- **Unit I: Introduction to Microprocessor**
  - Definition and History of Microprocessors
  - Basic Components of a Digital Computer
  - Basic Components of a Microprocessor
  - Architectures
    - Von Neumann Architecture
    - Harvard Architecture
    - Von Neumann vs Harvard Architecture
  - Instruction Formats & Related Terms
    - Instruction Format
      - Opcode (Operation Code)
      - Operand
      - Example
    - Instruction Cycle
    - Machine Cycle
    - T-State (Clock Cycle)
  - RISC vs. CISC
- **Unit II: Working of 8085 Microprocessor**
  - Pin Diagram of 8085
  - Block Diagram of 8085
    - Registers (Accumulator, Flags, Program Counter, Stack Pointer)
      - Q2a: Explain 16 bits registers of 8085 microprocessor.
    - Internal Architecture
    - The Flag Register
  - Working of the 8085
    - Demultiplexing of Lower Order Address Bus & Data Bus
      - Q2a: Explain function of ALE signal with diagram.
    - Instruction Fetching, Decoding and Execution
      - Instruction Fetching
      - Instruction Decoding
      - Instruction Execution
      - Example: ADD B Instruction
  - Microprocessor vs. Microcontroller
- **Unit III: Microcontroller Architecture**
  - General Block Diagram of a Microcontroller
  - Pin Diagram of 8051
  - 8051 Microcontroller Block Diagram
    - ALU (Arithmetic Logic Unit) & Timing and Control Unit
    - Instruction Register and Instruction Decoder
    - Accumulator (A)
    - Register B
    - PC (Program Counter)
    - SP (Stack Pointer)
    - DPTR (Data Pointer)
    - Special Function Registers (SFRs)
    - Program Status Word (PSW)
    - Explain clock and reset circuit for 8051 with sketch. (3)
    - I/O Ports
  - I/O Ports structure: Port 0, Port 1, Port2, Port 3.
    - Port-0 Pin Structure:
    - Port 1 Pin Structure
    - Port 2 Pin Structure
    - Port 3 Pin Structure
  - Memory Organization

- Draw and Explain program and data memory of 8051. (4)
- Draw and Explain External Memory Addressing and Decoding Logic of 8051.
- Draw and explain internal RAM architecture of the 8051 microcontroller. (4)

- Stack, Stack Pointer, and Stack Operations
- Timers/Counters
  - TCON Register
  - TMOD Register
- Modes of Operation
  - Explain different timer modes of 8051 microcontroller.
  - Timers/Counters logic diagram and its operation in various modes.
- Serial Communication
  - Modes
  - Explain Serial Communication in various modes.
  - SCON Register
- PCON Register
- Interrupts
  - Q3c: Explain interrupts of 8051 microcontroller.
  - Interrupt Vector Addresses
  - Interrupt structure, vector address, priority and operation.
  - IE Register
- Priorities
  - IP Register

Unit IV: 8051 Programming

- Addressing Modes
  - Immediate addressing mode
  - Register addressing mode
  - Direct addressing mode
  - Indirect addressing mode
  - Indexed addressing mode
  - Relative addressing mode
  - Bit addressing mode
- 8051 Instruction Set
  - Data Transfer Instructions
    - Compare MOV, MOVX and MOVC instruction using one example of each.
  - Arithmetic Instructions
  - Logical Instructions
  - Program Branching Instructions
  - Boolean or Bit-manipulation Instructions
  - Machine Control
- Assembly Language Programming Examples
  - Refer Dedicated Notes for Programming Examples

Unit V: Interfacing & Applications of Microcontroller

- Input Devices
  - Push Button Switches
    - Interface Input Devices with 8051 micro-controller: Switch, Push-button & DIP.
  - LM35 Temperature Sensor
    - Q5c: Interface LM 35 with 8051 microcontroller and explain block diagram of temperature controller.
    - Explain temperature sensor LM35 in brief. (3)
- Output Devices
  - LEDs
    - Q5b: Interface 8 leds with 8051 microcontroller and write a program to turn on and off.
    - Interface Output devices with 8051 microcontroller: LED.
  - 7-Segment Displays
    - Q5b: Interface 7 segment with 8051 microcontroller.
    - Draw block diagram to interface one common anode seven segment LED with port P0 of 8051.

Unit I: Introduction to Microprocessor

Definition and History of Microprocessors

Definition of a Microprocessor

A microprocessor is a single integrated circuit (IC) that incorporates the core functions of a computer's central processing unit (CPU).

- **The "Brain" of a Computer:** It executes instructions, performs calculations, and manages the flow of data within a computer system.
- **Small and Powerful:** Microprocessors pack millions or even billions of transistors into a tiny chip, enabling complex processing in compact devices.
- **Essential for Modern Devices:** They power a vast range of devices from smartphones and laptops to cars, appliances, and industrial equipment.
- **Components:** Typical components of a microprocessor include:
  - Arithmetic Logic Unit (ALU) - Performs arithmetic and logical operations
  - Control Unit (CU) - Decodes instructions and coordinates the operations of other units
  - Registers - Small, high-speed memory locations for temporary data storage
- **Responsible for:**

- **Fetching instructions:** Retrieving instructions from the computer's memory.
- **Decoding instructions:** Translating instructions into a form the microprocessor understands.
- **Executing instructions:** Performing calculations and logical operations based on the instructions.
- **Controlling data flow:** Managing the movement of data between memory, input devices, output devices, and the microprocessor itself.

History

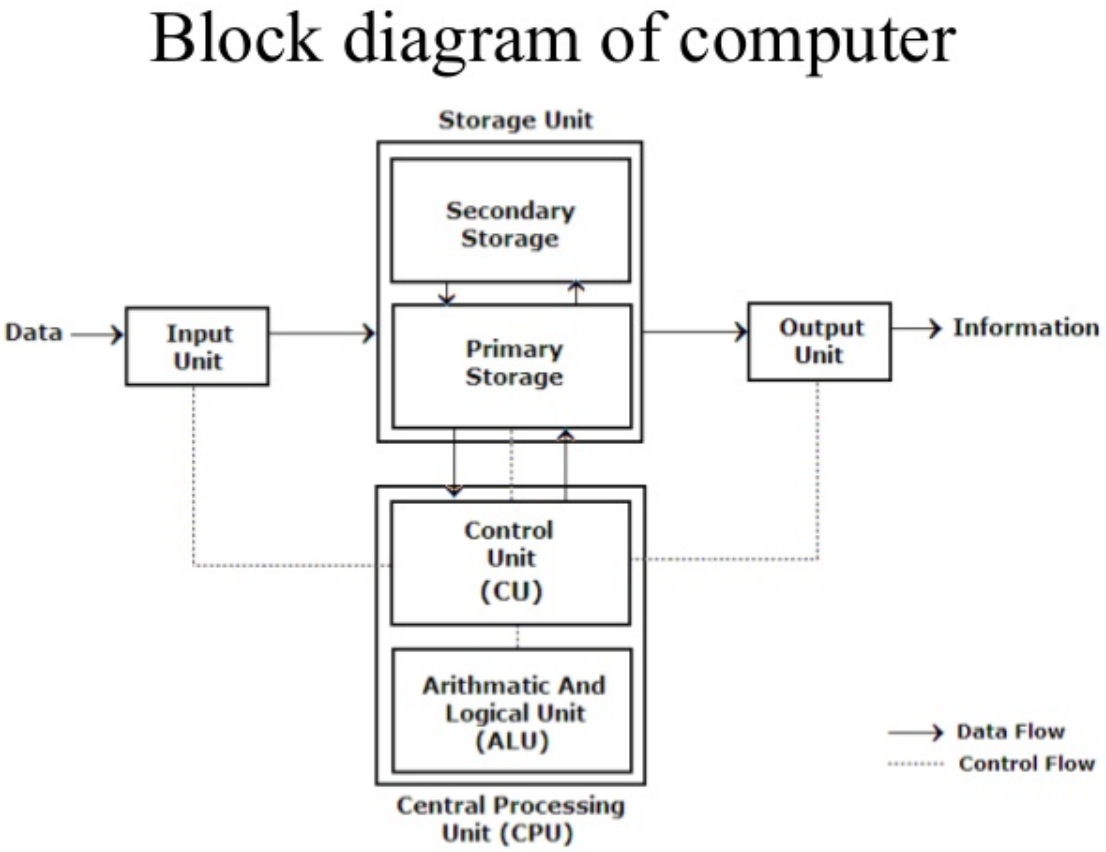
The evolution of microprocessors is a fascinating story of technological advancement:

- **Early Computers (1940s-1950s):** The first computers were massive, filling entire rooms. They used vacuum tubes for processing, which were bulky, power-hungry, and prone to failure.
- **Transistors (1950s-1960s):** The invention of the transistor revolutionized electronics. Transistors were smaller, faster, and more reliable than vacuum tubes, leading to smaller and more powerful computers.
- **Integrated Circuits (1960s):** Integrated circuits (ICs) combined multiple transistors, resistors, and other components onto a single chip. This enabled further miniaturization of computers.
- **The First Microprocessor (1971):** Intel released the Intel 4004, the first commercially available microprocessor on a single chip. While limited in power by today's standards, it paved the way for the computing revolution.
- **Rapid Advancement (1970s-1980s):** This period saw exponential growth in microprocessor performance with the introduction of iconic processors like the Intel 8080, Zilog Z80, and Motorola 6800. These processors found their way into the first personal computers.
- **Modern Era (1990s-Present):** Microprocessors have become incredibly powerful, with billions of transistors on a single chip. They power not only our computers but also smartphones, tablets, smart devices, cars, and countless other technologies.

Key milestones in microprocessor history:

- **1971:** Intel 4004 (4-bit)
- **1974:** Intel 8080 (8-bit)
- **1978:** Intel 8086 (16-bit) - foundation of the x86 architecture used in many PCs today.
- **1993:** Intel Pentium (32-bit) – brought significant performance gains
- **2000s:** Introduction of multi-core processors
- **Present:** Continued focus on performance, power efficiency, and specialized microprocessors for tasks like AI and machine learning.

Basic Components of a Digital Computer



A digital computer is a versatile device capable of performing calculations and logical operations at incredible speeds. To achieve this, a computer relies on several fundamental components working together:

- **Input Unit:** This unit bridges the gap between the user and the computer. It allows data and instructions to be entered into the system. Some common input devices include:
    - Keyboard
    - Mouse
    - Touchscreen
    - Scanner
    - Microphone
- **Storage Unit:** The storage unit preserves data, instructions, and results for short-term and long-term use. It's divided into two main categories:
    - **Primary Storage (Main Memory):** This fast, but relatively expensive memory temporarily holds the currently running programs, input data, and intermediate calculations. Since primary storage is volatile, data is lost when the computer powers down. RAM (Random Access Memory) is the most common type of primary storage.
    - **Secondary Storage (Auxiliary Memory):**

      This type of storage acts as a permanent repository for programs, data, and the operating system. It's slower than primary storage but offers larger capacity at a lower cost. Examples include:
        - Hard Disk Drives (HDD)
        - Solid-State Drives (SSD)

        - Optical Disks (CDs, DVDs)
- **Central Processing Unit (CPU):** The CPU is the "brain" of the computer responsible for controlling and executing instructions. It contains two primary parts:
    - **Control Unit (CU):** The orchestrator of the CPU. It fetches instructions from memory, decodes them, and generates signals to coordinate the activities of the other components within the system.
    - **Arithmetic Logic Unit (ALU):** The heart of calculations. The ALU performs arithmetic operations (addition, subtraction, etc.) and logical operations (AND, OR, NOT, etc.).
- **Output Unit:** The output unit presents the results of processing to the user in a human-readable form. Examples include:
    - Monitor (display)
    - Printer
    - Speakers

How These Components Work Together

1. **Input:** A user enters data or instructions through an input device like a keyboard or mouse.
2. **Storage:** Data and instructions are temporarily stored in the main memory (RAM) for quick access by the CPU.
3. **Processing:**
    - The Control Unit fetches an instruction from memory and decodes it.
    - The ALU executes the instruction, potentially involving calculations or logical comparisons.
    - Results might be stored back into memory (RAM or secondary storage).
4. **Output:** The processed results are presented to the user through an output device, such as a monitor or printer.

Basic Components of a Microprocessor

CPU (Central Processing Unit)

- **The Brain:** The CPU is the heart of a microprocessor, responsible for interpreting and executing instructions. Think of it as the decision-maker and coordinator of the entire system.
- Key Components:
    - **Control Unit (CU):** The manager that fetches instructions from memory, decodes them, and controls the flow of data and operations throughout the processor.
    - **Arithmetic Logic Unit (ALU):** The "calculator" within the CPU that performs all arithmetic (addition, subtraction, etc.) and logical (AND, OR, NOT, etc.) operations.

ALU (Arithmetic and Logic Unit)

- **The Calculator:** The ALU is a core part of the CPU, dedicated to carrying out the calculations and logic comparisons that drive computations within the microprocessor.
- Operations:
    - Arithmetic: Addition, subtraction, multiplication, division, etc.
    - Logical: AND, OR, XOR, NOT, comparisons, etc.

Control Unit

- **The Orchestrator:** The control unit is another essential part of the CPU. It directs all operations within the microprocessor.
- Responsibilities:
    - **Instruction Fetching:** Retrieves instructions from memory.
    - **Instruction Decoding:** Interprets instructions to determine what needs to be done.
    - **Control Signals:** Generates signals to coordinate the ALU, memory, and other components, ensuring everything works in sync.

Memory Unit (RAM, ROM)

- **Data and Code Storage:** The memory unit is where the microprocessor stores important data and instructions.
- Types:
    - **RAM (Random Access Memory):** Temporary, fast storage used for currently running programs and data. It's volatile, meaning data disappears when the power goes off.
    - **ROM (Read-Only Memory):** Permanent storage that typically holds the computer's startup instructions (BIOS) and other essential data that shouldn't change.

Input/Output (I/O) Units

- **Communication Bridge:** These units facilitate communication between the microprocessor and the outside world.
- Input Devices:
    - Keyboard
    - Mouse
    - Scanner
    - Microphone
    - Network interface card
- Output Devices:
    - Monitor
    - Printer
    - Speakers
    - Network interface card

How It All Works Together

1. **Fetch:** The Control Unit fetches an instruction from memory (RAM).
2. **Decode:** The Control Unit decodes the instruction to figure out the required operation.
3. **Execute:**
    - If it's a calculation or logical operation, the ALU gets involved.
    - Data may be moved between memory, the ALU, and internal registers (tiny, super-fast memory within the CPU).
4. **Store:** Results might be written back to memory or sent to an output device.

Architectures

Von Neumann Architecture

- **Key Features:**
  - **Single Unified Memory:** Both instructions and data reside in the same memory space.
  - **Single Bus:** A shared bus is used for transferring both data and instructions.
  - **Sequential Execution:** Instructions are fetched and executed one at a time.
- **Components:**
  - Central Processing Unit (CPU) with Control Unit (CU) & Arithmetic Logic Unit (ALU)
  - Unified Memory
  - Input/Output (I/O) devices
  - Bus
- **Advantages:**
  - **Simplicity:** Easier to design and implement.
  - **Flexibility:** Programs can modify their own code, enabling dynamic behavior.
- **Limitations:**
  - **The von Neumann Bottleneck:** Limited bandwidth due to the shared data and instruction bus, potentially slowing down processing.
  - **Security Concerns:** Less separation between code and data can increase vulnerability to some types of cyber-attacks.

Harvard Architecture

- **Key Features:**
  - **Separate Memories:** Distinct memory units for instructions and data.
  - **Dedicated Buses:** Separate buses for fetching instructions and accessing data.
  - **Parallel Execution:** The CPU can access instructions and data simultaneously.
- **Components:**
  - Central Processing Unit (CPU) with Control Unit (CU) & Arithmetic Logic Unit (ALU)
  - Separate instruction and data memory
  - Dedicated buses for each memory
  - Input/Output (I/O) devices
- **Advantages:**
  - **Speed and Efficiency:** Parallel access offers faster execution and eliminates the bottleneck present in Von Neumann architectures.
  - **Enhanced Security:** Improved isolation between code and data can aid security measures.
  - **Deterministic Behavior:** Reliable timing and performance make it ideal for real-time systems.
- **Limitations:**

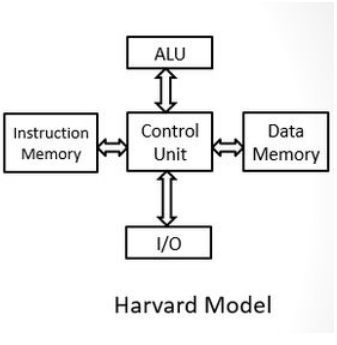
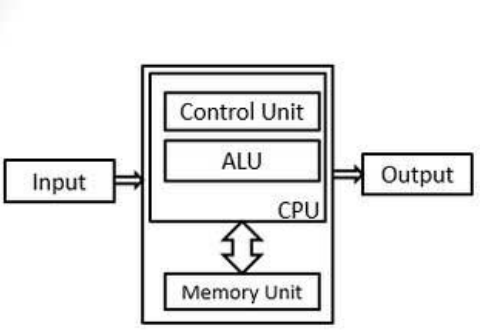
- **Increased Complexity:** More complex to design due to additional memory units and buses.
- **Less Flexible for Self-Modifying Code:** Separating code and data makes it more difficult for programs to modify their instructions on the fly.

Von Neumann vs Harvard Architecture

Similarities

- **Fundamental Components:** Both architectures include a CPU (with CU and ALU), memory, and I/O devices.
- **Stored-Program Concept:** Both can store programs in memory and execute them.

Differences Summary Table

Feature	Von Neumann Architecture	Harvard Architecture
Memory Structure	Single unified memory for instructions and data	Separate memory spaces for instructions and data
Buses	Single bus for instructions and data	Separate, dedicated buses for instructions and data
Instruction Processing	Sequential	Potential for parallel instruction fetch and data access
Performance	Potential bottleneck due to shared bus	Faster, eliminates the bottleneck
Complexity	Simpler to design and implement	Increased hardware complexity
Flexibility	Programs can self-modify code	Less flexible for self-modifying code
Security	Less isolation between code and data	Improved isolation
Applications	General-purpose computers, laptops, servers.	Embedded systems, microcontrollers, digital signal processors (DSPs).
Diagram	 Harvard Model	 Von Neumann Model

Instruction Formats & Related Terms

Instruction Format

A microprocessor instruction is a fundamental command encoded in binary that tells the microprocessor to perform a specific operation. Instructions generally have two core parts:

Opcode (Operation Code)

Specifies the operation the microprocessor should perform (e.g., add, subtract, move data, compare). The opcode is a unique binary pattern assigned to a particular action.

Operand

Data the operation acts upon. An operand could be:

- **Immediate Value:** Data directly included in the instruction itself.
- **Register:** A small, fast memory location inside the processor.
- **Memory Address:** A location in the main memory.

Example

Consider a simple 'ADD' instruction in a hypothetical microprocessor:

```
ADD  R1,  #5
```

- **Opcode:** 'ADD' tells the processor to perform an addition operation.
- **Operands**:
  - 'R1' is a register, indicating one value for the addition is stored in register R1.
  - '#5' is an immediate value, specifying the second value for the addition.

Instruction Cycle

The instruction cycle is the complete sequence of steps a microprocessor takes to process a single instruction. It involves:

1. **Fetch:** The Control Unit retrieves the instruction's opcode from memory.
2. **Decode:** The Control Unit decodes the opcode to understand the required operation.
3. **Execute**:

   The instruction is carried out. This may involve:
   - Reading data from memory or registers.
   - Performing calculations or logical operations in the ALU.
   - Writing results back to memory or registers.

Machine Cycle

A machine cycle represents a single, indivisible action performed by the microprocessor necessary to carry out part of an instruction's operation. Some examples of machine cycles include:

- **Memory Read:** Fetching data from memory.

- **Memory Write:** Storing data into memory.
- **I/O Read:** Reading data from an input device.
- **I/O Write:** Sending data to an output device.

An instruction cycle often comprises multiple machine cycles.

T-State (Clock Cycle)

A T-state is the fundamental unit of time in a microprocessor, measured by a single period of the processor's internal clock. Each machine cycle typically takes one or more T-states. Faster clocks mean more T-states per second, facilitating faster processing.

Relationship

- **Instructions** are built from opcodes and operands.
- An **instruction cycle** consists of the steps to execute one complete instruction.
- A **machine cycle** is a smaller unit of action within an instruction cycle.
- **T-States** are the fundamental timing unit, with a machine cycle usually encompassing multiple T-states.

Instructions in the 8085 microprocessor can be 1, 2, or 3 bytes long. The structure varies depending on the specific instruction and the addressing modes used.

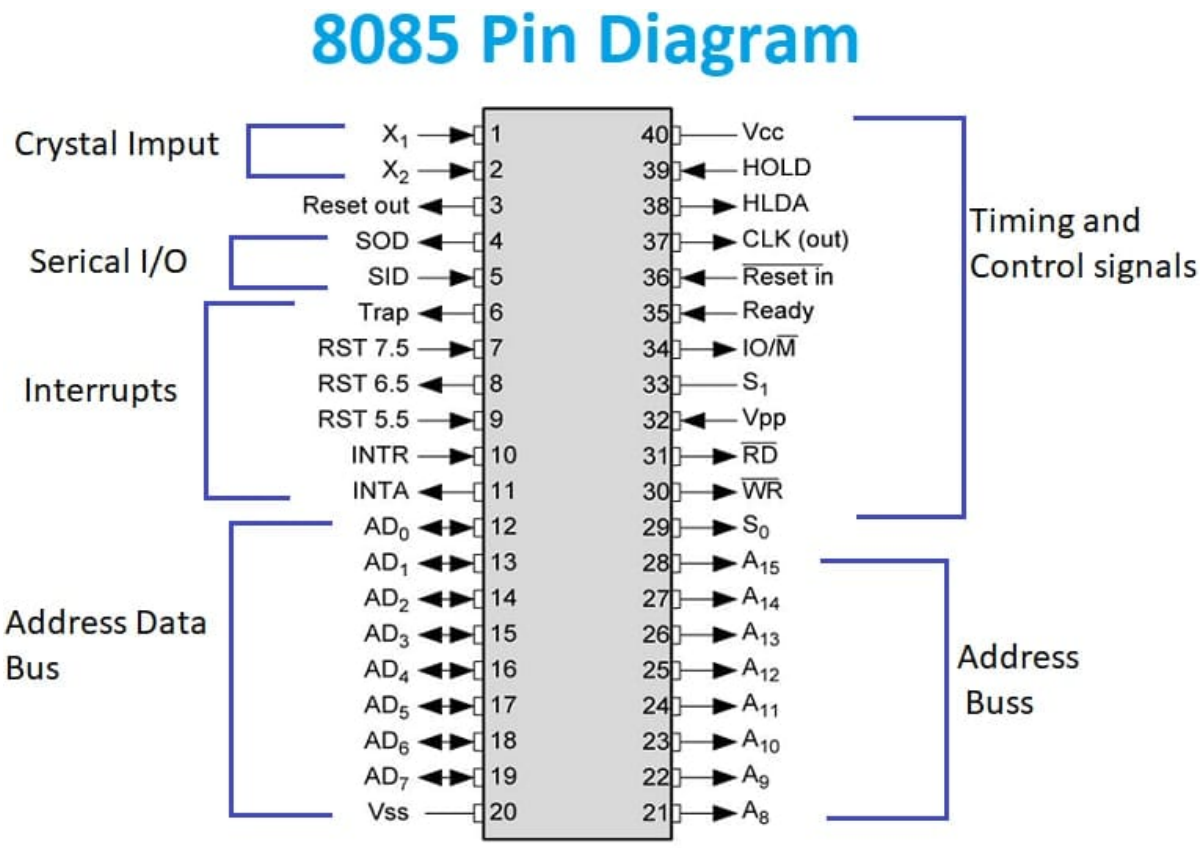
RISC vs. CISC

Feature	RISC (Reduced Instruction Set Computer)	CISC (Complex Instruction Set Computer)
Instruction Set	Smaller, simpler instructions. Focus on individual operations.	Larger, more complex instructions capable of multiple operations within a single command.
Addressing Modes	Limited addressing modes.	Extensive addressing modes for flexible data access.
Execution	Emphasizes hardware optimization. Instructions often execute in one clock cycle.	Utilizes microcode to implement complex instructions, potentially requiring multiple clock cycles per instruction.
Compiler Design	Relies on simpler instructions, shifting complexity to the compiler.	Simplifies compiler design by offloading complexity to processor hardware.
Memory Access	Load/Store architecture: Data must be explicitly moved between registers and memory for operations.	Instructions can operate directly on memory.
Pipelining	Highly efficient pipelining.	Pipelining can be less efficient due to variable-length instructions.

Feature	RISC (Reduced Instruction Set Computer)	CISC (Complex Instruction Set Computer)
Register Usage	Large number of general-purpose registers for fast operand access.	Fewer registers, often with specialized purposes.
Speed	Simplicity enables faster instruction execution, higher overall throughput.	Compact code due to complex instructions can optimize memory usage.
Power Efficiency	Efficient due to simpler design and execution.	Reduced instruction count can lower power consumption in some cases.
Design Cost	Fewer transistors and simpler design can reduce development time and cost.	More transistors and complex design can increase development time and cost.
Code Size	Simpler instructions may require longer sequences to achieve the same task, increasing program size.	Complex instructions can increase complexity and potential for errors, impacting performance.
Compiler	Burden placed on the compiler to generate efficient code.	Can hide hardware complexity, simplifying software development.
Applications	High-performance computing, smartphones, embedded systems, devices where speed and power efficiency are crucial.	Legacy systems, applications prioritizing code density (smaller program size).

Unit II: Working of 8085 Microprocessor

Pin Diagram of 8085

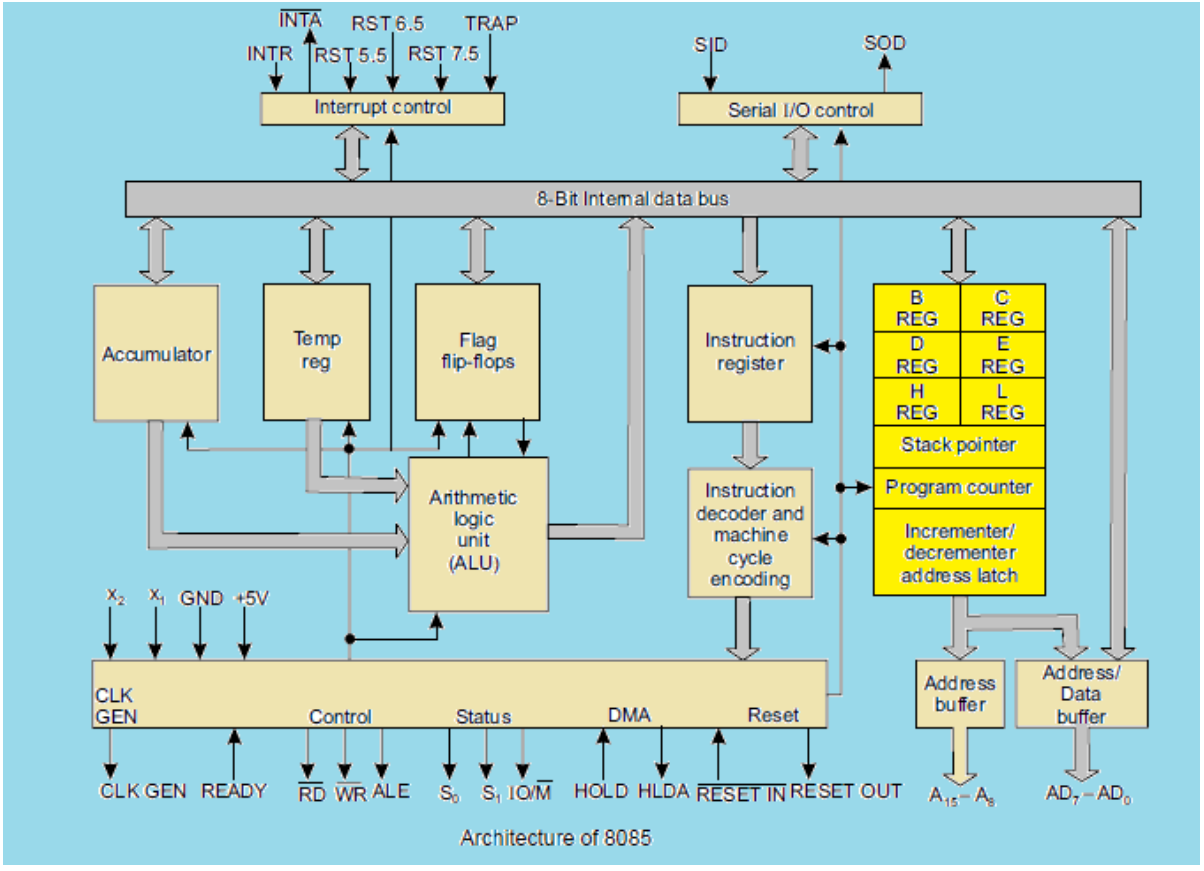


Explanation of Pin Groups

1. **Address Bus (A8-A15):** The upper 8-bits of the 16-bit address bus used for addressing memory and I/O devices.
2. **Multiplexed Address/Data Bus (AD0-AD7):** These pins serve two functions:
    - During the first clock state (T1), they carry the lower 8-bits of the address.
    - During subsequent clock states, they serve as the data bus for data transfer.
3. **Control and Status Signals**
    - **ALE (Address Latch Enable):** Indicates that the AD0-AD7 lines contain a valid address.
    - **RD (Read):** Indicates a read operation from memory or I/O.
    - **WR (Write):** Indicates a write operation to memory or I/O.
    - **IO/M (IO/Memory Select):** Distinguishes between memory (IO/M = 0) and I/O (IO/M = 1) operations.
    - **S0, S1 (Status signals):** These, along with IO/M, indicate the type of machine cycle (opcode fetch, memory read, I/O write, etc.).
4. **Power Supply and Clock**
    - **VCC:** +5V power supply.
    - **VSS:** Ground (0V).
    - **X1, X2:** Connections for a crystal or external clock source to drive the internal clock generator.
    - **CLK (OUT):** Clock output signal for synchronizing external devices.

5. **Interrupts**
    - **TRAP:** Highest priority non-maskable interrupt.
    - **RST 7.5, RST 6.5, RST 5.5:** Maskable interrupts with decreasing priority.
    - **INTR:** General maskable interrupt.
    - **INTA:** Interrupt acknowledge signal sent by the 8085.
6. **Serial I/O**
    - **SID (Serial Input Data):** Input line for serial data.
    - **SOD (Serial Output Data):** Output line for serial data.
7. **Reset**
    - **RESET IN:** When low, resets the microprocessor, clearing the program counter and registers.
    - **RESET OUT:** Indicates that the microprocessor is being reset.
8. **DMA (Direct Memory Access)**
    - **HOLD:** Input from a DMA device to request control of buses.
    - **HLDA:** Acknowledge signal, indicating the 8085 has relinquished control of buses.

Block Diagram of 8085



Architecture of 8085

Key Components and their Functions

1. **Accumulator:** An 8-bit register that's central to arithmetic and logical operations performed by the ALU.
2. **Arithmetic and Logic Unit (ALU):** Performs arithmetic operations (addition, subtraction, etc.) and logical operations (AND, OR, NOT, etc.). It sets flags (Carry, Zero, Sign, etc.) based on the results.

3. **Temporary Register:** A temporary holding location for data used during instruction execution.
4. **Instruction Register:** Holds the currently fetched instruction.
5. **Instruction Decoder and Machine Cycle Encoder:** Decodes the instruction in the instruction register and generates control signals to coordinate the microprocessor's actions during a machine cycle.
6. **Register Array:** Contains six general purpose 8-bit registers (B, C, D, E, H, and L), which can be used individually or in pairs (BC, DE, HL) for 16-bit operations.
7. **Program Counter (PC):** A 16-bit register that holds the memory address of the next instruction to be fetched.
8. **Stack Pointer (SP):** A 16-bit register pointing to the top of the stack in memory. The stack is used for storing return addresses of subroutines and temporarily storing data.
9. **Timing and Control Unit:** Generates timing and control signals for all operations within the microprocessor and synchronizes with external devices.
10. **Interrupt Control:** Handles incoming interrupt requests (if any), acknowledging them and allowing them to temporarily disrupt the current program execution.
11. **Serial I/O Control:** Facilitates serial input and output, useful for slower communication with certain types of peripherals.
12. **Address Bus (A8 - A15):** The upper 8-bits of the 16-bit address bus, used to send the most significant portion of an address.
13. **Address/Data Bus (AD0 - AD7):** A multiplexed bus. It carries the lower 8 bits of an address during the beginning of a machine cycle and data during data transfer operations.

How it Works (Simplified)

1. **Fetch:** The PC provides an address; the instruction is fetched from memory and placed into the Instruction Register.
2. **Decode:** The Instruction Decoder decodes the instruction to understand what needs to be done.
3. **Execute:** The Control Unit generates signals to coordinate the ALU, registers, and other components as they perform the necessary operations.
4. **Repeat:** The process continues, fetching and executing instructions sequentially.

Registers (Accumulator, Flags, Program Counter, Stack Pointer)

Q2a: Explain 16 bits registers of 8085 microprocessor.

16-Bit Registers in the 8085

The 8085 microprocessor, while primarily an 8-bit processor, features several 16-bit registers that are crucial for memory addressing and specific operations:

- **Program Counter (PC):**
    - Holds the 16-bit memory address of the next instruction to be fetched and executed by the processor.
    - Essential for maintaining the correct sequence of program execution.

- **Stack Pointer (SP):**
    - Points to the current top of the stack in memory.
    - The stack is a Last-In, First-Out (LIFO) data structure used for storing return addresses during subroutine calls, temporary data, and interrupt handling.
- **Register Pairs (BC, DE, HL):**
    - While B, C, D, E, H, and L are individual 8-bit registers, they can be paired together to form 16-bit registers:
        - BC
        - DE
        - HL
    - These register pairs allow for operations on 16-bit data and for holding 16-bit memory addresses.

Key Functions of 16-bit Registers

1. **Memory Addressing:** The 8085 has a 16-bit address bus, meaning it can address up to 64KB of memory. The 16-bit registers are used to store and manipulate memory addresses for data storage and retrieval.
2. **Subroutine Calls and Returns:** When a subroutine is called (using instructions like CALL), the processor needs to store the address where it should return to after the subroutine is finished. The Program Counter is pushed onto the stack for safekeeping.
3. **Data Manipulation:** Some instructions treat these register pairs as a single unit for performing 16-bit operations (e.g., addition, loading immediate 16-bit values).

Internal Architecture

The Flag Register

The Flag register in the 8085 is an 8-bit register, with only 5 bits actively used as flags. These flags act as individual flip-flops that are set (1) or reset (0) to reflect specific conditions arising from arithmetic, logical, and other operations performed by the ALU (Arithmetic and Logic Unit).

The 5 Flags:

1. **Sign Flag (S):**
    - Set (1) if the result of an operation is negative (the Most Significant Bit, or MSB, of the result is 1).
    - Reset (0) if the result is positive.
2. **Zero Flag (Z):**
    - Set (1) if the result of an operation is zero.
    - Reset (0) if the result is not zero.
3. **Auxiliary Carry Flag (AC):**
    - Set (1) if there is a carry-out from the lower nibble (lower 4 bits) into the upper nibble (upper 4 bits) of a result.
    - Used primarily in instructions that perform decimal arithmetic.
4. **Parity Flag (P):**

    - Set (1) if the result has even parity (contains an even number of 1s).
    - Reset (0) if the result has odd parity.
5. **Carry Flag (CY):**
    - Set (1) if there is a carry-out from the most significant bit (MSB) of a result during addition, or a borrow during subtraction.
    - Reset (0) otherwise.

How the Flags are Used:

- **Conditional Jumps:** Instructions like JZ (Jump if Zero), JNZ (Jump if Not Zero), JC (Jump if Carry), etc. use the status of these flags to determine whether to branch to different parts of the program.
- **Decision Making:** The processor can examine flag states to modify calculations or behaviors based on previous operations.

Example:

```
; Assume the accumulator (A) holds the value 50
SUB B  ; Subtract the value in register B from the accumulator
JZ LABEL  ; If the result is zero, jump to the code section marked as LABEL
```

Working of the 8085

Demultiplexing of Lower Order Address Bus & Data Bus

Why Demultiplexing is Needed

The Intel 8085 utilizes a multiplexed address/data bus to reduce the number of pins required. The lower 8 lines (AD0-AD7) carry two types of information:

1. **Address (during T1 state):** During the first clock cycle of a machine cycle, these lines hold the lower 8 bits of a 16-bit memory or I/O address.
2. **Data (during subsequent states):** In the remaining clock cycles, those same lines transmit or receive the actual data being sent to or from a memory location or I/O device.

Demultiplexing Process

Demultiplexing is the process of separating the address and data information so the 8085 and external devices can operate correctly. Here's how it's achieved:

1. **The ALE Signal:** During the first clock cycle (T1), the 8085 asserts the ALE (Address Latch Enable) control signal. This signal goes high.
2. **External Latch:** An external latch circuit (e.g., 8282 or 74LS373 octal latch) is connected to the AD0-AD7 lines. When the ALE signal goes high, this latch captures and holds the lower 8 bits of the address.
3. **Address Decoded:** The latched lower-order address bits, along with the higher-order address bits (A8-A15), provide the complete 16-bit address for memory or I/O devices.
4. **Data Bus Freed:** After the T1 state, the ALE signal goes low. The AD0-AD7 lines are now free to be used as a data bus for the remainder of the machine cycle.

Diagram

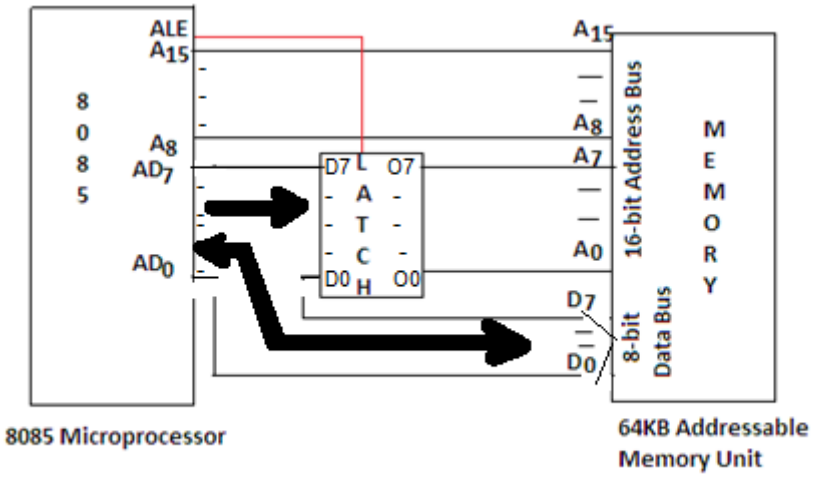


Figure -2 : De-Multiplexing the Address and Data Bus

Key Points

- Demultiplexing enables the 8085 to interface with memory and I/O devices correctly by separating the address and data functions of the same physical bus lines.
- The ALE signal plays a crucial role in timing the latching of address information.

Q2a: Explain function of ALE signal with diagram.

What is the ALE Signal?

- The ALE signal is a control signal generated by the 8085 microprocessor.
- It is a positive-going pulse that occurs during the first clock cycle (T1 state) of each machine cycle.

Purpose of the ALE Signal

The primary function of the ALE signal is to demultiplex the lower-order address/data bus (AD0-AD7). This bus is shared (multiplexed) to carry both:

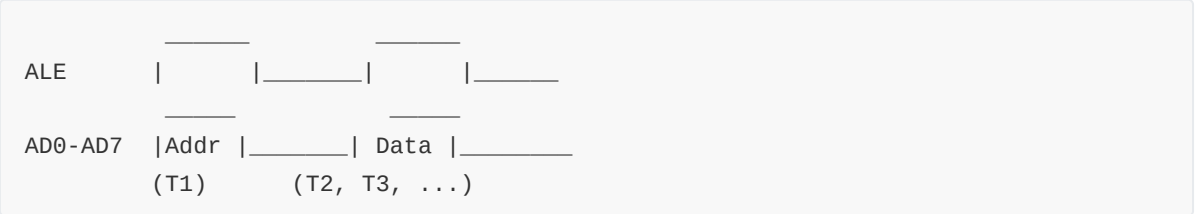
1. **Lower 8-bits of the Address (during T1 state):** The 8085 needs to send out the 16-bit address of a memory location or I/O port. The lower 8 bits of the address are carried on lines AD0-AD7.
2. **Data (during subsequent states):** The same lines are used to transmit or receive actual data to/from the memory or I/O device.

How ALE Demultiplexes the Bus

1. **T1 State:**
    - The ALE signal goes high.
    - The 8085 places the lower 8 bits of the address on lines AD0-AD7.
    - An external latch (usually an 8282 or 8283 octal latch) connected to these lines "latches" or captures this address information.
2. **Subsequent States (T2, T3, ...):**
    - ALE goes low.
    - The lower-order address lines (AD0-AD7) are now free to be used as a data bus for transferring data.

Diagram

A simple timing diagram can help visualize this:



Key Points:

- The ALE signal is crucial for the 8085 to correctly interface with memory and I/O devices.
- The external latch holds the lower order address bits, freeing the 8085 to continue its fetch or write operation.

Instruction Fetching, Decoding and Execution

Instruction Fetching

1. **Program Counter (PC):** The PC, a 16-bit register, holds the memory address of the next instruction to be fetched.
2. **Memory Address Register (MAR):** The contents of the PC are copied into the MAR.
3. **Memory Read:** The 8085's control unit sends a read signal to the memory, and the instruction code at the address specified by the MAR is placed on the data bus.
4. **Instruction Register (IR):** The instruction code is transferred from the data bus to the Instruction Register.
5. **PC Increment:** The PC is incremented to point to the next instruction in memory.

Instruction Decoding

1. **Instruction Decoder:** The instruction code in the IR is interpreted by the 8085's instruction decoder circuitry. It identifies the specific operation to be performed (opcode) and the operands involved.
2. **Control Signals:** The instruction decoder generates appropriate control signals to coordinate the upcoming execution. These signals control the flow of data within the 8085, directing the ALU, registers, and the timing of operations.

Instruction Execution

The execution phase varies significantly depending on the specific instruction. Here's a general breakdown of the kinds of steps involved:

- **Operand Fetching:** If the instruction uses operands (data), additional machine cycles may be involved in fetching these from either:
    - **Registers:** Accessed directly within the microprocessor.
    - **Memory:** The MAR is loaded with the memory address of the operand, and another memory read operation is performed.
- **ALU Operations:** For arithmetic or logical instructions, the ALU is engaged to perform the required calculation or comparison.
- **Result Storage:** The results of an operation may be stored in:
    - **Accumulator:** A special register within the 8085.

    - **Other General-Purpose Registers**
    - **Memory:** Another memory write operation might be needed.
- **Update Status Flags:** The ALU sets flags (Zero, Carry, Sign, etc.) to reflect the results of its operations, which can be used for conditional branching later.

Example: ADD B Instruction

Let's assume the instruction "ADD B" (add the value in register B to the accumulator) is being executed:

1. **Fetch:** The opcode for ADD B is fetched from memory and placed in the IR.
2. **Decode:** The instruction decoder determines that this is an addition operation and that the operand is in register B.
3. **Execute:**
    - The contents of register B are fetched.
    - The ALU performs the addition between the accumulator's current value and the value from register B.
    - The result is stored back into the accumulator.

Instruction Fetching

1. **Program Counter (PC):** The PC, a 16-bit register, holds the memory address of the next instruction to be fetched.
2. **Memory Address Register (MAR):** The contents of the PC are copied into the MAR.
3. **Memory Read:** The 8085's control unit sends a read signal to the memory, and the instruction code at the address specified by the MAR is placed on the data bus.
4. **Instruction Register (IR):** The instruction code is transferred from the data bus to the Instruction Register.
5. **PC Increment:** The PC is incremented to point to the next instruction in memory.

Instruction Decoding

1. **Instruction Decoder:** The instruction code in the IR is interpreted by the 8085's instruction decoder circuitry. It identifies the specific operation to be performed (opcode) and the operands involved.
2. **Control Signals:** The instruction decoder generates appropriate control signals to coordinate the upcoming execution. These signals control the flow of data within the 8085, directing the ALU, registers, and the timing of operations.

Instruction Execution

The execution phase varies significantly depending on the specific instruction. Here's a general breakdown of the kinds of steps involved:

- **Operand Fetching:** If the instruction uses operands (data), additional machine cycles may be involved in fetching these from either:
    - **Registers:** Accessed directly within the microprocessor.
    - **Memory:** The MAR is loaded with the memory address of the operand, and another memory read operation is performed.

- **ALU Operations:** For arithmetic or logical instructions, the ALU is engaged to perform the required calculation or comparison.
- **Result Storage:** The results of an operation may be stored in:
    - **Accumulator:** A special register within the 8085.
    - **Other General-Purpose Registers**
    - **Memory:** Another memory write operation might be needed.
- **Update Status Flags:** The ALU sets flags (Zero, Carry, Sign, etc.) to reflect the results of its operations, which can be used for conditional branching later.

Example: ADD B Instruction

Let's assume the instruction "ADD B" (add the value in register B to the accumulator) is being executed:

1. **Fetch:** The opcode for ADD B is fetched from memory and placed in the IR.
2. **Decode:** The instruction decoder determines that this is an addition operation and that the operand is in register B.
3. **Execute:**
    - The contents of register B are fetched.
    - The ALU performs the addition between the accumulator's current value and the value from register B.
    - The result is stored back into the accumulator.

Microprocessor vs. Microcontroller

Core Distinction

- **Microprocessor:** A Central Processing Unit (CPU) on a chip. It's the "brain" of a computer system, designed for general-purpose computing and requires external components to form a functional system.
- **Microcontroller:** A self-contained "computer-on-a-chip." It integrates a CPU, memory, and peripherals, optimized for embedded control applications.

Key Features

Feature	Microprocessor	Microcontroller
System Design	Core of a complex system	Often the entire system
Complexity	Less complex internally	More complex internally due to integrated components
Instruction Set	Larger, versatile instruction set for diverse operations	Smaller, tailored instruction set for specific applications
Memory	External RAM, ROM, flash required	On-chip RAM, ROM, often with flash memory
Peripherals	Requires external interfacing	Built-in peripherals (timers, ADCs, DACs, communication ports)

Feature	Microprocessor	Microcontroller
Power Consumption	Generally higher power consumption	Optimized for low power operation
Cost	Generally lower cost	Can be higher due to integrated components
Flexibility	Highly flexible for various tasks	More specialized, less adaptable to diverse use cases
Applications	Desktop computers, laptops, servers, complex systems	Embedded systems, appliances, medical devices, IoT devices
Examples	Intel Core Series, AMD Ryzen, IBM Power	Atmel AVR, PIC, ARM Cortex-M, Texas Instruments MSP430

Additional Considerations

- **Programming:** Microcontrollers often require more low-level knowledge of hardware for efficient programming.
- **Performance:** Microprocessors generally excel in raw computational performance, while microcontrollers prioritize power efficiency and responsiveness.
- **Bit Handling:** Microcontrollers frequently offer better support for bit-level operations on I/O pins.

Illustrative Analogy

Imagine building a custom robot:

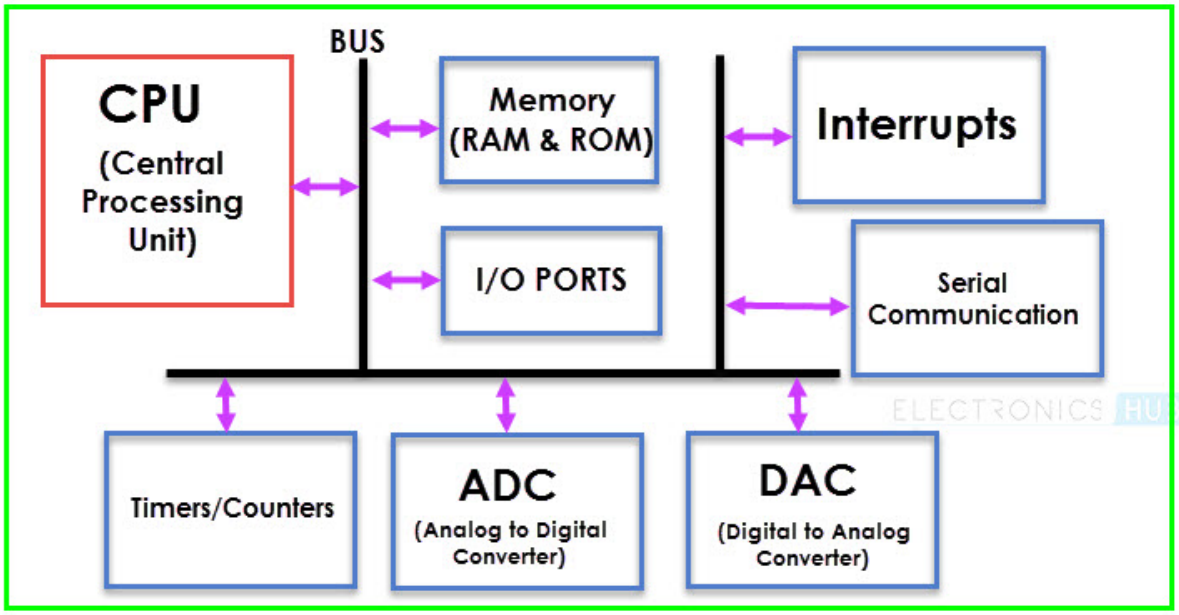
- **Microprocessor:** Like buying the high-performance brain for your robot. You'd still need to buy sensors, motors, a power supply, and design the entire body.
- **Microcontroller:** Like buying a pre-assembled robot kit with a basic brain, sensors, and motors. You focus on programming behavior, potentially adding some external components if needed.

When to Choose Which

- **Microprocessor:** Need high computational power, flexibility for a variety of tasks, or working with large amounts of data.
- **Microcontroller:** Self-contained solution, low-power, real-time control, or cost-sensitive applications are priorities.

Unit III: Microcontroller Architecture

General Block Diagram of a Microcontroller



Understanding the Core Components and Their Interconnections

- **CPU (Central Processing Unit)**
  - **ALU (Arithmetic Logic Unit):** The heart of calculations and logic operations. Performs arithmetic (addition, subtraction, etc.), logical (AND, OR, NOT) comparisons, and data manipulation.
  - **Control Unit (CU):** The manager that fetches instructions from memory, decodes them, and generates control signals to orchestrate the actions of all components within the microcontroller.
  - **Registers:** Small, incredibly fast memory units built into the CPU for temporarily storing data, instructions, and intermediate results.
- **Memory**
  - **ROM (Read-Only Memory):** Non-volatile memory used to permanently store the microcontroller's firmware (program code) and essential data.
  - **RAM (Random Access Memory):** Volatile memory for storing temporary data and variables used while the microcontroller is running a program.
  - **Flash Memory:** In modern microcontrollers, flash memory often replaces traditional ROM for its flexibility, offering reprogrammable program memory.
- **I/O Ports (Input/Output Ports):** Versatile pins that can be configured as either inputs or outputs.
  - **Inputs:** Connect to external sensors, switches, keypads, and other input devices that provide information to the microcontroller.
  - **Outputs:** Drive LEDs, displays, motors, actuators and other devices for interacting with the external world.
- **Bus System:** A communication network of wires connecting the CPU, memory, I/O ports, and other components.
  - **Address Bus:** Carries memory addresses to specify locations for data transfer.

    - **Data Bus:** Transfers data between the various components.
    - **Control Bus:** Transmits control signals (read/write, timing, etc.) to synchronize operations.
- **Timers/Counters:** Specialized modules for:
    - **Generating Accurate Time Delays:** Essential for controlling the timing of operations.
    - **Counting External Events:** Measuring frequencies, pulse widths, and more.
    - **Output Waveform Generation:** Like Pulse Width Modulation (PWM) for motor control, dimming LEDs, etc.
- **Serial Port:** Facilitates communication with other devices using standard protocols like:
    - **UART:** Universal Asynchronous Receiver/Transmitter (common for simple serial communication).
    - **SPI, I2C:** For interfacing with various sensors and peripherals.
- **Interrupts:** Signals that allow the microcontroller to respond to high-priority events immediately
    - **External Interrupts:** Triggered by changes on input pins (e.g., button presses).
    - **Internal Interrupts:** Generated by timers, ADCs, or other peripherals.
- **ADC (Analog to Digital Converter):** Converts incoming analog signals (e.g., from temperature sensors) into digital values that the CPU can process.
- **DAC (Digital to Analog Converter):** Transforms digital data from the CPU into analog signals useful for driving devices like speakers or creating smooth control voltages.
- **On-Chip Oscillator:** Provides the internal clock signal that drives the microcontroller's timing.
- **Special Function Registers (SFRs):** Registers dedicated to controlling the configuration and operation of the microcontroller's peripherals.

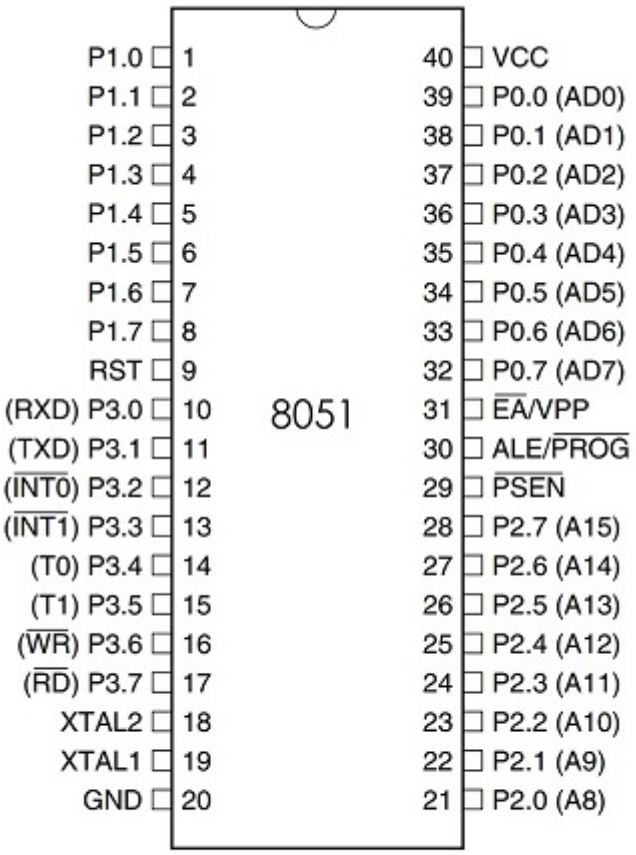
Pin Diagram of 8051

The following image shows the 8051 Microcontroller Pin Diagram with respect to a 40 – pin Dual In-line Package (DIP).

Pins 1 – 8 (PORT 1):

- Pins 1 to 8 are the PORT 1 Pins of 8051. PORT 1 Pins consists of 8 – bit bidirectional Input / Output Port with internal pull – up resistors. In older 8051 Microcontrollers, PORT 1 doesn’t serve any additional purpose but just 8 – bit I/O PORT.
- In some of the newer 8051 Microcontrollers, few PORT 1 Pins have dual functions. P1.0 and P1.1 act as Timer 2 and Timer 2 Trigger Input respectively.
- P1.5, P1.6 and P1.7 act as In-System Programming Pins i.e. MOSI, MISO and SCK respectively.

Pin 9 (RST): Pin 9 is the Reset Input Pin. It is an active HIGH Pin i.e. if the RST Pin is HIGH for a minimum of two machine cycles, the microcontroller will be reset. During this time, the oscillator must be running.



40 - PIN DIP

Pins 10 – 17 (PORT 3): Pins 10 to 17 form the PORT 3 pins of the 8051 Microcontroller. PORT 3 also acts as a bidirectional Input / Output PORT with internal pull-ups. Additionally, all the PORT 3 Pins have special functions. The following table gives the details of the additional functions of PORT 3 Pins.

PORT 3 Pin	Function	Description
P3.0	RXD	Serial Input
P3.1	TXD	Serial Output
P3.2	INT0	External Interrupt 0
P3.3	INT1	External Interrupt 1
P3.4	T0	Timer 0
P3.5	T1	Timer 1
P3.6	WR	External Memory Write
P3.7	RD	External Memory Read

Pins 18 & 19: Pins 18 and 19 i.e. XTAL 2 and XTAL 1 are the pins for connecting external oscillator. Generally, a Quartz Crystal Oscillator is connected here.

Pin 20 (GND): Pin 20 is the Ground Pin of the 8051 Microcontroller. It represents 0V and is connected to the negative terminal (0V) of the Power Supply.

Pins 21 – 28 (PORT 2): These are the PORT 2 Pins of the 8051 Microcontroller. PORT 2 is also a Bidirectional Port i.e. all the PORT 2 pins act as Input or Output. Additionally, when external memory is interfaced, PORT 2 pins act as the higher order address byte. PORT 2 Pins have internal pull-ups.

Pin 29 (PSEN): Pin 29 is the Program Store Enable Pin (PSEN). Using this pins, external Program Memory can be read.

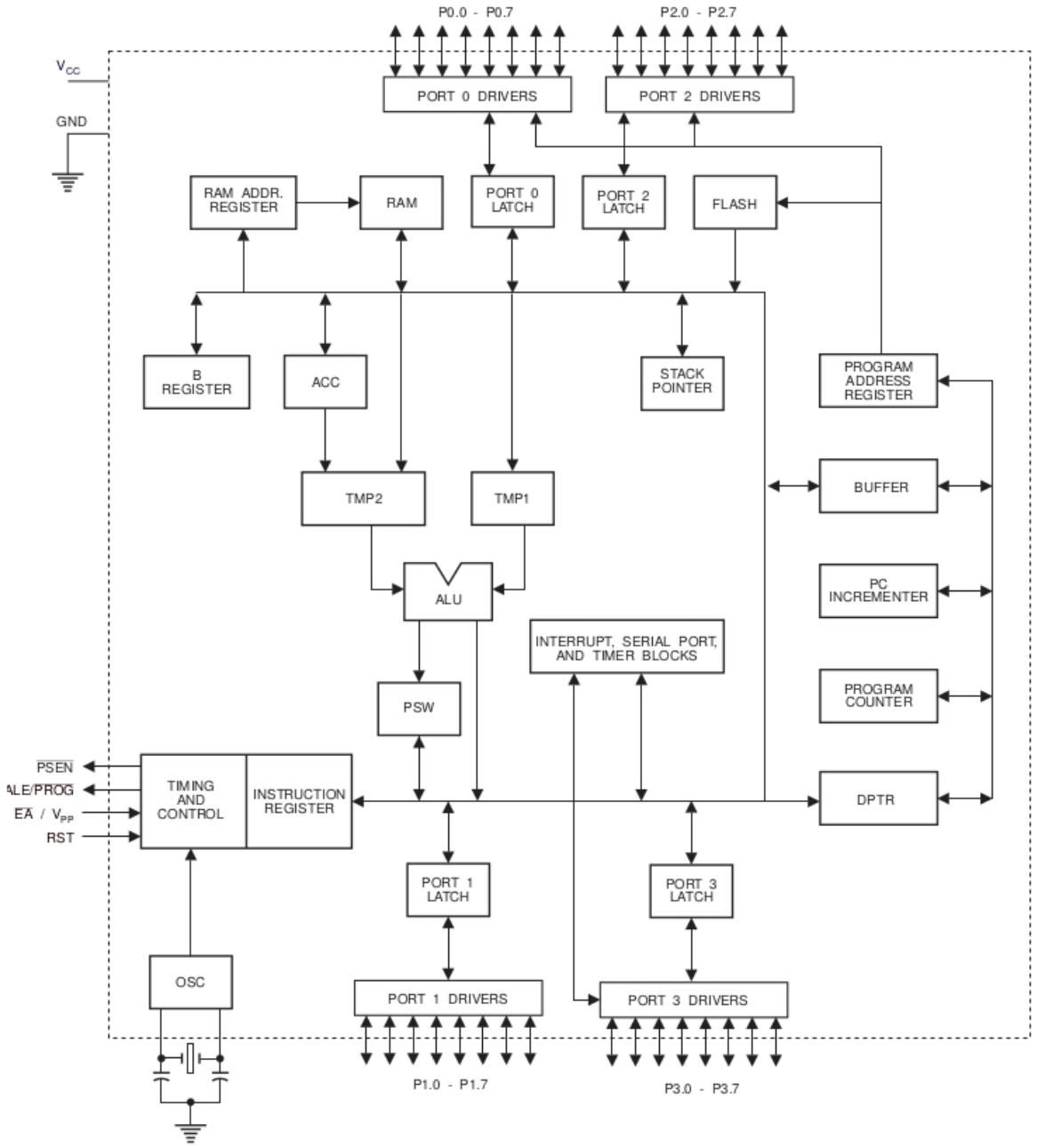
Pin 30 (ALE/PROG): Pin 30 is the Address Latch Enable Pin. Using this Pins, external address can be separated from data (as they are multiplexed by 8051). During Flash Programming, this pin acts as program pulse input (PROG).

Pin 31 (EA/VPP): Pin 31 is the External Access Enable Pin i.e. allows external Program Memory. Code from external program memory can be fetched only if this pin is LOW. For normal operations, this pins is pulled HIGH. During Flash Programming, this Pin receives 12V Programming Enable Voltage (VPP).

Pins 32 – 39 (PORT 0): Pins 32 to 39 are PORT 0 Pins. They are also bidirectional Input / Output Pins but without any internal pull-ups. Hence, we need external pull-ups in order to use PORT 0 pins as I/O PORT. In addition to acting as I/O PORT, PORT 0 also acts as lower order address/data bus when external memory is accessed.

Pin 40 (VCC): Pin 40 is the power supply pin to which the supply voltage is given (+5V).

8051 Microcontroller Block Diagram



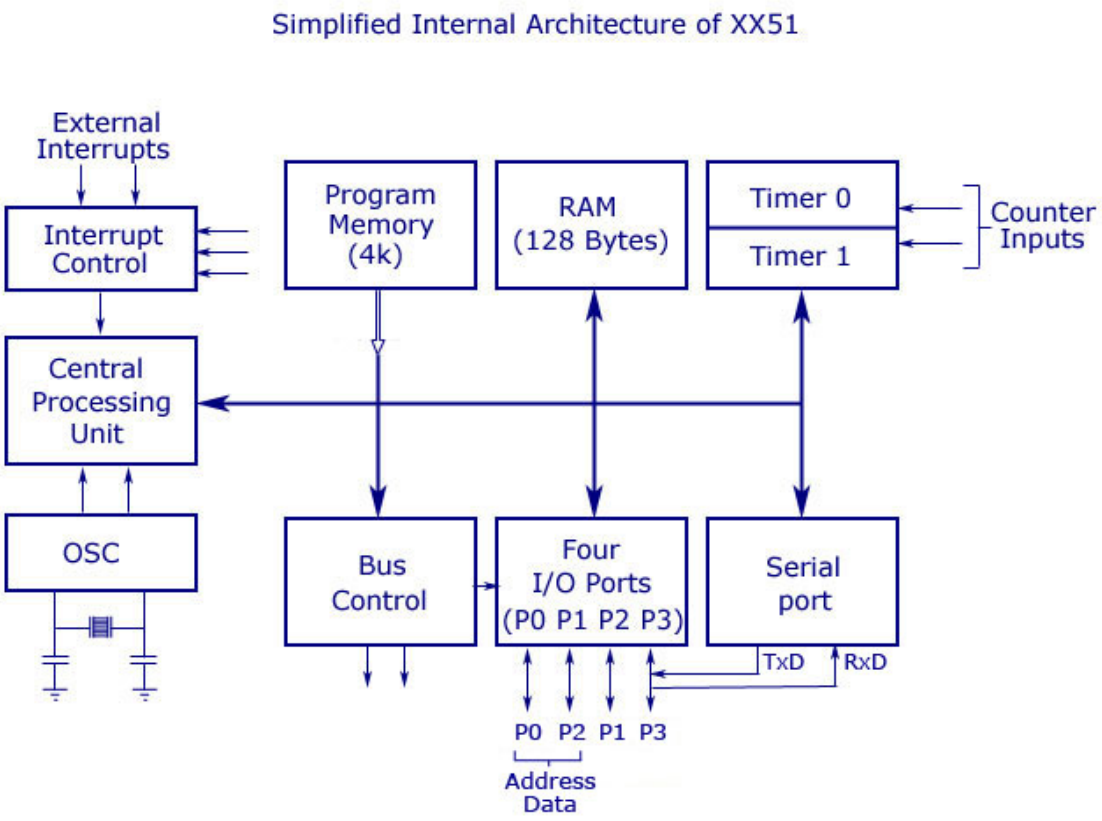
Central Processing Unit (CPU)

- **ALU (Arithmetic Logic Unit):** The heart of calculations and logic operations. Performs arithmetic (addition, subtraction, etc.), logical (AND, OR, NOT) comparisons, and data manipulation.
- **Instruction Decoder:** Decodes the instruction and generates control signals for other parts of the CPU.
- **Timing and Control Unit:** Manages the fetch-decode-execute cycle of the CPU and synchronizes actions with other blocks.

Registers

- **Accumulator (A):** An 8-bit register used for most arithmetic and logical operations.
- **B Register:** An 8-bit temporary register used for multiplication, division, and other data manipulation.

- **Program Status Word (PSW):** Holds status flags like carry, overflow, parity, and register bank selection.
- **Stack Pointer (SP):** Points to the top of the stack in RAM, used for subroutine calls and data storage.
- **Program Counter (PC):** Keeps track of the memory address of the next instruction to be fetched.
- **DPTR:** This is 16 bit register made up of two 8 bit registers – DPH & DPL. This register is used to point to Internal or External memory location.
- **SFR:** Special Function Registers (SFRs) are special registers that contains control and status bits for Timer/Counter (TCON, TMOD), Interrupts (IE, IP), Serial Communication (SCON) and Power Control (PCON).
- **Instruction Register:** Holds the currently fetched instruction.



Memory

- **Internal RAM (128 bytes):** Stores temporary data and variables during program execution.
  - **Register Banks 0-3:** Four sets of eight 8-bit general-purpose registers.
  - **Bit-addressable area (20h-2Fh):** 16 bytes with individually addressable bits.
  - **General-purpose area (30h-7Fh):** Remaining 80 bytes for data and variables.
- **Internal ROM (typically 4KB):** Stores the program code of the microcontroller.

Input/Output (I/O)

- **Ports 0-3 (P0-P3):** Four 8-bit I/O ports that can be individually configured as input or output pins.

Timers/Counters

- **Timers/Counters 0 and 1 (T0, T1):** 16-bit timers/counters that can be used for various purposes like measuring time intervals, counting external events, and generating waveforms.

Serial Port (UART)

- **TXD (Transmit Data):** Transmits serial data out of the microcontroller.
- **RXD (Receive Data):** Receives serial data into the microcontroller.
- **SBUF (Serial Buffer):** Temporarily stores data during serial transmission or reception.

Interrupts

- **External Interrupts (INT0, INT1):** Triggered by signals on external pins.
- **Timer Interrupts (TF0, TF1):** Triggered when timers overflow or reach a specific value.
- **Serial Interrupts (RI, TI):** Triggered by events related to serial communication.
- **Interrupt Control Logic:** Manages the enabling/disabling, prioritizing, and handling of interrupts.

Additional Notes

- **Bus Structure:** The 8051 uses an internal data bus to connect the CPU, memory, and I/O blocks. Instructions and data flow along this bus under the control of the CPU.
- **Reset:** The RESET input initializes the 8051, setting registers and the Program Counter to their starting states.
- **Oscillator:** The XTAL1 and XTAL2 inputs connect to the crystal and other components that generate the clock signal for the microcontroller.

ALU (Arithmetic Logic Unit) & Timing and Control Unit

8051 ALU (Arithmetic Logic Unit)

- **The Computational Heart:** The ALU is responsible for executing the core calculations and logical manipulations within the 8051 microcontroller. As its name suggests, it handles two primary types of operations:
- **Arithmetic Operations**
  - Addition (`ADD`)
  - Subtraction (`SUBB`)
  - Multiplication (`MUL`)
  - Division (`DIV`)
  - Incrementing (`INC`)
  - Decrementing (`DEC`)
- **Logical Operations**
  - AND (`ANL`)
  - OR (`ORL`)
  - Exclusive-OR (`XRL`)
  - NOT (`CLR` , `CPL`)
  - Comparisons (for setting flags in the PSW register)
- **Key Components:**

    - **Accumulator:** Data typically flows in and out of the Accumulator (A Register) during ALU operations.
    - **Temporary Register:** Holds the second operand during calculations.
    - **Arithmetic/Logic Circuits:** The actual hardware circuitry that performs the computations and manipulations.
- **Relationship to PSW:** The ALU interacts closely with the PSW register. ALU operations set flags like Carry (CY), Auxiliary Carry (AC), Overflow (OV), and Parity (P) in the PSW, which then become decision points for control flow in the program.

8051 Timing and Control Unit

- **Conductor of the Orchestra:** The Timing and Control Unit is the mastermind that synchronizes all the actions within the 8051 microcontroller. It ensures everything happens at the right time and in the correct sequence.
- **Key Functions:**
    - **Instruction Fetching and Decoding:** Fetches instructions from memory and interprets their meaning, figuring out what actions need to be performed.
    - **Machine Cycle and State Control:** Divides instructions into smaller steps (machine cycles) and generates timing signals that tell each part of the microcontroller when to do what.
    - **Signal Generation:** Produces the precise electrical pulses that activate registers, the ALU, data buses, and other components within the microcontroller.
    - **Interrupt Handling:** Coordinates what happens when an external event (like a timer overflow or a button press) interrupts the currently executing program.
- **Crystal Oscillator's Role:** The Timing and Control Unit relies on a crystal oscillator, generating a steady clock signal. This clock signal determines the fundamental speed at which the microcontroller operates.

Instruction Register and Instruction Decoder

Instruction Register (IR)

- **Temporary Holding Area:** The Instruction Register is a special, temporary holding space within the microcontroller where the currently fetched instruction resides. It's closely connected to the Instruction Decoder.
- **Size:** The size of the Instruction Register typically matches the width of instructions for the microcontroller. In the case of the 8051, instructions can be 1, 2, or 3 bytes long, so the Instruction Register has to accommodate that.
- **Two Main Parts (often):** In some architectures, the Instruction Register is divided into:
    - **Shift Register:** Shifts in the instruction, bit by bit, as it's retrieved from memory. This process is synchronized with the microcontroller's clock.
    - **Hold Register:** Holds the fully fetched instruction once the shifting is complete, making it available to the Instruction Decoder.

Instruction Decoder

- **The Translator:** The Instruction Decoder is the circuit that analyzes the instruction currently residing in the Instruction Register. It breaks the instruction down into its meaningful components:

- **Opcode:** The opcode (operation code) is the part of the instruction that tells the microcontroller what fundamental operation to perform (ADD, MOV, JUMP, etc.).
- **Operands:** Operands are the pieces of data the instruction acts upon. This could be register names, immediate data (values hard-coded into the instruction), or memory addresses.
- **Decoding Process:** The Instruction Decoder possesses 'knowledge' of all valid instructions in the microcontroller's instruction set. It compares the opcode to this knowledge base to determine:
    - What the operation is
    - What type of operands are involved
    - How many machine cycles are likely needed for execution
- **Control Signal Generation:** The Instruction Decoder produces control signals that activate different parts of the microcontroller, ensuring the correct actions are taken for the specified instruction. These signals will direct things like:
    - Data transfers from registers to the ALU
    - ALU operation selection (add, subtract, etc.)
    - Setting of flags in the PSW register
    - Flow of data to and from memory

The Dance of Fetching, Decoding, and Execution

1. **Fetch:** An instruction is retrieved from program memory, often with the help of the Program Counter.
2. **Load:** The instruction is shifted into the Instruction Register.
3. **Decode:** The Instruction Decoder analyzes the instruction and generates the appropriate control signals.
4. **Execute:** The microcontroller, directed by the control signals, carries out the steps required by the instruction.

Accumulator (A)

- **The Workhorse:** The Accumulator is the central hub for most arithmetic, logical, and data transfer operations within the microcontroller. If you think of the microcontroller as a tiny calculator, the Accumulator is where you see the numbers being entered and the results being displayed.
- **Key Operations:**
    - **Arithmetic:** Addition, subtraction, incrementing, decrementing.
    - **Logical:** AND, OR, XOR, NOT (complements), bit rotations, shifts.
    - **Data Movement:** Transfers data to and from internal RAM or external memory.
- **Special Role in Instructions:** Many instructions in the microcontroller's instruction set implicitly use the Accumulator as either the source of data, the destination for the result, or both.

Register B

- **Versatile Assistant:** The B Register serves as a secondary register, often used to temporarily hold values to assist in calculations or data manipulation.
- **Specialized Tasks:**
  - **Multiplication and Division:** The B Register is essential for the `MUL AB` (multiply) and `DIV AB` (divide) instructions. It holds one of the operands and, in the case of division, stores the remainder of the operation.
  - **Data Manipulation:** It can be used as a temporary holding space for values during complex operations that might involve multiple steps.

Key Points

- **Size:** Both the Accumulator and B Register are usually 8-bit registers. This means they can each store a single byte of data (a value from 0 to 255).
- **Not General Purpose:** Unlike general-purpose registers (like R0, R1, etc. in the 8051), the Accumulator and B Register have more defined roles due to their connection to specific instructions.

PC (Program Counter)

- **The Program's Navigator:** The Program Counter holds the address of the next instruction to be executed by the microcontroller. It's like the microcontroller's bookmark within the program.
- **How it Works:**
  1. **Fetch:** The PC sends its current address to fetch the next instruction from program memory.
  2. **Increment:** By default, the PC is automatically incremented after fetching, preparing it for the following instruction.
  3. **Control Flow Changes:** Instructions like jumps (`JMP`) and calls (`CALL`) can change the PC's value, altering the program's execution order.
- **Size:** The PC is 16 bits wide in the 8051, allowing it to address up to 64KB of program memory.

SP (Stack Pointer)

- **LIFO Storage:** The Stack Pointer points to the current 'top' of the stack. The stack is a last-in, first-out (LIFO) data structure within the 8051's internal RAM.
- **Key Operations:**
  - **PUSH:** Adds data to the top of the stack. The SP is then decremented to point to the new top.
  - **POP:** Removes data from the top of the stack. The SP is incremented as data is removed.
- **Essential for:**
  - **Subroutines (CALL and RET):** Stores the return address when a subroutine is called so the program knows where to resume after the subroutine finishes.

    - **Interrupt Handling:** Stores register values when an interrupt occurs, preserving the state before the interrupt routine is executed.
- **Size:** The SP is 8 bits wide in the 8051.

DPTR (Data Pointer)

- **Accessing External Data:** The DPTR is a special 16-bit register used for addressing external memory (data memory outside the 8051's internal space).
- **Key Functions:**
    - **Indirect Addressing:** The value in DPTR acts as a pointer. Instructions like `MOVX` (move external data) use DPTR to specify the source or destination address in external memory.
    - **Lookup Tables:** DPTR is useful for storing the starting address of tables or data structures located in external memory.

Special Function Registers (SFRs)

What is an SFR?

- **Special Function Registers (SFRs)** are unique memory locations within the 8051 microcontroller's architecture. Unlike general-purpose RAM, SFRs directly control and configure various hardware peripherals and functions of the microcontroller.
- **Location:** They occupy the address space from 80H to FFH within the internal RAM.

Important 8051 SFRs

Here's a breakdown of the most common 8051 SFRs, along with their roles:

1. Accumulator (A)

- **Address:** E0H
- **Function:** The heart of most arithmetic and logical operations in the 8051. It acts as a source or destination for data.

2. Program Status Word (PSW)

- **Address:** D0H
- **Function:** Contains critical flags indicating the status of the microcontroller, including:
    - CY (Carry Flag)
    - AC (Auxiliary Carry Flag)
    - F0 (User-definable flag)
    - RS1, RS0 (Register Bank select bits)
    - OV (Overflow Flag)
    - P (Parity Flag)

3. B Register (B)

- **Address:** F0H
- **Function:** Often used in conjunction with the Accumulator:
    - Multiplication and division operations
    - Temporary storage of data

4. Timer Registers

- **TH0, TL0 (Timer 0):** 98H, 99H
- **TH1, TL1 (Timer 1):** 8AH, 8BH
- **Function:** Generate time delays, count external events, and form the basis of baud rate generation for serial communication.

5. Serial Port Registers

- **SBUF:** 99H
  - Holds the data to be transmitted (write) or received data (read) during serial communication.
- **SCON:** 98H
  - Controls the mode of serial communication (framing, baud rate, etc.).

6. Interrupt Registers

- **IE (Interrupt Enable):** A0H
  - Enables or disables specific interrupts.
- **IP (Interrupt Priority):** B0H
  - Determines the priority level of different interrupt sources.

7. Port Registers (P0, P1, P2, P3)

- **P0:** 80H
- **P1:** 90H
- **P2:** A0H
- **P3:** B0H
- **Function:** Control input and output operations on the 8051's I/O pins.

8. Power Control Register (PCON)

- **Address:** 87H
- **Function:** Manages power-saving modes of the 8051 (idle mode, power-down mode).

Note: The exact set of SFRs can vary slightly depending on the specific 8051 microcontroller variant you are using.

How SFRs Work

You can interact with SFRs in your programs just like regular memory locations, using assembly language instructions or C extensions (like `sfr`, `sfr16`, and `sbit`). By manipulating the values in SFRs, you effectively configure the operation of the 8051.

Program Status Word (PSW)

Address: 0D0H (Bit addressable)							
PSW.7	PSW.6	PSW.5	PSW.4	PSW.3	PSW.2	PSW.1	PSW.0
CY	AC	F0	RS1	RS0	OV	-	P

The PSW register is a vital SFR (Special Function Register) in the functioning of a microcontroller. It reflects the status of the operation that is being carried out in the processor. The PSW register is bit and byte addressable. The physical address of PSW starts from D0H. The individual bits are then accessed using D1, D2 ... D7.

Bit	Description
CY	Carry - Is set if data is coming out of bit 7 of Acc during an Arithmetic operation.
AC	Auxiliary carry - This bit is set if data is coming out from bit 3 to bit 4 of Acc during an Arithmetic operation.
F0	Flag 0 - User defined flag
RS1, RS0	Register Bank select bits
OV	Overflow - OV flag is set if there is a carry from bit 6 but not from bit 7 of an Arithmetic operation. It's also set if there is a carry from bit 7 (but not from bit 6) of Acc.
P	Parity - This bit will be set if ACC has odd number of 1's after an operation. If not, bit will remain cleared.

Register Bank Selection:

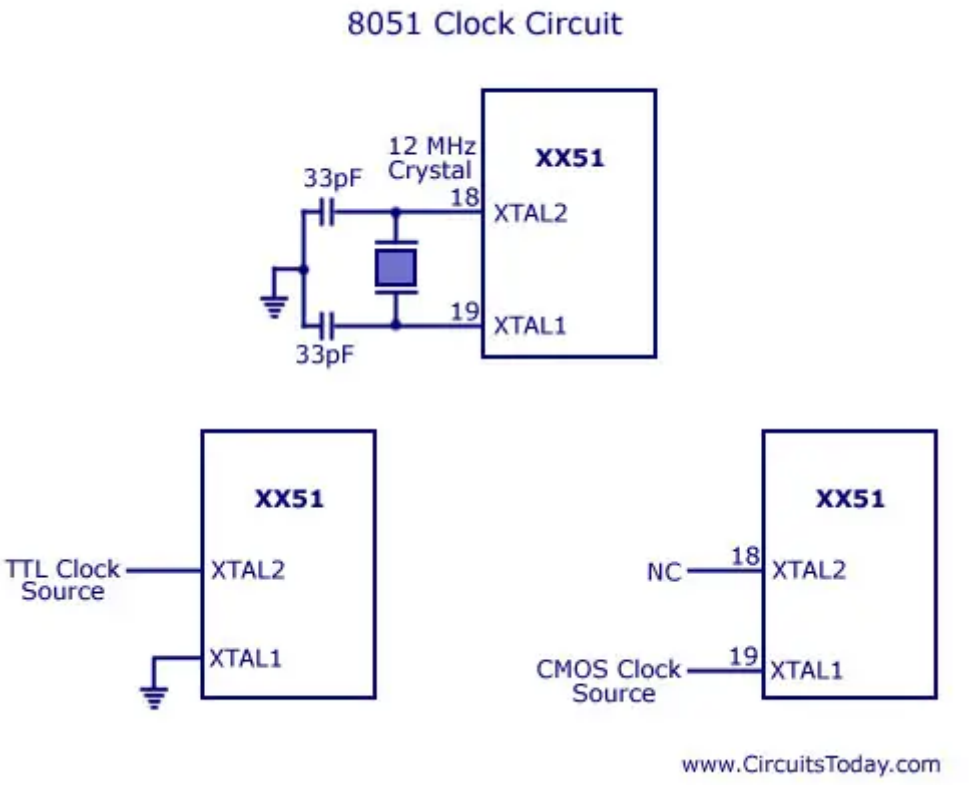
RS1 (PSW.4)	RS0(PSW.3)	Register Bank Selected
0	0	RB0
0	1	RB1
1	0	RB2
1	1	RB3

Explain clock and reset circuit for 8051 with sketch. (3)

Clock Circuit

- **Crystal Oscillator:** The foundation of the timing for the 8051 is a crystal oscillator connected to the XTAL1 and XTAL2 pins of the microcontroller.
    - The crystal, along with small capacitors (usually in the 20-30pF range), provides a stable and precise clock frequency.
    - Common crystal frequencies for 8051 systems are 11.0592 MHz or 12 MHz.
- **Internal Clock Generation:** The 8051 has an internal clock generator that takes the external crystal oscillator's signal and divides it down. This ensures that the microcontroller and its various components operate at the correct internal clock speed.

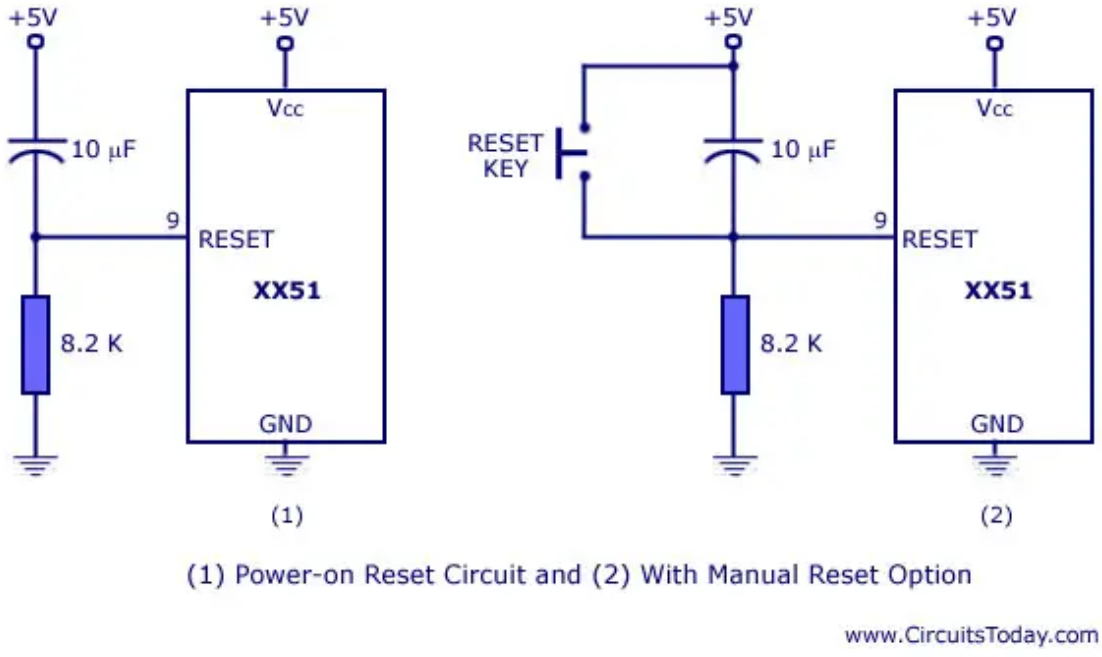
Diagram



Reset Circuit

- **RC Network:** A simple resistor-capacitor (RC) network is often used for the reset circuit.
  - When power is first applied, the capacitor begins to charge. This holds the RESET pin low for a short period, guaranteeing the 8051 starts in a known state.
  - Once the capacitor voltage reaches a threshold, the RESET pin goes high, allowing the microcontroller to begin executing code.
- **Supervisory Circuit (Optional):** For more robust reset control, a dedicated supervisory circuit/IC provides more precise monitoring of the power supply voltage. This ensures reliable resets if the power supply fluctuates or becomes unstable.

Diagram



Explanation

1. **Power On:** When the system powers on, the capacitor of the reset circuit is initially discharged, holding the RESET pin low.
2. **Reset:** This low level on the RESET pin forces the 8051 microcontroller into a reset state. Internal registers are cleared, and the Program Counter begins at address 0000H.
3. **Capacitor Charging:** The capacitor in the reset circuit starts charging through the resistor.
4. **Reset Released:** Once the capacitor charges beyond the RESET pin's threshold voltage, the pin goes high. The 8051 starts executing code from the beginning of its program memory.
5. **Clock Stabilization:** While the reset circuit is active, the crystal oscillator begins to oscillate and the clock stabilizes. The 8051's internal clock generator uses this signal to provide the necessary timing for the microcontroller's operation.

Key Points

- The clock and reset circuits are essential for the correct initialization and operation of an 8051 microcontroller system.
- Simple and inexpensive reset circuits can be designed using just a capacitor and resistor.
- Supervisory circuits offer improved power monitoring and enhanced reset reliability.

I/O Ports

General I/O Port Features

- **Bidirectional:** All four I/O ports (Port 0, Port 1, Port 2, and Port 3) are bidirectional. Each pin can be configured as either an input or an output.
- **Latches:** Each port has an associated latch that holds the output data. When a value is written to a port, it is stored in this latch, driving the output pins.
- **Internal Pull-ups (Except Port 0):** Ports 1, 2, and 3 have built-in pull-up resistors. When configured as inputs, these resistors weakly pull the pins high. If you need a strong pull-down for a '0' input, you'll need to add external resistors.
- **Dual Functionality:** Some port pins serve additional purposes, as explained below.

Port 0 (P0)

- **Address/Data Bus Duties:** Port 0 shares its pins to serve as:
  - **The lower 8-bits of the address bus (AD0-AD7)** when connecting to external memory.
  - **An 8-bit data bus (D0-D7)** for external memory read/write operations.
- **Open Drain:** Port 0's output drivers have an open-drain configuration. This means they can actively drive a pin low (logic '0'), but for high outputs (logic '1'), an external pull-up resistor is required.
- **Needs Pull-Ups:** When used as general-purpose I/O, Port 0 needs external pull-up resistors.

Port 1 (P1)

- **Standard I/O:** Primarily used as a general-purpose I/O port.
- **No Additional Functions:** Pins of Port 1 don't have other roles like addressing.

Port 2 (P2)

- **Address Bus Duties:** When external memory is used, Port 2 provides the upper 8-bits of the 16-bit address (A8-A15).
- **Limited I/O Availability:** In systems with external memory, Port 2 loses its ability to be used for general-purpose input/output.

Port 3 (P3)

- **Diverse Roles:** Pins on Port 3 have multiple alternate functions, making it quite versatile:
  - Serial Communication (RXD, TXD)
  - Timer/Counter External Inputs
  - Control Signals for External Memory (RD, WR)
  - Interrupts

Important Notes:

- **Initial State:** Upon reset, all I/O ports are configured as inputs.
- **Configuring as Outputs:** To use a port pin as an output, you need to write a '1' to the corresponding bit in the port's SFR.
- **Configuring as Inputs:** To use a port pin as an input, you must write a '1' to the corresponding bit in the port's SFR, ensuring the internal pull-ups are working as intended.

Example (C code):

```
#include <reg51.h> // Header file for 8051 SFRs

// Configure P1.0 as output, the rest of Port 1 as input
P1 = 0x01;

// Write a logic 1 (high) to P1.0
P1_0 = 1;

// Read the value from P1.5
unsigned char input_value = P1_5;
```

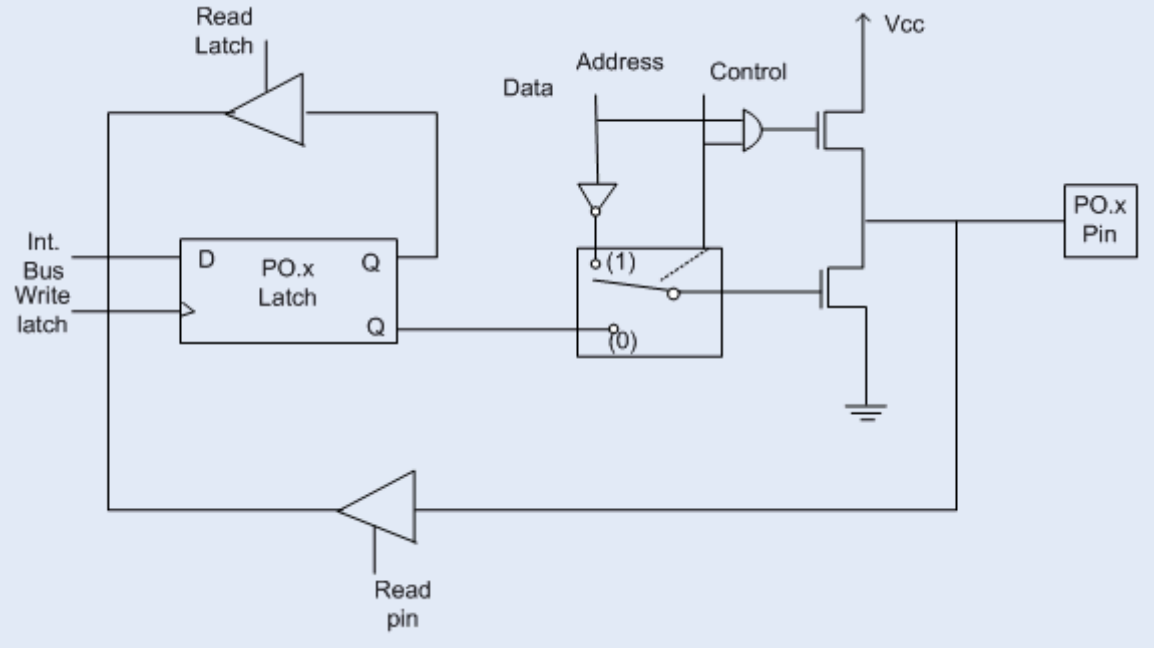
I/O Ports structure: Port 0, Port 1, Port2, Port 3.

Each port of 8051 has bidirectional capability. Port 0 is called 'true bidirectional port' as it floats (tristated) when configured as input. Port-1, 2, 3 are called 'quasi bidirectional port'.

Port-0 Pin Structure:

- **Dual Purpose:**
  - **General Purpose I/O:** Can be configured as a standard 8-bit bidirectional input/output port.
  - **Address/Data Bus:** Serves as the lower 8-bits of the address bus (AD0-AD7) and the data bus (D0-D7) when interfacing with external memory.
- **Open-Drain Outputs:** Port 0 pins use an open-drain configuration for outputs. This means they can actively drive a pin low (logic '0'), but require an external pull-up resistor to achieve a high output (logic '1').

- **Latch:** Each Port 0 pin is connected to a latch. Data written to the P0 SFR (Special Function Register) is held in this latch.
- **Internal Diagram (Simplified):**



Operation

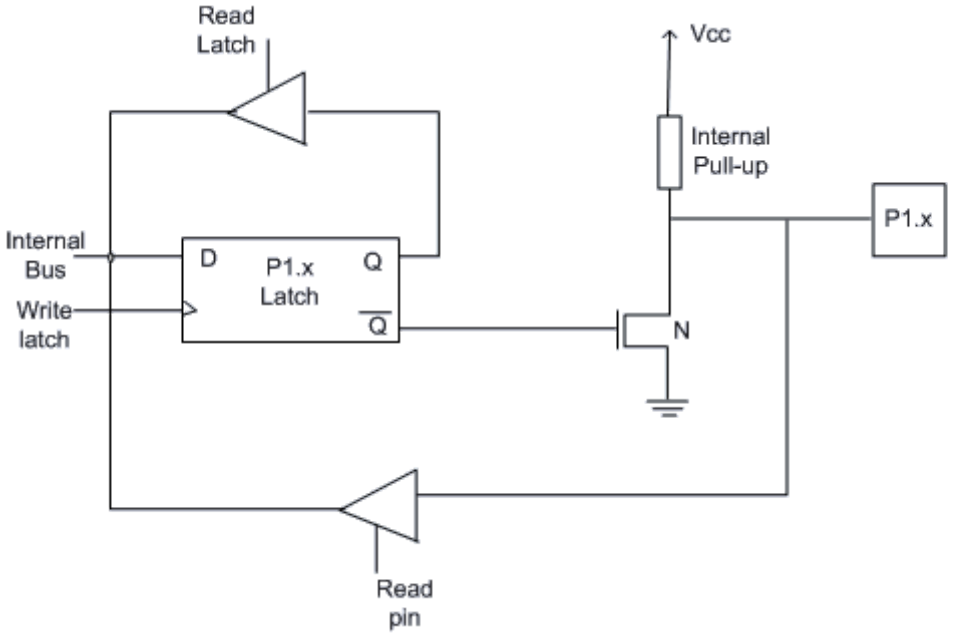
- **Input Mode:**
  1. To configure as input, write a '1' to the corresponding latch bit.
  2. Both output MOSFETs are turned off, resulting in a high-impedance state.
  3. External devices or pull-up resistors determine the pin's voltage level.
- **Output Mode:**
  1. **Writing '0':** The lower MOSFET turns on, pulling the pin to ground (logic '0').
  2. **Writing '1':**
     - Both MOSFETs turn off, resulting in a high-impedance state.
     - An external pull-up resistor is **required** to achieve a high output (logic '1').
- **External Memory Interfacing:**
  1. A control signal (likely ALE) determines if Port 0 functions in address/data mode.
  2. When acting as the address/data bus:
     - **'0' Output:** Lower MOSFET on, upper MOSFET off.
     - **'1' Output:** Lower MOSFET off, upper MOSFET on (the bus itself will pull the line high).

Key Points

- **Pull-up Resistors:** Port 0 absolutely requires external pull-up resistors when used as general-purpose I/O in situations where you need to output a logic '1'.
- **Versatility with Tradeoffs:** The dual-functionality of Port 0 offers flexibility, but adds a layer of complexity when interfacing external memory.

Port 1 Pin Structure

- **Dedicated I/O:** Port 1 is a simple 8-bit bidirectional I/O port. Its pins do not have any additional alternate functionality like serving as address lines or special control signals.
- **Internal Pull-up Resistors:** A crucial feature of Port 1 is that each pin is connected to a weak internal pull-up resistor. These resistors are automatically enabled when the port pin is configured as an input.
- **Internal Diagram (Simplified):**



Operation

- **Input Mode:**
  1. To configure as input, write a '1' to the corresponding latch bit.
  2. The internal pull-up resistor weakly pulls the pin towards a high voltage level (logic '1').
  3. To read a logic '0', an external device must be strong enough to overcome the internal pull-up and pull the pin to ground.
- **Output Mode:**
  1. To configure as output, write a '0' or '1' to the corresponding latch bit.
  2. The internal pull-up resistor is effectively overridden.
  3. **Writing '0':** The output driver actively pulls the pin low.
  4. **Writing '1':** The output becomes high-impedance, but the internal pull-up resistor weakly pulls the pin towards a high state.

Important Considerations

- **Weak Pull-ups:** The internal pull-up resistors on Port 1 are relatively weak. If a connected external device attempts to strongly drive a pin low, it might not be able to fully bring the voltage to a valid logic '0' level.
- **Sinking Current:** When a Port 1 pin is configured as an input and an external device drives it low, the external circuitry needs to be able to sink the current flowing through the internal pull-up resistor.

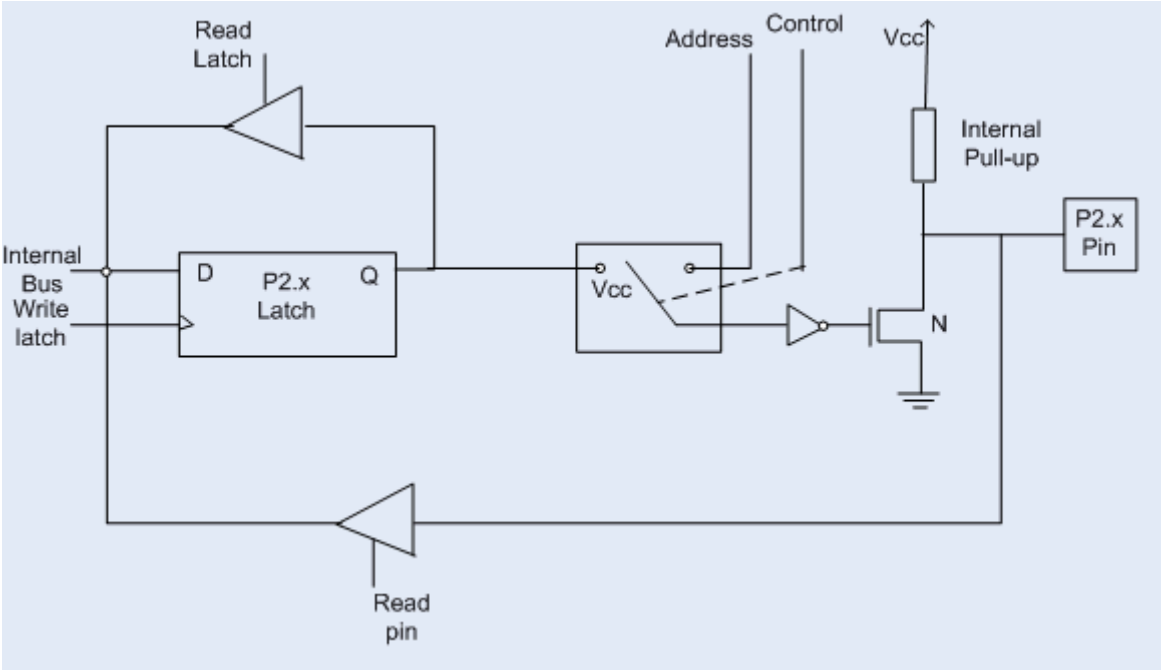
- **Potential for Incorrect Readings:** If an external device is not strong enough or is configured incorrectly, the input may not register a true '0' even when the external device intends to drive it low.

Recommendations

- **Input Considerations:** If using Port 1 for inputs where strong logic '0' signals are needed, consider either:
    - Disabling the internal pull-ups (if software/hardware allows it) and using external pull-down resistors.
    - Using a different port without built-in pull-ups.
- **Output Considerations:** Port 1 can drive outputs effectively, but keep in mind that writing a '1' relies on the internal pull-up or an external pull-up to achieve the high state.

Port 2 Pin Structure

- **Dual Roles:**
    1. **Higher Order Address Bus:** When the 8051 is interfaced with external memory, Port 2 provides the upper 8-bits of the 16-bit address (A8-A15).
    2. **General Purpose I/O:** If external memory is not in use, Port 2 can function as a standard 8-bit bidirectional I/O port.
- **Internal Pull-up Resistors:** Similar to Port 1, each pin of Port 2 has an internal pull-up resistor that is active when the pin is configured as an input.
- **Internal Diagram (Simplified):**



Operation

- **Input Mode:**
    1. To configure as input, write a '1' to the corresponding latch bit.
    2. The internal pull-up weakly pulls the pin high (logic '1').
    3. External devices must be strong enough to overcome the pull-up resistor to drive a logic '0'.

- **Output Mode:**
    1. To configure as output, write a '0' or '1' to the corresponding latch bit.
    2. The output driver actively drives the pin high or low, overriding the internal pull-up.
- **External Memory Interfacing:**
    1. When used as the higher address byte, the Port 2 latch holds the address information.
    2. Latch values remain stable during external memory operations.

Important Considerations

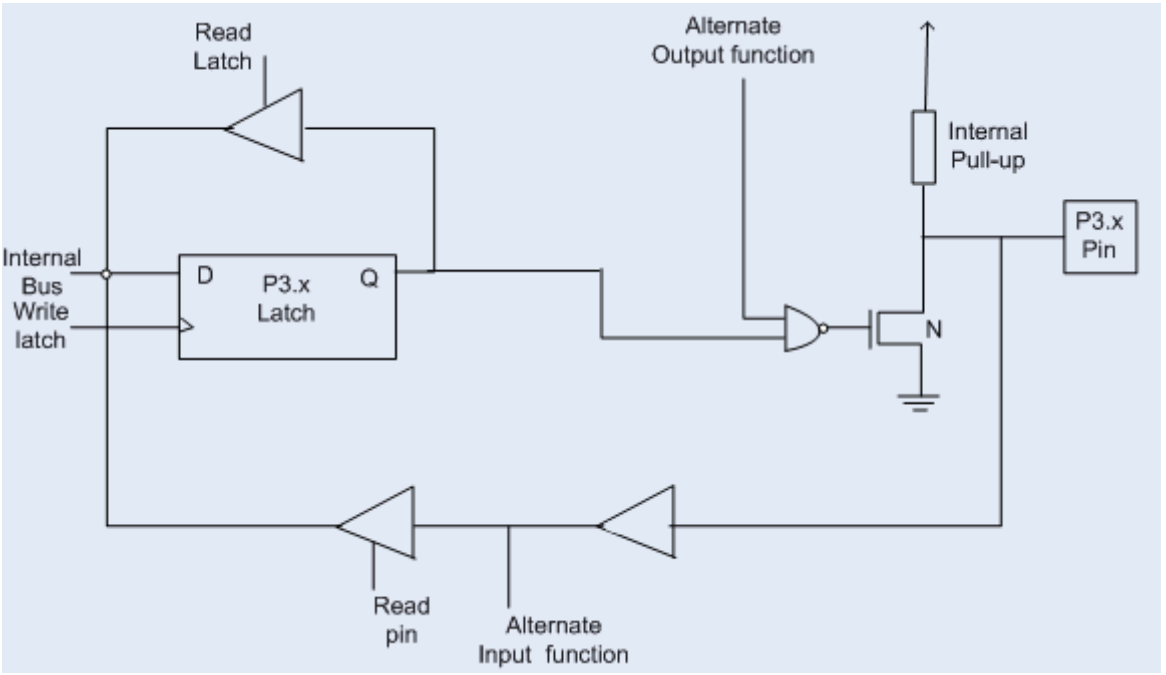
- **Limited Current Capacity:** As with Port 1, the internal pull-ups on Port 2 mean it's not ideal for driving heavy loads or sinking significant current, especially in input mode.
- **Conflict with External Memory:** When using external memory, Port 2's general-purpose I/O functionality is effectively unavailable.
- **Design Trade-offs:** The dual-role capability of Port 2 adds flexibility, but requires careful consideration of its function in relation to other system requirements.

Recommendations

- **Input Considerations:** The same recommendations for Port 1 apply to Port 2. Consider external pull-down resistors or disabling the internal pull-ups if reliable '0' inputs are crucial and your external devices are weak drivers.
- **External Memory Considerations:** If using external memory, avoid relying on Port 2 for general-purpose inputs.

Port 3 Pin Structure

- **Multifunctional:** Port 3, unlike Ports 1 and 2, is the most versatile port on the 8051. Each of its 8 pins (P3.0-P3.7) can serve either as a general-purpose I/O pin or take on a specialized alternate function.
- **Internal Pull-ups:** Each pin on Port 3 has a weak internal pull-up resistor, similar to Ports 1 and 2. This pull-up is active when the pin is configured as an input.
- **Alternate Function Control:**
    - **Latch:** Each Port 3 bit has a corresponding latch bit. Writing a '1' to the latch allows the alternate function to be used.
    - **Priority:** If multiple alternate functions compete for the same pin, a priority system exists to determine which function takes precedence.
- **Internal Diagram (Simplified):**



Alternate Functions of Port 3

PORT 3 Pin	Function	Description
P3.0	RXD	Serial Data Receive for UART communication.
P3.1	TXD	Serial Data Transmit for UART communication.
P3.2	INT0	External Interrupt 0 input.
P3.3	INT1	External Interrupt 1 input.
P3.4	T0	Timer/Counter 0 external input.
P3.5	T1	Timer/Counter 1 external input.
P3.6	WR	Write strobe for external memory.
P3.7	RD	Read strobe for external memory.

Operation

- **Input Mode:**
    1. Write a '1' to the pin's latch bit.
    2. The internal pull-up pulls the pin high.
    3. External devices must overcome the pull-up to drive a strong logic '0'.
- **Output Mode:**
    1. Write a '0' or '1' to the pin's latch bit.
    2. Output drivers actively drive the pin high or low.
- **Alternate Function Mode:**
    1. Write a '1' to the corresponding latch bit to enable the alternate function.
    2. The pin is now dedicated to its special role (serial communication, interrupt, etc.).

Important Notes

- **Flexibility and Tradeoffs:** Port 3's versatility comes at the cost of reduced I/O capability if many alternate functions are in use.
- **Configuration:** Careful software configuration is essential to determine whether a Port 3 pin acts as general-purpose I/O or in its alternate function role.

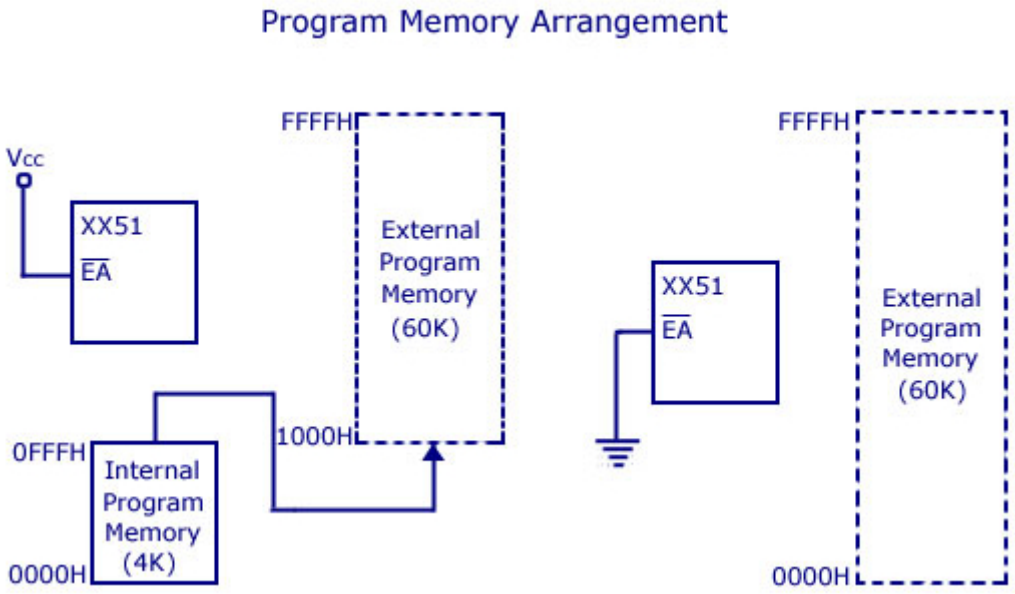
Memory Organization

Draw and Explain program and data memory of 8051. (4)

Program Memory (ROM)

- **Purpose:** The program memory is where the 8051 stores the instructions that make up the program it's executing. Think of it as the microcontroller's 'recipe book' of code.
- **Types:**
  - **Internal ROM:** Most 8051 derivatives have some amount of built-in program memory (often around 4KB).
  - **External ROM:** If a program is too large to fit in the internal ROM, the 8051 can interface with external memory chips to expand its program storage.
- **Non-volatile:** This means that the program code remains stored even when the 8051 loses power.
- **Access Control:** The external memory is accessed through the External Access (EA) pin. By default, the EA pin is connected to VCC, so the microcontroller fetches instructions from internal memory first. If the program size exceeds 4KB, the microcontroller will automatically switch to external memory. To force the microcontroller to use external memory only, connect the EA pin to GND.

Diagram (Program Memory)



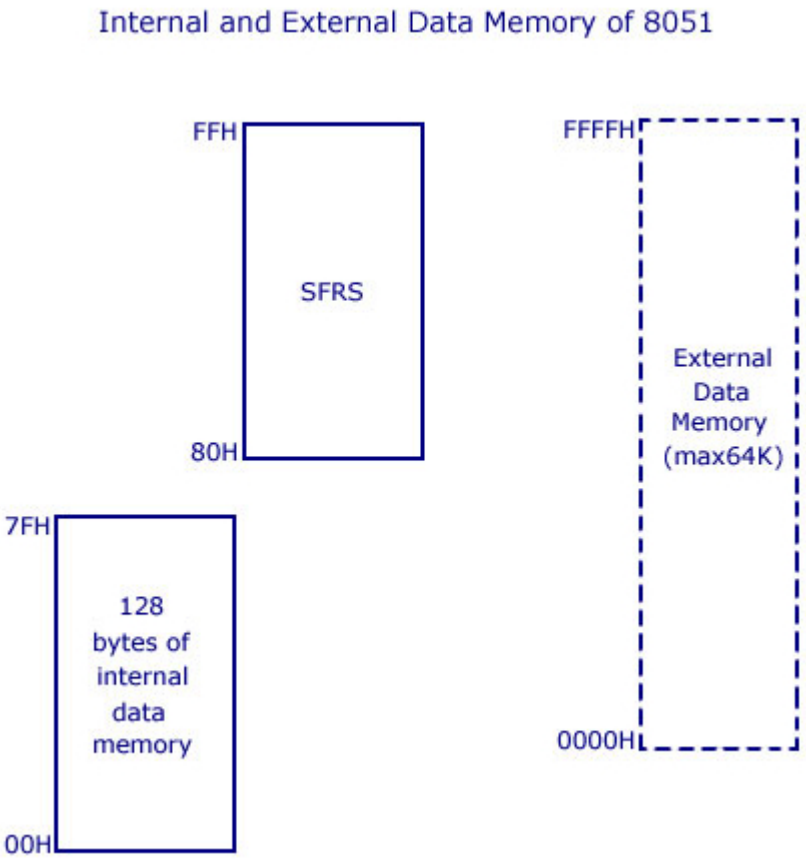
A simplified visual representation of program memory might look like this:

```
+--------------------+
| Program Memory (ROM)|
+--------------------+
|  Instruction 1      |
|  Instruction 2      |
|        ...          |
|  Instruction N      |
+--------------------+
```

Data Memory (RAM)

- **Purpose:** The data memory acts as the 8051's workspace. It holds temporary variables, intermediate calculations, and other data the program needs while running.
- **Types**
  - **Internal RAM:** The 8051 has a limited amount of internal RAM (usually 128 bytes).
  - **External RAM:** Like with program memory, the 8051 can utilize external RAM for additional data storage.
- **Volatile:** Data in RAM is lost when the 8051 loses power.

Structure of Internal Data Memory



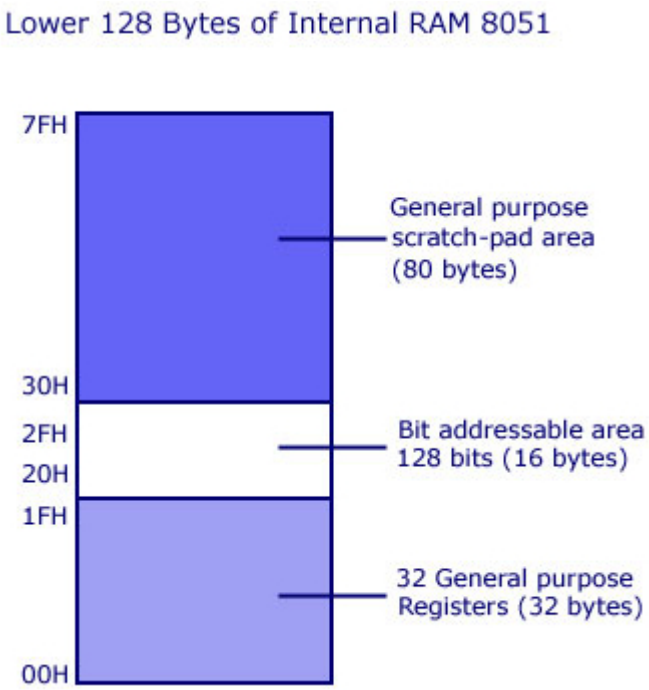
The internal RAM is divided into several important areas:

1. **Register Banks (00H - 1FH):**
   - Four banks of 8 general-purpose registers (R0-R7).
   - Only one bank is active at a time.
   - Used for frequently accessed data and arithmetic operations.
2. **Bit-Addressable Area (20H - 2FH):**

- 128 single bits that can be addressed individually.
- Efficient for storing flags, status bits, or control signals.

3. **Scratch Pad Area (30H-7FH):**
   - General-purpose area for variables and temporary data.
   - Also includes the stack (used for storing return addresses during function calls and interrupts).

Diagram (Data Memory)




```
+--------------------+
|  Data Memory (RAM) |
+--------------------+
| Register Banks 0-3 | (00H - 1FH)
+--------------------+
| Bit-Addressable    | (20H - 2FH)
+--------------------+
| Scratch Pad        | (30H - 7FH)
+--------------------+
```

Key Points

- **Memory Access:** The 8051 uses specific instructions and addressing modes to interact with both its program and data memory.
- **Limited Internal Resources:** The internal RAM and ROM of the 8051 are relatively small, highlighting the potential need for external memory in more complex applications.
- **Memory Trade-offs:** Program and data memory share the same external memory address space, often necessitating careful planning of how memory is used by a program.

Draw and Explain External Memory Addressing and Decoding Logic of 8051.

External Memory Interfacing in the 8051

The 8051 microcontroller offers limited internal program and data memory, which might not be sufficient for complex applications. To expand its memory capacity, the 8051 can be interfaced with external memory devices like ROM and RAM. This capability allows you to store larger programs and work with more data.

Key Components Involved:

- **Microcontroller:** The 8051 itself, responsible for controlling data flow and program execution.
- **External Memory:** ROM chips for program storage and RAM chips for data storage. Both can be up to 64KB in size.
- **Address Decoding Logic:** Circuitry that translates the microcontroller's memory addresses into specific chip select signals for each external memory device.
- **Control Signals:** Signals like PSEN (Program Store Enable), RD (Read), and WR (Write) from the microcontroller to control external memory operations.
- **Data Bus:** A bidirectional bus that carries data between the microcontroller and external memory.

Addressing and Decoding Process:

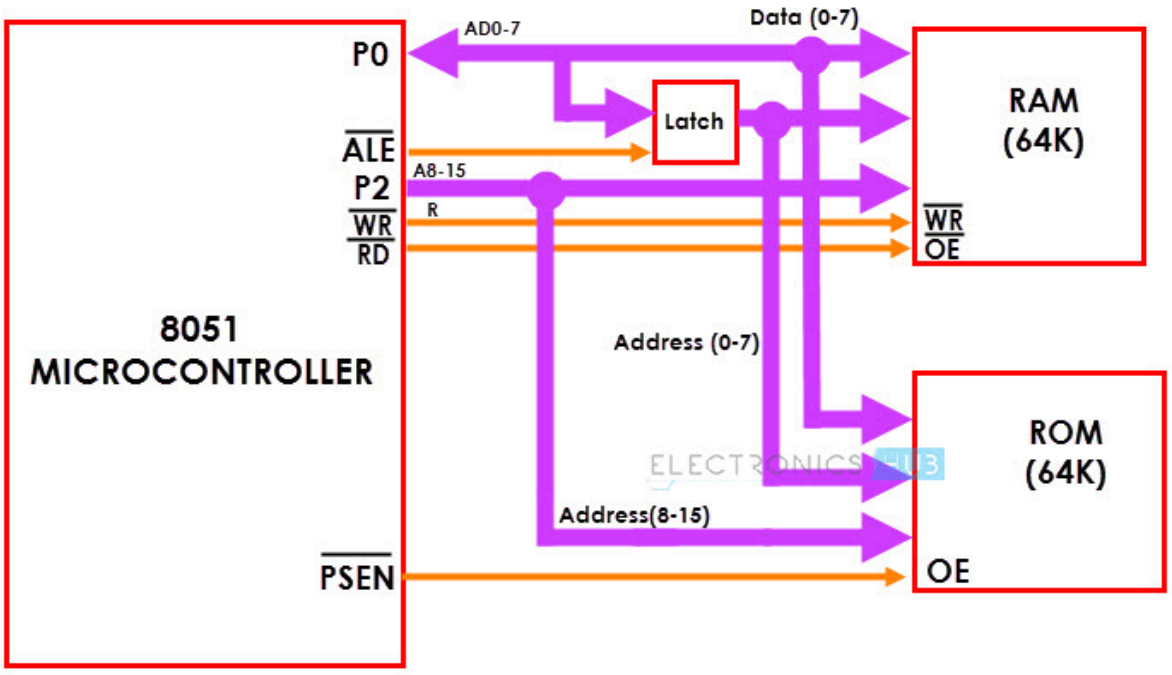
1. **Memory Access Initiation:** The microcontroller initiates a memory access operation, specifying an address and indicating whether it's a read or write operation.
2. **Address Bus Decoding:** The address decoding logic receives the address from the microcontroller.
3. **Chip Select Generation:** Based on the decoded address and the memory map, the decoding logic generates individual chip select signals for the appropriate ROM or RAM chip(s).
4. **Control Signal Assertion:** The microcontroller asserts control signals like PSEN, RD, or WR along with the data (for write operations) onto the control and data buses.
5. **Data Transfer:** The selected memory chip(s) perform the read or write operation based on the control signals and data provided.
6. **Data Bus Interaction:** The data is transferred between the microcontroller and the selected memory chip(s) on the data bus.

Example:

Imagine the microcontroller wants to read data from byte address 40000 (64KB ROM, 0-31KB for ROM, 32KB-63KB for RAM) in external memory.

1. The address 40000 is sent to the address decoding logic.
2. The logic recognizes it's within the ROM address range (0-31KB) and generates a chip select for the ROM chip.
3. The microcontroller asserts the RD (read) signal and places the address 40000 on the address bus.
4. The selected ROM chip reads the data at byte address 40000 and places it on the data bus.
5. The microcontroller reads the data from the data bus and stores it internally.

The below image shows a simplified block diagram of interfacing 64KB ROM and 64KB RAM with the 8051:



- **Microcontroller:** Represented by the 8051 block.
- **External Memory:** ROM and RAM blocks labeled as "64K ROM" and "64K RAM".
- **Address Decoding Logic:** Not explicitly shown but implied by the connections between the address bus and chip select signals.
- **Control Signals:** PSEN, RD, and WR signals are shown from the microcontroller.
- **Data Bus:** Represented by the bidirectional "Data (0-7)" lines.

We know that a typical 8051 Microcontroller has 4KB of ROM and 128B of RAM

The designer of an 8051 Microcontroller based system is not limited to the internal RAM and ROM present in the 8051 Microcontroller. There is a provision of connecting both external RAM and ROM i.e. Data Memory and Program.

The reason for interfacing external Program Memory or ROM is that complex programs written in high – level languages often tend to be larger and occupy more memory.

Another important reason is that chips like 8031 or 8032, which doesn't have any internal ROM, have to be interfaced with external ROM.

A maximum of 64KB of Program Memory (ROM) and Data Memory (RAM) each can be interface with the 8051 Microcontroller.

The following image shows the block diagram of interfacing 64KB of External RAM and 64KB of External ROM with the 8051 Microcontroller.

Draw and explain internal RAM architecture of the 8051 microcontroller. (4)

Internal RAM Organization

The 8051 family of microcontrollers typically includes 128 bytes of internal RAM, although some derivatives like the 8052 offer an extended 256 bytes. This internal RAM is organized into several distinct sections:

1. **Register Banks (00H - 1FH):**
    - Four banks of eight general-purpose registers (R0-R7).
    - Each bank can be selected using two bits in the Program Status Word (PSW) register.
    - Used for storing temporary data and intermediate results during calculations.
2. **Bit-Addressable Area (20H to 2FH):**
    - 16 bytes of RAM where each bit can be individually addressed (128 individual bits in total).
    - Useful for storing single-bit variables (like flags or control signals).
3. **General Purpose RAM (30H - 7FH):**
    - The remaining 80 bytes of general-purpose RAM.
    - Used for variable storage, temporary data, and even as a small stack if needed.

Key Points

- **Speed:** Internal RAM is extremely fast to access compared to external RAM, as it's located directly on the microcontroller chip.
- **Limited Size:** The internal RAM in 8051 is limited. Programs with larger data requirements often need external RAM.
- **Flexibility:** The bit-addressable area provides fine-grained control over individual bits, ideal for control and status flags.

How Internal RAM Is Used

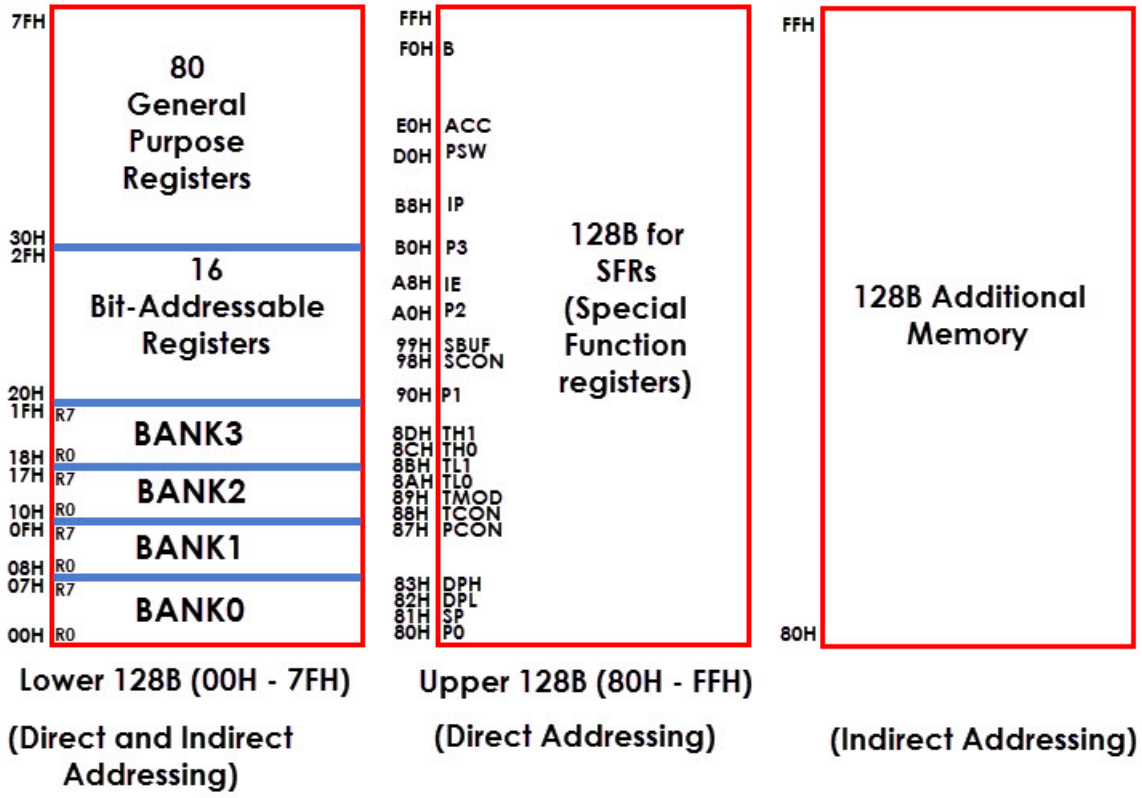
- **Arithmetic and Logical Operations**: The register banks are heavily used by the ALU for arithmetic and logical operations.
- **Temporary Storage:** All sections of the internal RAM can be used for temporarily storing data during calculation or program execution.
- **Stack:** Although the 8051 has a dedicated hardware stack, the general-purpose RAM can also be used as a stack area in constrained situations.
- **Flags and Control:** The bit-addressable area often houses individual control flags and status bits for the 8051 or its peripherals.

Example

```
MOV R1, #50H  ; Move the value 50H into register R1
ADD A, R1     ; Add the value in R1 to the accumulator
MOV 35H, A    ; Store the result in general-purpose RAM location 35H
SETB PSW.2    ; Set bit 2 (Carry flag) in the Program Status Word
```

Absolutely! Here's a heavily refined and improved version of the information on the 8051's data memory structure, incorporating insights to make it clearer and more accurate:

Data Memory (RAM) in the 8051 Microcontroller



The 8051's data memory (RAM) serves as a workspace for storing temporary data, variables, and intermediate results during program execution. Most modern 8051 variants provide 256 bytes of internal RAM, which is organized into the following distinct areas:

1. Working Registers (00H - 1FH)

- **Register Banks:** The first 32 bytes of RAM are divided into four register banks (Bank 0, Bank 1, Bank 2, Bank 3). Each bank contains eight general-purpose registers (R0-R7).
- **Addressing:**
    - **By Name:** Access registers by name (R0, R1, etc.) after selecting the appropriate bank using the RS0 and RS1 bits in the Program Status Word (PSW) register.
    - **By Address** Access registers directly by their address (e.g., 12H for R2 in Bank 2), regardless of the currently selected bank.

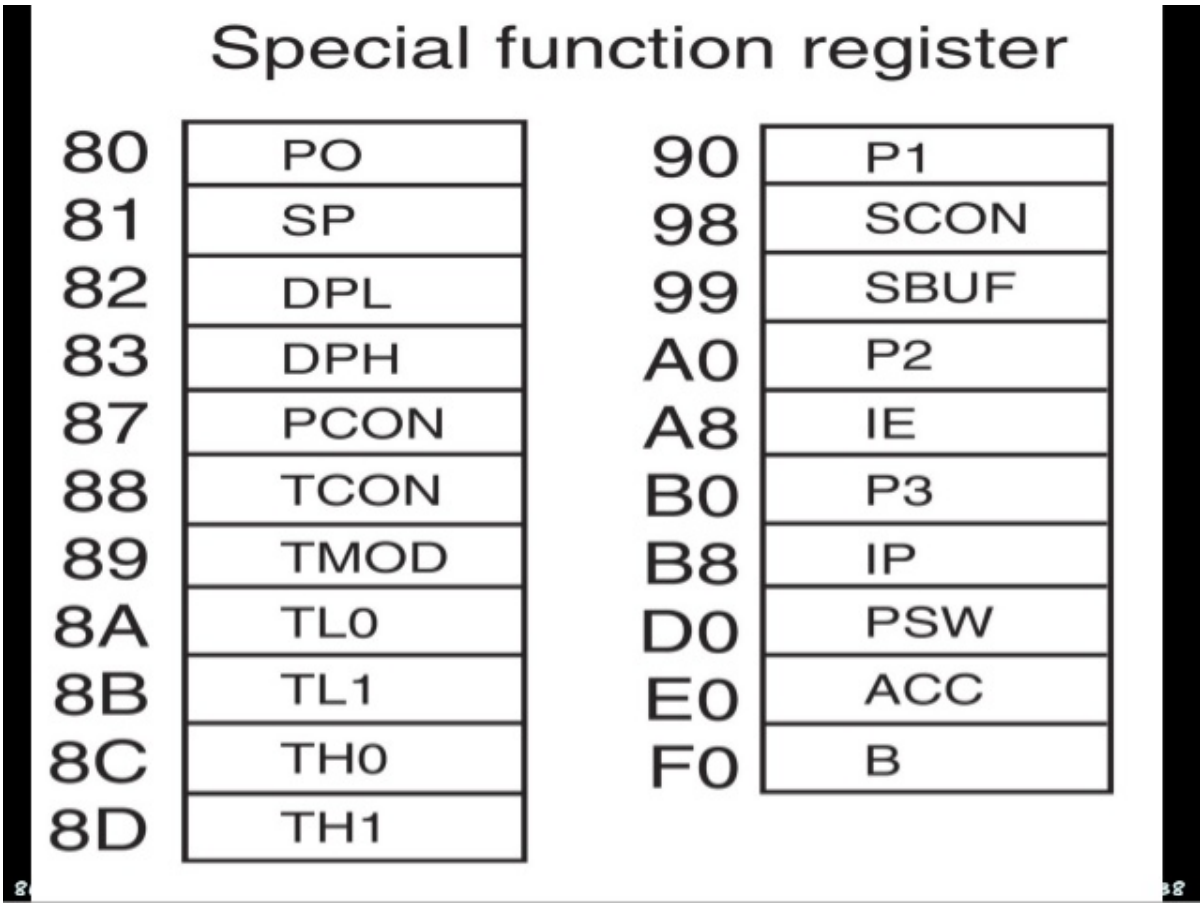
2. Bit-Addressable Memory (20H - 2FH)

- **Individual Bit Control:** This area contains 128 individually addressable bits (00H - 7FH within the byte range 20H-2FH). This is efficient for storing single-bit values like flags or control signals.

3. General Purpose RAM (Scratchpad) (30H - 7FH)

- **Flexible Storage:** This 80-byte area provides general-purpose data storage for variables and temporary data.
- **Stack:** The stack, used for storing function call return addresses and temporary storage during interrupts, also resides within this area.

4. Special Function Registers (SFRs) (80H - FFH)



- **Hardware Control:** SFRs occupy the upper 128 bytes of RAM and directly control various hardware functions of the 8051, such as:
    - I/O Ports (P0, P1, P2, P3)
    - Program Status Word (PSW)
    - Accumulator (A)
    - Interrupt Control (IE, IP)
    - Power Management (PCON)
- **Direct Addressing Only:** SFRs can only be accessed using their specific addresses. Unused addresses within this range are reserved and cannot be used for general-purpose data storage.

Additional Notes

- **Indirect Addressing:** The lower 128 bytes of RAM (working registers, bit-addressable area, and scratchpad) can be addressed both directly (by their address) and indirectly (using a register to hold the address).
- **Limited RAM Capacity:** The 8051's internal RAM is relatively small. Many applications require interfacing external RAM to support larger datasets.
- **Variant Differences:** Some 8051 variants may have an additional 128 bytes of RAM sharing the same address space as SFRs. This extra RAM is usually only accessible via indirect addressing.

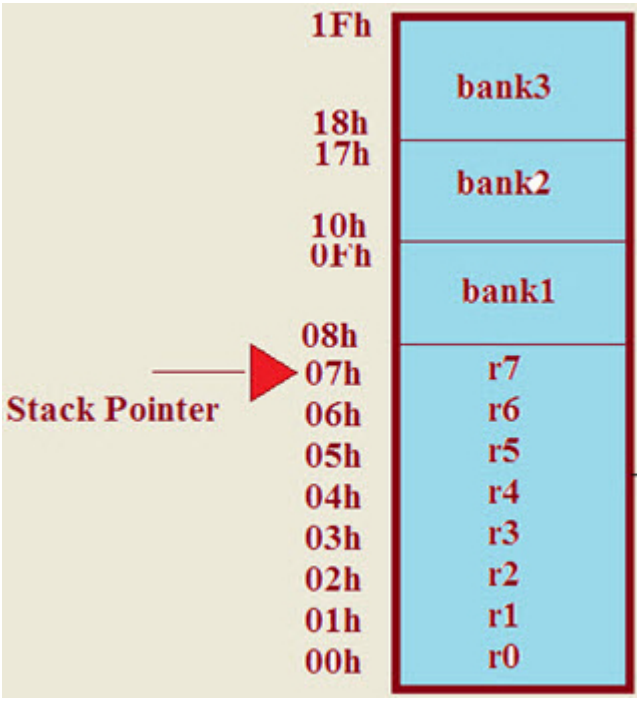
Stack, Stack Pointer, and Stack Operations

What is the Stack?

- **LIFO Structure:** The stack is a section of the 8051's internal RAM that follows a Last-In, First-Out (LIFO) principle. Imagine it like a stack of plates; you always add and remove from the top.
- **Purpose:**
    - **Temporary Storage:** The stack stores data temporarily during program execution.
    - **Function Calls:** It saves the return address when a function (subroutine) is called, allowing the program to return to the correct point after the function completes.
    - **Interrupts:** When an interrupt occurs, the 8051 temporarily pushes the current program counter (PC) onto the stack, allowing it to later resume execution where it was interrupted.

Stack Pointer (SP)

- **Address Tracker:** The Stack Pointer (SP) is an 8-bit register that holds the address of the top of the stack (the last item added).
- **Initialization:** Upon reset, the SP is usually initialized to 07H within the 8051's internal RAM.
- **Dynamic:** The SP changes automatically during stack operations (push and pop).

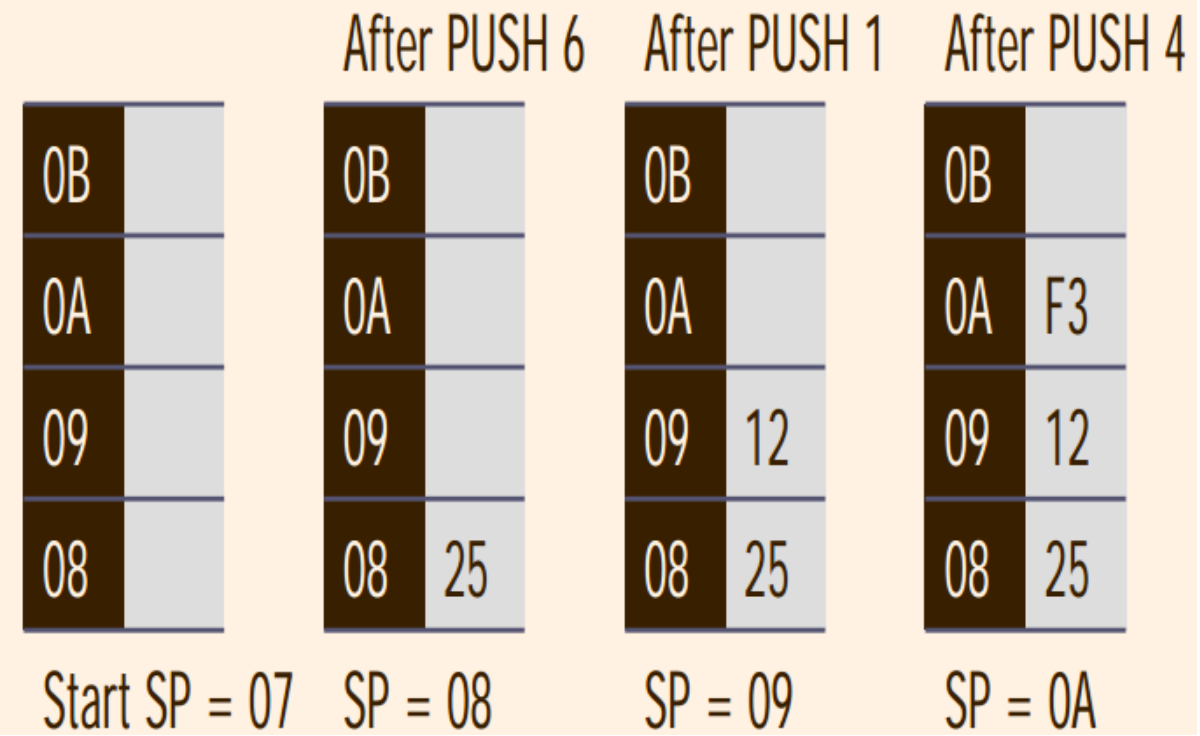


Stack Operations

1. **PUSH Instruction:**
    - **Stores Data:** The PUSH instruction puts a byte of data onto the top of the stack.
    - **SP Modification:**
        1. The SP is incremented.
        2. The data is then stored at the memory location now pointed to by the SP.

PUSH Example:

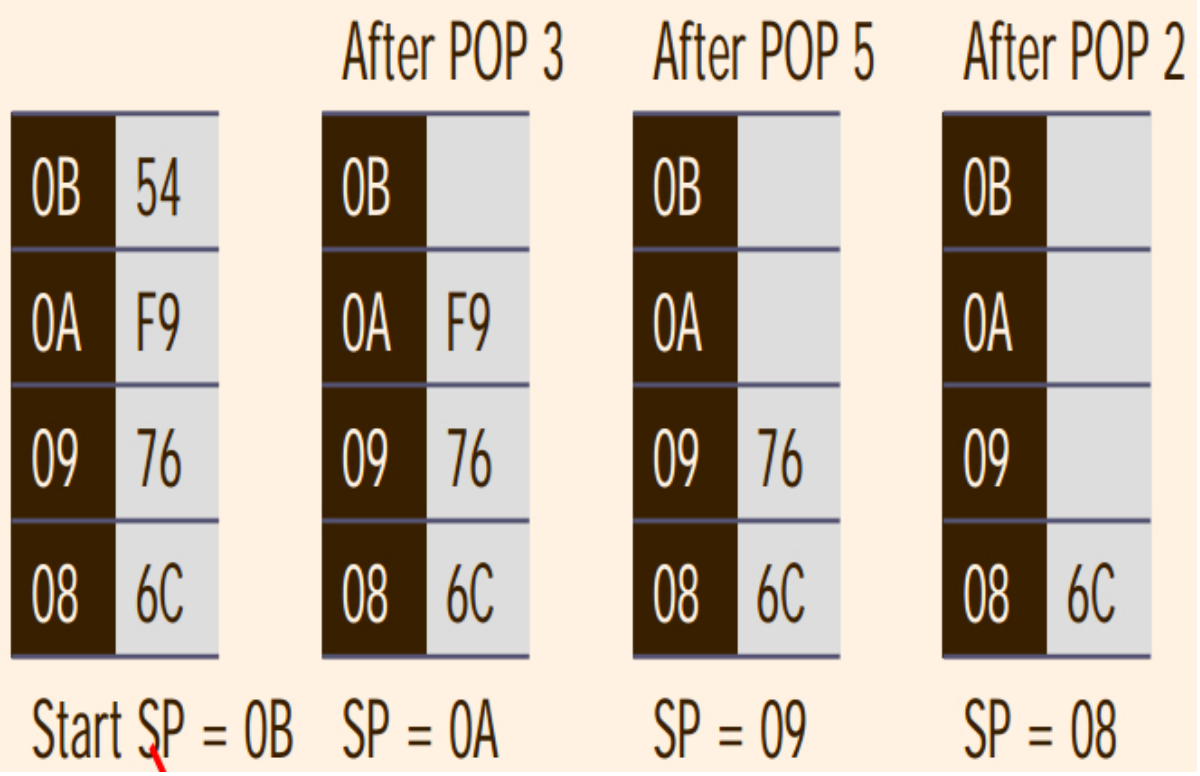
```
MOV R6, #25H
MOV R1, #12H
MOV R4, #0F3H
PUSH R6
PUSH R1
PUSH R4
```



2. **POP Instruction:**
    - **Retrieves Data:** The POP instruction removes a byte of data from the top of the stack.
    - **SP Modification:**
        1. The data at the location pointed to by the SP is retrieved.
        2. The SP is then decremented.

POP Example:

```
POP R3 ; POP stack into R3
POP R5 ; POP stack into R5
POP R2 ; POP stack into R2
```



Common Uses of the Stack

- **Temporary Storage:** Storing the contents of registers during calculations when there aren't enough registers available.
- **Subroutine Calls:** When a subroutine (function) is called using the 'CALL' instruction, the return address (next instruction after the call) is automatically pushed onto the stack. The 'RET' instruction pops this return address, so execution continues correctly.
- **Interrupt Handling:** When an interrupt occurs, the 8051 automatically pushes the Program Counter (PC) onto the stack, allowing seamless return to the interrupted code after the interrupt service routine.

Important Considerations

- **Limited Stack Size:** The 8051's internal RAM is small, which limits the size of the stack. Be mindful of stack usage to avoid overflow conditions.
- **Stack Overflow:** This occurs if you try to PUSH data when the stack is full. This can lead to unpredictable behavior.
- **Stack Underflow:** This occurs if you try to POP data when the stack is empty. This can also result in errors.
- **Stack Size:** The 8051's internal RAM for the stack is limited; it's crucial to prevent stack overflow.
- **Initialization:** The SP is initialized to 07H when the 8051 resets; your code often needs to set it to a custom location.

Example: Swapping Two Numbers

```
MOV SP, #70H   ; Initialize Stack Pointer (assuming safe RAM space)

MOV A, #25H    ; Load the first number into the accumulator
PUSH A         ; Push the first number onto the stack

MOV A, #30H    ; Load the second number into the accumulator
PUSH A         ; Push the second number onto the stack

POP B         ; Pop the top (second) number into register B
POP A          ; Pop the original (first) number into the accumulator
```

Timers/Counters

TCON Register

What is the TCON Register?

- The TCON (Timer Control) register is an 8-bit, bit-addressable register present in 8051 microcontrollers.
- It's primarily responsible for controlling the operation of the microcontroller's internal timers and counters.

TCON Register Structure

The 8 bits of the TCON register are assigned specific functions:

- **TF1 (Timer 1 Overflow Flag):** Set to '1' when Timer 1 overflows. Cleared by software.
- **TR1 (Timer 1 Run Control Bit):** Controls the Run/Stop status of Timer 1.
  - '1' = Timer 1 is running
  - '0' = Timer 1 is stopped
- **TF0 (Timer 0 Overflow Flag):** Set to '1' when Timer 0 overflows. Cleared by software.
- **TR0 (Timer 0 Run Control Bit):** Controls the Run/Stop status of Timer 0.
  - '1' = Timer 0 is running
  - '0' = Timer 0 is stopped
- **IE1 (External Interrupt 1 Edge Flag):** Set to '1' when an external interrupt 1 occurs on a falling edge transition. Cleared by software.
- **IT1 (External Interrupt 1 Type Control Bit):** Configures external interrupt 1 trigger type.
  - '1' = Falling edge triggered
  - '0' = Low-level triggered.
- **IE0 (External Interrupt 0 Edge Flag):** Set to '1' when an external interrupt 0 occurs on a falling edge transition. Cleared by software.
- **IT0 (External Interrupt 0 Type Control Bit):** Configures external interrupt 0 trigger type.
  - '1' = Falling edge triggered
  - '0' = Low-level triggered

Key Functions of TCON Register

1. **Timer/Counter Start/Stop:** The TR1 and TR0 bits enable you to start and stop Timer 1 and Timer 0, respectively.

2. **Overflow Monitoring:** The overflow flags TF1 and TF0 indicate when a timer has reached its maximum count and rolled over. These are often used to generate interrupts.
3. **External Interrupt Configuration:** The IE0, IT0, IE1, and IT1 bits control how external interrupts are triggered and detected by the microcontroller.

Example: Setting up Timer 0 as an interval timer

1. **Set the Mode:** To use Timer 0 in a specific mode, you'll configure the TMOD register (Timer Mode Register). Let's assume you want Timer 0 as a 16-bit interval timer.
2. **Load Initial Value:** Load the desired starting count into the TH0 (Timer 0 High Byte) and TL0 (Timer 0 Low Byte) registers.
3. **Start the Timer:** Set TR0 (Timer 0 Run Control Bit) in the TCON register to '1' to begin the timer.
4. **Interrupt Handling (Optional):** If you want an interrupt to be generated when the timer overflows, set the interrupt enable bits in relevant registers and create an interrupt service routine (ISR). The TF0 flag in TCON will be set when an overflow occurs.

TCON register is also one of the registers whose bits are directly in control of timer operation. Only 4 bits of this register are used for this purpose, while rest of them is used for interrupt control.

TCON Register:

Address: 088H (Bit addressable)							
TCON.7	TCON.6	TCON.5	TCON.4	TCON.3	TCON.2	TCON.1	TCON.0
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

Description of All the Bits of TCON:

Flag	Function
TF1	Timer 1 Overflow flag. Set when timer rolls from all 1's to 0. Cleared when processor vectors to execute interrupt service routine located at program address 001Bh.
TR1	Timer 1 run control bit. Set to 1 by program to enable timer to count; cleared to 0 by program to halt timer.
TF0	Timer 0 Overflow flag. Set when timer rolls from all 1's to 0. Cleared when processor vectors to execute interrupt service routine located at program address 000Bh.
TR0	Timer 0 run control bit. Set to 1 by program to enable timer to count; cleared to 0 by program to halt timer.
IE1	External interrupt 1 Edge flag. Set to 1 when a high-to-low edge signal is received on port 3.3 (INT1). Cleared when processor vectors to interrupt service routine at program address 0013h. Not related to timer operations.
IT1	External interrupt 1 signal type control bit. Set to 1 by program to enable external interrupt 1 to be triggered by a falling edge signal. Set to 0 by program to enable a low-level signal on external interrupt 1 to generate an interrupt.

Flag	Function
IE0	External interrupt 0 Edge flag. Set to 1 when a high-to-low edge signal is received on port 3.2 (INT0). Cleared when processor vectors to interrupt service routine at program address 0003h. Not related to timer operations.
IT0	External interrupt 0 signal type control bit. Set to 1 by program to enable external interrupt 1 to be triggered by a falling edge signal. Set to 0 by program to enable a low-level signal on external interrupt 0 to generate an interrupt.

TMOD Register

What is the TMOD Register?

- The TMOD (Timer Mode) register is an 8-bit, bit-addressable Special Function Register (SFR) within 8051 microcontrollers.
- Its primary role is to select and configure the operating modes of the two built-in timers: Timer 0 and Timer 1.

TMOD Register Structure

The TMOD register has a specific function assigned to each of its 8 bits:

- **Bits 7-4 (Timer 1 Configuration)**
  - **Gate:** Controls Timer 1 gating for external control (described later).
  - **C/T̄:** Selects Timer vs. Counter mode for Timer 1.
    - '1' = Counter mode (counts external pulses)
    - '0' = Timer mode (counts internal machine cycles)
  - **M1 M0:** Selects the operating mode of Timer 1 (Modes 0, 1, 2, or 3)
- **Bits 3-0 (Timer 0 Configuration)**: Same structure as Timer 1 configuration bits above, but control the settings for Timer 0.

Operating Modes

The TMOD register allows you to configure each timer into one of four operating modes:

- **Mode 0 (13-bit Timer):**
  - Timer register is effectively 13 bits (THx: 8 bits, TLx: 5 bits)
  - This mode is often used for simple timing or event counting where extremely long delays are not required.
- **Mode 1 (16-bit Timer):**
  - The standard 16-bit timer/counter mode, providing the full range of counting.
- **Mode 2 (8-bit Auto-Reload Timer):**
  - TLx is reloaded with the value in THx automatically after each overflow. Useful for generating fixed periodic events.
- **Mode 3 (Split Timer):**
  - Timer 1 is stopped. TL0 is used as an 8-bit Timer/Counter and can be controlled independently, while TH0 runs as a separate 8-bit timer (usually controlled by the system clock).

Gate Bit

The Gate bit for each timer provides additional control:

- **Gate = '0':** The timer runs continuously when the TRx bit (in TCON) is set to '1'.
- **Gate = '1':** The timer runs only when the TRx bit is '1' AND an external pin (INT0 or INT1) receives a high-to-low transition.

Example: Configuring Timer 0 as a 16-bit timer with gating

1. **Setting the Mode:** To configure Timer 0 as a 16-bit timer, set the corresponding bits in the TMOD register as follows:

```
TMOD = 0x01; // Assuming you want Timer 0 in Mode 1, Timer 1 is not important
here
```

2. **Enabling Gating:** If you want to control Timer 0 with an external signal on INT0 pin:

```
TMOD |= 0x08; // Set the Gate bit for Timer 0
```

This register contains bits controlling the operation of timer 0 & 1. To select the operating mode and the timer/counter operation of the timers we use TMOD register. Timer 0 and timer 1 are two timer registers in 8051. Both of these registers use the same register called TMOD to set various timer operation modes.

TMOD is an 8-bit register. The lower 4 bits are for Timer 0. The upper 4 bits are for Timer 1.

In each case, The lower 2 bits are used to set the timer mode. The upper 2 bits to specify the operation.

TMOD Register:

Address: 089H (Bit addressable)							
TMOD.7	TMOD.6	TMOD.5	TMOD.4	TMOD.3	TMOD.2	TMOD.1	TMOD.0
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Timer1	Timer0						
GATE	C/T	M1	M0	GATE	C/T	M1	M0

Description of All the Bits of TMOD:

Timer	**Bit**	**Function**
Timer1	GATE	GATE enables and disables Timer by means of a signal brought to the INTx pin: 1 – Timer operates only if the INTx bit is set. 0 – Timer operates regardless of the logic state of the INTx bit.
	C/T	C/T selects pulses to be counted up by the timer/counter: 1 – Timer counts pulses brought to the Tx(Timer) pin. 0 – Timer counts pulses from the internal oscillator.

Timer	Bit	Function
	M1	M1, M0 These two bits select the operational mode Timer.
	M0	M1, M0 These two bits select the operational mode Timer.
Timer0	GATE	GATE enables and disables Timer by means of a signal brought to the INTx pin: 1 – Timer operates only if the INTx bit is set. 0 – Timer operates regardless of the logic state of the INTx bit.
	C/T	C/T selects pulses to be counted up by the timer/counter: 1 – Timer counts pulses brought to the Tx(Timer) pin. 0 – Timer counts pulses from the internal oscillator.
	M1	M1, M0 These two bits select the operational mode Timer.
	M0	M1, M0 These two bits select the operational mode Timer.

Timer Mode Control Bits:

M1	M0	Mode	Operating Mode
0	0	0	13-bit Mode
0	1	1	16-bit Mode
1	0	2	8-bit Auto Reaload Mode
1	1	3	Split Timer Mode

Modes of Operation

Explain different timer modes of 8051 microcontroller.

Mode 0: 13-Bit Timer

- **Configuration:** The timer register is split into two parts:
  - Five high-order bits (THx)
  - Eight low-order bits (TLx), with the top 3 bits of TLx written as zeroes.
- **Operation:** The 5 bits of TLx are automatically incremented. When TLx overflows, it increments THx. This forms a 13-bit timer.
- **Use Cases:** Often used for event counting or generating baud rates in serial communication, particularly when interfacing with legacy systems.

Mode 1: 16-Bit Timer

- **Configuration:** The full 16-bits of the Timer register (THx and TLx) function as a single timer unit.
- **Operation:** Each clock pulse increments the entire register.
- **Use Cases** General-purpose time delays, long interval measurements, anything requiring 16-bit precision timing.

Mode 2: 8-Bit Auto-Reload Timer

- **Configuration:**
  - THx holds a fixed reload value.
  - TLx operates as the 8-bit timer.
- **Operation:**
  - TLx counts up. When it overflows, it's automatically reloaded with the value stored in THx.
  - This creates a recurring time interval.
- **Use Cases:** Generating fixed, predictable time delays or timing periodic events.

Mode 3: Split 8-bit Timers

- **Configuration:**
  - Timer 0 is split into two independent 8-bit timers/counters: TL0 and TH0.
  - Timer 1 remains as a 16-bit timer if needed.
- **Operation:**
  - TL0 and TH0 function as two separate timers, often with TL0 used as a timer and TH0 used as a counter.
- **Use Cases:**
  - Situations requiring two independent timers
  - Generating baud rates (TL0) while counting external events (TH0)

Key Control Registers

- **TMOD (Timer Mode):** This register selects the operating mode for Timer 0 and Timer 1.
- **TCON (Timer Control):** Contains flags and start/stop control bits for the timers.

How to Select a Mode

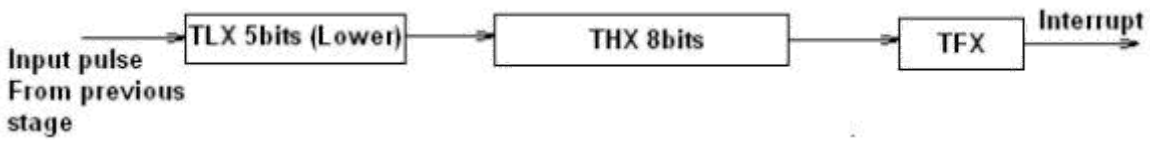
Mode selection depends on:

- **Timing Precision:** 16-bit vs. 8-bit
- **Recurring Intervals:** Auto-reload mode vs. manual restart.
- **Number of Timers Needed:** Split timer mode provides two independent 8-bit timers if needed within Timer 0.

Timers/Counters logic diagram and its operation in various modes.

TIMER MODE 0 (13 bit mode)

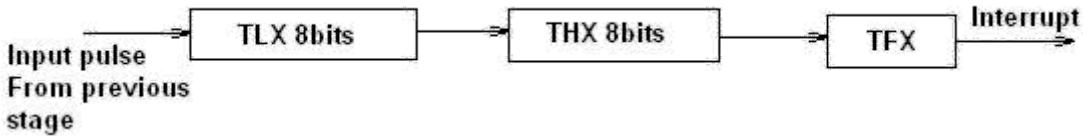
MODE 0 is a 13 bit mode. In this mode the THx acts as an 8 bit timer & TLx acts as a 5 bit timer. The TLx counts up to 31 & then resets to 00 & increment THx by 1. Suppose you load 0 in the timer then the timer will overflow in 2^13 i.e. 8192 machine cycles.



Mode 0 is exactly same like mode 1 except that it is a 13-bit timer instead of 16-bit. The 13-bit counter can hold values between 0000 to 1FFFH in TH-TL. Therefore, when the timer reaches its maximum of 1FFH, it rolls over to 0000, and TF is raised.

TIMER MODE 1 (16 bit mode)

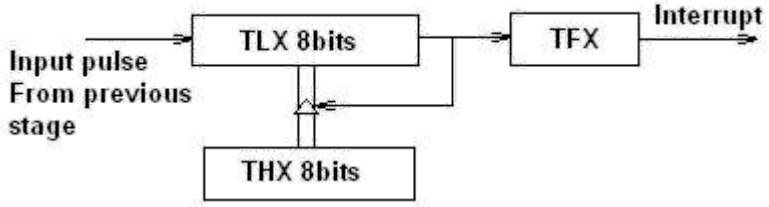
MODE 1 is similar to MODE 0 except it is a 16 bit mode. In this mode the THx & TLx both acts as an 8 bit timer. The TLx counts upto 255 & then resets to 00 & increment THx by 1. Since this is a full 16 bit timer we can get maximum of 2^16 i.e. 65536 Machine cycle before the timer overflows.



It is a 16-bit timer; therefore it allows values from 0000 to FFFFH to be loaded into the timer's registers TL and TH. After TH and TL are loaded with a 16-bit initial value, the timer must be started. We can do it by "SETB TR0" for timer 0 and "SETB TR1" for timer 1. After the timer is started. It starts count up until it reaches its limit of FFFFH. When it rolls over from FFFF to 0000H, it sets high a flag bit called TF (timer flag). This timer flag can be monitored. When this timer flag is raised, one option would be stop the timer with the instructions "CLR TR0" or CLR TR1 for timer 0 and timer 1 respectively. Again, it must be noted that each timer flag TF0 for timer 0 and TF1 for timer1. After the timer reaches its limit and rolls over, in order to repeat the process the registers TH and TL must be reloaded with the original value and TF must be reset to 0.

TIMER MODE 2 (8 bit mode)

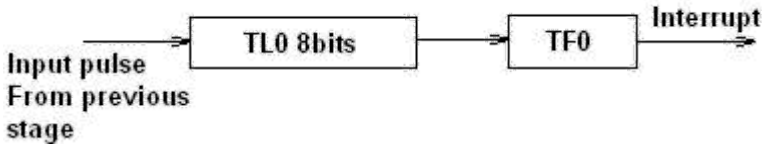
In this Mode TLx acts as the timer & THx contains the Reload Value i.e. THx is loaded in TLx everytime it overflows i.e. when TLx reaches 255 & is incremented then instead of resetting it to 0 it will be reset to the value stored in THx. This mode is very commonly used for generating baud rate used in serial communication.



It is an 8 bit timer that allows only values of 00 to FFH to be loaded into the timer's register TH. After TH is loaded with 8 bit value, the 8051 gives a copy of it to TL. Then the timer must be started. It is done by the instruction "SETB TR0" for timer 0 and "SETB TR1" for timer1. This is like mode 1. After timer is started, it starts to count up by incrementing the TL register. It counts up until it reaches its limit of FFH. When it rolls over from FFH to 00. It sets high the TF (timer flag). If we are using timer 0, TF0 goes high; if using TF1 then TF1 is raised. When Tl register rolls from FFH to 00 and TF is set to 1, TL is reloaded automatically with the original value kept by the TH register. To repeat the process, we must simply clear TF and let it go without any need by the programmer to reload the original value. This makes mode 2 auto reload, in contrast in mode 1 in which programmer has to reload TH and TL.

TIMER MODE 3 (Split Mode)

Timer mode “3” is known as split-timer mode. Timers 0 and 1 may be programmed to be in mode 0, 1, or 2 independently of a similar mode for the other timer. But in mode 3 the timers do not operate independently, if mode 3 is chosen for timer 0. When Timer 0 is placed in mode 3, it essentially becomes two separate 8-bit timers. Timer 0 is TL0 and Timer 1 is TH0. Both timers count from 0 to 255 and overflow back to 0. All the bits that are related to Timer 1 will now be tied to TH0. Now placing timer 1 in mode 3 causes it to stop counting, the control bit TR1 and the Timer 1 flag TF1 are now used by timer 0. So even if you use Timer 1 in Mode 0, 1 or 2 you won’t be able to START or STOP the timer & no INTERRUPT will be generated by Timer 1. The real Timer 1 will be incremented every machine cycle no matter what.



Mode 3 is also known as a split timer mode. Timer 0 and 1 may be programmed to be in mode 0, 1 and 2 independently of similar mode for other timer. This is not true for mode 3; timers do not operate independently if mode 3 is chosen for timer 0. Placing timer 1 in mode 3 causes it to stop counting; the control bit TR1 and the timer 1 flag TF1 are then used by timer0.

Serial Communication

What is Serial Communication?

- Serial communication is a method of transmitting data between devices where the bits of data are sent sequentially over a single communication line or channel. This contrasts with parallel communication where multiple bits are sent at the same time over multiple lines.

Key Modes of Serial Communication

1. **Simplex Mode**
    - **Unidirectional transmission:** Data flows in only one direction, from the transmitter to the receiver.
    - **Example:** A TV broadcasting station transmits signals to countless television sets (receivers). TVs can't transmit back to the station.
2. **Half-Duplex Mode**
    - **Bidirectional, but not simultaneous:** Both devices can transmit and receive, but not at the same time. They must take turns.
    - **Example:** Walkie-talkies. A user presses a button to talk (transmit) and releases the button to listen (receive). Both users cannot talk at the same time.
3. **Full-Duplex Mode**
    - **Simultaneous bidirectional transmission:** Both devices can transmit and receive data at the same time.
    - **Example:** Modern phone calls. Both parties can talk and listen simultaneously.

Asynchronous vs. Synchronous Communication

Within serial communication, there's an important distinction between asynchronous and synchronous modes:

- **Asynchronous:**
    - No shared clock signal between devices.

    - Data is framed using start and stop bits to signal the beginning and end of a data packet.
    - Good for irregular data transmission with potential gaps between bytes.
- **Synchronous:**
    - Devices share a clock signal that synchronizes the timing of data transmission.
    - Data bytes flow in a continuous stream without the need for start and stop bits.
    - Ideal for high-speed, continuous data transmission.

Example: UART Communication (Typically Asynchronous)

- **Hardware:** A Universal Asynchronous Receiver/Transmitter (UART) is a common hardware component for serial communication.
- **Protocol:** Data is framed with a start bit, 5-9 data bits, an optional parity bit (for error checking), and one or more stop bits.
- **Transmission:**
    - The transmitter sends out the start bit (logic low)
    - Data bits are sent one by one (least significant bit first)
    - Parity bit (if used) is sent
    - Stop bit (logic high) signals packet's end
- **Example Use Case:** Computer sending commands to a microcontroller over a serial connection.

Note: Other serial protocols exist, such as SPI (Serial Peripheral Interface) and I2C (Inter-Integrated Circuit), each with specific features and applications.

Modes

The SCON Register's Role

The SCON (Serial Control) register holds bits that determine the operating mode, baud rate, and other serial communication settings for the 8051's integrated UART. The pertinent bits are:

- **SM0, SM1:** Serial Mode selection bits.
- **REN:** Receive enable bit.

Serial Modes in 8051

The 8051 supports four primary serial communication modes, as outlined below:

1. Mode 0 (Shift Register Mode)

- **Synchronous:** Data is transmitted and received with clock pulses generated on the TxD pin (transmit data pin) of the 8051. RxD (receive data pin) is used for receiving data.
- **Framing:** Data transmission occurs in bytes (8-bit frames) without the overhead of start and stop bits.
- **Baud Rate:** Fixed at 1/12th of the microcontroller's oscillator frequency.

2. Mode 1 (10-bit UART)

- **Asynchronous:** No shared clock signal between devices. Start and stop bits frame each byte for synchronization.
- **Framing:** 1 start bit, 8 data bits, and 1 stop bit.

- **Baud Rate:** Variable, usually determined by using Timer 1 to generate the baud-rate ticks.
- **Common Use:** General-purpose serial communication with external devices.

3. Mode 2 (11-bit UART)

- **Asynchronous:** Same principle as Mode 1.
- **Framing:** 1 start bit, 8 data bits, a programmable 9th bit, and 1 stop bit.
- **9th Bit:** Can be used as an extra data bit, a parity bit (for error checking), or for multiprocessor communication.
- **Baud Rate:** Variable, often calculated using Timer 1.

4. Mode 3 (9-bit UART)

- **Similar to Mode 2:** Asynchronous with a programmable 9th bit.
- **Framing:** 1 start bit, 8 data bits, and 1 stop bit.
- **Key Difference:** The 9th bit is always transmitted as '1'.
- **Baud Rate:** Variable, based on calculations with Timer 1.

Mode Selection & Configuration Example

Let's configure the 8051 for serial communication in Mode 1, a classic UART setup:

1. **Mode Setting:**

```
SCON = 0x50; // SM0 = 0, SM1 = 1 (Mode 1), REN = 1 (enable reception)
```

2. **Baud Rate Calculation:** Determine the desired baud rate and calculate the appropriate reload value to load into the TH1 register used as the baud rate generator. (Consult your microcontroller datasheet for the calculation formula).

Remember:

- To transmit data, load the byte to be sent into the SBUF (Serial Buffer) register. Hardware handles the rest.
- Reception often involves setting up interrupts to detect when the RI flag in SCON is set, indicating received data.

Explain Serial Communication in various modes.

The serial port of 8051 is full duplex, i.e., it can transmit and receive simultaneously. The register SBUF is used to hold the data. The special function register SBUF is physically two registers. One is, write-only and is used to hold data to be transmitted out of the 8051 via TXD. The other is, read-only and holds the received data from external sources via RXD. Both mutually exclusive registers have the same address 099H.

Data Transmission:

Transmission of serial data begins at any time when data is written to SBUF.

Pin P3.1 (Alternate function bit TXD) is used to transmit data to the serial data network.

TI is set to 1 when data has been transmitted. This signifies that SBUF is empty so that another byte can be sent.

Data Reception :

Reception of serial data begins if the receive enable bit is set to 1 for all modes.

Pin P3.0 (Alternate function bit RXD) is used to receive data from the serial data network.

Receive interrupt flag, RI, is set after the data has been received in all modes. The data gets stored in SBUF register from where it can be read.

Serial Data Modes:

8051 micro controller communicate with another peripheral device through RXD and TXD pin of port3.controller have four mode of serial communication.this four mode of serial communication are below.

1. Serial data mode 0-fixed buad Rate.
2. Serial data mode 1-variable baud rate.
3. Serial data mode 2 -fixed baud Rate.
4. Serial Data mode 3 -variable baud rate.

Serial Data Mode-0 (Baud Rate Fixed):

In this mode, the serial port works like a shift register and the data transmission works synchronously with a clock frequency of fosc /12.

Serial data is received and transmitted through RXD. 8 bits are transmitted/ received aty a time.

Pin TXD outputs the shift clock pulses of frequency fosc /12, which is connected to the external circuitry for synchronization.

The shift frequency or baud rate is always 1/12 of the oscillator frequency.

Serial Data Mode-1 (standard UART mode) (baud rate is variable):

In mode-1, the serial port functions as a standard Universal Asynchronous Receiver Transmitter (UART) mode. 10 bits are transmitted through TXD or received through RXD.

The 10 bits consist of one start bit (which is usually '0'), 8 data bits (LSB is sent first/received first), and a stop bit (which is usually '1'). Once received, the stop bit goes into RB8 in the special function register SCON. The baud rate is variable.

Serial Data Mode-2 Multiprocessor (baud rate is fixed):

In this mode 11 bits are transmitted through TXD or received through RXD.

The various bits are as follows: a start bit (usually '0'), 8 data bits (LSB first), a programmable 9 th (TB8 or RB8)bit and a stop bit (usually '1').

While transmitting, the 9 th data bit (TB8 in SCON) can be assigned the value '0' or '1'. For example, if the information of parity is to be transmitted, the parity bit (P) in PSW could be moved into TB8.

On reception of the data, the 9 th bit goes into RB8 in 'SCON', while the stop bit is ignored.

The baud rate is programmable to either 1/32 or 1/64 of the oscillator frequency.

fbaud = (2^SMOD/64)*fosc

Serial Data Mode-3 - Multi processor mode (Variable baud rate):

In this mode 11 bits are transmitted through TXD or received through RXD.

The various bits are: a start bit (usually '0'), 8 data bits (LSB first), a programmable 9 th bit and a stop bit (usually '1').

Mode-3 is same as mode-2, except the fact that the baud rate in mode-3 is variable (i.e., just as in mode-1).

SCON Register

What is the SCON Register?

- The SCON (Serial Control) register is an 8-bit, bit-addressable Special Function Register (SFR) responsible for managing serial communication in 8051 microcontrollers.
- It holds settings and status flags that control how the microcontroller sends and receives data serially.

SCON Register Structure

Here's how the SCON register's bits function:

- **SM0, SM1 (Serial Mode Selection Bits):** These bits define the serial communication mode for the 8051. There are four primary modes:
  - **Mode 0:** 8-bit shift register for serial port output, clock for serial port input is generated internally.
  - **Mode 1:** 10-bit UART mode (8 data bits, 1 start bit, 1 stop bit).
  - **Mode 2:** 11-bit UART mode (8 data bits, 1 start bit, 1 programmable stop bit, 1 additional bit for addressing or other purposes)
  - **Mode 3:** Similar to mode 2, but with 9 data bits.
- **SM2 (Enable Multiprocessor Communication):** Specifically designed for multiprocessor systems to distinguish between data from other processors and address information.
- **REN (Receive Enable):**
  - '1' = Enables serial reception
  - '0' = Disables serial reception.
- **TB8 (Transmit Bit 8):** In Modes 2 and 3, this is the 9th data bit that is transmitted.
- **RB8 (Receive Bit 8):** There are two interpretations:
  - Modes 1, 2, and 3: This is the 9th data bit received.
  - Mode 0: RB8 becomes the stop bit when received.
- **TI (Transmit Interrupt Flag):**
  - '1' = Signals that the transmit buffer is empty, ready for new data (set by hardware).
  - '0' = Transmission is in progress (cleared by software).
- **RI (Receive Interrupt Flag):**
  - '1' = Signals that the receive buffer is full (set by hardware).
  - '0' = No data in the buffer to be read (cleared by software).

Example: Setting up Serial Communication in Mode 1 (10-bit UART)

1. **Mode Selection:** Set the SM0 and SM1 bits in the SCON register:

```
SCON = 0x50; // Mode 1: 10-bit UART, Receiver Enabled
```

2. **Baud Rate Calculation:** Determine the desired baud rate and calculate the appropriate reload value for the TH1 register (which acts as the baud rate generator). Consult your 8051 microcontroller datasheet for the calculation formula.
3. **Enabling Reception (if needed):** Set the REN bit in SCON to '1' to enable incoming serial data reception.
4. **Transmitting Data:**
    - Wait for the TI flag in SCON to become '1' (meaning the transmit buffer is empty).
    - Load the data byte to be transmitted into the SBUF register. The hardware handles the rest for you.

Note: To receive data, you'll usually create an interrupt service routine (ISR) that triggers when the RI flag in SCON is set.

Serial Port Control Register (SCON): Register SCON controls serial data communication. The serial port control and status register is the Special Function Register SCON. This register contains not only the mode selection bits, but also the 9th data bit for transmit and receive (TB8 and RB8), and the serial ports interrupt bits (TI and RI).

SCON Register:

Address: 098H (Bit addressable)							
SCON.7	SCON.6	SCON.5	SCON.4	SCON.3	SCON.2	SCON.1	SCON.0
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
SM0	SM1	SM2	REN	TB8	RB8	TI	RI

Description of All the Bits of SCON:

SM0	**These 2 bits determine the framing of data by specifying number of bits per**
SM1	character and start and stop bits. they take following combo.SM0
SM2	This enables multiprocessing capabilities of 8051. Usually set to 0
REN	Also referred to as SCON.4 as SCON is a bit addressable register. This is receive enable. When high or 1 it allows 8051 to receive data from RxD pin. Used or access as SET SCON.4 and CLR SCON.4. very useful in blocking external serial reception.
TB8	Transfer bit 8. Used for serial mode 2 and 3 not generally used so set it always to 0
RB8	Receive bit 8. Again used for serial mode 2 and 3 not used so set it to 0
TI	Transmit interrupt. Important flag bit in SCON register. When 8051 finishes transfer of 8 bit character, it raises the T1 flag to indicate that it is ready to transfer another byte. Is used at beginning of stop bit.
RI	Receive interrupt. Another important flag bit in SCON register. When 8051 finishes receiving data i.e when data is successfully stored in SBUF it raises R1 flag to indicate byte is received and to be picked before it gets lost.

SMO	SM1	Mode	Baud Rate	Description
0	0	Mode 0	Fixed Baud Rate (fosc/12)	8-Bit Synchronous Shift Register Mode
0	1	Mode 1	Variable Baud Rate (Can be set by Timer 1)	8-bit Standard UART mode
1	0	Mode 2	Fixed Baud Rate (fosc/64) or (fosc/32)	9-bit Multiprocessor Comm. mode
1	1	Mode 3	Variable Baud Rate (Can be set by Timer 1)	9-bit Multiprocessor Comm. mode

PCON Register

What is the PCON Register?

- The PCON (Power Control) register is an 8-bit Special Function Register (SFR) primarily used to manage power-saving modes within the 8051 microcontroller.
- It also includes a few additional control bits for baud rate adjustment and general-purpose usage.

PCON Register Structure

Here's a breakdown of the bits within the PCON register:

- **SMOD (Serial Mode Doubler):**
  - '1' = Doubles the baud rate for serial communication (UART) when Timer 1 is used for baud rate generation. Useful for increasing communication speeds.
  - '0' = Normal baud rate.
- **GF1 (General Purpose Flag 1), GF0 (General Purpose Flag 0):**
  - These bits can be set and cleared by software for various purposes chosen by the programmer. They have no predefined function assigned to them.
- **PD (Power-Down Mode):**
  - '1' = Enables Power-Down Mode. In this state, the oscillator is stopped to reduce power consumption dramatically.
  - '0' = Disables Power-Down Mode, the microcontroller runs normally.
- **IDL (Idle Mode):**
  - '1' = Enables Idle Mode. The CPU stops functioning, but peripherals like timers, serial ports, and interrupts remain active. This mode reduces power consumption while maintaining some functionality.
  - '0' = Disables Idle Mode.

Key Points about Power Modes

- **Exiting Power-Down Mode:** The microcontroller can only exit Power-Down mode with a hardware reset.
- **Exiting Idle Mode:** The microcontroller exits Idle mode upon an interrupt or a hardware reset.

Examples

1. **Enabling Power-Down Mode**

```
PCON |= 0x01; // Set the PD bit (bit 0) of PCON to '1'
```

2. **Enabling Idle Mode**

```
PCON |= 0x02; // Set the IDL bit (bit 1) of PCON to '1'
```

3. **Doubling Serial Communication Baud Rate**

```
PCON |= 0x80; // Set the SMOD bit (bit 7) of PCON to '1' (assuming you want
to double the baud rate)
```

Important Note: It is crucial to check your specific microcontroller datasheet, as certain manufacturers might have slightly different or additional assignments for the remaining unused bits in the PCON register.

Power Mode control Register (PCON): Register PCON controls processor powerdown, sleep modes and serial data bandrate. Only one bit of PCON is used with respect to serial communication. The seventh bit (b7)(SMOD) is used to generate the baud rate of serial communication.

The PCON or Power Control register, as the name suggests is used to control the 8051 Microcontroller's Power Modes and is located at 87H of the SFR Memory Space. Using two bits in the PCON Register, the microcontroller can be set to Idle Mode and Power Down Mode.

During Idle Mode, the Microcontroller will stop the Clock Signal to the ALU (CPU) but it is given to other peripherals like Timer, Serial, Interrupts, etc. In order to terminate the Idle Mode, you have to use an Interrupt or Hardware Reset.

In the Power Down Mode, the oscillator will be stopped and the power will be reduced to 2V. To terminate the Power Down Mode, you have to use the Hardware Reset.

Apart from these two, the PCON Register can also be used for few additional purposes. The SMOD Bit in the PCON Register is used to control the Baud Rate of the Serial Port.

There are two general purpose Flag Bits in the PCON Register, which can be used by the programmer during execution.

PCON Register:

Address: 087H (Byte addressable)							
PCON.7	PCON.6	PCON.5	PCON.4	PCON.3	PCON.2	PCON.1	PCON.0
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
SMOD	-	-	-	GF1	GF0	PD	IDL

Description of All the Bits of PCON:

Bit	Function
SMOD	Serial baud rate MODify bit – If SMOD = 1, the Baud rate is doubled when the serial port is used in mode 1,2 and 3
GF1	General Purpose Flag Bit 1
GF0	General Purpose Flag Bit 0
PD	Power Down Mode. If set, the oscillator is stopped. A reset or an interrupt can cancel this mode.
IDL	Idle Mode. If set, the CPU is stopped. A reset or an interrupt can cancel this mode.

Interrupts

Q3c: Explain interrupts of 8051 microcontroller.

What is an Interrupt?

- An interrupt is an event that temporarily suspends the normal execution of a program and forces the 8051 to execute a special routine called an Interrupt Service Routine (ISR).
- Interrupts allow the microcontroller to respond quickly to important events (e.g., button presses, timer overflow, data received) without needing to constantly poll for them in the main code.

Types of Interrupts in the 8051

1. **External Interrupts:**
    - **INT0 (Pin P3.2):** Triggered by a low-to-high transition on the INT0 pin.
    - **INT1 (Pin P3.3):** Triggered by a low-to-high transition on the INT1 pin.
2. **Timer Interrupts:**
    - **TF0 (Timer 0 Overflow):** Triggered when Timer 0 overflows.
    - **TF1 (Timer 1 Overflow):** Triggered when Timer 1 overflows.
3. **Serial Interrupt:**
    - **RI/TI (Receive Interrupt/Transmit Interrupt):** Triggered when the serial port finishes receiving a byte (RI) or transmitting a byte (TI).

Interrupt Process

1. **Trigger:** An interrupt source (external pin, timer overflow, etc.) is triggered.
2. **Completion of Current Instruction:** The 8051 completes executing its current instruction.
3. **Saving State:** The microcontroller automatically pushes the current Program Counter (PC) onto the stack.
4. **Jump to ISR:** The 8051 jumps to the pre-determined memory address of the corresponding Interrupt Service Routine (ISR).
5. **ISR Execution:** The ISR code executes, handling the event that triggered the interrupt.
6. **Returning:** After the ISR completes, a 'RETI' instruction pops the PC value from the stack, resuming the original program flow.

Interrupt Control Registers

- **IE (Interrupt Enable):** Enables or disables specific interrupts globally and individually within the system.
- **IP (Interrupt Priority):** Assigns priority levels to each interrupt source. If multiple interrupts occur simultaneously, the one with higher priority is serviced first.

Key Points

- **Priority:** The 8051 has a fixed interrupt priority structure (e.g., INT0 has the highest priority).
- **Masking:** Interrupts can be turned on or off selectively using the IE register.
- **Nesting:** Interrupts can potentially interrupt other interrupts, depending on their priority.

Example

Imagine an 8051 system monitoring a sensor. A timer interrupt might trigger periodically to read the sensor value, while an external interrupt could signal a critical threshold being exceeded, requiring immediate action.

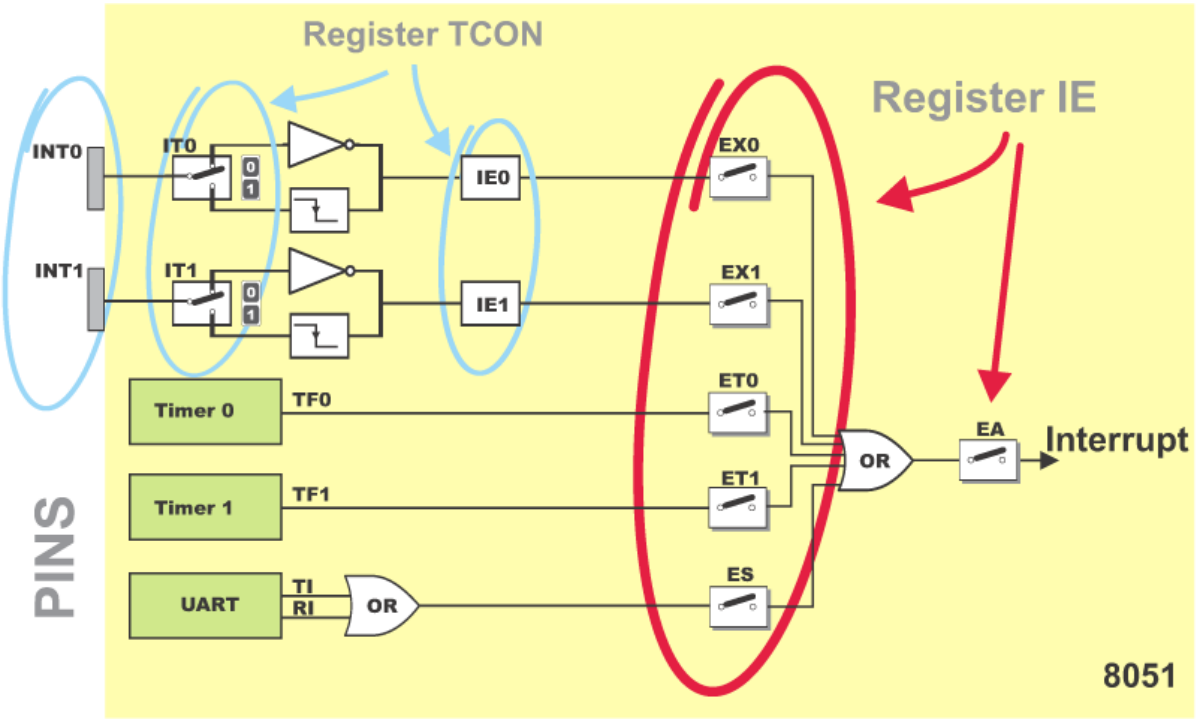
Interrupt Vector Addresses

Interrupt structure, vector address, priority and operation.

An interrupt is an event that occurs randomly in the flow of continuity. It is just like a call you have when you are busy with some work and depending upon call priority you decide whether to attend or neglect it.

Same thing happens in microcontrollers. 8051 architecture handles 5 interrupt sources, out of which two are internal (Timer Interrupts), two are external and one is a serial interrupt. Each of these interrupts has their interrupt vector address. Highest priority interrupt is the Reset, with vector address 0x0000.

Now, it is necessary to explain a few details referring to external interrupts- INT0 and INT1. If the IT0 and IT1 bits of the TCON register are set, an interrupt will be generated on high to low transition, i.e. on the falling pulse edge (only in that moment). If these bits are cleared, an interrupt will be continuously executed as far as the pins are held low.



Vector Address: This is the address where controller jumps after the interrupt to serve the ISR (interrupt service routine).

Interrupt	Flag	Interrupt vector address
Reset	-	0000H
INT0 (Ext. int. 0)	IE0	0003H
Timer 0	TF0	000BH
INT1 (Ext. int. 1)	IE1	0013H
Timer 1	TF1	001BH
Serial	TI/RI	0023H

Reset: Reset is the highest priority interrupt, upon reset 8051 microcontroller start executing code from 0x0000 address.

Internal interrupt (Timer Interrupt): 8051 has two internal interrupts namely timer0 and timer1. Whenever timer overflows, timer overflow flags (TF0/TF1) are set. Then the microcontroller jumps to their vector address to serve the interrupt. For this, global and timer interrupt should be enabled.

Serial interrupt: 8051 has serial communication port and have related serial interrupt flags (TI/RI). When the last bit (stop bit) of a byte is transmitted, TI serial interrupt flag is set and when last bit (stop bit) of receiving data byte is received, RI flag get set.

IE Register

What is the IE Register?

- The IE (Interrupt Enable) register is an 8-bit, bit-addressable Special Function Register (SFR) within 8051 microcontrollers.

- Each bit in this register controls the enabling or disabling of specific interrupts within the system.

IE Register Structure

Here's the breakdown of the IE Register's bit functionality:

- **EA (Enable All):**
  - '1' = Enables all interrupt sources (if their individual bits are also set to '1').
  - '0' = Disables all interrupts, regardless of other bit settings.
- **Unused (3 bits):** These bits are typically reserved and have no assigned functionality.
- **ES (Enable Serial Interrupt):**
  - '1' = Enables serial port interrupt.
  - '0' = Disables serial port interrupt.
- **ET1 (Enable Timer 1 Interrupt):**
  - '1' = Enables the interrupt generated by Timer 1 overflow.
  - '0' = Disables the Timer 1 interrupt.
- **EX1 (Enable External Interrupt 1):**
  - '1' = Enables the external interrupt 1.
  - '0' = Disables the external interrupt 1.
- **ET0 (Enable Timer 0 Interrupt):**
  - '1' = Enables the interrupt generated by Timer 0 overflow.
  - '0' = Disables the Timer 0 interrupt.
- **EX0 (Enable External Interrupt 0):**
  - '1' = Enables the external interrupt 0.
  - '0' = Disables the external interrupt 0.

How Interrupts Work with IE

1. **Global Enable:** The EA bit in the IE register must be set to '1' for any interrupt to function.
2. **Individual Enable:** Even if EA is set to '1', a specific interrupt request will be recognized only if the corresponding bit in the IE register is also set to '1'.

Examples

1. **Enabling All Interrupts**

```
IE = 0xFF;  // Set all bits in IE to '1'
```

2. **Enabling Only Timer 0 and External Interrupt 1**

```
IE = 0x89;  // Sets the ET0 and EX1 bits, the rest are '0'
```

Important Note:

- Interrupts must also be configured in other registers for them to be active. For example:
  - Timer interrupts require the timers to be started (TRx = '1' in TCON).

    - External interrupts may need edge or level triggering configured (ITx bits in TCON).
- The 8051 has a priority system for multiple simultaneous interrupts. You can control the priority using the IP (Interrupt Priority) register.

IE Register:

Address: 0A8H (Byte addressable)							
IE.7	IE.6	IE.5	IE.4	IE.3	IE.2	IE.1	IE.0
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
EA	-	-	ES	ET1	EX1	ET0	EX0

Description of All the Bits of IE:

EA	**Global interrupt enable/disable Bit**
ES	Enable Serial Interrupt Bit
ET1	Enable Timer1 Interrupt Bit
EX1	Enable External Interrupt 1 Bit
ET0	Enable Timer0 Interrupt Bit
EX0	Enable External Interrupt 1 Bit

Priorities

IP Register

What is the IP Register?

- The IP (Interrupt Priority) register is an 8-bit, bit-addressable Special Function Register (SFR) used to manage the priority of interrupt sources in 8051 microcontrollers.
- When multiple interrupts occur simultaneously, the IP register helps the system determine which interrupt to handle first.

IP Register Structure

Each bit in the IP register is assigned a specific interrupt source, providing two levels of priority (high or low):

- **Unused (3 bits):** These bits are typically reserved and have no assigned functionality.
- **PS (Serial Interrupt Priority):**
    - '1' = High priority.
    - '0' = Low priority.
- **PT1 (Timer 1 Interrupt Priority):**
    - '1' = High priority.
    - '0' = Low priority.

- **PX1 (External Interrupt 1 Priority):**
  - '1' = High priority.
  - '0' = Low priority.
- **PT0 (Timer 0 Interrupt Priority):**
  - '1' = High priority.
  - '0' = Low priority.
- **PX0 (External Interrupt 0 Priority):**
  - '1' = High priority.
  - '0' = Low priority.

How Interrupt Priorities Work with IP

1. **Interrupt Occurrence:** When one or more interrupts occur, the 8051 checks the corresponding bits in the IP register.
2. **Priority Handling**
   - Higher priority interrupts always take precedence over lower priority interrupts.
   - If multiple interrupts of the same priority level occur, then the 8051 uses a predefined internal polling sequence to determine the order for servicing the interrupts.

Examples

1. **Configuring Timer 0 as Highest Priority, External Interrupt 1 as Lowest**

```
IP = 0x12; // Sets PT0 to '1' (high), PX1 to '0' (low), others remain '0'
```

2. **Setting All Interrupts to Low Priority**

```
IP = 0x00; // All bits set to '0' for low priority
```

Important Notes:

- The IP register only determines the priority among simultaneously occurring interrupts. The interrupt itself still needs to be enabled globally (EA bit in the IE register) and individually (Ex and ETx bits in the IE register).
- The priority structure and internal polling sequence for the 8051 microcontroller can be found in your specific microcontroller's datasheet.

Priority to the interrupt can be assigned by using interrupt priority register (IP)

Interrupt priority after Reset:

Priority	Interrupt source	Intr. bit / flag
1	External Interrupt 0	INT0
2	Timer Interrupt 0	TF0
3	External Interrupt 1	INT1
4	Timer Interrupt 1	TF1

Priority	Interrupt source	Intr. bit / flag
5	Serial interrupt	(TI/RI)

In the table, interrupts priorities upon reset are shown. As per 8051 interrupt priorities, lowest priority interrupts are not served until microcontroller is finished with higher priority ones. In a case when two or more interrupts arrives microcontroller queues them according to priority.

It is not possible to forseen when an interrupt request will arrive. If several interrupts are enabled, it may happen that while one of them is in progress, another one is requested. In order that the microcontroller knows whether to continue operation or meet a new interrupt request, there is a priority list instructing it what to do. The priority list offers 3 levels of interrupt priority:

- Reset! The apsolute master. When a reset request arrives, everything is stopped and the microcontroller restarts.
- Interrupt priority 1 can be disabled by Reset only.
- Interrupt priority 0 can be disabled by both Reset and interrupt priority 1.

The IP Register (Interrupt Priority Register) specifies which one of existing interrupt sources have higher and which one has lower priority. Interrupt priority is usually specified at the beginning of the program. According to that, there are several possibilities:

- If an interrupt of higher priority arrives while an interrupt is in progress, it will be immediately stopped and the higher priority interrupt will be executed first.
- If two interrupt requests, at different priority levels, arrive at the same time then the higher priority interrupt is serviced first.
- If the both interrupt requests, at the same priority level, occur one after another, the one which came later has to wait until routine being in progress ends.
- If two interrupt requests of equal priority arrive at the same time then the interrupt to be serviced is selected according to the following priority list:
    - External interrupt INT0
    - Timer 0 interrupt
    - External Interrupt INT1
    - Timer 1 interrupt
    - Serial Communication Interrupt

IP Register:

Address: 0B8H (Byte addressable)							
IP.7	IP.6	IP.5	IP.4	IP.3	IP.2	IP.1	IP.0
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
-	-	-	PS	PT1	PX1	PT0	PX0

Description of All the Bits of IP:

PS	Serial Interrupt Priority Bit
PT1	Timer1 Interrupt Priority Bit
PX1	External Interrupt 1 Priority Bit
PT0	Timer0 Interrupt Priority Bit
PX0	External Interrupt 0 Priority Bit

Unit IV: 8051 Programming

Addressing Modes

What is an Addressing Mode?

An Addressing Mode is a way to locate a target Data, which is also called as Operand. The 8051 Family of Microcontrollers allows five types of Addressing Modes for addressing the Operands. They are:

- Register Addressing
- Direct Addressing
- Indirect Addressing
- Immediate Addressing
- Indexed(Base Relative Addressing with DPTR) Addressing

Key Addressing Modes in the 8051

1. **Register Addressing**
    - **How it Works:** The operand of the instruction directly specifies one of the 8051's registers (A, B, R0-R7).
    - **Example:** `MOV A, R2` (Copy the contents of R2 into the accumulator)
    - **Fast and Efficient:** No additional memory accesses are needed.
2. **Direct Addressing**
    - **How it Works:** The instruction contains an 8-bit address that directly points to a location in the internal RAM or Special Function Registers (SFRs).
    - **Example:** `MOV 45H, A` (Store the value in the accumulator into internal RAM location 45H)
    - **Accesses only first 256 bytes:** Limited to accessing the lower portion of internal RAM and SFRs.
3. **Indirect Addressing**
    - **How it Works:** The instruction specifies a register (R0 or R1) that holds the memory address of where the data actually resides.
    - **Example:** `MOV A, @R0` (Copy the byte pointed to by the address in R0 into the accumulator).
    - **Flexibility:** Allows dynamic calculation of data locations.
4. **Immediate Addressing**

- **How it Works:** The data to be used is embedded directly within the instruction itself. Preceded by the '#' symbol.
- **Example:** `MOV A, #60H` (Load the value 60H into the accumulator).
- **Convenient for constants:** Useful for loading fixed values.

5. **Indexed(Base Relative Addressing with DPTR)**
   - **How it Works**: Used for accessing external RAM. The Data Pointer (DPTR) provides a 16-bit base address, and an 8-bit offset within the instruction specifies a location relative to that base.
   - **Example**: `MOVX A, @DPTR` (Copy byte from external RAM pointed to by DPTR into the accumulator)
   - **Expanded Memory:** Access up to 64KB of external memory

Immediate addressing mode

Immediate Addressing Mode: What is it?

- In immediate addressing mode, the data (or value) to be operated on is directly included within the instruction itself.
- The symbol "#" usually indicates that the value following it is immediate data.

Key Advantages

- **Speed:** Immediate addressing is fast because the data is immediately available to the processor; no additional memory fetching is needed.
- **Simplicity:** This is a simple addressing mode, great for using fixed constants within your code.

Examples

Here are some examples of 8051 instructions using immediate addressing mode:

1. **Loading a value into the accumulator:**

```
MOV A, #50H  ; Load the value 50 (hexadecimal) into the accumulator (register A)
```

2. **Adding an immediate value to a register:**

```
ADD R2, #10  ; Add the value 10 to register R2
```

3. **Moving data using the Data Pointer (DPTR):**

```
MOV DPTR, #2500H ; Load the immediate value 2500H into the Data Pointer,
                 ; pointing to an external memory location
```

Important Note: In the 8051, immediate data is generally limited to 8-bits (0-255 or 00H to FFH).

When to Use Immediate Addressing Mode

Immediate addressing is ideal in the following situations:

- **Working with constants:** When you know the exact value at the time of writing the code.

- **Initializing variables:** Setting initial values for variables at the start of your program.
- **Performing simple calculations:** When you need to add or subtract small, fixed values.

Register addressing mode

Register Addressing Mode: The Basics

- In this mode, the operands (the data the instruction works with) are stored within the 8051's internal registers.
- The 8051 has a set of general-purpose registers named R0 through R7, along with the accumulator (A).

Why Use It

- **Speed:** Register addressing is the fastest addressing mode since data is accessed directly from the CPU's internal registers. No time is spent fetching data from external memory.
- **Efficiency:** It uses fewer instruction bytes, making your code more compact.

Example Instructions

Let's see register addressing in action:

1. **Moving data between registers:**

```
MOV R5, R1  ; Move the contents of register R1 into register R5
```

2. **Adding the contents of two registers:**

```
ADD A, R6   ; Add the contents of register R6 to the accumulator (A) and
store the result in the accumulator
```

3. **Clearing a register:**

```
CLR R0      ; Clear register R0 (set its value to 0)
```

Key Points

- Register addressing mode is heavily used in 8051 programs because of its speed and efficiency.
- You can't directly move data between two registers that aren't the Accumulator (A). You'll often see instructions temporarily using the accumulator to facilitate data transfers between registers.

Direct addressing mode

Direct Addressing Mode: Core Concept

- In direct addressing mode, the instruction contains the direct 8-bit address of the data within the 8051's internal RAM or Special Function Registers (SFRs).

Restrictions

- **Address Space:** Direct addressing can only access the following:
    - Internal RAM locations from 00H to 7FH.

- Special Function Registers (SFRs) from 80H to FFH.

Advantages

- **Reasonable speed:** It's slower than register addressing, but still relatively fast since you're accessing the internal memory.
- **Variable data:** Useful when working with variables whose location in memory might change.

Examples

1. **Loading data from Internal RAM:**

```
MOV A, 35H  ; Load the contents of internal RAM location 35H into the
accumulator (A)
```

2. **Storing data in Internal RAM:**

```
MOV 50H, A  ; Store the contents of the accumulator (A) into internal RAM
location 50H
```

3. **Controlling an output port:**

```
MOV P1, #90H ; Send the value 90H (hexadecimal) to port 1 (SFR address 90H)
```

Important Notes

- Direct addressing mode does **not** use the "#" symbol to differentiate between immediate data and addresses.
- SFRs (Special Function Registers) control the 8051's hardware peripherals and are accessed using direct addressing.

Indirect addressing mode

Indirect Addressing Mode: The Basics

- In indirect addressing mode, the instruction doesn't contain the actual data address. Instead, it holds the address of a register that points to where the data is located in memory.
- The "@" symbol precedes the register name to indicate indirect addressing.
- Indirect addressing supports two register pairs in the 8051:
  - **R0 and R1:** For accessing internal RAM
  - **DPTR (Data Pointer):** For accessing internal RAM and external RAM (if present)

Why Use It

- **Flexibility:** This is the key advantage. It allows you to calculate or dynamically change the memory location to be accessed during program execution.
- **Data Structures:** Perfect for arrays, tables, and other data structures where you need to access data elements sequentially.

Examples

1. **Accessing internal RAM using R0:**

```
MOV R0, #40H  ; Load the value 40H into register R0
MOV A, @R0    ; Load the contents of the internal RAM location pointed to by
R0 (which is 40H) into the accumulator
```

2. **Accessing external RAM using DPTR:**

```
MOV DPTR, #3000H  ; Load the value 3000H into the Data Pointer
MOVX A, @DPTR     ; Load the contents of external RAM location 3000H into the
accumulator
                      ; (note the use of MOVX for external memory)
```

3. **Table Lookup:**

```
MOV R0, #TABLE_START  ; Load the starting address of a table into R0
MOV A, @R0            ; Load the first element of the table into the
accumulator
INC R0                ; Increment R0 to point to the next element
; ... repeat as needed
```

Key Points

- Indirect addressing gives you much more flexibility for accessing data in different memory locations.
- Using 'MOVX' is necessary for accessing external memory with indirect addressing.

Indexed addressing mode

Indexed Addressing Mode: Combining Base + Offset

- Indexed addressing mode provides a way to access data in the Program Memory (code memory) of the 8051.
- It combines the contents of a base register (either the Data Pointer - DPTR – or the Program Counter - PC) with the contents of the Accumulator (A) to form the effective address of the data you want to access.

Why use it?

- **Accessing data tables in program memory:** Ideal for working with tables or arrays of data stored in the code space of the 8051.

Key Points

- The instruction "MOVC" is used for indexed addressing in the 8051. The 'C' indicates that the instruction is accessing Code memory.

Examples

1. **Using the Data Pointer (DPTR):**

```
MOV DPTR, #MY_TABLE   ; Load the starting address of 'MY_TABLE' into DPTR
MOV A, #03H           ; Load index '3' into accumulator
MOVC A, @A+DPTR       ; Fetch the data at the address (DPTR value + 3)
                          ; from program memory and load it into the
accumulator
```

2. **Using the Program Counter (PC):**

```
MOV A, #05H           ; Load index '5' into accumulator
MOVC A, @A+PC         ; Fetch the data at the address (PC value + 5)
                          ; from program memory and load it into the
accumulator
```

Explanation of Example 1

- Let's say 'MY_TABLE' starts at program memory address 2000H.
- With the index (offset) of 3 in the accumulator, the instruction `MOVC A, @A+DPTR` will access the data stored at address 2003H in program memory.

Important Notes

- Indexed addressing is limited to accessing data within the program memory of the 8051 microcontroller.
- It cannot be used to modify the program memory itself (ROM is read-only).

Relative addressing mode

Relative Addressing Mode: Jumping by Offset

- In relative addressing mode, the instruction contains an 8-bit signed offset (a value that can be positive or negative). This offset is added to the Program Counter (PC) to determine the address of the next instruction to be executed.
- Primarily used for conditional jumps and branching within your code.

Why Use It?

- **Code Relocatability:** Code using relative addressing becomes position-independent. This means you can move the code block to a different memory location without modifying branch instructions.
- **Conditional Branching:** Ideal for instructions like `SJMP` (Short Jump), `JZ` (Jump if Zero), etc.

Example

```
HERE:   MOV A, R0   ; Some instructions...
        JZ  TARGET  ; Jump to 'TARGET' if the accumulator is zero
        ...         ; More instructions...
TARGET: INC R5      ; Target location if the jump was taken
```

Explanation

- Let's say the instruction `JZ TARGET` is at memory location 1000H.
- The assembler figures out the offset needed to reach `TARGET` from the current location and includes it as a second byte in the instruction.
- If `TARGET` is 10 bytes ahead, the offset will be +10. If `TARGET` is 5 bytes behind, the offset will be -5.
- The range of a relative jump is limited to -128 to +127 bytes from the current instruction.

Key Points

- Relative addressing makes your code more compact, as you don't need full target addresses within the jump instructions.
- Watch out for jump range limits! It can only jump a limited distance forward or backward.

Bit addressing mode

Bit Addressing Mode: Manipulating Individual Bits

- Bit addressing mode provides fine-grained control by allowing you to directly address and manipulate individual bits within:
  - **Internal RAM (bit-addressable area):** Locations 20H to 2FH
  - **Special Function Registers (SFRs):** Many SFRs have individual bits that control specific hardware functions.

Key Points

- Bit addresses range from 00H to 7FH.
- Bit addressing uses a direct addressing approach, where the instruction includes the full bit address.

Instructions for Bit Addressing

- **SETB:** Sets a specified bit to 1.
- **CLR:** Clears a specified bit to 0.
- **CPL:** Complements a specified bit (changes 0 to 1, or 1 to 0).

Examples

1. **Controlling a bit in the P1 SFR (Port 1):**

```
SETB P1.0  ; Set bit 0 of Port 1 (making output pin P1.0 go high)
CLR P1.5   ; Clear bit 5 of Port 1 (making output pin P1.5 go low)
```

2. **Checking the status of a flag bit in the TCON SFR:**

```
JB TF0, OVERFLOW_ROUTINE  ; Jump to 'OVERFLOW_ROUTINE' if the Timer 0
overflow flag (TF0) in the TCON register is set.
```

Why Use Bit Addressing

- **Direct Hardware Control:** Modifying specific bits in SFRs allows you to configure and control various hardware peripherals of the 8051 microcontroller.
- **Efficient use of RAM:** You can pack multiple flags or status indicators into a single byte of internal RAM.

Important Notes

- Not all SFRs are bit-addressable. You'll need to consult the 8051 datasheet for details.
- Remember, bit addresses are different from regular byte addresses.

8051 Instruction Set

- The 8051 has a variable-length instruction set. Instructions range from 1 to 3 bytes long.

- There are 111 core instructions in the 8051.
- These instructions are broadly classified into the following groups:

1. Data Transfer Instructions

- **Moving data between registers:** MOV, XCH, XCHD
- **Moving data between internal RAM and registers:** MOV
- **Moving data to/from external RAM (if present):** MOVX
- **Stack operations:** PUSH, POP
- **Loading immediate values:** MOV
- **Absolute and relative jumps:** LJMP, SJMP
- **Conditional jumps:** JZ, JNZ, JC, JBC, etc.
- **Subroutine calls and returns:** ACALL, LCALL, RET, RETI

2. Arithmetic Instructions

- **Addition:** ADD, ADDC (add with carry)
- **Subtraction:** SUBB (subtract with borrow)
- **Increment/Decrement:** INC, DEC
- **Multiplication:** MUL
- **Division:** DIV
- **Comparison:** CJNE

3. Logical Instructions

- **AND:** ANL
- **OR:** ORL
- **XOR:** XRL

4. Program Branching Instructions

- **Unconditional jumps:** LJMP, SJMP
- **Conditional jumps based on flags and accumulator status:** JZ (Jump if Zero), JNZ (Jump if Not Zero), JC (Jump if Carry), etc.
- **Conditional jumps based on single bits:** JB (Jump if Bit set), JNB (Jump if Bit Not set), JBC (Jump if Bit and Clear)
- **Subroutine calls:** ACALL, LCALL
- **Subroutine returns:** RET, RETI

5. Boolean or Bit Manipulation Instructions

- **Setting bits:** SETB
- **Clearing bits:** CLR
- **Complementing (inverting) bits:** CPL
- **Logical operations on bits:** ANL, ORL, XRL,
- **Conditional jumps based on individual bit states:** JB, JNB, JBC

The following nomenclatures for register, data, address and variables are used while write instructions.

- A: Accumulator
- B: "B" register
- C: Carry bit
- Rn: Register R0 - R7 of the currently selected register bank
- Direct: 8-bit internal direct address for data. The data could be in lower 128bytes of RAM (00 - 7FH) or it could be in the special function register (80 - FFH).
- @Ri: 8-bit external or internal RAM address available in register R0 or R1. This is used for indirect addressing mode.
- #data8: Immediate 8-bit data available in the instruction.
- #data16: Immediate 16-bit data available in the instruction.
- Addr11: 11-bit destination address for short absolute jump. Used by instructions AJMP & ACALL. Jump range is 2 kbyte (one page).
- Addr16: 16-bit destination address for long call or long jump.
- Rel: 2's complement 8-bit offset (one - byte) used for short jump (SJMP) and all conditional jumps.
- bit: Directly addressed bit in internal RAM or SFR

Data Transfer Instructions

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.

These instructions move data between various registers, internal RAM, external RAM, and I/O ports of the 8051. Here's a breakdown of the key types:

1. Register-to-Register Transfers

- **MOV instruction:** The most versatile data transfer instruction.
- **Examples:**
  - `MOV A, R5` - Copies the contents of register R5 into the accumulator.
  - `MOV R2, #45H` - Loads the immediate value 45H into register R2.
  - `MOV P1, A` - Copies the accumulator's contents to port P1 (output).

2. Direct Addressing

- **MOV instruction with 'direct' addressing mode:** Accesses internal RAM or Special Function Registers (SFRs).
- **Examples:**
  - `MOV 50H, A` - Stores the value in the accumulator to internal RAM location 50H.
  - `MOV ACC, 55H` - Loads the byte from internal RAM location 55H into the accumulator.
  - `MOV TMOD, #01H` - Sets Timer 0 into mode 1.

3. Indirect Addressing

- **MOV instruction using registers as pointers:** The register holds the address of the data.
- **Examples:**
  - `MOV A, @R0` - Copies the byte pointed to by register R0 into the accumulator.
  - `MOV @R1, 33H` - Stores the value 33H at the address pointed to by R1.

4. External Memory Transfers

- **MOVX instruction:** Used to access external RAM.
- **Examples:**
  - `MOVX A, @DPTR` – Copies a byte from external RAM (address in DPTR) into the accumulator.
  - `MOVX @DPTR, A` – Copies the contents of the accumulator into external RAM (address in DPTR).

5. Special Data Transfers

- **PUSH instruction:** Pushes data onto the stack (internal RAM).
- **POP instruction:** Pops data off the stack.

Example: Data Sorting Routine

Consider a simple routine to sort three numbers stored at internal RAM locations 40H, 41H, and 42H:

```
COMPARE: MOV A, 40H    ; Load the first number
         MOVC A, @A+DPTR ; Load the second number (assuming DPTR points to RAM)
         JC  SWAP        ; Jump to SWAP if the first is greater than the second

         MOV A, 41H     ; Load the second number
         MOVC A, @A+DPTR ; Load the third number
         JC  SWAP        ; Jump to SWAP if the second is greater than the third
         ; ... (rest of your code)

SWAP:    ; ... (Code to swap values)
```

Mnemonic	Description	Byte	Cycle
MOV A,Rn	Moves the register to the accumulator	1	1
MOV A,direct	Moves the direct byte to the accumulator	2	2
MOV A,@Ri	Moves the indirect RAM to the accumulator	1	2
MOV A,#data	Moves the immediate data to the accumulator	2	2
MOV Rn,A	Moves the accumulator to the register	1	2
MOV Rn,direct	Moves the direct byte to the register	2	4
MOV Rn,#data	Moves the immediate data to the register	2	2
MOV direct,A	Moves the accumulator to the direct byte	2	3
MOV direct,Rn	Moves the register to the direct byte	2	3

Mnemonic	Description	Byte	Cycle
MOV direct,direct	Moves the direct byte to the direct byte	3	4
MOV direct,@Ri	Moves the indirect RAM to the direct byte	2	4
MOV direct,#data	Moves the immediate data to the direct byte	3	3
MOV @Ri,A	Moves the accumulator to the indirect RAM	1	3
MOV @Ri,direct	Moves the direct byte to the indirect RAM	2	5
MOV @Ri,#data	Moves the immediate data to the indirect RAM	2	3
MOV DPTR,#data	Moves a 16-bit data to the data pointer	3	3
MOVC A,@A+DPTR	Moves the code byte relative to the DPTR to the accumulator (address=A+DPTR)	1	3
MOVC A,@A+PC	Moves the code byte relative to the PC to the accumulator (address=A+PC)	1	3
MOVX A,@Ri	Moves the external RAM (8-bit address) to the accumulator	1	3-10
MOVX A,@DPTR	Moves the external RAM (16-bit address) to the accumulator	1	3-10
MOVX @Ri,A	Moves the accumulator to the external RAM (8-bit address)	1	4-11
MOVX @DPTR,A	Moves the accumulator to the external RAM (16-bit address)	1	4-11
PUSH direct	Pushes the direct byte onto the stack	2	4
POP direct	Pops the direct byte from the stack/td>	2	3
XCH A,Rn	Exchanges the register with the accumulator	1	2
XCH A,direct	Exchanges the direct byte with the accumulator	2	3
XCH A,@Ri	Exchanges the indirect RAM with the accumulator	1	3
XCHD A,@Ri	Exchanges the low-order nibble indirect RAM with the accumulator	1	3

Compare MOV, MOVX and MOVC instruction using one example of each.

MOV: The MOV instruction moves data bytes between the two specified operands. The byte specified by the second operand is copied to the location specified by the first operand. The source data byte is not affected.

Examples:

MOV A,Rn	**Moves the register to the accumulator**
MOV A,direct	Moves the direct byte to the accumulator
MOV A,@Ri	Moves the indirect RAM to the accumulator
MOV A,#data	Moves the immediate data to the accumulator
MOV Rn,A	Moves the accumulator to the register
MOV Rn,direct	Moves the direct byte to the register
MOV Rn,#data	Moves the immediate data to the register
MOV direct,A	Moves the accumulator to the direct byte
MOV direct,Rn	Moves the register to the direct byte
MOV direct,direct	Moves the direct byte to the direct byte
MOV direct,@Ri	Moves the indirect RAM to the direct byte
MOV direct,#data	Moves the immediate data to the direct byte
MOV @Ri,A	Moves the accumulator to the indirect RAM
MOV @Ri, direct	Moves the direct byte to the indirect RAM
MOV @Ri,#data	Moves the immediate data to the indirect RAM
MOV DPTR,#data	Moves a 16-bit data to the data pointer

MOVX: The MOVX instruction transfers data between the accumulator and external data memory. External memory may be addressed via 16-bits in the DPTR register or via 8-bits in the R0 or R1 registers. When using 8-bit addressing, Port 2 must contain the high-order byte of the address.

Examples:

MOVX A, @Ri	**Moves the external RAM (8-bit address) to the accumulator**
MOVX A, @DPTR	Moves the external RAM (16-bit address) to the accumulator
MOVX @Ri, A	Moves the accumulator to the external RAM (8-bit address)
MOVX @DPTR, A	Moves the accumulator to the external RAM (16-bit address)

MOVC: The MOVC instruction moves a byte from the code or program memory to the accumulator.

Examples:

MOVC A,@A+DPTR	**Moves the code byte relative to the DPTR to the accumulator (address=A+DPTR)**
MOVC A,@A+PC	Moves the code byte relative to the PC to the accumulator (address=A+PC)

Arithmetic Instructions

Arithmetic instructions perform several basic operations such as addition, subtraction, division, multiplication etc. After execution, the result is stored in the first operand. For example: ADD A,R1 - The result of addition (A+R1) will be stored in the accumulator.

Core Arithmetic Instructions

- **ADD (Addition):** Adds the value of a source operand to the accumulator (A).
  - Example: `ADD A, R2` (Adds contents of register R2 to the accumulator)
- **ADDC (Addition with Carry):** Adds a source operand plus the previous carry flag (CF) to the accumulator (A)
  - Example: `ADDC A, #30H` (Adds 30H and the carry flag to the accumulator)
- **SUBB (Subtraction with Borrow):** Subtracts the value of a source operand and the carry flag (CF) from the accumulator (A).
  - Example: `SUBB A, @R0` (Subtracts the value pointed to by R0 and the carry flag from the accumulator)
- **INC (Increment):** Increments the value of a register or direct memory location by 1.
  - Example: `INC R5` (Increments the value of register R5)
- **DEC (Decrement):** Decrements the value of a register or direct memory location by 1.
  - Example: `DEC 50H` (Decrements the value at internal RAM location 50H)
- **MUL (Multiplication):** Multiplies the accumulator (A) with register B. Result stored in both the accumulator and register B.
  - Example: `MUL AB`
- **DIV (Division):** Divides the accumulator (A) by register B. Quotient is stored in the accumulator and the remainder in register B.
  - Example: `DIV AB`

Important Notes

- Arithmetic operations affect the following flags:
  - C (Carry Flag)
  - AC (Auxiliary Carry Flag)
  - OV (Overflow Flag)
- **DAA (Decimal Adjust Accumulator):** A special instruction used after addition to adjust the result if you're working with BCD (Binary Coded Decimal) numbers.

Example: A Simple Calculation

```
MOV A, #25H   ; Load 25 into the accumulator
ADD A, #10H   ; Add 10 to the accumulator (result 35 in Accumulator)
INC A         ; Increment the accumulator (result 36 in Accumulator)
DEC R0        ; Decrement register R0
```

Mnemonic	Description	Byte	Cycle
ADD A,Rn	Adds the register to the accumulator	1	1

Mnemonic	Description	Byte	Cycle
ADD A,direct	Adds the direct byte to the accumulator	2	2
ADD A,@Ri	Adds the indirect RAM to the accumulator	1	2
ADD A,#data	Adds the immediate data to the accumulator	2	2
ADDC A,Rn	Adds the register to the accumulator with a carry flag	1	1
ADDC A,direct	Adds the direct byte to the accumulator with a carry flag	2	2
ADDC A,@Ri	Adds the indirect RAM to the accumulator with a carry flag	1	2
ADDC A,#data	Adds the immediate data to the accumulator with a carry flag	2	2
SUBB A,Rn	Subtracts the register from the accumulator with a borrow	1	1
SUBB A,direct	Subtracts the direct byte from the accumulator with a borrow	2	2
SUBB A,@Ri	Subtracts the indirect RAM from the accumulator with a borrow	1	2
SUBB A,#data	Subtracts the immediate data from the accumulator with a borrow	2	2
INC A	Increments the accumulator by 1	1	1
INC Rn	Increments the register by 1	1	2
INC Rx	Increments the direct byte by 1	2	3
INC @Ri	Increments the indirect RAM by 1	1	3
DEC A	Decrements the accumulator by 1	1	1
DEC Rn	Decrements the register by 1	1	1
DEC Rx	Decrements the direct byte by 1	1	2
DEC @Ri	Decrements the indirect RAM by 1	2	3
INC DPTR	Increments the Data Pointer by 1	1	3
MUL AB	Multiplies A and B	1	5
DIV AB	Divides A by B	1	5
DA A	Decimal adjustment of the accumulator according to BCD code	1	1

Logical Instructions

What are Logical Instructions?

Logical instructions perform logic operations on individual bits within registers or between a register and an immediate value. These include:

- **AND:** Bitwise logical AND operation.
- **OR:** Bitwise logical OR operation.
- **XOR:** Bitwise logical XOR (Exclusive OR) operation.
- **NOT:** Bitwise inversion (Complement)
- **Rotate/Shift:** Move bits within a register or memory location

How they work

Each bit of the first operand is compared to the corresponding bit of the second operand according to the following truth tables:

Input A	Input B	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Key Takeaways:

- **AND:** Outputs 1 only when both inputs are 1.
- **OR:** Outputs 1 when at least one input is 1.
- **XOR:** Outputs 1 when the inputs are different.

Examples

1. **Masking Bits (AND)**

```
MOV A, #53H  ; A = 0101 0011
ANL A, #0FH  ; AND with 0000 1111 (mask to keep only the lower 4 bits)
; A now holds 0000 0011
```

2. **Setting Bits (OR)**

```
MOV A, #7BH  ; A = 0111 1011
ORL A, #80H  ; OR with 1000 0000 (set the most significant bit)
; A now holds 1111 1011
```

3. **Toggling Bits (XOR)**

```
MOV A, #96H  ; A = 1001 0110
XOR A, #05H  ; XOR with 0000 0101 (toggle specific bits)
; A now holds 1001 0011
```

4. **Rotating Bits**

```
MOV A, #0AH  ; A = 0000 1010
RL A         ; Rotate left through carry (assume Carry flag is 0)
; A now holds 0001 0100
```

Common Uses

- **Testing if specific bits are set or clear.**
- **Manipulating flags (e.g., setting the Carry flag).**
- **Isolating sections of data within a byte.**
- **Implementing simple cryptographic functions.**

Mnemonic	Description	Byte	Cycle
ANL A,Rn	AND register to accumulator	1	1
ANL A,direct	AND direct byte to accumulator	2	2
ANL A,@Ri	AND indirect RAM to accumulator	1	2
ANL A,#data	AND immediate data to accumulator	2	2
ANL direct,A	AND accumulator to direct byte	2	3
ANL direct,#data	AND immediae data to direct register	3	4
ORL A,Rn	OR register to accumulator	1	1
ORL A,direct	OR direct byte to accumulator	2	2
ORL A,@Ri	OR indirect RAM to accumulator	1	2
ORL direct,A	OR accumulator to direct byte	2	3
ORL direct,#data	OR immediate data to direct byte	3	4
XRL A,Rn	Exclusive OR register to accumulator	1	1
XRL A,direct	Exclusive OR direct byte to accumulator	2	2
XRL A,@Ri	Exclusive OR indirect RAM to accumulator	1	2
XRL A,#data	Exclusive OR immediate data to accumulator	2	2
XRL direct,A	Exclusive OR accumulator to direct byte	2	3
XORL direct,#data	Exclusive OR immediate data to direct byte	3	4
CLR A	Clears the accumulator	1	1
CPL A	Complements the accumulator (1=0, 0=1)	1	1
SWAP A	Swaps nibbles within the accumulator	1	1
RL A	Rotates bits in the accumulator left	1	1
RLC A	Rotates bits in the accumulator left through carry	1	1

Mnemonic	Description	Byte	Cycle
RR A	Rotates bits in the accumulator right	1	1
RRC A	Rotates bits in the accumulator right through carry	1	1

Program Branching Instructions

Program Branching instructions, often also called jump instructions, allow you to alter the normal sequential flow of program execution. They cause the Program Counter (PC) to jump to a different memory location, breaking the usual 'execute the next instruction' pattern.

Types of Program Branching Instructions

1. **Unconditional Branching:** upon their execution a jump to a new location from where the program continues execution is executed.
    - **LJMP (Long Jump):** Jumps to the specified 16-bit address.
        - *Example:* `LJMP 2050H` (jumps to memory location 2050H)
2. **Conditional Branching:** a jump to a new program location is executed only if a specified condition is met. Otherwise, the program normally proceeds with the next instruction.
    - **These depend on the status of flags (Carry, Parity, Overflow, etc.) set by previous operations**
    - **Examples:**
        - `JC LABEL` (Jump if Carry flag is set)
        - `JNC LABEL` (Jump if Carry flag is not set)
        - `JZ LABEL` (Jump if Zero flag is set)
        - `JNZ LABEL` (Jump if Zero flag is not set)
3. **Short Jump (Relative Jump)**
    - **SJMP:** Jumps to an address within a limited range relative (+127 or -128 bytes) to the current instruction.
    - *Example:* `SJMP LOOP_START` (jumps to a label relatively nearby)

Example: Conditional Loop

```
MOV R0, #10    ; Initialize a counter
LOOP:
   ; ... some code here ...
   DJNZ R0, LOOP  ; Decrement and jump if not zero
```

Explanation

1. The counter register R0 is loaded with 10.
2. The code in the `LOOP` section executes.
3. `DJNZ R0, LOOP`
    - Decrements R0 by one.
    - If the Zero flag is NOT set (R0 is not zero), jumps back to the `LOOP` label.

Key Points

- Branching instructions are core to creating loops, decision structures (if-else), and subroutines within programs.
- The destination of a jump can be an explicit address (e.g., `LJMP 2050H`) or often a label that the assembler translates to the correct address.

Mnemonic	Description	Byte	Cycle
ACALL addr11	Absolute subroutine call	2	6
LCALL addr16	Long subroutine call	3	6
RET	Returns from subroutine	1	4
RETI	Returns from interrupt subroutine	1	4
AJMP addr11	Absolute jump	2	3
LJMP addr16	Long jump	3	4
SJMP rel	Short jump (from –128 to +127 locations relative to the following instruction)	2	3
JC rel	Jump if carry flag is set. Short jump.	2	3
JNC rel	Jump if carry flag is not set. Short jump.	2	3
JB bit,rel	Jump if direct bit is set. Short jump.	3	4
JBC bit,rel	Jump if direct bit is set and clears bit. Short jump.	3	4
JMP @A+DPTR	Jump indirect relative to the DPTR	1	2
JZ rel	Jump if the accumulator is zero. Short jump.	2	3
JNZ rel	Jump if the accumulator is not zero. Short jump.	2	3
CJNE A,direct,rel	Compares direct byte to the accumulator and jumps if not equal. Short jump.	3	4
CJNE A,#data,rel	Compares immediate data to the accumulator and jumps if not equal. Short jump.	3	4
CJNE Rn,#data,rel	Compares immediate data to the register and jumps if not equal. Short jump.	3	4
CJNE @Ri,#data,rel	Compares immediate data to indirect register and jumps if not equal. Short jump.	3	4
DJNZ Rn,rel	Decrements register and jumps if not 0. Short jump.	2	3
DJNZ Rx,rel	Decrements direct byte and jump if not 0. Short jump.	3	4
NOP	No operation	1	1

Boolean or Bit-manipulation Instructions

Similar to logic instructions, bit-oriented instructions perform logic operations. The difference is that these are performed upon single bits.

Core Boolean/Bit Instructions

- **ANL (Logical AND):** Performs a bitwise AND operation between a source operand and the accumulator.
  - Example: `ANL A, 52H` (ANDs the accumulator with the value 52H)
  - Example: `ANL P1.2, C` (ANDs port bit P1.2 with the Carry flag)
- **ORL (Logical OR):** Performs a bitwise OR operation between a source operand and the accumulator.
  - Example: `ORL A, #03H` (ORs the accumulator with the value 03H)
  - Example: `ORL 25H, C` (ORs the memory location 25H with the Carry flag)
- **XRL (Logical XOR):** Performs a bitwise XOR operation between a source operand and the accumulator.
  - Example: `XRL A, R3` (XORs the accumulator with register R3)
- **CLR (Clear):** Sets a specified register or bit to 0.
  - Example: `CLR A` (Clears the accumulator)
  - Example: `CLR P2.0` (Sets the bit P2.0 to zero)
- **SETB (Set):** Sets a specified bit to 1.
  - Example: `SETB 30H` (Sets a bit in the bit-addressable area of RAM)
  - Example: `SETB P3.7` (Sets the bit P3.7 to one)
- **CPL (Complement):** Inverts the value of a particular bit (changes 0 to 1 and 1 to 0).
  - Example: `CPL A` (Inverts all bits in the accumulator)
  - Example: `CPL P1.5` (Inverts the bit P1.5)

Bit-Based Conditional Jumps

- **JB (Jump if Bit set):** Jumps if the specified bit is '1'.
  - Example: `JB P2.3, LABEL`
- **JNB (Jump if Bit not set):** Jumps if the specified bit is '0'.
- **JBC (Jump if Bit set and Clear):** Jumps if the specified bit is '1', then clears the bit.

Example: Controlling an Output Pin

```
MOV P1, #00H    ; Initialize Port 1 output as 0

; ... some other code ...

SETB P1.5       ; Turn ON the output connected to P1.5

; ... more code ...

CLR P1.5        ; Turn OFF the output connected to P1.5
```

Important Notes

- Bit manipulation instructions work with the bit-addressable areas of internal RAM (20H-2FH) and specific bits within SFRs.
- Bit operations often control hardware functionality, flags, and status bits.

Mnemonic	Description	Byte	Cycle
CLR C	Clears the carry flag	1	1
CLR bit	Clears the direct bit	2	3
SETB C	Sets the carry flag	1	1
SETB bit	Sets the direct bit	2	3
CPL C	Complements the carry flag	1	1
CPL bit	Complements the direct bit	2	3
ANL C,bit	AND direct bit to the carry flag	2	2
ANL C,/bit	AND complements of direct bit to the carry flag	2	2
ORL C,bit	OR direct bit to the carry flag	2	2
ORL C,/bit	OR complements of direct bit to the carry flag	2	2
MOV C,bit	Moves the direct bit to the carry flag	2	2
MOV bit,C	Moves the carry flag to the direct bit	2	3

Machine Control

These instructions don't primarily target data manipulation, but instead, they control the processor's behavior.

Key Machine Control Instructions

- **NOP (No Operation):** Does nothing for one machine cycle (consumes some time). Uses:
  - Introducing small delays
  - Aligning instructions in memory
- **AJMP (Absolute Jump):** Similar to LJMP, but only allows a 2-byte address, limiting jump range. It's designed for jumping within code segments.
- **ACALL (Absolute Call):** Similar to LCALL, but uses a 2-byte address for calling subroutines within code segments.
- **RET (Return):** Returns from a subroutine called using ACALL or LCALL.
- **RETI (Return from Interrupt):** Returns from an interrupt service routine.
- **SJMP (Short Jump):** Allows relative jumps within a range of -128 to +127 bytes from the current instruction. Used for shorter jumps within code.
- **EI (Enable Interrupt):** Globally enables interrupts.
- **DI (Disable Interrupt):** Globally disables interrupts.

- **SIM (Set Interrupt Mask):** Used for finer-grained control of specific interrupt sources (setting the interrupt mask) and sending serial data out.
- **RIM (Read Interrupt Mask):** Used to read the status of the interrupt mask and read serial data in.
- **HLT (Halt):** Places the microcontroller in a low-power halt state. An interrupt or reset is needed to resume execution.

Note: SIM and RIM provide more specialized functionality than simple enable and disable.

Examples

1. **Generating a small delay:**

```
MOV R0, #50   ; Load counter value
DELAY: NOP    ; One machine cycle delay
       DJNZ R0, DELAY ; Decrement and jump if not zero
```

2. **Enabling interrupts:**

```
EI   ; Enable interrupts globally
```

3. **Handling an interrupt routine:**

```
ORG 0013H  ; Interrupt vector for timer 1
; Timer 1 Interrupt Service Routine (ISR)
; ... Code to handle the timer interrupt
RETI       ; Return from interrupt
```

Important Considerations

- Incorrect use of machine control instructions can lead to unexpected behavior of your microcontroller.
- Interrupts require careful planning to avoid conflicts and ensure proper program execution.

Mnemonic	Description	Bytes	Cycles
NOP	No Operation	1	1
EJMP addr16	Extended Jump (not frequently used)	3	2
AJMP addr11	Absolute Jump (within code segment)	2	2
ACALL addr11	Absolute Call (within code segment)	2	2
RET	Return from subroutine	1	2
RETI	Return from interrupt	1	2
SJMP rel8	Short Relative Jump (-128 to +127 range)	2	2
EI	Enable Interrupts (globally)	1	1
DI	Disable Interrupts (globally)	1	1
SIM	Set Interrupt Mask (specific interrupt control)	1	1

Mnemonic	Description	Bytes	Cycles
RIM	Read Interrupt Mask (specific interrupt status)	1	1
HLT	Halt (low-power mode)	1	1

Assembly Language Programming Examples

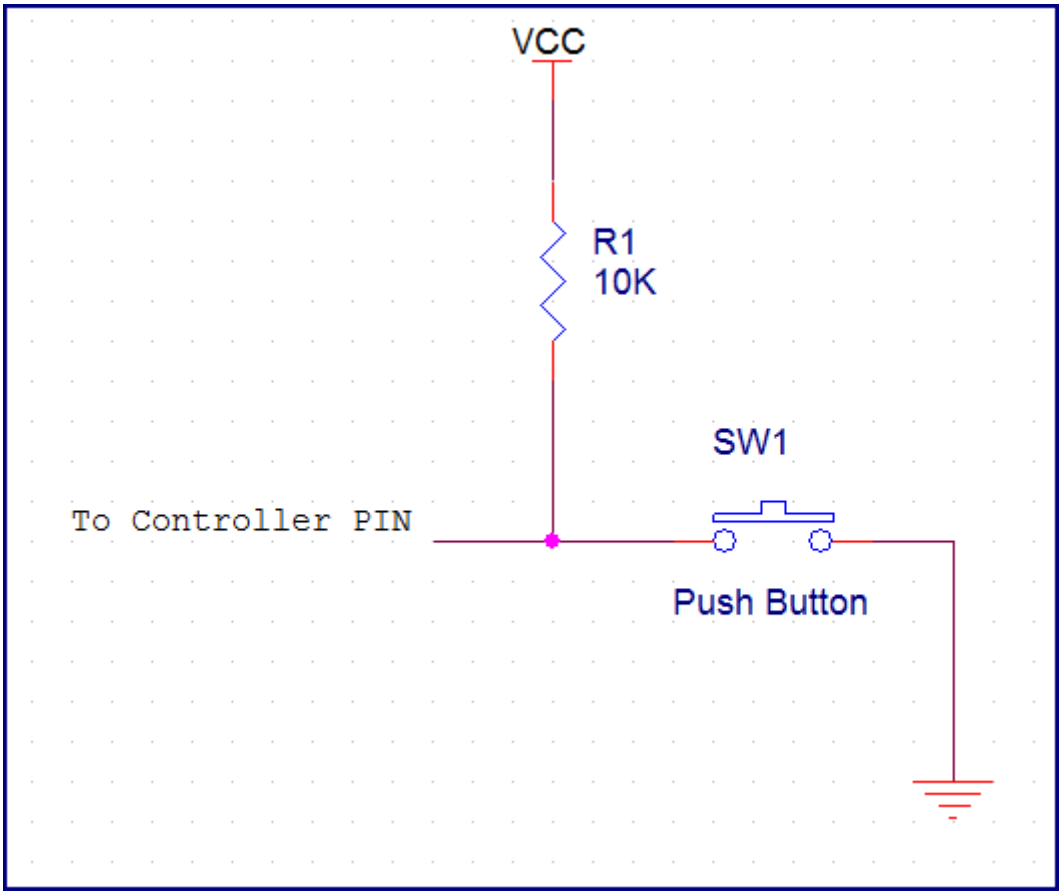
Refer Dedicated Notes for Programming Examples

Unit V: Interfacing & Applications of Microcontroller

Input Devices

Push Button Switches

Interface Input Devices with 8051 micro-controller: Switch, Push-button & DIP.



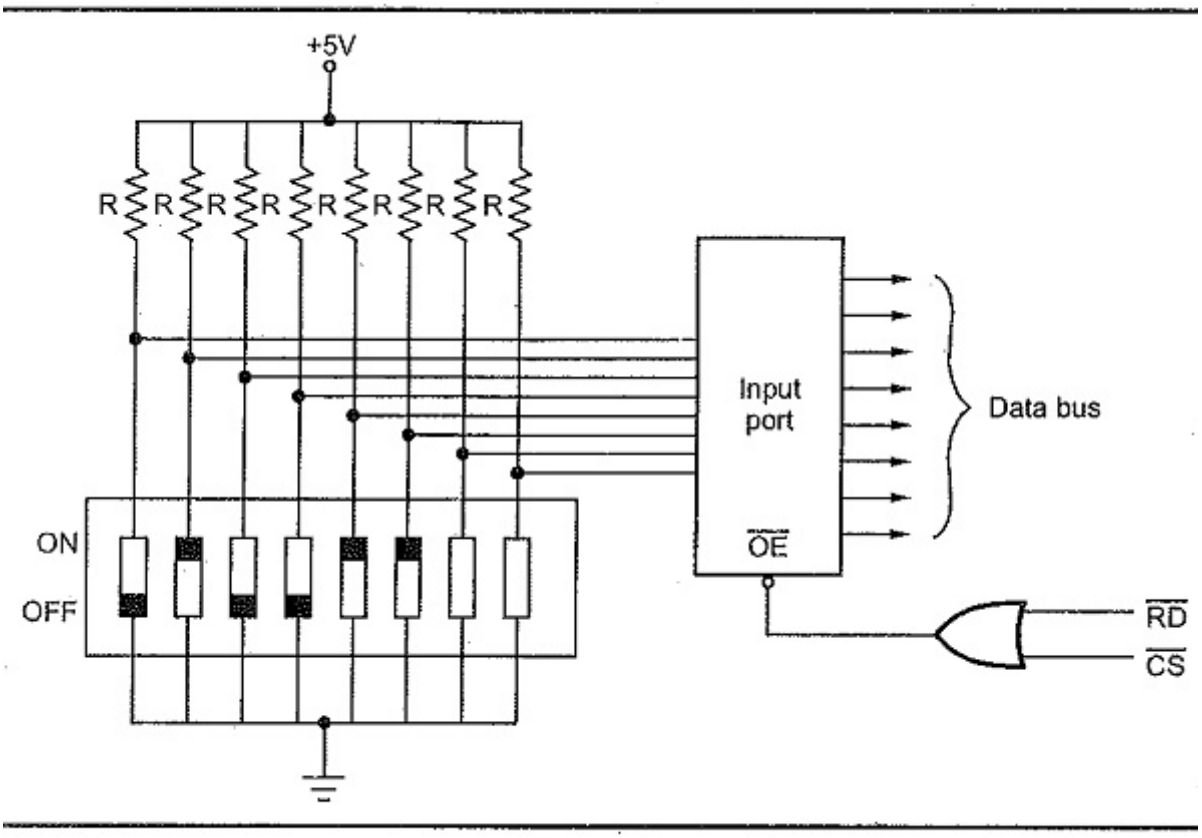
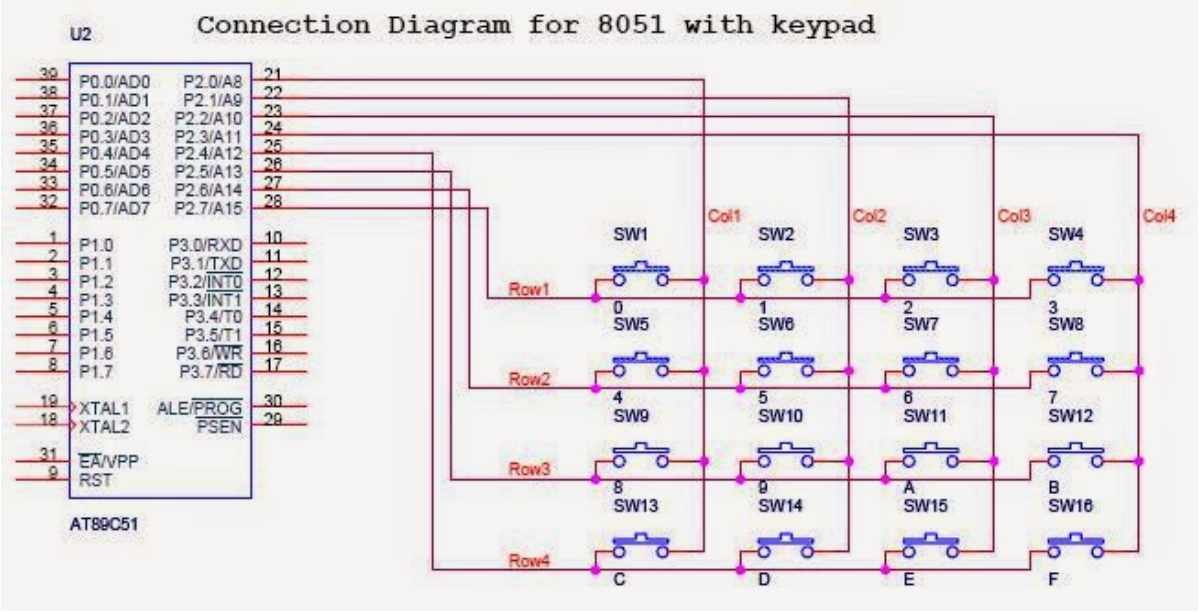


Fig. 15.23 (a) Interfacing DIP switches



LM35 Temperature Sensor

Q5c: Interface LM 35 with 8051 microcontroller and explain block diagram of temperature controller.

Interfacing LM35 with 8051

1. **Connections:**
   - Connect the Vout pin of the LM35 to one of the 8051's analog input channels (ADC).
   - Connect the VSS pin of the LM35 to ground.
   - Connect the VS pin of the LM35 to the power supply (+5V).
2. **ADC Configuration:**

- Select the ADC channel connected to the LM35.
- Set the ADC's resolution (e.g., 10-bit).
- In your program, initiate the ADC conversion process.

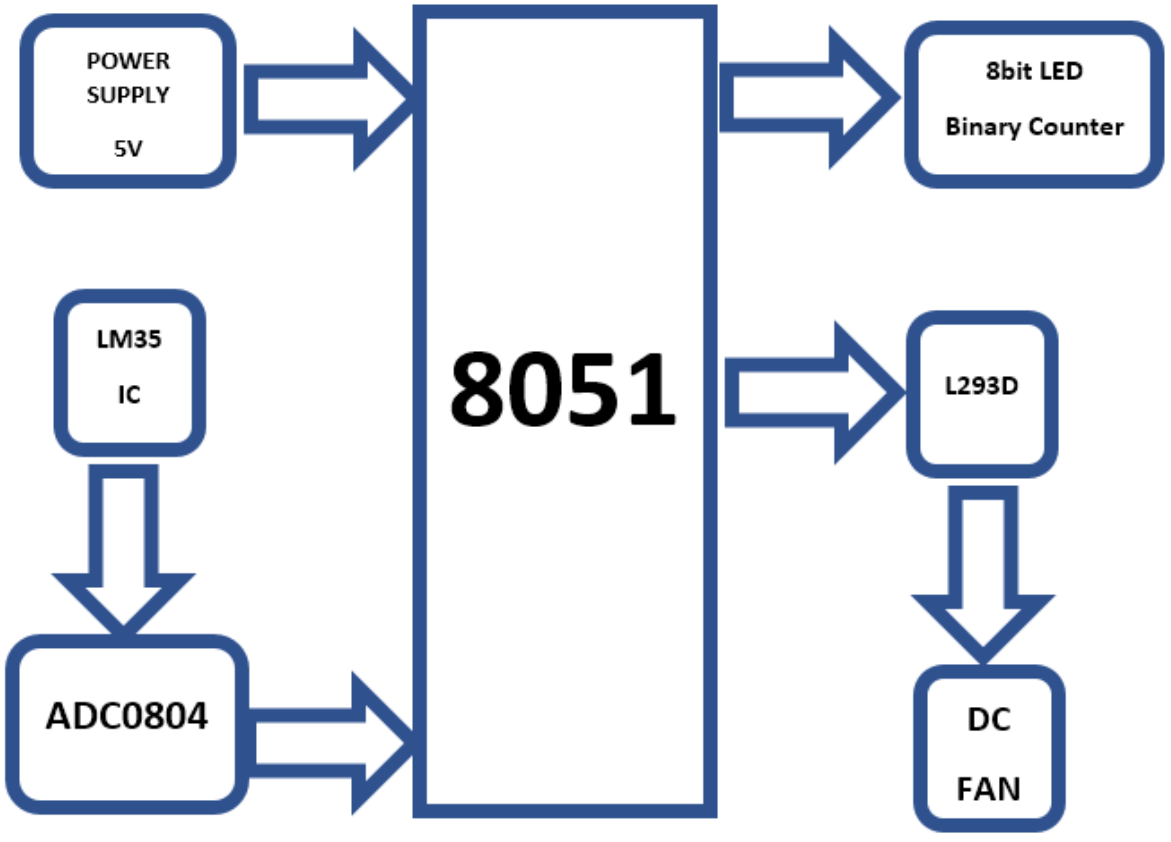
3. **Reading and Conversion:**
   - After the conversion completes, read the digital value from the ADC data register.
   - The LM35 outputs 10mV per degree Celsius. To convert the digital ADC reading to temperature:
     - Scale the ADC value based on its resolution and reference voltage.
     - Divide by 10 to get the temperature in Celsius.

Code Snippet (Illustrative)

```
; ... (ADC Initialization)
START_CONVERSION:
    SETB ADC_START_BIT  ; Trigger ADC conversion
    JBC ADC_BUSY_BIT, START_CONVERSION ; Wait for conversion to complete

    MOV A, ADC_DATA_REG   ; Read ADC result
    ; ... (Calculate temperature from ADC value)
```

Block Diagram: Temperature Controller



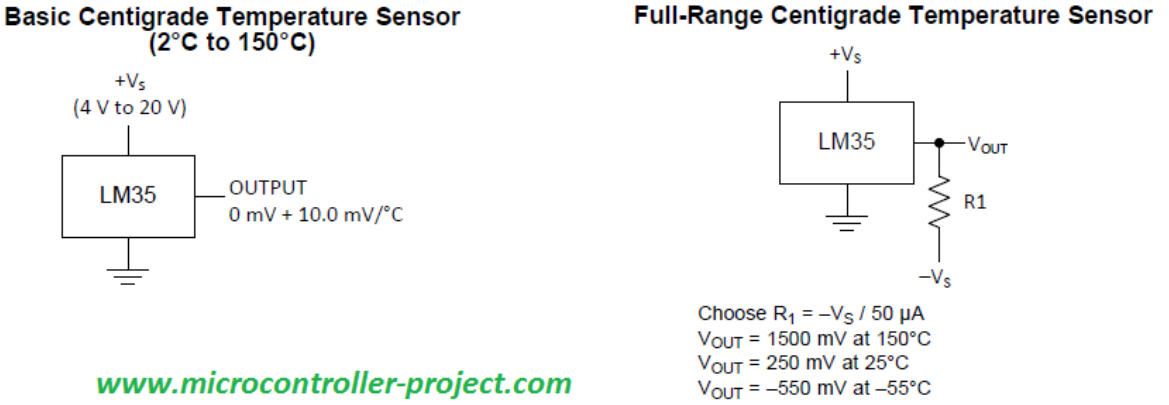
- **Temperature Sensor (LM35):** Measures the ambient temperature and generates an analog voltage proportional to the temperature.
- **ADC (Analog-to-Digital Converter):** Part of the 8051 microcontroller, it converts the analog voltage from the LM35 into a digital value.
- **8051 Microcontroller:**

    - Reads the temperature from the ADC.
    - Compares the measured temperature with a desired setpoint.
    - Generates control signals based on the comparison.
- **Control Output (Relay, etc.):** Controls a device (e.g., heater, fan) to regulate the temperature. Could be a simple on/off relay or more complex control like PWM.
- **Display (Optional):** A display (LCD, 7-segment) to show the current temperature or setpoint.

How it Works

1. The LM35 senses temperature and sends the analog signal to the ADC.
2. The ADC converts the analog signal to a digital value.
3. The 8051 microcontroller reads this value, calculates the temperature, and compares it to the desired setpoint.
4. If the temperature deviates from the setpoint, the 8051 sends control signals to turn a heater or cooler on or off, aiming to bring the temperature back to the setpoint.

Explain temperature sensor LM35 in brief. (3)



Output Devices

LEDs

Q5b: Interface 8 leds with 8051 microcontroller and write a program to turn on and off.

Hardware Setup

1. **8 LEDs:** Choose standard LEDs considering the current requirements of the 8051's I/O ports.
2. **Current-Limiting Resistors:** Calculate the appropriate resistor values for your specific LEDs to prevent damage (search for an online "LED resistor calculator" if needed).
3. **8051 Microcontroller:** We'll assume you have an 8051 development board with an I/O port (e.g., Port 1).
4. **Connections:**
    - Connect one leg of each LED to a separate pin on Port 1 of the 8051 (P1.0 to P1.7).
    - Connect the other leg of each LED through a current-limiting resistor to ground.

Programming (Assembly)

Here's a simple 8051 assembly program to repeatedly turn the LEDs on and then off with a delay:

```
ORG 0000H  ; Program starts at address 0000H

MAIN_LOOP:
    MOV A, #FFH   ; Set all port pins high (LEDs on)
    MOV P1, A     ; Send data to Port 1
    CALL DELAY    ; Call a delay subroutine

    MOV A, #00H   ; Set all port pins low (LEDs off)
    MOV P1, A
    CALL DELAY

    SJMP MAIN_LOOP  ; Jump back to the beginning

DELAY:  ; Simple delay subroutine - adjust for desired time
    MOV R0, #200   ; Adjust these values for timing
    MOV R1, #150
DLY_LOOP: DJNZ R1, DLY_LOOP
          DJNZ R0, DLY_LOOP
          RET  ; Return from subroutine

END
```

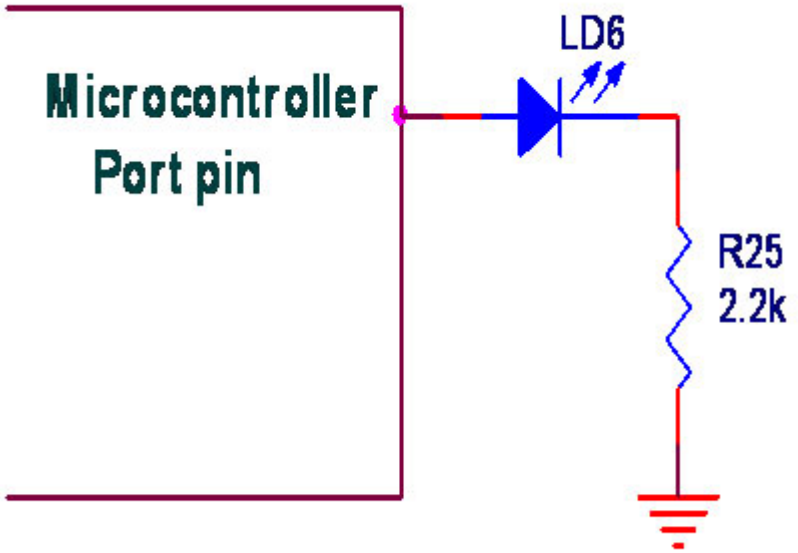
Explanation

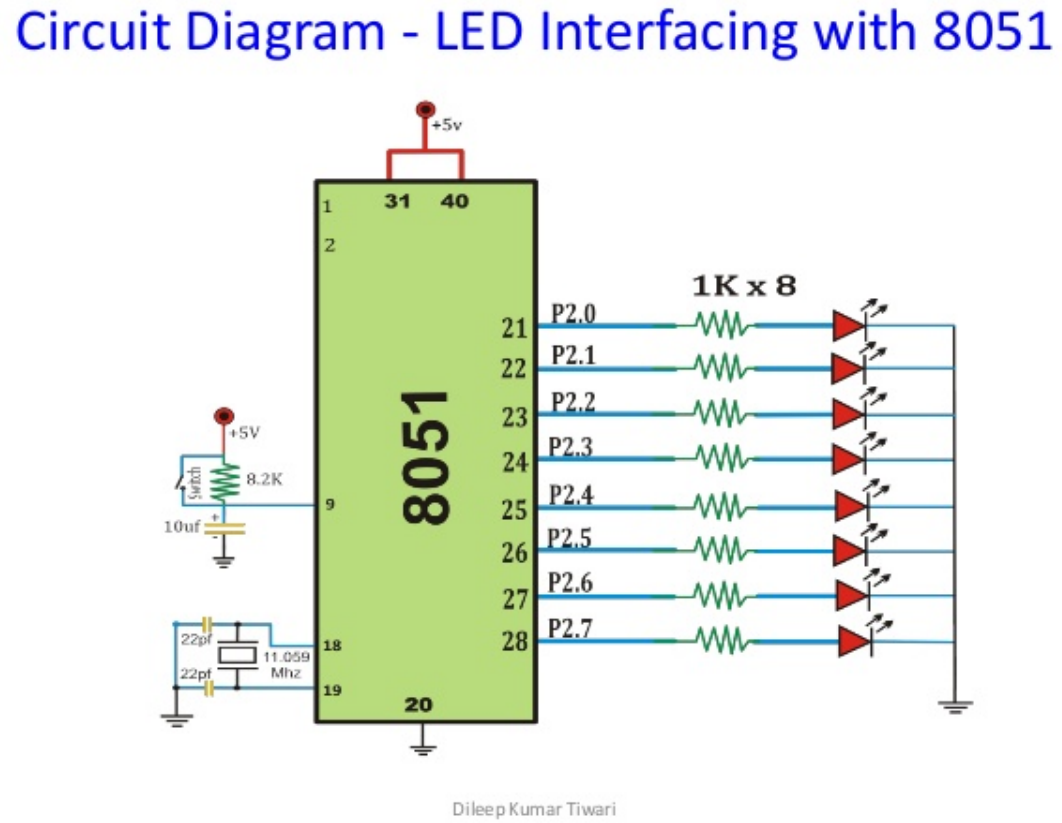
1. **MAIN_LOOP:**
    - `MOV A, #FFH`: Loads the accumulator with FFH (all bits 1), which will turn all LEDs on.
    - `MOV P1, A`: Sends this value to Port 1.
    - `CALL DELAY`: Calls a subroutine to create a delay.
    - `MOV A, #00H`: Loads the accumulator with 00H (all bits 0), which will turn all LEDs off.
    - `SJMP MAIN_LOOP`: Creates an infinite loop.
2. **DELAY Subroutine:**
    - Provides a simple software delay using nested loops. You might want a more precise timer-based delay in a real application.

Key Points:

- **Port Choice:** I used Port 1(P1); adapt the code if you connect the LEDs to a different port.
- **LED Polarity:** If your LEDs light up in the opposite manner, reverse the logic (use 00H to turn them on and FFH to turn them off)
- **Delay Adjustment:** Modify the values in the DELAY subroutine to change the on and off duration.

Interface Output devices with 8051 microcontroller: LED.





7-Segment Displays

Q5b: Interface 7 segment with 8051 microcontroller.

Assumptions

- **Common Anode Display:** The segments have a common positive connection. We'll control them by sinking current (connecting to ground) through the 8051's pins.
    - Adapt the segment patterns if you have a Common Cathode display.
- **Single Digit:** We'll interface a single digit display. This can be extended for multiple digits using multiplexing techniques.

- **Connections:** We'll assume you'll connect the 7-segment pins (a through g) to a port of the 8051 (e.g., Port 1).

Hardware Setup

1. **7-Segment Display:** Choose a common anode 7-segment LED display.
2. **Current-Limiting Resistors:** Calculate and use resistors in series with each segment LED to prevent damage. Search for a "LED resistor calculator" to find the right values.
3. **Connections:**
    - Connect the anodes of all segments (a through g) to the corresponding pins of Port 1 (P1.0 through P1.6) of the 8051.
    - Connect the common anode pin to the power supply (+5V).
    - Connect each segment's cathode through the resistor to ground.

Lookup Table

Create a lookup table in your program memory that maps the digit you want to display (0-9) to the corresponding segment patterns:

```
SEGMENT_PATTERNS:
    DB 0C0H ; Pattern for 0 (abcdefg)
    DB 0F9H ; Pattern for 1
    DB 0A4H ; Pattern for 2
     ; ... Add patterns for 3-9
```

Note: For a common anode display, '1' means the segment should be ON, so it's connected to ground.

Assembly Code Example

```
ORG 0000H

; Assume display is connected to Port 1
DISPLAY_PORT EQU P1

; ... (Segment patterns lookup table from above)

MAIN_LOOP:
    MOV R0, #2   ; Example: Load the digit 2 to display
    MOV A, @R0   ; Point to the segment pattern in the table
    ADD A, SEGMENT_PATTERNS  ; Calculate the address
    MOVC A, @A+DPTR          ; Fetch the segment pattern
    MOV DISPLAY_PORT, A      ;  Send the pattern to the display port

    ; ... (Add display refreshing if you want to multiplex multiple digits)
END
```

Explanation

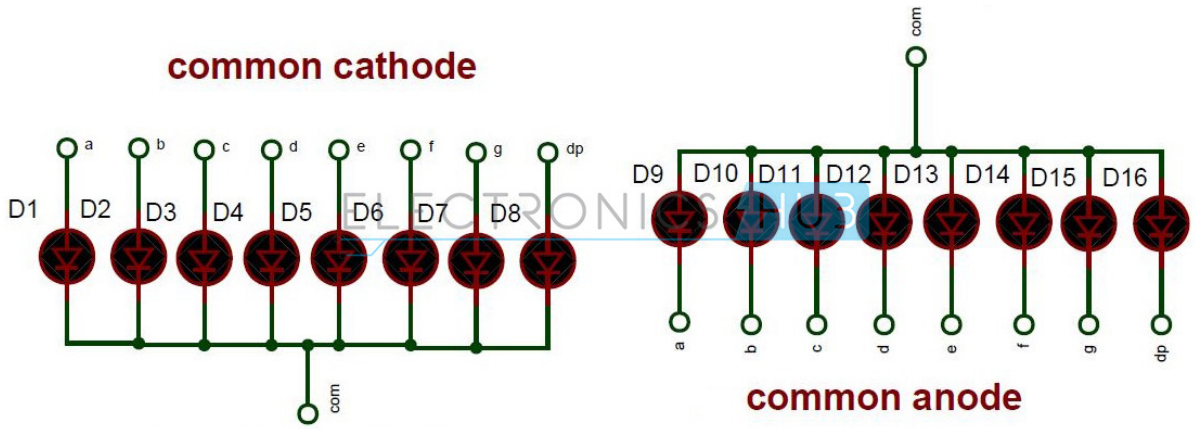
- **Table Usage:** The code loads the digit to be displayed into R0, uses indirect addressing to get the corresponding pattern from the lookup table, and sends it to the display port.

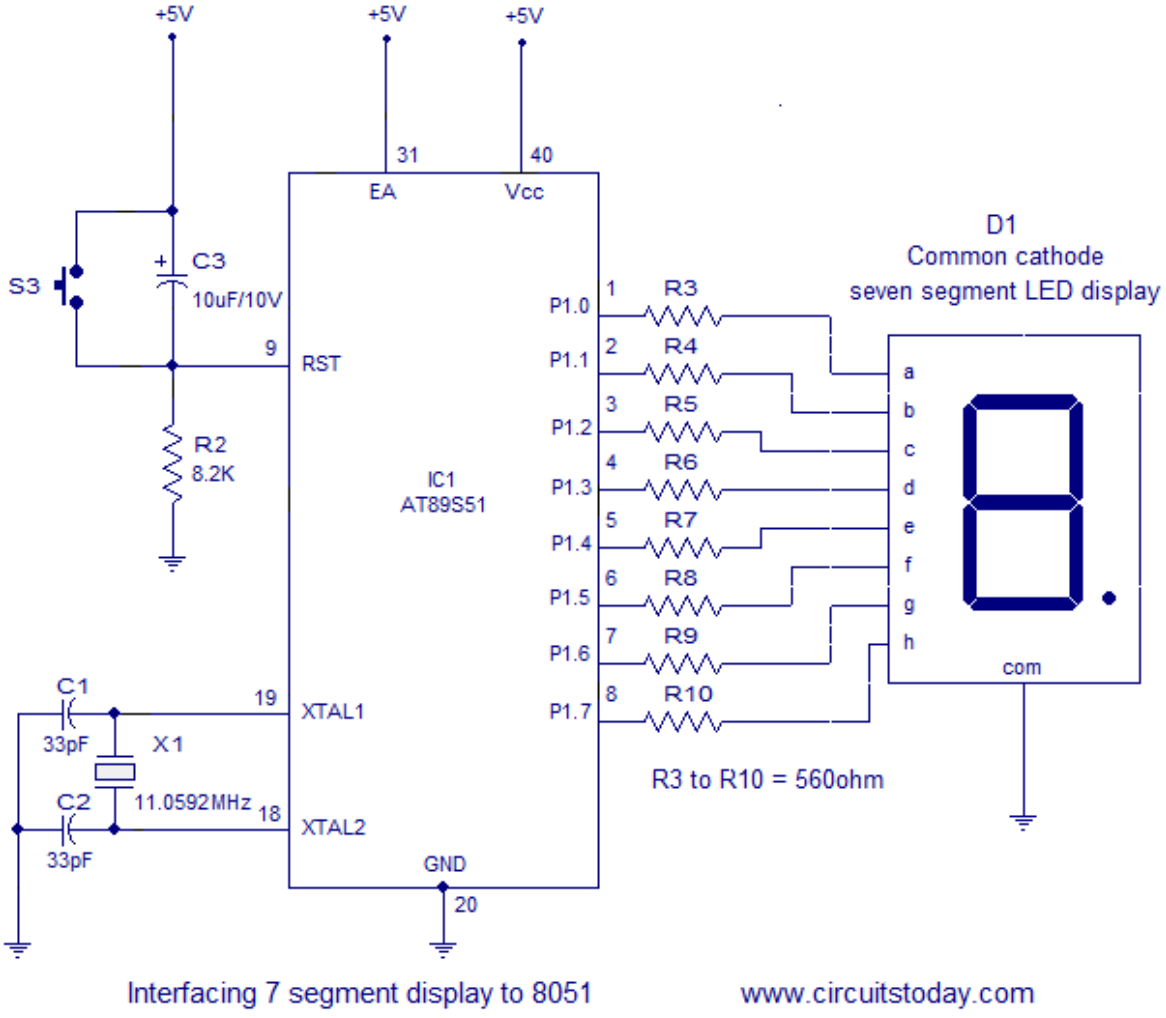
- **Multiplexing:** If you have multiple 7-segment displays, you need to switch between them rapidly and update the display port accordingly to create the illusion they are all on simultaneously.

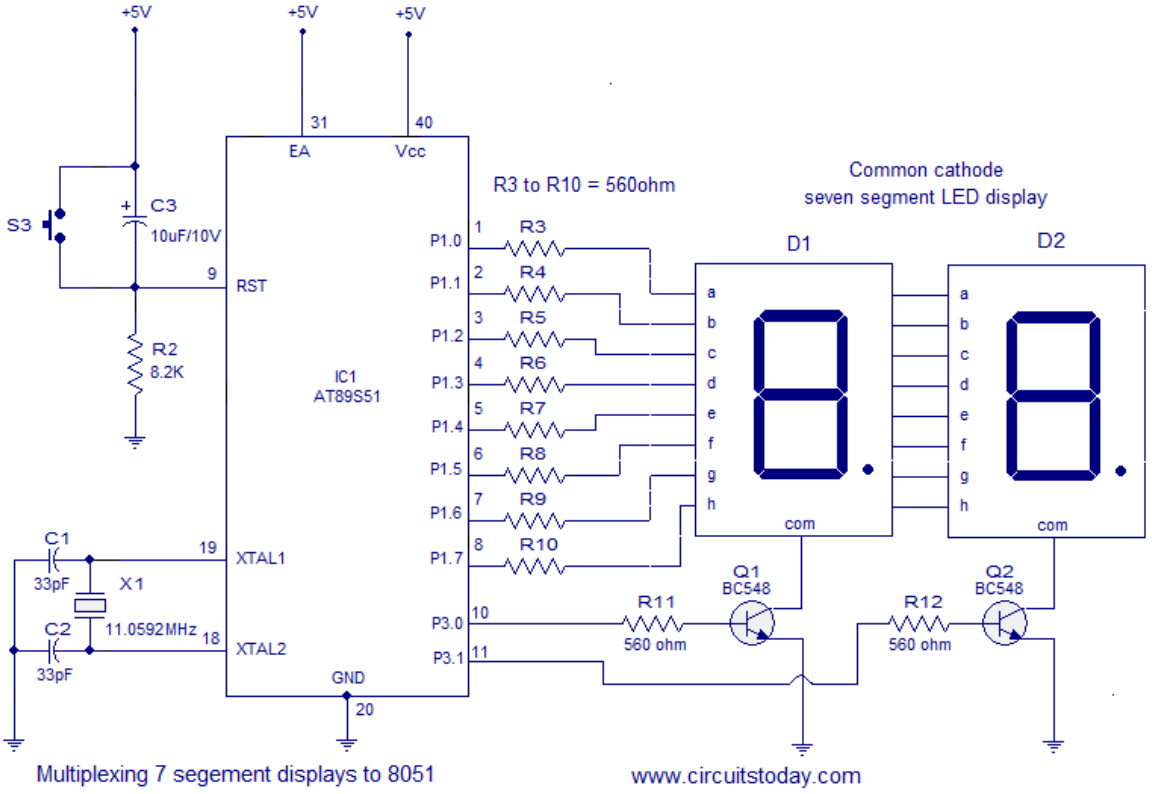
Important:

- **Port Output:** Ensure the port you use is configured as output.
- **Resistors:** Don't forget the current-limiting resistors!

Draw block diagram to interface one common anode seven segment LED with port P0 of 8051.







LCDs

Q5c: Interface LCD with 8051 microcontroller and write a program to display “welcome”.

Hardware Setup

- **LCD:** We'll assume a standard 16x2 character LCD module with a common HD44780 compatible controller.
- **Connections:**
  - **Data Pins (D0-D7):** Connect to an 8051 I/O Port (e.g., Port 2)
  - **Control Pins:**
    - RS (Register Select): Connect to an 8051 pin (e.g., P1.0)
    - RW (Read/Write): Connect to an 8051 pin (e.g., P1.1)
    - E (Enable): Connect to an 8051 pin (e.g., P1.2)
  - **Contrast Adjustment (Vo):** Connect to a potentiometer for controlling display contrast.
  - **Backlight (if present):** Power according to its requirements.

Assembly Programming

Here's an 8051 assembly program to initialize the LCD and display "Welcome":

```
ORG 0000H

; Constants
LCD_PORT EQU P2
RS_PIN   EQU P1.0
RW_PIN   EQU P1.1
E_PIN    EQU P1.2
```

```
; --- LCD Initialization ---
LCD_INIT:
    MOV A, #38H  ; Function set: 8-bit, 2 lines, 5x7 font
    CALL LCD_CMD
    MOV A, #0CH  ; Display on, cursor off, no blinking
    CALL LCD_CMD
    MOV A, #01H  ; Clear display
    CALL LCD_CMD
    MOV A, #06H  ; Entry mode: Increment cursor
    CALL LCD_CMD
    RET

; --- Send Command to LCD  ---
LCD_CMD:
    CLR RS_PIN
    CLR RW_PIN
    MOV LCD_PORT, A
    SETB E_PIN     ; Pulse Enable
    CLR E_PIN
    CALL DELAY     ; Small delay (important for LCD)
    RET

; --- Send Data (Character) to LCD ---
LCD_DATA:
    SETB RS_PIN
    CLR RW_PIN
    MOV LCD_PORT, A
    SETB E_PIN      ; Pulse Enable
    CLR E_PIN
    CALL DELAY      ; Small delay
    RET

; --- Simple Delay Subroutine ---
DELAY:
    MOV R5, #50    ; Adjust these for approximate delay
    DLOOP: MOV R6, #200
           DJNZ R6, DLOOP
           DJNZ R5, DLOOP
           RET

; -- Main Program --
MAIN:
    CALL LCD_INIT

    MOV A, #80H     ; Set cursor to first line, first position
    CALL LCD_CMD

    MOV A, #'W'    ; Load characters to send
    CALL LCD_DATA
    MOV A, #'e'
    CALL LCD_DATA
    MOV A, #'l'
    CALL LCD_DATA
    ; ... (Send rest of "come")

END ; Add an infinite loop if needed for display to stay
```

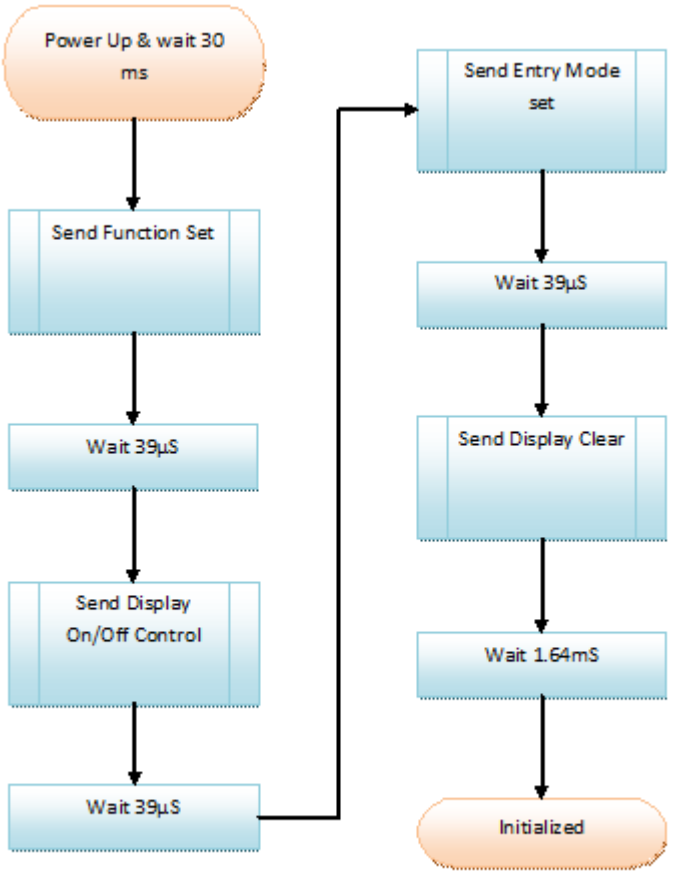
Explanation

- **Constants:** `LCD_PORT` is defined to make the code adaptable if your connections change.
- **Subroutines:** These encapsulate the interaction details with the LCD (sending commands, sending data), making the main program cleaner. You'll need to fill in the details of these subroutines according to your LCD's datasheet.
- **Main Program:**
  - Calls `LCD_INIT` to configure the LCD.
  - Sends commands to select the desired display mode (2 lines, 5x7 font) and clear the display.
  - Sends each character of the message "Welcome" to the LCD using `LCD_DATA`.

Key Points

- **LCD Datasheet:** You MUST adapt the initialization and command sequences based on your specific LCD module.
- **8051 Timing:** You might need short delays within the subroutines to ensure the LCD processes commands correctly.
- **Subroutine Implementation:** The core logic of sending commands/data to the LCD involves setting the RS/RW lines, placing data on the data port, and pulsing the Enable pin.

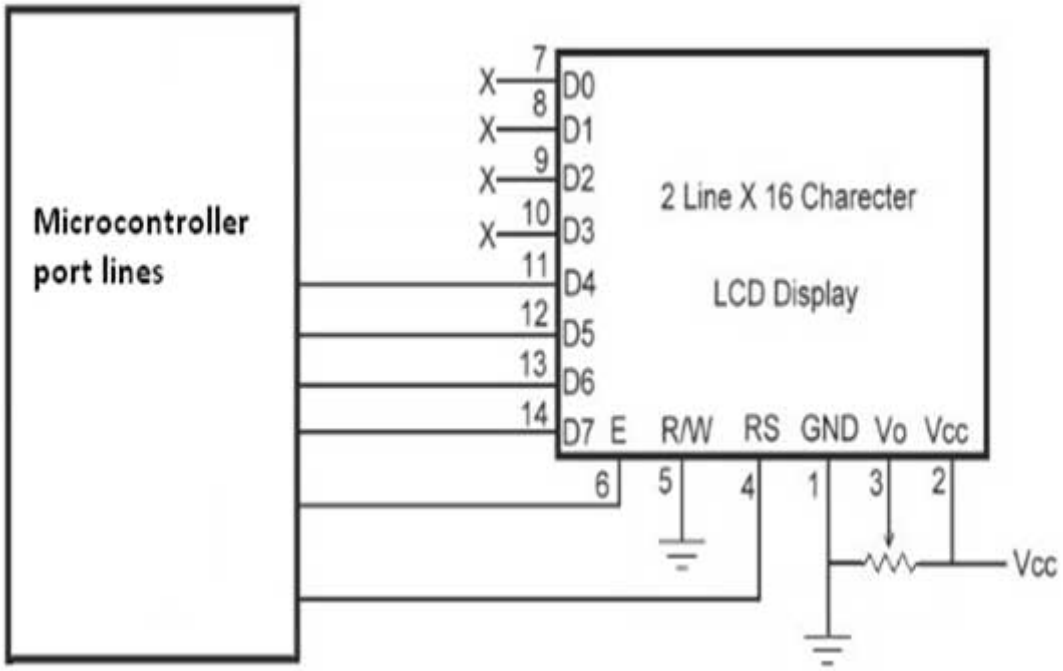
Write steps to Initialize LCD. (3)

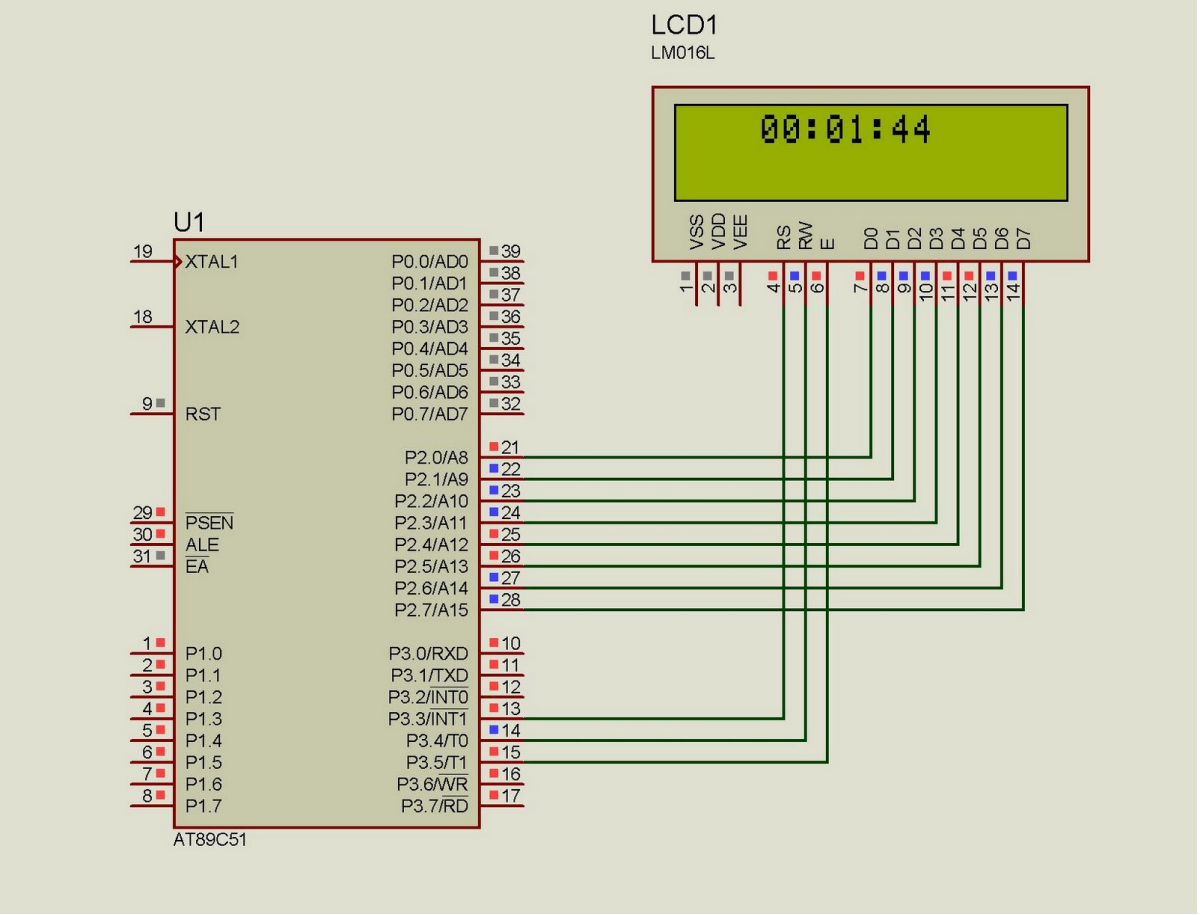


LCD Initialization Flow Chart

```
void LCD_init()
{
    LCD_data = 0x38; // Function set: 2 Line, 8-bit, 5x7 dots
    LCD_rs = 0; // Selected command register
    LCD_rw = 0; // We are writing in data register
    LCD_en = 1; // Enable H->
    LCD_en = 0;
    LCD_busy(); // Wait for LCD to process the command
    LCD_data = 0x0F; // Display on, Curson blinking command
    LCD_rs = 0; // Selected command register
    LCD_rw = 0; // We are writing in data register
    LCD_en = 1; // Enable H->
    LCD_en = 0;
    LCD_busy(); // Wait for LCD to process the command
    LCD_data = 0x01; // Clear LCD
    LCD_rs = 0; // Selected command register
    LCD_rw = 0; // We are writing in data register
    LCD_en = 1; // Enable H->
    LCD_en = 0;
    LCD_busy(); // Wait for LCD to process the command
    LCD_data = 0x06; // Entry mode, auto increment with no shift
    LCD_rs = 0; // Selected command register
    LCD_rw = 0; // We are writing in data register
    LCD_en = 1; // Enable H->
    LCD_busy();
}
```

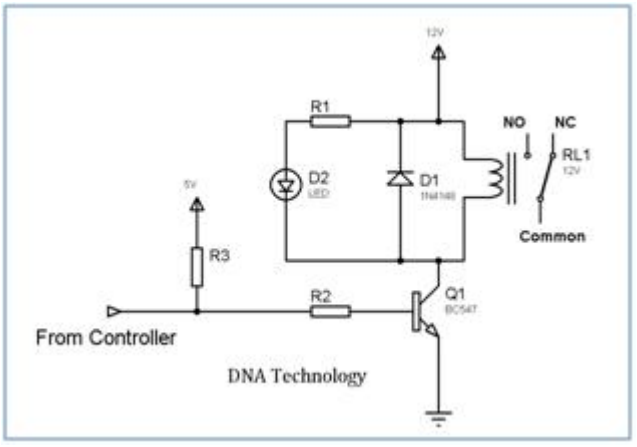
Explain interfacing of LCD in brief. (4)

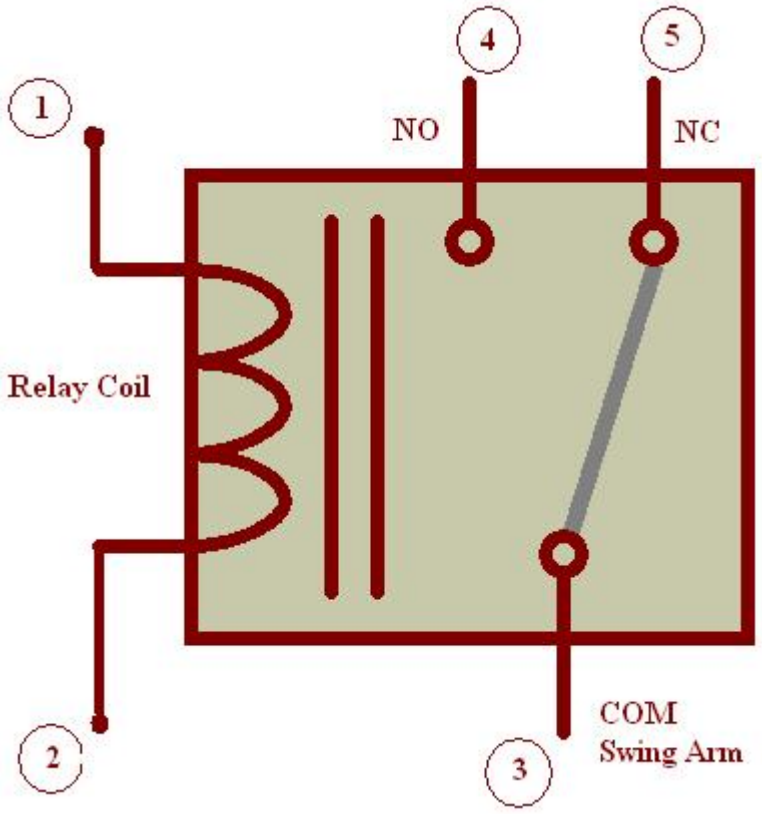


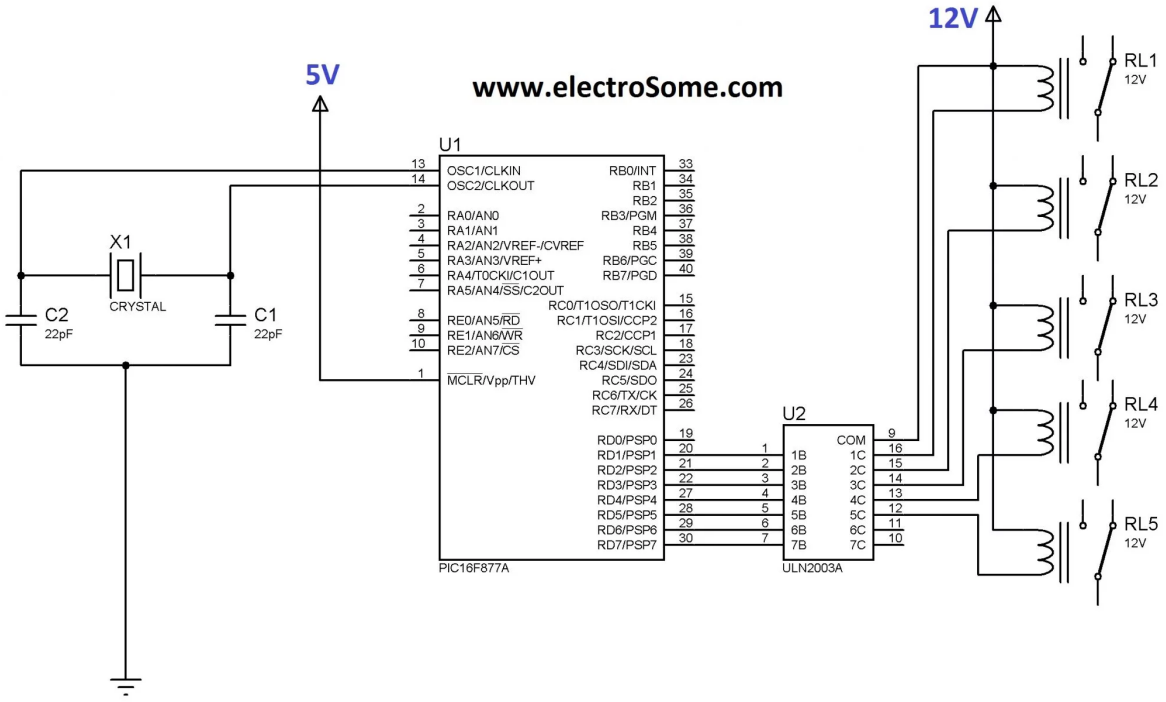


Relays

Draw circuit diagram for interfacing of relay with 8051. (3)

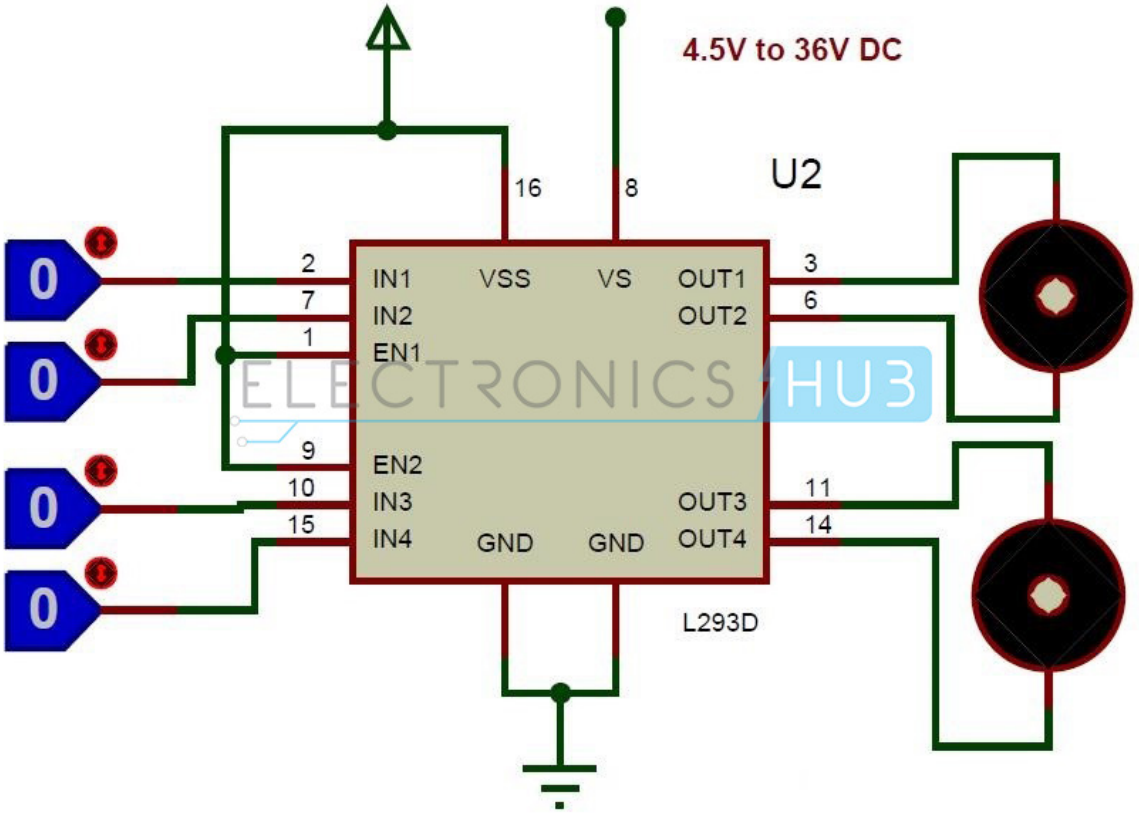


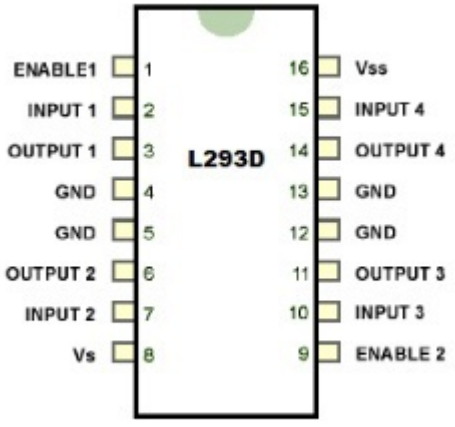

1
2
3
4
5
NO
NC
Relay Coil
COM
Swing Arm


12V
5V
www.electroSome.com
U1
PIC16F877A
U2
ULN2003A
X1
CRYSTAL
C2
22pF
C1
22pF
RL1
RL2
RL3
RL4
RL5
12V

DC Motors

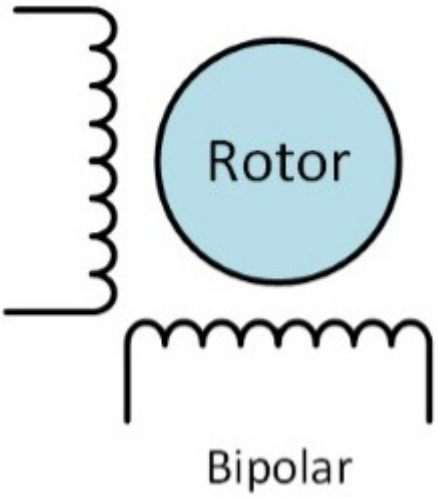
Draw diagram of interfacing dc motor and explain in brief. (4)



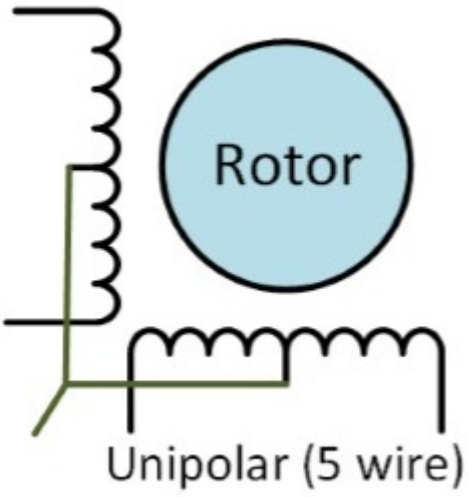


Stepper Motors

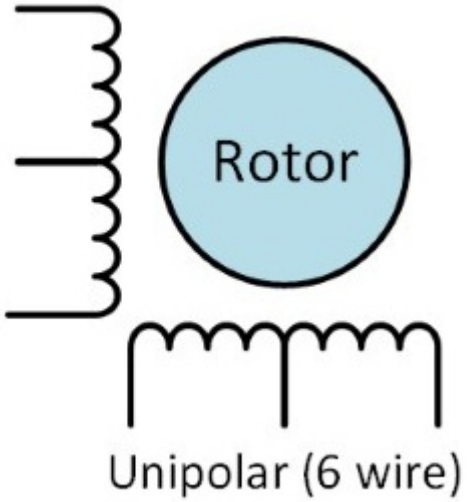
Draw diagram of interfacing stepper motor and explain in brief. (4)



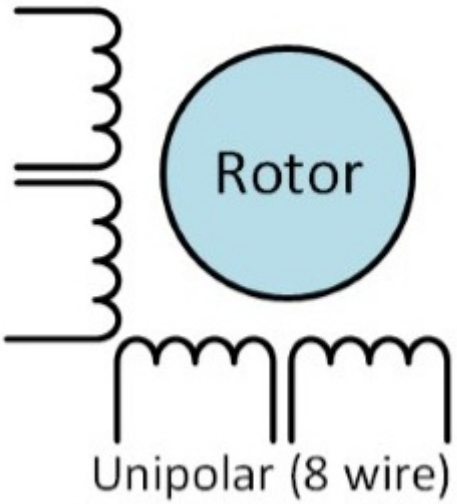
Bipolar



Unipolar (5 wire)



Unipolar (6 wire)



Unipolar (8 wire)

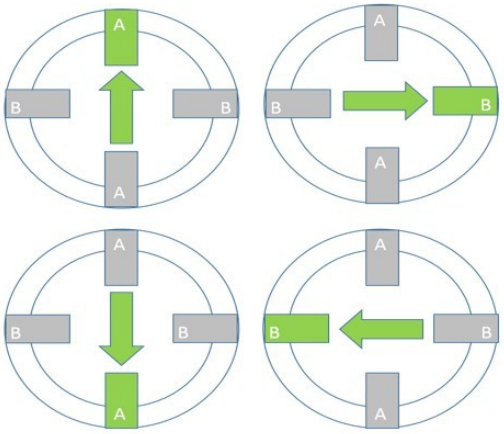


Fig 1 – One phase on – full step

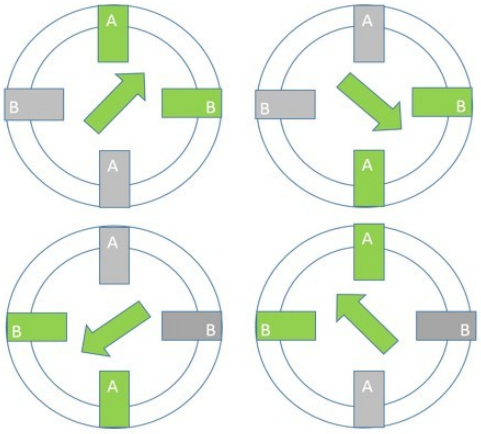


Fig2 – Two phase on – full step

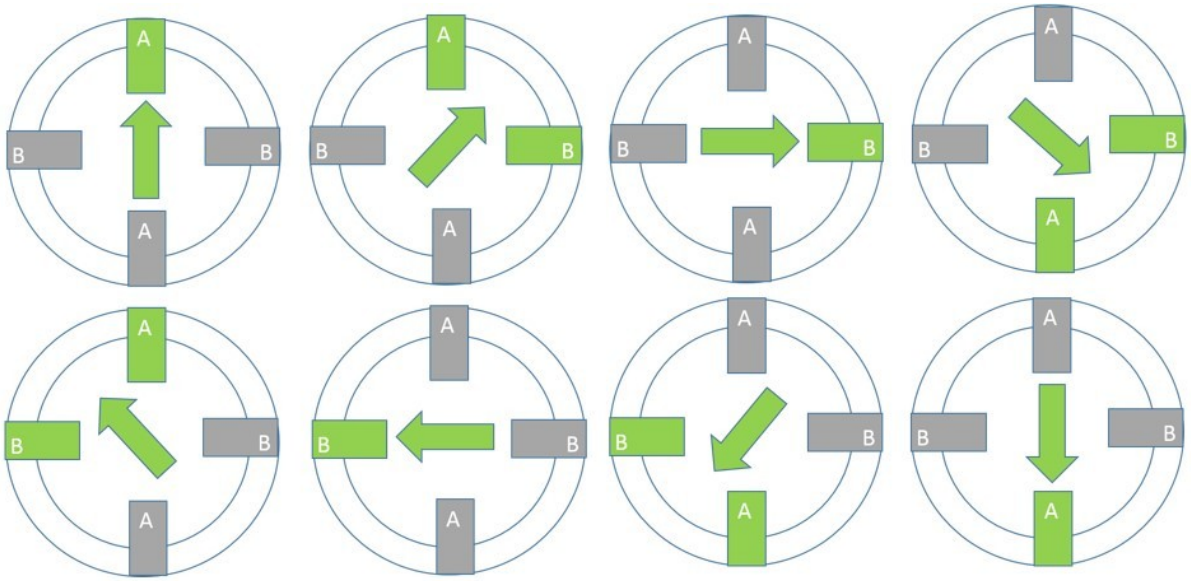
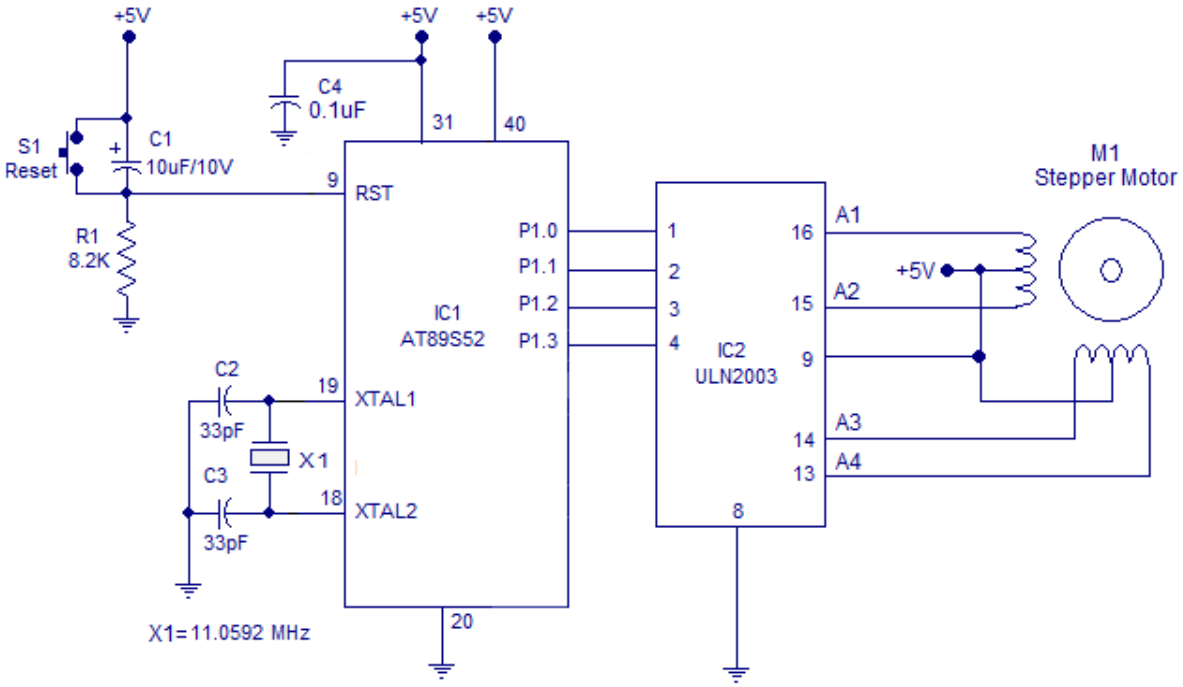


Fig3 - One-two phase on - half step



Wave Drive

A	B	C	D
1	0	0	0
0	1	0	0
0	1	0	0
0	0	0	1

Full Drive

A	B	C	D
1	1	0	0

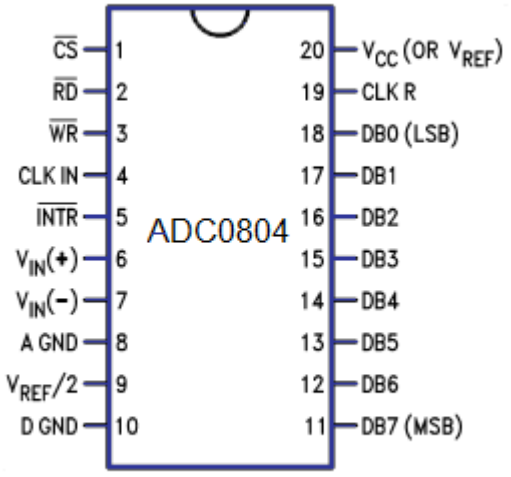
A	B	C	D
0	1	1	0
0	0	1	1
1	0	0	1

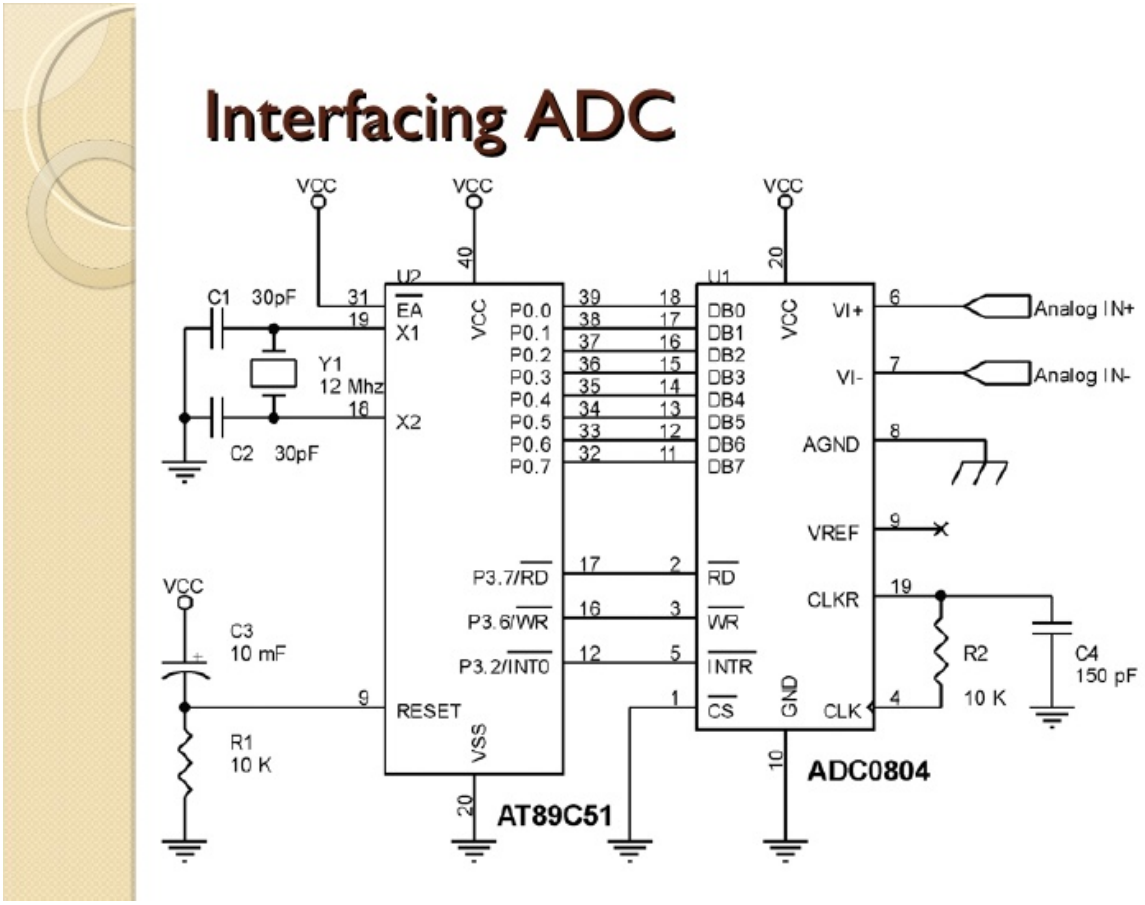
Half Drive

A	B	C	D
1	0	0	0
1	1	0	0
0	1	0	0
0	1	1	0
0	0	1	0
0	0	1	1
0	0	0	1
1	0	0	1

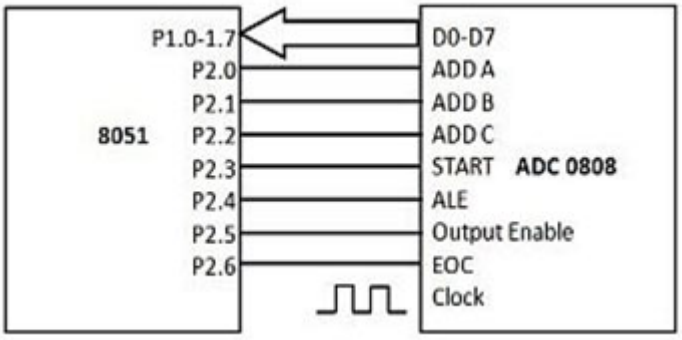
ADC and DAC Interfacing

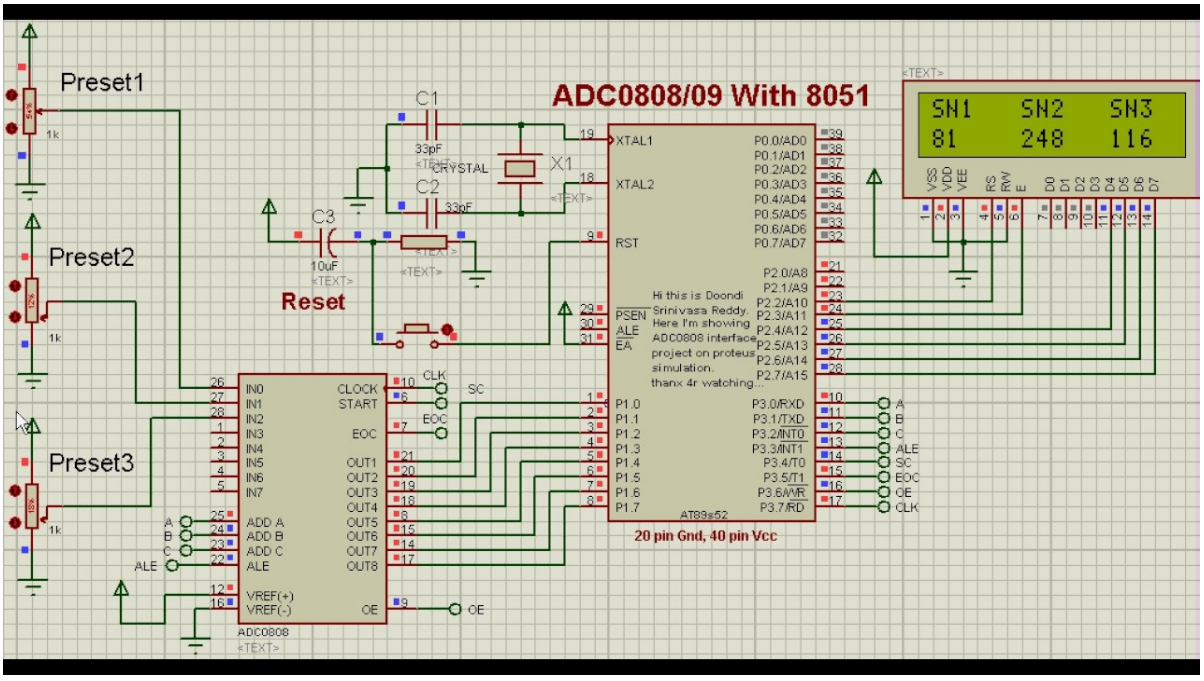
Draw circuit diagram for interfacing ADC 0804 with 8051. (3)



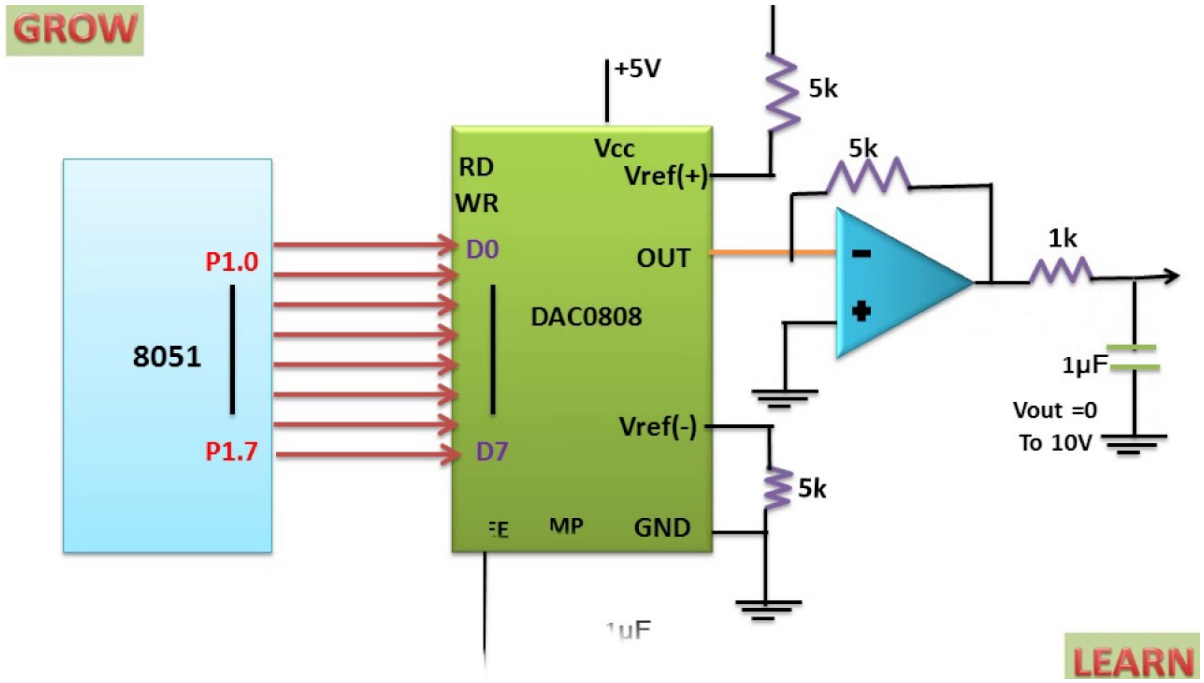


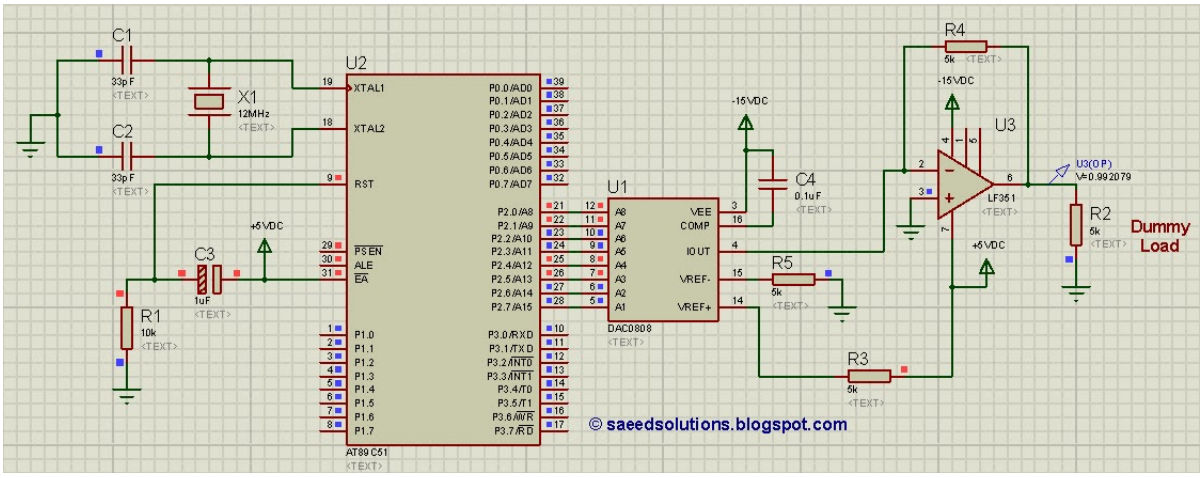
Interface ADC 0808 with 8051 and write a program to read digital output. (3)





Interface DAC 0808 with 8051 and write a program to create ramp signal. (4) & Draw interface diagram of DAC 0808 with 8051 microcontroller and write a program to generate Triangular wave. (7)





ADC0804, DAC0808

Real-World Applications

List applications of mocrocontrollers in various fields. (2)

- Consumer Electronics Products: Toys, Cameras, Robots, Washing Machine, Microwave Ovens etc. [any automatic home appliance]
- Instrumentation and Process Control: Oscilloscopes, Multi-meter, Leakage Current Tester, Data Acquisition and Control etc.
- Medical Instruments: ECG Machine, Accu-Check etc.
- Communication: Cell Phones, Telephone Sets, Answering Machines etc.
- Office Equipment: Fax, Printers etc.
- Multimedia Application: Mp3 Player, PDAs etc.
- Automobile: Speedometer, Auto-breaking system etc.

Industrial Automation

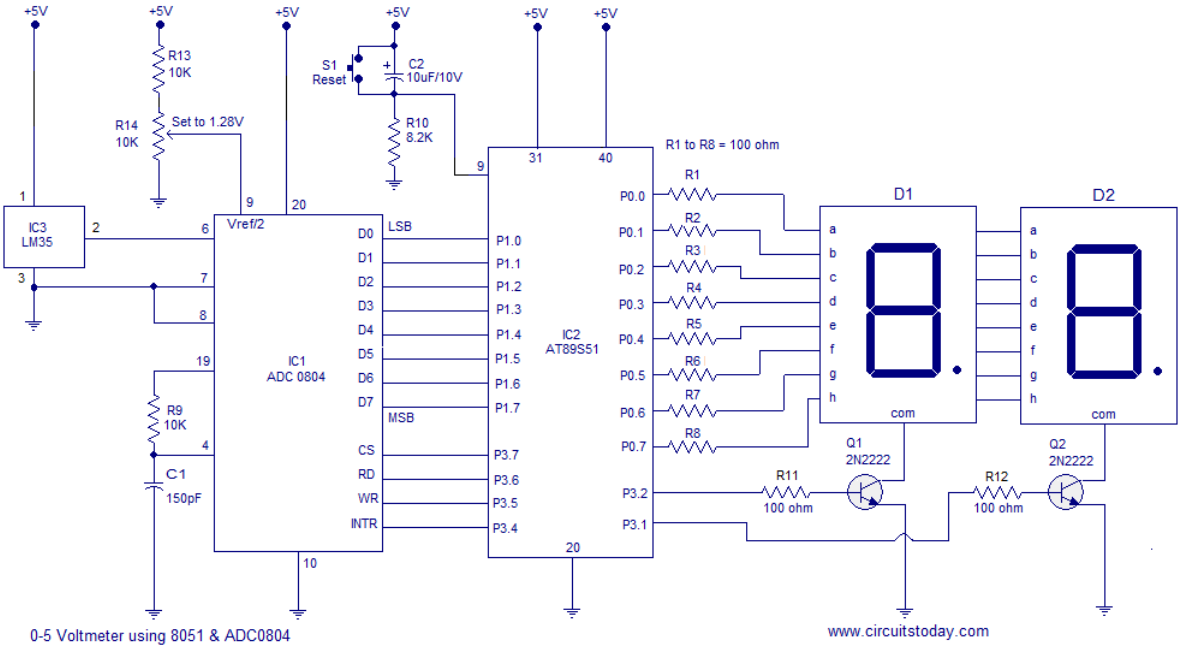
Embedded Systems

Consumer Electronics

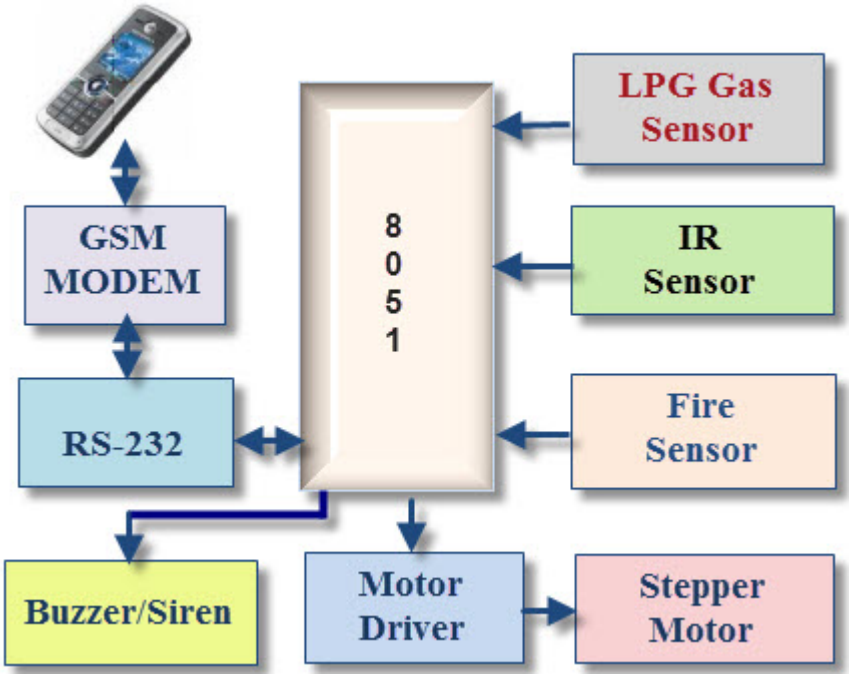
Automotive Systems

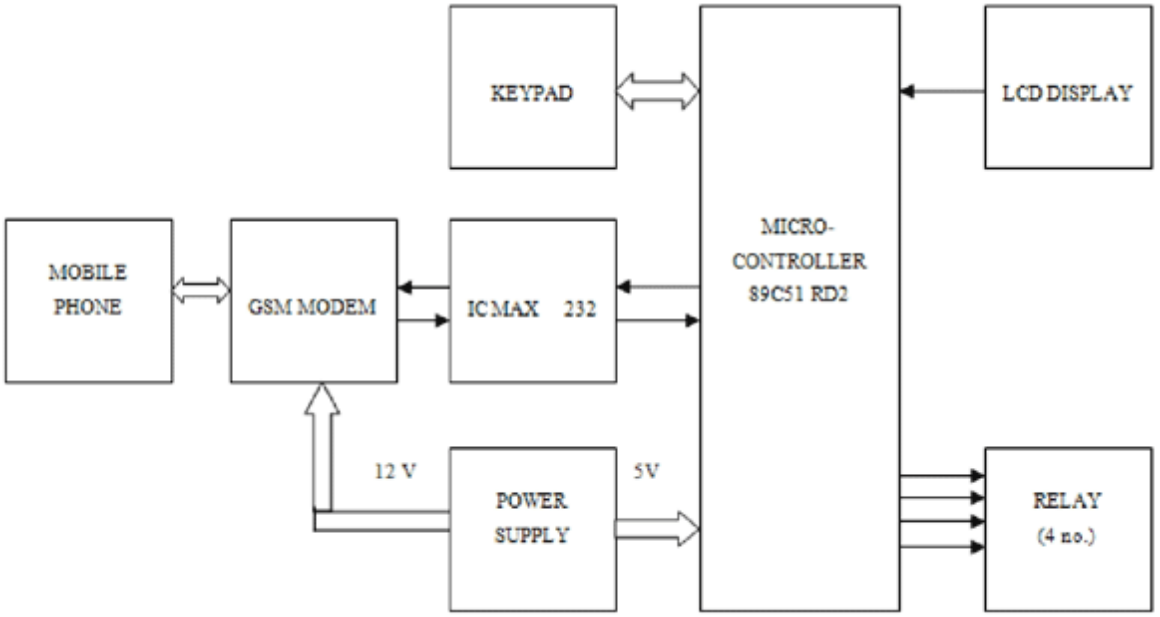
Miscellaneous

Draw block diagram of room temperature indicator system using 8051, LM35, ADCC0804, Microcontroller, 7 segment LED. (3)

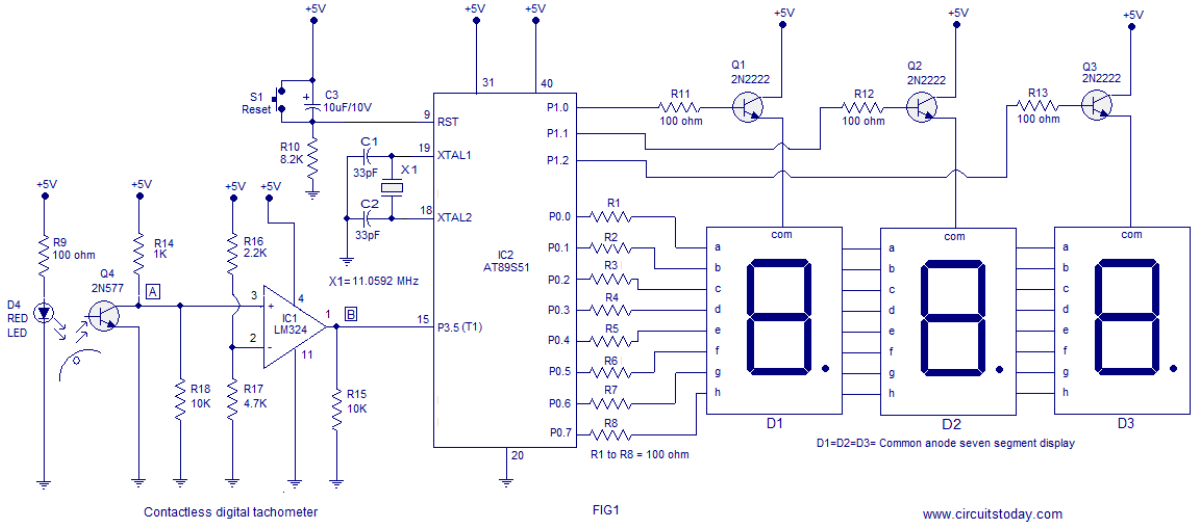


Explain GSM based security system GSM Modem, Microcontroller, Relay, Switches. (3)





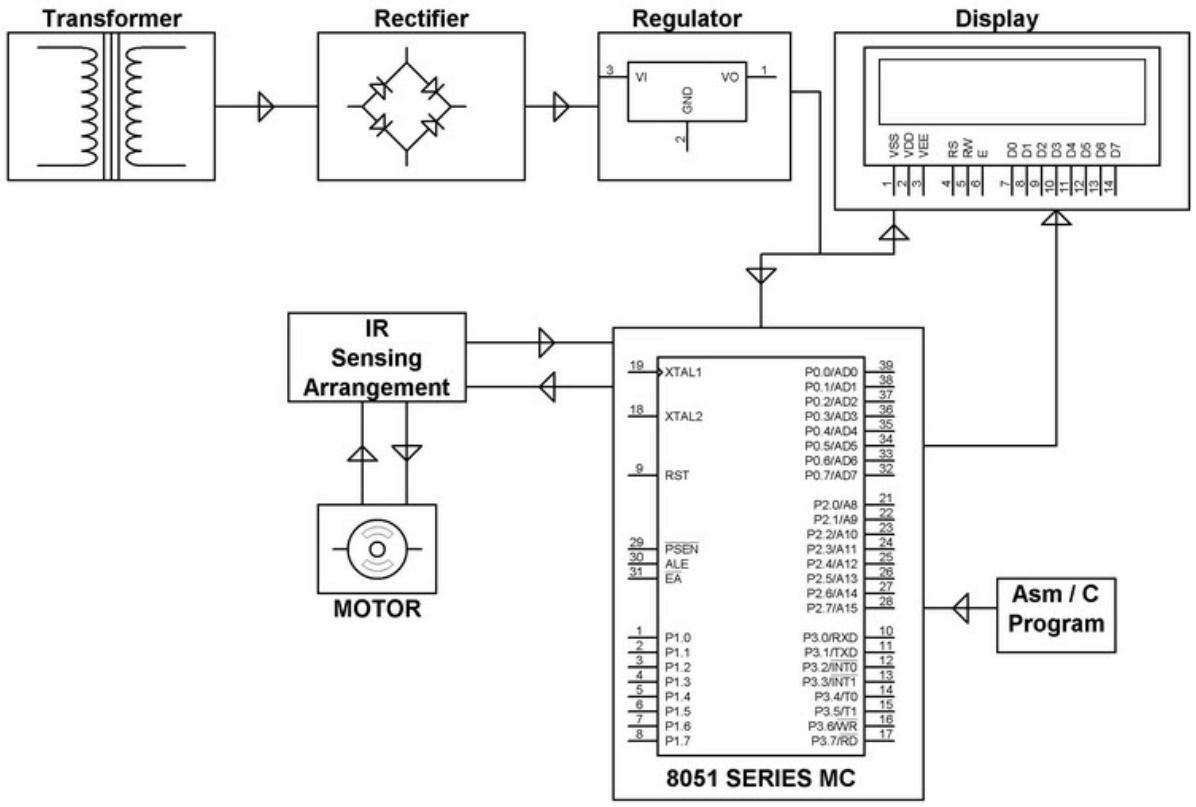
Explain RPM meter using Photo interrupter, Microcontroller, 7 Segment LED. (3)



Contactless digital tachometer

FIG1

www.circuitstoday.com



Draw circuit diagram of application based on RTC DS1307.