Course Coordinator: Prof. Mahesh P.Yanagimath

Module-2

Assembly programming and instruction of 8051

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

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

Assembling and running an 8051 program

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 8051assembler. 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.

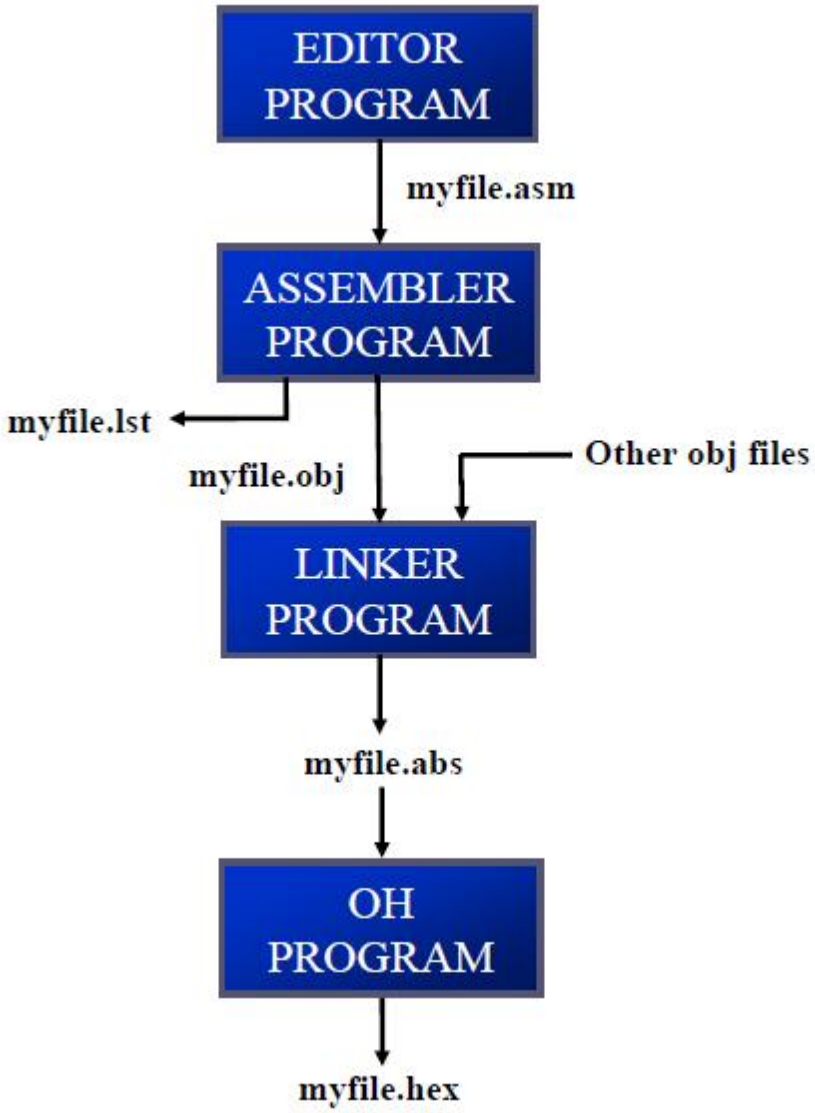

Fig: Flowchart showing Assembly programming.

8051 Data types and directives

The 8051 microcontroller has only one data type. It is 8 bits, and 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) to be processed by the CPU.

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, or ASCII

formats. For decimal, the "D" after the decimal number is optional, but using "B" (binary) and "H" (hexadecimal) for the others is required. Regardless of which is used, the assembler will convert the numbers into hex. To indicate ASCII, simply place the characters in quotation marks ('like this'). The assembler will assign the ASCII code for the numbers or characters automatically. The DB directive is the only directive that can be used to define ASCII strings larger than two characters; therefore, it should be used for all ASCII data definitions.

Assembler directives

The following are widely used directives of the 8051.

ORG (origin)

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

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 so that when the label appears in the program, its constant value will be substituted for the label. The following uses EQU for the counter constant and then the constant is used to load the R3 register.

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

What is the advantage of using EQU?

Assume that there is a constant (a fixed value) used in many different places in the program, and the programmer wants to change its value throughout. By the use of EQU, the programmer can change it once and the assembler will change all of its occurrences, rather than search the entire program trying to find every occurrence.

END directive

Another important pseudocode is the END directive. This indicates to the assembler the end of the source (asm) file. The END directive is the last line of an 8051 program, meaning that in the source code anything after the END directive is ignored by the assembler. Some assemblers use "END" (notice the dot) instead of "END".

Instruction Syntax

8051 instruction set is coded set that have been defined by manufacturer of 8051.

An 8051 instruction syntax is shown below.

Mnemonic	Destination operand	Source operand

Operand is data address it specifies the destination for the data i.e being copied form the source and also it gives us source address. Destination and source addresses are separated by (,) comma. Assembler will come to know when one operand name ends and other begins.

Data transfer instructions

MOV <dest-byte>,<src-byte>-

Function: Move byte variable

Description: The byte variable indicated by the second operand is copied into the location specified by the first operand. The source byte is not affected. No other register or flag is affected.

1. mov direct , A
2. mov A, @R_i
3. mov A, R_n
4. mov direct, direct
5. mov A, #data

EX: MOV 30h, A

MOV A,@R0 ; moves the content of memory pointed to by Ro into A

MOV A, R_1; moves the content of Register R_1 to Accumulator A

MOV 20h,30h; moves the content of memory location 30h to 20h

MOV A,#45h; moves 45h to Accumulator A

MOV <dest-bit>,<src-bit>

Function: Move bit data

Description: MOV <dest-bit>,<src-bit> copies the Boolean variable indicated by the second operand into the location specified by the first operand. One of the operands must be the carry flag; the other may be any directly addressable bit. No other register or flag is affected.

Example: MOV P1.3,C; moves the carry bit to 3rd bit of port1

MOV DPTR,#data16

Function: Load Data Pointer with a 16-bit constant

Description: MOV DPTR,#data16 loads the Data Pointer with the 16-bit constant indicated. The 16-bit constant is loaded into the second and third bytes of the instruction. The second byte (DPH) is the high-order byte, while the third byte(DPL) holds the lower-order byte. No flags are affected.

This is the only instruction which moves 16 bits of data at once.

Example: The instruction,MOV DPTR, # 4567H loads the value 4567H into the Data Pointer. DPH holds 45H, and DPL holds 67H.

MOVC A,@A+ <base-reg>

Function: Move Code byte

Description: The MOVC instructions load the Accumulator with a code byte or constant from program memory. The address of the byte fetched is the sum of the original unsigned 8-bit Accumulator contents and the contents of a 16-bit base register, which may be either the Data Pointer or the PC. In the latter case, the PC is incremented to the address of the following instruction before being added with the Accumulator; otherwise the base register is not altered. Sixteen-bit addition is performed so a carry-out from the low-order eight bits may propagate through higher-order bits. No flags are affected.

MOVC A,@A+PC

(PC) (PC) + 1

(A) ((A) + (PC))

MOVX <dest-byte>,<src-byte>

Function: Move External

Description: The MOVX instructions transfer data between the Accumulator and a byte of external data memory, which is why "X" is appended to MOV. There are two types of instructions, differing in whether they provide an 8-bit or 16-bit indirect address to the external data RAM.

This form of MOVX is faster and more efficient when accessing very large data arrays (up to 64K bytes), since no additional instructions are needed to set up the output ports.

Example:

MOVX A,@DPTR

MOVX @DPTR, A

(A) ((DPTR))

PUSH direct

Function: Push onto stack

Description: The Stack Pointer is incremented by one. The contents of the indicated variable is then copied into the internal RAM location addressed by the Stack Pointer. No flags are affected.

Example: On entering an interrupt routine, the Stack Pointer contains 09H. The Data Pointer holds the value 0123H. The following instruction sequence,

PUSH DPL

PUSH DPH

leaves the Stack Pointer set to 0BH and stores 23H and 01H in internal RAM locations 0AH and 0BH respectively.

POP direct

Function: Pop from stack.

Description: The contents of the internal RAM location addressed by the Stack Pointer is read, and the Stack Pointer is decremented by one. The value read is then transferred to the directly addressed byte indicated. No flags are affected.

Example: The Stack Pointer originally contains the value 32H, and internal RAM locations 30H through 32H contain the values 20H, 23H, and 01H, respectively. The following instruction sequence,

POP DPH

POP DPL leaves the Stack Pointer equal to the value 30H and sets the Data Pointer to 0123H.

Arithmetic Group of Instructions

ADD A,<src-byte>

Function: Add

Description: ADD adds the byte variable indicated to the Accumulator, leaving the result in the Accumulator. The carry and auxiliary-carry flags are set, respectively, if there is a carry-out from bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occurred. OV is set if a carry-out of bit 7 but not bit 6; otherwise, OV is cleared. When adding signed integers, OV indicates a negative number produced as the sum of two positive operands, or a positive sum from two negative operands. Four source operand addressing modes are allowed: register, direct, register-indirect, or immediate.

Example: The Accumulator holds 0C3H (1100001lB), and register 0 holds 0AAH (10101010B). The following instruction,

ADD A,R0

leaves 6DH (01101101B) in the Accumulator with the AC flag cleared and both the carry flag and OV set to 1.

ADD A, direct

(A) (A) + (direct)

ADD A, @Ri

(A) (A) + data

ADDC A, <src-byte>

Function: Add with Carry

Description: ADDC simultaneously adds the byte variable indicated, the carry flag and the Accumulator contents, leaving the result in the Accumulator. The carry and auxiliary-carry flags

are set respectively, if there is a carry-out from bit 7 or bit 3, and cleared otherwise. When adding signed integers, OV indicates a negative number produced as the sum of two positive operands or a positive sum from two negative operands. Four source operand addressing modes are allowed: register, direct, register-indirect, or immediate.

Example: The Accumulator holds 0C3H (11000011B) and register 0 holds 0AAH (10101010B) with the carry flag set. The following instruction,

ADDC A,R0

leaves 6EH (01101110B) in the Accumulator with AC cleared and both the Carry flag and OV set to 1.

ADDC A,Rn

Operation: ADDC

- (A) + (C) + (Rn)

ADDC A, direct

Operation: ADDC

(A) (A) + (C) + (direct)

ADDC A,@Ri

Operation: ADDC

(A) (A) + (C) + ((Ri))

ADDC A, #data

Operation: ADDC

(A) (A) + (C) + #data

SUBB A,<src-byte>

Function: Subtract with borrow

Description: SUBB subtracts the indicated variable and the carry flag together from the Accumulator, leaving the result in the Accumulator. SUBB sets the carry (borrow) flag if a borrow is needed for bit 7 and clears C otherwise. (If C was set *before* executing a SUBB

instruction, this indicates that a borrow was needed for the previous step in a multiple-precision subtraction, so the carry is subtracted from the Accumulator along with the source operand.) AC is set if a borrow is needed for bit 3 and cleared otherwise. The source operand allows four addressing modes: register, direct, register-indirect, or immediate.

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

Instructions	**OpCode**	**Bytes**	**Flags**
SUBB A,*#data*	0x94	2	C, AC, OV
SUBB A,*iram addr*	0x95	2	C, AC, OV
SUBB A,@R0	0x96	1	C, AC, OV
SUBB A,@R1	0x97	1	C, AC, OV
SUBB A,R0	0x98	1	C, AC, OV
SUBB A,R1	0x99	1	C, AC, OV
SUBB A,R2	0x9A	1	C, AC, OV
SUBB A,R3	0x9B	1	C, AC, OV
SUBB A,R4	0x9C	1	C, AC, OV
SUBB A,R5	0x9D	1	C, AC, OV
SUBB A,R6	0x9E	1	C, AC, OV
SUBB A,R7	0x9F	1	C, AC, OV

SUBB A,Rn

Operation: SUBB

(A) (A) - (C) - (Rn)

SUBB A, direct

Operation: SUBB

(A) (A) - (C) - (direct)

SUBB A,@Ri

Operation: SUBB

(A) (A) - (C) - ((Ri))

MUL AB

Function: Multiply

Description: MUL AB multiplies the unsigned 8-bit integers in the Accumulator and register B. The low-order byte of the 16-bit product is left in the Accumulator, and the high-order byte in B. If the product is greater than 255 (0FFH), the overflow flag is set; otherwise it is cleared. The carry flag is always cleared.

Example: Originally the Accumulator holds the value 80 (50H). Register B holds the value 160 (0A0H). The instruction,

MUL AB

will give the product 12,800 (3200H), so B is changed to 32H (00110010B) and the Accumulator is cleared. The overflow flag is set, carry is cleared.

DIV AB

Function: Divide

Description: DIV AB divides the unsigned eight-bit integer in the Accumulator by the unsigned eight-bit integer in register B. The Accumulator receives the integer part of the quotient; register B receives the integer remainder. The carry and OV flags are cleared. *Exception*: if B had originally contained 00H, the values returned in the Accumulator and B-register are undefined and the overflow flag are set. The carry flag is cleared in any case.

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

DA A

Function: Decimal-adjust Accumulator for Addition

Description: DA A adjusts the eight-bit value in the Accumulator resulting from the earlier addition of two variables (each in packed-BCD format), producing two four-bit digits. Any ADD or ADDC instruction may have been used to perform the addition. If Accumulator bits 3 through 0 are greater than nine or if the AC flag is one, six is added to the Accumulator producing the proper BCD digit in the low-order nibble. This internal addition sets the carry flag if a carry-out of the low-order four-bit field propagates through all high-order bits, but it does not clear the carry flag otherwise.

If the carry flag is now set, or if the four high-order bits now exceed nine, these high-order bits are added with six, producing the proper BCD digit in the high-order nibble. Again, this sets the carry flag if there is a carry-out of the high-order bits, but does not clear the carry.

SWAP A

Function: Swap nibbles within the Accumulator

Description: SWAP A interchanges the low- and high-order nibbles (four-bit fields) of the Accumulator (bits 3 through 0 and bits 7 through 4). The operation can also be thought of as a 4-bit rotate instruction. No flags are affected.

Example: The Accumulator holds the value 0C5H (11000101B). The instruction, SWAP A leaves the Accumulator holding the value 5CH (01011100B

Operation: SWAP

(A3-0) D (A7-4)

XCH A,<byte>

Function: Exchange Accumulator with byte variable

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

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

XCHD A,@Ri

Function: Exchange Digit

Description: XCHD exchanges the low-order nibble of the Accumulator (bits 3 through 0), generally representing a hexadecimal or BCD digit, with that of the internal RAM location indirectly addressed by the specified register. The high-order nibbles (bits 7-4) of each register are not affected. No flags are affected.

Example: R0 contains the address 20H. The Accumulator holds the value 36H (00110110B). Internal RAM location 20H holds the value 75H (01110101B). The following instruction, XCHD A,@R0 leaves RAM location 20H holding the value 76H (01110110B) and 35H (00110101B) in the Accumulator.

CPL A

Function: Complement Accumulator

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

Example: The Accumulator contains 5CH (01011100B).

The following instruction, CPL A leaves the Accumulator set to 0A3H (10100011B).

CPL bit

Function: Complement bit

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

Example: Port 1 has previously been written with 5BH (01011101B). The following instruction sequence, CPL P1.1CPL

P1.2 leaves the port set to 5BH (01011011B).

DEC byte

Function: Decrement

Description: DEC byte decrements the variable indicated by 1. An original value of 00H underflows to 0FFH. No flags are affected.

Example: Register 0 contains 7FH (01111111B). Internal RAM locations 7EH and 7FH contain 00H and 40H, respectively. The following instruction sequence,

DEC R0

DEC @R0

leaves register 0 set to 7EH and internal RAM locations 7EH and 7FH set to 0FFH and 3FH.

DEC A

DEC Rn

DEC direct

DEC @Ri

INC <byte>

Function: Increment

Description: INC increments the indicated variable by 1. An original value of 0FFH overflows to 00H. No flags are affected.

Example: Register 0 contains 7EH (011111110B). Internal RAM locations 7EH and 7FH contain 0FFH and 40H, respectively. The following instruction sequence,

INC R0

INC @R0 leaves register 0 set to 7FH and internal RAM locations 7EH and 7FH holding 00H and 41H, respectively.

INC A Operation: INC (A) (A) + 1

INC DPTR

Function: Increment Data Pointer

Description: INC DPTR increments the 16-bit data pointer by 1. A 16-bit increment (modulo 216) is performed, and an overflow of the low-order byte of the data pointer (DPL) from 0FFH to 00H increments the high-order byte (DPH). No flags are affected. This is the only 16-bit register which can be incremented.

Example: Registers DPH and DPL contain 12H and 0FEH, respectively. The following instruction sequence,

INC DPTR

INC DPTR

INC DPTR changes DPH and DPL to 13H and 01H.

NOP

Function: No Operation

Description: Execution continues at the following instruction. Other than the PC, no registers or flags are affected.

Logical instructions

ANL <dest-byte>,<src-byte>

Function: Logical-AND for byte variables

Description: ANL performs the bitwise logical-AND operation between the variables indicated and stores the results in the destination variable. No flags are affected. The two operands allow six addressing mode combinations. When the destination is the Accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the Accumulator or immediate data.

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

When the destination is a directly addressed byte, this instruction clears combinations of bits in any RAM location or hardware register. The mask byte determining the pattern of bits to be cleared would either be a constant contained in the instruction or a value computed in the Accumulator at run-time. The following instruction, ANL P1,#01110011B clears bits 7, 3, and 2 of output port 1.

Instructions	OpCode	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 A,R1	0x59	1	None
ANL A,R2	0x5A	1	None
ANL A,R3	0x5B	1	None
ANL A,R4	0x5C	1	None
ANL A,R5	0x5D	1	None

ANL A,R6	0x5E	1	None
ANL A,R7	0x5F	1	None
ANL C,*bit addr*	0x82	2	C
ANL C,/*bit addr*	0xB0	2	C

ANL A,Rn

Operation: AND the content of accumulator with the content of Register specified.

ANL A,@Ri

Operation: ANL

ANL direct,#data

Operation: ANL

(direct) #data ^ (direct)

ORL <dest-byte> <src-byte>

Function: Logical-OR for byte variables

Description: ORL performs the bitwise logical-OR operation between the indicated variables, storing the results in the destination byte. No flags are affected.

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

The instruction,ORL P1,#00110010B sets bits 5, 4, and 1 of output Port 1.

ORL A, Rn ; or the content of Accumulator and Register Rn and store the result in Accumulator

ORL A, direct ; or the content of Accumulator and the memory and store the result in Accumulator

ORL A, @Ri ; or the content of accumulator and the memory location whose address is specified in Ri

ORL C,<src-bit>

Function: Logical-OR for bit variables

Description: Set the carry flag if the Boolean value is a logical 1; leave the carry in its current state otherwise. A slash (/) preceding the operand in the assembly language indicates that the logical complement of the addressed bit is used as the source value, but the source bit itself is not affected. No other flags are affected.

Example:

ORL C, ACC.7 ;OR CARRY WITH THE ACC. BIT 7

ORL C, /OV ;OR CARRY WITH THE INVERSE OF OV.

SETB

Operation:	**SETB**
Function:	Set Bit
Syntax:	SETB *bitaddr*

Description: Sets the specified bit.

XRL <dest-byte>,<src-byte>

Function: Logical Exclusive-OR for byte variables

Description: XRL performs the bitwise logical Exclusive-OR operation between the indicated variables, storing the results in the destination. No flags are affected. The two operands allow six addressing mode combinations. When the destination is the Accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the Accumulator or immediate data.

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

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
XRL A,R1	0x69	1	None
XRL A,R2	0x6A	1	None
XRL A,R3	0x6B	1	None
XRL A,R4	0x6C	1	None
XRL A,R5	0x6D	1	None
XRL A,R6	0x6E	1	None
XRL A,R7	0x6F	1	None

Rotate Instructions

RL A

Function: Rotate Accumulator Left

Description: The eight bits in the Accumulator are rotated one bit to the left. Bit 7 is rotated into the bit 0 position. No flags are affected.

Example: The Accumulator holds the value 0C5H (11000101B). The following instruction,

RL A leaves the Accumulator holding the value 8BH (10001011B) with the carry unaffected.

RLC A

Function: Rotate Accumulator Left through the Carry flag

Description: The eight bits in the Accumulator and the carry flag are together rotated one bit to the left. Bit 7 moves into the carry flag; the original state of the carry flag moves into the bit 0 position. No other flags are affected.

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

RRC A

Function: Rotate Accumulator Right through Carry flag

Description: The eight bits in the Accumulator and the carry flag are together rotated one bit to the right. Bit 0 moves into the carry flag; the original value of the carry flag moves into the bit 7 position. No other flags are affected.

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

Jump Instructions

Unconditional Branch Instructions: It is a jump in which control is transferred unconditionally to target instruction. Basically there are three unconditional jump instructions 1) AJMP 2) LJMP 3)SJMP

Operation:	**AJMP**		
Function:	Absolute Jump Within 2K Block		
Syntax:	AJMP *code address*		
Instructions	**OpCode**	**Bytes**	**Flags**
AJMP *page0*	0x01	2	None
AJMP *page1*	0x21	2	None
AJMP *page2*	0x41	2	None
AJMP *page3*	0x61	2	None
AJMP *page4*	0x81	2	None
AJMP *page5*	0xA1	2	None
AJMP *page6*	0xC1	2	None
AJMP *page7*	0xE1	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 unchanged. 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(Long jump)

Operation: **LJMP**
Function: Long Jump
Syntax: LJMP *code address.*

Description: LJMP jumps unconditionally to the specified *code address*. It can jump in any memory from 0000H to FFFFH.

SJMP

SJMP

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

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

Conditional Jump Instructions

These instructions jumps to the specified location after checking a condition.

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

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

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

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

Operation: **JNB**
Function: Jump if Bit Not Set
Syntax: JNB *bit addr,reladdr*

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

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

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

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

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

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

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

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

Instructions	**OpCode**	**Bytes**	**Flags**
DJNZ *iram addr*,*reladdr*	0xD5	3	None
DJNZ R0,*reladdr*	0xD8	2	None
DJNZ R1,*reladdr*	0xD9	2	None
DJNZ R2,*reladdr*	0xDA	2	None
DJNZ R3,*reladdr*	0xDB	2	None
DJNZ R4,*reladdr*	0xDC	2	None
DJNZ R5,*reladdr*	0xDD	2	None
DJNZ R6,*reladdr*	0xDE	2	None
DJNZ R7,*reladdr*	0xDF	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.

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
CJNE @R1,*#data,reladdr*	0xB7	3	C
CJNE R0,*#data,reladdr*	0xB8	3	C
CJNE R1,*#data,reladdr*	0xB9	3	C
CJNE R2,*#data,reladdr*	0xBA	3	C
CJNE R3,*#data,reladdr*	0xBB	3	C
CJNE R4,*#data,reladdr*	0xBC	3	C
CJNE R5,*#data,reladdr*	0xBD	3	C
CJNE R6,*#data,reladdr*	0xBE	3	C
CJNE R7,*#data,reladdr*	0xBF	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.

CALL INSTRUCTIONS

Another control transfer instruction is the CALL instruction, which is used to call a 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.

In the 8051 there are two instructions for call: LCALL (long call) and ACALL (absolute call). Deciding which one to use depends on the target address. Each instruction is explained next.

LCALL (long call)

In this 3-byte instruction, the first byte is the opcode and the second and third bytes are used for the address of the target subroutine. Therefore, LCALL can be used to call subroutines located anywhere within the 64K-byte address space of the 8051. To make sure that after execution of the called subroutine the 8051 knows where to come back to, the processor automatically saves

on the stack the address of the instruction immediately below the LCALL. When a subroutine is called, control is transferred to that subroutine, and the processor saves the PC (program counter) on the stack and begins to fetch instructions from the new location. After finishing execution of the subroutine, the instruction RET (return) transfers control back to the caller. Every subroutine needs RET as the last instruction.

ACALL (absolute call)

ACALL is a 2-byte instruction in contrast to LCALL. Since ACALL is a 2-byte instruction, the target address of the subroutine must be within 2K bytes because only 11 bits of the 2 bytes are used for the address. There is no difference between ACALL and LCALL in terms of saving the program counter on the stack or the function of the RET instruction. The only difference is that the target address for LCALL can be anywhere within the 64K-byte address space of the 8051 while the target address of ACALL must be within a 2K-byte range.

RET

Function: Return From Subroutine

Syntax: RET

Description: RET is used to return from a subroutine previously called by LCALL or ACALL. Program execution continues at the address that is calculated by popping the topmost 2 bytes off the stack. The most-significant-byte is popped off the stack first, followed by the least-significant-byte.

RETI

Operation: **RETI**

Function: Return From Interrupt

Syntax: RETI

Description: RETI is used to return from an interrupt service routine. RETI first enables interrupts of equal and lower priorities to the interrupt that is terminating. Program execution continues at the address that is calculated by popping the topmost 2 bytes off the stack. The most-significant-byte is popped off the stack first, followed by the least-significant-byte.