# Microprocessor & Microcontroller (DI04000111)

Gujarat Technological University

# Table of Contents I

# Table of Contents II

# Table of Contents III

# Table of Contents IV

# Table of Contents V

# Table of Contents VI

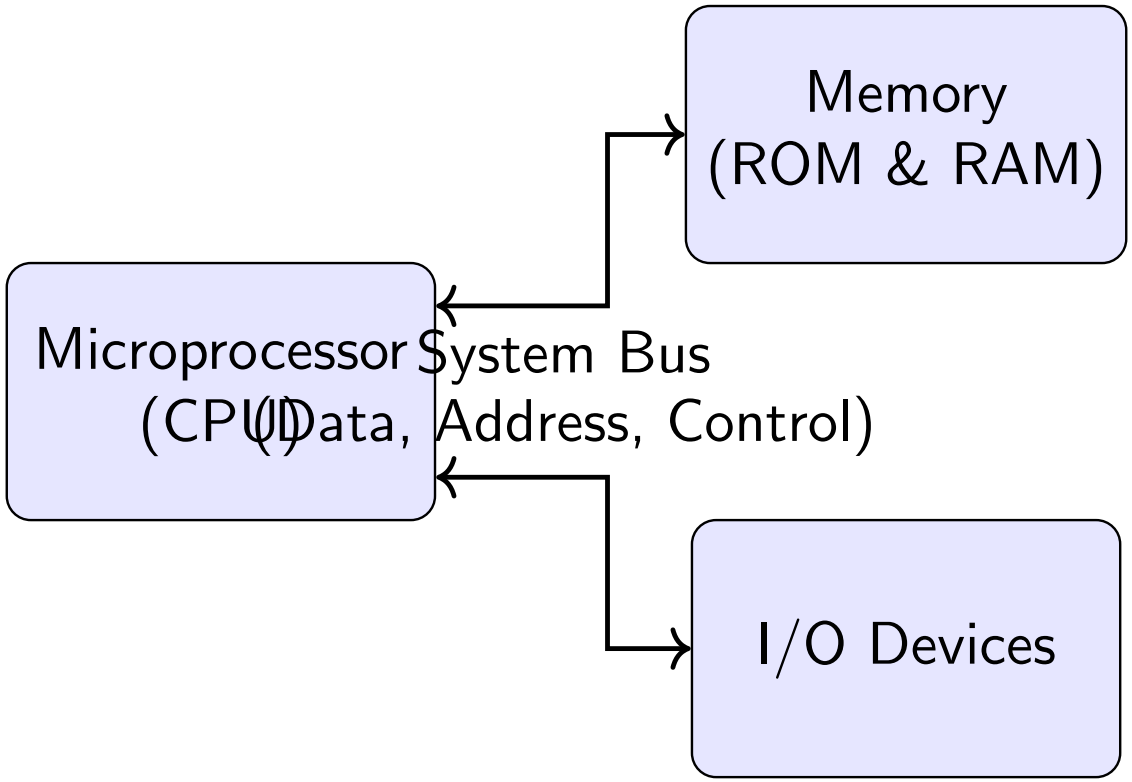
# What is a Microprocessor?

## Definition

A microprocessor is a multipurpose, programmable, clock-driven, register-based electronic device that reads binary instructions from a storage device called memory, accepts binary data as input, and processes data according to those instructions to provide results as output.

- Acts as the **Central Processing Unit (CPU)** on a single integrated circuit (IC).
- Consists of Arithmetic Logic Unit (ALU), Control Unit, and Registers.
- Executes instructions sequentially.

# Microprocessor-Based System (Microcomputer)




## Components

- **Microprocessor:** The brain of the system.
- **Memory:** Stores instructions (ROM) and data (RAM).
- **I/O Devices:** Facilitates communication with the outside world.

# History and Evolution of Microprocessors

- **First Generation (1971-1973):**
  - 4-bit processors using PMOS technology.
  - E.g., Intel 4004 (First commercially available microprocessor).
- **Second Generation (1974-1978):**
  - 8-bit processors using NMOS technology. Faster and denser.
  - E.g., Intel 8085, Motorola 6800.
- **Third Generation (1979-1980):**
  - 16-bit processors using HMOS technology.
  - E.g., Intel 8086, Zilog Z8000.

# History and Evolution of Microprocessors (Contd.)

- **Fourth Generation (1981-1995):**
  - 32-bit processors using HCMOS technology.
  - Introduction of caching and pipelining.
  - E.g., Intel 80386, 80486, Pentium.
- **Fifth Generation (1995-Present):**
  - 64-bit processors.
  - Multi-core processors, superscalar architecture.
  - E.g., Intel Core i3/i5/i7/i9, AMD Ryzen, ARM Cortex.

# Comparison of Intel Processors (Early Generations)

| Processor | Year | Data Bus | Address Bus | Memory Capacity |
|---|---|---|---|---|
| Intel 4004 | 1971 | 4-bit | 12-bit | 4 KB |
| Intel 8008 | 1972 | 8-bit | 14-bit | 16 KB |
| Intel 8080 | 1974 | 8-bit | 16-bit | 64 KB |
| Intel 8085 | 1976 | 8-bit | 16-bit | 64 KB |
| Intel 8086 | 1978 | 16-bit | 20-bit | 1 MB |

## Key Takeaway

As technology advanced, bus sizes increased, leading to exponentially larger memory capacities and faster processing speeds.

# Basic Components of a Microprocessor System

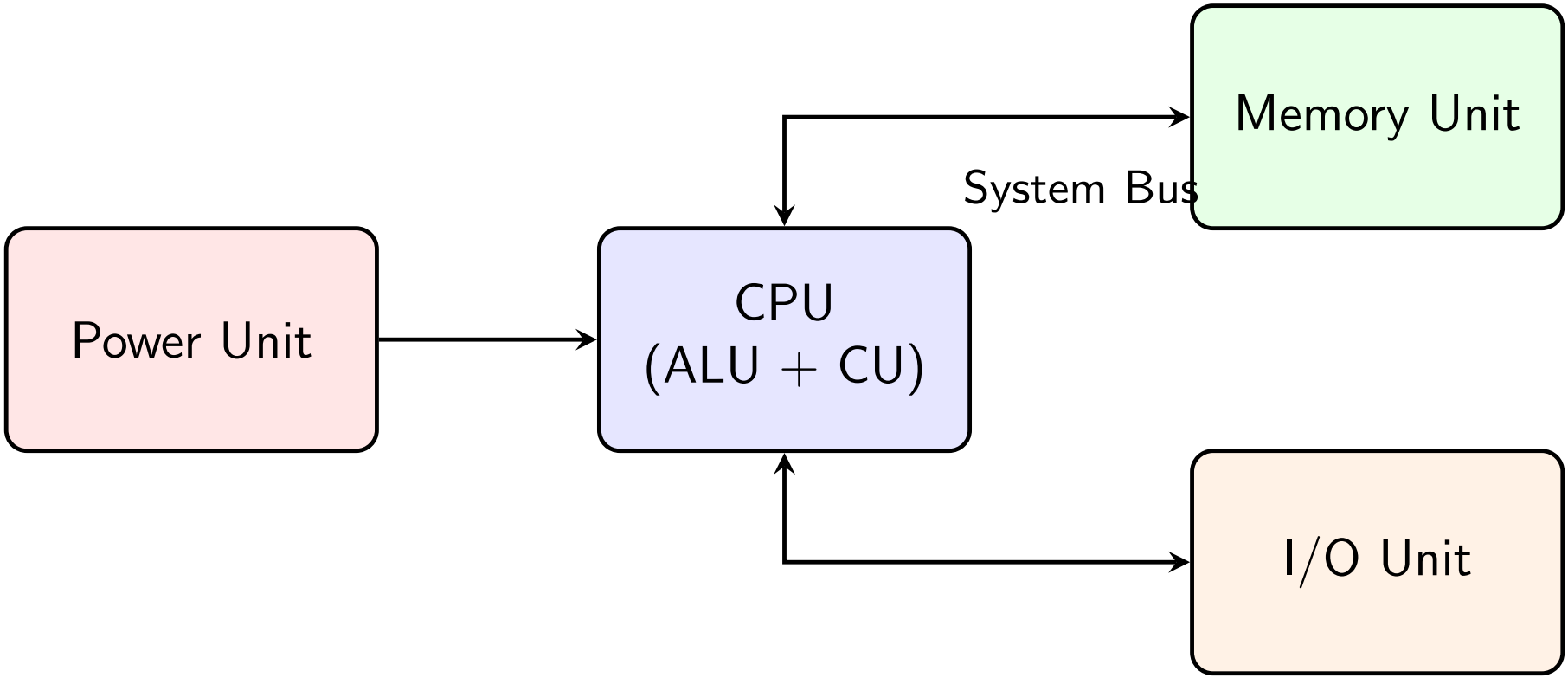
## Overview

A typical microprocessor-based system (or microcomputer) consists of several essential functional units that work together to process information.

The major components are:

- **CPU (Central Processing Unit)**: The "brain" of the system.
- **Memory Unit**: Stores instructions (program) and data.
- **Input/Output (I/O) Unit**: Communicates with the outside world.
- **Power Unit**: Provides necessary electrical power to all components.

# System Block Diagram




## System Bus

The components communicate over a set of parallel electrical lines called the **System Bus** (Data, Address, and Control buses).

# Central Processing Unit (CPU)

## What is the CPU?

The Central Processing Unit is the core component of any computer system. In a microcomputer, the CPU is implemented as a single integrated circuit (IC) known as the **Microprocessor**.

- **Main Functions:**
  - Fetches instructions from memory.
  - Decodes the fetched instructions.
  - Executes the instructions.
  - Controls the timing and data flow.

The CPU itself is divided into two primary sub-units: the **Arithmetic Logic Unit (ALU)** and the **Control Unit (CU)**.

# Arithmetic Logic Unit (ALU)

## Function of ALU

The ALU performs all the mathematical and logical operations required by the instructions.

| **Operation Type** | **Examples** |
|---|---|
| Arithmetic | Addition, Subtraction, Increment, Decrement |
| Logical | AND, OR, NOT, XOR |
| Shift / Rotate | Shift Left, Shift Right, Rotate through Carry |

- Receives operands from registers (like the Accumulator).
- Generates the result and updates the **Status Flags** (Carry, Zero, Sign, etc.).
- The result is typically stored back into a register or memory.

# Control Unit (CU)

## Role of the Control Unit

The Control Unit directs the operation of the processor. It does not execute instructions itself but tells the other parts of the system how to do so.

- Generates necessary **timing and control signals**.
- Synchronizes all operations with the system clock.
- Manages the flow of data between the CPU, memory, and I/O devices.
- **Instruction Decoder:** A key part of the CU that translates the fetched instruction (machine code) into a sequence of micro-operations.

# Memory Unit

## Purpose

The memory unit stores the instructions that the CPU must execute (the program) as well as the data those instructions manipulate.

Primary Memory is broadly classified into two types:

- **Read-Only Memory (ROM):**
  - Non-volatile (retains data without power).
  - Typically stores firmware, bootloaders, or fixed application code.
- **Random Access Memory (RAM):**
  - Volatile (loses data when powered off).
  - Used for temporary data storage, variables, and the stack.

# Input and Output (I/O) Unit

## Connecting to the Outside World

The I/O unit provides the interface between the microprocessor and external devices (peripherals).

- **Input Devices:** Allow external data to be fed into the system (e.g., Keyboards, Sensors, Switches, ADCs).
- **Output Devices:** Allow the system to send data out to the user or environment (e.g., Displays, LEDs, Motors, DACs).

## I/O Interfacing

Because external devices often operate at different speeds and voltage levels than the CPU, **I/O Interfaces** or **Ports** (buffers, latches) are used to synchronize and manage the data transfer.

# Power Unit

## Power Supply

All electronic components in the microprocessor system require a stable DC voltage to operate.

- The **Power Unit** converts AC mains voltage to the required regulated DC voltage(s).
- Standard microprocessors (like the classic 8085) typically require a stable **+5V DC** supply.
- Modern low-power microcontrollers might operate on +3.3V, +1.8V, or even lower.
- Must provide adequate current to drive the CPU, Memory, and all I/O peripherals.
- Often includes protection circuitry against over-voltage or short-circuits.

# Summary

- A microprocessor system is formed by combining a **CPU** with **Memory** and **I/O Units**, all powered by a **Power Unit**.
- The **CPU** is the brain, split into the **ALU** (computations) and **Control Unit** (timing and orchestration).
- **Memory** holds the program instructions and runtime data.
- **I/O Units** enable interaction with users or other systems.
- These units communicate over a shared set of electrical lines known as the **System Bus**.

# Von Neumann Architecture

## Overview

The Von Neumann architecture, named after John von Neumann, is a computer architecture based on the 1945 description of the stored-program computer concept.

- **Single Memory Space:** Uses a single memory to store both instructions and data.
- **Shared Bus:** Employs a single data bus and a single address bus for both instructions and data fetches.
- **Execution:** Sequential execution (one instruction at a time).
- **Bottleneck:** The "Von Neumann bottleneck" occurs because the CPU cannot read an instruction and read/write data at the same time.

# Harvard Architecture

## Overview

The Harvard architecture is a computer architecture with physically separate storage and signal pathways for instructions and data.

- **Separate Memories:** Distinct memory blocks for data and instructions.
- **Multiple Buses:** Features separate data and address buses for instructions and data.
- **Simultaneous Access:** The CPU can read an instruction and perform a data memory access at the same time.
- **Speed:** Generally faster execution due to parallel instruction and data fetching, eliminating the Von Neumann bottleneck.

# Von Neumann vs. Harvard Architecture

| **Feature** | **Von Neumann** | **Harvard** |
|---|---|---|
| Memory Space | Single for data and instructions | Separate for data and instructions |
| Buses | Shared data and address buses | Separate data and address buses |
| Space Requirement | Requires less physical space | Requires more physical space |
| Execution Speed | Slower (sequential access) | Faster (simultaneous access) |
| Hardware Complexity | Simpler | More complex |

# CISC Architecture

## Complex Instruction Set Computer (CISC)

CISC is an architectural design where single instructions can execute several low-level operations or are capable of multi-step operations or addressing modes within single instructions.

- **Instruction Size:** Variable instruction length.
- **Hardware Emphasis:** Focuses on hardware complexity to reduce software complexity.
- **Memory:** Requires less memory to store instructions because instructions are dense.
- **Clock Cycles:** Execution of a single instruction may take multiple clock cycles.
- **Example:** Intel x86 processors.

# RISC Architecture

## Reduced Instruction Set Computer (RISC)

RISC is a microprocessor architecture that utilizes a small, highly-optimized set of instructions, rather than a more specialized set of instructions often found in other types of architectures.

- **Instruction Size:** Fixed instruction length (usually 32-bit).
- **Software Emphasis:** Focuses on software complexity; requires more instructions to perform tasks.
- **Memory:** Requires more memory to store a larger number of simpler instructions.
- **Clock Cycles:** Aims for single-cycle execution of instructions.
- **Registers:** Contains a large number of registers to minimize memory access.
- **Example:** ARM, MIPS.

# RISC vs. CISC Architecture

| **Feature** | **RISC** | **CISC** |
|---|---|---|
| Instruction Set | Small, highly optimized | Large, complex |
| Instruction Length | Fixed | Variable |
| Hardware/Software | Emphasis on software | Emphasis on hardware |
| Execution Time | Often single cycle | Multiple cycles per instruction |
| Registers | Large number of registers | Fewer registers |
| Power Consumption | Generally lower | Generally higher |

# 8085 Microprocessor Pin Configuration

## Overview of 8085 Pins

The Intel 8085 is a 40-pin IC (Integrated Circuit) packaged in a Dual In-line Package (DIP). It operates on a single +5V power supply.

- The pins can be broadly classified into six groups:
  1. Address Bus
  2. Data Bus
  3. Control and Status Signals
  4. Power Supply and Clock Signals
  5. Interrupts and Externally Initiated Signals
  6. Serial I/O Ports

# Address Bus and Data Bus

## Address Bus ($A_{15} - A_8$)

- 16-bit address bus allows addressing up to $2^{16} = 64$ KB of memory.
- The upper 8 bits ($A_{15} - A_8$) are unidirectional.
- Used to carry the Most Significant Bits (MSB) of the memory/IO address.

## Multiplexed Address/Data Bus ($AD_7 - AD_0$)

- The lower 8 bits of the address bus are multiplexed with the 8-bit data bus.
- Bi-directional lines.
- Serves as address bus ($A_7 - A_0$) during the first clock cycle ($T_1$) of a machine cycle.
- Serves as data bus ($D_7 - D_0$) during subsequent clock cycles ($T_2, T_3$).

# Demultiplexing the Bus

## Address Latch Enable (ALE)

- A positive going pulse generated every time the 8085 begins an operation (machine cycle).
- When ALE is HIGH, the multiplexed bus $AD_7 - AD_0$ carries the lower-order address ($A_7 - A_0$).
- This address is latched into an external latch (like 74LS373) using the ALE signal.
- Once latched, the bus becomes available as the data bus ($D_7 - D_0$) for the rest of the machine cycle.

# Control and Status Signals

## Control Signals

- **$\overline{\text{RD}}$ (Read):** Active low signal. Indicates that the selected memory or I/O device is to be read and data is available on the data bus.
- **$\overline{\text{WR}}$ (Write):** Active low signal. Indicates that the data on the data bus is to be written into the selected memory or I/O location.
- **IO/$\overline{\text{M}}$:** Selects between I/O devices and Memory.
  - HIGH: I/O operation.
  - LOW: Memory operation.

# Status Signals

Table: Status Signals $S_1$ and $S_0$

| $\mathbf{IO/\overline{M}}$ | $S_1$ | $S_0$ | **Machine Cycle** |
|---|---|---|---|
| 0 | 0 | 0 | Halt |
| 0 | 0 | 1 | Memory Write |
| 0 | 1 | 0 | Memory Read |
| 1 | 0 | 1 | I/O Write |
| 1 | 1 | 0 | I/O Read |
| 0 | 1 | 1 | Opcode Fetch |
| 1 | 1 | 1 | Interrupt Acknowledge |

## Purpose

$S_1$ and $S_0$ identify the type of machine cycle currently being executed by the microprocessor.

# Power Supply and Clock Signals

## Power Supply

- $V_{CC}$: +5V power supply.
- $V_{SS}$: Ground Reference.

## Clock Signals

- $X_1, X_2$: Terminals to connect a crystal, LC, or RC network to set the internal clock generator frequency. (Operating frequency is half the crystal frequency).
- **CLK OUT:** Clock Output. Can be used as the system clock for other devices.

# Interrupts and Serial I/O Signals

## Interrupt Signals

- The 8085 has five hardware interrupt pins, ordered by priority:
  1. **TRAP** (Highest priority, Non-maskable)
  2. **RST 7.5**
  3. **RST 6.5**
  4. **RST 5.5**
  5. **INTR** (Lowest priority)
- **$\overline{\text{INTA}}$ (Interrupt Acknowledge):** Used to acknowledge an INTR interrupt.

## Serial I/O

- **SID (Serial Input Data):** Used to accept single-bit data from external devices.
- **SOD (Serial Output Data):** Used to send single-bit data to external devices.

# Other Externally Initiated Signals

- **READY:** Used to synchronize slower peripherals with the microprocessor. If LOW, the CPU enters wait states.
- **HOLD:** Indicates that another master (e.g., DMA controller) is requesting the use of the address and data buses.
- **HLDA (Hold Acknowledge):** Output signal acknowledging the HOLD request. The CPU relinquishes the buses.
- **$\overline{\text{RESET IN}}$:** Resets the program counter to zero and resets the interrupt enable and HLDA flip-flops.
- **RESET OUT:** Indicates that the CPU is being reset. Can be used to reset other devices in the system.

# Introduction to 8085 Architecture

## Overview

The 8085 is an 8-bit general-purpose microprocessor capable of addressing 64 KB of memory. Its internal architecture determines how it processes data and communicates with memory and I/O devices.

- Consists of various functional blocks working together.
- Key components include the ALU, Timing and Control Unit, Registers, and Interrupt Control.
- Operates on an 8-bit data bus and a 16-bit address bus.
- Uses a +5V single power supply.

# 8085 Block Diagram Overview

## Main Functional Units

The internal architecture of the 8085 can be divided into the following blocks:

1. **Arithmetic Logic Unit (ALU)**: Performs math and logic operations.
2. **Register Array**: Stores temporary data during execution.
3. **Instruction Register & Decoder**: Fetches and interprets instructions.
4. **Timing and Control Unit**: Generates signals to coordinate operations.
5. **Interrupt Control**: Handles external interrupt requests.
6. **Serial I/O Control**: Manages serial data communication.

# Arithmetic Logic Unit (ALU)

## Functionality

The ALU performs computing functions. It includes the logic circuitry to perform arithmetic and logical operations on 8-bit data.

- **Operations**: Addition, subtraction, logical AND, OR, EXCLUSIVE-OR, increment, decrement, and shift.
- **Inputs**: Receives data from the Accumulator and a temporary register.
- **Outputs**: Results are typically stored back into the Accumulator.
- **Flags**: The status of the result affects the Flag Register.

# Registers in 8085

## General Purpose Registers

The 8085 has six 8-bit general-purpose registers to store 8-bit data during program execution.

- **Registers**: B, C, D, E, H, and L.
- Can be used as individual 8-bit registers.
- **Register Pairs**: Can be combined as BC, DE, and HL to perform 16-bit operations.
- **HL Pair**: Acts as a default memory pointer for many instructions (M pointer).

# Special Purpose Registers

## Accumulator (A)

An 8-bit register that is part of the ALU. Used to store one of the operands and the result of operations.

- **Program Counter (PC)**: A 16-bit register that holds the memory address of the next instruction to be executed.
- **Stack Pointer (SP)**: A 16-bit register pointing to the top of the stack memory (used for subroutines and interrupts).
- **Instruction Register (IR)**: Holds the opcode of the instruction currently being decoded and executed.

# Flag Register

## Status Flags

The ALU includes five flip-flops called flags that are set or reset based on data conditions after an arithmetic or logical operation.

| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|
| S | Z | X | AC | X | P | X | CY |

- **S (Sign)**: Set if the result is negative (MSB = 1).
- **Z (Zero)**: Set if the ALU result is 0.
- **AC (Auxiliary Carry)**: Carry from bit 3 to 4 (used in BCD math).
- **P (Parity)**: Set if the result has an even number of 1s.
- **CY (Carry)**: Set if there is a carry out of the MSB.

# Timing and Control Unit

## The "Brain" of the Microprocessor

This unit coordinates the activities of all other blocks by generating the necessary timing and control signals.

- Communicates with memory and peripherals.
- **Clock Signals**: Generates internal clock (CLK) and external clock (CLK OUT).
- **Control Signals**: Generates read ($\overline{RD}$), write ($\overline{WR}$), and ALE (Address Latch Enable).
- **Status Signals**: Generates IO/$\overline{M}$, S0, and S1 to indicate the type of machine cycle in progress.

# Interrupt Control & Serial I/O

## Interrupt Control

Handles external requests that interrupt the normal execution sequence.

- 8085 has 5 hardware interrupts: TRAP, RST 7.5, RST 6.5, RST 5.5, and INTR.
- Prioritizes incoming interrupts and transfers control to corresponding service routines.

## Serial I/O Control

Manages serial data transfer.

- Features two lines for serial communication: SID (Serial Input Data) and SOD (Serial Output Data).

# Comparison: Microprocessor vs Microcontroller

## What is a Microprocessor?

- A silicon chip that contains a CPU (ALU, Registers, Control Unit).
- Lacks internal memory (RAM, ROM) and peripheral interfaces on the chip itself.
- Examples: Intel 8085, 8086, Core i7, ARM Cortex-A series.

## What is a Microcontroller?

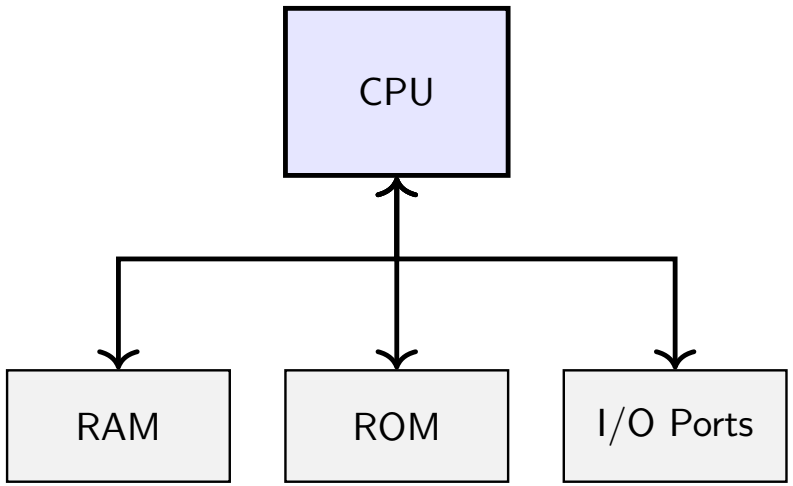
- Often referred to as a complete "computer on a chip."
- Integrates CPU, RAM, ROM, I/O ports, and timers on a single Integrated Circuit (IC).
- Examples: 8051, ATmega328 (used in Arduino), PIC microcontrollers.

# Key Differences

| Feature | Microprocessor (MPU) | Microcontroller (MCU) |
|---|---|---|
| **Integration** | Only CPU on chip. | CPU, Memory, Timers, I/O on chip. |
| **Architecture** | Typically Von Neumann architecture. | Typically Harvard architecture. |
| **Flexibility** | Highly flexible; external memory scales. | Fixed internal memory and I/O. |
| **Cost** | High (requires external components). | Low (everything is integrated). |
| **Power** | High power consumption. | Low power consumption (battery friendly). |
| **Performance** | High processing speed (GHz range). | Lower processing speed (MHz range). |

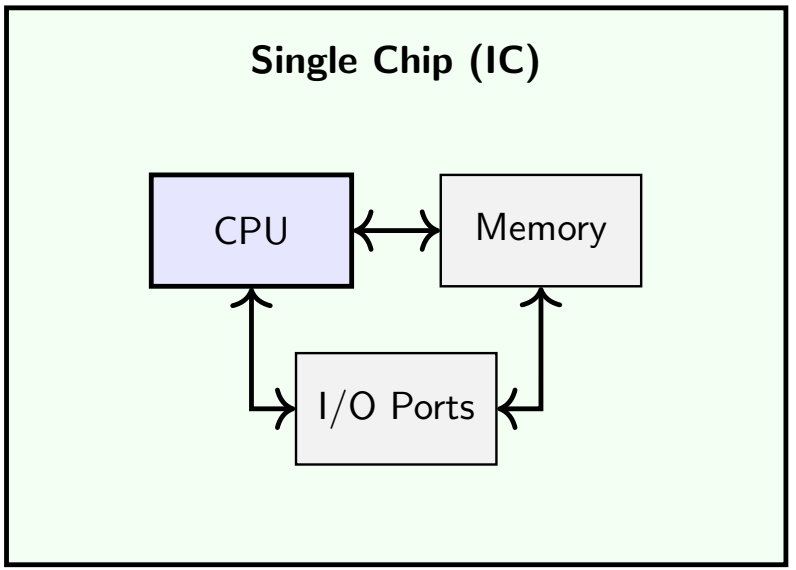
# System Block Diagrams

## Microprocessor System




Relies on external components connected via buses.

## Microcontroller




Everything packaged into one single chip.

# Instruction Set & Programming

## Microprocessor Instructions

- Instructions are designed to move data between external memory and the CPU.
- Processing large amounts of data (Data intensive).
- Large number of instructions addressing varied memory locations.

## Microcontroller Instructions

- Contains bit-level manipulation instructions.
- Highly optimized for controlling I/O pins (Control intensive).
- Easier to program for specific real-time tasks.

# Applications & Use Cases

## Microprocessor Applications (General Purpose)

- Systems requiring high computational power and flexibility.
- Personal computers, laptops, and enterprise servers.
- High-performance gaming consoles and smartphones.
- Complex tasks like video editing, AI processing, and big data.

## Microcontroller Applications (Embedded Systems)

- Application-specific devices performing dedicated tasks.
- Home appliances: Washing machines, microwaves, refrigerators.
- Automotive: Engine control units (ECU), ABS, airbags.
- Consumer electronics: Remote controls, smart watches, IoT devices.

# Introduction to Microcontrollers

## What is a Microcontroller?

A microcontroller (MCU) is a compact integrated circuit designed to govern a specific operation in an embedded system. It acts as a small computer on a single metal-oxide-semiconductor (MOS) integrated circuit chip.

**Core Differences from Microprocessors:**

- Integrates CPU, memory (RAM/ROM), and I/O peripherals on a single chip.
- Designed for specific, dedicated tasks rather than general-purpose computing.
- Optimized for lower power consumption, size, and cost.

# Common Features of Microcontrollers

Microcontrollers typically share several fundamental built-in features that allow them to function as standalone systems:

- On-chip Oscillator / Clock
- Program Memory (ROM/Flash)
- Data Memory (RAM)
- Input/Output (I/O) Ports
- Reset Circuitry
- Special Function Registers (SFRs)
- Timers and Counters
- Interrupt Controller

# On-chip Oscillator

## Clock Generation

The on-chip oscillator provides the fundamental timing and synchronization for all internal operations of the microcontroller.

- Typically requires an external crystal resonator or ceramic resonator connected to specific pins (e.g., XTAL1 and XTAL2 in 8051).
- Some modern microcontrollers have fully internal RC oscillators requiring no external components.
- The oscillator frequency determines the instruction execution speed.
- Example: In an 8051 operating at 12 MHz, one machine cycle takes 1 microsecond (12 clock cycles per machine cycle).

# Program Memory and Data Memory

Microcontrollers typically utilize a Harvard architecture, separating program and data memory.

## Program Memory (ROM)

- Non-volatile memory (Flash, EEPROM, or OTP ROM).
- Stores the executable code (firmware) and constant data.
- Retains contents even when power is removed.

## Data Memory (RAM)

- Volatile memory (SRAM).
- Used for storing temporary data, variables, and the stack during execution.
- Loses contents when power is turned off.

# I/O Ports

## Purpose of I/O Ports

Input/Output ports allow the microcontroller to interact with the external world, such as reading sensors or driving displays and actuators.

- Usually organized into 8-bit groups (e.g., Port 0, Port 1, Port 2, Port 3).
- Pins can often be configured individually as either inputs or outputs.
- **Alternate Functions:** Many I/O pins are multiplexed to serve dual purposes (e.g., standard I/O, UART TX/RX, Interrupt inputs, Timer inputs).
- Ports are typically controlled via Special Function Registers (SFRs).

# Reset Operation

## System Reset

A reset brings the microcontroller to a known initial state, ensuring predictable execution from the beginning of the program.

- Initializes all internal registers to their default startup values.
- Clears the Program Counter (PC), forcing execution to start from memory address `0x0000`.
- **Types of Reset:**
  - Power-on Reset (POR)
  - Manual Reset (via a dedicated Reset pin)
  - Watchdog Timer Reset (if the system hangs)
  - Brown-out Reset (if supply voltage drops below a threshold)

# Special Function Registers (SFRs)

## What are SFRs?

SFRs are specific memory locations dedicated to controlling and monitoring the microcontroller's hardware peripherals and core functions.

- They act as the control panel for the MCU.
- Examples of SFRs include:
  - Accumulator (A) and B Register for arithmetic.
  - Program Status Word (PSW) for flags (Carry, Zero, etc.).
  - Port latches (P0, P1, P2, P3).
  - Timer registers (TH0, TL0, TMOD, TCON).
  - Serial control registers (SCON, SBUF).
- Writing to or reading from these registers directly alters the hardware behavior.

# Timers and Counters

## Timers vs. Counters

Hardware modules that can count clock pulses. They operate independently of the CPU, saving processing power.

**Timer Mode:**

- Counts internal machine cycles.
- Used for generating precise time delays.
- Used for baud rate generation.

**Counter Mode:**

- Counts external pulses on a specific pin.
- Used for event counting (e.g., counting products on a conveyor belt).
- Used for frequency measurement.

# Interrupts

## Interrupt Mechanism

An interrupt is a mechanism by which an external or internal event can temporarily halt the CPU's current task to execute a high-priority routine.

- **Interrupt Service Routine (ISR):** The specific function executed when an interrupt occurs.
- Once the ISR finishes, the CPU resumes its previous task exactly where it left off.
- **Common Sources:**
  - External hardware pins (External Interrupts).
  - Timers overflowing.
  - Serial communication (byte received/transmitted).
- Interrupts provide a much more efficient alternative to polling.

# Summary of Microcontroller Features

| **Feature** | **Primary Function** |
|---|---|
| Oscillator | Provides system clock and synchronization |
| Program Memory | Stores executable code (Non-volatile) |
| Data Memory | Stores variables and stack (Volatile) |
| I/O Ports | Interfacing with external devices |
| Reset | Initializes system to a known state |
| SFRs | Controls and configures hardware peripherals |
| Timers/Counters | Generates delays and counts external events |
| Interrupts | Handles asynchronous high-priority events efficiently |

# Arithmetic Logic Unit (ALU)

## Overview

The ALU performs arithmetic and logical operations in the 8051 Microcontroller.

- 8-bit processing unit.
- Performs arithmetic functions: Addition, Subtraction, Multiplication, and Division.
- Logical functions: AND, OR, XOR, NOT, Rotate, and Clear.
- Works closely with Accumulator (Register A), Register B, and PSW.
- Automatically updates the Program Status Word (PSW) flags based on operation results.

# Program Counter (PC)

## Functionality

The Program Counter is a crucial 16-bit register that tracks program execution.

- **Size:** 16 bits wide, capable of addressing up to 64 KB of program memory ($2^{16} = 65536$ locations).
- **Role:** Holds the address of the next instruction to be executed from ROM.
- **Operation:** Automatically increments after fetching an instruction.
- Unlike other registers, the PC does not have an internal address and is inaccessible directly by the programmer.
- Modified by jump, call, and return instructions.

# Data Pointer (DPTR)

## What is DPTR?

The Data Pointer is a 16-bit user-accessible register used for addressing external data and code memory.

- **Structure:** Composed of two 8-bit registers:
  - **DPH** (Data Pointer High)
  - **DPL** (Data Pointer Low)
- **Primary Use:** Used as a base register in indirect jumps and block data transfers.
- Can be accessed as a single 16-bit register or two separate 8-bit registers.
- Essential for interfacing with external RAM and ROM components.

# Program Status Word (PSW)

## Flag Register

An 8-bit register containing status flags that reflect the current state of the ALU.

- **CY (Carry Flag):** Set during addition/subtraction if there's a carry/borrow out of the MSB.
- **AC (Auxiliary Carry):** Used for BCD arithmetic; set if carry from bit 3 to bit 4.
- **F0 (Flag 0):** General-purpose user flag.
- **RS1, RS0 (Register Bank Select):** Used to select one of the four register banks.
- **OV (Overflow Flag):** Indicates overflow in signed arithmetic operations.
- **P (Parity Flag):** Set/cleared by hardware to ensure an even number of 1s in the Accumulator.

# Internal RAM

## On-chip Data Memory

The standard 8051 provides 128 bytes of internal RAM for data storage.

- **Address Range:** 00H to 7FH.
- **Organization:**
  - **00H - 1FH:** 32 bytes allocated for four register banks (Bank 0 to 3).
  - **20H - 2FH:** 16 bytes (128 bits) of bit-addressable memory space.
  - **30H - 7FH:** 80 bytes of general-purpose Scratchpad RAM.
- Fast access compared to external RAM.
- Used for stack operations, temporary variables, and general data.

# Internal ROM

## On-chip Program Memory

Non-volatile memory used to store the application code (firmware).

- **Capacity:** Standard 8051 contains 4 KB of internal ROM.
- **Address Range:** 0000H to 0FFFH.
- **EA Pin Dependency:** If the External Access ($\overline{EA}$) pin is tied high, execution begins from Internal ROM. If low, execution relies solely on External ROM.
- Stores instruction opcodes, look-up tables, and constants.
- Variants like 8751 use EPROM, and modern versions like AT89C51 use Flash memory for easy reprogramming.

# Special Function Registers (SFRs)

## Control and Status

SFRs are used to control the operation of timers, serial ports, and I/O ports.

- Located in the upper 128 bytes of direct memory (Addresses 80H to FFH).
- **Key SFRs include:**
  - Accumulator (A) and Register B.
  - PSW (Program Status Word).
  - SP (Stack Pointer).
  - Port registers (P0, P1, P2, P3).
  - Timer/Counter registers (TCON, TMOD, TL0, TH0, etc.).
- Only specific addresses within this range are implemented; unused addresses are reserved.

# General Purpose Registers (GPRs)

## Register Banks

Used for fast and temporary data storage and manipulation during program execution.

- 32 bytes of internal RAM are dedicated to 4 register banks (Bank 0 to Bank 3).
- Each bank contains 8 registers named **R0 through R7**.
- **Selection:** Active bank is selected using the RS1 and RS0 bits in the PSW.
- **Default:** On power-up, Bank 0 (addresses 00H-07H) is selected by default.
- Accessing R0-R7 is faster and requires fewer instruction bytes than accessing direct memory addresses.

# Introduction to 8051 Internal Blocks

## Overview

The 8051 microcontroller integrates several essential blocks on a single chip, forming a complete computational system.

- **Processing:** Arithmetic Logic Unit (ALU)
- **Control & Pointers:** Program Counter (PC), Data Pointer (DPTR), Program Status Word (PSW)
- **Memory:** Internal RAM, Internal ROM
- **Registers:** Special Function Registers (SFRs), General Purpose Registers

# Arithmetic Logic Unit (ALU)

## Functionality of the ALU

The ALU in the 8051 is an 8-bit unit that performs arithmetic and logical operations.

- **Arithmetic Operations:** Addition, Subtraction, Multiplication, Division.
- **Logical Operations:** AND, OR, XOR, NOT, Rotate, Clear, Swap.
- The ALU primarily operates on data in the Accumulator (Register A) and another register (often Register B for math operations).
- It updates flags in the Program Status Word (PSW) based on the result of operations (e.g., carry, zero, overflow).

# Program Counter (PC)

## What is the PC?

The Program Counter is a 16-bit register that holds the address of the next instruction to be executed.

- Being 16 bits wide, it can access up to $2^{16} = 64$ KB of program memory (ROM).
- Upon reset, the PC is initialized to `0000H`, meaning execution starts from memory location `0000H`.
- The PC is automatically incremented after an instruction is fetched.
- It is the only register that does not have an internal address and cannot be accessed directly by instructions like `MOV`.

# Data Pointer (DPTR)

## Data Pointer (DPTR) Register

The DPTR is a 16-bit register used to hold memory addresses for accessing external data and code memory.

- It is the only user-accessible 16-bit register in the 8051.
- It can be used as a single 16-bit register (DPTR) or as two separate 8-bit registers:
  - **DPH:** Data Pointer High byte.
  - **DPL:** Data Pointer Low byte.
- Mainly used with instructions like `MOVX` (to access external RAM) and `MOVC` (to look up tables in ROM).

# Program Status Word (PSW)

## PSW Register

The Program Status Word is an 8-bit register (also known as the flag register) that reflects the current status of the CPU.

- It contains both status bits (flags) and control bits.
- **Key Flags:**
  - **CY (Carry Flag):** Set if an operation results in a carry out of the MSB.
  - **AC (Auxiliary Carry):** Set for BCD arithmetic carry from bit 3 to bit 4.
  - **F0 (Flag 0):** General-purpose user flag.
  - **RS1, RS0 (Register Bank Select):** Selects one of four working register banks.
  - **OV (Overflow Flag):** Set if a signed arithmetic operation overflows.
  - **P (Parity Flag):** Reflects the parity (number of 1s) in the Accumulator (1 = odd parity).

# Internal Memory: RAM

## 8051 Internal RAM

The standard 8051 has 128 bytes of internal RAM, addressed from 00H to 7FH.

- The RAM is divided into three distinct areas:
  1. **Working Registers (00H-1FH):** 32 bytes divided into 4 banks of 8 registers each (R0-R7).
  2. **Bit-Addressable Area (20H-2FH):** 16 bytes, providing 128 individually addressable bits (00H-7FH).
  3. **General Purpose RAM (30H-7FH):** 80 bytes used for data storage and the stack.

# Internal Memory: ROM

## 8051 Internal ROM

The standard 8051 includes 4 KB of on-chip Program Memory (ROM).

- Used for storing the application program (code) and constant data (like lookup tables).
- Address range: `0000H` to `0FFFH`.
- The `EA` (External Access) pin determines whether the processor executes from internal or external ROM:
  - `EA` = 1 (High): Executes from internal ROM up to `0FFFH`, then jumps to external ROM if addresses go beyond.
  - `EA` = 0 (Low): Ignores internal ROM and fetches all instructions from external memory.

# Special Function Registers (SFRs)

## What are SFRs?

SFRs are 8-bit registers used to control and configure the operation of the 8051's internal hardware peripherals.

- They reside in the upper 128 bytes of RAM memory space (address `80H` to `FFH`).
- **Note:** While they share addresses with upper RAM in the 8052, standard 8051 can only access them via direct addressing.
- **Important SFRs include:**
  - **ACC (Accumulator):** Primary register for math and logic.
  - **B Register:** Used for multiplication and division.
  - **Port Latches (P0, P1, P2, P3):** Control the I/O pins.
  - **TCON, TMOD:** Timer control and mode registers.
  - **SCON, SBUF:** Serial port control and data buffer.

# General Purpose Registers (R0 - R7)

## Working Register Banks

The lower 32 bytes of internal RAM are divided into 4 banks (Bank 0 to Bank 3), each containing 8 registers named R0 through R7.

- Only one bank can be active at a time, selected by the `RS1` and `RS0` bits in the PSW.
- **Default Bank:** Upon reset, Bank 0 is selected (addresses `00H` to `07H`).
- **Usage:** These registers are used for temporary data storage and efficient data manipulation during execution.
- Fast context switching during interrupts can be achieved by simply changing the active register bank in the PSW.

# 8051 Microcontroller Pinout Overview

- The 8051 microcontroller is generally available in a **40-pin Dual Inline Package (DIP)**.
- Requires a single +5V power supply.
- Offers 32 bidirectional I/O pins organized as four 8-bit ports (Port 0 to Port 3).
- Contains specialized pins for control signals, clock, and power.

## Categorization of Pins

- Power and Ground (2 pins)
- Clock Source (2 pins)
- Reset (1 pin)
- Control Signals (3 pins)
- I/O Ports (32 pins)

# Power Supply and Clock Pins

## Power Pins

- **Vcc (Pin 40):** Main power supply pin. Requires a +5V DC source.
- **GND (Pin 20):** Ground reference pin. Connects to 0V.

## Oscillator Pins

- **XTAL1 (Pin 19) & XTAL2 (Pin 18):** Connect to an external quartz crystal oscillator.
- Used to generate the internal clock frequency.
- Typically, an 11.0592 MHz or 12 MHz crystal is connected along with two stabilizing capacitors (usually 33 pF).
- Alternatively, an external clock source can be applied to XTAL1 while leaving XTAL2 unconnected.

# Reset Pin (RST)

## RST (Pin 9)

- Active high reset input pin.
- Used to bring the microcontroller to a known default state.
- To reset the 8051, a high pulse (+5V) must be applied to this pin for at least two machine cycles (24 oscillator periods).
- Upon reset, most registers (including Program Counter) are cleared, except for the Stack Pointer, which is initialized to 07H.
- Usually connected to a power-on reset circuit (a resistor-capacitor combination) and a manual reset button.

# Memory Control Pins

- **$\overline{\text{EA}}$ / VPP (External Access - Pin 31):**
  - Active low pin. Dictates whether the CPU fetches code from internal or external ROM.
  - If tied High (Vcc): Executes from Internal ROM (up to 4KB).
  - If tied Low (GND): Forces execution entirely from External ROM.
  - *VPP:* Receives 12V programming pulse during flash programming.
- **ALE / $\overline{\text{PROG}}$ (Address Latch Enable - Pin 30):**
  - Used to demultiplex the lower address and data bus (AD0-AD7) on Port 0 during external memory access.
  - Pulses high twice every machine cycle.
  - *$\overline{\text{PROG}}$:* Acts as a programming pulse input when writing to internal flash memory.
- **$\overline{\text{PSEN}}$ (Program Store Enable - Pin 29):**
  - Active low output signal used to read external program memory (ROM).
  - Connected to the $\overline{\text{OE}}$ (Output Enable) pin of external ROM.

# Input/Output Ports Overview

## General Properties of Ports

- Four 8-bit ports: **Port 0, Port 1, Port 2, and Port 3**.
- All 32 pins are bidirectional (can act as input or output).
- UPON RESET, all ports are configured as *Inputs* (all bits set to '1').
- To use a port pin as an input, a '1' must be written to its corresponding bit in the Special Function Register (SFR).

# Port 0 and Port 1

## Port 0 (Pins 32-39)

- True bidirectional I/O port (Open-drain).
- Requires **external pull-up resistors** (typically 10kΩ) when used as simple I/O.
- **Alternate Function:** Multiplexed low-order address and data bus (**AD0** - **AD7**) during external memory access (no pull-ups needed here).

## Port 1 (Pins 1-8)

- Standard bidirectional I/O port with **internal pull-up resistors**.
- Dedicated entirely to basic I/O operations (does not have alternate functions in the standard 8051, unlike other ports).
- Some later variants use P1.0 and P1.1 for Timers, but strictly standard 8051 uses them just for I/O.

# Port 2 and Port 3

## Port 2 (Pins 21-28)

- Bidirectional I/O port with internal pull-up resistors.
- **Alternate Function:** Outputs the high-order address byte (**A8 - A15**) during external memory access.

## Port 3 (Pins 10-17)

- Bidirectional I/O port with internal pull-up resistors.
- **Crucial Alternate Functions:** Each pin in Port 3 is multiplexed with special control signals (serial communication, interrupts, timers, and memory control).

# Port 3 Alternate Functions

| Pin | Name | Alternate Function |
|---|---|---|
| P3.0 (Pin 10) | RXD | Serial Input Data (UART) |
| P3.1 (Pin 11) | TXD | Serial Output Data (UART) |
| P3.2 (Pin 12) | $\overline{\text{INT0}}$ | External Hardware Interrupt 0 |
| P3.3 (Pin 13) | $\overline{\text{INT1}}$ | External Hardware Interrupt 1 |
| P3.4 (Pin 14) | T0 | Timer 0 External Input |
| P3.5 (Pin 15) | T1 | Timer 1 External Input |
| P3.6 (Pin 16) | $\overline{\text{WR}}$ | External Data Memory Write Strobe |
| P3.7 (Pin 17) | $\overline{\text{RD}}$ | External Data Memory Read Strobe |

# Overview of 8051 Memory Organization

- The 8051 microcontroller employs a **Harvard Architecture**.
- This implies separate physical and logical memory spaces for **Data** and **Code**.
- **Internal ROM (Program Memory):** Stores instructions and constant data.
- **Internal RAM (Data Memory):** Stores variables, temporary data, and system registers.
- Both memory spaces can be expanded using external memory chips.

# Internal ROM (Program Memory)

## Characteristics

- Standard 8051 features **4 KB** of internal ROM.
- The address range for this internal ROM is **0000H to 0FFFH**.
- The Program Counter (PC) is 16-bit, allowing up to **64 KB** of addressable program memory (internal + external).

## Execution Control ($\overline{EA}$ Pin)

- If $\overline{EA}$ (External Access) is connected to $V_{CC}$ (High): Executes from internal ROM (0000H to 0FFFH), then fetches from external ROM (1000H to FFFFH).
- If $\overline{EA}$ is connected to GND (Low): Bypasses internal ROM and fetches exclusively from external memory (0000H to FFFFH).

# Internal RAM (Data Memory) Structure

- The basic 8051 has **128 bytes** of internal RAM.
- Address range: **00H to 7FH**.
- This space is highly organized and serves multiple specific functions.

## Division of 128-Byte Internal RAM

1. **Register Banks** (00H – 1FH): 32 bytes
2. **Bit-Addressable Area** (20H – 2FH): 16 bytes
3. **General Purpose Scratchpad RAM** (30H – 7FH): 80 bytes

# 1. Register Banks (00H – 1FH)

- The first 32 bytes are divided into **four register banks** (Bank 0 to Bank 3).
- Each bank contains 8 registers, denoted **R0 through R7**.
- Only one register bank can be active at a time.
- Bank selection is controlled by bits **RS0 and RS1** in the Program Status Word (PSW).

| RS1 | RS0 | Active Bank | Address Range |
|---|---|---|---|
| 0 | 0 | Bank 0 (Default) | 00H – 07H |
| 0 | 1 | Bank 1 | 08H – 0FH |
| 1 | 0 | Bank 2 | 10H – 17H |
| 1 | 1 | Bank 3 | 18H – 1FH |

# 2. Bit-Addressable RAM (20H – 2FH)

- This 16-byte area consists of addresses **20H to 2FH**.
- Total bits available = 16 bytes $\times$ 8 bits/byte = **128 bits**.
- These 128 individual bits can be directly addressed from **00H to 7FH**.
- **Usage:** Highly efficient for flags, Boolean variable storage, and bit-oriented control operations.
- Instructions like `SETB bit`, `CLR bit`, and `JB bit, label` are commonly used here.

# 3. Scratchpad RAM (30H – 7FH)

- Consists of **80 bytes** from address **30H to 7FH**.
- Used for general data storage and variable management during program execution.
- Also utilized as the default area for the **Stack**.

## The Stack in 8051

- The Stack Pointer (SP) is initialized to **07H** on reset.
- Pushing data increments the SP first (`PUSH` operations typically start storing data from 08H onwards if not relocated).
- It is standard practice to initialize the SP to **2FH** or higher to avoid overwriting register banks and bit-addressable memory.

# Special Function Registers (SFRs)

- Located in the memory space from **80H to FFH**.
- Accessed only via **Direct Addressing**.
- **Purpose:** Control and configuration of microcontroller peripherals (Timers, Serial Port, I/O ports, Interrupts).

## Key SFRs

- **Accumulator (A):** Math and logic operations (E0H).
- **B Register (B):** Multiplication and division (F0H).
- **PSW (Program Status Word):** Flags and bank selection (D0H).
- **P0, P1, P2, P3:** Port latches for I/O operations.

*Note: SFR addresses ending in 0 or 8 (e.g., 80H, 90H, A8H) are also bit-addressable.*

# Introduction to 8051 Memory Architecture

## Harvard Architecture

The 8051 microcontroller uses the **Harvard Architecture**, meaning it has separate memory spaces for Program (Code) and Data.

- **Program Memory (ROM):** Used to store instructions and permanent data (e.g., lookup tables).
- **Data Memory (RAM):** Used for temporary data storage, variables, and stack operations during program execution.

## Key Benefit

Separate buses for code and data allow the CPU to fetch instructions and read/write data simultaneously, improving execution speed.

# Internal ROM Organization

## Overview of Program Memory

Standard 8051 has **4 KB of on-chip ROM**. The program counter (PC) is 16-bit wide, allowing the 8051 to address up to 64 KB of program memory ($0000H$ to $FFFFH$).

- **Internal ROM Range:** Addresses $0000H$ to $0FFFH$.
- **External ROM Range:** If external memory is used, it extends from $1000H$ to $FFFFH$ (assuming internal ROM is used).

## The $\overline{EA}$ (External Access) Pin

- If $\overline{EA} = 1$ ($V_{cc}$): CPU first fetches from Internal ROM ($0000H - 0FFFH$). If PC exceeds $0FFFH$, it automatically fetches from External ROM.
- If $\overline{EA} = 0$ ($GND$): CPU fetches instructions entirely from External ROM ($0000H - FFFFH$), ignoring internal ROM.

# Internal RAM Overview

## On-Chip Data Memory

The standard 8051 features **256 bytes of internal RAM**, which is divided into two distinct regions.

- **Lower 128 Bytes (00*H* to 7*FH*):** Directly and indirectly addressable. This is the true internal RAM used by the programmer for data storage.
- **Upper 128 Bytes (80*H* to *FFH*):** Directly addressable only. This space is reserved for **Special Function Registers (SFRs)**.

## Addressing Note

Indirect addressing of addresses 80*H* to *FFH* access external RAM (or extra internal RAM on enhanced 8052), not the SFRs.

# Detailed Map of Lower 128 Bytes RAM

The lower 128 bytes of RAM ($00H$ to $7FH$) are highly structured and serve specific purposes:

| Address Range | Size | Purpose |
|---|---|---|
| $00H - 1FH$ | 32 Bytes | Register Banks (Bank 0 to Bank 3) |
| $20H - 2FH$ | 16 Bytes | Bit-Addressable RAM (128 bits) |
| $30H - 7FH$ | 80 Bytes | General Purpose / Scratchpad RAM |

- **Stack memory** usually resides in the General Purpose RAM area, starting by default at $07H$ (though typically reinitialized to start above the bit-addressable region).

# Register Banks ($00H$ to $1FH$)

- The lowest 32 bytes are grouped into **4 Register Banks** (Bank 0, 1, 2, and 3).
- Each bank consists of **8 registers** named $R0$ through $R7$.
- Only one bank is active at any given time.

| **Register Bank** | **Address Range** |
|---|---|
| Bank 0 (Default) | $00H - 07H$ |
| Bank 1 | $08H - 0FH$ |
| Bank 2 | $10H - 17H$ |
| Bank 3 | $18H - 1FH$ |

## Bank Selection

The active bank is selected using bits RS1 and RS0 in the **Program Status Word (PSW)** register. By default, upon reset, Bank 0 is selected.

# Bit-Addressable RAM ($20H$ to $2FH$)

## Bit-Level Control

The 8051 provides powerful bit-manipulation capabilities. 16 bytes of internal RAM are designated as bit-addressable space.

- **Total Bits:** 16 bytes $\times$ 8 bits/byte $=$ 128 bits.
- **Bit Addresses:** $00H$ to $7FH$.
- **Byte Addresses:** $20H$ to $2FH$.

## Usage

Software flags and boolean variables are typically stored here. Instructions like `SETB`, `CLR`, and `JB` operate directly on these bit addresses. For example, bit address $00H$ corresponds to the LSB of byte address $20H$.

# General Purpose Scratchpad RAM ($30H$ to $7FH$)

## Scratchpad Area

The remaining 80 bytes of the lower internal RAM are used for general data storage and are byte-addressable only.

- **Address Range:** $30H$ to $7FH$.
- **Usage:** Storing application variables, intermediate calculation results, and data arrays.
- **Stack Location:** The default Stack Pointer (SP) value upon reset is $07H$. When a `PUSH` or `CALL` occurs, the SP increments to $08H$ (Bank 1). To preserve register banks, programmers typically initialize SP to a higher address, usually $2FH$ or $30H$.

# Special Function Registers (SFRs) (80$H$ to $FFH$)

## What are SFRs?

SFRs are registers that control the operation of the microcontroller's hardware peripherals (Timers, Serial Port, I/O Ports) and CPU state.

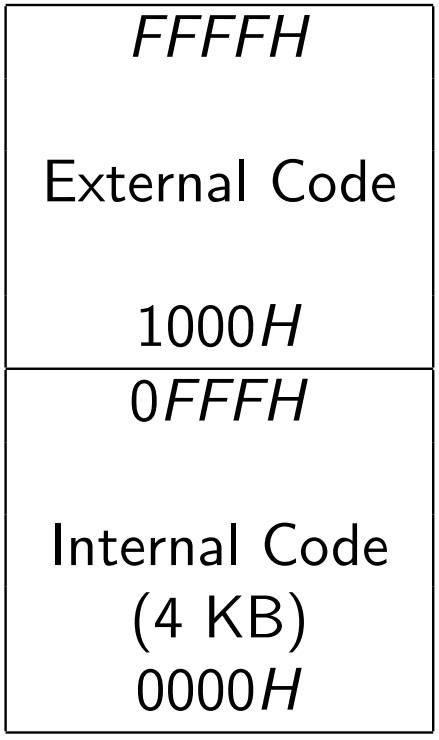
- **Location:** Upper 128 bytes of internal RAM space.
- **Access: Direct addressing only**.
- **Bit-Addressability:** SFRs whose addresses end in $0H$ or $8H$ (e.g., $80H, 88H, 90H$) are bit-addressable.

**Crucial SFRs:**

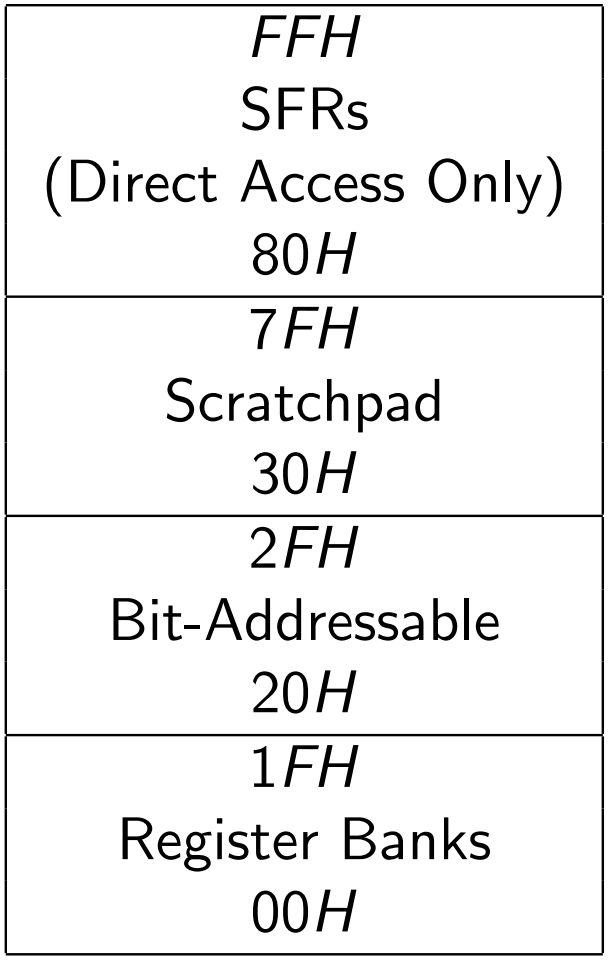
- **ACC ($E0H$):** Accumulator for arithmetic/logic ops.
- **B ($F0H$):** Used with ACC for multiply/divide.
- **PSW ($D0H$):** Program Status Word (flags).
- **SP ($81H$):** Stack Pointer.
- **DPTR:** Data Pointer (DPL at $82H$, DPH at $83H$).

# Visual Summary of Memory Architecture

**Program Memory (ROM)**




**Data Memory (RAM)**



# Introduction to the Stack in 8051

## What is a Stack?

- The stack is a section of RAM used by the CPU to store information temporarily.
- It operates on a **LIFO** (Last In, First Out) principle.
- In the 8051, the stack can be located anywhere in the internal RAM.
- It is widely used to save context (registers and return addresses) during subroutine calls and interrupts.

## Why do we need a Stack?

- To preserve data temporarily without needing extra explicit memory addresses.
- Essential for `CALL` instructions and interrupt handling to store the Program Counter (PC) safely.

# The Stack Pointer (SP)

## Definition and Role

- The **Stack Pointer (SP)** is an 8-bit register in the Special Function Register (SFR) memory space (Address: `81H`).
- It holds the internal RAM address that points to the top of the stack.
- On system reset, the SP is initialized to `07H`.

## Important Characteristics

- Because SP initializes to `07H`, the first data pushed onto the stack will be stored at `08H`.
- Locations `08H` to `1FH` normally belong to Register Banks 1, 2, and 3.
- If these banks are used by the program, the SP must be manually initialized to another area (e.g., `SP = 2FH` or higher) to avoid data corruption.

# Stack Operation: PUSH

## The PUSH Instruction

- **Syntax:** `PUSH <direct address>`
- **Function:** Stores the contents of the specified direct address onto the stack.

## Mechanism of PUSH

1. The SP is incremented by 1 (`SP = SP + 1`).
2. The content of the direct address is copied into the internal RAM location pointed to by the new SP value.

## Example

```
MOV SP, #2FH    ; Initialize SP to 2FH
MOV R6, #25H    ; Load 25H into R6
PUSH 06H        ; PUSH R6 onto stack (SP becomes 30H, RAM[30H] = 25H)
```

# Stack Operation: POP

## The POP Instruction

- **Syntax:** `POP <direct address>`
- **Function:** Retrieves the top value from the stack and places it into the specified direct address.

## Mechanism of POP

1. The data at the RAM location pointed to by the SP is copied into the specified direct address.
2. The SP is decremented by 1 (`SP = SP - 1`).

## Example

```
POP 03H          ; Pop top of stack into R3 (RAM[SP] → R3, then SP = SP - 1)
```

# Stack Dynamics: Subroutines and Interrupts

## Subroutine Calls (CALL/RET)

- When an `LCALL` or `ACALL` is executed, the 8051 automatically PUSHes the 16-bit Program Counter (PC) onto the stack (low byte first, then high byte).
- The SP is incremented by 2.
- When `RET` is executed, it POPs the top two bytes back into the PC and decrements the SP by 2.

## Stack Overflow and Underflow

- **Overflow:** Pushing too much data, exceeding the available RAM (e.g., beyond `7FH` in standard 8051). Can corrupt other data or SFRs.
- **Underflow:** Popping more times than pushed. Can lead to retrieving garbage data or crashing the program return sequence.

# Summary of Stack Memory Allocation

| Internal RAM Address | Typical Usage |
|---|---|
| 00H - 07H | Register Bank 0 (Default) |
| 08H - 1FH | Register Banks 1-3 (Default Stack Area) |
| 20H - 2FH | Bit-Addressable RAM |
| 30H - 7FH | General Purpose RAM (Preferred Stack Area) |

Table: 8051 Internal RAM Map and Stack Allocation

- **Best Practice:** Always set SP to 2FH at the start of your program if using multiple register banks.
- Example: `MOV SP, #2FH` (Stack starts from 30H).

# Introduction to Special Function Registers (SFRs)

- **What are SFRs?**
  Special Function Registers (SFRs) are hardware registers used by the 8051 microcontroller to control and monitor its operations.
- **Memory Location:**
  Located in the upper 128 bytes of internal RAM (address range `80H` to `FFH`).
- **Access Method:**
  SFRs can be accessed using direct addressing only. Some are also bit-addressable.
- **Purpose:**
  They control specific functions like timers, serial communication, interrupts, and I/O ports.

# Overview of 8051 SFR Memory Map

## Key Characteristics

- Only 21 SFRs are implemented in the standard 8051 architecture.
- The remaining locations between 80H and FFH are undefined and should not be used.
- SFRs ending with addresses 0H or 8H (e.g., 80H, 88H, 90H) are usually **bit-addressable**.

| Address | SFR Name | Bit Addressable? |
|---|---|---|
| E0H | Accumulator (ACC) | Yes |
| F0H | B Register | Yes |
| D0H | Program Status Word (PSW) | Yes |
| 81H | Stack Pointer (SP) | No |

# The Accumulator (ACC) and B Register

## Accumulator (A)

- Address: `E0H`
- Highly utilized 8-bit register.
- Used for all arithmetic and logic operations.
- Default destination for many instructions.
- Data transfer between 8051 and external memory goes through A.

## B Register (B)

- Address: `F0H`
- Primarily used in multiply (`MUL`) and divide (`DIV`) operations.
- Example: `MUL AB` multiplies A and B, storing the 16-bit result in A (low byte) and B (high byte).
- Can be used as a general-purpose register otherwise.

# Program Status Word (PSW)

- Address: `D0H` (Bit-addressable)
- Acts as the flag register containing status bits reflecting the current state of the CPU.

| **CY** (Bit 7) | **AC** (Bit 6) | **F0** (Bit 5) | **RS1** (Bit 4) | **RS0** (Bit 3) | **OV** (Bit 2) | - (Bit 1) | **P** (Bit 0) |
|---|---|---|---|---|---|---|---|
| Carry Flag | Aux Carry | User Flag 0 | Reg Bank 1 | Reg Bank 0 | Overflow | Reserved | Parity Flag |

## Register Bank Selection

`RS1` and `RS0` bits determine the active register bank (0-3) in the lower RAM (`00H-1FH`).

# Data Pointer (DPTR) and Program Counter (PC)

- **Data Pointer (DPTR)**
  - A 16-bit register composed of two 8-bit registers: `DPH` (`83H`) and `DPL` (`82H`).
  - Primarily used to hold 16-bit addresses for external data memory or external code memory access.
- **Program Counter (PC)**
  - A 16-bit register, but **NOT** considered an SFR (does not have an SFR address).
  - Holds the address of the next instruction to be executed.
  - Automatically increments after fetching an instruction.

# Stack Pointer (SP)

## Function

The Stack Pointer is an 8-bit register that indicates the current top of the stack in internal RAM.

- **Address:** `81H`
- **Default Value:** Upon reset, SP is initialized to `07H`.
- **Operation:**
  The stack grows upwards in memory. When a `PUSH` or `CALL` instruction is executed, SP is incremented first, and then data is stored.
- **Usage note:**
  Since `07H` overlaps with Register Bank 1, it's common practice to initialize SP to a higher address (like `2FH` or higher) to avoid overwriting general-purpose registers.

# I/O Port Registers (P0, P1, P2, P3)

- The 8051 has four 8-bit bidirectional I/O ports, each controlled by a corresponding SFR.
- **Addresses (All bit-addressable):**
  - `P0 : 80H`
  - `P1 : 90H`
  - `P2 : A0H`
  - `P3 : B0H`
- **Operation:**
  Writing to the port register outputs data to the pins. Reading the port register reads the logic level on the pins (provided a '1' was previously written to make it an input).
- **Alternate Functions:**
  Port 3 pins have alternate functions (e.g., Interrupts, Serial Rx/Tx, Timer inputs) which are enabled when the corresponding bit in P3 is set to '1'.

# Control Registers: Timers and Serial

## Timer/Counter Registers

- **TMOD (89H):** Timer Mode Register. Selects modes (0, 1, 2, 3) for Timer 0 and Timer 1.
- **TCON (88H):** Timer Control Register. Controls timer start/stop and contains timer overflow flags. Bit-addressable.
- **TH0/TL0, TH1/TL1:** 8-bit registers holding the actual timer values.

## Serial & Interrupt Control

- **SCON (98H):** Serial Control. Configures UART mode, Rx/Tx control flags. Bit-addressable.
- **SBUF (99H):** Serial Data Buffer. Holds data to transmit or data received.
- **IE (A8H) / IP (B8H):** Interrupt Enable and Priority registers.

# Summary: Applications of SFRs

- **ALU Operations:** Managed via ACC, B, and PSW.
- **Memory Access:** Facilitated by DPTR (for external data/code) and SP (for stack operations).
- **External Interfaces:** Controlled through P0, P1, P2, P3 registers.
- **Timing & Counting:** Configured and monitored via TMOD, TCON, and the Timer count registers.
- **Communication:** Handled by SCON and SBUF for UART interfacing.
- **Event Management:** Driven by IE and IP for hardware and software interrupts.

## Conclusion

Mastering SFRs is essential for programming the 8051, as they are the direct interface between the software and the hardware peripherals.

# Introduction to Special Function Registers (SFRs)

- **What are SFRs?** Registers mapped into the upper 128 bytes of internal RAM (addresses 80H to FFH) in the 8051 microcontroller.
- **Purpose:** Used to control and monitor the status of the microcontroller's on-chip peripherals.
- **Access:** They can be accessed using direct addressing mode only.
- **Bit-addressable SFRs:** Some SFRs (whose addresses end in 0 or 8 in hex) can have their individual bits accessed, while others are byte-addressable only.

## Key Point

SFRs are the control center for all on-chip features like timers, serial port, interrupts, and I/O ports.

# Core CPU Registers: ACC and B Register

- **Accumulator (ACC or A - Address E0H):**
  - The most heavily used register in the 8051.
  - Used for all arithmetic and logic operations.
  - Holds one of the operands and the result of the operation.
  - Bit-addressable.
- **B Register (Address F0H):**
  - Primarily used with the Accumulator for multiplication (`MUL AB`) and division (`DIV AB`) operations.
  - Can also be used as a simple general-purpose scratchpad register.
  - Bit-addressable.

# Program Status Word (PSW - Address D0H)

- Also known as the flag register. It is bit-addressable.
- Contains status flags that reflect the current state of the CPU after arithmetic or logic operations.

| CY | AC | F0 | RS1 | RS0 | OV | - | P |
|---|---|---|---|---|---|---|---|
| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |

- **CY (Carry Flag):** Set if there's a carry out of the MSB.
- **AC (Auxiliary Carry):** Set if carry from bit 3 to bit 4 (used in BCD).
- **RS1, RS0 (Register Bank Select):** Selects one of the 4 working register banks (Bank 0-3).
- **OV (Overflow Flag):** Set on arithmetic overflow.
- **P (Parity Flag):** Even parity flag for the accumulator.

# Memory Pointers: SP and DPTR

- **Stack Pointer (SP - Address 81H):**
  - 8-bit register that points to the top of the stack in internal RAM.
  - Increments before data is pushed onto the stack (`PUSH` or `CALL`).
  - Decrements after data is popped (`POP` or `RET`).
  - Default value upon reset is 07H.
- **Data Pointer (DPTR):**
  - A 16-bit register used to hold memory addresses for external data memory or program memory.
  - Consists of two 8-bit SFRs:
    - **DPH (Address 83H):** High byte of DPTR.
    - **DPL (Address 82H):** Low byte of DPTR.
  - Essential for `MOVX` (external RAM) and `MOVC` (ROM) instructions.

# I/O Port Latches (P0, P1, P2, P3)

- The 8051 has four 8-bit bidirectional I/O ports.
- Each port has a corresponding SFR that acts as a latch to hold output data or read input pins.
- All port SFRs are bit-addressable, allowing individual pin control.

## Port Addresses

- **P0 (Address 80H):** Port 0 latch. Also acts as lower address/data bus (AD0-AD7) for external memory.
- **P1 (Address 90H):** Port 1 latch. General-purpose I/O.
- **P2 (Address A0H):** Port 2 latch. Also acts as high-order address byte (A8-A15).
- **P3 (Address B0H):** Port 3 latch. Pins have alternate functions (RxD, TxD, INT0, INT1, T0, T1, WR, RD).

# Timer/Counter SFRs

- The 8051 has two 16-bit timers/counters (Timer 0 and Timer 1).
- **TCON (Timer Control - Address 88H):** Bit-addressable. Contains control bits and flags for timers and external interrupts (e.g., TR0, TR1, TF0, TF1).
- **TMOD (Timer Mode - Address 89H):** Byte-addressable. Used to set the operating mode of Timer 0 and Timer 1 (Modes 0, 1, 2, 3) and Gate control.

## Timer Data Registers

- **Timer 0:** Accessed via **TH0** (8CH, High byte) and **TL0** (8AH, Low byte).
- **Timer 1:** Accessed via **TH1** (8DH, High byte) and **TL1** (8BH, Low byte).

# Serial Port and Interrupt SFRs

- **Serial Port Control:**
  - **SCON (Address 98H):** Bit-addressable. Configures the serial port mode, reception enable, and holds transmit/receive interrupt flags (TI, RI).
  - **SBUF (Address 99H):** Byte-addressable. Serial buffer for transmitting and receiving data. Writing to SBUF initiates transmission. Reading SBUF gets received data.
- **Interrupt Control:**
  - **IE (Interrupt Enable - Address A8H):** Bit-addressable. Enables or disables specific interrupts (e.g., EA for global enable, EX0, ET0, ES).
  - **IP (Interrupt Priority - Address B8H):** Bit-addressable. Assigns priority levels (high or low) to each interrupt source.

# Applications of SFRs in Programming

- **Arithmetic/Logic:** Using ACC and B for calculations.
- **Branching:** Checking PSW flags (like CY or P) for conditional jumps (`JC`, `JNC`).
- **Memory Access:** Using DPTR to fetch look-up tables from ROM or interface external RAM.
- **I/O Interfacing:** Reading switches via P1 or controlling LEDs via P2.
- **Timing & Delays:** Configuring TMOD and loading THx/TLx to generate precise time delays.
- **Communication:** Setting up SCON and using SBUF to send/receive data via UART.

## Summary

Efficient 8051 programming requires a thorough understanding of SFRs, their addresses, and whether they are bit-addressable.

# Introduction to Addressing Modes

## What is an Addressing Mode?

- An **addressing mode** specifies how to calculate the effective memory address of an operand by using information held in registers and/or constants contained within a machine instruction or elsewhere.
- The 8051 microcontroller provides various addressing modes to access memory efficiently.

## 8051 Addressing Modes

The 8051 supports the following addressing modes:

1. Immediate Addressing
2. Register Addressing
3. Direct Addressing
4. Indirect Addressing
5. Indexed Addressing
6. Relative Addressing
7. Bit Addressing

# Immediate Addressing Mode

## Concept

- The operand comes **immediately** after the opcode.
- The data is provided directly in the instruction itself.
- A # symbol precedes the data to indicate it is an immediate value.

## Examples

```
MOV A, #25H     ; Load 25H into accumulator
MOV R4, #62     ; Load decimal 62 into register R4
MOV DPTR, #4521H ; Load 16-bit value 4521H into DPTR
```

## Note

Immediate data can be 8-bit or 16-bit (for DPTR). If an 8-bit data starts with a letter (A-F), it must be preceded by a 0 (e.g., #0A5H).

# Register Addressing Mode

## Concept

- Involves the use of registers to hold the data to be manipulated.
- The registers are typically R0 through R7 (from the currently selected register bank) and the Accumulator (A).

## Examples

```
MOV A, R0     ; Copy content of R0 into Accumulator
MOV R2, A     ; Copy content of Accumulator into R2
ADD A, R5     ; Add content of R5 to Accumulator
```

## Important Restrictions

- Moving data between two Rn registers directly is not allowed (e.g., `MOV R4, R7` is invalid).
- The size of source and destination must match.

# Direct Addressing Mode

## Concept

- The data is stored in memory, and its **direct 8-bit address** is specified in the instruction.
- Used to access internal RAM locations (00H to 7FH) and Special Function Registers (SFRs) (80H to FFH).
- Note the absence of the # sign.

## Examples

```
MOV R0, 40H   ; Save content of RAM location 40H into R0
MOV 56H, A    ; ...
MOV 04H, #12H; ...
```

# Indirect Addressing Mode

## Concept

- A register is used as a pointer to the data.
- Only registers **R0 and R1** are used for 8-bit addresses (Internal RAM).
- The `@` symbol is placed before the register name to indicate indirect addressing.

## Examples

```
MOV A, @R0    ; Move data from memory whose address
              ; is in R0 into A
MOV @R1, B    ; Move content of B into memory loc
              ; pointed to by R1
```

## Advantage

Very useful for iterating through blocks of memory (e.g., arrays or buffers) using loops.

# Indexed Addressing Mode

## Concept

- Extensively used for accessing data elements of look-up table entries located in the program ROM space of the 8051.
- Uses a base register (`DPTR` or `PC`) and an offset (`A`).
- The effective address is the sum of the base register and the offset.

## Examples

```
MOVC A, @A+DPTR ; Copy code byte found at ROM address
                ; formed by adding A and DPTR into A
MOVC A, @A+PC   ; ...
```

## Instruction Type

Note that `MOVC` (Move Code) is used instead of `MOV` since data is being accessed from ROM (Code memory).

# Relative Addressing Mode

## Concept

- Mainly used with jump and call instructions (e.g., `SJMP`, `JZ`, `CJNE`).
- The instruction specifies an offset (usually an 8-bit signed number: -128 to +127 bytes) relative to the Program Counter (PC).
- The PC is updated to branch to the target address: `Target Address = PC + Offset`.

## Examples

```
HERE: INC R2
      CJNE R2, #10, HERE ; ...
```

## Purpose

Allows for position-independent code. The code can be loaded anywhere in memory without changing the jump addresses.

# Bit Addressing Mode

## Concept

- The 8051 is unique in its ability to manipulate individual bits.
- Certain internal RAM locations (20H to 2FH) and Special Function Registers (SFRs) are bit-addressable.
- Bit addresses range from 00H to 7FH (for RAM) and 80H to FFH (for SFRs).

## Examples

```
SETB P1.0     ; Set bit 0 of Port 1 to 1
CLR 25H       ; Clear bit address 25H to 0
MOV C, 67H    ; ...
```

# Comparison of Addressing Modes

| Mode | Syntax Example | Key Characteristic |
|---|---|---|
| Immediate | `MOV A, #20H` | Data provided directly |
| Register | `MOV A, R0` | Uses Rn registers (R0-R7) |
| Direct | `MOV R0, 40H` | 8-bit RAM/SFR address given |
| Indirect | `MOV A, @R0` | Register (R0/R1) holds address |
| Indexed | `MOVC A, @A+DPTR` | A + DPTR/PC for ROM tables |
| Relative | `SJMP LABEL` | Offset relative to PC (-128 to +127) |
| Bit | `SETB P1.2` | Operates on single bits |

Table: Summary of 8051 Addressing Modes

# Overview of Instruction Execution

## What is an Instruction?

An instruction is a command given to the microcontroller to perform a specific operation on given data.

- Microcontrollers execute programs sequentially.
- A program consists of a series of **instructions**.
- Every instruction comprises two fundamental parts:
  - **Opcode** (Operation Code)
  - **Operand** (Data or Address)
- The execution of these instructions is governed by strict timing sequences defined by **T-states, Machine Cycles**, and **Instruction Cycles**.

# Opcode (Operation Code)

## Definition

The **Opcode** represents the specific operation or action to be performed by the microcontroller. It tells the CPU *what to do*.

- It is the first part of an instruction.
- The CPU decodes the opcode to understand the required action (e.g., addition, moving data, jumping to an address).
- Each mnemonic in assembly language (like `MOV`, `ADD`, `SUBB`) corresponds to a unique binary opcode understood by the 8051.

**Example:** In the instruction `ADD A, R0`, the operation is **ADD**, which has a specific hexadecimal/binary opcode representation in machine language.

# Operand

## Definition

The **Operand** represents the data upon which the operation (opcode) is to be performed, or the location (address) of that data. It tells the CPU *where to find the data or what data to use*.

- It is the second part of an instruction (though some instructions have no operands).
- An operand can be:
  - Immediate data (e.g., `#55H`)
  - A register (e.g., `A`, `R1`, `R2`)
  - A memory address (e.g., `30H`)

**Example:** In the instruction `MOV A, #45H`, the opcode is `MOV`, and the operands are `A` (destination) and `#45H` (source data).

# Opcode vs. Operand

| **Feature** | **Opcode** | **Operand** |
|---|---|---|
| **Meaning** | Operation Code | Data or Address |
| **Function** | Specifies the operation to be performed. | Specifies the data on which the operation is performed. |
| **Requirement** | Mandatory in every instruction. | Optional (some instructions don't need operands, e.g., `NOP`). |
| **Analogy** | The "Verb" (Action) | The "Noun" (Object/Subject) |

# Instruction Timing Vocabulary

To understand how fast an 8051 microcontroller executes instructions, we must define three fundamental timing parameters:

1. **T-State** (Clock Period)
2. **Machine Cycle**
3. **Instruction Cycle**

## Hierarchical Relationship

Multiple T-states make up a Machine Cycle.
One or more Machine Cycles make up an Instruction Cycle.

# T-State (Clock Cycle)

## Definition

A **T-state** (Timing State) is defined as one subdivision of the operation performed in one clock period. It is the most basic unit of time for a microcontroller.

- It is strictly dependent on the frequency of the connected oscillator (crystal).
- **Formula:** $T = \frac{1}{f_{osc}}$
- If an 8051 is connected to a 12 MHz crystal oscillator:
  - $f_{osc} = 12$ MHz
  - T-state $= \frac{1}{12 \times 10^6} = 0.083$ $\mu$s (or 83.33 ns)

# Machine Cycle

## Definition

A **Machine Cycle** is the time required by the microcontroller to complete one basic operation, such as fetching an opcode from memory, reading a memory location, or writing to a memory location.

- In the standard 8051 architecture, **one Machine Cycle consists of 12 oscillator periods (T-states)**.
- These 12 T-states are grouped into 6 states ($S_1$ to $S_6$), where each state has 2 phases ($P_1, P_2$).
- **Duration Calculation:**
  - Machine Cycle $= 12 \times$ T-state
  - For a 12 MHz crystal: $12 \times 0.083\ \mu s = 1\ \mu s$.

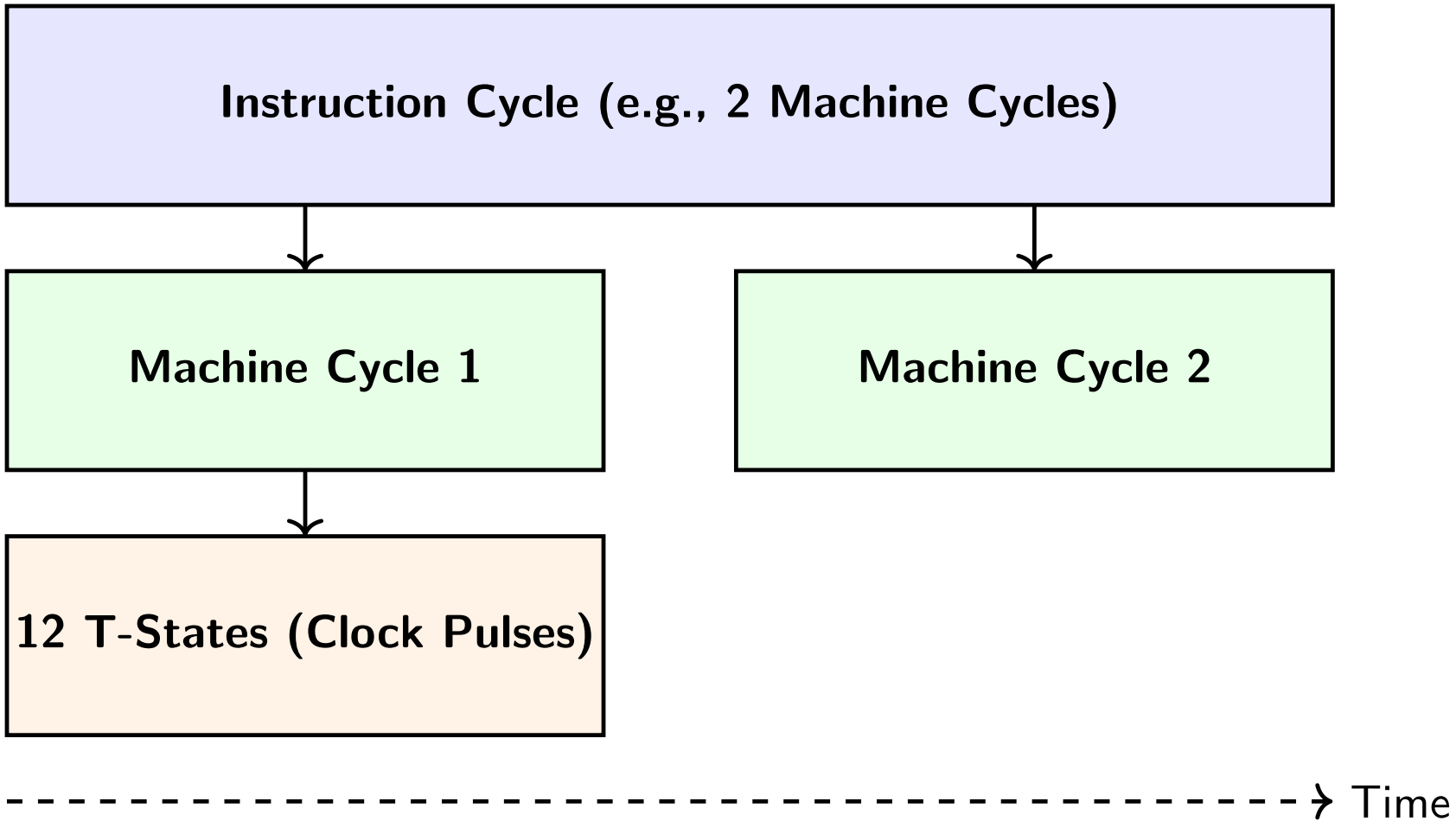
# Instruction Cycle

## Definition

An **Instruction Cycle** is the total time required by the microcontroller to fetch, decode, and completely execute a single instruction.

- Different instructions require a different amount of time to execute.
- An instruction cycle is always made up of one or more machine cycles.
- In the 8051 microcontroller:
  - **1-Cycle Instructions:** Take 1 machine cycle (e.g., `ADD A, R1`)
  - **2-Cycle Instructions:** Take 2 machine cycles (e.g., `DJNZ R1, label`)
  - **4-Cycle Instructions:** Take 4 machine cycles (e.g., `MUL AB`, `DIV AB`)

# Visualizing Timing Relationships




- **T-state:** The heartbeat (smallest tick).
- **Machine Cycle:** A full heartbeat cycle (12 ticks).
- **Instruction Cycle:** The task completed over 1 or more heartbeat cycles.

# Summary Example

## Example: `MOV A, #25H` at 12 MHz

- **Opcode:** `MOV` (The operation is a data transfer)
- **Operand:** `A` (Destination Register) and `#25H` (Immediate 8-bit Data)
- **T-state:** For 12 MHz oscillator, T-state $= \frac{1}{12\text{ MHz}} = 83.33$ ns.
- **Machine Cycle:** $12 \times 83.33$ ns $= 1$ $\mu$s.
- **Instruction Cycle:** The `MOV A, #data` instruction is a 1-byte, 1-machine cycle instruction. Thus, it takes exactly 1 $\mu$**s** to execute.

# Introduction to Instruction Execution

## Executing an Instruction

To execute an instruction, the microcontroller performs a series of operations:

- Fetches the instruction from memory.
- Decodes the instruction.
- Executes the instruction.

## Key Terms

Understanding how instructions are formed and executed requires defining several fundamental concepts: Opcode, Operand, T-state, Machine Cycle, and Instruction Cycle.

# Opcode and Operand

## Opcode (Operation Code)

- The part of an instruction that specifies the operation to be performed.
- Tells the microprocessor/microcontroller **what to do**.
- Example: `MOV`, `ADD`, `SUBB`.

## Operand

- The data on which the operation is to be performed.
- Tells the microprocessor/microcontroller **where to find the data** or what the data is.
- Can be data, a memory address, or a register.
- Example: In `MOV A, R1`, `A` and `R1` are operands.

# Instruction Format

## General Format

An instruction typically consists of an Opcode followed by zero, one, or two Operands.

**Instruction = Opcode + Operand(s)**

| Instruction | Opcode | Operand(s) |
|---|---|---|
| `INC A` | `INC` | `A` |
| `ADD A, #45H` | `ADD` | `A, #45H` |
| `NOP` | `NOP` | None |

# T-State (Time State)

## Definition

- A T-state is defined as one subdivision of the operation performed in one clock period.
- It is the basic unit to calculate the time taken for execution.
- These subdivisions are internal states synchronized with the system clock.
- One T-state = One Clock Period (Oscillator period).

## Formula

$$\text{T-state} = \frac{1}{\text{Oscillator Frequency}}$$

For example, if crystal frequency is 12 MHz:

$$\text{T-state} = \frac{1}{12\ \text{MHz}} = 0.0833\ \mu\text{s}$$

# Machine Cycle

## Definition

- The time required to complete one basic operation (e.g., fetching an opcode, reading from memory, writing to memory) is called a Machine Cycle.
- An instruction execution consists of one or more machine cycles.

## 8051 Machine Cycle

- In standard 8051, one machine cycle consists of **12 oscillator periods** (or 12 T-states).
- It is further divided into 6 states ($S1$ to $S6$), where each state has two phases ($P1$ and $P2$).
- Time for one machine cycle at 12 MHz:

$$1 \text{ MC} = 12 \times (\text{T-state}) = 12 \times 0.0833 \ \mu\text{s} = 1 \ \mu\text{s}$$

# Instruction Cycle

## Definition

- The total time required by the microcontroller to fetch, decode, and completely execute an instruction is known as the Instruction Cycle.
- An instruction cycle is made up of one or more machine cycles.

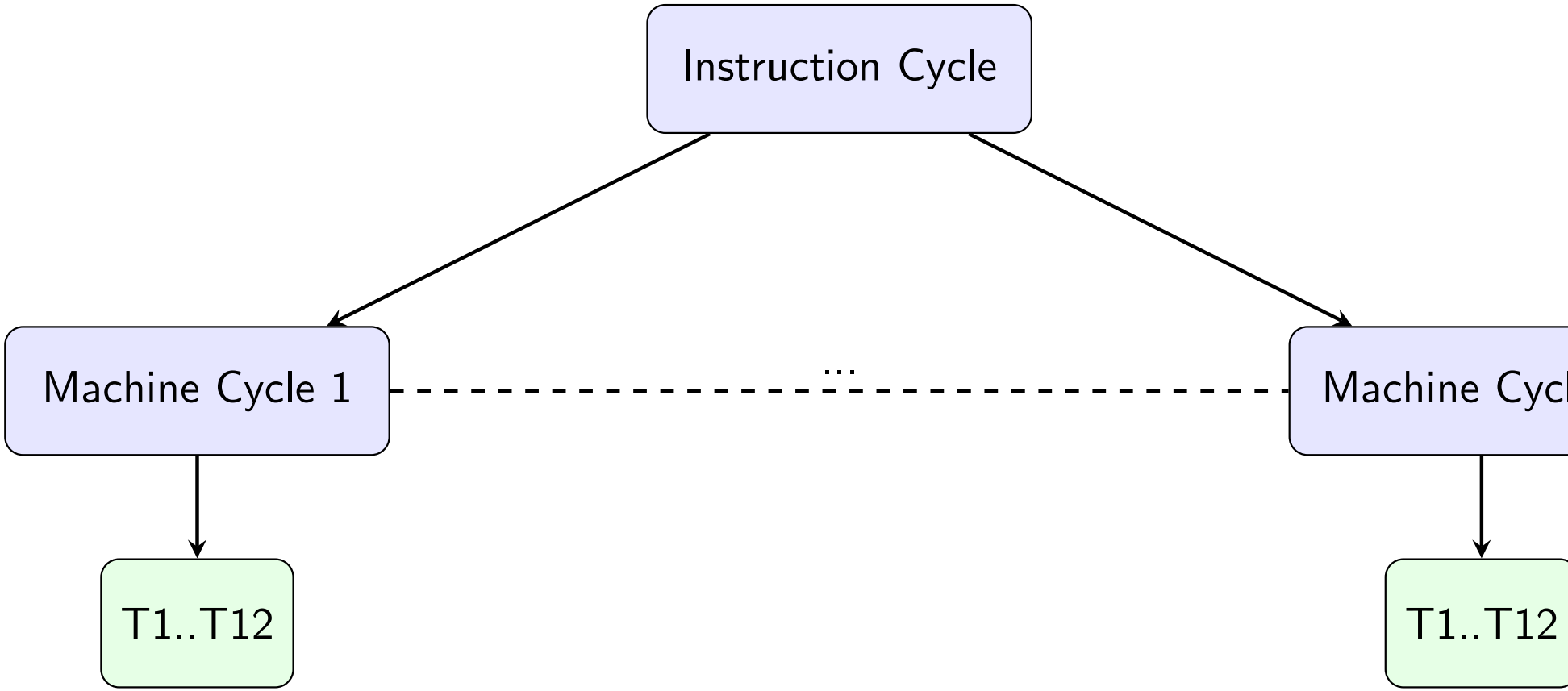
## Relationship

**Instruction Cycle = Fetch Cycle + Execution Cycle**

or

**Instruction Cycle $\geq$ Machine Cycle $\geq$ T-State**

# Relationship Diagram




## Summary

An Instruction Cycle comprises multiple Machine Cycles, and each Machine Cycle in 8051 is made of exactly 12 T-States (oscillator clock periods).

# Examples of 8051 Instructions

| Instruction | Bytes | Machine Cycles | Description |
|---|---|---|---|
| `NOP` | 1 | 1 | No operation |
| `ADD A, R1` | 1 | 1 | Add R1 to Acc |
| `MOV A, #55H` | 2 | 1 | Move immediate data |
| `MOV DPTR, #1234H` | 3 | 2 | Load 16-bit data |
| `MUL AB` | 1 | 4 | Multiply A and B |
| `DIV AB` | 1 | 4 | Divide A by B |

## Execution Time

For a 12 MHz crystal, a 1-machine cycle instruction takes 1 $\mu$s, and a 4-machine cycle instruction (like `MUL AB`) takes 4 $\mu$s.

# 8051 Instruction Set Overview

- The 8051 microcontroller has a rich instruction set of 111 instructions.
- They can be classified into five major groups based on their functionality:
  1. **Data Transfer Instructions**
  2. **Arithmetic Instructions**
  3. **Logical Instructions**
  4. **Branching (Boolean and Jump) Instructions**
  5. **Machine Control Instructions**
- Most instructions require 1 or 2 machine cycles to execute.

# Data Transfer Instructions

## Purpose

Used to move data between registers, memory, and I/O ports without altering the data.

- **MOV**: Move data within internal RAM or registers.
  - `MOV A, R0` (Move content of R0 to Accumulator)
  - `MOV R1, #45H` (Load immediate data 45H to R1)
- **MOVX**: Move data to/from external RAM.
  - `MOVX A, @DPTR` (Read from external memory into A)
- **MOVC**: Move code (read-only data) from program memory.
  - `MOVC A, @A+DPTR`
- **PUSH / POP**: Stack operations.

# Arithmetic Instructions

## Overview

Perform basic arithmetic operations like addition, subtraction, multiplication, and division.

- **ADD / ADDC**: Add or add with carry.
  - `ADD A, R2` (A $\leftarrow$ A + R2)
- **SUBB**: Subtract with borrow.
  - `SUBB A, #10H` (A $\leftarrow$ A - 10H - C)
- **INC / DEC**: Increment or decrement a register or memory location.
- **MUL AB**: Multiply A and B (16-bit result in B and A).
- **DIV AB**: Divide A by B (Quotient in A, Remainder in B).

# Logical Instructions

## Functionality

Perform bitwise logical operations such as AND, OR, XOR, and bit rotations.

- **ANL** (AND Logical), **ORL** (OR Logical), **XRL** (XOR Logical).
  - `ANL A, #0FH` (Mask upper nibble)
- **CLR / CPL**: Clear (set to 0) or Complement (invert bits).
  - `CLR A` (A $\leftarrow$ 00H)
- **Rotate Instructions**:
  - **RL A** / **RR A**: Rotate Accumulator left/right.
  - **RLC A** / **RRC A**: Rotate Accumulator left/right through Carry.

# Branching Instructions (1/2)

## Control Flow

Used to alter the sequential execution of the program. Include jumps, calls, and returns.

- **Unconditional Jumps**:
  - **SJMP** (Short Jump): $\pm$128 bytes range.
  - **LJMP** (Long Jump): 64KB range.
  - **AJMP** (Absolute Jump): 2KB range.
- **Subroutines**:
  - **LCALL** / **ACALL**: Call subroutine.
  - **RET**: Return from subroutine.
  - **RETI**: Return from interrupt.

# Branching Instructions (2/2)

- **Conditional Jumps**:
  - **JZ** / **JNZ**: Jump if A is Zero / Not Zero.
  - **JC** / **JNC**: Jump if Carry is set / not set.
  - **JB** / **JNB**: Jump if specified Bit is set / not set.
  - **CJNE**: Compare and Jump if Not Equal.
    - `CJNE A, #50H, TARGET`
  - **DJNZ**: Decrement and Jump if Not Zero. (Used for loops).
    - `DJNZ R1, LOOP`

# Machine Control Instructions

## Definition

Instructions that control CPU operation but do not affect registers or memory directly (except PC).

- **NOP** (No Operation):
  - Does nothing, simply consumes 1 machine cycle.
  - Useful for creating short time delays.
  - Used for aligning code or filling empty spaces in memory.

# Data Transfer Programs: Overview

## Introduction to Data Transfer

Data transfer operations are fundamental to programming the 8051 microcontroller. They involve moving data between various memory spaces and registers without modifying the data itself.

- **Internal RAM (Data Space):** Accessed using `MOV` instructions.
- **External RAM (XData Space):** Accessed using `MOVX` instructions.
- **ROM/Program Memory (Code Space):** Accessed using `MOVC` instructions.
- **Stack Operations:** Managed using `PUSH` and `POP` instructions.

# Internal RAM Data Transfer

## The `MOV` Instruction

The `MOV destination, source` instruction is the most common way to transfer data within the 8051's internal memory (Registers, Internal RAM, and SFRs).

- **Immediate Data:** `MOV A, #55H` (Load 55H into Accumulator)
- **Register to Register:** `MOV R0, A` (Copy A to R0)
- **Direct Addressing:** `MOV 40H, A` (Copy A to memory location 40H)
- **Indirect Addressing:** `MOV @R0, A` (Copy A to location pointed by R0)

# External RAM Data Transfer

## The MOVX Instruction

The MOVX instruction is exclusively used for reading from or writing to external data memory. It requires the use of the DPTR (Data Pointer) for 16-bit addresses or R0/R1 for 8-bit addresses.

**Example: Writing to External RAM**

```
MOV DPTR, #4500H ; ...
MOV A, #99H      ; Load immediate data 99H into A
MOVX @DPTR, A    ; ...
```

**Example: Reading from External RAM**

```
MOV DPTR, #4500H ; ...
MOVX A, @DPTR    ; ...
```

# ROM Data Transfer (Look-up Tables)

## The MOVC Instruction

The MOVC instruction reads data from program memory (ROM). It is commonly used for accessing look-up tables.

**Example: Reading from a Look-up Table**

```
MOV DPTR, #TABLE ; Point DPTR to the start of TABLE
MOV A, #03H      ; ...
MOVC A, @A+DPTR  ; A gets the value from TABLE+3

...

ORG 0500H
TABLE: DB 10H, 20H, 30H, 40H, 50H
```

# Stack Operations and Exchange

## Stack Instructions

The `PUSH` and `POP` instructions are used to save and restore data from the stack, primarily during subroutine calls and interrupts.

- `PUSH direct`: Increments SP, then stores data. (e.g., `PUSH 0E0H` saves Accumulator)
- `POP direct`: Retrieves data, then decrements SP. (e.g., `POP 0E0H` restores Accumulator)

## Exchange Instructions

The `XCH` instruction swaps the contents of the Accumulator with a register, direct memory, or indirect memory location.

- `XCH A, R2`: Swaps A and R2.
- `XCHD A, @R0`: Swaps the lower nibble of A with the lower nibble of the location pointed to by R0.

# Programming Example: Block Transfer

## Problem Statement

Transfer a block of 10 bytes from internal RAM starting at 30H to internal RAM starting at 50H.

```
        MOV R0, #30H    ; Source pointer
        MOV R1, #50H    ; Destination pointer
        MOV R2, #10     ; Counter (10 bytes)

LOOP:
        MOV A, @R0      ; Read byte from source
        MOV @R1, A      ; Write byte to destination
        INC R0          ; Increment source pointer
        INC R1          ; Increment destination pointer
        DJNZ R2, LOOP   ; ...
```

# Data Manipulation Overview

## Arithmetic and Logic Instructions

Data manipulation involves altering the data using Arithmetic and Logic Unit (ALU) operations.

- **Arithmetic:** `ADD`, `ADDC`, `SUBB`, `INC`, `DEC`, `MUL`, `DIV`. These operations often affect the PSW (Program Status Word) flags (Carry, Auxiliary Carry, Overflow, Parity).
- **Logic: Byte level:** `ANL`, `ORL`, `XRL`, `CPL`, `CLR`. **Bit level:** `RL`, `RR`, `RLC`, `RRC`, `SWAP`.

*Note: The Accumulator (A) is almost always the destination for arithmetic and most logic instructions.*

# Programming Example: Data Manipulation

## Problem Statement

Add two 16-bit numbers. Number 1 is at 30H (LSB) and 31H (MSB). Number 2 is at 40H (LSB) and 41H (MSB). Store result in 50H (LSB) and 51H (MSB).

```
MOV A, 30H      ; Load LSB of Number 1
ADD A, 40H      ; Add LSB of Number 2
MOV 50H, A      ; Store LSB of Result

MOV A, 31H      ; Load MSB of Number 1
ADDC A, 41H     ; ...
MOV 51H, A      ; Store MSB of Result
```

# Introduction to Data Transfer and Manipulation

- **Data Transfer:** Moving data between registers, memory, and I/O ports without altering its value.
- **Data Manipulation:** Modifying data using arithmetic or logical operations.
- In the 8051 microcontroller, these operations form the foundation of any non-trivial assembly language program.
- Efficient programming requires understanding different addressing modes and the specific instruction sets for these operations.

# Key Data Transfer Instructions

## The MOV Instruction

The fundamental instruction for data transfer in 8051.

- **Syntax:** `MOV destination, source`
- **Examples:**
  - `MOV A, R0` (Register to Accumulator)
  - `MOV R1, #55H` (Immediate data to Register)
  - `MOV DPTR, #1234H` (16-bit immediate data to Data Pointer)
  - `MOV 40H, A` (Accumulator to direct memory address)

# Advanced Data Transfer: External Memory

- **MOVX (Move External):** Used to access external RAM.
  - `MOVX A, @DPTR` (Read from external RAM into Accumulator)
  - `MOVX @DPTR, A` (Write from Accumulator to external RAM)
  - `MOVX A, @R0` (8-bit address for external RAM)
- **MOVC (Move Code):** Used to read from ROM (Code Memory).
  - Typically used for Look-Up Tables.
  - `MOVC A, @A+DPTR`
  - `MOVC A, @A+PC`

# Program Example: Block Data Transfer

## Task: Transfer 10 bytes from internal RAM 40H to 50H

```
        MOV R0, #40H      ; Source address pointer
        MOV R1, #50H      ; Destination address pointer
        MOV R2, #10       ; Counter for 10 bytes
LOOP:   MOV A, @R0        ; Get data from source
        MOV @R1, A        ; Store data to destination
        INC R0            ; Increment source pointer
        INC R1            ; ...
        DJNZ R2, LOOP     ; ...
```

# Data Manipulation: Arithmetic & Logic

**Arithmetic Manipulations**

- `ADD A, source`: Add to Accumulator
- `SUBB A, source`: Subtract with borrow
- `INC / DEC`: Increment or decrement operand
- `MUL AB / DIV AB`: Multiply and Divide

**Logical Manipulations**

- `ANL`: Bitwise AND
- `ORL`: Bitwise OR
- `XRL`: Bitwise XOR
- `CPL A`: Complement Accumulator
- `RL / RR`: Rotate Left / Right

# Program Example: Data Manipulation (Masking)

## Task: Extract the upper nibble (4 bits) of a byte in A

- When manipulating data, we often need to extract specific bits.
- We use logical AND (`ANL`) for masking.

```
; ...
ANL A, #0F0H        ; ...
                    ; Result in A: 80H (1000 0000)
SWAP A              ; ...
                    ; Result in A: 08H (0000 1000)
```

# Data Exchange Instructions

- The `XCH` instruction swaps data between the Accumulator and a byte of memory or register.
  - `XCH A, R0`: Exchange Accumulator and R0
  - `XCH A, 30H`: Exchange Accumulator and direct address 30H
  - `XCH A, @R1`: Exchange Accumulator and indirectly addressed RAM
- `XCHD` **(Exchange Digit):** Swaps the lower nibble (4 bits) between the Accumulator and indirectly addressed RAM.
  - `XCHD A, @R0`
  - Useful in BCD (Binary Coded Decimal) arithmetic.

# Stack Operations for Data Transfer

- The stack is heavily used for temporary data storage and during subroutine calls.
- Operations always use direct addressing for the source/destination.

## PUSH and POP

- `PUSH direct`: Increments Stack Pointer (SP) and stores data at new SP location.
- `POP direct`: Retrieves data from current SP location and decrements SP.

**Example: Swapping R0 and R1 using Stack**

```
PUSH 00H     ; ...
PUSH 01H     ; Push R1 onto stack
POP 00H      ; Pop into R0 (gets original R1)
POP 01H      ; Pop into R1 (gets original R0)
```

# Summary of Data Manipulation and Transfer

- **Data Transfer (`MOV`, `MOVX`, `MOVC`)** handles movement across internal RAM, external RAM, and ROM.
- **Data Manipulation** leverages Arithmetic and Logic instructions to process the data effectively.
- **Addressing Modes** (Immediate, Register, Direct, Indirect, Indexed) directly influence how efficiently data transfer programs can be written.
- Complex programs like block transfers combine data transfer instructions with loop counters and pointer incrementing.

# Introduction to Arithmetic Instructions

- The 8051 microcontroller provides a comprehensive set of arithmetic instructions.
- **Basic Operations:** Addition, Subtraction, Increment, Decrement, Multiplication, and Division.
- **Key Register:** The Accumulator (Register A) is heavily used in arithmetic operations.
- **Flags Affected:** The Program Status Word (PSW) flags such as Carry (CY), Auxiliary Carry (AC), and Overflow (OV) are updated based on the result.

## Common Arithmetic Mnemonics

`ADD, ADDC, SUBB, INC, DEC, MUL, DIV, DA A`

# Addition Operations

- `ADD A, source`: Adds the source operand to the Accumulator. The result is stored in the Accumulator.
- Source can be a register (e.g., `R0-R7`), direct address, indirect address (`@R0, @R1`), or immediate data (`#data`).
- `ADDC A, source`: Add with Carry. Adds the source operand and the Carry flag (CY) to the Accumulator. Useful for multi-byte addition.

## Examples

```
ADD A, #34H ; A = A + 34H
ADD A, R2   ; A = A + R2
ADDC A, 40H ; A = A + [40H] + CY
```

# Subtraction Operations

- `SUBB A, source`: Subtract with Borrow. Subtracts the source operand and the Carry flag (CY) from the Accumulator.
- The 8051 does NOT have a regular `SUB` instruction without borrow. If you want a simple subtract, you must clear the Carry flag first (`CLR C`).
- Result is stored in the Accumulator.

## Example: Simple Subtraction

```
CLR C          ; Clear CY flag before subtraction
MOV A, #55H ; Load A with 55H
SUBB A, #12H ; A = 55H - 12H - 0 = 43H
```

# Increment and Decrement

- `INC operand`: Increments the operand by 1.
- `DEC operand`: Decrements the operand by 1.
- Operands can be Accumulator, register, direct memory, or indirect memory.
- **Note:** `INC` and `DEC` do not affect the Carry (CY) flag (except `INC DPTR`, but it has no `DEC DPTR` counterpart).

## Examples

```
INC A    ; A = A + 1
DEC R5   ; R5 = R5 - 1
INC DPTR ; DPTR = DPTR + 1 (16-bit increment)
```

# Multiplication and Division

- `MUL AB`: Multiplies the unsigned 8-bit integers in Accumulator (A) and Register B.
  - The 16-bit product is stored in A (low byte) and B (high byte).
  - If the product $>$ 255 (FFH), the Overflow (OV) flag is set to 1.
- `DIV AB`: Divides the unsigned 8-bit integer in A by the unsigned 8-bit integer in B.
  - The integer quotient is stored in A.
  - The integer remainder is stored in B.
  - If B $=$ 00H before division, OV flag is set to 1 (divide by zero error).

# Introduction to Logical Instructions

- Logical instructions perform bitwise Boolean operations on operands.
- **Basic Operations:** AND, OR, XOR, Clear, Complement, Rotate, and Swap.
- These instructions generally do not affect the flags in the PSW, except for operations that explicitly involve the Carry flag.
- Often used for masking bits (setting or clearing specific bits without affecting others).

## Common Logical Mnemonics

`ANL, ORL, XRL, CLR, CPL, RL, RLC, RR, RRC, SWAP`

# Bitwise Logical Operations (AND, OR, XOR)

- `ANL destination, source`: Logical AND. Used to clear specific bits (masking).
- `ORL destination, source`: Logical OR. Used to set specific bits.
- `XRL destination, source`: Logical XOR. Used to toggle (invert) specific bits.
- Destination is typically the Accumulator or a direct memory address.

## Examples with Accumulator

```
ANL A, #0FH ; Clear upper nibble, keep lower nibble
ORL A, #80H ; Set the most significant bit (MSB)
XRL A, #FFH ; Toggle all bits (equivalent to CPL A)
```

# Rotate Instructions

- Rotate instructions shift the bits of the Accumulator left or right.
- `RL A`: Rotate Left without Carry. Bit 7 goes to Bit 0.
- `RLC A`: Rotate Left through Carry. Bit 7 goes to CY, CY goes to Bit 0.
- `RR A`: Rotate Right without Carry. Bit 0 goes to Bit 7.
- `RRC A`: Rotate Right through Carry. Bit 0 goes to CY, CY goes to Bit 7.

## Applications

Used in serial data transmission/reception, mathematical operations (e.g., multiply/divide by 2), and data bit manipulation.

# Clear, Complement, and Swap

- `CLR A`: Clears the Accumulator (A = 00H).
- `CPL A`: Complements (inverts) all bits in the Accumulator (1's complement).
- `SWAP A`: Swaps the lower nibble (bits 0-3) with the upper nibble (bits 4-7) of the Accumulator.

## SWAP Example

```
MOV A, #72H ; A = 0111 0010 (72H)
SWAP A      ; A = 0010 0111 (27H)
```

*Note: `SWAP A` is extremely useful when converting between packed BCD and ASCII data formats.*

# Masking in 8051 Microcontroller

- **Definition:** Masking is the process of modifying or retaining specific bits in a byte while clearing or setting others.
- **Why use Masking?**
  - To isolate specific bits (e.g., checking status flags).
  - To change specific bits without affecting the entire byte.
- **Logical Operations for Masking:**
  - `ANL` (Logical AND): Used to **clear** specific bits.
  - `ORL` (Logical OR): Used to **set** specific bits.
  - `XRL` (Logical Exclusive OR): Used to **toggle** specific bits.

# Masking Examples

## Using AND for Masking (Clearing Bits)

To clear the upper nibble (bits 4-7) of the accumulator and keep the lower nibble:

```
MOV A, #85H     ; A = 1000 0101 (85H)
ANL A, #0FH     ; A = A AND 0000 1111 (0FH)
                ; Result: A = 0000 0101 (05H)
```

## Using OR for Masking (Setting Bits)

To set the highest bit (bit 7) of the accumulator:

```
MOV A, #05H     ; A = 0000 0101 (05H)
ORL A, #80H     ; A = A OR 1000 0000 (80H)
                ; Result: A = 1000 0101 (85H)
```

# Stack Operation in 8051

- **What is a Stack?** A section of internal RAM used for temporary storage of data and addresses.
- **Stack Pointer (SP):**
  - 8-bit register that holds the address of the top of the stack.
  - Upon reset, `SP` is initialized to `07H`.
  - This means the first stack entry is at address `08H`.
- **LIFO Structure:** Last In, First Out.
- **Key Operations:**
  - `PUSH`: Storing data onto the stack.
  - `POP`: Retrieving data from the stack.

# PUSH and POP Instructions

## PUSH Operation

- SP is incremented by 1 (`SP = SP + 1`).
- Data from the specified direct address is copied to the RAM location pointed to by SP.

```
PUSH 0E0H    ; Push Accumulator (address E0H) onto stack
```

## POP Operation

- Data from the RAM location pointed to by SP is copied to the specified direct address.
- SP is decremented by 1 (`SP = SP - 1`).

```
POP 0E0H     ; Pop data from stack into Accumulator
```

# Conditional Programming Overview

- Conditional instructions allow the program to make decisions and alter the flow of execution based on specific conditions.
- Typically, conditions are based on:
  - The Carry Flag (CY).
  - The Accumulator (A) being zero or not.
  - The state of individual bits in bit-addressable memory or SFRs.
  - Comparison results.
- **Common Jump Instructions:**
  - `JZ` / `JNZ` (Jump if Zero / Jump if Not Zero)
  - `JC` / `JNC` (Jump if Carry / Jump if No Carry)
  - `JB` / `JNB` (Jump if Bit Set / Jump if Bit Not Set)
  - `CJNE` (Compare and Jump if Not Equal)
  - `DJNZ` (Decrement and Jump if Not Zero)

# Conditional Branching Examples

## Testing Accumulator Zero Flag

```
    MOV A, R0
    JZ IS_ZERO     ; Jump to IS_ZERO if A = 0
    ; Do something if A is not zero
    SJMP CONTINUE
IS_ZERO:
    ; Do something if A is zero
CONTINUE:
```

## Testing Carry Flag

```
    CLR C
    ADD A, R1
    JC CARRY_SET   ; Jump if addition produced a carry
    ; Handle no carry
    SJMP DONE
CARRY_SET:
    ; Handle carry
DONE:
```

# Compare and Decrement Branches

## CJNE: Compare and Jump if Not Equal

Compares first two operands; branches if they are not equal. Carry flag is set if *operand*1 $<$ *operand*2.

```
    CJNE A, #50H, NOT_EQUAL
    ; Executes if A == 50H
    SJMP DONE
NOT_EQUAL:
    ; Executes if A != 50H
DONE:
```

## DJNZ: Decrement and Jump if Not Zero

Used for looping. Decrements operand; if result is not zero, jumps.

```
    MOV R2, #10     ; Loop 10 times
LOOP_START:
    ; Loop body here
    DJNZ R2, LOOP_START ; ...
```

# Masking in 8051

## What is Masking?

Masking is the process of selectively modifying, extracting, or testing specific bits in a byte while leaving other bits unchanged.

- Commonly achieved using logical operations like **AND**, **OR**, and **XOR**.
- Useful for configuring specific pins of I/O ports or checking status flags.
- The `ANL` (Logical AND) instruction is typically used to clear (mask) bits.
- The `ORL` (Logical OR) instruction is used to set bits.

# Masking Operations Examples

## Clearing Bits (Using ANL)

To clear the higher nibble (bits 4-7) of the Accumulator and keep the lower nibble:

```
ANL A, #0FH   ; A = A AND 00001111B
```

## Setting Bits (Using ORL)

To set bit 0 and bit 1 of the Accumulator without affecting other bits:

```
ORL A, #03H   ; A = A OR 00000011B
```

# Stack Operation in 8051

## The Stack

The stack is a section of RAM used for temporary storage of data and return addresses. It operates on a Last-In, First-Out (LIFO) principle.

- Managed by the 8-bit **Stack Pointer (SP)**.
- Upon reset, the SP is initialized to `07H`.
- This means the first data pushed onto the stack will be stored at address `08H`.
- The stack grows upwards in the 8051 memory space.

# Stack Instructions: PUSH and POP

## PUSH Instruction

Stores data onto the stack.

- SP is incremented first (`SP = SP + 1`).
- Data is then stored at the new address pointed to by SP.

```
PUSH 0E0H   ; Push Accumulator onto stack
```

## POP Instruction

Retrieves data from the stack.

- Data at the address pointed to by SP is read.
- SP is then decremented (`SP = SP - 1`).

```
POP 0F0H    ; Pop stack content to B register
```

# Conditional Programming

## Overview

Conditional programming involves making decisions and altering the program flow based on certain conditions (usually the status of flags).

- Flags like Carry (C), Zero (Z), and Parity (P) in the Program Status Word (PSW) are often tested.
- **Jump Instructions:** These instructions transfer control to another part of the program depending on the tested condition.
- Essential for implementing loops, `if-else` logic, and polling operations.

# Common Conditional Jump Instructions

| Instruction | Description |
|---|---|
| `JZ rel` | Jump if Accumulator is Zero (`A = 0`) |
| `JNZ rel` | Jump if Accumulator is Not Zero (`A` $\neq$ `0`) |
| `JC rel` | Jump if Carry flag is set (`CY = 1`) |
| `JNC rel` | Jump if Carry flag is not set (`CY = 0`) |
| `JB bit, rel` | Jump if the specified bit is set (`bit = 1`) |
| `JNB bit, rel` | Jump if the specified bit is not set (`bit = 0`) |

# Conditional Programming Example

## Example: Looping with DJNZ

The DJNZ (Decrement and Jump if Not Zero) instruction is widely used for creating loops.

```
    MOV R2, #10      ; Load loop counter (10 iterations)
AGAIN:
    ADD A, #05H      ; Add 5 to Accumulator
    DJNZ R2, AGAIN   ; ...
```

- R2 acts as the loop counter.
- The loop executes 10 times until R2 becomes 0.

# Introduction to Bit Manipulation

## Overview

The 8051 microcontroller is renowned for its outstanding bit-manipulation capabilities. It can operate on individual bits directly, unlike many general-purpose microprocessors.

- Ideal for control applications (e.g., turning on/off a single relay, LED, or motor).
- The Boolean Processor in the 8051 has its own accumulator (the Carry Flag).
- Specialized instructions exist to set, clear, complement, and move individual bits.

# Bit-Addressable Memory in 8051

## Bit-Addressable RAM

16 bytes of internal RAM, from address **20H to 2FH**, are bit-addressable.

- This provides $16 \times 8 = 128$ individually addressable bits.
- Bit addresses range from **00H to 7FH**.
- Example: Bit 0 of RAM location 20H is bit address 00H. Bit 7 of 2FH is bit address 7FH.

## Bit-Addressable SFRs

Special Function Registers with addresses ending in **0H or 8H** are bit-addressable.

- Examples: A (E0H), B (F0H), P0 (80H), P1 (90H), P2 (A0H), P3 (B0H), PSW (D0H), IE (A8H), IP (B8H), TCON (88H), SCON (98H).

# Single-Bit Instructions (Boolean Operations)

| Instruction | Description |
|---|---|
| `CLR bit` | Clear bit (bit = 0) |
| `CLR C` | Clear carry flag (C = 0) |
| `SETB bit` | Set bit (bit = 1) |
| `SETB C` | Set carry flag (C = 1) |
| `CPL bit` | Complement bit (bit = NOT bit) |
| `CPL C` | Complement carry flag (C = NOT C) |

- **Example:** `SETB P1.0` sets pin 0 of Port 1 high.
- **Example:** `CPL P2.5` toggles the state of pin 5 of Port 2.

# Bit-Level Logical Operations

The Carry Flag (C) acts as the single-bit accumulator for logical operations.

| **Instruction** | **Description** |
|---|---|
| `ANL C, bit` | AND carry with bit ($C \leftarrow C$ AND $bit$) |
| `ANL C, /bit` | AND carry with inverse of bit ($C \leftarrow C$ AND $\overline{bit}$) |
| `ORL C, bit` | OR carry with bit ($C \leftarrow C$ OR $bit$) |
| `ORL C, /bit` | OR carry with inverse of bit ($C \leftarrow C$ OR $\overline{bit}$) |
| `MOV C, bit` | Move bit to carry flag ($C \leftarrow bit$) |
| `MOV bit, C` | Move carry flag to bit ($bit \leftarrow C$) |

# Bit-Level Jump Instructions

Conditional jumps based on the state of a specific bit are highly useful for polling or decision-making.

- `JC rel` : Jump if Carry is set ($C = 1$).
- `JNC rel` : Jump if Carry is not set ($C = 0$).
- `JB bit, rel` : Jump if the specified `bit` is set (1).
- `JNB bit, rel` : Jump if the specified `bit` is not set (0).
- `JBC bit, rel` : Jump if `bit` is set (1), and then **clear** the bit. Highly useful for flag polling.

## Example Application

Waiting for a switch press (connected to P1.2, active low):

```
WAIT: JB P1.2, WAIT   ; Loop here as long as P1.2 is high.
```

# Control Programs: Overview

## Definition

Control programs manage the flow of execution in an assembly language program based on conditions or loops, rather than executing instructions purely sequentially.

Key structures in 8051 control programming:

1. **Unconditional Jumps:** Branch to a target address without checking any condition.
2. **Conditional Jumps:** Branch to a target address only if a specific condition is met.
3. **Looping Constructs:** Repeating a block of code a specific number of times.
4. **Subroutine Calls:** Jumping to a reusable block of code and returning.

# Unconditional Jump Instructions

There are three types of unconditional jumps based on target address range:

| **Instruction** | **Bytes** | **Target Range** |
|---|---|---|
| `SJMP rel` (Short) | 2 | $-128$ to $+127$ bytes from PC. |
| `AJMP addr11` (Absolute) | 2 | Anywhere within the current 2KB block of memory. |
| `LJMP addr16` (Long) | 3 | Anywhere in the 64KB program memory space. |

## Indirect Jump

`JMP @A+DPTR`

Jumps to the address formed by adding the Accumulator to the Data Pointer. Useful for lookup tables and switch-case structures.

# Conditional Jump Instructions (Byte-Level)

These instructions test a byte (usually the Accumulator) and jump based on the result.

- `JZ rel` : Jump if Accumulator is Zero ($A = 00H$).
- `JNZ rel` : Jump if Accumulator is Not Zero ($A \neq 00H$).
- `CJNE dest, src, rel` : Compare and Jump if Not Equal.
  - Compares two operands. If they are not equal, execution jumps to the relative address.
  - It also affects the Carry flag: $C = 1$ if `dest < src`, $C = 0$ if `dest >= src`.
  - Used extensively for looping and magnitude comparisons.

# Looping with DJNZ

## Decrement and Jump if Not Zero (DJNZ)

This is the most common instruction used for implementing counters and loops in 8051.

**Syntax:** `DJNZ Rn, rel` or `DJNZ direct, rel`

- It decrements the specified register or memory location by 1.
- If the result is not zero, it jumps to the relative address.
- If the result is zero, it falls through to the next instruction.

## Example: Loop 10 times

```
  MOV R2, #10    ; Initialize counter
LOOP: ...         ; Body of the loop
  ...
  DJNZ R2, LOOP    ; Decrement R2, jump to LOOP if not zero
```

# Summary

- **Bit Manipulation:** The 8051 has a powerful Boolean processor. Addressable bits reside in RAM (20H-2FH) and specific SFRs.
- **Bit Instructions:** `SETB`, `CLR`, `CPL` modify individual bits, while `JB`, `JNB`, `JC` enable bit-level decision making.
- **Program Control:** Jumps (`SJMP`, `LJMP`) alter the flow unconditionally.
- **Conditional Branching:** `JZ`, `JNZ`, `CJNE` form the basis of logical branching.
- **Looping:** `DJNZ` is highly optimized for implementing iterative loops.

**Course: Microprocessor & Microcontroller (DI04000111)**

**Unit 3: 8051 Programming and Instruction Set**

# Lecture 26: Introduction to 8051 C programming

# Lecture Agenda

- Why use C instead of Assembly?
- Structure of an 8051 C Program
- Data Types in 8051 C
- Basic C Programming Examples for 8051
- Summary

# Why C over Assembly for 8051?

## Advantages of C

- Easier to write, read, maintain, and debug than Assembly.
- Code can be easily ported to other microcontrollers.
- Large libraries and functions available, accelerating development.
- Data structures like arrays and structs are natively supported.

## Trade-off

- Slightly larger code size and less precise timing compared to Assembly.

# Structure of 8051 C Program

- **Header files:** `#include <reg51.h>` is required for 8051 SFRs.
- **Global variables** and macro definitions.
- **Function declarations**.
- **The** `main()` | `function` - Execution starts here.
- **Infinite loop:** An embedded C program usually runs an infinite loop like `while(1)`.
- **Subroutines / Functions**.

# 8051 Specific Data Types in C

| Keyword | Description / Example |
|---|---|
| `sbit` | Used to define single bits within SFRs.<br>Example: `sbit LED = P1^0;` |
| `sfr` | Used to define 8-bit Special Function Registers.<br>Example: `sfr P1 = 0x90;` |
| `bit` | A single bit variable in bit-addressable memory. |

## Standard Data Types

Standard data types like `unsigned char` (8-bit) and `unsigned int` (16-bit) are highly recommended over signed types when dealing with hardware.

# Example 1: Toggling an LED

**Task:** Toggle an LED connected to Pin 1.0 continuously.

## Code Snippet

```
#include <reg51.h>
sbit LED = P1^0; // Define LED pin

void delay() {
    unsigned int i;
    for(i=0; i<30000; i++);
}

void main() {
    while(1) {
        LED = ~LED;
        delay();
    }
}
```

# Example 2: Reading a Switch

**Task:** Read a switch on P2.0 and control an LED on P1.0.

## Code Snippet

```
#include <reg51.h>
sbit SW = P2^0;
sbit LED = P1^0;

void main() {
  SW = 1; // Configure P2.0 as input
  while(1) {
      LED = SW; // Mirror switch to LED
  }
}
```

# Summary

## Key Takeaways

- C programming offers higher productivity and portability over Assembly.
- Embedded C programs require hardware specific headers like `reg51.h`.
- Special data types like `sbit` and `sfr` are used to access 8051 hardware.
- Infinite loops like `while(1)` are used to keep the program running.

# Introduction to 8051 C Programming

## Why C instead of Assembly?

- It is easier and less time-consuming to write in C than in Assembly.
- C code is easier to modify and update.
- C provides a library of built-in functions, saving time on rewriting standard operations.
- C programming allows programmers to write code that is portable, with minimal modifications needed for different microcontrollers.
- The compiler handles complex tasks such as register allocation and memory management.

# Structure of a C Program for Microcontrollers

## Standard Components

A typical C program for an 8051 microcontroller contains:

- **Include Directives:** e.g., `#include <reg51.h>` to define SFRs.
- **Global Variables / Macros:** Declarations accessible throughout the program.
- **Function Prototypes:** Declarations of user-defined functions.
- **Main Function:** The entry point, `void main(void)`, often containing an infinite loop (`while(1)`) since microcontrollers run continuously.
- **Interrupt Service Routines (ISRs):** Specialized functions executed in response to hardware interrupts.

# Data Types in 8051 C

| Data Type | Size (Bits) | Range/Description |
|---|---|---|
| `unsigned char` | 8 | 0 to 255 (Ideal for SFRs/ports) |
| `char` | 8 | -128 to +127 |
| `unsigned int` | 16 | 0 to 65535 |
| `int` | 16 | -32768 to +32767 |
| `sbit` | 1 | Single bit (0 or 1) of an SFR |
| `bit` | 1 | Single bit in bit-addressable RAM |
| `sfr` | 8 | 8-bit Special Function Register |

# Accessing Ports and SFRs

## The <reg51.h> Header File

To work with the 8051 in C, we include the standard header file which maps the SFR addresses to names.

## Using sbit

We can define individual pins using the `sbit` keyword (if supported by the specific C compiler like Keil).

```
#include <reg51.h>
sbit LED = P1^0;   // Define LED at Port 1, Pin 0
sbit SW = P2^3;    // Define Switch at Port 2, Pin 3
```

# Basic Example: Toggling a Port

## Program: Toggle all pins of Port 1 continuously

```
#include <reg51.h>

void delay(unsigned int time) {
    unsigned int i, j;
    for(i = 0; i < time; i++)
        for(j = 0; j < 1275; j++);
}

void main() {
    while(1) {              // Infinite loop
        P1 = 0x55;          // ...
        delay(100);         // Wait
        P1 = 0xAA;          // ...
        delay(100);         // Wait
    }
}
```

# Basic Example: Reading and Writing

## Program: Read switch on P1.0 and output to LED on P2.0

```
#include <reg51.h>

sbit SW = P1^0;          // Switch connected to P1.0
sbit LED = P2^0;         // LED connected to P2.0

void main() {
    SW = 1;              // Configure P1.0 as input
    LED = 0;             // Initialize LED as off

    while(1) {
        if (SW == 0)     // ...
            LED = 1;     // Turn ON LED
        else
            LED = 0;     // Turn OFF LED
    }
}
```

# Introduction to Timers and Counters in 8051

- The 8051 microcontroller has two 16-bit timers/counters: **Timer 0** and **Timer 1**.
- They can be used either as:
  - **Timers:** To generate time delays. The clock source is the internal oscillator.
  - **Counters:** To count events happening outside the microcontroller. The clock source is an external input pin.
- Each 16-bit timer is accessed as two separate 8-bit registers:
  - Timer 0: TH0 (High byte) and TL0 (Low byte)
  - Timer 1: TH1 (High byte) and TL1 (Low byte)

# Timer/Counter Registers

## 16-bit Structure

Since the 8051 is an 8-bit microcontroller, a 16-bit register is accessed as two 8-bit registers.

| Timer | High Byte (8-bit) | Low Byte (8-bit) |
|---|---|---|
| Timer 0 | TH0 | TL0 |
| Timer 1 | TH1 | TL1 |

- These registers can be accessed like any other SFR (Special Function Register).
- Examples: `MOV TL0, #05H`, `MOV TH1, #4AH`

# TMOD (Timer Mode) Register

- The **TMOD** register is used to set the various operating modes of Timer 0 and Timer 1.
- It is an 8-bit register. The lower 4 bits are for Timer 0, and the upper 4 bits are for Timer 1.

## TMOD Register Format

| **Timer 1** | | | | **Timer 0** | | | |
|---|---|---|---|---|---|---|---|
| GATE | C/$\overline{T}$ | M1 | M0 | GATE | C/$\overline{T}$ | M1 | M0 |

# TMOD Register Bits Description

- **GATE:** When set to 1, the timer operates only when the external `INT` pin is high and `TR` (Timer Run) is set. When 0, the timer runs regardless of the `INT` pin, provided `TR` is set.
- **C/$\overline{\text{T}}$ (Counter/Timer):**
  - 1 = Counter operation (input from external pins T0 or T1).
  - 0 = Timer operation (internal clock, oscillator frequency / 12).
- **M1, M0:** Mode select bits.

| M1 | M0 | Mode |
|---|---|---|
| 0 | 0 | Mode 0: 13-bit Timer/Counter |
| 0 | 1 | Mode 1: 16-bit Timer/Counter |
| 1 | 0 | Mode 2: 8-bit Auto-reload |
| 1 | 1 | Mode 3: Split Timer |

# TCON (Timer Control) Register

- The **TCON** register contains status and control bits for both timers.
- Unlike TMOD, TCON is bit-addressable.

## TCON Register Format

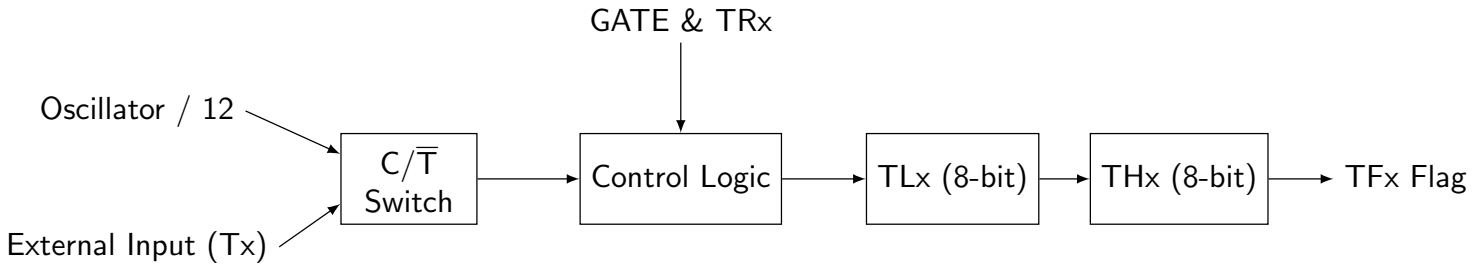
| TF1 | TR1 | TF0 | TR0 | IE1 | IT1 | IE0 | IT0 |
|---|---|---|---|---|---|---|---|

- **TR1/TR0:** Timer 1/0 Run control bit. Set to 1 to start the timer, clear to 0 to stop.
- **TF1/TF0:** Timer 1/0 Overflow Flag. Set by hardware when timer overflows. Must be cleared by software.
- (IE and IT bits are related to external interrupts).

# Timer/Counter Logic Diagram (Mode 1 - 16-bit)

## Mode 1 Operation

In Mode 1, the timer functions as a 16-bit timer or counter. The registers TH and TL are cascaded.



# Introduction to 8051 Timer Modes

- The 8051 microcontrollers have two 16-bit timers/counters: **Timer 0** and **Timer 1**.
- These timers can operate in four different modes.
- The operating mode is selected by configuring the **M1** and **M0** bits in the **TMOD** (Timer Mode) register.

| M1 | M0 | Mode | Description |
|---|---|---|---|
| 0 | 0 | 0 | 13-bit Timer/Counter |
| 0 | 1 | 1 | 16-bit Timer/Counter |
| 1 | 0 | 2 | 8-bit Auto-reload Timer/Counter |
| 1 | 1 | 3 | Split Timer Mode |

# Mode 0: 13-bit Timer/Counter

## Characteristics

- Operates as a 13-bit timer/counter.
- Uses all 8 bits of THx and lower 5 bits of TLx (where x = 0 or 1).
- The upper 3 bits of TLx are ignored.
- The maximum count is $2^{13} - 1 = 8191$ (1FFFH).

- When the timer reaches its maximum count (1FFFH), it rolls over to 0000H.
- The timer overflow flag (TFx) in the TCON register is set on rollover.
- Mode 0 is mainly provided for compatibility with the older 8048 microcontroller and is rarely used in modern applications.

# Mode 1: 16-bit Timer/Counter

## Characteristics

- Operates as a full 16-bit timer/counter.
- Uses all 8 bits of THx and all 8 bits of TLx.
- The maximum count is $2^{16} - 1 = 65535$ (FFFFH).

- The timer counts from its initial loaded value up to FFFFH.
- When it rolls over from FFFFH to 0000H, the TFx (Timer Overflow Flag) is set.
- **Manual Reloading:** In Mode 1, the timer must be reloaded with the initial value in software every time it overflows if a continuous periodic delay is required.

# Mode 2: 8-bit Auto-Reload Timer/Counter

## Characteristics

- Operates as an 8-bit auto-reload timer/counter.
- Uses only the TLx register for counting.
- The THx register holds the "reload value" and does not change during counting.
- Maximum count for TLx is $2^8 - 1 = 255$ (FFH).

- **Operation:**
  1. TLx counts up until it overflows from FFH to 00H.
  2. The TFx flag is set.
  3. The value in THx is automatically copied (reloaded) into TLx by hardware.
  4. TLx resumes counting from this reloaded value continuously.
- This mode is extensively used for generating baud rates for serial communication.

# Mode 3: Split Timer Mode

## Timer 0 in Mode 3

- Timer 0 acts as two independent 8-bit timers: TL0 and TH0.
- **TL0** uses Timer 0 control bits: C/$\overline{T}$0, GATE0, TR0, INT0, and TF0.
- **TH0** is locked into a timer mode (cannot be a counter) and uses Timer 1's run control (TR1) and overflow flag (TF1).

## Timer 1 in Mode 3

- If Timer 0 is in Mode 3, Timer 1 cannot set the TF1 flag or generate interrupts (as its flag is used by TH0).
- Timer 1 can still be used in Modes 0, 1, or 2 for applications like baud rate generation, where no interrupt is required and it runs continuously.

# Summary of Timer Modes

| Mode | Type | Key Feature |
|---|---|---|
| **0** | 13-bit | Legacy compatibility (8048). Uses 8 bits of THx and 5 bits of TLx. |
| **1** | 16-bit | Standard 16-bit timer. Requires manual software reload after overflow. |
| **2** | 8-bit Auto-reload | Hardware automatically reloads TLx from THx on overflow. Great for baud rates. |
| **3** | Split Timer | Splits Timer 0 into two 8-bit timers. Timer 1 loses its interrupt flag. |

# Introduction to 8051 Timers/Counters

- The 8051 microcontroller has two 16-bit Timers/Counters: **Timer 0** and **Timer 1**.
- They can be used either as **Timers** to generate a time delay or as **Counters** to count events happening outside the microcontroller.
- Both Timer 0 and Timer 1 are 16 bits wide, formed by two 8-bit registers:
  - Timer 0: `TH0` (High byte) and `TL0` (Low byte)
  - Timer 1: `TH1` (High byte) and `TL1` (Low byte)
- The operation mode is configured using the **TMOD** (Timer Mode) register.

# The TMOD Register

## Timer Mode Register (TMOD)

- TMOD is an 8-bit register used to set the various timer operation modes.
- The lower 4 bits are for Timer 0, and the upper 4 bits are for Timer 1.

| GATE | C/$\overline{T}$ | M1 | M0 | GATE | C/$\overline{T}$ | M1 | M0 |
|---|---|---|---|---|---|---|---|
| Timer 1 | | | | Timer 0 | | | |

- **GATE**: Timer gating control. When set, timer only runs while `INTx` pin is high.
- **C/$\overline{T}$**: Counter/Timer selector (1 = Counter, 0 = Timer).
- **M1, M0**: Mode selector bits.

# Timer/Counter Operating Modes

The mode selector bits **M1** and **M0** determine the operating mode of the timers.

| M1 | M0 | Mode | Description |
|---|---|---|---|
| 0 | 0 | Mode 0 | 13-bit Timer/Counter |
| 0 | 1 | Mode 1 | 16-bit Timer/Counter |
| 1 | 0 | Mode 2 | 8-bit Auto-Reload Timer/Counter |
| 1 | 1 | Mode 3 | Split Timer Mode |

# Mode 0: 13-bit Timer/Counter

- Mode 0 is a **13-bit Timer/Counter** mode.
- It uses 8 bits of the high byte (THx) and a 5-bit prescaler from the low byte (TLx).
- As the timer ticks, the 5 bits in TLx count from 0 to 31. When it overflows, it increments THx.
- When the entire 13-bit register rolls over from `1FFFH` to `0000H`, the timer overflow flag (`TFx`) is set.
- This mode was primarily used for compatibility with the older 8048 microcontroller and is rarely used in modern 8051 applications.

# Mode 1: 16-bit Timer/Counter

- Mode 1 operates as a full **16-bit Timer/Counter**.
- It uses all 8 bits of THx and all 8 bits of TLx.
- The timer counts from `0000H` to `FFFFH`.
- When it exceeds `FFFFH`, it rolls over to `0000H` and sets the timer overflow flag (`TFx`) in the TCON register.

## Steps to use Mode 1

1. Load the TMOD value register indicating which timer is to be used and in which mode.
2. Load registers TLx and THx with initial count values.
3. Start the timer (e.g., `SETB TRx`).
4. Monitor the `TFx` flag to see if it is raised (`JNB TFx, target`).
5. Stop the timer, clear the `TFx` flag, and repeat.

# Mode 2: 8-bit Auto-Reload

- Mode 2 is an **8-bit Auto-Reload Timer/Counter**.
- It uses only the TLx register (8 bits) as a timer/counter.
- The THx register holds the *reload value* and does not change during counting.
- When TLx overflows from `FFH` to `00H`:
  - The timer overflow flag (`TFx`) is set.
  - The value in THx is automatically loaded (auto-reloaded) into TLx.
- This mode is highly useful for applications requiring repetitive, uniform delays, such as setting baud rates for serial communication.

# Mode 3: Split Timer Mode

- Mode 3 is known as the **Split Timer Mode**.
- **Timer 0 in Mode 3**:
  - Timer 0 is split into two independent 8-bit timers: TL0 and TH0.
  - TL0 functions as an 8-bit timer/counter controlled by standard Timer 0 bits (TR0, TF0).
  - TH0 functions strictly as an 8-bit *timer* controlled by Timer 1 bits (TR1, TF1).
- **Timer 1 in Mode 3**:
  - If Timer 0 is in Mode 3, Timer 1 simply stops if its mode is set to Mode 3.
  - Timer 1 can still be used in Modes 0, 1, or 2, but it cannot generate interrupts (since TH0 took over TF1). It is often used as a baud rate generator in this configuration.

# Summary of Timer Control (TCON)

## Timer Control Register (TCON)

The upper 4 bits of TCON are used for timer control:

- **TF1 (Bit 7)**: Timer 1 Overflow flag. Set by hardware on overflow, cleared by software.
- **TR1 (Bit 6)**: Timer 1 Run control bit. Set/cleared by software to start/stop Timer 1.
- **TF0 (Bit 5)**: Timer 0 Overflow flag.
- **TR0 (Bit 4)**: Timer 0 Run control bit.

**Note:** To start Timer 0, use `SETB TR0`. To stop it, use `CLR TR0`. The same logic applies to Timer 1 using `TR1`.

# Introduction to Serial Communication

- In serial communication, data is transferred one bit at a time over a single data line.
- It is widely used for long-distance communication as it requires fewer wires compared to parallel communication.
- Two main methods:
  - **Synchronous:** A common clock signal synchronizes the transmitter and receiver.
  - **Asynchronous:** No common clock; start and stop bits frame each byte of data (e.g., UART).

## Simplex, Half-Duplex, and Full-Duplex

- **Simplex:** Data flows in one direction only.
- **Half-Duplex:** Data flows in both directions, but one at a time.
- **Full-Duplex:** Data flows in both directions simultaneously (supported by 8051).

# 8051 Serial Communication Hardware

- The 8051 microcontroller has a built-in UART (Universal Asynchronous Receiver-Transmitter) for full-duplex serial communication.
- It uses two pins of Port 3:
  - **RxD (P3.0):** Receive Data pin.
  - **TxD (P3.1):** Transmit Data pin.
- Data transmission and reception are handled by the **SBUF (Serial Buffer)** register.

## SBUF Register

- SBUF is an 8-bit register.
- Writing a byte to SBUF automatically initiates serial transmission.
- Reading from SBUF gives the byte that has been received.
- Physically, there are two separate SBUF registers (one for Tx, one for Rx), but they share the same address (99H).

# SCON (Serial Control) Register

- **SCON** is an 8-bit, bit-addressable register used to configure the serial port.

| SM0 | SM1 | SM2 | REN | TB8 | RB8 | TI | RI |
|---|---|---|---|---|---|---|---|
| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |

- **SM0, SM1:** Serial mode specifier bits.
- **SM2:** Used for multiprocessor communication (usually 0).
- **REN:** Receive Enable. Set to 1 to allow data reception.
- **TB8:** 9th bit transmitted (in Mode 2 and 3).
- **RB8:** 9th bit received (in Mode 2 and 3).
- **TI:** Transmit Interrupt flag. Set by hardware at the end of transmission. Must be cleared by software.
- **RI:** Receive Interrupt flag. Set by hardware when a byte is received. Must be cleared by software.

# Serial Communication Modes

The 8051 supports four serial communication modes, determined by SM0 and SM1 in SCON:

| SM0 | SM1 | Mode | Description | Baud Rate |
|---|---|---|---|---|
| 0 | 0 | 0 | 8-bit Shift Register | Fixed ($f_{osc}/12$) |
| 0 | 1 | 1 | 8-bit UART (1 start, 1 stop) | Variable (set by Ti |
| 1 | 0 | 2 | 9-bit UART (1 start, 1 stop) | Fixed ($f_{osc}/64$ or $f_o$ |
| 1 | 1 | 3 | 9-bit UART (1 start, 1 stop) | Variable (set by Ti |

- **Mode 1** is the most commonly used mode for standard PC and peripheral communication.

# Baud Rate Calculation for Mode 1

- In Mode 1, the baud rate is generated using **Timer 1** in **Mode 2** (8-bit auto-reload).
- The UART divides the machine cycle frequency by 32 (or 16 if SMOD=1).

## Formula

$$\text{Baud Rate} = \frac{2^{\text{SMOD}}}{32} \times \frac{\text{Oscillator Frequency}}{12 \times (256 - \text{TH1})}$$

- With an 11.0592 MHz crystal (ideal for serial communication) and SMOD = 0:
  - Timer 1 frequency = 11.0592 MHz/12 = 921.6 kHz
  - UART clock = 921.6 kHz/32 = 28,800 Hz
  - Baud Rate = 28800/(256 − TH1)

**Common TH1 values for 11.0592 MHz:**

- 9600 baud → `TH1 = -3` (FDH)
- 4800 baud → `TH1 = -6` (FAH)
- 2400 baud → `TH1 = -12` (F4H)

# Programming the Serial Port (Mode 1)

## Steps for Transmission

1. Configure TMOD register to use Timer 1 in Mode 2 (e.g., `TMOD = 0x20`).
2. Load TH1 with the value for the desired baud rate (e.g., `TH1 = 0xFD` for 9600).
3. Configure SCON for Mode 1 (e.g., `SCON = 0x50` enables REN and Mode 1).
4. Start Timer 1 (`TR1 = 1`).
5. Write a byte to SBUF to start transmission (`SBUF = data`).
6. Wait until TI flag becomes 1 (`while(TI==0);`).
7. Clear TI flag (`TI = 0`) and repeat for next byte.

## Steps for Reception

1. Configure TMOD for Timer 1 in Mode 2 and TH1 for baud rate.
2. Configure SCON for Mode 1 with REN=1 (`SCON = 0x50`).
3. Start Timer 1 (`TR1 = 1`).

# Example: Transmitting a Character (C Language)

## Code snippet to send 'A' at 9600 baud

```
#include <reg51.h>

void main(void) {
    TMOD = 0x20;        // Timer 1, Mode 2 (Auto-reload)
    TH1 = 0xFD;         // ...
    SCON = 0x50;        // ...
    TR1 = 1;            // Start Timer 1

    while (1) {
        SBUF = 'A';     // Load data to SBUF
        while (TI == 0); // ...
        TI = 0;         // ...
    }
}
```

# Example: Receiving Data (Assembly Language)

## Code snippet to receive a byte and send to Port 1

```
    MOV TMOD, #20H    ; Timer 1, Mode 2
    MOV TH1, #-3      ; 9600 baud rate
    MOV SCON, #50H    ; 8-bit, 1 stop, REN enabled
    SETB TR1          ; Start timer 1
WAIT:
    JNB RI, WAIT      ; Wait until byte is received
    MOV A, SBUF       ; ...
    MOV P1, A         ; Send data to Port 1
    CLR RI            ; Clear RI flag
    SJMP WAIT         ; Wait for next byte
```

# Introduction to 8051 Interrupts

## What is an Interrupt?

An interrupt is an external or internal event that interrupts the microcontroller to inform it that a device needs its service.

- When an interrupt occurs, the microcontroller temporarily suspends its main program, saves its state, and executes an **Interrupt Service Routine (ISR)**.
- After finishing the ISR, the microcontroller returns to the main program.

## 8051 Interrupt Sources

The 8051 has 5 primary interrupt sources:

1. External Interrupt 0 (`INT0`)
2. Timer 0 Overflow Interrupt (`TF0`)
3. External Interrupt 1 (`INT1`)
4. Timer 1 Overflow Interrupt (`TF1`)
5. Serial Port Interrupt (`RI/TI`)

# Interrupt Vector Addresses

- When an interrupt occurs, the microcontroller jumps to a fixed memory location called the **Interrupt Vector Address**.
- The programmer must place the ISR (or a jump to it) at this address.

| **Interrupt Source** | **Flag Name** | **Vector Address (Hex)** |
|---|---|---|
| External Interrupt 0 | IE0 | 0003H |
| Timer 0 Overflow | TF0 | 000BH |
| External Interrupt 1 | IE1 | 0013H |
| Timer 1 Overflow | TF1 | 001BH |
| Serial Communication | RI/TI | 0023H |

# Interrupt Priorities in 8051

## Default Priority

If multiple interrupts arrive simultaneously, the 8051 services them based on a predefined polling sequence (hardware priority):

1. External Interrupt 0 (Highest)
2. Timer 0 Interrupt
3. External Interrupt 1
4. Timer 1 Interrupt
5. Serial Communication (Lowest)

- This default order can be altered using the **Interrupt Priority (IP)** register.
- A higher priority interrupt can interrupt a lower priority ISR, but a lower priority interrupt cannot interrupt a higher priority ISR.

# Interrupt Priority (IP) Register

- The IP register allows the programmer to assign either a **high (1)** or **low (0)** priority level to each interrupt.
- It is a bit-addressable register.

| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|
| - | - | PT2 | PS | PT1 | PX1 | PT0 | PX0 |

- **PS**: Serial Port Interrupt Priority
- **PT1**: Timer 1 Interrupt Priority
- **PX1**: External Interrupt 1 Priority
- **PT0**: Timer 0 Interrupt Priority
- **PX0**: External Interrupt 0 Priority

*(PT2 is for Timer 2 in 8052).*

# Interrupt Enable (IE) Register

- Upon reset, all interrupts are disabled. They must be enabled by software.
- The **IE (Interrupt Enable)** register is used to enable or disable interrupts.

| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|
| EA | - | ET2 | ES | ET1 | EX1 | ET0 | EX0 |

- **EA (Enable All)**: If 0, no interrupt is acknowledged. If 1, each interrupt source is individually enabled or disabled by its specific bit.
- **ES**: Enable Serial Interrupt
- **ET1 / ET0**: Enable Timer 1 / Timer 0 Interrupt
- **EX1 / EX0**: Enable External Interrupt 1 / 0

# Interrupt Operations

## Sequence of Execution

When an enabled interrupt occurs and is accepted by the CPU:

1. The current instruction is finished.
2. The CPU pushes the **Program Counter (PC)** onto the stack to save the return address.
3. The CPU jumps to the fixed **Interrupt Vector Address** corresponding to the interrupt.
4. The **Interrupt Service Routine (ISR)** is executed.
5. The ISR must end with the `RETI` (Return from Interrupt) instruction.
6. `RETI` pops the saved PC from the stack, and execution resumes in the main program.

# The RETI Instruction

- **RETI** is crucial at the end of every ISR.
- It performs two main tasks:
  1. Pops the high and low bytes of the Program Counter (PC) from the stack.
  2. Clears the internal interrupt-in-progress flip-flop, allowing the microcontroller to acknowledge new interrupts of the same or lower priority.
- **Difference from RET:** A simple `RET` instruction returns to the main program but does *not* clear the internal interrupt status, effectively blocking future interrupts of that level.

# Introduction to Interrupt Service Routine (ISR)

- An **Interrupt Service Routine (ISR)** (or interrupt handler) is a special subroutine that executes when a specific interrupt occurs.
- Unlike normal subroutines, an ISR is not called by the `CALL` instruction but is triggered by hardware or software events.
- When an interrupt is activated, the microcontroller pauses its current task and jumps to the ISR.
- After completing the ISR, the microcontroller resumes the main program from where it left off.

## Key Characteristics

- Ends with `RETI` (Return from Interrupt) instead of `RET`.
- Does not require parameters to be passed.
- Must execute quickly to avoid blocking other operations.

# ISR vs. Normal Subroutine

| **Normal Subroutine** | **Interrupt Service Routine (ISR)** |
|---|---|
| Invoked by a `CALL` instruction | Triggered by an interrupt event |
| Can be located anywhere in memory | Must start at a specific vector addre |
| Ends with the `RET` instruction | Ends with the `RETI` instruction |
| Executes synchronously | Executes asynchronously |
| Can pass parameters | Cannot pass parameters directly |

- `RETI` restores the Program Counter (PC) and clears the interrupt-in-progress flag.
- `RET` only restores the PC.

# Interrupt Vector Table (IVT) in 8051

- The **Interrupt Vector Table** is a dedicated memory area that stores the starting addresses of ISRs for each interrupt source.
- When an interrupt occurs, the 8051 automatically jumps to the corresponding address in the IVT.

| **Interrupt Source** | **Vector Address** | **Flag Bit** |
|---|---|---|
| External Interrupt 0 (`INT0`) | `0003H` | `IE0` |
| Timer 0 Overflow (`TF0`) | `000BH` | `TF0` |
| External Interrupt 1 (`INT1`) | `0013H` | `IE1` |
| Timer 1 Overflow (`TF1`) | `001BH` | `TF1` |
| Serial Communication (`RI/TI`) | `0023H` | `RI/TI` |

- Usually, an `LJMP` instruction is placed at the vector address to jump to the actual ISR location, as only 8 bytes are allocated between vectors.

# Steps in Executing an Interrupt

When an interrupt is triggered, the 8051 performs the following steps automatically:

1. It finishes the instruction currently being executed.
2. It pushes the current **Program Counter (PC)** onto the stack.
3. It clears the interrupt flag (for Timer and External edge-triggered interrupts).
4. It sets an internal "interrupt-in-progress" flip-flop to prevent interrupts of the same or lower priority.
5. It loads the PC with the vector address of the interrupt.
6. It starts executing the ISR from that address.
7. Upon reaching `RETI`, it pops the PC from the stack and clears the internal flip-flop, resuming the main program.

# Programming an ISR in Assembly

## Basic Structure of an Assembly Program with ISR

```
        ORG 0000H          ; Reset vector
        LJMP MAIN          ; Jump to main program

        ORG 000BH          ; Timer 0 Interrupt Vector
        LJMP T0_ISR        ; Jump to Timer 0 ISR

        ORG 0030H          ; Main program starts here
MAIN:   ...                ; ...
        ...
        SJMP $             ; Wait here for interrupt

T0_ISR:                    ; Timer 0 ISR body
        ...                ; Perform task
        RETI               ; Return from interrupt
```

# Programming an ISR in C (Keil C51)

- In C, an ISR is defined as a regular function with a special `interrupt` keyword and an interrupt number.

| **Interrupt Source** | **Interrupt Number** |
|---|---|
| External Interrupt 0 | 0 |
| Timer 0 | 1 |
| External Interrupt 1 | 2 |
| Timer 1 | 3 |
| Serial Communication | 4 |

## Example Syntax

```
void timer0_ISR(void) interrupt 1 {
    // ISR code goes here
    // e.g., reload timer values, toggle LED
}
```

# Context Saving in ISR

- An ISR often modifies CPU registers (e.g., Accumulator, PSW, R0-R7).
- If the main program is using these registers, modifying them in the ISR will corrupt the main program's data.
- **Context Saving:** The process of temporarily storing register values before modifying them.

## Assembly Example

```
T0_ISR:
        PUSH ACC          ; Save Accumulator
        PUSH PSW          ; Save Program Status Word
        ; ... ISR tasks that use A and PSW ...
        POP PSW           ; Restore PSW
        POP ACC           ; Restore Accumulator
        RETI
```

- Note: In Keil C, context saving is automatically handled by the compiler.

# Best Practices for ISR Programming

1. **Keep it Short and Fast:** An ISR should execute as quickly as possible to allow other interrupts to be serviced and the main program to proceed.
2. **Avoid Delays:** Never use polling or long delay loops (e.g., `delay_ms`) inside an ISR.
3. **Use Flags:** Let the ISR set a flag variable, and let the main program perform the heavy processing based on that flag.
4. **Save Context:** Always push registers to the stack before using them (in Assembly) and pop them in reverse order before `RETI`.
5. **Register Banks:** In 8051, you can switch register banks inside the ISR (using `using` keyword in C or modifying `RS0`/`RS1` in Assembly) to avoid pushing/popping R0-R7.

# Interrupt Service Routine (ISR) Programming

- An **Interrupt Service Routine (ISR)**, also known as an Interrupt Handler, is a specific subroutine executed when an interrupt occurs.
- Unlike normal subroutines called via `CALL` instructions, an ISR is triggered by hardware or software events.
- Each interrupt has a fixed memory location associated with it, known as the **Interrupt Vector**.
- When an interrupt is triggered, the microcontroller suspends its current task, saves the Return Address on the stack, and branches to the corresponding vector location.

# The Interrupt Vector Table in 8051

## Vector Locations

The 8051 directs execution to specific addresses in the lower memory space based on the interrupt source.

| **Interrupt Source** | **Vector Address (Hex)** |
|---|---|
| Reset (System Start) | 0000H |
| External Interrupt 0 (INT0) | 0003H |
| Timer 0 Interrupt | 000BH |
| External Interrupt 1 (INT1) | 0013H |
| Timer 1 Interrupt | 001BH |
| Serial Communication (RI/TI) | 0023H |

*Note: Only 8 bytes of space are available between most vector locations.*

# Placing the ISR

- Because the space between interrupt vectors is very small (usually 8 bytes), we cannot fit a large ISR directly at the vector address.
- **Solution:** Place an unconditional jump (`LJMP` or `SJMP`) at the vector address, pointing to the actual ISR located elsewhere in memory.

## Example Structure in Assembly

```
ORG 0000H
LJMP MAIN      ; Bypass interrupt vectors
ORG 0003H      ; Vector for INT0
LJMP EX0_ISR    ; Jump to actual ISR
```

# Context Saving in ISRs

- Since an interrupt can occur at any unpredictable time, the ISR must not alter the registers and flags being used by the main program.
- The `PSW` (Program Status Word) and Accumulator (`A`) are frequently used and must be preserved.
- **Context Saving:** Use `PUSH` instructions at the start of the ISR to save registers onto the stack.
- **Context Restoring:** Use `POP` instructions at the end of the ISR (in reverse order) to restore the registers.

# Example: Preserving Context

## Assembly Code Snippet

```
EXO_ISR:
   PUSH ACC       ; Save Accumulator
   PUSH PSW       ; Save Program Status Word
   ; ... Perform Interrupt Tasks ...
   POP PSW        ; Restore Program Status Word
   POP ACC        ; Restore Accumulator
   RETI           ; Return from Interrupt
```

# Returning from an ISR (RETI)

- The RETI (Return from Interrupt) instruction must be used at the end of an ISR instead of a normal RET.
- **What RETI does:**
  1. Pops the return address from the stack into the Program Counter (PC).
  2. **Clears the internal interrupt-in-progress flip-flop**, alerting the microcontroller that the ISR has finished.
- If RET is used instead of RETI, the microcontroller will not clear the interrupt-in-progress state, blocking further interrupts of the same or lower priority.

# Writing ISRs in C (Keil C51)

- Writing an ISR in C is much simpler because the compiler handles context saving, restoring, and placing the `RETI` instruction automatically.
- Use the `interrupt` keyword followed by the interrupt number.

| **Interrupt Source** | **Interrupt Number** |
|---|---|
| External 0 | 0 |
| Timer 0 | 1 |
| External 1 | 2 |
| Timer 1 | 3 |
| Serial Comm | 4 |

# C Language ISR Example

## Timer 0 Interrupt Example in C

```
void timer0_isr (void) interrupt 1
{
  // Toggle an LED or update a counter
  P1_0 = ~P1_0;
  // Keil C51 will automatically save/restore registers
  // and append RETI.
}
```

- **Rule:** An ISR cannot take arguments (`void`) and cannot return a value (`void`).

# Introduction to Switches and Push Buttons

- Switches and push buttons are primary input devices used in embedded systems.
- They allow users to interact with the microcontroller to provide logic HIGH or LOW inputs.
- **Types of Switches:**
  - SPST (Single Pole Single Throw)
  - SPDT (Single Pole Double Throw)
  - Push Buttons (Momentary Switches)
- **Push Button Characteristics:**
  - Normally Open (NO): Circuit is broken until the button is pressed.
  - Normally Closed (NC): Circuit is complete until the button is pressed.

# Switch Interfacing Concepts

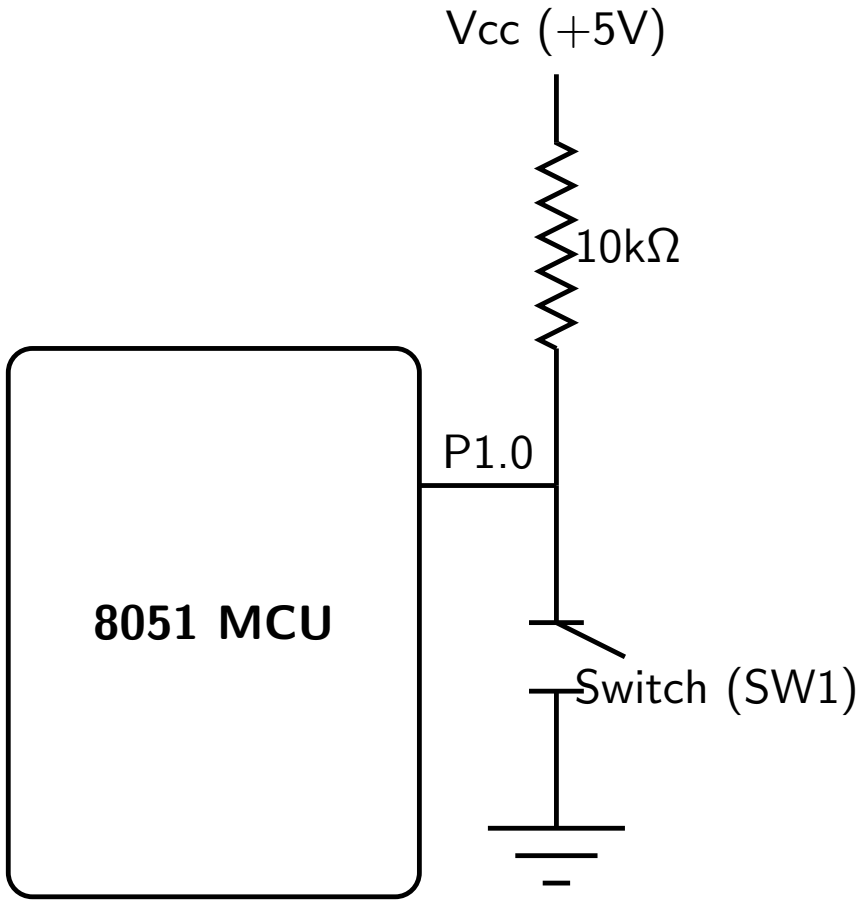
## Port Configuration for Input

- To read input from a pin on the 8051, the corresponding port pin must be configured as an input pin.
- This is done by writing a '1' to the port pin.
- Once configured as input, the microcontroller can read the logic level present at the pin.

## Pull-Up and Pull-Down Resistors

- **Pull-Up Resistor:** Connects the pin to Vcc, providing a default HIGH state. When the switch is closed, it connects the pin to Ground (LOW).
- **Pull-Down Resistor:** Connects the pin to Ground, providing a default LOW state. When the switch is closed, it connects the pin to Vcc (HIGH).
- 8051's Ports 1, 2, and 3 have internal pull-up resistors, while Port 0 requires external pull-ups.

# Interfacing Diagram: Push Button to 8051




- The switch is connected between Port pin P1.0 and Ground.
- A pull-up resistor keeps the pin HIGH when the switch is open.
- Pressing the switch shorts the pin to Ground, making it LOW.

# Programming Example: Assembly Language

**Task:** Read a switch connected to P1.0 and turn on an LED connected to P2.0 when the switch is pressed.

## Assembly Code

```
        ORG 0000H          ; Start of program
        SETB P1.0          ; Configure P1.0 as input pin
        CLR P2.0           ; Initialize LED (Turn OFF)
BACK:   JB P1.0, BACK      ; ...
        SETB P2.0          ; Turn ON LED (if active high)
HERE:   JNB P1.0, HERE     ; Wait until switch is released
        CLR P2.0           ; Turn OFF LED
        SJMP BACK          ; Repeat the process
        END                ; End of program
```

- `SETB P1.0` ensures P1.0 is ready to read input.
- `JB` (Jump if Bit set) and `JNB` (Jump if Bit not set) are used to check the pin state.

# Programming Example: C Language

**Task:** Read a switch connected to P1.0 and turn on an LED connected to P2.0 when the switch is pressed.

## C Code (Embedded C)

```
#include <reg51.h>
sbit SW = P1^0;   // Define Switch at P1.0
sbit LED = P2^0;  // Define LED at P2.0

void main(void) {
    SW = 1;       // Configure P1.0 as input
    LED = 0;      // Turn OFF LED initially

    while(1) {    // Infinite loop
        if(SW == 0) {  // Check if switch is pressed
            LED = 1;   // Turn ON LED
        } else {
            LED = 0;   // Turn OFF LED
        }
    }
}
```

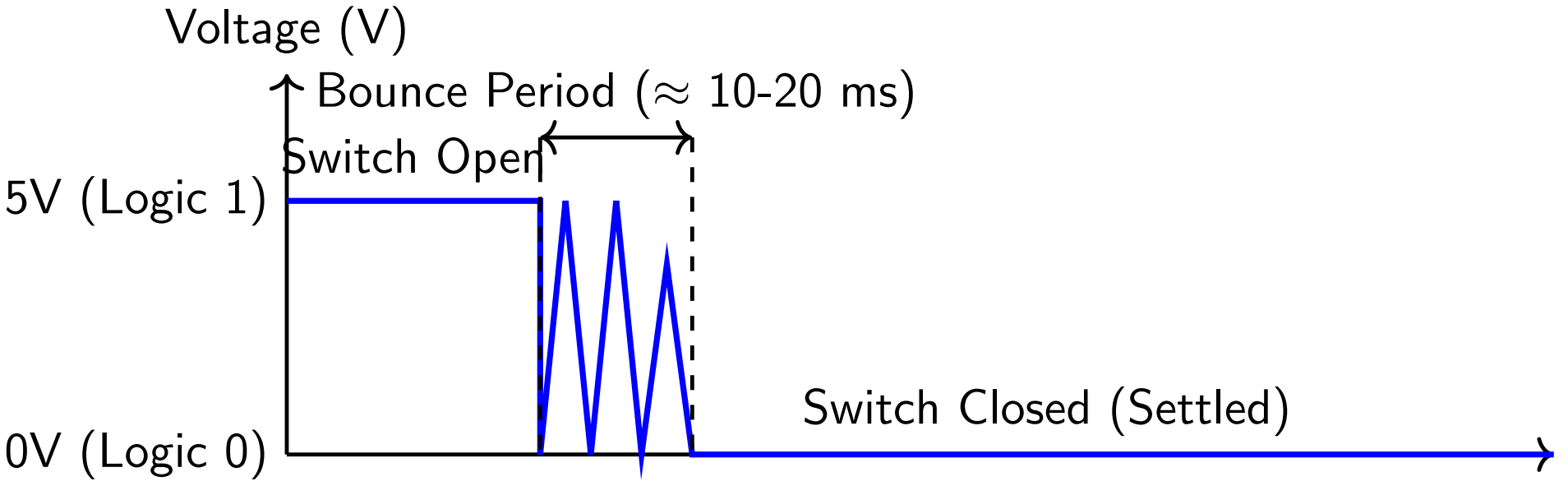
# The Switch Bouncing Problem

- Mechanical switches consist of moving metal contacts.
- When a switch is closed or opened, the contacts do not make or break a connection instantly.
- They **"bounce"** rapidly against each other several times before settling into the final state.

## Effect of Bouncing

- This rapid bouncing causes the logic level at the microcontroller pin to fluctuate rapidly between HIGH and LOW.
- A microcontroller is fast enough to read these fluctuations as multiple separate button presses.
- This can cause errors in applications like counters, menu navigation, or toggling states.

# Switch Bouncing Waveform




- The bounce duration typically lasts between 10ms to 20ms.
- **Debouncing** is required to ignore these false transitions and read only the true switch state.

# Debouncing Techniques

## 1. Hardware Debouncing

- Uses external electronic components to filter out the high-frequency voltage fluctuations.
- **RC Filter:** A resistor and capacitor network acts as a low-pass filter, slowing down the voltage transition.
- **SR Latch (Flip-Flop):** Provides a clean, bounce-free output but requires SPDT switches.
- **Schmitt Trigger:** Used with an RC filter to provide a sharp transition between logic levels.

## 2. Software Debouncing

- The most common and cost-effective method (requires no extra hardware).
- The microcontroller detects the initial button press, waits for a short delay (e.g., 20ms), and checks the button state again.
- If the button is still in the pressed state after the delay, it is considered a valid press.

# Software Debouncing: C Language Example

## Debounced Switch Reading in C

```
#include <reg51.h>
sbit SW = P1^0;

void delay(unsigned int msec) {
    unsigned int i, j;
    for(i=0; i<msec; i++)
        for(j=0; j<1275; j++); // ...
}

void main() {
    SW = 1; // Input pin
    while(1) {
        if(SW == 0) {          // Initial press detection
            delay(20);         // ...
            if(SW == 0) {      // ...
                // ...
                while(SW == 0); // ...
            }
```

# Summary

- Switches and push buttons are fundamental input devices for the 8051 microcontroller.
- A port pin must be configured as an input by writing a '1' to it before reading a switch.
- Proper interfacing requires pull-up or pull-down resistors to ensure stable logic levels.
- Mechanical switches suffer from the **bouncing** effect, causing multiple false triggers.
- **Debouncing** (hardware or software) is essential to ensure reliable and accurate reading of switch inputs. Software delays of 10-20ms are commonly used to achieve this.

# Introduction to Output Interfacing

- The 8051 microcontroller is widely used to interact with the real world through input/output devices.
- Today we cover interfacing common output devices:
  - LEDs (Light Emitting Diodes)
  - Relays (Electromechanical switches)
  - 7-Segment Displays
  - LCDs (Liquid Crystal Displays)
- These devices allow microcontrollers to display data or control high-power external circuits.

# Interfacing LED with 8051

## Overview

An LED (Light Emitting Diode) requires a small current (typically 10 to 20 mA) and a forward voltage to illuminate.

- Microcontroller pins can act as current sources or current sinks.
- **Current Sinking (Active Low):** Cathode connected to pin, Anode to Vcc via resistor. Logic 0 turns LED ON. (Recommended for 8051)
- **Current Sourcing (Active High):** Anode connected to pin, Cathode to GND via resistor. Logic 1 turns LED ON.
- A current limiting resistor (e.g., 330 $\Omega$) is necessary to protect the LED and the port pin.

# Interfacing Relay with 8051

## What is a Relay?

A relay is an electromechanical switch used to control high-power devices (like AC motors, heaters) using a low-power microcontroller signal.

- **Why use a Driver?** The 8051 cannot provide enough current (typically requires $> 50$ mA) to energize a relay coil directly.
- **Driver Circuit:** A transistor (e.g., NPN like BC547) or a driver IC (like ULN2003) is used.
- **Freewheeling Diode:** Placed across the relay coil to protect the transistor from back-EMF generated when the coil is de-energized.

# Relay Interfacing Circuit Diagram

- When 8051 pin outputs Logic 1: Transistor turns ON $\rightarrow$ Relay coil is energized $\rightarrow$ Contacts switch (NO closes, NC opens).
- When 8051 pin outputs Logic 0: Transistor turns OFF $\rightarrow$ Relay coil de-energizes $\rightarrow$ Contacts return to normal state.

*(A conceptual diagram showing 8051 Pin $\rightarrow$ Base Resistor $\rightarrow$ NPN Transistor $\rightarrow$ Relay Coil with parallel diode)*

# Interfacing 7-Segment Display

## 7-Segment LED Display

Contains 7 individual LEDs arranged in a figure '8', plus an optional decimal point (DP).

- **Common Anode (CA):** All anodes are tied to Vcc. To light a segment, apply Logic 0 to the respective cathode.
- **Common Cathode (CC):** All cathodes are tied to GND. To light a segment, apply Logic 1 to the respective anode.
- **Lookup Table:** A hexadecimal value is sent to the port to display digits (0-9) or letters (A-F).

# 7-Segment Interfacing Logic

Table: Common Anode Hex Codes (Port mapping: P1.0=a ... P1.7=dp)

| Digit | a | b | c | d | e | f | g | dp | Hex Code |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0xC0 |
| 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0xF9 |
| 2 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0xA4 |
| 3 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0xB0 |

- The hex values are stored in a lookup table in the program memory.
- The microcontroller simply fetches the required code and outputs it on the selected port.

# Interfacing LCD with 8051 (16x2 Character LCD)

## Overview

A 16x2 LCD can display 16 characters per line on 2 lines. It is widely used over 7-segment displays for displaying text and complex characters.

- The LCD uses an HD44780 or compatible controller.
- **Data Pins:** D0 to D7 (8-bit data bus) or D4 to D7 (4-bit mode).
- **Control Pins:**
  - **RS (Register Select):** 0 $\rightarrow$ Command register, 1 $\rightarrow$ Data register.
  - **RW (Read/Write):** 0 $\rightarrow$ Write to LCD, 1 $\rightarrow$ Read from LCD.
  - **EN (Enable):** A high-to-low pulse latches data into the LCD.

# LCD Initialization and Commands

- Sending a command:
  1. Put command byte on data bus.
  2. Set RS = 0 (Command).
  3. Set RW = 0 (Write).
  4. Pulse Enable (EN) pin (High $\rightarrow$ Low).
- **Common LCD Commands (Hex):**
  - `0x38`: 2 lines, 5x7 matrix (8-bit mode).
  - `0x0C`: Display ON, Cursor OFF.
  - `0x01`: Clear display screen.
  - `0x80`: Force cursor to beginning of 1st line.

# Summary

- **LEDs** require appropriate current limiting and sinking/sourcing configuration.
- **Relays** need driver circuits (transistors) and freewheeling diodes to control high-power AC/DC loads safely.
- **7-Segment Displays** use lookup tables to convert numerical data into visual segment patterns.
- **LCDs** provide complex text output and require initialization routines and specific control signal timing (RS, RW, EN).

# Introduction to Output Interfacing

- **Microcontroller 8051 Output Ports:** The 8051 has four bidirectional ports (P0, P1, P2, P3) that can be used for interfacing output devices.
- **Current Limitations:** A typical 8051 output pin can sink a few milliamperes of current but cannot source sufficient current to drive most output devices directly.
- **Need for Interfacing Components:** Due to voltage and current constraints, intermediate components such as transistors, driver ICs (like ULN2003), or relays are necessary to drive heavier loads.
- **Common Output Devices:**
  - LEDs (Light Emitting Diodes)
  - 7-Segment Displays
  - LCDs (Liquid Crystal Displays)
  - Relays (to control AC/DC loads)

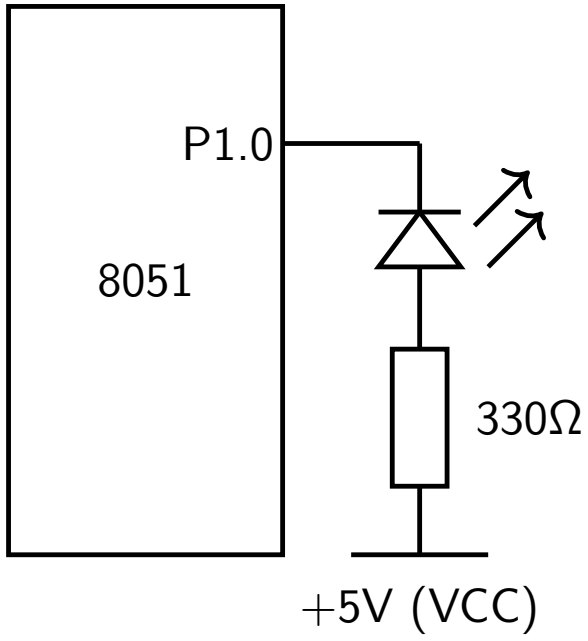
# LED Interfacing with 8051

## Basic Concept

An LED (Light Emitting Diode) is a semiconductor device that emits light when current passes through it. It requires a current limiting resistor to prevent damage.

- **Current Sinking Configuration (Active Low):**
  - The LED's anode is connected to VCC through a resistor.
  - The cathode is connected to an 8051 port pin.
  - The 8051 pin must output a logic '0' (LOW) to turn the LED ON.
  - This is preferred since 8051 pins can sink more current than they can source.
- **Current Sourcing Configuration (Active High):**
  - The LED's anode is connected to an 8051 port pin through a resistor.
  - The cathode is connected to ground.
  - The 8051 pin must output a logic '1' (HIGH) to turn the LED ON.

# LED Interfacing Example Circuit




## Assembly Code Snippet (Blinking LED)

```
BACK: CLR P1.0      ; Turn LED ON (Active Low)
      ACALL DELAY   ; Wait for some time
      SETB P1.0     ; Turn LED OFF
      ACALL DELAY   ; Wait for some time
      SJMP BACK     ; Repeat
```

# 7-Segment LED Display Interfacing

## Overview

A 7-segment display consists of seven LEDs arranged in a figure '8', plus an optional decimal point (DP). It is used to display numbers 0-9 and some hex characters (A-F).

- **Segments:** Labeled a, b, c, d, e, f, g, and dp.
- **Common Anode (CA):**
  - All anodes are connected together to +5V (VCC).
  - A logic '0' is required on the corresponding segment pin to turn it ON.
- **Common Cathode (CC):**
  - All cathodes are connected together to GND (0V).
  - A logic '1' is required on the corresponding segment pin to turn it ON.

# 7-Segment Display: Generating Digits

Assuming a **Common Anode** configuration connected to Port 1 (P1.0 to P1.7 = a to dp):

| Digit | dp (P1.7) | g (P1.6) | f | e | d | c | b | a (P1.0) | Hex Cod |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | C0H |
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | F9H |
| 2 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | A4H |
| 3 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | B0H |
| 4 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 99H |
| 5 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 92H |

## Programming Technique

Usually, an array (lookup table) containing these hex codes is stored in memory, and the code uses indexed addressing (e.g., `MOVC A, @A+DPTR`) to fetch and output the correct code.

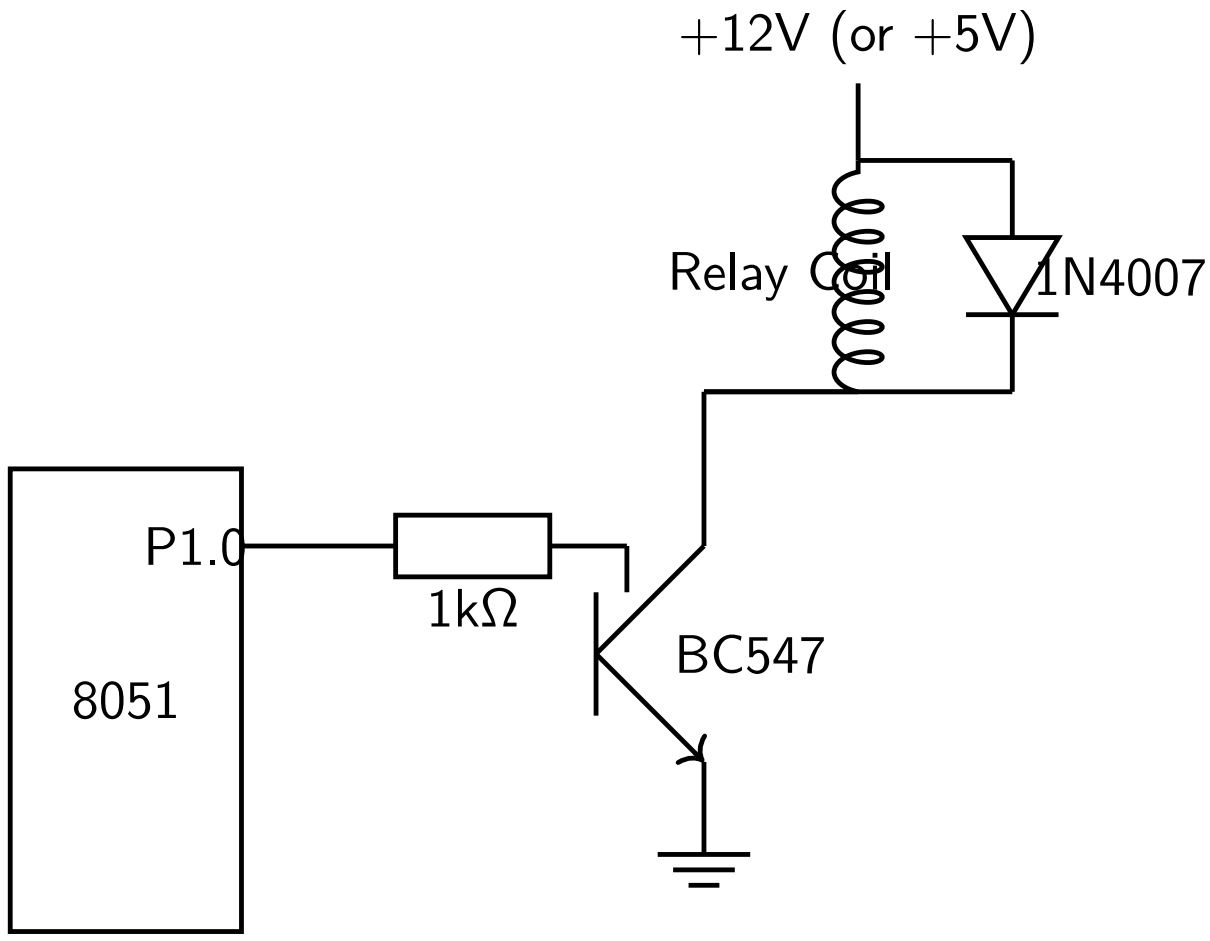
# Relay Interfacing with 8051

## What is a Relay?

A relay is an electromechanical switch operated by a relatively small electrical current that can turn on or off a much larger current (e.g., AC mains 230V appliances).

- **Problem:** An 8051 cannot drive a relay directly because the relay coil requires more current (e.g., 50-100 mA) and voltage (e.g., 5V, 12V) than an 8051 pin can provide (a few mA at 5V).
- **Solution:** Use a driver circuit.
  - **Transistor Switch:** A NPN transistor (like BC547) configured as a switch is commonly used. The 8051 pin drives the base of the transistor.
  - **Freewheeling Diode:** When the relay coil is de-energized, it generates a high back EMF spike that can damage the transistor. A diode (e.g., 1N4007) is placed in parallel with the coil (reverse biased) to discharge this energy.
  - **Driver IC:** A ULN2003 (Darlington pair array) can be used to drive multiple relays.

# Relay Interfacing Circuit Diagram




To turn the relay **ON**, output `logic 1` to P1.0. This biases the transistor ON, completing the circuit for the relay coil.

# LCD Interfacing with 8051 (16x2 Alphanumeric)

## 16x2 LCD Overview

A 16x2 LCD can display 16 characters per line on 2 lines. It utilizes the Hitachi HD44780 controller, making it standard for interfacing.

- **Important Pins:**
  - **VSS, VDD:** Ground and +5V respectively.
  - **VEE (Contrast):** Connected to a potentiometer to adjust display contrast.
  - **RS (Register Select):** `RS=0` selects Command Register. `RS=1` selects Data Register.
  - **R/W (Read/Write):** `R/W=0` for Write operations (most common). `R/W=1` for Read.
  - **E (Enable):** A high-to-low pulse is required to latch data or command into the LCD.
  - **D0-D7:** 8-bit bidirectional data bus.

# LCD Operation Modes & Interfacing Connections

- **Operation Modes:**
  - **8-bit Mode:** All 8 data lines (D0-D7) are used. Data is sent as an 8-bit byte. Faster, but requires 11 pins of the 8051 (8 data + 3 control).
  - **4-bit Mode:** Only 4 data lines (D4-D7) are used. Data is sent as two 4-bit nibbles (higher nibble, then lower). Slower, but saves 4 pins on the 8051.
- **Typical Connections (8-bit Mode):**
  - LCD Data Bus (D0-D7) $\rightarrow$ 8051 Port 2 (P2.0 - P2.7)
  - LCD RS pin $\rightarrow$ 8051 P3.0
  - LCD R/W pin $\rightarrow$ Ground (permanently configured for writing)
  - LCD E pin $\rightarrow$ 8051 P3.1

# Programming the LCD (Initialization)

Before sending data, the LCD must be initialized with specific command codes.

## Common Commands (Sent with RS=0)

- `38H`: Initialize LCD in 8-bit mode, 2 lines, 5x7 matrix.
- `0CH`: Display ON, Cursor OFF.
- `0EH`: Display ON, Cursor Blinking.
- `01H`: Clear display screen.
- `06H`: Increment cursor (shift cursor to right).
- `80H`: Force cursor to beginning of 1st line.
- `C0H`: Force cursor to beginning of 2nd line.

# Subroutines for LCD Operation

## Command Write Subroutine

1. Load command byte into Data Port (e.g., P2).
2. Clear RS pin (RS=0) to select Command Register.
3. Clear R/W pin (R/W=0) to select Write operation.
4. Set Enable pin (E=1), wait briefly, clear Enable pin (E=0) to create a High-to-Low pulse.
5. Wait for the LCD to process the command (Delay).

## Data Write Subroutine

1. Load character ASCII byte into Data Port (e.g., P2).
2. Set RS pin (RS=1) to select Data Register.
3. Clear R/W pin (R/W=0) to select Write operation.
4. Generate a High-to-Low pulse on Enable pin (E).
5. Wait for the LCD to process the character (Delay).

# Introduction to DAC (Digital to Analog Converter)

- **Purpose:** Converts digital signals (binary data) processed by microcontrollers into continuous analog signals (voltage or current).
- **Applications:** Audio processing, motor speed control, power supply control, and waveform generation.
- **DAC0808:** An 8-bit monolithic digital-to-analog converter.
- **Output Type:** Current output. To convert this current into voltage, an external operational amplifier (Op-Amp, e.g., LM741) is typically used as a current-to-voltage (I-to-V) converter.

## Resolution

For an 8-bit DAC, there are $2^8 = 256$ possible output levels. Resolution $= V_{ref}/256$.

# DAC0808 Features and Pin Diagram

**Key Features:**

- 8-bit resolution.
- Fast settling time: 150 ns.
- Relative accuracy: $\pm 0.19\%$ maximum error.
- Requires reference current/voltage to operate.

**Important Pins:**

- **D0-D7:** Digital input data pins (D0 is LSB, D7 is MSB).
- $V_{CC}$, $V_{EE}$**:** Power supply pins ($+5V$ and $-15V$ typical).
- $I_{OUT}$**:** Analog current output.
- $V_{REF(+)}$ **and** $V_{REF(-)}$**:** Reference voltage inputs to set the full-scale output current.

# Interfacing DAC0808 with 8051

- **Data Connection:** The 8 digital inputs (D0-D7) of the DAC0808 are connected to one of the 8051's ports, commonly **Port 1** (P1.0 - P1.7) or **Port 2**.
- **Current to Voltage:** The $I_{OUT}$ pin of the DAC is connected to the inverting input of an Op-Amp (like LM741) with a feedback resistor to produce an analog voltage output.
- **Algorithm for Waveform Generation:**
  1. Output a digital value to the port connected to the DAC.
  2. Call a delay (determines frequency).
  3. Change the digital value according to the desired waveform (e.g., toggle for square wave, increment/decrement for triangular wave).
  4. Repeat continuously.

# Generating a Square Wave

## Assembly Program

This program generates a square wave with a 50% duty cycle by toggling the DAC input between 00H (0V) and FFH (Full Scale Voltage). Assume DAC is connected to Port 1.

```
ORG 0000H
MAIN:   MOV P1, #00H      ; ...
        ACALL DELAY       ; Call delay for low time
        MOV P1, #0FFH     ; ...
        ACALL DELAY       ; Call delay for high time
        SJMP MAIN         ; ...

DELAY:  MOV R2, #255      ; ...
AGAIN:  DJNZ R2, AGAIN
        RET
END
```

# Generating a Triangular Wave

## Assembly Program

This program generates a triangular wave by ramping up the output to max (FFH) and then ramping it down to min (00H).

```
ORG 0000H
MAIN:   MOV A, #00H         ; Start with 00H
UP:     MOV P1, A           ; Send to DAC
        INC A               ; Increment A (ramp up)
        CJNE A, #0FFH, UP; ...

DOWN:   MOV P1, A           ; Send to DAC
        DEC A               ; Decrement A (ramp down)
        CJNE A, #00H, DOWN; ...
        SJMP MAIN           ; Repeat sequence
END
```

# Introduction to ADC0804 (Analog to Digital Converter)

- **Purpose:** Converts continuous analog signals (like temperature, pressure from sensors) into discrete digital numbers that the microcontroller can process.
- **ADC0804:** A commonly used 8-bit successive approximation analog-to-digital converter.
- **Resolution:** 8-bit means it maps the input voltage to one of 256 discrete digital values (00H to FFH).
- **Step Size:** Minimum change in analog voltage required to change the digital output by 1 LSB.
  Step Size $= \frac{V_{ref}}{2^n} = \frac{5V}{256} \approx 19.53$ mV

# ADC0804 Features and Control Pins

- **Clock:** It has a built-in clock generator (requires an external RC circuit, usually $R = 10k\Omega, C = 150pF$ for approx $606kHz$).
- **Vin(+) & Vin(-):** Differential analog inputs. Single-ended input connects Vin(-) to ground and signal to Vin(+).
- **Control Pins (Active Low):**
  - $\overline{CS}$ **(Chip Select):** Must be low to activate the ADC.
  - $\overline{RD}$ **(Read):** High-to-Low pulse enables the output buffers to read data.
  - $\overline{WR}$ **(Write):** Low-to-High pulse starts the conversion process (Start of Conversion - SOC).
  - $\overline{INTR}$ **(Interrupt):** Goes low when conversion is complete (End of Conversion - EOC).

# Interfacing ADC0804 with 8051

## Connection Scheme

- **Data Pins (D0-D7):** Connected to Port 1 (P1.0 - P1.7) to read the 8-bit digital output.
- **Control Pins:** Connected to Port 2.
  - $\overline{RD} \rightarrow$ P2.5
  - $\overline{WR} \rightarrow$ P2.6
  - $\overline{INTR} \rightarrow$ P2.7
  - $\overline{CS} \rightarrow$ Grounded (if ADC is the only device) or controlled via another port pin.

## Steps to Read Data

1. Make $\overline{CS} = 0$ (Select Chip).
2. Apply a Low-to-High pulse on $\overline{WR}$ to start conversion.
3. Monitor $\overline{INTR}$ pin. Wait until it goes low (Conversion complete).
4. Apply a High-to-Low pulse on $\overline{RD}$ to bring data to ADC pins.
5. Read data from the data port (Port 1).

# ADC0804 Interfacing Assembly Program

```
; ...
MYDATA EQU P1
RD     BIT P2.5
WR     BIT P2.6
INTR   BIT P2.7

ORG 0000H
MOV MYDATA, #0FFH ; Make P1 an input port

READ_ADC:
    CLR WR        ; WR = 0 (Start conversion pulse)
    SETB WR       ; WR = 1

WAIT:
    JB INTR, WAIT ; ...

    CLR RD        ; RD = 0 (Enable output buffer)
    MOV A, MYDATA ; ...
    SETB RD       ; RD = 1
```

# Summary of DAC and ADC Interfacing

- **DAC0808** requires continuous digital data updates from the microcontroller to generate continuous analog waveforms.
- External Op-Amp is critical for DAC0808 to convert output current to voltage.
- **ADC0804** requires a precise handshaking sequence ($\overline{WR}$ pulse to start, wait for $\overline{INTR}$, $\overline{RD}$ pulse to read).
- Both components allow the digital 8051 microcontroller to interact with the analog real world, enabling applications like sensor monitoring and signal generation.

# Introduction to DAC0808

- **DAC0808** is an 8-bit Digital-to-Analog Converter (DAC).
- Converts an 8-bit digital input into a proportional analog current output.
- Internally uses an R-2R ladder network.
- Requires an external operational amplifier (Op-Amp) to convert the output current into a voltage.
- Supports a wide power supply range (typically $+5$V and $-15$V).
- Fast settling time, suitable for generating waveforms (square, triangular, sine).

# DAC0808 Pin Description

- **Pins 5-12 (A1-A8):** 8-bit digital input (A1 is MSB, A8 is LSB).
- **Pin 4 ($I_O$):** Analog current output.
- **Pin 2 (GND):** Ground reference.
- **Pin 13 ($V_{CC}$), Pin 3 ($V_{EE}$):** Positive and negative power supplies.
- **Pin 14 ($V_{REF(+)}$), Pin 15 ($V_{REF(-)}$):** Reference voltage input pins.

## Output Current Equation

$$I_O = I_{REF} \times \left(\frac{A_1}{2} + \frac{A_2}{4} + \cdots + \frac{A_8}{256}\right)$$

- Where $I_{REF} = \frac{V_{REF}}{R_{REF}}$

# Interfacing DAC0808 with 8051

- The 8 data input pins of DAC0808 are connected to one of the 8051 ports (e.g., Port 1).
- The current output pin $I_O$ (Pin 4) is connected to the inverting input of an Op-Amp (like LM741).
- The Op-Amp acts as a current-to-voltage converter.
- Output voltage $V_{\text{out}} = I_O \times R_f$ (where $R_f$ is the feedback resistor).
- By sending different digital values (00H to FFH) to Port 1, we can obtain varying analog voltages.

# Programming DAC0808: Square Wave Generation

## Assembly Logic

- Send 00H (0V) and FFH (Max Voltage) to Port 1 alternately with some delay.

**Code Snippet:**

```
BACK: MOV A, #00H          ; Load 00H for 0V
      MOV P1, A            ; Send to DAC
      ACALL DELAY          ; Wait to maintain 0V level
      MOV A, #0FFH         ; Load FFH for Max Voltage
      MOV P1, A            ; Send to DAC
      ACALL DELAY          ; Wait to maintain Max V level
      SJMP BACK            ; Repeat continuously
```

# Introduction to ADC0804

- **ADC0804** is an 8-bit Analog-to-Digital Converter (ADC).
- It uses the successive approximation method for conversion.
- Resolution of 8 bits means 256 steps for the input voltage range.
- Built-in clock generator; requires only an external resistor and capacitor to set the clock frequency.
- Works on a single +5V power supply.
- Conversion time depends on the clock frequency (typically around 100 $\mu$s).

# ADC0804 Pin Description and Control Signals

- **D0-D7 (Pins 11-18):** 8-bit digital data output.
- $V_{IN(+)}$ **&** $V_{IN(-)}$ **(Pins 6, 7):** Differential analog voltage inputs.
- **Control Pins:**
  - $\overline{\textbf{CS}}$ **(Chip Select, Pin 1):** Active low, enables the ADC chip.
  - $\overline{\textbf{RD}}$ **(Read, Pin 2):** Active low, enables the output buffers to read data.
  - $\overline{\textbf{WR}}$ **(Write, Pin 3):** Active low pulse starts the A/D conversion.
  - $\overline{\textbf{INTR}}$ **(Interrupt, Pin 5):** Active low, signals End of Conversion (EOC).

# Interfacing ADC0804 with 8051

- Data lines D0-D7 of ADC0804 are connected to an 8051 port (e.g., Port 1) configured as input.
- Control pins $\overline{RD}$, $\overline{WR}$, and $\overline{INTR}$ are connected to 8051 control pins (e.g., P2.5, P2.6, P2.7). $\overline{CS}$ can be grounded if only one ADC is used.
- **Conversion Steps:**
  1. Make $\overline{CS} = 0$ (if not grounded).
  2. Send a high-to-low pulse on $\overline{WR}$ to start conversion.
  3. Wait until $\overline{INTR}$ goes low (End of Conversion).
  4. Send a low pulse on $\overline{RD}$ to read the digital data from D0-D7.

# Programming ADC0804

## Assembly Code Snippet

Assuming $\overline{RD}$ = P2.5, $\overline{WR}$ = P2.6, $\overline{INTR}$ = P2.7, Data Port = P1 (Input). $\overline{CS}$ is assumed grounded.

**Code Snippet:**

```
MOV P1, #0FFH          ; Make P1 an input port
CLR P2.6               ; WR = 0
SETB P2.6              ; WR = 1 (L-to-H pulse starts
conversion)
WAIT: JB P2.7, WAIT      ; Wait till INTR = 0 (EOC)
CLR P2.5               ; RD = 0 (Read Enable)
MOV A, P1              ; Read ADC data into Accumulator
SETB P2.5              ; RD = 1
```

# Introduction to DC Motors

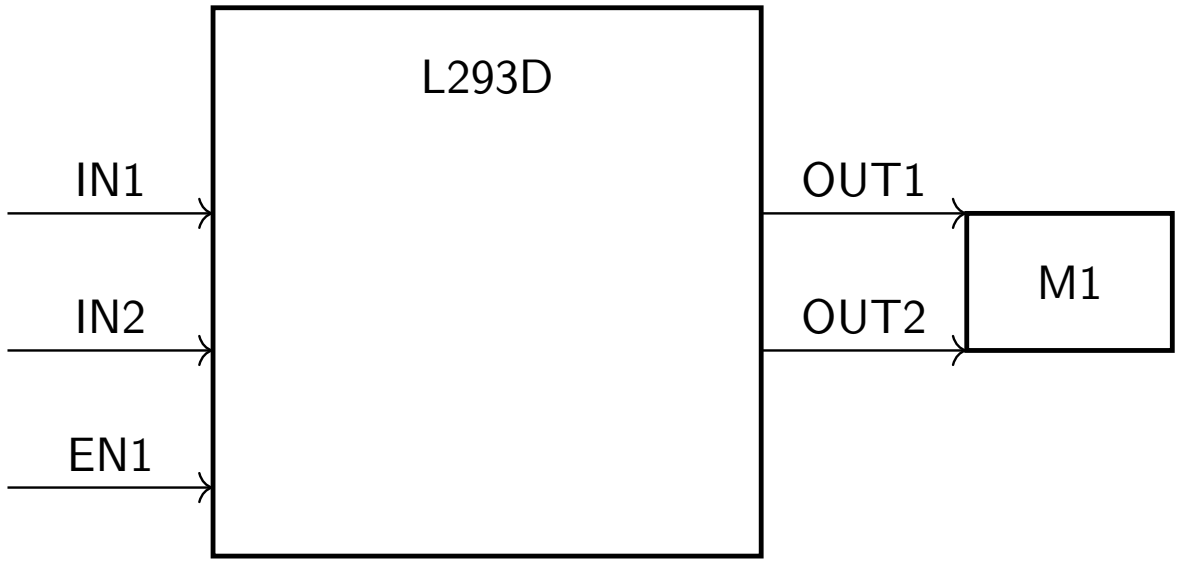
- A DC motor is an electrical machine that converts direct current electrical energy into mechanical energy.
- When current flows through the coil, a magnetic field is produced, which interacts with the permanent magnets to create rotation.
- **Direction of Rotation:** Determined by the polarity of the applied voltage. Reversing the polarity reverses the direction.
- **Speed Control:** Can be achieved by varying the average voltage applied to the motor, typically using Pulse Width Modulation (PWM).

## Microcontroller Limitation

Microcontrollers like 8051 cannot drive DC motors directly because motors require high current (e.g., 500 mA to a few Amps), whereas 8051 pins can source/sink only a few milliamperes.

# Interfacing DC Motor: Motor Drivers

- To bridge the gap between the 8051 and the motor, a **Motor Driver IC** is required.
- Common choice: **L293D** (Dual H-Bridge Motor Driver).
- **Features of L293D:**
  - Can drive two DC motors simultaneously.
  - Provides bidirectional drive currents of up to 600-mA.
  - Has internal clamp diodes to protect against back EMF.



# Controlling DC Motor Direction

Table: L293D Logic for Motor 1

| EN1 | IN1 (Pin 1) | IN2 (Pin 2) | Motor Status |
|---|---|---|---|
| 0 | X | X | Stop (Freewheeling) |
| 1 | 0 | 0 | Stop (Braking) |
| 1 | 1 | 0 | Clockwise Rotation |
| 1 | 0 | 1 | Anti-Clockwise Rotation |
| 1 | 1 | 1 | Stop (Braking) |

- IN1 and IN2 are connected to 8051 port pins (e.g., P1.0 and P1.1).
- EN1 is connected to 5V (for continuous operation) or an 8051 pin (for PWM speed control).

# C Code Example: DC Motor Direction

## C Code for Clockwise and Anti-Clockwise Control

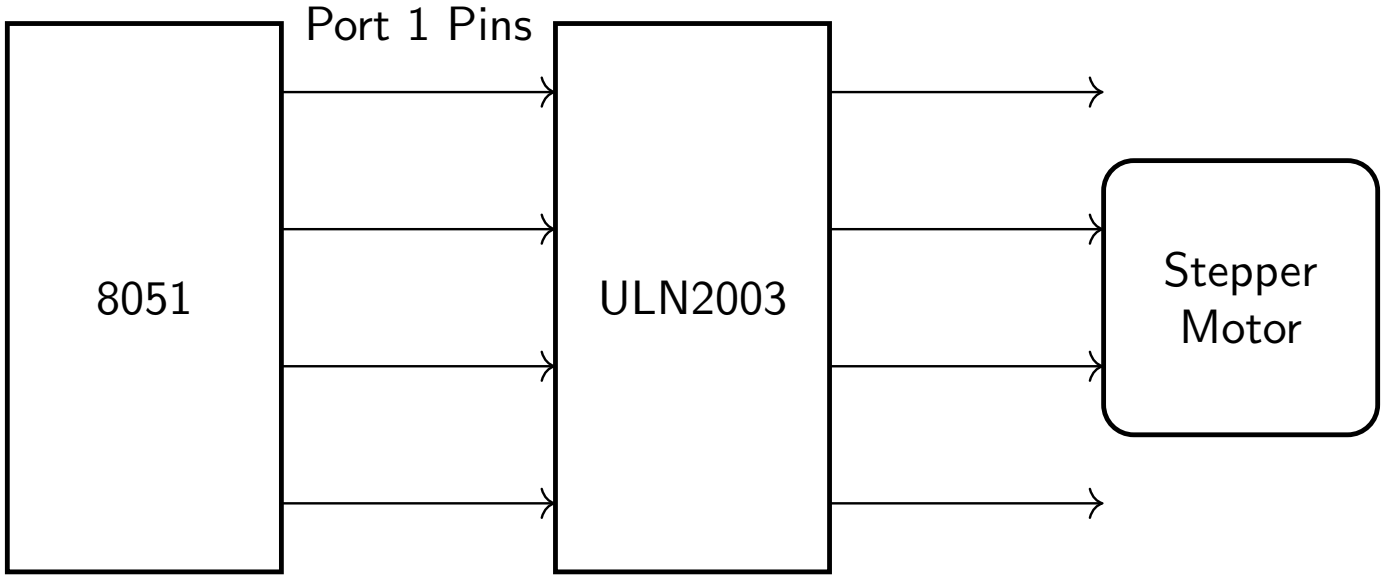
```
#include <reg51.h>
sbit IN1 = P1^0;  // Connect to L293D IN1
sbit IN2 = P1^1;  // Connect to L293D IN2

void delay(unsigned int ms) {
    unsigned int i, j;
    for(i = 0; i < ms; i++)
        for(j = 0; j < 1275; j++);
}

void main() {
    while(1) {
        IN1 = 1; IN2 = 0; // Clockwise
        delay(2000);
        IN1 = 0; IN2 = 0; // Stop
        delay(1000);
        IN1 = 0; IN2 = 1; // Anti-Clockwise
        delay(2000);
    }
}
```

# Introduction to Stepper Motors

- A stepper motor is a brushless DC electric motor that divides a full rotation into a number of equal steps.
- The motor's position can be commanded to move and hold at one of these steps without any feedback sensor (an open-loop controller).
- **Key Characteristics:**
  - **Step Angle:** The angle by which the rotor moves for each applied pulse (e.g., $1.8^\circ$, $7.5^\circ$).
  - Steps per revolution $= 360^\circ /$Step Angle.
- **Types based on winding:**
  - **Unipolar:** Has a center tap on the windings. Current flows in one direction (uses ULN2003).
  - **Bipolar:** No center tap. Current must be reversed to change magnetic polarity (uses H-bridge like L293D).

# Interfacing Stepper Motor (Unipolar)

- Unipolar stepper motors typically have 5 or 6 wires (4 coils and 1 or 2 common wires).
- The 8051 needs a driver like **ULN2003** (Darlington Transistor Array) to provide the necessary current.




## Operation

By energizing the coils in a specific sequence, the rotor turns step by step.

# Driving Sequences for Unipolar Stepper Motors

There are different sequences to drive a stepper motor:

Table: Full Step Sequence (2-Phase ON)

| Step | Coil A | Coil B | Coil C | Coil D | Hex Value |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 0x0C |
| 2 | 0 | 1 | 1 | 0 | 0x06 |
| 3 | 0 | 0 | 1 | 1 | 0x03 |
| 4 | 1 | 0 | 0 | 1 | 0x09 |

- Sends 2-phase on sequence. Higher torque but draws more power.
- For **Half-Step**, 8 sequences are used (interleaving 1-phase and 2-phase ON), doubling the resolution.

# C Code Example: Stepper Motor Control

## Rotating Stepper Motor continuously (Full Step)

```
#include <reg51.h>
#define motor_port P2

void delay(unsigned int ms) {
    unsigned int i, j;
    for(i=0; i<ms; i++)
        for(j=0; j<1275; j++);
}

void main() {
    unsigned char sequence[4] = {0x0C, 0x06, 0x03, 0...
    int i;
    while(1) {
        for(i = 0; i < 4; i++) {
            motor_port = sequence[i]; // ...
            delay(10); // Adjust delay to change speed
        }
    }
}
```

# Applications of Motor Interfacing

- **DC Motors:**
  - Robotics (wheels, drive systems).
  - Conveyor belts.
  - Cooling fans.
  - Automated doors and windows.
- **Stepper Motors:**
  - 3D Printers and CNC machines (precise positioning).
  - Hard disk drives and optical disc drives.
  - Robotics (robotic arms, precise angular movements).
  - Camera autofocus mechanisms.

## Summary

DC motors are excellent for continuous rotation and speed control, while stepper motors excel in applications requiring precise position control and repeatability.

# Objectives

## Learning Outcomes

By the end of this lecture, students will be able to:

- Understand the principles of DC and Stepper motors.
- Interface a DC motor with the 8051 using an H-bridge driver (e.g., L293D).
- Control the speed and direction of a DC motor.
- Interface a Stepper motor with the 8051 using a driver IC (e.g., ULN2003).
- Implement different stepping sequences (full-step, half-step) for precise control.

# Introduction to DC Motors

## Overview

A Direct Current (DC) motor is a rotary electrical machine that converts direct current electrical energy into mechanical energy.

- **Operation:** Relies on the forces produced by magnetic fields.
- **Direction:** Controlled by reversing the polarity of the applied voltage.
- **Speed:** Proportional to the applied voltage (often controlled via PWM).
- **Limitation:** A microcontroller cannot drive a DC motor directly because the motor requires high current (typically $> 250$mA), whereas the 8051 can only source/sink a few milliamperes.

# Motor Driver IC: L293D

## Why use a Motor Driver?

Motor drivers act as current amplifiers. They take a low-current control signal and provide a higher-current drive signal to the motor.

- The **L293D** is a popular 16-pin Motor Driver IC.
- Contains two H-bridge circuits.
- Can drive two DC motors simultaneously in both forward and reverse directions.
- Provides internal protection diodes to prevent back EMF from damaging the microcontroller.

# L293D Control Logic

Table: L293D Operation Logic for One Motor

| Enable (EN) | Input 1 (IN1) | Input 2 (IN2) | Motor Status |
|---|---|---|---|
| 0 | X | X | Stop (Freewheeling) |
| 1 | 0 | 0 | Stop (Brake) |
| 1 | 1 | 0 | Rotate Clockwise |
| 1 | 0 | 1 | Rotate Anti-Clockwise |
| 1 | 1 | 1 | Stop (Brake) |

- **EN:** Enable pin (must be High for operation).
- **IN1, IN2:** Logic inputs from the microcontroller.

# DC Motor Interfacing Circuit

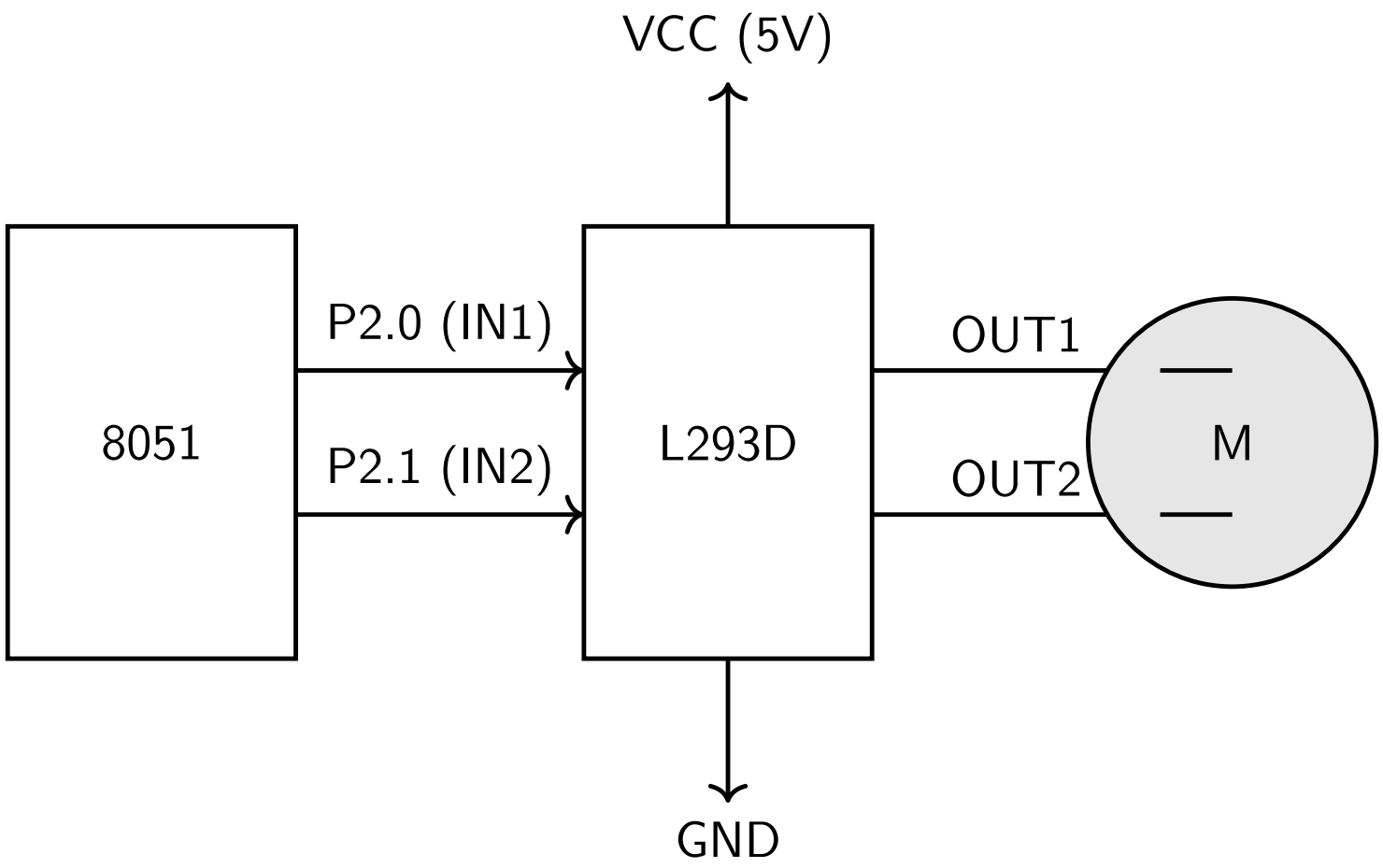



Figure: Interfacing 8051 with a DC Motor using L293D

# Speed Control using PWM

## Pulse Width Modulation (PWM)

PWM is a technique to control the average voltage supplied to the motor by rapidly switching the power on and off.

- **Duty Cycle:** The ratio of 'ON' time to the total time period.
- Duty Cycle (%) $= \frac{T_{ON}}{T_{ON}+T_{OFF}} \times 100$
- **Effect:** Higher duty cycle $\rightarrow$ higher average voltage $\rightarrow$ higher motor speed.
- **Implementation in 8051:** Can be generated using software delays or hardware timers.

# Introduction to Stepper Motors

## Overview

A Stepper motor is a brushless DC electric motor that divides a full rotation into a number of equal steps.

- **Precision:** The motor's position can be commanded to move and hold at one of these steps without any feedback sensor (open-loop control).
- **Step Angle:** The angle by which the rotor turns when one pulse is applied. Common step angles: 1.8° (200 steps/rev), 7.5° (48 steps/rev).
- **Types:** Permanent Magnet (PM), Variable Reluctance (VR), Hybrid.

# Types of Stepper Motors (Coil Configuration)

- **Unipolar Stepper Motor:**
  - Has a center tap on each of the two coils.
  - Current flows in only one direction through the coil halves.
  - Typically 5 or 6 wires.
  - Easier to control using simple transistor arrays like ULN2003.
- **Bipolar Stepper Motor:**
  - No center tap.
  - Current must be reversed in the coils to reverse the magnetic polarity.
  - Typically 4 wires.
  - Requires an H-bridge driver (like L293D or L298N) for control.

# Stepping Sequences

## Full-Step Sequence (Wave Drive)

Only one coil is energized at a time. Low power consumption but lower torque.

| Step | Coil A | Coil B | Coil C | Coil D |
|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 | 0 |
| 3 | 0 | 0 | 1 | 0 |
| 4 | 0 | 0 | 0 | 1 |

## Normal Full-Step (Two-Phase On)

Two coils energized simultaneously. Higher torque.

| Step | Coil A | Coil B | Coil C | Coil D |
|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 |
| 2 | 0 | 1 | 1 | 0 |
| 3 | 0 | 0 | 1 | 1 |
| 4 | 1 | 0 | 0 | 1 |

# Half-Step Sequence

## Half-Stepping

Alternates between energizing one coil and two coils. Doubles the number of steps per revolution, providing smoother rotation and higher resolution.

| Step | Coil A | Coil B | Coil C | Coil D |
|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 |
| 2 | 1 | 1 | 0 | 0 |
| 3 | 0 | 1 | 0 | 0 |
| 4 | 0 | 1 | 1 | 0 |
| 5 | 0 | 0 | 1 | 0 |
| 6 | 0 | 0 | 1 | 1 |
| 7 | 0 | 0 | 0 | 1 |
| 8 | 1 | 0 | 0 | 1 |

# Stepper Motor Interfacing Circuit

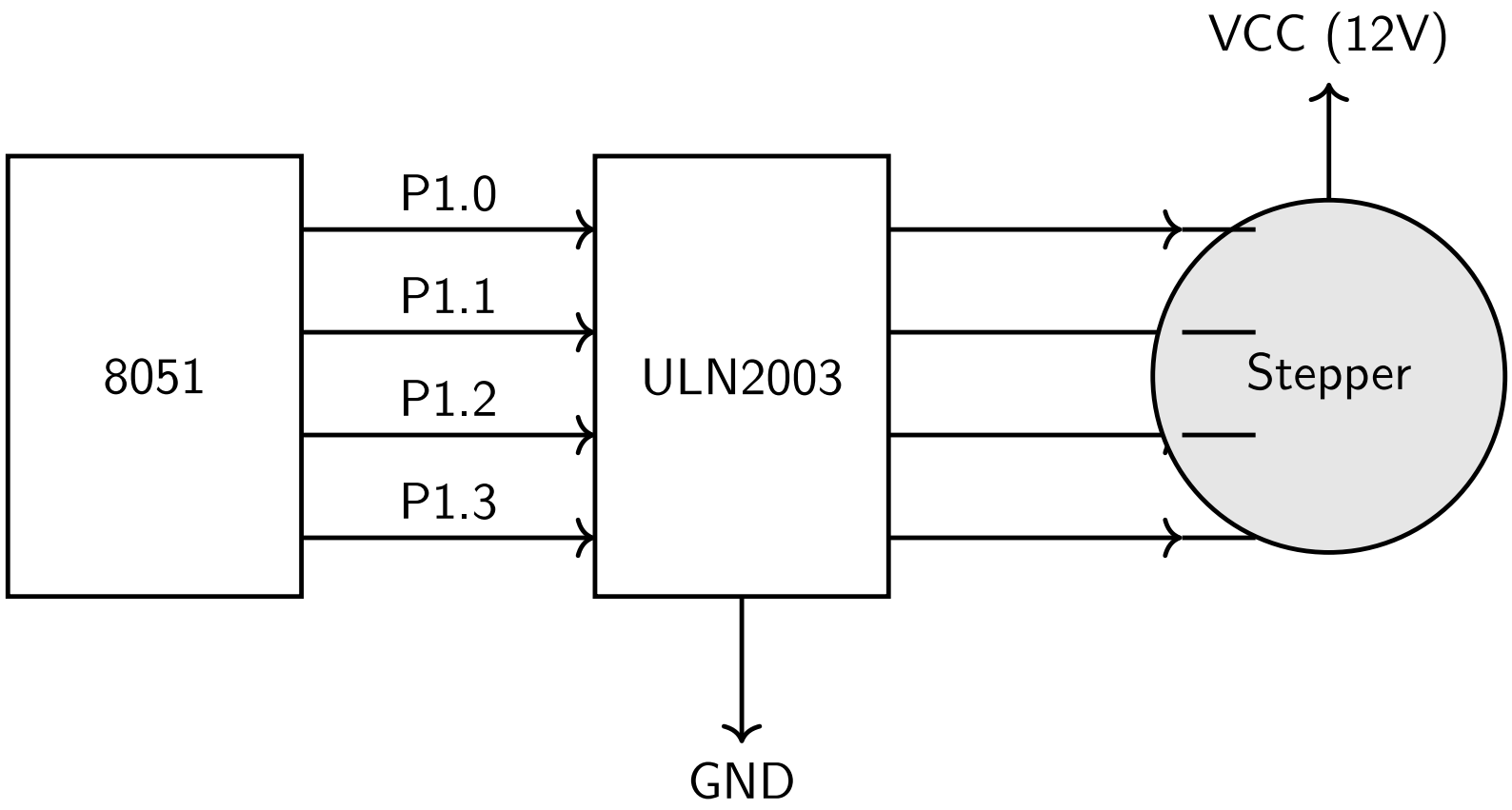



Figure: Interfacing a Unipolar Stepper Motor using ULN2003

# Key Interfacing Concepts

- **ULN2003:** A Darlington transistor array used to amplify current for a unipolar stepper motor. It acts as an open-collector driver.
- **Speed Control:** The speed of the stepper motor is controlled by the delay between consecutive steps. A shorter delay results in a higher speed.
- **Direction Control:** The direction is controlled by the sequence of steps. To reverse direction, simply reverse the stepping sequence.
- **Current Limitation:** Ensure the driver IC and power supply can handle the peak current requirements of the motor coils.

# Summary

## Key Takeaways

- **DC Motors** provide continuous rotation but require an H-bridge (like L293D) for bidirectional control due to high current needs.
- **PWM** is used to control the speed of a DC motor by varying the average voltage.
- **Stepper Motors** move in discrete steps, allowing precise open-loop position control.
- **Unipolar Stepper Motors** can be driven using a simple Darlington array like ULN2003.
- The choice of **stepping sequence** (wave, full-step, half-step) affects the torque, resolution, and power consumption of the stepper motor.

# Introduction to Microcontroller Applications

- Microcontrollers are ubiquitous in modern technology, serving as the "brains" of embedded systems.
- The 8051 microcontroller, due to its simplicity, robust architecture, and extensive support, is widely used in numerous applications.
- Key advantages that make microcontrollers ideal for these applications include:
  - Low cost and power consumption.
  - Compact size.
  - High reliability and flexibility.
- They interface with various sensors, actuators, and displays to control specific tasks in different domains.

# Applications in Industrial Automation

## Overview

Industrial environments demand robust, reliable, and real-time control systems. Microcontrollers like the 8051 are extensively used to automate processes and machinery.

- **Programmable Logic Controllers (PLCs):** Used for controlling manufacturing processes, assembly lines, and robotic devices.
- **Motor Control:** Precise control of stepper motors and DC motors in conveyor belts and CNC machines.
- **Data Acquisition Systems:** Collecting data from temperature, pressure, and flow sensors for monitoring and control.
- **Safety Systems:** Implementing failsafe mechanisms and emergency shutdown systems.

# Applications in Consumer Electronics

## Everyday Devices

Consumer electronics heavily rely on microcontrollers to provide smart, user-friendly features and energy efficiency.

- **Home Appliances:** Washing machines, microwave ovens, and refrigerators use microcontrollers for timing, temperature control, and user interface management.
- **Entertainment Systems:** Televisions, set-top boxes, and remote controls.
- **Personal Devices:** Digital cameras, smartwatches, and portable media players.
- **HVAC Systems:** Thermostats and air conditioning units for automated temperature regulation.

# Applications in the Automotive Industry

## Modern Vehicles

Modern cars contain dozens of microcontrollers, often communicating over networks like CAN (Controller Area Network) to manage various vehicle functions.

- **Engine Control Units (ECUs):** Optimizing fuel injection, ignition timing, and emissions control.
- **Safety Systems:** Anti-lock Braking Systems (ABS), airbag deployment, and Electronic Stability Control (ESC).
- **Comfort and Convenience:** Climate control, power windows, seat adjustment, and infotainment systems.
- **Advanced Driver-Assistance Systems (ADAS):** Parking sensors, adaptive cruise control, and lane departure warnings.

# Applications in Medical Devices

- The medical field requires highly precise and reliable equipment. Microcontrollers play a critical role in patient care and diagnostics.
- **Monitoring Equipment:** Heart rate monitors, blood pressure monitors, and pulse oximeters.
- **Diagnostic Tools:** Portable ultrasound machines and blood glucose meters.
- **Therapeutic Devices:** Infusion pumps, pacemakers, and automated external defibrillators (AEDs).
- **Laboratory Equipment:** Centrifuges and automated analyzers for sample testing.

# Other Prominent Fields

- **Aerospace and Defense:** Flight control systems, navigation equipment, and radar systems where reliability is paramount.
- **Internet of Things (IoT):** Smart home devices, smart meters, and connected sensors that gather and transmit data over the internet.
- **Telecommunications:** Modems, routers, and mobile phones.
- **Office Automation:** Printers, scanners, and photocopy machines for motor control and user interface.

# Summary of Microcontroller Applications

| **Field** | **Examples of Applications** |
|---|---|
| Industrial Automation | PLCs, Motor Control, Data Acquisition |
| Consumer Electronics | Washing Machines, Microwaves, Remotes |
| Automotive | ECUs, ABS, Airbag Systems, Infotainment |
| Medical | Blood Pressure Monitors, Pacemakers |
| IoT | Smart Plugs, Environmental Sensors |

Table: Summary of Microcontroller usage across various sectors.

# Introduction to Microcontroller Applications

## The Ubiquity of the 8051

The 8051 microcontroller is renowned for its versatility, simplicity, and low cost. Even today, it remains a foundational building block for countless embedded systems across various industries.

**Key Domains of Application:**

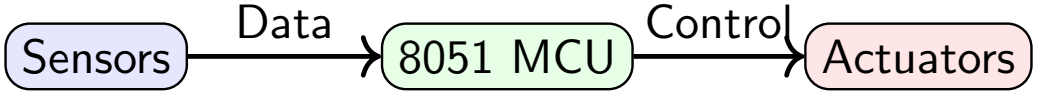
- **Industrial Automation:** Process control, robotics, assembly lines.
- **Consumer Electronics:** Home appliances, entertainment systems, smart devices.
- **Automotive Systems:** Engine control, dashboard displays, safety systems.
- **Healthcare & Medical:** Patient monitoring, diagnostic equipment.
- **Communication:** Networking devices, modems, mobile phones.

# Applications in Industrial Automation

## Why 8051 in Industry?

Industrial environments require robust, deterministic, and real-time control. The 8051's dedicated I/O ports and interrupt handling make it ideal for these tasks.

- **Motor Control:** Controlling the speed and direction of DC motors, stepper motors, and servo motors used in conveyor belts and robotic arms.
- **Process Monitoring:** Interfacing with sensors (temperature, pressure, flow) and taking automated actions (e.g., opening valves, triggering alarms).
- **Programmable Logic Controllers (PLCs):** The core logic for many small-scale PLCs was traditionally built around 8051-compatible architectures.



# Applications in Consumer Electronics

## Enhancing Daily Life

Microcontrollers are the "hidden brains" inside almost every modern household appliance, providing user interfaces, timing, and automated functions.

| **Appliance** | **Role of 8051 Microcontroller** |
|---|---|
| **Washing Machines** | Controls wash cycles, water level sensing, motor timing, and digital displays. |
| **Microwave Ovens** | Keypad scanning, LCD display control, timer management, and magnetron power control. |
| **Air Conditioners** | Temperature regulation via thermistors, fan speed control, and remote control signal decoding (IR). |
| **Smart Toys** | Voice recognition, motor control for movement, and interactive light displays. |

# Applications in Automotive Systems

## Driving Innovation

Modern vehicles contain dozens of microcontrollers communicating over networks (like CAN bus) to ensure safety, efficiency, and comfort. While 32-bit MCUs dominate complex tasks, 8-bit MCUs like the 8051 handle essential subsystems.

- **Body Control Modules:** Managing power windows, central locking systems, windshield wipers, and interior lighting.
- **Dashboard Instrumentation:** Driving stepper motors for speedometers and tachometers, and controlling digital instrument clusters.
- **Infotainment & Comfort:** HVAC (Heating, Ventilation, and Air Conditioning) control panels and basic audio system interfaces.
- **Sensor Interfaces:** Reading wheel speed sensors (for ABS), tire pressure sensors, and fuel level sensors.

# Applications in Healthcare and Medical Devices

## Precision and Reliability

Medical devices demand extreme reliability and accuracy. Microcontrollers are heavily utilized in both diagnostic and therapeutic equipment.

- **Patient Monitoring:** Portable ECG machines, pulse oximeters, and digital blood pressure monitors rely on MCUs for data acquisition and display.
- **Drug Delivery Systems:** Infusion pumps use microcontrollers to precisely control the flow rate of medication based on programmed parameters.
- **Diagnostic Tools:** Digital thermometers and blood glucose meters use MCUs to process sensor data and output human-readable results on LCDs.

**Design Considerations:** Medical applications require rigorous testing, fail-safe mechanisms, and often low power consumption for portable devices.

# Summary of 8051 Applications

- The **8051 microcontroller** is a deeply embedded workhorse across numerous sectors.
- **Industrial Automation:** Focuses on motor control, sensor interfacing, and process regulation.
- **Consumer Electronics:** Powers the logic and user interfaces of everyday household appliances.
- **Automotive:** Handles localized control tasks like windows, lighting, and dashboard indicators.
- **Healthcare:** Enables portable diagnostics and precise monitoring equipment.
- **Key Takeaway:** Despite the rise of advanced 32-bit processors, the 8-bit 8051 architecture remains highly relevant for tasks requiring straightforward logic, low cost, and reliable control.

# Introduction to 8051 Applications

## Overview

The 8051 microcontroller is widely used in various real-world applications due to its versatility, low cost, and ease of interfacing with external components.

**Applications Discussed in this Lecture:**

- Automatic Street Light Control (using LDR)
- Fire Alarm System (using Smoke Sensor & Buzzer)
- Water Level Controller (with Motor Control)
- Digital Stopwatch (Start, Stop, Reset)

# Automatic Street Light Control

## Concept

An Automatic Street Light Controller uses a Light Dependent Resistor (LDR) to detect the ambient light level and turns the streetlights ON or OFF accordingly, saving energy.

**Components Required:**

- 8051 Microcontroller
- LDR (Light Dependent Resistor)
- ADC (Analog to Digital Converter, e.g., ADC0804) or a Comparator (e.g., LM358)
- Relay Module (to switch the AC load)
- Transistor (e.g., BC547) to drive the relay

# Interfacing Street Light Control with 8051

## Working Mechanism

- The LDR forms a voltage divider circuit. Its resistance increases in darkness and decreases in light.
- A comparator or ADC converts the analog voltage into a digital signal (High/Low).
- The 8051 reads this digital signal.
- If the signal indicates "Darkness", the 8051 sends a HIGH signal to the relay driver to turn ON the street light.
- If "Daylight" is detected, it sends a LOW signal, turning OFF the light.

# Fire Alarm System

## Concept

A Fire Alarm System uses sensors to detect the presence of fire or smoke in an environment and triggers an alarm (buzzer) to alert occupants.

**Components Required:**

- 8051 Microcontroller
- Smoke/Gas Sensor (e.g., MQ-2) or Temperature Sensor (e.g., LM35)
- Buzzer (for audible alert)
- Transistor to drive the buzzer
- LED Indicators

# Interfacing Fire Alarm System with 8051

## Working Mechanism

- The MQ-2 smoke sensor provides an analog output or a digital output (via an onboard comparator).
- The digital output pin of the sensor is connected to an input pin of the 8051.
- The microcontroller continuously polls this input pin.
- Upon detecting smoke/gas (logic HIGH from sensor), the 8051 activates the buzzer and warning LEDs.
- When the smoke clears, the buzzer is deactivated.

# Water Level Controller

## Concept

A Water Level Controller monitors the level of water in a tank and automatically switches the water pump (motor) ON when the tank is empty and OFF when it is full.

**Components Required:**

- 8051 Microcontroller
- Conductive Probes or Float Switches (for level sensing)
- Relay Module (to control the water pump)
- LEDs (for Low, Medium, High level indication)

# Interfacing Water Level Controller with 8051

## Working Mechanism

- Sensors are placed at different levels in the tank (e.g., Bottom, Middle, Top).
- As water reaches a sensor, the circuit completes, sending a logic signal to the 8051.
- **Tank Empty condition:** The 8051 turns ON the relay to start the pump.
- **Intermediate Levels:** The 8051 updates the LED indicators.
- **Tank Full condition:** The 8051 turns OFF the relay to stop the pump, preventing overflow.

# Digital Stopwatch

## Concept

A Digital Stopwatch measures elapsed time with precision. It typically includes buttons for Start, Stop, and Reset operations, and uses a multiplexed 7-segment display or an LCD.

**Components Required:**

- 8051 Microcontroller
- Push Buttons (Start, Stop, Reset)
- Display (e.g., 4-digit Multiplexed 7-Segment Display or 16x2 LCD)
- Crystal Oscillator (for accurate timing)

# Implementing Digital Stopwatch with 8051

## Working Mechanism

- **Timing Generation:** 8051 internal timers (Timer 0 or Timer 1) are configured in Mode 1 or Mode 2 to generate precise delays (e.g., 10ms interrupts).
- **Button Inputs:** External interrupts or polled I/O pins monitor the Start, Stop, and Reset buttons.
- **Start:** Enables the timer.
- **Stop:** Disables the timer.
- **Reset:** Clears the time variables and updates the display to 00:00.
- **Display Update:** The time variables (milliseconds, seconds, minutes) are continuously refreshed on the display.

# Summary

## Key Takeaways

- **Street Light Control:** Uses LDR for ambient light detection and relays for switching.
- **Fire Alarm System:** Integrates smoke sensors (MQ-2) and buzzers for safety alerts.
- **Water Level Controller:** Automates pump operation using level probes and relay interfacing.
- **Digital Stopwatch:** Demonstrates the application of 8051 timers, interrupts, and multiplexed displays for accurate timekeeping.

# Introduction to 8051 Applications

## Overview

The 8051 microcontroller is widely used in various embedded system applications due to its versatility and rich set of peripherals (timers, serial ports, interrupts, I/O ports).

**Applications covered in this lecture:**

- **Automatic Street Light Control** using LDR sensor
- **Fire Alarm System** using smoke sensor and buzzer
- **Water Level Controller** with motor control and indicators
- **Digital Stopwatch** with start/stop/reset functionality

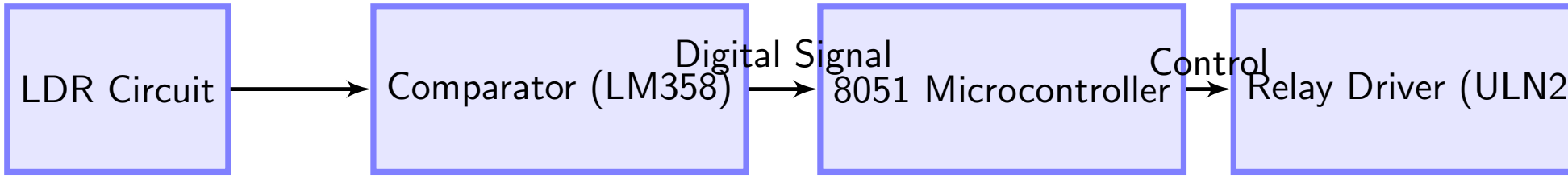
# Automatic Street Light Control: Concept

## Principle of Operation

Automatic street lights turn ON during the night and OFF during the day to save electricity and reduce manual intervention.

- **Key Component:** Light Dependent Resistor (LDR)
- **LDR Characteristics:**
  - Resistance decreases when light intensity increases (Daytime).
  - Resistance increases when it is dark (Nighttime).
- **Signal Conditioning:** Since the 8051 requires digital inputs, an Op-Amp (like LM358) in comparator mode or an ADC is used to convert the analog LDR signal into a digital HIGH/LOW.

# Automatic Street Light Control: Interfacing




## System Flow

1. LDR senses ambient light.
2. Comparator sends `HIGH` (dark) or `LOW` (light) to an 8051 pin (e.g., `P1.0`).
3. The 8051 reads the pin and outputs a signal to a relay driver (e.g., `P2.0`).
4. Relay toggles the high-voltage street light ON or OFF.

# Fire Alarm System: Concept

## Objective

Detect the presence of fire or smoke in a facility and alert the occupants through an audible buzzer or visual indicators.

**Core Components:**

- **Sensors:** Smoke sensor (e.g., MQ-2) or Flame sensor.
- **Microcontroller:** 8051 for processing the sensor data.
- **Output Devices:** Buzzer for audio alert, LEDs for visual warning.

## Sensor Working

The smoke/gas sensor's analog output is fed into a comparator (on-board module). When smoke crosses a threshold, the module outputs a digital `HIGH`.

# Fire Alarm System: Implementation

- **Input:** Smoke sensor DOUT connected to `P1.1`.
- **Output:** Buzzer connected to `P2.1` (via transistor if high current is needed).

| Condition | Sensor Output | 8051 Action | Buzzer State |
|---|---|---|---|
| No Smoke | `LOW` (0) | Set P2.1 = 0 | OFF |
| Smoke Detected | `HIGH` (1) | Set P2.1 = 1 | ON (Beep) |

Table: Fire Alarm Logic Truth Table

*Note: Proper debouncing and hysteresis logic in software can prevent false triggering.*

# Water Level Controller: Concept

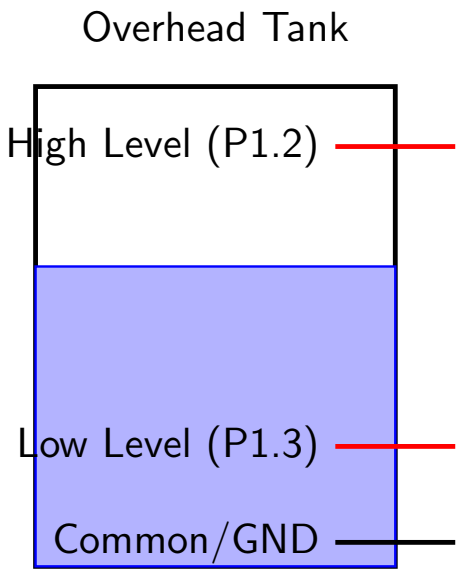
## Purpose

Automatically control a water pump to maintain the water level in an overhead tank, preventing overflow and dry running.

**System Requirements:**

- **Sensors:** Probes or ultrasonic sensors for different levels (Low, High).
- **Actuator:** Water Pump Motor (controlled via Relay).
- **Indicators:** LEDs showing Empty, Half, Full states.

# Water Level Controller: Control Logic




## Operational States

- **Tank Empty (Below Low Level):** Both probes OPEN. 8051 turns ON the Motor Relay.
- **Tank Filling:** Water touches Low Level, but not High. Motor stays ON.
- **Tank Full (Touches High Level):** Both probes shorted to GND. 8051 turns OFF the Motor Relay.
- Motor remains OFF until water drops below the Low Level again.

# Digital Stopwatch: Concept

## Overview

A digital stopwatch measures time intervals. It requires precise timing, display mechanisms, and user inputs to control the state.

**Components:**

- **Microcontroller:** 8051 (uses internal Timers).
- **Display:** 7-Segment Displays (Multiplexed) or LCD (16x2).
- **Buttons:** Three push-buttons for **Start**, **Stop**, and **Reset**.

**Timing Mechanism:** Timer 0 or Timer 1 of the 8051 is configured in Mode 1 (16-bit timer) or Mode 2 (8-bit auto-reload) to generate accurate delays (e.g., 10 ms or 1 second interrupts).

# Digital Stopwatch: Functionality

- **Start Button:** Enables the 8051 timer interrupt (`TR0 = 1`). The display counter increments.
- **Stop Button:** Disables the timer (`TR0 = 0`). The display holds the current value.
- **Reset Button:** Sets the counter variables (MM:SS:ms) to 0 and updates the display.

## Software Implementation Steps

1. Configure external interrupts or polling loop for the push-buttons.
2. Set up a timer to overflow at a known frequency (e.g., 100 Hz for 10ms resolution).
3. In the ISR (Interrupt Service Routine), increment the millisecond counter.
4. Handle carry-over: milliseconds $\rightarrow$ seconds $\rightarrow$ minutes.
5. Update the multiplexed 7-segment display continuously in the main loop.