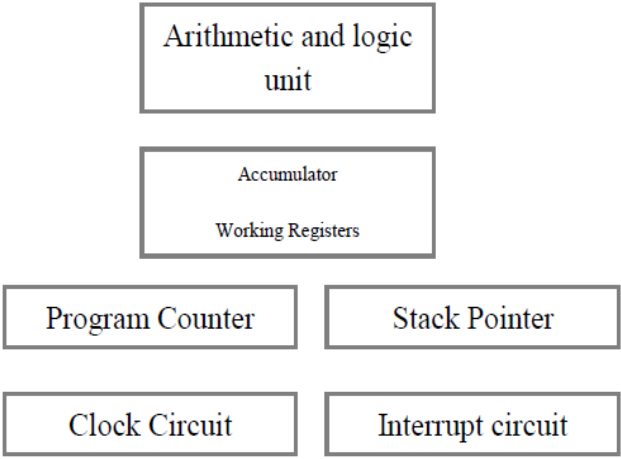
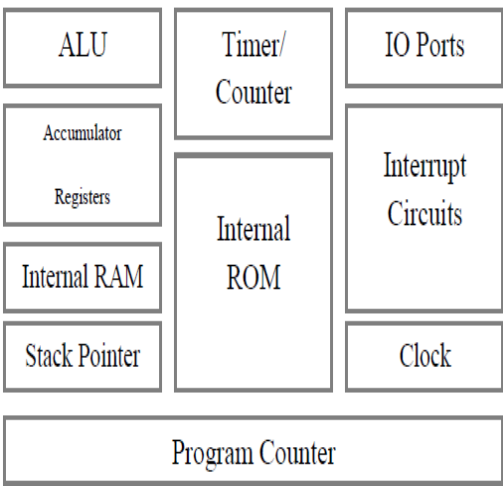


## Unit 1: 8051 MICROCONTROLLER ARCHITECTURE:

### Introduction to Microprocessors and Microcontrollers

#### Comparison of Microprocessor and Microcontrollers

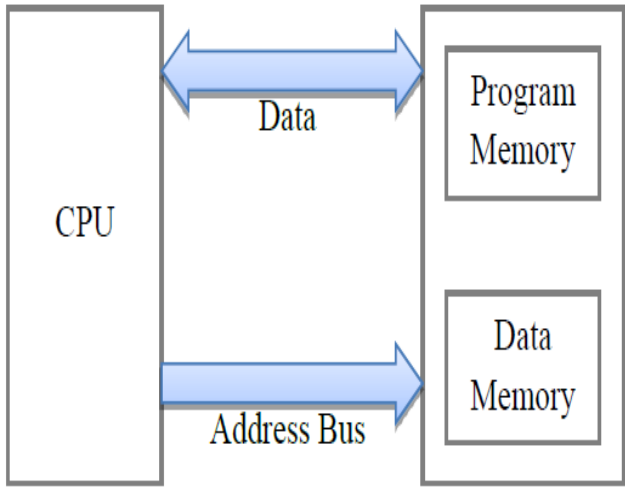
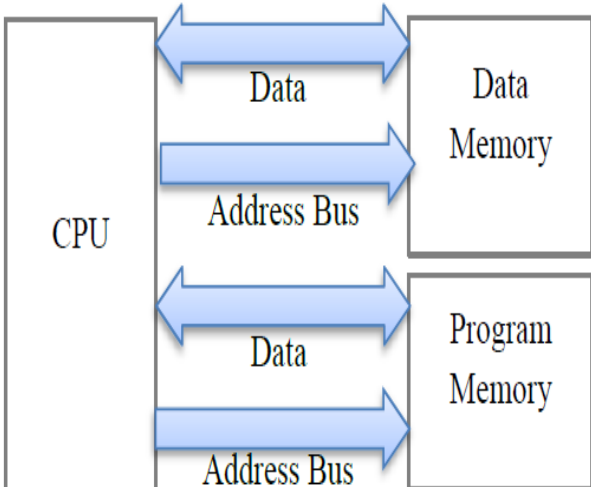
Microprocessor	Microcontroller
	
Block diagram of microprocessor	Block diagram of microcontroller
Microprocessor contains ALU, General purpose registers, stack pointer, program counter, clock timing circuit, interrupt circuit	Microcontroller contains the circuitry of microprocessor, and in addition it has built in ROM, RAM, I/O Devices, Timers/Counters etc.
It has many instructions to move data between memory and CPU	It has few instructions to move data between memory and CPU
Few bit handling instruction	It has many bit handling instructions
Less number of pins are multifunctional	More number of pins are multifunctional
Single memory map for data and code (program)	Separate memory map for data and code (program)
Access time for memory and IO are more	Less access time for built in memory and IO.
Microprocessor based system requires additional hardware	It requires less additional hardwares
More flexible in the design point of view	Less flexible since the additional circuits which is residing inside the microcontroller is fixed for a particular microcontroller
Large number of instructions with flexible addressing modes	Limited number of instructions with few addressing modes
Example: 8086	Example: 8051

## RISC AND CISC CPU ARCHITECTURES

Microcontrollers with small instruction set are called reduced instruction set computer (RISC) machines and those with complex instruction set are called complex instruction set computer (CISC). Intel 8051 is an example of CISC machine whereas microchip PIC 18F87X is an example of RISC machine.

RISC	CISC
Instruction takes one or two cycles	Instruction takes multiple cycles
Only load/store instructions are used to access memory	In additions to load and store instructions, memory access is possible with other instructions also.
Instructions executed by hardware	Instructions executed by the micro program
Fixed format instruction	Variable format instructions
Few addressing modes	Many addressing modes
Few instructions	Complex instruction set
Most of the have multiple register banks	Single register bank
Highly pipelined	Less pipelined
Complexity is in the compiler	Complexity in the microprogram

## HARVARD & VON- NEUMANN CPU ARCHITECTURE

Von-Neumann (Princeton architecture)	Harvard architecture
	
It uses single memory space for both instructions and data	It has separate program memory and data memory
It is not possible to fetch instruction code and data	Instruction code and data can be fetched simultaneously

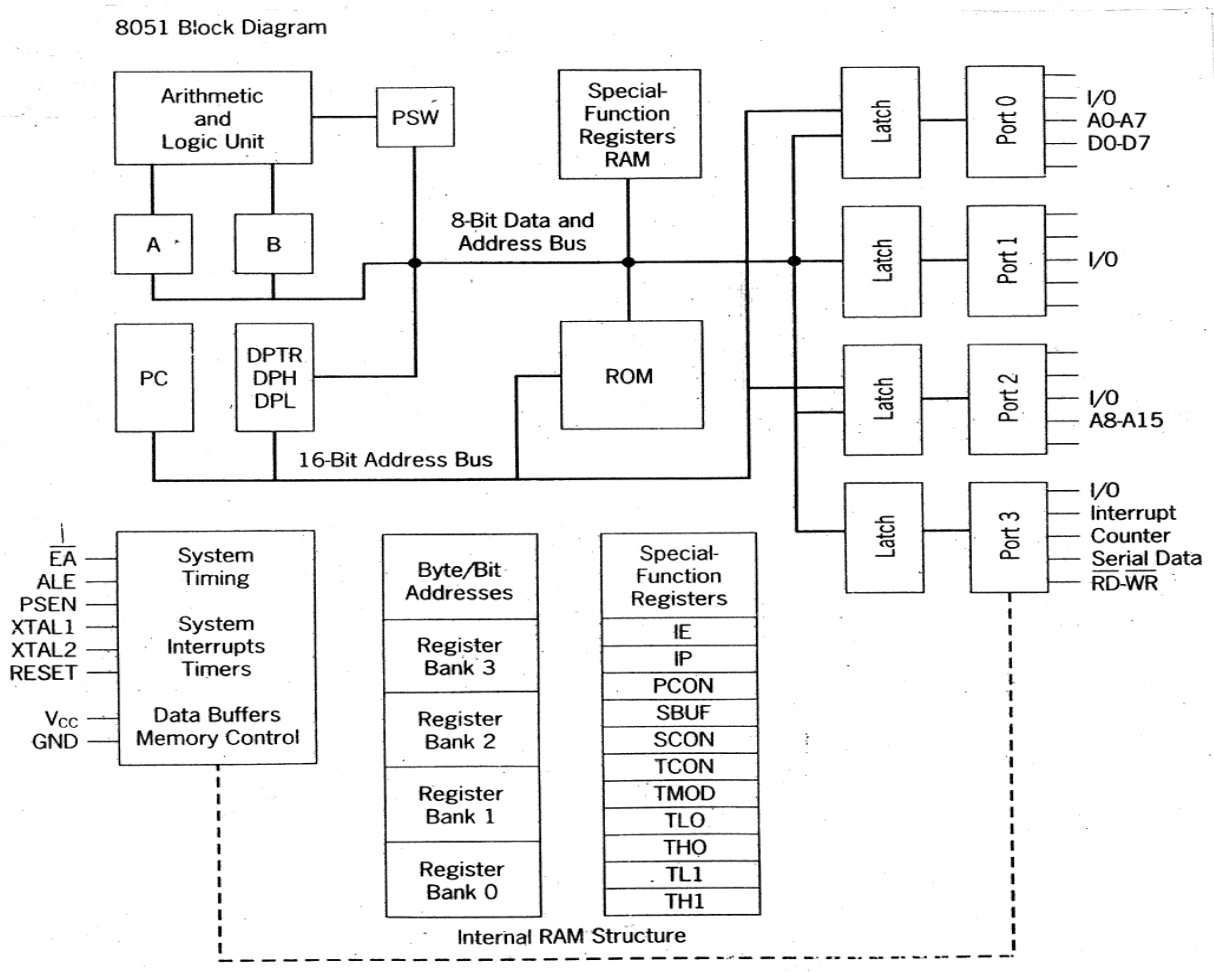
Execution of instruction takes more machine cycle	Execution of instruction takes less machine cycle
Uses CISC architecture	Uses RISC architecture
Instruction pre-fetching is a main feature	Instruction parallelism is a main feature
Also known as control flow or control driven computers	Also known as data flow or data driven computers
Simplifies the chip design because of single memory space	Chip design is complex due to separate memory space
Eg. 8085, 8086, MC6800	Eg. General purpose microcontrollers, special DSP chips etc.

### **COMPUTER SOFTWARE:**

A set of instructions written in a specific sequence for the computer to solve a specific task is called a program and software is a collection of such programs. The program stored in the computer memory in the form of binary numbers is called machine instructions. The machine language program is called object code. An assembly language is a mnemonic representation of machine language. Machine language and assembly language are low level languages and are processor specific.

The assembly language program the programmer enters is called source code. The source code (assembly language) is translated to object code (machine language) using assembler. Programs can be written in high level languages such as C, C++ etc. High level language will be converted to machine language using compiler or interpreter. Compiler reads the entire program and translate into the object code and then it is executed by the processor. Interpreter takes one statement of the high level language as input and translate it into object code and then executes.

## THE 8051 ARCHITECTURE



**Fig 1. 8051 Microcontroller Architecture**

**Registers:** In the CPU, registers are used to store information temporarily. It can be byte of data or an address. The vast majority of registers are 8-bit in size. Commonly used registers are A(accumulator), B, R0, R1, R2, R3, R4, R5, R6, R7, DPTR(data pointer) and PC(program counter).

### Salient features of 8051 microcontroller:

- Eight bit CPU
- On chip clock oscillator
- 4Kbytes of internal program memory (code memory) [ROM]
- 128 bytes of internal data memory [RAM]
- 64 Kbytes of external program memory address space.

- 64 Kbytes of external data memory address space.
- 32 bi directional I/O lines (can be used as four 8 bit ports or 32 individually addressable I/O lines)
- Two 16 Bit Timer/Counter :T0, T1
- Full Duplex serial data receiver/transmitter
- Four Register banks with 8 registers in each bank.
- Sixteen bit Program counter (PC) and a data pointer (DPTR)
- 8 Bit Program Status Word (PSW)
- 8 Bit Stack Pointer
- Five vector interrupt structure (RESET not considered as an interrupt.)
- 8051 CPU consists of 8 bit ALU with associated registers like accumulator 'A' , B register, PSW, SP, 16 bit program counter, stack pointer.
- ALU can perform arithmetic and logic functions on 8 bit variables.
- 8051 has 128 bytes of internal RAM which is divided into
  - Working registers [00 – 1F]
  - Bit addressable memory area [20 – 2F]
  - General purpose memory area (Scratch pad memory) [30-7F]
- 8051 has 4 K Bytes of internal ROM. The address space is from 0000 to 0FFFh. If the program size is more than 4 K Bytes 8051 will fetch the code automatically from external memory.
- Accumulator is an 8 bit register widely used for all arithmetic and logical operations. Accumulator is also used to transfer data between external memory. B register is used along with Accumulator for multiplication and division. A and B registers together is also called MATH registers.
- PSW (Program Status Word). This is an 8 bit register shown in fig 2. which contains the arithmetic status of ALU and the bank select bits of register banks.



**Fig 2. Bits of Program Status Word (PSW)**

CY - carry flag

AC - auxiliary carry flag

F0 - available to the user for general purpose

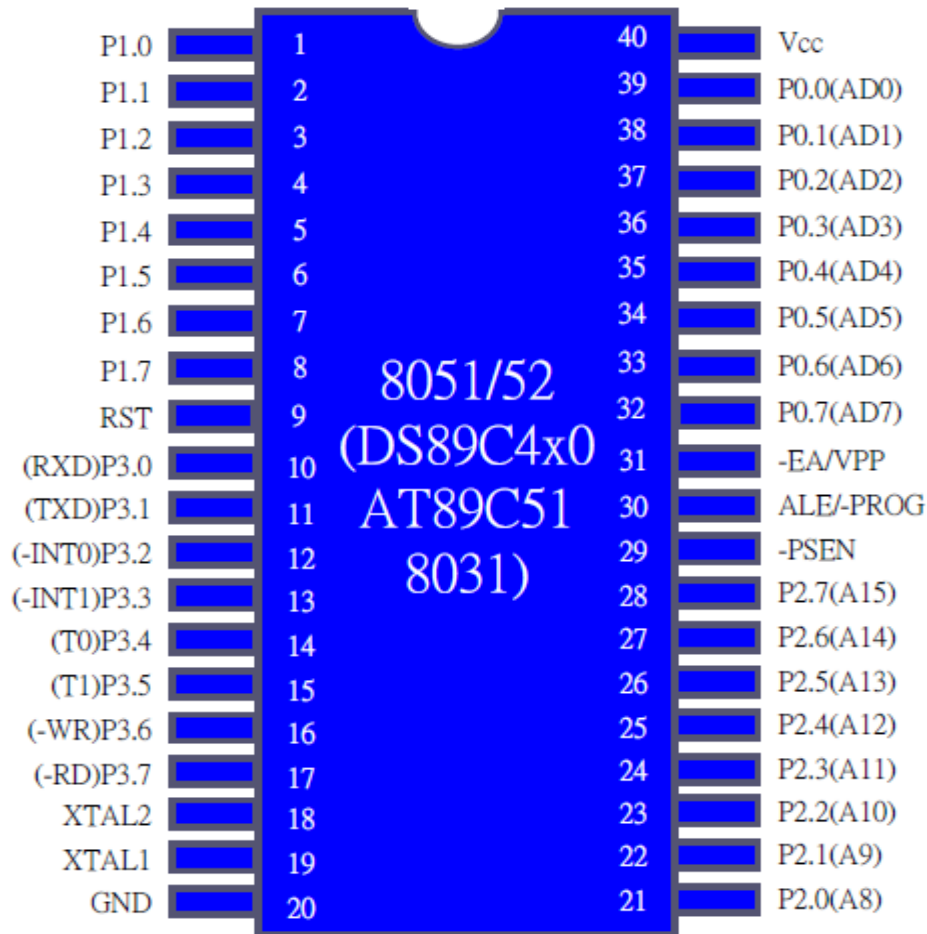
RS1, RS0 - register bank select bits

OV - overflow flag

P – Parity flag

- Stack Pointer (SP) – it contains the address of the data item on the top of the stack. Stack may reside anywhere on the internal RAM. On reset, SP is initialized to 07 so that the default stack will start from address 08 onwards.
- Data Pointer (DPTR) – DPH (Data pointer higher byte), DPL (Data pointer lower byte). This is a 16 bit register which is used to furnish address information for internal and external program memory and for external data memory.
- Program Counter (PC) – 16 bit PC contains the address of next instruction to be executed. On reset PC will set to 0000. After fetching every instruction PC will increment by one.

## PIN DIAGRAM



**Fig 3. Pin diagram of 8051 Microcontroller**

### Pin Description:

<i>Pin Number</i>	<b>Description of each pin</b>
<i>Pins 1-8</i>	<b>PORT 1.</b> Each of these pins can be configured as an input or an output.
<i>Pin 9</i>	<b>RESET.</b> A logic one on this pin disables the microcontroller and clears the contents of most registers. In other words, the positive voltage on this pin resets the microcontroller. By applying logic zero to this pin, the program starts execution from the beginning.
<i>Pins 10-17</i>	<b>PORT 3.</b> Similar to port 1, each of these pins can serve as general input or output. Besides, all of them have alternative functions
<i>Pin 10</i>	<b>RXD.</b> Serial asynchronous communication input or Serial synchronous communication output.
<i>Pin 11</i>	<b>TXD.</b> Serial asynchronous communication output or Serial synchronous communication

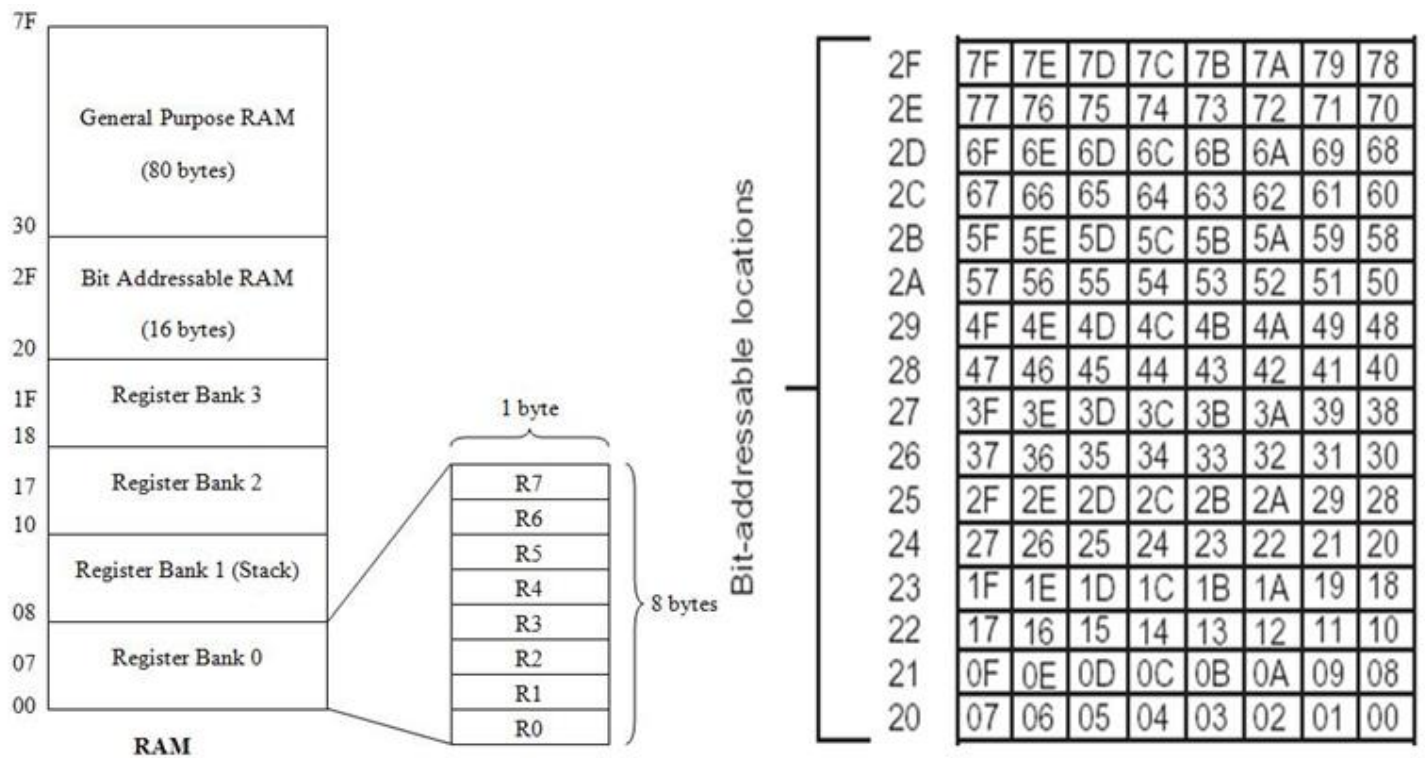
	clock output.
<b>Pin 12</b>	<b>INT0.</b> External Interrupt 0 input
<b>Pin 13</b>	<b>INT1.</b> External Interrupt 1 input
<b>Pin 14</b>	<b>T0. Counter 0 clock input</b>
<b>Pin 15</b>	<b>T1. Counter 1 clock input</b>
<b>Pin 16</b>	<b>WR. Write to external (additional) RAM</b>
<b>Pin 17</b>	<b>RD. Read from external RAM</b>
<b>Pin 18, 19</b>	<b>XTAL2, XTAL1.</b> Internal oscillator input and output. A quartz crystal which specifies operating frequency is usually connected to these pins.
<b>Pin 20</b>	<b>GND. Ground.</b>
<b>Pin 21-28</b>	<b>Port 2.</b> If there is no intention to use external memory then these port pins are configured as general inputs/outputs. In case external memory is used, the higher address byte, i.e. addresses A8-A15 will appear on this port. Even though memory with capacity of 64Kb is not used, which means that not all eight port bits are used for its addressing, the rest of them are not available as inputs/outputs.
<b>Pin 29</b>	<b>PSEN.</b> If external ROM is used for storing program then a logic zero (0) appears on it every time the microcontroller reads a byte from memory.
<b>Pin 30</b>	<b>ALE.</b> Prior to reading from external memory, the microcontroller puts the lower address byte (A0-A7) on P0 and activates the ALE output. After receiving signal from the ALE pin, the external latch latches the state of P0 and uses it as a memory chip address. Immediately after that, the ALE pin is returned its previous logic state and P0 is now used as a Data Bus.
<b>Pin 31</b>	<b>EA.</b> By applying logic zero to this pin, P2 and P3 are used for data and address transmission with no regard to whether there is internal memory or not. It means that even there is a program written to the microcontroller, it will not be executed. Instead, the program written to external ROM will be executed. By applying logic one to the EA pin, the microcontroller will use both memories, first internal then external (if exists).
<b>Pin 32-39</b>	<b>PORT 0.</b> Similar to P2, if external memory is not used, these pins can be used as general inputs/outputs. Otherwise, P0 is configured as address output (A0-A7) when the ALE pin is driven high (1) or as data output (Data Bus) when the ALE pin is driven low (0).
<b>Pin 40</b>	<b>VCC.</b> +5V power supply.



## MEMORY ORGANIZATION

The 8051 microcontroller has three types of internal memory. Internal RAM is of 128 bytes in size (fig 4.), Special Function Register (SFR) is 128 bytes in size and internal ROM is of 4 Kbytes in size.

### Internal RAM organization



**Fig 4. Internal RAM Memory (128 bytes)**

**Register Banks: 00h to 1Fh.** The 8051 has 4 register banks Register Bank 0, Register Bank 1, Register Bank 2 and Register Bank 3 as shown in fig 5. Each register bank has 8 general-purpose registers R0 through R7 (R0, R1, R2, R3, R4, R5, R6, and R7). Selection of register bank can be done through RS1,RS0 bits of PSW. On reset, the default Register Bank 0 will be selected.

R7	1F	BANK 3
R6	1E	
R5	1D	
R4	1C	
R3	1B	
R2	1A	
R1	19	
R0	18	
R7	17	BANK 2
R6	16	
R5	15	
R4	14	
R3	13	
R2	12	
R1	11	
R0	10	
R7	0F	BANK 1
R6	0E	
R5	0D	
R4	0C	
R3	0B	
R2	0A	
R1	09	
R0	08	
R7	07	BANK 0
R6	06	
R5	05	
R4	04	
R3	03	
R2	02	
R1	01	
R0	00	

**Fig 5. Register Banks and their Registers.**

**Bit Addressable RAM: 20h to 2Fh.** The 8051 supports a special feature which allows access to bit variables. This is where individual memory bits in Internal RAM can be set or cleared. In all there are 128 bits numbered 00h to 7Fh (bit address) as shown in fig 6. Being bit variables any one variable can have a value 0 or 1. A bit variable can be set with a command such as SETB and cleared with a command such as CLR.

Example instructions are:

*SETB 25h; sets the bit 25h (becomes 1)*

*CLR 25h; clears bit 25h (becomes 0)*

*Note, bit 25h is actually bit 5 of Internal RAM location 24h.*

The Bit Addressable area of the RAM is just 16 bytes of Internal RAM located between 20h and 2Fh (byte address).

Bit-addressable locations	2F	7F	7E	7D	7C	7B	7A	79	78
	2E	77	76	75	74	73	72	71	70
	2D	6F	6E	6D	6C	6B	6A	69	68
	2C	67	66	65	64	63	62	61	60
	2B	5F	5E	5D	5C	5B	5A	59	58
	2A	57	56	55	54	53	52	51	50
	29	4F	4E	4D	4C	4B	4A	49	48
	28	47	46	45	44	43	42	41	40
	27	3F	3E	3D	3C	3B	3A	39	38
	26	37	36	35	34	33	32	31	30
	25	2F	2E	2D	2C	2B	2A	29	28
	24	27	26	25	24	23	22	21	20
	23	1F	1E	1D	1C	1B	1A	19	18
	22	17	16	15	14	13	12	11	10
	21	0F	0E	0D	0C	0B	0A	09	08
	20	07	06	05	04	03	02	01	00

**Fig 6. Bit Addressable Memory**

**General Purpose RAM:** 30h to 7Fh. Even if 80 bytes of Internal RAM memory are available for general-purpose data storage, user should take care while using the memory location from 00 -2Fh since these locations are also the default register space, stack space, and bit addressable space. It is a good practice to use general purpose memory from 30 – 7Fh. The general purpose RAM can be accessed using direct or indirect addressing modes.

### **Special Function Register (SFR):**

The operations that do not use 128 byte internal RAM locations from 00H to 7FH, are grouped into specific internal registers called as Special Function Register as shown in fig. 7. It extends from location 80H to FFH. Few locations are only byte accessible, for example DPL register (82H). Whereas few registers are bit accessible, for example Accumulator register (E0H).

Bit accessible SFRs have address for each bit. The byte address of such registers is same as address of LSB of the same register. For example, accumulators byte address and bit address of LSB is E0H.

# Special Function Registers

Byte address	Bit address									Byte address	Bit address								
98	9F	9E	9D	9C	9B	9A	99	98	SCON	FF									
										F0	F7	F6	F5	F4	F3	F2	F1	F0	B
90	97	96	95	94	93	92	91	90	P1	E0	E7	E6	E5	E4	E3	E2	E1	E0	ACC
8D	not bit addressable								TH1										
8C	not bit addressable								TH0	D0	D7	D6	D5	D4	D3	D2	-	D0	PSW
8B	not bit addressable								TL1										
8A	not bit addressable								TL0	B8	-	-	-	BC	BB	BA	B9	B8	IP
89	not bit addressable								TMOD										
88	8F	8E	8D	8C	8B	8A	89	88	TCON	B0	B7	B6	B5	B4	B3	B2	B1	B0	P3
87	not bit addressable								PCON										
										A8	AF	-	-	AC	AB	AA	A9	A8	IE
83	not bit addressable								DPH										
82	not bit addressable								DPL	A0	A7	A6	A5	A4	A3	A2	A1	A0	P2
81	not bit addressable								SP										
80	87	86	85	84	83	82	81	80	P0	99	not bit addressable								SBUF

Fig 7. Special Function Registers (SFR)

## STACK MEMORY

A stack is a last in first out memory. In 8051 internal RAM space can be used as stack. The address of the stack is contained in a register called stack pointer. Instructions PUSH and POP are used for stack operations. When a data is to be placed on the stack, the stack pointer increments before storing the data on the stack so that the stack grows up as data is stored (pre-increment). As the data is retrieved from the stack the byte is read from the stack, and then SP decrements to point the next available byte of stored data (post decrement). The stack pointer is set to 07 when the 8051 resets. So that default stack memory starts from address location 08 onwards (to avoid overwriting the default register bank ie., bank 0).

The storing of a CPU register in the stack is called a PUSH. SP is pointing to the last used location of the stack. As we push data onto the stack, the SP is incremented by one. Loading the contents of the stack back into a CPU register is called a POP. With every pop, the top byte of the stack is copied to the register specified by the instruction and the stack pointer is decremented by one.

Example:

```
MOV R6, #25H
MOV R1, #12H
MOV R4, #0F3H
PUSH 6
PUSH 1
PUSH 4
```

	After PUSH 6	After PUSH 1	After PUSH 4																																
<table><tr><td>0B</td><td></td></tr><tr><td>0A</td><td></td></tr><tr><td>09</td><td></td></tr><tr><td>08</td><td></td></tr></table>	0B		0A		09		08		<table><tr><td>0B</td><td></td></tr><tr><td>0A</td><td></td></tr><tr><td>09</td><td></td></tr><tr><td>08</td><td>25</td></tr></table>	0B		0A		09		08	25	<table><tr><td>0B</td><td></td></tr><tr><td>0A</td><td></td></tr><tr><td>09</td><td>12</td></tr><tr><td>08</td><td>25</td></tr></table>	0B		0A		09	12	08	25	<table><tr><td>0B</td><td></td></tr><tr><td>0A</td><td>F3</td></tr><tr><td>09</td><td>12</td></tr><tr><td>08</td><td>25</td></tr></table>	0B		0A	F3	09	12	08	25
0B																																			
0A																																			
09																																			
08																																			
0B																																			
0A																																			
09																																			
08	25																																		
0B																																			
0A																																			
09	12																																		
08	25																																		
0B																																			
0A	F3																																		
09	12																																		
08	25																																		
Start SP = 07	SP = 08	SP = 09	SP = 0A																																

POP 3 ; POP stack into R3

POP 5 ; POP stack into R5

POP 2 ; POP stack into R2

	After POP 3	After POP 5	After POP 2																																
<table><tr><td>0B</td><td>54</td></tr><tr><td>0A</td><td>F9</td></tr><tr><td>09</td><td>76</td></tr><tr><td>08</td><td>6C</td></tr></table>	0B	54	0A	F9	09	76	08	6C	<table><tr><td>0B</td><td></td></tr><tr><td>0A</td><td>F9</td></tr><tr><td>09</td><td>76</td></tr><tr><td>08</td><td>6C</td></tr></table>	0B		0A	F9	09	76	08	6C	<table><tr><td>0B</td><td></td></tr><tr><td>0A</td><td></td></tr><tr><td>09</td><td>76</td></tr><tr><td>08</td><td>6C</td></tr></table>	0B		0A		09	76	08	6C	<table><tr><td>0B</td><td></td></tr><tr><td>0A</td><td></td></tr><tr><td>09</td><td></td></tr><tr><td>08</td><td>6C</td></tr></table>	0B		0A		09		08	6C
0B	54																																		
0A	F9																																		
09	76																																		
08	6C																																		
0B																																			
0A	F9																																		
09	76																																		
08	6C																																		
0B																																			
0A																																			
09	76																																		
08	6C																																		
0B																																			
0A																																			
09																																			
08	6C																																		
Start SP = 0B	SP = 0A	SP = 09	SP = 08																																

The reason of incrementing SP after push is to make sure that the stack is growing toward RAM location 7FH, from lower to upper addresses and to ensure that the stack will not reach the bottom of RAM and consequently run out of stack space. If the stack pointer were decremented after push, RAM locations 7, 6, 5, etc. which belongs to R7 to R0 of bank 0, the default register bank will be over written.

The stack memory can be relocated to another location by changing the content of stack pointer. For example, MOV SP, #5FH, will start the stack memory from 60H instead of default 08H.

## **Data Types and Assembler Directives:**

**Data Type:** The 8051 microcontroller has only one data type. It is 8 bits. If data is bigger than 8 bit, it is split into sets of 8 bits.

### **Assembler Directives:**

**1. 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 format. For decimal, “D” after the decimal number is optional but “B” for binary numbers and “H” for Hexadecimal numbers is required. Regardless of which is used, assembler will convert the numbers into hex. The ASCII characters are placed inside the quotation marks, either single or double quote. The assembler will assign the ASCII code for the numbers or characters automatically. It is the only directive to define ASCII strings larger than two characters.

Example:

	ORG 500H	
DATA1:	DB 28	; DECIMAL
DATA2:	DB 0000101B	; BINARY
DATA3:	DB 39H	; HEXADECIMAL
DATA4:	DB ‘2591’	; ASCII NUMBERS
DATA5:	DB “DSCE”	; ASCII CHARACTERS
	END	

### **2. ORG (origin):**

The ORG directive is used to indicate the beginning of the address from where the programs will be stored in the internal ROM memory. The number that comes after ORG can be either in hex and decimal. If the number is not followed by H, it is decimal and the assembler will convert it to hex.

Example:

	ORG 0000H
--	-----------

### 3. 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.

```
Example:          COUNT EQU 25
                  MOV R3, #COUNT
```

**4. END:**

This indicates to the assembler the end of the source (asm) file. The END directive is the last line of an 8051 program. Anything after the END directive is ignored by the assembler.

**Program Status Word (PSW) Register:**

MSB						LSB	
CY	AC	F0	RS1	RS0	OV	-	P

CY	PSW.7	carry flag
AC	PSW.6	auxiliary carry flag
F0	PSW.5	available to the user for general purpose
RS1	PSW.4	register bank select bit 1.
RS0	PSW.3	register bank select bit 0.
OV	PSW.2	overflow flag
-	PSW.1	user-definable bit
P	PSW.0	Parity flag

## **UNIT – 2: 8051 INSTRUCTION SET & ASSEMBLY PROGRAMMING**

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**

Various methods of accessing the data are called addressing modes.

8051 addressing modes are classified as follows.

1. Immediate addressing.
2. Register addressing.
3. Direct addressing.
4. Indirect addressing.
5. Relative addressing.
6. Absolute addressing.
7. Long addressing.



8. Indexed addressing.
9. Bit inherent addressing.
10. Bit direct addressing.

### ***1. Immediate addressing.***

In this addressing mode the data is provided as a part of instruction itself. In other words data immediately follows the instruction.

Eg.    MOV A, #30H  
      ADD A, #83 ; # Symbol indicates the data is immediate.

Data can only be a source, it can not be destination.

Eg.    Mov #30h, A is an invalid instruction.

### ***2. Register addressing.***

In this addressing mode the registers are part of the instruction. Source or destination can be any register from R0 to R7. But the other register has to be either A or B register. Both source and destination can not be general purpose registers (R0 to R7)

Eg.    MOV A,R0  
      ADD A,R6

R0 – R7 will be selected from the current selection of register bank. The default register bank will be bank 0.

MOV R0, R6 is a invalid register.

### ***3. Direct addressing***

In this addressing mode, address is part of the instruction. There are two ways to access the internal memory. Using direct address and indirect address. Using direct addressing mode we can not only address the internal memory but SFRs also. In direct addressing, an 8 bit internal data memory address is specified as part of the instruction and hence, it can specify the address only in the range of 00H to FFH. In this addressing mode, data is obtained directly from the memory.

Eg.    MOV A,60h  
      ADD A,30h

#### **4. Indirect addressing**

The indirect addressing mode uses a register to hold the actual address that will be used in data movement. Registers R0 and R1 and DPTR are the only registers that can be used as data pointers. Indirect addressing cannot be used to refer to SFR registers. Both R0 and R1 can hold 8 bit address and DPTR can hold 16 bit address.

Eg.    MOV A,@R0  
       ADD A,@R1  
       MOVX A,@DPTR

This addressing mode is applicable for both source and destination. MOV @R0, A is a valid instruction.

#### **5. Indexed addressing.**

In indexed addressing, either the program counter (PC), or the data pointer (DTPR)—is used to hold the base address, and the A is used to hold the offset address. Adding the value of the base address to the value of the offset address forms the effective address. Indexed addressing is used with JMP or MOVC instructions. Look up tables are easily implemented with the help of index addressing.

Eg.    MOVC A, @A+DPTR *// copies the contents of memory location pointed by the sum of the accumulator A and the DPTR into accumulator A.*

       MOVC A, @A+PC *// copies the contents of memory location pointed by the sum of the accumulator A and the program counter into accumulator A.*

#### **6. Relative Addressing.**

Relative addressing is used only with conditional jump instructions. The relative address, (offset), is an 8 bit signed number, which is automatically added to the PC to make the address of the next instruction. The 8 bit signed offset value gives an address range of +127 to —128 locations. The jump destination is usually specified using a label and the assembler calculates the jump offset accordingly. The advantage of relative addressing is that the program code is easy to relocate and the address is relative to position in the memory.

Eg.    SJMP LOOP1  
       JC BACK

### **7. Absolute addressing**

Absolute addressing is used only by the AJMP (Absolute Jump) and ACALL (Absolute Call) instructions. These are 2 bytes instructions. The absolute addressing mode specifies the lowest 11 bit of the memory address as part of the instruction. The upper 5 bit of the destination address are the upper 5 bit of the current program counter. Hence, absolute addressing allows branching only within the current 2 Kbyte page ( $2^{11}=2\text{Kbyte}$ ) of the program memory. Upper 5 bits specifies the page number. Total of  $2^5=32$  pages are available in the memory. Each page size is 2 Kbytes. Total size of internal ROM is 64 Kbytes.

Eg.   AJMP LOOP1  
      ACALL LOOP2

### **8. Long Addressing**

The long addressing mode is used with the instructions LJMP and LCALL. These are 3 byte instructions. The address specifies a full 16 bit destination address so that a jump or a call can be made to a location within a 64 Kbyte code memory space.

Eg.   LJMP FINISH  
      LCALL DELAY

### **9. Bit Inherent Addressing**

In this addressing, the address of the flag which contains the operand, is implied in the opcode of the instruction.

Eg.   CLR C ; *Clears the carry flag to 0*

### **10. Bit Direct Addressing**

In this addressing mode the direct address of the bit is specified in the instruction. The RAM space 20H to 2FH which are bit accessible and most of the special function registers are bit addressable. Bit address values are between 00H to 7FH.

Eg.   CLR 07h ; *Clears the bit 7 of 20h RAM space*  
      SETB 07H ; *Sets the bit 7 of 20H RAM space.*

## **INSTRUCTION SET**

Various instruction set of 8051 microcontrollers are:

1. Data transfer instructions
2. Arithmetic instructions
3. Logical instructions
4. Branch instructions
5. Subroutine instructions
6. Bit manipulation instructions

### **1. Data transfer instructions:**

These instructions are used to transfer data within internal RAM memory.

**Syntax: mov destination, source.**

It can be used to transfer both bit of data as well as byte of data.

a. Move the contents of a register Rn to A

- i. MOV A,R2
- ii. MOV A,R7

b. Move the contents of a register A to Rn

- i. MOV R4,A
- ii. MOV R1,A

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

- i. MOV A, #45H
- ii. MOV R6, #51H
- iii. MOV 30H, #44H
- iv. MOV @R0, #0E8H
- v. MOV DPTR, #0F5A2H
- vi. MOV DPTR, #5467H

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

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

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

- i. MOV R3, 65H
- ii. MOV 45H, R2

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

- i. MOV 47H, 65H
- ii. MOV 45H, @R0

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

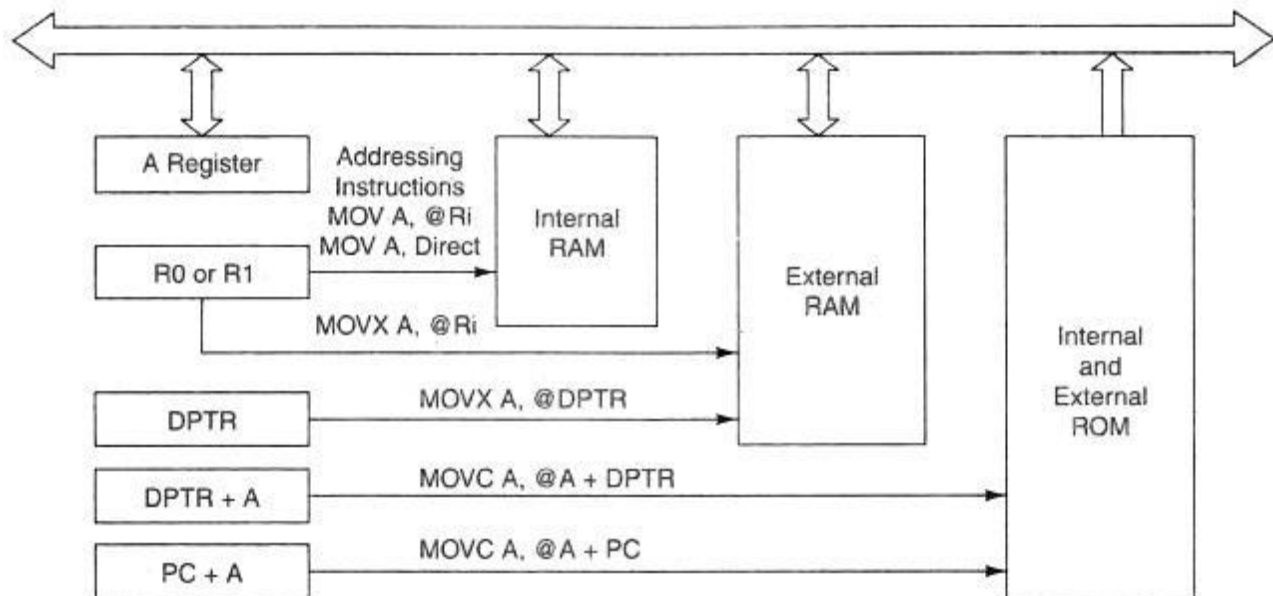
- i. MOVX A, @R1
- ii. MOVX @R0, A
- iii. MOVX A, @DPTR
- iv. MOVX @DPTR, A

h. Move the contents of program memory to A

- i. MOVC A, @A+PC
- ii. MOVC A, @A+DPTR

i. Move bit of data

- i. mov c, #1b ; destination is carry bit and source is 1bit data.



#### j. Push and Pop instructions

[SP]=07 //CONTENT OF SP IS 07 (DEFAULT VALUE)

MOV R6, #25H [R6]=25H //CONTENT OF R6 IS 25H

MOV R1, #12H [R1]=12H //CONTENT OF R1 IS 12H

MOV R4, #0F3H [R4]=F3H //CONTENT OF R4 IS F3H

PUSH 6 [SP]=08 [08]=[06]=25H //CONTENT OF 08 IS 25H

PUSH 1 [SP]=09 [09]=[01]=12H //CONTENT OF 09 IS 12H

PUSH 4 [SP]=0A [0A]=[04]=F3H //CONTENT OF 0A IS F3H

POP 6 [06]=[0A]=F3H [SP]=09 //CONTENT OF 06 IS F3H

POP 1 [01]=[09]=12H [SP]=08 //CONTENT OF 01 IS 12H

POP 4 [04]=[08]=25H [SP]=07 //CONTENT OF 04 IS 25H

#### k. Exchange instructions

The content of source ie., register, direct memory or indirect memory will be exchanged with the contents of destination ie., accumulator.

i. XCH A,R3

ii. XCH A,@R1

iii. XCH A,54h

l. Exchange digit. Exchange the lower order nibble of Accumulator (A0-A3) with lower order nibble of the internal RAM location which is indirectly addressed by the register.

i. XCHD A,@R1

ii. XCHD A,@R0

m. Swap instruction

This instruction will swap upper nibble with lower nibble. For example, if content of A=09h,

i. SWAP A; after execution, content of A = 90h.

## **2. Arithmetic instructions**

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

### ***Addition***

In this group, we have instructions to

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

i. ADD A, #45H

ii. ADDC A, #0B4H

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

i. ADD A, R5

ii. ADDC A, R2

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

i. ADD A, 51H

ii. ADDC A, 75H

iii. ADD A, @R1

iv. ADDC A, @R0

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

### **Decimal Arithmetic:**

#### **1. 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.

## 2. 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 will result in hexadecimal result.

For example:           MOV A, #17H  
                          ADD A, #28H

The result stored in A is 3FH but not 45H. To correct the result to get the proper BCD number 6 is added to the lower nibble or upper nibble or both if it is higher than 9. For example 3FH+06=45H.

The DA(**D**ecimal **A**djust **A**fter **A**ddition) 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 it is higher than 9.

For example:           MOV A, #17H  
                          ADD A, #28H  
                          DA A

## *Subtraction*

In this group, we have instructions to

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

- i. SUBB A, #45H
- ii. SUBB A, #0B4H

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

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

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

- i. SUBB A, 51H



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

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

### ***Multiplication***

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

Eg.    MOV A,#45H ;[A]=45H  
          MOV B,#0F5H ;[B]=F5H  
          MUL AB ;[A] x [B] = 45 x F5 = 4209  
                ;[A]=09H, [B]=42H

### ***Division***

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

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

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

1. INC A
2. INC Rn
3. INC DIRECT
4. INC @Ri
5. INC DPTR

INC increments the value of source by 1. If the initial value of register is FFh, incrementing the value will cause it to reset to 0. The Carry Flag is not set when the value "rolls over" from 255 to 0.

In the case of "INC DPTR", the value two-byte unsigned integer value of DPTR is incremented. If the initial value of DPTR is FFFFh, incrementing the value will cause it to reset to 0.

***Decrement:*** *decrements the operand by one.*

1. DEC A
2. DEC R<sub>n</sub>
3. DEC DIRECT
4. DEC @R<sub>i</sub>

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

### **3. Logical instructions**

#### ***Logical AND***

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

1. ANL A,#DATA
2. ANL A, R<sub>n</sub>
3. ANL A,DIRECT
4. ANL A,@R<sub>i</sub>
5. ANL DIRECT,A
6. ANL DIRECT, #DATA

#### ***Logical OR***

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

1. ORL A,#DATA
2. ORL A, R<sub>n</sub>
3. ORL A,DIRECT

4. ORL A,@Ri
5. ORL DIRECT,A
6. ORL DIRECT, #DATA

### **Logical Ex-OR**

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

1. XRL A,#DATA
2. XRL A,Rn
3. XRL A,DIRECT
4. XRL A,@Ri
5. XRL DIRECT,A
6. XRL DIRECT, #DATA

### **Logical NOT**

**CPL** complements *operand*, leaving the result in *operand*. If *operand* is a single bit then the state of the bit will be reversed. If *operand* is the Accumulator then all the bits in the Accumulator will be reversed.

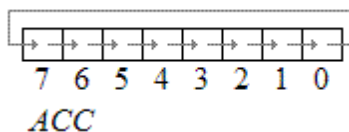
1. CPL A,
2. CPL C,
3. CPL bit address

**SWAP A** – Swap the upper nibble and lower nibble of A.

### **Rotate Instructions**

#### **Rotate Right (RR A)**

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



MOV A,#36H ;A = 0011 0110

RR A ;A = 0001 1011

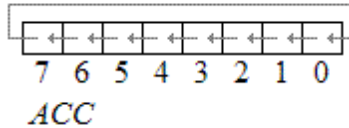
RR A ;A = 1000 1101

RR A ;A = 1100 0110

RR A ;A = 0110 0011

### Rotate Left (RL A)

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



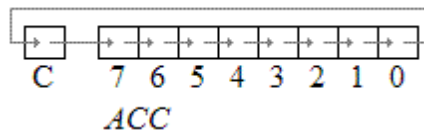
MOV A,#72H ;A = 0111 0010

RL A ;A = 1110 0100

RL A ;A = 1100 1001

### Rotate Right through Carry (RRC A)

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



CLR C ;make CY = 0

MOV A,#26H ;A = 0010 0110

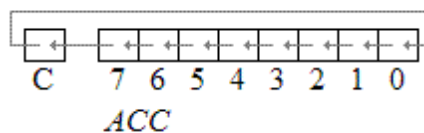
RRC A ;A = 0001 0011 CY = 0

RRC A ;A = 0000 1001 CY = 1

RRC A ;A = 1000 0100 CY = 1

### Rotate Left through Carry (RLC A)

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



```

CLR C ;make CY = 0
MOV A,#26H ;A = 0010 0110
RLC A ;A = 0100 1100 CY = 0
RLC A ;A = 1001 1000 CY = 0
RLC A ;A = 0011 0000 CY = 1

```

#### **4. Branch instructions**

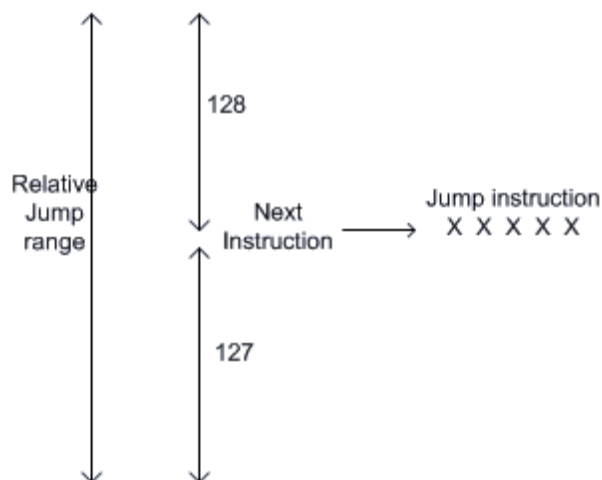
##### **Jump and Call Program Range**

There are 3 types of jump instructions. They are:-

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

##### ***Relative Jump***

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



**Ranges of Jump instruction**

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

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

**Disadvantages of the absolute jump: -**

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

**Instructions that use Relative Jump**

**SJMP <relative address>;** *this is unconditional jump*

*The remaining relative jumps are conditional jumps*

**JC <relative address>**

**JNC <relative address>**

**JB bit, <relative address>**

**JNB bit, <relative address>**

**JBC bit, <relative address>**

**CJNE <destination byte>, <source byte>, <relative address>**

**DJNZ <byte>, <relative address>**

**JZ <relative address>**

**JNZ <relative address>**

***Short Absolute Jump***

In this case only 11 bits of the absolute jump address are needed. The absolute jump address is calculated in the following manner.

In 8051, 64 kbyte of program memory space is divided into 32 pages of 2 kbyte each. The hexadecimal addresses of the pages are given as follows:-

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

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

Advantage: The instruction length becomes 2 bytes.

Example of short absolute jump: -

ACALL <address 11>

AJMP <address 11>

### ***Long Absolute Jump/Call***

Applications that need to access the entire program memory from 0000H to FFFFH use long absolute jump.

Since the absolute address has to be specified in the op-code, the instruction length is 3 bytes (except for JMP @ A+DPTR). This jump is not re-locatable.

Example: -

LCALL <address 16>

LJMP <address 16>

JMP @A+DPTR

Another classification of jump instructions is

1. Unconditional Jump
2. Conditional Jump

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

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

*eg: LJMP 3000H*

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

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

## 2. Conditional Jump instructions.

JBC Jump if bit = 1 and clear bit

JNB Jump if bit = 0

JB Jump if bit = 1

JNC Jump if CY = 0

JC Jump if CY = 1

CJNE reg,#data Jump if byte  $\neq$  #data

CJNE A,byte Jump if A  $\neq$  byte

DJNZ Decrement and Jump if A  $\neq$  0

JNZ Jump if A  $\neq$  0

JZ Jump if A = 0

*All conditional jumps are short jumps.*

### **Bit level jump instructions:**

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

JB bit, rel ; jump if the direct bit is set to the relative address specified.

JNB bit, rel ; jump if the direct bit is clear to the relative address specified.

JBC bit, rel ; jump if the direct bit is set to the relative address specified and then clear the bit.



## **5. Subroutine CALL And RETURN Instructions**

Subroutines are handled by CALL and RET instructions

There are two types of CALL instructions

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

This is long call instruction which unconditionally calls the subroutine located at the indicated 16 bit address.

This is a 3 byte instruction. The LCALL instruction works as follows.

- a. During execution of LCALL,  $[PC] = [PC] + 3$ ; (if address where LCALL resides is say, 0x3254; during execution of this instruction  $[PC] = 3254h + 3h = 3257h$ )
- b.  $[SP] = [SP] + 1$ ; (if SP contains default value 07, then SP increments and  $[SP] = 08$ )
- c.  $[[SP]] = [PC7-0]$ ; (lower byte of PC content ie., 57 will be stored in memory location 08.)
- d.  $[SP] = [SP] + 1$ ; (SP increments again and  $[SP] = 09$ )
- e.  $[[SP]] = [PC15-8]$ ; (higher byte of PC content ie., 32 will be stored in memory location 09.)

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

- f.  $[PC] = \text{address (16 bit)}$ ; the new address of subroutine is loaded to PC. No flags are affected.

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

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

- a. During execution of SCALL,  $[PC] = [PC] + 2$ ; (if address where LCALL resides is say, 0x8549; during execution of this instruction  $[PC] = 8549h + 2h = 854Bh$ )
- b.  $[SP] = [SP] + 1$ ; (if SP contains default value 07, then SP increments and  $[SP] = 08$ )
- c.  $[[SP]] = [PC7-0]$ ; (lower byte of PC content ie., 4B will be stored in memory location 08.)
- d.  $[SP] = [SP] + 1$ ; (SP increments again and  $[SP] = 09$ )
- e.  $[[SP]] = [PC15-8]$ ; (higher byte of PC content ie., 85 will be stored in memory location 09.)

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

- f.  $[PC10-0] = \text{address (11 bit)}$ ; the new address of subroutine is loaded to PC. No flags are affected.

## **RET instruction**

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

g.  $[PC15-8] = [[SP]]$  ;content of current top of the stack will be moved to higher byte of PC.

h.  $[SP]=[SP]-1$ ; (SP decrements)

i.  $[PC7-0] = [[SP]]$  ;content of bottom of the stack will be moved to lower byte of PC.

j.  $[SP]=[SP]-1$ ; (SP decrements again)

## **6. Bit manipulation instructions**

8051 has 128 bit addressable memory. Bit addressable SFRs and bit addressable PORT pins. It is possible to perform following bit wise operations for these bit addressable locations.

### **1. LOGICAL AND**

a.  $ANL\ C, BIT(BIT\ ADDRESS)$  ; ‘LOGICALLY AND’ CARRY AND CONTENT OF BIT ADDRESS, STORE RESULT IN CARRY

b.  $ANL\ C, /BIT$  ; ‘LOGICALLY AND’ CARRY AND COMPLEMENT OF CONTENT OF BIT ADDRESS, STORE RESULT IN CARRY

### **2. LOGICAL OR**

a.  $ORL\ C, BIT(BIT\ ADDRESS)$  ; ‘LOGICALLY OR’ CARRY AND CONTENT OF BIT ADDRESS, STORE RESULT IN CARRY

b.  $ORL\ C, /BIT$  ; ‘LOGICALLY OR’ CARRY AND COMPLEMENT OF CONTENT OF BIT ADDRESS, STORE RESULT IN CARRY

### **3. CLR bit**

a.  $CLR\ bit$  ; CONTENT OF BIT ADDRESS SPECIFIED WILL BE CLEARED.

b.  $CLR\ C$  ; CONTENT OF CARRY WILL BE CLEARED.

### **4. CPL bit**

a.  $CPL\ bit$  ; CONTENT OF BIT ADDRESS SPECIFIED WILL BE COMPLEMENTED.

b.  $CPL\ C$  ; CONTENT OF CARRY WILL BE COMPLEMENTED.

## Instructions that affect flag bits

<b>Instruction</b>	<b>CY</b>	<b>OV</b>	<b>AC</b>
ADD	X	X	X
ADDC	X	X	X
SUBB	X	X	X
MUL	0	X	
DIV	0	X	
DA	X		
RPC	X		
PLC	X		
SETB C	1		
CLR C	0		
CPL C	X		
ANL C, bit	X		
ANL C, /bit	X		
ORL C, bit	X		
ORL C, /bit	X		
MOV C, bit	X		
CJNE	X		