

## Module – 2

### 8051 Instruction Set

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.

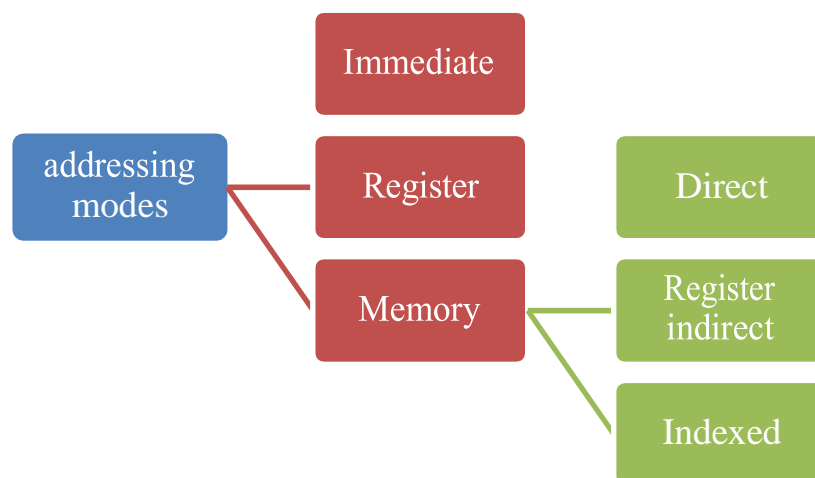
**OPCODE:** Opcode is the symbolic representation of the operation. The assembler converts the opcode to a unique binary code (machine language).

**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.

### Addressing Modes

The CPU can **access data in various ways**, which are called addressing modes



#### Immediate Addressing Mode

- The source operand is a constant
- The immediate data must be preceded by the pound sign, “#”
- Can load information into any registers, including 16-bit DPTR register
  - DPTR can also be accessed as two 8-bit registers, the high byte DPH and low byte DPL

```

MOV A, #25H           ;load 25H into A
MOV R4, #62           ;load 62 into R4
MOV B, #40H           ;load 40H into B
MOV DPTR, #4521H      ;DPTR=4512H
MOV DPL, #21H         ;This is the same
MOV DPH, #45H         ;as above

; Illegal!! Value > 65535 (FFFFH)
MOV DPTR, #68975

```

We can use EQU directive to access immediate data

```

Count EQU 30
...
MOV R4, #COUNT      ;R4=1EH
MOV DPTR, #MYDATA     ;DPTR=200H

ORG 200H
MYDATA: DB "America"

```

We can use EQU directive to access immediate data

```
MOV P1, #55H
```

### Register Addressing Mode

- Use registers to hold the data to be manipulated

```

MOV A, R0      ;copy contents of R0 into A
MOV R2, A      ;copy contents of A into R2
ADD A, R5      ;add contents of R5 to A
ADD A, R7      ;add contents of R7 to A
MOV R6, A      ;save accumulator in R6

```

- The source and destination registers must match in size

**MOV DPTR, A will give an error**

```

MOV DPTR, #25F5H
MOV R7, DPL
MOV R6, DPH

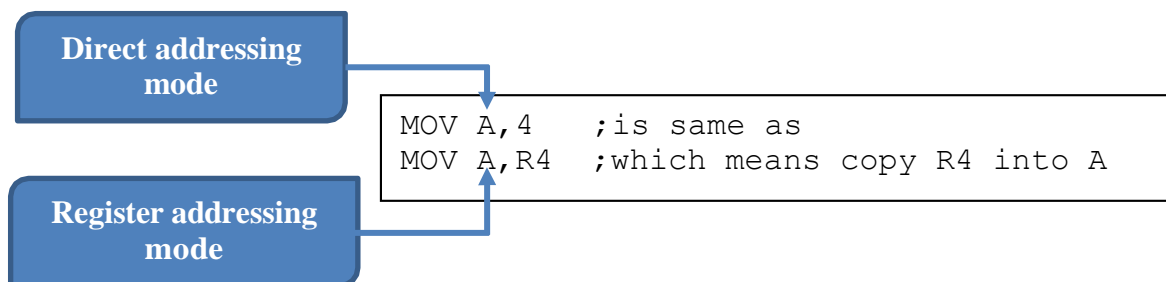
```

- The movement of data between Rn registers is not allowed

**MOV R4, R7 is invalid**

### Accessing Memory- Direct Addressing Mode

- It is most often used, the direct addressing mode, to access RAM locations 30H – 7FH
  - The entire 128 bytes of RAM can be accessed
  - The register bank locations are accessed by the register names



- In contrast to immediate addressing mode
  - There is no “#” sign in the operand

```
MOV R0, 40H      ; save/move content of 40H in R0
MOV 56H, A       ; save/move content of A in 56H
```

- The SFR (Special Function Register) can be accessed by their names or by their addresses

```
MOV 0E0H, #55H  ; is the same as
MOV A, #55h     ; load 55h into A
MOV 0F0H, R0    ; is the same as
MOV B, R0       ; copy R0 into B
```

- The SFR registers have addresses between 80H and FFH
  - Not all the address space of 80 to FF is used by SFR
  - The unused locations 80H to FFH are reserved and must not be used by the 8051 programmer.

#### Example 2-1

Write code to send 55H to ports P1 and P2, using (a) their names (b) their addresses

#### Solution :

```
(a)  MOV A, #55H      ; A=55H
      MOV P1, A        ; P1=55H
      MOV P2, A        ; P2=55H
```

#### (b) P1 address=90H; P2 address=A0H

```
MOV A, #55H      ; A=55H
MOV 90H, A       ; P1=55H
MOV 0A0H, A      ; P2=55H
```

### Accessing Memory- Stack and Direct Addressing Mode

- Only direct addressing mode is allowed for pushing or popping the stack
  - **PUSH A is invalid**
  - Pushing the accumulator onto the stack must be coded as **PUSH 0E0H**

#### Example 2-2

Show the code to push R5 and A onto the stack and then pop them back them into R2 and B, where B = A and R2 = R5

#### Solution:

```
PUSH 05    ;push R5 onto stack
PUSH 0E0H  ;push register A onto stack
POP 0F0H   ;pop top of stack into B
           ;now register B = register A
POP 02     ;pop top of stack into R2
           ;now R2=R5
```

### Accessing Memory- Register-Indirect Addressing Mode

- A register is used as a pointer to the data
  - **Only register R0 and R1 are used** for this purpose
  - **R2 – R7 cannot be used** to hold the address of an operand located in RAM
- When R0 and R1 hold the addresses of RAM locations, they **must be preceded by the “@”** sign

MOV A, @R0	; move <b>contents of</b> RAM whose
	; Address is held by <b>R0 into A</b>
MOV @R1, B	;move <b>contents of B into</b> RAM
	; whose address is held by <b>R1</b>

#### Example 2-3

Write a program to copy the value 55H into RAM memory locations 40H to 41H using (a) direct addressing mode, (b) register indirect addressing mode without a loop, and (c) with a loop

#### Solution:

```
(a)
    MOV A, #55H        ;load A with value 55H
    MOV 40H, A         ;copy A to RAM location 40H
    MOV 41H, A         ;copy A to RAM location 41H

(b)
    MOV A, #55H        ;load A with value 55H
    MOV R0, #40H       ;load the pointer. R0=40H
    MOV @R0, A         ;copy A to RAM R0 points to
    INC R0             ;increment pointer. Now R0=41h
    MOV @R0, A         ;copy A to RAM R0 points to

(c)
    MOV A, #55H        ;A=55H
    MOV R0, #40H       ;load pointer. R0=40H,
    MOV R2, #02        ;load counter, R2=3
AGAIN: MOV @R0, A      ;copy 55 to RAM R0 points to
    INC R0             ;increment R0 pointer
    DJNZ R2, AGAIN     ;loop until counter = zero
```

- The **advantage** is that it **makes accessing data dynamic** rather than **static as in direct addressing mode**
  - **Looping is not possible in direct addressing mode**

**Example 2-4**

Write a program to clear 16 RAM locations starting at RAM address 60H

**Solution:**

```

                CLR A                ;A=0
                MOV R1,#60H          ;load pointer, R1=60H
                MOV R7,#16           ;load counter, R7=16
AGAIN:         MOV @R1,A            ;clear RAM R1 points to
                INC R1               ;increment R1 pointer
                DJNZ R7,AGAIN        ;loop until counter=zero

```

**Example 2-5**

Write a program to copy a block of 10 bytes of data from 35H to 60H

**Solution:**

```

                MOV R0,#35H          ;source pointer
                MOV R1,#60H          ;destination pointer
                MOV R3,#10           ;counter
BACK:          MOV A,@R0            ;get a byte from source
                MOV @R1,A           ;copy it to destination
                INC R0               ;increment source pointer
                INC R1               ;increment destination pointer
                DJNZ R3,BACK         ;keep doing for ten bytes

```

- **R0 and R1** are the **only registers that can be used for pointers** in register indirect addressing mode
- Since **R0 and R1 are 8 bits wide**, their use is **limited** to access any information in the **internal RAM**
- Whether accessing **externally connected RAM** or **on-chip ROM**, we need **16-bit pointer**
  - In such case, the **DPTR** register is used.
- Indexed addressing mode is widely used in **accessing data elements of look-up table** entries located in the program ROM.
- The look-up table allows access to elements of a frequently used table with minimum operations
- The instruction used for this purpose is **MOVC A,@A+DPTR**
  - Use instruction **MOVC**, “**C**” means **code**
  - The contents of A are added to the 16-bit register DPTR to form the 16-bit address of the needed data

**Example 2-6**

Write a program to get the x value from P1 and send  $x^2$  to P2, continuously

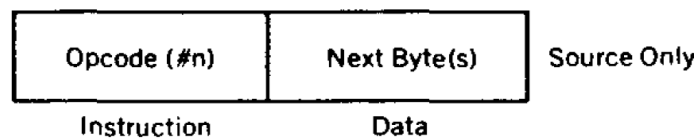
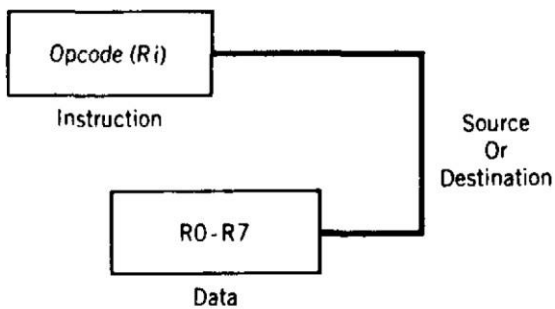
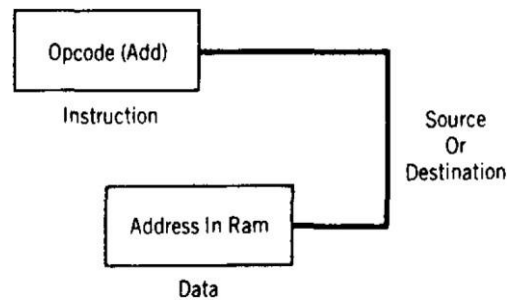
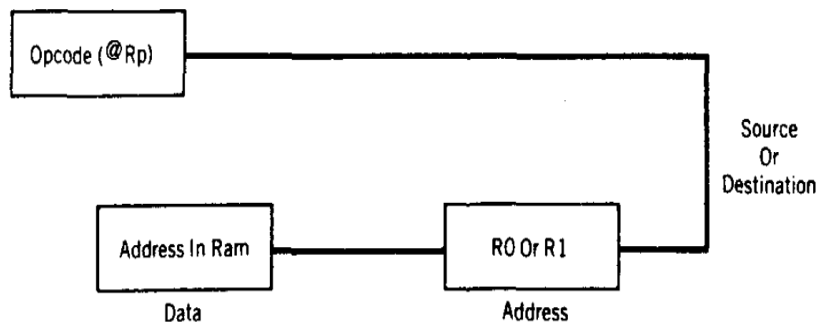
**Solution:**

```

ORG 0
    MOV DPTR, #300H ; LOAD TABLE ADDRESS
    MOV A, #0FFH    ; A=FF
    MOV P1, A       ; CONFIGURE P1 INPUT PORT
BACK: MOV A, P1      ; GET X
    MOV A, @A+DPTR  ; GET X SQUARE FROM TABLE
    MOV P2, A       ; ISSUE IT TO P2
    SJMP BACK       ; KEEP DOING IT

    ORG 300H
XSQR_TABLE:
    DB 0, 1, 4, 9, 16, 25, 36, 49, 64, 81
    END

```

**Immediate Addressing Mode****Register Addressing Mode****Direct Addressing Mode****Indirect Addressing Mode**

## Data Transfer instructions

- The MOV opcodes involve data transfers within the following four distinct physical parts of 8051 memory :
  - **Internal RAM**
  - **Internal special. function registers**
  - **External RAM**
  - **Internal and external ROM**
- Finally, the following five types of opcodes are used to move data:
  - **MOV**
  - **MOVB**
  - **MOVC**
  - **PUSH and POP**
  - **XCH**

### Data Transfer using immediate and register addressing modes

Mnemonic	Operation
MOV A, #n	Copy the immediate data byte n to the A register
MOV A, Rr	Copy data from register Rr to register A
MOV Rr, A	Copy data from register A to register Rr
MOV Rr, #n	Copy the immediate data byte n to register Rr
MOV DPTR, #nn	Copy the immediate 16-bit number nn to the DPTR register.

- A data MOV **does not alter the contents** of the data source address.
- A **copy** of the data is made from the source and moved to the destination address.
- The contents of the destination address are **replaced** by the source address contents.

Mnemonic	Operation
MOV A, #0F1h	<b>Move the immediate data byte F1h to the A register</b>
MOV A, R0	<b>Copy the data in register R0 to register A</b>
MOV DPTR, #0ABCDh	<b>Move the immediate data bytes ABCDh to the DPTR</b>
MOV R5, A	<b>Copy the data in register A to register R5</b>
MOV R3, #1Ch	<b>Move the immediate data byte 1Ch to register R3</b>

- It is **impossible** to have immediate data as a destination.
- All numbers **must** start with a decimal number (0-9), or the assembler assumes the number is a **label**.
- Register-to-register moves using the register addressing mode occur between registers **A and R0 to R7**.

Data Transfer using direct, immediate and register addressing modes

Mnemonic	Operation
MOV A, addr	Copy data from direct address add to register A
MOV add, A	Copy data from register A to direct address addr
MOV Rr, addr	Copy data from direct address add to register Rr
MOV addr, Rr	Copy data from register Rr to direct address addr
MOV addr, #n	Copy immediate data byte n to direct address addr

- MOV instructions that refer to direct addresses above 7Fh that are not SFRs will result in errors.
- The SFRs are physically on the chip; all other addresses above 7Fh do not physically exist.
- Moving data to a port changes the port *latch*; moving *data* from a port gets data from the port *pins*.
- Moving data from a direct address to itself is not predictable and could lead to errors.

Mnemonic	Operation
MOV A, 80h	Copy data from the port 0 pins to register A
MOV 80h, A	Copy data from register A to the port 0 latch
MOV 3Ah, #3Ah	Copy immediate data byte 3Ah to RAM location 3Ah
MOV R0, 12h	Copy data from RAM location 12h to register R0
MOV 8Ch, R7	Copy data from register R 7 to timer 0 high byte
MOV 5Ch, A	Copy data from register A to RAM location 5Ch
MOV 0A8h, 77h	Copy data from RAM location 77h to IE register

Data Transfer using register indirect addressing modes

- The indirect addressing mode uses a register to **hold** the actual address that will finally be used in the data move;
- The **register itself is not the address**, but rather the number in the register.
- Indirect addressing for MOV opcodes **uses register R0 or R1**, often called "*data pointers*," to hold the address of one of the data locations, which could be a RAM or an SFR address.
- The mnemonic symbol used for indirect addressing is the "*at*" sign, which is printed as @.



Mnemonic	Operation
MOV @Rp, #n	Copy the immediate byte n to the address in Rp
MOV @Rp, addr	Copy the contents of add to the address in Rp
MOV @Rp, A	Copy the data in A to the address in Rp
MOV addr, @Rp	Copy the contents of the address in Rp to addr
MOV A, @Rp	Copy the contents of the address in Rp to A

- The number in register Rp must be a RAM or an SFR address.
- Only registers R0 or R1 may be used for indirect addressing

Mnemonic	Operation
MOV A, @R0	Copy the contents of the address in R0 to the A register
MOV @R1, #35h	Copy the number 35h to the address in R1
MOV addr, @R0	Copy the contents of the address in R0 to addr
MOV @R1, A	Copy the contents of A to the address in R1
MOV @R0, 80h	Copy the contents of the port 0 pins to the address in R0

### Data Transfer - External Data Moves

- Registers R0, R1, and the DPTR can be used to *hold the address of the data byte in external RAM*.
- R0 and R1 are limited to external RAM address ranges of 00h to 0FFh, while the DPTR register can address the RAM space of 0000h to 0FFFFh.
- An *X* is added to the MOV mnemonics to serve as a reminder that the data move is *external* to the 8051

Mnemonic	Operation
MOVX A, @Rp	Copy the contents of the external address in Rp to A
MOVX A, @DPTR	Copy the contents of the external address in DPTR to A
MOVX @Rp, A	Copy data from A to the external address in Rp
MOVX @DPTR, A	Copy data from A to the external address in DPTR

- All external data moves must involve the A register.
- Rp can address 256 bytes; DPTR can address 64K bytes.
- MOVX is normally used with external RAM or I/O addresses.

Mnemonic	Operation
MOVX @DPTR, A	Copy data from A to the 16-bit address in DPTR
MOVX @R0, A	Copy data from A to the 8-bit address in R0
MOVXA, @R1	Copy data from the 8-bit address in R1 to A
MOVX A, @DPTR	Copy data from the 16-bit address in DPTR to A

### Data Transfer - Code Memory Read-Only Data Moves

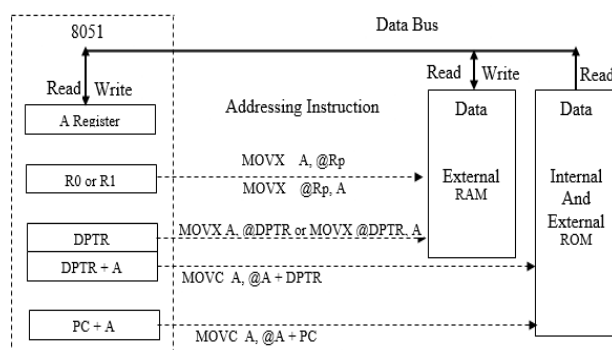
- Data moves between RAM locations and 8051 registers are made by using MOV and MOVX opcodes. The **data is usually of a temporary** or "scratch pad" nature and **disappears when the system is powered down**.
- There are times when access to a **preprogrammed mass of data is needed**, such as when using **tables of predefined bytes**. This data must be permanent to be of repeated use and is **stored in the program ROM**.
- Access to this data is made possible by using indirect addressing and the **A register in conjunction with either the PC or the DPTR**.
- In both cases, the number in register **A is added to the pointing register to form the address** in ROM where the desired data is to be found.
- The **data is then fetched from the ROM** address so formed and **placed in the A** register.
- The original data in A is **lost**, and the addressed data takes its place.

Mnemonic	Operation
MOVC A, @A+DPTR	Copy the code byte, found at the ROM address formed by adding A and the DPTR, to A
MOVC A, @A+PC	Copy the code byte, found at the ROM address formed by adding A and the PC, to A

Mnemonic	Operation
MOV DPTR, #1234h	Copy the immediate number 1234h to the DPTR
MOV A, #56h	Copy the immediate number 56h to A
MOVC A, @A+DPTR	Copy the contents of address 128Ah to A
MOVC A, @A+PC	Copies the contents of address 4059h to A if the PC contained 4000h and A contained 58h when the opcode is executed

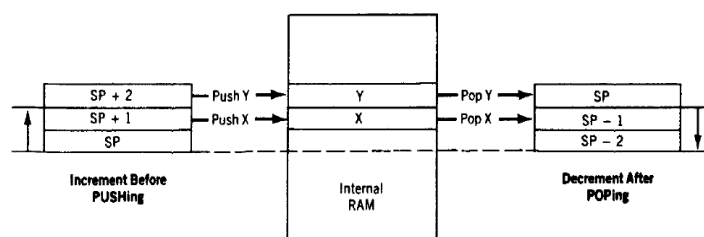
### Data Transfer - External Data Moves

- The PC is incremented by one (to point to the next instruction) *before* it is added to A to form the final address of the code byte.
- All data is moved *from* the code memory *to* the A register.
- **MOVC** is normally used with internal or external ROM and can address 4K of internal or 64K bytes of external code.



## Data Transfer - PUSH and POP Opcodes

- The PUSH and POP opcodes specify the direct address of the data.
- The data moves between an area of internal RAM, known as the stack, and the specified direct address.
- The stack pointer special-function register (SP) contains the address in RAM where data *from* the source address will be PUSHed, or where data to be POPed *to* the destination address is found.
- The SP register actually is used in the indirect addressing made but is *not* named in the mnemonic. It is *implied* that the SP holds the indirect address whenever PUSHing or POPing.
- A PUSH opcode copies data from the source address to the stack.
- SP is *incremented* by one *before* the data is copied to the internal RAM location contained in SP so that the data is stored from low addresses to high addresses in the internal RAM.
- The stack grows *up* in memory as it is PUSHed. Excessive PUSHing can make the stack exceed 7Fh (the top of internal RAM), after which point data is lost.
- A POP opcode copies data from the stack to the destination address.
- SP is *decremented* by one *after* data is copied from the stack RAM address to the direct destination to ensure that data placed on the stack is retrieved in the same order as it was stored.



Mnemonic	Operation
PUSH add	Increment SP; copy the data in add to the internal RAM address contained in SP
POP add	Copy the data from the internal RAM address contained in SP to add; decrement the SP

Mnemonic	Operation
MOV 81h, #30h	Copy the immediate data 30h to the SP
MOV R0, #0ACh	Copy the immediate data ACh to R0
PUSH 00h	SP = 31h; address 31h contains the number ACh
PUSH 00h	SP = 32h; address 32h contains the number ACh
POP 01h	SP = 31h; register R1 now contains the number ACh
POP 80h	SP = 30h; port 0 latch now contains the number ACh

- When the SP reaches FFh it "rolls over" to 00h (R0).
- RAM ends at address 7Fh; PUSHes above 7Fh result in errors.
- The SP is usually set at addresses above the register banks.
- The SP may be PUSHed and POPed to the stack.
- Note that direct addresses, *not* register names, must be used for most registers. The stack mnemonics have no way of knowing which bank is in use.

### **Data Transfer - Data Exchanges**

- MOV, PUSH, and POP opcodes all involve copying the data found in the source address to the destination address; the original data in the source is not changed.
- Exchange instructions actually move data in two directions: from source to destination and from destination to source.
- All addressing modes except immediate may be used in the XCH (exchange) opcodes

<b>Mnemonic</b>	<b>Operation</b>
XCH A,Rr	Exchange data bytes between register Rr and A
XCH A,add	Exchange data bytes between add and A
XCH A,@Rp	Exchange data bytes between A and address in Rp
XCHD A,@Rp	Exchange lower nibble between A and address in Rp

- All exchanges are *internal* to the 8051.
- All exchanges use register A.
- When using XCHD, the upper nibble of A and the upper nibble of the address location in Rp do not change.

### **Arithmetic instructions**

The 8051 can perform addition, subtraction. Multiplication and division operations on 8 bit numbers. The 24 arithmetic opcodes are grouped into as follows

<b>Mnemonic</b>	<b>Operation</b>
INC destination	<b>Increment destination by 1</b>
DEC destination	<b>Decrement destination by 1</b>
ADD/ ADDC destination ,source	<b>Add source to destination without/with carry (C) flag</b>
SUBB destination, source	<b>Subtract, with carry, source from destination</b>
MUL AB	<b>Multiply the contents of registers A and B</b>
DIV AB	<b>Divide the contents of register A by the contents of register B</b>
DA A	<b>Decimal Adjust the A register</b>

**Arithmetic instructions - Addition**

- All addition is done with the A register as the destination of the result.
- All addressing modes may be used for the source: an immediate number, a register, a direct address, and an indirect address.
- Some instructions include the carry flag as an additional source of a single bit that is included in the operation at the *least* significant bit position.

Mnemonic	Operation
ADD A, #n	Add A and the immediate number n; put the sum in A
ADD A, Rr	Add A and register Rr; put the sum in A
ADD A, addr	Add A and the address contents; put the sum in A
ADD A, @Rp	Add A and the contents of the address in Rp; put the sum in A

These instructions affect 3 bits in PSW:

C = 1 if result of add is greater than FF

AC = 1 if there is a carry out of bit 3

OV = 1 if there is a carry out of bit 7, but not from bit 6, or vice versa.

Program Status Word (PSW)								
Bit	7	6	5	4	3	2	1	0
Flag	CY	AC	F0	RS1	RS0	OV	F1	P
Name	Carry Flag	Auxiliary Carry Flag	User Flag 0	Register Bank Select 1	Register Bank Select 0	Overflow flag	User Flag 1	Parity Bit

**Addition of Unsigned Numbers**

**ADD A, source;      A = A + source**

- The instruction ADD is used to add two operands
- Destination operand is always in register A
- Source operand can be a register, immediate data, or in memory
- Memory-to-memory arithmetic operations are never allowed in 8051 Assembly language

**Example 2-7**

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

MOV A, #0F5H ; A=F5 hex

ADD A, #0BH ; A=F5+0B=00

**Solution:**

```

F5H      1111 0101
+ 0BH    + 0000 1011
100H     0000 0000

```

CY = 1, since there is a carry out from D7  
 PF = 1, because the number of 1s is zero (an even number), PF is set to 1.  
 AC = 1, since there is a carry from D3 to D4

**Example 2-8**

Assume that RAM locations 40 – 44H have the following values.

Write a program to find the sum of the values. At the end of the program, register A should contain the low byte and R7 the high byte.

40 = (7D) | 41 = (EB) | 42 = (C5) | 43 = (5B) | 44 = (30)

**Solution:**

```

MOV R0, #40H    ;load pointer
MOV R2, #5      ;load counter
CLR A           ;A=0
MOV R7, A       ;clear R7
AGAIN: ADD A, @R0 ;add the byte ptr to by R0
      JNC NEXT   ;if CY=0 don't add carry
      INC R7     ;keep track of carry
NEXT:  INC R0    ;increment pointer
      DJNZ R2, AGAIN ;repeat until R2 is zero

```

**ADDC and Addition of 16-Bit Numbers**

- When adding two 16-bit data operands, the propagation of a carry from lower byte to higher byte is concerned

$$\begin{array}{r}
 1 \\
 3C \ E7 \\
 + \ 3B \ 8D \\
 \hline
 78 \ 74
 \end{array}$$

When the first byte is added (E7+8D=74, CY=1). The carry is propagated to the higher byte, which result in 3C + 3B + 1 = 78 (all in hex)

**Example 2-9**

Write a program to add two 16-bit numbers. Place the sum in R7 and R6;

R6 should have the lower byte.

**Solution:**

```

CLR C           ;make CY=0
MOV A, #0E7H    ;load the low byte now A=E7H
ADD A, #8DH     ;add the low byte
MOV R6, A       ;save the low byte sum in R6
MOV A, #3CH     ;load the high byte
ADDC A, #3BH    ;add with the carry
MOV R7, A       ;save the high byte sum

```

**BCD Number System**

The binary representation of the digits 0 to 9 is called BCD (Binary Coded Decimal)

**Unpacked BCD**

- In unpacked BCD, the lower 4 bits of the number represent the BCD number, and the rest of the bits are 0
- Ex. 00001001 and 00000101 are unpacked BCD for 9 and 5

**Packed BCD**

- In packed BCD, a single byte has two BCD number in it, one in the lower 4 bits, and one in the upper 4 bits
- Ex. 0101 1001 is packed BCD for 59H

- Adding two BCD numbers must give a BCD result

```
MOV A, #17H
ADD A, #28H
```

Adding these two numbers gives 0011 1111B (3FH), Which is not BCD!

The result above should have been  $17 + 28 = 45$  (0100 0101).

To correct this problem, the programmer must add 6 (0110) to the low digit:  $3F + 06 = 45H$ .

### DA Instruction

- DA A ;decimal adjust for addition
- The DA instruction is provided to correct the aforementioned problem associated with BCD addition
- The DA instruction will add 6 to the lower nibble or higher nibble if need

#### Example:

```
MOV A, #47H ;A=47H first BCD operand
MOV B, #25H ;B=25H second BCD operand
ADD A, B ;hex (binary) addition (A=6CH)
DA A ;adjust for BCD addition (A=72H)
```

DA works only after an ADD, but not after INC

- The "DA" instruction works only on A.
- In other word, while the source can be an operand of any addressing mode, the destination must be in register A in order for DA to work.

### Summary of DA instruction

- After an ADD or ADDC instruction
  - If the lower nibble (4 bits) is greater than 9, or if AC=1, add 0110 to the lower 4 bits
  - If the upper nibble is greater than 9, or if CY=1, add 0110 to the upper 4 bits

#### Example:

HEX	BCD	
29	0010 1001	
+ 18	+ 0001 1000	
41	0100 0001	AC=1
+ 6	+ 0110	
47	0100 0111	

Since AC=1 after the addition, "DA A" will add 6 to the lower nibble. The final result is in BCD format.



**Example 2-10**

Assume that 5 BCD data items are stored in RAM locations starting at 40H, as shown below. Write a program to find the sum of all the numbers. The result must be in BCD.

40=(71) | 41=(11) | 42=(65) | 43=(59) | 44=(37)

**Solution:**

```

MOV R0, #40H      ;Load pointer
MOV R2, #5         ;Load counter
CLR A              ;A=0
MOV R7, A          ;Clear R7
AGAIN: ADD A, @R0   ;add the byte pointer to by R0
DA A               ;adjust for BCD
JNC NEXT           ;if CY=0 don't accumulate carry
INC R7             ;keep track of carries
NEXT: INC R0        ;increment pointer
      DJNZ R2, AGAIN ;repeat until R2 is 0

```

**Subtraction of Unsigned Numbers**

- In many microprocessor there are two different instructions for subtraction: SUB and SUBB (subtract with borrow)
- In the 8051 we have only SUBB
- The 8051 **uses adder circuitry** to perform the subtraction

**SUBB A, source ; A = A – Source – CY**

- To make SUB out of SUBB, we have to make CY=0 prior to the execution of the instruction
- Notice that we use the CY flag for the borrow

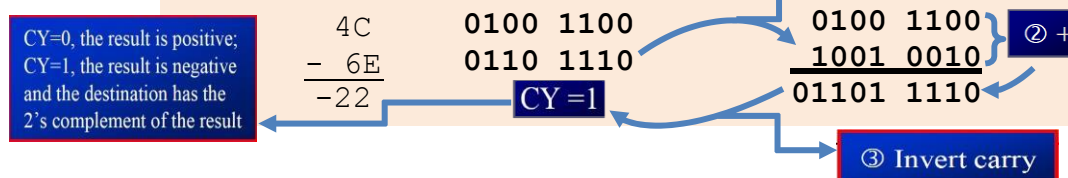
**SUBB when CY = 0**

- Take the 2's complement of the subtrahend (source operand)
- Add it to the minuend (A)
- Invert the carry

```

CLR C
MOV A, #4CH      ;load A with value 4CH
SUBB A, #6EH      ;subtract 6E from A
JNC NEXT         ;if CY=0 jump to NEXT
CPL A             ;if CY=1, take 1's complement
INC A             ;and increment to get 2's comp
NEXT: MOV R1, A   ;save A in R1

```

**Solution:**



**SUBB when CY = 1**

This instruction is used for multi-byte numbers and will take care of the borrow of the lower operand

**Code to subtract 1296h from 2762h (2762H - 1296H).**

```
CLR    C
MOV    A, #62H    ; A=62H
SUBB   A, #96H    ; 62H-96H=CCH with CY=1
MOV    R7, A      ; save the result
MOV    A, #27H    ; A=27H
SUBB   A, #12H    ; 27H-12H-1=14H
MOV    R6, A      ; save the result
```

$A = 62H - 96H - 0 = CCH$   
CY = 1

**Solution:**

We have  $2762H - 1296H = 14CCH$ .

$A = 27H - 12H - 1 = 14H$   
CY = 0

**Multiplication of Unsigned Numbers**

The 8051 supports byte by byte multiplication only

- The byte are assumed to be unsigned data

**MUL A B**; A\*B, 16-bit result in B, A

**Code to multiply 25h and 65h.**

```
MOV    A, #25H    ; load 25H to reg. A
MOV    B, #65H    ; load 65H to reg. B
MUL    AB         ; 25H * 65H = E99 where
                  ; B = 0EH and A = 99H
```

Multiplication	Operand1	Operand2	Result
Byte x byte	A	B	B = high byte A = low byte

**Division of Unsigned Numbers**

The 8051 supports byte over byte division only

- The byte are assumed to be unsigned data

**DIV A B**; A/B, Result(R, Q) in B, A

**Code to Divide 95 by 10.**

```
MOV    A, #95      ; load 95 to reg. A
MOV    B, #10      ; load 10 to reg. B
DIV    AB          ; B=05 (Rem) and A=09 (Quo)
```

Division	Numerator	Denominator	Quotient	Remainder
Byte / byte	A	B	A	B

CY is always 0  
If  $B \neq 0$ , OV = 0  
If  $B = 0$ , OV = 1 indicates error

**Example 2-11**

Write a program to get hex data in the range of 00 – FFH from port 1 and convert it to decimal. Save it in R7, R6 and R5.

**Solution:**

```
MOV A,    #0FFH
MOV P1,   A    ;make P1 an input port
MOV A,    P1   ;read data from P1
MOV B,    #10  ;B=0A hex
DIV AB    ;divide by 10
MOV R7,   B    ;save lower digit
MOV B,    #10
DIV AB    ;divide by 10 once more
MOV R6,   B    ;save the next digit
MOV R5,   A    ;save the last digit
```

**Logical instructions**

- A large part of machine control concerns sensing the on-off states of external switches, making decisions based on the switch states, and then turning external circuits on or off.
- Sensing and control implies a need for **byte and bit** opcodes that operate on data using Boolean operators.
- All 8051 RAM areas, both data and SFRs, may be manipulated using byte opcodes.
- Many of the SFRs, and a unique internal RAM area that is bit addressable, may be operated upon at the **individual** bit level.
- Bit operators are notably **efficient** when speed of response is needed.
- Bit operators yield compact program code that **enhances** program **execution speed**.

**ANL destination, source ; dest = dest AND source**

- This instruction will perform a logic AND on the two operands and place the result in the destination
  - The destination is normally the accumulator
  - The source operand can be a register, in memory, or immediate

```
MOV A, #35H ; A = 35H
ANL A, #0FH ; A = A AND 0FH
```

```
35H 00110101
0FH 00001111
0FH 00000101
```

ANL is often used to mask (set to 0) certain bits of an operand

**ORL destination, source; dest = dest OR source**

- This instruction will perform a logic OR on the two operands and place the result in the destination
  - The destination is normally the accumulator
  - The source operand can be a register, in memory, or immediate

```
MOV A, #04H ; A = 04H
ORL A, #68H ; A = A OR 68H = 6CH
```

```
04H 00000100
68H 01101000
6CH 01101100
```

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

### **XRL destination, source; dest = dest XOR source**

- This instruction will perform a logic XOR on the two operands and place the result in the destination
  - The destination is normally the accumulator
  - The source operand can be a register, in memory, or immediate

```
MOV A, #54H ; A = 04H
XRL A, #78H ; A = A XOR 78H = 2CH
```

```
54H 01010100
78H 01111000
2CH 00101100
```

XRL instruction can be used to toggle certain bits of an operand. (1's Complement)

The XRL instruction can be used to clear the contents of a register by XORing it with itself.

Show how **XRL A, A** clears A, assuming that AH = 45H.

```
45H 0100 0101
45H 0100 0101
-----
00H 0000 0000
```

### **Example 2:12**

**Read and test P1 to see whether it has the value 45H. If it does, send 99H to P2; otherwise, it stays cleared.**

Solution:

```
MOV P2, #00      ; clear P2
MOV P1, #0FFH    ; make P1 an input port
MOV R3, #45H     ; R3=45H
MOV A, P1        ; read P1
XRL A, R3
JNZ EXIT        ; jump if A is not 0
MOV P2, #99H
EXIT: ...
```

XRL can be used to see if two registers have the same value

If both registers have the same value, 00 is placed in A. JNZ and JZ test the contents of the accumulator.

**CPL A** ; complements the register A

This is called 1's complement

- Source and destination both will be the accumulator
- Content of A will 1's complemented and stored back in A itself.

```
MOV A, #56H
CPL A           ;now A=A9H
                ;0101 0110 (56H)
                ;becomes 1010 1001 (A9H)
```

- To get the 2's complement, all we have to do is to add 1 to the 1's complement

**CJNE destination, source, *Label***

- The actions of comparing and jumping are combined into a single instruction called **CJNE** (compare and jump if not equal)
- The CJNE instruction compares two operands, and jumps if they are not equal
- The destination operand can be in the accumulator or in one of the Rn registers
- The source operand can be in a register, in memory, or immediate
  - The operands themselves remain unchanged
- It changes the CY flag to indicate if the destination operand is larger or smaller.
- Notice in the CJNE instruction that any Rn register can be compared with an immediate value
- There is no need for register A to be involved
- The compare instruction is really a subtraction, except that the operands remain unchanged
- Flags are changed according to the execution of the SUBB instruction.

### Example 2 – 13

Write a program to read the temperature and test it for the value 75.

According to the test results, place the temperature value into the registers indicated by the following.

If T = 75 then A = 75

If T < 75 then R1 = T

If T > 75 then R2 = T

**Solution:**

```
MOV P1, #0FFH ;make P1 an input port
MOV A, P1      ;read P1 port
CJNE A, #75, OVER; jump if A is not 75
SJMP EXIT      ;A=75, exit
OVER:          JNC NEXT      ;if CY=0 then A>75
MOV R1, A      ;CY=1, A<75, save in R1
SJMP EXIT      ; and exit
NEXT:          MOV R2, A      ;A>75, save it in R2
EXIT: ...
```

**RR A ; rotate right A**

In rotate right

- The 8 bits of the accumulator are rotated right one bit, and
- Bit D0 exits from the LSB and enters into MSB, D7



```
MOV A, #36H      ;A = 0011 0110
RR A             ;A = 0001 1011 = 1Bh
RR A             ;A = 1000 1101 = 8Dh
RR A             ;A = 1100 0110 = C6h
RR A             ;A = 0110 0011 = 63h
```

**RL A ; rotate left A**

In rotate left

- The 8 bits of the accumulator are rotated left one bit, and
- Bit D7 exits from the MSB and enters into LSB, D0



```
MOV A, #72H      ;A = 0111 0010
RL A             ;A = 1110 0100
RL A             ;A = 1100 1001
```

**RRC A ; rotate right A, through carry**

- Bits are rotated from left to right
- They exit the LSB to the carry flag, and the carry flag enters the MSB



```
CLR C           ;make CY = 0
MOV A, #26H     ;A = 0010 0110
RRC A           ;A = 0001 0011 CY = 0
RRC A           ;A = 0000 1001 CY = 1
RRC A           ;A = 1000 0100 CY = 1
```

**RLC A ; rotate left A, through carry**

- Bits are rotated from left to right
- They exit the MSB to the carry flag, and the carry flag enters the LSB

**Example 2- 14**

Write a program that finds the number of 1s in a given byte.

```
MOV R1, #0
MOV R7, #8      ;count=08
MOV A, #97H
AGAIN: RLC A
      JNC NEXT      ;check for CY
      INC R1        ;if CY=1 add to count
NEXT: DJNZ R7, AGAIN
```

## SWAP A

It swaps the lower nibble and the higher nibble

- In other words, the lower 4 bits are put into the higher 4 bits and the higher 4 bits are put into the lower 4 bits
- SWAP works only on the accumulator (A)

### Example 2- 15

(a) Find the contents of register A in the following code.

(b) In the absence of a SWAP instruction, how would you exchange the nibbles?

Write a simple program to show the process.

#### Solution:

(a)

```
MOV A, #72H      ; A = 72H
SWAP A           ; A = 27H
```

(b)

```
MOV A, #72H      ; A = 0111 0010
RL A              ; A = 1110 0100
RL A              ; A = 1100 1001
RL A              ; A = 1001 0011
RL A              ; A = 0010 0111
```

**Example 2 - 16** Assume that register A has packed BCD, write a program to convert packed BCD to two ASCII numbers and place them in R2 and R6.

#### Solution:

```
MOV A, #29H      ; A=29H, packed BCD
MOV R2, A        ; keep a copy of BCD data
ANL A, #0FH      ; mask the upper nibble (A=09)
ORL A, #30H      ; make it an ASCII, A=39H('9')
MOV R6, A        ; save it
MOV A, R2        ; A=29H, get the original data
ANL A, #0F0H     ; mask the lower nibble
RR A             ; rotate right
RR A             ; rotate right
RR A             ; rotate right
RR A             ; rotate right
ORL A, #30H      ; A=32H, ASCII char. '2'
MOV R2, A        ; save ASCII char in R2
```

} **SWAP A**

## Branch instructions

- Repeating a sequence of instructions a certain number of times is called a loop.
- Loop action is performed by **DJNZ reg, Label**
  - The register is decremented by one
  - If it is not zero, it jumps to the target address referred to by the label
  - Prior to the start of loop the register is loaded with the counter for the number of repetitions
  - Counter can be R0 – R7 or RAM location

```

;This program adds value 3 to the ACC ten times
      MOV  A, #0          ;A=0, clear ACC
      MOV  R2, #10        ;load counter R2=10
AGAIN: ADD  A, #03        ;add 03 to ACC
      DJNZ R2, AGAIN      ;repeat until R2=0, 10 times
      MOV  R5, A          ;save A in R5

```

A loop can be repeated a maximum of 255 times, if R2 is FFH

### Branch instructions – Nested loop

If we want to repeat an action more times than 256, we use a loop inside a loop, which is called **nested loop**.

- We use multiple registers to hold the count

#### Example 2 – 17

Write a program to

- load the accumulator with the value 55H, and
- complement the ACC 700 times

```

      MOV  A, #55H        ;A=55H
      MOV  R3, #10        ;R3=10, outer loop count
NEXT: MOV  R2, #70        ;R2=70, inner loop count
      CPL  A              ;complement A register
      DJNZ R2, AGAIN      ;repeat it 70 times
      DJNZ R3, NEXT

```

### Conditional Jumps

#### JZ label ; jump if A=0

Jump only if a certain condition is met.

```

      MOV  A, R0          ;A=R0
      JZ   OVER          ;jump if A = 0
      MOV  A, R1          ;A=R1

```

OVER:

Can be used only for register A, not any other register

Determine if R5 contains the value 0. If so, put 55H in it.

```

      MOV  A, R5          ;copy R5 to A
      JNZ  NEXT          ;jump if A is not zero
      MOV  R5, #55H
NEXT: ...

```

#### JNC label ; jump if no carry, CY=0

- If CY = 0, the CPU starts to fetch and execute instruction from the address of the label
- If CY = 1, it will not jump but will execute the next instruction below JNC

**Example 2 – 18**

Find the sum of the values 79H, F5H, E2H. Put the sum in registers R0 (low byte) and R5 (high byte).

```

MOV A, #0           ;A=0
MOV R5, A           ;clear R5
ADD A, #79H         ;A=0+79H=79H
JNC N_1             ;if CY=0, add next number
INC R5              ;if CY=1, increment R5
N_1: ADD A, #0F5H    ;A=79+F5=6E and CY=1
JNC N_2             ;jump if CY=0
INC R5              ;if CY=1, increment R5 (R5=1)
N_2: ADD A, #0E2H    ;A=6E+E2=50 and CY=1
JNC OVER            ;jump if CY=0
INC R5              ;if CY=1, increment 5
OVER: MOV R0, A      ;now R0=50H, and R5=02

```

**Example 2 – 19**

Program using JNB and JBC instructions.

```

LOOP:  MOV A, #10h    ; A = 10h
        MOV R0, A     ; RO = 10h
ADDA:  ADD A, R0       ; add RO to A
        JNC ADDA      ; if the carry flag is 0, then 'no
                        ; carry' (NC) is true
                        ; jump to address ADDA
                        ; jump until A is F0h
                        ; C flag is set on the next ADD & 'no
                        ; carry' (NC) is false
                        ; do the next instruction

ADDR:  MOV A, #10h    ; A = 10h; do program again using JNB
        ADD A, R0     ; add RO to A (RO already equals 10h)
        JNB OD7h, ADDR ; D7h is the bit address of the carry
                        ; flag
        JBC OD7h, LOOP ; the carry bit is 1; the jump to LOOP
                        ; is taken, and the carry flag is
                        ; cleared to 0

```

**Unconditional Jumps**

- The unconditional jump is a jump in which control is transferred unconditionally to the target location.
- *Unconditional jumps* do not test any bit or byte to determine whether the jump should be taken.
- The jump is *always* taken.

**LJMP (long jump)**

- 3-byte instruction
  - First byte is the opcode
  - Second and third bytes represent the 16-bit target address (Any memory location from 0000 to FFFFH)



**SJMP** (short jump)

- 2-byte instruction
- First byte is the opcode
- Second byte is the relative target address
  - 00 to FFh (forward +127 and backward -128 bytes from the current PC)

**Calculating Short Jump Address**

- To calculate the target address of a short jump (SJMP, JNC, JZ, DJNZ, etc.)
  - The second byte is added to the PC of the instruction immediately below the jump
- If the target address is more than -128 to +127 bytes from the address below the short jump instruction
  - The assembler will generate an error stating the jump is out of range

Line	PC	Opcode	Mnemonic	Operand
01	0000		ORG	0000
02	0000	7800	MOV	R0, #0
03	0002	7455	MOV	A, #55H
04	0004	6003	JZ	NEXT
05	0006	08	INC	R0
06	0007	04	AGAIN: INC	A
07	0008	04		INC A
08	0009	2477	NEXT: ADD	A, #77H
09	000B	5005	JNC	OVER
10	000D	E4	CLR	A
11	000E	F8	MOV	R0, A
12	000F	F9	MOV	R1, A
13	0010	FA	MOV	R2, A
14	0011	FB	MOV	R3, A
15	0012	2B	OVER: ADD	A, R3
16	0013	50F2	JNC	AGAIN
17	0015	80FE	HERE: SJMP	HERE
18	0017		END	

**Bit manipulation instructions**

Bit manipulation 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.

Instructions that are used for signal-bit operations are

Instruction	Function
SETB bit	Set the bit (bit = 1)
CLR bit	Clear the bit (bit = 0)
CPL bit	Complement the bit (bit = NOT bit)
JB bit, target	Jump to target if bit = 1 (jump if bit)
JNB bit, target	Jump to target if bit = 0 (jump if no bit)
JBC bit, target	Jump to target if bit = 1, clear bit (jump if bit, then clear)

- The JNB and JB instructions are widely used single-bit operations
- They allow you to monitor a bit and make a decision depending on whether it's 0 or 1
  - These two instructions can be used for any bits of I/O ports 0, 1, 2, and 3

Mnemonic	Examples	Description
MOV A,PX	MOV A,P2	Bring into A the data at P2 pins
JNB PX.Y, ..	JNB P2.1,TARGET	Jump if pin P2.1 is low
JB PX.Y, ..	JB P1.3,TARGET	Jump if pin P1.3 is high
MOV C,PX.Y	MOV C,P2.4	Copy status of pin P2.4 to CY

### Simple Assembly language program examples (without loops) to use these instructions.

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 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 store data FFH into RAM memory locations 50H to 58H using direct addressing mode

```

ORG 0000H      ; Set program counter 0000H
MOV A, #0FFH   ; Load FFH into A
MOV 50H, A     ; Store contents of A in location 50H
MOV 51H, A     ; Store contents of A in location 51H
MOV 52H, A     ; Store contents of A in location 52H
MOV 53H, A     ; Store contents of A in location 53H
MOV 54H, A     ; Store contents of A in location 54H
MOV 55H, A     ; Store contents of A in location 55H
MOV 56H, A     ; Store contents of A in location 56H
MOV 57H, A     ; Store contents of A in location 57H
MOV 58H, A     ; Store contents of A in location 58H
END

```

3. 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 41H.
MOV A, #00      ; Load 00H into A
ADDC A, #00      ; Add the immediate data and the carry flag to A
MOV 42H, A      ; If result is positive, store 00H, else store 01H in 42H
END

```

4. 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

```

5. Write a program to store data FFH into RAM memory locations 50H to 58H using indirect addressing mode.

```

                ORG 0000H      ; Set program counter 0000H
                MOV A, #0FFH    ; Load FFH into A
                MOV R0, #50H    ; Load pointer, R0-50H
                MOV R5, #08H    ; Load counter, R5-08H
Start:          MOV @R0, A      ; Copy contents of A to RAM pointed by R0
                INC R0          ; Increment pointer
                DJNZ R5, start  ; Repeat until R5 is zero
                END

```

6. 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 60H 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 53H
END

```

7. 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

```

8. Write a program to compute  $1 + 2 + 3 + N$  (say  $N=15$ ) and save the sum at 70H

```

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

```

9. 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
BMUL A, B      ; 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 53H
END

```

**10. Write a program to find the average of five 8 bit numbers. Store the result in 55H. (Assume that after adding five 8 bit numbers, the result is 8 bit only).**

```

                ORG 0000H
                MOV 40H,#05H
                MOV 41H,#55H
                MOV 42H,#06H
                MOV 43H,#1AH
                MOV 44H,#09H
                MOV R0,#40H
                MOV R5,#05H
                MOV  B,R5
                CLR  A
Loop:          ADD  A,@R0
                INC  R0
                DJNZ R5,Loop
                DIV  A B
                MOV 55H,A
                END

```

**11. Write a program to find the cube of an 8 bit number program is as follows**

```

ORG 0000H
MOV R1,#N
MOV A,R1
MOV B,R1
MUL A B
MOV R2, B
MOV B, R1
MUL A B
MOV 50,A
MOV 51,B
MOV A,R2
MOV B, R1
MUL AB
ADD A, 51H
MOV 51H, A
MOV 52H, B
MOV A, #00H
ADDC A, 52H
MOV 52H, A
END

```

**12. Write a program to exchange the lower nibble of data present in external memory 6000H and 6001H**

```

ORG 0000H           ; Set program counter 00h
MOV DPTR, #6000H    ; Copy address 6000H to DPTR
MOVX A, @DPTR       ; Copy contents of 6000H to A
MOV R0, #45H        ; Load pointer, R0=45H
MOV @R0, A          ; Copy cont of A to RAM pointed by 80
INC DPL             ; Increment pointer
MOVX A, @DPTR       ; Copy contents of 6001H to A
XCHD A, @R0         ; Exchange lower nibble of A with RAM pointed by R0
MOVX @DPTR, A       ; Copy contents of A to 6001H
DEC DPL             ; Decrement pointer
MOV A, @R0          ; Copy cont of RAM pointed by R0 to A
MOVX @DPTR, A       ; Copy cont of A to RAM pointed by DPTR
END

```

**13. Write a program to count the number of 1's and 0's of 8 bit data stored in location 6000H.**

```

ORG 0000           ; Set program counter 0000H
MOV DPTR, #6000h   ; Copy address 6000H to DPTR
MOVX A, @DPTR      ; Copy number to A
MOV R0, #08        ; Copy 08 in R0
MOV R2, #00        ; Copy 00 in R2
MOV R3, #00        ; Copy 00 in R3
CLR C              ; Clear carry flag
BACK: RLC A        ; Rotate A through carry flag
      INC R2       ; If CF = 0, increment R2
      AJMP NEXT2
NEXT: INC R3       ; If CF = 1, increment R3
NEXT2: DJNZ R0, BACK ; Repeat until R0 is zero
      END

```

**14. Write a program to shift a 24 bit number stored at 57H-55H to the left logically four places. Assume that the least significant byte of data is stored in lower address.**

```

ORG 0000H           ; Set program counter 0000h
MOV R1, #04         ; Set up loop count to 4
again: MOV A, 55H    ; Place the least significant byte of data in A
      CLR C         ; Clear the carry flag
      RLC A         ; Rotate contents of A (55h) left through carry
      MOV 55H, A
      MOV A, 56H
      RLC A         ; Rotate contents of A (56H) left through carry
      MOV 56H, A
      MOV A, 57H
      RLC A         ; Rotate contents of A (57H) left through carry
      MOV 57H, A
      DJNZ R1, again ; Repeat until R1 is zero
      END

```

**15. Two 8 bit numbers are stored in location 1000h and 1001h of external data memory. Write a program to find the GCD of the numbers and store the result in 2000h.**

#### ALGORITHM

*Step 1 :Initialize external data memory with data and DPTR with address*

*Step 2 :Load A and TEMP with the operands*

*Step 3 :Are the two operands equal? If yes, go to step 9*

*Step 4 :Is (A) greater than (TEMP) ? If yes, go to step 6*

*Step 5 :Exchange (A) with (TEMP) such that A contains the bigger number*

*Step 6 :Perform division operation (contents of A with contents of TEMP)*

*Step 7 :If the remainder is zero, go to step 9*

*Step 8 :Move the remainder into A and go to step 4*

*Step 9 :Save the contents 'of TEMP in memory and terminate the program*

```

                ORG    0000H                ; Set program counter 0000H
                TEMP EQU 70H
                TEMPI EQU 71H
                MOV    DPTR, #1000H        ; Copy address 1000H to DPTR
                MOVX   A, @DPTR            ; Copy First number to A
                MOV    TEMP, A             ; Copy First number to temp INC DPTR
                MOVX   A, @DPTR            ; Copy Second number to A
LOOPS:          CJNE  A, TEMP, LOOP1        ; (A) != (TEMP) branch to LOOP1
                AJMP   LOOP2              ; (A) = (TEMP) branch to LOOP2
LOOP1:          JNC    LOOP3              ; (A) > (TEMP) branch to LOOP3
                NOV    TEMPI, A           ; (A) < (TEMP) exchange (A) with (TEMP)
                MOV    A, TEMP
                MOV    TEMP, TEMPI
LOOP3:          MOV    B, TEMP
                DIV    AB                 ; Divide (A) by (TEMP)
                MOV    A, B               ; Move remainder to A
                CJNE  A, #00, LOOPS        ; (A) != 00 branch to LOOPS
LOOP2:          MOV    A, TEMP
                MOV    DPTR, #2000H
                MOVX   @DPTR, A           ; Store the result in 2000H
                END

```

\*\*\*\*\*

