Introduction to Number Systems

There are different characteristics that define a number system include ***the number of independent digits used in the number system, the place values of the different digits constituting the number and the maximum numbers that can be written with the given number of digits.*** Among the three characteristic parameters, the most fundamental is the number of independent digits or symbols used in the number system.. It is known as the *radix* or *base* of the number system. The decimal number system with which we are all so familiar can be said to have a radix of 10 as it has 10 independent digits, i.e. 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. Similarly, the binary number system with only two independent digits, 0 and 1, is a radix-2 number system. The octal and hexadecimal number systems have a radix (or base) of 8 and 16 respectively. We will see in the following sections that the radix of the number system also determines the other two characteristics. The place values of different digits in ***the integer part of the number are given by (r^0, r^1,r^2, r^3) and so on, starting with the digit adjacent to the radix point. For the fractional part, these are (r^{-1}, r^{-2},r^{-3}, r^{-4}) and so on,*** again starting with the digit next to the radix point. Here, r is the radix of the number system. Also, maximum numbers that can be written with n digits in a given number system are equal to r^n.

Basic Number system

1. Decimal Number System

The decimal number system is a radix-10 number system and therefore has 10 different digits or symbols. These are 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. All higher numbers after ‘9’ are represented in terms of these 10 digits only. The process of writing higher-order numbers after ‘9’ consists in writing the second digit (i.e. ‘1’) first, followed by the other digits, one by one, to obtain the next 10 numbers from ‘10’ to ‘19’. The next 10 numbers from ‘20’ to ‘29’ are obtained by writing the third digit (i.e. ‘2’) first, followed by digits ‘0’ to ‘9’, one by one. The process continues until

we have exhausted all possible two-digit combinations and reached '99'. Then we begin with three-digit combinations. The first three-digit number consists of the lowest two-digit number followed by '0' (i.e. 100), and the process goes on endlessly. The place values of different digits in a mixed decimal number, starting from the decimal point, ***are (10^0, 10^1,10^2, 10^3) and so on (for the integer part) and (10^{-1}, 10^{-2},10^{-3}, 10^{-4})and so on (for the fractional part).***

Example: the decimal number 3586.265 can be expressed as

$3586 = 6\times10^0 + 8\times 10^1 + 5\times 10^2 + 3\times 10^3 = 6+80+500+3000 = 3586$

$265 = 2\times10^{-1} + 6\times10^{-2} + 5\times10^{-3} = 0.2+0.06+0.005 = 0.265$

2. Binary Number System

The binary number system is a radix-2 number system with '0' and '1' as the two independent digits. All larger binary numbers are represented in terms of '0' and '1'. The procedure for writing higher order binary numbers after '1' is similar to the one explained in the case of the decimal number system. For example, the first 16 numbers in the binary number system would be 0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011, 1100, 1101, 1110 and 1111. The next number after 1111 is 10000, which is the lowest binary number with five digits. This also proves the point made earlier that a maximum of only 16 = (2^4) numbers could be written with four digits. Starting from the binary point, the place values of different digits in a mixed binary number ***are (2^0, 2^1,2^2, 2^3) and so on (for the integer part) (2^{-1}, 2^{-2},2^{-3}, 2^{-4}) and so on (for the fractional part).***

3. Octal Number System

The octal number system has a radix of 8 and therefore has eight distinct digits. All higher-order numbers are expressed as a combination of these on the same pattern

as the one followed in the case of the binary and decimal number systems. The independent digits are 0, 1, 2, 3, 4, 5, 6 and 7. The next 10 numbers that follow '7', for example, would be 10, 11, 12, 13, 14, 15, 16, 17, 20 and 21. In fact, if we omit all the numbers containing the digits 8 or 9, or both, from the decimal number system, we end up with an octal number system. The place values for the different digits in the octal number system ***are (8^0, 8^1,8^2, 8^3) and so on (for the integer part) (8^{-1}, 8^{-2},8^{-3}, 8^{-4}) and so on (for the fractional part).***

4. Hexadecimal Number System

The hexadecimal number system is a radix-16 number system and its 16 basic digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F. The place values or weights of different digits in a mixed hexadecimal number ***are (16^0, 16^1,16^2, 16^3) and so on (for the integer part) (16^{-1}, 16^{-2},16^{-3}, 16^{-4}) and so on (for the fractional part). The decimal equivalent of A, B, C, D, E and F are 10, 11, 12, 13, 14 and 15 respectively, for obvious reasons.*** The hexadecimal number system provides a condensed way of representing large binary numbers stored and processed inside the computer. One such example is in representing addresses of different memory locations. Let us assume that a machine has 64K of memory. Such a memory has 64K ($= 2^{16} = 65\ 536$) memory locations and needs 65 536 different addresses. These addresses can be designated as 0 to 65 535 in the decimal number system and 00000000 00000000 to 11111111 11111111 in the binary number system. The decimal number system is not used in computers and the binary notation here appears too cumbersome and inconvenient to handle. In the hexadecimal number system, 65 536 different addresses can be expressed with four digits from 0000 to FFFF. Similarly, the contents of the memory when represented in hexadecimal form are very convenient to handle.

Binary-to-Decimal Conversion

Example : determined The decimal equivalent of the binary number $(1101)_2$?

$(1101)_2 = 1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3 = (13)_{10}$

Example : determined The decimal equivalent of the binary number $(01010)_2$?

$(01010)_2 = 0 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 + 0 \times 2^4 = (10)_{10}$

Example : determined The decimal equivalent of the binary number $(1001.0101)_2$?

The decimal equivalent of the binary number $(1001.0101)_2$ is determined as :

- The integer part = 1001
- The decimal equivalent = $1 \times 2^0 + 0 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 = 1 + 0 + 0 + 8 = 9$
- The fractional part = .0101
- Therefore, the decimal equivalent = $0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4}$

$= 0 + 0.25 + 0 + 0.0625 = 0.3125$

- Therefore, the decimal equivalent of $(1001.0101)_2 = 9.3125$

Decimal-to-Binary Conversion

As outlined earlier, the integer and fractional parts are worked on separately. For the integer part, the binary equivalent can be found by successively dividing the integer part of the number by 2 and recording the remainders until the quotient becomes '0'. The remainders written in reverse order constitute the binary equivalent. For the fractional part, it is found by successively multiplying the fractional part of the decimal number by 2 and recording the carry until the result of multiplication is '0'. The carry sequence written in forward order constitutes the binary equivalent of the fractional part of the decimal number. If the result of multiplication does not seem to be heading towards zero in the case of the fractional part, the process may be continued only until the requisite number of equivalent bits has been obtained. This

method of decimal–binary conversion is popularly known as the double-dabble method. The process can be best illustrated with the help of an example.

Example : *find the binary equivalent of* $(13.375)_{10}$

• The integer part = 13

		Remainder
13 ÷ 2	= 6	1
6 ÷ 2	= 3	0
3 ÷ 2	= 1	1
1 ÷ 2	= 0	1

The binary equivalent of = $(13)_{10}$= $(1101)_2$

• The fractional part = .375

• 0.375 × 2 = 0.75 with a carry of 0

• 0.75 × 2 = 0.5 with a carry of 1

• 0.5 × 2 = 0 with a carry of 1

• The binary equivalent of (0.375)10= (.011)2

• Therefore, the binary equivalent of (13.375)10 = (1101.011)2

Homework :

- *find the binary equivalent of* $(10)_{10}$?
- *find the binary equivalent of* $(25)_{10}$?
- *find the binary equivalent of* $(0.101)_{10}$?
- *find the binary equivalent of* $(110.011)_{10}$?
- *find the binary equivalent of* $(10.01)_{10}$?
- *find the binary equivalent of* $(0.125)_{10}$?
- *find the binary equivalent of* $(0.17)_{10}$?

Octal-to-Decimal Conversion

The decimal equivalent of the octal number $(137.21)_8$ is determined as follows:

• The integer part = 137

• The decimal equivalent = $7 \times 8^0 + 3 \times 8^1 + 1 \times 8^3 = 7 + 24 + 64 = 95$

• The fractional part = .21

• The decimal equivalent = $2 \times 8^{-1} + 1 \times 8^{-2} = 0.265$

• Therefore, the decimal equivalent of $(137.21)_8 = (95.265)_{10}$

Decimal-to-Octal Conversion

The process of decimal-to-octal conversion is similar to that of decimal-to-binary conversion. The progressive division in the case of the integer part and the progressive multiplication while working on the fractional part here are by '8' which is the radix of the octal number system. Again, the integer and fractional parts of the decimal number are treated separately. The process can be best illustrated with the help of an example.

Example

We will find the octal equivalent of $(73.75)_{10}$?

• The integer part = 73

		Remainder
73 ÷ 8	= 9	1
9 ÷ 8	= 1	1
1 ÷ 8	= 0	1

• The octal equivalent of $(73)_{10} = (111)_8$

• The fractional part = 0.75

• 0.75 × 8 = 0 with a carry of 6

• The octal equivalent of $(73)_{10} = (111)_8$

• The octal equivalent of $(0.75)_{10} = (.6)_8$

• Therefore, the octal equivalent of $(73.75)_{10} = (111.6)_8$

Homework :

- *find the octal equivalent of* $(105)_{10}$?
- *find the decimal equivalent of* $(127)_8$?
- *find the decimal equivalent of* $(75.23)_8$?
- *find the decimal equivalent of* $(10.01)_8$?

Hexadecimal-to-Decimal Conversion

The decimal equivalent of the hexadecimal number $(1E0.2A)_{16}$ is determined as follows:

• The integer part = 1E0

• The decimal equivalent = $0 \times 16^0 + 14 \times 16^1 + 1 \times 16^2 = 0 + 224 + 256 = 480$

• The fractional part = 2A

• The decimal equivalent = $2 \times 16^{-1} + 10 \times 16^{-2} = 0.164$

• Therefore, the decimal equivalent of $(1E0.2A)_{16} = (480.164)_{10}$

Decimal-to-Hexadecimal Conversion

The process of decimal-to-hexadecimal conversion is also similar. Since the hexadecimal number system has a base of 16, the progressive division and multiplication factor in this case is 16. The process is illustrated further with the help of an example.

Example

Let us determine the hexadecimal equivalent of $(82.25)_{10}$?

• The integer part = 82

	Remainder
82 ÷ 16 = 5	2
5 ÷ 16 = 0	5

• The hexadecimal equivalent of $(82)_{10} = (52)_{16}$

• The fractional part = 0.25

• $0.25 \times 16 = 0$ with a carry of 4

• Therefore, the hexadecimal equivalent of $(82.25)_{10} = (52.4)_{16}$

Homework :

- *find the* hexadecimal *equivalent of* $(100)_{10}$?
- *find the decimal equivalent of* $(\mathrm{A1C})_{16}$?
- *find the decimal equivalent of* $(\mathrm{AF.3C})_{16}$?

Binary–Octal and Octal–Binary Conversions

An octal number can be converted into its binary equivalent by replacing each octal digit with its three-bit binary equivalent. We take the three-bit equivalent because the base of the octal number system is 8 and it is the third power of the base of the binary number system, i.e. 2. All we have then to remember is the three-bit binary equivalents of the basic digits of the octal number system. A binary number can be converted into an equivalent octal number by splitting the integer and fractional parts into groups of three bits, starting from the binary point on both sides. The 0s can be added to complete the outside groups if needed.

Example

Let us find the binary equivalent of $(374.26)_8$ *and the octal equivalent of* $(1110100.0100111)_2$?

Solution

• The given octal number = $(374.26)_8$

• The binary equivalent = $(011\ 111\ 100.010\ 110)_2 = (011111100.010110)_2$

• Any 0s on the extreme left of the integer part and extreme right of the fractional part of the equivalent binary number should be omitted. Therefore, $(011111100.010110)_2 = (11111100.01011)_2$

• The given binary number = $(1110100.0100111)_2 = (1110100.0100111)_2$

$= (1\ 110\ 100.010\ 011\ 1)_2 = (001\ 110\ 100.010\ 011\ 100)_2 = (164.234)_8$

Homework :

- *find the Binary equivalent of* $(23)_8$?

- *find the Binary equivalent of* $(756)_8$?
- *find the Binary equivalent of* $(75.23)_8$?
- *find the octal equivalent of* $(0001.1010)_2$?

Hex–Binary and Binary–Hex Conversions

A hexadecimal number can be converted into its binary equivalent by replacing each hex digit with its four-bit binary equivalent. We take the four-bit equivalent because the base of the hexadecimal number system is 16 and it is the fourth power of the base of the binary number system. All we have then to remember is the four-bit binary equivalents of the basic digits of the hexadecimal number system. A given binary number can be converted into an equivalent hexadecimal number by splitting the integer and fractional parts into groups of four bits, starting from the binary point on both sides. The 0s can be added to complete the outside groups if needed.

Example

Let us find the binary equivalent of $(17E.F6)_{16}$ *and the hex equivalent of* $(1011001110.011011101)_2$ *?*

• The given hex number = $(17E.F6)_{16}$

• The binary equivalent = $(0001\ 0111\ 1110.1111\ 0110)_2$

$=(000101111110.11110110)_2 = (101111110.1111011)_2$

• The 0s on the extreme left of the integer part and on the extreme right of the fractional part have been omitted.

• The given binary number = $(1011001110.011011101)_2$

$= (10\ 1100\ 1110.0110\ 1110\ 1)_2$

• The hex equivalent = $(0010\ 1100\ 1110.0110\ 1110\ 1000)_2 = (2CE.6E8)_{16}$

Hex–Octal and Octal–Hex Conversions

For hexadecimal–octal conversion, the given hex number is firstly converted into its binary equivalent which is further converted into its octal equivalent. An

alternative approach is firstly to convert the given hexadecimal number into its decimal equivalent and then convert the decimal number into an equivalent octal number. The former method is definitely more convenient and straightforward. For octal–hexadecimal conversion, the octal number may first be converted into an equivalent binary number and then the binary number transformed into its hex equivalent. The other option is firstly to convert the given octal number into its decimal equivalent and then convert the decimal number into its hex equivalent. The former approach is definitely the preferred one. Two types of conversion are illustrated in the following example.

Example

Let us find the octal equivalent of $(2F.C4)_{16}$ *and the hex equivalent of* $(762.013)_8$?

Solution

• The given hex number = $(2F.C4)_{16}$.

• The binary equivalent = $(0010\ 1111.1100\ 0100)_2$

= $(00101111.11000100)_2$ = $(101111.110001)_2$ = $(101\ 111.110\ 001)2$ = $(57.61)_8$.

• The given octal number = $(762.013)_8$.

• The octal number = $(762.013)_8$ = $(111\ 110\ 010.000\ 001\ 011)_2$

= $(111110010.000001011)_2$ = $(0001\ 1111\ 0010.0000\ 0101\ 1000)_2$ = $(1F2.058)_{16}$.

Digital Arithmetic

Basic Rules of Binary Addition and Subtraction

The basic principles of binary addition and subtraction are similar to what we all know so well in the case of the decimal number system. In the case of addition, adding ‘0’ to a certain digit produces the same digit as the sum, and, when we add ‘1’ to a certain digit or number in the decimal number system, the result is the next higher digit or number, as the case may be. For example, 6 + 1 in decimal equals ‘7’ because ‘7’ immediately follows ‘6’ in the decimal number system. Also, 7 + 1 in octal equals ‘10’ as, in the octal number system, the next adjacent higher number

after '7' is '10'. Similarly, 9 + 1 in the hexadecimal number system is 'A'. With this background, we can write the basic rules of binary addition as follows:

1) 0 + 0 = 0.
2) 0 + 1 = 1.
3) 1 + 0 = 1.
4) 1 + 1 = 0 with a carry of '1' to the next more significant bit.
5) 1 + 1 + 1 = 1 with a carry of '1' to the next more significant bit.

Table (1) summarizes the sum and carry outputs of all possible three-bit combinations. We have taken three-bit combinations as, in all practical situations involving the addition of two larger bit numbers, we need to add three bits at a time. Two of the three bits are the bits that are part of the two binary numbers to be added, and the third bit is the carry-in from the next less significant bit column.

Table(1) *Binary addition of three bits.*

A	B	Carry- in (Cin)	Sum	Carry-out (Cout)
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

The basic principles of binary subtraction include the following:

1) 0 − 0 = 0.
2) 1 − 0 = 1.
3) 1 − 1 = 0.
4) 0 − 1 = 1 with a borrow of 1 from the next more significant bit.

The above-mentioned rules can also be explained by recalling rules for subtracting decimal numbers. Subtracting '0' from any digit or number leaves the digit or number unchanged. This explains the first two rules. Subtracting '1' from any digit or number in decimal produces the immediately preceding digit or number as the answer. In general, the subtraction operation of larger-bit binary numbers also involves three bits, including the two bits involved in the subtraction, called the minuend (the upper bit) and the subtrahend (the lower bit), and the borrow-in. The subtraction operation produces the difference output and borrow-out, if any. Table (2) summarizes the binary subtraction operation. The entries in Table (2) can be explained by recalling the basic rules of binary subtraction mentioned above, and that the subtraction operation involving three bits, that is, the minuend (A), the subtrahend (B) and the borrow-in (Bin), produces a difference output equal to (A − B – Bin). It may be mentioned here that, in the case of subtraction of larger-bit binary numbers, the least significant bit column always involves two bits to produce a difference output bit and the borrow-out bit. The borrow-out bit produced here becomes the borrow-in bit for the next more significant bit column, and the process continues until we reach the most significant bit column.

Table 3.2 Binary subtraction.

A (Minuend)	B(Subtrahend)	(Bin) Borrow-in	D(Different)	Bout (Borrow-out)
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Number Systems – Some Common Terms

In this section we will describe some commonly used terms with reference to different number systems.

Binary Number System

Bit is an abbreviation of the term ***'binary digit'*** and is the smallest unit of information. It is either '0' or '1'. A *byte* is a string of eight bits. The byte is the basic unit of data operated upon as a single unit in computers. A *computer word* is again a string of bits whose size, called the 'word length' or 'word size', is fixed for a specified computer, although it may vary from computer to computer. The word length may equal one byte, two bytes, four bytes or be even larger. The *1's complement* of a binary number is obtained by complementing all its bits, i.e. by replacing 0s with 1s and 1s with 0s. For example, the *1's complement of* $(10010110)_2$ *is* $(01101001)_2$. *The 2's complement of a binary number is obtained by adding '1' to its 1's complement. The 2's complement of* $(10010110)_2$ *is* $(01101010)_2$.

Example :

1101 *1101*

-1011 *1's complement* → *+ 0100*

1 0001

+ 1 *The final carry has been added*

0010

Example :

1011 *1011*

-1101 *1's complement* → *+ 0010*

0 1101 *1's complement* → - 0010

Since there are no carry take 1's complement and put minus (-0010)

Homework :

- *1111 – 1011 = find the result using 1's complement ?*
- *1011 – 1010 = find the result using 1's complement ?*
- *110 – 11 = find the result using 1's complement ?*

Example :

1101
-1011 *2's complement* → *0100 +1* = *0101*

1 0010

The final carry has been disregarded the result is 0010

Example :

101
-111 *2's complement* → *000 +1* = *001*

0 110 *2's complement* → - 010

Since there are no carry take 2's complement and put minus (-010)

Homework :

- *101 – 100 = find the result using 2's complement ?*
- *1110 – 11 = find the result using 1's complement ?*

Decimal Number System

Corresponding to the 1's and 2's complements in the binary system, in the decimal number system we have the 9's and 10's complements. *The 9's complement of a given decimal number is obtained by subtracting each digit from 9. For example, the 9's complement of* $(2496)_{10}$ *would be* $(7503)_{10}$. *The 10's complement is obtained by adding '1' to the 9's complement. The 10's complement of* $(2496)_{10}$ *is* $(7504)_{10}$.

Example :

75 *75*

- 28 *9's complement* → *+ 71*

1 46

1 *The final carry has been added*

47

Example :

75 *75*

- 28 *10's complement* → *+ 72*

1 47

The final carry has been disregarded the result is 47

Homework :

- *28 – 75 = find the result using 10's complement ?*
- *28 – 75 = find the result using 9's complement ?*

Octal Number System

In the octal number system, we have the *7's and 8's complements. The 7's complement of a given octal number is obtained by subtracting each octal digit from 7. For example, the 7's complement of* $(562)_8$ *would be* $(215)_8$. *The 8's complement is obtained by adding '1' to the 7's complement. The 8's complement of* $(562)_8$ *would be* $(216)_8$.

Hexadecimal Number System

The 15's and 16's complements are defined with respect to the hexadecimal number system. *The 15's complement is obtained by subtracting each hex digit from 15. For example, the 15's complement of* $(3BF)_{16}$ *would be* $(C40)_{16}$. *The 16's complement is obtained by adding '1' to the 15's complement. The 16's complement of* $(2AE)_{16}$ *would be* $(D52)_{16}$.

Sign-Bit Magnitude

In the sign-bit magnitude ***representation of positive and negative decimal numbers, the MSB represents the 'sign', with a '0' denoting a plus sign and a '1' denoting a minus sign.*** The remaining bits represent the magnitude. In eight-bit representation, while MSB represents the sign, the remaining seven bits represent the magnitude. For example, the eight-bit representation of +9 would be 00001001, and that for −9 would be 10001001. An n−bit binary representation can be used to represent decimal numbers in the range of − ($2^{n-1}-1$) to + ($2^{n-1}-1$). That is, eight-bit representation can be used to represent decimal numbers in the range from −127 to +127 using the sign-bit magnitude format.

1's Complement

In the 1's complement format, the positive numbers remain unchanged. The negative numbers are obtained by taking the 1's complement of the positive counterparts. For example, +9 will be represented as 00001001 in eight-bit notation, and −9 will be represented as 11110110, which is the 1's complement of 00001001. Again, n-bit notation can be used to represent numbers in the range from − ($2^{n-1}-1$) to + ($2^{n-1}-1$). using the 1's complement format. The eight-bit representation of the 1's complement format can be used to represent decimal numbers in the range from −127 to +127.

2's Complement

In the 2's complement representation of binary numbers, the MSB represents the sign, with a '0' used for a plus sign and a '1' used for a minus sign. The remaining bits are used for representing magnitude. Positive magnitudes are represented in the same way as in the case of sign-bit or 1's complement representation. Negative magnitudes are represented by *the 2's complement of their positive counterparts. For example, +9 would be represented as 00001001, and −9*

would be written as 11110111. Please note that, if the 2's complement of the magnitude of +9 gives a magnitude of −9, then the reverse process will also be true, i.e. the 2's complement of the magnitude of −9 will give a magnitude of +9. The n-bit notation of the 2's complement format can be used to represent all decimal numbers in the range from $-(2^{n-1}-1)$ to $+(2^{n-1})$. The 2's complement format is very popular as it is very easy to generate the 2's complement of a binary number and also because arithmetic operations are relatively easier to perform when the numbers are represented in the 2's complement format.

Example

Find the decimal equivalent of the following binary numbers expressed in the 2's complement format:

(a) 00001110;

(b) 10001110.

Solution

(a) The MSB bit is '0', which indicates a plus sign. The magnitude bits are 0001110. The decimal equivalent = $0\times2^0 + 1\times2^1 + 1\times2^2 + 1\times2^3 + 0\times2^4 + 0\times2^5 + 0\times2^6$ = 0+2+4+8+0+0+0 = 14 Therefore, 00001110 represents +14

(b) The MSB bit is '1', which indicates a minus sign. The magnitude bits are therefore given by the 2's complement of 0001110, i.e. 1110010 The decimal equivalent = $0\times2^0 + 1\times2^1 + 0\times2^2 + 0\times2^3 + 1\times2^4 + 1\times2^5 + 1\times2^6$ = 0+2+0+0+16+32+64 = 114 Therefore, 10001110 represents −114.

Addition Using the 2's Complement Method

The 2's complement is the most commonly used code for processing positive and negative binary numbers. It forms the basis of arithmetic circuits in modern computers. When the decimal numbers to be added are expressed in 2's complement form, the addition of these numbers, following the basic laws of binary addition, gives correct results. Final carry obtained, if any, while adding MSBs should be disregarded. To illustrate this, we will consider the following four different cases:

1. Both the numbers are positive.
2. Larger of the two numbers is positive.
3. The larger of the two numbers is negative.
4. Both the numbers are negative.

Case 1

• Consider the decimal numbers +37 and +18.

• The 2's complement of +37 in eight-bit representation = 00100101.

• The 2's complement of +18 in eight-bit representation = 00010010.

• The addition of the two numbers, that is, +37 and +18, is performed as follows

```
  00100101
+ 00010010
----------
= 00110111
```

• The decimal equivalent of (00110111)2 is (+55), which is the correct answer.

Case 2

• Consider the two decimal numbers +37 and -18.

• The 2's complement representation of +37 in eight-bit representation = 00100101.

• The 2's complement representation of −18 in eight-bit representation = 11101110.

• The addition of the two numbers, that is, +37 and −18, is performed as follows:

```
  00100101
+ 11101110
----------
= 00010011    The final carry has been disregarded.
```

• The decimal equivalent of (00010011)2 is +19, which is the correct answer.

Case 3

• Consider the two decimal numbers +18 and −37.

• −37 in 2's complement form in eight−bit representation = 11011011.

• +18 in 2's complement form in eight−bit representation = 00010010.

• The addition of the two numbers, that is, −37 and +18, is performed as follows:

```
  11011011
+ 00010010
----------
  11101101
```

• The decimal equivalent of (11101101)2, which is in 2's complement form, is −19, which is the correct answer.

Case 4

• Consider the two decimal numbers −18 and −37.

• −18 in 2's complement form is 11101110.

• −37 in 2's complement form is 11011011.

• The addition of the two numbers, that is, −37 and −18, is performed as follows:

```
  11011011
+ 11101110
----------
  11001001
```

• The final carry in the ninth bit position is disregarded.

• The decimal equivalent of (11001001)2, which is in 2's complement form, is −55, which is thecorrect answer.

It may also be mentioned here that, in general, 2's complement notation can be used to perform addition when the expected result of addition lies in the range from −2n−1 to +(2n−1− 1), n being the number of bits used to represent the numbers. As an example, eight-bit 2's complement arithmetic cannot be used to perform addition if the result of addition lies outside the range from −128 to +127.

Different steps to be followed to do addition in 2's complement arithmetic are summarized as follows:

1. Represent the two numbers to be added in 2's complement form.
2. Do the addition using basic rules of binary addition.
3. Disregard the final carry, if any.
4. The result of addition is in 2's complement form.

Example 3.1

Perform the following addition operations:

1. $(275.75)_{10}$+ $(37.875)_{10}$.

2. $(AF1.B3)_{16}$+ $(FFF.E)_{16}$.

Solution

1. As a first step, the two given decimal numbers will be converted into their equivalent binary numbers :

$(275.75)_{10}$ = $(100010011.11)_2$ and $(37.875)_{10}$ = $(100101.111)_2$ The two binary numbers can be rewritten as (100010011.110)2 and (000100101.111)2 to have the same number of bits in their integer and fractional parts. The addition of two numbers is performed as follows:

100010011_110

000100101_111

100111001_101

The decimal equivalent of (100111001.101)2 is (313.625)10.

2. (AF1.B3)16

=(101011110001.10110011)2 and (FFF.E)16

=(111111111111.1110)2. (1111111111

11.1110)2 can also be written as (111111111111.11100000)2 to have the same number of bits in

the integer and fractional parts. The two numbers can now be added as follows:

0101011110001_10110011

0111111111111_11100000

1101011110001_10010011

The hexadecimal equivalent of (1101011110001.10010011)2 is (1AF1.93)16, which is equal to the

hex addition of (AF1.B3)16 and (FFF.E)16.

Example 3.2

Find out whether 16-bit 2's complement arithmetic can be used to add 14 276 and 18 490.

Solution

The addition of decimal numbers 14 276 and 18 490 would yield 32 766. 16-bit 2's complement

arithmetic has a range of −215 to +(215− 1), i.e. −32 768 to +32 767. The expected result is inside

the allowable range. Therefore, 16-bit arithmetic can be used to add the given numbers.

Example 3.3

Add −118 and −32 firstly using eight-bit 2's complement arithmetic and then using 16-bit 2's

complement arithmetic. Comment on the results.

Solution

• −118 in eight-bit 2's complement representation = 10001010.

• −32 in eight-bit 2's complement representation = 11100000.

• The addition of the two numbers, after disregarding the final carry in the ninth bit position, is

01101010. Now, the decimal equivalent of (01101010)2, which is in 2's complement form, is +106.

The reason for the wrong result is that the expected result, i.e. −150, lies outside the range of

eight-bit 2's complement arithmetic. Eight-bit 2's complement arithmetic can be used when the
expected result lies in the range from −27 to + (27 − 1), i.e. −128 to +127. −118 in 16-bit 2's
complement representation = 1111111110001010.

• −32 in 16-bit 2's complement representation = 1111111111100000.

• The addition of the two numbers, after disregarding the final carry in the 17th position, produces
1111111101101010. The decimal equivalent of (1111111101101010)2, which is in 2's complement
form, is −150, which is the correct answer. 16-bit 2's complement arithmetic has produced the
correct result, as the expected result lies within the range of 16-bit 2's complement notation.

3.3 Subtraction of Larger-Bit Binary Numbers

Subtraction is also done columnwise in the same way as in the case of the decimal number system.
In the first step, we subtract the LSBs and subsequently proceed towards the MSB. Wherever the
subtrahend (the bit to be subtracted) is larger than the minuend, we borrow from the next adjacent
higher bit position having a '1'. As an example, let us go through different steps of subtracting (1001)2
from (1100)2.

In this case, '1' is borrowed from the second MSB position, leaving a '0' in that position. The
borrow is first brought to the third MSB position to make it '10'. Out of '10' in this position,

‘1’ is taken to the LSB position to make ‘10’ there, leaving a ‘1’ in the third MSB position.

10−1 in the LSB column gives ‘1’, 1−0 in the third MSB column gives ‘1’, 0−0 in the second

MSB column gives ‘0’ and 1−1 in the MSB also gives ‘0’ to complete subtraction. Subtraction

of mixed numbers is also done in the same manner. The above-mentioned steps are summarized

as follows:

1. 1 1 0 0 2. 1 1 0 0

1 0 0 1 1 0 0 1

1 1 1

3. 1 1 0 0 4. 1 1 0 0

1 0 0 1 1 0 0 1

0 1 1 0 0 1 1

3.3.1 Subtraction Using 2’s Complement Arithmetic

Subtraction is similar to addition. Adding 2’s complement of the subtrahend to the minuend and

disregarding the carry, if any, achieves subtraction. The process is illustrated by considering six

different cases:

1. Both minuend and subtrahend are positive. The subtrahend is the smaller of the two.
2. Both minuend and subtrahend are positive. The subtrahend is the larger of the two.
3. The minuend is positive. The subtrahend is negative and smaller in magnitude.
4. The minuend is positive. The subtrahend is negative and greater in magnitude.
5. Both minuend and subtrahend are negative. The minuend is the smaller of the two.
6. Both minuend and subtrahend are negative. The minuend is the larger of the two.

Case 1

• Let us subtract +14 from +24.

• The 2's complement representation of +24 = 00011000.

• The 2's complement representation of +14 = 00001110.

• Now, the 2's complement of the subtrahend (i.e. +14) is 11110010.

• Therefore, +24 − (+14) is given by

00011000

+ 11110010

00001010

with the final carry disregarded.

• The decimal equivalent of (00001010)2 is +10, which is the correct answer.

Case 2

• Let us subtract +24 from +14.

• The 2's complement representation of +14 = 00001110.

• The 2's complement representation of +24 = 00011000.

• The 2's complement of the subtrahend (i.e. +24) = 11101000.

• Therefore, +14 − (+24) is given by

00001110

+ 11101000

11110110

• The decimal equivalent of (11110110)2, which is of course in 2's complement form, is −10 which

is the correct answer.

Case 3

• Let us subtract −14 from +24.

• The 2's complement representation of +24 = 00011000 = minuend.

• The 2's complement representation of −14 = 11110010 = subtrahend.

• The 2's complement of the subtrahend (i.e. −14) = 00001110.

• Therefore, +24 − (−14) is performed as follows:

00011000

+ 00001110

00100110

• The decimal equivalent of (00100110)2 is +38, which is the correct answer.

Case 4

• Let us subtract −24 from +14.

• The 2's complement representation of +14 = 00001110 = minuend.

• The 2's complement representation of −24 = 11101000 = subtrahend.

• The 2's complement of the subtrahend (i.e. −24) = 00011000.

• Therefore, +14 − (−24) is performed as follows:

00001110

+ 00011000

00100110

• The decimal equivalent of (00100110)2 is +38, which is the correct answer.

Case 5

• Let us subtract −14 from −24.

• The 2's complement representation of −24 = 11101000 = minuend.

• The 2's complement representation of −14=11110010 = subtrahend.

• The 2's complement of the subtrahend = 00001110.

• Therefore, −24 − (−14) is given as follows:

11101000

+ 00001110

11110110

• The decimal equivalent of (11110110)2, which is in 2's complement form, is −10, which is the

correct answer.

Case 6

• Let us subtract −24 from −14.

• The 2's complement representation of −14 = 11110010 = minuend.

• The 2's complement representation of −24=11101000 = subtrahend.

• The 2's complement of the subtrahend = 00011000.

• Therefore, −14 − (−24) is given as follows:

11110010

+ 00011000

00001010

with the final carry disregarded.

• The decimal equivalent of (00001010)2, which is in 2's complement form, is +10, which is the

correct answer.

It may be mentioned that, in 2's complement arithmetic, the answer is also in 2's complement

notation, only with the MSB indicating the sign and the remaining bits indicating the magnitude. In

2's complement notation, positive magnitudes are represented in the same way as the straight binary

numbers, while the negative magnitudes are represented as the 2's complement of their straight binary

counterparts. A '0' in the MSB position indicates a positive sign, while a '1' in the MSB position

indicates a negative sign.

The different steps to be followed to do subtraction in 2's complement arithmetic are summarized

as follows:

1. Represent the minuend and subtrahend in 2's complement form.

2. Find the 2's complement of the subtrahend.

3. Add the 2's complement of the subtrahend to the minuend.

4. Disregard the final carry, if any.

5. The result is in 2's complement form.

6. 2's complement notation can be used to perform subtraction when the expected result of subtraction

lies in the range from −2n−1 to +(2n−1− 1), n being the number of bits used to represent the

numbers.

Example 3.4

Subtract (1110.011)2 from (11011.11)2 using basic rules of binary subtraction and verify the result by

showing equivalent decimal subtraction.

Solution

The minuend and subtrahend are first modified to have the same number of bits in the integer and

fractional parts. The modified minuend and subtrahend are (11011.110)2 and (01110.011)2 respectively:

11011_110

− 01110_011

01101_011

The decimal equivalents of (11011.110)2 and (01110.011)2 are 27.75 and 14.375 respectively. Their

difference is 13.375, which is the decimal equivalent of (01101.011)2.

Example 3.5

Subtract (a) (−64)10 from (+32)10 and (b) (29.A)16 from (4F.B)16. Use 2's complement arithmetic.

Solution:

(a) (+32)10in 2's complement notation = (00100000)2.

(−64)10 in 2's complement notation = (11000000)2.

The 2's complement of (−64)10

= (01000000)2.

(+32)10

− (−64)10 is determined by adding the 2's complement of (−64)10 to (+32)10.

Therefore, the addition of (00100000)2 to (01000000)2 should give the result. The operation is

shown as follows:

00100000

+ 01000000

01100000

The decimal equivalent of (01100000)2 is +96, which is the correct answer as +32 − (−64)=+96.

(b) The minuend = (4F.B)16

= (01001111.1011)2.

The minuend in 2's complement notation = (01001111.1011)2.

The subtrahend = (29.A)16

= (00101001.1010)2.

The subtrahend in 2's complement notation = (00101001.1010)2.

The 2's complement of the subtrahend = (11010110.0110)2.

(4F.B)16

−(29.A)16 is given by the addition of the 2's complement of the subtrahend to the minuend.

01001111_1011

+ 11010110_0110

00100110_0001

with the final carry disregarded. The result is also in 2's complement form. Since the result is a

positive number, 2's complement notation is the same as it would be in the case of the straight

binary code.

The hex equivalent of the resulting binary number = (26.1)16, which is the correct answer.

Binary Coded Decimal

The binary coded decimal (BCD) is a type of binary code used to represent a given decimal number in an equivalent binary form. BCD-to-decimal and decimal-to-BCD conversions are very easy and straightforward. It is also far less cumbersome an exercise to represent a given decimal number in an equivalent BCD code than to represent it in the equivalent straight binary form discussed in the previous chapter.

The BCD equivalent of a decimal number is written by replacing each decimal digit in the integer and fractional parts with its four-bit binary equivalent. As an example, the BCD equivalent of (23.15)10 is written as (0010 0011.0001 0101)BCD. The BCD code described above is more precisely known as the 8421 BCD code, with 8, 4, 2 and 1 representing the weights of different bits in the four-bit groups, starting from MSB and proceeding towards LSB. This feature makes it a weighted code, which means that each bit in the four-bit group representing a given decimal digit has an assigned

Decimal	**8421 BCD code**	**4221 BCD code**	**5421 BCD code**
0	0000	0000	0000
1	0001	0001	0001
2	0010	0010	0010
3	0011	0011	0011
4	0100	1000	0100
5	0101	0111	1000
6	0110	1100	1001
7	0111	1101	1010
8	1000	1110	1011
9	1001	1111	1100

weight. Other weighted BCD codes include the 4221 BCD and 5421 BCD codes. Again, 4, 2, 2 and 1 in the 4221 BCD code and 5, 4, 2 and 1 in the 5421 BCD code

represent weights of the relevant bits. Table 2.1 shows a comparison of 8421, 4221 and 5421 BCD codes. As an example, (98.16)10 will be written as 1111 1110.0001 1100 in 4221 BCD code and 1100 1011.0001 1001 in 5421 BCD code. Since the 8421 code is the most popular of all the BCD codes, it is simply referred to as the BCD code.

BCD-to-Binary Conversion

A given BCD number can be converted into an equivalent binary number by first writing its decimal equivalent and then converting it into its binary equivalent. we will find the binary equivalent of the BCD number 0010 1001.0111 0101:

• BCD number: 0010 1001.0111 0101.

• Corresponding decimal number: 29.75.

• The binary equivalent of 29.75 can be determined to be 11101 for the integer part and .11 for the fractional part.

• Therefore, (0010 1001.0111 0101)BCD =(11101.11)2.

Binary-to-BCD Conversion

The process of binary-to-BCD conversion is the same as the process of BCD-to-binary conversion executed in reverse order. A given binary number can be converted into an equivalent BCD number by first determining its decimal equivalent and then writing the corresponding BCD equivalent

Excess-3 Code

The excess-3 code is another important BCD code. It is particularly significant for arithmetic operations as it overcomes the shortcomings encountered while using the 8421 BCD code to add two decimal digits whose sum exceeds 9. The excess-3 code has no such limitation, and it considerably simplifies arithmetic operations. Table (1) lists the excess-3 code for the decimal numbers 0–9. The excess-3 code for a given decimal number is determined by adding ‘3’ to each decimal digit in the given

number and then replacing each digit of the newly found decimal number by its four-bit binary equivalent. It may be mentioned here that, if the addition of '3' to a digit produces a carry, as is the case with the digits 7, 8 and 9, that carry should not be taken forward. The result of addition should be taken as a single entity and subsequently replaced with its excess-3 code equivalent. As an example, let us find the excess-3 code for the decimal number 597:

• The addition of '3' to each digit yields the three new digits/numbers '8', '12' and '10'.
• The corresponding four-bit binary equivalents are 1000, 1100 and 1010 respectively.
• The excess-3 code for 597 is therefore given by: 1000 1100 1010=100011001010.

Also, it is normal practice to represent a given decimal digit or number using the maximum number of digits that the digital system is capable of handling. For example, in four-digit decimal arithmetic, 5 and 37 would be written as 0005 and 0037 respectively. The corresponding 8421 BCD equivalents would be 0000000000000101 and 0000000000110111 and the excess-3 code equivalents would be 0011001100111000 and 0011001101101010. Corresponding to a given excess-3 code, the equivalent decimal number can be determined by first splitting the number into four-bit groups, starting from the radix point, and then subtracting 0011 from each four-bit group. The new number is the 8421 BCD equivalent of the given excess-3 code, which can subsequently be converted into the equivalent decimal number. As an example, following these steps, the decimal equivalent of excess-3 number 01010110.10001010 would be 23.57. Another significant feature that makes this code attractive for performing arithmetic operations is that the complement of the excess-3 code of a given decimal number yields the excess-3 code for 9's complement of the decimal number. As adding 9's complement of a decimal number

B to a decimal number A achieves A – B, the excess-3 code can be used effectively for both addition and subtraction of decimal numbers.

Table (1) lists the excess-3 code for the decimal numbers 0–9.

Decimal number	***Excess-3 code***	***Decimal number***	***Excess-3 code***
0	0011	5	1000
1	0100	6	1001
2	0101	7	1010
3	0110	8	1011
4	0111	9	1100

Example

*Find (a) the excess-3 equivalent of (237.75)*10 *and (b) the decimal equivalent of the excess-3 number 110010100011.01110101.*

Solution

(a) Integer part=237. The excess-3 code for (237)10 is obtained by replacing 2, 3 and 7 with the four-bit binary equivalents of 5, 6 and 10 respectively. This gives the excess-3 code for (237)10 as: 0101 0110 1010=010101101010. Binary Codes **23** Fractional part=.75. The excess-3 code for (.75)10 is obtained by replacing 7 and 5 with the four-bit binary equivalents of 10 and 8 respectively. That is, the excess-3 code for (.75)10 =.10101000. Combining the results of the integral and fractional parts, the excess-3 code for (237.75)10 =010101101010.10101000.

(b) The excess-3 code=110010100011.01110101=1100 1010 0011.0111 0101. Subtracting 0011 from each four-bit group, we obtain the new number as:
1001 0111 0000.0100 0010. Therefore, the decimal equivalent=(970.42)10.

Gray Code

The Gray code was designed by Frank Gray at Bell Labs and patented in 1953. It is an unweighted binary code in which two successive values differ only by 1 bit. Owing to this feature, the maximum error that can creep into a system using the binary Gray code to encode data is much less than the worst-case error encountered in the case of straight binary encoding. Table (3) lists the binary and Gray code equivalents of decimal numbers 0–15. An examination of the four-bit Gray code numbers, as listed in Table (3), shows that the last entry rolls over to the first entry. That is, the last and the first entry also differ by only 1 bit. This is known as the *cyclic property* of the Gray code. Although there can be more than one Gray code for a given word length, the term was first applied to a specific binary code for non-negative integers and called the *binary-reflected Gray code* or simply the Gray code.

There are various ways by which Gray codes with a given number of bits can be remembered. One such way is to remember that the least significant bit follows a repetitive pattern of '2' (11, 00, 11, _ _ _), the next higher adjacent bit follows a pattern of '4' (1111, 0000, 1111, _ _ _) and so on. We can also generate the n-bit Gray code recursively by prefixing a '0' to the Gray code for n−1 bits to obtain the first 2n−1 numbers, and then prefixing '1' to the reflected Gray code for n−1 bits to obtain the remaining 2n−1 numbers. The reflected Gray code is nothing but the code written in reverse order.

Table (3) Gray code.

Decimal	***Binary***	***Gray***	***Decimal***	***Binary***	***Gray***
0	0000	0000	8	1000	1100
1	0001	0001	9	1001	1101
2	0010	0011	10	1010	1111
3	0011	0010	11	1011	1110
4	0100	0110	12	1100	1010
5	0101	0111	13	1101	1011
6	0110	0101	14	1110	1001
7	0111	0100	15	1111	1000

Binary–Gray Code Conversion

A given binary number can be converted into its Gray code equivalent by going through the following steps:

1. Begin with the most significant bit (MSB) of the binary number. The MSB of the Gray code equivalent is the same as the MSB of the given binary number.

2. The second most significant bit, adjacent to the MSB, in the Gray code number is obtained by adding the MSB and the second MSB of the binary number and ignoring the carry, if any. That is, if the MSB and the bit adjacent to it are both ‘1’, then the corresponding Gray code bit would be a ‘0’.

3. The third most significant bit, adjacent to the second MSB, in the Gray code number is obtained by adding the second MSB and the third MSB in the binary number and ignoring the carry, if any.

4. The process continues until we obtain the LSB of the Gray code number by the addition of the LSB and the next higher adjacent bit of the binary number.

The conversion process is further illustrated with the help of an example showing step-by-step conversion of (1011)2 into its Gray code equivalent:

Binary 1011

Gray code 1- - -

Binary 1011

Gray code 11- -

Binary 1011

Gray code 111-

Binary 1011

Gray code 1110

Gray Code–Binary Conversion

A given Gray code number can be converted into its binary equivalent by going through the following steps:

1. Begin with the most significant bit (MSB). The MSB of the binary number is the same as the MSB of the Gray code number.

2. The bit next to the MSB (the second MSB) in the binary number is obtained by adding the MSB in the binary number to the second MSB in the Gray code number and disregarding the carry, if any.

3. The third MSB in the binary number is obtained by adding the second MSB in the binary number to the third MSB in the Gray code number. Again, carry, if any, is to be ignored.

4. The process continues until we obtain the LSB of the binary number. The conversion process is further illustrated with the help of an example showing step-by-step conversion of the Gray code number 1110 into its binary equivalent:

Gray code 1110

Binary 1- - -

Gray code 1110

Binary 10 - -

Gray code 1110

Binary 101

Gray code 1110

Binary 1011

Example

Find (a) the Gray code equivalent of decimal 13 and (b) the binary equivalent of Gray code number 1111.

Solution

(a) The binary equivalent of decimal 13 is 1101.

Binary–Gray conversion

Binary 1101

Gray 1- - -

Binary 1101

Gray 10 - -

Binary 1101

Gray 101 –

Binary 1101

Gray 1011

(b) *Gray–binary conversion*

Gray 1111

Binary 1- - -

Gray 1111

Binary 10- -

Gray 1111

Binary 101-

Gray 1111

Binary 1010

Logic Gates

The Logic gates are electronic circuits that can be used to implement the most elementary logic expressions, also known as Boolean expressions. The logic gate is the most basic building block of combinational logic. There are three basic logic gates, namely the OR gate, the AND gate and the NOT gate. Other logic gates that are derived from these basic gates are the NAND gate, the NOR gate, the EXCLUSIVEOR gate and the EXCLUSIVE-NOR gate.

Positive and Negative Logic

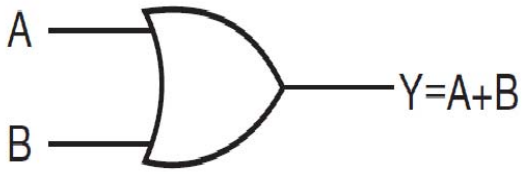
The binary variables, as we know, can have either of the two states, i.e. the logic '0' state or the logic '1' state. These logic states in digital systems such as computers, for instance, are represented by two different voltage levels or two different current levels. If the more positive of the two voltage or current levels represents a logic '1' and the less positive of the two levels represents a logic '0', then the logic system is referred to as a *positive logic system*. If the more positive of the two voltage or current levels represents a logic '0' and the less positive of the two levels represents a logic '1', then the logic system is referred to as a *negative logic system*. The following examples further illustrate this concept. If the two voltage levels are 0 V and +5 V, then in the positive logic system the 0 V represents a logic '0' and the +5 V represents a logic '1'. In the negative logic system, 0 V represents a logic '1' and +5 V represents a logic '0'. If the two voltage levels are 0 V and −5 V, then in the positive logic system the 0 V represents a logic '1' and the −5 V represents a logic '0'. In the negative logic system, 0 V represents a logic '0' and −5 V represents a logic '1'. It is interesting to note, as we will discover in the latter part of the chapter, that a positive OR is a negative AND. That is, OR gate hardware in the positive logic system behaves like an AND gate in the negative logic system. The reverse is also true. Similarly, a positive NOR is a negative NAND, and vice versa.

OR Gate

An OR gate performs an ORing operation on two or more than two logic variables. The OR operation on two independent logic variables A and B is written as Y = A+B and reads as Y equals A OR B and not as A plus B. An OR gate is a logic circuit with two or more inputs and one output. The output of an OR gate is LOW only when all of its inputs are LOW. For all other possible input combinations, the output is HIGH. This statement when interpreted for a positive logic system means the following.

The output of an OR gate is a logic '0' only when all of its inputs are at logic '0'. For all other possible input combinations, the output is a logic '1'. Figure below shows the circuit symbol and the truth table of a two-input OR gate. The operation of a two-input OR gate is explained by the logic expression:

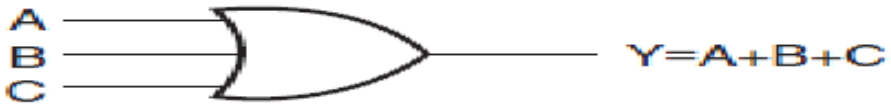
Y = A+B



A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

As an illustration, if we have four logic variables and we want to know the logical output of (A+ B+C +D), then it would be the output of a four-input OR gate with A, B, C and D as its inputs. Y=A+B Figures (a) and (b) show the circuit symbol of

three-input and four-input OR gates. Figure (c) shows the truth table of a three-input OR gate. Logic expressions explaining the functioning of threei nput and four-input OR gates are Y = A+B+C and Y = A+B+C +D.



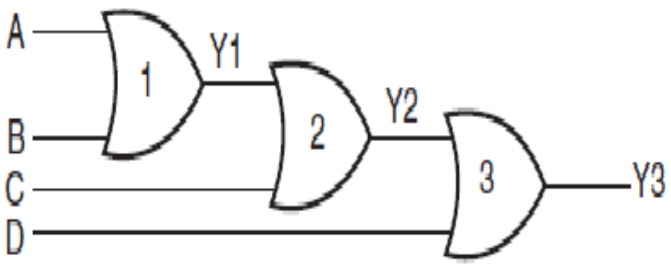
(a)

(b)

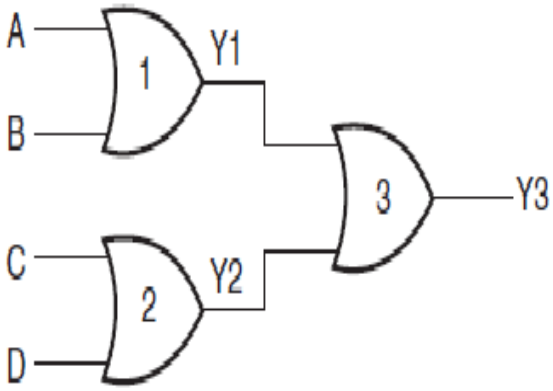
A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

(c)

Q/How would you hardware-implement a four-input OR gate using two-input OR gates only?

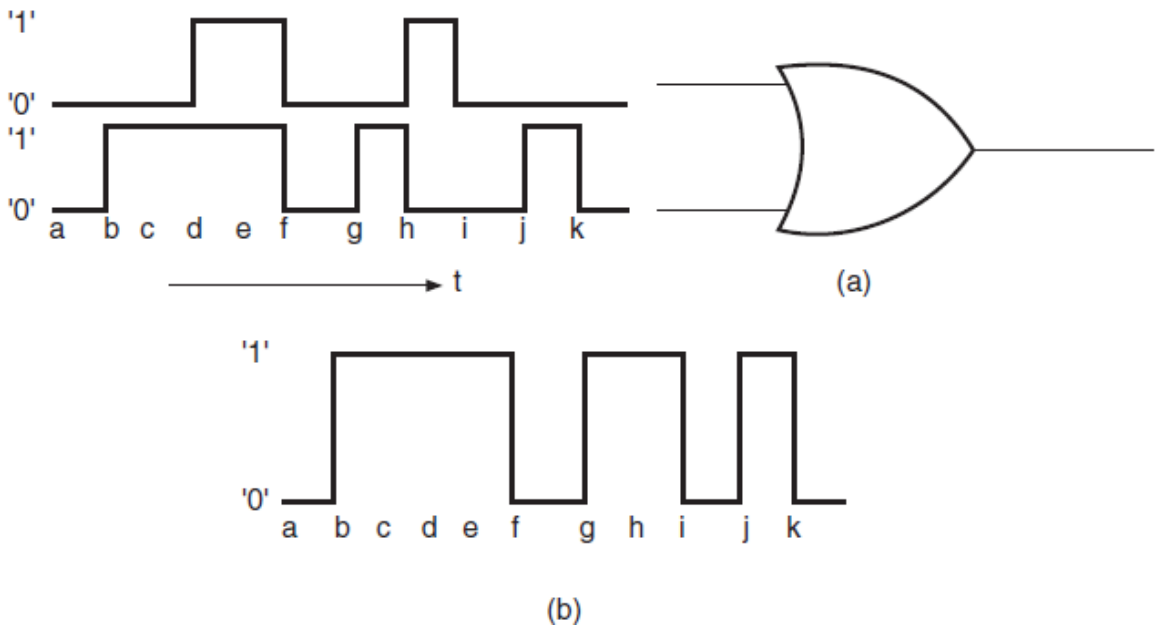


(a)



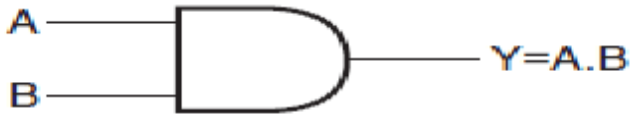
(b)

Example : ***Draw the output waveform for the OR gate and the given pulsed input waveforms of Fig.***



AND Gate

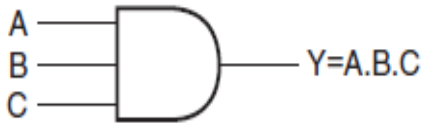
An AND gate is a logic circuit having two or more inputs and one output. The output of an AND gate is HIGH only when all of its inputs are in the HIGH state. In all other cases, the output is LOW. When interpreted for a positive logic system, this means that the output of the AND gate is a logic '1' only when all of its inputs are in logic '1' state. In all other cases, the output is logic '0'. The logic symbol and truth table of a two-input AND gate are shown below The AND operation on two independent logic variables A and B is written as Y = A.B and reads as Y equals A AND B and not as A multiplied by B. Here, A and B are input logic variables and Y is the output. An AND gate performs an ANDing operation:



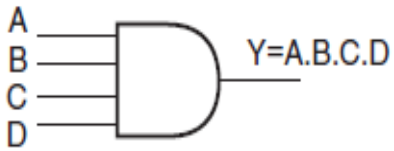
(a)

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

(b)



(a)



(b)

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

(c)

• for a two-input AND gate, Y = A.B;

• for a three-input AND gate, Y = A.B.C;

• for a four-input AND gate, Y = A.B.C.D.

If we interpret the basic definition of OR and AND gates for a negative logic system, we have an interesting observation. We find that an OR gate in a positive logic system is an AND gate in a negative logic system. Also, a positive AND is a negative OR.

Example :

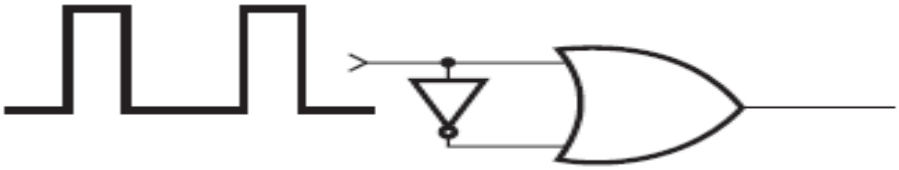
Show the logic arrangement for implementing a four-input AND gate using two-input AND gates only.

NOT Gate

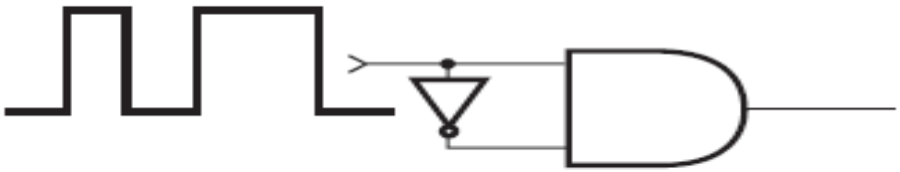
A NOT gate is a one-input, one-output logic circuit whose output is always the complement of the input. That is, a LOW input produces a HIGH output, and vice versa. When interpreted for a positive logic system, a logic '0' at the input produces a logic '1' at the output, and vice versa. It is also known as a 'complementing circuit' or an 'inverting circuit'.

The NOT operation on a logic variable X is denoted as $\overline{X}$ or X'. That is, if X is the input to a NOT circuit, then its output Y is given by $Y = \overline{X}$ or X' and reads as Y equals NOT X. Thus, if X = 0, Y = 1 and if X = 1, Y = 0.

Q /For the logic circuit arrangements of Figs (a) and (b), draw the output waveform.



(a)

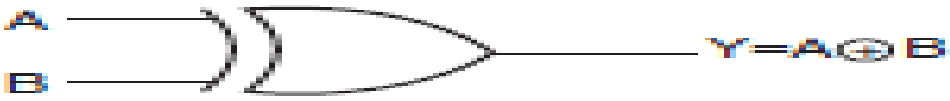


(b)

EXCLUSIVE-OR Gate

The EXCLUSIVE-OR gate, commonly written as EX-OR gate, is a two-input, one-output gate. Figures (a) and (b) respectively show the logic symbol and truth table of a two-input EX-OR gate. As can be seen from the truth table, the output of an EX-OR gate is a logic '1' when the inputs are unlike and a logic '0' when the inputs are like. Although EX-OR gates are available in integrated circuit form only as two-input gates, unlike other gates which are available in multiple inputs also, multiple-input EX-OR logic functions can be implemented using more than one two-input gates. The truth table of a multiple-input EX-OR function can be expressed as follows. The output of a multiple-input EX-OR logic function is a logic '1' when the number of 1s in the input sequence is odd and a logic '0' when the number of 1s in the input sequence is even, including zero. That is, an all 0s input sequence also produces a logic '0' at the output. Figure(c) shows the truth table of a four-input EX-OR function. The output of a two-input EX-OR gate is expressed by

$$Y = A \oplus B = \overline{A}B + A\overline{B}$$



(a)

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

(b)

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

(c)

Q: How do you implement three-input and four-input X-OR logic functions with the help of two-input EX-OR gates?

Q: How can you implement a NOT circuit using a two-input EX-OR gate?

NAND Gate

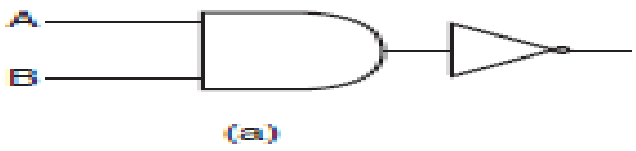
NAND stands for NOT AND. An AND gate followed by a NOT circuit makes it a NAND gate [Fig. (a)]. Figure (b) shows the circuit symbol of a two-input NAND gate. The truth table of a NAND gate is obtained from the truth table of an AND gate by complementing the output entries [Fig. (c)]. The output of a NAND gate is a logic

'0' when all its inputs are a logic '1'. For all other input combinations, the output is a logic '1'. NAND gate operation is logically expressed as

$Y = \overline{A.B}$

In general, the Boolean expression for a NAND gate with more than two inputs can be written as

$Y = \overline{A.B.C.D}$



(a)

(b)

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

(c)

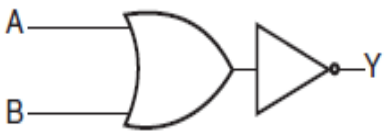
NOR Gate

NOR stands for NOT OR. An OR gate followed by a NOT circuit makes it a NOR gate [Fig. (a)]. The truth table of a NOR gate is obtained from the truth table of an OR gate by complementing the output entries. The output of a NOR gate is a logic '1' when all its inputs are logic '0'. For all other input combinations, the output is a logic '0'. The output of a two-input NOR gate is logically expressed as

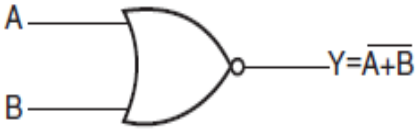
$Y = \overline{A + B}$

In general, the Boolean expression for a NOR gate with more than two inputs can be written as

$Y = \overline{A + B + C + D}$



(a)



(b)

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

(c)

EXCLUSIVE-NOR Gate

EXCLUSIVE-NOR (commonly written as EX-NOR) means NOT of EX-OR, i.e. the logic gate that we get by complementing the output of an EX-OR gate. The truth table of an EX-NOR gate is obtained from the truth table of an EX-OR gate by complementing the output entries. Logically, The output of a two-input EX-NOR gate is a logic '1' when the inputs are like and a logic '0' when they are unlike. In general, the output of a multiple-input EX-NOR logic function is a logic '0' when the number of 1s in the input sequence is odd and a logic '1' when the number of 1s in the input sequence is even including zero. That is, an all 0s input sequence also produces a logic '1' at the output.

$$Y = \overline{A \oplus B} = AB + \overline{A}\,\overline{B}$$

(a)

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

(b)

H.W :

Show the logic arrangements for implementing:
(a) a four-input NAND gate using two-input AND gates and NOT gates;
(b) a three-input NAND gate using two-input NAND gates;
(c) a NOT circuit using a two-input NAND gate;
(d) a NOT circuit using a two-input NOR gate;
(e) a NOT circuit using a two-input EX-NOR gate.

H.W:

How do you implement a three-input EX-NOR function using only two-input EX-NOR gates?

Boolean Algebra and Simplification Techniques

Boolean algebra is mathematics of logic. It is one of the most basic tools available to the logic designer and thus can be effectively used for simplification of complex logic expressions. Other useful and widely used techniques based on Boolean theorems include the use of Karnaugh maps in what is known as the mapping method of logic simplification.

Equivalent and Complement of Boolean Expressions

Two given Boolean expressions are said to be *equivalent* if one of them equals '1' only when the other equals '1' and also one equals '0' only when the other equals '0'. They are said to be the *complement* of each other if one expression equals '1' only when the other equals '0', and vice versa. The complement of a given Boolean expression is obtained by complementing each literal, changing all '.' to '+' and all '+' to '.', all 0s to 1s and all 1s to 0s. The examples below give some Boolean expressions and their complements:

$\bar{A}B+A\bar{B} = \overline{\bar{A}B + A\bar{B}} = (A+\bar{B}).(\bar{A}+B)$

Where given Boolean expression

$(A+B).(\bar{A}+\bar{B}) = \overline{(A + B).(\bar{A} + \bar{B})} = AB+\bar{A}\,\bar{B}$

Example

Find (a) the complement of $[(A\,\bar{B} +\bar{C})\,D + \bar{E}]\,F$.

Solution

(a) The complement of $(A\,\bar{B} +\bar{C})\,D + \bar{E})\,F$ ***is given by*** $[\,(\bar{A}+B)\,C +\bar{D}]\,E+ \bar{F}$

Example :

Simplify (AB+CD). (($\overline{A}$+$\overline{B}$).($\overline{C}$+$\overline{D}$)) = 0

Theorems of Boolean Algebra

The theorems of Boolean algebra can be used to simplify many a complex Boolean expression and also to transform the given expression into a more useful and meaningful equivalent expression.

Theorem 1 (Operations with '0' and '1'):

(a) 0.X = 0

(b) 1+X = 1

Theorem 2 (Operations with '0' and '1'):

(a) 1.X = X
(b) 0+X = X

Theorem 3 (Idempotent or Identity Laws):

(a) X.X.X… ……. .X = X
(b) X+X+X +··· +X = X

Theorem 4 (Complementation Law):

(a) $\overline{X}$X = 0
(b) X+$\overline{X}$ = 1

Theorem 5 (Commutative Laws)

(a) X+Y = Y +X

(b) X.Y = Y.X

Theorem 6 (Associative Laws):

(a) X+(Y +Z)= Y +(Z+X) = Z+(X+Y)

(b) X.(Y.Z)= Y .(Z.X) = Z.(X.Y)

Theorem 7 (Distributive Laws)

(a) X.(Y +Z) = X.Y +X.Z

(b) X+Y.Z = (X+Y) .(X+Z)

Theorem 8 :

(a) $X.Y + \bar{X}Y = X$

(b) $(X+Y).(X+\bar{Y}) = X$

Theorem 9 :

(a) $(X+\bar{Y}).Y = X.Y$

(b) $X.\bar{Y} + Y = X+Y$ [$X.\bar{Y} + Y(1+X) = X.\bar{Y} + Y + XY = X(\bar{Y} + Y) + Y$]

Theorem 10 (Absorption Law or Redundancy Law) :

(a) X+X.Y = X

(b) X.(X+Y) = X

Theorem 11:

(a) Z.X+ Z.$\bar{X}$.Y = Z.X+Z.Y

(b) (Z+X) . (Z+$\bar{X}$+Y) = (Z+X).(Z+Y)

Theorem 12 (Consensus Theorem):

(a) X.Y + $\bar{X}$.Z +Y.Z = X.Y +X.$\bar{Z}$

(b) (X+Y).($\bar{X}$+Z).(Y +Z) = (X+Y).($\bar{X}$+Z)

Theorem 13 (DeMorgan's Theorem):

(a) $\overline{(X1 + X2 + X3 + \cdots \ldots \ldots \ldots + Xn)} = \bar{X1}.\bar{X2}.\bar{X3}\ldots\ldots\ldots\ldots\ldots\ldots\bar{Xn}$

(b) $\overline{(X1.X2.X3.X4 \ldots \ldots \ldots \ldots \ldots \ldots \ldots Xn)} = \bar{X1}+\bar{X2}+\bar{X3}\ldots\ldots\ldots\ldots\ldots\ldots+\bar{Xn}$

Theorem 14 (Transposition Theorem):

(a) X.Y + $\bar{X}$.Z = (X+Z) . ($\bar{X}$+Y)

(b) (X+Y).($\bar{X}$+Z) = X.Z+ $\bar{X}$Y

Theorem 15 (Involution Law) :

X = $\bar{\bar{X}}$

****

Example : *apply Demorgan's theorm to simplify*

$Y = \overline{\overline{A + B\bar{C}} + D.\ \overline{(E + \bar{F})}}$

Solution :

$Y = \overline{\overline{A + B\bar{C}} + D\ \overline{(E + \bar{F})}} = (\ \overline{\overline{A + B\bar{C}}}\ ).(\ D.\ \overline{\overline{(E + \bar{F})}}\ )$

$= (A+B\bar{C}).(\bar{D}+(E+\bar{F}))$

Example : *simplify the expression , using Boolean algebra techniques ?*

Y = AB + A(B+C) +B(B+C)

Y = AB +AB +AC+BB+BC

Y = AB+AC +B+BC

Y = AB +AC +B(1+C)

Y= AB +AC +B

Y = AC +B(1+C)

Y = AC+B

Sum-of-Products Boolean Expressions

A sum-of-products expression contains the sum of different terms, with each term being either a single literal or a product of more than one literal. It can be obtained from the truth table directly by considering those input combinations that produce a

logic ‘1’ at the output. Each such input combination produces a term. Different terms are given by the product of the corresponding literals.

Example : *Convert each of the following Boolean expression to SOP form ?*

$Y = A\bar{B}C + \bar{A}\bar{B} + AB\bar{C}\bar{D}$

$Y = A\bar{B}C(D+\bar{D}) + \bar{A}\bar{B}(D+\bar{D})\,(C+\bar{C}) + AB\bar{C}\bar{D}$

$Y = A\bar{B}CD + A\bar{B}C\bar{D} + [\bar{A}\bar{B}D + \bar{A}\bar{B}\bar{D}]\,(C+\bar{C}) + AB\bar{C}\bar{D}$

$Y = A\bar{B}CD + A\bar{B}C\bar{D} + [\bar{A}\bar{B}DC + \bar{A}\bar{B}\bar{D}C] + [\bar{A}\bar{B}\bar{C}D + \bar{A}\bar{B}\bar{D}\bar{C}] + AB\bar{C}\bar{D}$

Example : *find the truth table of next equation*

$\boldsymbol{Y = ABC + A\bar{B}C + \bar{A}B\bar{C} + \bar{A}\bar{B}\bar{C}}$

Decimal	ABC	Y
0	000	1
1	001	0
2	010	1
3	011	0
4	100	0
5	101	1
6	110	0
7	111	1

Product-of-Sums Expressions

A product-of-sums expression contains the product of different terms, with each term being either a single literal or a sum of more than one literal. It can be obtained from the truth table by considering those input combinations that produce a logic ‘0’ at the output. Each such input combination gives a term, and the product of all such terms gives the expression. Different terms are obtained by taking the sum of the corresponding literals. Here, ‘0’ and ‘1’ respectively mean the uncomplemented and

complemented variables, unlike sum-of-products expressions where '0' and '1' respectively mean complemented and uncomplemented variables.

Example : *find the (POS)?*

Decimal	ABC	Y
0	000	0
1	001	1
2	010	1
3	011	1
4	100	1
5	101	0
6	110	0
7	111	1

$$Y = (A+B+C) .(\bar{A}+B+\bar{C}) .(\bar{A}+\bar{B}+C)$$

∑and π Nomenclature

∑ and notations are respectively used to represent sum-of-products and ***π*** product-of-sums Boolean expressions. So for last example

Y= $\pi(0, 5,6)$

Y= $\sum(1,2,3,4,7)$

Karnaugh Map Method

A Karnaugh map is a graphical representation of the logic system. It can be drawn directly from either minterm (sum-of-products) or maxterm (product-of-sums) Boolean expressions. Drawing a Karnaugh map from the truth table involves an additional step of writing the minterm or maxterm expression depending upon whether it is desired to have a minimized sum-of-products or a minimized product of

sums expression. Having drawn the Karnaugh map, the next step is to form groups of 1s as per the following guidelines:

1. Each square containing a '1' must be considered at least once, although it can be considered as often as desired.
2. The objective should be to account for all the marked squares in the minimum number of groups.
3. The number of squares in a group must always be a power of 2, i.e. groups can have 1, 2, 4_ 8, 16, squares.
4. Each group should be as large as possible, which means that a square should not be accounted for by itself if it can be accounted for by a group of two squares; a group of two squares should not be made if the involved squares can be included in a group of four squares and so on.
5. 'Don't care' entries can be used in accounting for all of 1-squares to make optimum groups. They are marked 'X' in the corresponding squares. It is, however, not necessary to account for all 'don't care' entries. Only such entries that can be used to advantage should be used.

X / Y	0	1
0	$\overline{X}\overline{Y}$	$X\overline{Y}$
1	$\overline{X}Y$	XY

X / Y	0	1
0	m_0	m_1
1	m_2	m_3

Karnaugh map of two variables

YZ / X	00	01	11	10
0	m_0	m_1	m_3	m_2
1	m_4	m_5	m_7	m_6

Karnaugh map of three variables

Example : Simplify the Boolean function

F(X. Y. Z) = ∑ (2.3.4.5)

YZ / X	00	01	11	10
0	0	0	1	1
1	1	1	0	0

$$F = x'y + xy'$$

Example : *use Karnaugh map to minimize the following POS expression :*

(A+B+C). (A+B+$\bar{C}$). (A+ $\bar{B}$ + C) .(A+$\bar{B}$+$\bar{C}$).) .($\bar{A}$ +$\bar{B}$+C)

= (0+0+0). (0+0+1). (0+ 1 + 0) .(0+1+1).) .(1 +1+0)

AB / C	00	01	11	10
0	0	0	0	1
1	0	0	1	1

F=A. ($\bar{B} + C$) =A$\bar{B}$ +AC

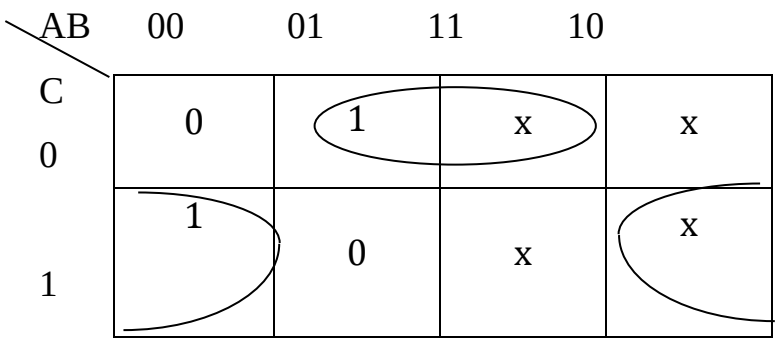
YZ / X	00	01	11	10
00	m_0	m_1	m_3	m_2
01	m_4	m_5	m_7	m_6
11	m_{12}	m_{13}	m_{15}	m_{14}
01	m_8	m_9	m_{11}	m_{10}

Karnaugh map of four variables

Don't care condition :

Example : find output function using the following truth table

Decimal	**ABC**	**Y**
0	000	0
1	001	1
2	010	1
3	011	0
4	100	x
5	101	x
6	110	x
7	111	x



$Y = \overline{B}C + B\overline{C}$

Combinational Circuits

A *combinational circuit* is one where the output at any time depends only on the present combination of inputs at that point of time with total disregard to the past state of the inputs. The logic gate is the most basic building block of combinational logic. The logical function performed by a combinational circuit is fully defined by a set of Boolean expressions.

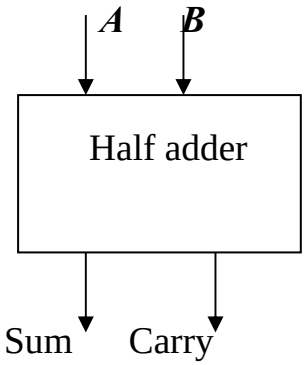
Arithmetic Circuits – Basic Building Blocks

Half-Adder

A *half-adder* is an arithmetic circuit block that can be used to add two bits. Such a circuit thus has two inputs that represent the two bits to be added and two outputs, with one producing the SUM output and the other producing the CARRY.

Sum = $\overline{A}B+A\overline{B}= A \oplus B$ Carry =AB

A	***B***	***S***	***C***
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Full Adder

A *full adder* circuit is an arithmetic circuit block that can be used to add three bits to produce a SUM and a CARRY output.

Sum = $\overline{A}B+A\overline{B}= A \oplus B$ Carry =AB

A	B	Carry- in (Cin)	Sum	Carry-out (Cout)
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Combinational Circuits

A *combinational circuit* is one where the output at any time depends only on the present combination of inputs at that point of time with total disregard to the past state of the inputs. The logic gate is the most basic building block of combinational logic. The logical function performed by a combinational circuit is fully defined by a set of Boolean expressions.

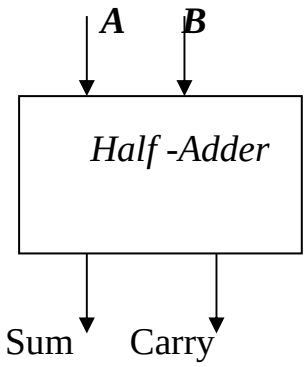
Arithmetic Circuits – Basic Building Blocks

Half-Adder

A *half-adder* is an arithmetic circuit block that can be used to add two bits. Such a circuit thus has two inputs that represent the two bits to be added and two outputs, with one producing the SUM output and the other producing the CARRY.

Sum = $\bar{A}B+A\bar{B}= A\oplus B$ **Carry =AB**

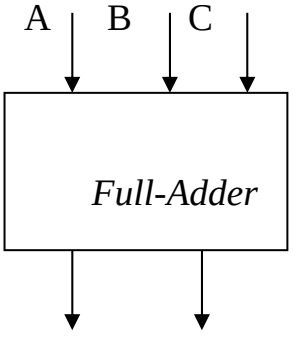
A	***B***	***S***	***C***
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Full Adder

A *full adder* circuit is an arithmetic circuit block that can be used to add three bits to produce a SUM and a CARRY output.

Sum = $\bar{A}\bar{B}C+\bar{A}B\bar{C}+A\bar{B}C+ABC= A\oplus B\oplus C$

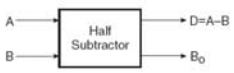


$$Carry = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC = AB + (A \oplus B).C$$ *Sum* *carry*

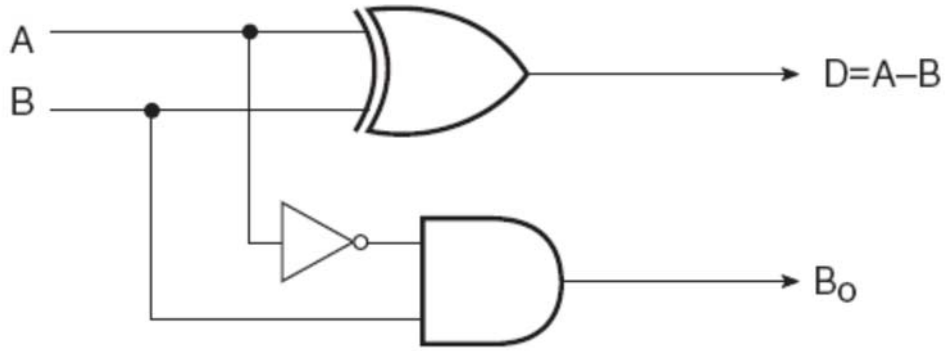
A	B	Carry- in (Cin)	Sum	Carry-out (Cout)
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Half-Subtractor

A half-subtractor is a combinational circuit that can be used to subtract one binary digit from another to produce a DIFFERENCE output and a BORROW output. The BORROW output here specifies whether a '1' has been borrowed to perform the subtraction. The truth table of a half-subtractor,. The Boolean expressions for the two outputs are given by the equations



A	B	D	B_O
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0



DIFFERENCE = $\bar{A}B+A\bar{B}= A\oplus B$ **BORROW**

$=\bar{A}B$

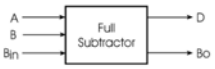
Full Subtractor

A *full subtractor* performs subtraction operation on two bits, a minuend and a subtrahend, and also takes into consideration whether a ‘1’ has already been borrowed by the previous adjacent lower minuend bit or not. As a result, there are three bits to be handled at the input of a full subtractor, namely the two bits to be subtracted and a borrow bit designated as Bin . There are two outputs, namely the

DIFFERENCE output D and the BORROW output Bo. The BORROW output bit tells whether the minuend bit needs to borrow a '1' from the next possible higher minuend bit.

$$\textbf{DIFFERENCE} = A \oplus B \oplus C$$

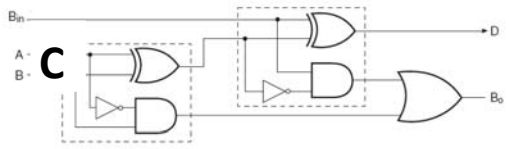
$$\textbf{BORROW} = \bar{A}B + (A \oplus B).\bar{C}$$



Minuend (A)	Subtrahend (B)	Borrow In (B_{in})	Difference (D)	Borrow Out (B_O)
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

C

C


Bin
A
B
C
D
Bo