**CHAPTER 02: INSTRUCTION SET OF 8051**

**Preview:**

In this chapter students will learn how to do assembly language programming for microcontroller 8051 using the different types of instructions. That is data transfer instructions, arithmetic instructions, Boolean type of instructions. What are different addressing modes of instructions like direct addressing mode, immediate addressing mode, register addressing mode. Also students will be able to simple programs such as addition, subtraction, multiplication, and division using assembly language and ‘C’. And how to execute using Keil, SPJ.

**2.1 Instruction Set:**

**2.1.1 Programmers Model of 8051:-**

When we consider programming model of 8051 we need to consider program memory.

* **Programe Memory:-**

In 8051 the program memory space is divided into internal and external memory. On-chip program memory is 4K. And we can connect 60K of external memory i.e total of 64K memory can be attached to 8051. This can be either on-chip or external depending on the value of the EA input pin.

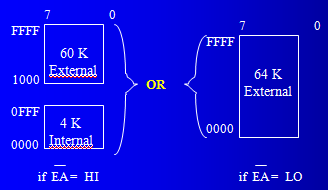


Fig 2.1: Program Memory of 8051

As shown in fig program memory can be accessed according to status of pin EA\*

Program execution Program execution from

from 0000 to 0FFF 1000 up to FFFF

If EA\* =0 (low) External program memory External program memory.

If EA\* =1 (high) Internal program memory External program memory.

The external 64K of memory can be accessed using MOVX instruction.

**2.1.2 Operand Types:**

8051 microcontroller has only one size of data type - 8 bits

* The size of each register is also 8 bits
* It is the job of the programmer to break down data larger than 8 bits

(00 to FFH, or 0 to 255 in decimal)

* The data types can be positive or negative

**2.1.3 Assembler Directives:**

* Directives are commands of assembler and they do not affect the operations of microcontroller.
* Assembler converts the assembly language program to machine language. Each instruction is converted into machine language but directive is use to facilitate or to enhance the operation and are converted into machine language.
* **DB (Define Byte)**
* The DB directive is the most widely used data directive in the assembler
* It is used to define the 8-bit data
* When DB is used to define data, the numbers can be in decimal, binary, hex, ASCII formats
* If the number is not followed by H, that is decimal and the assembler will convert it to hex
* The “D” after the decimal number is optional, but using “B” after binary and “H” after hexadecimal is required.
* ASCII code for characters is simply written in quotation marks.
* The Assembler will convert the numbers into hex
* **ORG (origin)**
* The ORG directive is used to indicate the beginning of the address
* The number that comes after ORG can be either in hex and decimal

**Example**

ORG 500H

DATA1: DB 28 ;DECIMAL (1C in Hex)

DATA2: DB 00110101B ;BINARY (35 in Hex)

DATA3: DB 39H ;HEX

ORG 510H

DATA4: DB “2591” ;ASCII NUMBERS

ORG 518H

DATA6: DB “My name is Joe” ;ASCII CHARACTERS

* **END:-**
* This indicates to the assembler the end of the source (asm) file
* The END directive is the last line of an 8051 program
* Mean that in the code anything after the END directive is ignored by the assembler
* **EQU (equate)**
* This is used to define a constant without occupying a memory location
* The EQU directive does not set aside storage for a data item but associates a constant value with a data label
* When the label appears in the program, its constant value will be substituted for the label.
* Assume that there is a constant used in many different places in the program, and

the programmer wants to change its value throughout

* By the use of EQU, one can change it once and the assembler will change all of its occurrences
* Use EQU for the counter constant
* The constant is used to load the R3 register

**Example**

COUNT EQU 25

MOV R3, #COUNT

* **DATA:**
* Data directive is used to assign symbol to different address of RAM memory.
* Since the size of RAM memory is 128 bytes from 00H to 7FH, hence memory location from 00H to 7FH can be assigned symbol.

**Example**

TABLE DATA 23H ; Memory location 23H is named as TABLE

ROW CODE F0H ; Memory location F0H is named as ROW

* **CODE:**
* CODE directive is used to assign symbol to different address in program memory.

**Example**

START CODE 00

LAST CODE 0123

**2.1.4 Addressing Modes:**

Addressing modes define the way in which operands can be accessed by instructions.

8051 has four addressing modes.

1. **Immediate Addressing:**

Data is immediately available in the instruction.  
**For example:-**

ADD A, #77 ; Adds 77 (decimal) to A and stores in A

ADD A, #4DH ;  Adds 4D (hexadecimal) to A and stores in A

MOV DPTR, #1000H ; Moves 1000 (hexadecimal) to data pointer

MOV DPTR, #1000H ; Moves 1000 (hexadecimal) to data pointer

***2.* Register Addressing:**   
 This way of addressing accesses the bytes in the current register bank.

* Data is available in the register specified in the instruction.
* The register bank is decided by 2 bits of Processor Status Word (PSW).   
  The address of data is available in the R0 or R1 registers as specified in the instruction.
* Instructions using register addressing are encoded using the three least significant bits of the instruction opcode to specify a register

### Some instructions are specific to a certain register, such as the accumulator, data pointer etc.

### The opcode is 00101111. 00101 indicates the instruction and the three lower bits, 111, specify the register

### Example:

### INC DPTR

### A 1-byte instruction adding 1 to the data pointer

### MUL AB

### A 1-byte instruction multiplying unsigned values in accumulator and register B

### MOV A, @R0

### Moves content of address pointed by R0 to A

### ADD A, R0

### Adds content of R0 to A and stores in A

**3. Direct Addressing:**

Direct addressing can access any on-chip memory location

The address of the data is available in the instruction.

**Example**

ADD A, 55H

MOV P1, A – Transfers the content of accumulator to Port 1 (address 90H )

MOV A, 088H ; Moves content of SFR TCON (address 088H)to A

### 4.  Register Indirect Addressing:

### 8051 has access to eight working registers (R0 to R7)

### How is a variable identified if its address is determined or modified while a program is running?

### 8051 solution is indirect addressing: R0 or R1 may operate as pointer registers (their content indicates an address in internal RAM where data are written or read)

### In 8051 assembly language, indirect addressing is represented by an @ before R0 or R1.

### Example:

### MOV A, @R0

### – Moves a byte of data from internal RAM at location whose address is in R0 to the accumulator

### MOV R0, #60H Loop: MOV @R0,#0 INC R0 CJNE R0,#80H ,Loop

**5.**

**Indexed Addressing**

* Indexed addressing uses a base register (either the program counter or data pointer) and an offset (the accumulator) in forming the effective address for a JMP or MOVC instruction

**Example:**

MOVC A, @A+DPTR

* This instruction moves a byte of data from code memory to the accumulator. The address in code memory is found by adding the accumulator to the data pointer

**2.1.5 Types of 8051 Instructions:**

8051 has about 111 instructions. These can be grouped into the following categories

1. Arithmetic Instructions
2. Logical Instructions
3. Data Transfer instructions
4. Boolean Variable Instructions
5. Program Branching Instructions

The following register, data, address and variables are used while write instructions

A: Accumulator

**B:** "B" register

**C**: Carry bit

**Rn:** Register R0 - R7 of the currently selected register bank

Direct: 8-bit internal direct address for data. The data could be in lower 128bytes of

RAM (00 - 7FH) or it could be in the special function register (80-FFH).

**@Ri :** 8-bit external or internal RAM address available in register R0 or R1.

This is used for indirect addressing mode.

**#data8:** Immediate 8-bit data available in the instruction.

**#data16:** Immediate 16-bit data available in the instruction.

**Addr11:** 11-bit destination address for short absolute jump.

Used by instructions AJMP & ACALL. Jump range is 2 kbyte (one page).

**Addr16:** 16-bit destination address for long call or long jump.

**Rel:** 2's complement 8-bit offset (one - byte) used for short jump (SJMP) and all conditional jumps.

**Bit:** Directly addressed bit in internal RAM or SFR

## Arithmetic Instructions

## 

## ADD

|  |  |
| --- | --- |
| **Operation:** | **ADD, ADDC** |
| **Function:** | Add Accumulator, Add Accumulator With Carry |
| **Syntax:** | ADD A,*operand* |
|  | ADDC A,*operand* |

**Description:-** Description: ADD and ADDC both add the value operand to the value of the Accumulator, leaving the resulting value in the

Accumulator. The value operand is not affected. ADD and ADDC function identically except that ADDC adds the value of operand as well as the value of the Carry flag whereas ADD does not add the Carry flag to the result.

The **Carry bit (C)** is set if there is a carry-out of bit 7. In other words, if the unsigned summed value of the Accumulator, operand and (in the case of ADDC) the Carry flag exceeds 255 Carry is set. Otherwise, the Carry bit is cleared.

The **Auxillary Carry (AC)** bit is set if there is a carry-out of bit 3. In other words, if the unsigned summed value of the low nibble of the Accumulator, operand and (in the case of ADDC) the Carry flag exceeds 15 the Auxillary Carry flag is set. Otherwise, the Auxillary Carry flag is cleared.

The **Overflow (OV)** bit is set if there is a carry-out of bit 6 or out of bit 7, but not both. In other words, if the addition of the Accumulator, operand and (in the case of ADDC) the Carry flag treated as signed values results in a value that is out of the range of a signed byte (-128 through +127) the Overflow flag is set. Otherwise, the Overflow flag is cleared.

## SUBB

**Operation :** SUBB

**Function :** Subtract from Accumulator with borrow

**Syntax :** SUBB A operand

**Description:** SUBB subtract the value of *operand* from the value of the Accumulator, leaving the resulting value in the Accumulator. The value *operand* is not affected.

The **Carry Bit (C)** is set if a borrow was required for bit 7, otherwise it is cleared. In other words, if the unsigned value being subtracted is greater than the Accumulator the Carry Flag is set.

The **Auxillary Carry (AC)** bit is set if a borrow was required for bit 3, otherwise it is cleared. In other words, the bit is set if the low nibble of the value being subtracted was greater than the low nibble of the Accumulator.

The **Overflow (OV)** bit is set if a borrow was required for bit 6 or for bit 7, but not both. In other words, the subtraction of two signed bytes resulted in a value outside the range of a signed byte (-128 to 127). Otherwise it is cleared.

## DEC

**Operation :** DEC

**Faction :** Decrement Register

**Syntax :** DEC register

**Description:** DEC 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). Note: The Carry Flag is NOT set when the value "rolls over" from 0 to 255.

## MUL

**Operation :** MUL

**Function :** Multiply Accumulator by B

**Syntax :** MUL A B

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **Op Code** | **Bytes** | **Flags** |
| MUL AB | 0xA4 | 1 | C, OV |

**Description:** Multiples the unsigned value of the Accumulator by the unsigned value of the "B" register. The least significant byte of the result is placed in the Accumulator and the most-significant-byte is placed in the "B" register.

The **Carry Flag (C)** is always cleared.

The **Overflow Flag (OV)** is set if the result is greater than 255 (if the most-significant byte is not zero), otherwise it is cleared.

## DIV

**Operation :** DIV

**Function :** Divide Accumulator by B

**Syntax :** DIV AB

**DIV**

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **OpCode** | **Bytes** | **Flags** |
| DIV AB | 0x84 | 1 | C, OV |

**Description:** Divides the unsigned value of the Accumulator by the unsigned value of the "B" register. The resulting quotient is placed in the Accumulator and the remainder is placed in the "B" register.

The **Carry flag (C)** is always cleared.

The **Overflow flag (OV)** is set if division by 0 was attempted, otherwise it is cleared.

**INC**

**Operation :** INC

**Function :** Increment register

**Syntax :** INC register

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **Op Code** | **Bytes** | **Flags** |
| INC *iram addr* | 0x05 | 2 | None |
| INC @R0 | 0x06 | 1 | None |
| INC @R1 | 0x07 | 1 | None |
| INC A | 0x04 | 1 | None |
| INC R0 | 0x08 | 1 | None |
| INC DPTR | 0xA3 | 1 | None |

**Description:** INC increments the value of *register* by 1. If the initial value of *register* is 255 (0xFF Hex), incrementing the value will cause it to reset to 0. Note: 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 65535 (0xFFFF Hex), incrementing the value will cause it to reset to 0. Again, the Carry Flag is NOT set when the value of DPTR "rolls over" from 65535 to 0.

**2] Logical instruction :**

**ANL**

**Operation :** ANL

**Function:** Bitwise AND

**Syntax:** ANL Operand, operand 2

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **Op Code** | **Bytes** | **Flags** |
| ANL  *iram addr*,A | 0x52 | 2 | None |
| ANL  *iram addr*,#*data* | 0x53 | 3 | None |
| ANL A,#*data* | 0x54 | 2 | None |
| ANL A,*iram addr* | 0x55 | 2 | None |
| ANL A,@R0 | 0x56 | 1 | None |
| ANL A,@R1 | 0x57 | 1 | None |
| ANL A,R0 | 0x58 | 1 | None |
| ANL C,*bit addr* | 0x82 | 2 | C |
| ANL C,/*bit addr* | 0xB0 | 2 | C |

operand and sets the corresponding bit in the resulting byte only if the bit was set in both of the original operands, otherwise the resulting bit is cleared.

## CLR

**Operation :** CLR

**Function :** Clear Register.

**Syntax :**  CRL register

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **Op Code** | **Bytes** | **Flags** |
| CLR *bit addr* | 0xC2 | 2 | None |
| CLR C | 0xC3 | 1 | C |
| CLR A | 0xE4 | 1 | None |

## Description:

## CLR:- Clears (sets to 0) all the bit(s) of the indicated register. If the register is a bit (including the carry bit), only the specified bit is affected. Clearing the Accumulator sets the Accumulator's value to 0.

## CPL

**Function :** Complement Register

**Syntax :** CPL operand

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **Op Code** | **Bytes** | **Flags** |
| CPL A | 0xF4 | 1 | None |
| CPL C | 0xB3 | 1 | C |
| CPL *bit addr* | 0xB2 | 2 | None |

**Description:**

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. This can be thought of as "Accumulator Logical Exclusive OR 255" or as "255-Accumulator." If the operand refers to a bit of an output Port, the value that will be complemented is based on the last value written to that bit, not the last value read from it

**Description:**  ANL does a bitwise "AND" operation between *operand1* and *operand2*, leaving the resulting value in *operand1*. The value of operand2 is not affected. A logical "AND" compares the bits of each

## ORL

**Operation :** ORL

**Function :** Bitwise OR

**Syntax :** PRL operand operand?

|  |  |  |  |
| --- | --- | --- | --- |
| ORL *iram addr*,A | 0x42 | 2 | None |
| ORL *iram addr*,#*data* | 0x43 | 3 | None |
| ORL A,#*data* | 0x44 | 2 | None |
| ORL A,*iram addr* | 0x45 | 2 | None |
| ORL A,@R0 | 0x46 | 1 | None |
| ORL A,@R1 | 0x47 | 1 | None |
| ORL A,R0 | 0x48 | 1 | None |
| ORL C,*bit addr* | 0x72 | 2 | C |
| ORL C,/*bit addr* | 0xA0 | 2 | C |

**Description:**

ORL does a bitwise "OR" operation between operand1 and operand2, leaving the resulting value in operand1. The value of operand2 is not affected. A logical "OR" compares the bits of each operand and sets the corresponding bit in the resulting byte if the bit was set in either of the original operands, otherwise the resulting bit is cleared.

## XRL

**Operation:** XRL

**Function:** Bitwise Exclusive OR

**Syntax:** XRL operand, operand2

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **OpCode** | **Bytes** | **Flags** |
| XRL iram addr,A | 0x62 | 2 | None |
| XRL iram addr,#data | 0x63 | 3 | None |
| XRL A,#data | 0x64 | 2 | None |
| XRL A,iram addr | 0x65 | 2 | None |
| XRL A,@R0 | 0x66 | 1 | None |
| XRL A,@R1 | 0x67 | 1 | None |
| XRL A,R0 | 0x68 | 1 | None |

**Description:** XRL does a bitwise "EXCLUSIVE OR" operation between *operand1* and *operand2*, leaving the resulting value in *operand1*. The value of operand2 is not affected. A logical "EXCLUSIVE OR" compares the bits of each operand and sets the corresponding bit in the resulting byte if the bit was set in either (but not both) of the original operands, otherwise the bit is cleared.

**ORL**

**Operation:** ORL

**Function:** ORL operand, operand2

**Syntax:** ORL operand, operand2

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **OpCode** | **Bytes** | **Flags** |
| ORL *iram addr*,A | 0x42 | 2 | None |
| ORL *iram addr*,#*data* | 0x43 | 3 | None |
| ORL A,#*data* | 0x44 | 2 | None |
| ORL A,*iram addr* | 0x45 | 2 | None |
| ORL A,@R0 | 0x46 | 1 | None |
| ORL A,@R1 | 0x47 | 1 | None |
| ORL A,R0 | 0x48 | 1 | None |
| ORL C,*bit addr* | 0x72 | 2 | C |
| ORL C,/*bit addr* | 0xA0 | 2 | C |

**Description:** ORL does a bitwise "OR" operation between operand1 and operand2, leaving the resulting value in operand1. The value of operand2 is not affected. A logical "OR" compares the bits of each operand and sets the corresponding bit in the resulting byte if the bit was set in either of the original operands, otherwise the resulting bit is cleared.

## RL

**Operation :** RL

**Function :** Rotate Accumulator Left

**Syntax :** RL A

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **OpCode** | **Bytes** | **Flags** |
| RL A | 0x23 | 1 | C |

**Description:** Shifts the bits of the Accumulator to the left. The left-most bit (bit 7) of the Accumulator is loaded into bit 0.

## RLC

|  |  |
| --- | --- |
| **Operation :** | **RLC** |
| **Function :** | Rotate Accumulator Left Through Carry |
| **Syntax :** | RLC A |

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **OpCode** | **Bytes** | **Flags** |
| RLC A | 0x33 | 1 | C |

**Description:** Shifts the bits of the Accumulator to the left. The left-most bit (bit 7) of the Accumulator is loaded into the Carry Flag, and the original Carry Flag is loaded into bit 0 of the Accumulator. This function can be used to quickly multiply a byte by 2.

## RR

|  |  |
| --- | --- |
| **Operation :** | **RR** |
| **Function :** | Rotate Accumulator Right |
| **Syntax :** | RR A |

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **OpCode** | **Bytes** | **Flags** |
| RR A | 0x03 | 1 | None |

**Description:** Shifts the bits of the Accumulator to the right. The right-most bit (bit 0) of the Accumulator is loaded into bit 7.

## RRC

|  |  |
| --- | --- |
| **Operation :** | **RRC** |
| **Function :** | Rotate Accumulator Right Through Carry |
| **Syntax :** | RRC A |

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **OpCode** | **Bytes** | **Flags** |
| RRC A | 0x13 | 1 | C |

**Description:** Shifts the bits of the Accumulator to the right. The right-most bit (bit 0) of the Accumulator is loaded into the Carry Flag, and the original Carry Flag is loaded into bit 7. This function can be used to quickly divide a byte by 2.

## SWAP

|  |  |
| --- | --- |
| **Operation :** | **SWAP** |
| **Function :** | Swap Accumulator Nibbles |
| **Syntax :** | SWAP A |

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **OpCode** | **Bytes** | **Flags** |
| SWAP A | 0xC4 | 1 | None |

**Description:** SWAP swaps bits 0-3 of the Accumulator with bits 4-7 of the Accumulator. This instruction is identical to executing "RR A" or "RL A" four times.

**Data Transfer Instructions**

## MOV

|  |  |
| --- | --- |
| **Operation :** | **MOV** |
| **Function :** | Move Memory |
| **Syntax :** | MOV *operand1*,*operand2* |

|  |  |  |  |
| --- | --- | --- | --- |
| MOV @R0,#data | 0x76 | 2 | None |
| MOV @R1,#data | 0x77 | 2 | None |
| MOV @R0,A | 0xF6 | 1 | None |
| MOV @R1,A | 0xF7 | 1 | None |
| MOV @R0,iram addr | 0xA6 | 2 | None |
| MOV @R1,iram addr | 0xA7 | 2 | None |
| MOV A,#data | 0x74 | 2 | None |
| MOV A,@R0 | 0xE6 | 1 | None |
| MOV C,bit addr | 0xA2 | 2 | C |
| MOV DPTR,#data16 | 0x90 | 3 | None |
| MOV Rn,#data | 0x78 | 2 | None |
| MOV Rn,iram addr | 0xA8 | 2 | None |
| MOV bit addr,C | 0x92 | 2 | None |
| MOV iram addr,#data | 0x75 | 3 | None |
| MOV iram addr,@R0 | 0x86 | 2 | None |
| MOV iram addr,@R1 | 0x87 | 2 | None |
| MOV iram addr,Rn | 0x88 | 2 | None |
| MOV iram addr,A | 0xF5 | 2 | None |
| MOV iram addr,iram addr | 0x85 | 3 | None |

**Description:** MOV copies the value of *operand2* into *operand1*. The value of *operand2* is not affected. Both *operand1* and *operand2* must be in Internal RAM. No flags are affected unless the instruction is moving the value of a bit into the carry bit in which case the carry bit is affected or unless the instruction is moving a value into the PSW register (which contains all the program flags).

\*\* Note: In the case of "MOV iram addr,iram addr", the operand bytes of the instruction are stored in reverse order. That is, the instruction consisting of the bytes 0x85, 0x20, 0x50

means "Move the contents of Internal RAM location 0x20 to Internal RAM location 0x50" whereas the opposite would be generally presumed.

## MOVC

|  |  |
| --- | --- |
| **Operation :** | **MOVC** |
| **Function :** | Move Code Byte to Accumulator |
| **Syntax :** | MOVC A,@A+*register* |

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **OpCode** | **Bytes** | **Flags** |
| MOVC A,@A+DPTR | 0x93 | 1 | None |
| MOVC A,@A+PC | 0x83 | 1 | None |

**Description:** MOVC moves a byte from Code Memory into the Accumulator. The Code Memory address from which the byte will be moved is calculated by summing the value of the Accumulator with either DPTR or the Program Counter (PC). In the case of the Program Counter, PC is first incremented by 1 before being summed with the Accumulator.

## MOVX

|  |  |
| --- | --- |
| **Operation :** | **MOVX** |
| **Function :** | Move Data To/From External Memory (XRAM) |
| **Syntax :** | MOVX *operand1*,*operand2* |

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **OpCode** | **Bytes** | **Flags** |
| MOVX @DPTR,A | 0xF0 | 1 | None |
| MOVX @R0,A | 0xF2 | 1 | None |
| MOVX @R1,A | 0xF3 | 1 | None |
| MOVX A,@DPTR | 0xE0 | 1 | None |
| MOVX A,@R0 | 0xE2 | 1 | None |
| MOVX A,@R1 | 0xE3 | 1 | None |

**Description:** MOVX moves a byte to or from External Memory into or from the Accumulator.

If *operand1* is @DPTR, the Accumulator is moved to the 16-bit External Memory address indicated by DPTR. This instruction uses both P0 (port 0) and P2 (port 2) to output the 16-bit address and data. If*operand2* is DPTR then the byte is moved from External Memory into the Accumulator.

If *operand1* is @R0 or @R1, the Accumulator is moved to the 8-bit External Memory address indicated by the specified Register. This instruction uses only P0 (port 0) to output the 8-bit address and data. P2 (port 2) is not affected. If *operand2* is @R0 or @R1 then the byte is moved from External Memory into the Accumulator.

## XCH

|  |  |
| --- | --- |
| **Operation :** | **XCH** |
| **Function :** | Exchange Bytes |
| **Syntax :** | XCH A,*register* |

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **OpCode** | **Bytes** | **Flags** |
| XCH A,@R0 | 0xC6 | 1 | None |
| XCH A,@R1 | 0xC7 | 1 | None |
| XCH A,R0 | 0xC8 | 1 | None |

**Description:** Exchanges the value of the Accumulator with the value contained in *register*.

## XCHD

|  |  |
| --- | --- |
| **Operation :** | **XCHD** |
| **Function :** | Exchange Digit |
| **Syntax :** | XCHD A,*[@R0/@R1]* |

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **OpCode** | **Bytes** | **Flags** |
| XCHD A,@R0 | 0xD6 | 1 | None |
| XCHD A,@R1 | 0xD7 | 1 | None |

**Description:** Exchanges bits 0-3 of the Accumulator with bits 0-3 of the Internal RAM address pointed to indirectly by R0 or R1. Bits 4-7 of each register are unaffected.

**Boolean Variable Instructions**

## SETB

|  |  |
| --- | --- |
| **Operation :** | **SETB** |
| **Function :** | Set Bit |
| **Syntax :** | SETB *bit addr* |

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **OpCode** | **Bytes** | **Flags** |
| SETB C | 0xD3 | 1 | C |
| SETB *bit addr* | 0xD2 | 2 | None |

**Description:** Sets the specified bit.

|  |  |  |  |
| --- | --- | --- | --- |
| **Mnemonics** | **Description** | **Bytes** | **Instruction Cycles** |
| CLR C | C-bit http://nptel.ac.in/courses/117104072/micro/lecture13/images/arrow.gif0 | 1 | 1 |
| CLR bit | bit http://nptel.ac.in/courses/117104072/micro/lecture13/images/arrow.gif0 | 2 | 1 |
| SET C | C http://nptel.ac.in/courses/117104072/micro/lecture13/images/arrow.gif1 | 1 | 1 |
| SET bit | bit http://nptel.ac.in/courses/117104072/micro/lecture13/images/arrow.gif1 | 2 | 1 |
| CPL C | Chttp://nptel.ac.in/courses/117104072/micro/lecture13/images/arrow.gif  http://nptel.ac.in/courses/117104072/micro/lecture13/images/c-bar.gif | 1 | 1 |
| CPL bit | bit http://nptel.ac.in/courses/117104072/micro/lecture13/images/arrow.gif  http://nptel.ac.in/courses/117104072/micro/lecture13/images/bit-bar.gif | 2 | 1 |
| ANL C, /bit | C http://nptel.ac.in/courses/117104072/micro/lecture13/images/arrow.gifC . http://nptel.ac.in/courses/117104072/micro/lecture13/images/bit-bar.gif | 2 | 1 |
| ANL C, bit | C http://nptel.ac.in/courses/117104072/micro/lecture13/images/arrow.gifC. bit | 2 | 1 |
| ORL C, /bit | Chttp://nptel.ac.in/courses/117104072/micro/lecture13/images/arrow.gif C + http://nptel.ac.in/courses/117104072/micro/lecture13/images/bit-bar.gif | 2 | 1 |
| ORL C, bit | C http://nptel.ac.in/courses/117104072/micro/lecture13/images/arrow.gifC + bit | 2 | 1 |
| MOV C, bit | Chttp://nptel.ac.in/courses/117104072/micro/lecture13/images/arrow.gif bit | 2 | 1 |
| MOV bit, C | bit http://nptel.ac.in/courses/117104072/micro/lecture13/images/arrow.gifC | 2 | 2 |

**Program Branching Instructions**

## ACALL

|  |  |
| --- | --- |
| **Operation:** | **ACALL** |
| **Function:** | Absolute Call Within 2K Block |
| **Syntax:** | ACALL *code address* |

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **OpCode** | **Bytes** | **Flags** |
| ACALL *page0* | 0x11 | 2 | None |

**Description:** ACALL unconditionally calls a subroutine at the indicated code address. ACALL pushes the address of the instruction that follows ACALL onto the stack, least-significant-byte first, most-significant-byte second. The Program Counter is then updated so that program execution continues at the indicated address.

The new value for the Program Counter is calculated by replacing the least-significant-byte of the Program Counter with the second byte of the ACALL instruction, and replacing bits 0-2 of the most-significant-byte of the Program Counter with 3 bits that indicate the page. Bits 3-7 of the most-significant-byte of the Program Counter remain unchaged.

Since only 11 bits of the Program Counter are affected by ACALL, calls may only be made to routines located within the same 2k block as the first byte that follows ACALL.

## LCALL

|  |  |
| --- | --- |
| **Operation :** | **LCALL** |
| **Function :** | Long Call |
| **Syntax :** | LCALL *code addr* |

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **OpCode** | **Bytes** | **Flags** |
| LCALL *code addr* | 0x12 | 3 | None |

**Description:** LCALL calls a program subroutine. LCALL increments the program counter by 3 (to point to the instruction following LCALL) and pushes that value onto the stack (low byte first, high byte second). The Program Counter is then set to the 16-bit value which follows the LCALL opcode, causing program execution to continue at that address.

## AJMP

|  |  |
| --- | --- |
| **Operation :** | **AJMP** |
| **Function :** | Absolute Jump Within 2K Block |
| **Syntax :** | AJMP *code address* |

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **OpCode** | **Bytes** | **Flags** |
| AJMP page0 | 0x01 | 2 | None |

**Description:** AJMP unconditionally jumps to the indicated code address. The new value for the Program Counter is calculated by replacing the least-significant-byte of the Program Counter with the second byte of the AJMP instruction, and replacing bits 0-2 of the most-significant-byte of the Program Counter with 3 bits that indicate the page of the byte following the AJMP instruction. Bits 3-7 of the most-significant-byte of the Program Counter remain unchaged.

Since only 11 bits of the Program Counter are affected by AJMP, jumps may only be made to code located within the same 2k block as the first byte that follows AJMP.

## LJMP

|  |  |
| --- | --- |
| **Operation:** | **LJMP** |
| **Function:** | Long Jump |
| **Syntax:** | LJMP *code addr* |

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **OpCode** | **Bytes** | **Flags** |
| LJMP *code addr* | 0x02 | 3 | None |

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

## CJNE

|  |  |
| --- | --- |
| **Operation :** | **CJNE** |
| **Function :** | Compare and Jump If Not Equal |
| **Syntax :** | CJNE *operand1*,*operand2*,*reladdr* |

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **OpCode** | **Bytes** | **Flags** |
| CJNE A,#data,reladdr | 0xB4 | 3 | C |
| CJNE A,iram addr,reladdr | 0xB5 | 3 | C |
| CJNE @R0,#data,reladdr | 0xB6 | 3 | C |

**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

## DJNZ

|  |  |
| --- | --- |
| **Operation :** | **DJNZ** |
| **Function :** | Decrement and Jump if Not Zero |
| **Syntax :** | DJNZ register,reladdr |

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **OpCode** | **Bytes** | **Flags** |
| DJNZ iram addr,reladdr | 0xD5 | 3 | None |
| DJNZ R0,reladdr | 0xD8 | 2 | None |

**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.

## JB

|  |  |
| --- | --- |
| **Operation :** | **JB** |
| **Function :** | Jump if Bit Set |
| **Syntax :** | JB bit addr, reladdr |

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **OpCode** | **Bytes** | **Flags** |
| JB bit addr,reladdr | 0x20 | 3 | None |

**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.

## JBC

|  |  |
| --- | --- |
| **Operation :** | **JBC** |
| **Function :** | Jump if Bit Set and Clear Bit |
| **Syntax :** | JB bit addr, reladdr |

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **OpCode** | **Bytes** | **Flags** |
| JBC bit addr,reladdr | 0x10 | 3 | None |

**Description:** JBC will branch to the address indicated by *reladdr* if the bit indicated by *bit addr* is set. Before branching to *reladdr* the instruction will clear the indicated bit. If the bit is not set program execution continues with the instruction following the JBC instruction.

## JC

|  |  |
| --- | --- |
| **Operation :** | **JC** |
| **Function :** | Jump if Carry Set |
| **Syntax :** | JC reladdr |

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **OpCode** | **Bytes** | **Flags** |
| JC *rel addr* | 0x40 | 2 | None |

**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.

## JNC

|  |  |
| --- | --- |
| **Operation :** | **JNC** |
| **Function :** | Jump if Carry Not Set |
| **Syntax :** | JNC reladdr |

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **OpCode** | **Bytes** | **Flags** |
| JNC *reladdr* | 0x50 | 2 | None |

**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.

## JNZ

|  |  |
| --- | --- |
| **Operation :** | **JNZ** |
| **Function :** | Jump if Accumulator Not Zero |
| **Syntax :** | JNZ reladdr |

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **OpCode** | **Bytes** | **Flags** |
| JNZ reladdr | 0x70 | 2 | None |

**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.

## JZ

|  |  |
| --- | --- |
| **Operation :** | **JZ** |
| **Function :** | Jump if Accumulator Zero |
| **Syntax :** | JNZ reladdr |

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **OpCode** | **Bytes** | **Flags** |
| JZ reladdr | 0x60 | 2 | None |

**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.

## NOP

|  |  |
| --- | --- |
| **Operation :** | **NOP** |
| **Function :** | None, waste time |
| **Syntax :** | No Operation |

|  |  |  |  |
| --- | --- | --- | --- |
| **Instructions** | **OpCode** | **Bytes** | **Flags** |
| NOP | 0x00 | 1 | None |

**Description:** NOP, as it's name suggests, causes No Operation to take place for one machine cycle. NOP is generally used only for timing purposes. Absolutely no flags or registers are affected.

**Programming 8051 using ‘C’ and ‘Assembly’**

In this there will be simple programs such as addition, subtraction, multiplication, division in assembly and ‘C’

**The step of Assembly language program are outlines 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

* 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

* The linker program takes one or more object code files and produce an absolute object file with the extension “abs”
* 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
* Recent Windows-based assemblers combine step 2 through 4 into one step

* **Programs for ADDITION using assembly language programming**

**Ex1**: Suppose the two data bytes are 30H and 20H then perform the addition of two 8 bit numbers

MOV A,#30H ; load the data 30H in A register

ADD A,#20H ; add the present data in A with 20H

LOOP**:** AJMP LOOP ; stop

**Ex 2:** Program that adda 77H with 44H

ADD A, #77 ; Adds 77 (decimal) to A and stores in A

ADD A, #4DH ;  Adds 4D (hexadecimal) to A and stores in A

**Ex3:**

MOV A, #25H ;load 25H into A

MOV R2, #34H ;load 34H into R2

ADD A, R2 ;add R2 to Accumulator ;(A = A + R2)

**Ex4:**

Show the status of the CY, AC and P flag after the addition of 38H and 2FH in the following instructions.

MOV A, #38H

ADD A, #2FH ;after the addition A=67H, CY=0

**Solution:**

38 00111000

+ 2F 00101111

67 01100111

CY = 0 since there is no carry beyond the D7 bit

AC = 1 since there is a carry from the D3 to the D4 bi

P = 1 since the accumulator has an odd number of 1s (it has five 1s)

* **Programs for SUBTRACTION using assembly language programming**

**Ex:** 1 Program to subtract 6E from 4C

CLR C

MOV A,#4C ;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

* **Programs for MULTIPLICATION using assembly language programming**

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

**Programs for DIVISION using assembly language programming**

**Ex1:**

1. 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.
2. Assuming that P1 has a value of FDH for data, analyze program.

**Solution:**

**(a)**

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

**(b) To convert a binary (hex) value to decimal, we divide it by 10**

repeatedly until the quotient is less than 10. After each division the

remainder is saves.

**Q R**

FD/0A = 19 3 (low digit)

19/0A = 2 5 (middle digit)

2 (high digit)

Therefore, we have FDH = 253.

**Execution of program using cross compiler like Keil IDE, SPJ, RIDE**

**Steps:**

1. Create a project in Keil software
2. Select microcontroller 8051 from ACTEL or 89c51 from ATMEL also select option to create hex file
3. Create a C file.
4. ADD that file into the project
5. Build the target
6. Debug the program
7. Run the the program
8. Check the output in peripheral register.

**QUESTION BANK**

**2M Questions**

1. List different addressing modes of 8051 with example
2. Compare use of XCH ,PUSH and POP instruction in detail.
3. List different addressing modes of 8051
4. List different types of instructions in 8051

**4M Questions**

1. List the addressing modes of 8051. Also give the example of

each mode. [ summer 12]

1. Write down the given instructions in different addressing modes with example. [winter 12]
2. Describe any four bit handling instructions of 8051 [ 2011-12]
3. Explain the function of following instructions [ Summer 2013]

MOV A,@Ri

INC @Ri

RETI

SWAP A

1. Explain function of DAA instruction.
2. Explain the following instruction

CJNE A, direct, rel

XCHD A, @Ri

**6M Questions**

1. Write an assembly language program to add two BCD numbers. Also store the result in data memory location.[Winter 2012]
2. Write an assembly language program in 8051 to add ten numbers in an array.[ SUMMER 2013]