

MODULE - 2

ADDRESSING MODES DIRECTIVES INSTRUCTION SET OF 8051 MICROCONTROLLER

NANDITHA KRISHNA

★ ADDRESSING MODES

- The CPU can access data in various ways.
- The data could be in a register or in a memory or be provided as an immediate value
- The various ways of accessing data is called as addressing modes.
- 8051 addressing modes are as follows-

- (1) Immediate addressing
- (2) Register addressing
- (3) Direct addressing
- (4) Indirect addressing
- (5) Relative addressing
- (6) Absolute addressing
- (7) Long addressing
- (8) Indexed addressing
- (9) Bit indirect addressing
- (10) Bit direct addressing
- (11) Inherent addressing

(1) IMMEDIATE ADDRESSING MODE -

- In this addressing mode the source operand is a constant
- As the name implies, when the instruction is assembled the operand comes immediately after opcode
- The immediate data must be preceded by the pound sign #

→ This addressing mode can be used to load information into any of the registers including (3) Rn register in 8bit ports

- (i) (r) MOV A, #34h
- (ii) MOV R4, #62
- (iii) MOV R5, #10h
- (iv) MOV DPTR, #0123h
- (v) MOV R1, #A0h

NOTE: DPTR register is a 16 bit, but it can be accessed as two 8 bit registers DPH and DPL where DPH is the high byte and DPL is the low byte.

(2) REGISTER ADDRESSING MODE

→ This involves the use of registers to hold the data to be manipulated

→ The registers A, DPTR, and R0 to R7 may be used as source and also destination

Ex(i) MOV A,R0 ; Copy the contents of R0 into A

(ii) MOV R2,A ; copy the contents of A into R2

(iii) ADD A,R5 ; add the contents of R5 to contents of A

(iv) ADD A,R7 ; add the contents of R7 to contents of A

(v) MOV R6,A ; 32bit accumulator R6 to A

(vi) MOV DPTR,#0123h

(vii) MOV R7,DPL

(viii) MOV R6,DPH

NOTE - we can move data between the accumulator & Rn register [n=0 to 7] but movement of data between Rn register is not allowed

R2 : MOV R6,R7 is invalid

(3) DIRECT ADDRESSING MODE -

- There are 128 bytes of RAM in 8051
- RAM has been assigned addresses 00 to 7Fh . The allocation of 128 bytes is as below -
- (i) RAM locations 00-1Fh are assigned to register banks & stack
- (ii) RAM locations 20-2Fh are set as bit addressable space to save single bit data.
- (iii) RAM locations 30-7Fh is used to save byte sized data
- The entire 128 bytes of RAM can be accessed using direct addressing mode , but RAM location 30 to 7Fh is most often used.
- Register bank locations are accessed by register names R0-R7 , but there is no such name for other RAM locations .
- In 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 instructions

Ex: (i) MOV R0, 40h ; Save content of RAM location 40h in R
(ii) MOV 56h, A ; Save content of A in RAM location 56h
(iii) MOV R7, 01h ; move content of RAM location 01h to R7
(iv) PUSH 0E0h
(v) POP 03h

. NOTE- '#' sign distinguishes between immediate & direct addressing mode.

→ RAM locations 0 to 7 are allocated to registers R0 - R7. These registers are accessed two ways as shown

- (i) MOV A, 4 is same as
MOV A, R4 ; copy R4 into A
- (ii) MOV A, 7 is same as
MOV A, R7 ; copy R7 into A
- (iii) MOV A, 2 is same as
MOV A, R2 ; copy R2 into A
- (iv) MOV A, 0 is same as
MOV A, R0 ; copy R0 into A
- (v) MOV @R2, R3 is same as
MOV R2, R3 : Invalid instruction

(4) REGISTER INDIRECT ADDRESSING MODE

- In this a register is used to hold the address of the data
 - The register itself is not the address, but rather the number in the register.
 - The instruction for indirect addressing mode uses 'MOV' opcode with register R0 or R1.
 - These registers R0 or R1 will hold RAM addresses from 00h to 7Fh.
 - The symbol used for indirect addressing is "@" at sign
- Ex: (i) MOV A, @R0 ; move contents of RAM location held by R0 into A
- (ii) MOV @R1, B ; move contents of B into RAM location whose address is held by R1
- (iii) MOV .@R1, A
- (iv) MOV 20h, @R1
- (v) MOV @R0, 03h

DRAWBACK

R_0 & R_1 are the only registers that can be used for pointers in this addressing mode. Since R_0 & R_1 are 8 bit code, their use is limited to accessing any information in the internal RAM.

- By using DPTR register we can access external memory.

ADVANTAGE

- It makes accessing data dynamic rather than static as in the case of direct addressing mode.
- Looping is possible in this addressing mode.

5) INDEXED ADDRESSING MODE

- This is widely used in accessing data elements of look-up table entries located in the program ROM space of 8051.

- The instructions used for this purpose is "MOV C"

Ex (i) $MOV A, @A + DPTR$

TE → The 16 bit register DPTR and register A are used to form the address of the data element stored in the on-chip ROM. Because the data elements are stored in the program code space ROM of 8051, instruction $MOV C$ is used instead of MOV . 'C' means code.

In the above example,

Add the contents of the accumulator with the contents of 16 bit register DPTR to form a program code memory location address of 16 bit. Then move the contents of this external memory address to accumulator.

(ii) $\text{MOVC A}, @A + PC$;

Add the contents of the accumulator with the
contents of the PC register to form a program code memory
location address. Move the contents of this external
memory address to the accumulator.

- only program memory can be accessed in the index addressing mode by MOVX instruction
Either DPTR or PC can be used as an Index register.
- 8051 has a total of 128 bytes of memory space
since 64k bytes of code added to 64k bytes
of data gives us 128k bytes.

NOTE - The major difference between the code & data space is that, data space cannot be shared between code & data.

(6) RELATIVE ADDRESSING MODE

- This is used only with conditional jump instructions.
- The relative address [offset] is an 8 bit signed number, which is automatically added to the PC to make the address of next instruction.
- The 8 bit signed offset value gives an address range of 127 to -128 locations.
- The jump destination is usually specified using label & the assembler calculates the jump offset.
- The advantage of relative addressing is that

the program code is easy to relocate and the address is relative to position in the memory.

Ex: SJMP LOOP1

JC BACK

1) ABSOLUTE ADDRESSING MODE

- This is used only by the AJMP [Absolute jump] and ACALL [Absolute call] instructions
- These are 2 byte instructions
- Absolute Addressing specifies the lowest 11 bit of memory address as a part of the instruction.
- The upper 5 bit of destination address are the upper 5 bit of current program counter.
- ∵ It allows branching only within the current 2 Kbyte page of program memory.

Ex: AJMP LOOP1

ACALL LOOP2

3) LONG ADDRESSING MODE

- This is used with the instructions LJMP and LCALL
- These are 3 byte instructions
- The address specifies a full 16 bit destination address so that a jump or a call can be made to a location within 64 K byte code memory space.

Ex: LJMP FINISH

LCALL DELAY

64 K Byte = $2^6 \times 2^{10}$ Byte = 2¹⁶ Byte

$\frac{2^{16}}{2^{16}} = 1$ Byte = 2⁰ Byte

(9) BIT INHERENT ADDRESSING MODE

→ In this the address of the flag which the operand is implied in the opcode of instruction.

Ex: CLR C ; Clears the carry flag to 0.

(10) BIT DIRECT ADDRESSING MODE

- In this addressing mode the direct address of the bit is specified in the instruction.
- The RAM space 20h to 2Fh and most of the special function registers are bit addressable.
- Bit address values are between 00h to 7Fh.

Ex: CLR 07h ; Clears the Bit 7 of 20h RAM space
SETB 07h ; Sets the bit 7 of 20h RAM space

(11) INHERENT ADDRESSING MODE

- This refers to a specific register such as accumulator or DPTR.

Ex: SWAP A ; Swaps within accumulator

★ INSTRUCTION SET

INSTRUCTION TIMINGS

- The 8051 internal operations & external read / write operations are controlled by the oscillator clock.
- The terms used in instruction set are -
 - T state
 - machine cycle
 - Instruction cycle

i) T-STATE

This is defined as one subdivision of the operation performed in one clock period. The term T-state & clock period is used.

ii) MACHINE CYCLE-

- This is defined as 12 oscillator periods.
- A machine cycle consists of six states & each state lasts for two oscillator periods.
- An instruction takes one to four machine cycles to execute an instruction.

iii) INSTRUCTION CYCLE-

- This is defined as the time required for completing the execution of an instruction.
- 8051 instruction cycle consists of one to four machine cycles.

Ex: If 8051 microcontroller is operated with 12MHz oscillator find the execution time for below instructions -

- (i) ADD A, 45H
- (ii) SUBB A, #55H
- (iii) MOV DPTR, #2000H
- (iv) MUL AB

Sols: since the oscillator frequency is 12MHz, the clock period is = $\frac{1}{12} \text{ MHz} = \underline{0.0833 \text{ sec.}}$

$$\text{Time for 1 machine cycle} = 0.0833 \text{ ms} \times 12 = \underline{1 \text{ ms.}}$$

<u>Instruction</u>	<u>No. of machine cycle</u>	<u>Execution time</u>
(i) ADD A, 45H	1	1 ms
(ii) SUBB A, #55H	2	2 ms
(iii) MOV DPTR, #2000H	2	2 ms
(iv) MUL AB	4	4 ms

★ 8051 INSTRUCTIONS

→ Based on the operations performed instruction
8051 is classified as,

- (1) Data transfer instructions
- (2) Arithmetic instructions
- (3) Logical instructions
- (4) Boolean instructions / Bit manipulation instruction
- (5) Program branching / Branch instructions / machine code
- (6) Subroutine instructions

(I) DATA TRANSFER INSTRUCTIONS

→ The instructions in this perform data transfer operations.

→ The instructions in this group are MOV, PUSH, POP,

(a) MOV :- MOV instruction copies data from one location to another location

Syntax - MOV operand 1, operand 2

format - MOV destination, source ; copies from source destination

(1) MOV A, Rn

→ This instruction moves the contents of Rn register accumulator. Rn register is not affected ; $(A) \leftarrow (R_n)$

NOTE: Rn may be R₀ to R₇ register of currently selected bank

Ex: MOV A, R₃

Before execution → R₃ = 58h , A = any value say 13h

After execution → A = 58h and R₃ = 58h

(6)

(2) MOV A, direct [direct address]

→ This instruction moves the contents of the address into A register ; $(A) \leftarrow (\text{direct})$

Ex: $\text{MOV A, } 60\text{h}$

Before execution $A \leftarrow \text{say } 10\text{h}$ After execution $A \leftarrow FF$
 $60\text{h} \leftarrow FF$

NOTE - Here 60h is a direct address. The direct address content 'FF' is moved into accumulator

(3) $\text{MOV A, } @R_i$

→ R_i is an internal register R_0 or R_1 ; $(A) \leftarrow ((R_i))$

Ex: $\text{MOV A, } @R_0$

Say $R_0 = 40\text{h} \leftarrow \text{address}$

$40\text{h} = FF \leftarrow \text{data}$

→ The address 40h is in register R_0

→ The data FF is in address 40h

Before execution

$R_0 \leftarrow 40\text{h}$

$40\text{h} \leftarrow FF$

$A \leftarrow 'xx'$

After execution

$R_0 \leftarrow 40\text{h}$

$40\text{h} \leftarrow FF$

$A \leftarrow FF$

(4) $\text{MOV A, } \# \text{data}$

→ 8 bit data is moved into accumulator ; $(A) \leftarrow \# \text{data}$

Ex: $\text{MOV A, } \#28$

Before Execution

$A \leftarrow 'xx'$

After Execution

$A \leftarrow 28$

(5) $\text{MOV R}_n, A$

→ The contents of accumulator is moved to register

R_n ; $(R_n) \leftarrow (A)$

→ R_n may be any register R₀-R₇ of the selected bank

Ex: MOV R₇, A

Before Execution

$$R_7 = 'xx'$$

$$A = FF$$

After Execution

$$R_7 = FF$$

$$A = FF$$

(6) MOV R_n, direct

→ R_n may be any register R₀ to R₇; (R_n) ← (direct)

Ex: MOV R₁, 40h

Before execution

$$40h = FF$$

$$R_1 = 'xx'$$

After execution

$$40h = FF$$

$$R_1 = FF$$

(7) MOV R_n, #data

→ R_n may be any register R₀ to R₇; (R_n) ← #data

Ex: MOV R₅, # 00

Before Execution

$$R_5 = 'xx'$$

After execution

$$R_5 = 00$$

(8) MOV direct, A

→ (direct) ← (A)

Ex: Mov 50h, R₃

Before Execution

$$50h = FF$$

$$R_3 = 00$$

After Execution

$$50h = 00$$

$$R_3 = 00$$

(9) MOV direct, direct

→ (direct) ← (direct)

Ex: MOV 30h, 50h

Before Execution

$$30h = 11$$

$$50h = 00$$

After Execution

$$30h = 00$$

$$50h = 00$$

(7)

) MOV direct, @ Ri

 $(\text{direct}) \leftarrow (\text{R}_i)$; R_i is only registers R₀ and R₁Ex: MOV 70h, @ R₀Before execution

$70h = 00$

$R_0 = 40h$

$40h = FF$

After Execution

$70h = FF$

$R_0 = 40h$

$40h = FF$

II) MOV direct, # data

 $\rightarrow (\text{direct}) \leftarrow \# \text{data}$ Ex: MOV 70h, #FFBefore execution

$70h = 'xx'$

After Execution

$70h = FF$

3) MOV @ R_i, direct $\rightarrow ((\text{R}_i)) \leftarrow (\text{direct})$ Ex: MOV @ R₁, 70hBefore execution

$R_1 = 40h$

$40h = 'xx'$

$70h = FF$

After execution

$R_1 = 40h$

$40h = FF$

$70h = FF$

3) MOV @ R_i, # data $\rightarrow ((\text{R}_i)) \leftarrow \# \text{data}$ Ex: MOV @ R₀, #00Before execution

$R_0 = 40h$

$40h = 'xx'$

After execution

$R_0 = 40h$

$40h = 00$

* MOV dest-bit, source-bit

 \Rightarrow Move bit data from source bit to destination bit

NOTE - one of the operands must be carry flag,
other may be any directly addressable
bit.

(14) MOV C, bit

$\rightarrow (C) \leftarrow (\text{bit})$

Ex : (i) MOV C, P1.4

<u>Before execution</u>	<u>After Execution</u>
$C = 'X'$	$C = 1$
$P1.4 = 1$	$P1.4 = 1$

(ii) MOV C, P0.7

<u>Before execution</u>	<u>After Execution</u>
$C = 'X'$	$C = 0$
$P0.7 = 0$	$P0.7 = 0$

(15) MOV bit, C

$\rightarrow (\text{bit}) \leftarrow (C)$

Ex : (i) MOV P1.2, C

<u>Before execution</u>	<u>After Execution</u>
$P1.2 = 'X'$	$P1.2 = 1$
$C = 1$	$C = 1$

(ii) MOV P3.7, C

<u>Before execution</u>	<u>After Execution</u>
$P3.7 = 'X'$	$P3.7 = 0$
$C = 0$	$C = 0$

(b) PUSH & POP INSTRUCTIONS

\rightarrow 'PUSH' is used for taking the values from register & storing in the starting address of the stack pointer i.e. 00h by using 'PUSH' operation.
For the next PUSH, it increments '+1' & stores

↓ Contd. next instructions will
be explained in page 8

the value in the next address of the stack pointer
ie 01h

'POP' → is used for placing the values from the stack pointer's maximum address to any other register's address, using POP again it decrements by 1 & the value is stored in any register given as POP

PUSH

[SP+2]
SP+1
SP

POP

SP
SP-1
SP-2

Ex: MOV R6, #25h
MOV R1, #12h
MOV R4, #F3h

[SP] = 07h ; Content of SP is 07 {default value}
[R6] = 25h ; Content of R6 is 25h
[R1] = 12h ; Content of R1 is 12h
[R4] = F3h ; Content of R4 is F3h

PUSH 6 [SP] = 08 [08] = [06] = 25h ; Content of 08 is 25h
PUSH 1 [SP] = 09 [09] = [01] = 12h ; Content of 09 is 12h
PUSH 4 [SP] = DA [0A] = [04] = F3h ; Content of DA is F3h

POP 6 [SP] = [06] = [DA] = F3h [SP] = D9 ; Content of 06 is F3h
POP 1 [SP] = [01] = [09] = 12h [SP] = D8 ; Content of 01 is 12h
POP 4 [SP] = [04] = [08] = 25h [SP] = 07 ; Content of 04 is 25h

(c) EXCHANGE INSTRUCTIONS [XCH] (contd. in Page 23)

→ The source ie register, direct memory or indirect memory will be exchanged with the contents of destination ie accumulator

Ex: XCH A, R3
XCH A, @ R1
XCH A, 54h

EXCHANGE DIGIT [XCHD]

Exchange the lower order nibble of accumulator [A₀-A₃] with lower order nibble of internal RAM location which is indirectly addressed by the register

Ex: XCHD A, @ R1
XCHD A, @ R0

(II) ARITHMETIC INSTRUCTIONS

→ 8051 can perform addition, subtraction, multiplication and division operations on numbers; increment & decrement operations.

(a) ADDITION

- memory format

- (1) Add the contents of A with immediate data or without carry
- (i) ADD A, #45h
 - (ii) ADDC A, #B4h

- (2) Add the contents of A with register R_n with or without carry

- (i) ADD A, R5
- (ii) ADDC A, R2

- (3) Add the contents of A with contents of memory with or without carry using direct & indirect addressing
- (i) ADD A, 51h
 - (ii) ADDC A, 75h
 - (iii) ADD A, @R1
 - (iv) ADDC A, @R0

NOTE 1 - CY, AC and OV flags will be affected by the operation.

NOTE 2 - OV is set ⁽¹⁾ if there is carry out of bit 6 no carry out from bit 7 or a carry out of bit 7 but no carry out from bit 6; otherwise OV=0

- (1) ADD A, R_n

→ (A) ← (A) + (direct) where R_n is R₀ to R₇

Ex: ADD A, R₄

(C) EXCHANGE INSTRUCTIONS

CONTD from page 2

1) XCH A, Rn

 $\rightarrow (A) \leftrightarrow (R_n)$

Ex: XCH A, R2

Before execution $A = FFh$ $R2 = 11h$ After Execution $A = 11h$ $R2 = FFh$ ~~[Exchange A, R2]~~

MOV R2, #11h

XCH A, R2

2) XCH A, direct

 $(A) \leftarrow (\text{direct})$

Ex: XCH A, 40h

Before execution $A = FFh$ $40h = 11h$ After Execution $A = 11h$ $40h = FFh$

3) XCH A, @ Ri

 $\rightarrow (A) \leftrightarrow ((R_i))$

Ex: XCH A, @ R1

Before Execution $40h = 11h$ $A = FFh$ After Execution $40h = FFh$ $A = 11h$

Program :
 MOV R1, #11h
 MOV R1, #40h
 MOV A, #FFh
 XCH A, @R1

4) XCHD A, @ Ri

\rightarrow The XCHD instruction exchanges only the lower nibble of accumulator [Bit 3-0] with lower nibble of the RAM location pointed by R_i (internal register R0 or R1).

\rightarrow The higher order nibbles [bit 7-4] of each registers are not affected

$\rightarrow A_{(3-0)} \leftrightarrow ((R_{i,3-0}))$

Ex: XCHD A, @ R1

Before execution

A = FFh

R1 = 50h

50h = 00h

After Execution

A = F0h

R1 = 50h

50h = 0Fh

(a) MOV INSTRUCTIONS

CONT'D from page ①

(16) MOV DPTR, # 16 bit value

→ The data pointer is loaded with 16 bit constant in

→ The DPH register holds high order byte while DPTR holds low order byte

→ $(DPTR) \leftarrow \# \text{data (0-15)}$

DPH $\leftarrow \# \text{data (15-8)}$

DPL $\leftarrow \# \text{data (7-0)}$

Ex: MOV DPTR, # 3456

After execution

DPH = 34h

DPL = 56h

(19) Mov x dest byte, source by

→ This instruction transfers data from external memory to register x appended to MOV

→ address of external memory can be 16 bit or 8 bit

MOVX A, @ Ri

$\rightarrow (A) \leftarrow ((Ri))$

Ex: MOVX A, @ R0

Program

MOV R0, # 50h

MOV 1234h, #FFh

MOVX A, @ DPTR

Before execution

R0 = 50h

50h = FFh

A = 'xx'

(17) MOVX A, @ DPTR

$\rightarrow (A) \leftarrow ((DPTR))$

Ex: MOVX A, @ DPTR

Program

MOV DPTR, # 1234

MOV 1234h, #FFh

MOVX A, @ DPTR

(20) MOVX (@Ri), A

$\rightarrow ((Ri)) \leftarrow (A)$

Ex: MOVX @ R1, A

Program

MOV R1, # 80h

MOV 1234h, #FFh

MOVX Y, @ R1

Before execution

1234h = FFh

DPTR = 1234

A = 'xx'

After execution

A = FFh

DPTR = 1234h

1234 = FFh

Before execution

R1 = 80h

80h = AAh

A = 'xx'

After execution

A = AA

R1 = 80

80h =

(18) MOVX @ DPTR, A

$\rightarrow ((DPTR)) \leftarrow (A)$

Program

MOV DPTR, # 1234

MOV 1234h, #FFh

MOVX A, @ DPTR, A

Before execution

DPTR $\leftarrow 1234h$

1234h \leftarrow FFh

A \leftarrow 'xx'

After execution

A \leftarrow FFh

DPTR = 1234h

1234 = FFh

Before Execution

$$A = 11$$

$$R_4 = 11$$

After Execution

$$A = 22 \quad \text{SF}$$

$$R_4 = 11$$

(1)

(ii) ADD A, R₇

Before Execution

$$A = 00$$

$$R_7 = FF$$

After Execution

$$A = FF \quad \text{SF}$$

$$R_7 = FF$$

ind.) ADD A, direct

DPL $\rightarrow (A) \leftarrow (A) + (\text{direct})$

Ex: ADD A, 40h

Before execution

$$A = 11$$

$$40h = 22$$

After execution

$$A = 33 \quad \text{SF}$$

$$40h = 22$$

iii) ADD A, @R_i

$(A) \leftarrow (A) + ((R_i))$; where $R_i = \text{internal register } R_0 \text{ or } R_1$

Ex: ADD A, @R₁

Before execution

$$A = 30$$

$$R_1 = 70h$$

$$70h = 20$$

After execution

$$A = 50 \quad \text{SF}$$

$$R_1 = 70h$$

$$70h = 20$$

iv) ADD A, #data

$\rightarrow (A) \leftarrow (A) + \#data$

Ex: ADD A, #01h

Before execution

$$A = 01$$

After execution

$$A = 02$$

v) ADDCA A, R_n

\rightarrow Simultaneously adds the contents of R_n register, carry flag & accumulator contents, result is stored in accumulator

$$\rightarrow (A) \leftarrow (A) + (C) + (Rn)$$

Ex (i) ADDCA, R7

Before Execution

$$A = 01$$

$$C = 1$$

$$R7 = 01$$

After Execution

$$A = 03$$

$$C = 1$$

$$R7 = 01$$

→ SUBTRACT

→ Subtract

with or

(i) SU

(ii) SU

(ii) ADD C, R0

Before execution

$$A = 01$$

$$C = 0$$

$$R0 = 01$$

After Execution

$$A = 02$$

$$C = 0$$

$$R0 = 01$$

→ Subtract

without

(i) SU

(ii) SU

(6) ADDC A, direct

$$\rightarrow (A) \leftarrow (A) + (C) + (\text{direct})$$

Ex (i) ADDC A, 80h

Before execution

$$A = 02$$

$$C = 1$$

$$80h = 02$$

After execution

$$A = 05$$

$$C = 1$$

$$80h = 02$$

(i)

(ii)

(iii)

(iv)

(7) ADDC A, @Ri

$$\rightarrow (A) \leftarrow (A) + (C) + ((Ri))$$

where $Ri = R_0 \text{ or } R_1$

Ex: ADDC A, @R0

Before execution

$$A = 01$$

$$C = 1$$

$$R0 = 70h$$

$$70h = 01$$

After execution

$$A = 03$$

$$C = 1$$

$$R0 = 70h$$

$$70h = 01$$

(i)

→

→

→

(8) ADDC A, #data

$$\rightarrow (A) \leftarrow (A) + (C) + \# \text{data}$$

Ex: ADDC A, #01h

Before execution

$$A = 01$$

$$C = 1$$

After execution

$$A = 03$$

$$C = 1$$

(b) SUBTRACTION

1) Subtract the contents of A with immediate data with or without carry

(i) SUBB A, #45H

(ii) SUBB A, #0B4H

2) Subtract the contents of A with register Rn with or without carry

(i) SUBB A, R5 $\rightarrow (A) = (A) - (R5) - (C)$

(ii) SUBB A, R2

3) Subtract the contents of A with contents of memory with or without carry using direct & indirect addressing

(i) SUBB A, 51H

(ii) SUBB A, 75H

(iii) SUBB A, @R1

(iv) SUBB A, @R0.

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

(1) SUBB A, source byte

→ Subtracts with borrow.

→ SUBB subtracts the source byte and the carry flag from the accumulator & puts the result in the accumulator

→ $(A) = (A) - (\text{Byte}) - (C)$ or $(A) = (A - \text{Byte} - C)$

→ SUBB instruction sets the carry flag according to the following :-

(i) destination > source
Byte Byte

CY=0 The result is
+ve

(ii) destination = Source Byte CY=0 The result is 0

(iii) destination < Source Byte CY=1 The result is 2's Complement

(C) MULTIPLICATION

MUL AB

→ This instruction multiplies two 8 bit unsigned which are stored in A and B register [contents of accumulator A is multiplied with the contents of register B]

→ Operation : $\begin{matrix} (A)_{7-0} \\ (B)_{15-8} \end{matrix} \leftarrow (A) \times (B)$

→ After multiplication the lower byte of the result be stored in accumulator and higher byte (15-8) result will be stored in B register

Ex: MOV A, #45H ; [A] = 45H
MOV B, #05H ; [B] = 05H
MUL AB ; [A] × [B] = 45 × 05 = 0225 H
[A] = 25H ; [B] = 02H.

Ex: MOV A, #50H
MOV B, #A0H
MUL AB

Before execution

$$50 \times A0 = 3200h$$
$$A = 50h$$
$$B = A0h$$

After execution

$$A = 00h \leftarrow (0-7) \text{ lower Byte}$$
$$B = 32h \leftarrow (15-8) \text{ higher Byte}$$

$$\text{Ex: } A = 100, B = 200 \Rightarrow 100 \times 200 = 20,000$$

$$A = 20h, B = 40h$$

(a) DIVISION

(11)

DIV AB

This instruction divides the 8 bit unsigned number which is stored in A by the 8 bit unsigned number which is stored in B register.

After division the result will be stored in accumulator and remainder will be stored in B register.

The accumulator receives the integer part of the quotient; register B receives the integer remainder.

The carry flag & overflow flags will be cleared.

Ex: MOV A, #45H

; [A] = 0E8H

MOV B, # 0F5H

; [B] = 1BH

DIV AB

; $[A]/[B] = E8/1B = 08H$ with remainder 10H

; [A] = D8H, [B] = 10H

Ex: A = 35, B = 10

DIV AB

A = 3, B = 5

$$\begin{array}{r} 3 \rightarrow \text{Quotient} \\ 10 \overline{) 35} \\ 30 \\ \hline 5 \rightarrow \text{Remainder} \end{array}$$

→ Here the quotient 3 is stored in accumulator & remainder B is stored in B-register

Ex: A = FBH, B = 12H
1 (251 dec) (18 dec)

DIV AB

A = 0DH (13 dec), B = 11H (17 dec)

$$\begin{array}{r} 13 \rightarrow \text{Quotient} \\ 18 \overline{) 251} \\ 234 \\ \hline 17 \rightarrow \text{Remainder} \end{array}$$

Ex: A = 97H (151 dec), B = 12H (18 dec)

DIV AB

A = 8, B = 7

* DA A [Decimal Adjust Accumulator]

→ When two BCD numbers are added, the answer is a non BCD number.

{ This instruction is used after addition of BCD to convert the result back to BCD }

→ To get the result in BCD we use DA A after addition.

→ The data is adjusted in the following 2 possible cases.

(i) If lower 4 bits (nibble) of A is greater than 9. Auxiliary carry (AC) = 1; 6 is added to lower nibble.

(ii) If upper 4 bits (nibble) of A is greater than 9. Carry [cy] is 1; 6 is added to upper nibble.

i.e., If $[A_{3-0} > 9] \vee [AC = 1]$ Then $(A_{3-0}) \leftarrow (A_{3-0}) + 6$

If $[A_{7-4} > 9] \vee [cy = 1]$ Then $(A_{7-4}) \leftarrow (A_{7-4}) + 6$

Ex(i)

MOV A, #47H ; A = 47h
 ADD A, #38H ; A = 47h + 38h = 7Fh ; Invalid BCD
 DA A ; A = 85h ; Valid BCD.

i.e., 47h

+ 38h

7 F h → Invalid BCD

+ 6 h → After DA A (add 6 to lower 4 bit).

8 5 h → Valid BCD

AC = 1

Since the lower nibble is greater than 9, DA adds 6 to A.

If the lower nibble is less than 9 but AC=1, it also adds 6 to the lower nibble.

(12)

Ex (ii) MOV A, #29h
ADD A, #18h
DA A

$$\begin{array}{r}
 29h \\
 + 18h \\
 \hline
 41h \rightarrow \text{Invalid BCD}
 \end{array}$$

$$\begin{array}{r}
 + 6 \\
 \hline
 47h \rightarrow \text{Valid BCD} \quad AC=1
 \end{array}$$

Ex (ii) MOV A, #52h
ADD A, #91h
DA A

$$\begin{array}{r}
 52h \\
 + 91h \\
 \hline
 E3h \rightarrow \text{Invalid BCD}
 \end{array}$$

$$\begin{array}{r}
 + 6 \\
 \hline
 143h \rightarrow \text{Valid BCD} \quad CY=1
 \end{array}$$

→ Add 6 {upper nibble is > 9}

Ex (iv) MOV A, #54h
ADD A, #87h
DA A

$$\begin{array}{r}
 54h \\
 87h \\
 \hline
 BB \rightarrow \text{Invalid BCD}
 \end{array}$$

$$\begin{array}{r}
 + 6 \\
 \hline
 141 \rightarrow \text{Valid BCD} \quad AC=1 \quad CY=1
 \end{array}$$

→ Add 6 {upper nibble > 9}
to both {lower nibble > 9}
upper & lower

(e) INCREMENT

- Increments the operand by one
- 'INC' increments the value of source by 1. If the initial value of register is FFh, incrementing the value will cause it to reset to 0.
- carry flag is not set when the value 'rolls over' from 255 to 0.
- In case of "INC DPTR", the value two byte unsigned integer value of DPTR is incremented. If the initial value of DPTR is FFFFh, incrementing the value causes it to reset to 0.

(1) INC A

$\rightarrow (A) \leftarrow (A) + 1$; accumulator increments by 1

Before Execution

$$A = 01$$

After Execution

$$A = 02$$

(2) INC R_n

$\rightarrow (R_n) \leftarrow (R_n) + 1$; register R_n (R₀ \rightarrow R₇) increments by 1

Ex: INC R0

Before Execution

$$R_0 = 30h$$

After Execution

$$R_0 = 31h$$

(3) INC direct

$\rightarrow (\text{direct}) \leftarrow (\text{direct}) + 1$; value in direct location is incremented by 1.

Ex: INC 40h

Before Execution

$$40h = 01$$

After Execution

$$40h = 02$$

(4) INC @ R_i where R_i \rightarrow internal register R₀ or R₁

$\rightarrow ((R_i)) \leftarrow ((R_i)) + 1$; value in R_i is incremented by 1.

Ex: INC @ R0

Before Execution

$$R_0 = 80h$$

$$80h = 01$$

After Execution

$$R_0 = 80h$$

$$80h = 02$$

(5) INC DPTR

\rightarrow This instruction increments the 16 bit register Content (DPTR) by 1. This is the only 16 bit register that can be incremented

$\rightarrow (DPTR) \leftarrow (DPTR) + 1$

Ex: INC DPTR

(i) Before execution

$$DPTR = 16$$

$$[DPH = 16, DPL = 1]$$

Before execution

$$DPTR =$$

$$[DPH = 00, DPL = 0]$$

f) DECREMENT

DEC decrements the initial value causing the carry flag from 0

1) DEC A

$$(A) \leftarrow$$

Befor

2) DEC

$$\rightarrow (R_n) \leftarrow$$

Ex:

Ex: INC DPTR

(B)

(i) Before Execution

DPTR = 16 FFh
[DPH = 16, DPL = FF]

After Execution

DPTR = 17 00h
[DPH = 17, DPL = 00]

1 (ii) Before Execution

DPTR = 00 FFh
[DPH = 00, DPL = FF]

After Execution

DPTR = 01 00h
[DPH = 01, DPL = 00]

(f) DECREMENT

DEC decrements the value of source by 1.

If the initial value is 0, decrementing the value will cause it to reset to FFh

carry flag is not set when the value "rolls over" from 0 to FFh.

(1) DEC A

$\rightarrow (A) \leftarrow (A) - 1$; decrements value of accumulator by 1

Before Execution

A = 05h

After Execution

A = 04h

(2) DEC Rn

$\rightarrow (R_n) \leftarrow (R_n) - 1$; decrements the value of register Rn by 1

Ex: DEC RD

Before Execution

RD = 80h

80h = 0Sh

After Execution

RD = 80h

80h = 04h.

(3) DEC direct

$$\rightarrow (\text{direct}) \leftarrow (\text{direct}) - 1$$

Ex: DEC 40h

Before Execution

$$40h = 05h$$

After Execution

$$40h = 04h$$

TE - 'AND'
some

) ANL A

(A) ←

Ex: ANL

Before

A =

40h:

ANL A

(A) ←

Ex: A

Be

(4) DEC @ Rⁱ

$$\rightarrow ((R^i)) \leftarrow ((R^i)) - 1$$

Ex: DEC @ R0

Before Execution

$$R0 = 80h$$

$$80h = 04h$$

After Execution

$$R0 = 80h$$

$$80h = 03h.$$

Ex: A

Be

(III) LOGICAL INSTRUCTIONS

→ 8051 performs logical AND, logical OR, logical NOT instructions

a) LOGICAL AND

→ ANL destination-byte, source-byte

→ logical-AND for byte variables.

→ ANL performs the bitwise logical-AND operation between the variables indicated & stores the result in the destination variables.

→ The value in source is not affected

(1) ANL A, Rn

$$\rightarrow (A) \leftarrow (A) \wedge (R_n)$$

Ex: ANL A, R5

Before Execution

$$A = 39$$

$$R5 = 09$$

After Execution

$$A = 09$$

$$R5 = 09$$

$$\begin{array}{r}
 0011\ 1001 \rightarrow \\
 0000\ 1001 \rightarrow \\
 \hline
 0000\ 1001 \rightarrow
 \end{array}$$

OTE - 'AND' instruction logically AND the bits of source & destination 14

(2) ANL A, direct

$$\rightarrow (A) \leftarrow (A) \wedge (\text{direct})$$

Ex: ANL A, 40h

Before execution

$$A = 39$$

$$40h = 09$$

After execution

$$A = 09$$

$$40h = 09$$

$$\begin{array}{r} 39 \rightarrow 00111001 \\ \wedge 09 \rightarrow 00000001 \\ \hline 00110000 \rightarrow 10h \end{array}$$

(3) ANL A, @ R_i; R_i → R₀ or R₁

$$\rightarrow (A) \leftarrow (A) \wedge ((R_i))$$

Ex: ANL A, @ R₀

Before execution

$$A = 39$$

$$R_0 = 80h$$

$$80h = 09$$

After execution

$$A = 09$$

$$R_0 = 80h$$

$$80h = 09$$

$$\begin{array}{r} A = 39 \rightarrow 00111001 \\ \wedge 09 \rightarrow 00000001 \\ \hline 00110000 \rightarrow 10h \end{array}$$

(4) ANL A, # data

$$\rightarrow (A) \leftarrow (A) \wedge \# \text{data}$$

Ex: ANL A, #09h

Before execution

$$A = 39$$

After execution

$$A = 09$$

(5) ANL direct, A

$$\rightarrow (\text{Direct}) \leftarrow (\text{Direct}) \wedge (A)$$

Ex: ANL 40h, A

Before execution

$$40h = 39$$

$$A = 09$$

After execution

$$40h = 09$$

$$A = 09$$

(6) ANL C, source_bit

- logical AND for bit variables
- In this instruction carry flag bit is ANDed with source bit & the result is placed in C, i.e. if source bit = 0, then CY=0 otherwise CY=1.

$$\rightarrow [C] \leftarrow [C] \wedge (\text{bit})$$

Ex(i) ANL C, ACC.7

Before execution

$$C = 1$$

$$A = 13$$

After execution

$$C = 0$$

$$A = 13$$

$$A = 13 = 0001\ 00$$

$$C = 1 = 11$$

$$\text{Result} \rightarrow 000100$$

Ex(ii) ANL C, P2.2

Before execution

$$C = 1$$

$$P2 = 00001011 \text{ B}$$

or

$$P2 = 0Bh$$

After execution

$$C = 0$$

$$P2 = 00000000 \text{ B}$$

$$P2 = 00000000 \text{ H}$$

$$C = 00000000 \text{ B}$$

$$\text{Result } C =$$

$$0$$

(7) ANL C, 1 Bit

NOTE: 1 Bit → means invert the bit data (NOT)

→ AND carry bit with inverse of bit data
[complement]

$$\rightarrow [C] \leftarrow [C] \wedge 1 (\text{Bit})$$

Ex:(i) ANL C, 1 ACC.7

Before execution

$$C = 1$$

$$Acc = 01111111 \text{ B}$$

or

$$Acc = 7Fh$$

After execution

$$C = 1$$

$$Acc.7 = 01111111$$

$$Acc.7 = 10000000$$

$$Acc.7 = 11111111$$

$$Acc.7 = 11111111$$

$$1 \quad C = 1$$

$$\text{Result} \rightarrow 1 \leftarrow C$$

(15)

x (ii) ANL C, P2.0

Before execution

$$C = 1$$

$$P2 = 0001\ 0000 \text{ B}$$

OR

$$P2 = 10 \text{ h}$$

After execution

$$C = 1$$

$$P2 = 10 \text{ h}$$

$$P2.0 = 0111\ 0000 [] \text{ B}$$

$$P2.0 = 0001\ 0000 [] \text{ B}$$

$$P2.0 = 0001\ 0000 [] \text{ B}$$

$$\wedge C = \underline{\quad} [] \rightarrow C$$

Result =

LOGICAL OR

→ ORL destination-byte, source-byte

→ ORL performs the bitwise logical 'or' operation between source & destination, leaving the resulting value in destination

→ The value of source is not affected

→ ORL instruction can be used to set certain bits of an operand to 1.

→ "OR" instruction logically OR the bits of source & destination

(i) ORL A, Rn

→ $(A) \leftarrow (A) \vee (Rn)$

→ This instruction performs a logical OR on the byte operands, bit by bit and stores the result in the destination

Ex: ORL A, R5

Before execution

$$A = 32 \text{ h}$$

$$R5 = 50 \text{ h}$$

After execution

$$A = 72 \text{ h}$$

$$R5 = 50 \text{ h}$$

$$A = 32 \Rightarrow 0011\ 0010$$

$$R5 = 50 \Rightarrow 0101\ 0000$$

$$\text{Result} \rightarrow 0111\ 0010$$

$$A = 72 \text{ h}$$

Program

MOV A, #32h

MOV R4, #50h

ORL A, R4

$\rightarrow (A) \leftarrow (\text{direct})$

Ex: ORL A, 30h

Program Before execution
 $A = 32h$
 $30h = 50h$

After execution
 $A = 72h$
 $50h = 50h$

(2) ORL A, @ R_i; R_i \rightarrow R₀ or R₁ only

$\rightarrow (A) \leftarrow (A) \vee ((R_i))$

Ex: ORL A, @ R₁

Before execution
 $A = 32h$
 $R_1 = 30h$
 $30h = 50h$

After execution
 $A = 72h$
 $R_1 = 30h$
 $30h = 50h$

(4) ORL A, # data

$\rightarrow (A) \leftarrow (A) \vee \# \text{data}$

Ex: ORL A, # 50h

Before execution
 $A = 32h$
 $\text{data} = 50h$

After execution
 $A = 72h$

(5) ORL direct, A

$\rightarrow (\text{direct}) \leftarrow (\text{direct}) \vee (A)$

Ex: ORL 30h, A

Before execution

MOV #30h
 MOV 30h, #50h

ORL 30h, A

After execution

$A = 72h$

$30h = 50h$

6) ORL direct, #data

(16)

$\rightarrow (direct) \leftarrow (direct) \vee \#data$

Ex: ORL 30h, #32h

Before execution		After execution
$30h, \#50h$		
$30h, \#32h$	$30h = 50h$	$30h = 72h$
	data = 32h	

7) ORL C, source-bit

Logical OR for bit variables.

The carry flag bit is 'OR'ed with a source bit and the result is placed in the carry flag.
∴ If the source bit is 1, CY is set
otherwise CY flag remains unchanged.

ORLC, bit

$\rightarrow (C) \leftarrow (C) \vee (bit)$

Ex (i) ORL C, ACC.3

Before execution

A = FF

CY = 0

i.e., A = 1111 1111

After execution

A = FF

CY = 1

Ex (ii) Before execution

A = F7

CY = 0

i.e., A = 1111 0111

After execution

A = F7

CY = 0

(8) ORLC, 1 bit
 $\rightarrow (C) \leftarrow (C) \vee (\overline{B \text{ bit}})$

Ex(i) ORLC C, 1 Acc. 3

Before execution

$$A = 00011000_6$$

i.e. $A = 18 \text{ h}$

$$CY = 0$$

After execution

$$CY = 0$$

$$A = 0001\boxed{0}000 \quad \text{Bit 3}$$

$$+ CY = \underline{\quad - 1 \quad}$$

$$\text{Result } CY = \underline{\quad 0 \quad}$$

Ex: XRL A, Rn

Before execution
 $A = 39$
 $R3 = 00$

(2) XRL A, direct

$$(A) \leftarrow (A) \vee (d)$$

Ex: XRL A, 30

Before

30h
 \downarrow
 $A, F, \dots, 39h$
 \downarrow
 $30h, \dots, 09h$
 \downarrow
 $1, 30h$

After execution

$$A = 0001\boxed{1}000 \quad \text{Bit 3}$$

$$CY = \underline{\quad + 0 \quad}$$

$$\text{Result } CY = \underline{\quad 1 \quad}$$

$$CY = 1$$

(3) XRL A, @

$$\rightarrow (A) \leftarrow (A)$$

Ex: XRL

Program
 \downarrow
 $1, F, \dots, 39h$
 \downarrow
 $F, \dots, 50h$
 \downarrow
 $50h, \dots, 09h$
 \downarrow
 $XRL A, @R1$

$$\rightarrow (A) \leftarrow$$

Ex: X

Program
 \downarrow
 $MOV A, \#$
 \downarrow
 $PLF, \#$

(C) LOGICAL XOR

\rightarrow XRL destination-byte, source-byte

\rightarrow XRL does a bitwise "EX-OR" operation between source and destination, leaving the resulting value in destination.

\rightarrow The value in source is not affected

\rightarrow 'XRL' instruction logically EX-OR the bits of source & destination.

(1) XRL A, Rn

$$\rightarrow (A) \leftarrow (A) \vee (Rn)$$

$A \rightarrow \text{XOR}$

$Rn \rightarrow \text{Register R}_0 \text{ to } R_7$

A	B	$A \text{ xor } B$
0	0	0
0	1	1
1	0	1
1	1	0

Ex: XRL A, R3

(17)

Before execution

$$A = 39h$$

$$R3 = 09h$$

After execution

$$A = 30h$$

$$R3 = 09h$$

$$\begin{array}{r} 00111001 \rightarrow A (39) \\ 00001001 \rightarrow R3(09) \\ \hline 00110000 \rightarrow A \text{ result} \\ 30h \end{array}$$

(2) XRL A, direct

$\rightarrow (A) \leftarrow (A) \vee (\text{direct})$

Ex: XRL A, 30h

Program

MOV F, #39h

MOV 30h, #09h

XL A, 30h

Before execution

$$A = 39h$$

$$30h = 09h$$

After execution

$$A = 30h$$

$$30h = 09h$$

$$\begin{array}{r} A = 30 = 00111001 \\ 30h = 09 = 00001001 \\ \hline A = 30 \rightarrow 00110000 \\ \text{Result} \end{array}$$

(3) XRL A, @ Ri ; $Ri \rightarrow RD \text{ or } RI$

$\rightarrow (A) \leftarrow (A) \vee ((Ri))$

Ex: XRL A, @ RI

Before execution

$$A = 39h$$

$$RI = 50h$$

$$50h = 09h$$

After execution

$$A = 30h$$

(4) XRL A, # data

$\rightarrow (A) \leftarrow (A) \vee \# \text{data}$

Ex: XRL A, # 09h

Before execution

$$A = 39h$$

$$\text{data} = 09h$$

After execution

$$A = 30h$$

MOV A, #39h

XRL A, #09h

(5) XRL direct, A

$\rightarrow (\text{direct}) \leftarrow (\text{direct}) \oplus (\text{A})$

Ex: XRL 50h, A

Mov A, #39h
Mov 50h, #09h
XRL 50h, A

Before execution

A = 39h

50h = 09h

After execution

A = 30h

(6) XRL direct, #data

$\rightarrow (\text{direct}) \leftarrow (\text{direct}) \oplus \# \text{data}$

Ex: XRL 50H, #09h

Mov 50h, #39h
XRL 50h, #09h

Before execution

50h = 39h

data = 09h

After execution

50h = 30h

NOTE - XRL is used to check whether the two registers have same value.

If the value is same then '0' is placed accumulator or location [destination]

★ ROTATE INSTRUCTIONS

(1) RL A

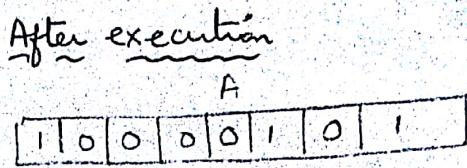
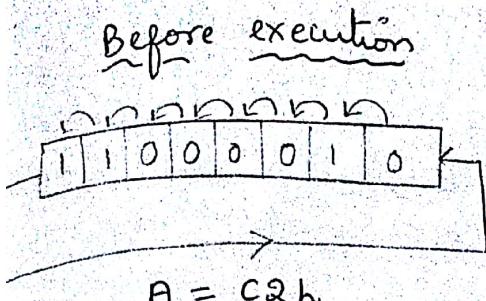
\rightarrow Rotate accumulator left

\rightarrow The eight bits in the accumulator are rotated one bit to the left.

Bit 7 is rotated into the bit 0 position

$\rightarrow (A_{n+1}) \leftarrow (A_n)$

$(A_0) \leftarrow (A_7)$ where $n = 0 \rightarrow 6$



$$A = 85h \\ \text{ie, } A = 10000101 \text{ b}$$

1) RR A

rotate accumulator right

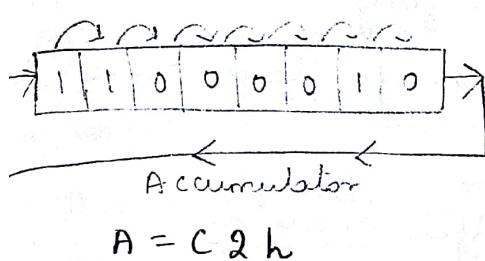
The eight bits in the accumulator are rotated one bit to the right.

Bit 0 is rotated into bit 1 position

$$(A_n) \leftarrow (A_{n+1}) ; n=0 \rightarrow 6$$

$$(A_1) \leftarrow (A_0)$$

Before execution



After execution



$$A = 61h$$

$$\text{ie, } A = 01100001 \text{ B}$$

1) RLC A

→ Rotate accumulator left through the carry flag

$$\rightarrow (A_{n+1}) \leftarrow (A_n)$$

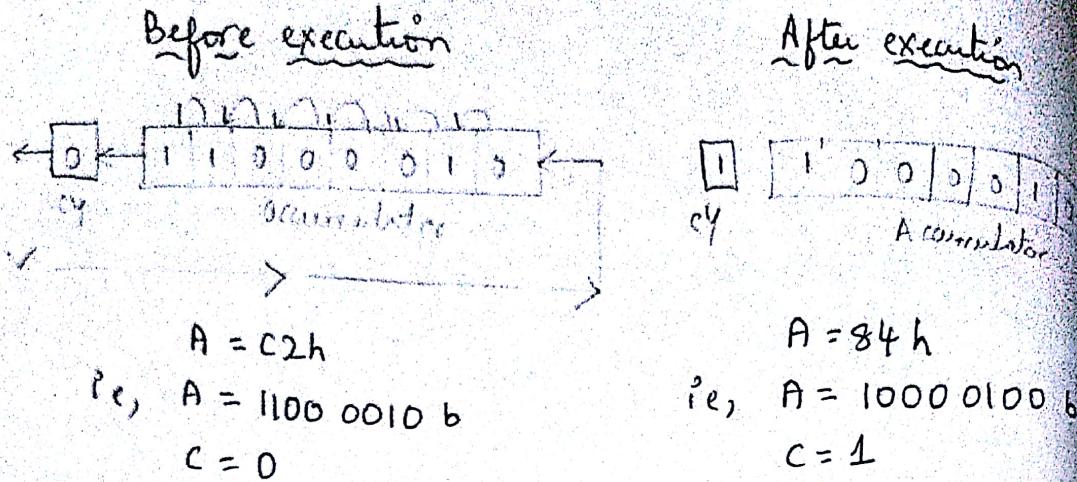
$$(A_0) \leftarrow (C)$$

$$(C) \leftarrow (A_7)$$

→ The 8 bits in the accumulator & the carry flag are together rotated one bit to the left

Bit 7 moves into the carry flag

The original state of the carry flag moves into the bit 0 position



(4) RRC A

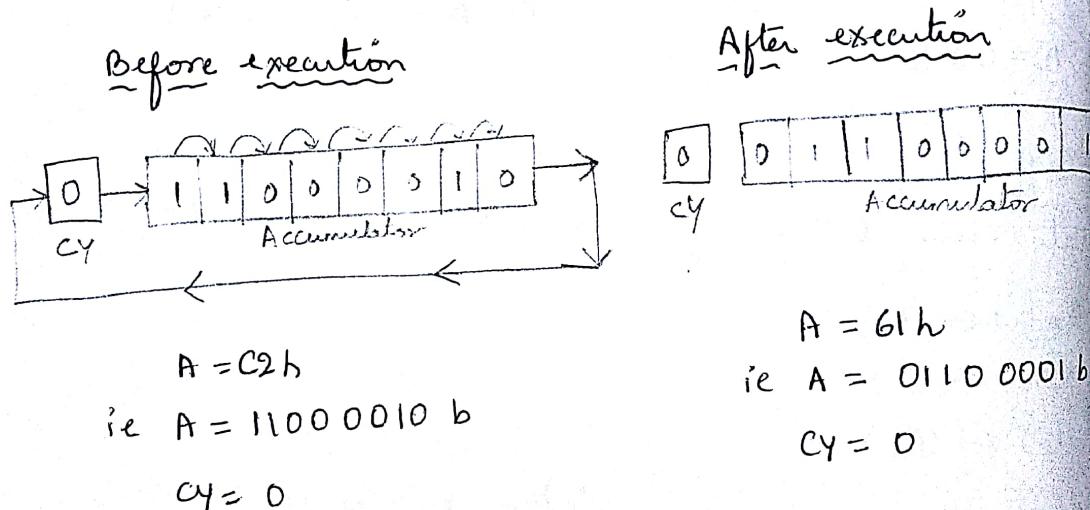
→ Rotate accumulator right through the carry flag

$$(A_n) \leftarrow (A_{n+1})$$

$$(A_7) \leftarrow (C)$$

$$(C) \leftarrow (A_0)$$

→ The 8 bits in the accumulator and the carry flag are together rotated 1-bit to the right
 Bit 0 moves into the carry flag
 The original state of the carry flag moves the bit 7 position



(5) NOP

→ NO operation

→ This instruction performs no operation & execution continues with the next instruction

It is sometimes used for timing delays to waste clock cycles.

This instruction only updates the program counter [PC] to point to the next instruction following

NOP

$$(PC) \leftarrow (PC) + 1$$

(d) LOGICAL NOT

→ CPL complements operand, leaving the result in operand

→ If operand is a single bit then the state of the bit will be reversed.

→ If operand is the accumulator then all the bits in the accumulator will be reversed

(1) CPL A

→ Complement accumulator

→ This instruction complements the contents of the accumulator

→ The result is 1's complement of the accumulator
ie, 0's become 1's and 1's become 0's

→ $(A) \leftarrow T(A)$ * $T(A)$ means $\text{NOT } (A)$

or

\overline{A}

Ex: CPL A

Before execution

MOV A, #FFh

A = FFh

After execution

A = 00h

CPL A

A = 00h

A = FFh

(2) CPL Bit

→ complement bit

→ This instruction complements a specified single bit

→ The bit can be any bit addressable location

Ex (i) CPL P0.3

Before execution

P0.3 = 1 (I/O pin)

After execution

P0.3 = 0 (I/O pin)

Ex (ii) CPL P3.3

Before execution

P3.3 = 0 (I/O pin)

After execution

P3.3 = 1 (I/O pin)

(3) CPL C

→ Complements carry bit

→ (C) ← T(C)

→

Before execution

(i) CY = 0

(ii) CY = 1

After execution

CY = 1

CY = 0.

(4) SWAP A

→ Swap nibbles within the accumulator

→ The swap instruction interchanges the lower nibble ($A_0 - A_3$) with the upper nibble ($A_4 - A_7$) inside register A

Ex: MOV A, #59h

SWAP A

Before execution

A = 59h.

i.e. A = 0101 1001 b

After execution

A = 95h

i.e., A = 1001 0101 b.

IV) BOOLEAN INSTRUCTIONS

(1) CLR A

→ clear accumulator

→ The accumulator is cleared [$A=00h$] , all bits of the accumulator are set to 0

, $(A) \leftarrow 0$

Ex: $\text{MOV } A, \#FFh$
 $\text{CLR } A$

Before execution

$A = FFh$

After execution

$A = 00h$

(2) CLR bit

→ clear bit

, This instruction clears the indicated bit

→ The bit can be the carry flag or any bit addressable location in 8051

→ $(C) \leftarrow 0$

Ex (i) $\text{CLR } C ; C4 = 0$

(ii) $\text{CLR P2.4} ; \text{clear P2.5 (Port 2's 5th Pin is cleared)}$

(iii) $\text{CLR P1.2} ; \text{clear P1.2 (P1.2 = 0)}$

(iv) $\text{CLR ACC.7} ; \text{clear D7 of accumulator (ACC.7 = 0)}$

Ex: $\text{MOV } A, \#FFh$

CLR ACC.7

Before execution

$Acc = [1]1111111$

After execution

$Acc = [0]1111111$

Ex: $\text{CLR } C$

Before Execution
 $C4 = 1$

After Execution
 $C4 = 0$

Ex: CLR P1.2

Before Execution

After Execution

(3) SETB C

→ $(C) \leftarrow 1$
→ sets the carry bit

Ex: Before execution

(i) $CY = 0$

(ii) $CY = 1$

After execution

$CY = 1$

$CY = 1$

(4) SETB Bit

→ $(Bit) \leftarrow 1$
→ Set specified bit

Ex: SETB P1.0

Before execution

$P1.0 = 0$ [Op pin]

After execution

$P1.0 = 1$ [I/p pin]

(V) BRANCH INSTRUCTIONS

* JUMP & CALL INSTRUCTIONS -

- Jump and call instructions change the flow of the program by changing the contents of program counter
- A jump permanently changes the program whereas call temporarily changes the program to allow another part of the program to run.
- jump instructions are classified into
 - (i) conditional jump
 - (ii) unconditional jump
- The jump instruction which changes the program flow if certain condition exists is called as

The jump instruction which changes the program flow irrespective of the condition [NOT depends on any condition] is called unconditional jump.

COMPARE INSTRUCTION → CJNE [Compare & jump if not equal]

This instruction compares the magnitude of the first two operands & changes program flow if their values are not equal.

The instructions that changes the program flow are :

- (a) Jump on bit conditions
- (b) compare byte & jump if not equal
- (c) decrement byte & jump if zero
- (d) Jump unconditionally
- (e) call a subroutine
- (f) Return from a subroutine

(I) JB bit, rel (rel → relative address)

→ jump if bit set

→ If the indicated bit is a one [1], jump to the address indicated ; otherwise proceed with the next instruction

The bit tested is not modified

$$\rightarrow (PC) \leftarrow (PC) + 3$$

If (bit) = 1

$$\text{Then, } (PC) \leftarrow (PC) + \text{rel}$$

Ex: ^{Say} Address

0300	JB ACC.0, down
0301	
0302	
0304	— down INC R0
0305	

After execution

ACC. D = 1

PC = 0304h

ACC. O = 0

PC = 0301h

Before execution

(i) PC = 0300h

Say ACC. O = 1

(ii) PC = 0300h

Say ACC. O = 0

NOTE - (1) If condition is false then processor
next instruction

(2) If condition is true, then processor
the specified address [label] & execute
address instruction

Ex :- SETB P1.5

Here : JB P1.5, Here
mov P2, # 55h

Ex: SETB P1.4

Here: JB P1.4, here
mov A, B

(2) JNB Bit, rel

→ Jump if bit NOT set.

→ If the indicated bit is a zero, branch to the indicated address ; otherwise proceed with the next instruction

The bit tested is not modified

→ $(PC) \leftarrow (PC) + 3$

If $(Bit) = 0$

Then

$(PC) \leftarrow (PC) + rel$

say address

0300

0301

0302

0303

JNB Acc.0, down
INC R0
↓
down : Dec RI

Before execution

(i) PC = 0300h

say ACC.0 = 1

Here ACC.0 = 1, the condition is false so the processor executes next instruction ie INC R0

(ii) PC = 0300h

say ACC.0 = 0

Here the condition is true, so the processor jumps to the address specified by the label & executes that instruction

Ex(iii) SETB ACC.0

here : JNB ACC.0, here

MOV R2, R3

Ex (iii) JNB ACC.0, Next

INC A

Next : DEC A

After execution

ACC.0 = 1

PC = 0301h

ACC.0 = 0 after execution
PC = 0303.

Ex(iv) CLR A

here : JNB ACC.0, here

MOV R2, R3

(3) TBC bit, rel rel → relative address

→ Jump if bit is set & clear bit

→ If the desired bit is high it will jump to the target address [specified address ie label]

at the same time the bit is cleared to zero

→ If the desired bit is low, then the processor proceeds with the next instruction ie executes

$\rightarrow (PC) \leftarrow (PC) + 3$
 If (bit) = 1
 Then, (bit) $\leftarrow 0$
 $(PC) \leftarrow (PC) + \text{rel}$

Ex: 0300 SET B Acc.7
 0301 JBC Acc.7, Next
 0302 MOV P1, A
 0303
 Next: INC R0

Before execution

Acc.7 = 1
PC = 0301

After execution

Acc.7 = 0
PC = 0303

(4) JC target or JC rel

- Jump if carry is set ie, $CY = 1$
- If the carry flag is set, branch to the address indicated, otherwise proceed with the next instruction

$\rightarrow (PC) \leftarrow (PC) + 2$

If $(C) = 1$
Then,

$(PC) \leftarrow (PC) + \text{rel}$

Ex: Say address

0300 SETB C
 0301 JC, Next
 ...
 0345 INC A

If $CY = 0$

If $CY = 1$

Next: INC R0

If $CY = 1$, then jumps to address specified by label

If $CY = 0$, then executes next instruction

Before execution

CY=1

PC = 0301

After execution

CY=1

PC = 0345

(23)

JNC target or JNC rel

→ Jump if NO carry [CY=0]

→ This instruction checks the carry flag, & if CY=0 it will jump to the target address.

$(PC) \leftarrow (PC) + 2$

If (C)=0

Then

$(PC) \leftarrow (PC) + \text{rel}$

Ex: MOV A, #0Fh

MOV B, #0FOh

Add a, b

JNC, down

INC R0

ADD C a, #30h

JNC, down

down; MOV 40h, a

→ This instruction checks the carry flag, if CY=0 [condition true], then jumps to the specified label.

If CY=1 [condition false] then executes next instruction

(6) JZ target or JZ rel

→ Jump if A=0 [Jump if Accumulator zero]

→ If all bits of the accumulator are zero, branch to the indicated address, otherwise proceed with the next instruction

$(PC) \leftarrow \text{part 1} + \text{rel}$

Ex: MOV A, #
 MOV B, #02h
 ADD A, B
 JZ, down
 INC R0
 down: MOV 40h, A

end

- If $A = 0$ [condition is true] then jumps to the label down specified by the label down in the above example ie MOV 40h, A
- If $A \neq 0$ [condition is false] then proceeds to the next instruction ie INC R0 in above example

(7) JNZ target or JNZ rel

- Jump if accumulator is NOT zero
- If any bit of the accumulator is 1 branch to the indicated address; otherwise proceed with next instruction
- ie If $A \neq 0$ [condition is true] then branch to the indicated address

If $A = 0$ [condition is false] then proceed to the next instruction

$$\rightarrow (PC) \leftarrow (PC) + 2$$

if $A \neq 0$

$$\text{Then } (PC) \leftarrow (PC) + \text{rel}$$

Ex: MOV A, #0Fh
 MOV B, #C0h
 ADD A, B
 JNZ, down
 INC R0
 ADD C - A, #01h

JNZ, down
 INC R1
 down: MOV 40h, A

If $A \neq 0$ [True condition], then jumps to the address specified by the label down ie $\text{Mov } 40h, A$ (24)
 If $A = 0$ [false condition], then proceeds with the next instruction ie $\text{INC } R0$ and $\text{INC } R1$

(b) DJNZ byte, target
 decrement & jump or DJNZ byte, rel-address
 In this instruction if not zero the result is a byte is decremented & if target address. NOT zero it will jump to the

$(PC) \leftarrow (PC) + 2$
 $(Rn) \leftarrow (Rn) - 1$

If $(Rn) \neq 0$ ie $(Rn) > 0$ or $(Rn) < 0$

Then

$(PC) \leftarrow (PC) + \text{rel}$

Ex: $\text{MOV } R0, \#30h$
 $\text{MOV } R1, \#40h$
 $\text{MOV } R3, \#05h$

Up: $\text{MOV } A, @R0$
 $\text{ADDCA}, \#01h$
 $\text{MOV } @R1, A$
 $\text{INC } R1$
 $\text{INC } R0$
 $\text{DJNZ } R3, \text{up}$
 end

It first decrements the contents of $R3$ register & then checks the condition ie $R3 \neq 0$. If $R3$ contents is not zero, then jumps to the indicated by the label.

* If $r_3 = 0$ [condition is false] then
next instruction ie end

(9) DJNZ direct, rel

$$\rightarrow (PC) \leftarrow (PC) + 2$$

$$(\text{direct}) \leftarrow (\text{direct}) - 1$$

If direct $\neq 0$ ie $(\text{direct}) > 0$ or $(\text{direct}) \neq 0$

Then, $(PC) \leftarrow (PC) + \text{rel}$

Ex: MOV R0, #30h

MOV R1, #40h

MOV 40h, #05h

up: mov a, @R0

ADD a, @

mov @R1, a

INC R1

INC R0

PJNZ 40h, up

end.

\rightarrow First decrements the contents of 40h [address] then checks the condition ie $40h \neq 0$

If contents of 40h address is not zero then

then jumps to the specified address

indicated by the label.

\rightarrow If $(40h) = 0$ [condition is false] then executes the next instruction ie END

NOTE -

The Target address can be no more than 127 bytes backward or 127 bytes forward until next instruction

1) ~~STMP~~ ~~and (PC)~~ ~~STMP~~ * 8 bit address
 short jump
 program control branches unconditionally to the
 address indicated
 $(PC) \leftarrow (PC) + 2$
 $(PC) \leftarrow (PC) + \text{rel}$
 $\text{org } 00h$
 2) JMP over
 $30h$
 org
 over: $\text{MOV A, } \#10h$

1) LJMP 16 bit address

long jump

LJMP causes an unconditional branch to the indicated address, by loading the high order & low order bytes of the PC

$(PC) \leftarrow (\text{address})_{6-15}$

The destination may be anywhere in the full 64K bytes program address space.

2) JMP @ A + DPTR

→ Jump Indirect

The jump instruction is unconditional jump to a target address. The target address is provided by total ~~was~~ sum of register A & DPTR. register:

→ Jump instructions are classified into,

(a) conditional jump

(b) unconditional jump

(a) CONDITIONAL JUMP

→ Depending upon the condition i.e True or false program branches [jumps] to the specified address
ie If condition is True → then JUMP to specified label

If condition is false → NO jump, executes Next instruction

→ All conditional jumps are short jumps, meaning that the target address cannot be more than -128 bytes backward to +127 bytes forward of the PC of the instruction following the jump.

* If the target address is beyond the -128 to +127 byte range, the assembler gives an error

* If the target address is ^{within} beyond -128 to +127 byte range, the assembler gives no error

(b) UNCONDITIONAL JUMP

→ Jumps to the specified address without any condition [unconditionally jumps to the specified address]

→ The unconditional jump instructions are

(1) SJMP 8 bit address

(2) LJMP 16 bit address

(3) JMP @ A + DPTR

SJMP

- 1) This is a 2-byte instruction. The 1st byte is the opcode & the second byte is the address number, which is added to the instruction following the SJMP to get the target address. (26)
- In this jump, the target address must be within 128 to +127 bytes of the PC of the instruction.

LJMP

- 1) This is a 3-byte instruction, the 1st byte is the opcode & the next two bytes are the target address.
- The LJMP is used to jump to any address location within the 64 Kbytes code space of 8051.
- Ex: LJMP 3000H

JMP @ A + DPTR

- 1) The target address is provided by the total sum of register A & DPTR register.
- This instruction is not widely used.

CONDITIONAL JUMPS

1) PUSH direct

- Push onto stack
- The stack pointer is incremented by one.
- The contents of the indicated variable is then

copied into the internal RAM location
by the stack pointer

$$\rightarrow (\text{SP}) \leftarrow (\text{SP}) + 1$$
$$(\text{SP}) \leftarrow \text{direct}$$

→ This instruction supports only direct addressing mode. The instructions such as 'PUSH A', 'PUSH R3' are illegal instructions.

Ex: PUSH 0E0h

where 0E0h is the RAM address belonging to

Ex: PUSH 03h

where 03h is the RAM address of R3 of Bank 0

(2) POP direct

→ POP from the stack

→ This copies the byte pointed to by stack pointer to the location where direct address is indicated, decrements SP by 1

$$\rightarrow (\text{direct}) \leftarrow ((\text{SP}))$$
$$(\text{SP}) \leftarrow (\text{SP}) - 1$$

→ This instruction supports only direct addressing. The instructions such as 'POP A' or 'POP R3' are illegal instructions.

Ex: POP 0E0h

where 0E0h is the RAM address belonging to register A

Ex: POP 03h

where 03h is the RAM address of R3 of Bank 0

SUBROUTINE INSTRUCTIONS

→ Subroutines are handled by CALL and RET instruction.

CALL INSTRUCTIONS

call instruction transfers control to a subroutine

There are two types of call

1) ACALL

2) LCALL

1) ACALL

Absolute call; Transfers control to a subroutine

A call is a 2 byte instruction

In A call, the target address is within 2k bytes of the current program counter [PC]

If a subroutine is called, the PC [which has the address of the instruction after the ACALL] is pushed onto the stack & the stack pointer [SP] is incremented by 2.

Then the program counter [PC] is loaded with the new address & control is transferred to the subroutine

At the end of the procedure [subroutine], when RET instruction is executed, PC is popped off the stack which returns control to the instruction after the call.

ACALL unconditionally calls a subroutine located at the indicated address. The instruction increments the PC twice to obtain the address of the following instruction, then pushes 16 bit result onto the stack & increments the

- stack pointer to store high order byte
- ACALL is a 2 byte instruction in which
are used for the opode & remaining 11 bits
used for the target subroutine address
 - A 11 bit address limits the range to 2¹¹ = 2k
 - $(PC) \leftarrow (PC) + 2$; address where LCALL resides, say $854E_{16}$
 $PC = 854E_{16} + 2h = 854E_{16} + 2h = 854F_{16}$
 - $(SP) \leftarrow (SP) + 1$; If SP has default value 00, then $SP = 01$
 - $(SP) \leftarrow (PC_{7-0})$; since byte of PC content 4B is ignored.
 - $(SP) \leftarrow (SP) + 1$; SP increments again. $[SP] = 02$
 - $(SP) \leftarrow (PC_{15-8})$; higher byte of PC content 85 will be stored in memory location 03.
 - $(PC_{10-0}) \leftarrow$ page address

(2) LCALL

- Long call; Transfers control to a subroutine
- LCALL is a 3 byte instruction in which one byte
is the opode & the other two bytes are the 16 bit
address of the target subroutine
- LCALL calls a subroutine located at the indicated
address.
- The instruction adds three to the program counter
to generate the address of the next instruction
then pushes the 16 bit result onto the stack
(1st lower byte) incrementing the stack pointer by 2 &
pushes higher byte
- The subroutine may therefore begin anywhere
in the full 64 k byte program memory

$(PC) \leftarrow (PC) + 3$; address where next instruction is stored
 $(PC) \leftarrow (SP) + 1$
 $(SP) \leftarrow (SP) - 1$
 $(SP) \leftarrow (PC_{7-0})$
 $(SP) \leftarrow (SP) + 1$
 $(SP) \leftarrow (PC_{15-0})$
 $(PC) \leftarrow \text{address}$
 RETURN INSTRUCTIONS
 (0-15)
 16 bit

RET

This instruction pops top two contents from the stack and load it to PC.

This instruction is used to return from a subroutine previously entered by instruction CALL or ACALL. The two top bytes of the stack are popped in the program counter [PC] & program execution continues at the new address.

After popping the top two bytes of the stack into the program counter, the stack pointer [SP] is decremented by 2.

$[PC_{15-8}] \leftarrow ((SP))$; Content of current top of the stack will be moved to higher byte of PC
 $(SP) \leftarrow (SP) - 1$; SP decrements
 $(PC_{7-0}) \leftarrow ((SP))$; Content of bottom of the stack will be moved to lower byte of PC
 $(SP) \leftarrow (SP) - 1$; SP decrements again

) RET I

Return from interrupt
This is used at the end of an interrupt service

routine [interrupt handler].

→ The top two bytes of the stack are popped
program counter (PC) ; stack pointer (SP) is decremented by 2.

$$\rightarrow (P_{15-8}) \leftarrow ((SP))$$

$$(SP) \leftarrow (SP) - 1$$

$$(P_{7-0}) \leftarrow ((SP))$$

$$(SP) \leftarrow (SP) - 1$$

NOTE - The RET instruction is used at the end of a subroutine associated with the ACALL and LCALL instructions, RETI must be used for the interrupt service subroutine.

★ COMPARE & JUMP INSTRUCTIONS

CJNE destination-byte, source-byte, target

→ compare & jump if not equal

→ The magnitudes of the source byte & destination byte are compared. If they are not equal it jumps to the target

(1) CJNE A, direct, rel address

$$\rightarrow (PC) \leftarrow (PC) + 3$$

If $(A) \neq (\text{direct})$

Then

$$(PC) \leftarrow (PC) + \text{relative address}$$

If $(A) < (\text{direct})$

Then

$$(C) \leftarrow 1$$

CJNE A, #data, rel address

$$\rightarrow (PC) \leftarrow (PC) + 3$$

If (A) < data

Then (PC) $\leftarrow (PC) + \text{relative address}$

If (A) < data

$$\text{Then } (C) \leftarrow 1$$

$$\text{else } (C) \leftarrow 0$$

3) CJNE Rn, #data, rel

$$\rightarrow (PC) \leftarrow (PC) + 3$$

If (Rn) < data

Then (PC) $\leftarrow (PC) + \text{relative address}$

If (Rn) < data

$$\text{Then } (C) \leftarrow 1$$

$$\text{else } (C) \leftarrow 0$$

4) CJNE @ Ri, #data, rel

$$\rightarrow (PC) \leftarrow (PC) + 3$$

If ((Ri)) < data

Then (PC) $\leftarrow (PC) + \text{relative address}$

If (Ri) < data

Then

$$(C) \leftarrow 1$$

else

$$(C) \leftarrow 0$$

★ DIFFERENCE BETWEEN JUMP & CALL INSTRUCTIONS

<u>JUMP</u>	<u>CALL</u>
<ul style="list-style-type: none"> (1) These permanently changes the program flow (2) These won't store return address on stack (3) This branches to the target address (4) Conditional jump instruction are available (5) Jump ranges <ul style="list-style-type: none"> (i) Relative : -128d to +127d (ii) Absolute → within a page (2kb) (iii) long → anywhere within 64kbyte (6) Ex: SJMP, JNC, JC, JZ, JB etc 	<ul style="list-style-type: none"> (1) These temporarily changes the program flow (2) These store the return address on stack (3) This is used to call subroutine (4) Conditional call instruction are not available (5) Call ranges <ul style="list-style-type: none"> (i) Acall : within a page [2¹⁶] (ii) Lcall : anywhere within 64¹⁶ (6) Ex: Acall, Lcall

★ DIFFERENCE BETWEEN CONDITIONAL JUMP & UNCONDITIONAL

<u>CONDITIONAL</u>	<u>UNCONDITIONAL</u>
<ul style="list-style-type: none"> (1) All conditional jumps are short jumps -128d to +127d (2) Conditional jump instruction changes the program flow if certain condition exists <p>Ex: JNZ, JZ, JNC, JC, JB.</p>	<ul style="list-style-type: none"> (1) Unconditional jump may be LJMP & SJMP (2) Unconditional jump instruction changes the program flow irrespective of the condition i.e. does not depend on any condition <p>Ex: SJMP, LJMP</p>

DATA TYPES & DIRECTIVES

(30)

DATA TYPES

8051 microcontroller supports only one data type
It is 8 bits [1 Byte] & size of each register is
also 8 bits

since data will be larger than 8 bits, programmer
has to break down data & process them separately
[00 to FF_h, or 0 to 255 in decimal]

The data type used by 8051 can be positive
or negative

MSB of 8 bit data indicates the sign of the data
ie If MSB=0, no is positive

If MSB=1, no is negative

In case of negative numbers, the numbers are
represented in 2's complement form.

Ex -1 in decimal \Rightarrow 1_d = 0000 0001_b

1^s Complement = 1111 1110_b

2^s Complement = 1111 1111_b

(-1) is represented by 1111 1111₂

→ For unsigned numbers the data can be 00 to FF_h or
00 to 255 in decimal.

→ For signed numbers ; data can be (1000 0000)_b =
D(0111 111)₂ i.e., +128 in 12-bit

in decimal.

(1) DB [Define Byte]

The DB directive is the most widely used
data directive in the assembler.
It is used to define the 8 bit data.

- DB is used to define data, the numbers can be in decimal, hex & ASCII format
- For decimal, 'D' after the decimal number is optional, but using binary (B) & hexadecimals (H) for the others is required
- Assembler will converts the numbers into hex
- To indicate ASCII, place the characters in 'quotient marks'
- The DB directive is only directive that used to define ASCII strings larger than two characters.
It should be used for all the ASCII data definitions

Ex:

ORG	500h	;	decimal [1C in hex]	
Data1:	DB	28	;	Binary [35 in Hex]
Data2:	DB	00110101B		
Data3:	DB	39h		
D	ORG	510h		
DATA4	DB	2591	;	ASCII numbers
	ORG	518h	;	
DATA6	DB	"My name is "	;	ASCII characters

- Either single or double quotes can be used around ASCII strings
- This is useful for strings which contain a single quote such as "O" clearly!
- DB is also used to allocate memory in 1 or more sized chunks

ASSEMBLER DIRECTIVES

(31)

There are some instructions which are not a part of instruction set.

These instructions are instructions to the assembler linker and loader. They are referred to as pseudo operations or as assembler directives.

- The assembler directives enable us to control the way in which a program assembles & lists
- They act during the assembly of a program & do not generate any executable machine code.
- most widely used directives of 8051 are :-

(1) ORG [origin]

- The ORG directive is used to indicate the beginning of the address. The number that comes after ORG can be either in hex or in decimal. If the number is not followed by H, it is decimal & converts it to hex.
- Some assemblers use ".ORG" (notice the .)
- instead of ORG for the origin directive

(2) EQU [equate] and SET

- This is used to define a constant without a label occupying a memory location
- The EQU directive does not set aside storage for a data but associates a constant value with data label so that when the label appears

in the program, its constant i.e. the ~~expr~~
→ need to ~~variables~~ will be substituted for ~~value~~
label.

Ex: COUNT EQU 25

MOV R3, #COUNT

→ when executing the instruction "MOV R3, #COUNT"
the register R3 will be loaded with the
value 25 (sign#).

→ Advantage of using EQU → Assume that there is
a constant [fixed value] used in many
different places in the program, & the
programmer wants to change its value throughout.

By the use of EQU, the programmer can change
it once & the assembler will change all of
its occurrences, rather than search the entire
program trying to find every occurrence.

→ SET directive allows redefinition of symbols at later stage

(3) END

→ This indicates to the assembler the end of source [asm] file.

→ END directive is the last line of 8051 program
means that in the source code anything
after the END directive is ignored by
the assembler.

→ Some assemblers use ".END" instead of "END"

→ If END statement is missing the assembler will
generate an error message

CODE

(32)

It assigns a name to the specified memory location in the program memory [Range 0 to 65535]

Ex: LIST CODE 1020h ; memory location 1020h in program memory is now referred to as LIST

DATA

It assigns a name to the specified location in internal RAM of 8051 [Range 0-255]

Ex: TEMP DATA 52H ; memory location 52H is now referred to as TEMP

IDATA

It assigns a name to the memory location whose address is located in the specified register (location)

Ex: MARKS IDATA 80 ; Register whose address is in register at address 80 is named as MARKS

XDATA

It assigns name to the specified memory location in the external RAM memory [Range 0 - 65535]

Ex: RESULT XDATA 1000h ; memory location 1000h in the external RAM memory is now referred to as RESULT

USING

It is used to define which register bank [Bank 0-7] will be used in the following program

Ex: USING 1 ; Bank 1 will be used

★ 8051 I/O PROGRAMMING

- In 8051 there are a total of four ports for operations
- As seen from the pin diagram, out of 40 pins a total of 32 pins are set aside for the four ports P0, P1, P2 and P3 where each port has 8 pins
- The rest of the pins are designated as Vcc, GND, XTAL1, XTAL2, RST, EA, ALE/PROG and PSEN
- A port is a pin where data can be transferred between 8051 and an external device
- The four ports P0, P1, P2 & P3 each use 8 pins, making them 8 bit ports
- All the ports upon RESET are configured as inputs ready to be used as input ports.
- When the first 0 is written to a port, it becomes an output.
To reconfigure it as an input, 1 must be sent to the port
- To use any of these ports as an input port it must be programmed.
- Each port has a D-type flip flop for each pin & each port pin can be either used as input or output pin under software control

PORT 0

Port 0 is a 8 bit, bit addressable Input output port.

It is also used as a bidirectional bus order address & data bus for external memory.

Input part

To use each pin as an input pin, first we must write '1' [logic high] to that bit ie

(1) writing a 1 to port 0 pin, the D-flip flop is high ie $Q=1$ and $\bar{Q}=0$

(2) Since $\bar{Q}=0$ & is connected to FET's gate M₁ & M₂. thus turning OFF both FET's

(3) when M₁ & M₂ are OFF, it acts like a open circuit & there will be no connection between port 0 pin & ground, thus input signal is directed to the tri state buffer TB1

Ex: MOV A, #0FFh

MOV P0, A

Output part

To use each pin as an output pin, first we must write '0' [logic zero] to that bit ie,

(1) writing a '0' to the port 0 pin, the D-flip flop output is low ie $Q=0$ & $\bar{Q}=1$

(2) since $\bar{Q}=1$ & is connected to the FET's gate M₁ & M₂. thus turning ON both FET's

(3) When M₁ & M₂ are ON, it acts like a short circuit thus port pin is connected to the ground

Any attempt to read the input pin will get the low ground signal regardless of the status of input pin

Eg: `MOV A, #00h`
`MOV P0, A`

As Address / data bus operation

- Port 0 is also used to carry the multiplexed address / data bus during access to external memory
- When 8051 access external memory, Port 0 carries the low order address whenever the ^{Pin} _{ALE} has a rising edge signal ie 
- Port 0 lines are further used to carry the bidirectional data bus to read & write to external memory

→ Port 0 address \rightarrow 80h

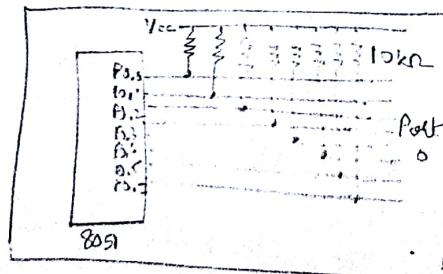
→ Port 0 bit address

P1.7	P1.6	P1.5	P1.4	P1.3	P0.2	P0.1	P0.0
87	86	85	84	83	82	81	80

NOTE - Port 0 is designated as ADD-A0-D7 to be used as both address & data

- (1) To use the pins of port 0 as both input & output each pin must be connected externally to 10kΩ resistor called as pull up resistor

(2)



PORT 0 WITH PULL UP RESISTORS

- (2) Port 0 does not need a pull up resistor when used to access external memory

PORT 1

- Port 1 occupies a total of 8 pins [pins 1 through 8]
- It can be used as input or output
- As in port 0, this port does not need any pull up resistors since it already has pull up resistors internally
- Upon reset, port 1 is configured as an input port.

As Input Port

- To use each pin as an input pin first we must write a '1' [logic high] to that bit i.e,
- Writing a '1' to port 1 pin, the D flip flop output is high i.e $Q=1 \& \bar{Q}=0$
- Since $\bar{Q}=0$ & is connected to FET gate M₁, thus turning OFF the FET
- When M₁ is off, it acts like open circuit [it blocks any path to the ground] thus input signal is directed to the tri-state buffer T_{B1}

E2: MOVA, 0FFh
MOV P1, A

As Output port

- To use each pin as an output pin, first we must write '0' [logic low] to that bit
- Writing a '0' to the port 1 pin, the D flip flop output $Q=0 \& \bar{Q}=1$
- Since $\bar{Q}=1$ & is directly connected to FET gate M₁, thus turning ON the FET

(3) When M1 is ON, it cuts like short circuit,
 port pin is connected to the ground
 ∴ Any attempt to read the input pin will
 always get the low ground signal regardless
 of the status of input pin
 Ex: `MOV A, #00h`
`MOV P1, A`

→ Port 1 address → 90h

→ Port 1 bit address

P1.7	P1.6	P1.5	P1.4	P1.3	P1.2	P1.1	P1.0
97	96	95	94	93	92	91	90

→ Port 1 pins do not have any dual function
 [only input & output function]

→ Port 1 pins have internal pull up resistors.

(III) PORT 2

→ Port 2 occupies total of 8 pins [Pins 21 through 28]

→ It can be used as input or output.

→ Just like port 1, port 2 does not need any pull up resistors since it has pull up resistors internally.

→ Upon reset port 2 is configured as an input port.

→ Port 2 pins are used to carry the higher order address [A8 - A15] bus during external memory ^{all} access.

As Input port

→ To use as input port, it must be programmed by writing 1 [logic high] to all its bits

→ Port 2 is configured first as an input port by

writing is to it.

(35)

data is received from that port & is sent to port 1 continuously.

port 2 address \rightarrow A0 h

port 2 bit address

port 2

	P2.7	P2.6	P2.5	P2.4	P2.3	P2.2	P2.1	P2.0
A7	A6	A5	A4	A3	A2	A1	A0	

I) PORT 3

- It occupies a total of 8 pins, pins 10 through 17
- It can be used as input or output
- It does not need any pull up resistors just as port 1 & port 2 did not

\Rightarrow port 3 is configured as an input port upon reset

\Rightarrow port 3 has additional function of providing important signals such as interrupts

As input port.

To use each pin as input pin, first we must write '1' [logic high] to that bit ie

write '1' to the port 3 pins, the D flip flop

(1) writing a '1' to the port 3 pins, the output is high ie $Q=1$

output is high ie $Q=1$

(2) since $Q=1$ is connected to one input of NAND gate, thus output of NAND gate is 0, & turns off the FET

(3) when FET is OFF, it acts like open circuit, thus

input signal is directed to the tri-state buffer

TBI.

Ex: `MOV A, #0FFh`
`MOV P2, A`

As output port

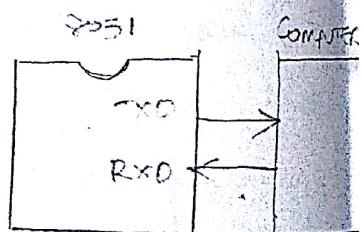
- To use each pin has an output pin, first we write '0' [logic low] to that bit i.e.
- (1) writing a '0' to the port 3 pin, the Q output $\oplus=0$, thus output of NAND gate is 1 [logic high] & turns ON the FET.
- (2) when FET is ON, it acts like short circuit, it provides the path to ground to the input pin. Any attempt to read the input pin will always get the low ground signal regardless of the state of the input pin
- Ex: MOV A, #0FFh
MOV P3, A

NOTE - For input-output operation, alternate output is always 1.

Alternate function of Port 3

Pin	Alternate Use
P3.0 - RXD	Serial data input
P3.1 - TXD	Serial data output
P3.2 - INT0	External interrupt 0
P3.3 - INT1	External interrupt 1
P3.4 - T0	External timer 0 input
P3.5 - T1	External timer 1 input
P3.6 - MWE	External memory write pulse
P3.7 - R/W	External memory read pulse

(i) RXD & TXD -



→ In 8051, RXD & TXD pins are used for serial communication.
 TXD: The data is transmitted out of 8051 through TXD pin.
 RXD: The data is received by 8051 through RXD pin.

(ii) INT0 & INT1 :

→ T...+.....L...+....T...L...

triggered by external circuits

T0 & T1

(36)

8051 has two 16 bit Timers / Counters

- T0 → Timer 0 register [16 bit]
- T1 → Timer 1 register [16 bit]

T0 & T1 can be used either as timers to generate a time delay or as counters to count events happening outside the microcontroller.

RD & WR

when $\overline{RD} = 0$, microcontroller reads the data from external RAM

when $\overline{WR} = 0$, microcontroller writes the data into external RAM.

NOTE

→ Port 3 address is $\rightarrow B_0$

→ Port 3 bit address

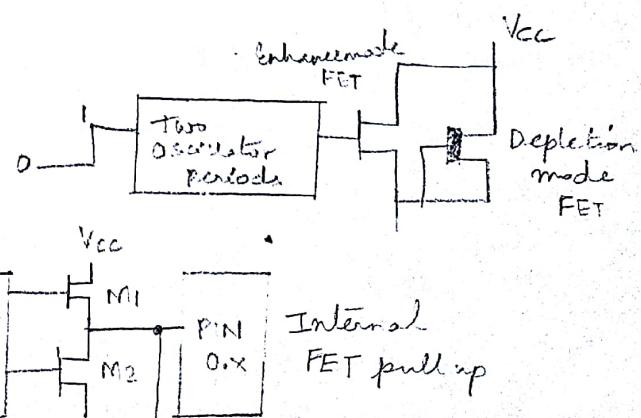
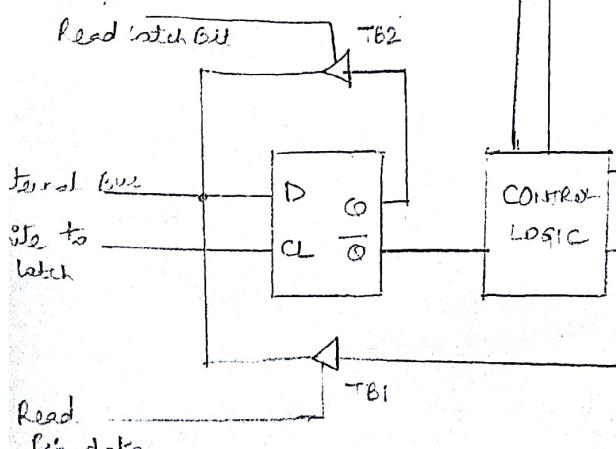
P3.7	P3.6	P3.5	P3.4	P3.3	P3.2	P3.1	P3.0
B7	B6	B5	B4	B3	B2	B1	B0

PIN CONFIGURATION OF I/O PORTS

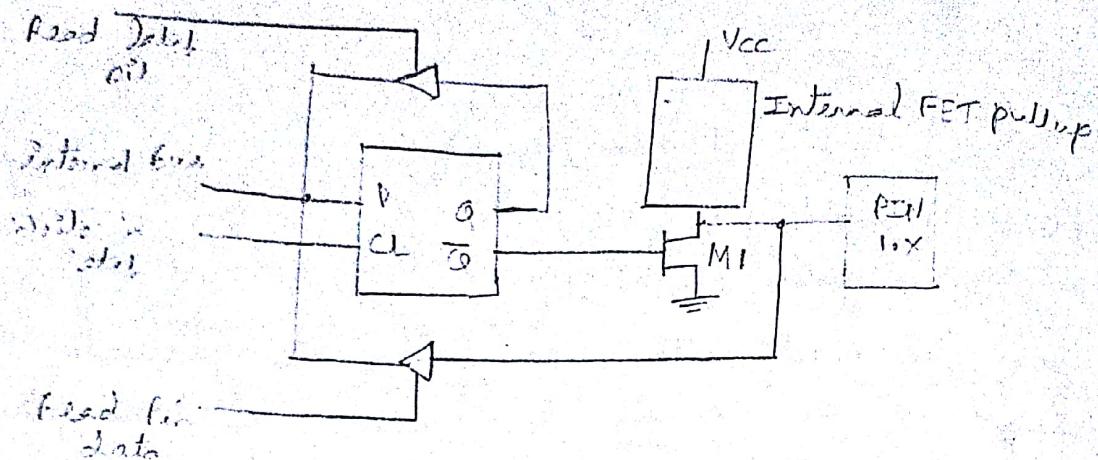
PORT0

Port 0 signals

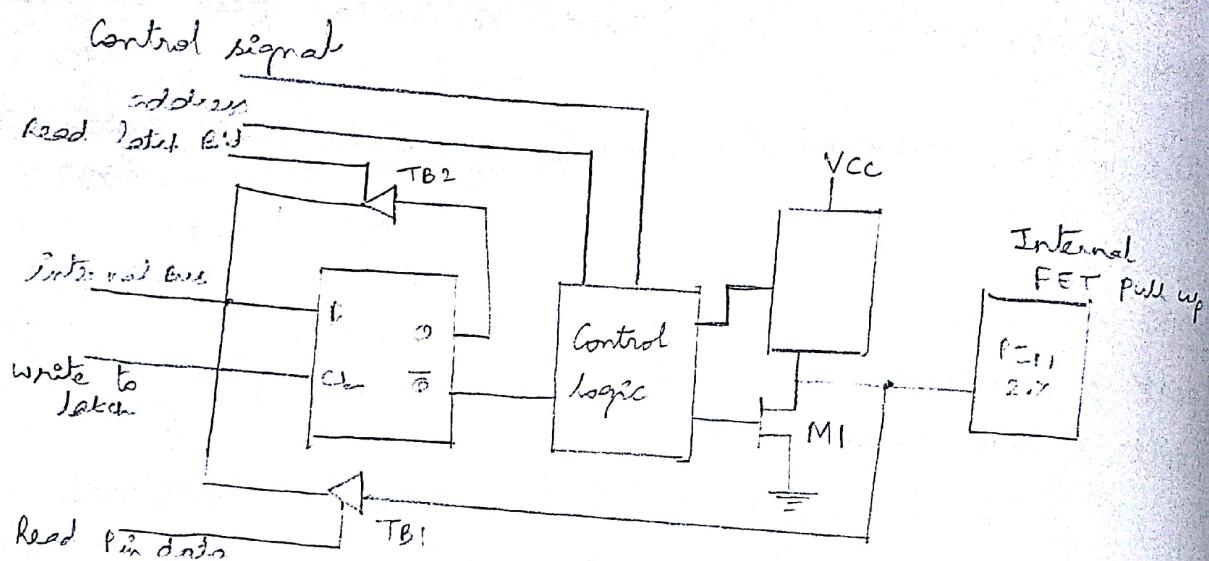
Address/data



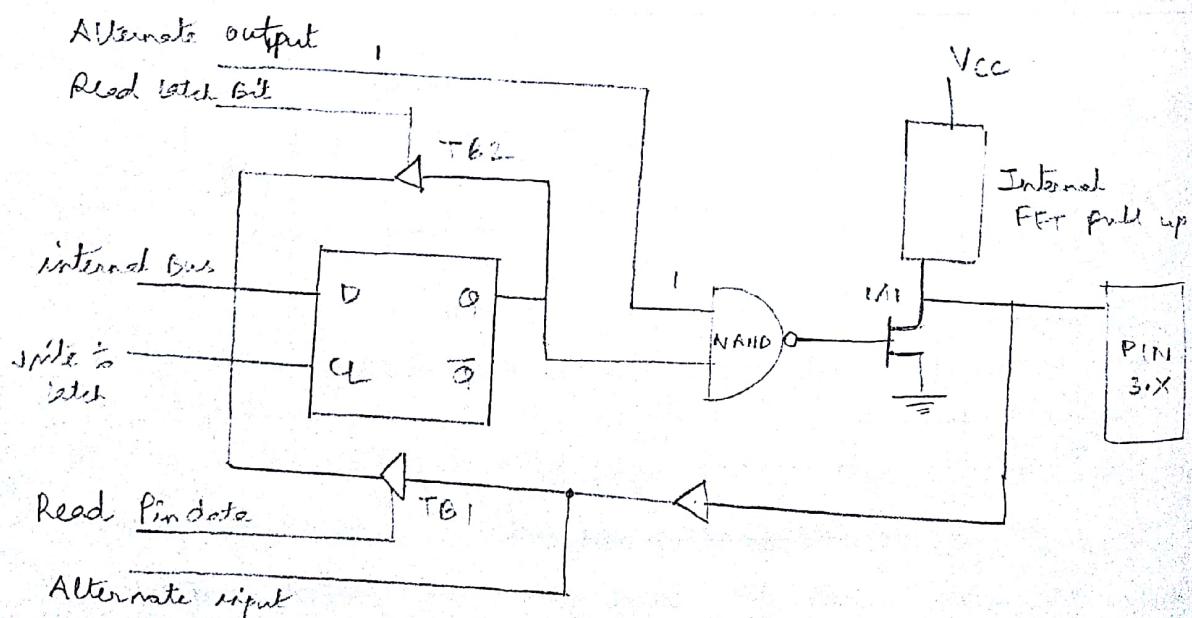
PORT 1



PORT 2



PORT 3



BIT ADDRESSING FOR I/O & RAM

Many microprocessors allow programs to access registers and I/O ports in byte size only, ie if we need to check a single bit of an I/O port, we need to read entire byte first & then manipulate the whole byte with some logic instructions to get hold of the desired single bit.

→ It is not the case with 8051, ie most important feature of 8051 is the ability to access the registers, RAM & I/O ports in bits instead of bytes.

BIT ADDRESSABLE RAM

→ Out of 128 byte internal RAM of 8051, only 16 bytes are bit addressable, rest are accessed in byte format.

→ Bit addressable RAM locations are 20H to 2FH

→ These 16 bytes provide 128 bits of RAM bit addressability, ie $16 \times 8 = 128$

→ They are addressed as 0 to 127 (decimal) or 00 to 7

∴ Bit addresses 0 to 7 → first byte of internal RAM location 20H

8 to 0Fh → bit addresses of second byte RAM location 21H

→ Internal RAM locations 20-2Fh are both byte-addressable & bit-addressable.

→ To avoid confusion regarding addresses 00-7Fh remember below-

(1) 128 bytes of RAM have byte addresses of 00-7Fh & can be accessed in byte size using various

addressing modes such as direct and register ^{indirect} addressing modes.
These 128 bytes are accessed using byte type instructions.

(2) 16 bytes of RAM location 20-2Fh also have bit addresses of 00-7Fh.

To access these 128 bits of RAM locations & other bits we can use only single bit instructions, such as

SETB.

single bit instructions use only one addressing mode & that is direct addressing mode.

BIT ADDRESSABLE I/O PORT

- 8051 has four 8 bit I/O port P0, P1, P2 and P3
- we can access entire 8 bits or any single bit without altering the rest.
- When accessing a port in single bit manner, we use SETB x.y where x → port number 0, 1, 2, or 3
y → desired bit number 0 to 7 for data bits D₀ to D₇
- D₀ is the LSB ; D₇ is the MSB.
- Every SFR register is assigned a byte address & ports P0-P3 are a part of SFR.
- SFR registers are byte addressable & also bit addressable
- Bit address for P0 are 80h to 87h
P1 are 90h to 97h - -

REGISTERS BIT ADDRESSABILITY

- Registers A, B, PSW, IP, IE, Acc, SCON and TCON are bit addressable.

... Refer Fig 2 (32)