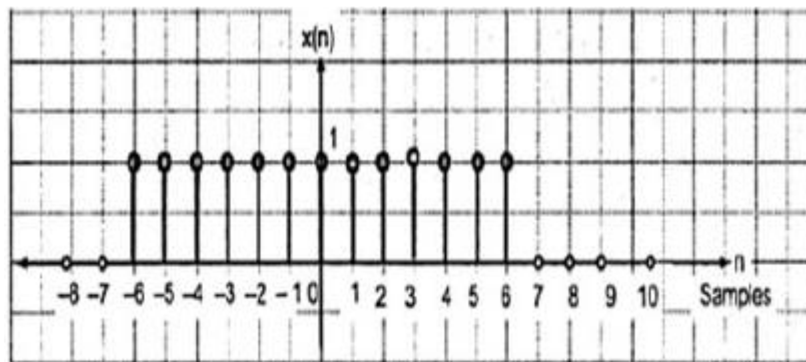
Discrete time signals can be classified according to the conditions or operations on the signals.

**Even and Odd Signals**

**Even Signal**

A signal is said to be even or symmetric if it satisfies the following condition;

$$x(-n) = x(n)$$

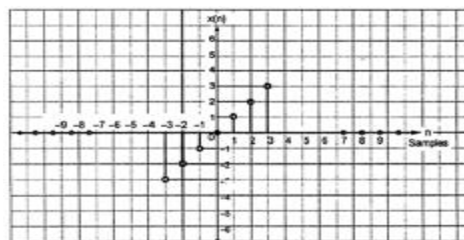



Here, we can see that x(-1) = x(1), x(-2) = x(2) and x(-n) = x(n). Thus, it is an even signal.

**Odd Signal**

A signal is said to be odd if it satisfies the following condition;

$$x(-n) = -x(n)$$



From the figure, we can see that x(1) = -x(-1), x(2) = -x(2) and x(n) = -x(-n). Hence, it is an odd as well as anti-symmetric signal.

**Periodic and Non-Periodic Signals**

A discrete time signal is periodic if and only if, it satisfies the following condition −

$$x(n+N)=x(n)$$

Here, x(n) signal repeats itself after N period. This can be best understood by considering a cosine signal −

$$x(n)=A\cos(2\pi f_0 n+\theta)$$

$$x(n+N)=A\cos(2\pi f_0(n+N)+\theta)=A\cos(2\pi f_0 n+2\pi f0N+\theta)$$

For the signal to become periodic, following condition should be satisfied;

$$x(n+N)=x(n)$$

$$\Rightarrow A\cos(2\pi f_0 n+2\pi f_0 N+\theta)=A\cos(2\pi f_0 n+\theta)$$

i.e. $2\pi f_0 N$ is an integral multiple of $2\pi$

$$2\pi f_0 N=2\pi K$$

$$\Rightarrow \quad N=K/f_0$$

Frequencies of discrete sinusoidal signals are separated by integral multiple of $2\pi$.

**Energy and Power Signals**

**Energy Signal**

Energy of a discrete time signal is denoted as E. Mathematically, it can be written as;

$$E=\sum_{n=-\infty}^{+\infty}|x(n)|^2$$

If each individual values of x(n) are squared and added, we get the energy signal. Here x(n) is the energy signal and its energy is finite over time i.e $0<E<\infty$

**Power Signal**

Average power of a discrete signal is represented as P. Mathematically, this can be written as;

$$P=\lim_{N\to\infty}\frac{1}{2N+1}\sum_{n=-N}^{+N}|x(n)|^2$$

Here, power is finite i.e. $0<P<\infty$. However, there are some signals, which belong to neither energy nor power type signal.

## Operations on Signals

The basic signal operations which manipulate the signal characteristics by acting on the independent variable(s) which are used to represent them. This means that instead of performing operations like addition, subtraction, and multiplication between signals, we will perform them on the independent variable. In our case, this variable is time (t).

### 1. Time Shifting

Suppose that we have a signal x(n) and we define a new signal by adding/subtracting a finite time value to/from it. We now have a new signal, y(n). The mathematical expression for this would be $x(n \pm n0)$.

Graphically, this kind of signal operation results in a positive or negative "shift" of the signal along its time axis. However, note that while doing so, none of its characteristics are altered. This means that the time-shifting operation results in the change of just the positioning of the signal without affecting its amplitude or span.

Let's consider the examples of the signals in the following figures in order to gain better insight into the above information.

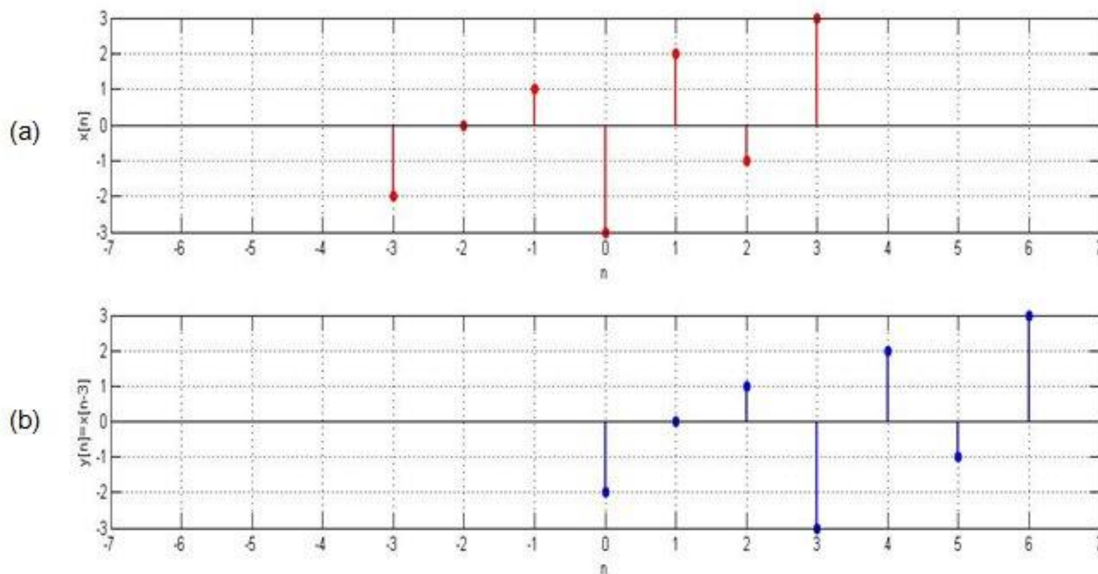



Figure 1. Original signal and its time-delayed version

Here the original signal, x[n], spans from n = -3 to n = 3 and has the values -2, 0, 1, -3, 2, -1, and 3, as shown in Figure 1(a).

### Time-Delayed Signals

Suppose that we want to move this signal right by three units (i.e., we want a new signal whose amplitudes are the same but are shifted right three times).

This means that we desire our output signal y[n] to span from n = 0to n = 6. Such a signal is shown as Figure 1(b) and can be mathematically written as y[n] = x[n-3].

This kind of signal is referred to as time-delayed because we have made the signal arrive three units late.

**Time-Advanced Signals**

On the other hand, let's say that we want the same signal to arrive early. Consider a case where we want our output signal to be advanced by, say, two units. This objective can be accomplished by shifting the signal to the left by two time units, i.e., y[n] = x[n+2].

The corresponding input and output signals are shown in Figure 2(a) and 2(b), respectively. Our output signal has the same values as the original signal but spans from n = -5 to n = 1 instead of n = -3 to n = 3. The signal shown in Figure 2(b) is aptly referred to as a time-advanced signal.

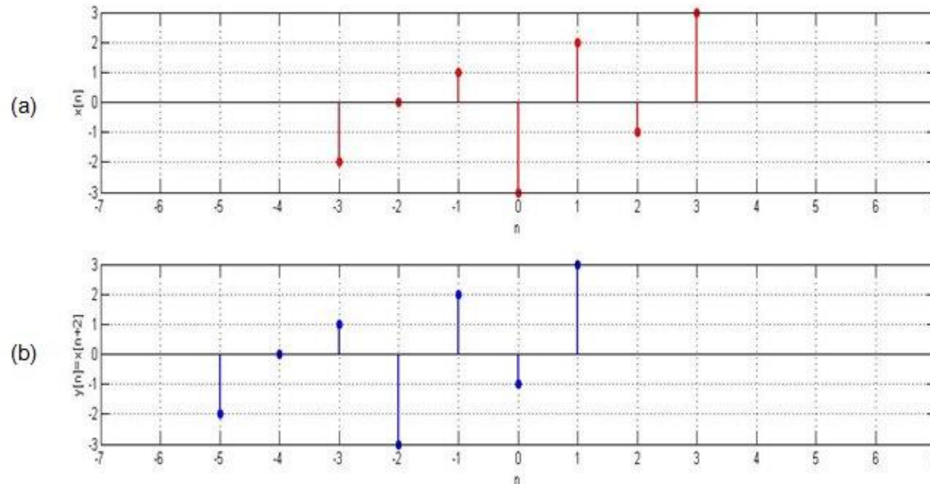



Figure 2. Original signal and its time-advanced version

For both of the above examples, note that the time-shifting operation performed over the signals affects not the amplitudes themselves but rather the amplitudes with respect to the time axis. We have used discrete-time signals in these examples, but the same applies to continuous-time signals.

**Practical Applications**

Time-shifting is an important operation that is used in many signal-processing applications. For example, a time-delayed version of the signal is used when performing autocorrelation. (You can learn more about autocorrelation in my previous article, Understanding Correlation.)

Another field that involves the concept of time delay is artificial intelligence, such as in systems that use Time Delay Neural Networks.

**2. Time Scaling**

Now that we understand more about performing addition and subtraction on the independent variable representing the signal, we'll move on to multiplication.

For this, let's consider our input signal to be a continuous-time signal x(t) as shown by the red curve in Figure 3.

Now suppose that we multiply the independent variable (t) by a number greater than one. That is, let's make t in the signal into, say, 2t. The resultant signal will be the one shown by the blue curve in Figure 3.

From the figure, it's clear that the time-scaled signal is contracted with respect to the original one. For example, we can see that the value of the original signal present at t = -3 is present at t = -1.5 and those at t= -2 and at t = -1 are found at t = -1 and at t = -0.5 (shown by green dotted-line curved arrows in the figure).

This means that, if we multiply the time variable by a factor of 2, then we will get our output signal contracted by a factor of 2 along the time axis. Thus, it can be concluded that the multiplication of the signal by a factor of n leads to the compression of the signal by an equivalent factor.

Now, does this mean that dividing the variable t by a number greater than 1 will cause the signal to become expanded? That is, if we divide the variable t by a factor of n, will we get a signal which is stretched by an equivalent factor?

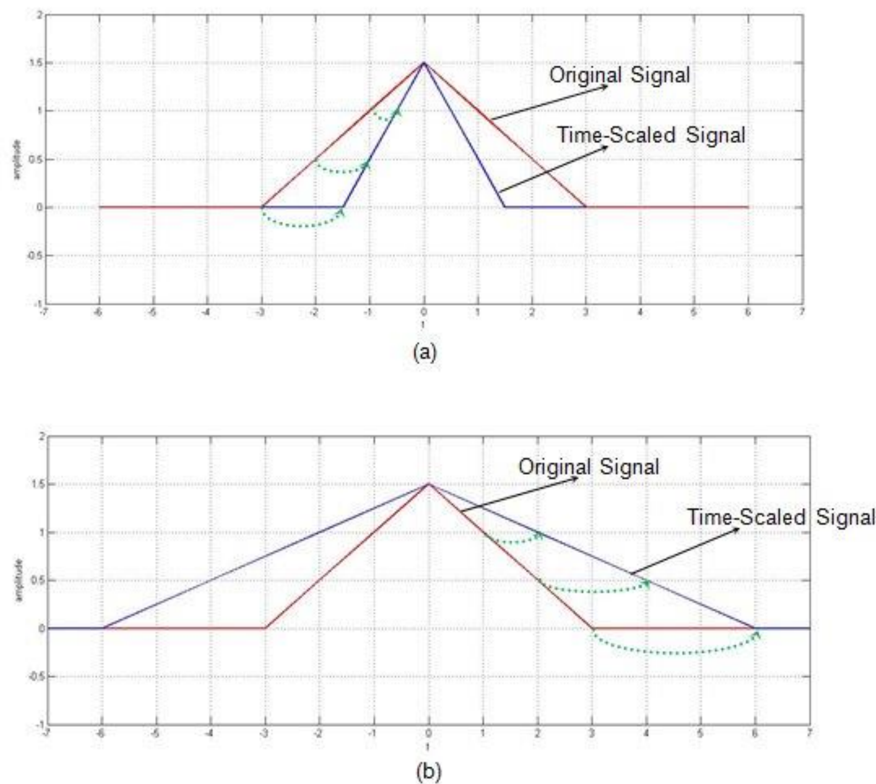



Figure 3. Original signal with its time-scaled versions

Let's check it out.

For this, let's consider our signal to be the same as the one in Figure 3 (the red curve in the figure). Now let's multiply its time-variable t by ½ instead of 2. The resultant signal is shown by the blue curve in Figure 3(b). You can see that, in this time-scaled signal indicated by the green dotted-line arrows in Figure 3(b), we have the values of the original signal present at the time instants t = 1, 2, and 3 to be found at t = 2, 4, and 6.

This means that our time-scaled signal is a stretched-by-a-factor-of-n version of the original signal. So the answer to the question posed above is "yes."

Although we have analyzed the time-scaling operation with respect to a continuous-time signal, this information applies to discrete-time signals as well. However, in the case of discrete-time signals, time-scaling operations are manifested in the form of decimation and interpolation.

**Practical Applications**

Basically, when we perform time scaling, we change the rate at which the signal is sampled. Changing the sampling rate of a signal is employed in the field of speech processing. A particular example of this would be a time-scaling-algorithm-based system developed to read text to the visually impaired.

Next, the technique of interpolation is used in Geodesic applications (PDF). This is because, in most of these applications, one will be required to find out or predict an unknown parameter from a limited amount of available data.

**3. Time Reversal**

Until now, we have assumed our independent variable representing the signal to be positive. Why should this be the case? Can't it be negative?

It can be negative. In fact, one can make it negative just by multiplying it by -1. This causes the original signal to flip along its y-axis. That is, it results in the reflection of the signal along its vertical axis of reference. As a result, the operation is aptly known as the time reversal or time reflection of the signal.

For example, let's consider our input signal to be x[n], shown in Figure 4(a). The effect of substituting –n in the place of n results in the signal y[n] as shown in Figure 4(b).

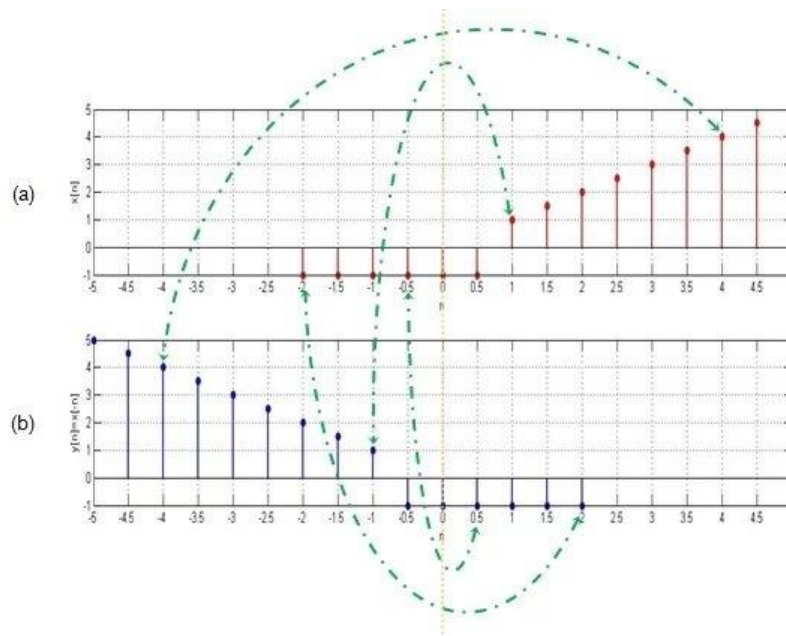



Figure 4. A signal with its reflection

**Analog frequency and Digital frequency**

The fundamental relation between the analog frequency, Ω , and the digital frequency, ω , is given by the following relation:

$$\omega = \Omega T \qquad (3.3a)$$

or alternately,

$$\omega = \Omega / f_s \qquad (3.3b)$$

where T is the sampling period, in sec., and fs =1/T is the sampling frequency in Hz.

Note, however, the following interesting points:

• The unit of Ω is radian/sec., whereas the unit of ω is just radians.

• The analog frequency, Ω , represents the actual physical frequency of the basic analog signal , for example, an audio signal (0 to 4 kHz) or a video signal (0 to 4 MHz). The digital frequency, ω , is the transformed frequency from Equation 3.3a or Equation 3.3b and can be considered as a mathematical frequency, corresponding to the digital signal.

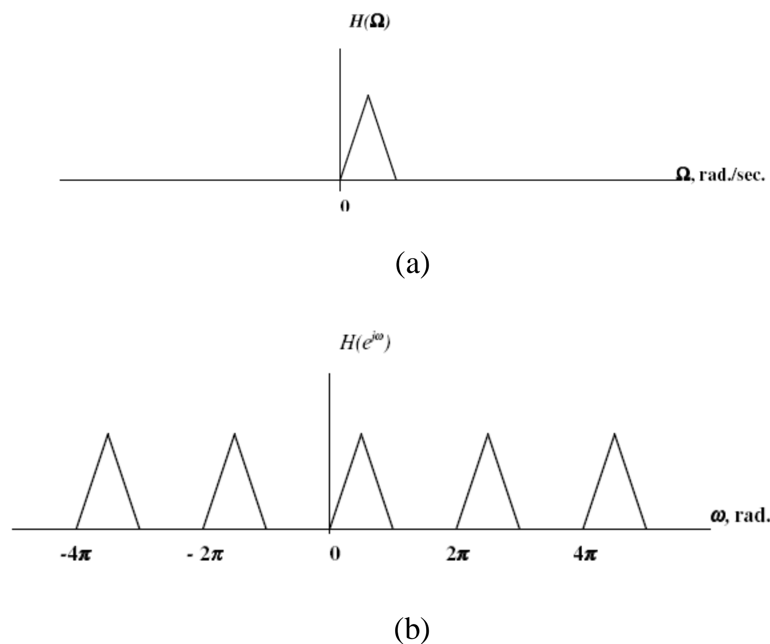



FIGURE 3.1 Analog frequency response and (b) digital frequency response

**Definition of Discrete time system**

System can be considered as a physical entity which manipulates one or more input signals applied to it. For example a microphone is a system which converts the input acoustic (voice or sound) signal into an electric signal. A system is defined mathematically as a unique operator or transformation that maps an input signal in to an output signal. This is defined as y(n) = T[x(n)] where x(n) is input signal, y(n) is output signal, T[] is transformation that characterizes the system behavior.

$$y(n) = T\,[x(n)]$$

$$\text{or,} \quad \mathbf{x(n)} \overset{T}{\rightarrow} \mathbf{y(n)}$$

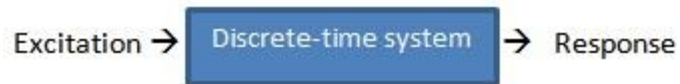
Where, T is the general rule or algorithm which is implemented on x(n) or the excitation to get the response y(n). For example, a few systems are represented as,

$$y(n) = -2x(n)$$

$$\text{or,} \quad y(n) = x(n-1) + x(n) + x(n+1)$$

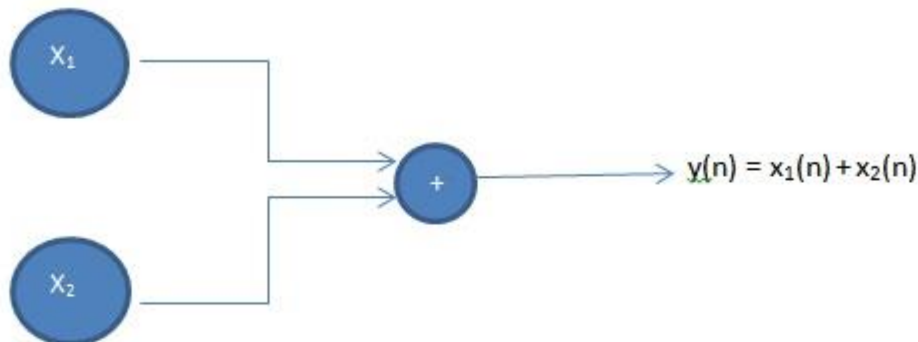
**Block Diagram representation of Discrete-time systems**

Digital Systems are represented with blocks of different elements or entities connected with arrows which also fulfills the purpose of showing the direction of signal flow,

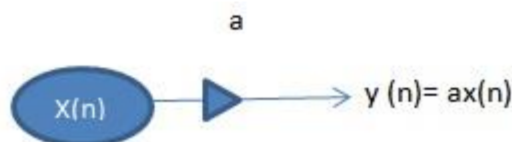



**Some common elements of Discrete-time systems are:-**

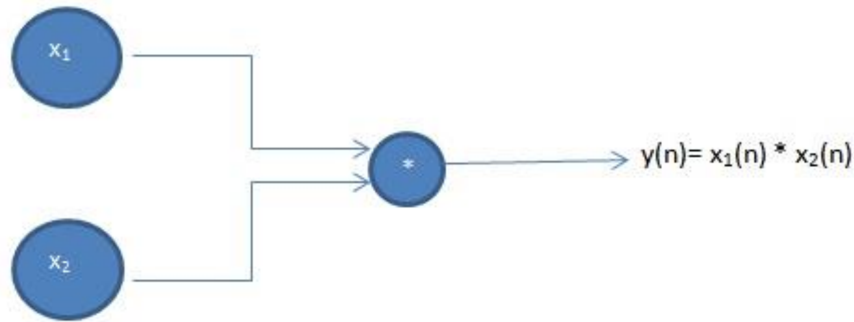
**Adder:** It performs the addition or summation of two signals or excitation to have a response. An adder is represented as,




**Constant Multiplier:** This entity multiplies the signal with a constant integer or fraction. And is represented as, in this example the signal x(n) is multiplied with a constant "a" to have the response of the system as y(n).




**Signal Multiplier:** This element multiplies two signals to obtain one.

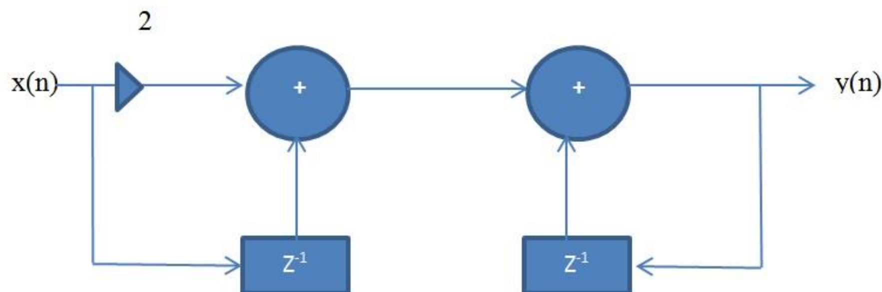



**Unit-delay element:** This element delays the signal by one sample i.e. the response of the system is the excitation of previous sample. This can element is said to have a memory which stores the excitation at time n-1 and recalls this excitation at the time n form the memory. This element is represented as,

**Unit-advance element:** This element advances the signal by one sample i.e. the response of the current excitation is the excitation of future sample. Although, as we can see this element is not physically realizable unless the response and the excitation are already in stored or recorded form.

Now that we have understood the basic elements of the Discrete-time systems we can now represent any discrete-time system with the help of block diagram. For example,

$$y(n) = y(n-1) + x(n-1) + 2x(n)$$




The above system is an example of Discrete-time system involving the unit delay of current excitation and also one unit delay of the current response of the system.

**Classification of Discrete-time Systems**

Discrete-time systems are classified on different principles to have a better idea about a particular system, their behavior and ultimately to study the response of the system.

**Relaxed system:** If $y(n_o -1)$ is the initial condition of a system with response y(n) and $y(n_o -1)=0$, then the system is said to be initially relaxed i.e. if the system has no excitation prior to $n_o$ .

**Static and Dynamic systems:** A system is said to be a Static discrete-time system if the response of the system depends at most on the current or present excitation and not on the past or future excitation. If there is any other scenario then the system is said to be a Dynamic discrete-time system. The static systems are also said to be memory-less systems and on the other hand dynamic systems have either finite or infinite memory depending on the nature of the system. Examples below will clear any arising doubts regarding static and dynamic systems.

$$y(n) = 2x(n) + nx^2(n) \qquad \{ \text{static-system} \}$$

$$y(n) = ax(n) \qquad \{ \text{static-system} \}$$

$$y(n) = ax(n) + bx(n-1) + cx(n+1) \qquad \{ \text{dynamic-system with finite memory} \}$$

$$y(n) = \sum_{k=0}^{n} x(n-k) \qquad \{ \text{dynamic-system with finite memory} \}$$

$$y(n) = \sum_{k=0}^{\infty} x(n-k) \qquad \{ \text{dynamic-system with in-finite memory} \}$$

The last example is the case of in-finite memory and the others are specified about their type depending on their characteristics.

**Time-variant and Time-invariant system:** A discrete-time system is said to be time invariant if the input-output characteristics do not change with time, i.e. if the excitation is delayed by k units then the response of the system is also delayed by k units. Let there be a system,

$$x(n) \quad \text{---->} \quad y(n) \qquad \forall\, x(n)$$

Then the relaxed system T is time-invariant if and only if,

$$x(n-k) \quad \text{---->} \quad y(n-k) \quad \forall\, x(n) \text{ and } k.$$

Otherwise, the system is said to be time-variant system if it does not follows the above specified set of rules. For example,

$$y(n) = ax(n) \qquad \{ \text{time-invariant} \}$$

$$y(n) = x(n) + x(n-3) \qquad \{ \text{time-invariant} \}$$

$$y(n) = nx(n) \qquad \{ \text{time-variant} \}$$

Note:- In order to check whether the system is time-invariant or time-variant the system must satisfy the "T[x(n-k)]=y(n-k)" condition, i.e. first delay the excitation by k units, then replace n

with (n-k) in the response and then equate L.H.S. and R.H.S. if they are equal then the system is time invariant otherwise not. For example in the last system above,

$$\text{L.H.S.} = T[x(n-k)] = nx(n-k)$$

{not (n-k)x(n-k) which is a general misconfusion}

$$\text{R.H.S.} = y(n-k) = (n-k)\, x(n-k)$$

So, the L.H.S. and R.H.S. are not equal hence the system is time-varient.

Note:- What about Folder, is it a time-variant or time-invariant system, let's see,

$$y(n) = x(-n)$$

$$\text{L.H.S.} = y(n-k) = x[-(n-k)] = x(-n+k)$$

$$\text{R.H.S.} = T[x(n-k)] = x(-n-k)$$

Thus, R.H.S. is not equal to L.H.S. so the system is time-variant.

**Linear and non-Linear systems:** A system is said to be a linear system if it follows the superposition principle i.e. the sum of responses (output) of weighted individual excitations (input) is equal to the response of sum of the weighted excitations. Pay attention to the above specified rule, according to the rule the following condition must be fulfilled by the system in order to be classified as a Linear system,

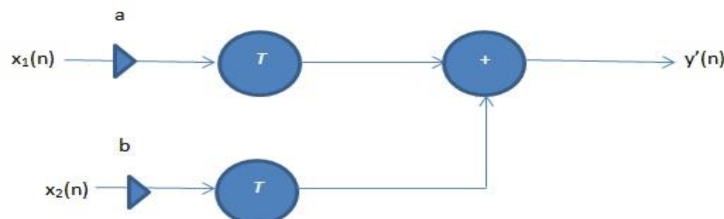
If, $y_1(n) = T[\, a x_1(n)\, ]$

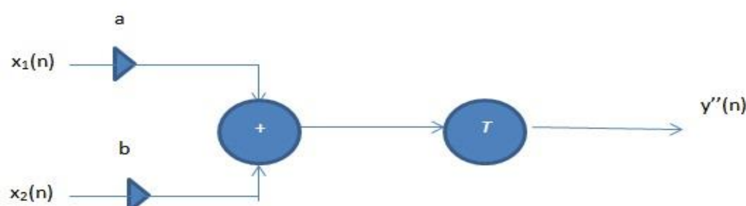
$y_2(n) = T[\, b x_2(n)\, ]$

and, $y(n) = T[a x_1(n) + b x_2(n)]$

Then, the system is said to be linear if ,

$$T[\, a x_1(n) + b x_2(n)] = T[\, a x_1(n)\, ] + T[\, b x_2(n)\, ]$$



So, if y'(n) = y''(n) then the system is said to be linear. I the system does not fulfills this property then the system is a non-Linear system. For example,

$$y(n) = x(n^2) \qquad \{\text{ linear }\}$$

$$y(n) = Ax(n) + B \qquad \{\text{non – linear }\}$$

$$y(n) = nx(n) \qquad \{\text{ linear }\}$$

The explanation of the above specified examples is left as an exercise for the reader.

Causal and non-Causal systems: A discrete-time system is said to be a causal system if the response or the output of the system at any time depends only on the present or past excitation or input and not on the future inputs. If the system T follows the following relation then the system is said to be causal otherwise it is a non-causal system.

$$y(n) = F[x(n), x(n-1), x(n-2), \ldots\ldots]$$

Where F[] is any arbitrary function. A non-causal system has its response dependent on future inputs also which is not physically realizable in a real-time system but can be realized in a recorded system. For example,

$$y(n) = \sum_{k=0}^{\infty} x(n-k) \qquad \{\text{ Causal }\}$$

$$y(n) = x(n) + x(n+1) \qquad \{\text{non-Causal }\}$$

$$y(n) = x(2n) \qquad \{\text{ non-Causal since, } y(n) = x(n+n) \ \}$$

Stable and Unstable systems: A system is said to be stable if the bounded input produces a bounded output i.e. the system is BIBO stable. If,

$$x(n) = M \quad \forall \quad -\infty < M < \infty$$

$$\text{then,} \qquad y(n) = N \quad \forall \quad -\infty < N < \infty$$

Then the system is said to be bounded system and if this is not the case then the system is unbounded or unstable

**ANALYSIS OF DISCRETE-TIME LINEAR TIME-INVARIANT SYSTEMS**

Systems are characterized in the time domain simply by their response to a unit sample sequence. Any arbitrary input signal can be decomposed and represented as a weighted sum of unit sample sequences.

Our motivation for the emphasis on the study of LTI systems is twofold. First there is a large collection of mathematical techniques that can be applied to the analysis of LTI systems. Second, many practical systems are either LTI systems or can be approximated by LTI systems.

As a consequence of the linearity and time-invariance properties of the system, the response of the system to any arbitrary input signal can be expressed in terms of the unit sample response of the system. The general form of the expression that relates the unit sample response of the system and the arbitrary input signal to the output signal, called the convolution sum

Thus we are able to determine the output of any linear, time-invariant system to any arbitrary input signal.

There are two basic methods for analyzing the behavior or response of a linear system to a given input signal.

The first method for analyzing the behavior of a linear system to a given input signal is first to decompose or resolve the input signal into a sum of elementary signals. The elementary signals are selected so that the response of the system to each signal component is easily determined. Then, using the linearity property of the system, the responses of the system to the elementary signals are added to obtain the total response of the system to the given input signal.

Suppose that the input signal $x(n)$ is resolved into a weighted sum of elementary signal components $\{x_k(n))$ so that

$$x(n) = \sum_k c_k x_k(n)$$

where the $\{c_k\}$ is the set of amplitudes (weighting coefficients) in the decomposition of the signal $x(n)$ . Now suppose that the response of the system to the elementary signal component $x_k(n)$ is $y_k(n)$. Thus

$$y_k(n) \equiv \mathcal{T}[x_k(n)]$$

assuming that the system is relaxed and that the response to $c_k x_k(n)$ is $c_k v_k(n)$ as a consequence of the scaling property of the linear system.

Finally, the total response to the input $x(n)$ is

$$\begin{aligned} y(n) &= \mathcal{T}[x(n)] = \mathcal{T}\left[\sum_k c_k x_k(n)\right] \\ &= \sum c_k \mathcal{T}[x_k(n)] \\ &= \sum_k c_k y_k(n) \end{aligned}$$

In **the above equation** we used the additivity property of the linear system.

**Resolution of a Discrete-Time Signal into Impulses**

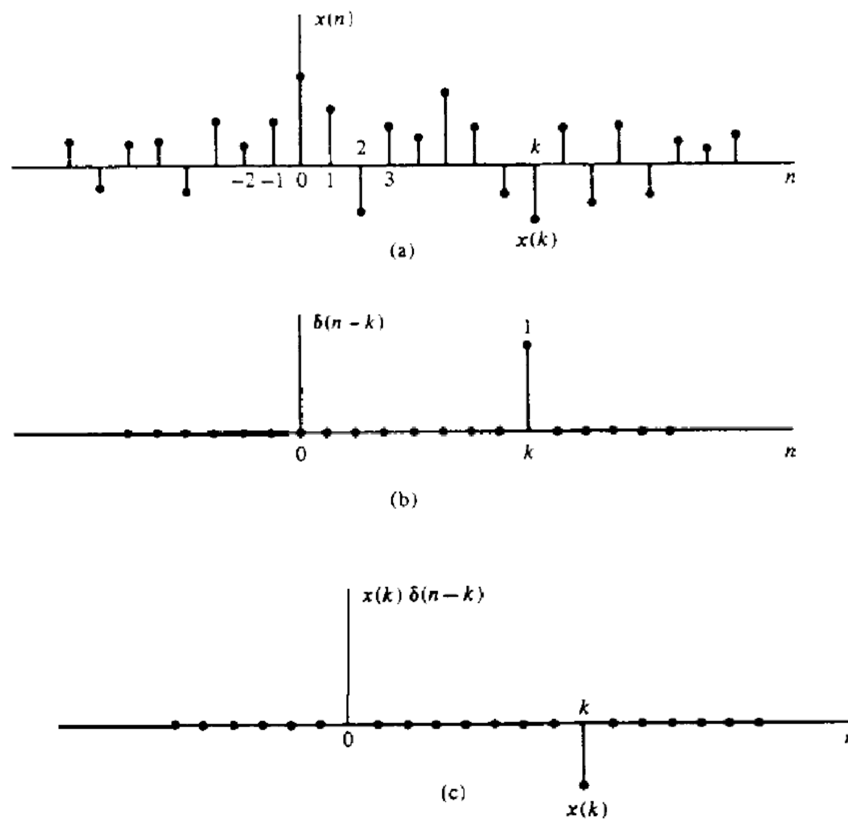
Suppose we have an arbitrary signal $x(n)$ that we wish to resolve into a sum of unit sample sequences. we

select the elementary signals $x_k(n)$ to be

$$x_k(n) = \delta(n - k)$$

where $k$ represents the delay of the unit sample sequence. To handle an arbitrary signal $x(n)$ that may have nonzero values over an infinite duration, the set of unit impulses must also be infinite, to encompass the infinite number of delays.

Now suppose that we multiply the two sequences $x(n)$ and $\delta(n - k)$. Since $\delta(n - k)$ is zero everywhere except at $n = k$. where its value is unity, the result of this multiplication is another sequence that is zero everywhere except at $n = k$. where its value is $x(k)$, as illustrated in Fig. below. Thus




**Multiplication of a signal $x(n)$ with a shifted unit sample sequence.**

If we repeat this multiplication over all possible delays, $-\infty < k < \infty$, and sum all the product sequences, the result will be a sequence equal to the sequence $x(n)$, that is,

$$x(n) = \sum_{k=-\infty}^{\infty} x(k)\delta(n - k)$$

**Example .**

**Consider the special case of a finite-duration sequence given as**

$$x(n) = \{2, \underset{\uparrow}{4}, 0, 3\}$$

**Resolve the sequence *x ( n )* into a sum of weighted impulse sequences.**

**Solution: Since the sequence *x ( n )* is nonzero for** the **time instants *n*** = **-1, 0. 2, we**

**need three impulses at delays *k*** = **- 1. 0, 2. Following (2.3.10) we find that**

$$x(n) = 2\delta(n+1) + 4\delta(n) + 3\delta(n-2)$$

**Response of LTI Systems to Arbitrary Inputs: The Convolution Sum**

we denote the response ***y( n,k )*** of the system to the input unit sample sequence at ***n***= ***k*** by the special symbol ***h(n.*** k), $-\infty < k < \infty$. That is,

$$y(n, k) \equiv h(n, k) = \mathcal{T}[\delta(n-k)]$$

***n*** is the time index and ***k*** is a parameter showing the location of the input impulse. If the impulse at the input is scaled by an amount ***ck*** **≡x(*k* )** th*e* response of the system is the correspondingly scaled output, that is,

$$c_k h(n, k) = x(k)h(n, k)$$

Finally, if the input is the arbitrary signal *x(n)* that is expressed as a sum of

weighted impulses. that is.

$$x(n) = \sum_{k=-\infty}^{\infty} x(k)\delta(n-k)$$

Then the response of the system to **x(n)** is the corresponding sum of weighted outputs, that is.

$$\begin{aligned} y(n) &= \mathcal{T}[x(n)] = \mathcal{T}\left[\sum_{k=-\infty}^{\infty} x(k)\delta(n-k)\right] \\ &= \sum_{k=-\infty}^{\infty} x(k)\mathcal{T}[\delta(n-k)] \\ &= \sum_{k=-\infty}^{\infty} x(k)h(n, k) \end{aligned}$$

The above equation follows from the superposition property of linear systems, and is known as the ***superposition summation.***

In the above equation we used the linearity property of the system bur nor its time invariance property.

Then by the time-invariance property, the response of the system to the delayed unit sample sequence $\delta(n - k)$ is

$$h(n-k) = \mathcal{T}[\delta(n-k)]$$

$$y(n) = \sum_{k=-\infty}^{\infty} x(k)h(n-k)$$

The formula above gives the response $y(n)$ of the LTI system as a function of the input signal $x(n)$ and the unit sample (impulse) response $h(n)$ is called a **convolution sum.**

The process of computing the convolution between $x(k)$ and $h(k)$ involves the following four steps.

1. ***Folding.*** Fold $h(k)$ about $k = 0$ to obtain $h(-k)$.

2. ***Shifting.*** Shift $h(-k)$ by $n_o$ to the right (left) if $no$ is positive (negative), to obtain $h(n_o - k)$.

**3,** ***Multiplication.*** Multiply $x(k)$ by $h(no - k)$ to obtain the product sequence

$$v_{no}(k) = x(k)h(n_o - k).$$

***4. Summation.*** Sum all the values of the product sequence $v_{no}(k)$ to obtain the

value of the output at time $n = n_o$.

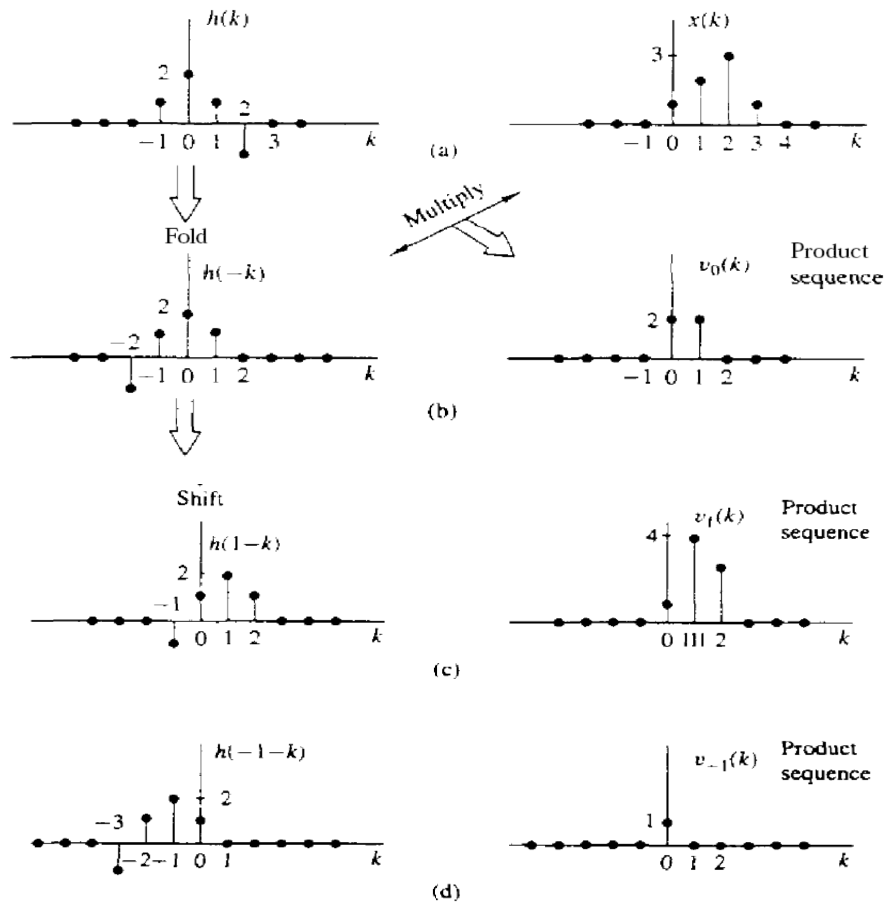
**Example .**

The impulse response of a linear time-invariant system is

$$h(n) = \{1, \underset{\uparrow}{2}, 1, -1\}$$

Determine the response of the system to the input signal

$$x(n) = \{\underset{\uparrow}{1}, 2, 3, 1\}$$




$$y(1) = \sum_{k=-\infty}^{\infty} v_1(k) = 8 \qquad y(-1) = 1$$

$$y(n) = \{\ldots, 0, 0, 1, \underset{\uparrow}{4}, 8, 8, 3, -2, -1, 0, 0, \ldots\}$$

# Filtering using Overlap-save and Overlap-add methods

In many applications one of the signals of a convolution is much longer than the other. For instance when filtering a speech signal $x_L[k]$ of length L with a room impulse response $h_N[k]$ of length $N \ll L$. In order to perform the convolution various techniques have been developed that perform the filtering on limited portions of the signals. These portions are known as partitions, segments or blocks. The respective algorithms are termed as *segmented* or *block-based* algorithms. The following section introduces two techniques for the block-based convolution of signals. The basic concept of these is to divide the convolution $y[k]=x_L[k] * h_N[k]$ into multiple convolutions operating on (overlapping) segments of the signal xL[k].

## Overlap-Add Algorithm

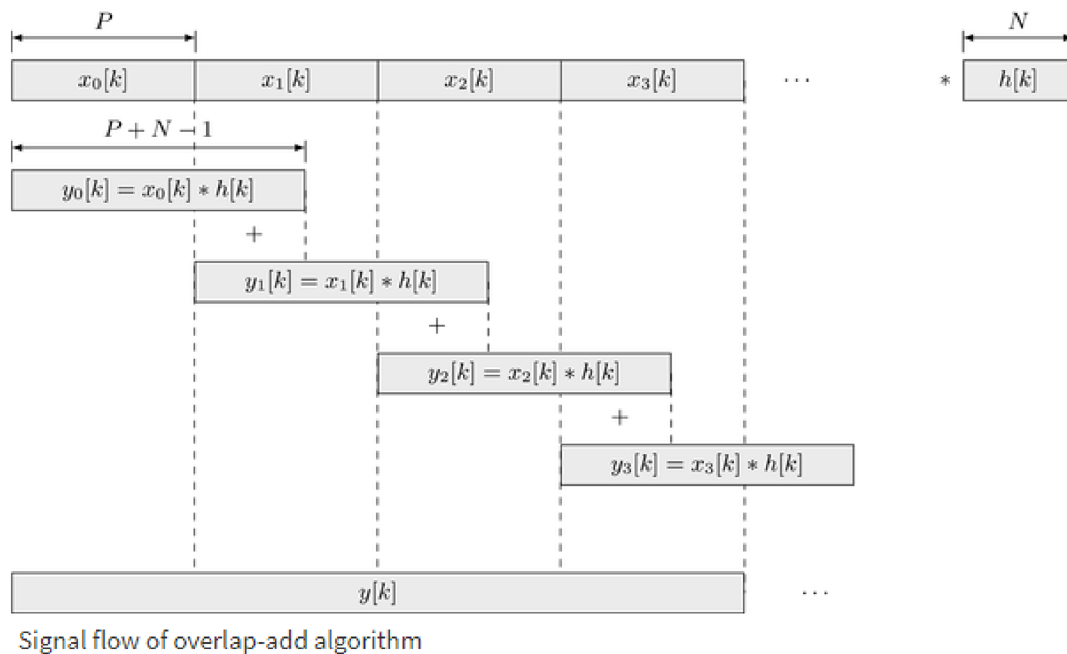
The overlap-add algorithm is based on splitting the signal $x_L[k]$ into non-overlapping segments $x_p[k]$ of length P.

$$x_L[k] = \sum_{p=0}^{L/P-1} x_p[k - p \cdot P]$$

where the segments $x_p[k]$ are defined as

$$x_p[k] = \begin{cases} x_L[k + p \cdot P] & \text{for } k = 0, 1, \dots, P-1 \\ 0 & \text{otherwise} \end{cases}$$

Note that $x_L[k]$ might have to be zero-padded so that its total length is a multiple of the segment length P. Introducing the segmentation of $x_L[k]$ into the convolution yields where $y_p[k]=x_p[k] * h_N[k]$. This result states that the convolution of $x_L[k]*hN[k]$ can be split into a series of convolutions $y_p[k]$ operating on the samples of one segment(block) only. The length of $y_p[k]$ is N+P−1. The result of the overall convolution is given by summing up the results from the segments shifted by multiples of the segment length P. This can be interpreted as an overlapped superposition of the results from the segments, as illustrated in the following diagram.



Signal flow of overlap-add algorithm

## Overlap-Save Algorithm

The overlap-save algorithm, also known as *overlap-discard algorithm*, follows a different strategy as the overlap-add technique introduced above. It is based on an overlapping segmentation of the input $x_L[k]$ and application of the periodic convolution for the individual segments.

Lets take a closer look at the result of the periodic convolution $x_p[k]*h_N[k]$, where $x_p[k]$ denotes a segment of length P of the input signal and $h_N[k]$ the impulse response of length N. The result of a linear convolution $x_p[k]*h_N[k]$ would be of length P+N−1. The result of the periodic convolution of period P for P>N would suffer from a circular shift (time aliasing) and superposition of the last N−1 samples to the beginning. Hence, the first N−1 samples are not equal to the result of the linear convolution. However, the remaining P−N+1 do so.

This motivates to split the input signal $x_L[k]$ into overlapping segments of length P where the p-th segment overlaps its preceding (p−1)-th segment by N−1 samples.

$$x_p[k] = \begin{cases} x_L[k + p \cdot (P - N + 1) - (N - 1)] & \text{for } k = 0, 1, \dots, P - 1 \\ 0 & \text{otherwise} \end{cases}$$

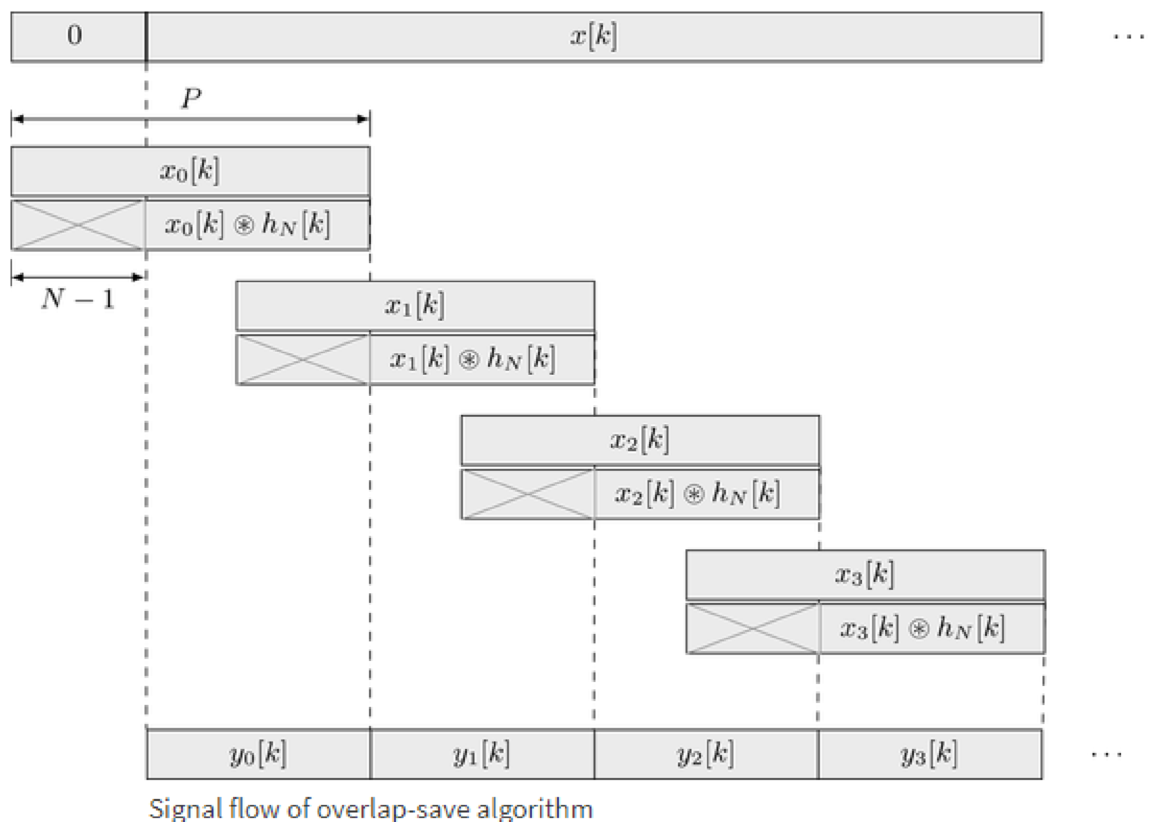
The part of the circular convolution $x_p[k]*h_N[k]$ of one segment $x_p[k]$ with the impulse response $h_N[k]$ that is equal to the linear convolution of both is given as

$$y_p[k] = \begin{cases} x_p[k] \circledast h_N[k] & \text{for } k = N - 1, N, \dots, P - 1 \\ 0 & \text{otherwise} \end{cases}$$

The output $y[k]$ is simply the concatenation of the $y_p[k]$

$$y[k] = \sum_{p=0}^{L/P-1} y_p[k - p \cdot (P - N + 1) + (N - 1)]$$

The overlap-save algorithm is illustrated in the following diagram.



Signal flow of overlap-save algorithm

For the first segment $x_0[k]$, N−1 zeros have to be appended to the beginning of the input signal $x_L[k]$ for the overlapped segmentation. From the result of the periodic convolution $x_p[k]*h_N[k]$ the first N−1 samples are discarded, the remaining P−N+1 are copied to the output y[k]. This is indicated by the alternative notation *overlap-discard* used for the technique

**Causal Linear Time-Invariant Systems**

In the case of a linear time-invariant system, causality can be translated to a condition on the impulse response. To determine this relationship, let us consider a linear time-invariant system having an output at time $n = n_0$ given by the convolution formula

$$y(n_0) = \sum_{k=-\infty}^{\infty} h(k)x(n_0 - k)$$

Suppose that we subdivide the sum into two sets of terms, one set involving present and past values of the input [i.e.. $x(n)$ for $n \leq n_o$] and one set involving future values of the input [i.e., $x(n)$. $n > n_o$]. Thus we obtain

$$\begin{aligned} y(n_0) &= \sum_{k=0}^{\infty} h(k)x(n_0 - k) + \sum_{k=-\infty}^{-1} h(k)x(n_0 - k) \\ &= [h(0)x(n_0) + h(1)x(n_0 - 1) + h(2)x(n_0 - 2) + \cdots] \\ &\quad + [h(-1)x(n_0 + 1) + h(-2)x(n_0 + 2) + \cdots] \end{aligned}$$

We observe that the terms in the first sum involve $x(n_0)$, $x(n_o - 1)$ . . . . , which are the present and past values of the input signal. On the other hand, the terms in the second sum involve the input signal components $x(n_o + 1)$, $x(n_o + 2)$. . . . . Now, if the output at time $n = n_0$ is depend only on the present and past inputs, then, clearly. the impulse response of the system must satisfy the condition

$$h(n) = 0 \qquad n < 0$$

Since $h(n)$ is the response of the relaxed linear time-invariant system to a unit impulse applied at $n = 0$, it follows that $h(n) = 0$ for $n < 0$ is both a necessary and a sufficient condition for causality. Hence an *LTI* ***system*** is ***causal if and only if its impulse response is zero for negative values of n.***

Since for a causal system, $h(n) = 0$ for $n < 0$. the limits on the summation of the convolution formula may be modified to reflect this restriction. Thus we have the two equivalent forms

$$\begin{aligned} y(n) &= \sum_{k=0}^{\infty} h(k)x(n - k) \\ &= \sum_{k=-\infty}^{n} x(k)h(n - k) \end{aligned}$$

Up to this point we have treated linear and time-invariant systems that are characterized by their unit sample response ***h(n).*** In turn h(n) allows us to determine the output ***y(n)*** of the system for any given input sequence ***x( n )*** by means of the convolution summation.

In the case of **FIR systems,** such a realization involves additions, Multiplications, and a finite number of memory locations. Consequently, an FIR system is readily implemented directly, as implied by the convolution summation.

If the system is **IIR**. however, its practical implementation as implied by convolution is clearly impossible. since it requires an infinite number of memory locations, multiplications, and additions. A question that naturally arises, then, is whether or not it is possible to realize IIR systems other than in the form suggested by the convolution summation. Fortunately, the answer is yes.

There is a practical and computationally efficient means for implementing a family of IIR systems, as will be demonstrated in this section, Within the general class of IIR systems. this family of discrete-time systems is more conveniently described by difference equations. This family or subclass of IIR systems is very useful in a variety of practical applications, including the implementation of digital filters, and the modeling of physical phenomena and physical systems.

**Recursive and Nonrecursive Discrete-Time Systems**

As indicated above, the convolution summation formula expresses the output of the linear time-invariant system explicitly and only in terms of the input signal.

However, this need not be the case, as is shown here. There are many systems where it is either necessary or desirable to express the output of the system not only in terms of the present and past values of the input, but also in terms of the already available past output values. The following problem illustrates this point.

Suppose that we wish to compute the ***cumulative average*** of a signal ***x ( n )*** in the interval $0 \leq k \leq n$, defined as

$$y(n) = \frac{1}{n+1} \sum_{k=0}^{n} x(k) \qquad n = 0, 1, \ldots$$

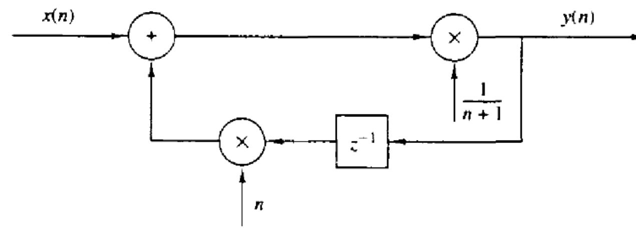
the computation of ***y( n )*** requires the storage of all the input samples ***x (k )*** for $0 \leq k \leq n$. Since *n* is increasing, our memory requirements grow linearly with time.

Our intuition suggests, however, that *y( n )* can be computed more efficiently

by utilizing the previous output value ***y(n - I )*** . Indeed, by a simple algebraic rearrangement , we obtain

$$\begin{aligned}(n+1)y(n) &= \sum_{k=0}^{n-1} x(k) + x(n) \\ &= ny(n-1) + x(n)\end{aligned}$$

$$y(n) = \frac{n}{n+1} y(n-1) + \frac{1}{n+1} x(n)$$




This is an example of a ***recursive system.***

***Difference Equations in Discrete□Time Systems***

Here a treatment of linear difference equations with constant coefficients and it is confined to first- and second-order difference equations and their solution. Higher-order difference equations of this type and their solution is facilitated with the Z□transform

**1-Recursive Method for Solving Difference Equations**

In mathematics, a *recursion* is an expression, such as a polynomial, each term of which is determined by application of a formula to preceding terms. The solution of a difference equation is often obtained by recursive methods. An example of a recursive method is Newton's method for solving non-linear equations. While recursive methods yield a desired result, they do not provide a **closed-form** solution. If a closed-form solution is desired, we can solve difference equations using the Method of Undetermined Coefficients, and this method is similar to the classical method of solving linear differential equations with constant coefficients.

**2-Method of Undetermined Coefficients**

A second-order difference equation has the form

$$y(n) + a_1 y(n-1) + a_2(n-2) = f(n) \qquad \text{(A.1)}$$

Where a*1*d $a_2$ are constants and the right side is some function of *n*. This differenc equation expresses the output *y*(*n*) at time *n* as the linear combination of two previous outputs *y*(*n*-1) and *y*(*n*-2). The right side of relation (A.1) is referred to as the **forcing function** The general (closed-form) solution of relation (A.1) is the same as that used for solving second-order differential equations. The three steps are as follows:

1. Obtain the **natural response** (complementary solution) in terms of two arbitrary real constants $k_1$ and $k_2$ , where $a_1$ and $a_2$ are also real constants, that is,

$$y_C(n) = k_1 a_1^n + k_2 a_2^n \qquad \text{(A.2)}$$

2. Obtain the forced response (particular solution) in terms of an arbitrary real constant $\mathbf{k_3}$ **,** that is,

$$y_P(n) = k_3 a_3^n \qquad \text{(A.3)}$$

where the right side of (A.3) is chosen with reference to Table A.1.

TABLE A.1 Forms of the particular solution for different forms of the forcing function

| Form of forcing function | Form of particular solution[a] |
|---|---|
| Constant | $k$ − a constant |
| $an^k$ − a is a constant | $k_0 + k_1 n + k_2 n^2 + \ldots + k_k n^k$ − $k_i$ is constant |
| $ab^{\pm n}$ − a and b are constants | Expression proportional to $b^{\pm n}$ |
| $a\cos(n\omega)$ or $a\sin(n\omega)$ | $k_1 \cos(n\omega) + k_2 \sin(n\omega)$ |

3. Add the natural response (complementary solution) $y_c(n)$ and the forced response (particular solution) $y_p(n)$to obtain the total solution, that is,

$$y(n) = y_C(n) + y_P(n) = k_1 a_1^n + k_2 a_2^n + y_P(n) \quad (A.4)$$

4. Solve for $k_1$ and $k_2$ in (A.4) using the given initial conditions. It is important to remember that the constants $k_1$ and $k_2$ must be evaluated from the total solution of (A.4), not from the complementary solution $y_c(n)$**.**

**Example 1**

Find the total solution for the second−order difference equation

$$y(n) - \frac{5}{6}y(n-1) + \frac{1}{6}y(n-2) = 5^{-n} \qquad n \ge 0 \quad (A.5)$$

subject to the initial conditions $y(-2) = 25$ and $y(-1) = 6$.

**Solution:**

1. We assume that the complementary solution $y_c(n)$ has the form

$$y_C(n) = k_1 a_1^n + k_2 a_2^n \quad (A.6)$$

The homogeneous equation of (A.5) is

$$y(n) - \frac{5}{6}y(n-1) + \frac{1}{6}y(n-2) = 0 \qquad n \ge 0 \quad (A.7)$$

Substitution of

$$y(n) = a^n$$

into (A.7) yields

$$a^n - \frac{5}{6}a^{n-1} + \frac{1}{6}a^{n-2} = 0 \quad (A.8)$$

Division of (A.8) by

$$a^{n-2}$$

**Yields**

$$a^2 - \frac{5}{6}a + \frac{1}{6} = 0 \qquad (A.9)$$

The roots of (A.9) are

$$a_1 = \frac{1}{2} \qquad a_2 = \frac{1}{3} \qquad (A.10)$$

and by substitution into (A.6) we obtain

$$y_C(n) = k_1\left(\frac{1}{2}\right)^n + k_2\left(\frac{1}{3}\right)^n = k_1 2^{-n} + k_2 3^{-n} \qquad (A.11)$$

2. Since the forcing function is

$$5^{-n},$$

, we assume that the particular solution is

$$y_P(n) = k_3 5^{-n} \qquad (A.12)$$

and by substitution into (A.5),

$$k_3 5^{-n} - k_3\left(\frac{5}{6}\right)5^{-(n-1)} + k_3\left(\frac{1}{6}\right)5^{-(n-2)} = 5^{-n}$$

Division of both sides by

$$5^{-n}$$

**Yields**

$$k_3\left[1 - \left(\frac{5}{6}\right)5 + \left(\frac{1}{6}\right)5^2\right] = 1$$

**Or k=1 and thus**

$$y_P(n) = 5^{-n} \qquad (A.13)$$

The total solution is the addition of (A.11) and (A.13), that is,

$$y(n) = y_C(n) + y_P(n) = k_1 2^{-n} + k_2 3^{-n} + 5^{-n} \quad (A.14)$$

To evaluate the constants $k_1$ and $k_2$ we use the given initial conditions, i.e., s $y(-2) = 25$ and $y(-1) = 6$. For $n = -2$, (A.14) reduces to

$$y(-2) = k_1 2^2 + k_2 3^2 + 5^2 = 25$$

from which

$$4k_1 + 9k_2 = 0 \quad (A.15)$$

For $n = -1$, (A.14) reduces to

$$y(-1) = k_1 2^1 + k_2 3^1 + 5^1 = 6$$

from which

$$2k_1 + 3k_2 = 1 \quad (A.16)$$

Simultaneous solution of (A.15) and (A.16) yields

$$k_1 = \frac{3}{2} \qquad k_2 = -\frac{2}{3} \quad (A.17)$$

and by substitution into (A.14) we obtain the total solution as

$$y(n) = y_C(n) + y_P(n) = \left(\frac{3}{2}\right)2^{-n} + \left(-\frac{2}{3}\right)3^{-n} + 5^{-n} \qquad n \geq 0 \quad (A.18)$$

## IMPLEMENTATION OF DISCRETE-TIME SYSTEMS

In practice, system design and implementation are usuaHy treated jointly rather than separately. Often, the system design is driven by the method of implementation and by implementation constraints, such as cost. hardware limitations, size limitations, and power requirements. At this point, we have not as yet developed the necessary analysis and design tools to treat such complex issues. However, we have developed sufficient background to consider some basic implementation methods for realizations of LTI systems described by linear constant-coefficient difference equations.

### Structures for the Realization of Linear Time-Invariant Systems

In this subsection we describe structures for the realization of systems described by linear constant-coefficient difference equations.

As a beginning, let us consider the first-order system

$$y(n) = -a_1 y(n-1) + b_0 x(n) + b_1 x(n-1)$$

which is realized as in Fig. a. This realization uses separate delays (memory) for both the input and output signal samples and it is called a ***direct form*** I ***structure.***

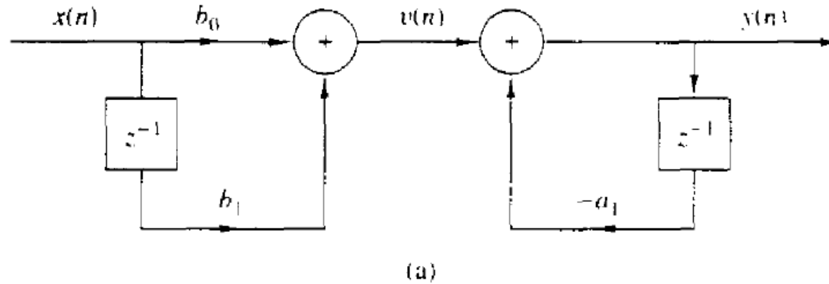
Note that this system can be viewed as two linear time-invariant systems in cascade.

The first is a nonrecursive, system described by the equation

$$v(n) = b_0 x(n) + b_1 x(n-1)$$

whereas the second is a recursive system described by the equation

$$y(n) = -a_1 y(n-1) + v(n)$$




(a)

Thus if we interchange the order of the recursive and nonrecursive systems, we obtain an alternative structure for the realization of the system described above**.** The resulting system is shown in Fig. b. From this figyre we obtain

the two difference equations

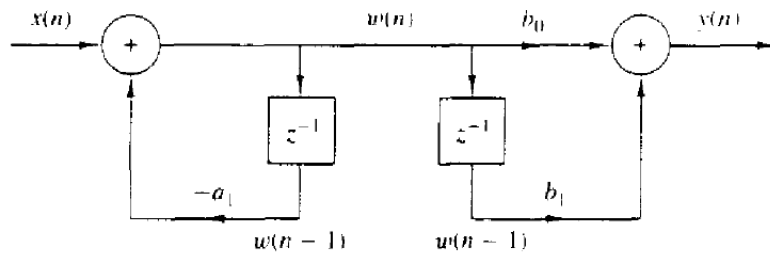
$$w(n) = -a_1 w(n-1) + x(n)$$

$$y(n) = b_0 w(n) + b_1 w(n-1)$$

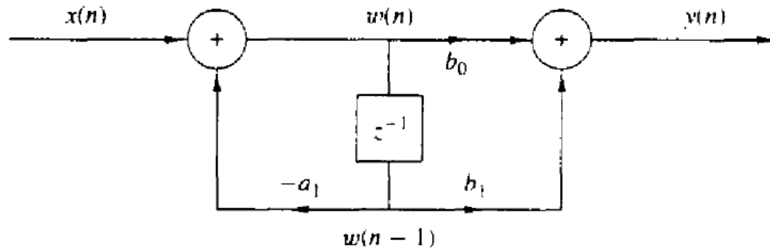
which provide an alternative algorithm for computing the output of the system described by the single difference equation given first. In other words. The last two difference equations are equivalent to the single difference equation .

A close observation of Fig. **a,b** reveals that the two delay elements contain the same input ***w(n)*** and hence the same output ***w(n-1)***. Consequently. these

two elements can be merged into one delay, as shown in Fig. **c.** In contrast




(b)




(c)

to the direct form I structure, this new realization requires only one delay for the auxiliary quantity ***w(n),*** and hence it is more efficient in terms of memory requirements. It is called the ***direct*** form 11 structure and it is used extensively in practical applications. These structures can readily be generalized for the general linear time-invariant recursive system described by the difference equation

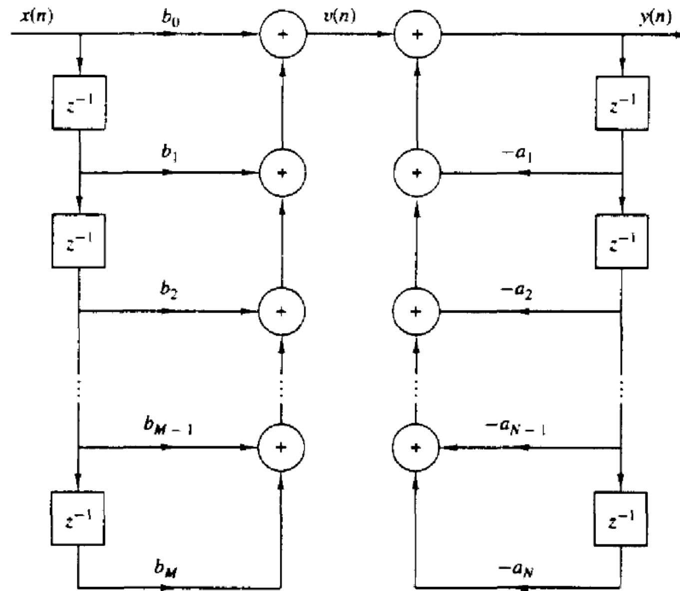
$$y(n) = -\sum_{k=1}^{N} a_k y(n-k) + \sum_{k=0}^{M} b_k x(n-k)$$

Figure below illustrates the direct form I structure for this system. This structure requires M + N delays and N + ***M*** + 1 multiplications. It can be viewed as the cascade of a nonrecursive system

$$v(n) = \sum_{k=0}^{M} b_k x(n-k)$$

and a recursive system

$$y(n) = -\sum_{k=1}^{N} a_k y(n-k) + v(n)$$




By reversing the order of these two systems as was previously done for the first-order system, **we** obtain the direct form **I1** structure shown in Fig. below for N > M. This structure is the cascade of a recursive system

$$w(n) = -\sum_{k=1}^{N} a_k w(n-k) + x(n)$$

followed **by** a nonrecursive system

$$y(n) = \sum_{k=0}^{M} b_k w(n-k)$$

We observe that if N $\geq$ M, this structure requires a number of delays equal to the order N of the system. However, if M > ***N,*** the required memory is specified by M. Figure above can easily by modified to handle this case. Thus the direct form **I1** structure requires M + ***N*** + 1 multiplications and max(M, N} delays. Because it requires the minimum number of delays for the realization of the system described by *given difference equation.*

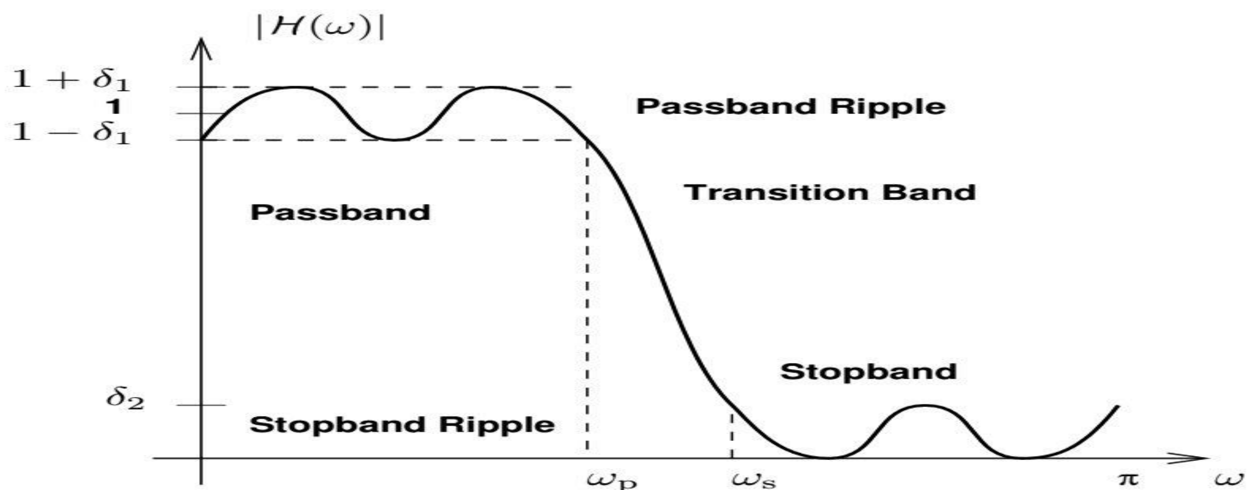
# MODULE – V

# IIR FILTER AND FIR FILTERS

**Introduction**

Filters are used in a wide variety of applications. Most of the time, the final aim of using a filter is to achieve a better frequency selectivity on the spectrum of the input signal. At this point of time, it is required to reviewing the frequency response of a practical filter. The below Figure (A) shows an example of a practical low pass filter.

In this example, frequency components in the pass band, from DC to $\omega_P$ will pass through the filter almost with no attenuation. The components in the stop band, above $\omega_S$ will experience significant attenuation. Note that the frequency response of a practical filter cannot be absolutely flat in the pass band or in the stop band. As shown in Figure (A), some ripples will be unavoidable and the transition band, $\omega p<\omega<\omega s$ cannot be infinitely sharp in practice.




Digital filter design involves four steps:

**1) Determining specifications**

First, we need to determine what specifications are required. This step completely depends on the application. This information is necessary to find the filter with minimum order for this application.

2) **Finding a transfer function**

With design specifications known, we need to find a transfer function which will provide the required filtering. The rational transfer function of a digital filter is as given below.

$$H(z) = \frac{\sum_{k=0}^{M-1} b_k z^{-k}}{\sum_{k=0}^{N-1} a_k z^{-k}}$$

**3) Choosing a realization structure**

Now that H(z) is known, we should choose the realization structure. In other words, there are many systems which can give the obtained transfer function and we must choose the appropriate one. For example, any of the direct form I, II, cascade, parallel, transposed, or lattice forms can be used to realize a particular transfer function. The main difference between the aforementioned realization structures is their sensitivity to using a finite length of bits. Note that in the final digital system, we will use a finite length of bits to represent a signal or a filter coefficient. Some realizations, such as direct forms, are very sensitive to quantization of the coefficients. However, cascade and parallel structures show smaller sensitivity and are preferred.

4) **Implementing the filter**

After deciding on what realization structure to use, we should implement the filter. You have a couple of options for this step: a software implementation (such as a MATLAB or C code) or a hardware implementation (such as a DSP, a microcontroller, or an ASIC).

it is necessary to take into account all fundamental characteristics of a signal to be filtered as these are very important when deciding which filter to use. In most cases, it is only one characteristic that really matters and it is whether it is necessary that filter has linear phase characteristic or not. it is necessary that a filter has linear phase characteristic to prevent loosing important information.When a signal to be filtered is analysed in this way, it is easy to decide which type of digital filter is best to use. Accordingly, if the phase characteristic is of the essence, FIR filters should be used as they have linear phase characteristic. Such filters are of higher order and more complex, therefore. The FIR Filters can be easily designed to have perfectly linear Phase. These filters can be realized recursively and Non-recursively. There are greater flexibility to control the Shape of their Magnitude response. Errors due to round off noise are less severe in FIR Filters, mainly because Feed back is not used

An FIR digital filter of order M may be implemented by programming the signal-flow-graph shown below. Its difference equation is:

y[n] = a0x[n] + a1x[n-1] + a2x[n-2] + ... + aMx[n-M]

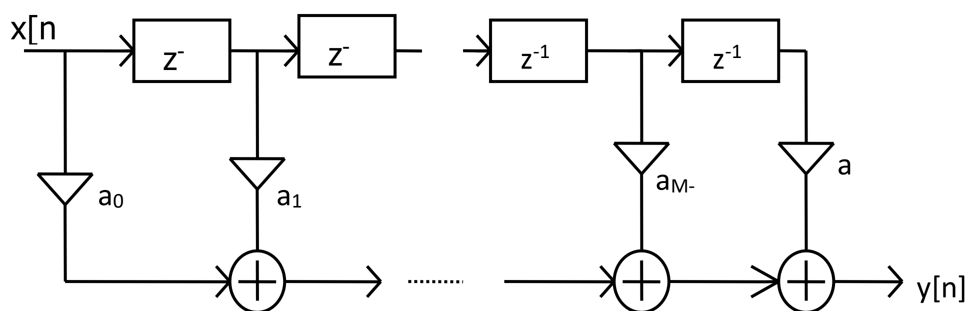



Fig. 4.1

Its impulse-response is {..., 0, ..., a0, a1, a2,..., aM, 0, ...} and its frequency-response is the DTFT of the impulse-response, i.e.

$$H(e^{j\Omega}) = \sum_{n=-\infty}^{\infty} h[n]e^{-j\Omega n} = \sum_{n=0}^{M} a_n e^{-j\Omega n}$$

Now consider the problem of choosing the multiplier coefficients. a0, a1,..., aM such that H( ejΩ ) is close to some desired or target frequency-response H′(ejΩ) say.  The inverse DTFT of H'(ejΩ) gives the required impulse-response :

$$h'[n] = \frac{1}{2\pi}\int_{-\pi}^{\pi} H'(e^{j\Omega})e^{j\Omega n}d\Omega$$

The methodology is to use the inverse DTFT to get an impulse-response {h′[n]} & then realise some approximation to it  Note that the DTFT formula is an integral, it has complex numbers and the range of integration is from -π to π, so it involves negative frequencies.

**What about the negative frequencies?**

Examine the DTFT formula for H(ejΩ).

$$H(e^{j\Omega}) = \sum_{n=-\infty}^{\infty} h[n]e^{-j\Omega n} \qquad \therefore \quad H(e^{-j\Omega}) = \sum_{n=-\infty}^{\infty} h[n]e^{j\Omega n}$$

If h[n] real then h[n]ejΩ is complex-conjugate of  h[n]e-jΩ.  Adding up terms gives H(e-jΩ ) as complex conj of H(ejΩ).

G(Ω) = G(-Ω) since G(Ω) = |H(ejΩ)| &  G(-Ω) = H(e-jΩ)|

Because of the range of integration (-π to π) of the DTFT formula, it is common to plot graphs of G(Ω) and ϕ(Ω) over the frequency range -π to π rather than 0 to π.  As G(Ω) = G(-Ω) for a real filter the gain-response will always be symmetric about Ω=0.

**4.2 Features of FIR Filter**

1. FIR filter always provides linear phase response. This specifies that the signals in the pass band will suffer no dispersion Hence when the user wants no phase distortion, then FIR filters are preferable over IIR. Phase distortion always degrade the system performance. In various applications like speech processing, data transmission over long distance FIR filters are more preferable due to this characteristic.

2. FIR filters are most stable as compared with IIR filters due to its non feedback nature.

3. Quantization Noise can be made negligible in FIR filters. Due to this sharp cutoff FIR filters can be easily designed.

4. Disadvantage of FIR filters is that they need higher ordered for similar magnitude response of IIR filters.

Difference equation of FIR filter of length M is given as

$$y(n)=\sum_{k=0}^{M-1} b_k\, x(n-k) \qquad (1)$$

And the coefficient  $b_k$ are related to unit sample response as    H(n) = bn for 0 ≤ n ≤ M-1,

= 0 otherwise

We can expand this equation as Y(n)= b0 x(n) + b1 x(n-1) + …….. + bM-1 x(n-M+1) (2)

System is stable only if system produces bounded output for every bounded input. This is stability definition for any system. Here h(n)={b0, b1, b2, } of the FIR filter are stable. Thus y(n) is bounded if input x(n) is bounded. This means FIR system produces bounded output for every bounded input. Hence FIR systems are always stable.

**The main features of FIR filter are,**

• They are inherently stable

• Filters with linear phase characteristics can be designed

• Simple implementation – both recursive and non-recursive structures possible

• Free of limit cycle oscillations when implemented on a finite-word length digital system

**Disadvantages:**

• Sharp cutoff at the cost of higher order

• Higher order leading to more delay, more memory and higher cost of implementation

Of these, the linear phase property is probably the most important. A filter is said to have a generalised linear phase response if its frequency response can be expressed in the form

$$H(e^{j\omega}) = A(e^{j\omega})e^{-j\alpha\omega + j\beta}$$

where $\alpha$ and $\beta$ are constants, and $A(e^{j\omega})$ is a real function of $\omega$. If this is the case, then

- If $A$ is positive, then the phase is

$$\sphericalangle H(e^{j\omega}) = \beta - \alpha\omega.$$

If $A$ is negative, then

$$\sphericalangle H(e^{j\omega}) = \pi + \beta - \alpha\omega.$$

In either case, the phase is a linear function of $\omega$.

It is common to restrict the filter to having a real-valued impulse response $h[n]$, since this greatly simplifies the computational complexity in the implementation of the filter.

A FIR system has linear phase if the impulse response satisfies either the even symmetric condition

$$h[n] = h[N - 1 - n],$$

or the odd symmetric condition

$$h[n] = -h[N - 1 - n].$$

The system has different characteristics depending on whether $N$ is even or odd. Furthermore, it can be shown that all linear phase filters must satisfy one of these conditions. Thus there are exactly four types of linear phase filters.

Consider for example the case of an odd number of samples in $h[n]$, and even symmetry. The frequency response for $N = 7$ is

$$\begin{aligned} H(e^{j\omega}) &= \sum_{n=0}^{6} h[n]e^{-j\omega n} \\ &= h[0] + h[1]e^{-j\omega} + h[2]e^{-j2\omega} + h[3]e^{-j3\omega} + h[4]e^{-j4\omega} \\ &\qquad + h[5]e^{-j5\omega} + h[6]e^{-j6\omega} \\ &= e^{-j3\omega}(h[0]e^{j3\omega} + h[1]e^{j2\omega} + h[2]e^{j\omega} + h[3] + h[4]e^{-j\omega} \\ &\qquad + h[5]e^{-j2\omega} + h[6]e^{-j3\omega}). \end{aligned}$$

The specified symmetry property means that $h[0] = h[6]$, $h[1] = h[5]$, and $h[2] = h[4]$, so

$$\begin{aligned} H(e^{j\omega}) &= e^{-j3\omega}(h[0](e^{j3\omega} + e^{-j3\omega}) + h[1](e^{j2\omega} + e^{-j2\omega}) \\ &\qquad + h[2](e^{j\omega} + e^{-j\omega}) + h[3]) \\ &= e^{-j3\omega}(2h[0]\cos(3\omega) + 2h[1]\cos(2\omega) + 2h[2]\cos(\omega)) \\ &= e^{-j3\omega}\sum_{n=0}^{3} a[n]\cos(\omega n), \end{aligned}$$

where $a[0] = h[3]$, and $a[n] = 2h[3-n]$ for $n = 1, 2, 3$. The resulting filter clearly has a linear phase response for real $h[n]$. It is quite simple to show that in general for odd values of $N$ the frequency response is

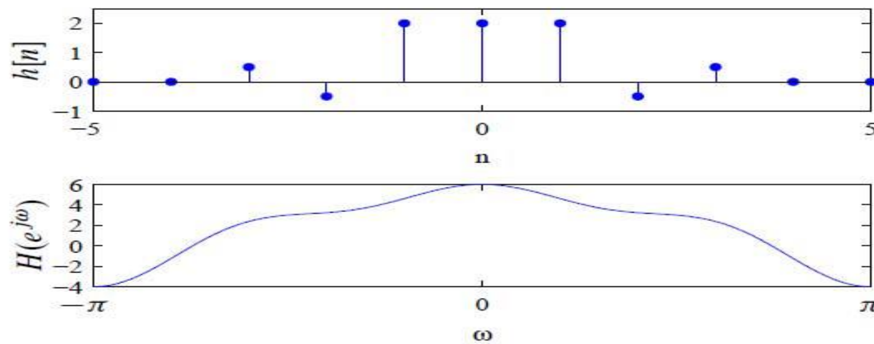
$$H(e^{j\omega}) = e^{-j\omega(N-1)/2}\sum_{n=0}^{(N-1)/2} a[n]\cos(\omega n),$$

for a set of real-valued coefficients $a[0], \ldots, a[(N-1)/2]$. As different values for $a[n]$ are selected, different linear-phase filters are obtained.

The cases of $N$ odd and $h[n]$ antisymmetric are similar to that presented, and the frequency responses are summarised in the following table:

| **Symmetry** | **N** | $H(e^{j\omega})$ | **Type** |
|---|---|---|---|
| Even | Odd | $e^{-j\omega(N-1)/2}\sum_{n=0}^{(N-1)/2} a[n]\cos(\omega n)$ | 1 |
| Even | Even | $e^{-j\omega(N-1)/2}\sum_{n=1}^{N/2} b[n]\cos(\omega(n-1/2))$ | 2 |
| Odd | Odd | $e^{-j[\omega(N-1)/2-\pi/2]}\sum_{n=0}^{(N-1)/2} a[n]\sin(\omega n)$ | 3 |
| Odd | Even | $e^{-j[\omega(N-1)/2-\pi/2]}\sum_{n=1}^{N/2} b[n]\sin(\omega(n-1/2))$ | 4 |

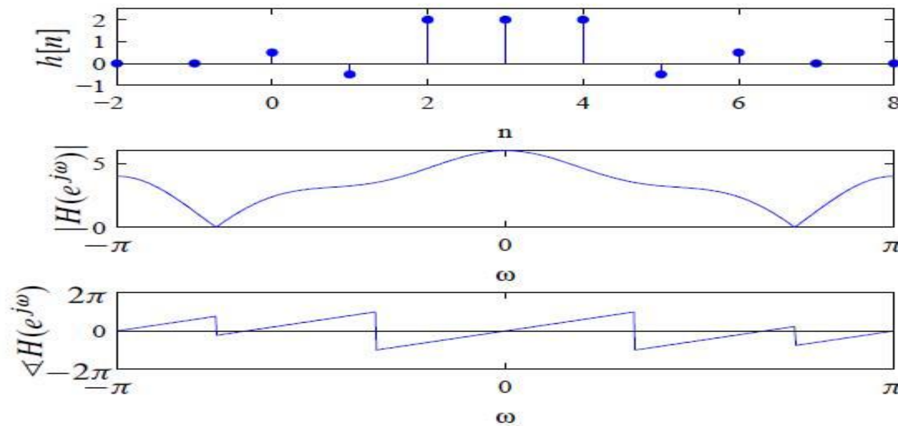
Recall that even symmetry implies $h[n] = h[N-1-n]$ and odd symmetry $h[n] = -h[N-1-n]$. Examples of filters satisfying each of these symmetry conditions are:




Recall from the properties of the Fourier transform this filter has a real-valued frequency response $A(e^{j\omega})$. Delaying this impulse response by $(N-1)/2$ results in a causal filter with frequency response

$$H(e^{j\omega}) = A(e^{j\omega})e^{-j\omega(N-1)/2}.$$

This filter therefore has linear phase.




**4.3 Symmetric and Anti-symmetric FIR filters**

Unit sample response of FIR filters is symmetric if it satisfies following condition.

h(n)= h(M-1-n), for n=0,1,2…………….M-1 2.

Unit sample response of FIR filters is Anti-symmetric if it satisfies following condition

h(n)= -h(M-1-n) for n=0,1,2.

FIR filters giving out Linear Phase characteristics: Symmetry in filter impulse response will ensure linear phase An FIR filter of length M with i/p x(n) & o/p y(n) is described by the difference_equation

$$y(n)= b_0\, x(n) + b_1\, x(n-1)+\ldots\ldots.+b_{M-1}\, x(n-(M-1)) = \sum_{k=0}^{M-1} b_k\, x(n-k) \qquad -(1)$$

Alternatively. it can be expressed in convolution form

$$y(n) = \sum_{k=0}^{M-1} h(k)x(n-k)$$

i.e $b_k = h(k)$, k=0,1,….M-1

**Choice of Symmetric and anti-symmetric unit sample response**

When we have a choice between different symmetric properties, the particular one is picked up based on application for which the filter is used. The following points give an insight to this issue.

- If h(n)=-h(M-1-n) and M is odd, Hr(w) implies that Hr(0)=0 & Hr(π)=0, consequently not suited for low pass and high pass filter. This condition is suited in Band Pass filter design.
- Similarly if M is even Hr(0)=0 hence not used for low pass filter
- Symmetry condition h(n)=h(M-1-n) yields a linear-phase FIR filter with non zero response at w = 0 if desired. Looking at these points, anti-symmetric properties are not generally preferred

**Poles & Zeros of linear phase sequences:**

The poles of any finite-length sequence must lie at z=0. The zeros of linear phase sequence must occur in conjugate reciprocal pairs. Real zeros at z=1 or z=-1 need not be paired (they form their own reciprocals), but all other real zeros must be paired with their reciprocals. Complex zeros on the unit circle must be paired with their conjugate (that form their reciprocals) and complex zeros anywhere else must occur in conjugate reciprocal quadruples. To identify the type of sequence from its pole-zero plot, all we need to do is check for the presence of zeros at z= □ and count their number. A type-2 seq must have an odd number of zeros at z=-1, a type-3 seq must have an odd number of zeros at z=-1 and z=1, and type-4 seq must have an odd number of zeros at z=1. The no. of other zeros if present (at z=1 for type=1 and type-2 or z=-1 for type-1 or type-4) must be even.

**Zeros of Linear Phase FIR Filters:**

Consider the filter system function

$$H(z) = \sum_{n=o}^{M-1} h(n) z^{-n}$$

Expanding this equation

$$H(z) = h(0) + h(1)z^{-1} + h(2)z^{-2} \quad ++ h(M-2)z^{-(M-2)} + h(M-1)z^{-(M-1)}$$

$$\sin ce\ for\ Linear-phase\ we\ need$$

$$h(n) = h(M-1-n) \quad i.e.,$$

$$h(0) = h(M-1);\ h(1) = h(M-2);\ldots\ldots h(M-1) = h(0);$$

$$then$$

$$H(z) = h(M-1) + h(M-2)z^{-1} + \ldots\ldots\ldots + h(1)z^{-(M-2)} + h(0)z^{-(M-1)}$$

$$H(z) = z^{-(M-1)}[h(M-1)z^{(M-1)} + h(M-2)z^{(M-2)} + \ldots\ldots + h(1)z + h(0)]$$

$$H(z) = z^{-(M-1)}[\sum_{n=0}^{M-1} h(n)(z^{-1})^{-n}] = z^{-(M-1)} H(z^{-1})$$

This shows that if z = z1 is a zero then $z=z1^{-1}$ is also a zero

The different possibilities: 1. If z1 = 1 then $z1 = z1^{-1} = 1$ is also a zero implying it is one zero

2. If the zero is real and |z|<1 then we have pair of zeros

3. If zero is complex and |z|=1then and we again have pair of complex zeros.

4. If zero is complex and |z|≠1 then and we have two pairs of complex zeros

### 4.4 FIR Filter Design Methods

The various method used for FIR Filer design are as follows

1. Fourier Series method
2. Windowing Method
3. DFT method
4. Frequency sampling Method. (IFT Method)

### 4.4.1. Design of an FIR low-pass digital filter

Assume we require a low-pass filter whose gain-response approximates the ideal 'brick-wall' gain-response in Figure 4.2.

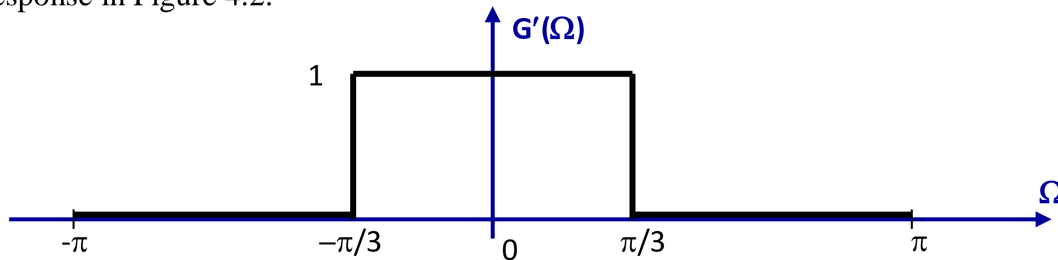



Fig. 4.2

If we take the phase-response φ′(Ω) to be zero for all Ω, the required frequency-response is:-

$$H'(e^{j\Omega}) = G'(\Omega)e^{j\phi(\Omega)} = \begin{cases} 1 & : \quad |\Omega| \le \pi/3 \\ 0 & : \; \pi/3 < |\Omega| < \pi \end{cases}$$

And by the inverse DTFT,

$$h'[n] = \frac{1}{2\pi}\int_{-\pi/3}^{\pi/3} 1 e^{j\Omega n} d\Omega = \begin{cases} 1/3 & : \text{ n} = 0 \\ (1/n\pi)\sin(n\pi/3) & : \text{ n} \ne 0 \end{cases}$$

= (1/3)sinc(n/3) for all n.

$$\text{where } \operatorname{sinc}(x) = \begin{cases} \dfrac{\sin(\pi x)}{\pi x} : x \ne 0 \\ 1 : x = 0 \end{cases}$$

A graph of sinc(x) against x is shown below:

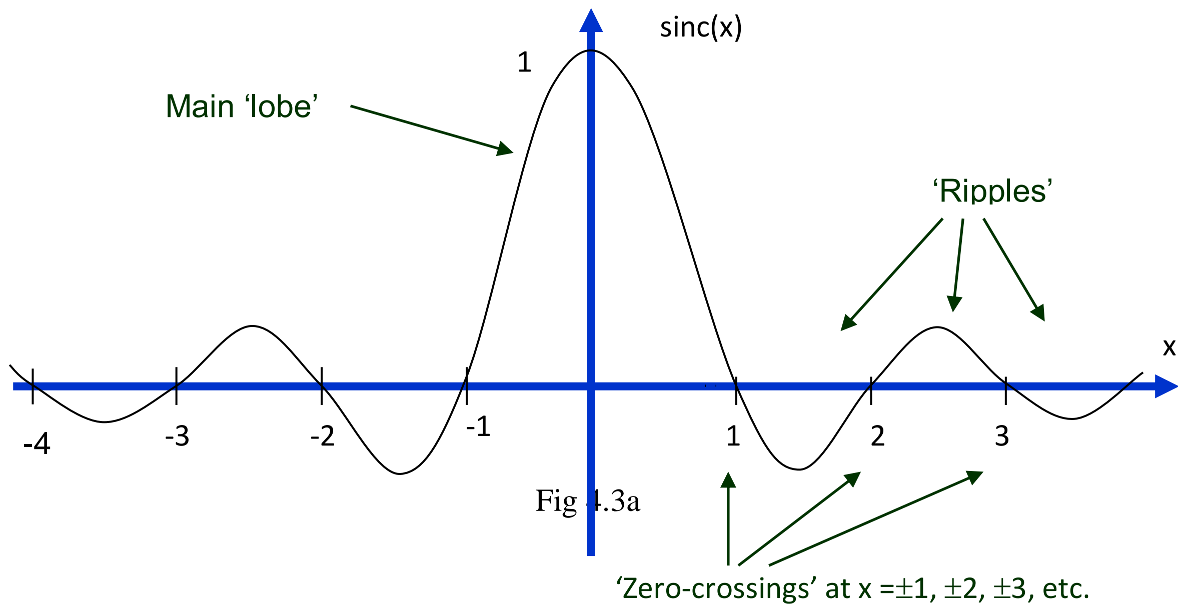



Fig 4.3a

The ideal impulse-response {h′[n]} with each sample h′[n] = (1/3) sinc(n/3) is therefore as follows:

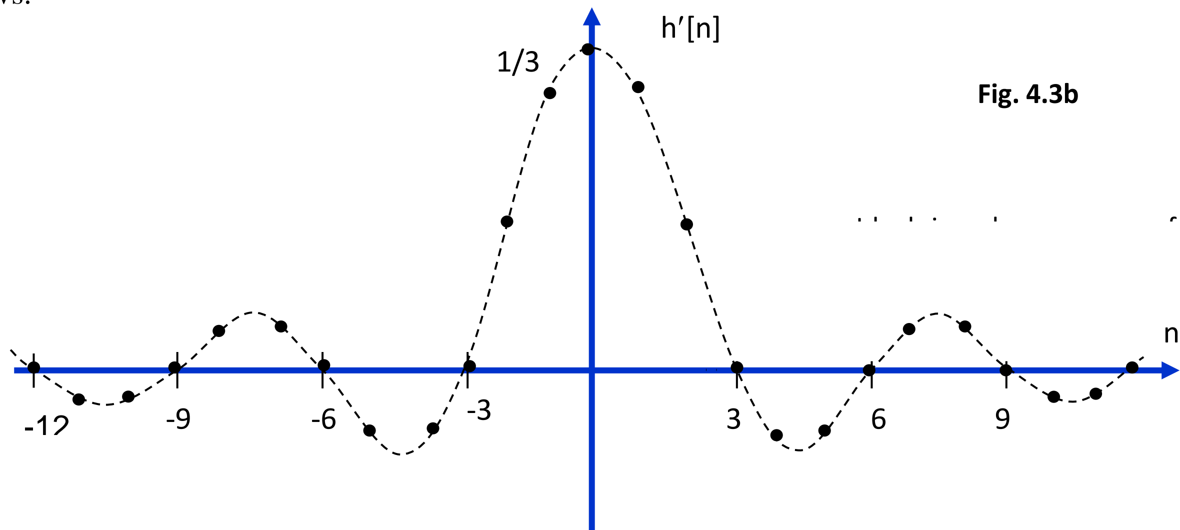



**Fig. 4.3b**

In Fourier series method, limits of summation index is -∞ to ∞. But filter must have finite terms.

Hence limit of summation index change to -Q to Q where Q is some finite integer. But this type of truncation may result in poor convergence of the series. Abrupt truncation of infinite series is equivalent

to multiplying infinite series with rectangular sequence. i.e at the point of discontinuity some oscillation may be observed in resultant series.

2. Consider the example of LPF having desired frequency response Hd (ω) as shown in figure. The oscillations or ringing takes place near band-edge of the filter.

3. This oscillation or ringing is generated because of side lobes in the frequency response W(ω) of the window function. This oscillatory behavior is called "Gibbs Phenomenon".

Reading from the graph, or evaluating the formula, we get:

**{h′[n]} = { ..., -0.055, -0.07, 0, 0.14, 0.28, <u>0.33</u>, 0.28, 0.14, 0, -0.07, -0.055, ... }**

A digital filter with this impulse-response would have exactly the ideal frequency-response we applied to the inverse-DTFT i.e. a ‘brick-wall’ low-pass gain response & phase = 0 for all Ω. But {h′[n]} has non-zero samples extending from n = -∞ to ∞, It is not a finite impulse-response. It is also not causal since h′[n] is not zero for all n < 0. It is therefore not realizable in practice.

**To produce a realizable impulse-response of even order M:**

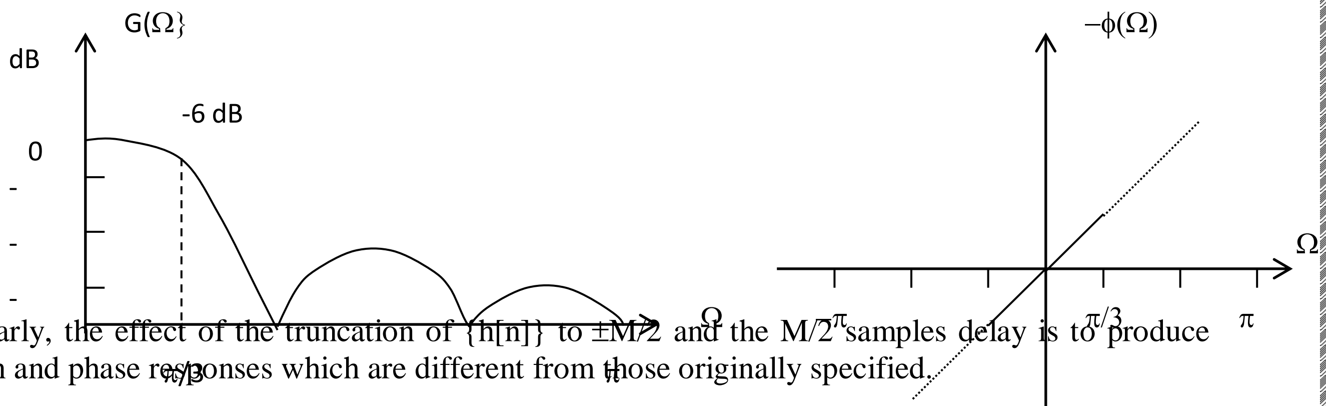
(1) Set $h[n] = \begin{cases} h'[n] & : \frac{-M}{2} \le n \le \frac{M}{2} \\ 0 & : \text{otherwise} \end{cases}$

(2) Delay resulting sequence by M/2 samples to ensure that the first non-zero sample occurs at n = 0.

The resulting causal impulse response may be realised by setting $a_n$ = h[n] for n=0,1,2,...,M. Taking M=4, for example, the finite impulse response obtained for the π/3 cut-off low-pass specification is : {..,0,..,0, <u>0.14</u>, 0.28, 0.33 , 0.28 , 0.14 , 0 ,..,0,..}

The resulting FIR filter is as shown in Figure 4.1 with $a_0$=0.14, $a_1$=0.28, $a_2$=0.33, $a_3$=0.28, $a_4$=0.14. ( Note: a 4th order FIR filter has 4 delays & 5 multiplier coefficients ).

The gain & phase responses of this FIR filter are sketched below.




Clearly, the effect of the truncation of {h[n]} to ±M/2 and the M/2 samples delay is to produce gain and phase responses which are different from those originally specified.

Considering the gain-response first, the cut-off rate is by no means sharp, and two ‘ripples’ appear in the stop-band, the peak of the first one being at about -21dB.

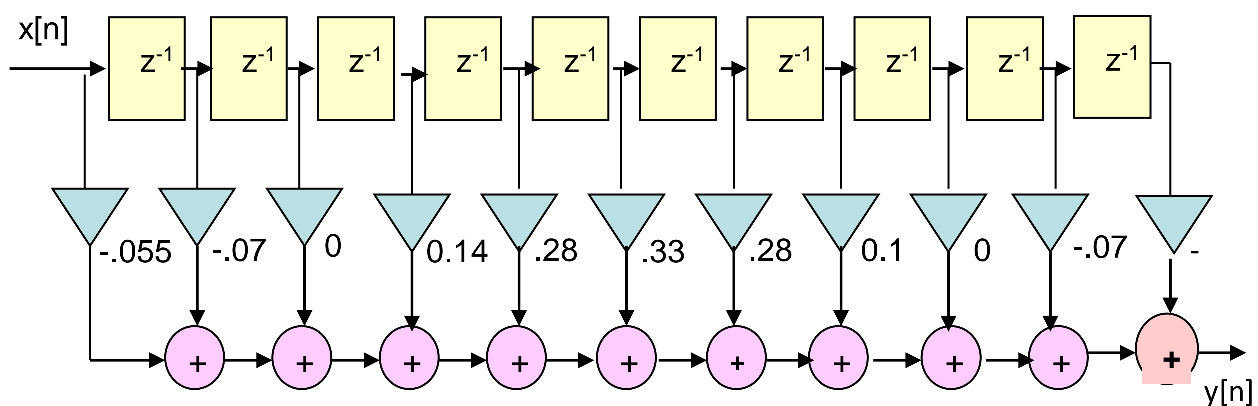
The phase-response is not zero for all values of Ω as was originally specified, but is linear phase ( i.e. a straight line graph through the origin ) in the pass-band of the low-pass filter ( $-\pi/3$ to $\pi/3$ ) with slope arctan( M/2 ) with M = 4 in this case. This means that $\phi(\Omega) = -(M/2)\Omega$ for $|\Omega| \le \pi/3$; i.e. we get a linear phase-response ( for $|\Omega| \le \pi/3$ ) with a phase-delay of M/2 samples.

It may be shown that the phase-response is linear phase because the truncation was done symmetrically about n=0.

Now let's try to improve the low-pass filter by increasing the order to ten. Taking 11 terms of { (1 / 3) sinc (n / 3) } we get, after delaying by 5 samples:

{...0,$\underline{-0.055}$,-.069, 0,.138,.276,.333,.276,.138,0,-.069,-.055,0,...}.

The signal-flow graph of the resulting $10^{th}$ order FIR filter is shown below:




Notice that the coefficients are again symmetric about the centre one (of value 0.33) and this again ensures that the FIR filter is linear phase.

( [-0.055, -0.069, 0, 0.138, 0.276, 0.333, 0.276, 0.138, 0, -0.069, -0.055] );

In may be seen in the gain-response, as reproduced below, that the cut-off rate for the $10^{th}$ order FIR filter is sharper than for the 4th order case, there are more stop-band ripples and, rather disappointingly, the gain at the peak of the first ripple after the cut-off remains at about -21 dB. This effect is due to a well known property of Fourier series approximations, known as Gibb's phenomenon. The phase-response is linear phase in the pass band ( $-\pi/3$ to $\pi/3$ ) with a phase delay of 5 samples. As seen in fig 4.6, going to 20th order produces even faster cut-off rates and more stop-band ripples, but the main stop-band ripple remains at about -21dB. This trend continues with $40^{th}$ and higher orders as may be easily verified. To improve matters we need to discuss '***windowing'.***

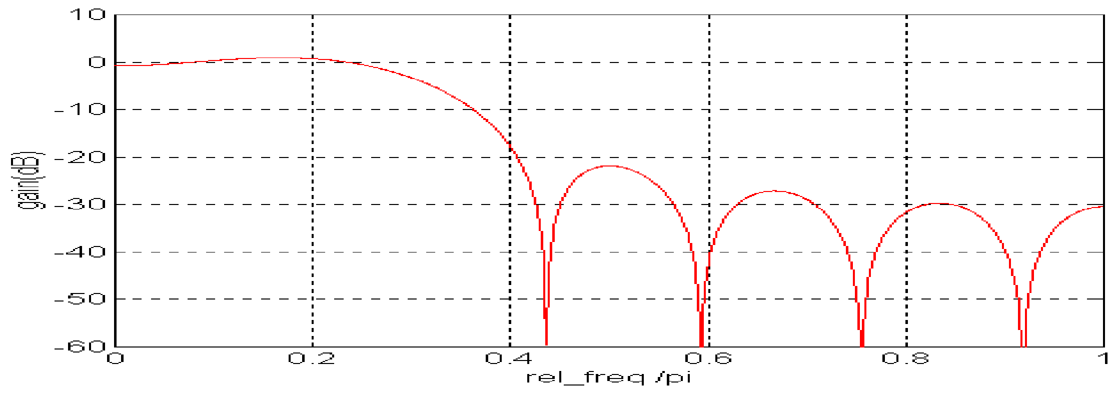



Fig 4.5: Gain response of tenth order low pass FIR filter with $\Omega_C = \pi/3$

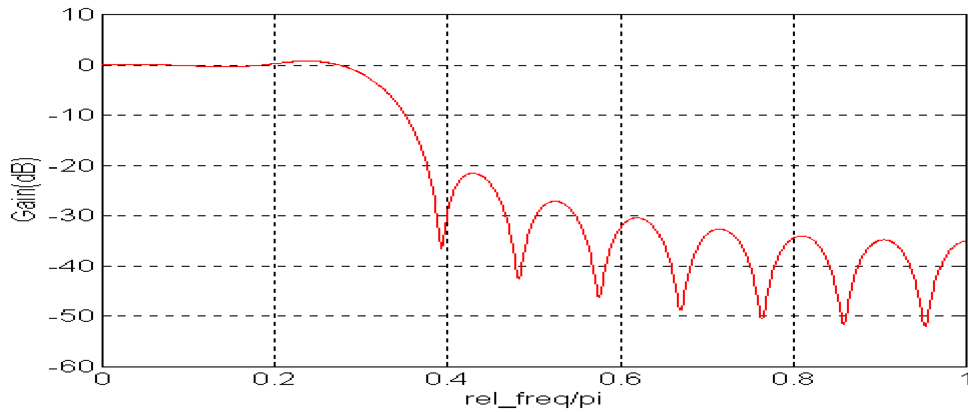


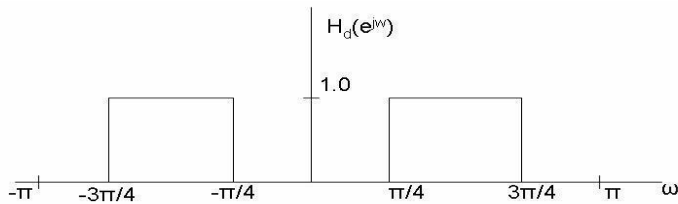

Fig 4.6: Gain response of 20th order low pass FIR filter with $\Omega_C = \pi/3$.

**<u>Exercise Problems</u>**

**Problem 1 : Design an ideal band pass filter with a frequency response:**

$$H_d(e^{j\Omega}) = \begin{cases} 1 & for \quad \frac{\pi}{4} \le |\Omega| \le \frac{3\pi}{4} \\ 0 & otherwise \end{cases}$$

Find the values of h(n) for M = 11 and plot the frequency response.




$$\textit{truncating to } 11 \textit{ samples we have } h(n) = h_d(n)$$

$$\textit{for } |n| \le 5 = 0 \textit{ otherwise}$$

For n = 0 the value of h(n) is separately evaluated from the basic Integration, h(0) = 0.5

Other values of h(n) are evaluated from h(n) Expression

h(1)=h(-1)=0

h(2)=h(-2)=-0.3183

h(3)=h(-3)=0

h(4)=h(-4)=0

h(5)=h(-5)=0

The transfer function of the filter is

$$H(z) = h(0) + \sum_{n=1}^{(N-1)/2} [h(n)\{z^{n} + z^{-n}\}]$$

$$= 0.5 - 0.3183(z^{2} + z^{-2})$$

*the transfer function of the realizable filter is*

$$H'(z) = z^{-5}[0.5 + 0.3183(z^{2} + z^{-2})]$$

$$= 0.3183 z^{-3} + 0.5 z^{-5} + 0.3183 z^{-7}$$

*the filter coeffients are*

$$h'(0) = h'(10) = h'(1) = h'(9) = h'(2) =$$
$$h'(8) = h'(4) = h'(6) = 0$$

$$h'(3) = h'(7) = 0.3183$$

$$h'(5) = 0.5$$

The magnitude response can be expressed as

$$|H(e^{j\omega})| = \sum_{n=1}^{(N-1)/2} a(n)\cos\omega n$$

*comparing this* exp *with*

$$|H(e^{j\omega})| = |z^{-5}[h(0) + 2\sum_{n=1}^{5} h(n)\cos\omega n]|$$

We have a(0)=h(0)

a(1)=2h(1)=0

$a(2)=2h(2)=-0.6366$

$a(3)=2h(3)=0$

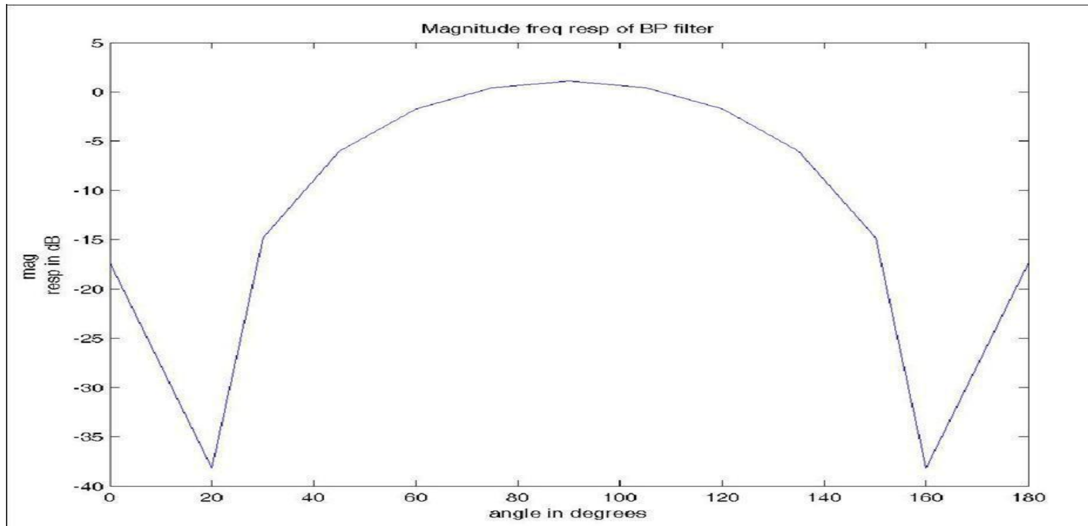
$a(4)=2h(4)=0$

$a(5)=2h(5)=0$

The magnitude response function is

$|H(e^{j\omega})| = 0.5 - 0.6366 \cos 2\omega$ which can plotted for various values of $\omega$ in degrees =[0 20 30 45 60 75 90 105 120 135 150 160 180];

$|H(e^{j\omega})|$ in dBs= [-17.3 -38.17 -14.8 -6.02 -1.74 0.4346 1.11 0.4346 -1.74 -6.02 -14.8 -38.17 -17.3];




**Problem 2: Design an ideal low pass filter with a freq response**

$$H_d(e^{j\omega}) = 1 \quad for \; -\frac{\pi}{2} \le \omega \le \frac{\pi}{2}$$

$$= 0 \quad for \; \frac{\pi}{2} \le |\omega| \le \pi$$

Find the values of h(n) for N =11. Find H(z). Plot the magnitude response
From the freq response we can determine hd(n),

□$n$

$$h_d(n) = \frac{1}{2\pi}\int_{-\pi/2}^{\pi/2} e^{j\omega n}\, d\omega = \frac{\sin \frac{\pi n}{2}}{\pi n}$$

Truncating hd(n) to 11 samples

h(0) = 1/2 h(1)=h(-1)=0.3183

h(2)=h(-2)=0

h(3)=h(-3)=0.106

h(4)=h(-4)=0

h(5)=h(-5)=0.06366

The realizable filter can be obtained by shifting h(n) by 5 samples to right h‟(n)=h(n-5)

h‟(n)= [0.06366, 0, -0.106, 0, 0.3183, 0.5, 0.3183, 0, -0.106, 0, 0.06366];

$$H'(z) = 0.06366 - 0.106z^{-2} + 0.3183z^{-4} + 0.5z^{-5} + 0.3183z^{-6} - 0.106z^{-8} + 0.06366z^{-10}$$

## <u>4.5 Windowing Technique:</u>

**FIR filter design using window functions**

Windowing is the quickest method for designing an FIR filter. A windowing function simply truncates the ideal impulse response to obtain a causal FIR approximation that is non- causal an infinitely long. Smoother window functions provide higher out-of band rejection in the filter response. However this smoothness comes at the cost of wider stop band transitions. Various

windowing method attempts to minimize the width of the main lobe (peak) of the frequency response. In addition, it attempts to minimize the side lobes (ripple) of the frequency response.

*The FIR filter design process via window functions can be split into several steps:*

- Defining filter specifications;
- Specifying a window function according to the filter specifications;
- Computing the filter order required for a given set of specifications;
- Computing the window function coefficients;
- Computing the ideal filter coefficients according to the filter order;
- Computing FIR filter coefficients according to the obtained window function and ideal filter coefficients;

If the resulting filter has too wide or too narrow transition region, it is necessary to change the filter order by increasing or decreasing it according to needs, and after that steps 4, 5 and 6 are iterated as many times as needed.

The final objective of defining filter specifications is to find the desired normalized frequencies (ωc, ωc1, ωc2), transition width and stop band attenuation. The window function and filter order are both specified according to these parameters. Accordingly, the selected window function must satisfy the given specifications. This point will be discussed in more detail in the next chapter .After this step, that is, when the window function is known, we can compute the filter order required for a given set of specifications. One of the techniques for computing is provided in chapter 2.3.When both the window function and filter order are known, it is possible to calculate the window function coefficients w[n] using the formula for the specified window function. This issue is also covered in the next chapter.After estimating the window function coefficients, it is necessary to find the ideal filter frequency samples. The expressions used for computing these samples are discussed in section 2.2.3 under Ideal filter approximation. The final objective of this step is to obtain the coefficients hd[n]. Two sequencies w[n] and hd[n] have the same number of elements. The next step is to compute the frequency response of designed filter h[n] using the following expression:

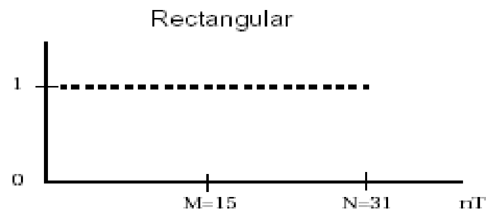
$$h[n] = w[n] \cdot h_d[n]$$

Lastly, the transfer function of designed filter will be found by transforming impulse response via Fourier transform:

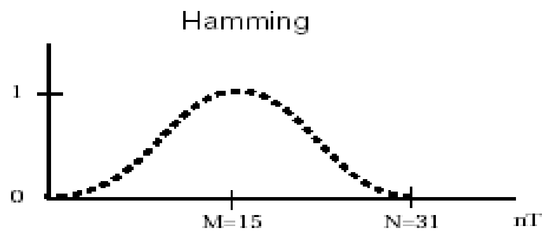
$$H(e^{j\omega}) = \sum_{n=0}^{N} h[n] \cdot e^{-jn\omega}$$

or via Z-transform $H(z) = \sum_{n=0}^{N} h[n] z^{-n}$ If the transition region of designed filter is wider than needed, it is necessary to increase the filter order, reestimate the window function coefficients and ideal filter frequency samples, multiply them in order to obtain the frequency response of designed filter and re estimate the transfer function as well. If the transition region is narrower than needed, the filter order can be decreased for the purpose of optimizing hardware and/or software resources. It is also necessary to re estimate the filter frequency coefficients after that. For the sake of precise estimates, the filter order should be decreased or increased by 1.

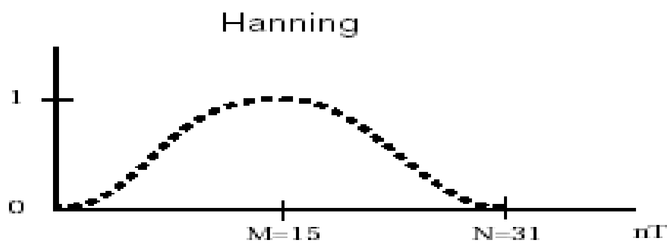
**Rectangular Window:** Rectangular This is the most basic of windowing methods. It does not require any operations because its values are either 1 or 0. It creates an abrupt discontinuity that results in sharp roll-offs but large ripples.




**Hamming Window**:This windowing method generates a moderately sharp central peak. Its ability to generate a maximally flat response makes it convenient for speech processing filtering.




**Hanning Window**: This windowing method generates a maximum flat filter design.




**Kaiser Window**: This windowing method is designed to generate a sharp central peak. It has reduced side lobes and transition band is also narrow. Thus commonly used in FIR filter design

- **Rectangular:**

$$w[n] = \begin{cases} 1 & 0 \le n \le N \\ 0 & \text{otherwise} \end{cases}$$

- **Bartlett (triangular):**

$$w[n] = \begin{cases} 2n/N & 0 \le n \le N/2 \\ 2 - 2n/N & N/2 < n \le N \\ 0 & \text{otherwise} \end{cases}$$

- **Hanning:**

$$w[n] = \begin{cases} 0.5 - 0.5\cos(2\pi n/N) & 0 \le n \le N \\ 0 & \text{otherwise} \end{cases}$$

- **Hamming:**

$$w[n] = \begin{cases} 0.54 - 0.46\cos(2\pi n/N) & 0 \le n \le N \\ 0 & \text{otherwise} \end{cases}$$

- **Kaiser:**

$$w[n] = \begin{cases} I_0[\beta(1 - [(n-\alpha)/\alpha]^2)^{1/2}] & 0 \le n \le N \\ 0 & \text{otherwise} \end{cases}$$

| Name of window function $w(n)$ | Mathematical definition |
|---|---|
| Rectangular | 1 |
| Hanning | $0.5 - 0.5\cos\left[\frac{2\pi n}{N-1}\right]$ |
| Hamming | $0.54 - 0.46\cos\left[\frac{2\pi n}{N-1}\right]$ |
| Blackman | $0.42 - 0.5\cos\left[\frac{2\pi n}{N-1}\right] + 0.08\cos\left[\frac{2\pi n}{N-1}\right]$ |

| Type of window | Approx. Transition width of the main lobe | Peak Side lobe (dB) |
|---|---|---|
| Rectangular | $4\pi/M$ | -13 |
| Bartlett | $8\pi/M$ | -27 |
| Hanning | $8\pi/M$ | -32 |

| | | |
|---|---|---|
| Hamming | $8\pi/M$ | -43 |
| Blackman | $12\pi/M$ | -58 |

Looking at the above table we observe filters which are mathematically simple do not offer best characteristics. Among the window functions discussed Kaiser is the most complex one in terms of functional description whereas it is the one which offers maximum flexibility in the design.

**Procedure for designing linear-phase FIR filters using windows:**

1. Obtain hd(n) from the desired freq response using inverse FT relation
2. Truncate the infinite length of the impulse response to finite length with

( assuming M odd) choosing proper window

$h(n) = h_d(n)w(n)$ *where*

$w(n)$ *is the window function defined for* $-(M-1)/2 \le n \le (M-1)/2$

3. Introduce h(n) = h(-n) for linear phase characteristics

4. Write the expression for H(z); this is non-causal realization

5. To obtain causal realization $H''(z) = z^{-(M-1)/2} H(z)$

### 4.6. Summary of ‘windowing’ design technique for FIR filters

To design an FIR digital filter of even order M, with gain response $G'(\Omega)$ and linear phase, by the windowing method,

1) Set $H'(e^{j\Omega}) = G'(\Omega)$ the required gain-response. This assumes $\phi'(\Omega) = 0$.
2) Inverse DTFT to produce the ideal impulse-response $\{h'[n]\}$.
3) Window to $\pm M/2$ using chosen window.
4) Delay windowed impulse-response by M/2 samples.
5) Realize by setting multipliers of FIR filter.

Instead of obtaining $H'(e^{j\Omega}) = G'(\Omega)$, we get $e^{-j\Omega M/2} G(\Omega)$ with $G(\Omega)$ a distorted version of $G'(\Omega)$ the distortion being due to windowing.

The phase-response is therefore $\phi(\Omega) = -\Omega M/2$ which is a linear phase-response with phase-delay M/2 samples at all frequencies $\Omega$ in the range 0 to $\pi$. This is because $-\phi(\Omega) / \Omega = M/2$ for all $\Omega$.

Notice that the filter coefficients, and hence the impulse-response of each of the digital filters we have designed so far are symmetric in that h[n] = h[M-n] for all n in the range 0 to M where M is the order. If M is even, this means that h[M/2 - n] = h[M/2 + n] for all n in the range 0 to M/2. The impulse response is then said to be 'symmetric' about sample M/2. The following example illustrates this for an example where M=6 and there are seven non-zero samples within {h[n]}: {… 0, …, 0, $\underline{2}$, -3, 5, 7, 5, -3, 2, 0, …,0, … }

The most usual case is where M is even, but, for completeness, we should briefly consider the case where M is odd. In this case, we can still say that {h[n]} is 'symmetric about M/2' even though sample M/2 does not exist. The following example illustrates the point for an example where M=5 and {h[n]} therefore has six non-zero sample:

(…, 0,…, 0, 1, 3, 5, 5, 3, 1, 0, …, 0, …}

When M is odd, h[(M-1)/2 - n] = h[(M+1)/2 + n] for n = 0, 1, …, (M-1)/2.

It may be shown that FIR digital filters whose impulse-responses are symmetric in this way are linear phase. We can easily illustrate this for either of the two examples just given. Take the second. Its frequency-response is the DTFT of {h[n]} i.e.

—

It is also possible to design FIR filters which are not linear phase. The technique described in this section is known as the 'windowing' technique or the 'Fourier series approximation technique'.