Microcontroller
8051

Prof. Bandgar S S
K.T.H.M College, Nashik - 422002
Why do we need to learn Microprocessors/controllers?

- The microprocessor is the core of computer systems.
- Nowadays many communication, digital entertainment, portable devices, are controlled by them.
- A designer should know what types of components he needs, ways to reduce production costs and product reliable.
Different aspects of a microprocessor/controller

- Hardware : Interface to the real world

- Software : Order how to deal with inputs
Three criteria in Choosing a Microcontroller

1. meeting the computing needs of the task efficiently and cost effectively
   • speed, the amount of ROM and RAM, the number of I/O ports and timers, size, packaging, power consumption
   • easy to upgrade
   • cost per unit

2. availability of software development tools
   • assemblers, debuggers, C compilers, emulator, simulator, technical support

3. wide availability and reliable sources of the microcontrollers.
Contents:

- Introduction
- Block Diagram and Pin Description of the 8051 Registers
- Memory mapping in 8051
- Stack in the 8051
- I/O Port Programming
- Timer
- Interrupt
The necessary tools for a microprocessor/controller

- CPU: Central Processing Unit
- I/O: Input /Output
- Bus: Address bus & Data bus
- Memory: RAM & ROM
- Timer
- Interrupt
- Serial Port
- Parallel Port
Microprocessors: General-purpose microprocessor

- CPU for Computers
- No RAM, ROM, I/O on CPU chip itself
- Example: Intel’s x86, Motorola’s 680x0
Microcontroller:

- A smaller computer
- On-chip RAM, ROM, I/O ports...
- Example: Motorola’s 6811, Intel’s 8051, Zilog’s Z8 and PIC 16X
Microprocessor vs. Microcontroller

**Microprocessor**
- CPU is stand-alone, RAM, ROM, I/O, timer are separate
- designer can decide on the amount of ROM, RAM and I/O ports.
- Expansive
- versatility
- general-purpose

**Microcontroller**
- CPU, RAM, ROM, I/O and timer are all on a single chip
- fix amount of on-chip ROM, RAM, I/O ports
- for applications in which cost, power and space are critical
- single-purpose
## Comparison

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>Microcontroller</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Data is in byte (8-bit) or word (16-bit) format</td>
<td></td>
</tr>
<tr>
<td>- Many instruction types and modes</td>
<td></td>
</tr>
<tr>
<td>- Hardware includes CPU only</td>
<td></td>
</tr>
<tr>
<td>- Mainly for processing</td>
<td></td>
</tr>
<tr>
<td>- Larger I/O scale – use system buses</td>
<td></td>
</tr>
<tr>
<td>- Memory is byte-oriented</td>
<td></td>
</tr>
<tr>
<td>- Instruction sets are more robust</td>
<td></td>
</tr>
<tr>
<td>- Data is in bits or byte format</td>
<td></td>
</tr>
<tr>
<td>- Less instruction types and modes</td>
<td></td>
</tr>
<tr>
<td>- Hardware includes CPU and peripherals</td>
<td></td>
</tr>
<tr>
<td>- Mainly for control</td>
<td></td>
</tr>
<tr>
<td>- Smaller I/O scale – use parallel and serial ports</td>
<td></td>
</tr>
<tr>
<td>- Memory is bit-oriented</td>
<td></td>
</tr>
<tr>
<td>- Instruction sets are more compact</td>
<td></td>
</tr>
</tbody>
</table>
External interrupts

Interrupt Control

CPU

OSC

On-chip ROM for program code

On-chip RAM

4 I/O Ports

Serial Port

Timer/Counter

Timer 1

Timer 0

Bus Control

4 I/O Ports

Address/Data

P0 P1 P2 P3

TxD RxD

Counter Inputs
The 8051 Block Diagram

- **Interrupt Control**
- **CPU**
- **OSC**
- **4K byte ROM**
- **128 byte RAM**
- **I/O Ports**
- **Serial Port**
- **External Interrupts**
- **Bus Control**
- **Counter Inputs**
- **Timer 1**
- **Timer 0**

- **TXD**
- **RXD**

- Addresses: P0, P2, P1, P3
<table>
<thead>
<tr>
<th>Feature</th>
<th>8051</th>
<th>8052</th>
<th>8031</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROM (on-chip program space in bytes)</td>
<td>4K</td>
<td>8K</td>
<td>0K</td>
</tr>
<tr>
<td>RAM (bytes)</td>
<td>128</td>
<td>256</td>
<td>128</td>
</tr>
<tr>
<td>Timers</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>I/O pins</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Serial port</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Interrupt sources</td>
<td>6</td>
<td>8</td>
<td>6</td>
</tr>
</tbody>
</table>
General Physical Features

- 4KB ROM
- 128 bytes internal RAM
  - 4 register banks of 8 bytes each (R0-R7)
  - 16 bytes of bit-addressable area
  - 80 bytes of general purpose memory
- Four 8-bit I/O ports (P0-P3)
- Two 16-bit timers (Timer0 & Timer1)
- One serial receiver-transmitter interface
- Five interrupt sources (2 external & 3 internal)
- One oscillator (generates clock signal)
Memory of 8051 can be increased externally:
- Increase memory space for codes (programs) by 64K
- Increase memory space for data by 64K

Boolean instructions work with 1 bit at a time

Assume clock frequency = 12MHz, it takes about 4 \( \mu \)s (i.e. 4 \( \times 10^{-6} \)s) to carry out a 8-bit multiplication instruction
Pin Description of the 8051

P1.0  1
P1.1  2
P1.2  3
P1.3  4
P1.4  5
P1.5  6
P1.6  7
P1.7  8
RST   9
(RXD)P3.0  10
(TXD)P3.1  11
(INT0)P3.2 12
(INT1)P3.3 13
(T0)P3.4  14
(T1)P3.5  15
(WR)P3.6  16
(RD)P3.7  17
XTAL2   18
XTAL1   19
GND     20

8051 (8031)

40 Vcc
39 P0.0(AD0)
38 P0.1(AD1)
37 P0.2(AD2)
36 P0.3(AD3)
35 P0.4(AD4)
34 P0.5(AD5)
33 P0.6(AD6)
32 P0.7(AD7)
31 EA/VPP
30 ALE/PROG
29 PSEN
28 P2.7(A15)
27 P2.6(A14)
26 P2.5(A13)
25 P2.4(A12)
24 P2.3(A11)
23 P2.2(A10)
22 P2.1(A9)
21 P2.0(A8)
The 8051 Logic Symbol

VSS  VCC  RST

XTAL1

P0.7  P0.6  P0.5  P0.4  P0.3  P0.2  P0.1  P0.0

EA

P1.7  P1.6  P1.5  P1.4  P1.3  P1.2  P1.1  P1.0

PSEN

ADDRESS AND DATA BUS

ALE

P2.7  P2.6  P2.5  P2.4  P2.3  P2.2  P2.1  P2.0

SECONDARY FUNCTIONS

PORT 1

PORT 2

PORT 3

RxD  TxD  INT0  INT1  T0  T1  WR  RD

Without alternate function

ADDRESS BUS
Some Important Pins

- **VCC** (pin 40 - provides supply voltage of +5V)
- **GND** (pin 20)
- **XTAL1 & XTAL2** (pins 19 & 18 - to crystal and then caps)
- **RST** (pin 9 - reset)
- **EA** (pin 31 - external access)
- **PSEN** (pin 29 - program store enable)
- **ALE** (pin 30 - address latch enable)
- **Ports 0-3**
Hardware Description

- Oscillator circuit
- Program counter (PC)
- Data pointer (DPTR)
- Accumulator ("A") register
- B register
- Flags
- Program status word (PSW)
- Internal memory (ROM, RAM, additional memory)
- Stack & stack pointer (SP)
- Special function register (SFR)
Using a quartz crystal oscillator
We can observe the frequency on the XTAL2 pin.
Oscillator Circuit

- The heart of the 8051
- Produces clock pulses
- Synchronize all 8051's internal operations

A single machine cycle consists of 12 crystal pulses!
Machine Cycle

- Machine cycle is the basic repetitive process that the CPU performs once it is powered on. A machine cycle consists of a fixed number of clock cycles (pulses). It is different for different kinds of CPU.
- The 8051 family needs 12 clock cycles for a machine cycle.
- The CPU takes one or more machine cycles to complete an instruction. More complex instructions require more number of machine cycles to complete the instruction. The number of machine cycles of the 8051 instructions are ranging from 1 to 4.
Example 4-1

Find the elapse time of the machine cycle for:
(a) XTAL = 11.0592 MHz
(b) XTAL = 16 MHz
(c) XTAL = 20 MHz

Solution:
(a) \( \frac{11.0592 \text{ MHz}}{12} = 921.6 \text{ kHz} \)
    Machine cycle = \( \frac{1}{921.6 \text{ kHz}} = 1.085 \mu\text{s} \)

(b) \( \frac{16 \text{ MHz}}{12} = 1.333 \text{ MHz} \)
    Machine cycle = \( \frac{1}{1.333 \text{ MHz}} = 0.75 \mu\text{s} \)

(c) \( \frac{20 \text{ MHz}}{12} = 1.667 \text{ MHz} \)
    Machine cycle = \( \frac{1}{1.667 \text{ MHz}} = 0.60 \mu\text{s} \)
Pins of 8051 (2/4)

- RST (pin 9): reset
  - It is an input pin and is active high (normally low).
    - The high pulse must be high at least 2 machine cycles.
  - It is a power-on reset.
    - Upon applying a high pulse to RST, the microcontroller will reset and all values in registers will be lost.
- Reset values of some 8051 registers
Figure (b). Power-On RESET Circuit

Vcc

+ 10 uF

8.2 K

30 pF

11.0592 MHz

30 pF

EA/VPP

X1

19

X2

18

RST

9
## RESET Value of Some 8051 Registers:

<table>
<thead>
<tr>
<th>Register</th>
<th>Reset Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>0000</td>
</tr>
<tr>
<td>ACC</td>
<td>0000</td>
</tr>
<tr>
<td>B</td>
<td>0000</td>
</tr>
<tr>
<td>PSW</td>
<td>0000</td>
</tr>
<tr>
<td>SP</td>
<td>0007</td>
</tr>
<tr>
<td>DPTR</td>
<td>0000</td>
</tr>
</tbody>
</table>

RAM are all zero.
Pins of 8051 (3/4)

- **/EA (pin 31)**: external access
  - There is no on-chip ROM in 8031 and 8032.
  - The /EA pin is connected to GND to indicate the code is stored externally.
  - /PSEN & ALE are used for external ROM.
  - For 8051, /EA pin is connected to Vcc.
  - “/” means active low.

- **/PSEN (pin 29)**: program store enable
  - This is an output pin and is connected to the OE pin of the ROM.
ALE (pin 30): address latch enable
  – It is an output pin and is active high.
  – 8051 port 0 provides both address and data.
  – The ALE pin is used for de-multiplexing the address and data by connecting to the G pin of the 74LS373 latch.

I/O port pins
  – The four ports P0, P1, P2, and P3.
  – Each port uses 8 pins.
  – All I/O pins are bi-directional.
The 8051 has four I/O ports

- Port 0 (pins 32-39): P0 (P0.0 ~ P0.7)
- Port 1 (pins 1-8): P1 (P1.0 ~ P1.7)
- Port 2 (pins 21-28): P2 (P2.0 ~ P2.7)
- Port 3 (pins 10-17): P3 (P3.0 ~ P3.7)

- Each port has 8 pins.
  - Named P0.X (X=0,1,...,7), P1.X, P2.X, P3.X
  - Ex: P0.0 is the bit 0 (LSB) of P0
  - Ex: P0.7 is the bit 7 (MSB) of P0
  - These 8 bits form a byte.

- Each port can be used as input or output (bi-direction).
Hardware Structure of I/O Pin

- Each pin of I/O ports
  - Internal CPU bus: communicate with CPU
  - A D latch store the value of this pin
    - D latch is controlled by “Write to latch”
      - Write to latch = 1: write data into the D latch
  - 2 Tri-state buffer:
    - TB1: controlled by “Read pin”
      - Read pin = 1: really read the data present at the pin
    - TB2: controlled by “Read latch”
      - Read latch = 1: read value from internal latch
  - A transistor M1 gate
    - Gate=0: open
    - Gate=1: close
Registers

Some 8-bit Registers of the 8051

Some 8051 16-bit Register
Program Counter (PC)

- **PC** is a 16-bit register
- **PC** is the only register that does not have an internal address
- Holds the address of the memory location to fetch the program instruction
- Program ROM may be **on the chip** at addresses 0000H to 0FFFH (4Kbytes), external to the chip for addresses that exceed 0FFFH
- Program ROM may be **totally external** for all addresses from 0000H to FFFFFH
- **PC** is **automatically incremented** (+1) after every instruction byte is fetched
**DPTR** is a 16-bit register

**DPTR** is made up of two 8-bit registers: **DPH** and **DPL**

**DPTR** holds the memory addresses for internal and external code access and external data access (eg. MOVCA, @A+DPTR     MOVX A, @DPTR     MOVX @DPTR, A)

**DPTR** is under the control of program instructions and can be specified by its 16-bit name, or by each individual byte name, **DPH** and **DPL**

**DPTR** does not have a single internal address; **DPH** and **DPL** are each assigned an address (83H and 82H)
**Stack and Stack Pointer (SP)**

- **SP** is a 8-bit register used to hold an internal RAM address that is called the “**top of the stack**”
- **Stack** refers to an area of internal RAM that is used in conjunction with certain opcodes to store and retrieve data quickly
- **SP** holds the internal RAM address where the last byte of data was stored by a stack operation
- When data is to be placed on the stack, the **SP** increments before storing data on the stack so that the stack **grows up** as data is stored
- As data is retrieved from the stack, the byte is read from the stack, and then the **SP** decrements to point to the next available byte of stored data
- **SP** = 07H after reset
**Stack in the 8051**

- The register used to access the stack is called **SP** (stack pointer) register.

- The stack pointer in the 8051 is only 8 bits wide, which means that it can take value 00 to FFH. When 8051 powered up, the SP register contains value 07.

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00H</td>
<td>Scratch pad RAM</td>
</tr>
<tr>
<td>07H</td>
<td>Register Bank 0 (Stack)</td>
</tr>
<tr>
<td>08H</td>
<td>Register Bank 1</td>
</tr>
<tr>
<td>10H</td>
<td>Register Bank 2</td>
</tr>
<tr>
<td>17H</td>
<td>Register Bank 3</td>
</tr>
<tr>
<td>18H</td>
<td>Bit-Addressable RAM</td>
</tr>
<tr>
<td>20H</td>
<td></td>
</tr>
<tr>
<td>2FH</td>
<td></td>
</tr>
<tr>
<td>30H</td>
<td></td>
</tr>
<tr>
<td>7FH</td>
<td></td>
</tr>
</tbody>
</table>

Register Bank 0

Register Bank 1

Register Bank 2

Register Bank 3
Stack Operation

Storing Data on the Stack (Increment then store)
Internal RAM (Get then decrement)
Getting Data From the Stack

SP = 0A
Address 0A
SP = 0A

SP = 09
Address 09
SP = 09

SP = 08
Address 08
SP = 08

SP = 07
Address 07
SP = 07
Example 2-8

Show the stack and stack pointer for the following. Assume the default stack area.

MOV R6, #25H
MOV R1, #12H
MOV R4, #0F3H
PUSH 6
PUSH 1
PUSH 4

After PUSH 6: SP = 07
After PUSH 1: SP = 08
After PUSH 4: SP = 0A
Example 2-10

Show the stack and stack pointer for the following.

MOV     SP, #5FH
MOV     R2, #25H
MOV     R1, #12H
MOV     R4, #0F3H
PUSH    2
PUSH    1
PUSH    4

<table>
<thead>
<tr>
<th></th>
<th>After PUSH 2</th>
<th>After PUSH 1</th>
<th>After PUSH 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>63</td>
<td>63</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>62</td>
<td>62</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>61</td>
<td>61</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>60</td>
<td>60</td>
</tr>
</tbody>
</table>

Start SP = 5F  SP = 60  SP = 61  SP = 62
Example 2-9

Examine the stack, show the contents of the registers and \( SP \) after execution of the following instruction. All values are in hex.

\[
\begin{align*}
\text{POP} & \quad 3 & \text{;POP stack into R3} \\
\text{POP} & \quad 5 & \text{;POP stack into R5} \\
\text{POP} & \quad 2 & \text{;POP stack into R2}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Start SP = 0B</th>
<th>After POP 3</th>
<th>After POP 5</th>
<th>After POP 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0B 54</td>
<td>0B 54</td>
<td>0B 54</td>
<td>0B 54</td>
</tr>
<tr>
<td>0A F9</td>
<td>0A F9</td>
<td>0A F9</td>
<td>0A F9</td>
</tr>
<tr>
<td>09 76</td>
<td>09 76</td>
<td>09 76</td>
<td>09 76</td>
</tr>
<tr>
<td>08 6C</td>
<td>08 6C</td>
<td>08 6C</td>
<td>08 6C</td>
</tr>
<tr>
<td>SP = 0A</td>
<td>SP = 09</td>
<td>SP = 08</td>
<td></td>
</tr>
</tbody>
</table>

05 ??
04 ??
03 ??
02 ??
Accumulator (A Register)

- Most versatile CPU register and is used for many operations, including addition, integer multiplication and division, and Boolean bit manipulations
- A register is also used for all data transfer between the 8051 and any external memory
B Register

- **B** register is used with the **A** register for multiplication and division operations (e.g. MUL AB   DIV AB)
- No other special function other than as a location where data may be stored
Flags

- Flags are **1-bit registers** provided to store the results of certain program instructions.
- Other instructions can test the condition of the flags and make decisions based on the flag states.
- Flags are grouped inside the **program status word (PSW)** and the **power control (PCON)** registers for convenient addressing.

**Math flags**: respond automatically to the outcomes of math operations (**CY, AC, OV, P**)

**User flags**: general-purpose flags that may be used by the programmer to record some event in the program (**F0, GF0, GF1**)
**Program Status Word (PSW)**

PSW contains the math flags, user program flag F0, and the register select bits (RS1, RS0) that identify which of the four general-purpose register banks is currently in use by the program.

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CY</td>
<td>AC</td>
<td>F0</td>
<td>RS1</td>
<td>RS0</td>
<td>OV</td>
<td>--</td>
<td>P</td>
</tr>
</tbody>
</table>
**Program Status Word (PSW)**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Symbol</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>CY</td>
<td>Carry Flag; used in arithmetic, JUMP, ROTATE, and BOOLEAN instruction</td>
</tr>
<tr>
<td>6</td>
<td>AC</td>
<td>Auxiliary carry flag; used for BCD arithmetic</td>
</tr>
<tr>
<td>5</td>
<td>F0</td>
<td>User flag 0</td>
</tr>
<tr>
<td>4</td>
<td>RS1</td>
<td>Register bank select bit 1</td>
</tr>
<tr>
<td>3</td>
<td>RS0</td>
<td>Register bank select bit 0</td>
</tr>
<tr>
<td>2</td>
<td>OV</td>
<td>Overflow flag; used in arithmetic instructions</td>
</tr>
<tr>
<td>1</td>
<td>--</td>
<td>Reserved for future use</td>
</tr>
<tr>
<td>0</td>
<td>P</td>
<td>Parity flag; shows parity of register A: 1 = Odd Parity</td>
</tr>
</tbody>
</table>
The 8051 maintains even parity with the accumulator A, ie; the number of ones in the accumulator together with the parity bit (in the program status word, PSW) is always even.
# Instruction that Affect Flag Bits

<table>
<thead>
<tr>
<th>Instruction</th>
<th>CY</th>
<th>OV</th>
<th>AC</th>
<th>Instruction</th>
<th>CY</th>
<th>OV</th>
<th>AC</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>ADD</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>ADDC</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>SETB C</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBB</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>CLR C</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL</td>
<td>0</td>
<td>X</td>
<td></td>
<td>CPL C</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIV</td>
<td>0</td>
<td>X</td>
<td></td>
<td>ANL C, bit</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DA</td>
<td>X</td>
<td></td>
<td></td>
<td>ORL C, bit</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RRC</td>
<td>X</td>
<td></td>
<td></td>
<td>ORL C, bit</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RLC</td>
<td>X</td>
<td></td>
<td></td>
<td>CJNE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV C, bit</td>
<td>X</td>
<td></td>
<td></td>
<td>Note: X can be 0 or 1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Special Function Registers (SFR)

- 8051 has 21 SFRs which occupy the addresses from 80H to FFH (128bytes)
- Not all of the addresses from 80H to FFH are used for SFRs
- Attempt to use the “empty” addresses may get unpredictable result
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC*</td>
<td>Accumulator</td>
<td>0E0H</td>
</tr>
<tr>
<td>B*</td>
<td>B register</td>
<td>0F0H</td>
</tr>
<tr>
<td>PSW*</td>
<td>Program status word</td>
<td>0D0H</td>
</tr>
<tr>
<td>SP</td>
<td>Stack pointer</td>
<td>81H</td>
</tr>
<tr>
<td>DPTR</td>
<td>Data pointer 2 bytes</td>
<td></td>
</tr>
<tr>
<td>DPL</td>
<td>Low byte</td>
<td>82H</td>
</tr>
<tr>
<td>DPH</td>
<td>High byte</td>
<td>83H</td>
</tr>
<tr>
<td>P0*</td>
<td>Port 0</td>
<td>80H</td>
</tr>
<tr>
<td>P1*</td>
<td>Port 1</td>
<td>90H</td>
</tr>
<tr>
<td>P2*</td>
<td>Port 2</td>
<td>0A0H</td>
</tr>
<tr>
<td>P3*</td>
<td>Port 3</td>
<td>0B0H</td>
</tr>
<tr>
<td>IP*</td>
<td>Interrupt priority control</td>
<td>0B8H</td>
</tr>
<tr>
<td>IE*</td>
<td>Interrupt enable control</td>
<td>0A8H</td>
</tr>
<tr>
<td>TMOD</td>
<td>Timer/counter mode control</td>
<td>89H</td>
</tr>
<tr>
<td>TCON*</td>
<td>Timer/counter control</td>
<td>88H</td>
</tr>
<tr>
<td>T2CON*</td>
<td>Timer/counter 2 control</td>
<td>0C8H</td>
</tr>
<tr>
<td>T2MOD</td>
<td>Timer/counter mode control</td>
<td>0C9H</td>
</tr>
<tr>
<td>TH0</td>
<td>Timer/counter 0 high byte</td>
<td>8CH</td>
</tr>
<tr>
<td>TL0</td>
<td>Timer/counter 0 low byte</td>
<td>8AH</td>
</tr>
<tr>
<td>TH1</td>
<td>Timer/counter 1 high byte</td>
<td>8DH</td>
</tr>
<tr>
<td>TL1</td>
<td>Timer/counter 1 low byte</td>
<td>8BH</td>
</tr>
<tr>
<td>TH2</td>
<td>Timer/counter 2 high byte</td>
<td>0CDH</td>
</tr>
<tr>
<td>TL2</td>
<td>Timer/counter 2 low byte</td>
<td>0CCH</td>
</tr>
<tr>
<td>RCAP2H</td>
<td>T/C 2 capture register high byte</td>
<td>0CBH</td>
</tr>
<tr>
<td>RCAP2L</td>
<td>T/C 2 capture register low byte</td>
<td>0CAH</td>
</tr>
<tr>
<td>SCON*</td>
<td>Serial control</td>
<td>98H</td>
</tr>
<tr>
<td>SBUF</td>
<td>Serial data buffer</td>
<td>99H</td>
</tr>
<tr>
<td>PCON</td>
<td>Power control</td>
<td>87H</td>
</tr>
</tbody>
</table>

* Bit-addressable
### Special Function Register Map

**Bit addressable**

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>F8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E8</td>
<td></td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E0</td>
<td></td>
<td></td>
<td>ACC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D8</td>
<td></td>
<td></td>
<td></td>
<td>PSW</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PSW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>IP</td>
<td></td>
</tr>
<tr>
<td>B0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P3</td>
<td></td>
</tr>
<tr>
<td>A8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>IE</td>
<td></td>
</tr>
<tr>
<td>A0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P2</td>
<td></td>
</tr>
<tr>
<td>98</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SBUF</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P1</td>
<td></td>
</tr>
<tr>
<td>88</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TMOD</td>
<td>TL0</td>
</tr>
<tr>
<td>80</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SP</td>
<td>DPH</td>
</tr>
</tbody>
</table>
## Value of SFR at Reset

<table>
<thead>
<tr>
<th>REGISTER</th>
<th>RESET VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>000H</td>
</tr>
<tr>
<td>ACC</td>
<td>00H</td>
</tr>
<tr>
<td>B</td>
<td>00H</td>
</tr>
<tr>
<td>PSW</td>
<td>00H</td>
</tr>
<tr>
<td>SP</td>
<td>07H</td>
</tr>
<tr>
<td>DPTR</td>
<td>00000H</td>
</tr>
<tr>
<td>P0–P3</td>
<td>FFH</td>
</tr>
<tr>
<td>IP</td>
<td>XXX000000B</td>
</tr>
<tr>
<td>IE</td>
<td>0XX00000B</td>
</tr>
<tr>
<td>TMOD</td>
<td>00H</td>
</tr>
<tr>
<td>TCON</td>
<td>00H</td>
</tr>
<tr>
<td>TH0</td>
<td>00H</td>
</tr>
<tr>
<td>TL0</td>
<td>00H</td>
</tr>
<tr>
<td>TH1</td>
<td>00H</td>
</tr>
<tr>
<td>TL1</td>
<td>00H</td>
</tr>
<tr>
<td>SCON</td>
<td>00H</td>
</tr>
<tr>
<td>SBUF</td>
<td>Indeterminate</td>
</tr>
<tr>
<td>PCON (NMOS)</td>
<td>0XXXXXXX00B</td>
</tr>
<tr>
<td>PCON (CMOS)</td>
<td>0XXX00000B</td>
</tr>
</tbody>
</table>
A functioning computer must have memory for program code bytes, commonly in ROM, and RAM memory for variable data that can be altered as the program runs.

8051 has internal RAM (128 bytes) and ROM (4Kbytes).

8051 uses the same address but in different memories for code and data.

Internal circuitry access the correct memory based on the nature of the operation in progress.

Can add memory externally if needed.
RAM memory space allocation in the 8051
8051 Internal RAM Organisation

- 128 bytes internal RAM
  - 4 register banks of 8 bytes each (R0-R7)
  - 16 bytes of bit-addressable area
  - 80 bytes of general purpose memory
Internal ROM

- Internal ROM occupies the code address space from 0000H to 0FFFH (Size = 4K byte)
- Program addresses higher than 0FFFH will automatically fetch code bytes from external program memory
- Code bytes can also be fetched exclusively from an external memory by connecting the external access pin (EA) to ground
Memory mapping in 8051

- ROM memory map in 8051 family

4k
- 0000H to 0FFFH

8k
- 0000H to 1FFFH

32k
- 0000H to 7FFFH

- 8751 AT89C51 from Atmel Corporation
- 8752 AT89C52 from Atmel Corporation
- DS5000-32 from Dallas Semiconductor
D Latch:
A Pin of Port 1

- Read latch
- Internal CPU bus
- Write to latch
- Read pin
- TB2
- Vcc
- Load(L1)
- M1
- P1.X pin
- 8051 IC

D Q
P1.X
Clk Q

P0.x
Writing “1” to Output Pin P1.X

1. write a 1 to the pin

Internal CPU bus

Read latch

Write to latch

P1.X

D

Q

Clk

Q

1

0

TB2

Vcc

Load(L1)

M1

TB1

Vcc

8051 IC

2. output pin is

P1.X pin

output 1

1. write a 1 to the pin

Internal CPU bus

Read latch

Write to latch

P1.X

D

Q

Clk

Q

1

0

TB2

Vcc

Load(L1)

M1

TB1

Vcc

8051 IC

2. output pin is

P1.X pin

output 1
Writing “0” to Output Pin P1.X

1. write a 0 to the pin

2. output pin is ground

8051 IC
Reading “High” at Input Pin

1. write a 1 to the pin MOV P1,#0FFH
2. MOV A,P1 external pin=High
3. Read pin=1 Read latch=0
   Write to latch=1

8051 IC
Reading “Low” at Input Pin

1. write a 1 to the pin
   MOV P1,#0FFH

2. MOV A,P1
   external pin=Low

3. Read pin=1 Read latch=0
   Write to latch=1

8051 IC
A Pin of Port 0

Read latch

Internal CPU bus

Write to latch

Read pin

8051 IC

D Q

P1.X

Clk Q

TB2

M1

P0.X pin

P1.x

TB1
I/O Ports (P0 - P3)

One of the most useful features of the 8051 is that it consists of 4 I/O ports (P0 - P3)

- All ports are bidirectional (they can take input and to provide output)
- All ports have multiple functions (except P1)
- All ports are bit addressable
- On **RESET** all the ports are configured as **output**
- When a bit latch is to be used as an **input**, a “1” **must be** written to the corresponding latch by the program to **configure it as input** (eg. MOV P1, #0FFH)
Port 0

- Occupies a total of 8 pins (Pins 32-39)
- Can be used for:
  - Input only
  - Output only
  - Input and output at the same time (i.e. some pins for input and the others for output)
- Can be used to handle both address and data
- Need pull-up resistors
Dual Role of Port 0

- When connecting an 8051 to an external memory, port 0 provides both address and data (AD0 – AD7)
- When ALE = 0, it provides data D0 – D7
- When ALE = 1, it provides data A0 – A7
- ALE is used for demultiplexing address and data with the help of a 74LS373 latch
Port 0 as an Input Port

In the following code, port 0 is configured first as an input port by writing 1s to it, and then data is received from that port and sent to P1.

```
BACK:
  MOV A, #0FFH
  MOV P0, A
  MOV A, P0
  MOV P1, A
  SJMP BACK
```

FFH = 111111112
The following code will continuously send out to port 0 the alternating values 55H and AAH

```
BACK:  MOV A, #55H
       MOV P0, A
       ACALL DELAY
       CPL A
       SJMP BACK
```

\[ AAH = 10101010_2 \]
\[ 55H = 01010101_2 \]
Port 1

- Occupies a total of 8 pins (Pins 1-8)
- Can be used as input or output
- **Does not need** any pull-up resistors
- Upon reset, port 1 is configured as an output port
- No alternative functions
Port 1 as an Input Port

In the following code, port 1 is configured first as an input port by writing 1s to it, and then data is received from that port and saved in R7, R6, and R5.

```
MOV A, #0FFH
MOV P1, A
MOV A, P1
MOV R7, A
ACALL DELAY
MOV A, P1
MOV R6, A
ACALL DELAY
MOV A, P1
MOV R5, A
```
Port 1 as an Output Port

The following code will continuously send out to port 1 the alternating values 55H and AAH

<table>
<thead>
<tr>
<th>BACK:</th>
<th>MOV</th>
<th>A, #55H</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MOV</td>
<td>P1, A</td>
</tr>
<tr>
<td></td>
<td>ACALL</td>
<td>DELAY</td>
</tr>
<tr>
<td></td>
<td>CPL</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>SJMP</td>
<td>BACK</td>
</tr>
</tbody>
</table>
Port 2

- Occupies a total of 8 pins (Pins 21-28)
- Similar function as Port 1
- Can be used as input or output
- **Does not need** any pull-up resistors
- Upon reset, **port 2** is configured as an output port
Dual Role of Port 2

- When connecting an 8051 to an external memory, **port 2** provides both address (A8 – A15)
- It is **used along with P0** to provide the 16-bit address
- When P2 is used for the upper 8 bits of the 16-bit address, it cannot be used for I/O
Port 2 as an Input Port

In the following code, port 2 is configured first as an input port by writing 1s to it, and then data is received from that port and sent to P1.

```
MOV A, #0FFH
MOV P2, A
A, P2
P1, A
SJMP BACK
```
### Port 2 as an Output Port

The following code will continuously send out to port 2 the alternating values 55H and AAH

<table>
<thead>
<tr>
<th>BACK:</th>
<th>MOV A, #55H</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>P2, A</td>
</tr>
<tr>
<td>ACALL</td>
<td>DELAY</td>
</tr>
<tr>
<td>CPL</td>
<td>A</td>
</tr>
<tr>
<td>SJMP</td>
<td>BACK</td>
</tr>
</tbody>
</table>
**Port 3**

- Occupies a total of 8 pins (Pins 10-17)
- Similar function as Port 1 and Port 2
- Can be used as input or output
- Does not need any pull-up resistors
- Upon reset, **port 3** is configured as an output port
- Pins can be individually programmable for other uses
- Most commonly be used to provide some important signals (e.g. interrupts)
## Port 3 Alternate Functions

<table>
<thead>
<tr>
<th>P3 Bit</th>
<th>Function</th>
<th>Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>P3.0</td>
<td>RxD</td>
<td>10</td>
</tr>
<tr>
<td>P3.1</td>
<td>TxD</td>
<td>11</td>
</tr>
<tr>
<td>P3.2</td>
<td>INT0</td>
<td>12</td>
</tr>
<tr>
<td>P3.3</td>
<td>INT1</td>
<td>13</td>
</tr>
<tr>
<td>P3.4</td>
<td>T0</td>
<td>14</td>
</tr>
<tr>
<td>P3.5</td>
<td>T1</td>
<td>15</td>
</tr>
<tr>
<td>P3.6</td>
<td>WR</td>
<td>16</td>
</tr>
<tr>
<td>P3.7</td>
<td>RD</td>
<td>17</td>
</tr>
</tbody>
</table>
Other Pins

- P1, P2, and P3 have internal pull-up resistors.
  - P1, P2, and P3 are not open drain.
- P0 has no internal pull-up resistors and does not connect to Vcc inside the 8051.
  - P0 is open drain.
  - Compare the figures of P1.X and P0.X.
    However, for a programmer, it is the same to program P0, P1, P2 and P3.
- All the ports upon RESET are configured as output.
Port 0 with Pull-Up Resistors

Vcc

10 K

DS5000
8751
8951

P0.0
P0.1
P0.2
P0.3
P0.4
P0.5
P0.6
P0.7

Port 0
Timer:

[Diagram showing a sequential logic circuit with Q, Clock, and Flag signals and a timing chart indicating the behavior of Q0, Q1, Q2, and Flag over time.]
## Timers Programming (5/10)

### M1 & M0 – Mode bits

<table>
<thead>
<tr>
<th>M1</th>
<th>M0</th>
<th>Mode</th>
<th>Operating Mode</th>
</tr>
</thead>
</table>
| 0  | 0  | 0    | 13-bit timer/counter mode  
Timer value range from 0000H to 1FFFH in TH - TL |
| 0  | 1  | 1    | 16-bit timer/counter mode  
Timer value range from 0000H to FFFFH in TH - TL |
| 1  | 0  | 2    | 8-bit auto reload timer/counter mode  
THx holds a value that is to be reloaded into TLx each time it overflows. |
| 1  | 1  | 3    | Split timer mode |

Modes 1 & 2 are used most widely.
**TMOD Register:**

- **Gate**: When set, timer only runs while INT(0,1) is high.
- **C/T**: Counter/Timer select bit.
- **M1**: Mode bit 1.
- **M0**: Mode bit 0.
**TCON Register:**

<table>
<thead>
<tr>
<th></th>
<th>TF1</th>
<th>TR1</th>
<th>TF0</th>
<th>TR0</th>
<th>IE1</th>
<th>IT1</th>
<th>IE0</th>
<th>IT0</th>
</tr>
</thead>
</table>

- **TF1**: Timer 1 overflow flag.
- **TR1**: Timer 1 run control bit.
- **TF0**: Timer 0 overflow.
- **TR0**: Timer 0 run control bit.
- **IE1**: External interrupt 1 edge flag.
- **IT1**: External interrupt 1 type flag.
- **IE0**: External interrupt 0 edge flag.
- **IT0**: External interrupt 0 type flag.
Interrupt:

Program execution without interrupts:

Program execution with interrupts:

ISR: Interrupt Service Routine
# Interrupt Vector Table for the 8051

- **Interrupt vector table** holds the addresses of ISR

<table>
<thead>
<tr>
<th>Priority</th>
<th>Interrupt</th>
<th>Flag</th>
<th>ROM location</th>
<th>Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Reset</td>
<td>RST</td>
<td>0000H</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>External 0 (INT0)</td>
<td>IE0</td>
<td>0003H</td>
<td>P3.2 (12)</td>
</tr>
<tr>
<td>3</td>
<td>Timer 0</td>
<td>TF0</td>
<td>000BH</td>
<td>---</td>
</tr>
<tr>
<td>4</td>
<td>External 1 (INT1)</td>
<td>IE1</td>
<td>0013H</td>
<td>P3.3 (13)</td>
</tr>
<tr>
<td>5</td>
<td>Timer 1</td>
<td>TF1</td>
<td>001BH</td>
<td>---</td>
</tr>
<tr>
<td>6</td>
<td>Serial port</td>
<td>RI or TI</td>
<td>0023H</td>
<td>---</td>
</tr>
</tbody>
</table>
## Interrupt Enable Register:

<table>
<thead>
<tr>
<th>EA</th>
<th>ET2</th>
<th>ES</th>
<th>ET1</th>
<th>EX1</th>
<th>ET0</th>
<th>EX0</th>
</tr>
</thead>
</table>

- **EA**: Global enable/disable.
- **---**: Undefined.
- **ET2**: Enable Timer 2 interrupt.
- **ES**: Enable Serial port interrupt.
- **ET1**: Enable Timer 1 interrupt.
- **EX1**: Enable External 1 interrupt.
- **ET0**: Enable Timer 0 interrupt.
- **EX0**: Enable External 0 interrupt.
Interrupt Priority

Upon reset, the priorities of interrupt source are assigned from top to bottom as in the following Table 8, i.e. if INT0 and INT1 are activated at the same time, INT0 is first responded.

Table 8.1: 8051 Interrupt Priority Upon Reset

<table>
<thead>
<tr>
<th>Interrupt Source</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>External Interrupt 0</td>
<td>INT0</td>
</tr>
<tr>
<td>Timer Interrupt 0</td>
<td>TF0</td>
</tr>
<tr>
<td>External Interrupt 1</td>
<td>INT1</td>
</tr>
<tr>
<td>Timer Interrupt 1</td>
<td>TF1</td>
</tr>
<tr>
<td>Serial Communication</td>
<td>(RI+TI)</td>
</tr>
</tbody>
</table>

(INT0 > TF0 > INT1 > TF1 > SERIAL(RI+TI))
Setting Interrupt Priority with the IP register

- The sequence of last Table can be changed by assigning a higher priority to any one of the interrupts.
- It is done by setting high at the corresponding bit in the **IP** (interrupt priority) / Bit-addressable register.

<table>
<thead>
<tr>
<th>IP.7</th>
<th>IP.6</th>
<th>IP.5</th>
<th>IP.4</th>
<th>IP.3</th>
<th>IP.2</th>
<th>IP.1</th>
<th>IP.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>--</td>
<td>PT2</td>
<td>PS</td>
<td>PT1</td>
<td>PX1</td>
<td>PT0</td>
<td>PX0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PT2</th>
<th>Timer 2 (8052 only),</th>
</tr>
</thead>
<tbody>
<tr>
<td>PT1 &amp; PT0</td>
<td>Timer 1 &amp; Timer 0 interrupts</td>
</tr>
<tr>
<td>PX1 &amp; PX0</td>
<td>External interrupts 1 &amp; 0,</td>
</tr>
<tr>
<td>PS</td>
<td>Serial port interrupt</td>
</tr>
</tbody>
</table>
Addressing Modes

- Addressing mode: a method that...
  - Points out where the operands (i.e. source and destination) are, and
  - How these operands should be accessed
- The opcode in an instruction specifies what addressing mode will be used
Addressing Modes

- Immediate addressing (eg. MOV A, #55H)
- Register addressing (eg. MOV A, R0)
- Direct addressing (eg. MOV A, 30H)
- Register indirect addressing (eg. MOV A, @R0)
- Indexed addressing (eg. MOVC A, @A+PC)
- Absolute addressing (eg. ACALL address11)
- Long addressing (eg. LCALL addr16)
- Relative addressing (Eg. SJMP relative)
Immediate addressing mode

- The operand comes immediately after the op-code.
- The immediate data must be preceded by the pound sign, "#".

```
MOV A,#25H ;load 25H into A
MOV R4,#62 ;load the decimal value 62 into R4
MOV B,#40H ;load 40H into B
MOV DPTR,#4521H ;DPTR=4512H
```
Register addressing mode

- Register addressing mode involves the use of registers to hold the data to be manipulated.

```
MOV A,R0  ;copy the contents of R0 into A
MOV R2,A  ;copy the contents of A into R2
ADD A,R5  ;add the contents of R5 to contents of A
ADD A,R7  ;add the contents of R7 to contents of A
MOV R6,A  ;save accumulator in R6
```
ACCESSING MEMORY USING VARIOUS ADDRESSING MODES

- Direct addressing mode
- There are 128 bytes of RAM in the 8051.
- The RAM has been assigned addresses 00 to 7FH.
  - 1. RAM locations 00 - 1 FH are assigned to the register banks and stack.
  - 2. RAM locations 20 - 2FH are set aside as bit-addressable space to save singlebit data.
  - 3. RAM locations 30 - 7FH are available as a place to save byte-sized data.
Direct addressing mode

- It is most often used to access RAM locations 30 - 7FH.
- This is due to the fact that register bank locations are accessed by the register names of R0 - R7.
- There is no such name for other RAM locations so must use direct addressing.
Direct addressing mode

- In the direct addressing mode, the data is in a RAM memory location whose address is known, and this address is given as a part of the instruction.

```
MOV R0,40H ;save content of RAM location 40H in R0
MOV 56H,A ;save content of A in RAM location 56H
MOV R4,7FH ;move contents of RAM location 7FH to R4
MOV A,4   ;is same as
MOV A,R4  ;which means copy R4 into A
MOV A,7   ;is same as
MOV A,R7  ;which means copy R7 into A
```
Special Function Registers

- In the 8051, registers A, B, PSW, and DPTR are part of the group of registers commonly referred to as SFR.
- The SFR can be accessed by their names or by their addresses.
- For example, register A has address E0H and register B has been designated the address F0H.

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

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

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

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

MOV P1, A ;is the same as
MOV 90H,A ;which means copy reg A to P1
```
Another major use of direct addressing mode is the stack.

In the 8051 family, only direct addressing mode is allowed for pushing onto the stack.

An instruction such as "PUSH A" is invalid. Pushing the accumulator onto the stack must be coded as "PUSH 0E0H."

Direct addressing mode must be used for the POP instruction as well.

"POP 04" will pop the top of the stack into R4 of bank 0.
Stack and Direct Addressing Mode

Only direct addressing mode is allowed for pushing onto the stack.

- PUSH A      (Invalid)
- PUSH 0E0h   (Valid)
- PUSH R3     (Invalid)
- PUSH 03     (Valid)
- POP R4      (Invalid)
- POP 04      (Valid)

PUSH direct
POP direct
Register indirect addressing mode

- A register is used as a pointer to the data.
- If the data is inside the CPU, only registers R0 and R1 are used for this purpose.
- R2 - R7 cannot be used to hold the address of an operand located in RAM when using indirect addressing mode.
- When RO and R1 are used as pointers they must be preceded by the @ sign.

```
MOV A,@R0 ;move contents of RAM location whose address is held by R0 into A
MOV @R1,B ;move contents of B into RAM location whose address is held by R1
```
Register Indirect Addressing (eg. ADD A, @R0)

Before

ACC

10

R0

31

After

ACC

22

R0

31

Program memory

Data memory

Addresses

200
ADD A, @R0

201

30

31

32

12

ADD A, @R0
Indexed Addressing

- Using a **base register** (starting point) and an **offset** (how much to parse through) to form the effective address for a JMP or MOV instruction.
- Used to parse through an array of items or a look-up table.
- Usually, the **DPTR** is the base register and the “**A**” is the offset.
- **A** increases/decreases to parse through the list.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address Formulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVC A, @A+DPTR</td>
<td>@A+DPTR</td>
</tr>
<tr>
<td>MOVC A, @A+PC</td>
<td>@A+PC</td>
</tr>
<tr>
<td>JMP</td>
<td>@A+DPTR</td>
</tr>
</tbody>
</table>
# Examples of Indexed Addressing

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVCA, @A + DPTR</td>
<td>Copy the code byte, found at the ROM address formed by adding register A and the DPTR register, to A</td>
</tr>
<tr>
<td>MOVCA, @A + PC</td>
<td>Copy the code byte, found at the ROM address formed by adding A and the PC, to A</td>
</tr>
<tr>
<td>JMP @A + DPTR</td>
<td>Jump to the address formed by adding A to the DPTR, this is an unconditional jump and will always be done.</td>
</tr>
</tbody>
</table>
Indexed Addressing

Example: MOVC A, @A + DPTR

Before

Program memory

00

DPTR

56

31

10

ACC

After

ACC

56
Example 6-10  (look-up table)

Write a program to get the \( x \) value from P1 and send \( x^2 \) to port P2, continuously.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORG 0000h</td>
<td></td>
</tr>
<tr>
<td>MOV DPTR, #300h</td>
<td>load look-up table address</td>
</tr>
<tr>
<td>MOV A, #0FFh</td>
<td>A = FF</td>
</tr>
<tr>
<td>MOV P1, A</td>
<td>configure P1 as input port</td>
</tr>
<tr>
<td>MOV A, P1</td>
<td>get X</td>
</tr>
<tr>
<td>MOV A, @A+DPTR;</td>
<td>get X square from table</td>
</tr>
<tr>
<td>MOV P2, A</td>
<td>issue it to port P2</td>
</tr>
<tr>
<td>SJMP BACK</td>
<td>keep doing it</td>
</tr>
</tbody>
</table>

ORG 0000h

TABLE: DB 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

END
Absolute Addressing

- Only used with the instructions **ACALL** and **AJMP**
- Similar to indexed addressing mode
- The largest “jump” that can be made is 2K

Long Addressing

- Only used with the instructions **LCALL** and **LJMP**
- Similar to indexed addressing mode
- The largest “jump” that can be made is 64K

\[ 2^{11} = 2048 = 2K \]
Absolute vs Long Addressing

- Absolute addressing: 11-bit address in 2-byte instruction
- Long addressing: 16-bit address in 3-byte instruction
- Range of the “jump” of long is greater than absolute
- Yet absolute mode has shorter code (2 bytes), hence faster execution
Relative Addressing

- Used in jump ("JMP") instructions
- Relative address: an 8-bit value (-128 to +127)
- You may treat relative address as an offset
- Labels indicate the JMP destinations (i.e. where to stop)
- Assembler finds out the relative address using the label

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SJMP relative</td>
<td>Jump short to relative address</td>
</tr>
<tr>
<td>DJNZ direct, relative</td>
<td>Decrement direct register and jump if not zero</td>
</tr>
<tr>
<td>DJNZ Rn, relative</td>
<td>Decrement Rn register and jump if not zero</td>
</tr>
</tbody>
</table>
Relative Addressing

- The relative address is added to the PC.
- The sum is the address of the next instruction to be executed.
- As a result, program skips to the desired line right away instead of going through each line one by one.
- Labels indicate the JMP destinations (i.e. where to stop).
Relative Addressing

Program counter + offset
= Effective address
= address of next instruction

Program Counter

Branch Opcode
Offset
Next Opcode

+ Offset

Next Instruction
### Examples of Relative Addressing

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SJMP NXT</td>
<td>Jump to relative address with the label 'NXT'; this is an unconditional jump and is always taken.</td>
</tr>
<tr>
<td>DJNZ R1, DWN</td>
<td>Decrement register R1 by 1 and jump to the relative address specified by the label 'DWN' if the result of R1 is not zero.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code Address</th>
<th>Op Codes</th>
<th>Mnemonics</th>
<th>Operands</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0030</td>
<td>74 89</td>
<td>MOV</td>
<td>A, #89h</td>
<td>; load A with data 89h</td>
</tr>
<tr>
<td>0032</td>
<td>00</td>
<td>NOP</td>
<td></td>
<td>; endless loop</td>
</tr>
<tr>
<td>0033</td>
<td>80 6F</td>
<td>SJMP</td>
<td>0032h</td>
<td></td>
</tr>
</tbody>
</table>
Assembler Directives

**ORG (origin)**

indicates the beginning of the address of the instructions. The number that comes after ORG can be either hex or decimal.

**EX:** ORG 0030

**END**

indicates to the assembler the end of the source assembly instructions.

**EQU (equate)**

used to define a constant without occupying a memory location. It does not set aside storage for a data item but associates a constant value with a data label so that when the label appears in the program, its constant value will be substituted for the label.

**EX:** COUNT EQU 25H

**DB (define byte)**

used to define 8-bit data and store them in assigned memory locations. Define data can be in decimal, binary, hex, or ASCII formats.

**EX:** ASCIICODE : DB “APPLE”
MYDATA DB 23H
**8051 Instruction Set**

- **ACALL**: Absolute Call
- **AjMP**: Absolute Jump
- **CJNE**: Compare & Jump if Not Equal
- **CPL**: Complement Register
- **DEC**: Decrement Register
- **DJNZ**: Dec. Reg. & Jump if Not Zero
- **JB**: Jump if Bit Set
- **JC**: Jump if Carry Set
- **JNB**: Jump if Bit Not Set
- **JNZ**: Jump if Acc. Not Zero
- **LCALL**: Long Call
- **MOV**: Move Memory
- **MOVX**: Move Extended Memory
- **NOP**: No Operation
- **POP**: Pop Value From Stack
- **RET**: Return From Subroutine
- **RL**: Rotate Accumulator Left
- **RR**: Rotate Accumulator Right
- **SETB**: Set Bit
- **SUBB**: Sub. From Acc. With Borrow
- **XCH**: Exchange Bytes
- **XRL**: Bitwise Exclusive OR
- **ADD, ADDC**: Add Acc. (With Carry)
- **ANL**: Bitwise AND
- **CLR**: Clear Register
- **DA**: Decimal Adjust
- **DIV**: Divide Accumulator by B
- **INC**: Increment Register
- **JBC**: Jump if Bit Set and Clear Bit
- **JMP**: Jump to Address
- **JNC**: Jump if Carry Not Set
- **JNZ**: Jump if Acc. Not Zero
- **JZ**: Jump if Accumulator Zero
- **LCALL**: Long Call
- **LJMP**: Long Jump
- **MOVC**: Move Code Memory
- **MUL**: Multiply Accumulator by B
- **ORL**: Bitwise OR
- **PUSH**: Push Value Onto Stack
- **RET**: Return From Subroutine
- **RET**: Return From Interrupt
- **RLC**: Rotate Acc. Left Through Carry
- **RRC**: Rotate Acc. Right Through Carry
- **SJMP**: Short Jump
- **SWAP**: Swap Accumulator Nibbles
- **XCHD**: Exchange Digits
- **Undefined**: Undefined Instruction
An Assembly language instruction consists of four fields:

\[
\text{[label : ] mnemonic [operands] [';comment']}
\]
Types Of Instructions

- Arithmetic instructions
- Logical instructions
- Rotate instructions
- Data movement instructions
- Program branching instructions
Data movement instructions

Moving data from a source to a destination

- MOV
- MOVX
- MOVC
- PUSH
- POP
- XCH
- XCHD
The MOV Instruction

MOV destination, source

- Used for data movement inside the 8051
- No change in original (source) data: a copy of it is made and the copy is then moved to the destination
- Source can be data/register/memory locations
- Destination can be register/memory locations
- Note that the destination cannot be an immediate data

The MOV Instruction
Examples of MOV Instruction

- **MOV A, 80h**
  - Copy data from 80h (port 0) to register A

- **MOV 80h, A**
  - Copy data from register A to RAM address 80h (port 0)

- **MOV 3Ah, #3Ah**
  - Copy immediate data 3Ah to RAM location 3Ah

- **MOV R0, 12h**
  - Copy data from RAM location 12h to R0

- **MOV 5Ch, A**
  - Copy data from register A to RAM location 5Ch

- **MOV 08Ah, 77h**
  - Copy data from RAM address 77h to 08Ah (IE register)
The MOVX Instruction

- MOVX destination, source
- "X" means the data movement is external to the 8051 → data movement is between the external RAM and the (internal) register A
- All MOVX instructions must involve register A
- All MOVX instructions must use indirect addressing mode
- Operation is similar to the MOV instruction
**Examples of MOVX Instruction**

- **MOVX  @DPTR, A**
  - Copy data from A to the 16-bit address in DPTR

- **MOVX  @R0, A**
  - Copy data from A to the 8-bit address in R0

- **MOVX  A, @R1**
  - Copy data from the 8-bit address in R1 to A

- **MOVX  A, @DPTR**
  - Copy data from the 16-bit address in DPTR to A
The MOVC Instruction

- MOVC destination, source
- “C” means the data movement is from the source address in code ROM to register A
- Used with data transfer between internal/external ROM and register A, e.g. reading a table from the program memory
Examples of MOVC Instruction

- **MOVC A, @A+DPTR** (Copy the code byte to A)
  - This code byte is found at the ROM address formed by adding A and the DPTR

- **MOVC A, @A+PC** (Copy the code byte to A)
  - This code byte is found at the ROM address formed by adding A and the PC

Note that the **PC** is incremented by 1 before added to A to form the final address of the code byte.
The PUSH Instruction

- **PUSH source**
  Copy data from the source address to the **stack**
- **SP** is incremented by 1 **before** source data is copied to the **stack**
- Too many **PUSH** operations may overflow the **stack**, (i.e. **stack** runs out of memory)
The POP Instruction

- **POP  destination**
  Copy data from the **stack** to the destination address

- **SP** is decremented by 1 (−1) after data is copied from the stack
Examples

**MOV 81h, #30h** ; Copy the immediate data 30h to SP

**MOV R0, #0ACh**; Copy the immediate data ACh to R0 ; (i.e. 00h)

**PUSH 00h** ; **SP**=31h, address 31h contains the ; number ACh

**PUSH 00h** ; **SP**=32h, address 32h contains the ; number ACh

**POP 01h** ; **SP**=31h, address R1 (i.e. 01h) ; contains the number ACh

**POP 80h** ; **SP**=30h, port 0 latch (i.e. 80h) ; contains the number ACh
Data Exchanging

- XCH and XCHD are “data exchange” instructions
  → data movement is bi-directional (i.e. source ↔ destination)
- Data exchanging operations are internal to the 8051
- All data exchanging operations must use register A
The XCH and XCHD Instructions

- **XCH** destination, source
- **XCHD** destination, source

- **XCH**: Data exchange between register A and the addressed byte
- **XCHD**: Data exchange between the lower-nibble of A and the addressed byte
  - (Upper-nibble of A remains unchanged)
Examples

- **XCH**    A, R7
  Exchange bytes between registers A and R7

- **XCH**    A, 0F0h
  Exchange bytes between registers A and B

- **XCH**    A, @R1
  Exchange bytes between register A and address in R1

- **XCHD**   A, @R1
  Exchange lower-nibble in register A and the address in R1
Arithmetic Instructions

There are 24 arithmetic opcodes which are grouped into the following types:

- ADD and ADDC
- SUBB
- MUL
- DIV
- INC
- DEC
- DA
## Instructions that Affecting Flags (1/2)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>Flags Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>C, AC, OV</td>
</tr>
<tr>
<td>ADDC</td>
<td>C, AC, OV</td>
</tr>
<tr>
<td>SUBB</td>
<td>C, AC, OV</td>
</tr>
<tr>
<td>MUL</td>
<td>C = 0, OV</td>
</tr>
<tr>
<td>DIV</td>
<td>C = 0, OV</td>
</tr>
<tr>
<td>DA A</td>
<td>C</td>
</tr>
<tr>
<td>SETB C</td>
<td>C = 1</td>
</tr>
<tr>
<td>MOV C, bit</td>
<td>C</td>
</tr>
</tbody>
</table>
The **ADD** and **ADDC** Instructions

- **ADD**  \( A, \text{source} \quad ; A = A + \text{source} \)
- **ADDC**  \( A, \text{source} \quad ; A = A + \text{source} + C \)

- A register must be involved in additions
- The **C** flag is set to 1 if there is a carry out of bit 7
- The **AC** flag is set to 1 if there is a carry out of bit 3
- **ADD** is used for ordinary addition
- **ADDC** is used to add a carry after the LSB addition in a multi-byte process
**Example 6-1**

Show how the flag register is affected by the following instructions.

MOV A, #0F5h ; A = F5h
ADD A, #0Bh ; A = F5 + 0B = 00

<table>
<thead>
<tr>
<th>Solution</th>
<th>F5h</th>
<th>1111 0101</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ 0Bh</td>
<td>+ 0000 1011</td>
<td></td>
</tr>
<tr>
<td>100h</td>
<td>0000 0000</td>
<td></td>
</tr>
</tbody>
</table>

After the addition, register A (destination) contains 00 and the flags are as follows:

- CY = 1 since there is a carry out from D7
- P = 0 because the number of 1s is zero
- AC = 1 since there is a carry from D3 to D4
The DA Instruction

DA A

- The action is to "decimal adjust" the register A
- Used after the addition of two BCD numbers

Example 6.4:

MOV A, #47h ; A=47h first BCD operand
MOV B, #25h ; B=25h second BCD operand
ADD A, B ; hex (binary) addition (A=6Ch)
DA A ; adjust for BCD addition (A=72h)
## Example 6.4 of DA Instruction

<table>
<thead>
<tr>
<th>Hex</th>
<th>BCD</th>
</tr>
</thead>
<tbody>
<tr>
<td>47</td>
<td>0100 0111</td>
</tr>
<tr>
<td>+ 25</td>
<td>+ 0010 0101</td>
</tr>
<tr>
<td>6C</td>
<td>0110 1100</td>
</tr>
<tr>
<td>+ 6</td>
<td>+ 0110</td>
</tr>
<tr>
<td>72</td>
<td>0111 0010</td>
</tr>
</tbody>
</table>

Offset decimal 6!
The SUBB Instruction

SUBB A, source

No borrow: \( A = A - \text{source} \)

With borrow:
\[ A = A - \text{source} - \text{carry} \quad \text{(i.e. borrow)} \]

Note that the 8051 uses the 2’s complement method to do subtraction

After execution:
The C flag is set to 1 if a borrow is needed into bit 7
The AC flag is set to 1 if a borrow is needed into bit 3

<table>
<thead>
<tr>
<th>SUBB</th>
<th>A, #data</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBB</td>
<td>A, direct</td>
</tr>
<tr>
<td>SUBB</td>
<td>A, @Ri, where i = 0 or 1</td>
</tr>
<tr>
<td>SUBB</td>
<td>A, Rn, where n = 0, 1, 7</td>
</tr>
</tbody>
</table>
The MUL Instruction

- **MUL**   **AB**

- Uses registers **A** and **B** as both source and destination registers

- Numbers in **A** and **B** are multiplied, then put the lower-order byte of the product in **A** and the high-order byte in **B**

- The **OV** flag is set to 1 if the product > FFh

- Note that the **C** flag is 0 at all times
The DIV Instruction

DIV AB

Similarly, it uses registers A and B as both source and destination registers.

The number in A is divided by B. The quotient is put in A and the remainder (if any) is put in B.

The OV flag is set to 1 if B has the number 00h (divide-by-zero error).

Note that the C flag is 0 at all times.
The INC and DEC Instructions

- To increment (INC) or decrement (DEC) the internal memory location specified by the operand.
- No change with all the arithmetic flags in this operation.
- e.g. INC 7Fh ;content in 7Fh increased by 1
  DEC R1 ;content in R1 decreased by 1

<table>
<thead>
<tr>
<th>INC  A</th>
</tr>
</thead>
<tbody>
<tr>
<td>INC  direct</td>
</tr>
<tr>
<td>INC  @Ri where i=0,or 1</td>
</tr>
<tr>
<td>INC  Rn where n=0,,7</td>
</tr>
</tbody>
</table>
Logical Instructions

- **ANL** destination, source
  Destination = destination **AND** source

- **ORL** destination, source
  Destination = destination **OR** source

- **XRL** destination, source
  Destination = destination **XOR** source

- Usually, the destination is register **A** or a direct address in the internal RAM

<table>
<thead>
<tr>
<th>Logical Instruction</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANL direct, A</td>
<td>ANL direct, #data</td>
</tr>
<tr>
<td>ANL A, #datan</td>
<td>ANL A, direct</td>
</tr>
<tr>
<td>ANL A, @Ri</td>
<td>where i=0, or 1</td>
</tr>
<tr>
<td>ANL A, Rn</td>
<td>where n=0, 7</td>
</tr>
</tbody>
</table>
Logical Instructions

- The source operand can be any of the 4 addressing modes (i.e. immediate/register/direct/indirect)

- **ANL** can be used to clear (0) certain bits

- **ORL** can be used to set (1) certain bits

### Examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ANL A,R0</th>
<th>ORL A,R0</th>
<th>XRL A,R0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A before:</td>
<td>10010111</td>
<td>10010111</td>
<td>10010111</td>
</tr>
<tr>
<td>R0 before:</td>
<td>11110010</td>
<td>11110010</td>
<td>11110010</td>
</tr>
<tr>
<td>A afterwards:</td>
<td>10010010</td>
<td>11110111</td>
<td>01100101</td>
</tr>
</tbody>
</table>
The CLR and CPL Instructions

**CLR A**
All bits in register A are cleared

**CPL A**
All bits in register A are complemented (inverted)

Note that CLR and CPL instructions operate on register A only
Contents in register A is rotated one bit position to the left or to the right (operated in A only)

The bit shifted out is used as the new bit shifted in

May include the C flag in the operation

Useful in inspecting the bits in a byte one by one

Also useful for multiplication and division in powers of 2
The Rotate Instructions

- **RL A**
  Rotates A one bit position to the left

- **RLC A**
  Rotates A and the carry flag one bit position to the left

- **RR A**
  Rotates A one bit position to the right

- **RRC A**
  Rotates A and the carry flag one bit position to the right

Note that for **RLC** and **RRC**, you have to know the C flag first
The Rotate Instructions

**RL A**

Before: 10011100
After: 00111001

**RLC A**

Before: 10011100
   CY = 0
After: 00111000
   CY = 1

**RR A**

Before: 10011100
   After: 00111001

Before: 10011100
   CY = 1
After: 11001110
   CY = 0

**RRC A**

Before: 10011100
   CY = 1
After: 11001110
   CY = 0
The SWAP Instruction

- Swapping the lower-nibble (lower 4 bits) and the higher-nibble (upper 4 bits) of register A.

Register A = 5Eh (original value) after SWAP Register A = E5h
Program branching instructions / Program Flow Control instructions

- Branching
- Subroutine
Program Branching is caused when a JUMP instruction is executed. It directs the program flow to the immediate destination address. Two kinds of Jump instructions:

- **CONDITIONAL JUMP**
- **UNCONDITIONAL JUMP**
a jump to the target location in which the control of the jump is valid only if a condition is met,
  e.g. the status of Carry Flag, result of comparison, etc.

All conditional jumps are short jumps and the range of jump to the target address must be within -128 to +127 bytes.
Conditional jump instructions for 8051 are summarized as below:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>JZ</td>
<td>Jump if A equal 0</td>
</tr>
<tr>
<td>JNZ</td>
<td>Jump if A not equal 0</td>
</tr>
<tr>
<td>DJNZ</td>
<td>Decrement and jump, if A not equal 0</td>
</tr>
<tr>
<td>CJNE A, byte</td>
<td>Jump, if A not equal to byte</td>
</tr>
<tr>
<td>CJNE reg, #data</td>
<td>Jump, if byte not equal to #data</td>
</tr>
<tr>
<td>JC</td>
<td>Jump, if CY =1</td>
</tr>
<tr>
<td>JNC</td>
<td>Jump, if CY =0</td>
</tr>
<tr>
<td>JB</td>
<td>Jump, if bit equal to 1</td>
</tr>
<tr>
<td>JNB</td>
<td>Jump, if bit equal to 0</td>
</tr>
<tr>
<td>JBC</td>
<td>Jump, if bit equal to 1 and clear bit</td>
</tr>
</tbody>
</table>
## Conditional jumps

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JZ &lt;rel addr&gt;</td>
<td>Jump if a = 0</td>
</tr>
<tr>
<td>JNZ &lt;rel addr&gt;</td>
<td>Jump if a != 0</td>
</tr>
<tr>
<td>JC &lt;rel addr&gt;</td>
<td>Jump if C = 1</td>
</tr>
<tr>
<td>JNC &lt;rel addr&gt;</td>
<td>Jump if C != 1</td>
</tr>
<tr>
<td>JB &lt;bit&gt;,&lt;rel addr&gt;</td>
<td>Jump if bit = 1</td>
</tr>
<tr>
<td>JNB &lt;bit&gt;,&lt;rel addr&gt;</td>
<td>Jump if bit != 1</td>
</tr>
<tr>
<td>JBC &lt;bit&gt;,&lt;rel addr&gt;</td>
<td>Jump if bit =1, clear bit</td>
</tr>
<tr>
<td>CJNE A, direct,&lt;rel addr&gt;</td>
<td>Compare A and memory, jump if not equal</td>
</tr>
</tbody>
</table>
CONDITIONAL JUMP

Example

NEXT:  ADD A, #7EH
       JNC OVER

OVER:  ADD A, R3
       CJN A, #80H, E NEXT

Jump to label "OVER" if no carry

Jump to label "NEXT" if A ≠ #80H
“Looping” is widely used in the 8051 to repeat a sequence of instructions at a preset number of times.

To perform the looping action, instructions DJNZ and CJNE are commonly used as a counting machine.

MOV R2, #10
AGAIN: ADD A, #03
DJNZ R2, AGAIN

No. of Looping is given by the value of R2
Nested LOOPING

- Nested loop means there are loops inside a loop.
- For a 2-layer nested loop, 2 registers are used as counting machines.

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R3, #3</td>
<td></td>
</tr>
<tr>
<td>NEXT: MOV R2, #10</td>
<td></td>
</tr>
<tr>
<td>AGAIN: ADD A, #2</td>
<td></td>
</tr>
<tr>
<td>DJNZ R2, AGAIN</td>
<td></td>
</tr>
<tr>
<td>DJNZ R3 NEXT</td>
<td></td>
</tr>
</tbody>
</table>

Inner Loop; counted by the value of R2 (=10)

Outer Loop; counted by the value of R3 (=3)
Unconditional Jump instruction causes the program jump to a target location without a condition.

There are 3 kinds of unconditional jumps:

- **LJMP** (long jump)
  Allows the program counter jump to any address location within the 64K code space (16-bit address from 0000h to FFF_fh)

- **SJMP** (short jump)
  Allows the program counter jump either forward or backward within a range of -128 and 127 bytes relative to the original memory address.

- **AJMP** (absolute jump)
  Allows the program jump to the target address within a 2K bytes program memory page unconditionally.
Unconditional Jumps

- **SJMP <rel addr>** ; Short jump, relative address is 8-bit 2’s complement number, so jump can be up to 127 locations forward, or 128 locations back.

- **LJMP <address 16>** ; Long jump

- **AJMP <address 11>** ; Absolute jump to anywhere within 2K block of program memory

- **JMP @A + DPTR** ; Long indexed jump
Subroutines

Main: ...

acall sublabel

...

...

...

sublabel:....

...

ret

call to the subroutine

the subroutine
CALL Instructions

Another control transfer instruction is the CALL instruction, which is used to call a subroutine. Call is similar to a jump, but Call instruction pushes PC on stack before branching.

- **ACALL** (Absolute Call)
  - It is a 2-byte instruction.
  - It calls subroutines with a target address within 2K bytes from the current program counter (PC).
  - It apparently gives a 16-bit memory address to PC where the called subroutine starts.
  - Ex: *Absolute call*
    
    acall <address 1l>; stack ← PC
    ; PC ← address 1l
LCALL (Long Call)

• It is a 3-byte instruction, the first byte is the opcode and the second and third byte is the operand, which denotes the start of memory address of the subroutine call.

• It can be used to call subroutines located anywhere within the 64kbyte memory address.

Ex: Long call
    Lcall <address 16> ; stack ← PC
                         ; PC ← address 16
Return

- Return is also similar to a jump, but
  - Return instruction pops PC from stack to get address to jump to

**EX: ret ; PC ← stack**
Thanks......