Module-I

One's Complement: Complement all the bits .i.e. makes all 1s as 0s and all 0s as 1s

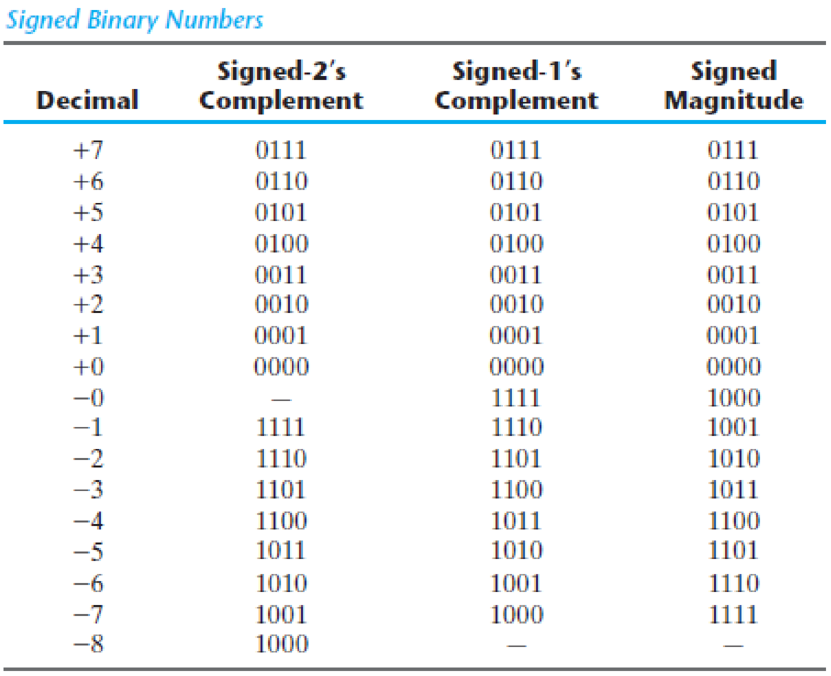
Two's Complement: One's complement+1

SIGNED BINARY NUMBERS

Positive integers (including zero) can be represented as unsigned numbers. However, to represent negative integers, we need a notation for negative values. There are three different format to represent a negative (signed) number. For example:

Three different ways to represent -9 with eight bits:

Signed magnitude representation:	10001001
signed-1's-complement representation:	11110110
signed-2's-complement representation:	11110111



Signed Binary Numbers

Decimal	Signed-2's Complement	Signed-1's Complement	Signed Magnitude
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
−0	—	1111	1000
−1	1111	1110	1001
−2	1110	1101	1010
−3	1101	1100	1011
−4	1100	1011	1100
−5	1011	1010	1101
−6	1010	1001	1110
−7	1001	1000	1111
−8	1000	—	—

Two's Complement as -ve Number

Two's complement is -ve number because binary addition of a *n*-bit number with it's complement gives n bit result with all bits = 0s

Highest Two's Complement format +ve Number: A highest positive arithmetic number is when at msb there is 0 and all remaining bits are 1s

Lowest Two's Complement format -ve Number : A lowest negative arithmetic number is when at msb there is 1 and all remaining bits are 0s

Therefore for 8-bits

- Maximum 8-bit number = *0111 1111*(+127)
- Minimum 8-bit number = 1000 0000 (−128)

Subtraction using 1′ and 2′ complements

Example: Calculate the following binary Subtraction: 11101.101 – 1011.11, then verify the result in decimal System.

Direct subtraction

$$\begin{array}{rr} & 11101.101 \\ - & 01011.110 \\ \hline & 10001.111 \end{array}$$

1's complement

$$\begin{array}{rr} & 11101.101 \\ + & 10100.001 \\ \hline & \boxed{1}\,10001.110 \\ + & 1 \\ \hline & 10001.111 \end{array}$$

2's complement

$$\begin{array}{rr} & 11101.101 \\ + & 10100.010 \\ \hline & \boxed{\cancel{1}}\,10001.111 \end{array}$$

Direct subtraction

$$\begin{array}{rr} & 29.625 \\ - & 11.750 \\ \hline & 17.875 \end{array}$$

9's complement

$$\begin{array}{rr} & 29.625 \\ + & 88.249 \\ \hline & \boxed{1}\,17.874 \\ + & 1 \\ \hline & 17.875 \end{array}$$

10's complement

$$\begin{array}{rr} & 29.625 \\ + & 88.250 \\ \hline & \boxed{\cancel{1}}\,17.875 \end{array}$$

Important Note:

- When using the complement methods in subtraction and having no additional 1 in the extreme left cell, then, this means a negative result.
- In this case, the solution is the negative of 1's complement of the result (if using 1's complement initially), or the negative of 2's complement of the result (if using 2's complement initially).

  It is shown in the following examples, where the results are –ve.

Direct subtraction

$$\begin{array}{rr} & 01101.101 \\ - & 11011.110 \\ \hline & \end{array}$$

1's complement

$$\begin{array}{rr} & 01101.101 \\ + & 00100.001 \\ \hline & \boxed{0}\,10001.110 \end{array}$$

2's complement

$$\begin{array}{rr} & 01101.101 \\ + & 00100.010 \\ \hline & \boxed{0}\,10001.111 \end{array}$$

BINARY CODES

In the coding, when numbers, letters or words are represented by a specific group of symbols, it is said that the number, letter or word is being encoded. The group of symbols is called as a code. The digital data is represented, stored and transmitted as group of binary bits. This group is also called as binary code. The binary code is represented by the number as well as alphanumeric letter.

Binary-Coded Decimal Code

Binary Coded Decimal (BCD) as the name implies is a way of representing Decimal numbers in a 4 bit binary code. BCD numbers are useful when sending data to display devices for example. The numbers 0 through 9 are the only valid BCD values. Notice in the table that the binary and BCD values are the same for the numbers 0 to 9. When we exceed the value of 9 in BCD each digit in the BCD number is now represented by a 4 bit binary value.

In this code each decimal dig it is represented by a 4-bit binary number. BCD is a way to express each of the decimal digits with a binary code. In the BCD, with four bits we can represent sixteen numbers (0000 to 1111). But in BCD code only first ten of these are used (0000 to 1001). The remaining six code combinations i.e. 1010 to 1111 are invalid in BCD.

Number	Binary	BCD
0	0000	0000
1	0001	0001
2	0010	0010
3	0011	0011
4	0100	0100
5	0101	0101
6	0110	0110
7	0111	0111
8	1000	1000
9	1001	1001
10	1010 – invalid BCD number	0001 0000
11	1011 – invalid BCD number	0001 0001

BCD Addition

Consider the addition of two decimal digits in BCD, together with a possible carry from a previous less significant pair of digits. Since each digit does not exceed 9, the sum cannot be greater than 9 + 9 + 1 = 19, with the 1 being a previous carry. Suppose we add the BCD digits as if they were binary numbers. Then the binary sum will produce a result in the range from 0 to 19. In binary, this range will be from 0000 to 10011, but in BCD, it is from 0000 to 1 1001, with the first (i.e., leftmost) 1 being a carry and the next four bits being the BCD sum. When the binary sum is equal to or less than 1001 (without a carry), the corresponding BCD digit is correct. However, when the binary sum is greater than or equal to 1010, the result is an invalid BCD digit. The addition of 6 = (0110)2 to the binary sum converts it to the correct digit and also produces a carry as required. This is because a carry in the most significant bit position of the binary sum and a decimal carry differ by 16 - 10 = 6. Consider the following three BCD additions:

4	0100	4	0100	8	1000
+5	+0101	+8	+1000	+9	1001
9	1001	12	1100	17	10001
			+0110		+0110
			10010		10111

In each case, the two BCD digits are added as if they were two binary numbers. If the binary sum is greater than or equal to 1010, we add 0110 to obtain the correct BCD sum and a carry. In the first example, the sum is equal to 9 and is the correct BCD sum. In the second example, the binary sum produces an invalid BCD digit. The addition of 0110 produces the correct BCD sum, 0010 (i.e., the number 2), and a carry. In the third example, the binary sum produces a carry. This condition occurs when the sum is greater than or equal to 16. Although the other four bits are less than 1001, the binary sum requires a correction because of the carry. Adding 0110, we obtain the required BCD sum 0111 (i.e., the number 7) and a BCD carry.
The addition of two *n*-digit unsigned BCD numbers follows the same procedure.
Consider the addition of 184 + 576 = 760 in BCD:

BCD	1	1		
	0001	1000	0100	184
	+0101	0111	0110	+576
Binary sum	0111	10000	1010	
Add 6		0110	0110	
BCD sum	0111	0110	0000	760

The first, least significant pair of BCD digits produces a BCD digit sum of 0000 and a carry for the next pair of digits. The second pair of BCD digits plus a previous carry produces a digit sum of 0110 and a carry for the next pair of digits. The third pair of digits plus a carry produces a binary sum of 0111 and does not require a correction.
Advantages of BCD Codes: It is very similar to decimal system. We need to remember binary equivalent of decimal numbers 0 to 9 only.

Disadvantages of BCD Codes: The addition and subtraction of BCD have different rules. The BCD arithmetic is little more complicated. BCD needs more number of bits than binary to represent the decimal number. So BCD is less efficient than binary.

Gray Codes

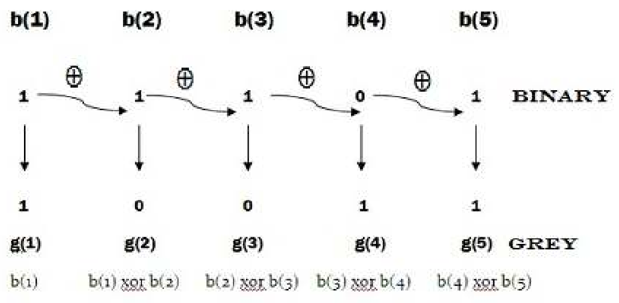
It is the non-weighted code and it is not arithmetic codes. That means there are no specific weights assigned to the bit position. It has a very special feature that has only one bit will chang e, eachtime the decimal number is incremented as shown in the table . As only one bit chang es at a time, the g ray code is called as a unit distance code. The g ray code is a cyclic code. Gray code cannot be used for arithmetic operation

Decimal	BCD Code	Gray Code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101

Binary to Gray Conversion:

Follow the below Steps to convert Binary number to gray code.
Lets Consider the Binary number B_1 B_2 B_3 B_4 ... B_n and the Gray code is G_1 G_2 G_3 G_4 ... G_n

1. Most significant bit (B_1) is same as the most significant bit in Gray Code ($B_1 = G_1$)
2. To find next bit perform Ex-OR (Exclusive OR) between the Current binary bit and previous bit.
   $G_n = B_n$ (Ex-OR) B_{n-1}
3. Look the below Image for Binary to Gray code Conversion



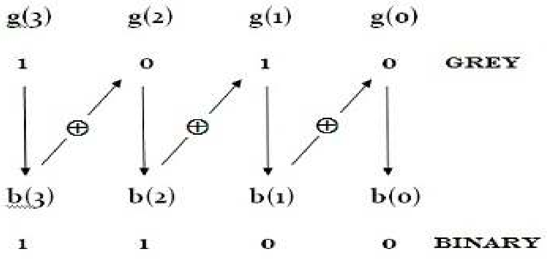
Gary to Binary Conversion

Follow the below steps to convert Gray Code to Binary

1. Most significant bit (G_1) is same as the most significant bit in Binary Code ($G_1 = B_1$)
2. The next number can be obtain by taking Exclusive OR operation between the previous binary bit, and the current gray code bit and write down the value.

Repeat the Above Step until you find B_n

Look at the below example for Converting Binary to Gray Code.



i.e $b(3) = g(3)$

$b(2) = b(3) \oplus g(3)$

$b(1) = b(2) \oplus g(1)$

$b(0) = b(1) \oplus g(0)$

Application of Gray code: Gray code is popularly used in the shaft position encoders. A shaft position encoder produces a code word which represents the angular position of the shaft.

Excess-3 code

The Excess-3 code is also called as XS-3 code. It is non-weighted code used to express decimal numbers. The Excess-3 code words are derived from the 8421 BCD code words adding (0011)2 or (3)10 to each code word in8421. The excess-3 codes are obtained as follows

Decimal Number $\longrightarrow$ BCD Code $\longrightarrow$ Exess-3 Code

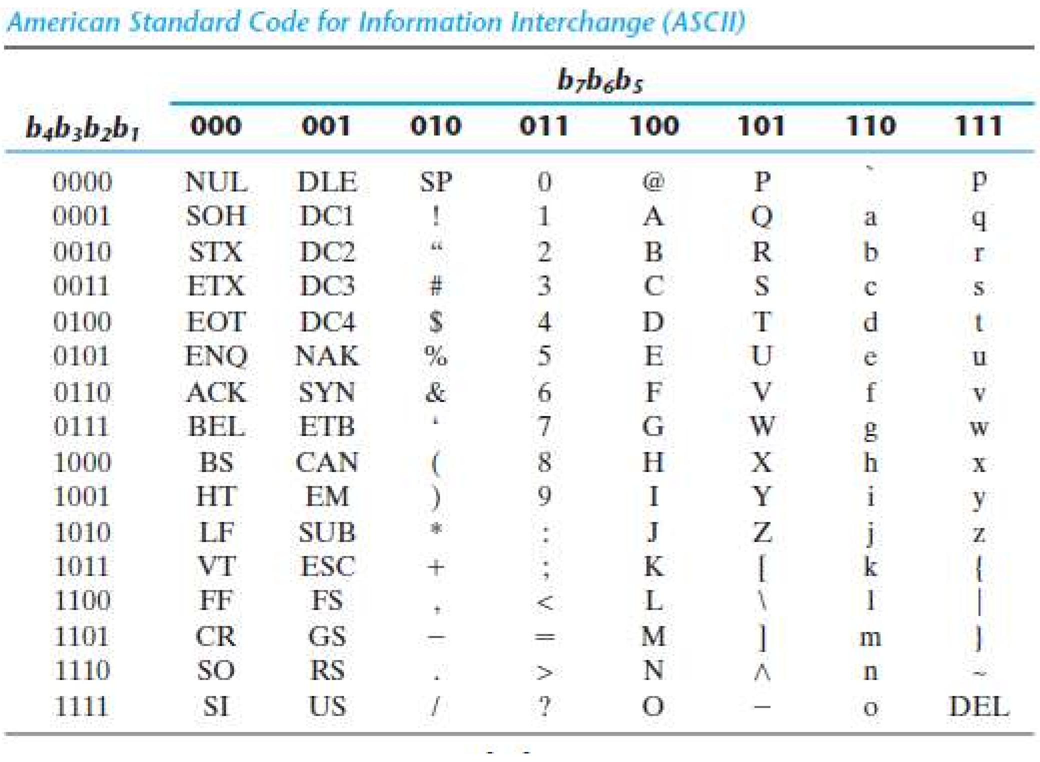
Decimal	**BCD Code**	**Excess-3(BCD+0011)**
0	0000	0011
1	0001	0100
2	0010	0101
3	0011	0110
4	0100	0111
5	0101	1000
6	0110	1001
7	0111	1010
8	1000	1011
9	1001	1100

ASCII Code

(American Standard Code for Information Interchange)

ASCII code is a 7-bit code whereas. ASCII code is more commonly used worldwide. This standard binary code for the alphanumeric characters is the American Standard Code for Information Interchange (ASCII), which uses seven bits to code 128 characters, as shown in Table given below. The seven bits of the code are designated by *b*1 through *b*7, with *b*7 the most significant bit. The letter *A*, for example, is represented in ASCII as 1000001 (column 100, row 0001). The ASCII code also contains 94 graphic characters that can be printed and 34 nonprinting characters used for various control functions.

The graphic characters consist of the 26 uppercase letters (A through Z), the 26 lowercase letters (a through z), the 10 numerals (0 through 9), and 32 special printable characters, such as %, *, and $.



American Standard Code for Information Interchange (ASCII)

	$b_7b_6b_5$							
$b_4b_3b_2b_1$	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	“	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	‘	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	–	=	M	]	m	}
1110	SO	RS	.	>	N	∧	n	~
1111	SI	US	/	?	O	—	o	DEL

The 34 control characters are designated in the following ASCII table with abbreviated names

Control Characters

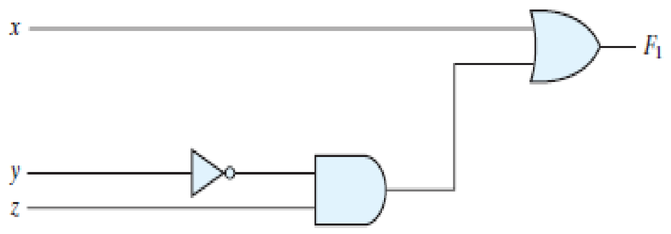
NUL	Null	DLE	Data-link escape
SOH	Start of heading	DC1	Device control 1
STX	Start of text	DC2	Device control 2
ETX	End of text	DC3	Device control 3
EOT	End of transmission	DC4	Device control 4
ENQ	Enquiry	NAK	Negative acknowledge
ACK	Acknowledge	SYN	Synchronous idle
BEL	Bell	ETB	End-of-transmission block
BS	Backspace	CAN	Cancel
HT	Horizontal tab	EM	End of medium
LF	Line feed	SUB	Substitute
VT	Vertical tab	ESC	Escape
FF	Form feed	FS	File separator
CR	Carriage return	GS	Group separator
SO	Shift out	RS	Record separator
SI	Shift in	US	Unit separator
SP	Space	DEL	Delete

BOOLEAN FUNCTIONS

Boolean algebra is an algebra that deals with binary variables and logic operations. A Boolean function described by an algebraic expression consists of binary variables, the constants 0 and 1, and the logic operation symbols. For a given value of the binary variables, the function can be equal to either 1 or 0. As an example, consider the Boolean function

$$F1 = x + y'z$$

A Boolean function can be transformed from an algebraic expression into a circuit diagram composed of logic gates connected in a particular structure. The logic-circuit diagram (also called a schematic) for $F1$ is shown in Figure using different logic gates.



Truth Tables for F_1

x	y	z	F_1
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

CANONICAL AND STANDARD FORMS

Minterms and Maxterms

A binary variable may appear either in its normal form (x) or in its complement form (x')
Now consider two binary variables x and y combined with an AND operation. Since each variable may appear in either form, there are four possible combinations: $x'y'$, $x'y$, xy' and xy. Each of these four AND terms is called a *minterm*, or a *standard product*. In a similar manner, n variables can be combined to form 2n minterms. The 2n different minterms may be determined by a method similar to the one shown in Table given below for three variables. Each minterm is obtained from an AND term of the n variables, with each variable being primed if the corresponding bit of the binary number is a 0 and unprimed if a 1. A symbol for each minterm is also shown in the table and is of the form mj, where the subscript j denotes the decimal equivalent of the binary number of the minterm designated.
In a similar fashion, n variables forming an OR term, with each variable being primed or unprimed, provide 2n possible combinations, called *maxterms*, or *standard sums*. The eight maxterms for three variables, together with their symbolic designations, are listed in Table 2.3 . Any 2n maxterms for n variables may be determined similarly. It is important to note that (1) each maxterm is obtained from an OR term of the n variables, with each variable being unprimed if the corresponding bit is a 0 and primed if a 1, and (2) each maxterm is the complement of its corresponding minterm and vice versa.
A Boolean function can be expressed algebraically from a given truth table by forming a minterm for each combination of the variables that produces a 1 in the function and then taking the OR of all those terms. For example, the function $f1$ in Table 2.4 is

determined by expressing the combinations 001, 100, and 111 as *x_y_z*, *xy_z_*, and *xyz*, respectively. Since each one of these minterms results in $f1 = 1$, we have

$$f1 = x'y'z + xy'z' + xyz = m1 + m4 + m7 = \sum(1,4,7)$$

Minterms and Maxterms for Three Binary Variables

			Minterms		Maxterms	
x	y	z	Term	Designation	Term	Designation
0	0	0	$x'y'z'$	m_0	$x + y + z$	M_0
0	0	1	$x'y'z$	m_1	$x + y + z'$	M_1
0	1	0	$x'yz'$	m_2	$x + y' + z$	M_2
0	1	1	$x'yz$	m_3	$x + y' + z'$	M_3
1	0	0	$xy'z'$	m_4	$x' + y + z$	M_4
1	0	1	$xy'z$	m_5	$x' + y + z'$	M_5
1	1	0	xyz'	m_6	$x' + y' + z$	M_6
1	1	1	xyz	m_7	$x' + y' + z'$	M_7

Standard Forms

Another way to express Boolean functions is in *standard* form.

-SOP (*sum of products*)
-POS (*product of sums*)

The *sum of products* is a Boolean expression containing AND terms, called *product terms,* with one or more literals each. The *sum* denotes the ORing of these terms. An example of a function expressed as a sum of products is

$$F1 = y' + xy + x'yz'$$

The expression has three product terms, with one, two, and three literals. Their sum is, in effect, an OR operation. The logic diagram of a sum-of-products expression consists of a group of AND gates followed by a single OR gate. This configuration pattern is shown in Fig. 2.3 (a). Each product term requires an AND gate, except for a term with a single literal. The logic sum is formed with an OR gate whose inputs are the outputs of the AND gates and the single literal. It is assumed that the input variables are directly available in their complements, so inverters are not included in the diagram. This circuit configuration is referred to as a *two-level implementation*.

A *product of sums* is a Boolean expression containing OR terms, called *sum terms*. Each term may have any number of literals. The *product* denotes the ANDing of these terms. An example of a function expressed as a product of sums is

$$F2 = x\,(y' + z)\,(x' + y + z')$$

This expression has three sum terms, with one, two, and three literals. The product is an AND operation. The use of the words *product* and *sum* stems from the similarity of the AND operation to the arithmetic product (multiplication) and the similarity of the OR operation to the arithmetic sum (addition). The gate structure of the product-of-sums expression consists of a group of OR gates for the sum terms (except for a single literal), followed by an AND gate. **This standard type of expression results in a two-level structure of gates.**

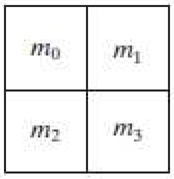
Gate-Level Minimization

Gate-level minimization is the design task of finding an optimal gate-level implementation of the Boolean functions describing a digital circuit. The map method presented here provides a simple, straightforward procedure for minimizing Boolean functions. This method may be regarded as a pictorial form of a truth table. The map method is also known as the *Karnaugh map* or *K-map*. A K-map is a diagram made up of squares, with each square representing one minterm of the function that is to be minimized.

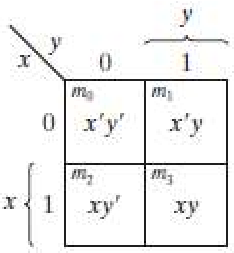
Two-Variable K-Map

There are four minterms for two variables; hence, the map consists of four squares, one for each minterm.. The 0 and 1 marked in each row and column designate the values of variables. Variable *x* appears primed in row 0 and unprimed in row 1. Similarly, *y* appears primed in column 0 and unprimed in column 1. If we mark the squares whose minterms belong to a given function, the two-variable map becomes another useful way to represent any one of the 16 Boolean functions of two variables. As an example, the function *xy* is shown in Figure (a). Since *xy* is equal to *m*3, a 1 is placed inside the square that belongs to *m*3. Similarly, the function *x* + *y* is represented in the map of Figure (b) by three squares marked with 1's. These squares are found from the minterms of the function:

$$m_1 + m_2 + m_3 = x'y + xy' + xy = x + y$$

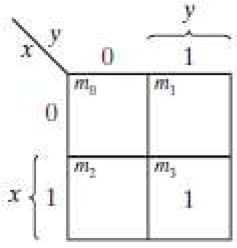


(a)

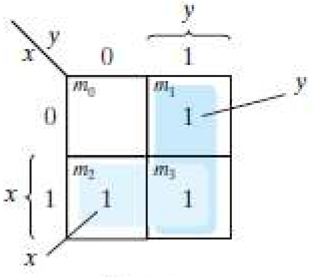


(b)

The three squares could also have been determined from the intersection of variable *x* in the second row and variable *y* in the second column, which encloses the area belonging to *x* or *y* . In each example, the minterms at which the function is asserted are marked with a 1.



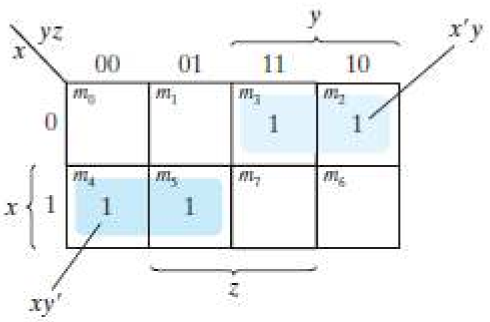
(a) *xy*



(b) *x* + *y*

Three-Variable K-Map

Example: Simplify the Boolean function
$F(x,y,z) = \sum(2,3,4,5)$



The simplified function is: $F(x,y,z) =$ x'y+yx'

Example
For the Boolean function
$F = A'C + A'B + AB'C + BC$

(a) Express this function as a sum of minterms.
(b) Find the minimal sum-of-products expression.

Note that F is a *sum of products*. Three product terms in the expression have two literals and are represented in a three-variable map by two squares each. The two squares corresponding to the first term, $A'C$, are found in Fig-a from the coincidence of A' (first row) and C (two middle columns) to give squares 001 and 011. Note that, in marking 1's in the squares, it is possible to find a 1 already placed there from a preceding term.
This happens with the second term, $A'B$, which has 1's in squares 011 and 010. Square 011 is common with the first term, $A'C$, though, so only one 1 is marked in it. Continuing in this fashion, we determine that the term $AB'C$ belongs in square 101, corresponding to minterm 5, and the term BC has two 1's in squares 011 and 111. The function has a total of five minterms, as indicated by the five 1's in the map of Figure . The minterms are read directly from the map to be 1, 2, 3, 5, and 7. The function can be expressed in sum-of-minterms form as $F(x,y,z) = \sum(1,2,3,5,7)$
The sum-of-products expression, as originally given, has too many terms. It can be simplified, as shown in the map, to an expression with only two terms:

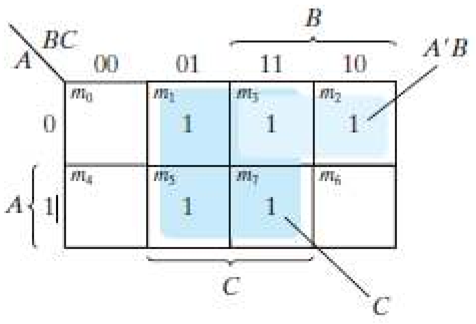
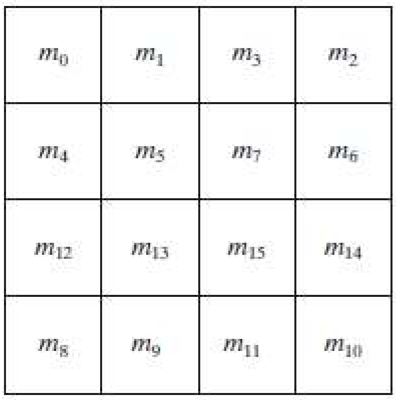


Fig-a

The simplified function is: $F = C + A'B$

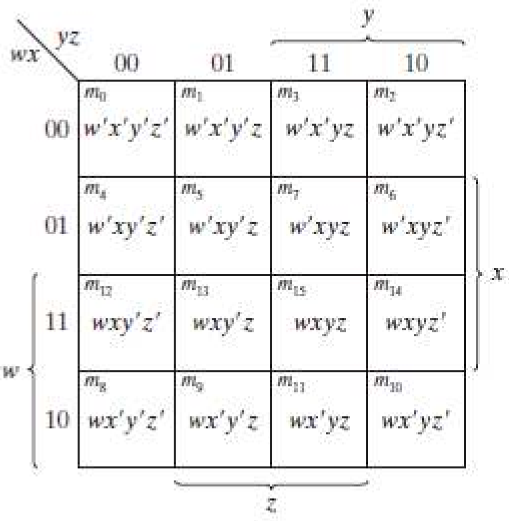
FOUR-VARIABLE K-MAP

Here for four variables we have 16 minterms. So a map of 16 squares is required. Lets say 4 variables are w,x,y,z



m_0 m_1 m_3 m_2
m_4 m_5 m_7 m_6
m_{12} m_{13} m_{15} m_{14}
m_8 m_9 m_{11} m_{10}

(a)

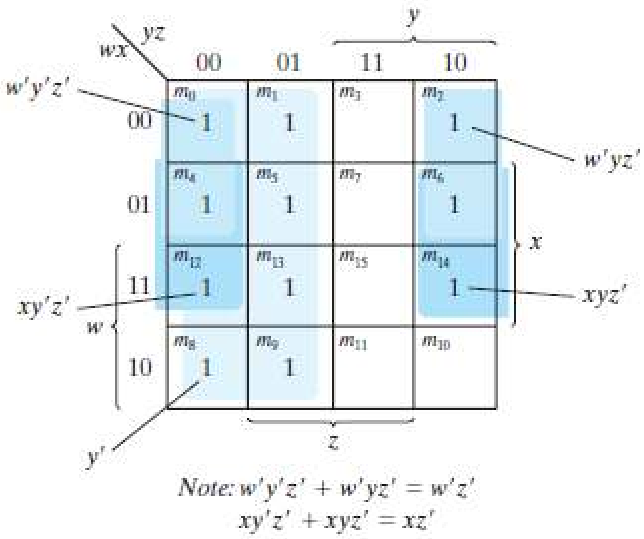


(b)

Example

Simplify the Boolean function: $F\ (x,y,z)= \sum(0,1,2,4,5,6,,9,12,13,14)$

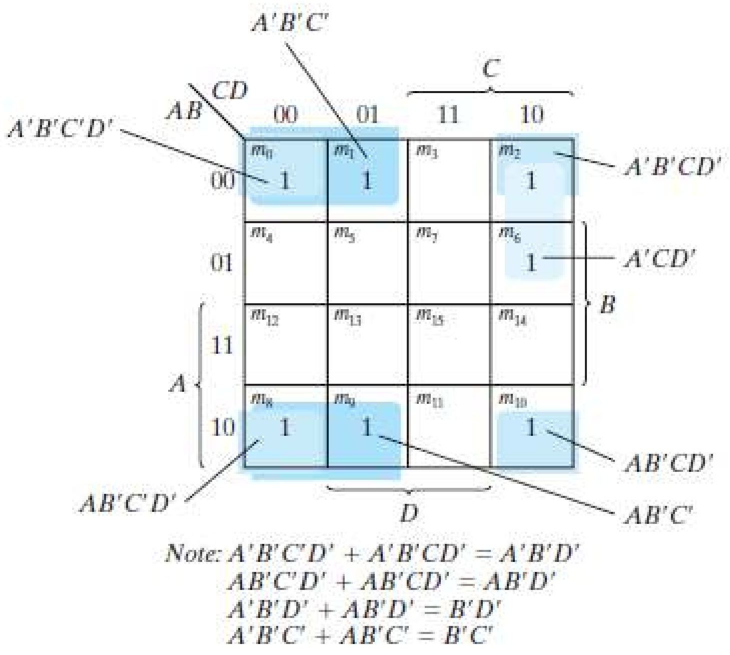
Eight adjacent squares marked with 1's can be combined to form the one literal term z'. The remaining three 1's on the right cannot be combined to give a simplified term; they must be combined as two or four adjacent squares. The larger the number of squares combined, the smaller is the number of literals in the term. In this example, the top two 1's on the right are combined with the top two 1's on the left to give the term $w'z'$. Note that it is permissible to use the same square more than once. We are now left with a square marked by 1 in the third row and fourth column (square 1110). Instead of taking this square alone (which will give a term with four literals), we combine it with squares already used to form an area of four adjacent squares. These squares make up the two middle rows and the two end columns, giving the term xz'.



The simplified function is: $F = y' + w'z' + xz'$

Example

Simplify the Boolean function: *F* = A’B’C’+ B’CD’ + A’BCD’ + AB’C’



The simplified function is: *F* = B’D’ + A’CD’ + B’C’

Five-Variable Map

A five-variable map needs 32 squares and a six-variable map needs 64 squares. It can be explained in the class.

PRODUCT-OF-SUMS SIMPLIFICATION

By using K-map the simplified function is in SOP format.
So if we want to get Final answer in POS format, we need to simplified for the F’ and at the end take complement of F’ to get F in POS form. It can be easily explained in the following example

Example

Simplify the following Boolean function into (a) sum-of-products form and (b) product-of-sums form:

$F(A,B,C,D) = \sum(0,1,2,5,8,9,10)$

The 1’s marked in the map of Figure represent all the minterms of the function. The squares marked with 0’s represent the minterms not included in *F* and therefore denote the complement of *F*. Combining the squares with 1’s gives the simplified function in sum-of-products form:

(a) $F = B'D' + B'C' + A'C'D$

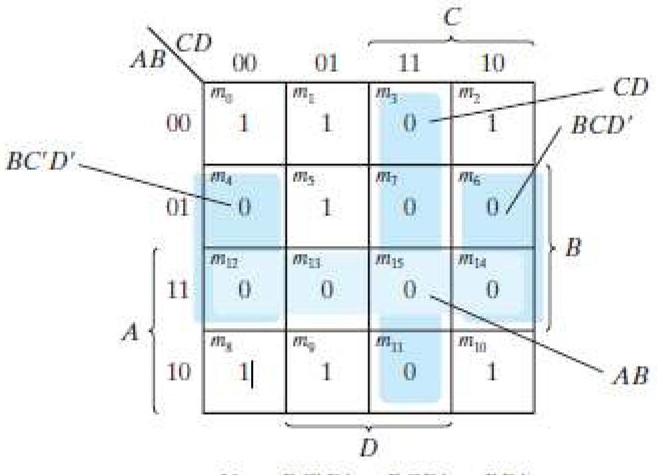
If the squares marked with 0's are combined, as shown in the diagram, we obtain the simplified complemented function:

$F\_ = AB + CD + BD'$

Applying DeMorgan's theorem, we obtain the simplified function in productof-sums form:

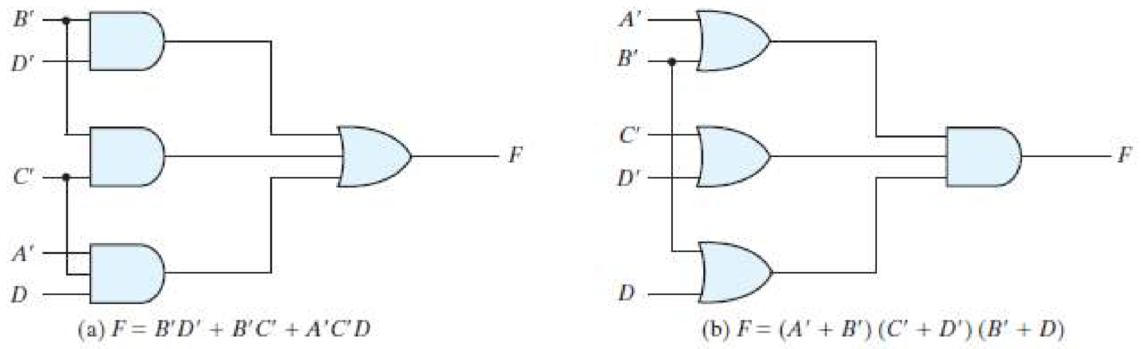
(b) $F = (A' + B')(C' + D')(B' + D)$

The gate-level implementation of the simplified expressions obtained in Example 3.7 is shown in Fig. 3.13 . The sum-of-products expression is implemented in (a) with a group of AND gates, one for each AND term. The outputs of the AND gates are connected to the inputs of a single OR gate. The same function is implemented in (b) in its product-of-sums



Note: $BC'D' + BCD' = BD'$

***F* (*A, B, C, D*)** $= \sum(\mathbf{0,1,2,5,8,9,10}) = B'D' + B'C' + A'C'\boldsymbol{D} =$
$(A' + B')(C' + D')(B' + \boldsymbol{D})$



(a) $F = B'D' + B'C' + A'C'D$ (b) $F = (A' + B')(C' + D')(B' + D)$

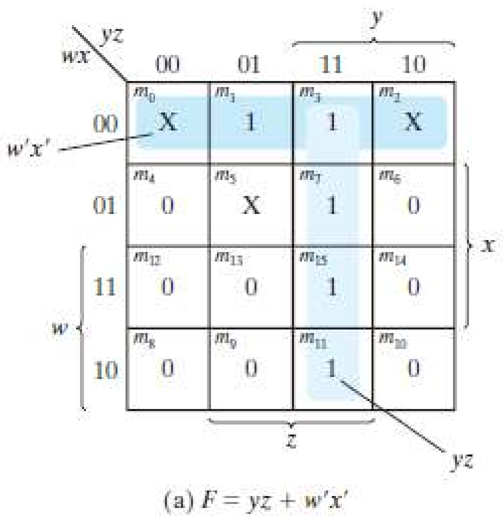
DON'T-CARE CONDITIONS

The logical sum of the minterms associated with a Boolean function specifies the conditions under which the function is equal to 1. The function is equal to 0 for the rest of the minterms. In practice, in some applications the function is not specified for certain combinations of the variables. As an example, the four-bit binary code for the decimal digits has six combinations that are not used and consequently are considered to be unspecified. Functions that have unspecified outputs for some input combinations are called *incompletely*

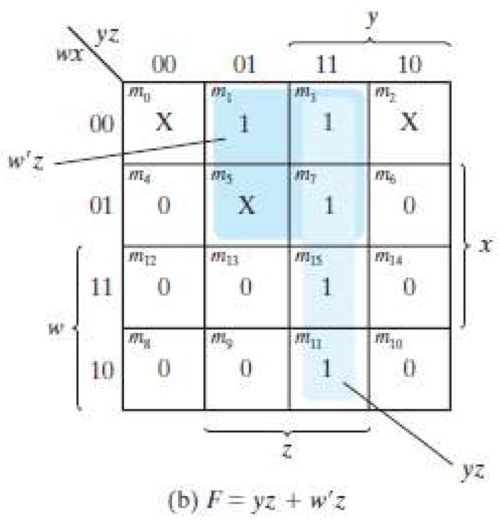
specified functions . these unspecified minterms are don't-care conditions and can be used on a map to provide further simplification of the Boolean expression.
A don't-care minterm is a combination of variables whose logical value is not specified. To distinguish the don't-care condition from 1's and 0's, an X is used. Thus, an X inside a square in the map indicates that we don't care whether the value of 0 or 1 is assigned to *F* for the particular minterm.
In choosing adjacent squares to simplify the function in a map, the don't-care minterms may be assumed to be either 0 or 1. When simplifying the function, we can choose to include each don't-care minterm with either the 1's or the 0's, depending on which combination gives the simplest expression.

Example
Simplify the Boolean function: F (w,x,y,z)= $\sum(1,3,7,11,15)$
which has the don't-care conditions: d (w,x,y,z)= $\sum(0, 2, 5)$

The minterms of *F* are the variable combinations that make the function equal to 1. The minterms of *d* are the don't-care minterms that may be assigned either 0 or 1(marked by X's) and the remaining squares are filled with 0's. To get the simplified expression in sum-of-products form, we must include all five 1's in the map, but we may or may not include any of the X's, depending on the way the function is simplified. The term *yz* covers the four minterms in the third column. The remaining minterm, *m*1, can be combined



(a) $F = yz + w'x'$



(b) $F = yz + w'z$

with minterm *m*3 to give the three-literal term *w'x'z*. However, by including one or two adjacent X's we can combine four adjacent squares to give a two-literal term. In Figure (a), don't-care minterms 0 and 2 are included with the 1's, resulting in the simplified function

$$F = yz + w'x'$$

In Figure (b), don't-care minterm 5 is included with the 1's, and the simplified function is now $F = yz + w'z$
Either one of the preceding two expressions satisfies the conditions stated for this example.

More examples will be done in the Class.

NAND AND NOR IMPLEMENTATION

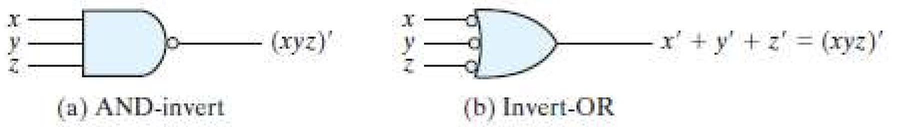
Digital circuits are frequently constructed with NAND or NOR gates rather than with AND and OR gates. NAND and NOR gates are easier to fabricate with electronic components and are the basic gates used in all IC digital logic families. Rules and procedures have been developed for the conversion from Boolean functions given in terms of AND, OR, and NOT into equivalent NAND and NOR logic diagrams.

NAND Circuits

The NAND gate is said to be a *universal* gate because any logic circuit can be implemented with it. To show that any Boolean function can be implemented with NAND gates, we need only show that the logical operations of AND, OR, and complement can be obtained with NAND gates alone. This is indeed shown in Figure.

A convenient way to implement a Boolean function with NAND gates is to obtain the simplified Boolean function in terms of Boolean operators and then convert the function to NAND logic. The conversion of an algebraic expression from AND, OR, and complement to NAND can be done by simple circuit manipulation techniques that change AND–OR diagrams to NAND diagrams.

Two equivalent graphic symbols for the NAND gate are shown in Figure . The AND-invert symbol has been defined previously and consists



The general procedure for converting a multilevel AND–OR diagram into an all-NAND diagram using mixed notation is as follows:

1. Convert all AND gates to NAND gates with AND-invert graphic symbols.
2. Convert all OR gates to NAND gates with invert-OR graphic symbols.
3. Check all the bubbles in the diagram. For every bubble that is not compensated by another small circle along the same line, insert an inverter (a one-input NAND gate) or complement the input literal.

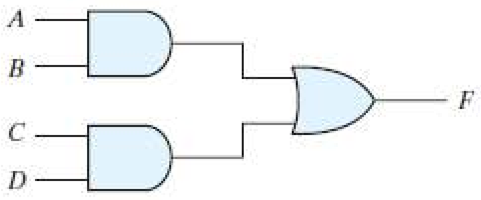
Two-Level Implementation

Example: The implementation of Boolean functions with NAND gates requires that the functions be in sum-of-products form**.**

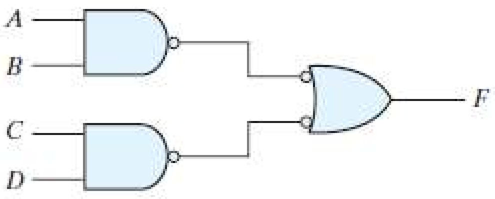
$$F = AB + CD$$

The function is implemented in Figure(a) with AND and OR gates.

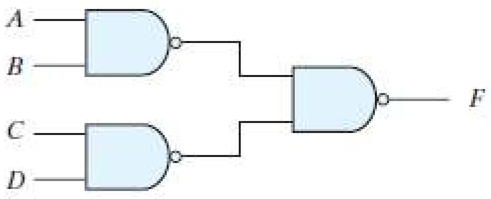
In Figure(b), the AND gates are replaced by NAND gates and the OR gate is replaced by a NAND gate with an OR-invert graphic symbol. Remember that a bubble denotes complementation and two bubbles along the same line represent double complementation, so both can be removed. Removing the bubbles on the gates of (b) produces the circuit of (a).



(a)



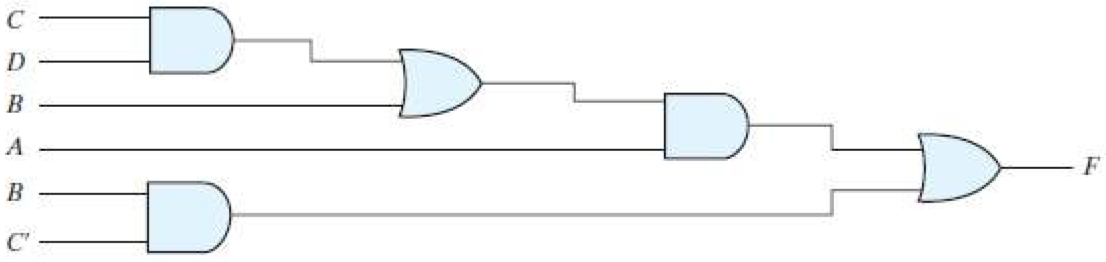
(b)



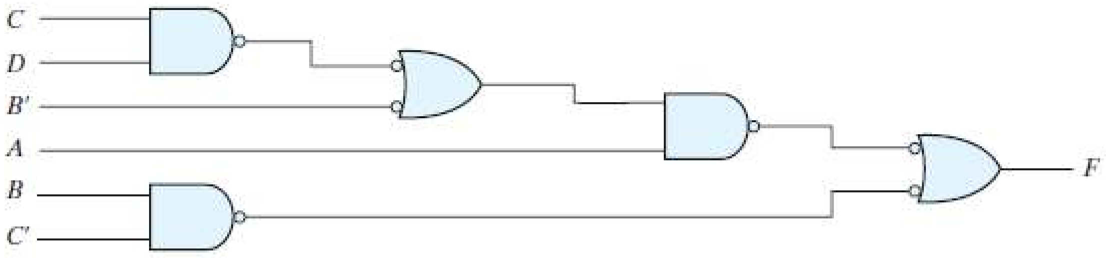
(c)

Multilevel NAND Circuits

Implement : $F = A(CD + B) + BC'$



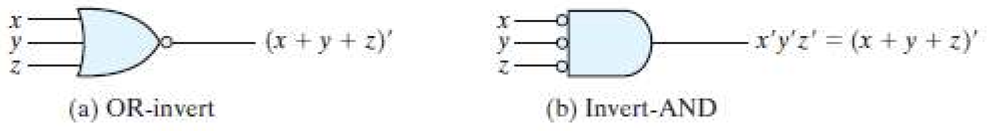
(a) AND–OR gates



(b) NAND gates

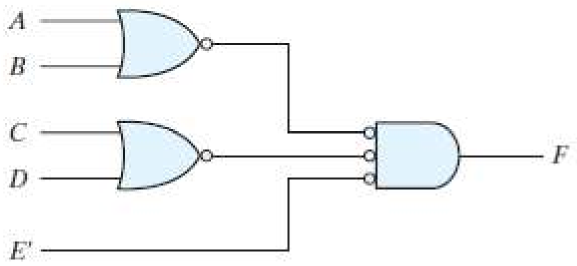
NOR Implementation

Two equivalent graphic symbols for the NOR gate are shown in Figure .

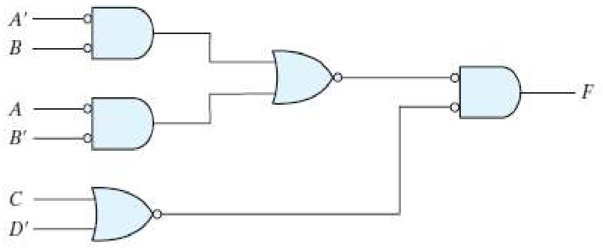


NOR implementation of a function expressed as a product of sums. Then the OR-AND pattern can be easily converted to NOR gates.For example:

$F = (A + B)(C + D)E$



Ex.The Boolean function for this circuit is:F = (AB_ + A_B)(C + D_)



THER TWO-LEVEL IMPLEMENTATIONS

The eight *nondegenerate* forms are as follows:

AND–OR	OR–AND
NAND–NAND	NOR–NOR
NOR–OR	NAND–AND
OR–NAND	AND–NOR

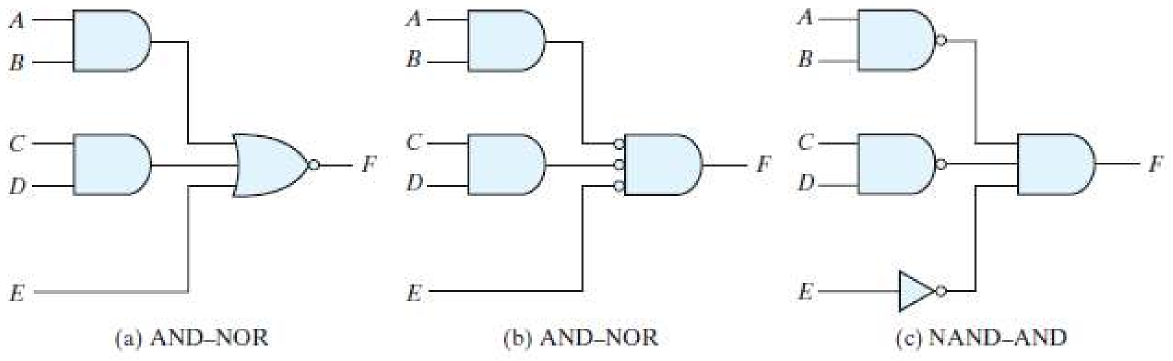
AND–OR–INVERT Implementation

The two forms, NAND–AND and AND–NOR, are equivalent and can be treated together. Both perform the AND–OR–INVERT function, as shown in Figure given below . The AND–NOR form resembles the AND–OR form, but with an inversion done by the bubble in the output of the NOR gate. It implements the function

$F = (AB + CD + E)$_

By using the alternative graphic symbol for the NOR gate, we obtain the diagram of Figure (b). Note that the single variable *E* is *not* complemented, because the only

change made is in the graphic symbol of the NOR gate. Now we move the bubble from the input terminal of the second-level gate to the output terminals of the first-level gates. An inverter is needed for the single variable in order to compensate for the bubble. Alternatively, the inverter can be removed, provided that input E is complemented. The circuit of Fig. 3.27 (c) is a NAND–AND form and was shown in Fig. 3.26 to implement the AND–OR–INVERT function.
An AND–OR implementation requires an expression in sum-of-products form. The AND–OR–INVERT implementation is similar, except for the inversion. Therefore, if the *complement* of the function is simplified into sum-of-products form (by combining the 0's in the map), it will be possible to implement $F\_$ with the AND–OR part of the function. When $F\_$ passes through the always present output inversion (the INVERT part), it will

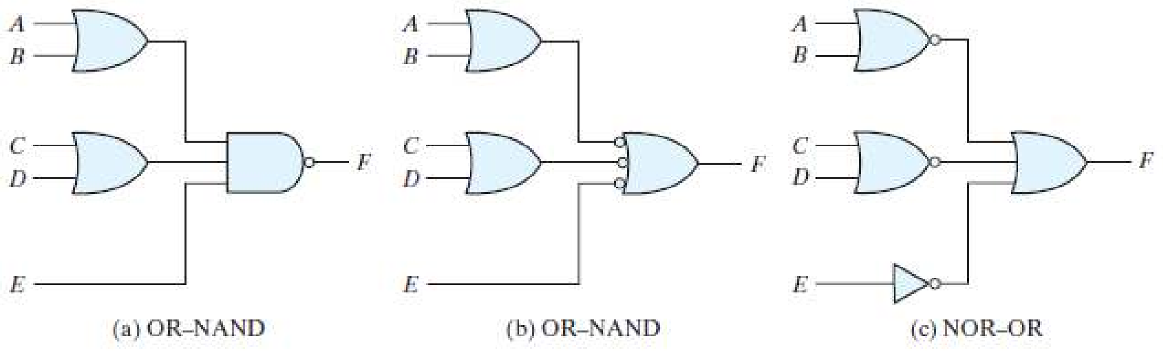


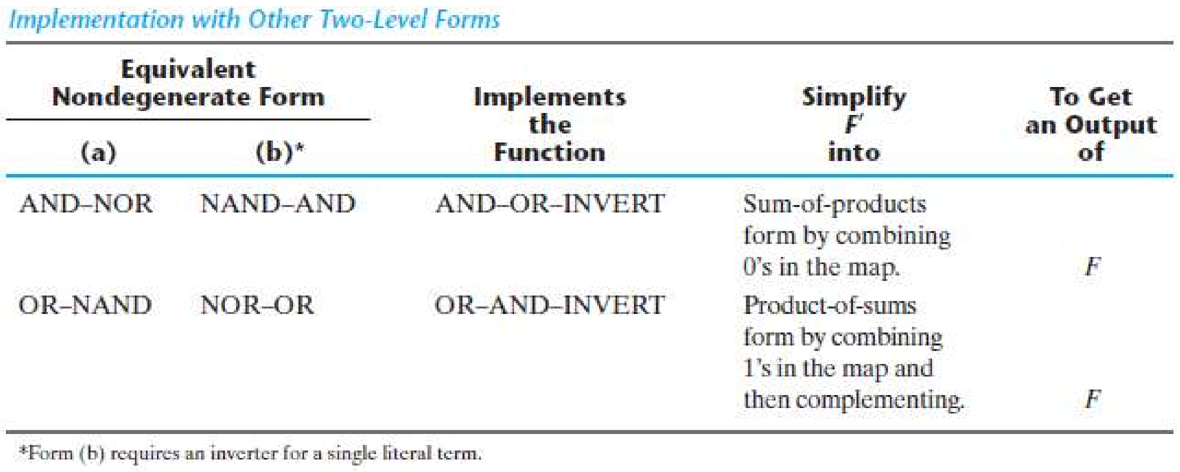
OR–AND–INVERT Implementation

The OR–NAND and NOR–OR forms perform the OR–AND–INVERT function, as shown in Fig. 3.28 . The OR–NAND form resembles the OR–AND form, except for the inversion done by the bubble in the NAND gate. It implements the function

$$F = 3(A + B)(C + D)E'$$

By using the alternative graphic symbol for the NAND gate, we obtain the diagram of Figure (b). The circuit in Figure(c) is obtained by moving the small circles from the inputs of the second-level gate to the outputs of the first-level gates. The circuit of Fig. (c) is a NOR–OR form and was shown in Fig. 3.26 to implement the OR–AND–INVERT function.
The OR–AND–INVERT implementation requires an expression in product-of-sums form. If the complement of the function is simplified into that form, we can implement F' with the OR–AND part of the function. When $F\_$ passes through the INVERT part, we obtain the complement of F', or F , in the output.




Implementation with Other Two-Level Forms

Equivalent Nondegenerate Form (a)	Equivalent Nondegenerate Form (b)*	Implements the Function	Simplify F' into	To Get an Output of
AND–NOR	NAND–AND	AND–OR–INVERT	Sum-of-products form by combining 0's in the map.	F
OR–NAND	NOR–OR	OR–AND–INVERT	Product-of-sums form by combining 1's in the map and then complementing.	F

*Form (b) requires an inverter for a single literal term.

EXCLUSIVE-OR FUNCTION

The exclusive-OR (XOR), denoted by the symbol⊕, is a logical operation that performs the following Boolean operation:

$$x \oplus y = xy' + x'y$$

The exclusive-OR is equal to 1 if only x is equal to 1 or if only y is equal to 1 (i.e., x and y differ in value), but not when both are equal to 1 or when both are equal to 0. The exclusive-NOR, also known as equivalence performs the following Boolean operation:

$(x \oplus y)' = xy + x'y'$

The exclusive-NOR is equal to 1 if both x and y are equal to 1 or if both are equal to 0.
The exclusive-NOR can be shown to be the complement of the exclusive-OR by means of a truth table or by algebraic manipulation:

$(x \oplus y)' = (xy' + x'y) = (x' + y)(x + y') = xy + x'y'$

The following identities apply to the exclusive-OR operation:

$x \oplus 0 = x$

$x \oplus 1 = x'$

$x \oplus x = 0$

$x \oplus x' = 1$

$x \oplus y' = x' \oplus y = (x \oplus y)'$

Any of these identities can be proven with a truth table or by replacing the ⊕ operation by its equivalent Boolean expression. Also, it can be shown that the exclusive-OR operation is both commutative and associative; that is,

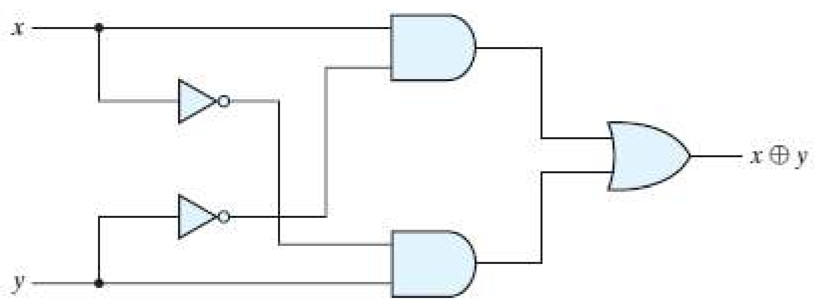
$$A \oplus B = B \oplus A$$

and

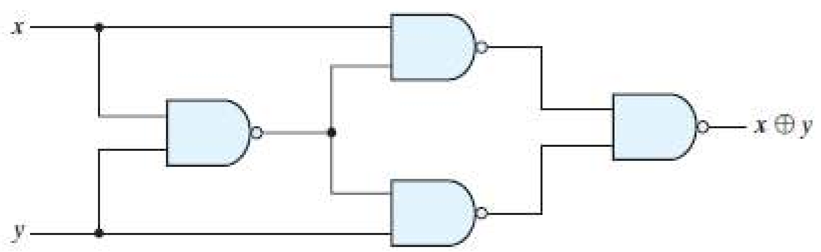
$$(A \oplus B) \oplus C = A \oplus (B \oplus C) = A \oplus B \oplus C$$

This means that the two inputs to an exclusive-OR gate can be interchanged without affecting the operation. It also means that we can evaluate a three-variable exclusive-OR operation in any order, and for this reason, three or more variables can be expressed without parentheses. This would imply the possibility of using exclusive-OR gates with three or more inputs. However, multiple-input exclusive-OR gates are difficult to fabricate with hardware. In fact, even a two-input function is usually constructed with other types of gates. A two-input exclusive-OR function is constructed with conventional gates using two inverters, two AND gates, and an OR gate, as shown in Fig (a). Figure (b) shows the implementation of the exclusive-OR with four NAND gates. The first NAND gate performs the operation $(xy)' = (x' + y')$. The other two-level NAND circuit

produces the sum of products of its inputs:
$(x' + y') + (x' + y') = xy' + x'y = x \oplus y$

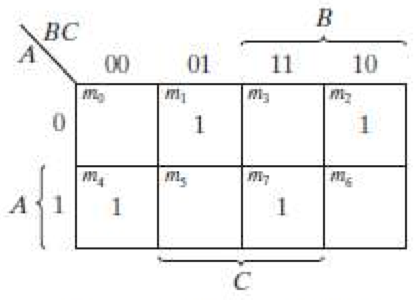


(a) Exclusive-OR with AND–OR–NOT gates

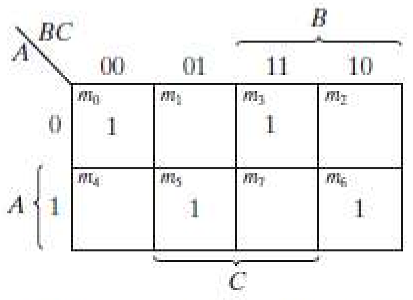


(b) Exclusive-OR with NAND gates

Odd Function
The exclusive-OR operation with three or more variables can be converted into an ordinary Boolean function by replacing the ⊕ symbol with its equivalent Boolean expression. In particular, the three-variable case can be converted to a Boolean expression
as follows:
$A \oplus B \oplus C = (AB' + A'B)C' + (AB + A'B')C$
$= AB'C' + A'BC' + ABC + A'B'C$
$= \sum(1, 2, 4, 7)$
The Boolean expression clearly indicates that the three-variable exclusive-OR function is equal to 1 if only one variable is equal to 1 or if all three variables are equal to 1. Contrary to the two-variable case, in which only one variable must be equal to 1, in the case of three or more variables the requirement is that an odd number of variables be equal to 1. As a consequence, the multiple-variable exclusive-OR operation is defined as an *odd function.*



(a) Odd function $F = A \oplus B \oplus C$



(b) Even function $F = (A \oplus B \oplus C)'$

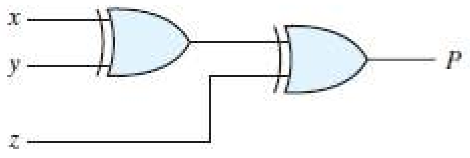
Parity Generation and Checking

Exclusive-OR functions are very useful in systems requiring error detection and correction codes. A parity bit is used for the purpose of detecting errors during the transmission of inary information. A parity bit is an extra bit included with a binary message to make the number of 1's either odd or even. The message, including the parity bit, is transmitted and then checked at the receiving end for errors. An error is detected if the checked parity does not correspond with the one transmitted. The circuit that generates the parity bit in the transmitter is called a *parity generator.* The circuit that checks the parity in the receiver is called a *parity checker.*

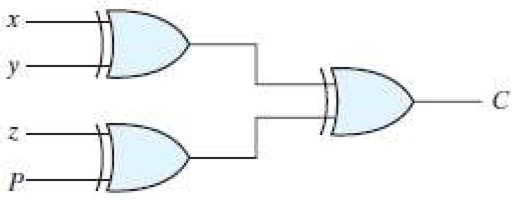
As an example, consider a three-bit message to be transmitted together with an even-parity bit. Table shows the truth table for the parity generator. The three bits *x*, *y*, and *z* constitute the message and are the inputs to the circuit. The parity bit *P* is the output. For even parity, the bit *P* must be generated to make the total number of 1's (including *P*) even. From the truth table, we see that *P* constitutes an

Even-Parity-Generator Truth Table

Three-Bit Message			**Parity Bit**
x	***y***	***z***	***P***
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



(a) 3-bit even parity generator



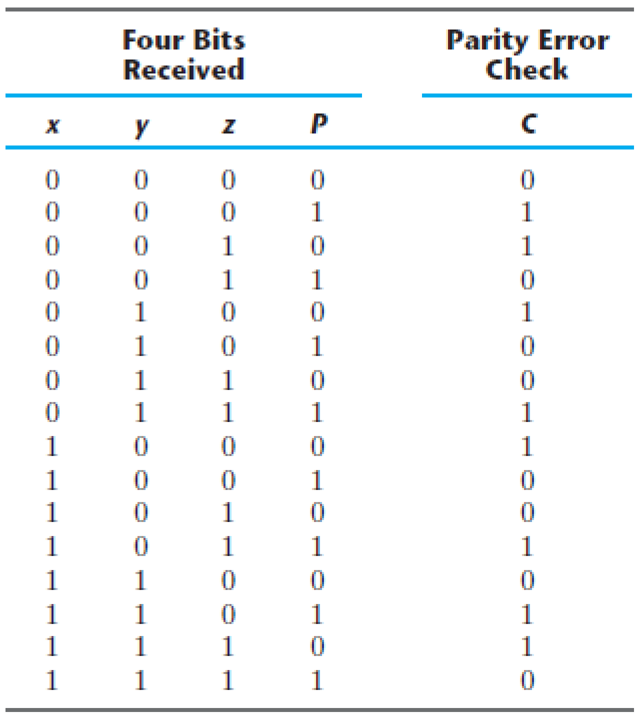
(b) 4-bit even parity checker

odd function because it is equal to 1 for those minterms whose numerical values have an odd number of 1's. Therefore, *P* can be expressed as a three-variable exclusive-OR function:

$$P = x \oplus y \oplus z$$

The logic diagram for the parity generator can be drawn using XOR gates. The three bits in the message, together with the parity bit, are transmitted to their destination, where they are applied to a parity-checker circuit to check for possible errors in the transmission. Since the information was transmitted with even parity, the four bits received must have an even number of 1's. An error occurs during the transmission if the four bits received have an odd number of 1's, indicating that one bit has changed in value during transmission. The output of the parity checker, denoted by *C* , will be equal to 1 if an error occurs—that is, if the four bits received have an odd number of 1's. The truth table for the even-parity checker is given below. From it, we see that the function C consists of the eight minterms with binary numerical values having an odd number of 1's. The table corresponds to the map of Fig.(a), which

Even-Parity-Checker Truth Table



Four Bits Received				Parity Error Check
x	y	z	P	C
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

represents an odd function. The parity checker can be implemented with exclusive-OR gates:

$$C = x \oplus y \oplus z \oplus P$$

The logic diagram of the parity checker can be drawn using XOR gates. It is obvious from the foregoing example that parity generation and checking circuits always have an output function that includes half of the minterms whose numerical values have either an odd or even number of 1's. As a consequence, they can be implemented with exclusive-OR gates. A function with an even number of 1's is the complement of an odd function. It is implemented with exclusive-OR gates, except that the gate associated with the output must be an exclusive-NOR to provide the required complementation.