MODULE-1

Introduction to Digital Design

Syllabus

Introduction to Digital Design: Binary Logic, Basic Theorems And Properties Of Boolean Algebra, Boolean Functions, Digital Logic Gates, Introduction, The Map Method, Four-Variable Map, Don't-Care Conditions, NAND and NOR Implementation, Other Hardware Description Language – Verilog Model of a simple circuit. Text book 1: 1.9, 2.4, 2.5, 2.8, 3.1, 3.2, 3.3, 3.5, 3.6, 3.9

Text book 1: **M. Morris Mano** & Michael D. Ciletti, **Digital Design With an Introduction to Verilog Design**, 5e, Pearson Education.

BINARY LOGIC

Definition of Binary Logic

- **Binary logic** consists of **binary variables** and a **set of logical operations.** The variables are designated by letters of the alphabet, such as A, B, C, x, y, z, etc., with each variable having **two distinct possible values: 1 and 0**.

There are **three basic** logical operations: **AND, OR, and NOT**. Each operation produces a binary result, denoted by z.

1. **AND**: This operation is represented by a **dot** or by the **absence** of an **operator**.

Example, x.y = z or xy = z is read "x AND y is equal to z."

If z = 1 if and only if x = 1 and y = 1; otherwise z = 0. The result of the operation x .y is z.

x, y, and z are binary variables and can be equal either to 1 or 0

2. **OR:** This operation is represented by a **plus** sign. For example, x + y = z is read "x OR y is equal to z," meaning that **z = 1 if x = 1 or if y = 1 or if both x = 1 and y = 1. If both x = 0 and y = 0, then z = 0.**

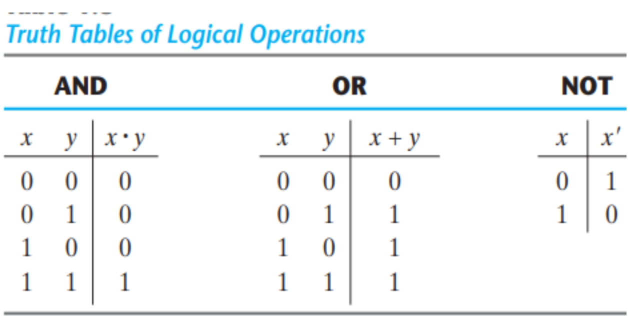
3. **NOT**: This operation is represented by a prime (or by an **overbar**).

For example, $x' = z$ (or $\bar{x} = z$) is read "not x is equal to z,"**.**

If x = 1, then z = 0, and if x = 0, then z = 1. The **NOT operation** is also referred to as the **complement operation**, since it changes a 1 to 0 and a 0 to 1.

Truth table

- A **truth table** is a table of all possible combinations of the variables, showing the relation between the values that the variables may take and the result of the operation.
- The truth tables for the operations AND and OR with variables x and y are obtained by listing all possible values that the variables may have when combined in pairs.



Truth Tables of Logical Operations

AND			OR			NOT	
x	y	$x \cdot y$	x	y	$x + y$	x	x'
0	0	0	0	0	0	0	1
0	1	0	0	1	1	1	0
1	0	0	1	0	1		
1	1	1	1	1	1		

Gates:

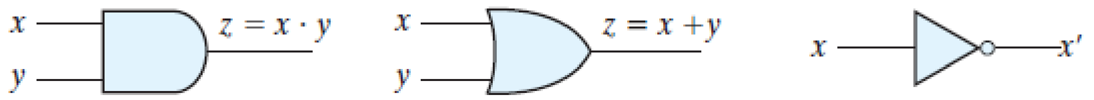
"A digital circuit having one or more input signals but only one output signal is called a gate".

Logic Gates:

"The gates which perform logical operation is called logic gates".

- It take binary input and gives binary outputs.
- The output of the logic gates can be understood using truth table, which contains inputs, outputs of logic circuits.

Logic gates are electronic circuits that operate on one or more input signals to produce an output signal.



(a) Two-input AND gate (b) Two-input OR gate (c) NOT gate or inverter

Fig: Symbols for digital logic circuits

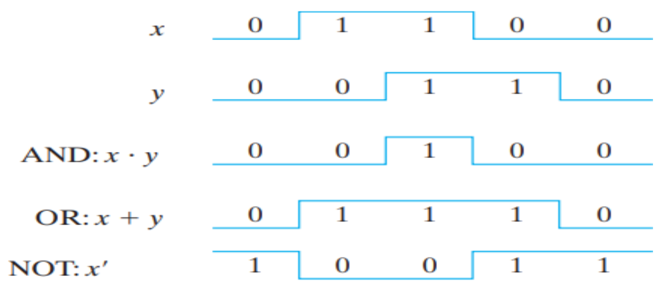
The **timing diagrams** illustrate the idealized response of each gate to the four input signal combinations. The **horizontal axis** of the timing diagram represents the **time**, and the **vertical axis** shows the **signal** as it changes between the two possible voltage levels.

The low level represents logic 0, the high level logic 1.

The **AND gate** responds with a logic 1 output signal when both input signals are logic 1.

The **OR gate** responds with a logic 1 output signal if any input signal is logic 1.

The **NOT gate** is commonly referred to as an inverter. The output signal inverts the logic sense of the input signal.



Input-Output Signals for gates Timing Diagram

Number System

Number system is a basis for counting varies items. Modern computers communicate and operate with binary numbers which use only the digits 0 &1. Basic number system used by humans is Decimal number system.

For Ex: Let us consider decimal number 18. This number is represented in binary as 10010.

We observe that binary number system take more digits to represent the decimal number. For large numbers we have to deal with very large binary strings. So this fact gave rise to three new number systems.

i) Octal number systems

ii) Hexa Decimal number system

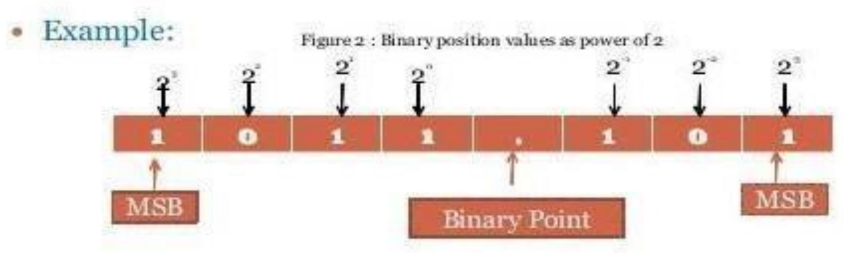
iii) Binary Coded Decimal number(BCD) system To

define any number system we have to specify

- Base of the number system such as 2,8,10 or 16.
- The base decides the total number of digits available in that number system.
- First digit in the number system is always zero and last digit in the number system is always base-1.

Binary number system:

The binary number has a radix of 2. As r = 2, only two digits are needed, and these are 0 and 1. In binary system weight is expressed as power of 2.



The left most bit, which has the greatest weight is called the Most Significant Bit (MSB). And the right most bit which has the least weight is called Least Significant Bit (LSB).

or Ex: $\mathbf{1001.01_2 = [\,(1) \times 2^3\,] + [\,(0) \times 2^2\,] + [\,(0) \times 2^1\,] + [\,(1) \times 2^0\,] + [\,(0) \times 2^{-1}\,] + [\,(1) \times 2^2\,]}$

$\mathbf{1001.01_2 = [\,1 \times 8\,] + [\,0 \times 4\,] + [\,0 \times 2\,] + [\,1 \times 1\,] + [\,0 \times 0.5\,] + [\,1 \times 0.25\,]}$

$\mathbf{1001.01_2 = 9.25_{10}}$

Decimal Number system

The decimal system has ten symbols: 0,1,2,3,4,5,6,7,8,9. In other words, it has a base of 10.

Octal Number System

Digital systems operate only on binary numbers. Since binary numbers are often very long, two shorthand notations, octal and hexadecimal, are used for representing large binary numbers. Octal systems use a base or radix of 8. It uses first eight digits of decimal number system. Thus it has digits from 0 to 7.

Hexa Decimal Number System

The hexadecimal numbering system has a base of 16. There are 16 symbols. The decimal digits 0 to 9 are used as the first ten digits as in the decimal system, followed by the letters A, B, C, D, E and F, which represent the values 10, 11,12,13,14 and 15 respectively.

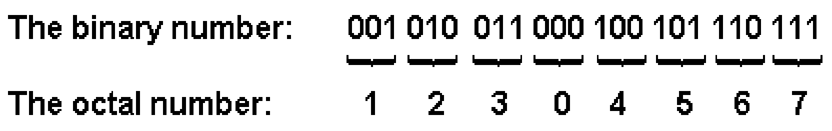
Decimal	Binary	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8

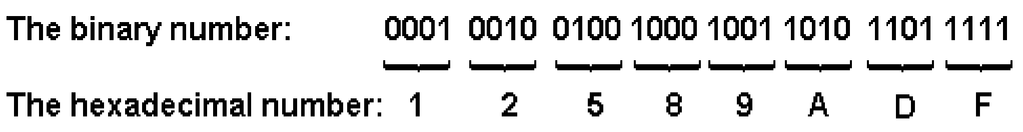
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Number Base conversions

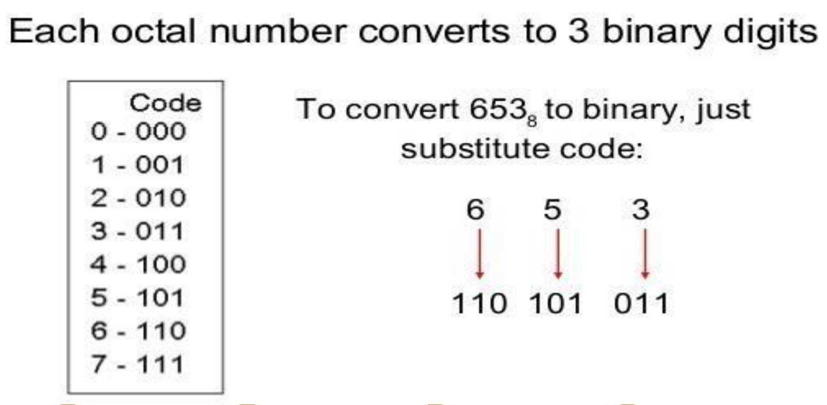
The human beings use decimal number system while computer uses binary number system. Therefore it is necessary to convert decimal number system into its equivalent binary.

i) Binary to octal number conversion
ii) Binary to hexa decimal number conversion


The binary number: 001 010 011 000 100 101 110 111
The octal number: 1 2 3 0 4 5 6 7


The binary number: 0001 0010 0100 1000 1001 1010 1101 1111
The hexadecimal number: 1 2 5 8 9 A D F

iii) Octal to binary Conversion


Each octal number converts to 3 binary digits
Code
0 - 000
1 - 001
2 - 010
3 - 011
4 - 100
5 - 101
6 - 110
7 - 111
To convert 653₈ to binary, just substitute code:
6 5 3
110 101 011

iv) Hexa to binary conversion

v) Octal to Decimal conversion

Ex: convert 4057.06_8 to octal

$=4x8^3+0x8^2+5x8^1+7x8^0+0x8^{-1}+6x8^{-2}$

=2048+0+40+7+0+0.0937

2095.0937_{10}

Decimal to Octal Conversion Ex: convert

378.93_{10} to octal

378_{10} to octal: Successive division:

```
      8 |  378
          |_____
      8 |47 ---   2
          |_________
   8 |5  ---    7        ↑
     |________
      0   ---    5
```

$=572_8$

0.93_{10} to octal :

0.93x8=7.44

0.44x8=3.52 ↓

0.53x8=4.16

0.16x8=1.28

$=0.7341_8$

$378.93_{10}=572.7341_8$

vi) Hexadecimal to Decimal Conversion

Ex: $5C7_{16}$ to decimal

$=(5\text{x}16^2)+(C\ \text{x}16^1)+ (7\ \text{x}16^0)$

=1280+192+7

$=147_{10}$

vii) Decimal to Hexadecimal Conversion

Ex: 2598.67510

```
 1 6  |2598
      |_______
16 16|2         -6
      |_______
     10          -2
```

= A26 (16)

0.67510=0.675x16 -- 10.8

=0.800x16 -- 12.8 ↓

=0.800x16 -- 12.8

=0.800x16 -- 12.8

=$0.ACCC_{16}$

2598.675_{10} = $A26.ACCC_{16}$

viii) Octal to hexadecimal conversion:

The simplest way is to first convert the given octal no. to binary & then the binary no. to hexadecimal.

Ex: 756.603_8

7	5	6	.	6	0	3
111	101	110	.	110	000	011
0001	1110	1110	.	1100	0001	1000
1	E	E	.	C	1	8

ix) Hexadecimal to octal conversion:

First convert the given hexadecimal no. to binary & then the binary no. to octal.

Ex: B9F.AE16

B	9	F	.	A	E		
1011	1001	1111	.	1010	1110		
101	110	011	111	.	101	011	100
5	6	3	7	.	5	3	4

=5637.534

Complements:

In digital computers to simplify the subtraction operation & for logical manipulation complements are used. There are two types of complements used in each radix system.

i) The radix complement or r's complement

ii) The diminished radix complement or (r-1)'s complement

Representation of signed no.s binary arithmetic in computers:

- Two ways of rep signed no.s
  1. Sign Magnitude form
  2. Complemented form
- Two complimented forms
  1. 1‘s compliment form
  2. 2‘s compliment form

Advantage of performing subtraction by the compliment method is reduction in the hardware.(instead of addition & subtraction only adding ckt‘s are needed.)
i. e, subtraction is also performed by adders only.

Instead of subtracting one no. from other the compliment of the subtrahend is added to minuend. In sign magnitude form, an additional bit called the sign bit is placed in front of the no. If the sign bit is 0, the no. is +ve, If it is a 1, the no is _ve.

Ex:

0	1	0	1	0	0	1

↓

Sign bit =+41 magnitude

↑

1	1	0	1	0	0	1

= -41

Note: manipulation is necessary to add a +ve no to a –ve no

Representation of signed no.s using 2's or 1's complement method:

If the no. is +ve, the magnitude is rep in its true binary form & a sign bit 0 is placed in front of the MSB.I f the no is _ve , the magnitude is rep in its 2‘s or 1‘s compliment form &a sign bit 1 is placed in front of the MSB.

Ex:

Given no.	Sign mag form	2‘s comp form	1‘s comp form
01101	+13	+13	+13
010111	+23	+23	+23
10111	-7	-7	-8
1101010	-42	-22	-21

Special case in 2’s comp representation:

Whenever a signed no. has a 1 in the sign bit & all 0‘s for the magnitude bits, the decimal equivalent is -2^n , where n is the no of bits in the magnitude .
Ex: 1000= -8 & 10000=-16

Characteristics of 2’s compliment no.s:

Properties:

1. There is one unique zero
2. 2‘s comp of 0 is 0
3. The leftmost bit can‘t be used to express a quantity . it is a 0 no. is +ve.
4. For an n-bit word which includes the sign bit there are $(2^{n-1}-1)$ +ve integers, 2^{n-1} –ve integers & one 0 , for a total of 2^n uniquestates.
5. Significant information is containd in the 1‘s of the +ve no.s & 0‘s of the _ve no.s
6. A _ve no. may be converted into a +ve no. by finding its 2‘s comp.

Signed binary numbers:

Decimal	Sign 2‘s comp form	Sign 1‘s comp form	Sign mag form
+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	0011	0011	0011
+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	--	--

Methods of obtaining 2's comp of a no:

- In 3 ways

1. By obtaining the 1‘s comp of the given no. (by changing all 0‘s to 1‘s & 1‘s to 0‘s) & then adding 1.
2. By subtracting the given n bit no N from 2^n
3. Starting at the LSB , copying down each bit upto & including the first 1 bit encountered , and complimenting the remaining bits.

Ex: Express -45 in 8 bit 2‘s comp form

+45 in 8 bit form is 00101101

I method:

1‘s comp of 00101101 & the add 1

```
00101101
11010010
       +1
——– – – – – – – – – –

11010011            is 2‘s comp form
```

II method:

Subtract the given no. N from

2^n 2^n = 100000000

Subtract 45= -00101101

III method:

Original no: 00101101

Copy up to First 1 bit 1

Compliment remaining :

1101001

____

Bits 11010011

-73.75 in 12 bit

2‘compform I method

01001001.1100

10110110.0011

```
                    +1
            ______________

            10110110.0100 is 2‘s
      II method:
      2^8 = 100000000.0000
Sub 73.75=-01001001.1100
            _____________

            10110110.0100 is 2‘s comp
      III method :
      Orginalno            :
                           01001001.1100
      Copy up to 1‘st bit       100
      Comp the remaining bits:      10110110.0
                                   ______________

                                   10110110.0100
```

2’s compliment Arithmetic:

- The 2‘s comp system is used to rep –ve no.s using modulus arithmetic . The word length of a computer is fixed. i.e, if a 4 bit no. is added to another 4 bit no . the result will be only of 4 bits. Carry if any , from the fourth bit will overflow called the Modulus arithmetic.
  Ex:1100+1111=1011
- In the 2‘s compl subtraction, add the 2‘s comp of the subtrahend to the minuend . If there is a carry out , ignore it , look at the sign bit I,e, MSB of the sum term .If the MSB is a 0, the result is positive.& it is in true binary form. If the MSB is a ` (carry in or no carry at all) the result is negative.& is in its 2‘s comp form. Take its 2‘s comp to find its magnitude in binary.

Ex: Subtract 14 from 46 using 8 bit 2‘s comp arithmetic:

```
+14    = 00001110
-14    = 11110010          2‘s comp

+46    = 00101110
-14    =+11110010          2‘s comp form of -14
-32     (1)0010000         ignore
         0                 carry
```

Ignore carry , The MSB is 0 . so the result is +ve. & is in normal binary form. So the result is +00100000=+32.

EX: Add -75 to +26 using 8 bit 2‘s comp arithmetic

```
+7      = 01001011
5       =10110101          2‘s comp
-75

+2      = 00011010
6       =+10110101         2‘s comp form of -75
-75     ___________
-49       11001111          No carry
```

No carry , MSB is a 1, result is _ve & is in 2‘s comp. The magnitude is 2‘s comp of 11001111. i.e, 00110001 = 49. so result is -49

Ex: add -45.75 to +87.5 using 12 bit arithmetic

```
+87.5 = 01010111.1000
-45.75=+11010010.0100
____    ______________

-41.75    (1)00101001.1100 ignore
 carry MSB is 0, result is +ve.  =+41.75
```

1’s compliment of n number:

- It is obtained by simply complimenting each bit of the no,.& also , 1‘s comp of a no, is subtracting each bit of the no. form 1.This complemented value rep the – ve of the original no. One of the difficulties of using 1‘s comp is its rep o f zero. Both 00000000 & its 1‘s comp 11111111 rep zero.
- The 00000000 called +ve zero& 11111111 called –ve

zero. Ex: -99 & -77.25 in 8 bit 1‘s comp

```
+99        =    01100011
-99        =    10011100

+77.25 =       01001101.0100
-77.25  =      10110010.1011
```

1’s compliment arithmetic:

In 1‘s comp subtraction, add the 1‘s comp of the subtrahend to the minuend. If there is a carryout , bring the carry around & add it to the LSB called the **end around carry.** Look at the sign bit (MSB) . If this is a 0, the result is +ve & is in true binary. If the MSB is a 1 (carry or no carry), the result is –ve & is in its is comp form .Take its 1‘s comp to get the magnitude inn binary.

Ex: Subtract 14 from 25 using 8 bit 1‘s EX: ADD -25 to +14

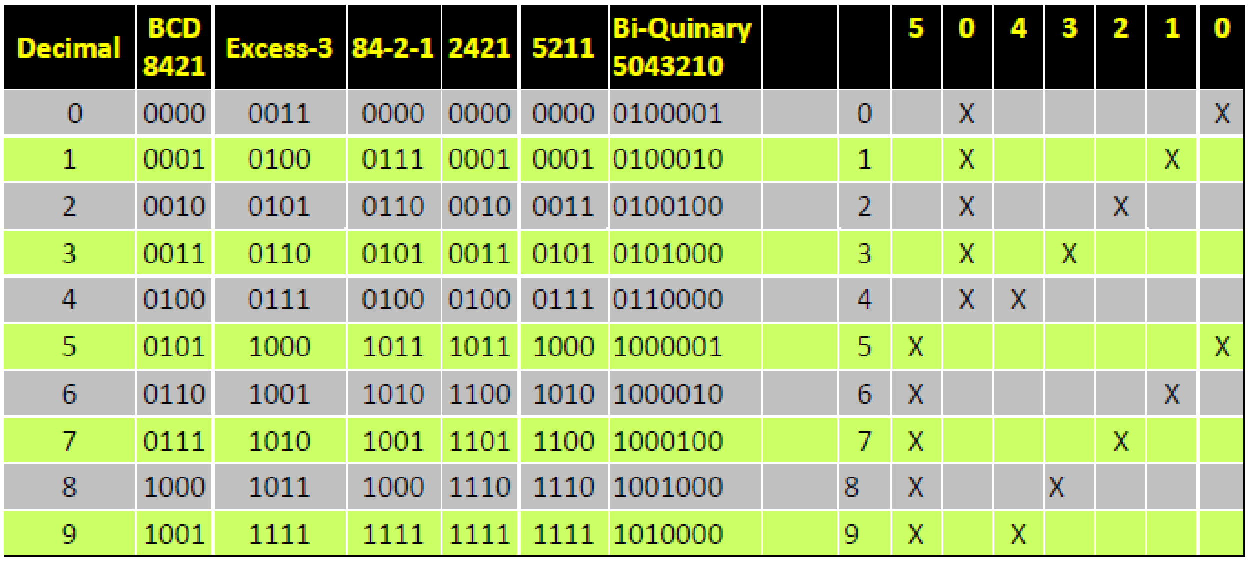
```
25      =     00011001              +14    = 00001110
-45     =     11110001              -25    =+11100110
-----         ----------------------------------------
+11           (1)00001010           -11       11110100

        +1
              ----------------------No carry   MSB =1
              00001011                     result=-ve=-
                                           11₁₀
  MSB is a 0 so result is +ve (binary )

        =+11₁₀
```

Binary codes

Binary codes are codes which are represented in binary system with modification from the original ones.

- Weighted Binary codes
- Non Weighted Codes

Weighted binary codes are those which obey the positional weighting principles, each position of the number represents a specific weight. The binary counting sequence is an example.



Decimal	BCD 8421	Excess-3	84-2-1	2421	5211	Bi-Quinary 5043210			5	0	4	3	2	1	0
0	0000	0011	0000	0000	0000	0100001		0		X					X
1	0001	0100	0111	0001	0001	0100010		1		X				X	
2	0010	0101	0110	0010	0011	0100100		2		X			X		
3	0011	0110	0101	0011	0101	0101000		3		X		X			
4	0100	0111	0100	0100	0111	0110000		4		X	X				
5	0101	1000	1011	1011	1000	1000001		5	X						X
6	0110	1001	1010	1100	1010	1000010		6	X					X	
7	0111	1010	1001	1101	1100	1000100		7	X				X		
8	1000	1011	1000	1110	1110	1001000		8	X			X			
9	1001	1111	1111	1111	1111	1010000		9	X		X				

Non weighted codes

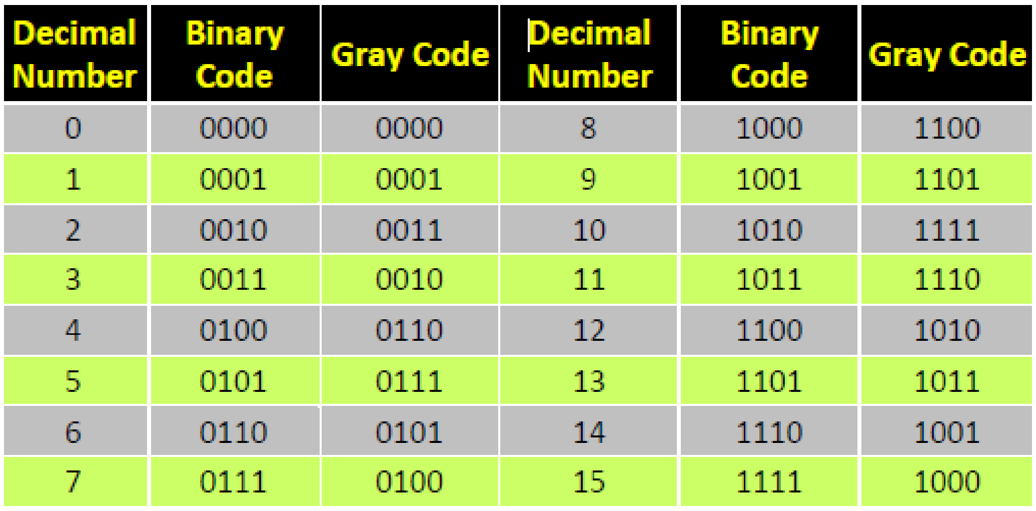
Non weighted codes are codes that are not positionally weighted. That is, each position within the binary number is not assigned a fixed value. Ex: Excess-3 code

Excess-3 Code

Excess-3 is a non weighted code used to express decimal numbers. The code derives its name from the fact that each binary code is the corresponding 8421 code plus 0011(3).

Gray Code

The gray code belongs to a class of codes called minimum change codes, in which only one bit in the code changes when moving from one code to the next. The Gray code is non-weighted code, as the position of bit does not contain any weight. The gray code is a reflective digital code which has the special property that any two subsequent numbers codes differ by only one bit. This is also called a unit- distance code. In digital Gray code has got a special place.



Decimal Number	Binary Code	Gray Code	Decimal Number	Binary Code	Gray Code
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

Gray Code MSB is binary code MSB.

- Gray Code MSB-1 is the XOR of binary code MSB and MSB-1.

MSB-2 bit of gray code is XOR of MSB-1 and MSB-2 bit of binary code. MSB-N bit of gray code is XOR of MSB-N-1 and MSB-N bit of binary code.

8421 BCD code (Natural BCD code):

Each decimal digit 0 through 9 is coded by a 4 bit binary no. called natural binary codes. Because of the 8,4,2,1 weights attached to it. It is a weighted code & also sequential . it is useful for mathematical operations. The advantage of this code is its case of conversion to & from decimal. It is less efficient than the pure binary, it require more bits.

Ex: 14→1110 in binary

But as 0001 0100 in 8421 ode.

The disadvantage of the BCD code is that , arithmetic operations are more complex than they are in pure binary . There are 6 illegal combinations 1010,1011,1100,1101,1110,1111 in these codes, they are not part of the 8421 BCD code system . The disadvantage of 8421 code is, the rules of binary addition 8421 no, but only to the individual 4 bit groups.

BCD Addition:

It is individually adding the corresponding digits of the decimal no,s expressed in 4 bit binary groups starting from the LSD . If there is no carry & the sum term is not an illegal code , no correction is needed .If there is a carry out of one group to the next group or if the sum term is an illegal code then 6_{10}(0100) is added to the sum term of that group & the resulting carry is added to the next group.

Ex: Perform decimal additions in 8421 code

(a)25+13

In BCD 25= 0010 0101

```
In BCD      +13  =+0001 001
                        1
               ___   ___________

            38      0011 1000
```

No carry , no illegal code . This is the corrected sum

(b). 679.6 + 536.8

```
679.6     =     0110          0111         1001   .0110 in BCD
+536.8  =       +0101         0011         0010    .1000 in BCD
___ _ _         _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
1216.4          1011          1010         0110    . 1110        illegal codes
                +0110      +  0011         +011    . + 0110      add 0110 to
                                              0                  each
            (1)0001     (1)000       (1)0101    . (1)0100      propagate carry
                        0
           /            /            /              /
             +1           +1           +1             +1
           ______________________________________________
           0001         0010         0001          0110  .    0100

            1            2            1            6     .     4
```

BCD Subtraction:

Performed by subtracting the digits of each 4 bit group of the subtrahend the digits from the corresponding 4- bit group of the minuend in binary starting from the LSD . if there is no borrow from the next group , then 6_{10}(0110)is subtracted from the difference term of this group.

(a)38-15

```
In BCD       38=  0011     1000
In BCD      -15 = -0001    0101
               ___   ___________

            23      0010   0011
```

No borrow, so correct difference.

.(b) 206.7-147.8

```
206.7      =    0010   0000   0110   .      0111               in BCD
-147.8 =        -0001  0100   0111   .      0110               in BCD
____ _ _        ____ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
58.9            0000   1011   1110   .      1111          borrows are
                                                          present
```

```
-0110  -0110 .      -0110        subtract
                                 0110
               ____________________________

               0101   1000  .    1001
```

BCD Subtraction using 9's & 10's compliment methods:

Form the 9's & 10's compliment of the decimal subtrahend & encode that no. in the 8421 code . the resulting BCD no.s are then added.

EX: 305.5 – 168.8

```
305.5  =      305.5
-168.8=       +83.1              9's comp of -168.8
              ——– – –
              (1)136.6
                   +1             end around carry
                    136.7              corrected difference
305.5₁₀  =    0011   0000 0101  .      0101
 +831.1₁₀ =  +1000   0011 0001  .    0001           9's comp of    8 in BCD
__- - -      ----------------        1
                                      68.
             +1011    0011  0110      0110     1011 is illegal code

             ___________________________________

     (1)0001     0011    0110  .    0110
                                         +1      End around carry
             ___________________________________

        0001     0011     0110   .    0111
                               = 136.7
```

Excess three(xs-3)code:

It is a non-weighted BCD code .Each binary codeword is the corresponding 8421 codeword plus 0011(3).It is a sequential code & therefore , can be used for arithmetic operations..It is a self-complementing code.s o the subtraction by the method of compliment addition is more direct in xs-3 code than that in 8421 code. The xs-3 code has six invalid states 0000,0010,1101,1110,1111.. It has interesting properties when used in addition & subtraction.

Ex: 267-175

```
267 = 0101 1001 1010
-175=   -0100 1010  1000
```

```
——— – –  – – –     – – –

 0000   1111    0010
 +0011 -0011  +0011
         ______________________

   0011   1100   +0011                   =92₁₀
```

Error – Detecting codes: When binary data is transmitted & processed,it is susceptible to noise that can alter or distort its contents. The 1's may get changed to 0's & 1's .because digital systems must be accurate to the digit, error can pose a problem. Several schemes have been devised to detect the occurrence of a single bit error in a binary word, so that whenever such an error occurs the concerned binary word can be corrected & retransmitted.

Parity: The simplest techniques for detecting errors is that of adding an extra bit known as parity bit to each word being transmitted.Two types of parity**:** Oddparity**,** evenparity forodd parity, the parity bit is set to a ‗0' or a ‗1' at the transmitter such that the total no. of 1 bit in the word including the parity bit is an odd no.For even parity, the parity bit is set to a ‗0' or a ‗1' at the transmitter such that the parity bit is an even no.

Decimal	8421 code	Odd parity	Even parity
0	0000	1	0
1	0001	0	1
2	0010	0	1
3	0011	1	0
4	0100	0	1
5	0100	1	0
6	0110	1	0
7	0111	0	1
8	1000	0	1
9	1001	1	0

When the digit data is received . a parity checking circuit generates an error signal if the total no of 1's is even in an odd parity system or odd in an even parity system. This parity check can always detect a single bit error but cannot detect 2 or more errors with in the same word.Odd parity is used more often than even parity does not detect the situation. Where all 0's are created by a short ckt or some other fault condition.

Ex: Even parity scheme

(a) 10101010 (b) 11110110 (c)10111001

Ans:

(a) No. of 1's in the word is even is 4 so there is no error
(b) No. of 1's in the word is even is 6 so there is no error
(c) No. of 1's in the word is odd is 5 so there is error

Ex: odd parity

(a)10110111 (b) 10011010 (c)11101010

Ans:

(a) No. of 1's in the word is even is 6 so word has error

(b) No. of 1's in the word is even is 4 so word has error

(c) No. of 1's in the word is odd is 5 so there is no error

DIGITAL LOGIC GATES

- Since Boolean functions are expressed in terms of AND, OR, and NOT operations, it is easier to implement a Boolean function with these type of gates.
- Factors to be weighed in considering the construction of other types of logic gates are (1) the feasibility and economy of producing the gate with physical components, (2) the possibility of extending the gate to more than two inputs, (3) the basic properties of the binary operator, such as commutativity and associativity, and (4) the ability of the gate to implement Boolean functions alone or in conjunction with other gates.
- The graphic symbols and truth tables of the eight gates are shown in Fig. 2.5. Each gate has one or two binary input variables, designated by x and y, and one binary output variable, designated by F.
- The **inverter** circuit inverts the logic sense of a binary variable, producing the NOT, or complement, function. The small circle in the output of the graphic symbol of an inverter (referred to as a bubble) designates the logic complement.
- The **NAND function** is the complement of the AND function, as indicated by a graphic symbol that consists of an AND graphic symbol followed by a small circle.
- The **NOR function** is the complement of the OR function and uses an OR graphic symbol followed by a small circle.
- NAND and NOR gates are used extensively as standard logic gates and are in fact far more popular than the AND and OR gates. This is because NAND and NOR gates are easily constructed with transistor circuits and because digital circuits can be easily implemented with them.
- The **exclusive-OR gate** has a graphic symbol similar to that of the OR gate, except for the additional curved line on the input side. The equivalence, or exclusive-NOR, gate is the complement of the exclusive-OR, as indicated by the small circle on the output side of the graphic symbol.

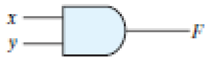
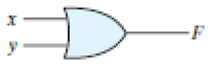
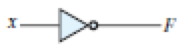
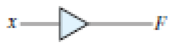
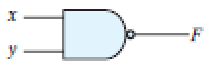
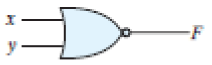
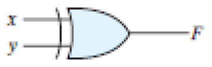
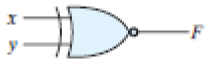
Name	Graphic symbol	Algebraic function	Truth table
AND		$F = x \cdot y$	x y | F 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR		$F = x + y$	x y | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
Inverter		$F = x'$	x | F 0 | 1 1 | 0
Buffer		$F = x$	x | F 0 | 0 1 | 1
NAND		$F = (xy)'$	x y | F 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR		$F = (x + y)'$	x y | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0
Exclusive-OR (XOR)		$F = xy' + x'y$ $= x \oplus y$	x y | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0
Exclusive-NOR or equivalence		$F = xy + x'y'$ $= (x \oplus y)'$	x y | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 1

FIGURE 2.5
Digital logic gates

THE MAP METHOD

- **The map method 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.
- Since any Boolean function can be expressed as a sum of minterms, it follows that a Boolean function is recognized graphically in the map from the area enclosed by those squares whose minterms are included in the function.
- Map represents a visual diagram of all possible ways a function may be expressed in standard form. By recognizing various patterns, the user can derive alternative algebraic expressions for the same function, from which the simplest can be selected.

- **The simplified expressions produced by the map are always in one of the two standard forms: sum of products or product of sums**.
- The simplest algebraic expression is an algebraic expression with a minimum number of terms and with the smallest possible number of literals(variable) in each term. This expression produces a circuit diagram with a minimum number of gates and the minimum number of inputs to each gate.

FOUR-VARIABLE K-MAP

- The map for Boolean functions of four binary variables (w, x, y, z) is shown in Fig. 3.8. In Fig. 3.8(a) are listed the 16 minterms and the squares assigned to each.
- In Fig. 3.8(b), the map is redrawn to show the relationship between the squares and the four variables. The rows and columns are numbered in a Gray code sequence, **with only one digit changing value between two adjacent rows or columns.**
- The minterm corresponding to each square can be obtained from the concatenation of the row number with the column number. For example, the numbers of the third row (11) and the second column (01), when concatenated, give the binary number 1101, the binary equivalent of decimal 13. Thus, the square in the third row and second column represents minterm m13.
- **One square represents one minterm, giving a term with four literals. Two adjacent squares represent a term with three literals.**
- Four adjacent squares represent a term with two literals. Eight adjacent squares represent a term with one literal.

- Sixteen adjacent squares produce a function that is always equal to No other combination of squares can simplify the function.

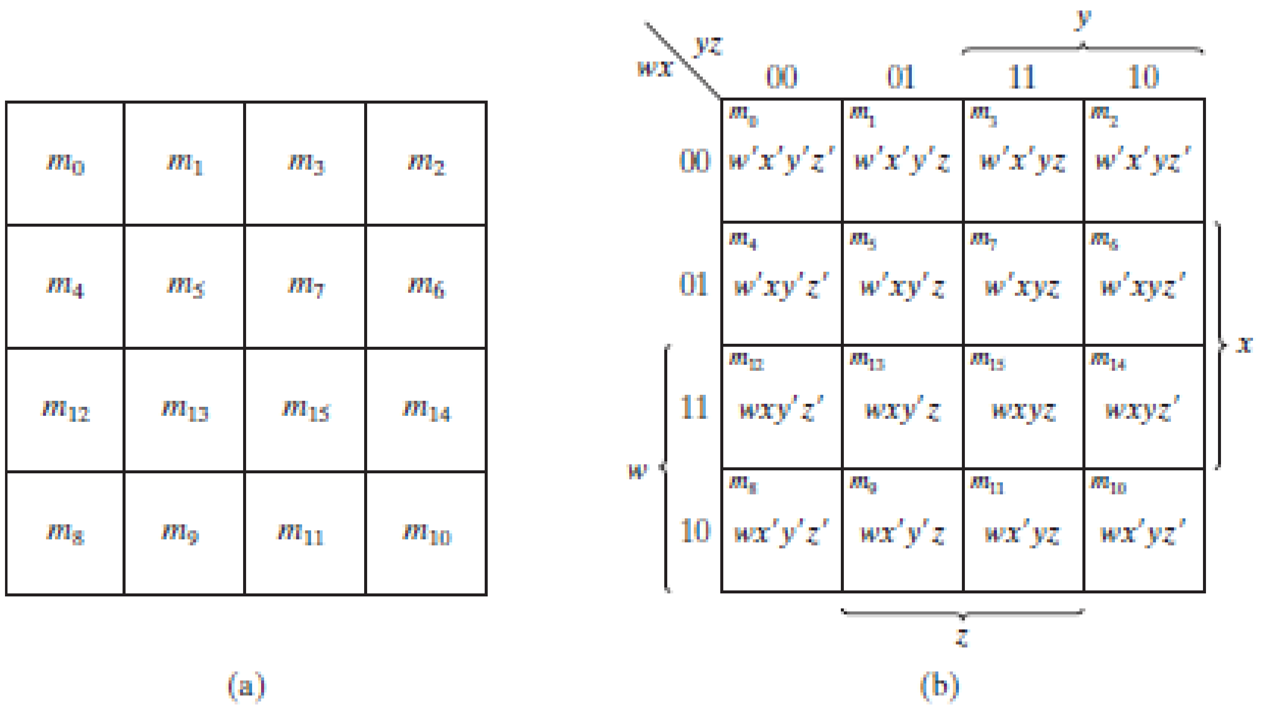


FIGURE 3.8
Four-variable map

1) Simplify the Boolean function

F (w, x, y, z) = Σm (0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)

Since the function has four variables, a four-variable map must be used. The minterms listed in the sum are marked by 1's in the map of Fig. 3.9.

Eight adjacent squares marked with 1's can be combined to form the one literal term y'. 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'.**

These squares make up the two middle rows and the two end columns, giving the term **xz'.** The simplified function is

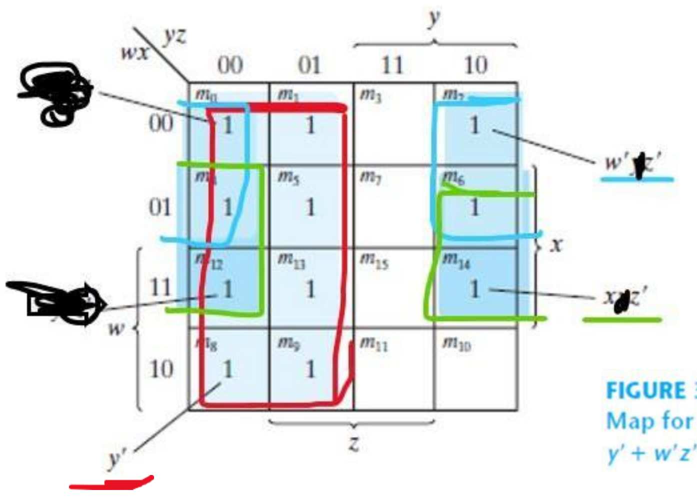


FIGURE 3.9
Map for Example 3.5, $F(w, x, y, z) = \Sigma(0,1, 2, 4, 5, 6, 8, 9, 12, 13, 14) = y' + w'z' + xz'$

2) Simplify the Boolean function

F = A'B'C' + B'CD' + A'BCD' + AB'C'

- F=A’B’C’(D+D’)+B’CD’(A+A’)+A’BCD’+AB’C’(D+D’)
- F=A’B’C’D’+A’B’C’D+AB’CD’+A’B’CD’ +A’BCD’+AB’C’D+AB’C’D’

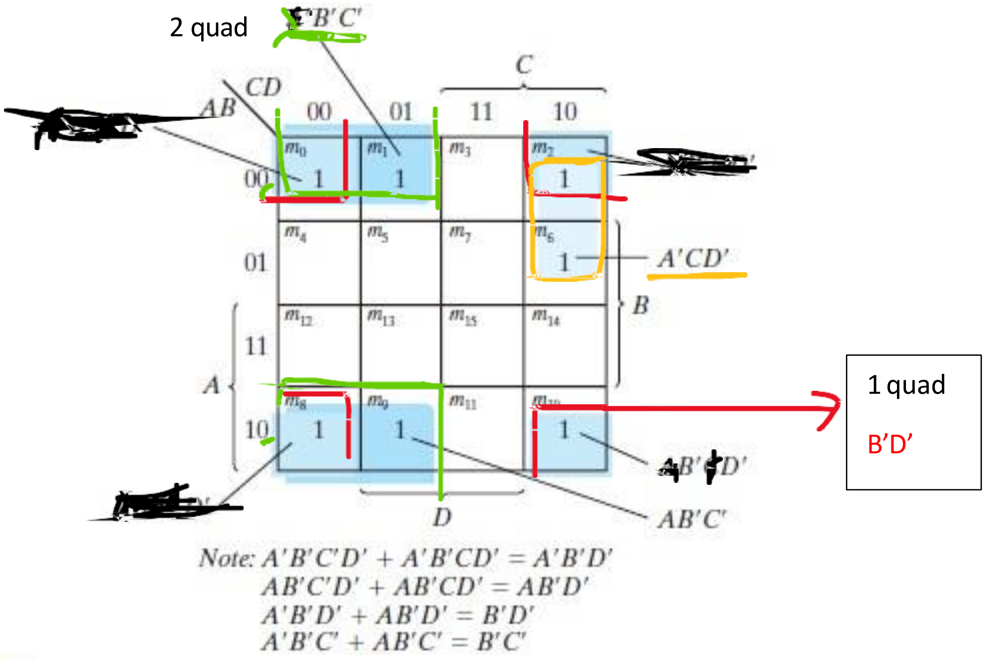
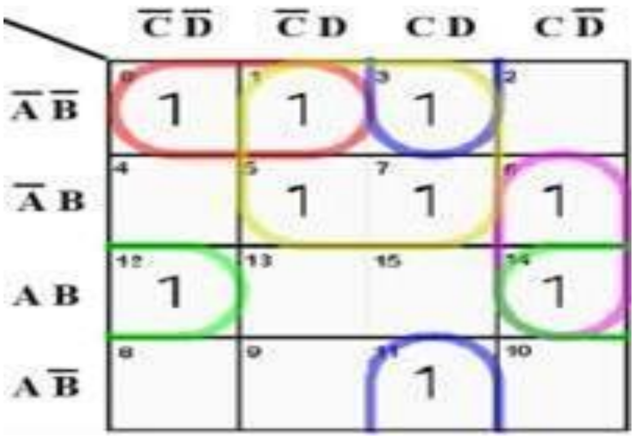


FIGURE 3.10
Map for Example 3.6, $A'B'C' + B'CD' + A'BCD' + AB'C' = B'D' + B'C' + A'CD'$

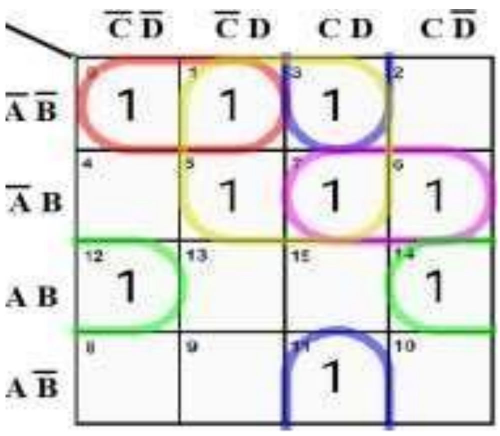
The simplified function is

F = B'D' + B'C' + A'CD'

3) Solve S= F(A,B,C)=Σ m(0, 1, 3, 5, 6, 7, 11, 12, 14) using Kmap and implement using basic gates.

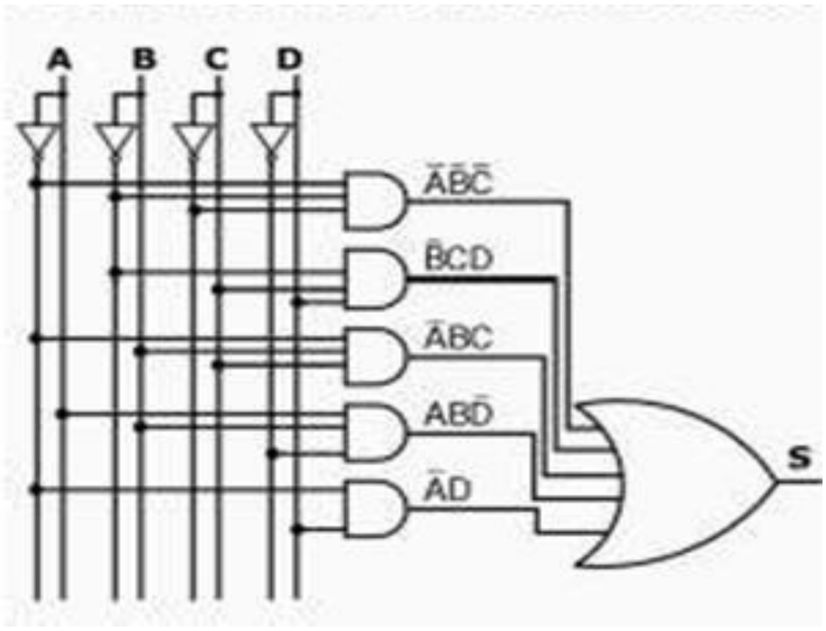


OR

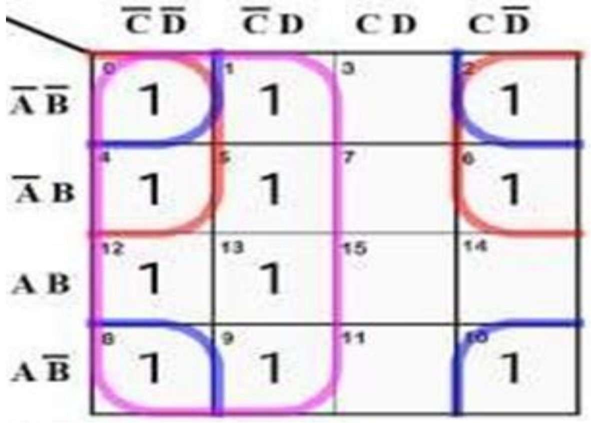


$S = \bar{A}\bar{B}\bar{C} + \bar{B}CD + BC\bar{D} + AB\bar{D} + \bar{A}D$ OR $S = \bar{A}\bar{B}\bar{C} + \bar{B}CD + \bar{A}BC + AB\bar{D} + \bar{A}D$

$S = \bar{A}\bar{B}\bar{C} + \bar{B}CD + \bar{A}BC + AB\bar{D} + \bar{A}D$

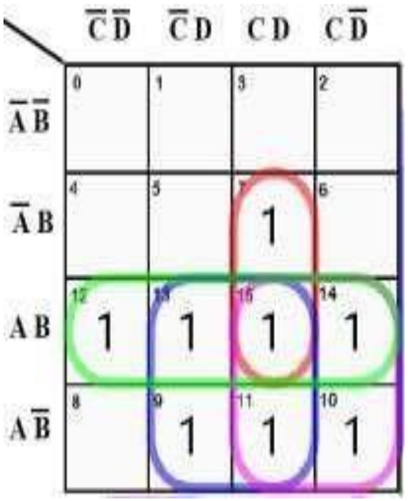


4) Solve S=Σ M(0,1, 2, 4, 5,6, 8,9,10,12,13) using Kmap



$S = \bar{A}\bar{D} + \bar{B}\bar{D} + \bar{C}$

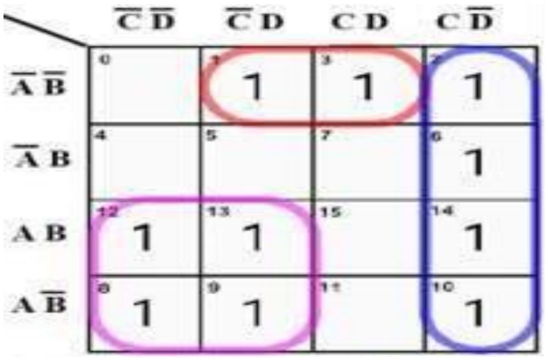
5) **Solve S= F(A,B,C,D)=Σm(7,9,10,11,12,13,14,15) using K map to get minimum SOP expression.**



$S = BCD + AD + AC + AB$

Solve S=F(A,B,C,D)=Σm(1,2,3,6,8,9,10,12,13,14) using K map

to get minimum SOP expression.



$S = \bar{A}\bar{B}D + C\bar{D} + A\bar{C}$

Prime Implicants

In choosing adjacent squares in a map, we must ensure that

(1) all the minterms of the function are covered when we combine the squares.
(2) the number of terms in the expression is minimized, and
(3) there are no redundant terms (i.e., Minterms already covered by other terms).

- **A prime implicant** is a **product term** obtained by combining the **maximum possible number of adjacent squares in the map.**
- The prime implicants of a function can be obtained from the map by combining all possible maximum numbers of squares.
- prime implicants are the building blocks used in the simplification of Boolean functions

Essential Prime Implicant:

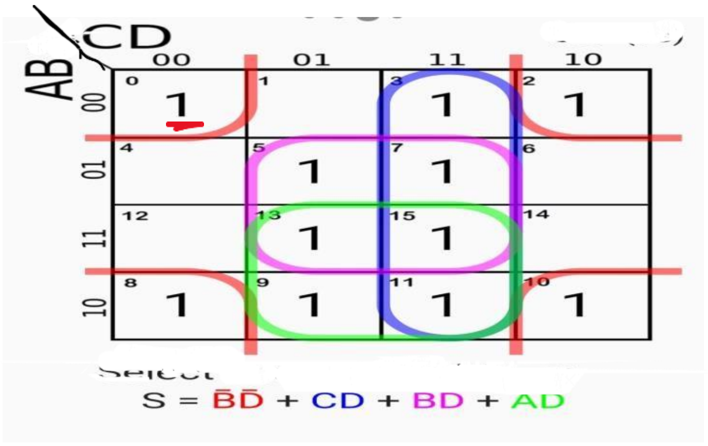
- An essential prime implicant is a prime implicant that covers **at least one minterm that no other prime implicant covers.**

essential prime implicants are a subset of prime implicants that are necessary to cover specific minterms in order to achieve a minimal representation of the Boolean function.

1) **Simplify following four-variable Boolean function:**

F(A, B, C, D) = Σm (0, 2, 3, 5, 7, 8, 9, 10, 11, 13, 15)

The simplified expression is obtained from the logical sum of the two essential prime implicants and any two prime implicants that cover minterms m3, m9, and m11..



Essential Prime Implicants are BD and B’D’

Prime Implicants are B’D’,CD,BD,AD

There are four possible ways that the function can be expressed with four product terms of two literals each:

F = BD + B'D' + CD + AD

= BD + B'D' + CD + AB'

= BD + B'D' + B'C + AD

= BD + B'D' + B'C + AB'

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. This pair of conditions assumes that all the combinations of the values for the variables of the function are valid.
- In some applications the function is not specified for certain combinations of the variables.
- Functions that have unspecified outputs for some input combinations are called incompletely specified functions.
- In most applications, we simply don't care what value is assumed by the function for the unspecified minterms. For this reason, it is customary to call the unspecified minterms of a function don't-care conditions. These don't-care conditions 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. Such a minterm cannot be marked with a 1 in the map, because it would require that the function always be a 1 for such a combination. Likewise, putting a 0 on the square requires the function to be 0. 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 min- terms 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.

Simplify the Boolean function

F (w, x, y, z) = (1, 3, 7, 11, 15)which has the don't-care conditions d

(w, x, y, z) = (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. The map simplification is shown in Fig. 3.15. The minterms of F are marked by 1's, those of d are marked by X's, and the remaining squares are filled with 0's. To get the simplified expres- sion 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, m1, can be combined

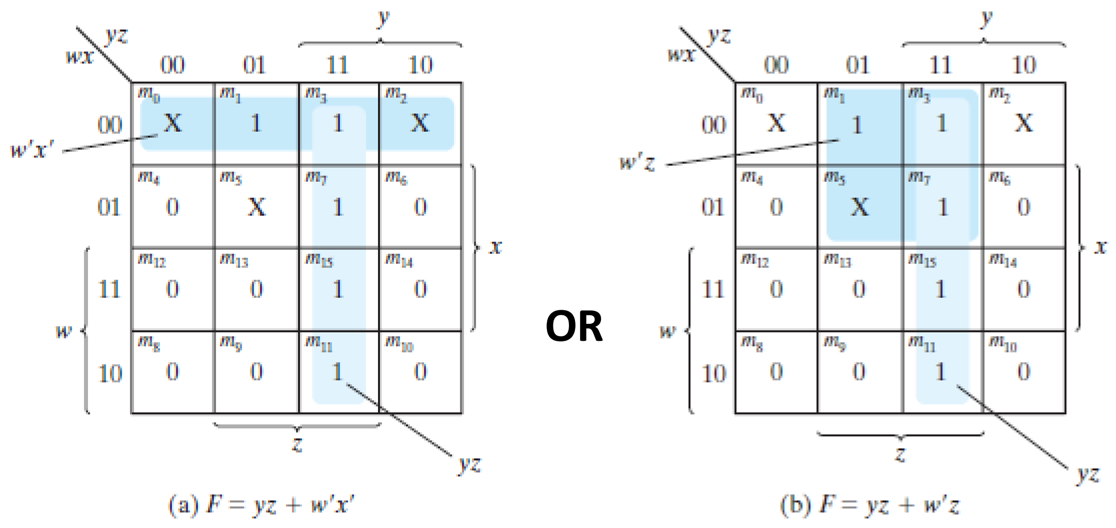


FIGURE 3.15
Example with don't-care conditions

with minterm m3 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 Fig. 3.15(a), don't-care minterms 0 and 2 are included with the 1's, resulting in the simplified function

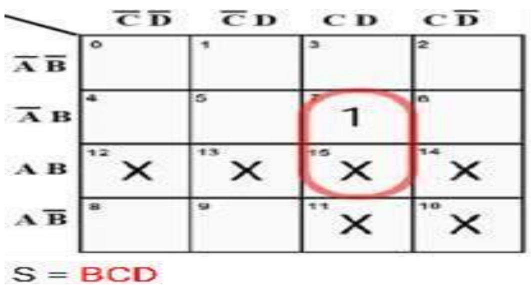
F = yz + w'x'

In Fig. 3.15(b), don't-care minterm 5 is included with the 1's, and the simplified function is now

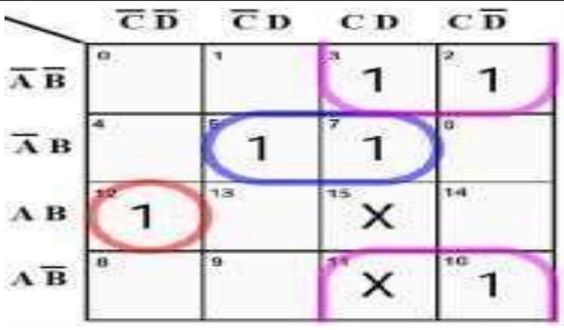
F = yz + w'z

Either one of the preceding two expressions satisfies the conditions stated for this example.

Solve S=F(A,B,C,D)=Σm(7)+d(10,11,12,13,14,15) using K map to get minimum SOP expression

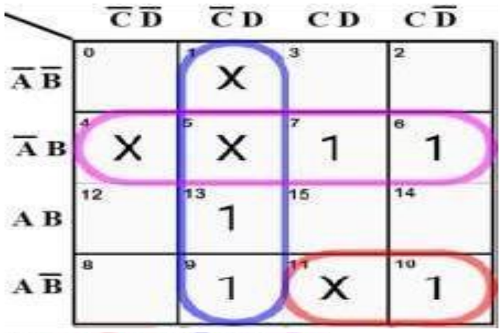


Solve S =F(A,B,C,D)=Σm(2,3,5,7,10,12)+d(11,15) using K map to get minimum SOP expression



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

Solve S=F(A,B,C,D)=Σm(6,7,9,10,13)+d(1,4,5,11) using K map to get minimum SOP expression.



$$S = A\bar{B}C + \bar{C}D + \bar{A}B$$

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.

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 Fig. 3.16.
- The complement operation is obtained from a one-input NAND gate that behaves exactly like an inverter. The AND operation requires two NAND gates. The first produces the NAND operation and the second inverts the logical sense of the signal. The OR operation is achieved through a NAND gate with additional inverters in each input.
- 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 Fig. 3.17. The AND-invert symbol has been defined previously and consists

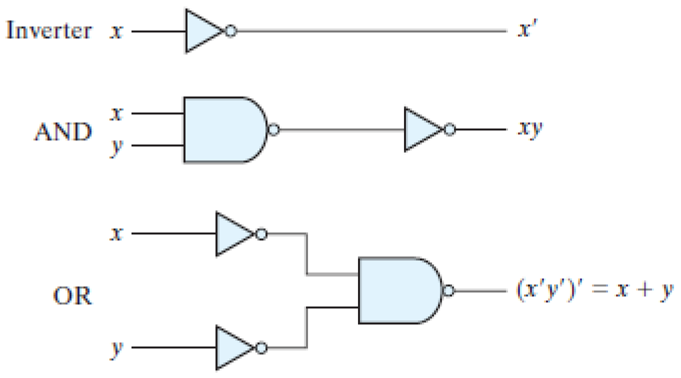


FIGURE 3.16
Logic operations with NAND gates

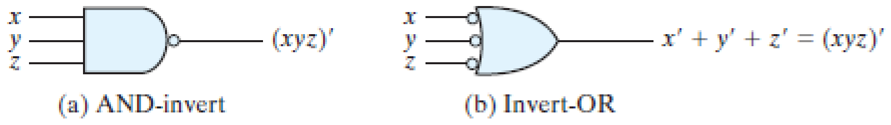


FIGURE 3.17
Two graphic symbols for a three-input NAND gate

of an AND graphic symbol followed by a small circle negation indicator referred to as a bubble.

- It is possible to represent a NAND gate by an OR graphic symbol that is preceded by a bubble in each input. The invert-OR symbol for the NAND gate follows DeMorgan's theorem and the convention that the negation indicator (bubble) denotes complementation. The two graphic symbols' representations are useful in the analysis and design of NAND circuits. When both symbols are mixed in the same diagram, the circuit is said to be in mixed notation.

Two-Level Implementation

- The implementation of Boolean functions with NAND gates requires that the **functions be in sum-of-products form**. To see the relationship between a sum- of-products expression and its equivalent NAND implementation, consider the logic diagrams drawn in Fig. 3.18. All three diagrams are equivalent and implement the function

  F = AB + CD

- The function is implemented in Fig. 3.18(a) with AND and OR gates. In Fig. 3.18(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). Therefore, the two diagrams implement the same function and are equivalent.

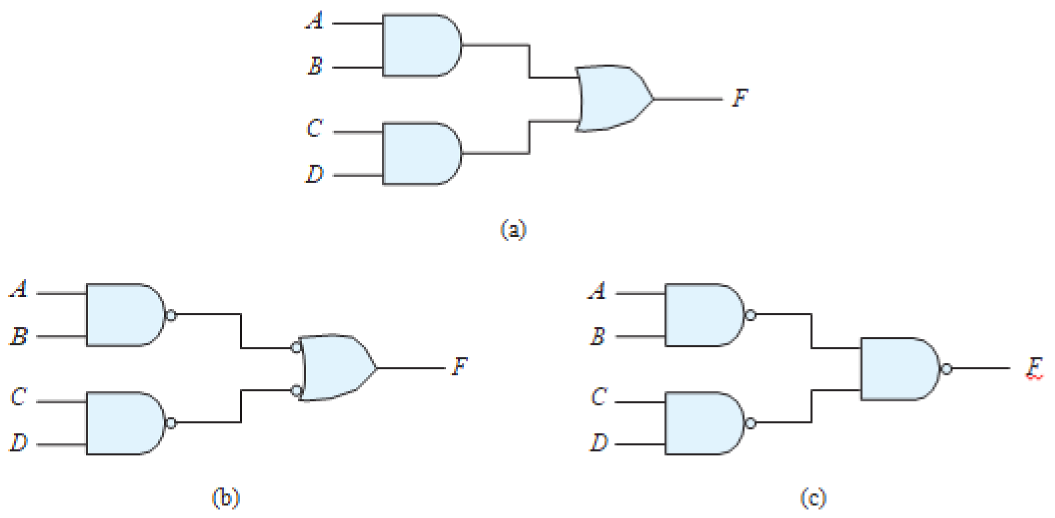


FIGURE 3.18
Three ways to implement *F* = *AB* + *CD*

- In Fig. 3.18(c), the output NAND gate is redrawn with the AND-invert graphic symbol. In drawing NAND logic diagrams, the circuit shown in either Fig. 3.18(b) or (c) is acceptable.
- The one in Fig. 3.18(b) is in mixed notation and represents a more direct relationship to the Boolean expression it implements. The NAND implementation in Fig. 3.18(c) can be verified algebraically.
- The function it implements can easily be converted to sum-of- products form by DeMorgan's theorem:

F = ((AB)'(CD)')' = AB + CD

Implement the following Boolean function with NAND gates:

F (x, y, z) = (1, 2, 3, 4, 5, 7)

- The first step is to simplify the function into sum-of-products form. This is done by means of the map of Fig. 3.19(a), from which the simplified function is obtained:

  F = xy' + x'y + z

- The two-level NAND implementation is shown in Fig. 3.19(b) in mixed notation. Note that input z must have a one-input NAND gate (an inverter) to compensate for the bubble in the second-level gate. An alternative way of drawing the logic diagram is given in Fig. 3.19(c).
- Here, all the NAND gates are drawn with the same graphic symbol. The inverter with input z has been removed, but the input variable is complemented and denoted by z'

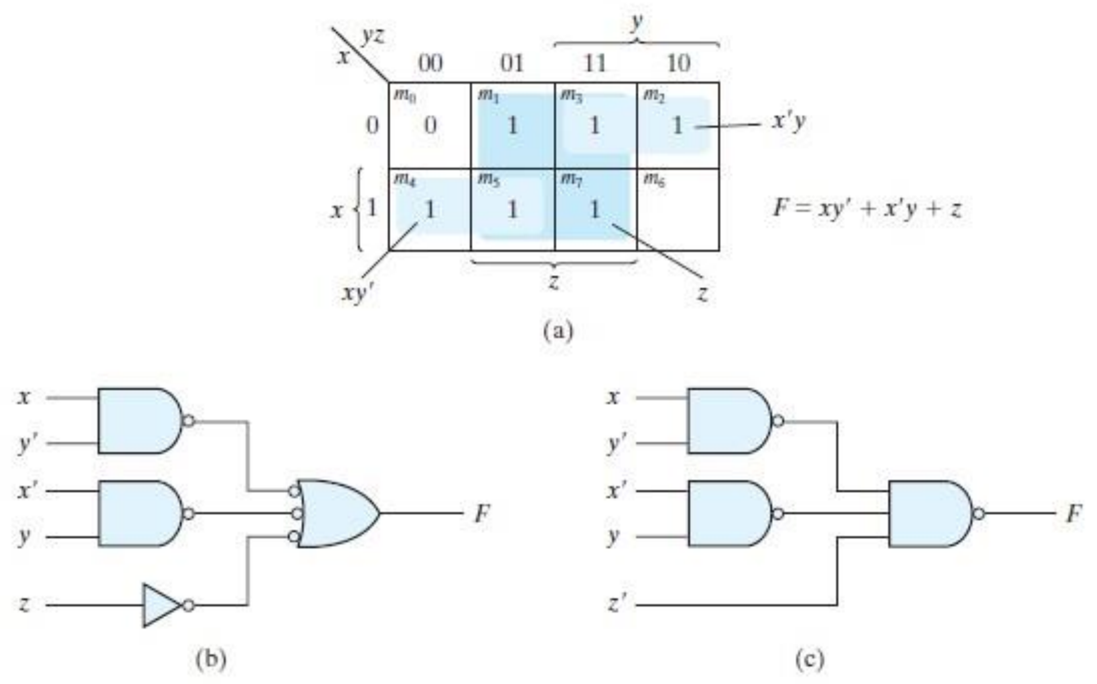


FIGURE 3.19
Solution to Example 3.9

- ➢ The procedure described in the previous example indicates that a Boolean function can be implemented with two levels of NAND gates. The **procedure** for **obtaining the logic diagram from a Boolean function is as follows:**

1. Simplify the function and express it in sum-of-products form.

2. **Draw a NAND gate** for each product term of the expression that has **at least two literals.** The inputs to each NAND gate are the literals of the term. This procedure produces a group of **first-level gates**.

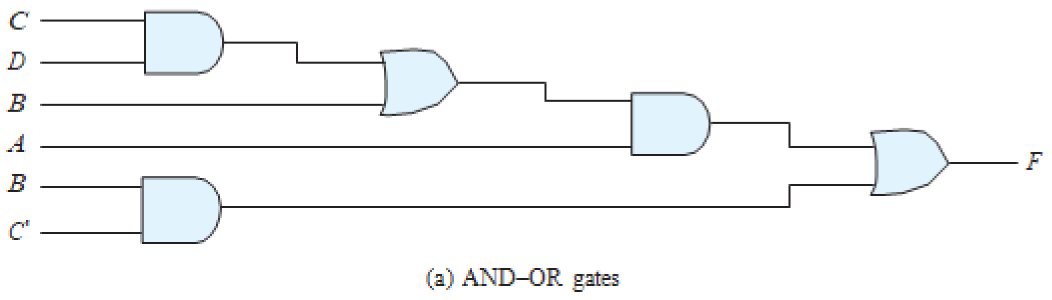
3. Draw a single gate using the **AND-invert or the invert-OR** graphic symbol in the **second level**, with inputs coming from outputs of first-level gates.

4. A term with a single literal requires an inverter in the first level. However, if the single literal is complemented, it can be connected directly to an input of the second-level NAND gate.

Multilevel NAND Circuits

- ➢ The standard form of expressing Boolean functions results in a two-level implementation. There are occasions, however, when the design of digital systems results in gating structures with three or more levels.
- ➢ The most common procedure in the design of multilevel circuits is to express the Boolean function in terms of AND, OR, and complement operations. The function can then be implemented with AND and OR gates. After that, if necessary, it can be converted into an all-NAND circuit.

Consider, for example, the Boolean function F = A (CD + B) + BC'



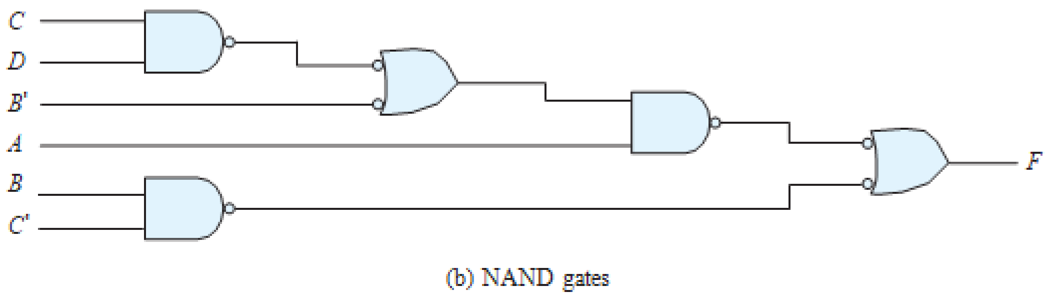


FIGURE 3.20
Implementing $F = A(CD + B) + BC'$

- The AND–OR implementation is shown in Fig. 3.20(a). There are four levels of gating in the circuit. The first level has two AND gates. The second level has an OR gate followed by an AND gate in the third level and an OR gate in the fourth level.
- A logic diagram with a pattern of alternating levels of AND and OR gates can easily be converted into a NAND circuit with the use of mixed notation, shown in Fig. 3.20(b).
- The procedure is to change every AND gate to an AND-invert graphic symbol and every OR gate to an invert-OR graphic symbol. The NAND circuit performs the same logic as the AND–OR diagram as long as there are two bubbles along the same line.
- The bubble associated with input B causes an extra comple- mentation, which must be compensated for by changing the input literal to B'.

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

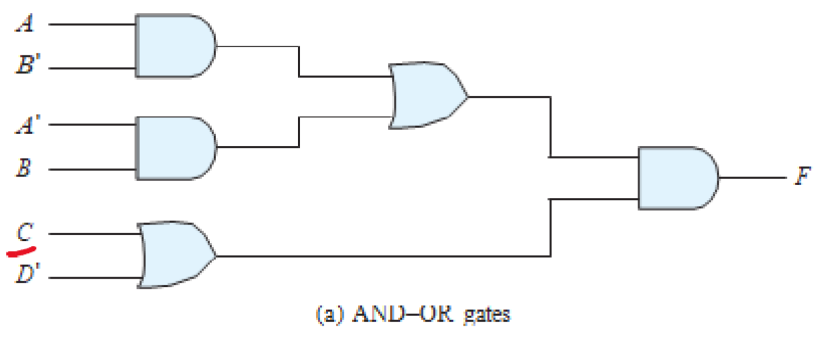
1. **Convert all AND gates** to **NAND** gates with AND-invert graphic symbols.

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.

As another example, consider the multilevel Boolean function

F = (AB' + A'B)(C + D')



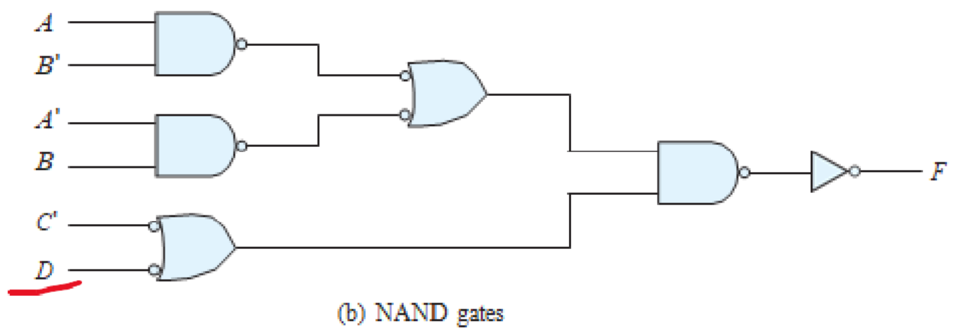


FIGURE 3.21
Implementing $F = (AB' + A'B)(C + D')$

- The AND–OR implementation of this function is shown in Fig. 3.21(a) with three levels of gating. The conversion to NAND with mixed notation is presented in Fig. 3.21(b) of the diagram.
- The two additional bubbles associated with inputs C and D' cause these two literals to be complemented to C' and D.
- The bubble in the output NAND gate complements the output value, so we need to insert an inverter gate at the output in order to complement the signal again and get the original value back.

NOR Implementation

- **The NOR operation is the dual of the NAND operation**. Therefore, all procedures and rules for NOR logic are the duals of the corresponding procedures and rules developed for NAND logic.
- **The NOR gate is another universal gate** that can be used to implement any Boolean function. The implementation of the complement, OR, and AND operations with NOR gates is shown in Fig. 3.22.
- The complement operation is obtained from a one- input NOR gate that behaves exactly like an inverter.The OR operation requires two NOR gates, and the AND operation is obtained with a NOR gate that has inverters in each input.
- The two graphic symbols for the mixed notation are shown in Fig. 3.23. The OR-invert symbol defines the NOR operation as an OR followed by a complement. The invert-AND symbol complements each input and then performs an AND

operation. The two symbols designate the same NOR operation and are logically identical because of DeMorgan's theorem.

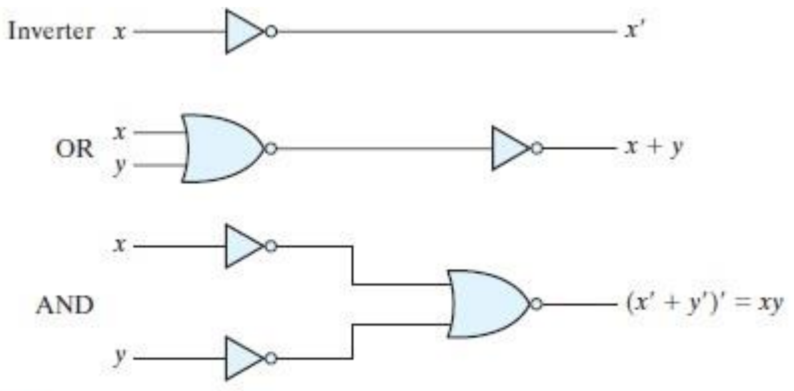


FIGURE 3.22
Logic operations with NOR gates

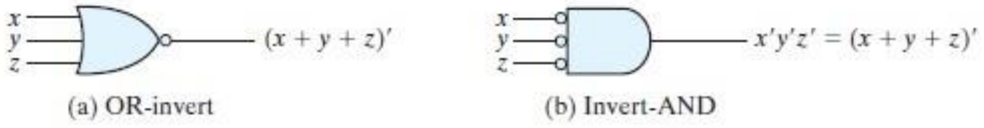


FIGURE 3.23
Two graphic symbols for the NOR gate

- A two-level implementation with NOR gates requires that the function be simplified **into product-of-sums form.**
- Remember that the simplified product-of-sums expression is obtained from the map by combining the 0's and complementing. A product-of-sums expression is implemented with a first level of OR gates that produce the sum terms followed by a second-level AND gate to produce the product.
- The transformation from the OR–AND diagram to a NOR diagram is achieved by **changing the OR gates to NOR gates** with OR-invert graphic symbols and the **AND gate to** a NOR gate with **an invert-AND** graphic symbol.
- A single literal term going into the second-level gate must be complemented.

Figure 3.24 shows the NOR implementation of a function expressed as a product of sums: F = (A + B)(C + D)E

The OR–AND pattern can easily be detected by the removal of the bubbles along the same line. Variable E is complemented to compensate for the third bubble at the input of the second-level gate.

The procedure for converting a multilevel AND–OR diagram to an all-NOR diagram is similar to the one presented for NAND gates. **For the NOR case, we must convert each OR gate to an OR-invert symbol and each AND gate to an invert-AND symbol.** Any bubble that is not compensated by another bubble along the same line needs an inverter, or the complementation of the input literal.

NOR implementation for the following function **F = (A + B)(C + D)E in Fig3.24**

NOR implementation for the following Boolean function is

F = (AB' + A'B)(C + D') in Fig:3.25

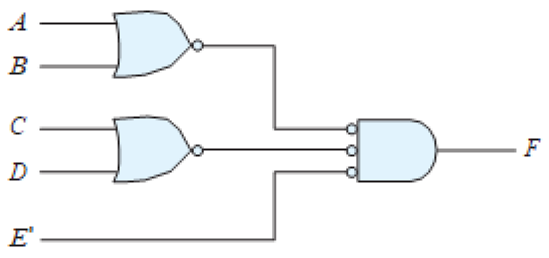


FIGURE 3.24
Implementing $F = (A + B)(C + D)E$

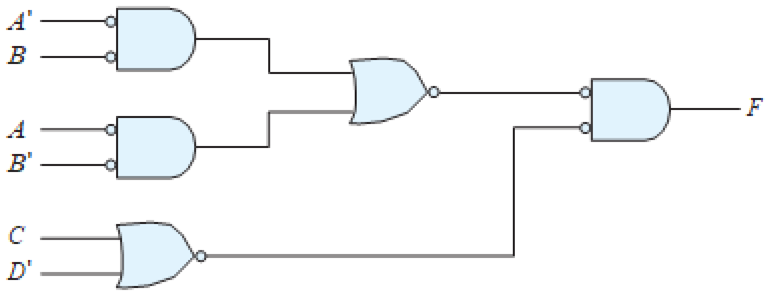


FIGURE 3.25
Implementing $F = (AB' + A'B)(C + D')$ with NOR gates

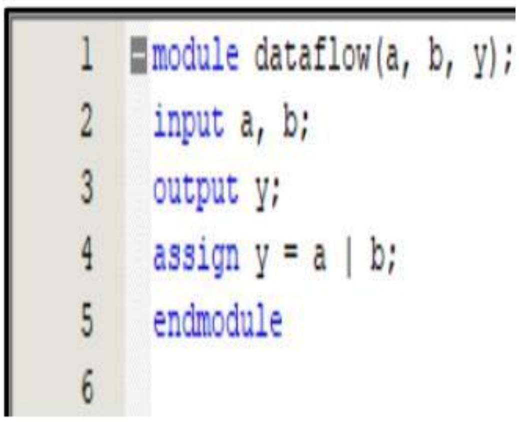
The equivalent AND–OR diagram can be recognized from the NOR diagram by removing all the bubbles. To compensate for the bubbles in four inputs, it is necessary to complement the corresponding input literals.

Hardware Description Language

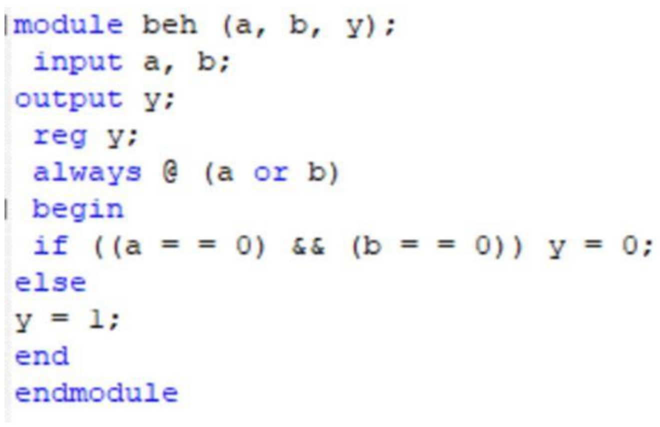
- A hardware description language **(HDL)** is a **computer-based language that describes the hardware of digital systems in a textual form.**
- It resembles an ordinary computer programming language, such as C, but is **specifically oriented to describing hardware structures and the behavior of logic circuits.**
- It can be **used to represent logic diagrams, truth tables, Boolean expressions, and complex abstractions of the behavior of a digital system.**
- One way to view an HDL is to observe that it describes a relationship between signals that are the inputs to a circuit and the signals that are the outputs of the circuit.
- For example, an HDL description of an AND gate describes how the logic value of the gate's output is determined by the logic values of its inputs.

- HDL is used to represent and document digital systems in a form that can be read by both humans and computers and is suitable as an exchange language between designers.
- Companies that design integrated circuits use proprietary and public HDLs. In the public domain, there are **two standard HDLs** that are supported by the IEEE: **VHDL and Verilog.**
- Verilog is an easier language than VHDL to describe, learn, and use, we have chosen it for this book.
- A **Verilog model is composed of text using keywords**, of which there are about 100.
- **Keywords** are **predefined lowercase identifiers** that define the language constructs. **Examples** of **keywords** are **module, endmodule, input, output, wire, and, or, and not.**
- Any text between **two forward slashes (//)** and the end of the line is interpreted as a **comment** and will have no effect on a simulation using the model
- **Multiline comments** begin with **/ * and terminate with * /.**
- **Verilog is case sensitive**, which means that uppercase and lowercase letters are distinguishable (e.g., not is not the same as NOT).
- A module is the fundamental descriptive unit in the Verilog language. It is declared by the keyword module and must always be terminated by the *keyword endmodule*.
- There 3 types of modelling:
- **Dataflow modeling** describes a system in terms of how data flows through the system.
- **Behavioral modeling** describes a system's behavior or function in an algorithmic fashion.
- **Structural modeling** describes a system in terms of its structure and interconnections between components.

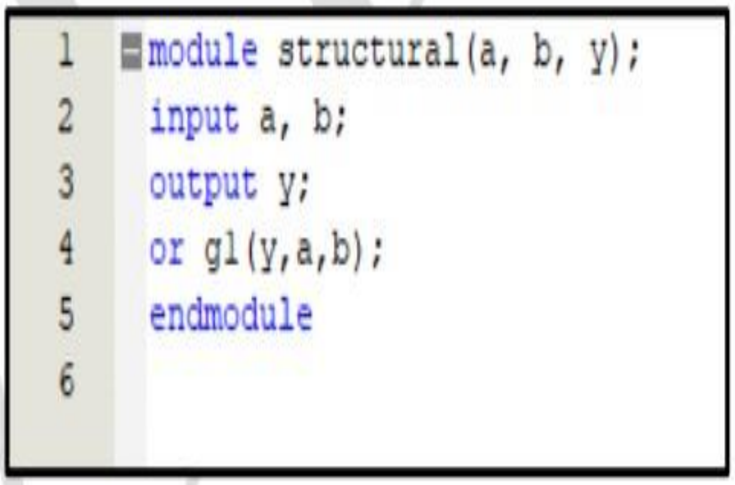
Write a Verilog program for OR gate using i)dataflow modelling ii)Behavioral modelling and iii)structural modelling.



```
module dataflow(a, b, y);
input a, b;
output y;
assign y = a | b;
endmodule
```

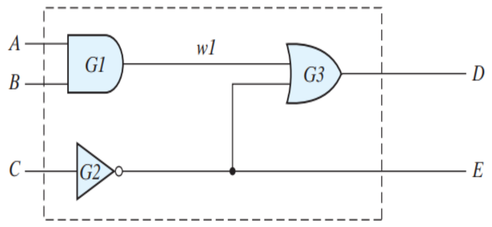


```
module beh (a, b, y);
 input a, b;
output y;
 reg y;
 always @ (a or b)
 begin
 if ((a = = 0) && (b = = 0)) y = 0;
else
y = 1;
end
endmodule
```



```
module structural(a, b, y);
input a, b;
output y;
or g1(y,a,b);
endmodule
```

2) **write a Verilog code for the following circuit.**



```
module Simple_Circuit (A, B, C, D, E);
output D, E;
input A, B, C;
wire w1;
and G1 (w1, A, B); // Optional gate instance name
not G2 (E, C);
or G3 (D, w1, E);
endmodule
```

HDL describes a circuit that is specified with the following two Boolean expressions:

$$E = A + BC + B'D$$
$$F = B'C + BC'D'$$

```
module beh(E, F, A, B, C, D);
output E, F;
input A, B, C, D;
assign E = A || (B && C) || ((!B) && D);
assign F = ((!B) && C) || (B && (!C) && (!D));
endmodule
```