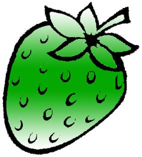
# STRAWBERRY



/strawberrydevelopers

/strawberry_app

For more visit:

Strawberrydevelopers.weebly.com

# 8051 Microcontrollers

Richa Upadhyay Prabhu

NMIMS's MPSTME

*richa.upadhyay@nmims.edu*

March 15, 2016

## 8051 INSTRUCTIONS

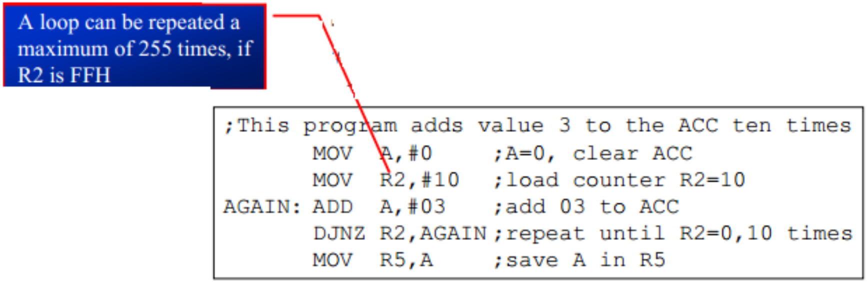
**JUMP, LOOP AND CALL INSTRUCTIONS**

## 8051 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
- 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



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

```
;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
```

## 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

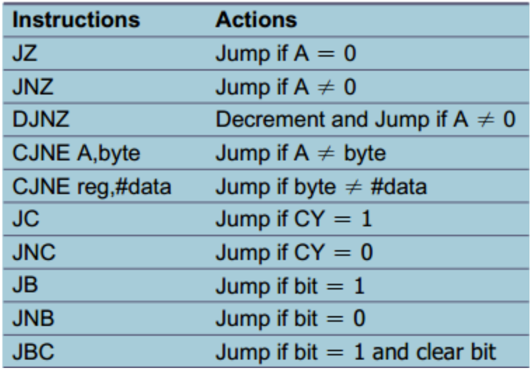
## Conditional Jumps

Jump only if a certain condition is met
**JZ label ;jump if A=0**

**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

## Conditional Jumps



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

All conditional jumps are short jumps; The address of the target must within -128 to +127 bytes of the contents of PC

## Unconditional Jumps

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

- **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)

## CALL INSTRUCTIONS

- Call instruction is used to call subroutine
- Subroutines are often used to perform tasks that need to be performed frequently. This makes a program more structured in addition to saving memory space.
- **LCALL(long call)**
  - 3-byte instruction
  - First byte is the opcode
  - Second and third bytes are used for address of target subroutine which is located anywhere within 64K byte address space
- **ACALL(absolute call)**
  - 2-byte instruction
  - 11 bits are used for address within 2K-byte range

## CALL INSTRUCTIONS

- When a subroutine is called, control is transferred to that subroutine
- the processor Saves on the stack the the address of the instruction immediately below the LCALL
- Begins to fetch instructions form the new location
- After finishing execution of the subroutine,
    - The instruction **RET** transfers control back to the caller
    - Every subroutine needs RET as the last instruction

## ADDRESSING MODES

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

- Immediate
- Register
- Direct
- Register indirect
- Indexed

## 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 is the 8051 only user-accessible 16-bit (2-byte) register
  - DPTR is used to point to data
  - When the 8051 accesses external memory it will access external memory at the address indicated by DPTR

## IMMEDIATE ADDRESSING MODE

- MOV A,#25H
- MOV R4,#62 ;
- MOV B,#40H ;
- MOV DPTR,#4521H ;
- MOV DPL,#21H ;
- MOV DPH,#45H ;
- Can also use immediate addressing mode to send data to 8051 ports
  - MOV P1,#55H

## REGISTER ADDRESSING MODE

- Use registers to hold the data to be manipulated
- The source and destination registers must match in size
- MOV A,R0
- MOV R2,A
- MOV A,4 ;is same as
- MOV A,R4 ;which means copy R4 into A
- The movement of data between Rn registers is not allowed
- MOV R4,R7 ;is invalid

## DIRECT ADDRESSING MODE

- It is most often used the direct addressing mode to access RAM locations 30 - 7FH
- The entire 128 bytes of RAM can be accessed
- The register bank locations are accessed by the register names
- Contrast this with immediate addressing mode. There is no # sign in the operand
- MOV R0,40H ;save content of 40H in R0
- MOV 56H,A ;save content of A in 56H

## REGISTER INDIRECT ADDRESSING MODE

- 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 contents of RAM whose
            ;address is held by R0 into A
MOV @R1,B   ;move contents of B into RAM
            ;whose address is held by R1
```

## REGISTER INDIRECT ADDRESSING MODE

- 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

## ADDRESSING MODE

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

## ADDRESSING MODE

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

```
(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
```

## ADDRESSING MODE

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

```
(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
```

## ADDRESSING MODE

(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
```

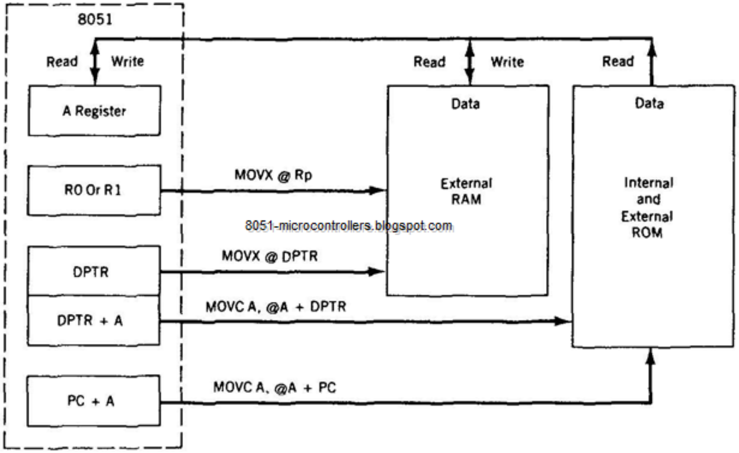
## INDEXED ADDRESSING MODE

- Indexed addressing mode is widely used in accessing data elements of look-up table entries located in the program ROM
- 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

## INDEXED ADDRESSING MODE

- In many applications, the size of program code does not leave any room to share the 64K-byte code space with data, in such cases external memory is used.
- It is accessed only by the MOVX instruction

## INDEXED ADDRESSING MODE



## ARITHMETIC & LOGIC INSTRUCTIONS

## ARITHMETIC INSTRUCTIONS : Addition of Unsigned Numbers

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

- 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

## Addition of Unsigned Numbers

PROBLEM :
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)

## Addition of Unsigned Numbers

Program:

```
        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
```

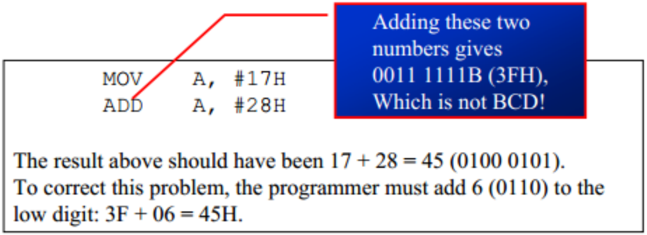
## Addition of Unsigned Numbers

- When adding two 16-bit data operands, the propagation of a carry from lower byte to higher byte is concerned
- ADDC A, source; add with carry

## Arithmetic Instructions

| UNPACKED BCD | PACKED BCD |
|---|---|
| 00001001 and 00000101 are unpacked BCD for 9 and 5 | 0101 1001 is packed BCD for 59H |

- Adding two BCD numbers must give a BCD result



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

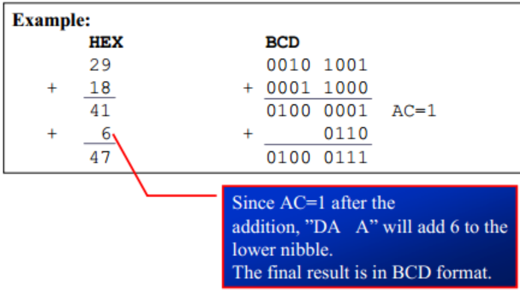
Adding these two numbers gives 0011 1111B (3FH), Which is not BCD!

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 Instructions

**DA A ;decimal adjust for addition**

- The DA instruction is provided to correct the aforementioned problem associated with BCD addition
- 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.

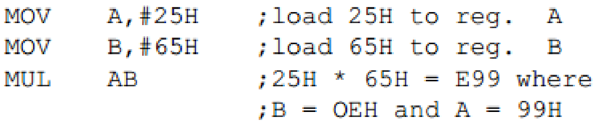
## Subtraction of Unsigned Numbers

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

- In the 8051 we have only SUBB

## Unsigned Multiplication

**MUL AB ;AxB, 16-bit result in B, A**



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

## Unsigned Multiplication

**DIV AB ;divide A by B, A/B**

```
MOV     A,#95    ;load 95 to reg.  A
MOV     B,#10    ;load 10 to reg.  B
MUL     AB       ;A = 09(quotient) and
                 ;B = 05(remainder)
```

## LOGIC AND COMPARE INSTRUCTIONS

ANL destination,source ;dest = dest AND source

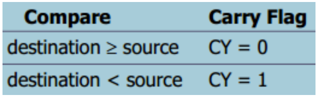
ORL destination,source; dest = dest OR source

CPL A ;complements the register A

## Compare Instruction

**CJNE destination,source,rel. addr.**

- compare and jump if 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



| Compare | Carry Flag |
|---|---|
| destination $\geq$ source | CY = 0 |
| destination $<$ source | CY = 1 |

## ROTATE INSTRUCTION

**RR A ;rotate right A**
**RL A ;rotate left A**
**RRC A ;rotate right through carry**
**RLC A ;rotate left through carry**

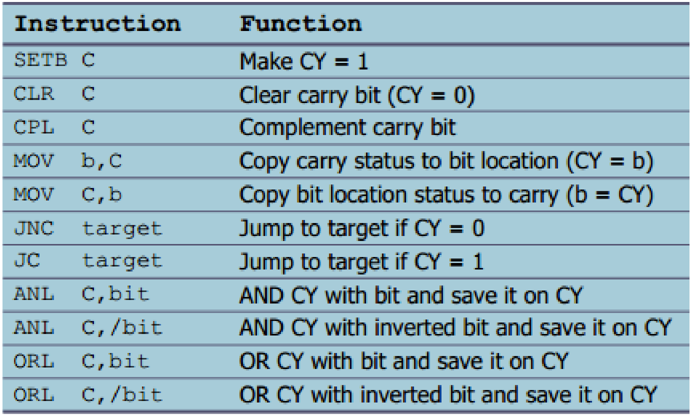
## SWAP INSTRUCTION

**SWAP A**

- It swaps the lower nibble and the higher nibble
- only on the accumulator (A)

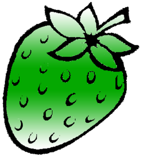
## Single-bit Operations with CY

There are several instructions by which the CY flag can be manipulated directly



| Instruction | Function |
|---|---|
| `SETB C` | Make CY = 1 |
| `CLR C` | Clear carry bit (CY = 0) |
| `CPL C` | Complement carry bit |
| `MOV b,C` | Copy carry status to bit location (CY = b) |
| `MOV C,b` | Copy bit location status to carry (b = CY) |
| `JNC target` | Jump to target if CY = 0 |
| `JC target` | Jump to target if CY = 1 |
| `ANL C,bit` | AND CY with bit and save it on CY |
| `ANL C,/bit` | AND CY with inverted bit and save it on CY |
| `ORL C,bit` | OR CY with bit and save it on CY |
| `ORL C,/bit` | OR CY with inverted bit and save it on CY |

# STRAWBERRY



/strawberrydevelopers

/strawberry_app

For more visit:

Strawberrydevelopers.weebly.com