8. Interrupt

A single microcontroller can serve several devices. There are two ways to do that: interrupts or polling. In the Interrupt method, whenever any device needs its service, the device notifies the microcontroller by sending an interrupt signal. Once the interrupt is accepted the microcontroller serves the device by executing an interrupt service routine (ISR). In polling method the microcontroller continuously monitor the status of a give device, when the condition is met it performs the service. This polling method is not efficient because it has to monitor all times the status of devices in round-robin fashion and priority assignment is not possible.

Interrupt Service Routine

For every interrupt, there must be an Interrupt service routine (ISR), Interrupt handler. For every interrupt, there is a fixed location in memory that holds the address of its ISR. The group of memory locations set aside to hold the addresses of ISRs is called the interrupt vector table.

Steps in executing an Interrupt

Once an interrupt is activated, microcontroller performs the following steps.

1. It finishes the instruction it is executing and save the address of the next instruction (PC) on the stack.
2. It also saves the the current status of all the interrupts internally.
3. It jumps to a fixed location in memory called the interrupt vector table that holds the address of ISR.
4. The microcontroller gets the address of the ISR from the interrupt vector table and jumps to it. It starts to execute the ISR until it reaches last instruction of subroutine RETI (return from the interrupt).
5. Upon executing the RETI instruction, the microcontroller returns to the place where it was interrupted. First it gets PC address from the stack by popping the top two bytes of the stack into the PC

Six Interrupts of 8051

The six interrupts in the 8051 are allocated as follows

1. Reset- when the reset pin is activated, the 8051 jumps to address location 0000. This is power-up reset.

2. Two interrupts are set aside for the timers: one for timer 0 and one for timer 1. Memory locations 000BH and 001BH in the interrupt vector table belongs to timer 0 and timer 1.respectively.
3. Two interrupts are set aside for hardware external hardware interrupts, Pin numbers 12 (P3.2) and 13(P3.3) in port 3 are for the external hardware interrupts INT0 and INT1, respectively. These external interrupts are also referred to as EX1 and EX2. Memory location 0003H and 0013H in the interrupt vector table are assigned to INT0 and INT1 respectively.
4. Serial communication has a single interrupt that belongs to both receive and transfer. The interrupt vector table location 0023H belongs to this interrupt.

Interrupt Vector Table for 8051

From the table it has been observed that only three bytes of ROM space is assigned to the reset pin. They are ROM address locations 0,1 and 2.

Table 4.2 Interrupt vector addresses

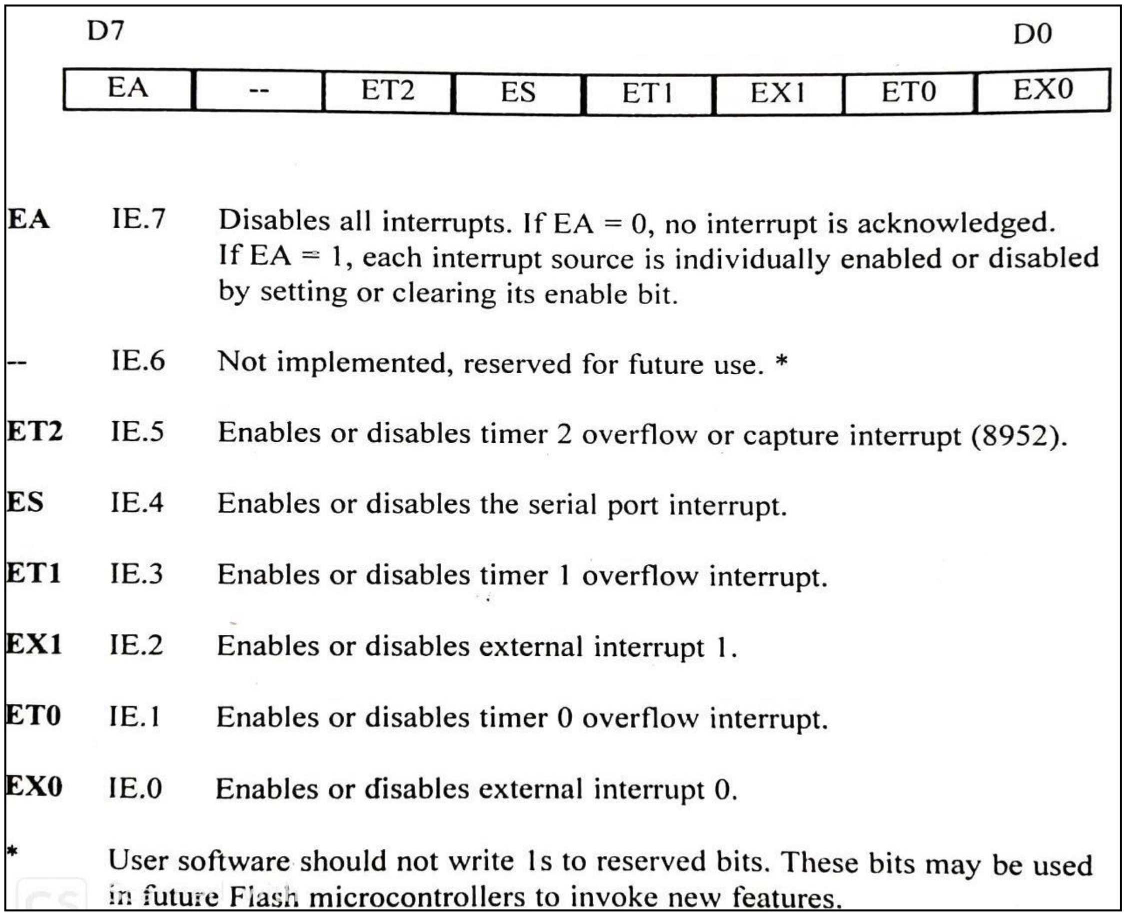
Interrupt	**ROM Location(Hex)**	**Pin**
Reset	0000	9
External hardware interrupt (INT0)	0003	12
Timer 0 interrupt (TF0)	000B	13
External hardware interrupt (INT1)	0013	
Timer 1 interrupt (TF1)	001B	
Serial COM interrupt (RI and TI)	0023	

Enabling and disabling an interrupt

Upon reset all interrupts are disabled. The interrupts must be enabled by software. There is a register IE (interrupt enable) that is responsible for enabling and disabling the interrupts.

To enable an interrupt following steps should be followed.

1. Bit D7of IE register (EA) must be set high to allow the rest of register to take effect.
2. If EA=1 , interrupts are enabled and will be responded to if their corresponding bits in IE are high. If EA=0, no interrupt will be responded to, even if the associated bit in the IE register is high.



D7 | | | | | | | D0

EA	--	ET2	ES	ET1	EX1	ET0	EX0

EA	IE.7	Disables all interrupts. If EA = 0, no interrupt is acknowledged. If EA = 1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit.
--	IE.6	Not implemented, reserved for future use. *
ET2	IE.5	Enables or disables timer 2 overflow or capture interrupt (8952).
ES	IE.4	Enables or disables the serial port interrupt.
ET1	IE.3	Enables or disables timer 1 overflow interrupt.
EX1	IE.2	Enables or disables external interrupt 1.
ET0	IE.1	Enables or disables timer 0 overflow interrupt.
EX0	IE.0	Enables or disables external interrupt 0.

* User software should not write 1s to reserved bits. These bits may be used in future Flash microcontrollers to invoke new features.

Fig. 4.21 Interrupt Enable Register

Interrupt Priority in the 8051

When the 8051 is powered up, the priorities are assigned, that are enlisted in table.

Table 4.3 Interrupt priority

Highest to Lowest priority	
External Interrupt 0	INT0
Timer Interrupt 0	TF0
External Interrupt 1	INT1
Timer Interrupt 1	TF1
Serial Communication	RI-TI

9. Programmer's Model

The CPU registers are used to store the data temporarily. The information may be data to be processed or address pointing the data to be fetched. The majority of registers are 8 bits. The 8-bit registers are shown in the diagram from MSB (most significant bit) D7 to the LSB (least significant bit) D0. The most widely used registers of 8051 are A (accumulator), B, R0, R1, R2, R3, R4, R5, R6, R7, DPTR (data pointer), and PC (program counter). All these registers are 8 bits except DPTR and the program counter. The accumulator is used to hold one operand before execution and hold the result after execution. The program counter points to the address of next instruction to be fetched. It is a auto increment register. As the size of program counter is 16 bit. 8051 can access the program addresses from 0000H-FFFFH. When 8051 is powered-up the program counter contents will be 0000H. This means that it expects the first opcode to be stored at ROM address 0000H. For this reason in the 8051 system, the first opcode must be burned memory location 0000H of program ROM since this is where it looks for the first instruction when it is booted.

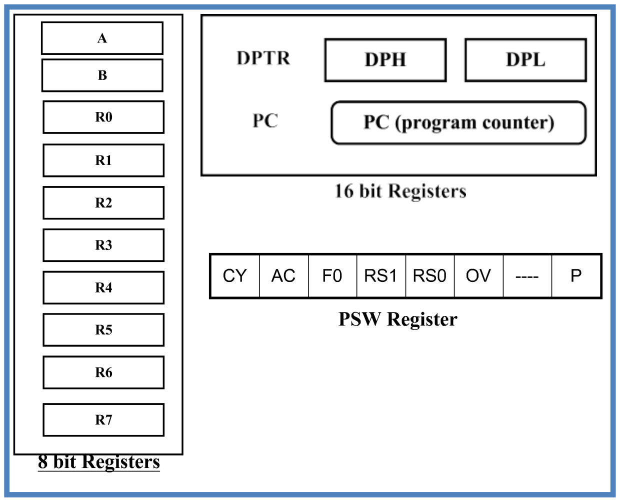
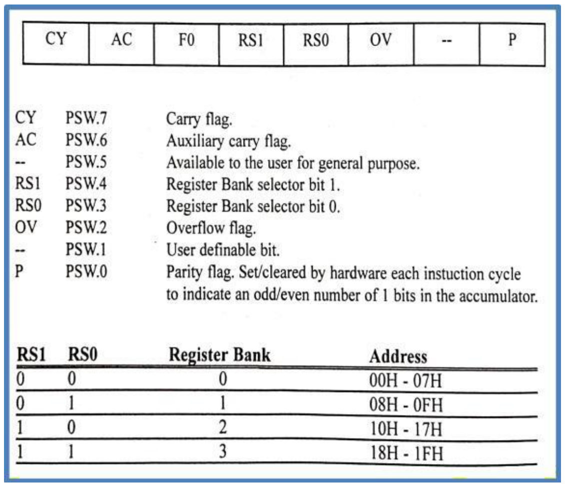


Fig. 4.22 Programmer's model



CY	AC	F0	RS1	RS0	OV	--	P

CY	PSW.7	Carry flag.
AC	PSW.6	Auxiliary carry flag.
--	PSW.5	Available to the user for general purpose.
RS1	PSW.4	Register Bank selector bit 1.
RS0	PSW.3	Register Bank selector bit 0.
OV	PSW.2	Overflow flag.
--	PSW.1	User definable bit.
P	PSW.0	Parity flag. Set/cleared by hardware each instuction cycle to indicate an odd/even number of 1 bits in the accumulator.

RS1	RS0	Register Bank	Address
0	0	0	00H - 07H
0	1	1	08H - 0FH
1	0	2	10H - 17H
1	1	3	18H - 1FH

Fig. 4.23 PSW register

PSW (program status word register)

The program status word register (PSW) is an 8-bit register. It is also referred as Flag register. Although this register is size of 8-bits, only 6bits are used by 8051. Two unused bits are user definable flags. Other 4 bits are called as conditional flags such as CY (carry), AC (auxiliary carry), P(parity) and OV(overflow).In this register the bits PSW.3 and PSW.4 are designated as RS0 and RS1 and used to select the banks. PSW.5 and PSW.1 bits are general purpose status flags and can be used by the programmer for any purpose.

10. Operand addressing

An addressing mode is a method of specifying the data source or destination in an instruction. There are 5 types of addressing modes is supported by 8051.

1. Register
2. Immediate
3. Direct (memory related)
4. Register Indirect (memory related)
5. Index register addressing

Register addressing mode

This addressing mode involves the use of registers to hold the data to be manipulated.

Examples:

MOV A, R0 ; Copy the contents of R0 int A

ADD A, R7 ; Add the contents of R7 to contents of A and the result is stored in A

Immediate addressing mode

In this addressing mode immediate data is specified in instruction as a source operand.

Examples:

```
MOV B, #40H            ; load 40H into B register

MOV DPTR, #2000H       ; load 2000H into DPTR
```

Direct addressing mode

As we know the on-chip RAM of 8051 is 128 byte, it can be accessed through memory address from 00H to FF H. The allocations of 128 bytes are as follows.

1. RAM location 00H-1FH are assigned to register banks and stack
2. RAM location 20H-2FH is set aside as bit-addressable space to save single bit data.
3. RAM location 30H-7F is available as place to save bite-sized data.

Although the entire 128 bytes of RAM can be accessed through direct addressing mode, it is most often used to access RAM location 30H-7FH. This is due to fact that register banks are accessed through their names.

Examples:

MOV R4, 70H	; move the contents of RAM location 70H to R4.
MOV 56H, A	; save the content of A in RAM location 56H
PUSH 05	; push R5 onto the stack

Register indirect addressing mode

In this mode the address (of 8bits) is indirectly specified in the instruction by the contents of pointer. This addressing mode so called because the source operand is from the address specified indirectly by another register in the instruction. The limitation is that only R0 and R1 register can be used in 8051 for indirect addressing. SFRs are directly accessible.

Examples

MOV R1, #55H	; load pointer R1=55H
MOV A, @R1	; the content of pointer is transferred to A

Index registers addressing

Suppose we need to access external data RAM and external code space of on-chip ROM 16 bit address must be required. In this case we have to use DPTR. This mode is widely used in accessing data elements of look-up table entries in the program ROM space of 8051.

Examples;

MOV DPTR, #0200H	; load DPTR with 0200
CLR A	; clear accumulator
MOVC A,@A+DPTR	; Move the content 0200 location into A

11. Instruction set

The instruction set of 8051 can be classified into following group.

1. Data Transfer Instructions
2. Arithmetic Instructions
3. Logic Instructions
4. Boolean Variable manipulation Instructions
5. Program flow control (Processor and Machine control) Instructions
6. Interrupt flow Control instruction

12.1 Data Transfer Instruction

Three types of the data transfer can be done by move instruction. First type is transfer within the internal RAM and SFRs, second type is transfer using code memory area (CODE) and the third is using the external data memory X-DATA).

MOV instruction

A MOV instruction means move (copy) the bits from one source to a destination.

Table 4.4 MOV instructions within the registers, internal RAM and SFRs in 8051

Instruction (Mnemonic)	Action	Addressing	Length in bytes	cycles
MOV A, Rn	Move Rn into A	Register	1	1
MOV Rn, A	Move into Rn from A	Register	1	1
MOV A, #data	Move immediate 8-bit data into A	Immediate	2	1
MOV Rn, #data	Move into Rn the data.	immediate	2	1
MOV A, direct	Move byte at the direct address into A	Direct	2	1
MOV Rn, direct	Move from direct address into Rn	Direct	2	2
MOV direct, A	Move byte to the direct address form A	Direct	2	1
MOV direct, Rn	Move a byte to the direct address from Rn	Direct	2	2
M OV direct, direct	Move byte to the direct address from the direct address	Direct	3	2
MOV direct, #data	Move immediate data byte to the direct address	Immediate	3	2
MOV a,@Ri	Move into A the byte from the address pointed by Ri	Indirect	2	2
MOV @Ri, A	Move A into address pointed by Ri	Indirect	1	1
MOV direct, @Ri	Move into direct address from address pointed by Ri	indirect	1	1
MOV @Ri, direct	Move from the direct address to the address poined by ri	Indirect	2	2
MOV @Ri, #data	Move data ino address pointed by Ri	immediate	2	2
MOV DPTR, data16	Mov e16 bit dat	immediate	3	2

MOVC-type Instruction

It moves the 8-bit code from one source at the program memory (internal and external) to the register A destination.

Table 4.5 MOVC Instructions for transfer from the program memory area address code or constant to accumulator in 8051

Instruction	Action	Addressing	Length in bytes	Cycles
MOVC A, @A+DPTR	Moves the code or constant into A the byte from the program memory address pointed	Indirect	1	2

	by hypothetical addition of DPTR with the A itself.			
MOVC A, @A+PC	Move the code or constant into A the byte from the program memory address pointed by hypothetical addition of PC with the A itself	Indirect	1	2

MOX-type Instructions

A MOVX instruction means move (copy) the 8-bit data into A and from A using the external data memory address using DPTR or Ri as the pointer

Table 4.6 MOVX instruction

Instruction	**Action**	**Addressing**	**Length in bytes**	**Cycles**
MOVX A, @DPTR	Move the external data byte (X-DATA) into A from the data memory address pointed by DPTR	Indirect	1	2
MOVX @DPTR,A	Move into the external data memory from A to the address pointed by DPTR	Indirect	1	2
MOVX A,@Ri	Move the external data byte into a from the memory address pointed by Ri	Indirect	1	2
MOVX @Ri, A	Move into the external data memory from A to the memory address pointed by Ri	Indirect	1	2

Table 4.7 PUSH and POP instructions for using the Stack Area employing SP

Instruction	**Action**	**Addressing**	**Length in bytes**	**Cycles**
PUSH direct	Move byte from a direct internal RAM or SFR into the stack after first incrementing the stack pointer by 1	Direct	2	2
POP direct	Move byte to a direct internal RAM or SFR into the stack and then decrement the stack pointer by 1.	Direct	2	2

XCH-type instructions

An XCH instruction is for exchanging the A register with a source using the register (direct or indirect addresing0 mode.

Table 4.8 XCH and XCHD instruction

Instruction	**Action**	**Addressing**	**Length in bytes**	**cycles**
XCH A@Ri	Exchange byte at A with the address pointed by Ri	Indirect	1	2
XCH A,Rn	Exchange byte at A with the register Rn	Register	1	2
XCH A, direct	Exchange byte at A with the byte at a direct address.	Direct	1	1
XCHD A,@Ri	Exchange lower hex-digits of the bytes at A with the address pointed by Ri	Indirect	1	2

12.2 Arithmetic Instruction

These instructions include 8 bit addition, subtraction, increment, decrement, multiply and division instruction.

Table 4.9 Arithmetic ADD, SUB,MUL, DIV, INC and DEC instruction s in 8051

Instruction	**Action**	**Addressing**	**Flags affected**	**Length (bytes)**	**Cycles**
ADD A,Rn	Add Rn into A	Register	C,AC,OV	1	1
ADD A, direct	Add the byte at the direct address into A	Direct	C,AC,OV	2	1
ADD A, @Ri	Add the byte from the address pointed by the Ri into A	Indirect	C,AC,OV	1	1
ADD A, #data	Add immediate data byte to the A	Immediate	C,AC,OV	2	1
ADDC A, Rn	Add CF(carry) bit and Rn into A	Register	C,AC,OV	1	1
ADDC A, direct	Add CF bit and byte at the direct address ito A	Direct	C,AC,OV	2	1
ADDC A @Ri	Add CF bit and the byte from the address pointed by the Ri	Indirect	C,AC,OV	1	1
ADDC A, #data	Add CF bit and immediate data byte to the A	Immediate	C,AC,OV	**2**	**1**
SBBB A,Rn	Subtract borrow at CF bit and Rn into A	Rgister	C,AC,OV	**1**	**1**
SBBB A, direct	Subtract borrow at CF bit and byte at the direct address into A	Direct	C,AC,OV	**2**	**1**
SBBB A, @Ri	Subtract borrow at C bit and byte at the byte from the address pointed	Indirect	C,AC,OV	**1**	**1**

	by the Ri into A				
SBBB A, #data	Subtract borrow at CF bit and immediate data byte into A	Immediate	C,AC,OV	**2**	**1**
INC A	Increment	Register	None	1	1
INC Rn	Increment Rn	Register	None	1	1
INC direct	Increment byte at the direct address	Direct	None	2	1
INC @Ri	Increment the byte at the address pointed by Ri	Indirect	None	1	1
DEC A	Decrement A	Register	None	1	1
DEC Rn	Decrement Rn	Register	None	1	1
DEC direct	Decrement byte at the direct address	Direct	None	2	1
DEC @Ri	Decrement the byte at the address pointed by the Ri	Indirect	None	1	1
MUL AB	Multiply A and B Result MSB in B and LSB in A	Register	OV	1	4
DIV AB	Divide A (Numerator) and B(denominator) Remainder in B Quotient in A	Register	OV	1	4
DAA	Decimal adjust accumulator	Register	C	1	1

12.3 Logical Instruction

Table gives features of 8-bit AND, OR and XOR instruction. These instructions have 4 addressing modes such as register, immediate, direct and indirect.

Table 4.10 ANL, ORL XRL instruction

Instruction	**Action**	**Addressing**	**Length in bytes**	**Cycles**
ANL A, Rn	AND Rn into A	Register	1	1
ANL A, direct	AND byte at the direct address into A	Direct	2	1
ANL A, @Ri	AND into the byte from the address pointed by the Ri	Indirect	1	1
ANL A, #data	AND immediate data byte into A	immediate	2	1
ANL direct, A	AND A into byte at the direct address	Direct	2	1
ANL direct, #data	AND immediate byte into byte at the direct address	Direct	3	2
ORL A, Rn	OR Rn into A	Register	1	1
ORL A, direct	OR byte at the direct address into A	Direct	2	1
ORL A, @Ri	OR into the byte from the address pointed by Ri	Indirect	1	1
ORL A, #data	OR immediate data byte to the A	immediate	2	1
ORL direct, A	OR A into byte at the direct address	Direct	2	1
ORL direct,#data	OR immediate byte into byte at the	Direct	3	2

	direct address			
XRL A, Rn	XOR Rn into A	Register	1	1
XRL A, direct	XOR byte at the direct address into A	Direct	2	1
XRL A, @Ri	XOR the byte at the address pointed by Ri into A	Indirect	1	1
XRL A, #data	XOR immediate data byte to the A	immediate	2	1
XRL direct, A	XOR A into byte at the direct address	Direct	2	1
XRL direct, #data	XOR immediate byte into byte at the direct address	Direct	3	2

12.4 Boolean Variable manipulation Instructions

These are also called as Boolean processing instruction.

Table 4.11 MOV, CLR, CPL,SETB,ANL, and ORL Boolean Processing Instruction

Instruction	**Action**	**Addressing**	**Length (bytes)**	**Cycles**
MOV C, bit	Move bit into CF	Direct bit addressing	2	1
MOV bit, C	Move CF into the bit	Direct bit addressing	2	2
CLR C	Clear CF	PSW Register CF bit addressing	1	1
CLR bit	Clear bit	Direct bit addressing	2	1
CPL C	Complement CF	PSW Register CF bit addressing	1	1
CPL bit	Complement bit	Direct bit addressing	2	1
SETB C	Set CF=1	PSW Register CF bit addressing	1	1
SETB bit	Set bit =1	Direct bit addressing	2	1
ANL C,bit	AND between CF and bit, place the result in CF	Direct bit addressing	2	2
ANL C, $\overline{\text{bit}}$	AND between CF and , place the result in C	Direct bit addressing	2	2
ORL C,bit	OR between CF and bit, place the result in C	Direct bit addressing	2	2
ORL C, $\overline{\text{bit}}$	OR between CF and $\overline{\text{bit}}$, place the result in C	Direct bit addressing	2	2

12.5 Control Transfer Instruction

In the main program other sub programs may be called to perform a particular task. When a sub program is called the processor will jump to a new address where this program is available and it has to accomplish program flow control transfer with help of JUMP and CALL instruction when some condition met.

Table 4.12 Delay-Cycle (NOP) instruction (No operation)

Instruction	**Action**	**Addressing**	**Length in bytes**	**Cycles**
NOP	No operation, PC gets the address of next instruction on incrementing at NOP.		1	1

Long, Absolute and Short Jump

8051 has three jump instructions: Long- it jumps to 16-bit address, Absolute- it jumps within 2 K bytes and Short- it jumps to address within 128 bytes above or below the present address.

Table 4.13 Long, absolute and short jump instructions

Instruction	**Action**	**Addressing**	**Length in bytes**	**Cycles**
LJMP addr16	Jump to the next address given by two bytes in the instruction	Direct 16 bit address	3	2
AJMP addr11	Jump to the next address	Direct 11-bit address	2	2
SJMP *rel*	Jump in the range between -128 and +127 from the address of next instruction	Direct 8-bit	2	2
JMP @A+DPTR	Jump in the next address given by addition of 8-bits of A with 16-bits of DPTR	Indirect 16-bit relative addreess		

Table 4.14 Conditional Short Relative Jumps

Instruction	Action	Addressing	Length in bytes	Cycles
JNZ rel	Jump to a relative address if a is not zero	Relative(offset)	2	2
JZ rel	Jump to a relative address if A is zero	Relative(offset)	2	2
JNC rel	Jump to a relative address if CF is not 1	Relative(offset)	2	2
JC rel	Jump to a relative address if CF=1	Relative(offset)	2	2
JB bit, rel	Jump to a relative address if addressed bit 1 (bit not set)	Relative(offset)	2	2
JNB bit,rel	Jump to a relative address if addressed bit 0 (bit not set)	Relative(offset)	2	2
JBC bit, rel	Jump to a relative address if addressed bit 1(bit set) and reset	Relative(offset)	2	2

	carry (make CF=0)			

Decrement and Conditional jump on Zero

Table 4.15 Instruction for decrement and then jump in program-loops in 8051

Instruction	Action	Addressing	Length in bytes	Cycles
DJNZ Rn, Rel	Decrement Rn and jump if Rn is still not zero.	Relative (offset)	2	2
DJNZ direct, Rel	Decrement byte at the direct and jump if byte is still not zero	Relative (offset)	2	2

Jump after comparison

Table 4.16 Compare then conditional jump after comparison

Instruction	Action	Addressing	Flag affected	Length in bytes	Cycles
CJNE A, #data, rel	Compare A and immediate data and jump if both are not equal.	Relative (offset)	C	3	2
CJNE Rn, #data, rel	Compare Rn and immediate data and jump if both are not equal.	Relative (offset	C	3	2
CJNE A, direct, rel	Compare the bytes at A and direct and jump if both are not equal	Relative (offset	C	3	2
CJNE @Ri, #data, rel	Compare byte from the address pointed by Ri and immediate data and jump if both are not equal	Relative (offset)	C	3	2

Call to a Routine

`Table 4.17 Long, absolute call and return instruction

Instruction	**Action**	**Addressing**	**Length in bytes**	**Cycles**
LCALL addr16	Call to the next address given by two bytes in the instruction	Direct 16-bit address	3	2
ACALL addr11	Call the next address given by 11 bits in	Direct 11	2	**2**

	the instruction.	bit address		
RET	Return to PC the saved PCL and PCH from the stack.	Stack address	1	**2**

12.6 Interrupt Control Flow (RETI instruction)

Table 4.18 RETI instruction

Instruction	**Action**	**Addressing**	**Length In bytes**	**cycles**
RETI	Return into PC the saved PCL and	Stack adddress	1	2

12. Programming

While the CPU can work only in binary, it can do so at a very high speed, however, it is quite tedious and slow for humans to deal with 0s and 1s in order to program the computer. A program that consists of 0s and 1s is called machine language. In the early days of the computer programmers coded programs in machine language. Although the hexadecimal system was used as a more efficient way to represent binary numbers, the process of working in machine code was still cumbersome for humans. Eventually, assembly language were developed which provided **mnemonics** for the machine code instructions. Plus other features which made programming faster and less prone to error. Assembly language is referred to as low level language because it deals directly with internal structure of CPU. Programmer needs assembler to convert the assembly language to machine language for execution purpose. Assembly language consists mnemonics optionally followed by one or two operands.

Programs

P1. Write an ALP (Assembly Language Program) to find the sum of values and store the result in A (lower byte and in R7 (higher byte). Assume that RAM locations 40-44 have the following values.

40=(7B), 41=(EC), 42=(C4), 43=(5B), 44=(30)

Solution:

```
        MOV     R0, #40H        ; load pointer
        MOV     R2, #05H        ; load counter
        CLR     A               ; A=0
        MOV     R7, A           ; clear R7
AGAIN:  ADD     A, @R0          ; add the byte pointer
        JNC     NEXT            ; if CY=0 it can jump to NEXT label
        INC     R7              ; increment counter
NEXT:   INC     R0              ; increment pointer
        DJNZ    R2,AGAIN        ; repeat until R2is zero
HERE:   SJMP    HERE
```

P2. Assume that 5 BCD data items are stored in RAM locations starting a 40H as shown below. Write an ALP to find the sum of all numbers. The result must be in BCD.

Solution:

```
        MOV     R0, #40H        ; load pointer
        MOV     R2, #05H        ; load counter
        CLR     A               ; A=0
        MOV     R7, A           ; clear R7
AGAIN:  ADD     A, @R0          ; add the byte pointer
        DA      A
        JNC     NEXT            ; if CY=0 it can jump to NEXT label
        INC     R7              ; increment counter
NEXT:   INC     R0              ; increment pointer
```

```
        DJNZ    R2,AGAIN        ; repeat until R2is zero
HERE:   SJMP    HERE
```

P3. Write an ALP to get hex data in the range of 00-FFH from port 1 and convert it to decimal. Save the digits in R7, R6 and R5, where the least significant digit in R7.

```
        MOV     A, #0FFH
        MOV     P1, A           ; make an P1 an input port
        MOV     A1, P1          ; read data from P1
        MOV     B, #0AH         ;  move 0AH to register b
        DIV     AB              ; divide by the contents of A by B
        MOV     R7, B           ; Save lower digit in R7 register
        MOV     B , #0AH        ;
        DIV     AB              ;
        MOV     R6, B           ; save the next digit
        MOV     R5, A           ; save the last digit
HERE:   SJMP    HERE
```

P4. Read and test P1 to see whether it has the value 45H. if it does send 99H to P2; otherwise, it stays cleared.

Solution:

```
        MOV     P2, 00H         ; clear P2
        MOV     P1, #0FFH       ; make P1 an input port
        MOV     R3, #45H        ; R3=45H
        MOV     A, P1           ; read P1
        XRL     A, R3           ;
```

```
          JNZ       EXIT
          MOV       P2, #99H
EXIT:     ……
```

P5. Find the 2's complement of the value 78 H

Solution:

```
          MOV       A, #78H             ; A=85H
          CPL       A                   ; make 1's complement a
          ADD       A, #01H             ; make 2's complemt
HERE:     SJMP      HERE
```

P6. Write an ALP to determine if register A contains the value 99H, if so, make R1=FFH otherwise make R1=0.

Solution:

```
          MOV       R1, #00H            ; clear R1
          CJNE      A, #99H, NEXT       ; if A is not equal 99H then jump
          MOV       R1, #0FFH           ;  make R1=FFH
NEXT      …..
```

P7. Assume that P1 is an input port connected to a temperature sensor. Write an ALP to read the temperature and test it for the value 75. According to the rest result, place the temperature value into the registers indicated by the following.

If T=75   then A=75

If T<75   then R1=T

If T>75 then R2=T

Solution:

```
        MOV     P1, # 0FFH          ; make P1 an input port
        MOV     A, P1               ; read P1 port, temperature
        CJNE    A, #75, OVER        ; jump if  A is not equal 75
        SJMP    EXIT
OVER:   JNC     NEXT                ; if CY=0, then A>75
        MOV     R1, A               ; if CY=1, A<75
        SJMP    EXIT                ; Exit
NEXT:   MOV     R2, A
EXIT    ……
```

P8. Write an ALP that finds the number of 1s in a given byte 97H.

Solution:

```
         MOV    R1, #00H            ; clear R1
        MOV     R7, 08H             ; Counter=08
        MOV     A, 97H
AGAIN:  RLC     A                   ; rotate through CY once
        JNC     NEXT                ; check for CY
        INC     R1                  ; if CY=1 then increment R1
NEXT:   DJNZ    R7, AGAIN           ; go through 8times
HERE:   SJMP    HERE
```

P9. Assume that register a has packed BCD 29H, write an ALP to convert packed BCD to ASCII numbers and place them in R2 and R6.

Solution:

```
          MOV       A, #29H              ; A=29H, packed BCD
          MOV       R2, A                ; keep a copy of BCD data in R2
          ANL       A, #0FH              ; mask the upper nibble (A=09)
          ORL       A, #30H              ; make it an ASCII, A=39H
          MOV       A, R6                ; save in R6
          MOV       A, R2                ; A=29H
          ANL       A, #0F0H             ; mask the lower nibble
          RR        A                    ; rotae right
          RR        A                    ; rotae right
          RR        A                    ; rotate right
          RR        A                    ; rotate right
          ORL       A, #30 H             ; A=32H
          MOV       R2, A                ; save the ASCII character in R2
HERE:     SJMP      HERE
```

P10. Write an ALP to create a square wave of 50% duty cycle on bit 0 of port 1.

Solution:

```
HERE:     SETB      P1.0                 ; set to high bit 0of port 1
          LCALL     DELAY                ; call the delay subroutine
          CLR       P1.0                 ; p1.0=0
          LCALL     DELAY
          SJMP      HERE
```

P11. Assume that the bit P2.2 is used to control the outdoor light and bit P2.5 to control the light inside the building. Write an ALP to turn on outside light and to turn the inside one.

Solution:

```
SETB    C           ; CY=1
ORL     C, P2.2     ; CY=P2.2
MOV     P2.2, C     ;  turn  it "on" if not already "on"
CLR     C           ; CY=0
ANL     C, P2.5     ; CY=P2.5 ANDed with CY
MOV     P2.5, C     ; turn it off if not already off.
```