5

Arithmetic and Logical Instructions

Objectives

- List the arithmetic and logical operations performed by the 8051
- Explain the concept of signed and unsigned numbers and range of these numbers supported by the 8051
- Perform the signed and unsigned addition and subtraction
- Perform the multiplication and division
- Appreciate the role of flags in the arithmetic operations
- Introduce binary coded decimal (BCD) numbers and BCD arithmetic
- Illustrate the use of increment and decrement instructions
- Discuss OR, AND, NOT and EX-OR operations and their applications
- Explain byte as well as bit level logical operations
- List and illustrate the rotate and swap operations performed by the 8051

Key Terms

- Addition
- AND/OR/NOT/EXOR
- Arithmetic Operations
- Auxiliary Carry Flag: AC
- BCD Arithmetic
- Byte/Bit Operations
- Carry Flag: C or CY
- Division
- Logical Operations
- Multiplication
- Overflow Flag: OV
- Rotate
- Signed Arithmetic
- Subtraction
- Unary Operations

The 8051 microcontroller supports basic arithmetic operations such as addition, subtraction, division, multiplication, increment, decrement and logical operations such as AND, OR, NOT, and EX-OR. The unsigned operations with only 8-bit binary integers are supported directly, however, signed and multi-byte operations can be performed with the help of overflow and carry flags. It also supports BCD operations. Thus, arithmetic and logical operations can be used for mathematical calculations and data manipulation.

5.1 | ARITHMETIC INSTRUCTIONS

This group of instructions performs arithmetic operations. The arithmetic operations modify arithmetic flags Carry (CY or C), Overflow (OV) and Auxiliary Carry (AC). These flags (1-bit register) are modified (set/reset) according to the result obtained in an operation. The status of the flags is used as a test condition to make decisions by the program flow control (branch) instructions. A programmer may interpret the flag in more than one way. For example, C flag shows a carry out during addition and indicates borrow during subtraction.

An important point to note is that the flags are stored in the PSW register. Any instruction that modifies byte or bit in the PSW register can change the flags.

First, the arithmetic operations for unsigned numbers are discussed because they are directly supported by the 8051 and in later section, signed operations are discussed. For unsigned numbers, all data bits are used to represent the magnitude of a number. For example, in an 8-bit unsigned number, entire 8 bits are used for magnitude only; therefore, 8-bit number can have any value between 00H to FFH (0 to 255 decimal).

5.1.1 Addition

The 8051 supports half as well as full addition. The half addition is used to add two operands (of 8 bits each) specified in an instruction. While adding two 8-bit numbers, it is possible to get 9-bit result. The 9^{th} bit is the carry and it is stored in the carry (CY or C) flag. The full addition includes carry flag in the addition. The full addition is used for multi-byte operations and it is discussed in the next section.

The mnemonic ADD is used to add two 8-bit operands (half addition). The format of this addition instruction is,

```
ADD A, source      // A=A+ source.
                   // Add contents of A with operand source and place result into A.
```

All addressing modes can be used to specify the *source*. **The register A is always the destination as well as one of the source operand.** The formats of addition instructions for different addressing modes are as follows:

```
ADD A, #data       // add contents of A with immediate data and store result in A
ADD A, #10H        // A= A+10H, If A=05H → A=05H+10H=15H, C=0
ADD A, Rn          // add contents of A with contents of Rn and store result in A
ADD A, R0          // A = A+ R0, If A= F5H, R0=10H → A=F5H+10H=05H, C=1
ADD A, direct      // add A with contents of address direct and store result in A
ADD A, 40H         // A= A + (40H), If A=10H, (40H) =15H → A=10H+15H=25H, C=0
ADD A, @Ri         // add A with contents of address in Ri, store result in A
ADD A, @R0         // A= A+ (R0), If A=20H, R0=10H, (10H) =30H
                   // → A=20H+ (10H) = 20H+30H=50H,C=0
```

After addition, the C flag is set to 1 if there is carry out from MSB (bit position 7), otherwise it is cleared to zero. The AC flag is set to 1, if there is carry out from the lower nibble (bit position 3). The OV flag is set to1 if there is a carry out from bit 7 but not from bit position 6, or if there is a carry out of bit position 6 but not from position 7. The conditions in which the flags are set to 1 are illustrated in Figure 5.1.

The condition in which OV flag is set is given as:

OV = C7 XOR C6.

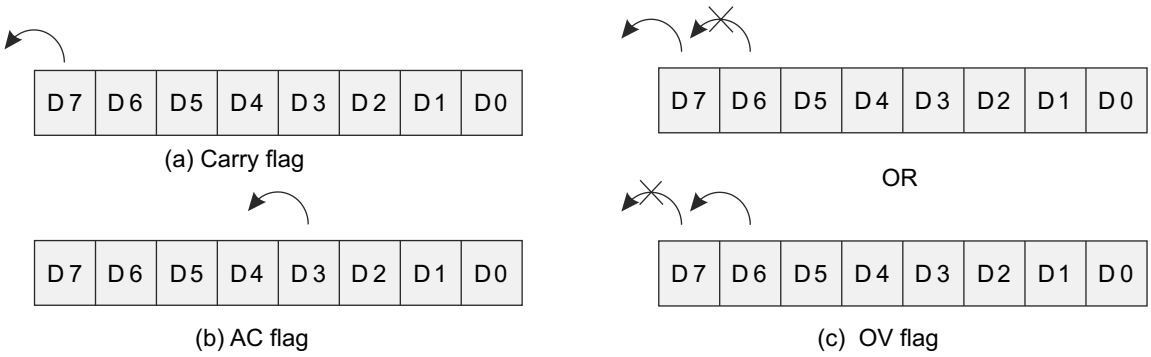


Fig. 5.1 *Conditions that set the flags*

Example 5.1

Assume R0=20H, (20H) =30H, (40H) = FFH, and A=50H. Illustrate the operation of ADD instruction for different addressing modes. Note that (XXH) is read as data at address XXH.

Solution:

Assuming the contents of register/memory locations as given above, before execution of **each** of the following instructions:

```
ADD A, #10H        // A= 50H+10H= 60H, C=0, AC=0
ADD A, R0          // A= 50H+20H= 70H, C=0, AC=0
ADD A, 40H         // A= 50H+FFH= 4FH, C=1, AC=0
ADD A, @R0         // the value present at address 20H is 30H, therefore
                   // A= 50H+30H= 80H, C=0, AC=0
```

Example 5.2

What will be the status of carry (CY) and auxiliary carry (AC) flags after execution of following instructions?

```
MOV A, #97H
ADD A, #80H
```

Solution:

The above two instructions add 97H with 80H and result will be placed in the A. The effect of addition on flags will be as shown below:

```
  97H  =   1001 0111 b
+ 80H  =   1000 0000 b
 117H     1 0001 0111 b
```

As can be seen from the above addition, there will be carry out from bit D7 and no carry from bit D3. Therefore, the carry flag is set, i.e. CY = 1 and auxiliary carry flag is reset, i.e. AC = 0.

Example 5.3

Write a program to add two numbers stored at internal RAM address 10H and 11H and store the result into internal RAM address 20H.

Solution:

Since the numbers to be added are stored in an internal RAM, we can access these numbers using direct addressing as shown in the program.

```
MOV A, 10H         // Read number stored at internal RAM address 10H into A
ADD A, 11H         // Add contents of A with data stored at address 11H and store result in to A
MOV 20H, A         // copy result of addition into internal RAM address 20H
```

Note that addition of two 8-bit numbers may generate 9-bit result (9^{th} bit in carry). The carry should be stored at suitable location when more than two bytes are added. This is illustrated in Example 5.5.

Addition of Multi-Byte Numbers

While dealing with multi-byte numbers (size of numbers is more than 8 bits) addition, we need to consider the propagation of carry from lower bytes to the higher bytes, i.e. we need to perform the full addition. The full addition includes carry

flag in the operation. The instruction ADDC is used for such operations. The suffix C after ADD indicate carry flag is also included in the addition. The operation of ADDC instructions is same as ADD instructions except that it also adds the contents of a carry flag with addition of both the operands. **ADDC means add with carry**. The formats of ADDC instructions for different addressing modes are,

ADDC A, #data	// add contents of A, immediate *data* and carry, store result in A
	// i.e. A = A + data + C
ADDC A, #10H	// A= A+10H+C, If A=05H, C=1→ A=05H+10H+1=16H, C=0
ADDC A, *Rn*	// add contents of A, contents of *Rn* and carry, store result in A
	// i.e. A= A+ *Rn* + C
ADDC A, R0	//A=A+ R0+C, If A=20H,R0=30H,C=0→ A=20H+30H+0=50H, C=0
ADDC A, *direct*	// add A, contents of address *direct* and carry, store result in A
	// i.e. A= A+ (direct) + C
ADDC A, 40H	// A= A + (40H) +C, If A=F0H, (40H) =15H, C=1 →
	// A=F0H+15H+1=06H, C=1
ADDC A, @*Ri*	// add contents A, contents of address in *Ri* and C, store result in A
	// A= A+ (*Ri*) + C
ADDC A, @R0	// A= A+ (R0) + C, If A=20H, R0=10H, (10H) =30H ,C=1
	//→ A=20H+ (10H)+1 = 20H+30H+1=51H,C=0

Multi-byte number addition is illustrated in Example 5.4.

Example 5.4

Write program to add two 16-bit numbers 42E1H and 255CH.

Solution:

The addition of these two 16-bit numbers will be performed in the following manner,

```
    1
  42 E1 H
+ 25 5C H
  68 3D H
```

When lower bytes are added, the result will be 3DH and CY is set (E1+5C=3D, CY=1). The carry is propagated to higher byte, and added to higher bytes, which results in 42 + 25 + 1 = 68H. The program to perform above addition is given below:

```
MOV A, #0E1H        // add lower bytes
ADD A, #5CH
MOV R5, A           // save lower byte result in to R5
MOV A, #42H         // add upper bytes including carry
ADDC A, #25H
MOV R7, A           // save higher byte of result in to R7
```

Usually, the ADD instruction is used for lower byte addition in multi-byte number addition. While ADDC can be used for all the bytes provided that the CY is properly initialized, i.e. for lower byte addition, ADDC can be used if carry is cleared before such addition as shown below:

```
CLR C               // initially carry is cleared
MOV A, #0E1H        // add lower bytes
ADDC A, #5CH
MOV R5, A           // save lower byte result in to R5
MOV A, #42H         // add upper bytes including carry
ADDC A, #25H
MOV R7, A           // save higher byte of result in to R7
```

Note: For this example, carry is not generated after addition of upper bytes; therefore it is not taken care of. Otherwise, it should have been considered as a part of result.

Example 5.5

Write a program to add three numbers stored at internal RAM address 10H, 11H and 12H and store the lower byte of result into internal RAM address 20H and upper bits into 21H.

Solution:

When we add three bytes, the maximum result will require 10 bits (For example, FFH+FFH+FFH=2FDH). The upper two bits of the result must be stored in 21H.

```
CLR C
MOV 21H, #00H       // clear 21H to store upper bits of addition
MOV A, 10H          // add contents of address 10H and 11H
ADD A, 11H
MOV R0, A           // store partial sum temporarily in R0
MOV A, 21H          // add carry (if any) to higher byte of result
ADDC A, #00H
MOV 21H, A          // store back
MOV A, R0           // retrieve partial sum (10H+11H)
ADD A, 12H          // add partial sum with contents of address 12H
MOV 20H, A          // store lower byte of sum in 20H
MOV A, 21H          // add carry (if any) to higher byte of result
ADDC A, #00H
MOV 21H, A          // store upper byte of result in 21H
```

Note: Novice programmers usually **erroneously** consider this type of addition (addition of multiple bytes) as multi-byte numbers addition!!

5.1.2 Subtraction

The mnemonic for subtraction is SUBB, **it means subtract with borrow**. The format of instruction is,

SUBB A, *source* // A= A – *source* – CY
// subtract carry and operand *source* from A and place result into A.

Similar to addition, all four addressing modes can be used for the *source* operand. **The A is always destination as well as one of the source operand.** The formats of subtraction instructions for different addressing modes are as follows:

SUBB A, #data	// subtract immediate *data* and carry from A, store result in A // i.e. A = A – data – C
SUBB A, #10H	// A= A–10H–C, If A=20H, C=1 → A=20H–10H–1=0FH, C=0
SUBB A, *Rn*	// subtract contents of *Rn* and carry from A, store result in A // i.e. A= A– *Rn* – C
SUBB A, R0	//A= A– R0–C, If A=30H, R0=20H,C=0 → A=30H–20H–0=10H, C=0
SUBB A, *direct*	// subtract contents of address *direct* and carry from A, store result in A // i.e. A= A– (direct) – C
SUBB A, 40H	// A= A – (40H) –C, If A=10H, (40H) =15H, C=0 → // A=10H–15H–0=FBH, C=1
SUBB A, @*Ri*	// subtract contents of address in *Ri* and C from A, store result in A // A= A– (Ri) – C
SUBB A, @R0	// A= A– (R0)–C, If A=50H, R0=10H, (10H) =30H, C=1 //→ A=50H– (10H)–1 = 50H–30H–1=1FH, C=0

Subtraction instruction treats carry flag as borrow and always subtract carry flag as a part of operation. In fact, meaning of the mnemonic SUBB is subtract with borrow. Many microcontrollers/processors have two different instructions for subtraction, SUB and SUBB. But in the 8051, we have only SUBB. To perform operation equivalent to SUB using SUBB we have to make CY=0 before execution of an instruction.

After subtraction,
CY=1 if borrow is needed into bit 7, CY=0 otherwise.
AC=1 if borrow is needed into bit 3, AC=0 otherwise.
OV=1 if borrow is needed into bit 7 and not in bit 6 or borrow in to bit 6 and not in bit 7, OV=0 otherwise.

Note: The Accumulator is always destination as well as one of the source operand for Addition and Subtraction.

Example 5.6

Write instructions to subtract 10H from 30H using immediate and register addressing.

Solution:

Using immediate addressing,

```
CLR C
MOV A, #30H        // A=30H
SUBB A, #10H       // A= A– 10H– C= 30H–10H– 0=20H
```

Using register addressing,

```
CLR C
MOV A, #30H        // A=30H
MOV R0, # 10H      // R0= 10H
SUBB A, R0         // A= A–R0–C= 30H–10H– 0 =20H
```

Note that carry flag is cleared before subtraction of 8-bit numbers.

In multi-byte subtraction C is cleared before subtraction of first byte and then included in subsequent higher byte operations.

5.1.3 Signed Arithmetic

All types of data considered so far in this text are assumed to be unsigned numbers, i.e. entire 8-bit operand represents magnitude only, but in real life the number can either be positive or negative. There should be some mechanism to represent and process such positive or negative numbers. The signed numbers are used to serve the purpose. In signed numbers, the most significant bit (MSB) is used to represent the sign (+ or –) and rest of the bits are used for the magnitude as shown in Figure 5.2. For the numbers greater than 8 bits, the leftmost bit of the most significant byte is used to represent the sign. The '0' in the MSB position indicate positive sign while '1' indicates negative sign.

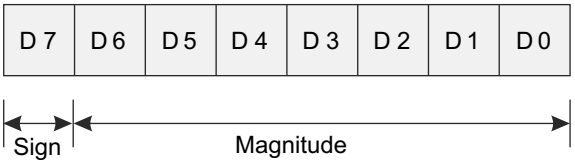


Fig. 5.2 *Signed numbers*

1. Positive Numbers

Bit D7 is '0' for positive numbers, since only seven bits (D0 to D6) are used for magnitude, the range of positive number that can be represented by 8-bit signed number is 0 to +127 as shown in Table 5.1.

If a positive number is larger than +127, a 16-bit or larger sized operand must be used.

Table 5.1 *8-bit positive numbers*

Decimal	Binary	Hex.
0	**0**000 0000	00
+1	**0**000 0001	01
+2	**0**000 0010	02
:	:	:
:	:	:
+126	**0**111 1110	7E
+127	**0**111 1111	7F

2. Negative Numbers

Bit D7 is 1 for negative numbers, however, negative numbers are not represented in true binary form, but it is represented in 2's complement form. Example 5.7 shows how negative numbers are represented.

Example 5.7

Represent –6 and –128 in 8–bit binary using 2's complement representation.

Solution:

The steps involved in this representation

1. Represent the magnitude of the number in 8-bit binary (without sign),

2. Complement each bit and
3. Add one to it.

The given number –6 is represented as,

```
  0000 0110        6 (magnitude) in 8 bit binary
  1111 1001        complement
+        1        add 1
  1111 1010
```

Thus, 11111010 (FAH) is the signed representation (or 2's complement) of – 6.

Similarly, for –128

```
  1000 0000 ——————→ 128 (magnitude),
  0111 1111 ——————→ invert each bit,
+        1 ——————→ add 1
  1000 0000
```

1000 0000 (80H) is the signed representation of –128.

On the contrary, if the 8-bit 2's complement signed number is given; the equivalent decimal number can be found as follows:

If the MSB is 0, the number is positive and other bits represent magnitude of the number. For example, consider 64H (0110 0100), since its MSB is 0, it is a positive number and remaining bits represent 100d, therefore the number is +100d.

If MSB is 1, a number is negative and the magnitude can be found by calculating 2's complement (because negative numbers are already represented in 2's complement) of a number. For example, consider FFH (1111 1111), since its MSB is 1, it is a negative number and magnitude is found from 2's complement of FFH, it is 1H (0000 0001), therefore the number represented by FFH is –1.

The other method is to find equivalent decimal number from the positional weight of each bit of a number. The positional weights for an 8-bit signed number are shown below:

–128	64	32	16	8	4	2	1
MSB							LSB

For example, consider 80H (1000 0000).

The decimal equivalent value of 80H is calculated from positional weights as

1× (–128) + 0×64 + 0×32 + 0×16 + 0×8 +1×4 + 0×2 + 0×1= –128d

Similarly, decimal equivalent for FFH (1111 1111) can be found as

1× (–128) + 1×64 + 1×32 + 1×16 + 1×8 + 1×4 + 1×2 + 1×1= –128+127= –1d

And for 7FH (0111 1111),

0× (–128) + 1×64 + 1×32 + 1×16 + 1×8 +1×4 +1×2 + 1×1 = +127d

From the above examples, we can say that range of byte-sized negative numbers is –1 to –128.

Table 5.2 shows byte-sized signed numbers.

The CY and OV flags are used to handle the unsigned and the signed operations. CY is generally used in the unsigned arithmetic while OV is used in the signed arithmetic.

Table 5.2 *8-bit signed numbers*

Decimal	Binary	Hex.
–128	1000 0000	80
–127	1000 0001	81
:	:	:
:	:	:
–1	1111 1111	FF
0	0000 0000	00
+1	0000 0001	01
+127	0111 1111	7F

THINK BOX 5.1

What are the signed and unsigned decimal representations of an 8-bit number 21H?

Both are 33

Discussion Question For a byte-sized signed number, why is there only 127 positive numbers, while negative numbers are 128?

Answer The number 0 is considered to have positive sign, so we may consider there are 128 (0, +1 to +127) positive numbers.

3. Overflow

A byte-sized signed number may range from –128d (1000 000b) to +127d (0111 1111b). If the result of the operation on signed number exceeds this range, an overflow will occur, which indicates error in a result. The 8051 indicate this error by raising overflow (OV) flag. This problem can be understood by Example 5.8.

Example 5.8

Illustrate the use of overflow flag using suitable example.

Solution:

Consider addition of two signed numbers +90H and +70H.

+90d	5AH	0101 1010 B
+70d	46H	0100 0110 B
+160d	A0H	1010 0000 B
		= A0H OV=1

A0 = –96 decimal

In this example, +90 is added to +70 and the result after addition is – 96 (assuming the signed numbers), which is incorrect, because the expected result is +160, which is larger than what an 8-bit number could represent or an 8-bit register could store (8-bit register could contain only up to +127). The 8051 indicates error by setting OV=1. It is up to the programmer to take care of erroneous result. The interpretation and recovery from an erroneous result is discussed in more detail in next section.

Let us understand signed arithmetic for positive (+ve) and negative (–ve) numbers for addition and subtraction. There are four types of operations in signed arithmetic as shown in Table 5.3.

Table 5.3 *Type of signed operations*

	Operation	Type of signed numbers	Remark
1	Addition	*Unlike* numbers	*Like* numbers means both numbers have same sign, *Unlike* numbers means opposite sign
2	Addition	*Like* numbers	
3	Subtraction	*Unlike* numbers	
4	Subtraction	*Like* numbers	

4. Addition of Unlike Signed Numbers

When *unlike* signed numbers are added then result will be always within a range of –128d to +127d, and the sign of the result will be always correct because the result (either positive or negative) will be smaller than larger of two original numbers. Addition of unlike signed numbers is illustrated in Examples 5.9 to 5.12.

Example 5.9

Add –02d with +30d.

Solution:

	(– 02d) =		1111 1110 B	= FEH
+	(+ 30d) =		0001 1110 B	= 1EH
	+28d	1)	0001 1100 B	1CH
			=+28d	

In the above addition, there is a carry out from bit 7 as well as from bit 6, therefore the overflow (OV) flag is 0, and the result is correct and for this condition, no action should be taken to correct the result.

> ***Note:*** Numbers without any suffix or with suffix 'D' are decimal numbers, and suffix 'B' and 'H' represents binary and hexadecimal numbers respectively.

Example 5.10

Add –128d with +127d.

Solution:

	(–128d) =	1000 0000B	=	80H
+	(+127d) =	0111 1111B	=	7FH
	–1d	1111 1111B		FFH
		= – 1d		

There is no carry out from bit 7 as well as bit 6, so OV=0 and C=0 and for this condition, no action should be taken to correct the sum because it is already correct.

Example 5.11

Add 0d with –1d.

Solution:

	(00d) =	0000 0000B	=	00H
+	(–1d) =	1111 1111B	=	FFH
	–1d	1111 1111B		FFH
		= –1d		

The OV=0, therefore the result is correct.

Example 5.12

Add +1d with –128d.

Solution:

	(+001d) =	0000 0001B	=	01H
+	(–128d) =	1000 0000B	=	80H
	–127d	1000 0001B		81H
		= –127d		

Again, it is seen that the result is correct.

In conclusion, when two unlike signed numbers are added, the result is always a correct signed number by neglecting the carry.

5. *Addition of Like Signed Numbers*

If two positive numbers are added, it is possible that sum may exceed +127d, i.e. overflow may occur. Addition of like signed numbers is illustrated in Examples 5.13 to 5.16.

Example 5.13

Add +40d with +70d.

Solution:

	(+40d) =	0010 1000B	=	28H
+	(+70d) =	0100 0110B	=	46H
	+110d	0110 1110B		6EH
		= +110d		

In the above case, there are no carry from bit 6 and 7 of sum. There, C=0 and OV=0 and the result is correct.

Example 5.14

Add +100d with +70d.

Solution:

	(+100d) =	0110 0100B	=	64H
+	(+070d) =	0100 0110B	=	46H
	+170d (expected)	1010 1010B		AAH
		= –86d (actual result)		

Here, the expected result is +170, which exceeds the maximum positive number (+127) that an 8-bit number can represent (or an 8-bit register can hold). So we get an incorrect result. This is indicated by OV=1, because there is carry out from bit 6 but not from bit 7.

The reason for getting an erroneous result in above example may also be understood this way: Since both the original numbers were positive and therefore the result should have been positive, but we got it as negative because of the excessive magnitude of the result (overflow) has modified (inverted) the sign bit.

Interpretation of the Result There are three ways the result may be interpreted. Though these methods (described next) are unconventional, they would help in instigating a better understanding of the topic.

(a) As we have seen that the overflow modifies (invert) the sign bit, therefore to interpret the result, let us change the sign of the erroneous result AAH (**1**010 1010 b). After inverting the sign bit the result will become 2AH (**0**010 1010 b) = +42d. This may be considered as the actual result is 42d higher (because of the + sign) than +128 (if we have some way of representing +128), i.e.

+128d
<u>+042d</u>
+170d

But, we cannot represent +128 using 8-bit number.
Considering it the other way, it may also be seen as +256+ (–86) = +170!

(b) If the carry is considered as a sign bit of the result, and eight bits of the result are considered to be only magnitude,
   (i) If the result is to be used in further arithmetic operations, the number should be re-sized to 16 bits by copying the carry to all bits of upper byte, i.e. in this example, carry is 0 and eight bit result is 10101010, the resultant 16 bits represents 00000000 10101010, which is +170 in 16-bit signed representation.
   (ii) If the result is not to be used in further arithmetic operations, i.e. if it is the end result, the carry may be used to determine sign of the expected result. In this example, carry is 0, therefore, the sign of expected result is positive and consider all bits of the result as magnitude only, i.e. 10101010=170, thus, combining the sign and the magnitude, the result would be +170.
   Or simply considering carry as a 9th bit, result would be correct, i.e. 010101010=+170.

(c) Using the fact that overflow flag modifies (inverts) the sign bit; we should invert it to get correct sign of the expected result. In the example, the sign bit is 1, after inverting it, we get it as 0 and therefore the sign of the expected result is positive. Now since the sign is positive, treat eight bits of the erroneous result as a magnitude, i.e. 10101010=170, thus, combining the sign and the magnitude, the result would be +170.

Example 5.15

Add –70d with –80d.

Solution:

(–70d) =	1011 1010B	= BAH
+ <u>(–80d)</u> =	<u>1011 0000B</u>	= <u>B0 H</u>
–150d (expected)	1)0110 1010B	6AH
	=+106d (actual result)	

Here, result exceeds –128d, so answer is incorrect. There is carry from bit position 7 and no carry from bit 6, so OV=1 and C=1.

The result is erroneous in above example because both the original numbers were negative and therefore the result should have been negative, but the result we got is positive because of the excessive magnitude of the result (overflow) has modified (inverted) the sign bit.

Interpretation of the Result There are again three ways that the result may be interpreted.

(a) Since the overflow modifies (invert) the sign bit, therefore to interpret the result, let us change the sign of the erroneous result 6AH (**0**110 1010 b). After inverting the sign bit, the result will become EAH (**1**110 1010 b) = –22d. This may be considered as the actual result; it is –22 lower (because of the – sign) than –128, i.e.

–128d
+ <u>–022d</u>
–150d

It may also be seen as –256+ (+106) = –150!

(b) If the carry is considered as a sign bit of the result, and eight bits of the result are considered to be only magnitude,
  (i) If the result is to be used in further arithmetic operations, the number should be resized to 16 bits by copying the carry to all bits of upper byte, i.e. in this example carry is 1 and eight bit result is 10101010, the resultant 16 bits represents 11111111 01101010, which is –150 in 16-bit signed representation.
  (ii) If the result is not to be used in further arithmetic operations, i.e. if it is the end result,
  The carry may be used to determine sign of the expected result. In this example, carry is 1, therefore the sign of expected result is negative and consider all bits of the result as magnitude after taking its 2's complement, i.e. 2's complement of 01101010 is 10010110 =150, thus, combining the sign and the magnitude, the result would be –150.
  Or simply 101101010= –150 if it is treated as 9- bit signed number.

(c) Using the fact that overflow flag modifies (inverts) the sign bit; we should invert it to get correct sign of the expected result. In the example, the sign bit is 0, after inverting it we get it as 1; therefore, the sign of the expected result is negative. Now since the sign is negative, find 2's complement of the erroneous result to get the magnitude, i.e. 2's complement of 01101010, which is 10010110=150, thus, combining the sign and the magnitude the result would be –150.

Example 5.16

Add –20d with –25d.

Solution:

(–20d)	=	1110 1100B	=	ECH
+ (–25d)	=	1110 0111B	=	E7H
–45d		1)1101 0011B		D3H
		= –45d		

Here, C=1 and OV=0. The result is correct.

In conclusion, when two like signed numbers are added, and after addition if OV=0, the result is always a correct signed number by neglecting the carry, otherwise (if OV=1) the result is incorrect.

6. *Subtraction of* Unlike *Signed Numbers*

If two unlike numbers are subtracted, it is possible that result may exceed range of –128d to +127d. The situation is similar to adding the *like* numbers. Subtraction of unlike signed numbers is illustrated in Examples 5.17 and 5.18.

Example 5.17

Subtract +100d from –70d.

Solution:

(–070d)	=	1011 1010B	=	BAH
– (+100d)	=	– 0110 0100B	=	64H
– 170d (expected)		0) 0101 0110B		56H
		= +86d (actual result)		

There is borrow into the bit portion of 6 but not into bit7. OV=1 and C=0. Because OV=1, the result is incorrect.

The result is erroneous in above example because we subtract larger number from smaller one, the result should have been negative, but the result we got is positive because the excessive magnitude of the result (overflow) has modified (inverted) the sign bit.

Interpretation of the Result There are again three ways that the result may be interpreted.

(a) As we have seen that the overflow modifies (invert) the sign bit, therefore to interpret the result, let us change the sign of the erroneous result 56H (**0**101 0110 b). After inverting the sign bit, the result will become D6H (**1**101 0110 b) = –42d. This may be considered as the actual result; it is –42 lower (because of the – sign) than –128, i.e.

	–128d
+	–042d
	–170d

It may also be seen as –256+ (+86) = –170!

(b) In a microcontroller, the subtraction is actually performed by adding 2's complement of the subtrahend to the minuend. Because of this, our example may be represented as follows:

(–070d)	=	1011 1010B	= BAH
+ (–100d)	=	+ 1001 1100B	= 9CH
–170d(expected)		1) 0101 0110B	56H
		= +86d (actual result)	

If the carry is considered as a sign bit of the result, and eight bits of the result are considered to be only magnitude,

(i) If the result is to be used in further arithmetic operations, the number should be resized to 16 bits by copying the carry to all bits of upper byte, i.e. in this example, carry is 1 and eight bit result is 01010110, the resultant 16 bits represents 11111111 01010110, which is –170 in 16-bit signed representation.

(ii) If the result is not to be used in further arithmetic operations, i.e. if it is the end result, the carry may be used to determine sign of the expected result. In this example, carry is 1, therefore the sign of expected result is negative and consider all bits of the result as magnitude after taking its 2's complement, i.e. 2's complement of 01010110 is 10101010 =170, thus, combining the sign and the magnitude the result would be –170.

Or simply, 101010110 = –170 if it is treated as 9-bit signed number.

(c) Using the fact that overflow flag modifies (inverts) the sign bit; we should invert it to get the correct sign of the expected result. In the example, the sign bit is 0, after inverting it we get it as 1; therefore, the sign of the expected result is negative. Now since the sign is negative, find 2's complement of the erroneous result to get the magnitude, i.e. 2's complement of 01010110, which is 10101010=170, thus, combining the sign and the magnitude the result would be –170.

Example 5.18

Subtract –50 from +80.

Solution:

(+80d)	=	0101 0000	=50H
– (–50d)	=	1100 1110	=CEH
+130d (expected)		1) 1000 0010	82H
		= –126d (actual result)	

There is a borrow into bit 7 and no borrow into bit 6. Therefore OV=1 and C=1.The result may be interrupted in a ways similar to above examples.

7. *Subtraction of* Like *Signed Numbers*

The situation here is similar to adding *unlike* numbers. The result will be always correct, i.e. always within the range –128 to +127. So magnitude and sign of the result need not be adjusted.

Subtraction of like signed numbers is illustrated in Examples 5.19 and 5.20.

Example 5.19

Subtract +120d from +101d.

Solution:

(+101d)	=	0110 0101B	= 65H
– (+120d)	=	0111 1000B	= 78H
–19d		1)1110 1101B	EDH
		= –19d	

Here OV=0 and C=1, neglecting carry, the result is correct.

Example 5.20

Subtract –116d from –61H.

Solution:

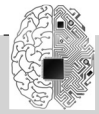
```
(–061d) =      1100 0011B   = C3H
– (–116d) =    1000 1100B   = 8CH
  +55d       0)0011 0111B     37H
                  =+55d
```

Here, OV=0 and C=0. The result is correct.
In conclusion, when two like signed numbers are subtracted, the result is always a correct signed number by neglecting the carry.

From the above discussion, we can make general note that if OV flag is set to 1, the result is incorrect as it is outside the range –128d to +127d.The OV=1 also suggests to complement the sign bit of the result to interpret the result.

8. Recovering a Result from Overflow

If we want to recover the result as well as use it in further arithmetic, the '(i)' part of the second method (to interpret the result) discussed above may be implemented easily in a program because its logic is same for all types of operations. However, if we want only to recover the result, i.e. if the result obtained is the end result and it is not to be used in further arithmetic, the '(ii)' part of the second method '(b)' may be used to implement the program.



THINK BOX 5.2

Show how the erroneous result obtained in signed arithmetic can be recovered and displayed in decimal on the display device (like LCD or monitor).

Overflow flag indicate the erroneous result, i.e. if OV=1, perform the following steps to recover a result.
Use the second method '(b)', ('(ii)' part of that) to interpret the result and write a program as follows:

- If carry is 0, send ASCII of '+' to the display device, treat the 8 bits of the original result as magnitude, convert it to BCD, then into ASCII.
- If carry is 1, send ASCII of '–' to the display device, find 2's complement of the original result and treat it as magnitude.

To avoid an overflow, a programmer has to predict the largest possible result and choose the size of the numbers accordingly, i.e. to support the larger numbers in signed operations one should go for 16-bit signed numbers (or even multi-byte signed numbers as per requirements).

THINK BOX 5.3

Why do we use same logic for addition (or subtraction) of unsigned and signed (2's complement) binary numbers?
Because rules for binary addition (or subtraction) remain same for all type of representations of a binary number. Computers understand only 0s and 1s. It is how we interpret the binary number that makes the difference in the meaning conveyed by them. For example, 1101 represent 11D for unsigned representation and –5D for signed representation. Following example shows how same binary numbers are interpreted in different ways for signed and unsigned representations.

```
  0110  (=6)      0110 (=  6)
+ 1011 (=11)   + 1011 (= –5)
1 0001 (=17)    1 0001 (=  1)
```

THINK BOX 5.4

Can we perform the subtraction operation without SUBB instruction? If yes, How?
Yes. Find two's complement of subtrahend and add it to minuend.

THINK BOX 5.5

How is multi-byte signed arithmetic performed?
Signed numbers are similar to the unsigned numbers except for the most significant bit of a most significant byte; therefore, lower bytes are treated as an unsigned number and overflow is checked only for most significant byte.

5.1.4 Decimal (Binary Coded Decimal—BCD) Arithmetic

Binary representation of decimal digits (0 to 9) is called binary coded decimal. BCD numbers are needed because humans prefer to use decimal number system (digits 0 to 9). Four bits are required to represent the decimal number from 0 to 9 as shown in Table 5.4.

Table 5.4 *BCD codes for decimal digits 0 to 9*

Decimal Digit	BCD Code
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

The terms *unpacked* and *packed BCD numbers* are widely used with the BCD numbers. In *unpacked BCD numbers,* the lower four bits of the number are used to represent the decimal digit and the upper four bits are 0. In *packed BCD numbers* two BCD numbers are placed (packed) into a single byte.

Examples of unpacked and packed BCD numbers are shown below.

```
0000 0001 ⟶ unpacked BCD for 1
0000 0101 ⟶ unpacked BCD for 5
   ↓    └──⟶ BCD code of number
Upper nibble
  is zero

0110 1000    packed BCD for 68
1000 0010    packed BCD for 82
  ↓    ↓
Both nibbles are used
```

The 8051 performs all the operations in pure binary. When BCD numbers are used in arithmetic, the result may be non-BCD as shown in Example 5.21.

Example 5.21

Explain the process of BCD addition.

Solution:

Consider addition of two BCD numbers 38 and 24.

```
  38 d     0011 1000 BCD     38H
+ 24 d     0010 0100 BCD     24H
  62 d     0101 1100         5CH
```

In the above addition, when two BCD numbers are added (two operands are assumed to be BCD), the result is 5C, which is non-BCD number because, in the BCD numbers we can use only 0 to 9 (0000 to 1001). The correct result should have been 62d.To correct the result, programmer must adjust result by adding 6 (0110) to the lower digit, i.e.

```
  5C
 +6
  62
```

A similar problem could have occurred in the upper digit of the result while performing addition. For example,

```
  72 d     0111 0010 BCD     72H
+ 41 d     0100 0001 BCD     41H
 113 d     1011 0011 BCD     B3H
```

Again, to get the correct result (72+41=113), we should add 6 to the upper digit, i.e.

```
  B3
+ 6
 113
```

This problem in above example is common while working with the BCD numbers. Therefore, the 8051 has instruction to adjust the BCD result automatically. Instruction DA A is there to correct the BCD addition problem discussed above.

> ***Note:*** Microcontrollers understand only binary numbers, they cannot differentiate between binary and BCD numbers, and it is the programmer who assumes the number to be binary or BCD and treats them accordingly.

DA A Decimal Adjust Accumulator for Addition

DA A works only on the contents of register A, it must be kept in mind that DA A must be used after the addition of BCD numbers and BCD numbers can never have any digit greater than 9. In other words, A to F digits are not allowed. The DA A instruction works only when used after ADD or ADDC instructions and does not give correct answer after SUBB, MUL, DIV or INC operations.

Operation of DA A DA A instruction performs the following operations. After ADD or ADDC instruction,

1. If lower nibble is greater than 9 or if AC=1, add 6 to lower nibble (4 bits)
2. If upper nibble is greater than 9 or if CY=1, add 6 to upper nibble.

The operation of DA A instruction for above-mentioned different conditions is illustrated in Example 5.22.

Example 5.22

Explain with suitable example the operations performed by DA A instruction when
(i) Lower nibble of A is greater than 9 after addition
(ii) AC=1 after addition
(iii) Upper nibble of A is greater than 9 after addition
(iv) C=1 after addition

Solution:

(i) When the lower nibble is greater than 9 after addition,

28 BCD	0010 1000	
+ 12 BCD	0001 0010	
40 BCD	0011 **1010**	3**A**; lower nibble is greater than 9 (invalid BCD)
	+ 0000 0110	add 6 to lower nibble (DA A will do it)
	0100 0000	40 BCD, the desired result

(ii) When AC=1 after addition,

28 BCD	0010 1000	
+ 19 BCD	0001 1010	
47 BCD	0100 0001	41; AC is 1 after addition (invalid result)
	+ 0000 0110	add 6 to lower nibble (DA A will do it)
	0100 0111	47 BCD, the desired result

(iii) When upper nibble is greater than 9,

82 BCD	1000 0010	
+ 21 BCD	0010 0001	
103 BCD	1010 0011	**A**3; upper nibble greater than 9 (invalid BCD)
	+ 0110 0000	add 6 to upper nibble (DA A will do it)
	1 0000 0011	103 BCD, the desired result

(iv) When CY flag is set after addition,

82 BCD	1000 0010	
+ 91 BCD	1001 0001	
173 BCD	**1** 0001 0011	carry flag set after addition (Invalid result)
	+ 0110 0000	add 6 to upper nibble (DA A will do it)
	1 0111 0011	173 BCD, the desired result

It is also possible to have both nibbles as non-BCD after addition as shown.

63 BCD	0110 0011	
+ 88 BCD	1000 1000	
151 BCD	1110 1011	**EB**; both nibbles greater than 9 (invalid BCD)
	+ 0110 0110	add 6 to both nibbles DA A will do it)
	1 0101 0001	151 BCD, the desired result

The AC flag is useful for DA A instruction only. It has no other use to the programmer.

Example 5.23

Illustrate the use of DA A instruction with a suitable example.

Solution:

Consider addition of two BCD numbers 28 and 12.

```
MOV A, #28H          // add 28 and 12
ADD A, #12H          // A= 3AH; non BCD
DA A                 // adjust the result in A to BCD
                     // A= 40
```

Note that the two 8-bit numbers added are assumed to be BCD. The result after addition (3AH) is non-BCD; therefore, to get correct BCD result, the DA A instruction is used.

Example 5.24

Discuss the operation performed by DA A instruction for the following.

```
MOV A, #99H
ADD A, #01H
DA A
```

Solution:

The DA A instruction will add 66H to the result after addition, i.e. 9AH+66H=100H

5.1.5 Multiplication

The 8051 supports 8-bit integer multiplication. Multiplication instruction uses only A and B registers as both source and destination for the operation. The bytes in A and B are assumed to be unsigned. Format of multiply instruction is as follows:

```
MUL AB          // multiply contents A with B; put the lower byte of result in
                // A and higher byte of result in B.
```

Use of OV in Multiplication

The overflow will be set t to 1, if A × B > FFH. **Here, the OV flag does not indicate that an error has occurred**. But, it shows that the result is larger than 8 bits and we need to consider B register for higher byte of the result. Example 5.25 shows the use of multiply instruction.

Discussion question Which flags are affected by the multiply instruction?

Answer The multiply instruction always clears the carry flag to zero and it will set the overflow flag if the result after multiply instruction is greater than 0FFH, else it will clear the overflow flag.

Example 5.25

Multiply the contents of R0 and R1 and place the result into R2 (MSByte) and R3 (LSByte).

Solution:

Let us consider multiplication of largest 8-bit numbers, i.e. assume that R0=FFH, R1=FFH

```
MOV A, R0       // A= FF
MOV B, R1       // B= FF, operands can only be in A and B
MUL AB          // A x B=FE01, B=FE, A=01, OV=1 to indicate that the result is greater than 8 bits
MOV R3, A       // store result LSB into R3
MOV R2, B       // store result MSB into R2
```

Repeat the above program for R0=05H, R1=04H

```
MOV A, R0       // A= 05
MOV B, R1       // B= 04
MUL AB          // A x B=0014, B=00, A=14, OV=0 to indicate that result is only 8 bits
MOV R3, A       // store result LSB into R3
MOV R2, B       // store result MSB into R2; may be ignored for this case
```

Example 5.26

Write a program to find the square of a number stored at internal RAM address 50H. Store the result at address 60H (LSByte) and 61H (MSByte). If the number is AAH, what will be the result and status of OV flag after finding the square of that number?

Solution:

The square of a number is found by multiplying the number with itself. In the 8051, multiply instruction require operands in A and B only; therefore we need to copy the number into A and B registers and then we will use the multiply instruction.

```
MOV A, 50H      // copy the number at address 50H into A
MOV B, A        // copy the same number into B
MUL AB          // find the square by multiplication
MOV 60H, A      // copy the result (LSByte) into address 60H
MOV 61H, B      // copy the result (MSByte) into address 61H
```

If the number is AA, then result will be 70E4H. Since result is greater than FFH the overflow flag will be set, i.e. OV=1 after multiplication.

5.1.6 Division

The 8051 supports 8-bit integer division. The divide instruction uses only A and B registers as both source and destination for the operation. The number in A is divided by number in B. Quotient (result) is placed in A and remainder is placed in B, again both numbers are assumed to be unsigned. Format of divide instruction is as follows,

```
DIV AB      // divide A by B, store the result in A and remainder in B
```

Use of OV in Division The OV flag is set to 1 to show that an attempt to divide by zero has been made, i.e. contents of B=00 before division. The contents of A and B are undefined when division by 0 is attempted.

Discussion Question Which flags are affected by the divide instruction?

Answer The divide instruction always clears the carry flag to zero and it will set the overflow flag if a divisor is 00H, else it will clear the overflow flag.

Example 5.27

Write instructions to divide 95 by 10.

Solution:

```
MOV A, #95      // A=95d = 5FH, number without any suffix is decimal number
MOV B, #10      // B=10d =0AH
DIV AB          // A/B, result is A=9 (quotient), B=5(remainder)
```

> ***Note:*** The original contents of A and B are lost in both multiply and divide instructions.

Example 5.28

Show the status of OV flag and contents of A and B after execution of the following instructions.

```
MOV A, #31H
MOV B, #03H
DIV AB
```

Solution:

The operation performed by the above program is $\frac{31H}{03H} = \frac{49d}{03d}$. Therefore, the result will be, A= 10H (Quotient) and B= 01H (Remainder). The OV=0, because no attempt has been made to divide by zero.

5.1.7 Increment and Decrement

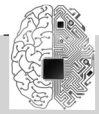
These are the most simple and special instructions for addition and subtraction. They are used to add 1 or subtract 1 from a specified operand. All addressing modes are supported. No flags are affected. These operations provide powerful way to repeat the operations when used with program flow control instructions to, i.e. increment or decrement until the desired result is obtained. The format of these instructions is as follows,

```
INC destination     // add one to the destination operand
DEC destination     // subtract one form the destination operand
```

The formats of increment and decrement instructions for different addressing modes are as follows:

INC A	// Increment the contents of A by 1, A=A+1
INC A	// A=A+1, If A=10H → A=11H
INC *Rn*	// Increment the contents of Rn by 1, *Rn*= *Rn*+1
INC R3	// R3= R3+1, If R3=1AH → R3=1BH
INC *@Ri*	// Increment the contents of address pointed by *Ri* by 1, (*Ri*)= (*Ri*) +1
INC @R0	// (R0)= (R0) +1, If R0=10H, (10H)=55H → (R0)=(10H)=56H
INC *direct*	// Increment the contents of direct address by 1, (direct)=(direct) +1
INC 40H	// (40H)=(40H)+1, If (40H)=1FH → (40H)=20H
INC DPTR	// Increment the contents of DPTR by 1, DPTR=DPTR+1
INC DPTR	// DPTR=DPTR+1, If DPTR=1000H → DPTR=1001H
DEC A	// Decrement the contents of A by 1, A=A–1
DEC A	// A=A–1, If A=10H → A=0FH
DEC *Rn*	// Decrement the contents of Rn by 1, *Rn*= *Rn*–1
DEC R3	// R3= R3–1, If R3=1AH → R3=19H
DEC *@Ri*	// Decrement the contents of address pointed by *Ri* by 1,(*Ri*)= (*Ri*) –1
DEC @R0	// (R0)= (R0) –1, If R0=10H, (10H)=55H → (R0)=(10H)=54H
DEC *direct*	// Decrement the contents of *direct* address by 1 (direct)=(direct) –1
DEC 40H	// (40H)=(40H) –1, If (40H)=1FH → (40H)=1EH

It should be noted that there is no DEC DPTR instruction.



THINK BOX 5.6

What is the difference between following two instructions?

- **INC A**
- **INC ACC**

Both instructions will do the same operation. But, INC A instruction is assembled as a 04 (1-byte instruction) and INC ACC is assembled as 05 E0 (2-byte instruction, direct addressing mode '**I**NC *direct*')

Example 5.29

Illustrate how increment and decrement instructions modify the operands.

Solution:

The following instructions illustrate how increment and decrement instructions modify the operands.

```
MOV A, #0FFH        // A= FFH
MOV R5, #10H        // R5= 10H
MOV R4, #10         // R4= 10d= 0AH
MOV R0, # 20H       // R0= 20H
MOV 20H, A          // (20H)= FFH
INC A               // A= FFH +1=00
DEC R5              // R5= 0FH
DEC R4              // R4= 09
DEC @ R0            // (20H)= FEH
INC 20H             // (20H)= FFH
INC 20H             // (20H)= 00H
DEC 20H             // (20H)=FFH
```

Observe that, when data FF is incremented by 1, it results into 00, i.e. 8-bit data overflow from FF to 00, it is modulo 8-bit operation, But remember that it (INC and DEC) does not affect any flag.

Example 5.30

Find the contents of the operands after execution of each of the following instructions.

```
MOV DPTR, #0FFFFH
DEC DPL
INC DPH
INC DPTR
DEC DPH
INC DPTR
```

Solution:

```
MOV DPTR, #0FFFFH      // DPTR=FFFFH
DEC DPL                // DPL=FEH
INC DPH                // DPH= 00H
INC DPTR               // DPTR=00FFH
DEC DPH                // DPH=FFH
INC DPTR               // DPTR=0000H
```

Note that DPTR overflow from FFFF to 0000.

Example 5.31

Find the contents of the destination operand after execution of each of the following instructions.

```
MOV R5, #10H
INC R5
INC R5
MOV R0, #20H
MOV A, #0FFH
MOV 20H, A
INC @ R0
INC A
MOV 20H, #00H
INC 20H
```

Solution:

```
MOV R5, #10H       // R5= 10H
INC R5             // R5= 11H
INC R5             // R5= 12H
MOV R0, #20H       // R0= 20H
MOV A, #0FFH       // A= FFH
MOV 20H, A         // (20H) = FFH
INC @ R0           // (20H) = 00H
INC A              // A=00H
MOV 20H, #00H      // (20H) = 00H
INC 20H            // (20H) =01H
```

Example 5.32

What should be the initial value of A if the value of A after execution of the following instructions is 00H?

```
MOV R0, A
SUBB A, R0
INC A
```

Solution:

A should have any value between 00H-FFH and CY=1.

To have value 00 after INC A, the value of A before execution of INC A (or after SUBB A, R0) should be FFH. Now, since values of A and R0 are same because of the first instruction, the carry (borrow) flag should be 1 to have value of A=FF after execution of SUBB instruction.

5.2 LOGICAL INSTRUCTIONS

Logic means evaluating the conditions using reasonable thinking and making a decision based on the evaluation. Binary logic deals with data that takes only two discrete values as being true or false (or, yes or no) and operations on such data. There are three basic logical operations: AND, OR and NOT. Many other logical operations are derived from these basic operations.

The 8051 supports logical operations such as AND, OR, EX-OR and NOT. These operations can be performed on a byte or on a single bit at a time. Single-bit operations are useful mostly in machine control applications where we need to monitor and control binary events such as to turn ON or OFF a device, to read status of an input switch.

5.2.1 Byte Operations

The operation specified in an instruction is performed on all 8 bits in a byte. The general format for these instructions are as follows:

ANL *destination, source*	// *destination* =*destination* AND *source*
ORL *destination, source*	// *destination* =*destination* OR *source*
XRL *destination, source*	// *destination* =*destination* EX-OR *source*

Note that these are all bit-wise operations, i.e.

destination bit *Dn* = destination bit *Dn* **OPERATION** source bit *Dn*

All four addressing modes can be used for the *source* operand. The A or a *direct* address in internal RAM can be the *destination*. For example,

1. AND Operation

The format of AND instruction for different addressing modes is given below:

ANL A, *#data*	// bitwise AND operation of A with *data*, A=A AND *data*
ANL A, #10H	// If A= FFH, → A= FFH AND 10H= 10H
ANL A, *Rn*	// bitwise AND operation of A with Rn, A=A AND *Rn*
ANL A, R0	// If A=55H, R0= 4AH, → A=55H AND 4AH= 40H
ANL A, *@Ri*	// bitwise AND operation of A with data pointed by Ri, A=A AND *(Ri)*
ANL A, @R1	// If A= 38H, R1=20H, (20H) =1FH, → A= 38H AND 1FH= 18H
ANL A, *direct*	// bitwise AND operation of A with data in *direct*, A=A AND (*direct*)
ANL A, 20H	// If A= E6H, (20H)=67H, → A=E6H AND 67H= 66H
ANL *direct, #data*	// bitwise AND operation of data in address *direct* with immediate *data* *// (direct) = (direct)* AND *data* (store result in address *direct*)
ANL 30H, #0E6H	// If (30H) = 0FH, → (30H) = 0FH AND E6H= 06H
ANL *direct*, A	// bitwise AND operation of data in address *direct* with A *(direct) = (direct)* AND A
ANL 10H, A	// If (10H) = 72H, A= 0FH, → (10H) = 72H AND 0FH= 02H

Example 5.33

Illustrate the use of ANL instruction.

Solution:

```
MOV A, #45H      A= 45H
ANL A, #0EH      A= 45H AND 0EH =04H
```

The operation of ANL instruction is explained below:

```
AND  45 H   01000101
     0EH    00001110
     04H    00000100
```

The AND operation (ANL instruction) is commonly used to mask (set to 0) certain bits of a result as shown in above example (see highlighted bits)

2. OR Operation

The format of OR instruction for different addressing modes is given below,

ORL A, *#data*	// bitwise OR operation of A with *data*, A=A OR *data*
ORL A, #10H	// If A= FFH, → A= FFH OR 10H= FFH
ORL A, *Rn*	// bitwise OR operation of A with *Rn*, A=A OR *Rn*
ORL A, R0	// If A = 55H, R0= 8AH, → A=55H OR 8AH= DFH
ORL A, *@Ri*	// bitwise OR operation of A with data pointed by *Ri*, A=A OR *(Ri)*
ORL A, @R1	// If A= 38H, R1=20H, → (20H) =1FH, A= 38H OR 1FH= 3FH
ORL A, *direct*	// bitwise OR operation of A with data in *direct*, A=A OR (*direct*)
ORL A, 20H	// If A= E6H, (20H) =67H, → A=E6H OR 67H= E7H
ORL *direct, #data*	// bitwise OR operation of data in address *direct* with immediate *data* *// (direct) = (direct)* OR *data* (store result in address *direct*)
ORL 30H, #0E6H	// If (30H) = 0FH, → (30H) = 0FH OR E6H= EFH
ORL *direct*, A	// bitwise OR operation of data in address *direct* with A *// (direct) = (direct)* OR A
ORL 10H, A	// If (10H) = 72H, A= 0FH, → (10H) = 72H OR 0FH= 7FH

Example 5.34

Illustrate the use of ORL instructions.

Solution:

```
MOV A, #34H     A= 34H
ORL A, #57H     A= 34H OR 57H =77H
    OR  34 H    00110100
        57 H    01010111
        77 H    01110111
```

The OR operation (ORL instruction) is commonly used to set certain bits of a result to 1 as shown in above example (see highlighted bits).

3. EX-OR Operation

The format of EX-OR instruction for different addressing modes is given below:

XRL A, *#data*	// bitwise X-OR operation of A with *data*, A=A X-OR *data*
XRL A, #10H	// If A= FFH, → A= FFH X-OR 10H= EFH
XRL A, *Rn*	// bitwise X-OR operation of A with Rn, A=A X-OR *Rn*
XRL A, R0	// If A = 55H, R0= 8AH, → A=55H X-OR 4AH= DFH
XRL A, *@Ri*	// bitwise X-OR op. of A with data pointed by Ri, A=A X-OR *(Ri)*
XRL A, @R1	// If A= 38H, R1=20H, (20H) =1FH, → A= 38H X-OR 1FH= 27H
XRL A, *direct*	// bitwise X-OR operation of A with data in *direct*, A=A X-OR (*direct*)
XRL A, 20H	// If A= E6H, (20H) =67H, → A=E6H X-OR 67H= 81H
XRL *direct, #data*	// bitwise X-OR operation of data in address *direct* with *data* *// (direct) = (direct)* X-OR *data* (store result in address *direct*)

XRL 30H, #0E6H	// If (30H) = 0FH, → (30H) = 0FH X-OR E6H= E9H
XRL *direct*, A	// bitwise X-OR operation of data in address *direct* with A // *(direct)* = *(direct)* X-OR A
XRL 10H, A	// If (10H) = 72H, A= 0FH, → (10H) = 72H X-OR 0FH= 7DH

Example 5.35

Illustrate the use of XRL instructions.

Solution:

```
MOV A, #0A4H        A= A4H
XRL A, #71H         A= A4H EX-OR 71H =D5H

EX-OR  A4 H         1010 0100
       71 H         0111 0001
       D5 H         1101 0101
```

The EX-OR operation (XRL instruction) is often used to invert certain bits of an operand, i.e. if any bit is EX-ORed with 1, it will be inverted (see highlighted bits). EX-OR operation may also be used to see if two registers (or two bits) have the same value. If two bits of same value are EX-ORed, the result will be always zero. We can use this result along with decision making instructions to take appropriate action.

No flags are affected by byte level logical instructions, expect when destination operand is PSW (direct address).

Example 5.36

Write single instruction for each of the following operat

(i) Clear bits 0,2,3,6 of the A

(ii) Set bits 0,1,5 of the contents of the address 20H

(iii) Complement bits 2,3,4,7 of the A

Make sure that other bits are not disturbed.

Solution:

(i) ANL A, #0B2H
(ii) ORL 20H, #23H
(iii) XRL A, #9CH

4. Logical Operations with Ports

When destination of the logical instruction is port SFR, the latch register (port structure will be discussed in Chapter 13) will be used as both source of data and destination to store the result. In such instructions, the port pins are not read. Consider the following instruction,

```
MOV P1, #0FFH     // port 1 latch =FFH
```

Assume that port P1 pins are connected to base of transistors (each pin with the different transistor) and the above instruction is executed. Since latch contains all 1's, the transistors will be ON and bases of all the transistors will be near to ground level (0.7 volts), therefore P0 pins will be at low (0) level, i.e. port pins will be at logic level 0 even though port latch is at level 1. Now consider instruction,

```
ANL P1, #0F0H
```

In above instruction P0 is initially source of data, so latch of P1 (FFH) is read, and then it will be logically ANDed with immediate value F0H and then result (F0H) will be written back to P0 latch register. For above instruction, if pins were read then result would have been 00 (00 AND 0F) which is incorrect. This issue is discussed in more detail in Section 13.2.

When an instruction uses port as a source, but not as a destination, microcontroller reads port pins (as the source of data) instead of port SFR. For example,

```
ANL A, P1
```

This instruction will AND A with contents of P1 pin (00) for above example, result will be 00 in the Accumulator.

All logical operations discussed till now had two operands, and that's why they are referred as binary operations. There are few instructions which require only one operand. They are referred as *unary operations*.

5.2.2 Unary Operations

Unary operations require single operand thus the operations are performed on a single operand. The source as well as destination for these operations is Accumulator.

1. Clear: The clear instruction is used to clear the contents of Accumulator, the format of this instruction is

```
CLR A          // clear accumulator, A=00H
```

2. Complement (NOT operation): Generally, complement is used to generate 1's complement of the data in an accumulator, i.e. All 0s will be replaced by 1 and 1s will be replaced by 0.

```
CPL A          // complement A, A= Ā
```

Example 5.37

What will be contents of A, after execution of following instructions?

```
MOV A, # 54H
CPL A
```

Solution:

These instructions will find complement of the number 54H.

```
MOV A, # 54H          // A= 54H = 0101 0100 b
CPL A                 // A= ABH = 1010 1011 b
```

The CPL A instruction will complement the contents of A. Therefore, A= ABH after execution of given instructions.

3. Rotate: Rotate operations are useful for monitoring bits of a data byte without using a logical test. The status of bits may be used in decision-making process for certain applications. The rotation can be 1 bit in left or right direction, with or without including carry flag in the rotation. The rotate instruction works only with the Accumulator. Total of nine bits are involved in the rotation operation when carry flag is included and eight bits when carry is not included. Usually, a carry flag is included in an operation when decision making is required because JC (jump if carry) and JNC (jump if no carry) are the instructions which makes a decision based on value of the carry flag. It is also used to perform rotation on multi-byte data. Rotate instructions may also be used to convert parallel data to serial data. The 8051 has four different rotate instructions as described in the following section.

(a) Rotate Accumulator Left by One Bit

```
RL A      // rotate A one bit position to the left, bit D0 to D1, bit
          D1 to D2, …, bit D6 to D7 and bit D7 to D0 as
          illustrated in Figure 5.3.
```

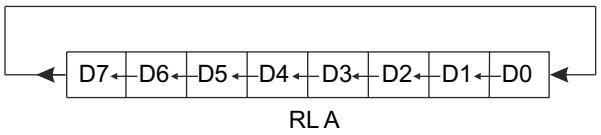


Fig. 5.3 *RL A instruction*

For example,

```
MOV A, #43H       // A= 0100 0011
RL A              // A=1000 0110
RL A              // A=0000 1101
```

(b) Rotate Accumulator Right by One Bit

```
RR A      // rotate A one bit position to the right, bit D0 to D7,
          bit D7 to D6, …, bit D2 to D1 and bit D1 to D0 as
          illustrated in Figure 5.4.
```

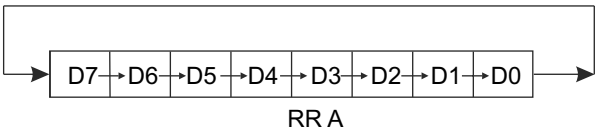


Fig. 5.4 *RR A instruction*

For example,

```
MOV A, #35H       // A= 0011 0101
RR A              // A= 1001 1010
```

(c) Rotate Accumulator Left through Carry by One Bit

```
RLC A     // rotate A one bit position to the left through carry flag, bit D0 to D1, bit D1 to D2, …, bit D6 to D7,
          // bit D7 to CY and CY to D0 as illustrated in Figure 5.5.
```

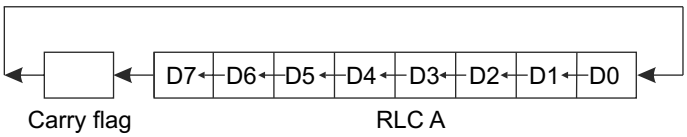


Fig. 5.5 *RLC A instruction*

For example,

```
CLR C              // CY=0
MOV A, #0D6H       // A= 1101 0110, CY=0
RLC A              // A= 1010 1100, CY=1
```

(d) Rotate Accumulator right through Carry by One Bit

RRC A // rotate A one bit position to the right through carry flag, bit D0 to CY, CY to D7, bit D7 to D6, …, bit // D2 to D1 and bit D1 to D0 as illustrated in Figure 5.6.

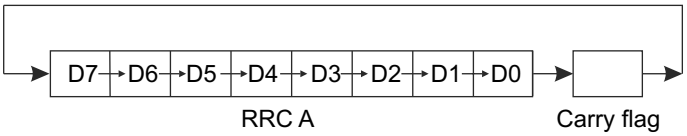


Fig. 5.6 *RRC A instruction*

For example,

```
SETB C             // CY=1
MOV A, #62H        // A= 0110 0010, CY=1
RRC A              // A= 1011 0001, CY=0
```

4. **SWAP:** SWAP A instruction swaps the nibbles of register A, i.e. it interchanges the upper nibble with the lower nibble of A. This operation is equivalent to 4-bit rotation in either left or right direction. It works only with the A register. The operation of swap instruction is illustrated in Figure 5.7.

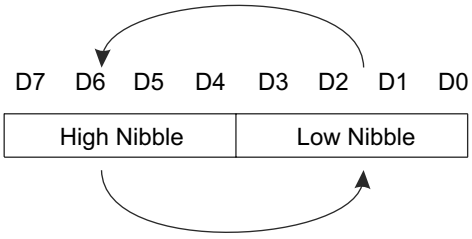


Fig. 5.7 *SWAP instruction*

For example,

```
A=53H (before execution)
SWAP A
A=35H (After execution)
MOV A, #53H      //A=53H
SWAP A           // A=35H
```

THINK BOX 5.7

Realize 'SWAP A' instruction using rotate instructions.
SWAP A can be realized by four RL A (or RR A) instructions.

Example 5.38

If A= 35H and C=1 before execution of the following instructions, write contents of destination operand and C after execution of each instruction.

```
RR A
RR A
RLC A
SWAP A
CPL C
RL A
SWAP A
RRC A
```

Solution:

Initially, A=35H=00110101b, C=1

```
RR A               // A=10011010b, C=1
RR A               // A=01001101b, C=1
```

```
RLC A          // A=10011011b, C=0
SWAP A         // A=10111001b, C=0
CPL C          // A=10111001b, C=1
RL A           // A=01110011b, C=1
SWAP A         // A=00110111b, C=1
RRC A          // A=10011011b, C=1
```

3. Summary of Arithmetic and Logical Instructions

Arithmetic and logical instructions are summarized in Tables 5.5 and 5.6 respectively.

Table 5.5 *Arithmetic instructions with examples*

Mnemonics	Operation	Addressing Modes			
		Direct	Indirect	Register	Immediate
ADD A, **<BYTE>**	A= A + **<BYTE>**	ADD A, **direct**	ADD A,**@Ri**	ADD A,**Rn**	ADD A,**#data**
		ADD A, **12H**	ADD A,**@R1**	ADD A,**R4**	ADD A,**#09H**
ADDC A, **<BYTE>**	A= A + **<BYTE>** + C	ADDC A, **direct**	ADDC A,**@Ri**	ADDC A,**Rn**	ADDC A,**#data**
		ADDC A, **12H**	ADDC A,**@R1**	ADDC A,**R4**	ADDC A,**#10H**
SUBB A, **<BYTE>**	A= A – **<BYTE>** – C	SUBB A, **direct**	SUBB A,**@Ri**	SUBB A,**Rn**	SUBB A,**#data**
		SUBB A, **12H**	SUBB A,**@R0**	SUBB A,**R7**	SUBB A,**#25H**
INC A	A= A + 1	Accumulator only			
INC <BYTE>	<BYTE>= <BYTE> + 1	INC **direct**	INC **@Ri**	INC **Rn**	
		INC **12H**	INC **@R1**	INC **R6**	
INC DPTR	DPTR= DPTR + 1	Data Pointer only			
DEC A	A= A – 1	A only			
DEC <BYTE>	<BYTE>= <BYTE> – 1	DEC **direct**	DEC **@Ri**	DEC **Rn**	
		DEC **35H**	DEC **@R0**	DEC **R3**	
MUL AB	B:A= B*A	A & B only			
DIV AB	A= Int [A/B]; B= Mod [A/B]	A & B only			
DA A	Decimal Adjust	A only			

Table 5.6 *Logical instructions with examples*

Mnemonics	Operation	Addressing Modes			
		Direct	Indirect	Register	Immediate
ANL A, <BYTE>	A= A AND **<BYTE>**	ANL A, **direct**	ANL A,**@Ri**	ANL A,**Rn**	ANL A,**#data**
		ANL A, **12H**	ANL A,**@R1**	ANL A,**R0**	ANL A,**#10H**
ANL <BYTE>, A	<BYTE>= <BYTE> AND **A**	ANL direct**, A**			
		ANL 10H, **A**			
ANL <BYTE>, #data	<BYTE>= <BYTE> AND **#data**	ANL direct**, #data**			
		ANL 10H, #**20H**			
ORL A, <BYTE>	A= A OR **<BYTE>**	ORL A, **direct**	ORL A,**@Ri**	ORL A,**Rn**	ORL A,**#data**
		ORL A, **12H**	ORL A,**@R0**	ORL A,**R2**	ORL A,**#10H**
ORL <BYTE>, A	<BYTE>= <BYTE> OR **A**	ORL direct, **A**			
		ORL 10H, **A**			
ORL <BYTE>, #data	<BYTE>= <BYTE> OR **#data**	ORL direct, **#data**			
		ORL 10H, #**20H**			

(*contd.*)

(**Table 5.6** *contd.*)

Mnemonics	Operation	Addressing Modes			
		Direct	Indirect	Register	Immediate
XRL A, <BYTE>	A= A XOR **<BYTE>**	XRL A, **direct**	XRL A,@**Ri**	XRL A,**Rn**	XRL A,**#data**
		XRL A, **12H**	XRL A,@**R0**	XRL A,**R2**	XRL A,**#25H**
XRL <BYTE>, A	<BYTE>= <BYTE> XOR **A**	XRL direct, **A**			
		XRL 10H, **A**			
XRL <BYTE>, #data	<BYTE>= <BYTE> XOR **#data**	XRL direct, **#data**			
		XRL 10H, #**20H**			
CLR A	A= 00H	Accumulator only			
CPL A	A= NOT A	Accumulator only			
RL A	Rotate A Left 1 bit	Accumulator only			
RLC A	Rotate A Left 1 bit through Carry	Accumulator only			
RR A	Rotate A Right 1 bit	Accumulator only			
RRC A	Rotate A Right 1 bit through Carry	Accumulator only			
SWAP A	Swap nibbles in A	Accumulator only			

POINTS TO REMEMBER

- The 8051 microcontroller supports basic arithmetic and logical operations such as addition, subtraction, division, multiplication, increment, decrement, AND, OR, NOT and EX-OR.
- The arithmetic operations modify arithmetic flags: carry (CY), overflow (OV) and auxiliary carry (AC).
- The register A is always destination as well as one of the source operand in the addition and subtraction operations. It is always one of the source and destination operand in a multiplication and division.
- Subtraction instruction treats the carry flag as borrow and always subtract carry flag as a part of an operation.
- Negative numbers are not represented in true binary form, but they are represented in 2' complement form.
- The CY and OV flags are there to handle unsigned and signed operations. CY is generally used in unsigned arithmetic while OV is used in signed addition and subtraction.
- Single byte sized signed number may range from –128d (1000 000b) to +127d (0111 1111b).
- In unpacked BCD numbers, the lower four bits of the number are used to represent the decimal digit and the upper four bits are 0. In packed BCD numbers, two BCD numbers are placed (packed) into single byte.
- Microcontrollers understand only binary numbers. They cannot differentiate between binary and BCD numbers; it is the programmer who assumes the number to be binary or BCD and treats them accordingly.
- AC flag is useful for DA A instruction only. It has no other use to the programmer.
- Increment and decrement instructions do not affect the flags.
- The 8051 support byte as well as bit level logical operations such as AND, OR, EX-OR and NOT.
- The AND operation is often used to mask (set to 0) certain bits of a result, the OR operation is often used to set certain bits of a result to 1, while the EX-OR operation is often used to invert certain bits of an operand.
- No flags are affected by byte level logical instructions, except when destination operand is PSW (direct address).
- When destination of the logical instruction is port SFR, the latch register will be used as both source of data and destination to store the result.
- When an instruction uses port as a source, but not as a destination, microcontroller reads port pins (as the source of data) instead of port SFR.
- Rotate operations are useful for monitoring bits of data byte without using a logical test. The status of bits may be used in decision making process for certain applications.

OBJECTIVE QUESTIONS

1. Addition instruction of the 8051 can affect,
   (a) carry flag (b) aux. carry flag (c) overflow flag (d) all of the above
2. The contents of the accumulator after execution of following instructions will be,
   MOV A, #0FH
   ANL A, #2CH
   (a) 11010111 (b) 11011010 (c) 00001100 (d) 00101000
3. Which of the following statements will add the accumulator with register R0?
   (a) ADD @R0, A (b) ADD A, @R0 (c) ADD R0, A (d) ADD A, R0
4. To mask LSB of the A, we must AND it with,
   (a) 7FH (b) 80H (c) FEH (d) FFH
5. To complement the A, we must EX-OR it with,
   (a) 7FH (b) 80H (c) FEH (d) FFH
6. To set MSB of the A, we must OR it with,
   (a) 00H (b) 01H (c) 80H (d) FFH
7. The contents of the accumulator after execution of following instructions will be,
   MOV A, #55H
   ORL A, 01H
   (a) 1B H (b) 55 H (c) 3B H (d) 4B H
8. The following command will rotate the 8 bits of the accumulator one position to the left
   RLC A
   (a) True (b) False
9. The following program will read data of port 1, determine whether bit 2 is high, and if so, send the number FFH to port 2,
   ```
   BACK:  MOV A, P1
          ANL A, #02H
          CJNE A, #02H, BACK
          MOV P2, #0FFH
   ```
   (a) True (b) False
10. DA A instruction adjusts the value in the accumulator resulting from an addition of two BCD numbers only.
    (a) True (b) False
11. For signed operations, -1 is represented in binary as,
    (a) 10000001 (b) 01111111 (c) 10000000 (d) 11111111
12. The contents of A and B after execution of following instructions will be,
    ```
    MOV A, #02H
    MOV B, #04H
    MUL AB
    ```
    (a) A=02H, B= 04H (b) A=08H, B= 04H (c) A=08H, B= 00H (d) A=00H, B= 08H
13. _____is used to indicate error in signed arithmetic operations.
    (a) AC flag (b) OV flag (c) CY flag (d) P flag
14. _____is used to indicate error in unsigned arithmetic operations.
    (a) AC flag (b) OV flag (c) CY flag (d) P flag
15. An alternate instruction for CLR C is,
    (a) CLR PSW.0 (b) CLR PSW.7 (c) CLR PSW.2 (d) CLR PSW.1
16. AC flag is used by,
    (a) arithmetic instructions only (b) logical instructions only
    (c) DA A instruction only (d) all instructions
17. INC instructions affect,
    (a) CY flag (b) AC flag (c) OV flag (d) None of the above
18. A = BAH and CY=0. After execution of instruction SUBB A ,#64H, the status of CY and OV flags will be,
    (a) CY=0, OV=0 (b) CY=0, OV=1 (c) CY=1, OV=0 (d) CY=1, OV=1

19. A = 65H and CY=0. After execution of instruction SUBB A, #78H, the contents of A will be,
    (a) EDH (b) DEH (c) 19H (d) 78H
20. Which of the following instructions require maximum execution time?
    (a) ADD A,@R0 (b) DA A (c) MUL AB (d) DIV AB
21. A = F0H, R0=30H and (30) = AAH. The contents of A after execution of an instruction XCH A, @ R0, will be,
    (a) AAH (b) F0H (c) 30H (d) None of above
22. A = F0H, R1=40H and (40) = 0FH. The contents of A after execution of instruction XCHD A, @ R1, will be,
    (a) 00H (b) F0H (c) 0FH (d) FFH
23. A =FEH and CY=1. The contents of A after execution of instruction RRA will be,
    (a) FCH (b) 7FH (c) F1H (d) FDH
24. If A =D6H and CY=0, the contents of A after execution of instruction RLC A will be,
    (a) ACH (b) CAH (c) ADH (d) DAH
25. If DPTR =20FFH, the contents of DPH after execution of instruction INC DPTR will be,
    (a) 20H (b) 21H (c) 00H (d) FFH

Answers to Objective Questions

1. (d)	2. (c)	3. (d)	4. (c)	5. (d)	6. (c)	7. (b)	8. (b)	9. (a)
10. (a)	11. (d)	12. (c)	13. (b)	14. (c)	15. (b)	16. (c)	17. (d)	18. (b)
19. (a)	20. (c), (d)	21. (a)	22. (d)	23. (b)	24. (a)	25. (b)		

REVIEW QUESTIONS WITH ANSWERS

1. Register A is always destination operand in addition and subtraction instructions. True/False.

A. True.

2. List arithmetic flags of the 8051.

A. Carry (CY), Auxiliary carry (AC), and Overflow (OV).

3. Arithmetic flags are affected only by arithmetic instructions. True/False.

A. False. Arithmetic flags are located in PSW register. Any instruction which can modify PSW will change the flags.

4. Write instruction/s to add numbers 10H and 20H and store the result in to internal RAM address 30H.

A.
```
MOV A,#10H
ADD A, #20H
MOV 30H, A
```

5. What will be the status of CY and AC flags after execution of following instructions?
MOV A, #58H
ADD A, #28H

A. CY=0, AC=1 (carry out from bit D3 to D4).

6. PSW may also be referred as a flag register. True/False.

A. True.

7. What is meant by user flag?

A. A programmer can alter (set or reset) the flag as per the requirements or it can be used to record one-bit event.

8. State the validity of the following instructions.
(a) ADD A, R0 (b) ADD R0, A (c) ADD R1, #05H (d) DEC DPTR

A. (a) Valid.
(b) Invalid, Register A is always destination operand in addition instruction.
(c) Invalid, Register A is always destination operand in addition instruction.
(d) Invalid, there no such instruction in 8051.

9. Discuss the role of overflow flag in a division operation.

A. When an attempt is made to divide some number by zero, the overflow flag will set to 1 to indicate that the result is incorrect (indeterminate).

10. Where should the operands of multiply instruction be stored? Where does it store the result?

A. One of the operand (either multiplicand or multiplier) should be in register A and other in B. The result is stored in B (MSByte) and A (LSByte).

11. State one common application of AND operation.

A. It is commonly used in masking (clear bits to 0).

12. The 8051 has signed multiplication instruction. True/False.

A. False, it has unsigned multiplication instruction.

13. EX-OR operation of number with itself always result in a zero. True/False.

A. True.

14. What is a limitation of the rotate instructions?

A. It works only with register A.

15. Represent decimal number 95 in packed and unpacked BCD format.

A. Packed BCD 1001 0101
Unpacked BCD: 0000 1001, 0000 0101

16. What are the limitations of DA A instruction?

A. It works with only register A and must only be used after ADD or ADDC instruction.

17. Show how the erroneous result obtained in signed arithmetic can be recovered using third method (to interpret the result) and displayed on the display device (like LCD or monitor).

A. Overflow flag indicates the erroneous result.
Write a program as follows:
- Get MSB of the result and complement it.
- If it is 0, send ASCII of '+' to the display device, treat the 8 bits of the original result as magnitude, convert it to BCD, then into ASCII.
- If result of step 1 is 1, send ASCII of '–' to the display device, find 2's complement of the original result and treat it as magnitude.

EXERCISE

1. List the steps of actions taken by DA A instruction with suitable example.
2. How does BCD addition differ from binary addition?
3. Discuss the importance of AC flag.
4. Name different math flags in PSW.
5. If OV=1 after division, what is the cause?
6. Where should the operands of divide instruction be stored? Where does it store the result?
7. Discuss the difference between following two instructions,
   (1) DEC A (2) SUBB A, #01H
8. Discuss common applications of OR and EX-OR operations.
9. Suggest different instruction/s to clear Accumulator.
10. Explain with suitable example how rotate instruction can be use to check whether number is odd or even.
11. Explain with suitable example how rotate instruction can be use to check whether number is positive or negative.
12. What is meant by packed and unpacked BCD numbers?
13. Write a program to multiply two 8-bit numbers stored at internal RAM address 10H and 11H. Store the result at address 12H(MSByte) and 13H(LSByte).
14. Write the instruction/s to perform following operations,
    (a) Mask bit D7 of R2
    (b) Set upper three bits at address 30H
    (c) Exchange the nibbles of R2
15. What will be the contents of register A after execution of each of the following instructions.
    CLR C
    MOV A, #55H
    ORL A, #0F0
    RL A
    RLC A
    RLC A
16. Show how the swap operation is realized using rotate instructions.