# MODULE 2: Assembly Programming and Instruction of 8051

## Structure

2.1 Introduction to 8051 assembly programming

2.2 Assembling and running an 8051 program

2.3 Data types and Assembler directives

2.4 Arithmetic, logic instructions and programs

2.5 Jump, loop and call instructions

2.6 IO port programming

## Objectives

- To explain in detail the execution of 8051 Assembly language instructions and data types
- To explain loop, conditional and unconditional jump and call, handling and manipulation of I/O instructions.

## 2.1 Introduction to 8051 assembly programming

- In the early days of the computer, programmers coded in machine language, consisting of 0s and 1s : Tedious, slow and prone to error.
- Assembly languages, which provided mnemonics for the machine code instructions, plus other features, were developed: An Assembly language program consist of a series of lines of Assembly language instructions
- Assembly language is referred to as a low level language: It deals directly with the internal structure of the CPU.

### 2.1.1 Assembly language instruction includes

- A mnemonic (abbreviation easy to remember) *f* the commands to the CPU, telling it what those to do with those items
- optionally followed by one or two operands *f* the data items being manipulated
- A given Assembly language program is a series of statements, or lines:
  1. Assembly language instructions *f*
     Tell the CPU what to do
  2. Directives (or pseudo-instructions)
     Give directions to the assembler

General syntax for 8051 assembly language is as follows:

**LABEL: OPCODE OPERAND; COMMENT**

- **LABEL:** (THIS IS NOT NECESSARY UNLESS THAT SPECIFIC LINE HAS TO BE ADDRESSED). The label is a symbolic address for the instruction. When the program is assembled, the label will be given specific address in which that instruction is stored. Unless that specific line of instruction is needed by a branching instruction in the program, it is not necessary to label that line. Eg: BACK, HERE
- **OPCODE:** Opcode is the symbolic representation of the operation. The assembler converts the opcode to a unique **binary code** (machine language). Eg:MOV, ADD
- **OPERAND:** While opcode specifies what operation to perform, operand specifies where to perform that action. The operand field generally contains the source and destination of the data. In some cases only source or destination will be available instead of both. The operand will be either **address of the data, or data itself.**
- **COMMENT:** Always comment will begin with ; or // symbol. To improve the program quality, programmer may always use comments in the program.

## 2.2 Assembling and running an 8051 program

**The steps of Assembly language program is outlined as follows:**

1. First we use an editor to type a program, many excellent editors or word processors are available that can be used to create and/or edit the program
   - Notice that the editor must be able to produce an ASCII file
   - For many assemblers, the file names follow the usual DOS conventions, but the source file has the extension "asm" or "src", depending on which assembly you are using.
2. The "asm" source file containing the program code created in step 1 is fed to an 8051 assembler *ƒ* The assembler converts the instructions into machine code. The assembler will produce an **object file and a list file** .The extension for the object file is "obj" while the extension for the list file is "lst"
3. Assembler require a third step called **linking** *ƒ*
   - The linker program takes one or more object code files and produce an **absolute object file** with the extension "abs" *ƒ*
   - This abs file is used by 8051 trainers that have a monitor program
4. Next the "abs" file is fed into a program called "OH" (object to hex converter) which creates a file with extension "hex" that is ready to burn into ROM

- This program comes with all 8051 assemblers
- *ƒ* Recent Windows-based assemblers combine step 2 through 4 into one step

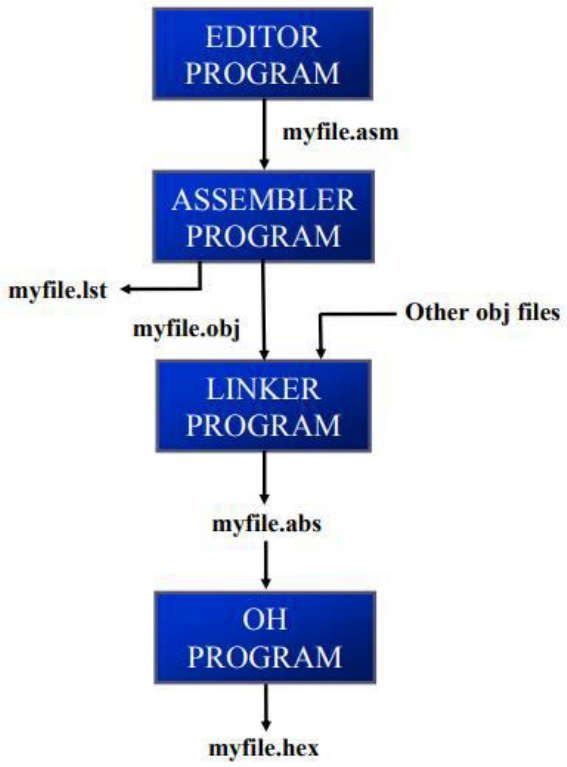



**Fig: 2.1: Steps of Assembly language program**

### 2.2.1 The lst (list) file

1. It lists all the opcodes and addresses as well as errors that the assembler detected.
2. The programmer uses the lst file to find the syntax errors or debug

## 2.3 Data types and Assembler directives

Assembler directives tell the assembler to do something other than creating the machine code for an instruction. In assembly language programming, the assembler directives instruct the assembler to

1. Process subsequent assembly language instructions
2. Define program constants
3. Reserve space for variables

**The following are the widely used 8051 assembler directives.**

1. **ORG (origin)**: The ORG directive is used to indicate the starting address. It can be used only when the program counter needs to be changed. The number that comes after ORG can be either in hex or in decimal.
   Eg: ORG 0000H;

2. **EQU and SET** : EQU and SET directives assign numerical value or register name to the specified symbol name.
   EQU is used to define a constant without storing information in the memory.
   The symbol defined with EQU should not be redefined.
   SET directive allows redefinition of symbols at a later stage.

3. **DB (DEFINE BYTE)**: The DB directive is used to define an 8 bit data. DB directive initializes memory with 8 bit values.
   The numbers can be in decimal, binary, hex or in ASCII formats.
   For decimal, the 'D' after the decimal number is optional, but for binary and hexadecimal, 'B' and ‘H’ are required.
   For ASCII, the number is written in quotation marks (‘LIKE This).
   DATA1: : DB 40H ; hex
   DATA2: DB 01011100B ; b i n a r y
   DATA3: DB 48 ; decimal
   DATA4: D B ' H E L L O ’ ; ASCII

4. **END**: The END directive signals the end of the assembly module. It indicates the end of the program to the assembler. Any text in the assembly file that appears after the END directive is ignored. If the END statement is missing, the assembler will generate an error message.

## 2.4 Arithmetic, logic instructions and programs

8051 Instructions The instructions of 8051 can be broadly classified under the following headings.

1. Data transfer instructions

2. Arithmetic instructions

3. Logical instructions

4. Branch instructions

5. Subroutine instructions

6. Bit manipulation instructions

**1. Data transfer instructions.**

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

i. MOV A,R2

ii. MOV A,R7

b. Move the contents of a register A to Rn

i. MOV R4,A

ii. MOV R1,A

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

i. MOV A, #45H

ii. MOV R6, #51H

iii. MOV 30H, #44H

iv. MOV @R0, #0E8H

v. MOV DPTR, #0F5A2H

vi. MOV DPTR, #5467H

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

i. MOV A, 65H
ii. MOV A, @R0
iii. MOV 45H, A
iv. MOV @R1, A

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

i. MOV R3, 65H

ii. MOV 45H, R2

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

i. MOV 47H, 65H

ii. MOV 45H, @R0

g. Move the contents of an external memory to A or A to an external memory

i. MOVX A,@R1

ii. MOVX @R0,A

iii. MOVX A,@DPTR

iv. MOVX@DPTR,A

h. Move the contents of program memory to A

i. MOVC A, @A+PC

ii. MOVC A, @A+DPTR

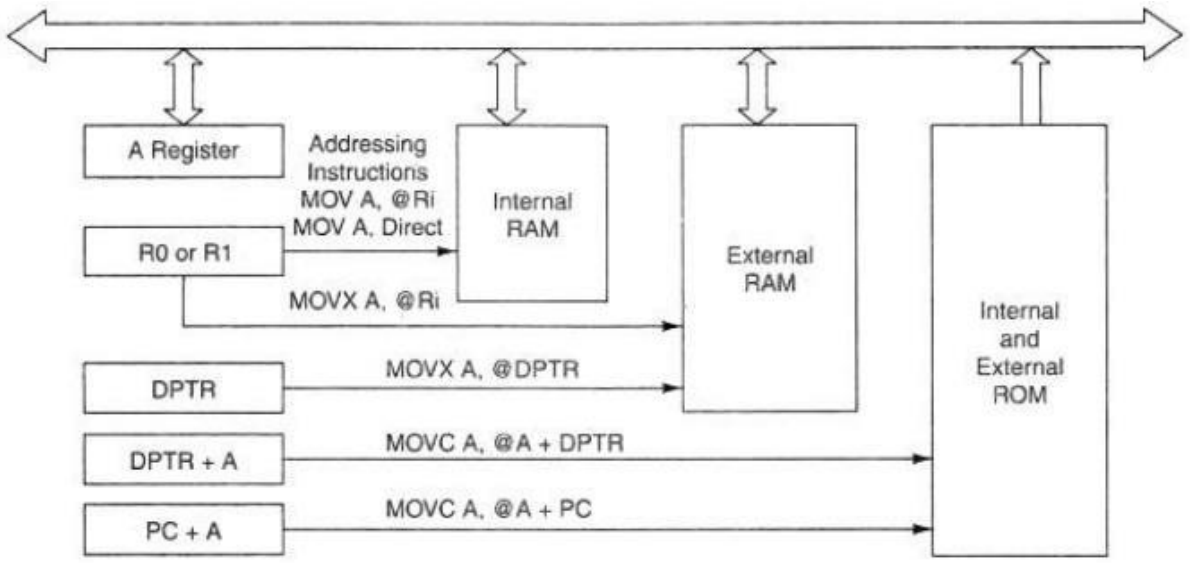



**Fig:2.2: Addressing using MOV,MOVX,MOVC**

### 2.4.1 Arithmetic instructions

The 8051 can perform addition, subtraction. Multiplication and division operations on 8 bit numbers:

**a) Addition:**

In this group, we have instructions to

i. Add the contents of A with immediate data with or without carry.

i. ADD A, #45H

ii. ADDC A, #OB4H

ii. Add the contents of A with register Rn with or without carry.

i. ADD A, R5

ii. ADDC A, R2

iii. Add the contents of A with contents of memory with or without carry using direct and indirect addressing

i. ADD A, 51H
ii. ADDC A, 75H
iii. ADD A, @R1
iv. ADDC A, @R0

**Example:** The Accumulator holds 0C3H (11000011B) and register 0 holds 0AAH (10101010B) with the carry flag set. The following instruction, ADDC A,R0 leaves **6EH (01101110B)** in the Accumulator with AC cleared and both the Carry flag and OV set to 1.

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

## Unsigned Addition

Unsigned numbers make use of the carry flag to detect when the result of an ADD operation is a number larger than FFh. If the carry is set to one after an ADD, then the carry can be added to a higher order byte so that the sum is not lost. For instance,

$$\begin{array}{rl} 95d = & 01011111b \\ \underline{189d} = & \underline{10111101b} \\ 284d \quad 1 & 00011100b = 284d \end{array}$$

The C flag is set to 1 to account for the carry out from the sum. The program could add the carry flag to another byte that forms the second byte of a larger number.

## Signed Addition

Signed numbers may be added two ways: addition of like signed numbers and addition of unlike signed numbers. If unlike signed numbers are added, then it is not possible for the result to be larger than −128d or +127d, and the sign of the result will always be correct. For example,

$$\begin{array}{rl} -001d = & 11111111b \\ \underline{+027d} = & \underline{00011011b} \\ +026d \quad & 00011010b = +026d \end{array}$$

Here, there is a carry from bit 7 so the carry flag is 1. There is also a carry from bit 6, and the OV flag is 0. For this condition, no action need be taken by the program to correct the sum.

Here, there is a carry from bit 7 so the carry flag is 1. There is also a carry from bit 6, and the OV flag is 0. For this condition, no action need be taken by the program to correct the sum.

If positive numbers are added, there is the possibility that the sum will exceed +127d, as demonstrated in the following example:

$$\begin{array}{rl} +100d = & 01100100b \\ \underline{+050d} = & \underline{00110010b} \\ +150d \quad & 10010110b = -106d \end{array}$$

Ignoring the sign of the result, the magnitude is seen to be +22d which would be correct if we had some way of accounting for the +128d, which, unfortunately, is larger than a single byte can hold. There is no carry from bit 7 and the carry flag is 0; there is a carry from bit 6 so the OV flag is 1.

An example of adding two positive numbers that do not exceed the positive limit is:

$$\begin{array}{rl} +045d = & 00101101b \\ \underline{+075d} = & \underline{01001011b} \\ +120d \quad & 01111000b = 120d \end{array}$$

Note that there are no carries from bits 6 or 7 of the sum; the carry and OV flags are both 0.

The result of adding two negative numbers together for a sum that does not exceed the negative limit is shown in this example:

$$\begin{array}{rl} -030d = & 11100010b \\ \underline{-050d} = & \underline{11001110b} \\ -080d \quad & 10110000b = -080d \end{array}$$

**b) Subtraction:**

SUBB A, <src-byte>

Function: Subtract with borrow

**Description**: SUBB subtracts the indicated variable and the carry flag together from the Accumulator, leaving the result in the Accumulator. SUBB sets the carry (borrow) flag if a borrow is needed for bit 7 and clears C otherwise. (If C was set before executing a SUBB instruction, this indicates that a borrow was needed for the previous step in a multiple-precision subtraction, so the carry is subtracted from the Accumulator along with the source operand.) AC is set if a borrow is needed for bit 3 and cleared otherwise. OV is set if a borrow is needed into bit 6, but not into bit 7, or into bit 7, but not bit 6

**Example:** The Accumulator holds 0C9H (11001001B), register 2 holds 54H (01010100B), and the carry flag is set. The instruction, SUBB A,R2 will leave the value **74H (01110100B)** in the accumulator, with the **carry flag and AC cleared but OV set**.

In this group, we have instructions to

i. Subtract the contents of A with immediate data with or without carry.

   i. SUBB A, #45H
   ii. SUBB A, #OB4H

ii. Subtract the contents of A with register Rn with or without carry.

   i. SUBB A, R5
   ii. SUBB A, R2

iii. Subtract the contents of A with contents of memory with or without carry using direct and indirect addressing

   i. SUBB A, 51H
   ii. SUBB A, 75H
   iii. SUBB A, @R1
   iv. SUBB A, @R0

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

## Unsigned Subtraction

Because the C flag is always subtracted from A along with the source byte, it must be set to 0 if the programmer does not want the flag included in the subtraction. If a multi-byte subtraction is done, the C flag is cleared for the first byte and then included in subsequent higher byte operations.

The result will be in true form, with no borrow if the source number is smaller than A, or in 2's complement form, with a borrow if the source is larger than A. These are *not* signed numbers, as all eight bits are used for the magnitude. The range of numbers is from positive 255d (C = 0, A = FFh) to negative 255d (C = 1, A = 01h).

The following example demonstrates subtraction of larger number from a smaller number:

```
          015d = 00001111b
SUBB      100d = 01100100b
        −085d  1 10101011b = 171d
```

*The C flag is set to 1, and the OV flag is set to 0. The 2's complement of the result is 085d.*

The reverse of the example yields the following result:

```
100d = 01100100b
015d = 00001111b
085d   01010101b = 085d
```

The C flag is set to 0, and the OV flag is set to 0. The magnitude of the result is in true form.

## Signed Subtraction

As is the case for addition, two combinations of unsigned numbers are possible when subtracting: subtracting numbers of like and unlike signs. When numbers of like sign are subtracted, it is impossible for the result to exceed the positive or negative magnitude limits of +127d or −128d, so the magnitude and sign of the result do not need to be adjusted, as shown in the following example:

```
          +100d = 01100100b      (Carry flag = 0 before SUBB)
SUBB      +126d = 01111110b
        −026d  1 11100110b = −026d
```

There is a borrow into bit positions 7 and 6; the carry flag is set to 1, and the OV flag is cleared.

**c) Multiplication**

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

**Eg**. MOV A,#45H ; [A]=45H

MOV B,#0F5H ; [B]=F5H

MUL AB ; [A] x [B] = 45 x F5 = 4209 ;[A]=09H, [B]=42H

**d) Division:**

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

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

**Function: Divide**

**Description**: DIV AB divides the unsigned eight-bit integer in the Accumulator by the unsigned eight-bit integer in register B. The Accumulator receives the integer part of the quotient; register B receives the integer remainder. The carry and OV flags are cleared.

**Exception:** if B had originally contained 00H, the values returned in the Accumulator and B register are undefined and the overflow flag are set. The carry flag is cleared in any case.

**Example:** The Accumulator contains 251 (0FBH or 11111011B) and B contains 18 (12H or 00010010B). The following instruction, DIV AB leaves 13 in the Accumulator (0DH or 00001101B) and the value 17 (11H or 00010001B) in B, since 251 = (13 x 18) + 17. Carry and OV are both cleared.

**e) DA A (Decimal Adjust After Addition)**

When two BCD numbers are added, the answer is a non-BCD number. To get the result in BCD, we use DA A instruction after the addition.

DA A works as follows.

- If lower nibble is greater than 9 or auxiliary carry is 1, 6 is added to lower nibble
- If upper nibble is greater than 9 or carry is 1, 6 is added to upper nibble.

Eg 1: MOV A,#23H
MOV R1,#55H
ADD A,R1 // [A]=78
DA A // [A]=78 no changes in the accumulator after da a

Eg 2: MOV A,#53H
MOV R1,#58H
ADD A,R1 // [A]=ABh
DA A // [A]=11, C=1 , ANSWER IS 111. **Accumulator data is changed after DA A**

**Increment: Increments the operand by one.**

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

**Eg:** INC A
INC Rn
INC DIRECT
INC @Ri
INC DPTR

**Decrement: decrements the operand by one**

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

Eg: DEC A
DEC Rn
DEC DIRECT
DEC @Ri

## 2.4.2 Logical Instructions

**a) Logical AND**

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

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

**Example:** If the Accumulator holds 0C3H (1100001lB), and register 0 holds 55H (01010101B), then the following instruction, ANL A,R0 leaves **41H (01000001B)** in the Accumulator

**b) Logical OR**

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

ORL A,#DATA

ORL A, Rn

ORL A,DIRECT

ORL A,@Ri

ORL DIRECT,A

ORL DIRECT, #DATA

**Example:** If the Accumulator holds 0C3H (11000011B) and R0 holds 55H (01010101B) then the following instruction, ORL A,R0 leaves the Accumulator holding the value **0D7H (1101011lB).**

**c) Logical Ex-OR**

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

XRL A,#DATA

XRL A,Rn

XRL A,DIRECT

XRL A,@Ri

XRL DIRECT,A

XRL DIRECT, #DATA

**Example:** If the Accumulator holds 0C3H (1100001lB) and register 0 holds 0AAH (10101010B) then the instruction, XRL A,R0 leaves the Accumulator holding the value **69H (01101001B).**

**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.

CPL A

CPL C

CPL bit address

**SWAP A**

Swap the upper nibble and lower nibble of A

**Function:** Swap nibbles within the Accumulator

Description: SWAP A interchanges the low- and high-order nibbles (four-bit fields) of the Accumulator (bits 3 through 0 and bits 7 through 4). The operation can also be thought of as a 4- bit rotate instruction. No flags are affected. Example: The Accumulator holds the value 0C5H (11000101B). The instruction, SWAP A leaves the Accumulator holding the value 5CH (01011100B)

**XCH A,<byte>**

**Function:** Exchange Accumulator with byte variable Description: XCH loads the Accumulator with the contents of the indicated variable, at the same time writing the original Accumulator contents to the indicated variable. The source/destination operand can use register, direct, or register-indirect addressing.

**Example:** R0 contains the address 20H. The Accumulator holds the value 3FH (0011111lB). Internal RAM location 20H holds the value 75H (01110101B). The following instruction, XCH A,@R0 leaves RAM location 20H holding the values 3FH (00111111B) and 75H (01110101B) in the accumulator.

**CPL A**

Function: Complement Accumulator

Description: CPLA logically complements each bit of the Accumulator (one's complement). Bits which previously contained a 1 are changed to a 0 and vice-versa. No flags are affected.

**Example:** The Accumulator contains 5CH (01011100B). The following instruction, CPL A leaves the Accumulator set to 0A3H (10100011B).

**CPL bit**

Function: Complement bit

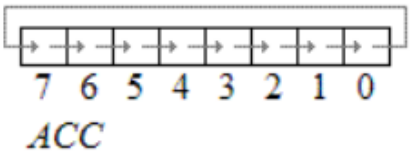
Description: CPL bit complements the bit variable specified. A bit that had been a 1 is changed to 0 and vice-versa. No other flags are affected. CLR can operate on the carry or any directly addressable bit.

**Example:** Port 1 has previously been written with 5BH (01011101B). The following instruction sequence, CPL P1.1CPL P1.2 leaves the port set to 5BH (01011011B).

**Rotate Instructions**

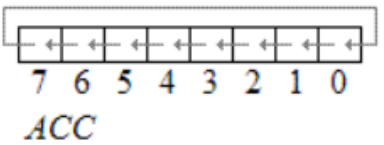
RR A

This instruction is rotate right the accumulator. Its operation is illustrated below. Each bit is shifted one location to the right, with bit 0 going to bit 7.




**RL A**

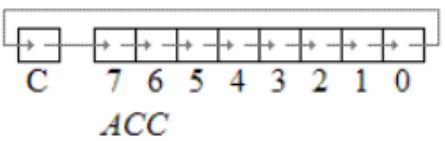
Rotate left the accumulator. Each bit is shifted one location to the left, with bit 7 going to bit 0




**Example:** The Accumulator holds the value 0C5H (11000101B). The following instruction, RL A leaves the Accumulator holding the value **8BH (10001011B)** with the carry unaffected.

**RRC A**

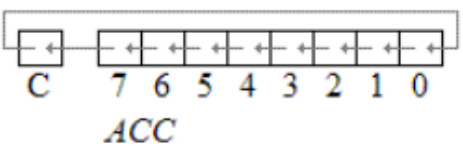
Rotate right through the carry. Each bit is shifted one location to the right, with bit 0 going into the carry bit in the PSW, while the carry was at goes into bit 7




**Example**: The Accumulator holds the value 0C5H (11000101B), the carry is zero. The following instruction, RRC A leaves the Accumulator holding the value **62 (01100010B)** with the carry set.

**RLC A**

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




**Example:** The Accumulator holds the value 0C5H(11000101B), and the carry is zero. The following instruction, RLC A leaves the Accumulator holding the value **8BH (10001010B)** with the carry set.

## 2.5 Branch (JUMP) Instructions

Jump and Call Program Range There are 3 types of jump instructions.

They are:-

1. Relative Jump
2. Short Absolute Jump
3. Long Absolute Jump

**1. Relative Jump**

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

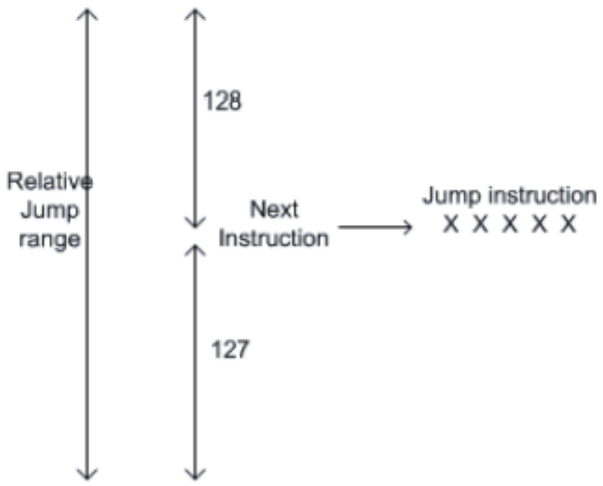



**Fig 2.3: Relative Jump Range**

**The advantages of the relative jump are as follows:-**

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

**Disadvantages of the absolute jump: -**

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

Instructions that use Relative Jump

SJMP ; this is unconditional jump

**Operation:** SJMP Function: Short Jump Syntax: SJMP reladdr

**Description:** SJMP jumps unconditionally to the address specified reladdr. Reladdr must be within -128 or +127 bytes of the instruction that follows the SJMP instruction

The remaining relative jumps are conditional jumps

JC <relative address>
JNC <relative address>
JB bit, <relative address>
JNB bit, <relative address>
JBC bit, <relative address>
CJNE <destination byte>, <source byte>, <relative address>
DJNZ <byte>, <relative address>
JZ <relative address>
JNZ <relative address>

**2. Short Absolute Jump**

- In this case only 11bits of the absolute jump address are needed. The absolute jump address is calculated in the following manner.
- In 8051, 64 kbyte of program memory space is divided into **32 pages of 2 kbyte each**.

The hexadecimal addresses of the pages are given as follows:-

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

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

**Advantage:**

The instruction length becomes 2 bytes.

**Example** of short absolute jump: -

ACALL <address 11>
AJMP <address 11>

**3. Long Absolute Jump/Call**

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

**Example: -**

```
LCALL <address 16>
LJMP  <address 16>
JMP @A+DPTR
```

**Operation**: LJMP

**Function**: Long Jump

**Syntax**: LJMP code address.

**Description:** LJMP jumps unconditionally to the specified code address.

**Another classification of jump instructions is**

1. Unconditional Jump
2. Conditional Jump

**1**. **The unconditional jump** is a jump in which control is transferred unconditionally to the target location.

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

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

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

**2. Conditional Jump instructions.**

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

**All conditional jumps are short jumps.**

**Operation: JNC**

Function: Jump if Carry Not Set

Syntax: JNC reladdr

Description: JNC branches to the address indicated by reladdr if the carry bit is not set. If the carry bit is set program execution continues with the instruction following the JNB instruction.

**Operation: JC**

Function: Jump if Carry Set

Syntax: JC reladdr

Description: JC will branch to the address indicated by reladdr if the Carry Bit is set. If the Carry Bit is not set program execution continues with the instruction following the JC instruction.

**Operation: JNB**

Function: Jump if Bit Not Set

Syntax: JNB bit addr, reladdr

Description: JNB will branch to the address indicated by reladdress if the indicated bit is not set. If the bit is set program execution continues with the instruction following the JNB instruction.

**Operation: JB**

Function: Jump if Bit Set

Syntax: JB bit addr, reladdr

Description: JB branches to the address indicated by reladdr if the bit indicated by bit addr is set. If the bit is not set program execution continues with the instruction following the JB instruction.

**Operation: JNZ**

Function: Jump if Accumulator Not Zero

Syntax: JNZ reladdr

Description: JNZ will branch to the address indicated by reladdr if the Accumulator contains any value except 0. If the value of the Accumulator is zero program execution continues with the instruction following the JNZ instruction.

**Operation: JZ**

Function: Jump if Accumulator Zero

Syntax: JNZ reladdr

Description: JZ branches to the address indicated by reladdr if the Accumulator contains the value 0. If the value of the Accumulator is non-zero program execution continues with the instruction following the JNZ instruction.

**Operation: DJNZ**

Function: Decrement and Jump if Not Zero

Syntax: DJNZ register, reladdr

Description: DJNZ decrements the value of register by 1. If the initial value of register is 0, decrementing the value will cause it to reset to 255 (0xFF Hex). If the new value of register is not 0 the program will branch to the address indicated by relative addr. If the new value of register is 0 program flow continues with the instruction following the DJNZ instruction.

**Operation: CJNE**

Function: Compare and Jump If Not Equal

Syntax: CJNE operand1,operand2,reladdr

Description: CJNE compares the value of operand1 and operand2 and branches to the indicated relative address if operand1 and operand2 are not equal. If the two operands are equal program flow continues with the instruction following the CJNE instruction. The Carry bit (C) is set if operand1 is less than operand2, otherwise it is cleared.

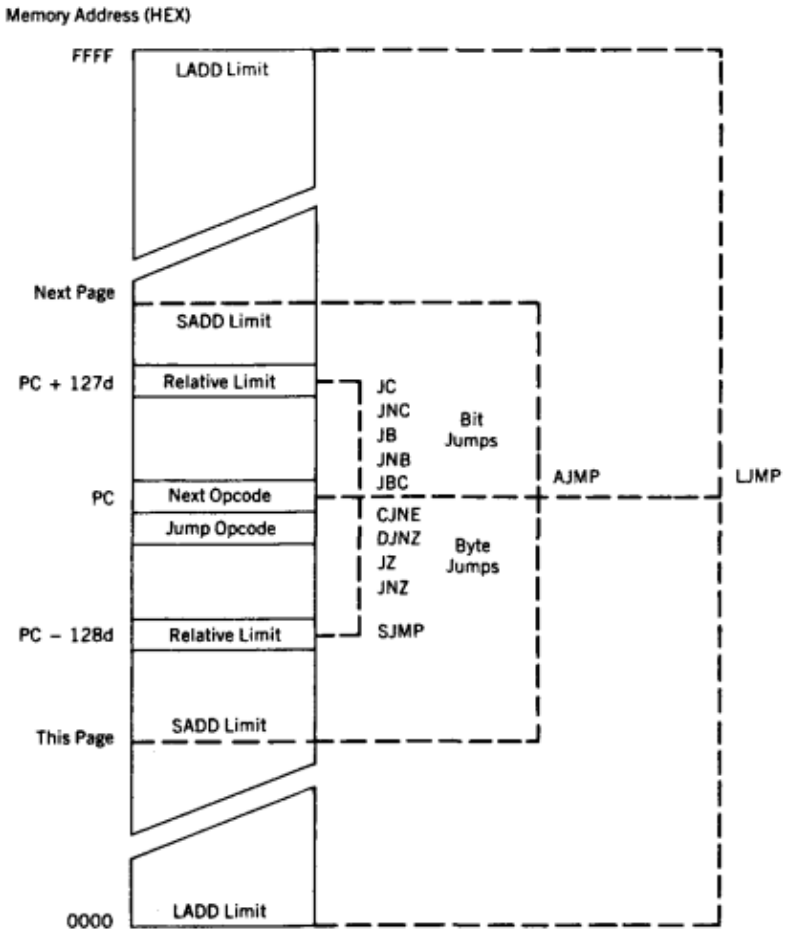



**Fig: 2.4: Jump Instruction Ranges**

## 2.5. 1 Bit level jump instructions

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

## Bit Jumps

*Bit jumps* all operate according to the status of the carry flag in the PSW or the status of any bit-addressable location. All bit jumps are relative to the program counter.

Jump instructions that test for bit conditions are shown in the following table:

| Mnemonic | Operation |
|---|---|
| JC radd | Jump relative if the carry flag is set to 1 |
| JNC radd | Jump relative if the carry flag is reset to 0 |
| JB b,radd | Jump relative if addressable bit is set to 1 |
| JNB b,radd | Jump relative if addressable bit is reset to 0 |
| JBC b,radd | Jump relative if addressable bit is set, and clear the addressable bit to 0 |

*Note that no flags are affected unless the bit in JBC is a flag bit in the PSW.* When the bit used in a JBC instruction is a port bit, the SFR latch for that port is read, tested, and altered.

## Byte Jumps

*Byte jumps*—jump instructions that test bytes of data—behave as bit jumps. If the condition that is tested is *true*, the jump is taken; if the condition is *false*,the instruction after the jump is executed. All byte jumps are relative to the program counter.

The following table lists examples of byte jumps:

| Mnemonic | Operation |
|---|---|
| CJNE A,add,radd | Compare the contents of the A register with the contents of the direct address; if they are *not* equal, then jump to the relative address; set the carry flag to 1 if A is less than the contents of the direct address; otherwise, set the carry flag to 0 |
| CJNE A,#n,radd | Compare the contents of the A register with the immediate number n; if they are *not* equal, then jump to the relative address; set the carry flag to 1 if A is less than the number; otherwise, set the carry flag to 0 |
| CJNE Rn,#n,radd | Compare the contents of register Rn with the immediate number n; if they are *not* equal, then jump to the relative address; set the carry flag to 1 if Rn is less than the number; otherwise, set the carry flag to 0 |
| CJNE @Rp,#n,radd | Compare the contents of the address contained in register Rp to the number n; if they are *not* equal, then jump to the relative address; set the carry flag to 1 if the contents of the address in Rp are less than the number; otherwise, set the carry flag to 0 |

### 2.5.2 Subroutine CALL and RETURN Instructions

Subroutines are handled by CALL and RET instructions There are two types of CALL instructions Call instructions may be included explicitly in the program as mnemonics or implicitly included using hardware interrupts.

Subroutine: Subroutine is a standalone program or small program in a main program

"**A Subroutine** is a program that may be used many times in the execution of a larger program.The subroutine could be written into the body of the main program everywhere it is needed resulting in the fastest possible code execution."

**1. LCALL address (16 bit)**

- This is long call instruction which unconditionally calls the subroutine located at the indicated 16 bit address.
- This is a 3 byte instruction.
- The LCALL instruction works as follows.

a. During execution of LCALL, [PC] = [PC]+3; (if address where LCALL resides is say, 0x3254; during execution of this instruction [PC] = 3254h + 3h = 3257h
b. [SP]=[SP]+1; (if SP contains default value 07, then SP increments and [SP]=08
c. [[SP]] = [$PC_{7-0}$]; (lower byte of PC content ie., 57 will be stored in memory location 08.
d. [SP]=[SP]+1; (SP increments again and [SP]=09)
e. [[SP]] = [$PC_{15-8}$]; (higher byte of PC content ie., 32 will be stored in memory location 09.

With these the address (0x3254) which was in PC is stored in stack.

f. [PC]= address (16 bit); the new address of subroutine is loaded to PC. No flags are affected.

**2. ACALL address (11 bit)**

This is absolute call instruction which unconditionally calls the subroutine located at the indicated 11 bit address. This is a 2 byte instruction. The SCALL instruction works as follows.

a. During execution of SCALL, [PC] = [PC]+2; (if address where LCALL resides is say, 0x8549; during execution of this instruction [PC] = 8549h + 2h = 854Bh
b. [SP]=[SP]+1; (if SP contains default value 07, then SP increments and [SP]=08
c. [[SP]] = [$PC_{7-0}$]; (lower byte of PC content ie., 4B will be stored in memory location 08.
d. [SP]=[SP]+1; (SP increments again and [SP]=09)
e. [[SP]] = [$PC_{15-8}$]; (higher byte of PC content ie., 85 will be stored in memory location 09.

With these the address (0x854B) which was in PC is stored in stack.

f. [$PC_{10-0}$]= address (11 bit); the new address of subroutine is loaded to PC. No flags are affected.

**RET instruction**

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

g. $[PC_{15-8}] = [[SP]]$ ;content of current top of the stack will be moved to higher byte of PC.
h. [SP]=[SP]-1; (SP decrements)
i. $[PC_{7-0}] = [[SP]]$ ;content of bottom of the stack will be moved to lower byte of PC.
j. [SP]=[SP]-1; (SP decrements again)

## 2.5.3 Calls and Stack

1. A call, hardware or software when initiated, causes a jump to the address where the subroutine is located.
2. At the end of the subroutine the program resumes operation at the opcode address immediately following the call.
3. Call can be located anywhere in the program space and used many times.
4. The stack area of internal RAM is used to automatically store the address, called the return address, of the instruction found immediately after the call.
5. **Stack and stack pointer** are often used to designate the top of the stack area in RAM that is pointed to by the stack pointer

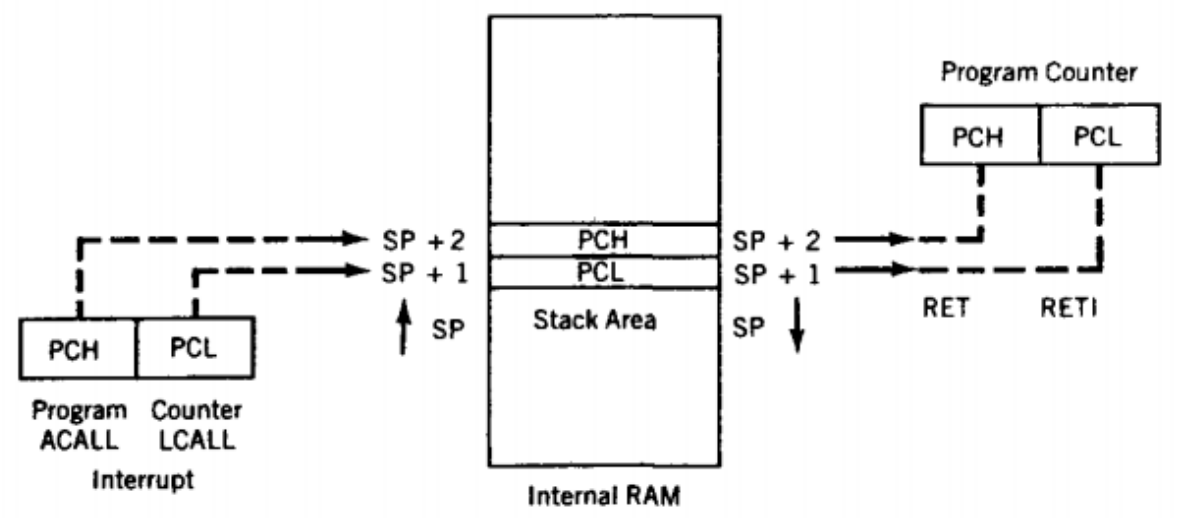



**Fig 2.5 : Storing and Retrieving Return address**

1. A call opcode occurs in the program software, or an interrupt is generated in the hardware circuitry.
2. The return address of the next instruction after the call instruction or interrupt is found in the program counter.
3. The return address bytes are pushed on the stack, *low* byte *first*.
4. The stack pointer is incremented for each push on the stack.
5. The subroutine address is placed in the program counter.
6. The subroutine is executed.
7. A RET (return) opcode is encountered at the end of the subroutine.

8. Two pop operations restore the return address to the PC from the stack area in internal RAM.
9. The stack pointer is decremented for each address byte pop.

## 2.6 I O port programming

1. I/O Port pins, Ports and Circuits: One major feature of a microcontroller is versatility built into the I/O circuits that connect the 8051 to the outside world.
2. Out of 40 pins 24 pins may each be used for one of two entirely different functions yielding a total pin configuration of 64.
3. But the port pins have been multiplexed to perform different functions to make 8051 as 40 Pin IC
   The port pin circuitry is as shown below

**Port-0**

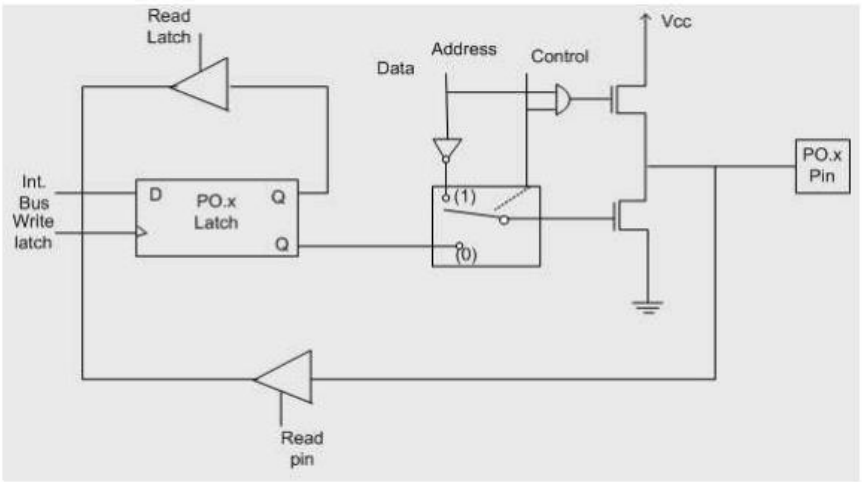



**Fig. 2. 6 : Port -0**

1. Port -0 has 8 pins (P0.0-P0.7).The structure of a Port-0 pin is shown in Fig.2.6 ..Port-0 can be configured as a normal bidirectional I/O port or it can be used for address/data interfacing for accessing external memory.
2. When control is '1', the port is used for address/data interfacing. When the control is '0', the port can be used as a normal bidirectional I/O port. Let us assume that control is '0'.
3. When the port is used as an input port, '1' is written to the latch. In this situation both the output MOSFETs are 'off'. Hence the output pin floats. This high impedance pin can be pulled up or low by an external source.
4. When the port is used as an output port, a '1' written to the latch again turns 'off' both the output MOSFETs and causes the output pin to float. An external pull-up is required to output a '1'.

5. But when '0' is written to the latch, the pin is pulled down by the lower MOSFET. Hence the output becomes zero. When the control is '1', address/data bus controls the output driver MOSFETs. If the address/data bus (internal) is '0', the upper MOSFET is 'off' and the lower MOSFET is 'on'.
6. The output becomes '0'. If the address/data bus is '1', the upper transistor is 'on' and the lower transistor is 'off'.
7. Hence the output is '1'. Hence for normal address/data interfacing (for external memory access) no pull-up resistors are required. Port-0 latch is written to with 1's when used for external memory access

**Port-1**

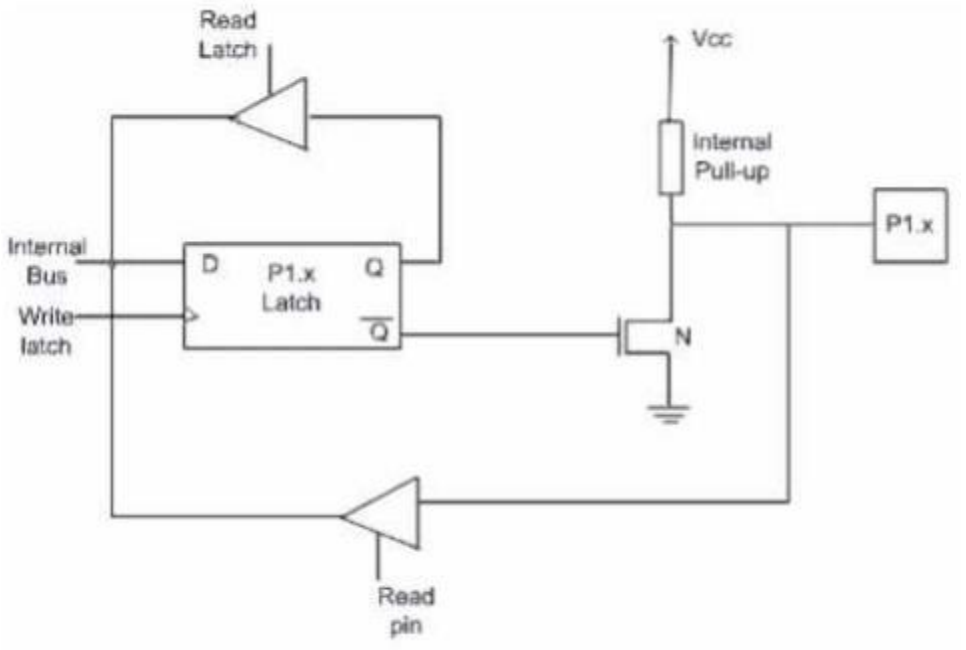



**Fig 2.7 : Port 1 Structure**

1. Port-1 has 8 pins (P1.1-P1.7) .The structure of a port-1 pin is shown in fig Fig.2.7
2. Port-1 does not have any alternate function i.e. it is dedicated solely for I/O interfacing. When used as output port, the pin is pulled up or down through internal pull-up.
3. To use port- 1 as input port, '1' has to be written to the latch. In this input mode when '1' is written to the pin by the external device then it reads fine.
4. But when '0' is written to the pin by the external device then the external source must sink current due to internal pull-up. If the external device is not able to sink the current the pin voltage may rise, leading to a possible wrong reading.

**Port-2**

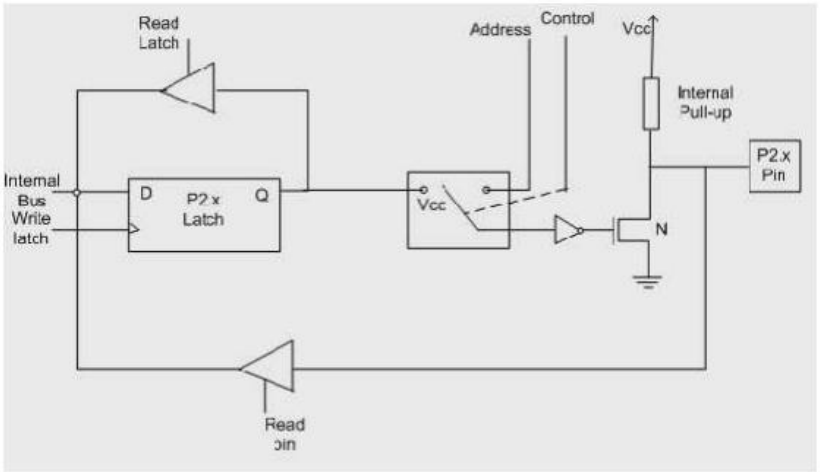



**Fig 2.8: Port 2 Structure**

1. Port-2 has 8-pins (P2.0-P2.7) . The structure of a port-2 pin is shown in Fig 2.8
2. Port-2 is used for higher external address byte or a normal input/output port. The I/O operation is similar to Port-1.
3. Port-2 latch remains stable when Port-2 pin are used for external memory access. Here again due to internal pull-up there is limited current driving capability.

**Port-3**

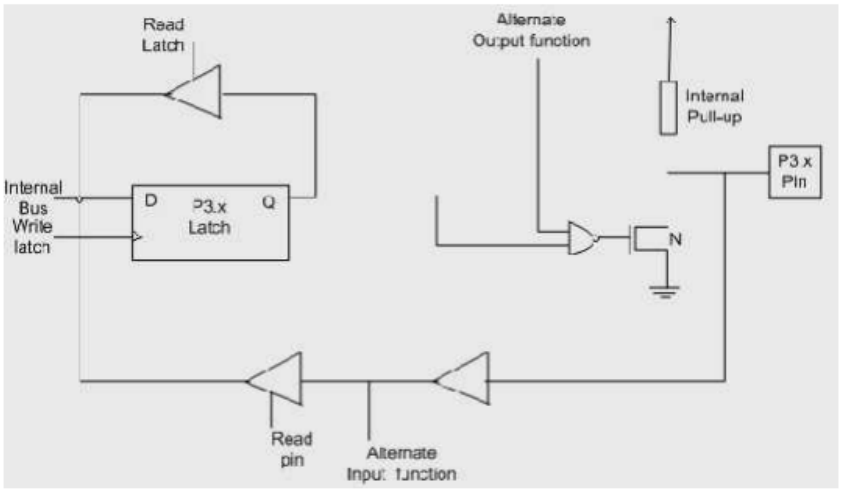



**Fig 2.9: Port 3 Structure**

1. Each pin of Port-3 can be individually programmed for I/O operation or for alternate function. The alternate function can be activated only if the corresponding latch has been written to '1'.
2. To use the port as input port, '1' should be written to the latch. This port also has internal pull-up and limited current driving capability.
3. Alternate functions of Port-3 pins –

| | |
|---|---|
| P3.0 | RxD |
| P3.1 | TxD |
| P3.2 | INT0 |
| P3.3 | INT1 |
| P3.4 | T0 |
| P3.5 | T1 |
| P3.6 | WR |
| P3.7 | RD |

Note:
1. Port 1, 2, 3 each can drive 4 LS TTL inputs.
2. Port-0 can drive 8 LS TTL inputs in address /data mode. For digital output port, it needs external pull-up resistors.
3. Ports-1,2and 3 pins can also be driven by open-collector or open-drain outputs.

- Each Port 3 bit can be configured either as a normal I/O or as a special function bit. Reading a port (port-pins) versus reading a latch.
- There is a subtle difference between reading a latch and reading the output port pin. The status of the output port pin is sometimes dependant on the connected load.
- For instance if a port is configured as an output port and a '1' is written to the latch, the output pin should also show '1'.
- If the output is used to drive the base of a transistor, the transistor turns 'on'. If the port pin is read, the value will be '0' which is corresponding to the base-emitter voltage of the transistor.
- Reading a latch: Usually the instructions that read the latch, read a value, possibly change it, and then rewrite it to the latch. These are called "read-modify-write" instructions.

### 2.6.1 Programs

1. Write a program to add the values of locations 50H and 51H and store the result in locations in 52h and 53H.

```
ORG 0000H ; Set program counter 0000H
MOV A,50H ; Load the contents of Memory location 50H into A
ADD ADD A,51H ; Add the contents of memory 51H with CONTENTS A
MOV 52H,A ; Save the LS byte of the result in 52H
MOV A, #00 ; Load 00H into A
ADDC A, #00 ; Add the immediate data and carry to A
MOV 53H,A ; Save the MS byte of the result in location 53h
END
```

2. Write a program to subtract a 16 bit number stored at locations 51H-52H from 55H-56H and store the result in locations 40H and 41H. Assume that the least significant byte of data or the result is stored in low address. If the result is positive, then store 00H, else store 01H in 42H.

```
ORG 0000H ; Set program counter 0000H
MOV A, 55H ; Load the contents of memory location 55 into A
CLR C ; Clear the borrow flag
SUBB A,51H ; Sub the contents of memory 51H from contents of A
MOV 40H, A ; Save the LSByte of the result in location 40H
MOV A, 56H ; Load the contents of memory location 56H into A
SUBB A, 52H ; Subtract the content of memory 52H from the content A
MOV 41H, A; Save the MSbyte of the result in location 41.
MOV A, #00 ; Load 005 into A
ADDC A, #00 ; Add the immediate data and the carry flag to A
MOV 42H, A ; If result is positive, store00H, else store 0lH in 42H
END
```

3. Write a program to add two 16 bit numbers stored at locations 51H-52H and 55H-56H and store the result in locations 40H, 41H and 42H. Assume that the least significant byte of data and the result is stored in low address and the most significant byte of data or the result is stored in high address.

```
ORG 0000H ; Set program counter 0000H
MOV A,51H ; Load the contents of memory location 51H into A
ADD A,55H ; Add the contents of 55H with contents of A
MOV 40H,A ; Save the LS byte of the result in location 40H
MOV A,52H ; Load the contents of 52H into A
ADDC A,56H ; Add the contents of 56H and CY flag with A
MOV 41H,A ; Save the second byte of the result in 41H
MOV A,#00 ; Load 00H into A
ADDC A,#00 ; Add the immediate data 00H and CY to A
MOV 42H,A ; Save the MS byte of the result in location 42H
END
```

4. Write a program to add two Binary Coded Decimal (BCD) numbers stored at locations 60H and 61H and store the result in BCD at memory locations 52H and 53H. Assume that the least significant byte of the result is stored in low address.

```
ORG 0000H ; Set program counter 0000H
 MOV A,60H ; Load the contents of memory location 6.0.H into A
ADD A,61H ; Add the contents of memory location 61H with contents of A
DA A ; Decimal adjustment of the sum in A
MOV 52H, A ; Save the least significant byte of the result in location 52H
MOV A,#00 ; Load 00H into .A
ADDC A,#00H ; Add the immediate data and the contents of carry flag to A
MOV 53H,A ; Save the most significant byte of the result in location 53:,
END
```

5. Write a program to clear 10 RAM locations starting at RAM address 1000H.

```
ORG 0000H ;Set program counter 0000H
MOV DPTR, #1000H ;Copy address 1000H to DPTR CLR A ;
Clear A MOV R6, #0AH ;Load 0AH to R6 again:
MOVX @DPTR,A ;Clear RAM location pointed by DPTR
INC DPTR ;Increment DPTR
DJNZ R6, again ;Loop until counter R6=0
END
```

6. Write a program to compute 1 + 2 + 3 + N (say N=15) and save the sum at70H

```
ORG 0000H ; Set program counter 0000H N EQU 15 MOV R0,#00 ;
Clear R0
CLR A ; Clear A
again: INC R0 ; Increment R0
ADD A, R0 ; Add the contents of R0 with A
CJNE R0,#N,again ; Loop until counter, R0, N
MOV 70H,A ; Save the result in location 70H
END
```

7. Write a program to multiply two 8 bit numbers stored at locations 70H and 71H and store the result at memory locations 52H and 53H. Assume that the least significant byte of the result is stored in low address.

```
ORG 0000H ; Set program counter 0000H
MOV A, 70H ; Load the contents of memory location 70h into A
MOV B, 71H ; Load the contents of memory location 71H into B
MUL AB ; Perform multiplication
MOV 52H,A ; Save the least significant byte of the result in location 52H
MOV 53H,B ; Save the most significant byte of the result in location 53
END
```

**Outcomes**

**At the end of the Module, the students will be able to:**

- **CO1: Interpret** the architectural features of 8051 microcontroller and its peripherals, Memory organization, memory interfacing and looping instructions. **(L4) MODULE 1, 2**
- **CO2: Develop** 8051 programs in assembly language to solve arithmetic and logical programs. **(L3) MODULE 1, 2**