SKDAV GOVT.POLYTECHNIC ROURKELA

DEPARTMENT OF ELECTRONICS AND TELECOMMUNICATION ENGINEERING

LECTURE NOTES

Year & Semester: 3RD Year, VI Semester

Subject Code/Name: ETT-602, MICROCONTROLLER, EMBEDDED SYSTEM & PLCS

CONTENTS

5.Peripherals

5.1 Explain Watchdog Timers, LCD Controllers,
5.2 Explain Analog-to-Digital converters
5.3 Explain Real- Time Clocks
5.4 Explain DS 12887 RTC chip & its interfacing
5.5 Motor Control: Relay and optoisolator, Stepper motor interfacing, DC MOTOR Interfacing

6. Programmable Logic Controllers (PLCs)

6.1 PLC Architecture
6.2 Explain the basic operation of PLC
6.3 Describe briefly PLC programming
6.4 Explain address of internal of a PLC
6.5 State the difference between a programmable controller and a computer
6.6 Explain PLC memory organization
6.7 Explain program scan of a PLC
6.8 Explain internal instruction of PLC
6.9 Program a ladder Rung diagram
6.10 Program PLC timer
6.11 Program PLC as a counter
6.12 Understand control instructions of PLC
6.13 Understand Data management instruction of PLC
6.14 Explain how I/O interface handles numerical data in PLC
6.15 Draw the solid state logic control circuit for the following problems and explain Motor control circuit to provide sequence control to Motor 1 and motor 2

PART-1

UNIT-1: INTRODUCTION TO EMBEDDED SYSTEM

1.1 EMBEDDED SYSTEMS OVERVIEW

System:

- A System is a way of working organizing or doing one or many tasks according to a fixed plan, program or set of rules.
- A system is also an arrangement in which all its units assembles & work together according to the plan or program.

Embedded system:

- An embedded system is a system that has embedded software & hardware, which makes it a system dedicated for an application or specific part of an applications or product or a part of a larger system.
- **Embedded system=programming in electronics.**

Shortlist of Embedded System:

Embedded system is found in a variety of common electronic devices such as:-

1. **Consumer Electronics-**

- Cell phones, pagers, digital cameras, video cassettes recorders, portable video games, calculators & personal digital assistants.

2. **Home Appliances:** -

- Microwave ovens, answering machines, thermostat, home security, washing machine, & lighting system.

3. **Office Automation:**-

- Printers, fax machine, photocopying machines, scanners, copier, biometric, surveillance cameras etc.

4. **Business: -**

- Cash register, alarm system, card reader, product scanner, curb side check in automated teller machine (ATM)

5. **Automobiles:-**

- Transmission control , cruise control, fuel injection, anti-lock brakes & reactive suspension
- Air bags, anti-lock braking system (ABs), engine control, door lock, GPS system, vehicular ad-hoc network (VANET)

6. Communication:-

- Mobile phone, network switches, WiFi, hotspots, telephone, MODEM etc.

Characteristics of Embedded System:-

Embedded system have several common characteristics

1. Single functional: -

- An embedded system usually executes only one program, repeatedly.

For Example: - a pager is always a pager

- It performs specialized operation & does the same job repeatedly.
- The Embedded system should perform the single given task throughout the life. This feature makes it dedicated & performs the accurately on time. A user of the system can't change the feature or functionality.

2. Tightly Constraint:-

- The circuit size should be small enough to fit on a single chip and must perform fast enough to process data in real time & consume minimum power to extend battery-life.

3. Reactive& Real time:-

- It should continuously react to the changes in the system environment and must compute certain results in real time without any delay
- A reactive system is reacting on a given input. Like in an over when we press the button to start cooking, it takes input & start acting. This type of system is the reactive system
- But what happens it the over responds in a random interval like some time it start in 30 sec & sometime in 5 minutes. The point is there should be some time limit to start action. This called **deadline**

Example of Embedded system:

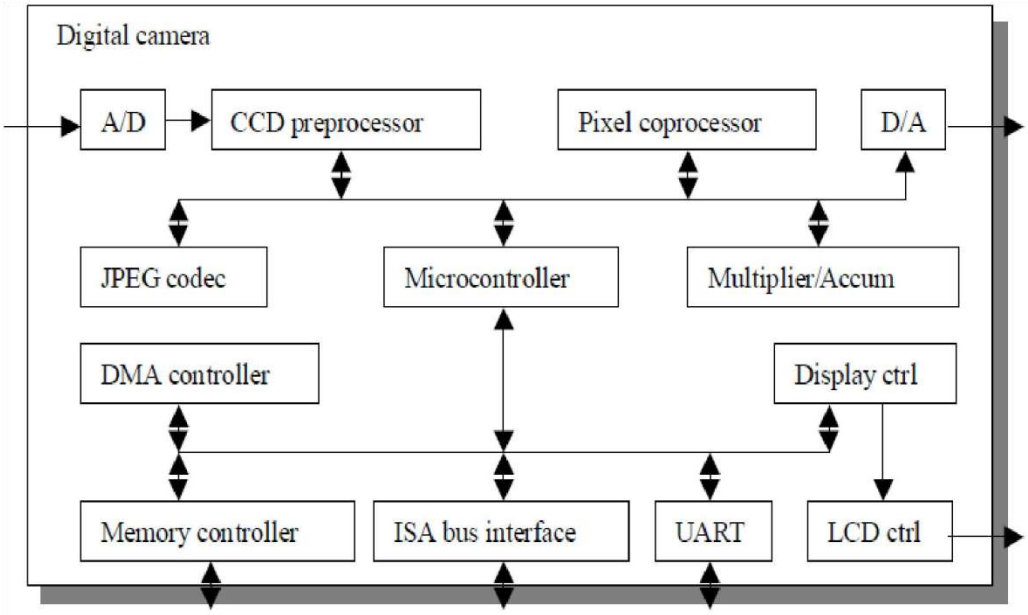
A DIGITAL CAMERA:-

- The A2D and D2A circuits convert analog images to digital and digital to analog respectively.
- The CCD processor is a charge-coupled device pre-processor

- The JPEG codec compresses and decompresses an image using the JPEG2 compression standard, enabling compact storage in the limited memory of the camera.
- The pixel coprocessor aids in rapidly displaying images.
- The memory controller controls access to a memory chip also found in the camera, while the DMA controller enables direct memory access without requiring the use of the microcontroller.
- The UART enables communication with a PC's serial port for uploading video frames, while the ISA bus interface enables a faster connection with a PC's ISA bus.
- The LCD ctrl and Display ctrl circuit control the display of images on the camera's liquid-crystal display device.
- A multiplier/Accumulator circuit assists with certain digital signal processing.
- The heart of the system is a **microcontroller,** which is a processor that controls the activities of all the other circuit. Each device as a processor designed for a particular task, while the microcontroller is a more general processor designed for general tasks.

➢ **This example illustrates some of the embedded system characteristics described above.**

- It performs a single function repeatedly. This system always acts as a digital camera, where in it captures. Compresses and stores frames, decompresses and display frames, and uploads frames.
- It is tightly constrained. The system must be low cost since consumer must be able to afford such a camera it must be small so that it fits within a standard- sized camera. It must be fast so that it can process numerous images in millisecond. It must consume little power so that the camera's battery will last a long time.
- This particular system does not possess a high degree of the characteristics of being reactive and real time, as it only needs to responds to the pressing of buttons by a user, Which even for an avid photographer is still quite slow with respect to processor speeds.



1.2 EMBEDDED SYSTEM TECHNOLOGIES:-

- Technology as a manner of accomplishing a task, especially using technical process, methods or knowledge.
- There are three technologies in the embedded system design:
    1. Processor technologies
    2. IC technologies
    3. Design technologies

1.3 PROCESSOR TECHNOLOGYY:-

- Processor technology involves the architecture of the computation engine used to implement a system's desired functionality.
- The term processor often linked to programmable software processors may also be used in association with several other non-programmable, digital systems.
- These processors vary in their specialization towards a particular application (for e.g. Mobile phones, radio, computer) and by doing so exhibit a variety of design metric.

- An embedded processor can be classified into 3 different forms on the basis of functionality.

→ General purpose processor
→ Special purpose processor
→ Application specific processor

GENERAL PURPOSE PROCESSORS--SOFTWARE:

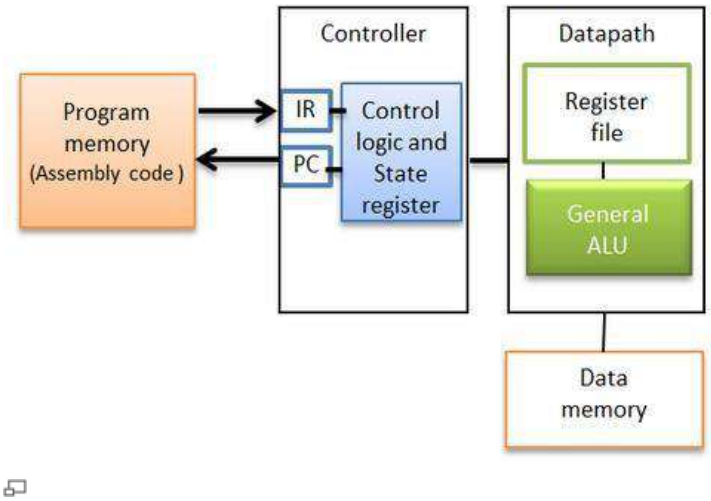
- The general –purpose processor is designed to serve several applications with the idea of being able to sell the maximum number of devices.
- One feature of such a processor is a program memory-the designer does not know what program will run on the processor, so the program cannot be inserted with a digital circuit.
- Another feature is a general data path this should have the capacity to handle a variety of computations-thus it will have large register file with additional general purpose arithmetic logic unit (ALU).
- An embedded system simply uses a general-purpose processor, by programming the processor's memory to carry out the required functionality.

- **Using a general-purpose processor in an embedded system may result in several design-merit advantages**.

1. **Design time** and **NRE (non-recurring engineering)→**
- One time cost to design, develop, manufacture.
- Cost is low, because the designer has to only write a program and any digital design is not necessary.
2. **Flexibility→**is high, because changing functionality requires only changing the program.
3. **Units cost** may be relatively low in small quantities, since the processor manufacture sells large quantities to customers.
4. **Performance** may be fast for computation-intensive applications, if using a fast processor, due to advanced architecture features and leading edge IC technology.

- **There are also some design-merit drawbacks.**
    1. **Unit cost→**may be too high for large quantities.
    2. **Performance→**may be slow for certain applications.

3. **Size** and **power→** may be large due to unnecessary processor hardware.

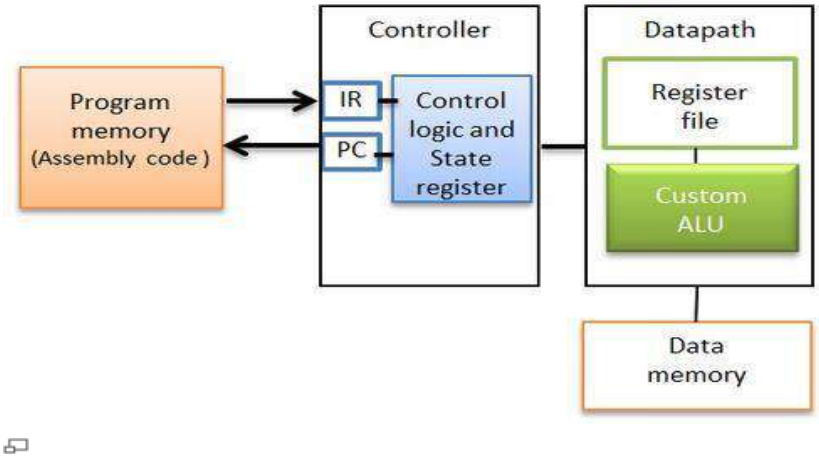


- Diagram shows a simple architecture of a general-purpose processor implementing the array summing functionality.
- The functionality is stored in a program memory.
- The controller fetches the current instructions, as indicated by the program counter (PC) into the instruction register (IR).
- It then configures the data path for these instructions and executes the instruction.
- Finally, it determines the appropriate next instruction address, sets the PC to this address, and fetches again.

SINGLE PURPOSE PROCESSOR--HARDWARE:

- A single-purpose processor is a digital circuit designed to execute exactly one program.
- **For example: -**consider the digital camera. All of the components other than the microcontroller are single-purpose processor. The JPEG codec, for example executes a single program that compresses and decompresses video frames.
- An embedded system creates a single-purpose processor by designing a custom digital circuit.

- Using a single-purpose processor in an embedded system result in several design metric benefits and drawbacks, which are essentially the inverse of those for general purpose processors.
- Performance may be fast, size and power may be small, and unit-cost may be low for large quantities, while design time and NRE cost may be high, flexibility is low,unit cost may be high for small quantities and performance may not match general purpose processor for some applications.



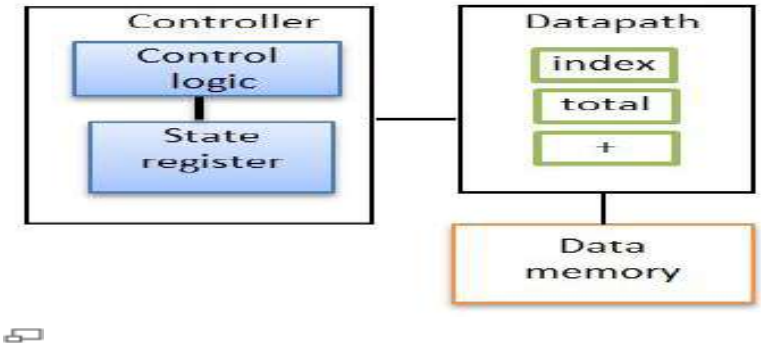
The architecture of such a single-purpose processor

- For example, since the example counts from 1 to N, we add an index register.
- The index register will be loaded with N, and will then count down to zero ('0'), at which time it will assert a status line read by the controller.
- The example has only one other value, we add only one register labeled total to the data path.
- Since the example's only arithmetic operation is addition, we add a single adder to the data path.
- Since the processor only executes this one program, we hardware the program directly into the control logic.

1.4 APPLICATION SPECIFIC PROCESSORS:-

- An application-specific instruction-set processor (or ASIP) can serve as a compromise between the other processor.

- An ASIP is designed for a particular class of applications with common characteristics, such as digital-signal processing, telecommunications, embedded control, etc.
- An ASIP in an embedded system can provide the benefit of flexibility while still achieving good performance, power and size.
- Such processors can require large NRE cost to build the processor itself.



The Block diagram of an application specific processor is given below. You can see there is no program memory in this. It is designed for particular types of input.

MICROCONTROLLERS:-

- Microcontroller is a microprocessor that has been optimized for embedded a control applications.
- Such applications typically monitor and set numerous single bit control signals but do not perform large amount of data computations. Thus microcontrollers tend to have simple data paths that excel bit-level operations and reading and writing external bits.
- Furthermore, they tend to incorporate on the microprocessor chip several peripheral components common in control applications like serial communication peripherals, times, counters, pulse width modulators and analog to digital converters. Such incorporation of peripherals enables single chip implementations and hence smaller and lower cost product.

DIGITAL SIGNAL PROCESSING (DSP):-

- Digital-signal processors (DSPs) are a common class of ASIP.
- A DSP is a processor designed to perform common operations on digital signals, which are the digital encodings of analog signals like video and audio. These

operations carry out common signal processing tasks like signal filtering, transformation, or combination.

- Such operations are usually math-intensive, including operations like multiply and add or shift and add.
- To support such operations, a DSP may have special purpose datapath components such a multiply-accumulate unit, which can perform a computation like T = T + M[i]*k using only one instruction.
- The use of an ASIP, while partially customized to the desired functionality, there is some inefficiency since the processor also contains features to support reprogramming.
- The data path may be customized. It may have an auto-incrementing register, a path that allows the add of a register plus a memory location in one instruction, fewer registers, and a simpler controller.

1.5 IC TECHNOLOGY:-

- Every processor must eventually be implemented on an IC.
- An IC (Integrated Circuit), often called a "chip," is a semiconductor device consisting of a set of connected transistors and other devices.
- A number of different processes exist to build semiconductors, the most popular of which is CMOS (Complementary Metal Oxide Semiconductor).
- Semiconductors consist of numerous layers as shown in the figure given below.

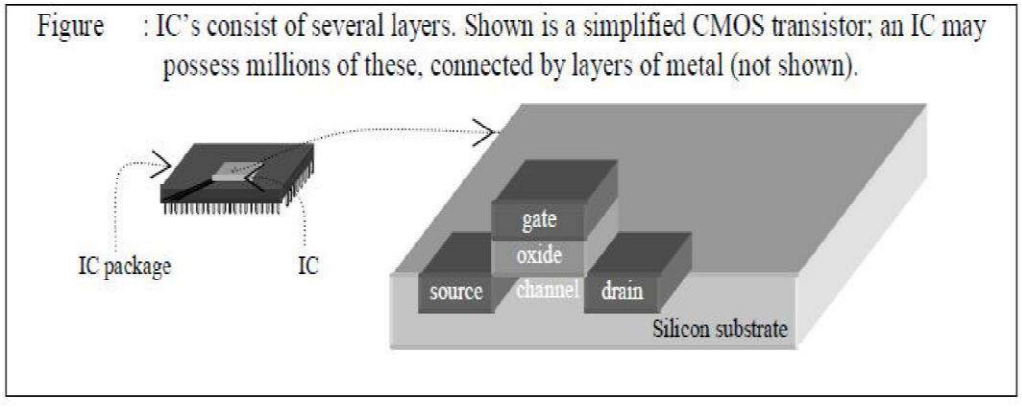


Figure : IC's consist of several layers. Shown is a simplified CMOS transistor; an IC may possess millions of these, connected by layers of metal (not shown).

- The bottom layers form the transistors. The middle layers form logic gates. The top layers connect these gates with wires. These layers can be created by depositing photo-sensitive chemicals on the chip surface and then shining light through masks to change regions of the chemicals. A set of masks is often called a layout. The narrowest line that we can create on a chip is called the feature size.

Full Custom / VLSI:-

- In a full-custom IC technology, we optimize all layers for our particular embedded system's digital implementation.
- Such optimization includes placing the transistors to minimize interconnection lengths, sizing the transistors to optimize signal transmissions and routing wires among the transistors.
- Once all the masks are completed, then we send the mask specifications to a fabrication plant that builds the actual ICs.
- Full-custom IC design, often referred to as VLSI (Very Large Scale Integration) design, has very high NRE cost and long turnaround times (typically months) before the IC becomes available, but can yield excellent performance with small size and power.
- It is usually used only in high-volume or extremely performance-critical applications.

SEMICUSTOM ASIC (GATE ARRAY AND STANDARD CELL):

- In an ASIC (Application-Specific IC) technology, the lower layers are fully or partially built, leaving us to finish the upper layers.
- In a gate array technology, the masks for the transistor and gate levels are already built (i.e., the IC already consists of arrays of gates).
- The remaining task is to connect these gates to achieve our particular implementation.
- In a standard cell technology, logic-level cells (such as an AND gate or an AND-ORINVERT combination) have their mask portions pre-designed, usually by hand.
- Thus, the remaining task is to arrange these portions into complete masks for the gate level, and then to connect the cells.
- ASICs are by far the most popular IC technology, as they provide for good performance and size, with much less NRE cost than full-custom IC's.

PLD:

- In a PLD (Programmable Logic Device) technology, layers implement a programmable circuit, where programming has a lower-level meaning than a software program.

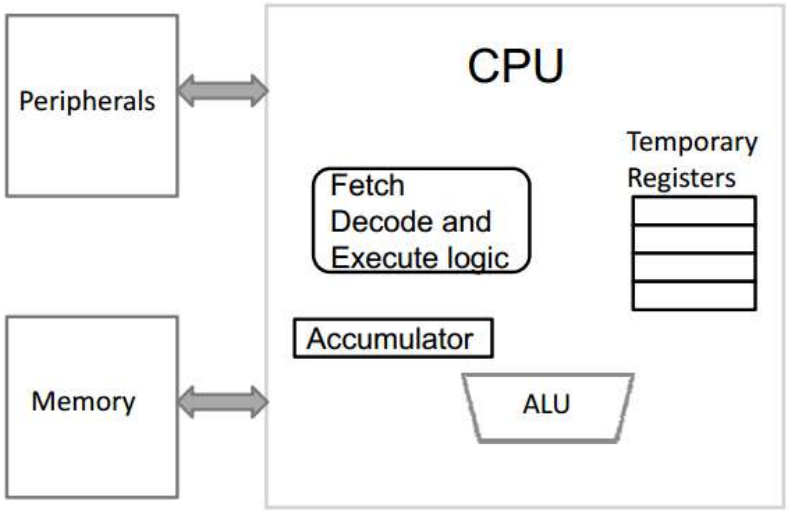
- The programming that takes place may consist of creating or destroying connections between wires that connect gates, either by blowing a fuse, or setting a bit in a programmable switch.
- Small devices, called programmers, connected to a desktop computer can typically perform such programming.
- PLD's of two types, simple and complex. One type of simple PLD is a PLA (Programmable Logic Array), which consists of a programmable array of AND gates and a programmable array of OR gates.
- Another type is a PAL (Programmable Array Logic), which uses just one programmable array to reduce the number of expensive programmable components.
- One type of complex PLD, growing very rapidly in popularity over the past decade, is the FPGA (Field Programmable Gate Array), which offers more general connectivity among blocks of logic, rather than just arrays of logic as with PLAs and PALs, and is thus able to implement far more complex designs. PLDs offer very low NRE cost and almost instant IC availability.
- They are typically bigger than ASICs, may have higher unit cost, may consume more power, and may be slower (especially FPGAs). They still provide reasonable performance, though, so are especially well suited to rapid prototyping.

UNIT-2: MICROCONTROLLER 8051 ARCHITECTURE

1.1 DIFFERENCE BETWEEN MICROCONTROLLER &GENERAL PURPOSE MICROPROCESSOR:

MICROCONTROLLER:

- Microcontroller is like a mini computer with a CPU along with RAM, ROM, serial ports, timers, and IO peripherals all embedded on a single chip.
- It's designed to perform application specific tasks that require a certain degree of control such as a TV remote, LED display panel, smart watches, vehicles, traffic light control, temperature control, etc.
- It's a high-end device with a microprocessor, memory, and input/output ports all on a single chip.
- It's the brains of a computer system which contains enough circuitry to perform specific functions without external memory.
- Since it lacks external components, the power consumption is less which makes it ideal for devices running on batteries.
- Simple speaking, a microcontroller is complete computer system with less external hardware.



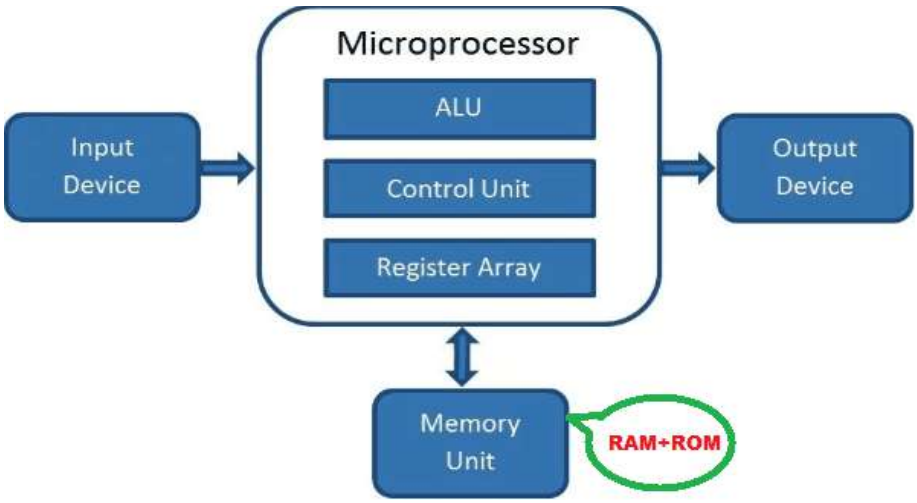
MICROPROCESSOR:

- A Microprocessor is a multipurpose, Programmable clock driven, register based electronic device,
- That read binary instruction from a storage device called memory, accepts binary data as input and processes data according to those instructions and provides results as outputs.
- Microprocessor is clock driven semiconductor device which for is manufactured by

using LSI and VLSI technique.

Or

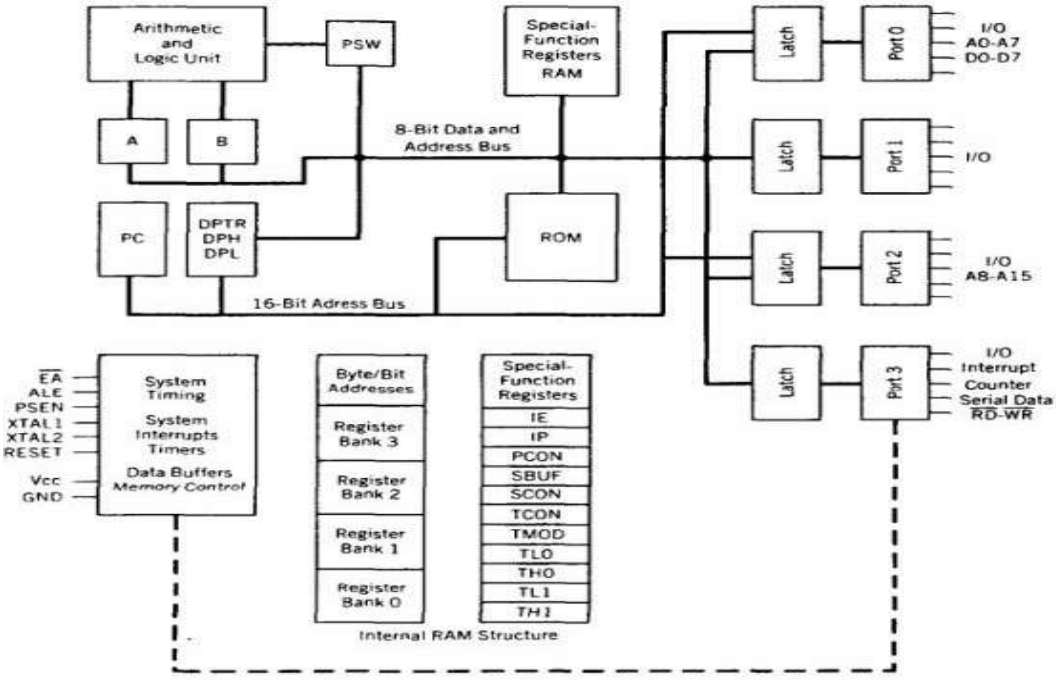
- Microprocessor is a silicon-based integrated chip with only a central processing unit. It's the heart of a computer system which is designed to perform loads of tasks that involve data.
- Microprocessors don't have RAM, ROM, IO pins, Timers, and other peripherals on the chip. They are to be added externally to make them functional.
- It consists of the ALU which handles all the arithmetic and logical operations; the Control Unit which manages and handles the flow of instructions throughout the system; and Register Array which stores the data from memory for fast access.
- They are designed for general purpose applications such as logical operations in computer system. In simple terms, it's a fully-functional CPU on a single integrated circuit that is used by a computer system to do its work.



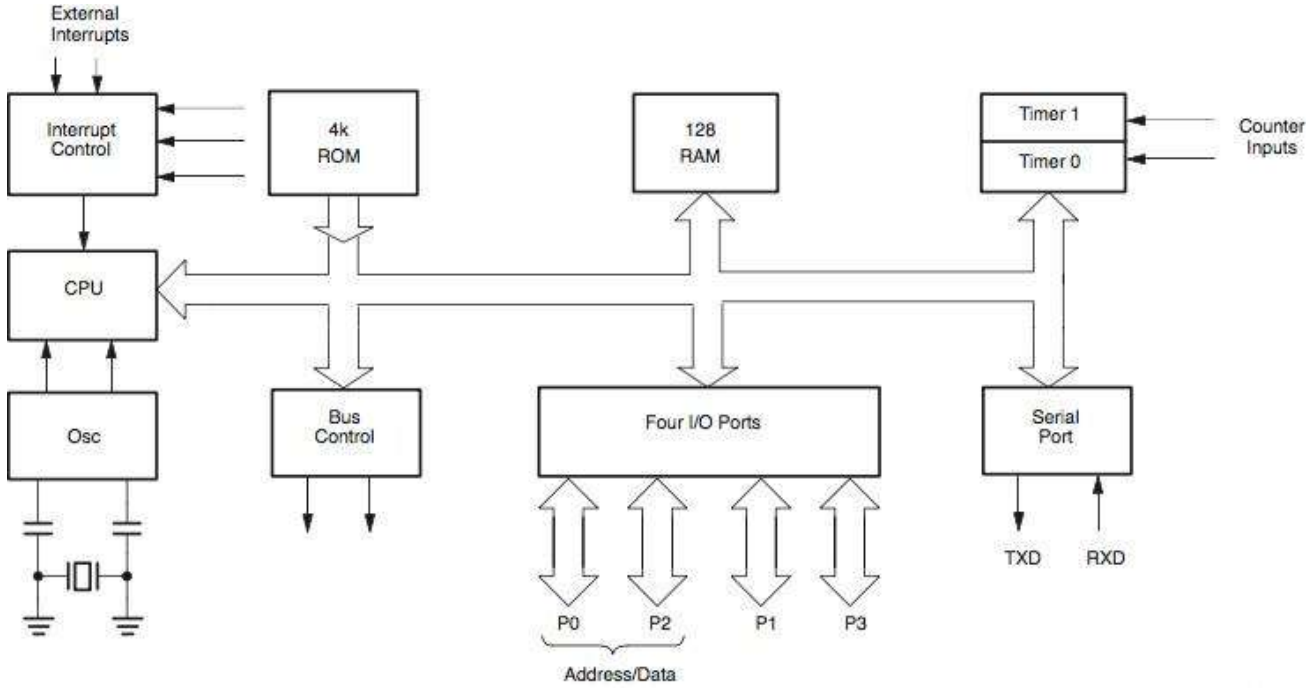
Difference between the microcontroller and microprocessor -

Microprocessor	Microcontroller
Microprocessor contains ALU, General purpose registers, stack pointer, program counter, clock timing circuit, interrupt circuit	Microcontroller contains the circuitry of microprocessor, and in addition it has built in ROM, RAM, I/O Devices, Timers/Counters etc.
It has many instructions to move data between memory and CPU	It has few instructions to move data between memory and CPU
Few bit handling instruction	It has many bit handling instructions
Less number of pins are multifunctional	More number of pins are multifunctional
Single memory map for data and code (program)	Separate memory map for data and code (program)
Access time for memory and IO are more	Less access time for built in memory and IO.
Microprocessor based system requires additional hardware	It requires less additional hardware's
More flexible in the design point of view	Less flexible since the additional circuits which is residing inside the microcontroller is fixed for a particular microcontroller
Large number of instructions with flexible addressing modes	Limited number of instructionswith few addressing modes

2. 2 ARCHITECTURE OF 8051:



ALTERNATE DIAGRAM FOR 8051 ARCHITECTURE /BLOCK DIAGRAM



- ✓ **8051** is a microcontroller. This means it has an internal processor, internal memory and an I/O section. The architecture of 8051 is thus divided into three main sections:

- The CPU
- Internal Memory
- I/O components.

CPU:

- 8051 has an 8 bit CPU.
- This is where all 8-bot arithmetic and logic operations are performed.
- It has the following components.

ALU – ARITHMETIC LOGIC UNIT:

- It performs 8-bit arithmetic and logic operations.
- It can also perform some bit operations.

- **Example:**
  ADD A, R0 ; Adds contents of A register and R0 register and stores the result in A register.
  ANL A, R0 ; Logically ANDs contents of A register and R0 register and stores the result in A register.
  CPL P0.0 ; Complements the value of P0.0 pin.

A – REGISTER (ACCUMULATOR):

- It is an 8-bit register.
- In most arithmetic and logic operations, A register hold the first operand and also gets the result of the operation.
- Moreover, it is the only register to be used for data transfers to and from external memory.

- **Example:**
  ADD A, R1 ; Adds contents of A register and R1 register and stores the result in A register.
  MOVX A, @DPTR ; A gets the data from External RAM location pointed by DPTR

B – REGISTER:

- It is an 8-bit register.
- It is dedicated for Multiplication and Division.
- It can also be used in other operations.

- **Example:**
  MUL AB; Multiplies contents of A and B registers. Stores 16-bit result in B and A registers.

  DIV AB; Divides contents of A by those of B. Stores quotient in A and remainder in B.

PC – PROGRAM COUNTER

- It is a 16-bit register.
- It holds address of the next instruction in program memory (ROM).
- PC gets automatically incremented as soon as any instruction is fetched.
- That's what makes the program move ahead in a sequential manner.
- In the case of a branch, a new address is loaded into PC.

DPTR – DATA POINTER

- It is a 16-bit register.
- It holds address data in data memory (RAM).
- DPTR is divided into two registers DPH (higher byte) and DPL (lower byte).
- It is typically used by the programmer to transfer data from External RAM.
- It can also be used as a pointer to a look up table in ROM, using Indexed addressing mode.
- **Example:**
  MOVX A, @DPTR ; A gets the data from External RAM location pointed by DPTR
  MOVC A, @A+DPTR ; A gets the data from ROM location pointed by A + DPTR

SP – STACK POINTER

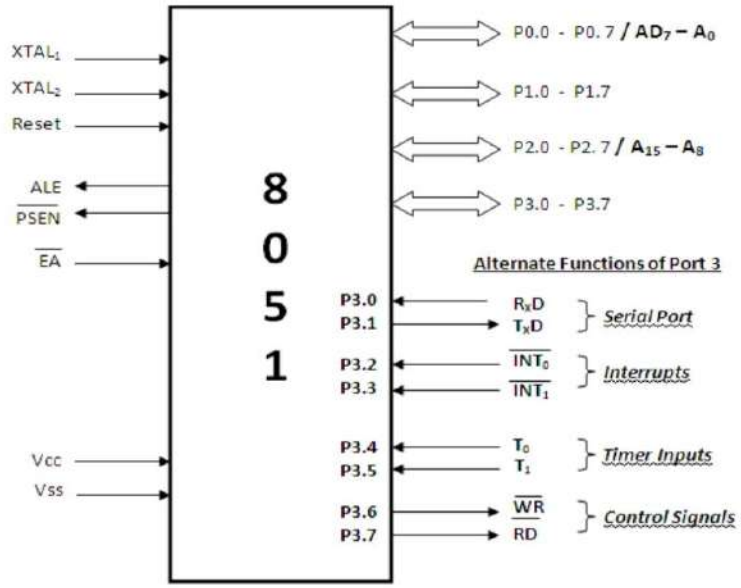
- It is an 8-bit register.
- It contains address of the top of stack. The Stack is present in the Internal RAM.
- Internal RAM has 8-bit addresses from 00H… 7FH. Hence SP is an 8-bit register.
- It is affected during Push and Pop operations. During a Push, SP gets incremented.
- During a Pop, SP gets decremented.

PSW – PROGRAM STATUS WORD

- It is an 8-bit register.
- It is also called the "Flag register", as it mainly contains the status flags. These flags indicate status of the current result.
- They are changed by the ALU after every arithmetic or logic operation. The flags can also be changed by the programmer.
- PSW is a bit addressable register.
- Each bit can be individually set or reset by the programmer.
- The bits can be referred to by their bit numbers (**PSW.4**) or by their name (**RS1**).

- **Example:**
  SETB PSW.3 ; Makes PSW.3← 1
  CLR PSW.4 ; Makes PSW.4←0

2.3 PIN DIAGRAM OF 8051:



- **8051 has 40 pins.**
  The function of these pins is briefly explained as follows.

<u>XTAL1 & XTAL2:</u>

- These are connected to the crystal oscillator.

- The typical operate in frequency is 12 MHz
- In Serial communication based applications, the operating frequency is chosen to be 11.0592 MHz, in order to derive the standard universal baud rates. This will be discussed in detail in the further chapters.

RESET:

- It is used to reset the 8051 microcontroller. On reset PC becomes 0000H.
- This address is called the reset vector address.
- From here, 8051 executes the BIOS program also called the Booting program or the monitor program.
- It is used to set-up the system and make it ready, to be used by the end-user.

ALE:

- It is used to enable the latching of the address. The address and data buses are multiplexed.
- This is done to reduce the number of pins on the 8051 IC.
- Once out of the chip, address and data have to be separated that is called de-multiplexing.
- This is done by a latch, with the help of ALE signal. ALE is "1" when the bus carries address and "0" when the bus carries data.
- This informs the latch, when the bus is carrying address so that the latch captures only address and not the data.

EA'

- It decides whether the first 4 KB of program memory space (0000H… 0FFFH) will be assigned to internal ROM or External ROM.
- If EA = 0, the External ROM begins from 0000H.
- In this case the Internal ROM is discarded. 8051 now uses only External ROM.
- If EA = 1, the External ROM begins from 1000H.
- In this case the Internal ROM is used. It occupies the space 0000H…0FFFH.
- In modern FLASH ROM versions, this pin also acts as VPP (12 Volt programming voltage) to write into the FLASH ROM.

PSEN'

- 8051 has a 16-bit address bus (A_{15}-A_0).

This should allow 8051 to access 64 KB of external Memory as 2^{16} = 64 KB.

Interestingly though, 8051 can access 64 KB of External ROM and 64 KB of External RAM, making a total of 128 KB.

- Both have the same address range 0000H to FFFFH.
- This does not lead to any confusion because there are separate control signals for External RAM and External ROM.
- RD and WR are control signals for External RAM.
- PSEN is the READ signal for External ROM.
- It is called Program Status Enable as it allows reading from ROM also known as Program Memory. Having separate control signals for External RAM and External ROM actually allows us to double the size of the external memory to a total of 128 KB from the original 64 KB.

VCC & GND:

- These are power supply pins.
- 8051 works at +5V / 0V power supply.

P0.0-P0.7

- These are 8 pins of Port 0.
- We can perform a byte operation (8-bit) on the whole port 0.
- We can also access every bit of port 0 individually by performing bit operations like set, clear, complement etc.
- The bits are called P0.0… P0.7.
- Additionally, Port 0 also has an alternate function.
- It carries the multiplexed address data lines.
- A0-A7 (the lower 8 bits of address) and D0-D7 (8 bits of data) are multiplexed into AD0-AD7.
- In any operation address and data are not issued simultaneously. First, address is given, then data is transferred. Using a common bus for both, reduces the number of pins.
- To identify if the bus is carrying address or data, we look at the ALE signal. If ALE = 1, the bus carries address,
- If ALE = 0, the bus carries data.

P1.0-P1.7

- These are 8 pins of Port 1.

- We can perform a byte operation (8-bit) on the whole port 1.
- We can also access every bit of port 1 individually by performing bit operations like set, clear, complement etc. on P1.0… P1.7.
- Port 1 also has NO alternate function

P2.0-P2.7

- These are 8 pins of Port 2.
- We can perform a byte operation (8-bit) on the whole port 2.
- We can also access every bit of port 2 individually by performing bit operations like set, clear, complement etc. on P2.0… P2.7.
- Additionally, Port 2 also has an alternate function. It carries the higher order address lines A8-A15.

P3.0-P3.7

- These are 8 pins of Port 3.
- We can perform a byte operation (8-bit) on the whole port 3. We can also access every bit of port 3 individually.
- The bits are called P0.0… P0.7.
- The various pins of Port 3 have a lot of alternate functions.

P3.0 (RXD) and P3.1 (TXD):

- They are used to receive and transmit serial data.
- This forms the serial port of 8051.

P3.2 (INT0) and P3.3 (INT1):

- They are external hardware interrupts of 8051.
- If they occur simultaneously, INTO is by default higher priority.

P3.4 (T0) and P3.5 (T1):

- They are used **timer clock inputs**.
- They provide external clock inputs to Timer 0 and Timer 1.

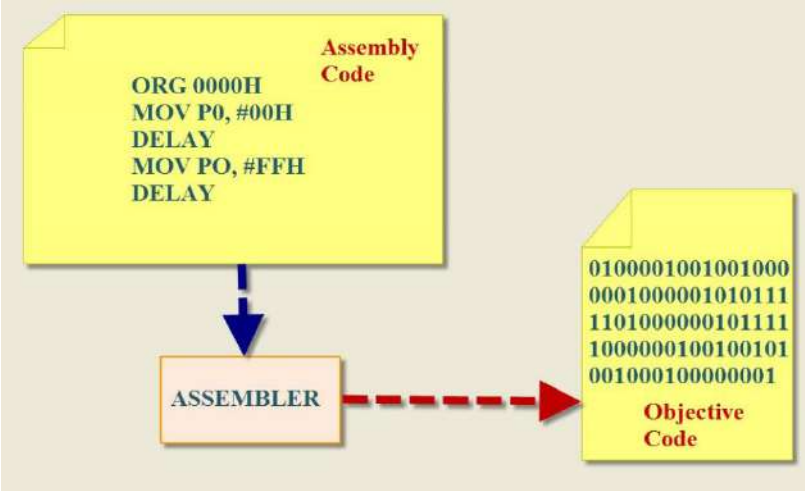
P3.6 (WR) and P3.7 (RD):

- They are used as **control signals for External RAM**.
- 8051 can access 64 KB External RAM from 0000H to FFFFH.

2.4 8051 PROGRAMMING MODEL:

- The assembly language is a low-level programming language used to write program code in terms of mnemonics.

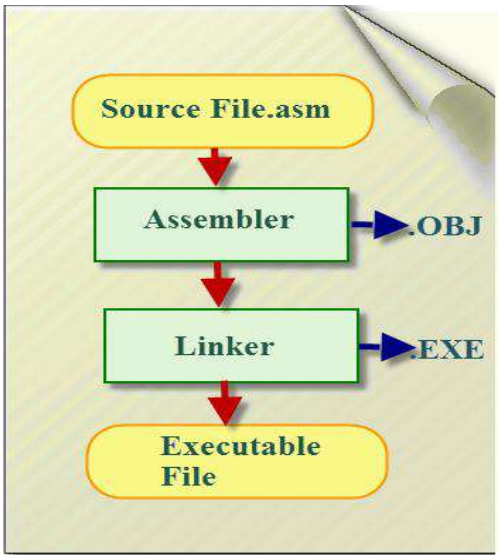
- Even though there are many high-level languages that are currently in demand, assembly programming language is popularly used in many applications.
- It can be used for direct hardware manipulations. It is also used to write the 8051 programming code efficiently with less number of clock cycles by consuming less memory compared to the other high-level languages.



8051 Programming

8051 Programming in Assembly Language

- The assembly language is a fully hardware related programming language.
- The embedded designers must have sufficient knowledge on hardware of particular processor or controllers before writing the program.
- The assembly language is developed by mnemonics; therefore, users cannot understand it easily to modify the program.



8051 Programming in Assembly Language

- Assembly programming language is developed by various compilers and the "keiluvison" is best suitable for microcontroller programming development.
- Microcontrollers or processors can understand only binary language in the form of '0s or 1s'; an assembler converts the assembly language to binary language, and then stores it in the microcontroller memory to perform the specific task.

2.5 EXPLAIN 8051 REGISTER BANKS AND STACK:

MEMORY ORGANIZATION:

- During the runtime, microcontroller uses two different types of memory: one for holding the program being executed **(ROM memory),** and the other for temporary storage of data and auxiliary variables **(RAM memory).**
- Depending on the particular model from 8051 family, this is usually few kilobytes of ROM and 128/256 bytes of RAM.
- This amount is built-in and is sufficient for common tasks performed "independently" by the MCU.
- However, 8051 can address up to 64KB of external memory.

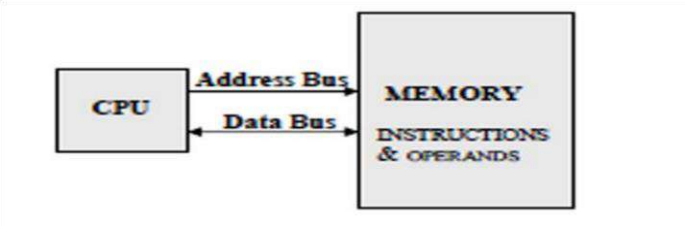
MEMORY ARCHITECTURE:

1. VON-NEUMAN ARCHITECTURE:

- Von Neumann architectures are computer architectures that use the same

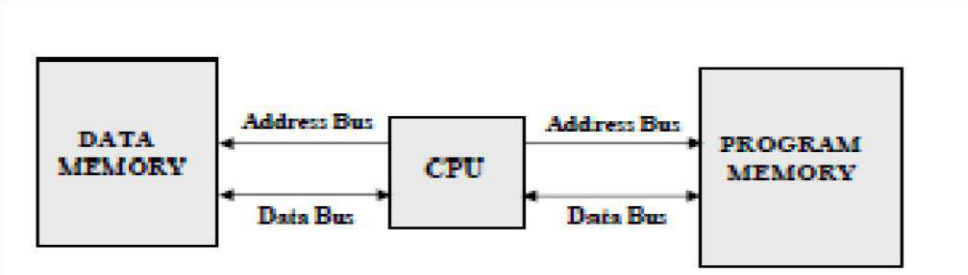
storage device for both instructions and data.

- By treating the instructions in the same way as the data, the machine could easily change the instructions.
- In other words the machine was reprogram able. Because the machine did not distinguish between instructions and data, it allowed a program to modify or replicate a program.



2. HARVARD ARCHITECTURE

- The term Harvard architecture originally referred to computer architectures that uses physically separate storage devices for their instructions and data.
- Harvard architecture has separate data and instruction busses, allowing transfers to be performed simultaneously on both buses.



- The Harvard architecture executes instructions in fewer instruction cycles than the Von Neumann architecture.
- This is because a much greater amount of instruction parallelism is possible in the Harvard architecture.
- Parallelism means that fetches for the next instruction can take place during the execution of the current instruction, without having to either wait for a "dead" cycle of the instruction's execution or stop the processor's operation while the next instruction is being fetched.

MEMORY ORGANIZATION:

- The 8051 has two types of memory and these are **Program Memory** and **Data Memory.**
- Program Memory (ROM) is used to permanently save the program being executed, while Data Memory (RAM) is used for temporarily storing data and intermediate

results created and used during the operation of the microcontroller.

- Depending on the model in use (we are still talking about the 8051 microcontroller family in general) at most a few Kb of ROM and 128 or 256 bytes of RAM is used.

- All 8051 microcontrollers have a 16-bit addressing bus and are capable of addressing 64 kb memory.

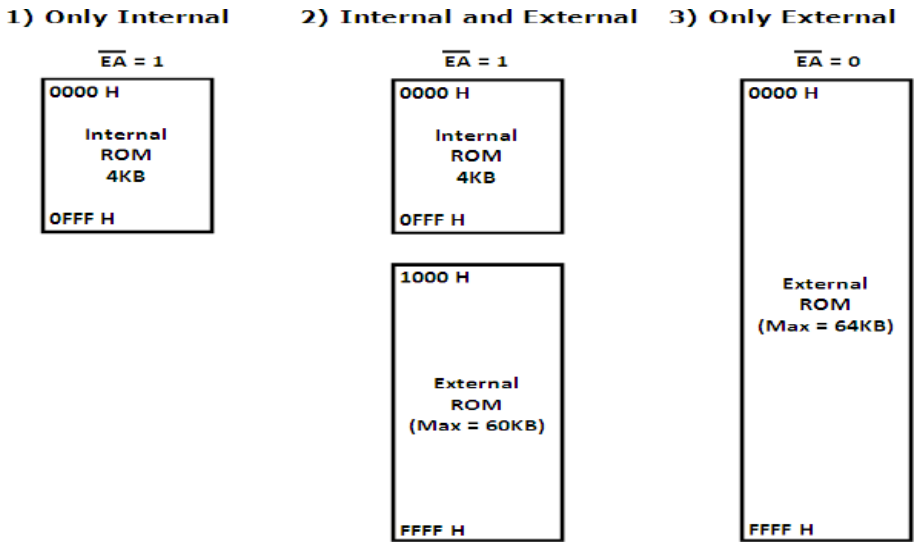
MEMORY ORGANISATION-RAM STRUCTURE:

✓ 8051 operates with 4 different memories:

- Internal ROM
- External ROM
- Internal RAM
- External RAM

- Being based on Harvard Model, 8051 stores programs and data in separate memory spaces. Programs are stored in ROM, whereas data is stored in RAM.
- Microcontrollers are used in appliances.
- Washing machines, remote controllers, microwave ovens are some of the
- **EXAMPLES:**
  Here programs are generally permanent in nature and very rarely need to be modified. Moreover, the programs must be retained even after the device is completely switched off. Hence programs are stored in permanent (non-volatile) memory like ROM.
- Data on the other hand is continuously changed at runtime. For example current temperature, cooking time etc. in an oven.
- Such data is not permanent in nature and will certainly be modified in every usage of the device.
- **Hence Data is stored in writeable memory like RAM.**
- However, sometimes there is permanent data, such as ASCII codes or 7-segment display codes. Such data is stored in ROM, in the form of Look up tables and is accessed using a dedicated addressing mode called Indexed Addressing mode. We will discover this in more depth in further topics.
- We are now going to take a closer look at all four memories.

ROM ORGANIZATION / CODE MEMORY / PROGRAM MEMORY



- ✓ **We can implement ROM in three different ways in 8051**.

1. ONLY INTERNAL ROM:

- 8051 has 4 KB internal ROM.
- In many cases this size is sufficient and there is no need for connecting External ROM. Such systems use only Internal ROM of 8051.
- All addresses from 0000H… 0FFFH will be accessed from Internal ROM. Any address beyond that will be invalid.
- In such systems **EA** will be "1" as Internal ROM is being used.

2. INTERNAL AND EXTERNAL ROM:

- 8051 has 4 KB internal ROM.
- In many cases this size is may be insufficient and we may need to add some External ROM. Such systems use a combination of Internal ROM and External ROM.
- The "total" ROM that can be accessed is 64 KB.
- Since we are using the Internal ROM of 4 KB, the maximum amount of External ROM that can be connected is 60 KB.
- All addresses from 0000H… 0FFFH will be accessed from Internal ROM. Addresses 1000H… FFFH will be accessed from External ROM.
- In such systems EA will be "1" as Internal ROM is being used.

3. ONLY EXTERNAL ROM:

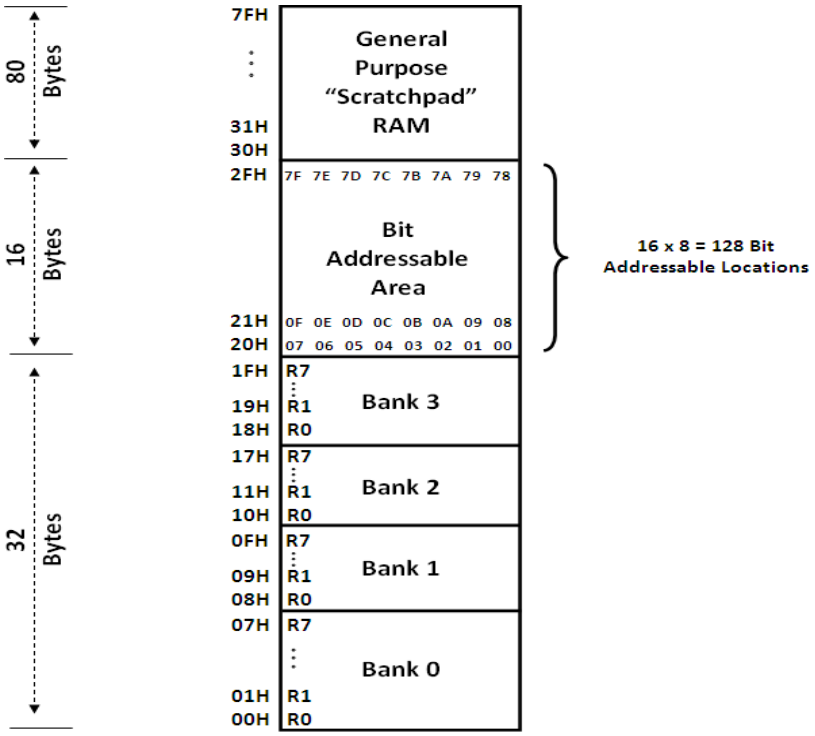
- This is the most interesting case.
- Though 8051 has 4 KB of Internal ROM, the user may choose the discard it and connect only External ROM.
- This may happen due to several reasons.
- The program stored in the Internal ROM may have become invalid or outdated, or the system may need to be upgraded etc.
- Such systems use only External ROM, and the Internal ROM is discarded. Here we can connect up to 64 KB of External ROM.
- All addresses from 0000H… FFFFH will be accessed from External ROM. But do keep in mind, that the Internal ROM is still present in 8051.
- We need to clearly indicate to 8051 that the Internal ROM must be ignored and every address from 0000H… FFFFH must be accessed externally. This is indicated by us to 8051 using **EA.**
- By making **EA = 0**, we inform 8051 that the Internal ROM must be discarded and all ROM must be accessed externally.

✓ Use of EA pin of 8051:

- The EA pin of 8051 decides whether the Internal ROM will be used or not.
- If the Internal ROM has to be used we must make **EA** = 1.
- Now 8051 will Access the internal ROM for all addresses from 0000H to 0FFFH and will only access external ROM for addresses 1000H and beyond.

- But if **EA** = 0, then the Internal ROM is completely discarded.
- Now 8051 will access the External ROM for all addresses from 0000H to FFFFH, hence discarding the internal ROM.
- 8051 checks **EA** pin during every ROM operation where the address is 0000H…
- 0FFFH. If **EA = 1,** this location is accessed from internal ROM.
- If **EA = 0,** this location is accessed from external ROM.
- If the address is 1000H or more, 8051 does not check **EA** as this location can only be present in External ROM.

STRUCTURE OF INTERNAL RAM:



- 8051 has a 128 Bytes of internal RAM. These are 128 locations of 1 Byte each.
- The address range is 00H… 7FH.
- This RAM is used for storing data.
- It is divided into three main parts: Register Banks, Bit addressable area and a general purpose area.

REGISTER BANKS:

- The first 32 locations (Bytes) of the Internal RAM from 00H… 1FH, are used by the programmer as general purpose registers.
- Having so many general purpose registers makes programming easier and faster.
- But as a downside, this also vastly increases the number of opcodes (refer my class lectures for detailed understanding of this).
- Hence the 32 registers are divided into 4 banks, each having 8 Registers R0… R7.
- The first 8 locations 00H… 07H are registers R0… R7 of bank 0.
- Similarly locations 08H… 0FH are registers R0… R7 of bank 1 and so on. A register can be addressed using its name, or by its address.
- E.g. Location 00H can be accessed as R0, if Bank 0 is the active bank.

MOV A, R0; "A" register gets data from register R0.
It can also be accessed as Location 00H, irrespective of which bank is the active

bank.

MOV A, 00H; "A" register gets data from Location 00H.

- The appropriate bank is selected by the RS1, RS0 bits of PSW.

Since PSW is available to the programmer, any Bank can be selected at run-time.

- Bank 0 is selected by default, on reset.

BIT ADDRESSABLE AREA:

- The next 16-bytes of RAM, from 20H… 2FH, is available as Bit Addressable Area.
- We can perform ordinary byte operations on these locations, as well as bit operations.
- As each location has 8-bits, we have a total of →16 X 8 = 128 Addressable Bits.
- These bits can be addressed using their individual address 00H … 7FH. SETB 00H; Will store a "1" on the LSB of location 20H
  CLR 07H; Will store a "0" on the MSB of location 20H

- Normal "BYTE" operations can also be performed at the addresses: 20H … 2FH. MOV 20H, #00H; Will store a "0" on all 8-bits of location 20H.

- Here is something very interesting to know and will also help you understand further topics. The entire internal RAM is of 128 bytes so the address range is 00H… 7FH.
- The bit addressable area has 128 bits so its bit addresses are also 00h… 7FH.

- This means every address 00H… 7FH can have two meanings, it could be a byte address or a bit address.

  - This does not lead to any confusion, because the instruction in which we use the address, will clearly indicate whether it is a bit operation or a byte operations.

  - SETB, CLR etc. are bit ops whereas ADD, SUB etc. are byte operations.
  - SETB 00H; this is a bit operation. It will make Bit location 00H contain a value "1".
  - MOV A, 00H; this is a byte operation. A" register will get 8-bit data from byte location 00H.

GENERAL PURPOSE AREA

- The general-purpose area ranges from location 30H … 7FH.
- This is an 80-byte area which can be used for general data storage.

SFR (SPECIAL FUNCTION REGISTER):

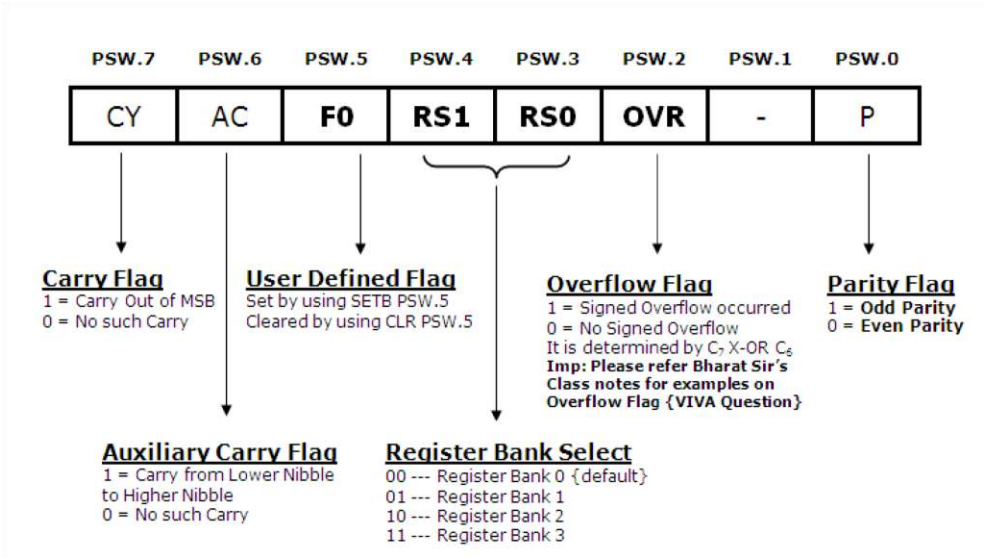
- 8051 has 21, 8-bit Special Function registers.

	NAME	FUNCTION	BYTE ADDRESS	BIT ADDRESS
Used for holding data and status during Programming	A*	Accumulator	0E0H	0E7H...0E0H
	B*	Arithmetic	0F0H	0F7H...0F0H
	PSW*	Program Status Word	0D0H	0D7H...0D0H
Used in instructions to point to memory	SP	Stack Pointer	81H	NA
	DPL	Address External Memory	82H	NA
	DPH	Address External Memory	83H	NA
Used by the respective I/O Ports	P0*	I/O Port latch	80H	87H...80H
	P1*	I/O Port latch	90H	97H...90H
	P2*	I/O Port latch	0A0H	0A7H...0A0H
	P3*	I/O Port latch	0B0H	0B7H...0B0H
Used by the Serial Port	SCON*	Serial Port Control	98H	9FH...98H
	SBUF	Serial Port Data Buffer	99H	NA
Used for Timer Control	TCON*	Timer/Counter Control	88H	8FH...88H
	TMOD	Timer/Counter Mode Control	89H	NA
	TL0	Timer 0 Low Byte	8AH	NA
	TL1	Timer 1 Low Byte	8BH	NA
	TH0	Timer 0 High Byte	8CH	NA
	TH1	Timer 1 High Byte	8DH	NA
Used for Interrupt Control	IE*	Interrupt Enable	0A8H	0AFH...0A8H
	IP*	Interrupt Priority	0B8H	0BFH...0B8H
Used for Power Control	PCON	Power Control	87H	NA

- SFRs are 8-bit registers. Each SFR has its own special function.
- They are placed inside the Microcontroller.
- They are used by the programmer to perform special functions like controlling the timers, the serial port, the I/O ports etc.
- As SFRs are available to the programmer, we will use them in instructions. This causes another problem. SFRs are registers after all, and hence using them would tremendously increase the number of opcodes to reduce the number of opcodes, SFRs are allotted addresses. These addresses must not clash with any other addresses of the existing memories
- Incidentally, the internal RAM is of 128 bytes and uses addresses only from 00H… 7FH. This gives an entire range of addresses from 80H… FFH completely unused and can be freely allotted to the SFRs. Hence SFRs are allotted addresses from 80H… FFH.

- It is not a co-incidence that these addresses are free. The Internal RAM was restricted to 128 bytes instead of 256 bytes so that these addresses are free for SFRs.
- To avoid this problem, even the bits of the SFRs are allotted addresses. These are bit addresses, which are different from byte addresses. These bit addresses must not clash with those of the bit addressable area of the Internal RAM. Amazingly, even the bit addresses in the Internal RAM are 00H… 7FH (again 128 bits), keeping bit addresses 80H… FFH free to be used by the SFR bits.
- So bit addresses 80H… FFH are allotted to the bits of various SFRs.
- Port 0 has a byte address of 80H and its bit addresses are from 80H… 87H.
- A byte operation at address 80H will affect entire Port0.
- **E.g.-**MOV A, P0; this refers to Byte address 80H that's whole Port 0. 12) A bit Operation at 80H will affect only P0.0.
- E.g. SETB P0.0; this refers to bit address 80H that's Port0.0

FLAG REGISTER (PSW) OF 8051:



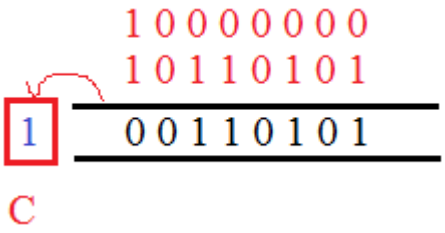
PSW – PROGRAM STATUS WORD

- It is an 8-bit register.
- It is also called the "Flag register", as it mainly contains the status flags.

- These flags indicate status of the current result.
- They are changed by the ALU after every arithmetic or logic operation.
- The flags can also be changed by the programmer.
- PSW is a bit addressable register.
- Each bit can be individually set or reset by the programmer.
- The bits can be referred to by their bit numbers (PSW.4) or by their name (RS1).

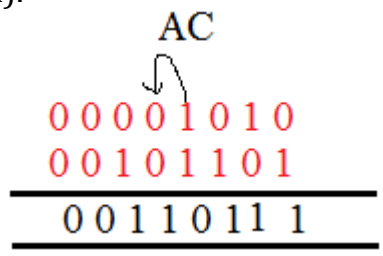
CY - CARRY FLAG

- It indicates the carry out of the MSB, after any arithmetic operation.
- If CY = 1, There was a carry out of the MSB
- If CY = 0, There was no carry out of the MSB



AC – AUXILIARY CARRY FLAG

- It indicates the carry from lower nibble (4-bits) to higher nibble.
- If the 8bits are numbered Bit 7 --- Bit 0, this is the carry from Bit 3 to Bit 4.
- If AC = 1, There was an auxiliary carry
- If AC = 0, There was no auxiliary carry
  - ✓ Note: It is particularly useful in an operation called DA A (Decimal Adjust after Addition).



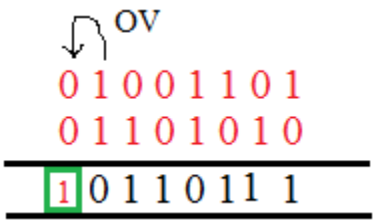
OVR - OVERFLOW FLAG

- It indicates if there was an overflow during a signed operation.
- An 8-bit signed number has the range -80H… 00H… +7FH. Any result,

out of this range causes an overflow.

- If OVR = 1, There was an overflow in the result If OVR = , There was no overflow in the result
- Overflow is determined by doing an Ex-Or between the 2nd last carry (C_6) and the last carry (C_7)

✓ Note: After an overflow, the Sign (MSB) of the result becomes wrong.



P - PARITY FLAG

- It indicates the Parity of the result.
- Parity is determined by the number of 1's in the result.
- If PF = 1, The result has ODD parity
- If PF = 0, The result has EVEN parity

F0 – USER DEFINED FLAG

- This flag is available to the programmer.
- It can be used by us to store any user defined information.
- For example: In an Air Conditioning unit, programmer can use this flag indicate whether the compressor is ON or OFF (1 or 0).
- This flag can be changed by simple instructions like SETB and CLR.
- SETB PSW.5; This makes F0 bit→1
- CLR PSW.5; This makes F0 bit→0

RS1, RS0 – REGISTER BANK SELECT

- The initial 32 locations (bytes) of the Internal RAM are available to the programmer as registers.
- Having so many registers makes programming easier and faster.
- Naming R0… R31, would tremendously increase the number of opcodes.
- Hence the registers are divided into 4 banks: Bank0… Bank3.

- Each bank has 8 registers named R0… R7.
- At a time, only of the four banks is the “active bank”.
- RS1 and RS0 are used by the programmer to select the active bank.

RS1 RS0	REGISTER BANK	SELECTED BY INSTRUCTIONS
0 0	Bank 0	CLR PSW.4 CLR PSW.3
0 1	Bank 1	CLR PSW.4 SETB PSW.3
1 0	Bank 2	SETB PSW.4 CLR PSW.3
1 1	Bank 3	SETB PSW.4 SETB PSW.3

NUMERICAL EXAMPLES FOR FLAG REGISTER:

Example 1:
32 H→0011 0001
23 H→0010 0011
54 H→0101 0100

- Flag Affected: CY=0, AC=0, OVR=0, P=1

Example 2:
39 H→0011 1001
27 H→0010 0111
60 H→0110 0000

- Flag Affected: CY=0, AC=1, OVR=0, P=0

Example 3:
42 H→0100 0010
44 H→0100 0100
86 H→1000 0110

- Flag Affected: CY=0, AC=0, OVR=1, P=1

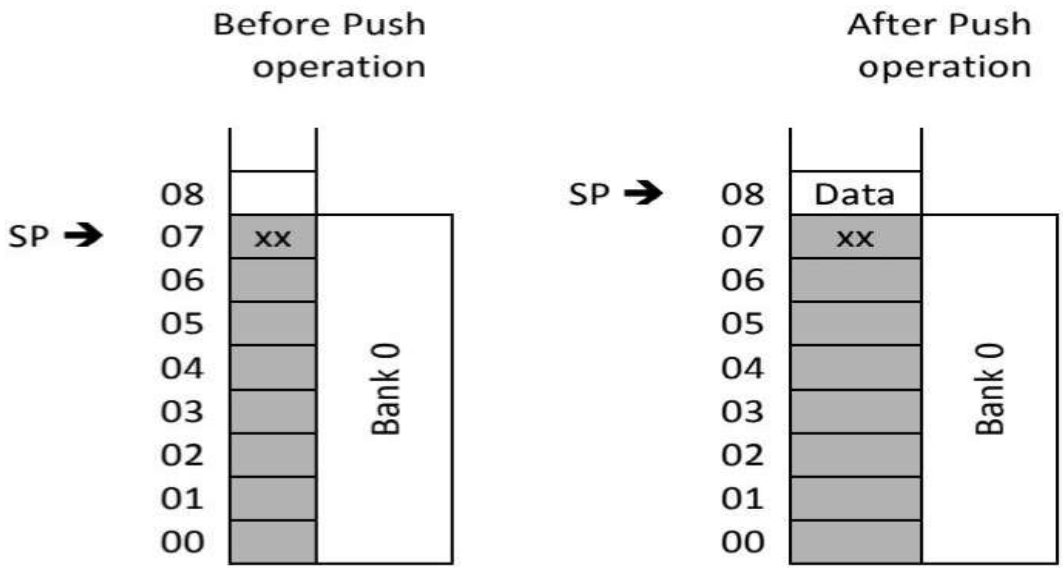
- The result 86H is out of range for a "Signed" Number as it has become greater than +7FH.
- Such an event is called a "Signed Overflow".
- In such a case the MSB of the result gives a wrong sign.
- Though the result is +ve (+86H) the MSB is "1" indicating that the result is –ve.
- Overflow is determined by doing an Ex-Or between the 2nd last Carry and the last Carry.
- Here the 2nd last Carry (the one coming into the MSB) is "1".
- The final carry (The one going out of the MSB) is "0". As "1" Ex-Or "0" = "1", the Overflow flag is "1".

STACK OF 8051:

- Another important element of the Internal RAM is the Stack.
- Stack is a set of memory locations operating in Last in First out (LIFO) manner.
- It is used to store return addresses during ISRs and also used by the programmer to store data during programs.
- In 8051, the Stack can only be present in the Internal RAM.
- This is because, SP which is an 8-bit register, can only contain an 8-bit address and External RAM has 16-bit address. (#Viva)
- On reset SP gets the value 07H.
- Thereafter SP is changed by every PUSH or POP operation in the following manner:

PUSH:	**POP:**
SP → SP + 1	Data→ [SP]
[SP] → New data	**SP→SP – 1**

- The reset value of SP is 07H because, on the first PUSH, SP gets incremented and then data is pushed on to the stack. This means the very first data will be stored at location 08H.
- This does not affect the default bank (0) and still gives the stack, the maximum space to grow.



- The programmer can relocate the stack to any desired location by simply putting a new value into SP register.

2.6 EXPLAIN THE PORT STRUCTURE & OPERATION, TIMER/COUNTERS, SERIAL INTERFACE& EXTERNAL MEMORY:

PORT STRUCTURE & OPERATION OF 8051:

8051 microcontrollers have 4 I/O ports each of 8-bit, which can be configured as input or output. Hence, total 32 input/output pins allow the microcontroller to be connected with the peripheral devices.

Pin configuration, i.e. the pin can be configured as 1 for input and 0 for output as per the logic state.

Input/output (I/O) pin:

All the circuits within the microcontroller must be connected to one of its pins except P0 port because it does not have pull-up resistors built-in.

Input pin:

Logic 1 is applied to a bit of the P register. The output FE transistor is turned off and the other pin remains connected to the power supply voltage over a pull-up resistor of high resistance.

Port 0:

- The P0 (zero) port is characterized by two functions –
- When the external memory is used then the lower address byte (addresses A0A7) is applied on it, else all bits of this port are configured as input/output.

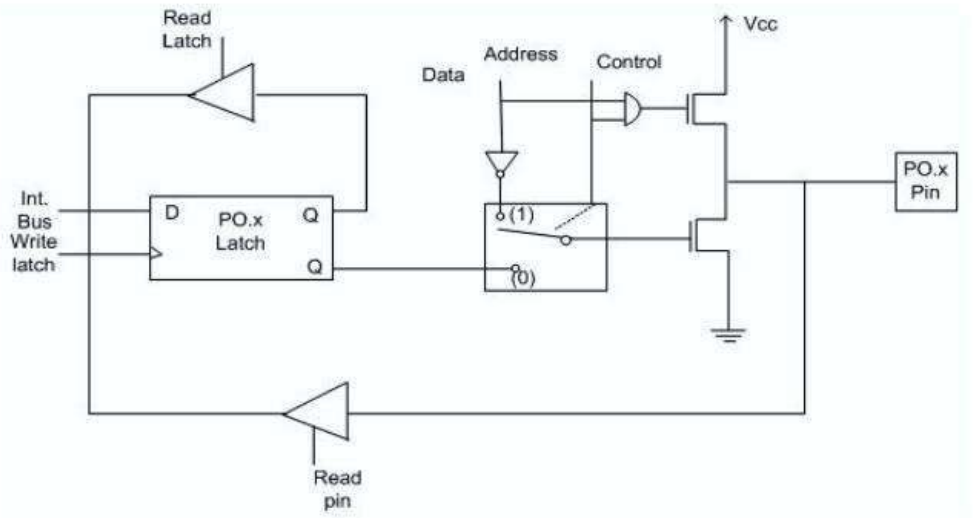
- When P0 port is configured as an output then other ports consisting of pins with built-in pull-up resistor connected by its end to 5V power supply, the pins of this port have this resistor left out.

PORT 0 as an Input Port:

- If any pin of this port is configured as an input, then it acts as if it "floats", i.e. the input has unlimited input resistance and in-determined potential.
- Let us assume that control is '0'. When the port is used as an input port, '1' is written to the latch. In this situation both the output MOSFETs are 'off'. Hence the output pin have floats hence whatever data written on pin is directly read by read pin.

PORT 0 as an Output Port:

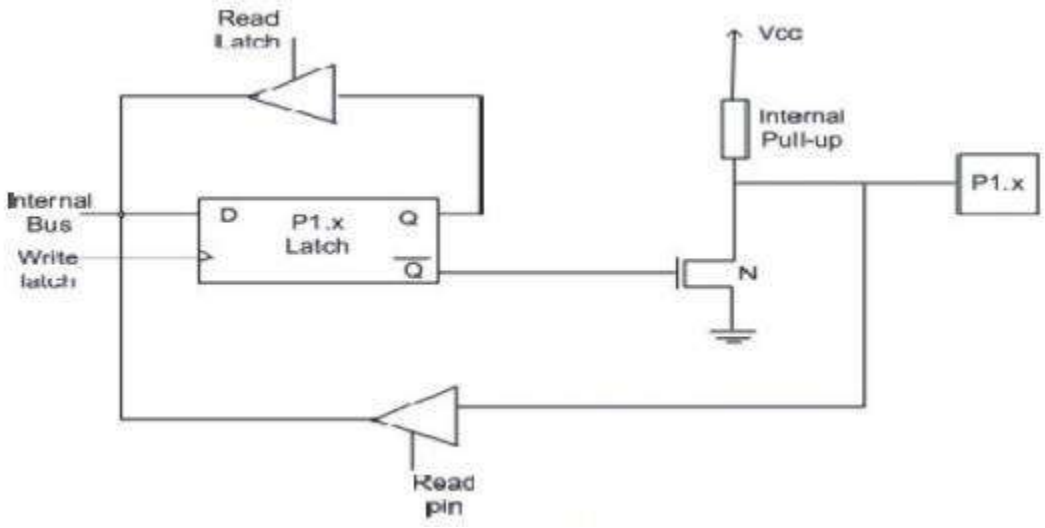
- When the pin is configured as an output, then it acts as an "open drain". By applying logic 0 to a port bit, the appropriate pin will be connected to ground (0V), and applying logic 1, the external output will keep on "floating".
- In order to apply logic 1 (5V) on this output pin, it is necessary to build an external pull up resistor.
- Suppose we want to write 1 on pin of Port 0, a '1' written to the latch which turns 'off' the lower FET while due to '0' control signal upper FET also turns off as shown in fig. above. Here we wants logic '1' on pin but we getting floating value so to convert that floating value into logic '1' we need to connect the pull up resistor parallel to upper FET . This is the reason why we needed to connect pull up resistor to port 0 when we want to initialize port 0 as an output port.
- If we want to write '0' on pin of port 0, when '0' is written to the latch, the pin is pulled down by the lower FET. Hence the output becomes zero.
- When the control is '1', address/data bus controls the output driver FETs. If the address/data bus (internal) is '0', the upper FET is 'off' and the lower FET is 'on'. The output becomes '0'. If the address/data bus is '1', the upper FET is 'on' and the lower FET is 'off'. Hence the output is '1'. Hence for normal address/data interfacing (for external memory access) no pull-up resistors are required.Port-0 latch is written to with 1's when used for external memory access.



Port 0 structure

Port 1:

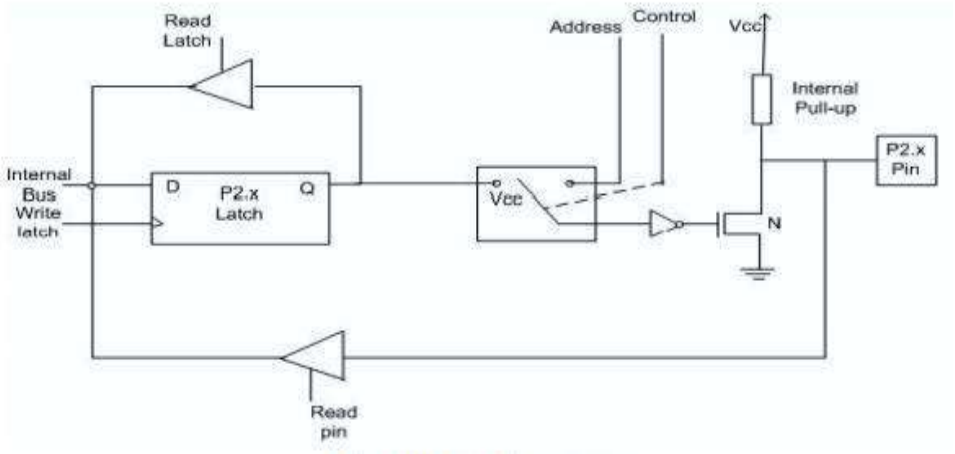
- P1 is a true I/O port as it doesn't have any alternative functions as in P0, but this port can be configured as general I/O only. It has a built-in pull-up resistor and is completely compatible with TTL circuits.
- Port-1 dedicated only for I/O interfacing. When used as output port, not needed to connect additional pull-up resistor like port 0.
- It have provided internally pull-up resistor as shown in fig. below. The pin is pulled up or down through internal pull-up when we want to initialize as an output port.
- To use port-1 as input port, '1' has to be written to the latch. In this input mode when '1' is written to the pin by the external device then it read fine.
- But when '0' is written to the pin by the external device then the external source must sink current due to internal pull-up.
- If the external device is not able to sink the current the pin voltage may rise, leading to a possible wrong reading.



Port 1 structure

Port 2:

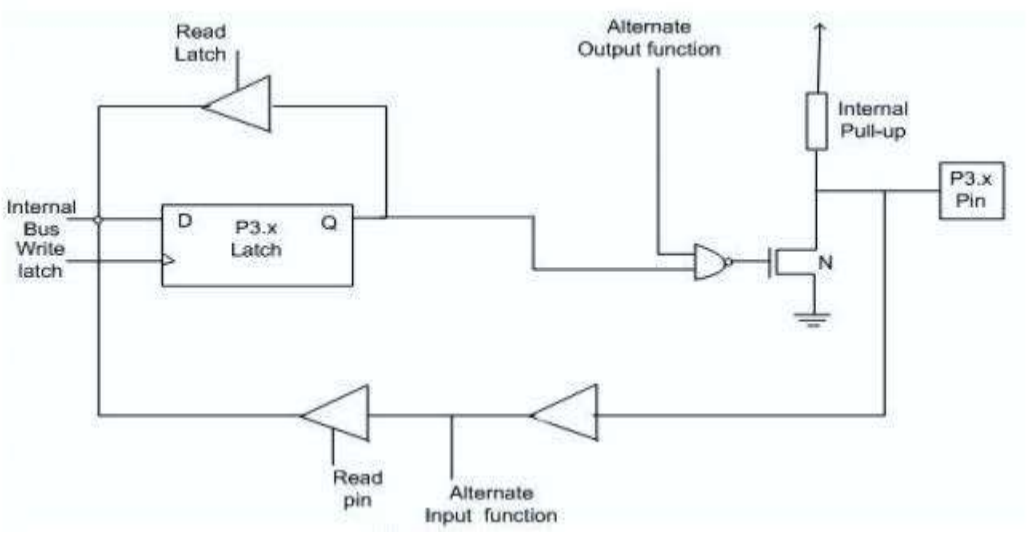
- P2 is similar to P0 when the external memory is used. Pins of this port occupy addresses intended for the external memory chip.
- This port can be used for higher address byte with addresses A8-A15. When no memory is added then this port can be used as a general input/output port similar to Port 1
- Port-2 we use for higher external address byte or a normal input/output port. The I/O operation is similar to Port-1. Port-2 latch remains stable when Port-2 pin are used for external memory access.



Port 2 structure

Port 3:

- In this port, functions are similar to other ports except that the logic 1 must be applied to appropriate bit of the P3 register.
- Following are the alternate functions of port 3:
- P3.0—–RXD
- P3.1—– TXD
- P3.2—– INT0’
- P3.3—– INT1’
- P3.4—– T0
- P3.5—– T1
- P3.6—– WR’
- P3.7—– RD’
- It works as an IO port same like Port 2 as well as it can do lots of alternate work which are discuss above. Only alternate function of port 3 makes its architecture different than other ports.
- Each pin of Port-3 can be individually programmed for I/O operation or for alternate function. The alternate function can be activated only if the corresponding latch has been written to '1'. To use the port as input port, '1' should be written to the latch. This port also has internal pull-up and limited current driving capability.



Port 3 structure

TIMER SECTION OF 8051:

- The 8051 has two timers: timer0 and timer1. They can be used either as timers or as

counters. Both timers are 16 bits wide. Since the 8051 has an 8-bit architecture, each 16-bit is accessed as two separate registers of low byte and high byte.

- There are two 16-bit timers and counters in 8051 microcontroller: **timer 0 and timer 1**. Both timers consist of 16-bit register in which the lower byte is stored in TL and the higher byte is stored in TH. Timer can be used as a counter as well as for timing operation that depends on the source of clock pulses to counters.
- 8051 has 2, 16-bit Up Counters T1 and T0.
- If the counter counts internal clock pulses it is known as timer.
- If it counts external clock pulses it is known as counter.
- Each counter is divided into 2, 8-bit registers TH1 - TL1 and TH0 - TL0.
- The timer action is controlled mainly by the TCON and the TMOD registers.

TCON - Timer Control (SFR) [Bit-Addressable As TCON.7 to TCON.0]

TF1	**TR1**	**TF0**	**TR0**	IE1	IT1	IE0	IT0

TF1 and TF0: (Timer Overflow Flag)

- Set (1) when Timer 1 or Timer 0 overflows respectively i.e. its bits roll over from all 1's to all 0's.
- Cleared (0) when the processor executes ISR (address 001BH for Timer 1 and 000BH for Timer 0).

TR1 and TR0: (Timer Run Control Bit)

- Set (1) - Starts counting on Timer 1 or Timer 0 respectively.
- Cleared (0) - Halts Timer 1 or Timer 0 respectively.

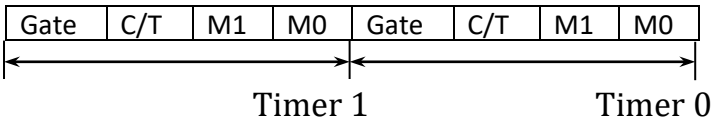
IE1 and IE0: (External Interrupt Edge Flag)

- Set (1) when external interrupt signal received at INT1 or INT0 respectively.
- Cleared (0) when ISR executed (address 0013H for Timer 1 and 0003H for Timer 0).

IT1 and IT0: (External Interrupt Type Control Bit)

- Set (1) - Interrupt at INT1 or INT0 must be -ve edge triggered.
- Cleared (0) - Interrupt at INT1 or INT0 must be low-level triggered.

TMOD - Timer Mode Control (SFR) [NOT Bit-Addressable]



C/T: (<u>Counter/Timer</u>)

- Set (1) - Acts as Counter (Counts external frequency on T1 and T0 pin inputs).
- Cleared (0) - Acts as Timer (Counts internal clock frequency, fosc/12).

Gate: (<u>Gate Enable Control bit</u>)

- Set (1) - Timer controlled by hardware i.e. INTX signal.
- Cleared (0) – Counting independent of INTX signal.

M1, M0: (<u>Mode Selection bits</u>)

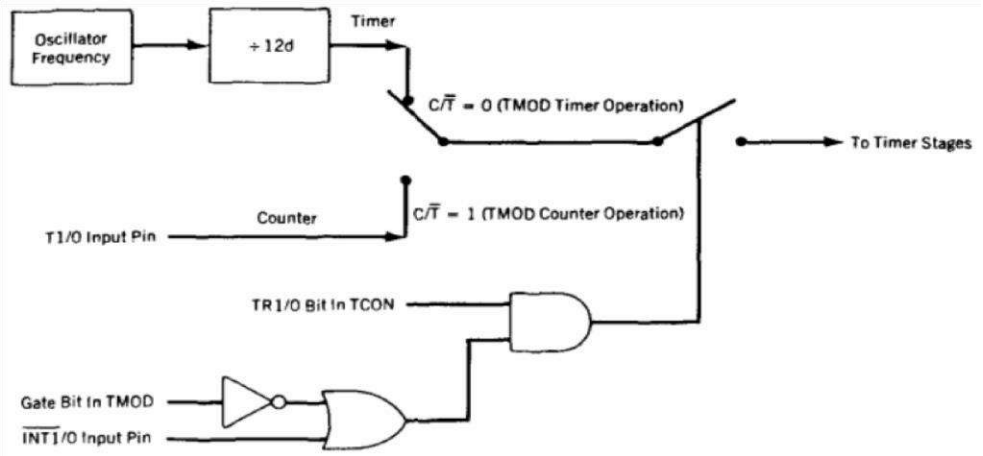
- Used to select the operational modes of the respective Timer.

M1 M0	Timer Mode
0 0	Mode 0
0 1	Mode 1
1 0	Mode 2
1 1	Mode 3

<u>Timer Counter Interrupts</u>

- To use the timer, a certain count value is placed in the Count Register.
- This value is the **→Max count—desired count+1**
- On each count (rising edge of the input clock) the counter increments its value.
- When the counter rolls over (i.e. form all 1's to all 0's) it is said to overflow.
- Thus the Timer Overflow Flag, TFX (TF1 or TF0) is set.
- If timer interrupt is enabled then the Timer Interrupt will occur on overflow.

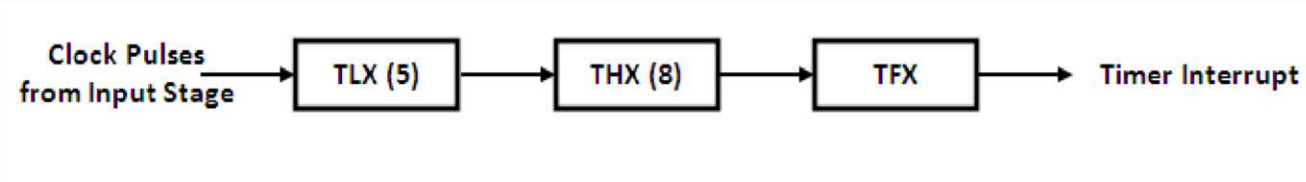
<u>Timer Counter Logic</u>



- As shown above, based on C/T bit the timer functions as a Counter or as a Timer.
- If it is a Timer, it will count the internal clock frequency of 8051 divided by 12_d (f/12).
- If it is a Counter, the input clock signal is applied at the TX (T1 or T0) input pins for Timer1 or
- Timer0 respectively. #please refer Bharat Sir's Lecture Notes for this...
- As shown the Timer is running only if the TRX bit (TR1 or TR0) is set.
- Also if the Gate bit is set in the TMOD then the INTX (INT1 or INT0) pin must be "high (1)" for the timer to count.

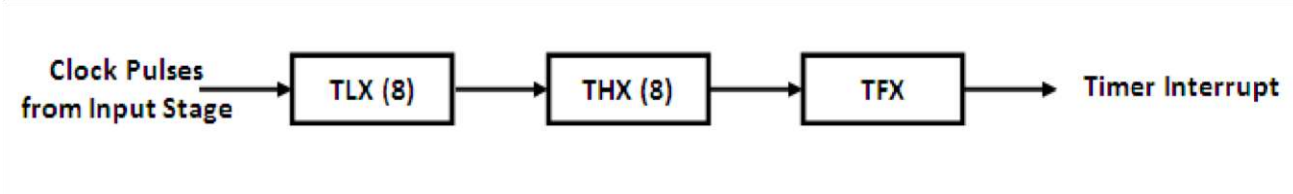
TIMER MODES:

a) Timer Mode 0 (13-bit Timer/Counter)



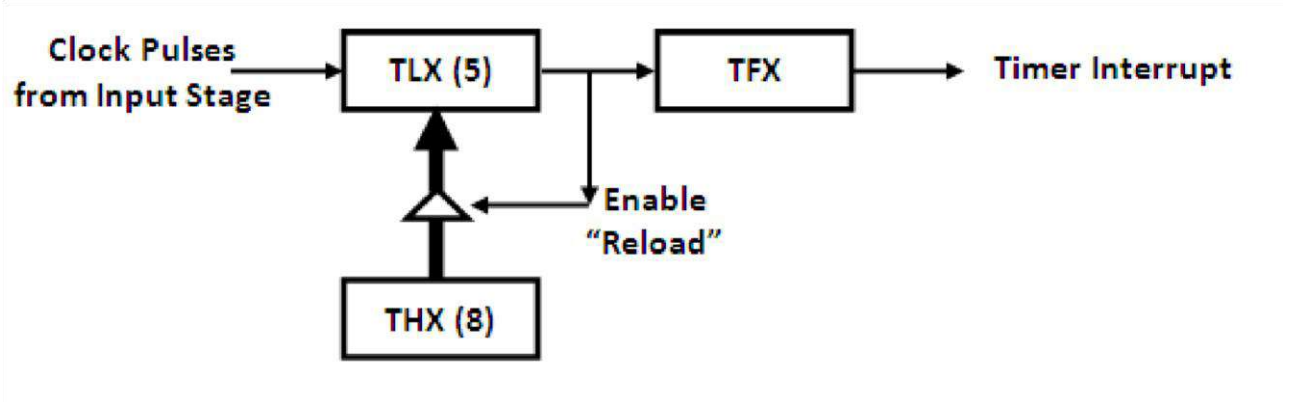
- THX is used as an 8-bit counter.
- TLX is used as a 5-bit pre-set. Hence 13-bits are used for counting.
- On each count the TLX increments.
- Each time TLX rolls-over, THX increments.
- Thus the input frequency is divided by 32 (5-bits of TLX and $2^5 = 32$).
- The timer overflow flag TFX is set only when THX overflows i.e. rolls from FFH to 00H. Max Count = $2^{13} = 8K = 8192$ (1FFFH). Hence Max Delay→ 8192(12/f)

b) Timer Mode 1 (16-bit Timer/Counter)



- All 16-bits of the Counter are used (8 bits of THX and 8 bits of TLX).
- On each count the 16-bit Timer increments.
- The timer overflow flag TFX is set when the Timer rolls-over from FFFFH to 0000H. Max Count => 2^{16} = 16K = 65536 (FFFFH). Hence Max Delay →65536(12/f).

c) Timer Mode 2 (Auto reload TL from TH)

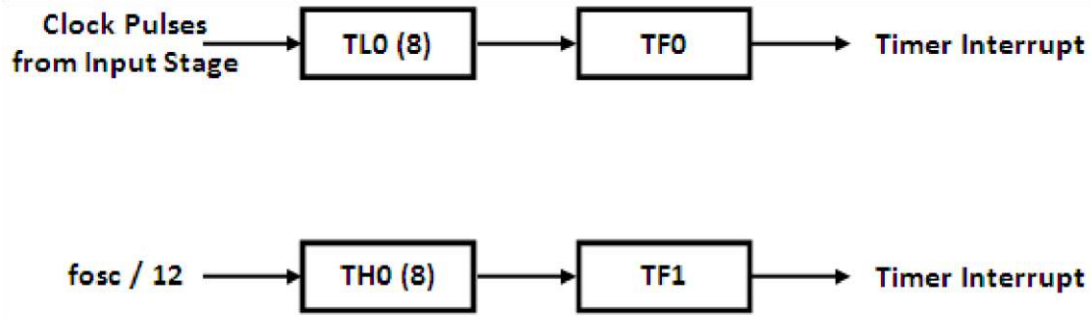


- TLX is used as an 8-bit counter.
- THX holds the count value to be reloaded.
- On each count TLX increments.
- When TLX rolls-over (i.e. from FFH to 00H), the following events take place:

  1. Timer overflow flag TFX is set, hence timer interrupt occurs.

  2. The value of THX is copied into TLX. Hence TLX is auto-reloaded form THX, and the process repeats.
- Thus the timer interrupt occurs at regular intervals "Continuously".
- This mode is used to generate a desired frequency using the Timer Flag. Max Count è 2^8 = 256 (FFH). Hence Max Delay è 256(12/f).

d) Timer Mode 3 (Two 8-bit Timers Using Timer0)



- Timer 0 is used as 2 separate 8-bit timers TH0 and TL0.
- TL0 uses the control bits (TR0 and TF0) of Timer 0.
- It can work as a Timer or a Counter.
- TH0 uses the control bits (TR1 and TF1) of Timer 1.
- It can work only as a Timer. #please refer Bharat Sir's Lecture Notes for this...
  Timer 1 can be in Mode 0, Mode 1, or Mode 2, but will not generate an interrupt.

8051 TIMER/COUNTER (HARDWARE DELAY) PROGRAMMING:

Example 1:

WAP to generate a delay of 20 µsec using internal timer-0 of 8051. After the delay send a "1" through Port3.1. Assume Suitable Crystal Frequency

NOTE:

- In 8051, if we select a Crystal of 12 MHz, then Timer freq will be $f_{osc}/12$→ 1MHz.
- Hence each count will require 1/1MHz è 1 µsec. Thus for 20 µsec, the Desired Count will be 20_d→ 14H. For an Up-Counter (Mode 1):
- **Count = Max Count – Desired Count + 1 Count = FFFF – 14 + 1**
- **<u>Count = FFECH</u>** #Please refer Bharat Sir's Lecture Notes for this...

```
                        ; Program
SOLN: MOV TMOD,         TMOD→(0000
#01H                    0001)2      ...Timer0
                        Mode1
      MOV TL0, #0ECH    ; Load lower byte of
                        Count
      MOV TH0, #0FFH    ; Load upper byte of
                        Count
```

```
MOV TCON, #10H          ; Program TCON→ (0001 0000)2…start Timer0
WAIT: JNB TCON.5, WAIT  ; Wait for overflow
SETB P3.1               ; Send a "1" through Port3.1
MOV TCON, #00H          ; Stop Timer0
HERE: SJMP HERE         ; End of program
```

Example 2:
WAP to generate a Square wave of 1 KHz from the TxD pin of 8051, Q11 usingTimer1. Assume Clock Frequency of 12 MHz

NOTE:

- For a Square wave of 1 KHz, the delay required is .5 msec.
- We know, each count will require 1/1MHz→ 1 µsec.
- Thus for 500 µsec, the Desired Count will be 500_d → 01F4H. For an Up-Counter (Mode 1):

- Count = Max Count – Desired Count + 1
- Count = FFFF – 01F4 + 1
- Count = FE0CH

```
SOLN: CLR P3.1          ; Clear Txd Line initially
MOV TMOD, #10H          ; Program TMOD→(0001 0000)2…Timer1 Mode1
REPEAT: MOV TL1, #0CH   ; Load lower byte of Count
MOV TH1, #0FEH          ; Load upper byte of Count
MOV TCON, #40H          ; Program TCON→ (0100 0000)2…start Timer1
WAIT: JNB TCON.7, WAIT  ; Wait for overflow
```

```
CPL P3.1                    ; Toggle Txd pin after
                              the delay
MOV TCON, #00H              ; Stop Timer1
SJMP REPEAT                 ; Repeat the process
```

Example 3:

WAP to generate a Rectangular wave of 1 KHz, having a 25% Duty Cycle from the TxD pin of 8051, using Timer1. Assume XTAL of 12 MHz

NOTE:

- For a Rectangular wave of 1 KHz, having 25% Duty Cycle: T_{ON} = 250 µsec; T_{OFF} = 750 µsec.
- **For T_{ON}**: Desired Count = 250_d è 00FAH
- Count $_{ON}$ = Max Count – Desired Count + 1
- Count $_{ON}$ = FFFF – 00FA + 1
- Count $_{ON}$ = FF06H
- **For T_{OFF}**: Desired Count = 750_d è 02EEH
- Count $_{OFF}$ = Max Count – Desired Count + 1
- Count $_{OFF}$ = FFFF – 02EE + 1
- **Count $_{OFF}$ = FD12H**

```
SOLN: MOV TMOD,             ; Program TMOD (0001
#10H                          0000)2...Timer1 Mode1


REPEAT: MOV TL1,            ; Load lower byte of
#06H                          Count ON
 MOV TH1, #0FFH             ; Load upper byte of
                              Count ON
 SETB P3.1                  ; Display"1"at Txd
 MOV TCON, #40H             ; Program TCON
                              (0100 0000)2...
                              startTimer1
ON: JNB TCON.7, ON          ; Maintain"1" at Txd
 CLR P3.1                   ; Clear Txd
 MOV TCON, #00H             ; Stop Timer1

 MOV TL1, #12H              ; Load lower byte of Count
```

```
MOV TH1, #0FDH          ; Load upper byte of Count OFF
MOV TCON, #40H          ; Program TCON (0100 0000)2…start Timer1
OFF: JNB TCON.7, OFF    ; Maintain "0" at Txd
MOV TCON, #00H          ; Stop Timer1

SJMP REPEAT             ; Repeat the process
```

- **NOTE:**
- If system frequency=12MHZ, it is clear that 1 count requires 1 msec.
- In mode, we have 16-bit count.
- Hence max pulses that can be desired is 2^{16}=65536
- Count=max count-desired count+1=65535-65535+1=0
- Thus we will get max delay if we load the count as 0000H, as it will have to "roll-over" back to 0000H to overflow.
- Hence max delay if XTAL is of 12 MHz.is 65536 µsec→65.536 msec.
- Similarly max delay if XTAL is of 11.0592 MHz…is 71106 µsec→71.106 msec.

Example 4:
WAP to generate a delay of 1 SECOND using Timer1. Assume Clock Frequency of 12 MHz (Popular Question in College!)

- **NOTE:**
- Max delay if XTAL is of 12 MHz … is 65536 µsec è 65.536 msec. Hence to get a delay of 1 second, we will have to perform the counting repeatedly in a loop.
- Let's keep the Desired Count 50000. (50 msec delay)
- Now 50000_d = C350H
- Count = Max Count – Desired Count + 1 Count = FFFF – C350 + 1
- **<u>Count = 3CB0H</u>**
- We will have to perform this counting 1sec/50msec times→**<u>20 times</u>**

```
SOLN: MOV TMOD, #10H    ; Program TMOD→(0001 0000)2...Timer1 Mode1

MOV R0, #14H            ; Load count 20 in R0
REPEAT: MOV TL1, #0B0H  ; Load lower byte of Count ON
```

```
MOV TH1, #3CH          ; Load upper byte of Count ON
MOV TCON, #40H         ; Program TCON (0100 0000)2…start Timer1
WAIT: JNB TCON.1, WAIT ; Wait for an overflow
MOV TCON, #00H         ; Stop Timer1
DJNZ R0, REPEAT        ; repeat the process 20 times
HERE: SJMP HERE:       ; End of program
```

Example 5

WAP to read the data from Port1, 10 times, each after a 1 sec delay. Store the data from RAM locations 20H onwards. When the operation is complete, ring an "Alarm" connected at Port3.1. Assume CLK = 12 MHz

- **NOTE:**
- As seen from the previous program, for a delay of 1 second, we have **Count = 3CB0H.** Counting has to be performed **20 times.**
- Also note that all ports of 8051 are o/p ports by default.
- To program a port as i/p ports, **all "1"s** must be sent through it.

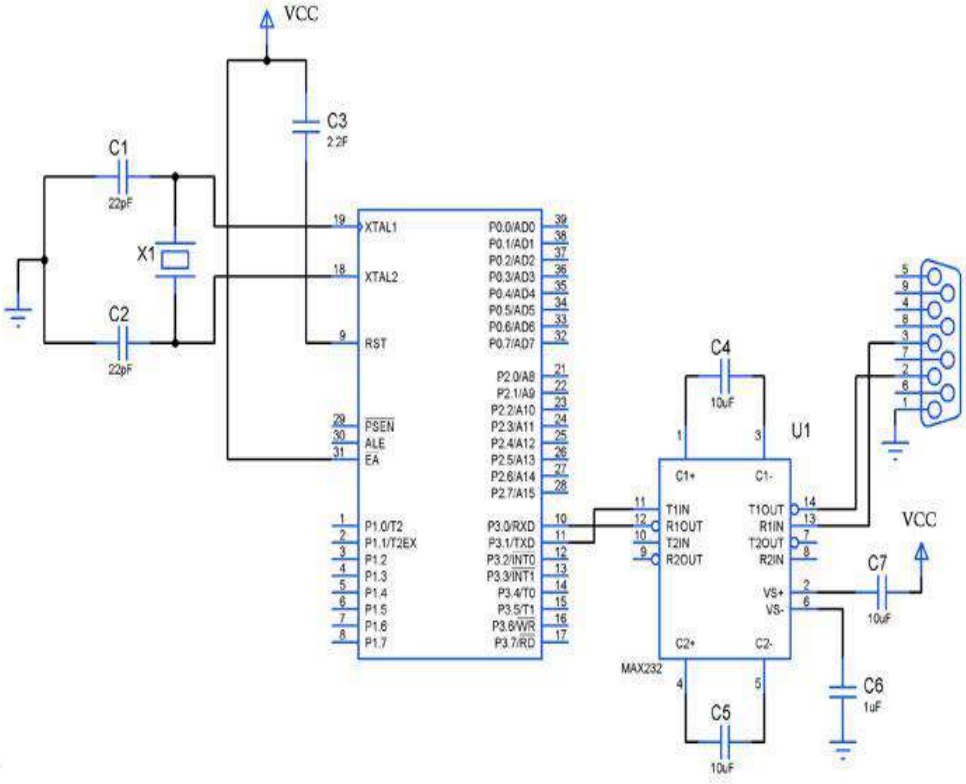
```
SOLN: CLR P3.1          ; Clear Port3.1 line
MOV TMOD, #10H          ; Program TMOD→(0001 0000)2...Timer1 Mode1
MOV 90H, #0FFH          ; Program Port1 as i/p by sending all "1"s through it

REPEAT: MOV R0, #0AH    ; Load Data Count of 10 in R0
MOV R1, #20H            ; Load Storage address in R1
MOV @R1, 90H            ; Read data from Port
INC R1                  ; Increment data storage address from next Iteration
ACALL DELAY             ; Call delay of 1 sec before going into next Iteration
```

```
 DJNZ R0, REPEAT          ; Repeat till all 10 bytes
                            are read
 SETB P3.1                ; Ring "Alarm" at Port3.1
HERE: SJMP HERE:          ; End of program

DELAY: MOV R2, #14H       ; Load count 20 in R0
REPEAT: MOV TL1,          ; Load lower byte of
#0B0H                       Count ON
 MOV TH1, #3CH            ; Load upper byte of
                            Count ON
 MOV TCON, #40H           ; Program TCON
                            (0100 0000)…start Timer1

WAIT: JNB TCON.1,         ; Wait for an overflow
WAIT
 MOV TCON, #00H           ; Stop Timer1
 DJNZ R2, REPEAT          ; Repeat the process 20
                          times
 RET                      ; End of delay routine
```

SERIAL INTERFACE & EXTERNAL MEMORY:

- Microcontrollers need to communicate with external devices such as sensors, computers and so on to collect data for processing.
- Data communication is generally done by means of two methods – Parallel and Serial mode.
- In parallel mode data bits are transferred faster using more data pins. But when comes to a Microcontroller, we cannot afford to dedicate many pins for data transfer.
- UART or Serial communication in 8051 microcontroller will allow the controller to send and receive data's just by using two pins
- Serial Communication uses only two data pins to establish communication between Microcontroller and external devices.

- In this mode of communication data is transferred one bit at a time. This article describes Interfacing of 8051 with PC to establish communication through its serial port RS232.



Using the Serial Port:

- 8051 provides a transmit channel and a receive channel of serial communication.
- The transmit data pin (TXD) is specified at P3.1, and the receive data pin (RXD) is at P3.0.
- The serial signals provided on these pins are TTL signal levels and must be boosted and inverted through a suitable converter (Max232) to comply with RS232 standard. All modes are controlled through SCON, the Serial Control register. The SCON bits are defined as SM0, SM1, SM2, REN, TB8, RB8, TI, and RI from MSB to LSB. The

RS232 AND MAX232:

- To establish communication between a controller and PC, we must use serial I/O protocol RS-232 which was widely used in PC and several devices. PC works on RS-232 standards which operates at a logic level of -25V to +25V. But Microcontrollers use TTL logic which works on 0-5V is not compatible with the RS-232 voltage levels.

- MAX232 is a specialized IC which offers intermediate link between the Microcontroller and PC. The transmitter of this IC will convert the TTL input level to RS-232 Voltage standards. Meanwhile the receiver of this IC will convert RS-232 input to 5V TTL logic levels. Read the complete working of MAX232 IC.

SCON REGISTER:

- It a bit addressable register used to set the mode in which serial communication takes place in the controller. The above figure shows the configuration of the SCON register. Here is the list of functions of each bit.

SM0	SM1	SM2	REN	TB8	RB8	TI	RI

- It a bit addressable register used to set the mode in which serial communication takes place in the controller. The above figure shows the configuration of the SCON register. Here is the list of functions of each bit.

SM0	SM1	Mode	Baud Rate
0	0	Serial Mode 0	1/12 Osc Frequency
0	1	Serial Mode 1	Determined By timer 1
1	0	Serial mode 2	1/64 or 1/32 Osc Frequency
1	1	Serial mode 3	Determined by timer 1

- **SM0, SM1:** Serial Mode control Bits
- **SM2:** Multiprocessor mode control bit, logic 1 enables Multi processor mode and 0 for normal mode.
- **REN:** Enables Serial reception. If set, it enables the reception otherwise the reception is disabled.
- **TB8:** It is the 9th bit of the data that is to be transmitted.
- **RB8:** It is used in modes 2 and 3, it is the 9th bit received by the microcontroller.
- **TI:** It is known as Transmit Interrupt flag which is set by hardware to indicate the end of a transmission. It has to be cleared by the software.
- **RI:** It is known as Receive Interrupt flag which is set by hardware to indicate the end of a reception. It has to be cleared by the software.

BAUD RATE:

- It is defined as number of bits transmitted or received per second and usually expressed in Bits per second bps. For mode 0 and mode 2 the baud rate is determined by means of 1/12, 1/32 or 1/64 of crystal frequency whereas for mode 1 and 3 it is determined by means of timer 1.

Baud Rate (bps)	TH1 (Hex value)
9600	FD
4800	FA
2400	F4
1200	E8

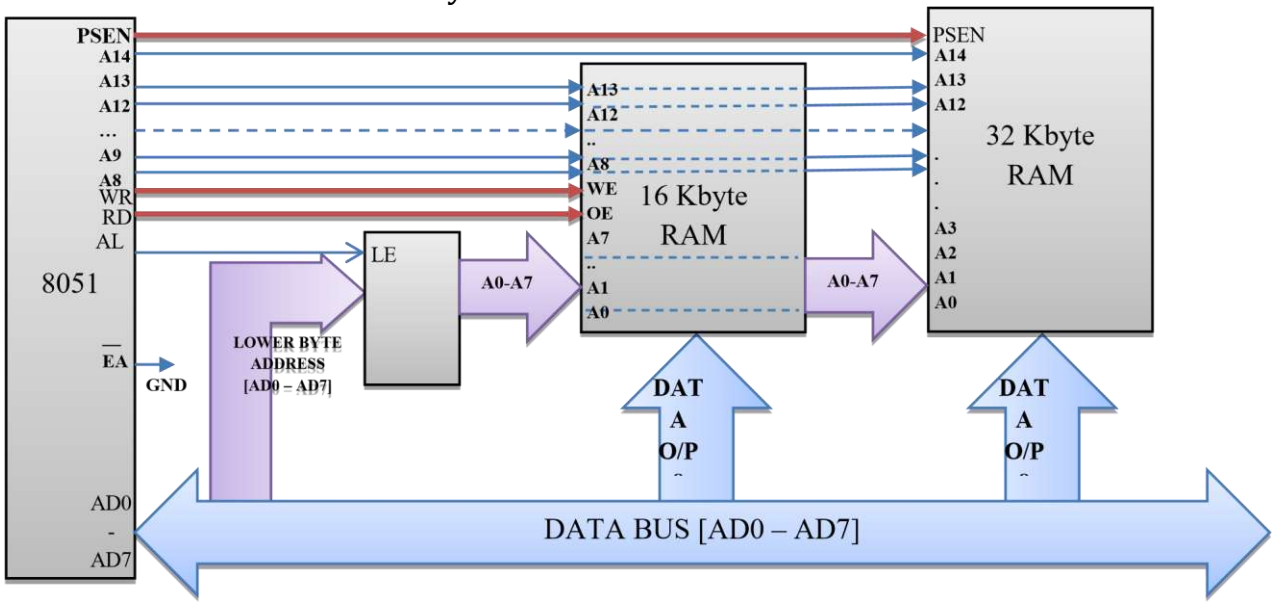
SBUF REGISTER:

- It is an 8 bit register that holds the data needed to be transmitted or the data that is received recently.
- The serial port of 8051 is full duplex so the microcontroller can transmit and receive data using the register simultaneously.

EXTERNAL MEMORY INTERFACING:

Example:
Interfacing of 16 K Byte of RAM and 32 K Byte of EPROM to 8051

- Number of address lines required for 16 Kbyte memory is 14 lines and that of 32Kbytes of memory is 15 lines.
- The connections of external memory is shown below.



- The lower order address and data bus are multiplexed. De-multiplexing is done by the latch.

- Initially the address will appear in the bus and this latched at the output of latch using ALE signal.

- The output of the latch is directly connected to the lower byte address lines of the memory.
- Later data will be available in this bus. Still the latch output is address itself.
- The higher byte of address bus is directly connected to the memory. The number of lines connected depends on the memory size.
- The **RD** and **WR** (both active low) signals are connected to RAM for reading and writing the data.
- PSEN of microcontroller is connected to the output enable of the ROM to read the data from the memory.
- **EA** (active low) pin is always grounded if we use only external memory. Otherwise, once the program size exceeds internal memory the microcontroller will automatically switch to external memory.

UNIT-3: 8051 ADDRESSING MODES & INSTRUCTION SET

3.1 ADDRESSING MODES OF 8051:

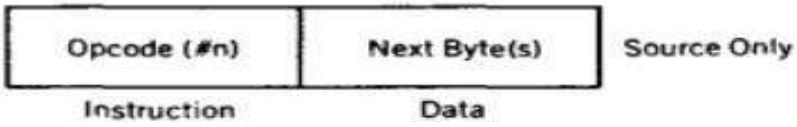
Addressing Modes is the manner in which operands are given in the instruction. 8051 supports the following 5 addressing modes:

1. Immediate addressing mode
2. Register addressing mode
3. Direct addressing mode
4. Indirect addressing mode
5. Index addressing mode

1. IMMEDIATE ADDRESSING MODE

- In this addressing mode, the Data is given in the Instruction itself.
- We put a "#" symbol, before the data, to identify it as a data value and not as an address.
- **Example**

MOV A, #35H ; A←35H
MOV DPTR, #3000H; DPTR← 3000H

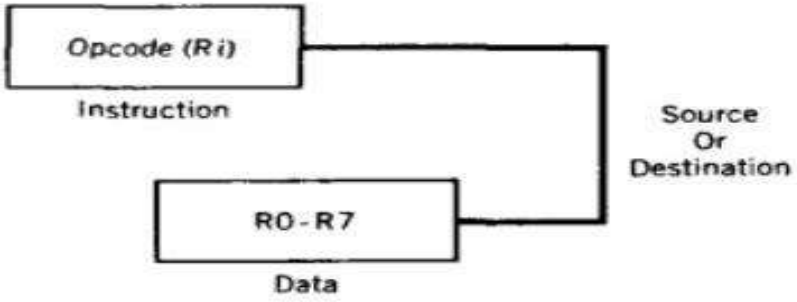


Immediate Addressing Mode

2. REGISTER ADDRESSING MODE

- In this addressing mode, Data is given by a Register in the instruction.
- The permitted registers are A, R7 … R0 of each memory bank.
- Note: Data transfer between two RAM registers is not allowed.
- **Example**

MOV A, R0 ; A← R0 … If R0 = 25H, then A gets the Value 25H.
MOV R5, A ; R5← A
MOV Rx, Ry ; NOT ALLOWED. That's because this would allow 64 combinations of register.
; As registers invite opcodes, this would need 64 opcodes!



Register Addressing Mode

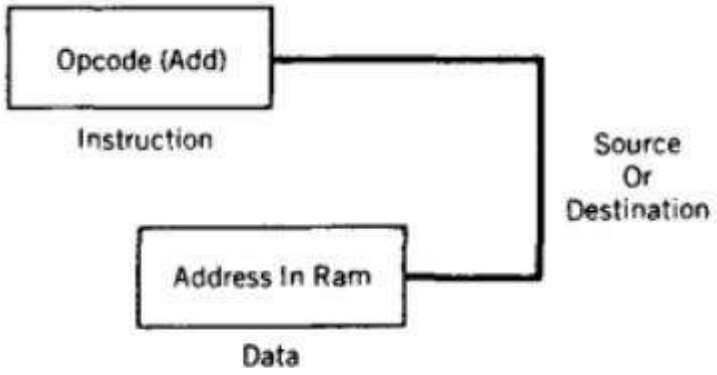
3. DIRECT ADDRESSING MODE

- Here, the address of the operand is given in the instruction.
- Only Internal RAM addresses (00H…7FH) and SFR addresses (from 80H to FFH) allowed.

- **Example**

  MOV A, 35 ; A←Contents of RAM location 35H
  MOV A, 80H ; A←contents of port 0 (SFR at address 80H)
  MOV 20H, 30H ; [20H]←[30H]

  i.e. Location 20H gets the contents of location 30H.

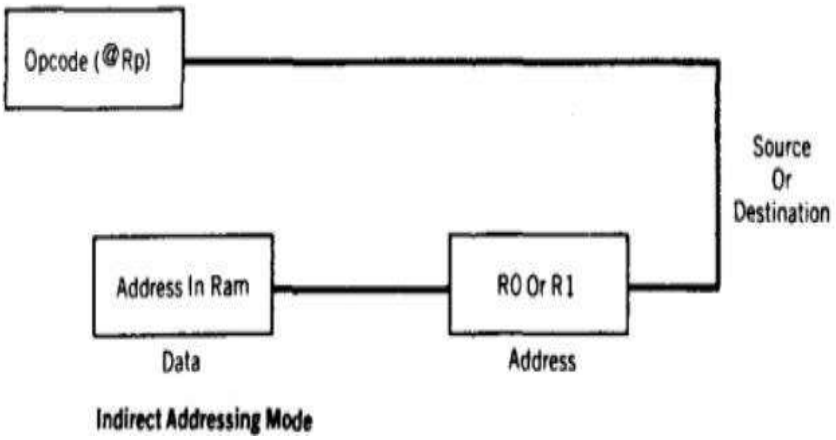


Direct Addressing Mode

4. INDIRECT ADDRESSING MODE

- Here, the address of the operand is given in a register.
- Internal RAM and External RAM can be accessed using this mode.
- The advantage of giving an address using a register is that we can increment the address in a loop, by simply incrementing the register, and

hence access a series of locations.



Indirect Addressing Mode

INTERNAL RAM: (8-BIT ADDRESS GIVEN BY R0 OR R1):

- ONLY R1 or R0, called as Data Pointers, can be used to specify address (00H … 7FH).
- A "@" sign is present before the register to indicate that the register is giving an address.
- **Example:**

MOV A, @R0 ; A← [R0]

; i.e. **A**←Contents of Internal RAM Location whose address is given by R0.

; if R0 = 25H, then A gets the contents of Location 25H from Internal RAM.

MOV @R1, A ; [R1]← A

; i.e. Internal RAM Location pointed by R1 gets value of A.

EXTERNAL RAM: (16 BIT ADDRESS GIVEN BY DPTR):

- For the External RAM, address is provided by R1 or R0, or by DPTR.
- If DPTR is used to give an address, then the full 64KB range of External RAM from 0000H… FFFFH is available. This is because DPTR is 16-bit and 2^{16} = 65536.
- An "X" is present in the instruction, to indicate External RAM.

- **Example**

MOVX A, @DPTR ; A ←[DPTR] ^

; A gets the contents of External RAM location whose address is given by DPTR.
; If DPTR=2000H, then A gets contents of location 0025H from the external RAM

MOVX @DPTR, A ; [DPTR] ^←A

; I.e. A is stored at the External RAM location whose address is given by DPTR.

EXTERNAL RAM: (8 BIT ADDRESS GIVEN BY R0 OR R1):

- If R0 or R1 is used to give an address, then only the first 256 locations of External RAM is available from 0000 H to 00FF H.
- This is because R0 or R1 are 8-bit and 2^8 = only 256.
- **Example**

MOVX A,@R0 ; A← [R0] ^

; i.e. **A** gets the contents of External RAM Location whose address is given by R0.

; If R0 = 25H, then A gets contents of Location 0025H from the External RAM

MOVX @R1, A ; [R1] ^←A

; i.e. **A** is stored at the External RAM Location whose address is given by R1

5. INDEXED ADDRESSING MODE

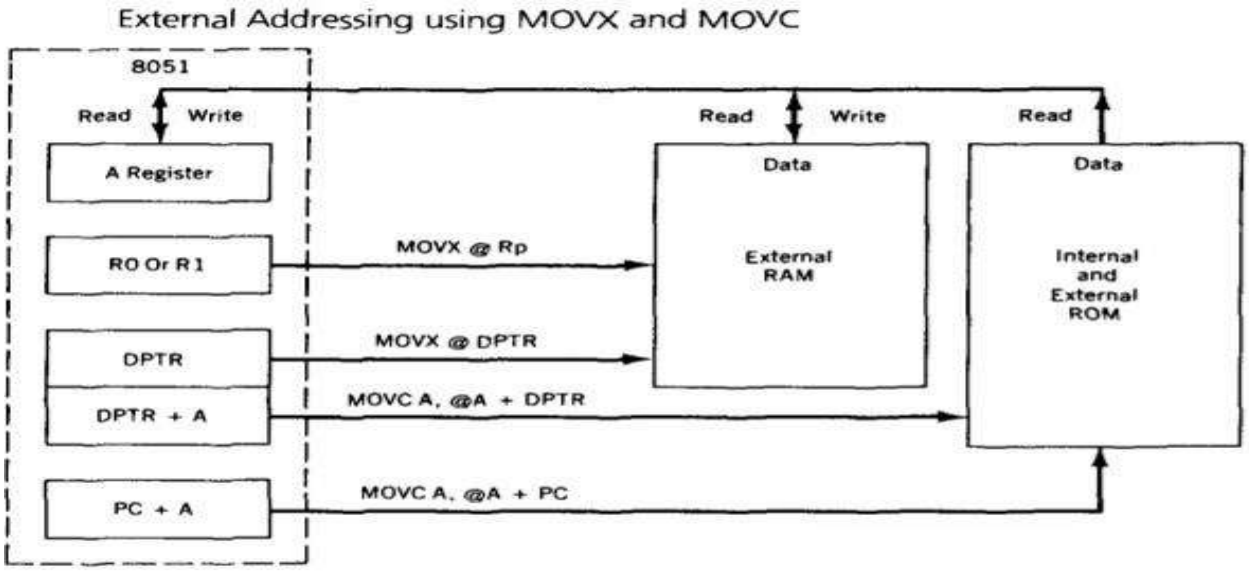
- This mode is used to access data from the Code memory (Internal ROM or External ROM).
- In this addressing mode, address is indirectly specified as a "SUM" of (A and DPTR) or (A and PC).
- This is very useful because ROM contains permanent data which is stored in the form of Look Up tables.
- To access a Look Up table, address is given as a SUM or two registers, where one acts as the base and the other acts as the index within the table.
- A "C" is present in such instructions, to indicate Code Memory.

- **Example**

MOVC A, @A+DPTR; A← Contents of a ROM Location pointed by A+DPTR.
; If DPTR = 0400H and A = 05H,
; Then **A** gets the contents of ROM Location whose address is 0405 H.

MOVC A, @A+PC ; A← Contents of a ROM Location pointed by A+PC.

- The same instruction may operate on Internal or External ROM, depending upon the address and on the value of **EA** pin of 8051.
- If the address is in the range of 0000… 0FFFH, then **EA** pin will decide if it operates on Internal
- ROM or External ROM. IF **EA** = 0, External ROM else Internal ROM. If Address is 1000H and more, it will certainly be External ROM.



3.2 INSTRUCTION SETS OF 8051:

- An 8051 Instruction consists of an Opcode (short of Operation – Code) followed by Operand(s) of size Zero Byte, One Byte or Two Bytes.
- The Op-Code part of the instruction contains the Mnemonic, which specifies the type of operation to be performed. All Mnemonics or the Opcode part of the instruction are of One Byte size.
- Coming to the Operand part of the instruction, it defines the data being processed by the instructions. The operand can be any of the following:
- No Operand
- Data value
- I/O Port
- Memory Location

- CPU register

There can multiple operands and the format of instruction is as follows:

MNEMONICS DESTINATION OPERANDS, SOURCE OPERANDS

- A simple instruction consists of just the **opcode.**
- Other instructions may include one or more operands.
- Instruction can be one-byte instruction, which contains only opcode, or two-byte instructions, where the second byte is the operand or three byte instructions, where the operand makes up the second and third byte.

Based on the operation they perform, all the instructions in the 8051 Microcontroller Instruction Set are divided into five groups. They are:

- Data Transfer instructions
- Arithmetic instructions
- Logical instructions
- Boolean or Bit Manipulation
- Program Branching instructions

✓ This Bit-Processing group is also known as Boolean Variable Manipulation.

✓ Like 8085, some instruction has two operands. The first operand is the Destination, and the second operator is Source.

✓ In the following examples, you will get some notations. The notations are like:

Rn = any register from R0 to R7

Ri = Either R0 or R1

d8 = Any 8-bit immediate data (00H to FFH)

d16 = 16-bit immediate data

a8 = 8-bit address

Bit = 8-bit address of bit which is bit addressable

rel = 8-bit signed displacement. The range is -128 to 127. It is relative to the fir

1. DATA TRANSFER INSTRUCTION:

- Data transfer instructions move the content of one register to another. The

register the content of which is moved remains unchanged. If they have the suffix "X" (MOVX), the data is exchanged with external memory.

Or

- The Data Transfer Instructions are associated with transfer with data between registers or external program memory or external data memory.

- In this group, the instructions perform data transfer operations of the following types.
- Move the contents of a register Rn to A

1. MOV A,R2
2. MOV A,R7

- Move the contents of a register A to Rn

1. MOV R4,A
2. MOV R1,A

- Move an immediate 8 bit data to register A or to Rn or to a memory location(direct or indirect)

1. MOV A, #45H
2. MOV R6, #51H
3. MOV @R0, #0E8H
4. MOV DPTR, #0F5A2H
5. MOV DPTR, #5467H
6. MOV 30H, #44H

- Move the contents of a memory location to A or A to a memory location using direct and indirect addressing

1. MOV A, 65H
2. MOV A, @R0
3. MOV 45H, A
4. MOV @R1, A

- Move the contents of a memory location to Rn or Rn to a memory location using direct addressing

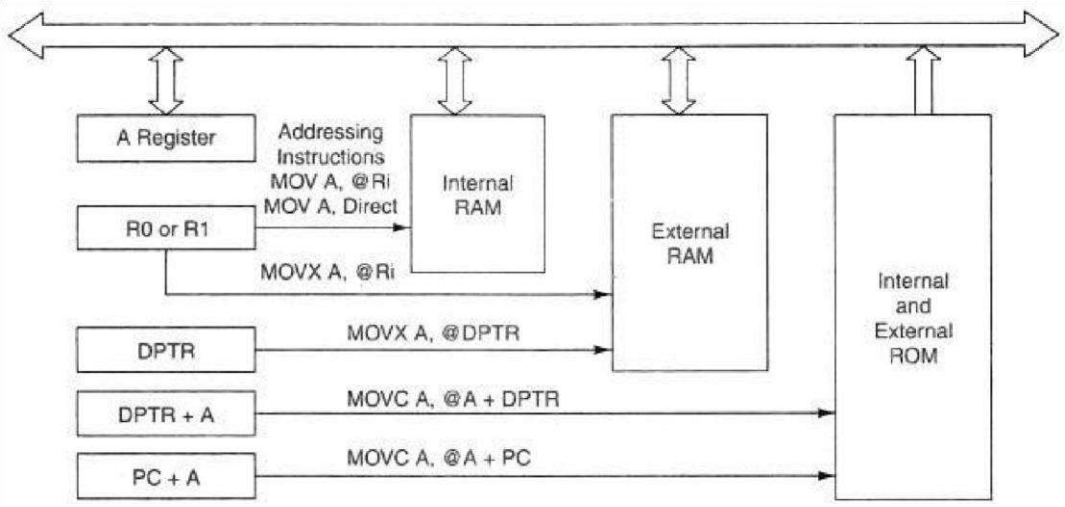
1. MOV R3, 65H
2. MOV 45H, R2

- Move the contents of memory location to another memory location using direct and indirect addressing

1. MOV 47H, 65H

2. MOV 45H, @R0

- Move the contents of an external memory to A or A to an external memory
  1. MOVX A,@R1
  2. MOVX @R0,A
  3. MOVX A,@DPTR
  4. MOVX@DPTR,A
- Move the contents of program memory to A
  1. MOVC A, @A+PC
  2. MOVC A, @A+DPTR



- **Push and Pop instructions**

	[SP]=07 (DEFAULT VALUE)	//CONTENT OF SP IS 07
MOV R6, #25H	[R6]=25H	//CONTENT OF R6 IS 25H
MOV R1, #12H	[R1]=12H	//CONTENT OF R1 IS 12H
MOV R4, #0F3H	[R4]=F3H	//CONTENT OF R4 IS F3H
PUSH 6	[SP]=08	[08]=[06]=25H //CONTENT OF 08 IS 25H
PUSH 1	[SP]=09	[09]=[01]=12H //CONTENT OF 09 IS 12H

PUSH 4	[SP]=0A	[0A]=[04]=F3H
	//CONTENT OF 0A IS F3H	
POP 6	[06]=[0A]=F3H	[SP]=09
	//CONTENT OF 06 IS F3H	
POP 1	[01]=[09]=12H [SP]=08	//CONTENT OF 01 IS 12H
POP 4	[04]=[08]=25H [SP]=07	//CONTENT OF 04 IS 25H

- **Exchange instructions**

  The content of source i.e. register, direct memory or indirect memory will be exchanged with the contents of destination i.e. accumulator.

  1. XCH A,R3
  2. XCH A,@R1
  3. XCH A,54h

- **Exchange digit.**

  Exchange the lower order nibble of Accumulator (A0-A3) with lower order nibble of the internal RAM location which is indirectly addressed by the register.

  1. XCHD A,@R1
  2. XCHD A,@R0

2. ARITHMETIC INSTRUCTIONS:

Using Arithmetic Instructions, we can perform addition, subtraction, multiplication and division. The arithmetic instructions also include increment by one, decrement by one and a special instruction called Decimal Adjust Accumulator.

Addition

In this group, we have instructions to

- Add the contents of A with immediate data with or without carry.
  1. ADD A, #45H
  2. ADDC A, #OB4H
- Add the contents of A with register Rn with or without carry.
  1. ADD A, R5
  2. ADDC A, R2
- Add the contents of A with contents of memory with or without carry using direct and indirect addressing

1. ADD A, 51H
2. ADDC A, 75H
3. ADD A, @R1
4. ADDC A, @R0
5.

- CY AC and OV flags will be affected by this operation.

Subtraction

In this group, we have instructions to

- Subtract the contents of A with immediate data with or without carry.
    1. SUBB A, #45H
    2. SUBB A, #OB4H
- Subtract the contents of A with register Rn with or without carry.
    1. SUBB A, R5
    2. SUBB A, R2
- Subtract the contents of A with contents of memory with or without carry using direct and indirect addressing
    1. SUBB A, 51H
    2. SUBB A, 75H
    3. SUBB A, @R1
    4. SUBB A, @R0

- CY AC and OV flags will be affected by this operation.

Multiplication

MUL AB.

- This instruction multiplies two 8 bit unsigned numbers which are stored in A and B register.
- After multiplication the lower byte of the result will be stored in accumulator and higher byte of result will be stored in B register.

Examples:

```
MOV A,#45H      ;[A]=45H
MOV B,#0F5H     ;[B]=F5H
MUL AB          ;[A] x [B] = 45 x F5 = 4209
                ;[A]=09H, [B]=42H
```

Division

DIV AB.

- This instruction divides the 8 bit unsigned number which is stored in A by the 8 bit unsigned number which is stored in B register.
- After division the result will be stored in accumulator and remainder will be stored in B register.

- **Examples:**

```
MOV A,#45H      ;[A]=0E8H
MOV B,#0F5H     ;[B]=1BH
DIV AB          ;[A] / [B] = E8 /1B = 08 H with remainder 10H
                ;[A] = 08H, [B]=10H
```

DAA (Decimal Adjust After Addition):

- When two BCD numbers are added, the answer is a non-BCD number.
- To get the result in BCD, we use DAA instruction after the addition. DAA works as follows.
  - If lower nibble is greater than 9 or auxiliary carry is 1, 6 is added to lower nibble.
  - If upper nibble is greater than 9 or carry is 1, 6 is added to upper nibble.

Examples 1:

```
MOV
A,#23H
MOV
R1,#55H
ADD         //
A,R1        [A]=78
DA A        //          no changes in the accumulator after
            [A]=78      DAA
```

Examples 2:

```
MOV
A,#53H
MOV
R1,#58H
ADD         //
```

A,R1	[A]=ABH
DA A	// [A] =11, C=1.
	Answer is 111. **Accumulator data is changed after DAA**

Increment:

Increments the operand by one.

INC A
INC Rn
INC DIRECT
INC @Ri
INC DPTR

- INC increments the value of source by 1.
- If the initial value of register is FFH, incrementing the value will cause it to reset to 0.
- The Carry Flag is not set when the value "rolls over" from 255 to 0.
- In the case of "INC DPTR", the value two-byte unsigned integer value of DPTR is incremented.
- If the initial value of DPTR is FFFFH, incrementing the value will cause it to reset to 0.

Decrement:

Decrements the operand by one.

DEC A
DEC Rn
DEC DIRECT
DEC @Ri

- DEC decrements the value of source by 1.
- If the initial value of is 0, decrementing the value will cause it to reset to FFH.
- The Carry Flag is not set when the value "rolls over" from 0 to FFH.

3. LOGICAL INSTRUCTIONS:

- The Logical Instructions, which perform logical operations like AND, OR, XOR,

NOT, Rotate, Clear and Swap.

- Logical Instruction are performed on Bytes of data on a bit-by-bit basis.

Logical AND

- ANL destination, source: ANL does a bitwise "AND" operation between source and destination, leaving the resulting value in destination.
- The value in source is not affected.
- "AND" instruction logically AND the bits of source and destination.

ANL A, #DATA ANL A, Rn
ANL A, DIRECT ANL A, @Ri
ANL DIRECT, A ANL DIRECT, #DATA

Logical OR

- **ORL destination, source:** ORL does a bitwise "OR" operation between source and destination, leaving the resulting value in destination.
- The value in source is not affected. “OR” instruction logically OR the bits of source and destination.

ORL A, #DATA ORL A, Rn
ORL A, DIRECT ORL A, @Ri
ORL DIRECT, A ORL DIRECT, #DATA

Logical Ex-OR

- **XRL destination, source:** XRL does a bitwise "EX-OR" operation between source and destination, leaving the resulting value in destination.
- The value in source is not affected. "XRL" instruction logically EX-OR the bits of source and destination.

XRL A, #DATA XRL A, Rn
XRL A, DIRECT XRL A, @Ri
XRL DIRECT, A XRL DIRECT, #DATA

Logical NOT

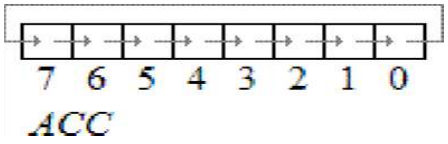
- **CPL** complements operand, leaving the result in operand.
- If operand is a single bit then the state of the bit will be reversed.
- If operand is the Accumulator then all the bits in the Accumulator will be reversed.
- CPL A, CPL C, CPL bit address

- **SWAP A** – Swap the upper nibble and lower nibble of A.

Rotate Instructions

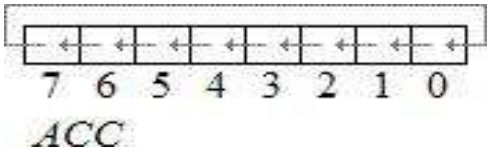
RRA

- This instruction is rotate right the accumulator.
- Each bit is shifted one location to the right, with bit 0 going to bit 7.



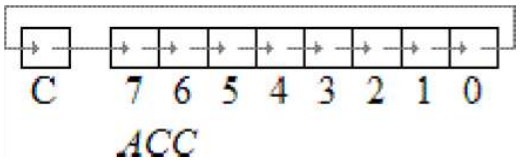
RLA

Rotate left the accumulator. Each bit is shifted one location to the left, with bit 7 going to bit 0



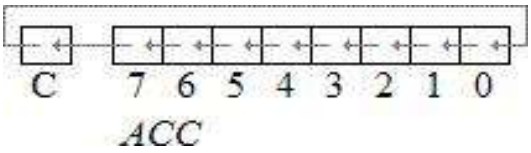
RRC A

Rotate right through the carry. Each bit is shifted one location to the right, with bit 0 going into the carry bit in the PSW, while the carry was at goes into bit 7



RLC A

Rotate left through the carry. Each bit is shifted one location to the left, with bit 7 going into the carry bit in the PSW, while the carry goes into bit 0.



4. BOOLEAN OR BIT MANIPULATION INSTRUCTIONS:

- As the name suggests, Boolean or Bit Manipulation Instructions will deal with bit

variables.

- We know that there is a special bit-addressable area in the RAM and some of the Special Function Registers (SFRs) are also bit addressable.
- 8051 has 128 bit addressable memory. Bit addressable SFRs and bit addressable PORT pins. It is possible to perform following bit wise operations for these bit addressable locations.

1. LOGICAL AND

a. **ANL C,BIT(BIT ADDRESS)** ; logically and' carry and content of bit address, Store result in carry

b. **ANL C, /BIT;** ; logically and' carry and complement of Content of bit address, store result in carry

2. LOGICAL OR

a. **ORL C,BIT(BIT ADDRESS)** ; logically or' carry and content of bit address, Store result in carry

b. **ORL C, /BIT** ; logically or' carry and complement of Content of bit address, store result in carry

3. CLR bit

a. **CLR bit** ; content of bit address specified will be cleared.

b. **CLR C** ; content of carry will be cleared.

4. CPL bit

a. **CPL bit** ; content of bit address specified will be complemented.

b. **CPL C** ; content of carry will be complemented.

5. PROGRAM BRANCHING (JUMP) INSTRUCTIONS:

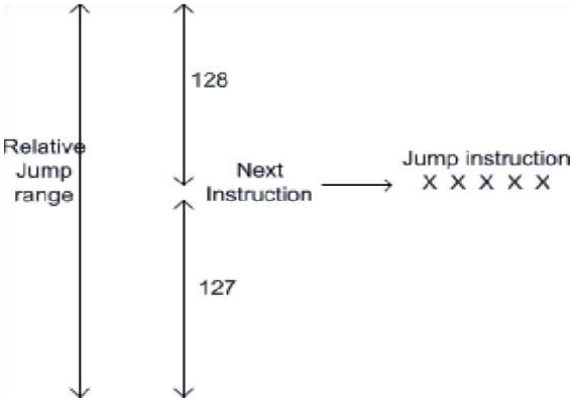
Jump and Call Program Range

There are 3 types of jump instructions. They are: 1. Relative Jump

2. Short Absolute Jump
3. Long Absolute Jump

Relative Jump

- Jump that replaces the PC (program counter) content with a new address that is greater than (the address following the jump instruction by 127 or less) or less than (the address following the jump by 128 or less) is called a relative jump.
- Schematically, the relative jump can be shown as follows: -



The advantages of the relative jump are as follows:-

- Only 1 byte of jump address needs to be specified in the 2's complement form, ie. For jumping ahead, the range is 0 to 127 and for jumping back, the range is -1 to -128.
- Specifying only one byte reduces the size of the instruction and speeds up program execution.
- The program with relative jumps can be relocated without reassembling to generate absolute jump addresses.

Disadvantages of the absolute jump: -

- Short jump range (-128 to 127 from the instruction following the jump instruction)

Instructions that use Relative Jump

- SJMP <relative address>; this is unconditional jump
- The remaining relative jumps are conditional jumps
  JC <relative address>
  JNC <relative address>
  JB bit, <relative address>
  JNB bit, <relative address>
  JBC bit, <relative address>
  CJNE <destination byte>, <source byte>, <relative address>

DJNZ <byte>, <relative address>
JZ <relative address> JNZ <relative address>

Short Absolute Jump

- In this case only 11bits of the absolute jump address are needed.
- The absolute jump address is calculated in the following manner.
- In 8051, 64 Kbyte of program memory space is divided into 32 pages of 2 Kbyte each.
- The hexadecimal addresses of the pages are given as follows:-

Page (Hex)	Address (Hex)
00	0000 - 07FF
01	0800 - 0FFF
02	1000 - 17FF
03	1800 - 1FFF
.	
.	
1E	F000 - F7FF
1F	F800 - FFFF

- It can be seen that the upper 5bits of the program counter (PC) hold the page number and the lower 11bits of the PC hold the address within that page. Thus, an absolute address is formed by taking page numbers of the instruction (from the program counter) following the jump and attaching the specified 11bits to it to form the 16-bit address.

Advantage:

- The instruction length becomes 2 bytes.
- **Example** of short absolute jump: - ACALL <address 11>

  AJMP <address 11>

Long Absolute Jump/Call

- Applications that need to access the entire program memory from 0000H to FFFFH use long absolute jump.
- Since the absolute address has to be specified in the op-code, the instruction length is 3 bytes (except for JMP @ A+DPTR). This jump is not re-locatable.

- **Example: -**

  LCALL <address 16>

LJMP <address 16>
JMP @A+DPTR

Another classification of jump instructions is

1. Unconditional Jump
2. Conditional Jump

1. unconditional jump:

It is a jump in which control is transferred unconditionally to the target location.

✓ **LJMP (long jump).**

- This is a 3-byte instruction. First byte is the op-code and second and third bytes represent the 16-bit target address which is any memory location from 0000 to FFFFH **e.g.: LJMP 3000H**

✓ **AJMP:**

- This causes unconditional branch to the indicated address, by loading the 11 bit address to 0 -10 bits of the program counter. The destination must be therefore within the same 2K blocks.

✓ **SJMP (short jump):**

- This is a 2-byte instruction. First byte is the op-code and second byte is the relative target address, 00 to FFH (forward +127 and backward -128 bytes from the current PC value). To calculate the target address of a short jump, the second byte is added to the PC value which is address of the instruction immediately below the jump.

2. Conditional Jump instructions.

JBC	Jump if bit = 1 and clear bit
JNB	Jump if bit = 0
JB	Jump if bit = 1
JNC	Jump if CY = 0
JC	Jump if CY = 1
CJNE reg, #data	Jump if byte ≠ #data
CJNE A, byte	Jump if A ≠ byte
DJNZ	Decrement and Jump if A ≠ 0

JNZ Jump if A ≠ 0

JZ Jump if A = 0

- All conditional jumps are short jumps.

Bit level jump instructions:

- Bit level JUMP instructions will check the conditions of the bit and if condition is true, it jumps to the address specified in the instruction. All the bit jumps are relative jumps.

JB bit, rel ; jump if the direct bit is set to the relative address specified.

JNB bit, rel ; jump if the direct bit is clear to the relative address specified.

JBC bit, rel ; jump if the direct bit is set to the relative address specified and then clear the bit

UNIT-4: 8051 ASSEMBLY LANGUAGE PROGRAMMING TOOLS

4.1 PROGRAM USING JUMP, LOOP AND CALL INSTRUCTION IN 8051:

In the sequence of instructions to be executed, it is often necessary to transfer program control to a different location. There are many instructions in the 8051 to achieve this goal. This chapter covers the control transfer instructions available in 8051 Assembly Language.

LOOP AND JUMP INSTRUCTIONS:

- The various types of control transfer instructions in assembly language include conditional and unconditional jumps and call instructions.
- The flow of a program will be proceed sequentially from instruction to instruction, unless a control transfer instruction is executed.
- The looping action will be performed using an instruction DJNZ which decrements a counter and jumps to the top of the loop if the value counter is not zero.
- Jump conditionally instructions based on the value of the carry flag, the accumulator, or bits of the I/O port.
- Unconditional jumps can be long or short, depending upon the relative value of the target address.

What is looping in the 8051 Assembly Language Programming?

- Loop is defined as repeating a sequence of instructions a certain number of times.
- The loop is one of most widely used action that any microprocessor performs. In the 8051, "DJNZ reg, label" performed the loop actions.
- In "DJNZ reg, label" instruction, the register is decrement, if it is not zero and jumps to the target address referred to by the label.
- The register is loaded with the counter of the number of repetitions prior to the start of the loop.
- In "DJNZ reg, label" instruction, the register is decrement and the decision to jump are combined into a single instruction.

- **<u>Example 1</u>:**
- **write a program to clear Accumulator [A], then Add 5 to the accumulator 20 times:**

This program adds value 3 to the Accumulator 20 times.

```
        MOV A,#0          ;Accumulator=0, Clear Accumulator
        MOV R2,#20        ;Load counter R2=20
AGAIN:  ADD A,#05         ;Add 05 to Accumulator
        DJNZ R2,AGAIN     ;Repeat Until R2=0 [20 times]
        MOV R5,A          ;Save Accumulator in R5
```

- The R2 register is used as a counter.
- The counter R2 is first set to 20.
- In each iteration, the instruction DJNZ decrements R2 and checks its value.
- If R2 is not zero, it jumps to the target address associated with label "AGAIN".
- This looping action continues until R2 becomes zero.
- After R2 becomes zero, it falls through the loop and executes the instruction immediately below it, in this case the "MOV R5, A" instruction.
- In the DJNZ instruction that the registers can be any of R0-R7. The counter can also be a RAM location.

Example 2:

Write a program to multiply 25 by 10 using the technique of repeated addition.

Solution:

Multiplication can be achieved by adding the multiplicand repeatedly, as many times as the multiplier.

E.g. 25x10=250 (FAH)

25+25+25+25+25+25+25+25+25+25=250

```
        MOV A,#0          ;Accumulator=0, Clear Accumulator
        MOV R2,#10        ;the multiplier is placed in R2
AGAIN:  ADD A,#25         ;Add the multiplicand to the Accumulator
        DJNZ R2,AGAIN     ;Repeat Until R2=0 (10 times)
        MOV R5,A          ;Save Accumulator in R5
```

The maximum number of times that a loop can be repeated is only 256 since the count register R2 is an 8-bit register and can hold only numbers from 0 to 255.

Example 3:

Write a program to add the first ten natural numbers.

Solution:

The first 10 natural numbers are 1, 2, 3…10

```
        MOV A,#0          ;Accumulator=0, Clear Accumulator
        MOV R2,#10        ; Load counter value in R2
        MOV R0, #0        ; initialize R0 to zero.
AGAIN: INC R0             ; increment R0 to hold the natural numbers
        ADD A,R0          ;Add first number to accumulator
        DJNZ R2,AGAIN     ;Repeat Until R2=0 (10 times)
        MOV 46H,A         ;Save the result (37H)in RAM location 46H
```

The maximum number of times that a loop can be repeated is only 256 since the count register R2 is an 8-bit register and can hold only numbers from 0 to 255.

<u>Loop Inside a Loop [NESTED LOOP]:</u>

The maximum count is 256. But if we want to repeat an action more times than 256, we use a loop inside a loop, which is called ***nested loop***. In nested loop, we use two registers to hold the count.

Example 4:

Write a program to load the accumulator with the value 66H, and complement the accumulator ACC 600 times.

Solution:

700 is larger than 255 [the maximum capacity of any register], we use two registers to hold the count. We are using R2, R3 for the count.

```
        MOV A,#66H        ;Accumulator A=66H
        MOV R3,#10        ;R3=10, the outer loop count
 NEXT: MOV R2,#60         ;R2=60, the inner loop count
AGAIN: CPL A              ;Complement A register
        DJNZ R2,AGAIN     ;Repeat inner loop 60 times
```

```
DJNZ R3,NEXT          ;Repeat outer loop 10, so 10X60=600 times CPL A run
```

- R2 is used to keep the inner loop count.
- In the instruction "DJNZ R2, AGAIN", whenever R2 becomes zero, it falls through and "DJNZ R3, NEXT" is executed.
- This instruction forces the CPU to load R2 with the count 60 and the inner loop starts again.
- This process will continue until R3 becomes zero and the outer loop is finished.

8051 Conditional Jump Instructions:

Instruction	Action
JZ	Jump if A= 0
JNZ	Jump if A≠ 0
DJNZ	Decrement and jump if register ≠ 0
CJNE A, data	Jump if A ≠ data
CJNE reg, #data	Jump if byte ≠ #data
JC	Jump if CY=1
JNC	Jump if CY=0
JB	Jump if bit =1
JNB	Jump if bit=0
JBC	Jump if bit= 1 & clear bit

Example 5:

In this instruction the content of register A is checked. If it is zero, it jumps to the target address.

```
MOV A,R0      ;A = R0
JZ OVER       ;Jump if accumulator A = 0
MOV A,R1      ;A = R1
JZ OVER       ;Jump if accumulator A = 0
.....
OVER:
```

- Either R0 or R1 is zero, it jumps to the label OVER. The JZ instruction can be used only for register accumulator A.

- It can used only check to see whether the Accumulator is zero. It does not apply to any other register.

Example 6:
Write a program in which if R4 register contains the value 0. Then put 55H in R4 register.

Solution:

```
        MOV A,R4        ;Copy R4 to accumulator A
        JNZ NEXT        ;Jump if accumulator is not zero
        MOV R4,#55H     ;Put value 55H into register R4
NEXT:   .........
```

JNC [Jump if No Carry i.e. Jumps if Carry Flag = 0]:

- In JNC instruction, the carry flag bit in the flag [PSW] register is used to make the decision whether to jump.
- In executing "JNC label", the processor looks are the carry flag to see it if is raised carry flag CY = 1, if it is not, the CPU starts to fetch and execute instructions from the address of the label.
- If carry flag CY=1, it will not jump but will execute the next instruction below JNC.

JC [Jump if Carry i.e. Jumps if Carry Flag = 1]:

- In JC instruction, if carry flag CY=1, it jumps to the target address.
- JB [Jump if bit is high]:
- JNB [Jump if bit is low]:

Example 6:
Write a program to find the sum of the values 78H, F4H and E1H. Put the sum in registers R0 [low byte] and R5 [high byte]:

Solution:

```
    MOV A,#0        ;Clear accumulator A=0
    MOV R5,A        ;Clear R5
```

```
ADD A,#78H              ;Accumulator A=0+78H=78H
JNC HERE_1              ;If No carry, add next number
INC R5                  ;If carry flag CY=1, increment R5
HERE_1:ADD A,#0F4H      ;A=78H+F4=6C and carry flag CY=1,as [78H+F4=16C]
JNC HERE_2              ;Jump if Carry flag CY=0
INC R5                  ;If carry flag CY=1,then increment R5, i.e. R5=1
HERE_2: ADD A,#0E1H     ;A=6C+E1=4D and carry flag CY=1,as [6C+E1=14D]
JNC OVER                ;Jump if carry flag CY=0
INC R5                  ;If carry flag CY=1, increment R5
OVER: MOV R0,A          ;Now R0=4D, and R5=02
END
```

Example 7:

Write a program to multiply the numbers 0ECH by 25H using the technique of repeated addition.

Solution:

```
        MOV     R1, #0          ;R1=0, this is the register to store the MSB
        MOV     A, #0           ;clear accumulator
        MOV     R0, #25H        ; the multiplier is placed in R0
AGAIN:ADD       A, #0ECH        ; add the multiplicand to the accumulator
        JNC     HERE            ; if no carry, then repeat the addition
        INC     R1              ; increment R1 for each carry generated
        DJNZ    R0,AGAIN        ;Repeat Until R0=0
        MOV     R1, #0          ;R1=0, this is the register to store the MSB
        MOV     A, #0           ;clear accumulator
        MOV     R0, #25H        ; the multiplier is placed in R0
AGAIN:ADD       A, #0ECH        ; add the multiplicand to the accumulator
        JNC     HERE            ; if no carry, then repeat the addition
        INC     R1              ; increment R1 for each carry generated
        DJNZ    R0,AGAIN        ;Repeat Until R0=0
```

```
MOV     R0, A          ; the LSB of the product is moved to R0
                       ; the MSB of the product is in R1
                       ; now R1=22H and R0=1CH
```

- They are all 2-byte instructions. In these instructions, the first byte is the opcode and the second byte is the relative address.
- The target address is relative to the value of the program counter (PC).
- To calculate the target address, the second byte is added to the PC (program counter) of the instruction immediately below the jump.

Example 8:
WAP to verify the jump forward address calculation.

LINE	PROGRAM COUNTER	OPCODE	MNEMONIC OPERAND
01	0000		ORG 0000
02	0000	7800	MOV R0,#01
03	0002	7455	MOV A,#66H
04	0004	6003	JZ NEXT
05	0006	08	INC R1
06	0007	04	AGAIN: INC A
07	0008	04	INC A
08	0009	2477	NEXT: ADD A,#77H
09	000B	5005	JNC OVER
10	000D	E4	CLR A
11	000E	F8	MOV R3,A
12	000F	F9	MOV R2,A
13	0010	FA	MOV R1,A
14	0011	FB	MOV R0,A
15	0012	2B	OVER ADD A,R3

16	0013	50F2	JNC AGAIN
17	0015	80FE	SJMP HERE
18	0017		END

Answer:

- The target address for a forward jump is calculated by adding the program counter (PC) of the following instruction to the second byte of the short jump instruction, which is called the relative address.
- JZ and JNC instructions both jump forward. In line 04, the instruction "JZ NEXT" has opcode of '60' and operand of '03' at the program counter (PC) address of '0004' and '0005'.
- The 03 is the relative address, relative to the address of the next instruction "INC R0" at program counter address of '0006'.
- By adding 0006 to 03, the target address so the label NEXT, 0009 is generated, where the jump will take place at NEXT: ADD A, #77H.
- In the same way, at line 09, the "JNC OVER instruction has opcode and operand of '50' and '05', where '50' is the opcode and '05' is the relative address.
- So 05 is added to 000D, the address if instruction "CLR A", resulting in 0012, the address of the label OVER of mnemonic operand 'OVER ADD A, R3'.

Backward Jumps Verification in Example 8:

- "JNC AGAIN" has opcode 50 and relative address F2H.
- This relative address F2H is added to program counter address 0015 or 15H, the address of the instruction below the jump, resulting 15H + F2H = 107, after dropping the carry, 15H + F2H = 07 (the carry is dropped).
- 0007 is the PC address of the 'AGAIN: INC A' mnemonic operand.
- In the same way, 'SJMP HERE' which has 80 and FE for the opcode and relative address.
- The program counter of the following instruction 0017H is added to FEH, the relative address, to get 0015H, the address of the HERE label (17H+FEH = 15H).
- FEH is -2 and 17H+ (-2) =15H.

Backward Jump Target Address Calculations:

- The displacement value is positive number in the case of forward jump, i.e. between 0 to 127, 00 to 7F in hex.
- The displacement value is negative in the case of backward jump, i.e. 0 to -128.
- The short jump, SJMP is forward or backward, the address of the target address can never be more than -128 to +127 bytes from the address associated with the instruction below the SJMP. The assembler will generate an error of out of range if any attempt is made to violate this rule.

CALL INSTRUCTIONS IN 8051 MICROCONTROLLER

- CALL instruction is another control transfer instruction.
- CALL instruction is used to call a subroutine.
- Need to performed the tasks frequently subroutines are used. Using a subroutine make a program more structured and helps in reducing memory space.
- There are two instructions for CALL in the 8051 programming. LCALL (Long Call) and ACALL (Absolute call).

LCALL (Long Call):

- LCALL is a 3-byte instruction. This first byte is the opcode and the second and third bytes are used for the address for the target subroutine.
- Anywhere Within the 64K byte address space of the 8051, LCALL can be used to call subroutines.
- The 8051 microcontroller knows where to come back to, it automatically saves on the stack the address of the instruction immediately below the LCALL.
- When a subroutine is called, the control is transferred to that subroutine, and the processor saves the program counter (PC) on the stack and begins to fetch instructions from the new location.
- After finishing execution of the subroutine, the instruction RET (Return) transfers control back to the caller.
- RET (Return) needs by every subroutine as the last instruction.

Example 1:

Write a program to toggle all the bits of port 1 by sending it the values 55H and AAH continuously. Put a time delay between each issuing of data to port 1.

Answer:

```
        ORG     0
```

```
BACK    MOV     A, #55H          ; load A with 55
        MOV     P1,A             ; send 55H to port 1
        LCALL   DELAY            ; time delay
        MOV     A, #0AAH         ; load a with AA (in hex)
        MOV     P1,A             ; send AAH to port 1
        LCALL   DELAY
        SJMP    BACK             ; keep doing this indefinitely
; ----------  this is the delay subroutine
        ORG     300H             ; put time delay at address 300H
DELAY:  MOV     R5, #0FFH        ; R5=255(FF in hex), the counter
AGAIN:  DJNZ    R5, AGAIN        ; stay here until R5 becomes 0
        RET                      ; return to caller ( when R5=0)
        END                      ; end of asm file
```

In the above example, following point should be noted.

- When the first "LCALL DELAY" is executed, the address of the instruction right below it, "MOV A, #0AAH", is pushed onto the stack, and the 8051 starts to execute instructions at address 300H.
- In the DELAY subroutine, first the counter R4 is set to 255 [FFH in hex]. 255 means the loop will be repeated 256 times. When R4 becomes zero, control falls to the RET instruction which pops address from the stack into the program counter and resumes executing the instructions after the CALL.
- The amount of time delay depends on the frequency of the 8051. We can increase the time delay by using nested loop **as shown below.**

```
DELAY:
        MOV     R4,#255          ; nested loop delay
NEXT:   MOV     R5, #255         ; R4= 255(FF in hex)
AGAIN:  DJNZ    R5, AGAIN        ;stay here until R5 becomes 0
        DJNZ    R4, NEXT         ;decrement R4
                                 ; keep loading R5 until R4=0
        RET                      ; return ( when R4=0 )
```

Rewriting <u>Example 1</u> more efficiently:

```
        ORG     0
```

```
        MOV     A, #55H         ; load A with 55H
BACK    MOV     P1, A           ; issue value in register A to port 1
        LCALL   DELAY           ; time delay
        CPL A                   ; complement register A i.e.55H becomes AAH
        SJMP    BACK            ; keep doing this indefinitely
                                ; ----------  this is the delay subroutine DELAY:
MOV     R5, #0FFH       ; R5=255(FF in hex), the counter
AGAIN:  DJNZ    R5, AGAIN       ; stay here until R5 becomes 0
        RET                     ; return to caller (when R5=0)
        END                     ; end of asm file
```

- The register accumulator A is loaded with 55H.
- By complementing 55H, we have AAH.
- In binary 55H=0101 0101. In binary AAH=1010 1010. So by complementing 0101 0101 (55H) we have 1010 1010 (AAH) and by complementing 1010 1010(AAH) we have again 0101 0101(55H).

Importance of Stack and the CALL Instruction:

To understand the importance of stack in microcontroller, let's examine the contents of the stack in microcontroller.

Example 2:
WAP Analyze the stack contents after the execution of the first LCALL.

```
001     0000                    ORG     0
002     0000    7455    BACK:   MOV     A, #55H     ; load A with 55H
003     0002    F590            MOV     P1, A       ; send 55H to port 1
004     0004    120300          LCALL   DELAY       ; time delay
005     0007    74AA            MOV     A, #0AAH    ; load A with AAH
006     0009    F590            MOV     P1, A       ; send AAH to port 1
007     000B    120300          LCALL   DELAY
008     0000    80F0            SJMP    BACK        ; keep doing this
009     0010
010     0010    ;---------- this is the delay subroutine
011     0300                    ORG 300H
012     0300            DELAY:
```

```
013    0300    7DFF              MOV     R5, #0FFH   ; R5=255
014    0302    DDFE    AGAIN:    DJNZ    R5, AGAIN   ; stay here
015    0304    22                RET                 ; return to caller
016    0305                      END                 ; end of asm file
```

- When the first LCALL is executed, the address of the instruction "MOV A, #0AAH" is saved on the stack.
- The low byte goes first and the high byte is last.
- The last instruction of the called instruction must be a RET instruction which directs the CPU to POP the top bytes of the stack into the program counter (PC) and resume executing at address 07.
- The stack frame after the first LCALL is given below.

```
0A
09          00
08          07
Stack Pointer= 09
```

PUSH and POP instructions in subroutines:

- The stack keeps track of where the CPU should return after completing the subroutine, where ever a subroutine is called.
- The number of PUSH and POP instructions must always match in any called subroutine.
- For every PUSH instruction executed, there is a POP instruction to be executed also.

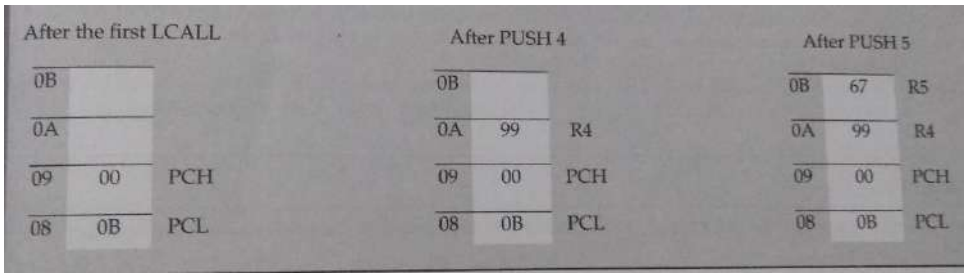
Example 2:
WAP analyze the stack for the first LCALL instruction.

```
01    0000                    ORG     0
02    0000    7455    BACK:   MOV     A, #55H     ; load A with 55H
03    0002    F590            MOV     P1, A       ; send 55H to port 1
04    0004    7C99            MOV     R4, #99H
05    0006    7D67            MOV     R5, #67H
06    0008    120300          LCALL DELAY         ; time delay
07    000B    74AA            MOV     A, #0AAH    ; load A with AA
```

```
08    000D    F590              MOV     P1, A       ; send AAH to port 1
09    000F    120300            LCALL   DELAY
10    0012    80EC              SJMP    BACK        ; keep doing this
11    0014          ;---------- this is the delay subroutine
12    0300                      ORG     300H
13    0300    C004    DELAY:    PUSH    4           ; PUSH R4
14    0302    C005              PUSH    5           ; PUSH R5
15    0304    7CFF              MOV     R4, #0FFH   ; R4=FFH ;
16    0306    7DFF    NEXT:     MOV     R5, #0FFH   ; R5=255H ;
17    0308    DDFE    AGAIN:    DJNZ    R5 AGAIN    ; stay here
18    030A    DCFA              DJNZ    R4, NEXT
19    030C    D005              POP     5           ; POP into R5
20    030E    D004              POP     4           ; POP into R4
21    0310    22                RET                 ; return to caller
22                              END                 ; end of asm file
```

Answer:

First notice that for the PUSH and POP instructions we must specify the direct address of the register being pushed or popped. Here is the stack frame.



CALLING SUBROUTINES:

- In assembly language programming it is common to have one main program and many subroutines that are called from the main program.
- This allow us to make each subroutine into a separate module.
- Each module can be tested separately and then brought together with the main program.
- More importantly, in a large program the modules can be assigned to different programmers in order to shorten development time.

- It needs to be emphasized that in using LCALL, the target address of the subroutine can be anywhere within the 64K-byte memory space of the 8051.this is not the case for other instructions.

```
; MAIN program calling subroutines
                ORG 0
MAIN:           LCALL           subroutine_1
                LCALL           subroutine_2
                LCALL           subroutine_3
HERE:           SJMP            HERE
; ---------------------------------End of MAIN
;
subroutine_1: ....
                ....
                RET
;---------------------------------End of subroutine 1
;
subroutine_2: ....
                ....
                RET
;---------------------------------End of subroutine 2
subroutine_3: ....
                ....
                RET
 ; ---------------------------------End of subroutine 3
                END             ; End of asm file
```

Absolute Call (ACALL):

- ACALL is a 2 byte instruction as compared to LCALL.
- LCALL is a 3 byte instruction.
- The target address of the subroutine must be within 2K bytes address because only 11 bits of the 2 bytes are used for the address.
- There is no difference between ACALL and LCALL in terms of saving a program counter on the stack or the function of the RET instruction.

- The only difference is that the target address for LCALL can be anywhere within the 64K byte address space of the 8051.
- The target address of the ACALL must be within 2K byte range.
- 8051 marketed by different companies, on-chip ROM is as low as 1K bytes.
- In this case, the use of ACALL instead of LCALL can save the number of bytes of program ROM space.

TIME DELAY CALCULATION IN 8051:

Machine Cycle in 8051 microcontrollers:

- The CPU takes a certain number of clock cycles to execute an instruction. These clock cycles are referred to as **machine cycles.**
- The length of the machine cycle depends on the frequency of the crystal oscillator connected to the 8051 system.
- The crystal oscillator, along with the on-chip circuitry, provide the clock source for the 8051 CPU.
- The crystal oscillator frequency can vary from 4MHz to 30MHz.
- To make the 8051 system compatible with the serial port of the personal computer PC, 11.0592MHz crystal oscillators is used.
- In the 8051, one machine cycle lasts 12 oscillator periods.
- So to calculate the machine cycle, we take 1/12 of the crystal frequency, then take the inverse of it results in time period i.e. frequency = 1/time period.

Example 1:

To find the time period of the machine cycle in each case for the following crystal frequency of different 8051 based systems: 11.0592 MHz, 16 MHz, and 20 MHz

Answer:

11.0592 MHz:
11.0592/12 = 921.6 KHz
Machine cycle = 1/921.6 KHz = 1.085μs

16 MHz:
16MHz/12 = 1.333 MHz
Machine cycle = 1/1.333 MHz = 0.75μs

20MHz:
20MHz/12 = 1.66 MHz
Machine Cycle = 1/1.66 MHz = 0.60μs

Example 2:

To find how long it takes to execute each of the following instructions, for a crystal frequency of 11.0592 MHz The machine cycle of a system of 11.0592 MHz is 1.085 μs.

INSTRUCTION	MACHINE CYCLE	TIME TO EXECUTE
MOVR2, #55H	1	1x1.085 us = 1.085μs
DEC R2	1	1x1.085 us = 1.085μs
DJNZ R2, target	2	2x1.085 us = 2.17μs
LJMP	2	2x1.085 us = 2.17μs
SJMP	2	2x1.085 us = 2.17μs
NOP	1	1x1.085 us = 1.085μs
MUL AB	4	4x1.085 us = 4.34μs

Calculate Exact Time Delay in 8051 microcontroller:

- The delay subroutine consists of 2 parts:
  - Setting a counter and
  - Creating a loop.

Example 3:
To find the size of the delay if the crystal frequency of 11.0592 MHz is connected.

```
        MOV     A, #55H          ; load A with 55H
AGAIN:  MOV     P1, A            ; issue value in register A to port 1
        ACALL   DELAY            ; time delay
        CPL     A                ; complement register A
        ASJMP   AGAIN            ; keep doing this indefinitely
; ---------Time Delay
DELAY:  MOV     R3, #225         ; load R3 with 255
HERE:   DJNZ    R3, HERE         ; stay here until R3 become 0
        RET                      ; return to caller
```

Answer:

- We have the following machine cycles for each instruction of the DELAY subroutine.

```
DELAY: MOV R2, #255          Machine Cycle = 1
HERE: DJNZ R2, HERE          Machine Cycle = 2
RET                          Machine Cycle = 1
```

- Therefore, we have a time delay of [(255 x 2) + 1 + 1] x 1.085 us = 555.52 us
- Very often we used to calculate the time delay based on the instructions inside the loop and ignore the clock cycles associated with the instructions outside the loop.

NOP Instruction:

NOP instruction is used to increase the delay in the loop. NOP means "No Operation" simply wastes time.

Loop Inside a Loop Delay:

This method is used to get a large delay i.e. is used to loop inside a loop, which is also called a **nested loop**.

Example 4:

To find the time delay for the following subroutine with 11.0592 MHz crystal frequency is connected to the 8051 system.

```
DELAY: MOV    R2, #255          Machine Cycle = 1
HERE:  NOP                      Machine Cycle = 1
       NOP                      Machine Cycle = 1
       NOP                      Machine Cycle = 1
       NOP                      Machine Cycle = 1
       DJNZ   R2, HERE          Machine Cycle = 2
       RET                      1
```

- The time delay inside the HERE loop is [255(1+1+1+1+2)] x 1.085 us = 1660.05µs
- The time delay of the two instructions outside the loop is:

= [1660.05 us+1+1)] x1.085 µs
= 1803.32425µs

Example 5:

For a crystal frequency of 11.0592 MHz, let's find the time delay in the following subroutine. The machine cycle is 1.085 us.

Answer:

DELAY: MOV R2, #200	Machine Cycle = 1
AGAIN: MOV R3, #250	Machine Cycle = 1
HERE: NOP	Machine Cycle = 1
NOP	Machine Cycle = 1
DJNZ R3, HERE	Machine Cycle = 2
DJNZ R2, AGAIN	Machine Cycle = 2
RET	Machine Cycle = 1

'HERE' Loop Calculations:
1+1+2, so [(1+1+2) x250] x 1.085μs = 1085μs.

'AGAIN' Loop Calculations:

- In this loop "MOV R3, #250" and "DJNZ R2, AGAIN" at the beginning and end of the AGAIN loop add [(1+2) x200] x1.085 us = 651us to the time delay.
- The AGAIN loop repeats the HERE loop 200 times so 200x1085μs = 217000μs.
- As a result the total time delay will be 217000μs + 651μs = 217651μs or 217.651 milliseconds.
- The time is approximate as we have ignored the first and the last instructions in the subroutine i.e. DELAY: "MOV R2, #200" and "RET".

4.2 I/O PORT PROGRAMMING:

- The 8051 family members come in packages such as PDIP (Dual in-line package), QFP (Quad flat Package) and LLC (Leadless Chip Carrier).
- All packages have 40 pins that are dedicated for various functions such as I/O, RD', WR', address, data, and interrupts.
- 20 pin version of 8051 also provided by some companies with a reduced number of I/O ports.
- In 8051 the total of 32 pins are set aside for the four ports P0, P1, P2 and P3. Each port has 8 pins.
- The rest of the pins are designated as Vcc, GND, XTAL1, XTAL2, RST, and EA'.
- These pins must be connected in order for the system to work.

RESET Value of Some 8051 Registers:

REGISTER	**RESET VALUE**
Program Counter (PC)	0000

Accumulator (ACC)	0000
B	0000
Program Status Word (PSW)	0000
Stack Pointer (SP)	0007
Data Pointer (DPTR)	0000

Functions of Input / Output I/O Port Pins:

- There are four ports in 8051 microcontroller named as P0, P1, P2 and P3.
- Each port has 8 pins. For Port 0 8 pins names as P0.0, P0.1, P0.2, P0.3, P0.4, P0.5, P0.6, and P0.7.
- All the ports upon RESET are ready to be configured or used as output.
- To use all these ports as an input port, it must be programmed.

PORT 0:

- Port 0 has 8 pins P0.0, P0.1, P0.2, P0.3, P0.4, P0.5, P0.6, and P0.7.
- This port can be used for input or output.
- Each pin of the port 0 must be connected externally to a 10K Ohm pull-up resistor, to use Port 0 as both input or output ports.
- This is because the Port 0 (P0) is an open drain.
- This is not the case in other ports P1, P2 and P3. 'Open drain' is a term used for MOS chips, as 'open collector' is used for TTL chips.
- To use Port 0 for both input and output, have to connect Port 0 to pull-up resistors.
- When external pull-up resistors connected upon reset, Port 0 is configures as output port.

Role of Port 0 as Input Port:

- To make Port 0 as input port, the pull up resistors are connected to port 0.
- The port must be programmed by writing 1 to all the bits.
- Let's examine the code example below in which the Port 0 is configured first as an input port by writing 1's to it and then data is received from that port and send to Port 1 (P1).

```
MOV A, #0FFH        ; Load accumulator with value FFH in hex or 255 in
                    Decimal
MOV P0, A           ; Make Port 0 as an input port by writing all 1's to it
BACK: MOV A, P0     ; Get data from Port 0
MOV P1, A           ; Send it to Port 1
SJMP BACK           ; Keep doing it repeatedly
```

The Dual Role of Port 0:

- Port 0 can be used to configure for both data and address. The Port 0 is also designated as AD0 - AD7.
- When connecting an 8051 to an external memory, port 0 provides both address and data.
- The 8051 multiplexes address and data through Port 0 to save pins.
- Address latch enable [ALE] indicates if Port 0 has address or data.
- When ALE=0, it provides data D0 - D7, but when ALE = 1, it has address A0 - A7. With the help of 74LS373 latch, ALE is used for demultiplexing address and data.

PORT 1:

- Port 1 has a total of 8 pins. P1.0, P1.1, P1.2, P.13, P1.4, P1.5, P1.6, and P1.7.
- Port 1 can be used as an input or output.
- As compares to Port 0, this port does not need any pull-up resistors.
- Port 1 already has internally connected pull-up resistors.
- Port 1 is configured as an output port when reset.

Example 1:

Let's examine the code, which will continuously send Output to Port 1 the alternating values 55H and AAH.

```
; Toggle all bits of P1 continuously
        MOV     A, #55H
BACK    MOV     P1, A
        ACALL   DELAY
        CPL     A               ; Complement (invert) register A
        SJMP    BACK
```

Port 1 as Input Port:

The Port 1 must be programmed by writing 1's to all the bits in order to make Port 1 as an input port.

```
MOV     A, #0FFH        ; Load Accumulator with FFH in hex
MOV     P1, A           ; Make P1 an input Port,
                        ; by writing all 1's to Port 1
MOV     A, P1           ; Get data from Port 1
MOV     R7, A           ; Save data in register R7
```

```
ACALL    DELAY          ; Wait
MOV      A, P1          ; Get another data from Port 1
MOV      R6, A          ; Save data in register R6
ACALL    DELAY          ; Wait
MOV      A, P1          ; Get another data from Port 1
MOV      R5, A          ; Save data in register R5
```

- The Port 1 is configures as input port by writing 1's to it, then data is received from that port and saved in R7, R6 and R5.

PORT 2:

- Port 2 also has total of 8 pins. P2.0, P2.1, P2.2, P2.3, P2.4, P2.5, P2.6, and P2.7.
- Port 2 can be used for input or output port.
- To make Port 2 as an input, the Port 2 must be programmed by writing 1's to all bits.

Example 3:

The code will send out continuously to Port 2 an alternating values 55H and AAH to toggle the bits or Port 2 continuously.

```
        MOV     A, #55H
BACK:   MOV     P2, A
        ACALL   DELAY
        CPL     A          ; complement register A
        SJMP    BACK
```

Port 2 as an Input Port:

The Port 2 must be programmed by writing 1's to all the port 2 bits to make Port 2 as an input port.

Example 4:

The Port 2 is configures as input port by writing all 1's, then data is received from that port and is sent to Port 1 continuously.

```
; Get a byte from P2 and send it to P1
        MOV     A, #0FFH    ; Load accumulator with value 0FFH in hex
        MOV     P2, A       ; Make Port 2 an input port by writing 1's
                            to all bits of port 2
BACK:   MOV     A, P2       ; Get data from Port 2
        MOV     P1, A       ; Send data to Port 1
        SJMP    BACK        ; Keep doing it repeatedly
```

Dual Role of Port 2:

- Port 2 has dual role. Port 2 is also designated as A8 - A15. This indicates that Port 2 has a dual function.
- An 8051/31 microcontroller is capable of accessing 64K bytes of external memory, it needs a path for the 16 bits of the address.
- P0 provides the lower 8 bits via A0 - A7 while Port 2 provides bits A8 - A15 of the address.
- When the 8031 is connected to external memory, Port 2 is used for the upper 8 bits of the 16-bit address, and it cannot be used for input / output operations.

PORT 3:

- Port 3 has total 8 Pins. P3.0, P3.1, P3.2, P3.3, P3.4, P3.5, P3.6, and P3.7.
- Port 3 can be sued as an input port or output port.
- Port 3 does need any pull-up resisters, just like the Port 1 and Port 2 does not require pull-up resistors.
- Only Port 0 require Pull-up resistors.
- Port 3 configured as an output port upon system reset.
- Port 3 has additional function of providing signals of interrupts.

Alternate Functions of Port 3:

P3 BIT	FUNCTION	PIN
P3.0	**RxD** (Receive serial communication signals)	10
P3.1	**TxD** (Transmit serial communication signals)	11
P3.2	**INT0'** (External Interrupt 0)	12
P3.3	**INT1'**(External Interrupt 1)	13
P3.4	**T0** (Timer 0)	14
P3.5	**T1** (Timer 1)	15
P3.6	**WR'**(Write signal of external memory)	16
P3.7	**RD'**(Read signal of external memory)	17

I/O BIT MANIPULATION PROGRAMMING IN 8051:

Bit Manipulation is a powerful and widely used feature of an 8051.

Ways of Accessing the Entire 8 bits data:

Example 1:
Let's examine an example in which the entire 8 bits of Port 1 data are accessed.

```
BACK:   MOV     A, #55H
        MOV     P1, A
        ACALL   DELAY
        MOV     A, #0AAH
        MOV     P1, A
        ACALL   DELAY
        SJMP    BACK
```

- In the example above, the code toggles the every bit of Port 1 continuously.
- This code can be **written in a more efficient manner** by accessing the port directly without going through the accumulator.

```
BACK:   MOV     P1, #55H
        ACALL   DELAY
        MOV     P1, #0AAH
        ACALL   DELAY
        SJMP    BACK
```

Read-Modify-Write Feature of an 8051:

- The 8051 ports can be accessed by the read-modify-write technique.
- This feature saves many lines of code by combining in a single instruction.
- The first action is reading the port, the second is modifying the port, and the third is writing to the port.

Example 2:

```
        MOV     P1, #55H        ; Load Port 1 with 55H or 01010101
                                  in Binary
AGAIN:  XRL     P1, #0FFH       ; EX-OR Port 1 with 1111 1111


        ACALL   DELAY           ; Wait
        SJMP AGAIN              ; Repeat the action again
```

- The code above first places 0101 0101 (binary0 into Port 1.
- Then the instruction "XRL P1, #0FFH" performs an XOR logic operation on Port 1 with 1111 1111 (Binary), and then writes the result back into Port 1.
- XOR of 55H and FFH gives AAH, and the XOR of AAH and FFH gives 55H.

Addressability of Ports as a Single Bit:

- This feature is used when it is need to access only 1 or 2 bits of the port instead of the entire 8 bits.
- This is a powerful feature of 8051, that the capability to access the single bit and individual bits of 8051 I/O ports without altering the rest of the bits in that port.

Example 3:

Let's examine the code in which the code toggles the Port bit P1.3 continuously.

```
BACK:   CPL     P1.3    ; Complement Port 1 bit 3 i.e. P1.3
        ACALL   DELAY   ; wait
        SJMP    BACK    ; go back to instruction first instruction again
```

Another way of writing the above program is:

```
AGAIN:  SETB    P1.3    ; Set the Bit 3 of Port 1 only to high i.e. P1.3
        ACALL   DELAY   ; Wait
        CLR     P1.3    ; Clear the Bit 3 of Port 1 to Low i.e. P1.3
        ACALL   DELAY   ; Wait
        SJMP    AGAIN   ; Repeat
```

Addressability of Ports as a Single-Bit:

P0	**P1**	**P2**	**P3**	**PORT BIT**
P0.0	P1.0	P2.0	P3.0	D0
P0.1	P1.1	P2.1	P3.1	D1
P0.2	P1.2	P2.2	P3.2	D2
P0.3	P1.3	P2.3	P3.3	D3
P0.4	P1.4	P2.4	P3.4	D4
P0.5	P1.5	P2.5	P3.5	D5
P0.6	P1.6	P2.6	P3.6	D6
P0.7	P1.7	P2.7	P3.7	D7

Let's examine the code to perform the following tasks:

1. Keep monitoring the Port 1 bit 3, P1.3 until it becomes high.
2. As P1.3 becomes high, write 55H value to P1.
3. Send a High-to-Low pulse to P2.0.

```
        SETB    P1.3        ; Make Port 1 bit 3 i.e. P1.3 an input
        MOV     A, #55H     ; Load accumulator with value 55H
```

```
AGAIN:    JNB     P1.3, AGAIN    ; Get out of the loop when P1.3=1,
                                   means high
          MOV     P1, A          ; Copy value in accumulator i.e. 55H
                                   to Port 1
          SETB    P2.0           ; Make/Set P2.0 High
          CLR     P2.0           ; Make/Clear P2.0 low i.e. High-to-Low
```

- The instruction "JNB P1.3,AGAIN", mean jumps to the target 'AGAIN' if no bit is set or if bit P1.3 is low, and stays in the loop as long as P1.3 becomes low.
- When P1.3 becomes high, it gets out of the loop, writes 55H to Port 1, and creates a High-to-Low pulse by the sequence of instructions SETB and CLR.

4.3 ARITHMETIC PROGRAMS:

- Unsigned numbers are defined as data in which all the bits are used to represent data and no bits are set aside for the positive or negative sign.
- This means that the operand can be between 00 and FFH (0 to 255 decimal) for 8-bit data.

ADDITION OF UNSIGNED NUMBERS:

- In the 8051, in order to add numbers together, the accumulator register (A) must be involved.
- The form of the ADD instruction is :
  ADD A, source ; A=A + source
- The instruction ADD is used to add two operands.
- The destination operand is always in register A while the source operand can be a register, immediate data, or in memory.
- Memory to memory arithmetic operations are never allowed in 8051 Assembly language.
- The instruction could change any of the ‘Auxiliary Carry' AC flag, 'Carry Flag' CY and 'Parity Flag' PF.
- Bits of the flag register, depending on the operands involved.
- The ADD instruction could change any of the above mentioned flags according to its operations of operands involved.

Example 1:

- Let’s see how the flag register is affected:

```
MOV     A, #80H     ; Load the value 80H into the accumulator register
ADD     A, #80H     ; add the value 80H with the value in accumulator
                    0F5H and save the result in register A
```

- **The destination register accumulator contains 00. The following flags are affected:**

Carry Flag CY = 1	Out from D7, there is a carry
Parity Flag PF = 1	An even number of once is zero
Auxiliary Carry AC = 0	No carry from D3 to D4

Addition of Individual Bytes in 8051:

- It is already clear that, the maximum value that an 8-bit register can hold is FFH in hex.
- The carry flag should be checked after the addition of each operand, in order to calculate the sum of any number of operands.

Example 2:

Let's assume that the RAM locations 30H to 44H have some values given below. The code program will find the sum of the values. When the program ends, the register A should contain the low byte and R2 contain the high byte.

```
        MOV     R0, #30H    ; Loading the pointer
        MOV     R3, #5      ; loading the counter
        CLR     A           ; Clear the accumulator register A
        MOV     R5, A       ; Clear the register R2
AGAIN:  ADD     A, @R0      ; Add the byte which is pointed to
                            By R0 register
        JNC     NEXT        ; if no carry, then do not accumulate
                            Carry, ignore and jump
        INC     R5          ; if there is a carry increment R7 by 1
                            To keep track or carries
NEXT:   INC     R0          ; Increment the R0 pointer
        DJNZ    R3, AGAIN   ; Repeat the loop again and again,
                            Until R3 is zero
        END                 ; End of the code program
```

When R0=30H and R3=5, Carry flag=0:

- In the instruction "AGAIN: ADD A, @R0", the 8AH value is added to the accumulator register A.
- The accumulator contains value 0H which is added to the value 8AH i.e. A=0+8AH=8AH.
- The carry flag is zero i.e. CY=0 and the instruction R5 is skipped, since there is no carry so R5=00, and the counter decrement by 1 value and jump to the label 'AGAIN' by the instruction "DJNZ R3, AGAIN" i.e. R3=04.

When R0=31H and R3=4, Carry flag=1:

- In the second iteration of the loop, 9CH is added to the accumulator register A by the instruction "AGAIN: ADD A, @R0".
- The accumulator already contains value 8AH which is added to the value 9CH i.e. A=9C+8AH=26H.
- The carry flag is set to 1 i.e. CY=1 and R5 cannot be skipped in this case and R5 is incremented by 1 i.e. R5=01 by the instruction "INC R5", and the counter decrement by 1 value again, and jump to the label 'AGAIN' by the instruction "DJNZ R3,AGAIN" i.e. R3=03.

When R0=32H and R3=3, Carry flag=0:

- In the third iteration of the loop, 3EH is added to the accumulator register A by the instruction "AGAIN: ADD A, @R0".
- The accumulator already contains value 26H which is added to the value 3EH i.e. A=26+3EH=64H.
- The carry flag is set to 0 i.e. CY=0 and R5 is skipped in this case and R5 is not incremented by 1, and the counter decrement by 1 value again, and jump to the label 'AGAIN' by the instruction "DJNZ R3,AGAIN" i.e. R3=02.

When R0=33H and R3=2, Carry flag=1:

- In the fourth iteration of the loop, 9DH is added to the accumulator register A by the instruction "AGAIN: ADD A, @R0".
- The accumulator already contains value 64H which is added to the value 9DH i.e. A=64H+9DH=01H.
- The carry flag is set to 1 again. i.e. CY=1 and R5 cannot be skipped in this case and R5 is incremented by 1 again i.e. R5=02 by the instruction "INC R5", and the counter decrement by 1 value again, and jump to the label 'AGAIN' by the instruction "DJNZ R3,AGAIN" i.e. R3=01.

When R0=34H and R3=1, Carry flag=1:

- In the fifth iteration of the loop, FFH is added to the accumulator register A by the instruction "AGAIN: ADD A, @R0".

- The accumulator already contains value 01H which is added to the value FFH i.e. A=01H+FFH=00H.
- The carry flag once again is set to 1. i.e. CY=1 and R5 cannot be skipped in this case and R5 is incremented by 1 i.e. R5=03 by the instruction "INC R5", So, the total accumulated carry becomes 03, as the carry occurs total of three times in this program.
- The counter is decrement by 1 value again, by the instruction "DJNZ R3, AGAIN" i.e. R3=00.
- This time it cannot jump to the label 'AGAIN' as R3=0, DJNZ as it is zero so the condition becomes false and it comes out from the loop.
- The program is end. When the loop is finished, the sum is held by the accumulator registers A and R5.
- The accumulator register has low byte saved while the R5 has high byte saved.

ADDITION OF 16-BIT NUMBERS & ADDC INSTRUCTION:

The ADDC instruction means addition with carry. The ADDC instruction is used when two 16-bit data operands are need to be added and the propagation of a carry from the lower byte to the higher byte.

Example 3:

Let's add the two 16 bit numbers:

Carry 1

```
   2B D6
 + 2A 7C
 -------
   56 52
```

- In the example above, when the first byte D6+7C is added, we get the result D6+7C=152.
- Then due to carry flag originally we get D6+7C=52. "1" of "152" will go to the higher byte as a carry.
- The carry flag CY=1. The carry is propagated to the higher byte then we have 1+2B+2A=56.
- All the values are in hex. i.e. 1H+2BH+2AH=56H.

Example 4:

Let's add two 16 bit numbers. First number is 2DF6H and 4C9DH. Put the sum in R5 and R4. R5 for higher byte and R4 for lower byte.

```
CLR     C          ; Clear the carry flag to zero i.e. CY = 0
MOV     A, #F6H    ; Load the low byte to the accumulator i.e. A=F6H
ADD     A, #9DH    ; add the low byte in the accumulator F6H with
                   ; Another low byte 9DH and place the result back in
                   ; Accumulator
```

```
MOV     R4,         ; Save the result of low bytes, sum of addition in R4
                    ; register
MOV     A, #2DH     ; Load the high byte to the accumulator register.
                    ; i.e. A=2DH

ADDC    A, #4CH     ; Add with carry, the high byte in the accumulator
                    ; 2DH with another high byte 4CH and place the
                    ; result back in accumulator i.e. A=1+2D+2C= 7A.
                    ; All values are in Hex.
MOV     R5, A       ; Save the result of high bytes, sum of addition in
                    ; R5 register
```

SUBTRACTION OF UNSIGNED NUMBERS:

- SUBB A, source ; A= A-Source-CY
- In microprocessor there are two different instruction: SUB AND SUBB (subtract with borrow).
- In Intel 8051 there is only SUBB. To make SUB out of SUBB, we have to make CY=0 prior to the execution of the instruction.
- Therefore, there are two cases for the SUBB instruction
    1. With CY=0
    2. With CY=1

SUBB (Subtract with borrow) when CY=0

- In subtraction, the 8051 microprocessor (indeed all Modem CPUs) use the 2's complement method.
- Although every CPU contains adder circuitry, it would be too cumbersome (and take too many transistors) to design separate subtracted circuitry.
- For this reason, the 8051 uses adder circuitry to perform the subtraction command.
- Assuming that the 8051 is executing a simple subtract instruction and that CY=0 prior to the execution of the instruction, one can summarize the steps of the hardware of the CPU in executing the SUBB instruction for unsigned numbers, as follows.
    1. Take the 2's complement of the subtrahend (source operand)
    2. Add it to the minuend (A)
    3. Invert the carry
- These three steps are performed for every SUBB instruction by the internal hardware of the 8051 CPU, regardless of the source of the operands, provided that the addressing mode is supported.
- After these three steps the result is obtained and the flags are set.

- If the CY=0 after the execution of SUBB, the result is positive; if CY=1, the result is negative and the destination has the 2's complement of the result.
- Normally, the result is left in 2's complement, but the CPL (complement) and INC instruction can be used to change it
- The CPL instruction perform the 1's complement of the operand; then the operand is incremented (INC) to get the 2's complement.

Example:
Show the steps involved in the following.

```
CLR      C            ; make CY=0
MOV      A, #3FH      ; load 3FH into A (A=3FH)
MOV      R3, #23H     ; load 23H into R3 (R3=3FH)
SUBB     A, R3        ; subtract A-R3, place result in A
```

Solution:

```
 A= 3F        0011 1111          0011 1111
R3=23         0010 0011        + 1101 1101      (2's complement)
    1C                       1  0001 1100
                             0   CF=0     (step 3)
```

The flag would be set as follows: CY=0, AC=0 and the programmer must look at the carry flag to determine if the result is positive or negative.

SUBB (Subtract with borrow) when CY=1:
- This instruction is used for multibyte numbers and will take care of the lower operand.
- If CY=1 prior to executing the SUBB instruction, it also subtracts 1 from the result.

Example:

```
        CLR      C
        MOV      A, #4CH          ; load A with value 4CH (A=4CH)
        SUBB     A, #6EH          ; subtract 6E from A
        JNC      NEXT             ; if CY=0 jump to NEXT target
        CPL      A                ; if CY=1 then take 1's complement
        INC      A                ; and increment to get 2's complement
NEXT:   MOV      R1, A            ; save A in R1
```

Solution:

- Following are the steps for "SUBB A, #6EH:

```
  4C          0100 1100                    0100 1100
- 6E          0110 1110   2's complement =  1001 0010
 - 22                                      0 1101 1110
```

- CY= 1, the result is negative, in 2's complement.

Example:

```
CLR     C                   ; CY=0
MOV     A, #62H             ; load A with value 62H (A=62H)
SUBB    A, #96H             ; 62H – 96H=CCH with CY=1
MOV     R7, A               ; save the result
MOV     A, #27H             ; A=27H
SUBB    A, #12              ; 27H-12H-1=14H
MOV     R6, A               ; save the result
```

Solution:

- After the SUBB, A=62H-96H=CCH and the carry flag is set high indicating there is a borrow.
- Since CY=1, when SUBB is executed the second time A=27H-12H-1=14H.
- Therefore, we have 2762H-1296H=14CCH.

UNSIGNED MULTIPLICATION AND DIVISION:

In multiplying or dividing two numbers in the 8051, the use of registers A and B is required since the multiplication and division instructions work only with these two registers.

MULTIPLICATION OF UNSIGNED NUMBERS:

- The 8051 supports byte-by-byte multiplication only.
- The bytes are assumed to be unsigned data.
- The syntax is

```
MUL AB      ; A x B, place 16-bit result in B and A
```

- In byte-by-byte multiplication, one of the operands must be in register A, and the second operand must be in register B.
- After multiplication, the result is in the A and B registers, the lower byte is in A, and the upper byte is in B.

Example:

Multiplies 25H by 65H and the result is a 16-bit data that is held by the A and B registers.

```
MOV     A, #25H         ; load 25H to register A
MOV     B, #65H         ; load 65H in register B
MUL     AB              ; 25H *65H=E99 where
                        ; B= 0EH and A =99H
```

DIVISION OF UNSIGNED NUMBERS:

- In the division of unsigned numbers, the 8051 supports byte over byte only.
- The syntax is

```
DIV AB          ; divide A by B
```

- When dividing a byte by a byte, the numerator must be in register A and the denominator must be in B.
- After the DIV instruction is performed, the quotient is in A and the remainder is in B.

```
MOV     A, #95      ; load 95 into A
MOV     B, #10      ; load 10 into B
DIV     AB          ; now A= 09 (quotient) and
                    ; B=05 (remainder)
```

- Following points for instruction "DIV AB"
  1. The instruction always makes CY=0 and OV=0 if the denominator is not 0.
  2. If the denominator is 0 (B=0), OV=1 indicates an error, and CY=0.in all microprocessor when dividing a number by 0 is to indicate that the invalid result of infinity.
  3. In the 8051, the OV flag is set to 1.

Example:

In a semester, a student has to take six courses. The marks of the student (out of 25) are stored in RAM locations 47H onwards. Find the average marks, and output it on port 1.

Solution:

```
MOV     R1, #06     ; R1 stores the number of courses
MOV     B, #06      ; only B can be used as the divisor register
MOV     R0, #47H    ; R0 acts as the pointer to the data
MOV     A, #0       ; Clear A
```

```
BACK:   ADD   A,@R0        ; add the data to the A register
        INC   R0           ; increment the pointer
        DJNZ  R1, BACK     ; repeat addition until R1=0
        DIV   AB           ; divide the sum by 6 to get the average
                           ; The quotient is in A, remainder in B
        MOV   P1, A        ; ignore the remainder, output the average
```

CONCEPT OF SIGNED NUMBER:

- All data items used unsigned numbers, meaning that the entire 8-bit operand was used for the magnitude.
- Many application required signed data.

Concept of signed number in computer:

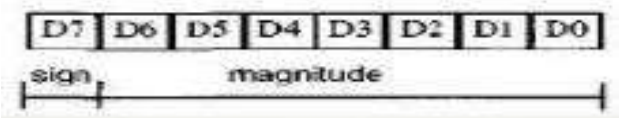
- In everyday life, numbers are used that could be positive or negative.
- **For example:-** a temperature of 5 degrees below zero can be represented as -5, and 20 degrees above zero as +20.
- Computers must be able to accommodate such numbers, that computer scientist have devised the following arrangement for the representation of signed positive and negative numbers.
- The most significant bit (MSB) is set aside for the sign (+ or -) while the rest of the bits are used for the magnitude.
- The sign is represented by 0 for positive (+) numbers and 1 for negative (-) numbers.
- Signed byte representation is discussed below:

Signed 8-bit operands:

- In signed byte operands, D7 (MSB) is the sign and D0-D6 are set aside for the magnitude of the number.
- If D7=0 the operand is positive, and if D7=1, it is negative.

Positive number:

- The range of positive number that can be represented by the format shown in figure below is 0 to +127.



- If a positive number is larger than +127, a 16-bit size operand must be used.
- Since the 8051 doesn't support 16-bit data.

Negative number:

- For negative numbersD7 is 1; however the magnitude is represented in its 2's complement.
- Convert to negative number representation (2's complement), follows these three steps:
    1. Write the magnitude of the numbers in 8-bit binary (no sign).
    2. Invert each bit.
    3. Add 1 to it.

Example 1:

-7

1. 0000 0111
2. 1111 1000
3. 1111 1001
4. F9H ; representation in hex

Example 2:

-56

1. 0011 1000
2. 1100 0111
3. 1100 1000
4. C8H ; representation in hex

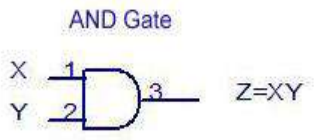
4.4 LOGIC PROGRAMS:

PROGRAM USING LOGIC AND COMPARE INSTRUCTION:

- Logic instruction are widely used instruction.
- Logic instructions such as AND, OR, exclusive-or (XOR) and complement.

AND:

- ANL destination, source ; dest=dest AND source
- This instruction will perform a logical AND on the two operands and place the result in the destination.
- The destination is normally the accumulator.



TRUTH TABLE

INPUTS		OUTPUT
X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

Example:

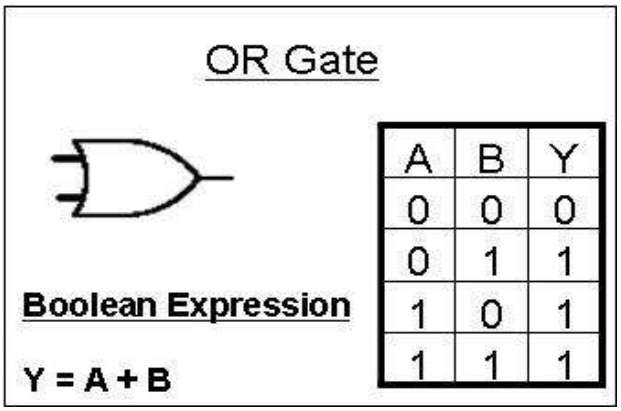
```
MOV     A, #35H          ; A=35H
ANL     A, #0FH          ; A = A AND 0FH (now A=05)
```

Solution:

```
35H     0011 0101
0FH     0000 1111
05H     0000 0101        35H AND 0FH= 05H
```

OR:

- ORL destination, source; dest=dest OR source
- The destination and source operands are ORed, and the result is placed in the destination.



A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

Example:

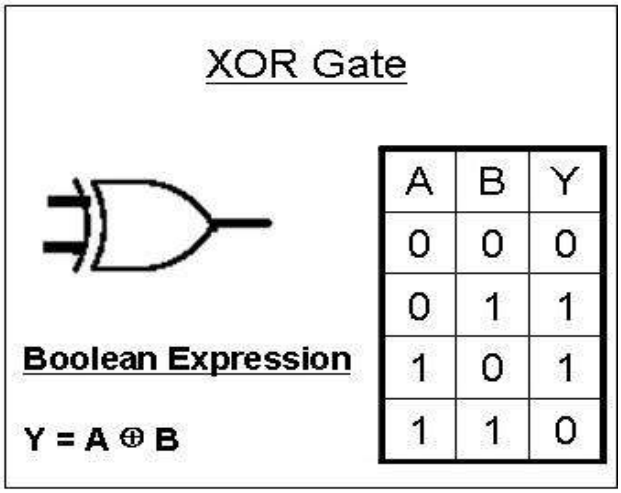
```
MOV     A, #04H          ; A=04H
ANL     A, #30H          ; A = A OR 30H (now A=43H)
```

Solution:

```
04H     0000 0100
30H     0011 0000
34H     0011 0100        04H OR 30H= 34H
```

XOR:

- XRL destination, source ; dest = dest XOR source
- This instruction will perform a XOR operation on the two operands and place the result in the destination.
- The destination is normally the accumulator.



Example 1:

```
MOV     A, #54H          ; A=54H
ANL      A, #78H          ; A = A XOR 78H (now A=2CH)
```

Solution:

```
54H     0101 0100
78H     0111 1000
2CH     0010 1100          54H OR 78H= 2CH
```

Example 2:

Read and test P1 to see whether it has the value 45H. If it does, send 99H to P2.

```
        MOV   P2, #00          ; clear P2
        MOV   P1, #0FFH        ; make P1 an input port
        MOV   R3, #45          ; R3=45H
        MOV   A, P1            ; read P1
        XRL    A, R3
        JNZ    EXIT            ; jump if A has value other than 0
        MOV    P2, #99
EXIT:…
```

- JNZ and JZ test the contents of the accumulator only.
- There is no such thing as a zero flag in the 8051
- Another widely used application of XRL is to toggle bits of an operand.

CPL A (complement accumulator):

- This instruction complements the contents of register A.

- The complement action changes the 0s to 1s and 1s to 0s. this is also called as 1's complement.

Example:
Find the 2's complement of the value 85H.

Solution:

```
MOV      A, #85H
CPL       A                  ; 1's complement
ADD       A, #1              ; 2's complement
```

85H= 1000 0101
1'S = 0111 1010
+ 1
0111 1011= 7BH

COMPARE INSTRUCTION:

- The 8051 has an instruction for the compare operation.it has the following syntax

CJNE destination, source, relative index

- In the 8051, the action of comparing and jumping are combined into a single instruction called CJNE (compare and jump if not equal)
- The CJNE instruction compare two operands, and jumps if they are not equal.
- In addition, it changes the CY flag to indicate if the destination operand is larger or smaller.
- The operand always remain unchanged.
- For example: after the execution of the instruction "CJNE A, NEXT", register A still has its original value.
- This instruction compares register A with value 67H and jump to the target address NEXT only if register A has a value other than 67H

Example

```
        MOV       A, #55H
        CNJE,      A, #99H, NEXT
        ……
        ……...
NEXT:
```

a) In a given example will it jump to NEXT?
b) What is in A after the CJNE instruction is executed?

Solution:

a) Yes, it jumps because 55H 99H are not equal.
b) A=55H, its original value before the comparison.

- In CJNE, the destination operand can be in the accumulator or in one of the Rn register.
- The source operand can be in a register, in memory or immediate.

Example 1:
10 hex number are stored in ram location 50H onwards. Write a program to find the biggest number in the set. The biggest number should finally be saved in 60H.

Solution:

```
        MOV     R0, #50H        ; R0 is the pointer to the data
        MOV     R1, #10         ; R1 is the counter
        MOV     B, #0           ; B=0
BACK:   MOV     A, #R0          ; move a number to A
        CJNE    A, B, LOOP      ; compare with B
LOOP:   JC      LOOP1           ; if A<B, jump to LOOP1
        MOV     B, A            ; if A>B, move it to B,
                                    ; i.e., the bigger number should be in B
        INC     R0                  ; increment the pointer
        DJNZ    R1, BACK        ; repeat until the counter=0
        SJMP    NEXT            ; jump to EXIT, the biggest number is in B
LOOP1:  INC     R0              ; this is another loop, taken when the bigger
                                    ; Number was already in B after a comparison
        DJNZ    R1, BACK        ; repeat until the counter=0
NEXT:   MOV     A, B            ; transfer the biggest number to the A register
        MOV     60H, A          ; transfer the result to RAM location 60H.
        END
```

Example 2:
Assume that P1 is an input port connected to a temperature sensor. Write a program to read the temperature and test it for the value 75. Accordingly to the test results, place the temperature value into the registers indicated by the following.
If T=75 then A=75
If T<75 then R1=T
If T>75 then R2=T

Solution:

```
        MOV     P1, #0FFH               ; make P1 an input port
```

```
        MOV     A, P1                   ; read P1 port, temperature
        CJNE    A, 65, OVER             ; compare with B

        SJMP    EXIT            ; A=75, exit
OVER:   JNC     NEXT                    ; if CY=0 then A>75
        MOV     R1, A                   ; if CY=1, A<75, save in R1
        SJMP    EXIT            ; and exit
NEXT:   MOV     R2, A           ; A>75, save it in R2
EXIT:   .....
```

Example 3:

Assume internal RAM memory location 40H-44H contain daily temperature for five days, as shown below. Search to see if any of the values equals 65. If value 65 does exists in the table, give its location R4, otherwise make R4=0

40H= (76) 41H=(79) 42H=(69) 43H=(65) 44H=(62)

Solution:

```
        MOV     R4, #0H                 ; R4=0
        MOV     R0, #40                 ; load pointer
        MOV     R2, #05                 ; load counter
        MOV     A, #65          ; move a number to A
BACK:   CJNE    A, @R0, NEXT    ; compare with B
        MOV     R4, R0          ; if 65 save address
        SJMP    EXIT            ; and exit
NEXT:   INC     R0              ; otherwise increment pointer
        DJNZ    R2, BACK        ; keep checking until count=0
EXIT    .....
```

Example 4:

Write a program to check if the character string of length 7, stored in RAM locations 50H onwards is a palindrome. If it is, output 'Y' to P1

Solution:

```
        MOV     R2, #03H        ; take half the string length as a counter value
        MOV     R0, #50H        ; take R0 as pointer to the forward reading
        MOV     R1, #56h        ; take R1 as pointer for the backward
                                  ; Reading of the string
BACK:   MOV     A, @R0          ; move into A the character pointed by R0
```

```
        MOV     B, @R1          ; move into B the character pointed by R1
        CJNE    A, B, NEXT      ; compare it with the character pointed by R1
        INC     R0                ; increment the forward counter
        DEC     R1              ; decrement the backward counter
        DJNZ    R2, BACK        ; repeat until all the character are compared
        MOV     P1, # 'Y'       ; since the string is a palindrome, output 'Y'
NEXT:   NOP                     ; if not equal, do nothing since it is not a
                                  ; Palindrome
        END
```

ROTATE AND SWAP INSTRUCTION:

ROTATE:

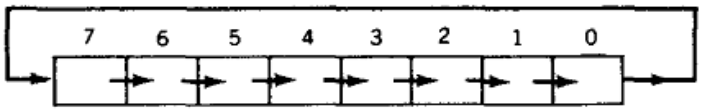
- It is a logical operation of 8085 microprocessor. It is a 1 byte instruction.
- This instruction does not require any operand after the opcode.
- It operates the content of accumulator and the result is also stored in the accumulator.
- The Rotate instruction is used to rotating the bits of accumulator.

- **Types of ROTATE Instruction:**
  There are 4 categories of the ROTATE instruction:

  - Rotate accumulator left (RLC),
  - Rotate accumulator left through carry (RAL),
  - Rotate accumulator right (RRC),
  - Rotate accumulator right through carry (RAR).
- Among these four instructions; two are for rotating left and two are for rotating right.

Rotate accumulator right through carry (RRA):

- In this instruction, each bit is shifted to the adjacent right position. Bit D0 becomes the carry bit and the carry bit is shifted into D7. Carry flag CY is modified according to the bit D0.



RR A

- RR A; rotate right A

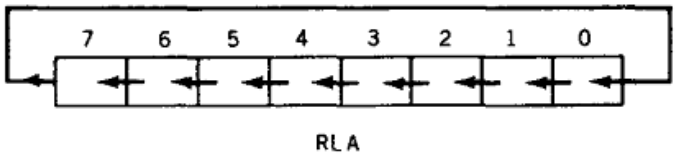
Example:

```
MOV         A, #36H     ; A=0011 0110
RR           A            ; A=0001 1011
RR           A            ; A=1000 1101
```

```
RR      A       ; A=1100 0110
RR      A       ; A=0110 0011
```

Rotate accumulator left through carry (RLA):

- In this instruction, each bit is shifted to the adjacent left position. Bit D7 becomes the carry bit and the carry bit is shifted into D0. Carry flag CY is modified according to the bit D7.



RL A

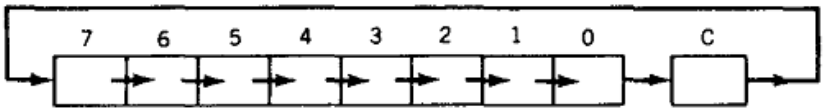
- RL A ; rotate left A

Example:

```
MOV     A, #72H     ; A=0111 0010
RL      A           ; A=1110 0100
RL      A           ; A=1100 1001
```

Rotate accumulator right (RRC):

- In this instruction, each bit is shifted to the adjacent right position. Bit D7 becomes D0. Carry flag CY is modified according to the bit D0.



RRC A Carry Flag

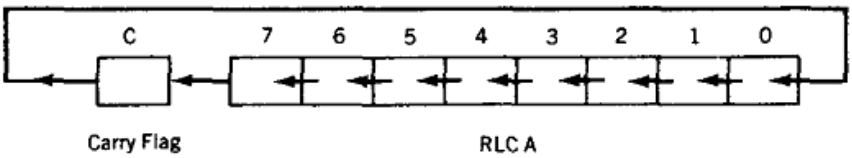
- RRC A ; rotate right through carry

Example:

```
CLR     C           ; make CY=0
MOV     A, #26H     ; A=0010 0110
RRC     A           ; A=0001 0011   CY=0
RRC     A           ; A=0000 1001   CY=1
RRC     A           ; A=1000 0100   CY=1
```

Rotate accumulator left (RLC):

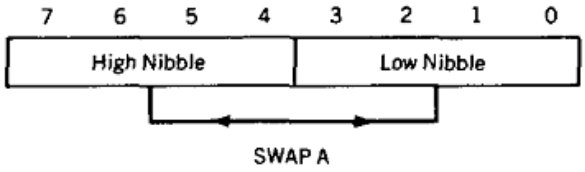
- In this instruction, each bit is shifted to the adjacent left position. Bit D7 becomes D0. Carry flag CY is modified according to the bit D7.




```
SETB        C              ; make CY=1
MOV         A, #15H        ; A=0001 0101
RLC         A              ; A=0010 1011    CY=0
RLC         A              ; A=0101 0110    CY=0
RLC         A              ; A=1010 1100    CY=0
RLC         A              ; A=0101 1000    CY=1
```

SWAP:

- The SWAP instruction exchanges the low-order and high-order nibbles within the accumulator. No flags are affected by this instruction.
- See Also: XCH, XCHD



- SWAP A

Example:
Register A= 5EH (original value) after SWAP Register A=E5H

BCD and ASCII Application program:

ASCII numbers:

- On ASCII keyboards, when the key '0' is activated, "011 0000" (30H) is provided to the computer. Similarly, 31H (011 0001) is provided for the key "1" and so on.
- ASCII is standard in the United State (and many other country), BCD numbers are universal.
- Since the keyboard, printers and monitors all use ASCII

Packed BCD to ASCII Conversion:

- The RTC provides the time of day (hour, minute, second) and the date (year, month, day) continuously, regardless of whether the power is on or off, this data is provided in packed BCD.
- Data to be displayed on a device such as an LCD, or to be printed by the printer, it must be in ASCII format.

- To convert packed BCD to ASCII, it must be converted to unpacked BCD is tagged with 011 0000(30H).

Packed BCD	Unpacked BCD	ASCII
29H	02H & 09H	32H & 39H
0010 1001	0000 0010 &	0011 0010 &
	0000 1001	0011 1001

ASCII to Packed BCD:

- To convert ASCII to BCD, it is first converted to unpacked BCD (to get rid of 3), and then combined to make packed BCD.
- For example, for 4 and 7 the keyboard gives 34 and 37, respectively. The goal is to produce 47H or "0100 0111", which is packed BCD.

Key	ASCII (hex)	Binary	BCD (unpacked)
0	30	011 0000	0000 0000
1	31	011 0001	0000 0001
2	32	011 0010	0000 0010
3	33	011 0011	0000 0011
4	34	011 0100	0000 0100
5	35	011 0101	0000 0101
6	36	011 0110	0000 0110
7	37	011 0111	0000 0111
8	38	011 1000	0000 1000
9	39	011 1001	0000 1001

Example 1:

Assume that register A has packed BCD. Write a program to convert packed BCD to two ASCB number and place them in R2 and R6.

Solution:

```
MOV   A, #29H          ; A=29H, packed BCD
MOV   R2, A            ; keep a copy of BCD data in R2
ANL   A, #0FH          ; mark the upper nibble (A=09)
ORL   A, #30H          ; make it an ASCII, A = 39H ('9')
MOV   R6, A            ; save it (R6=39H ASCII char)
MOV   A, R2            ; A=29H, get the original data
```

```
ANL   A, #0F0H        ; mask the lower nibble (A=20)
RR    A               ; rotate right
RR    A               ; rotate right
RR    A               ; rotate right
RR    A               ; rotate right, (A=02)
ORL   A, #30H         ; A=32H, ASCII char '2'
MOV   R2, A           ; save ASCII char in R2
```

Of course, in the above code we can replace all the RR instructions with a single "swap A" instruction.

Key	ACII	Unpacked BCD	Packed BCD
4	34	0000 0100	
7	37	0000 0111	01000111 or 47H

```
MOV   A, # '4'        ; A = 34H, hex for ASCII char 4
ANL   A, #0FH         ; mask upper nibble (A=04)
SWAP  A               ; A=40H
MOV   B, A
MOV   A, # '7'        ; R1 = 37H, hex for ASCII char7
ANL   A, #0FH         ; mask upper nibble (R1=07)
ORL   A, B            ; A=47H, packed BCD
```

After this conversion, the packed BCD numbers are processed and he result will be in packed BCD format. A special instruction, "DAA', requires that data be in packed BCD format.

Using a look-up table for ASCII:

- In some applications it is much easier to use a look-up table to get the ASCII character we need.
- This is a widening used concept in interfacing a keyboard to the microcontroller. This is shown in Example 6-35.

Example 2:

Assume that the lower three bits of PI are connected to three switched. Write a program to send the following ASCII characters to P2 based on the status of the switches.

000 '0'
000 '1'
000 '2'
011 '3'
100 '4'
101 '5'
110 '6'
111 '7'

Solution:

```
        MOV     DPTR,  #MYTABLE
        MOV     A, P1                   ; get 0W status
        ANL     A, #07H                 ; mask all but lower & bits
        MOVC    A, #A+DPTR              ; get the data from look-up-table
        MOV     P2, A                   ; display value
        SJMP    $                       ; stay here
; -------------------------------------------------
        ORG     400H
MYTABLE DB      '0', '1', '2', '3', '4', '5' , '6', '7'
        END
```

Binary (hex) to ASCII conversion:

- Many ADC (analog-to –digital converter) chips provide output data in binary (hex).
- To display the data on an LCD or PC screen, we need to convert it to ASCII.
- The following code shows the binary – to – ASCII conversion program.
- Notice that the subroutine gets a byte of 8 – bit binary (hex) data from P1 and converts it to decimal digits, and 2nd subroutine converts the decimal digits to ASCII digits and save them.

- The low digit are saved in the lower address location and the high digit in the higher address location.
- This is referred to as the little–Endian convention that is low-byte to low-location and high-byte to high- location.

Binary to ASCII Conversion Program:

Example 3:

```
; CONVERTING BIN (HEX) TO ASCII


RAM_ADDR        EQU 40H
ASCI_RESULT     EQU 50H
COUNT           EQU 3
; ----------------------------------------------main program
                ORG 0
                ACALL BIN_DRC_CONVRT
                ACALL DEC_ASCI_CONVRT
                SJMP  $
; ----------------------------------------------Converting BIN (HEX) TO DEC (00-FF TO 000-255)
```

BIN_DEC_CONVERT:

```
        MOV     R0, #RAM_ADDR   ; save DEC digits in these RAM Locations
        MOV     A, P1           ; read data from P1
        MOV     B, #10          ; B=0A hex (10 Dec)
        DIV     AB              ; divide by 10
        MOV     @R0, B          ; save lower digit
        INC     R0
        MOV     B, #10
        DIV     AB              ; divide by 10 once more
        MOV     @R0, B          ; save the next digit
```

```
        INC     R0
        MOV     @R0, A      ; save the last digit
        RET
; ----------------------------------------Converting DEC digits to displayable ASCII digits
```

DEC_ASCI_CONVERT:

```
        MOV     R0, #RAM_ADDR       ; addr of DEC data
        MOV     R1, #ASCI_RESULT    ; addr of ASCII data
        MOV     R2, #3              ; count
BACK:   MOV     A, @R0              ; get DEC digit
        ORL     A, #30H             ; make it an ASCII digit
        MOV     @R1, A              ; save it
        INC     R0                  ; next digit
        INC     R1                  ; next
        DJNZ    R2, BACK            ; repeat until the last one
        END
```

4.5 SIMPLE PROGRAMS:

Example 1:
WAP for the addition of 8-bit numbers located in two memory addresses.

```
        MOV     R0, #20H        ; set source address 20H to R0
        MOV     R1, #30H        ; set destination address 30H to R1
        MOV     A,@R0           ; take the value from source to register A
        MOV     R5, A           ; Move the value from A to R5
        MOV     R4, #00H        ; Clear register R4 to store carry
        INC     R0              ; Point to the next location
```

```
        MOV     A,@R0       ; take the value from source to register A
        ADD     A, R5       ; Add R5 with A and store to register A
        JNC     SAVE
        INC     R4          ; Increment R4 to get carry
        MOV     B, R4       ; Get carry to register B
        MOV     @R1, B      ; Store the carry first
        INC     R1          ; Increase R1 to point to the next address
SAVE:   MOV     @R1, A      ; Store the result
HALT:   SJMP HALT           ; Stop the program
```

Time delay program:

Example 2:
To find the size of the delay if the crystal frequency of 11.0592 MHz is connected.

```
        MOV     A, #55H         ; load A with 55H
AGAIN:  MOV     P1, A           ; issue value in register A to port 1
        ACALL   DELAY           ; time delay
        CPL     A               ; complement register A
        ASJMP   AGAIN           ; keep doing this indefinitely
; ---------Time Delay
DELAY:  MOV     R3, #225        ; load R3 with 255
HERE:   DJNZ    R3, HERE        ; stay here until R3 become 0
        RET                     ; return to caller
```

Answer:

- We have the following machine cycles for each instruction of the DELAY subroutine.

```
DELAY: MOV R2, #255         Machine Cycle = 1
HERE: DJNZ R2, HERE         Machine Cycle = 2
RET                         Machine Cycle = 1
```

- Therefore, we have a time delay of [(255 x 2) + 1 + 1] x 1.085 us = 555.52 us

- Very often we used to calculate the time delay based on the instructions inside the loop and ignore the clock cycles associated with the instructions outside the loop.

Example 3:

To find the time delay for the following subroutine with 11.0592 MHz crystal frequency is connected to the 8051 system.

```
DELAY: MOV     R2, #255          Machine Cycle = 1
HERE:  NOP                       Machine Cycle = 1
       NOP                       Machine Cycle = 1
       NOP                       Machine Cycle = 1
       NOP                       Machine Cycle = 1
       DJNZ    R2, HERE          Machine Cycle = 2
       RET                       1
```

- The time delay inside the HERE loop is [255(1+1+1+1+2)] x 1.085 us = 1660.05μs
- The time delay of the two instructions outside the loop is:

= [1660.05 us+1+1)] x1.085 μs
= 1803.32425μs

Square wave program:

Example 1:
8051 assembly level program to create a square wave of different duty cycle.

Assume duty cycle 50%
Assume 12MHz clock is connected to microcontroller
Use timer
Check output in P3.2

Solution:

```
        ORG     0000H
        MOV     TMOD, #01H
UP:     SETB    P3.2
        LCALL   DELAY
        SJMP    UP
       DELAY:
        MOV     TH0, #0FEH
        MOV     TL0, #0CH
        CLR     TF0
        SETB    TR0
HERE:   JNB     TF0, HERE
```

```
RET
END
```

Example 2:

Write a program to continuously generate a square wave of 2 kHz frequency on pin P1.5 using timer 1. Assume the crystal oscillator frequency to be 12 MHz

The period of the square wave is T = 1/ (2 kHz) = 500 μs. Each half pulse = 250 μs. The value n for 250 s is: 250 μs /1 μs = 250

65536 - 250 = FF06H.

TL = 06H and TH = 0FFH.

```
        MOV     TMOD, #10       ; Timer 1mode
AGAIN:  MOV     TL1, #06H       ; TL0 = 06H
        MOV     TH1, #0FFH      ; TH0 = FFH
        SETB    TR1             ; Start timer 1
BACK:   JNB     TF1, BACK       ; Stay until timer rolls over CLR
        CLR     TR1             ; Stop timer 1
        CPL     P1.5            ; Complement P1.5 to get

                                ; High, Low
        CLR     TF1             ; Clear timer flag 1
        SJMP    AGAIN           ; Reload timer
```

SIMPLE 8051 PROGRAMMING IN C:

- Compiler produce hex files that is downloaded into the ROM of the microcontroller. The size of the hex file produced by the compiler is one of the main concern of microcontroller programmers, for two reasons:

  1. Microcontroller have limited on-chip ROM.
  2. The code space for the 8051 is limited to 64Kbytes.

- While assembly language produces a hex file that is much smaller than C, programming in assembly language is tedious and time consuming.
- C programming, on the other hand, is less time consuming and much easier to write, but the hex file size produced is much larger than assembly language.
- **Some major reasons for writing programs in C instead of assembly:**

1. It is easier and less time consuming to write in C than assembly.
2. C is easier to modify and update.
3. The code available can be used in function libraries.
4. C code is portable to other microcontrollers with little or no modification.

PROGRAMS

Example 1:
Write an 8051 program to send values 00 – FF to port P1.

Solution:

```
#include <reg51.h>
 Void main (void)
 {
 Unsigned char z;
 for (z = 0; z<= 255; z++)
 P1 = z;
 }
```

Example 2:

Write an 8051 C program to send hex values for ASCII characters of 0, 1, 2, 3, 4, 5, A, B, C, & D to port P1.

Solution:

```
#include <reg51.h>
Void main (void)
  {
    Unsigned char mynum [] = “012345ABCD”;
    Unsigned char z;
    for (z=0; z<=10; z++
    for (z = 0; z<= 10; z++)
     P1 = mynum [z];
```

```
    }
```

Example 3:
Write an 8051 C program to toggle all the bits of P1 continuously.

Solution:

```
// Toggle P1 forever
#include <reg51.h>
Void main (void)
 {
 for (; ;)                    //repeat forever
 {
        P1 = 0x55;         //0x indicates the data is in hex (binary)
        P1 = 0x AA;
     }
 }
```

Signed int:

Signal int is a 16 – bit data type that uses the most significant bit (D15 of D15 – D0) to represent the – or + value a result , we have only 15 – bits for the magnitude of the number or values from -32,768 to +32,767.

Sbit (single bit):

- The Sbit keyword is a widely used 8051 C data type designed specifically to access single- bit addressable registers. It allows access to the single bits of the SFR registers.
- Among the SFRs that are widely used & are also bit – addressable are ports P0 – P3. Sbit can used to access individual bits of the ports

Example 4:
Write an 8051 C program to send values of -4 to +4 to port P1.

Solution:

```
 //sign numbers
```

```
#include <reg51.h>
Void main (void)
  {
  char mynum[]= { +1, -1, +2, -2, +3, -3, +4, -4};
  Unsigned char z;
   for (z = 0; z<= 8; z++)
    P1 = mynum [z];
  }
```

Example 5:
Write an 8051 C program to toggle bit D0 of the port P1 (P1.0) 50,000 times.

Solution:

```
#include <reg51.h>
Sbit MYBIT = P1^0;              //notice that Sbit is
                                //declared outside of main
Void main (void)
   {
    Unsigned int z;
     for (z = 0; z<= 50000; z++)
     {
      MYBIT = 0;
       MYBIT = 1;
            }
      }
```

Example 6:
Write an 8051 C Program to toggle bits of P1 continuously forever with some delay.

Solutions:

```
// Toggle P1 forever with some delay in between 'on' and 'off'.
#include <reg51.h>
Void main (void)
{
 Unsigned int x;
 for (; ;)                                        //repeat forever
        {
          P1 = 0x55;
           for (x = 0; x<40000; x++);              //delay forever
           P1=0xAA;
           For (x = 0; x=40000; x++);
        }
}
```

Example 7:
Write an 8051 program to toggle the bits of P1 ports continuously with a 250 ms delay.

Solution:
The program below is tested for the DS89C420 with XTAL = 11.0592 MHz

```
#include <reg51.h>
Void MSDeLay (unsigned int);
Void main (void)
        {
          While (1)       // repeat forever
                {
                 P1 = 0x55;
                 MSDelay (250);
```

```
            P1 = 0xAA;
            MSDelay (250);
          }
      }
Void MSDelay (unsigned int itime)
    {
      Unsigned int i, j;
      for ( i = 0 ; i<itime ;  i++)
      for (j=0 ; j<1275 ; j++i );
     }
```

Example 8:
Write an 8051 C program to toggle all the bits of P0 and P2 continuously with a 250ms delay.

Solution:

```
//This program is tested for the DS89C420 with XTAL =11.059 MHz
#include <reg51.h>
Void MSDelay (unsigned int);
Void main (void)
    {
      While (1)        // another way to do it forever
          {
            P0 = 0x55;
            P2 = 0x55;
            MSDelay (250);
          }
       }
Void MSDelay (unsigned int itime)
```

```
{
  Unsigned int i, j;
  for (i=0 ; i<itime; i++)
  for (j = 0; j<1275 ; j++) ;
}
```

4.6 TIMER PROGRAMMING:

- 8051 has 2, 16-bit Up Counters T1 and T0.
- If the counter counts internal clock pulses it is known as timer.
- If it counts external clock pulses it is known as counter.
- Each counter is divided into 2, 8-bit registers TH1 - TL1 and TH0 - TL0.
- The timer action is controlled mainly by the TCON and the TMOD registers.

Example 1:

WAP to generate a delay of 20 µsec using internal timer-0 of 8051. After the delay send a "1" through Port3.1. Assume Suitable Crystal Frequency

NOTE: In 8051, if we select a Crystal of 12 MHz, then Timer freq will be $f_{osc}/12$ è 1MHz. Hence each count will require 1/1MHz è 1 µsec. Thus for 20 µsec, the Desired Count will be 20 14H. For an Up-Counter (Mode 1):
Count = Max Count – Desired Count + 1 Count = FFFF – 14 + 1
<u>Count = FFECH</u> #Please refer Bharat Sir's Lecture Notes for this...

```
        MOV TMOD, #01H      ;   TMOD→(0000 0001)2 ...Timer0 Mode1
        MOV TL0, #0ECH      ;   Load lower byte of Count
        MOV TH0, #0FFH      ;   Load upper byte of Count
        MOV TCON, #10H      ;   Program TCON→ (0001 0000)2…start Timer0
WAIT:   JNB TCON.5, WAIT    ;   Wait for overflow
        SETB P3.1           ;   Send a "1" through Port3.1
        MOV TCON, #00H      ;   Stop Timer0
HERE:   SJMP HERE           ;   End of program
```

Example 2:

- **WAP to generate a Square wave of 1 KHz from the TxD pin of 8051, Q11 usingTimer1. Assume Clock Frequency of 12 MHz**

NOTE: For a Square wave of 1 KHz, the delay required is .5 msec.We know, each count will require 1/1MHz è 1 µsec.
Thus for 500 µsec, the Desired Count will be 500_d è 01F4H. For an Up-Counter (Mode 1):
Count = Max Count – Desired Count + 1
Count = FFFF – 01F4 + 1
Count = FE0CH

```
        CLR P3.1            ;   Clear Txd Line initially
        MOV TMOD, #10H      ;   Program TMOD (0001 0000)2 ... Timer1 Mode1
REPEAT: MOV TL1, #0CH       ;   Load lower byte of Count
        MOV TH1, #0FEH      ;   Load upper byte of Count
        MOV TCON, #40H      ;   Program TCON è (0100 0000)2 ... start Timer1
WAIT:   JNB TCON.7, WAIT    ;   Wait for overflow
        CPL P3.1            ;   Toggle Txd pin after the delay
        MOV TCON, #00H      ;   Stop Timer1
        SJMP REPEAT         ;   Repeat the process
```

Example 3:

WAP to generate a Rectangular wave of 1 KHz, having a 25% Duty Cycle from the TxD pin of 8051, using Timer1. Assume XTAL of 12 MHz

NOTE: For a Rectangular wave of 1 KHz, having 25% Duty Cycle: T_{ON} = 250 μsec; T_{OFF} = 750 μsec.

For T_{ON}: Desired Count = 250_d è 00FAH
Count $_{ON}$ = Max Count – Desired Count + 1
Count $_{ON}$ = FFFF – 00FA + 1
Count $_{ON}$ = FF06H

For T_{OFF}: Desired Count = 750_d è 02EEH
Count $_{OFF}$ = Max Count – Desired Count + 1
Count $_{OFF}$ = FFFF – 02EE + 1
Count $_{OFF}$ = FD12H

```
        MOV TMOD, #10H      ;   Program TMOD (0001 0000)2 ...
                                Timer1 Mode1

REPEAT: MOV TL1, #06H       ;   Load lower byte of Count ON
        MOV TH1, #0FFH      ;   Load upper byte of Count ON
        SETB P3.1           ;   Display"1"at Txd
        MOV TCON, #40H      ;   Program TCON (0100 0000)2 ... startTimer1
ON:     JNB TCON.7, ON      ;   Maintain"1" at Txd
        CLR P3.1            ;   Clear Txd
        MOV TCON, #00H      ;   Stop Timer1

        MOV TL1, #12H       ;   Load lower byte of Count
        MOV TH1, #0FDH      ;   Load upper byte of Count OFF
        MOV TCON, #40H      ;   Program TCON (0100 0000)2...start Timer1
OFF:    JNB TCON.7, OFF     ;   Maintain "0" at Txd
        MOV TCON, #00H      ;   Stop Timer1

        SJMP REPEAT         ;   Repeat the process
```

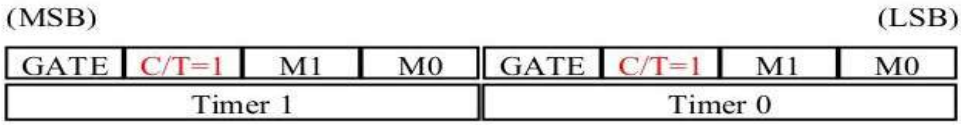
COUNTER PROGRAMMING:

- The timers can also be used as counters counting events happening outside the 8051. 8051 crystal is used as the source of the frequency.
- When the timer/counter is used as a timer, the
- When it is used as a counter, however, it is a pulse outside the 8051 that increments the TH, TL registers.

C/T bit in TMOD register:

- If C/T = 0, the timer gets pulses from the crystal. In contrast, when C/T = 1, the timer is used as a counter and gets its pulses from outside the 8051.
- Therefore, when C/T = 1, the counter counts up as pulses are fed from pins 14 & 15. These pins are called T0 (Timer 0 input) & T1 (Timer 1 input).
- These two pins belong to port 3. In the case of timer 0, when C/T = 1, pin P3.4 provides the clock pulse &the counter counts up for each clock pulse coming from that pin.
- Similarly, for Timer 1, when C/T = 1 each clock pulse coming in from pin P3.5 makes the counter count up.

Pin	Port Pin	Function	Description
14	P3.4	T0	Timer/Counter 0 external input
15	P3.5	T1	Timer/Counter 1 external input



Port 3 pins used for Timers 0 & 1

Example 1:

Design a counter for counting the pulses of an input signal. The pulses to be counted are fed to pin P3.4 XTAL 22 MHz

Solution:

; Tested for an AT89C51 with a crystal frequency of 22 MHz

- In this, Timer 1 is used as time based for timing 1 second.
- During this 1 second, Timer 0 is run as a counter, with input pulses fed into pin P3.4.
- At the end of 1 second, the values in TL0 &TH0 give the number of pulses that were received at pin P3.4 during this 1 second.

- This gives the frequency of the unknown signal, i.e., the no. the no. of pulses received in 1 second.

```
        ORG     0000H
RPT:    MOV     TMOD, #15H      ; timer 1 as timer &timer 0 as counter
        SETB    P3.4            ; make port P3.4 an input port
        MOV     TL0, #00        ; clear TL0
        MOV     TH0, #00        ; clear TH0
        SETB    TR0             ; start counter
        MOV     R0, #28         ; R0 = 28, time 1 second
AGAIN:  MOV     TL1, #00H       ; TL1 = 0
        MOV     TH1, #00H       ; TH1 = 0
        SETB    TR1             ; start timer 1
BACK:   JNB     TF1, BACK       ; test timer 1 flag
        CLR     TF1             ; clear timer 1 flag
        CLR     TR1             ; stop timer 1
        DJNZ    R0, AGAIN       ; repeat the loop until R0 = 0

        MOV     A, TL0          ; since 1 sec has elapsed, check TL0
        MOV     P2, A           ; move TL0 to port 2
        MOV     A, TH0          ; move TH0 to ACC
        MOV     P1, A           ; move it to port 1
        SJMP    RPT             ; repeat
        END
```

As the frequency varies, the values obtained at ports 1 & 2 vary.
The values obtained for 10 Hz, 25Hz, 100Hz … are respectively, 0AH, 19, 64H…

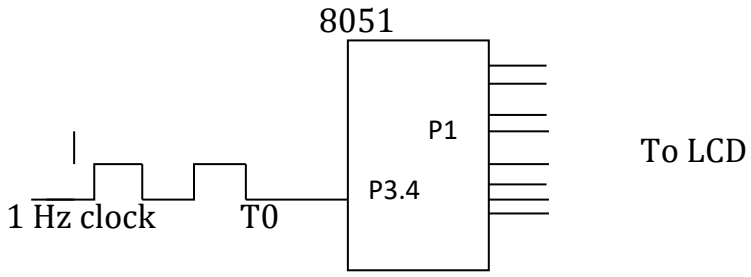
Example 2:

Assume that a 1- Hz frequency pulse is connected to input pin 3.4. Write a program to display counter 0 on an LCD. Set the initial value of TH0 to -60.
Solution:

To display the TL count on an LCD, we must convert 8-bit binary data to ASCII.

```
        ACALL   LCD _ SET _ UP          ; initialize the LCD
        MOV     TMOD, #00000110B        ; counter 0, mode 2, C/T = 1
        MOV     THO, #-60               ; counting 60 pulses
        SETB    P3.4                    ; make T0 as input
AGAIN:  SETB    TR0                     ; starts the counter
BACK:   MOV     A, TL0                  ; get copy of count TL0
        ACALL   CONV                    ; convert in R2, R3, R4
        ACALL   DISPLAY                 ; display on LCD
        JNB     TF0, BACK               ; loop if TF0 = 0
        CLR     TR0                     ; stop the counter 0
        CLR     TF0                     ; make TF0 = 0
; converting 8-bit binary to ASCII
; Upon return, R4, R3, R2 have ASCII data (R2 has LSD)

CONV:   MOV     B, #10                  ; divide by 10
        DIV     AB
        MOV     R2, B                   ; save low digit
        MOV     B, #10                  ; divide by 10 once more
        DIV     AB
        ORL     A, #30H                 ; make it ASCII
        MOV     R4, A                   ; save MSD
        MOV     A, B
        ORL     A, #30H                 ; make 2nd digit an ASCII
        MOV     R3, A                   ; save it
```

```
MOV     A, R2
ORL     A, #30H          ; make 3rd digit an ASCII
MOV     R2, A            ; save the ASCII
RET
```



- By using 60Hz it can generated seconds, minutes, hours.
- On the first round, it starts from 0, since on RESET. TL0 = 0.
- To solve this problem, load TL0 with -60 at the beginning of the program.

PROGRAMMING TIMER 0 & 1 IN 8051 C:

Example 1:
Write an 8051 C program to toggle all the bits of P1 port continuously with some delay. Use timer 0, 16-bit mode to generate.

Solution:

```
#include <reg51.h>

Void T0DeLay (unsigned int);

Void main (void)
    {
      While (1)      // repeat forever
          {
           P1 = 0x55;
           T0Delay (250);
           P1 = 0xAA;
           T0Delay ();
```

```
        }
    }
Void T0Delay ()
    {
TMOD=0x01;              // Timer 0, mode 1
TL0=0x00;               // load TL0
TH0=0x35;               // load TH0
TR0=1;                  // turn on T0
While (TF0==0);         // wait for TF0 to roll over
TR0=0;                  // turn off T0
TF0=0;                  // clear TF0

  }
```

FFFFH-3500H=CAFFH=51967+1=51968

51968 x 1.085µs=56.384 ms is the approximate delay.

Timers 0 & 1 delay using mode 1 (16-bit non-auto-reload):

Example 1:
Write an 8051 C program to toggle only bit P1.5 continuously every 50ms. Use Timer 0, mode1 (16-bit) to create the delay.

Solution:

```
#include <reg51.h>
Void T0MDelay (void):
Sbit mybit = P1^5;
Void main (void)
  {
While (1)
{
        mybit= ~mybit;     // toggle P1.5
        T0M1Delay ();     // Timer 0, mode 1 (16-bit)
```

```
        }
  }
```

Tested for AT89C51, XTAL=11.0592 MHz, using the Proview 32 compiler.

```
Void    T0M1Delay (void)
{
TMOD=0x01;                  // Timer 0, mode 1 (16-bit)
TMOD=0xFD;                   // load TL0
TMOD=0x4B;                   // load TH0
TR0=1;                       // turn on T0
While (TF0==0);              // wait for TF0 to roll over
TR0=0;                       // turn off T0
TF0=0;                       // clear TF0
}
```

Example 2:
Write an 8051 C program to toggle all bits of P2 continuously every 500ms. Use Timer 1, mode1 to create the delay.

Solution:

```
// tested for DS89C420, XTAL=11.0592 MHz, using the Proview 32 compiler.

#include <reg51.h>
Void T1MDelay (void):
Void main (void)
 {
Unsigned char x;
P2=0x55;
While (1)
{
P2= ~P2;
For (x=0; x<=20; x++)
```

```
T1M1Delay ();
        }
  }
Void    T1M1Delay (void)
{
TMOD=0x10;                  // Timer 1, mode 1 (16-bit)
TL1=0xFE;                   // load TL1
TH1=0xA5;                   // load TH1
TR1=1;                      // turn on T1
While (TF1==0);             // wait for TF1 to roll over
TR1=0;                      // turn off T1
TF1=0;                      // clear TF1
}
```

A5FEH=42494 in decimal
65536-42494=23042
23042x 1.085μs=25ms and 20x 25 ms=500ms

Timers 0 & 1 delay using mode 1 (8-bit auto-reload):

Example 1:
Write an 8051 C program to toggle only bit P1.5 continuously every 250ms. Use Timer 0, mode2 (8-bit auto-reload) to create the delay.

Solution:
// tested for DS89C420, XTAL=11.0592 MHz, using the Proview 32 compiler.

```
#include <reg51.h>
Void T0M2Delay (void):
Sbit mybit = P1^5;
Void main (void)
  {
Unsigned char x, y;
While (1)
 {
        mybit= ~mybit;                  // toggle P1.5
```

```
        for (x=0; x=250; x++)          // due to for loop overhead
        for (y=0; y=36; y++)            // we put 36 and not 40
        T0M2Delay ();
      }
  }
Void  T0M2Delay (void)
{
TMOD=0x02;              // Timer 0, mode 2 (8-bit auto-related)
TH0=-23;                // load TH0 (auto-reload value)
TR0=1;                  // turn on T0
While (TF0==0);         // wait for TF0 to roll over
TR0=0;                  // turn off T0
TF0=0;                  // clear TF0
}
```

256-23=233
23x1.085 μs=25 μs
25 μs x 250 x 40=250 ms by calculation.

- However, the scope output does not give us this result. This is due to overhead of the 'for loop' in C. to correct this problem, put 36 instead of 40.

Example 2:
Write an 8051 C program to toggle only bit P1.5 continuously every 250ms. Use Timer 0, mode2 (8-bit auto-reload) to create the delay.

Solution:

```
// tested for DS89C420, XTAL=11.0592 MHz, using the Proview 32 compiler.

#include <reg51.h>
Void T1M2Delay (void):
Sbit mybit = P2^7;
Void main (void)
  {
```

```
Unsigned char x;
While (1)
 {
            mybit= ~mybit;                    // toggle P2.7
            T1M2Delay ();
          }
    }
Void    T1M2Delay (void)
{
TMOD=0x20;                  // Timer 1, mode 2 (8-bit auto-related)
TH1=-184;                    // load TH1 (auto-reload value)
TR1=1;                       // turn on T1
While (TF1==0);              // wait for TF1 to roll over
TR1=0;                       // turn off T1
TF1=0;                       // clear TF1
}
```

1/2500=400 µs
400 µs/2=200 µs
200 µs/1.085 µs=184