Damietta University
Faculty of Computers
and Information

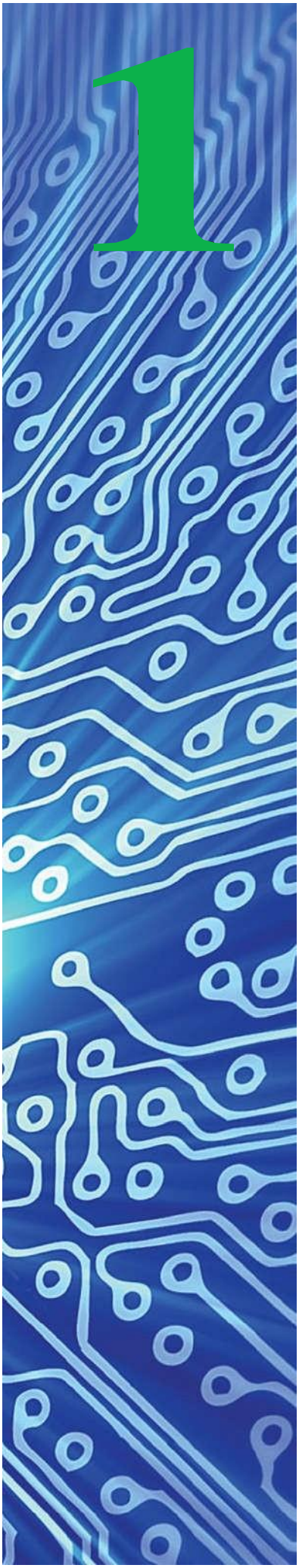
LOGIC DESIGN

Prepared By

Dr. Mona Nagy ElBedwehy

Lecturer of Computer Science, Faculty of
Computers and Information, Damietta University

Contents



1 Number Systems and Codes

OUTLINE

OBJECTIVES

Upon completion of this chapter, you should be able to:

- Determine the weighting factor for each digit position in the *decimal*, *binary*, *octal*, and *hexadecimal* numbering systems.
- Convert any number in one of the four number systems (decimal, binary, octal, and hexadecimal) to its equivalent value in any of the remaining three numbering systems.
- Describe the binary-coded decimal (BCD) number.

2.1 INTRODUCTION

Digital circuitry is the foundation of digital computers and many automated control systems. In a modern home, digital circuitry controls the appliances, alarm systems, and heating systems. Under the control of digital circuitry and microprocessors, newer automobiles have added safety features, are more energy efficient, and are easier to diagnose and correct when malfunctions arise.

Other uses of digital circuitry include the areas of automated machine control, energy monitoring and control, inventory management, medical electronics, and music. The area of medical electronics uses digital thermometers, life-support systems, and monitors. We have also seen more use of digital electronics in the reproduction of music. Digital reproduction is less susceptible to electrostatic noise and therefore can reproduce music with greater fidelity.

Digital electronics evolved from the principle that transistor circuitry could easily be fabricated and designed to output one of two voltage levels based on the levels placed at its inputs. The two distinct levels (usually +5 volts [V] and 0 V) are HIGH and LOW and can be represented by **1** and **0**.

The *binary* numbering system is made up of only **1**s and **0**s and is therefore used extensively in digital electronics. The other numbering systems and codes covered in this chapter represent groups of binary digits and therefore are also widely used.

2.2 Digital versus Analog

Digital systems are used in computation and data processing, control systems, communications, and measurement. Because digital systems are capable of greater accuracy and reliability than analogue systems, many tasks formerly done by analogue systems are now being performed digitally.

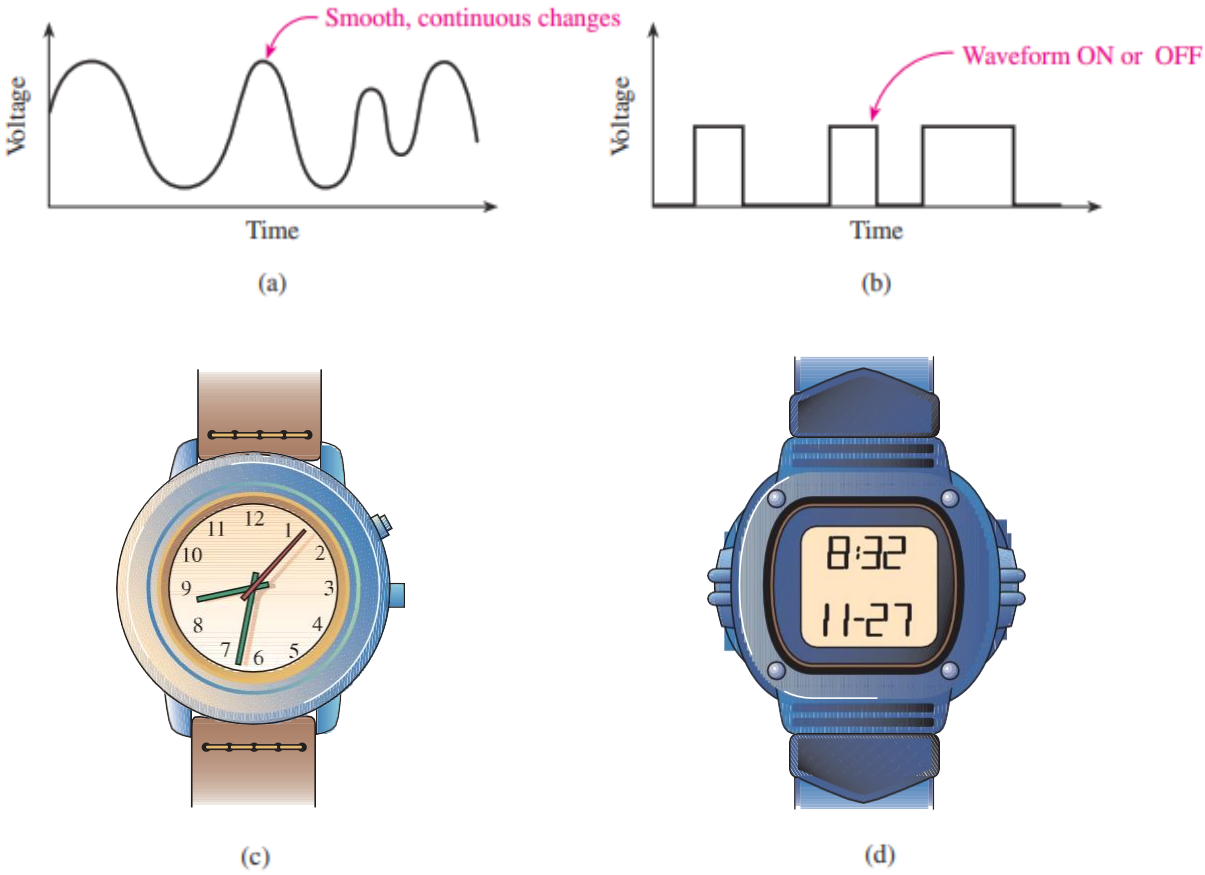


Figure 1-1 Analog versus digital: (a) analog waveform; (b) digital waveform; (c) analog watch; (d) digital watch.

Digital systems operate on discrete digits that represent numbers, letters, or symbols. They deal strictly with **ON** and **OFF** states, which we can represent by **0**s and **l**s. Analog systems measure and respond to continuously varying

electrical or physical magnitudes. Analog devices are integrated electronically into systems to continuously monitor and control such quantities as temperature, pressure, velocity, and position and to provide automated control based on the levels of these quantities. Figure 1-l shows some examples of digital and analog quantities.

Review Questions

1-1 List three examples of *analog* quantities.

1-2 Why do computer systems deal with *digital* quantities instead of *analog* quantities?

2.3 Number System

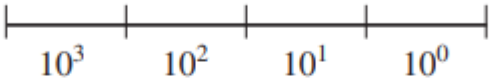
Numbers can be represented in different *bases*. If *base* of a number system is *r*, then the numbers present in that number system are ranging from **zero** to ***r*-1**. The total numbers present in that number system is ***r***. So, we will get various number systems, by choosing the values of bases as greater than or equal to two. Let us discuss about the popular number systems and how to represent a number in the respective number system.

2.4 Decimal Numbering System (Base 10)

In the *decimal* numbering system, each position contains **10** different possible digits. These *digits* are 0, l, 2, 3, 4, 5, 6, 7, 8, and 9. Each position in a multidigit number will have a *weighting* factor based on a power of **10.**

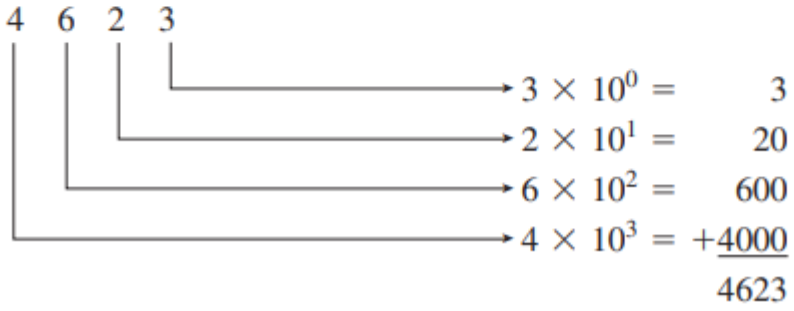
EXAMPLE 1–1

In a four-digit decimal number, the least significant position (rightmost) has a weighting factor of 10^0; the most significant position (leftmost) has a weighting factor of 10^3:



where $10^3 = 1000$
$10^2 = 100$
$10^1 = 10$
$10^0 = 1$

To evaluate the decimal number 4623, the digit in each position is multiplied by the appropriate weighting factor:



2.5 Binary Numbering System (Base 2)

All digital circuits and systems use this *binary* numbering system because it uses only the digits **0** and **1**, which can be represented simply in a digital system by two distinct voltage levels, such as +5 V = 1 and 0 V = 0. The *base* of this number system is **2**. The number lies to the left of the binary point is known as *integer part*. The number lies to the right of the binary point is known as *fractional part*. The weighting factors for binary positions are the powers of 2 shown in Table 1-1.

TABLE 1–1 Powers-of-2 Binary Weighting Factors

128	64	32	16	8	4	2	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

$2^0 = 1$
$2^1 = 2$
$2^2 = 4$
$2^3 = 8$
$2^4 = 16$
$2^5 = 32$
$2^6 = 64$
$2^7 = 128$

2.6 Octal Numbering System (Base 8)

The *octal* numbering system is a method of grouping binary numbers in groups of three. The eight allowable digits are 0, 1, 2, 3, 4, 5, 6, and 7.

The octal numbering system is used by manufacturers of computers that utilize 3-bit codes to indicate instructions or operations to be performed. By using the octal representation instead of binary, the user can simplify the task of entering/ reading computer instructions and thus save time. In Table 1- 2, we see that when the octal number exceeds 7, the least significant octal position resets to zero and the next most significant position increases by 1.

TABLE 1–2 Octal Numbering System

Decimal	Binary	Octal
0	000	0
1	001	1
2	010	2
3	011	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	10
9	1001	11
10	1010	12

2.7 Hexadecimal Numbering System (Base 16)

The *hexadecimal* numbering system, like the octal system, is a method of grouping bits to simplify entering and reading the instructions or data present in digital computer systems. Hexadecimal uses 4-bit groupings; therefore,

instructions or data used in 8-, 16-, or 32-bit computer systems can be represented as a two-, four-, or eight-digit hexadecimal code instead of using a long string of binary digits (see Table 1-3).

Hexadecimal (hex) uses 16 different digits and is a method of grouping binary numbers in groups of four. Because hex digits must be represented by a single character, letters are chosen to represent values greater than 9. The 16 allowable hex digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

TABLE 1–3 Hexadecimal Numbering System

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	0001 0000	1 0
17	0001 0001	1 1
18	0001 0010	1 2
19	0001 0011	1 3
20	0001 0100	1 4

2.8 Binary to Decimal Conversion

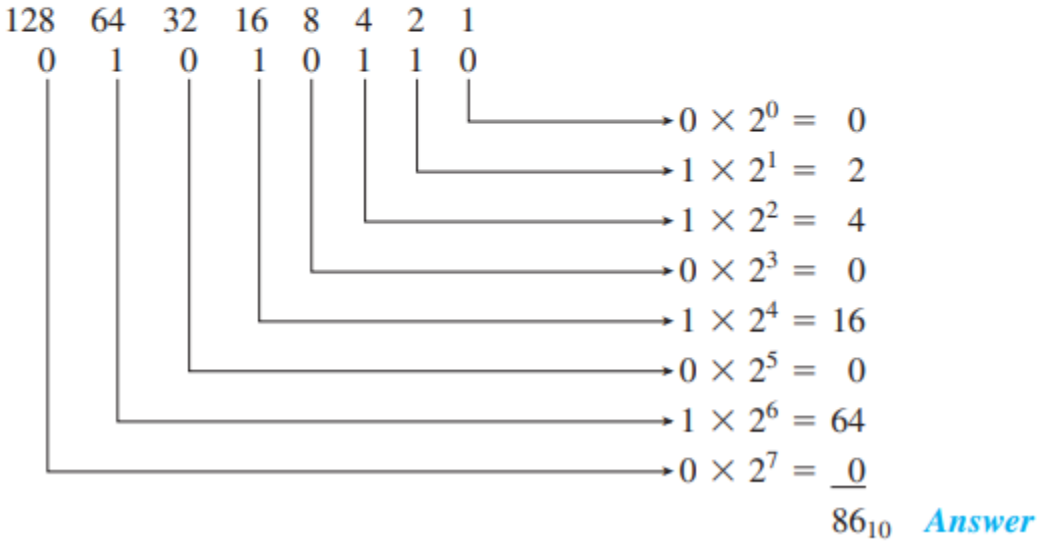
1. Multiply the bits of binary number with the respective positional weights.

2. Add all those products.

EXAMPLE 1–2

Convert the binary number 01010110_2 to decimal. (Notice the subscript 2 used to indicate that 01010110 is a base 2 number. A capital letter B can also be used, i.e., 01010110B.)

Solution: Multiply each binary digit by the appropriate weight factor and total the results.



86_{10} *Answer*

Review Questions

1-3. Why is the binary numbering system commonly used in digital electronics?

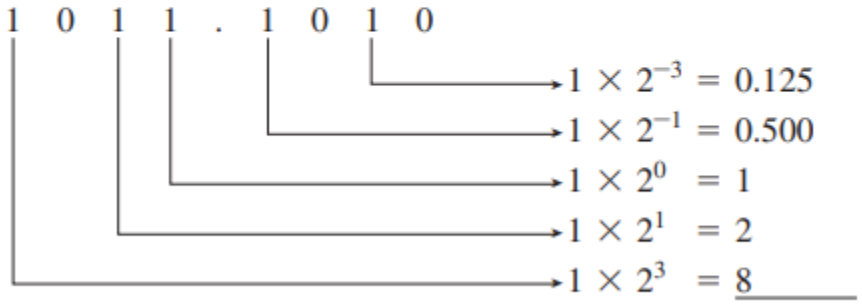
1-4. How are the weighting factors determined for each binary position in a base 2 number?

1-5. Convert $0110\ 1100_2$ to decimal.

EXAMPLE 1–3

Convert the fractional binary number 1011.1010_2 to decimal.

Solution: Multiply each binary digit by the appropriate weighting factor given in Figure 1–6, and total the results. (We skip the multiplication for the binary digit 0 because it does not contribute to the total.)

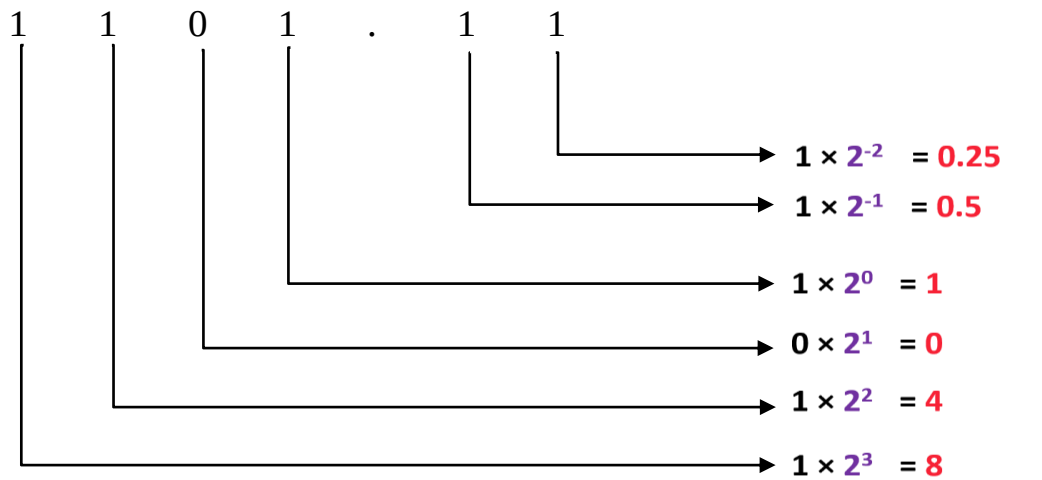


11.625_{10} ***Answer***

EXAMPLE 1 - 4

Convert the fractional binary number 1101.11_2 to decimal.

Solution:



13.75

2.9 Decimal to Binary Conversion

1. Divide the integer decimal number to be converted by the value of the new base.
2. Get the remainder from Step1 as the rightmost digit (least significant digit) (LSD) of new base number.
3. Divide the quotient of the previous divide by the new base.
4. Record the remainder from Step3 as the next digit (to the left) of the new base number.
5. Repeat Steps 3 and 4, getting remainders from right to left, until the quotient becomes zero in Step 3.
6. The last remainder thus obtained will be the Most Significant Digit (MSD) of the new base number.
7. Multiply the fractional part and successive fractions with the base of the new system.

EXAMPLE 1 - 5

Convert 152_{10} to binary using successive division.

Solution:

$152 \div 2 = 76$ remainder 0 (LSB)
$76 \div 2 = 38$ remainder 0
$38 \div 2 = 19$ remainder 0
$19 \div 2 = 9$ remainder 1
$9 \div 2 = 4$ remainder 1
$4 \div 2 = 2$ remainder 0
$2 \div 2 = 1$ remainder 0
$1 \div 2 = 0$ remainder 1 (MSB)

Answer: $1\ 0\ 0\ 1\ 1\ 0\ 0\ 0_2$

EXAMPLE 1 - 6

Convert the 58.25_{10} to binary.

Solution Here, the *integer* part is 58 and *fractional* part is 0.25.

Step 1: Division of 58 and successive quotients with base 2.

	Quotient	Remainder	
58	2		
29	2	0	LSB
14	2	1	
7	2	0	
3	2	1	
1	2	1	
0		1	MSB

Therefore, the *integer* part of equivalent binary number is 111010.

Step 2: Multiplication of 0.25 and successive fractions with base 2.

	Integer		Fraction	Coefficient
0.250 × 2 = 0. 500	0	+	0.500	0
0.500 × 2 = 1	1	+	0.000	1

$(58.25)_{10} = 111010.01_2$

EXAMPLE 1 - 7

Convert the 0.8675_{10} to binary.

Solution Multiplication of 0.8675 and successive fractions with base 2.

	Integer		Fraction	Coefficient
0.8675 × 2 = 1.375	1	+	0.375	1
0.375 × 2 = 0.750	0	+	0.750	0
0.750 × 2 = 1. 500	1	+	0.500	1
0.500 × 2 = 1	1	+	0.000	1

$(0.8675)_{10} = 0.1011_2$

2.10 Octal to Decimal Conversion

1. Multiply the bits of *octal* number with the respective positional *weights*.
2. Add all those products.

EXAMPLE 1 - 8

Convert $3\ 2\ 6_8$ to decimal.

Solution:

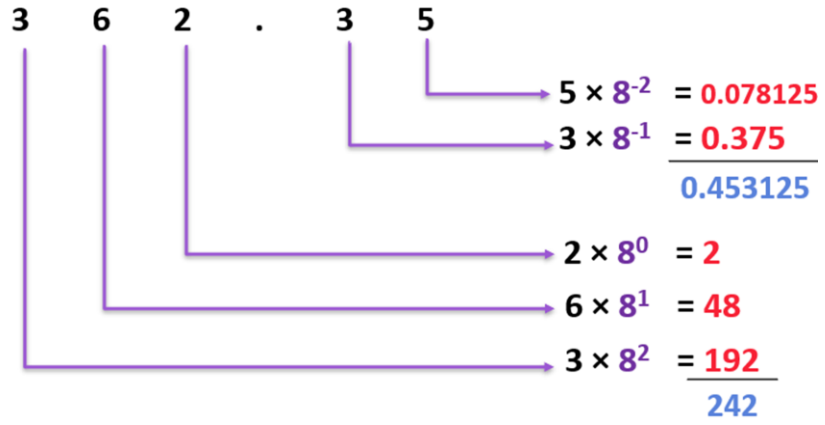
$$3\ 2\ 6$$

$$6 \times 8^0 = 6 \times 1 = 6$$
$$2 \times 8^1 = 2 \times 8 = 16$$
$$3 \times 8^2 = 3 \times 64 = \underline{192}$$
$$214_{10}$$ *Answer*

EXAMPLE 1 - 9

Convert 362.35_8 to decimal.

Solution



$(0.8675)_{10} = 0.1011_2$

2.11 Decimal to Octal Conversion

EXAMPLE 1 - 10

Convert the 359_{10} to octal.

Solution

	Quotient	Remainder	
359	8		
44	8	7	LSB
5	8	4	
0		5	MSB

$(359)_{10} = 547_8$

EXAMPLE 1 - 11

Convert the 153.513_{10} to octal.

Solution **Step 1** – Division of 153 and successive quotients with base 8.

	Quotient	Remainder	
153	8		
19	8	1	LSB
2	8	3	
0		2	MSB

Step 2 – Multiplication of 0.513 and successive fractions with *base* 8.

	Integer		Fraction	Coefficient
0.513 × 8 = 4.104	4	+	0.104	4
0.104 × 8 = 0.832	0	+	0.832	0
0.832 × 8 = 6.656	6	+	0.656	6
0.656 × 8 = 5.248	5	+	0.248	5
0.248 × 8 = 1.984	1	+	0.984	1

$(153.513)_{10} = 231.40651_8$

2.12 Binary to Octal Conversion

1. Divide the binary digits into groups of three (starting from the right).
2. Convert each group of three binary digits to one octal digit.

EXAMPLE 1 - 12

Convert $0\ 1\ 1\ 1\ 0\ 1_2$ to octal.

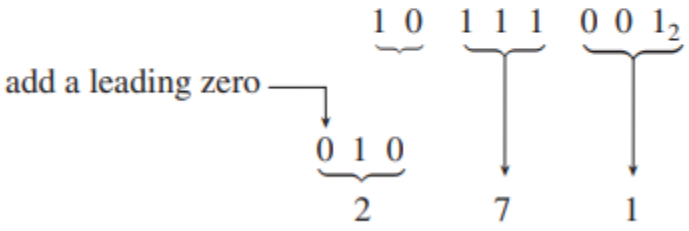
Solution:

$$\underbrace{0\ 1\ 1}_{3}\ \ \underbrace{1\ 0\ 1}_{5} = 35_8 \quad \textit{Answer}$$

EXAMPLE 1 - 13

Convert $1\ 0\ 1\ 1\ 1\ 0\ 0\ 1_2$ to octal.

Solution:



$= 271_8$ *Answer*

EXAMPLE 1 - 14

Convert 101110.01101_2 to octal.

Solution

Step 1 − Make the groups of 3 bits.

1	0	1	1	1	0	.	0	1	1	0	1	0

Step 2 − Write the *octal* digits corresponding to each group of 3 bits.

5	6	.	3	2

$(101110.01101)_2 = 56.32_8$

2.13 Octal to Binary Conversion

1. Convert each octal digit to a 3 digit binary number (the octal digits may be treated as decimal for this conversion).
2. Combine all the resulting binary groups (of 3 digits each) into a single binary number.

EXAMPLE 1 - 15

Convert 25_8 to binary.

Solution

2			5		
0	1	0	1	0	1

$(25)_8 = 010\ 101_2$

2.14 Decimal to Hexadecimal Conversion

EXAMPLE 1 - 16

Convert the 348_{10} to hexadecimal.

Solution

	Quotient	Remainder	
348	16		
21	16	12 = C	LSB
1	16	5	
0		1	MSB

$(348)_{10} = 15C_{16}$

2.15 Hexadecimal to Decimal Conversion

EXAMPLE 1 - 18

Convert 2 A 6_{16} to decimal.

Solution:

2 A 6

$6 \times 16^0 = 6 \times 1 = 6$

$A \times 16^1 = 10 \times 16 = 160$

$2 \times 16^2 = 2 \times 256 = 512$

678_{10} *Answer*

2.16 Binary to Hexadecimal Conversion

EXAMPLE 1 - 19

Convert 101110.01101_2 to hexadecimal.

Solution

Step 1 − Make the groups of 4 bits.

0 0 1 0 1 1 1 0 . 0 1 1 0 1 0 0 0

Step 2 − Write the *hex* digits corresponding to each group of 4 bits.

2 13=**E** . 6 8

$(101110.01101)_2 = 2E.68_{16}$

EXAMPLE 1 - 20

Convert 01101101_2 to hex.

Solution:

0 1 1 0 (6) 1 1 0 1 (D) $_2$ $= 6D_{16}$ *Answer*

2.17 Hexadecimal to Binary Conversion

EXAMPLE 1 - 21

Convert $(65.4C)_{16}$ to binary.

Solution

6	5	.	4	C
0 1 1 0	0 1 0 1	.	0 1 0 0	1 1 0 0

$(65.4C)_{16} = 0110\ 0101.0100\ 1100_2$

2.18 Binary-Coded-Decimal (BCD) System

The binary-coded-decimal (BCD) system is used to represent each of the 10 decimal digits as a 4-bit binary code. This code is useful for outputting to displays that are always numeric (0 to 9), such as those found in digital clocks or digital voltmeters. To form a BCD number, simply convert each decimal digit to its 4-bit binary code.

EXAMPLE 1 - 22

Convert $4\ 9\ 6_{10}$ to BCD.

Solution:

$$\underbrace{4}_{0100}\ \underbrace{9}_{1001}\ \underbrace{6}_{0110} = 0100\ \ 1001\ \ 0110_{BCD}$$ *Answer*

EXAMPLE 1 - 23

Convert $0111\ \ 0101\ \ 1000_{BCD}$ to decimal.

Solution:

$$\underbrace{0111}_{7}\ \underbrace{0101}_{5}\ \underbrace{1000}_{8} = 758_{10}$$ *Answer*

EXAMPLE 1 - 24

Convert $0110\ 0100\ 1011_{BCD}$ to decimal.

Solution:

0110	0100	1011
6	4	*

*This conversion is impossible because 1011 is not a valid binary-coded decimal. It is not in the range 0 to 9.

2.19 Comparison of Numbering Systems

Table 1–4 compares numbers written in the five number systems commonly used in digital electronics and computer systems.

TABLE 1–4 Comparison of Numbering Systems

Decimal	Binary	Octal	Hexadecimal	BCD
0	0000	0	0	0000
1	0001	1	1	0001
2	0010	2	2	0010
3	0011	3	3	0011
4	0100	4	4	0100
5	0101	5	5	0101
6	0110	6	6	0110
7	0111	7	7	0111
8	1000	1 0	8	1000
9	1001	1 1	9	1001
10	1010	1 2	A	0001 0000
11	1011	1 3	B	0001 0001
12	1100	1 4	C	0001 0010
13	1101	1 5	D	0001 0011
14	1110	1 6	E	0001 0100
15	1111	1 7	F	0001 0101
16	0001 0000	2 0	1 0	0001 0110
17	0001 0001	2 1	1 1	0001 0111
18	0001 0010	2 2	1 2	0001 1000
19	0001 0011	2 3	1 3	0001 1001
20	0001 0100	2 4	1 4	0010 0000

Problems

1. **Convert the following binary numbers to decimal:**

(a) 100001	(b) 10011.1	(c) 101010	(d) 1110.01
(e) 1100000	(f) 11111101	(g) 11110.010	(h) 11111.111

2. **Convert the following decimal numbers to binary and hexadecimal**

(a) 12	(b) 50	(c) 97	(d) 127
(e) 19.8	(f) 390	(g) 98.52	

3. **Convert the following hexadecimal numbers to binary and decimal**

(a) 46	(b) 54	(c) B4	(d) 1A.3
(e) FA	(f) ABC	(g) AB.CD	

4. **Convert the following binary numbers to hexadecimal and octal**

(a) 1111	(b) 11111	(c) 10101010	(d) 10101100
(e) 101110.11	(f) 00110.011		

5. **Convert the following hexadecimal numbers to decimal**

(a) 42	(b) 64	(c) 4D.16	(d) 2B1
(e) FF	(f) BC	(g) 6F11	(h) ABC.6

6. **Convert the following decimal numbers to hexadecimal**

(a) 3652	(b) 8925	(c) 79778

7. **Convert the following octal numbers to decimal and binary**

(a) 76.47	(b) 2673	(c) 25.4	(d) 174
(e) 67	(f) 456		

8. Convert the following decimal numbers to octal

(a) 23 (b) 45 (c) 65 (d) 84

9. Convert the following BCD numbers to decimal:

(a) $0011\ 0110_{BCD}$ (b) $0110\ 1001_{BCD}$ (c) $0111\ 0100_{BCD}$ (d) $1000\ 0001_{BCD}$

10. Convert the following decimal numbers to BCD:

(a) 87_{10} (b) 142_{10} (c) 94_{10} (d) 44_{10}

اسم الطالب: .....

رقم الطالب: .....

اسم المقرر: .....

رقم الشيت: .....

الرقم السري: .....

استمارة التقويم المستمر

Question No.	Degree	Signature
1.		
2.		
3.		
4.		
5.		
6.		
7.		
8.		
9.		
10.		

10

2

Binary Arithmetic

OUTLINE

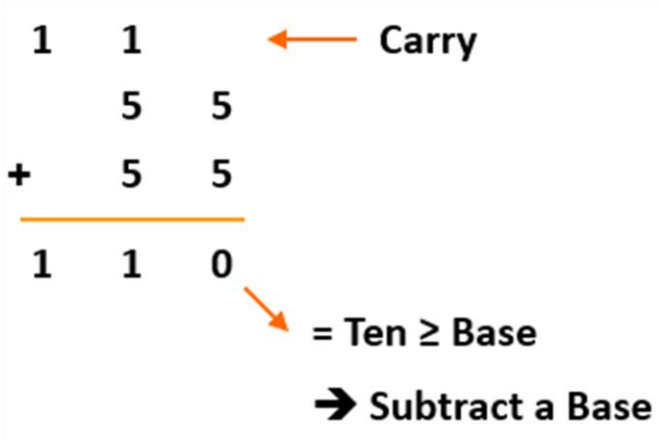
2.1 INTRODUCTION

Arithmetic operations in digital systems are usually done in binary because design of logic circuits to perform binary arithmetic is much easier than for decimal. Binary arithmetic is carried out in much the same manner as decimal, except the addition and multiplication tables are much simpler.

EXAMPLE 2 - 1

Compute $(55)_{10} + (55)_{10}$.

Solution



2.2 Binary Addition

The *addition* table for *binary* numbers is:

0 + 0 = 0

0 + 1 = 1

1 + 0 = 1

1 + 1 = 0 and carry 1 to the next column.

Carrying 1 to a column is equivalent to adding 1 to that column.

EXAMPLE 2 - 2

Compute $(10)_{10} + (7)_{10}$ in binary.

Solution

$(10)_{10} = (1010)_2$ and $(7)_{10} = (0111)_2$

```
  1  1  1
     1  0  1  0     = 10
+    0  1  1  1     = 7
-------------------
  1  0  0  0  1     = 17
                  ≥ (2)10
```

EXAMPLE 2 - 3

Compute $(61)_{10} + (23)_{10}$ in binary.

Solution

$(61)_{10} = (111101)_2$ and $(23)_{10} = (10111)_2$

```
  1  1  1  1  1  1
     1  1  1  1  0  1     = 61
+       1  0  1  1  1     = 23
-----------------------
  1  0  1  0  1  0  0     = 84
                       ≥ (2)10
```

2.3 Binary Subtraction

The subtraction table for binary numbers is

$0 - 0 = 0$

$0 - 1 = 1$ and borrow **1** from the next column

$1 - 0 = 1$

$1 - 1 = 0$

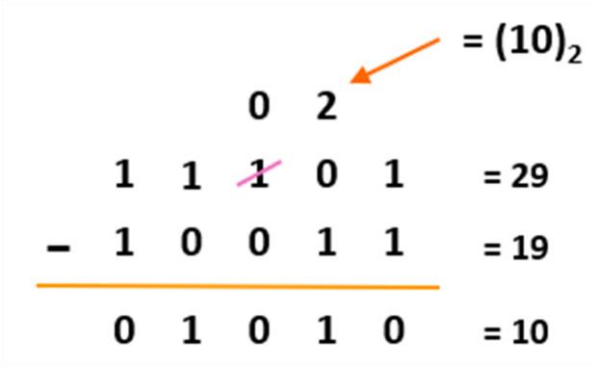
Borrowing 1 from a column is equivalent to subtracting 1 from that column.

EXAMPLE 2 - 4

Compute $(29)_{10} - (19)_{10}$ in binary.

Solution

$(29)_{10} = (11101)_2$ and $(19)_{10} = (10011)_2$

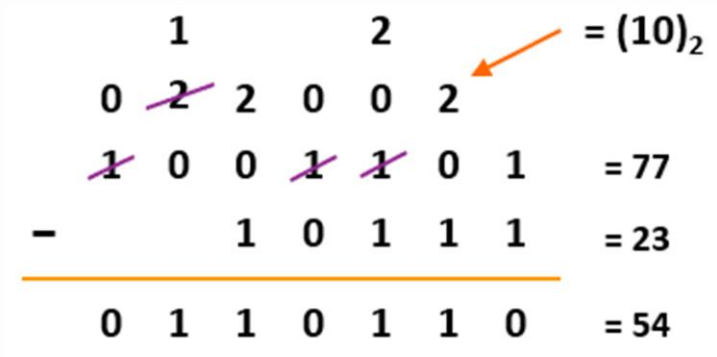


EXAMPLE 2 - 5

Compute $(77)_{10} - (23)_{10}$ in binary.

Solution

$(77)_{10} = (10001101)_2$ **and** $(23)_{10} = (10111)_2$

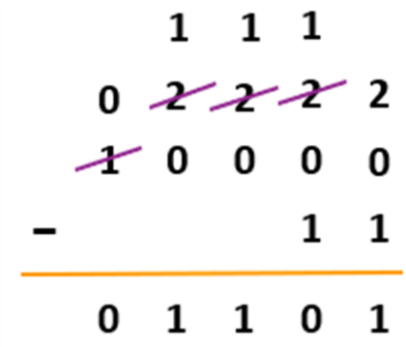


EXAMPLE 2 - 6

Compute $(16)_{10} - (3)_{10}$ in binary.

Solution

$(16)_{10} = (10000)_2$ **and** $(3)_{10} = (11)_2$



2.4 Binary Multiplication

The *multiplication* table for binary numbers is:

$$0 \times 0 = 0$$

$$0 \times 1 = 0$$

$$1 \times 0 = 0$$

$$1 \times 1 = 1$$

EXAMPLE 2 - 7

Compute $(10111)_2 \times (1010)_2$.

Solution

			1	0	1	1	1
x				1	0	1	0
			0	0	0	0	0
		1	0	1	1	1	
	0	0	0	0	0		
1	0	1	1	1			
1	1	1	0	0	1	1	0

Note that each partial product is either the multiplicand (10111) shifted over the appropriate number of places or is zero.

When adding up long columns of binary numbers, the sum of the bits in a single column can exceed 11_2, and therefore the carry to the next column can be greater than 1. For example, if a single column of bits contains five 1's, then adding up the 1's gives 101_2, which means that the sum bit for that column is 1, and the carry to the next column is 10_2. When doing binary multiplication, a common way to

avoid carries greater than 1 is to add in the partial products one at a time.

EXAMPLE 2 - 8

Compute $(1011)_2 \times (101)_2$.

Solution

		1	0	1	1
x			1	0	1
		1	0	1	1
	0	0	0	0	
1	0	1	1		
1	1	0	1	1	1

Compute $(11011.101)_2$

We multiply numbers by normal way, then we put the decimal point after the sum of the number of digits of the two numbers (that is, if the first number has a decimal point after 3 digits and the second number has a decimal point after 2 digits, then we put the decimal point in the result of the multiplication after 5 digits).

2.5 Binary Division

Binary division is similar to decimal division, except it is much easier because the only two possible quotient digits are 0 and 1. For example, if we start by comparing the divisor (1011) with the upper four bits of the dividend (1001), we find that we cannot subtract without a negative result, so we move the divisor one place to the right and try again. This time we can subtract 1011 from 10010 to give 111 as a result, so we put the first quotient bit of 1 above 10010. We then bring down the next dividend bit (0) to get 1110 and shift the divisor right. We then subtract 1011 from 1110 to get 11, so the second quotient bit is 1.

EXAMPLE 2 - 10

Compute $(10010)_2 \div (11)_2$.

Solution

```
          0  1  1  0
      ┌──────────────
11    │ 1  0  0  1  0
    - │    1  1
      ├──────────────
      0  0  1  1
         1  1
      ├──────────────
            0  0  0
            0  0  0
      ├──────────────
                  0
```

When we bring down the next dividend bit, the result is 110, and we cannot subtract the shifted divisor, so the third quotient bit is 0. We then bring down the last dividend bit and subtract 1011 from 1101 to get a final remainder of 10, and the last quotient bit is 1.

When dividing the decimal numbers in the binary system, we unify the number of digits after the decimal point by supplying the number 0, for example (10.1) and the number (111.10), in order to equal the number of decimal digits, we supply 0 for the number (10.1) so it becomes (10.10), after that we ignore the decimal point of both numbers becomes the problem $(11110)_2 \div (1010)_2$, then we solve the problem normal.

EXAMPLE 2 - 11

Compute $(1111)_2 \div (1010)_2$

Solution

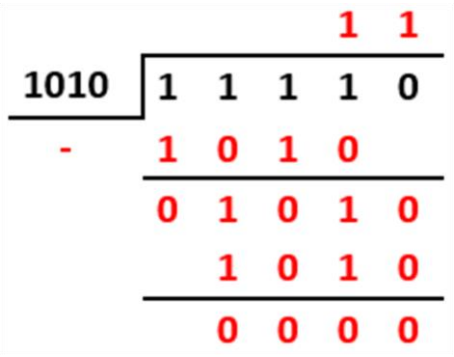
```
                1 . 1
        ---------------
  1010 | 1  1  1  1
       
    -    1  0  1  0
        ---------------
         0  1  0  1  0
            1  0  1  0
        ---------------
            0  0  0  0
```

$$(1111)_2 \div (1010)_2 = (1.1)_2$$

EXAMPLE 2 - 12

Compute $(111.10)_2 \div (10.1)_2$

Solution



$$(111.10)_2 \div (10.1)_2 = (11)_2$$

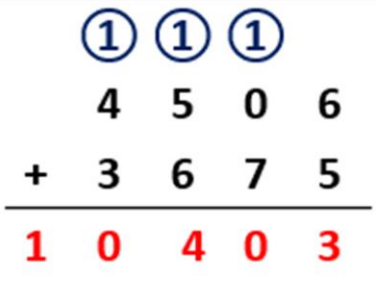
2.6 Octal Arithmetic: Addition

The addition in the octal number is as follows: $6 + 5 = 11_{10}$, and because $11 > 8$. We will subtract 8 from 11 because the base of the *octal* system is 8, so the result is 3 and we carry 1 to the next digit.

EXAMPLE 2 - 13

Compute $(4506)_8 + (3675)_8$

Solution



Second, $7 + 1 = 8_{10}$, and because 8 = 8. So, we will subtract 8 from 8 because the base of the octal system is 8, so the result is 0 and we carry 1 to the next digit.

Third, $6 + 6 = 12_{10}$, and because 12> 8. So, we will subtract 8 from 12 because the base of the octal system is 8, so the result is 4 and we carry 1 to the next digit.

Fourth, $7 + 1 = 8_{10}$, and because 8 = 8. We will subtract 8 from 8 because the base of the octal system is 8, so the result is 0 and we carry 1 to the next digit.

EXAMPLE 2 - 14

Compute $(1127)_8 + (3325)_8 + (503)_8$

Solution

$$\begin{array}{cccc} \textcircled{1} & & \textcircled{1} & \\ 1 & 1 & 2 & 7 \\ 3 & 3 & 2 & 5 \\ + & 5 & 0 & 3 \\ \hline 5 & 1 & 5 & 7 \end{array}$$

2.7 Octal Arithmetic: Subtraction

EXAMPLE 2 - 15

Compute $(5762)_8 - (3231)_8$

Solution

$$\begin{array}{cccc} 5 & 7 & 6 & 2 \\ - & 3 & 2 & 3 & 1 \\ \hline 2 & 5 & 3 & 1 \end{array}$$

2.8 Hexadecimal Arithmetic: Addition

EXAMPLE 2 - 16

Compute $(DF)_{16} + (AC)_{16}$

Solution

$$\begin{array}{cccc} & \textcircled{1} & & \\ & D & F & \\ + & A & C & \\ \hline 1 & 8 & B & \end{array}$$

EXAMPLE 2 - 17

Compute $(58)_{16} + (22)_{16}$

Solution

$$\begin{array}{ccc} & 5 & 8 \\ + & 2 & 2 \\ \hline & 7 & A \end{array}$$

2.9 Hexadecimal Arithmetic: Subtraction

EXAMPLE 2 - 18

Compute $(84)_{16} - (2A)_{16}$

Solution

$$\begin{array}{ccc} & {}^{7}\not{8} & {}^{20}\not{4} \\ - & 2 & A \\ \hline & 5 & A \end{array}$$

2.10 Signed Binary Numbers

We have considered only *unsigned* binary numbers that represent *positive* quantities. We will often want to represent both *positive* and *negative* numbers, requiring a different binary number system. Several schemes exist to represent *signed binary numbers*; the two most widely employed are called *sign/magnitude* and *two's complement*.

2.11 Sign/Magnitude Numbers

Sign/magnitude numbers are intuitively appealing because they match our custom of writing negative numbers with a *minus* sign followed by the *magnitude*. An *N*-bit sign/magnitude number uses the most significant bit as the *sign* and the remaining *N* - 1 bits as the *magnitude* (absolute value). A sign bit of **0** indicates *positive* and a sign bit of **1** indicates *negative*.

EXAMPLE 2 - 19

Write 5 and −5 as 4-bit sign/magnitude numbers.

Solution

Both numbers have a magnitude of $5_{10} = 101_2$. Thus,

$$5_{10} = 0101_2, \quad -5_{10} = 1101_2$$

Unfortunately, ordinary binary *addition* does not work for sign/magnitude numbers. For example, using ordinary addition on $\mathbf{-5_{10} + 5_{10}}$ gives $1101_2 + 0101_2 = 10010_2$, which is *nonsense*.

An *N*-bit sign/magnitude number spans the range $[-2^{N-1} + 1, 2^{N-1} - 1]$. Sign/magnitude numbers are slightly odd in that both $+0$ and -0 exist. Both indicate *zero*. As you may expect, it can be troublesome to have two different

representations for the same number.

2.12 Complements and Representation of Negative Numbers

Digital circuits perform binary arithmetic operations. It is possible to use the circuits designed for binary addition to perform binary subtraction. Only we must change the problem of subtraction into an equivalent addition problem. Up to this point we have been working with *unsigned* positive numbers. The most common methods for representing both positive and negative numbers are *sign* and *magnitude*, 2's complement, and 1's complement form of the binary numbers. In each of these methods, the *leftmost* bit of a number is **0** for *positive* numbers and **1** for *negative* numbers.

As discussed below, if n bits are used to represent numbers, then the *sign* and *magnitude* and 1's complement methods represent numbers in the range $-(2^{(n-1)} - 1)$ to $+(2^{(n-1)} - 1)$ and both have two representations for **0**, a positive **0** and a negative **0**. In 2's complement, numbers in the range $-2^{(n-1)}$ to $+(2^{(n-1)} - 1)$ are represented and there is only a positive **0**. If an operation, such as addition or subtraction, is performed on two numbers and the result is outside the range of representation, then we say that an *overflow* has occurred. For $n = 4$, **0**011 represents **+3** and **1**011 represents **−3**. Note that **1**000 represents *minus* **0**.

There are *two* types of complements for each base-r system:

1. Diminished radix complements.
2. Radix complement.

2.12.1 Radix Complement Representation

The r's complement of an n-digit number N in base r is defined as $r^n - N$ for

$N \neq 0$ and as 0 for $N = 0$. Comparing with the $(r - 1)$'s complement, we note that the r's complement is obtained by adding 1 to the $(r - 1)$'s complement, since $r^n - N = [(r^n - 1) - N] + 1$.

$$\bar{N} = r^n - N$$

For examples, find the 10's complement for the following decimal numbers.

$$\bar{N} = 10^6 - 012398_{10}$$
$$= 987602_{10}$$
$$\bar{N} = 10^3 - 320.52_{10}$$
$$= 679.48_{10}$$

Find the 2's complement for the following binary numbers.

$$\bar{N} = 2^7 - 1101100_2$$
$$= 10000000_2 - 1101100_2$$
$$= 0010100_2$$
$$\bar{N} = 2^3 - 101.1_2$$
$$= 1000_2 - 101.1_2$$
$$= 10.1_2$$

2.12.2 Diminished Radix Complement Representation

The *base* is called diminished radix complement if it decreases by one from the original *base*. For example, the *radix* for the binary system is **1**, and the base for the decimal system is 9. The diminished radix complement is denoted by the symbol $\bar{\bar{N}}$. It is calculated according to the following relationship:

$$\bar{\bar{N}} = r^n - N - r^{-m}$$

Where: *N* is the number that is represented by a base number system *r*, and this number consists of *n* integer digits and *m* fractional digits.

For examples,

$$\bar{N} = 10^3 - 320.52_{10} - 10^{-2}$$

$$= 1000_{10} - 320.52_{10} - 0.01_{10}$$

$$= 679.47_{10}$$

$$\bar{N} = 2^3 - 101.1_2 - 2^{-2}$$

$$= 1000_2 - 101.1_2 - 0.1_2 \quad = 10.0_2$$

2.13 Subtraction Using 1's Complement

Binary subtraction can be performed by adding the *1's complement* of the subtrahend to the minuend. If a *carry* is generated, **remove** the carry, **add** it to the result. This *carry* is called the *end-around carry*. Now if the subtrahend is larger than the minuend, then ***no carry*** is generated. The answer obtained is 1's complement of the true result and opposite in sign. We compute *1's complement* by replacing all 0s with 1s and all 1s with 0s.

EXAMPLE 2 - 20

Compute $1010100_2 - 1000011_2$ using 1's complement.

Solution Suppose $X = 1010100_2$ and $Y = 1000011_2$

Step1: Take the 1's complement of *Y.*

$$\begin{array}{rrrrrrrr} & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ & 0 & 1 & 1 & 1 & 1 & 0 & 0 \end{array}$$

Step2: Add 1's complement of Y to the X.

$$\begin{array}{rrrrrrrr} & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ + & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ \hline 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{array}$$

Step3: Remove the end carry and add 1 to the result.

$$\begin{array}{rrrrrrrr} & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ + & & & & & & & 1 \\ \hline & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{array}$$

EXAMPLE 2 - 21

Compute $1000011_2 - 1010100_2$ using 1's complement.

Solution Suppose $X = 1000011_2$ and $Y = 1010100_2$.

Step1: Take the 1's complement of Y.

$$\begin{array}{rrrrrrrr} & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \end{array}$$

Step2: Add 1's complement of Y to the X.

$$\begin{array}{rrrrrrrr} & & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ + & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ \hline & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \end{array}$$

Step3: There is No end carry. So, Take the 1's complement of the sum result.

$$\begin{array}{rrrrrrrr} \mathbf{1} & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ \\ \mathbf{1} & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{array}$$

2.14 Subtraction Using 2's Complement

Binary subtraction can be performed by adding the *2's complement* of the subtrahend to the minuend. If a *carry* is generated, **discard** the carry. Now if the subtrahend is larger than the minuend, then *no carry* is generated. The answer obtained is in 2's complement and is *negative*. To get a true answer take the 2's complement of the number and change the sign. The advantage of the 2's complement method is that the end-around carry operation present in the 1's complement method is not present here. We compute *2's complement* by finding *2's complement* to the binary number than add 1 to the 2's complement of the number.

EXAMPLE 2 - 22

Compute $1010100_2 - 1000011_2$ using 2's complement.

Solution Suppose $X = 1010100_2$ and $Y = 1000011_2$

Step1: Take the 2's complement of Y.

$$\begin{array}{ccccccc} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 \end{array}$$

$$\begin{array}{ccccccc} & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ + & & & & & & & 1 \\ \hline & 0 & 1 & 1 & 1 & 1 & 0 & 1 \end{array}$$

Step2: Add 2's complement of Y to the X.

$$\begin{array}{ccccccc} & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ + & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\ \hline \mathbf{1} & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{array}$$

Step3: Remove the end carry.

$$\mathbf{0\ 0\ 1\ 0\ 0\ 0\ 1}$$

EXAMPLE 2 - 23

Compute $1000011_2 - 1010100_2$ using 2's complement.

Solution Suppose $X = 1000011_2$ and $Y = 1010100_2$.

Step1: Take the 2's complement of Y.

	1	0	1	0	1	0	0
1	0	1	0	1	0	1	1

	1	0	1	0	1	0	1	1
+								**1**
	1	0	1	0	1	1	0	0

Step2: Add 2's complement of Y to the X.

		1	0	0	0	0	1	1
+	**1**	0	1	0	1	1	0	0
	1	1	1	0	1	1	1	1

Step3: There is No end carry. So, Take the 2's complement of the sum result.

1	0	0	1	0	0	0	0
1	0	0	1	0	0	0	1

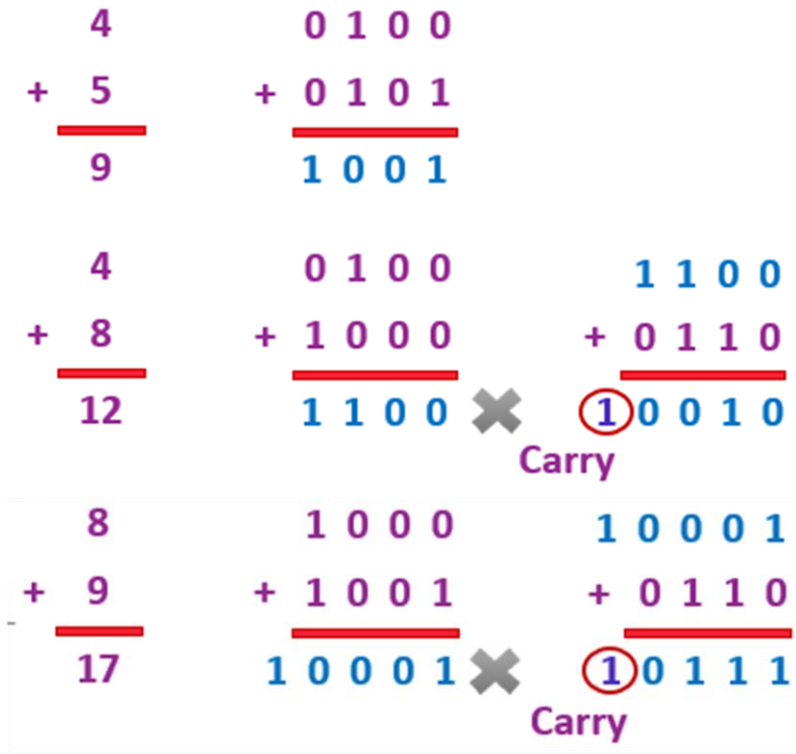
2.15 Binary Coded Decimal (BCD) Addition

At first the given number are to be added using the rule of binary. Consider the addition of two decimal digits in BCD. Each digit does not exceed 9. In second step we must judge the result of addition. Here two cases are shown to describe the rules of BCD Addition. In case 1 the result of addition of two binary number is greater than 9, which is not valid for BCD number. But the result of addition in case 2 is less than 9, which is valid for BCD numbers.

If the four-bit result of addition is greater than 9 and if a carry bit is present in the result then it is invalid and we have to add 6 whose binary equivalent is $(0110)_2$ to the result of addition. Then the resultant that we would get will be a valid binary coded number. In case 1 the result was $(1111)_2$, which is greater than 9 so we must add 6 or $(0110)_2$ to it.

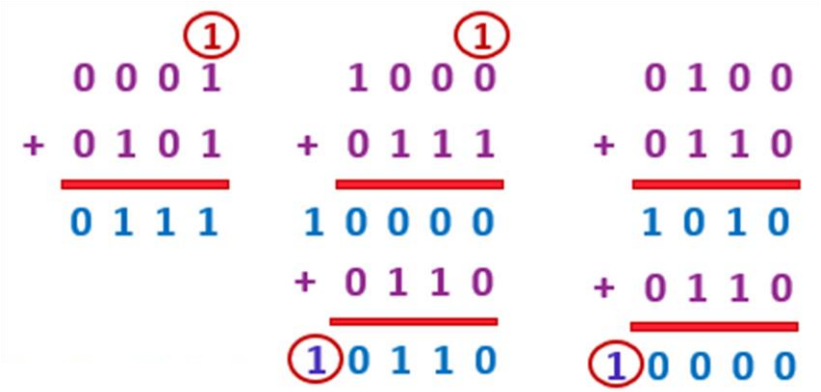
EXAMPLE 2 - 24

Compute:



EXAMPLE 2 - 25

Compute $184_{10} + 576_{10}$ in BCD.

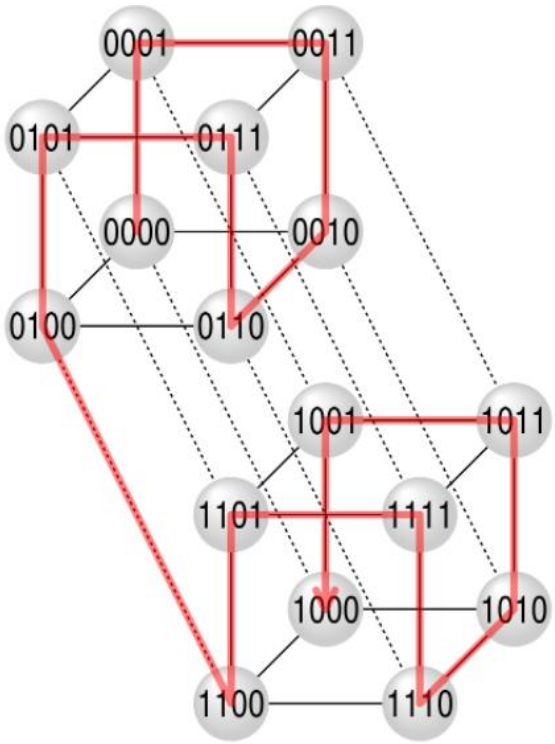


2.16 Gray Codes

The *reflected binary code* (RBC), also known as *reflected binary* (RB) *or Gray code* after Frank Gray, is an ordering of the binary numeral system such that two successive values differ in only one bit (binary digit). *Gray code* is an ordering of the binary numeral system such that two successive values differ in only one bit (binary digit). The advantage is that only one bit in the code group changes in going from one number to the next.

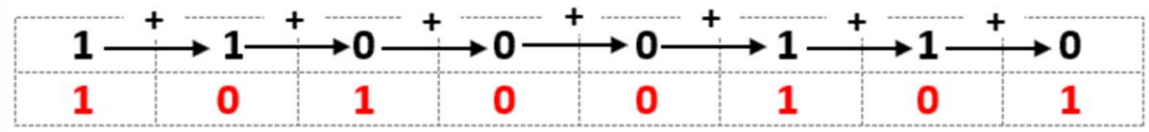
2.16.1 Binary to Gray Code Conversion

- The MSB (left-most) in the Gray code is the same as the corresponding MSB in the binary number.
- Going from left to right, add each adjacent pair of binary code bits to get the next Gray code bit.
- Discard carries.



EXAMPLE 2 - 26

Convert the binary number 11000110_2 to the Gray code.

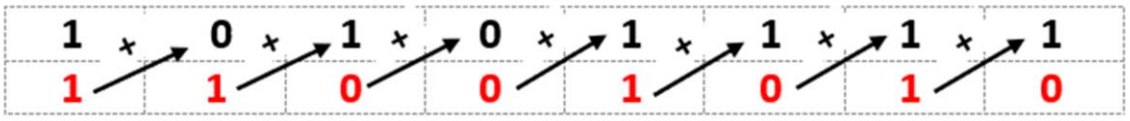


2.16.2 Gray to Binary Code Conversion

- The MSB (left-most) in the binary code is the same as the corresponding bit in the Gray code.
- Add each binary code bit generated to the Gray code bit in the next adjacent position.
- Discard carries.

EXAMPLE 2 – 27

Convert the Gray code 10101111_{gray} to binary.



Problems

1. Add, Subtract, and Multiply in binary

(a) 1111_2 and 1010_2
(b) 1111_2 and 1001_2
(c) 100100_2 and 10110_2
(d) 110110_2 and 11101_2
(e) 110010_2 and 11101_2
(f) 1101001_2 and 110110_2
(g) $(11010.1)_2$ and $(1011.01)_2$
(h) $(11011.101)_2$ and $(1110.11)_2$

2. Subtract in binary

(a) $11110100_2 - 1000111_2$
(b) $10100100_2 - 01110011_2$
(c) $1110110_2 - 111101_2$
(d) $10010011_2 - 01011001_2$
(e) $10010011_2 - 01011001_2$
(f) $11110011_2 - 10011110_2$

3. Divide in binary

(a) $11101001_2 \div 101_2$
(b) $0001101_2 \div 110_2$
(c) $110000001_2 \div 1110_2$
(d) $110000011_2 \div 1011_2$
(e) $1110010_2 \div 1001_2$
(f) $1110100_2 \div 1010_2$

4. Perform the following additions

(a) $25_{16} + 33_{16}$
(b) $43_{16} + 62_{16}$
(c) $A4_{16} + F5_{16}$
(d) $FC_{16} + AE_{16}$

5. Perform the following subtractions

(a) $60_{16} - 39_{16}$
(b) $A5_{16} - 98_{16}$
(c) $F1_{16} - A6_{16}$
(d) $AC_{16} - 10_{16}$

6. Add the following numbers in binary using 2's complement to represent negative numbers. Use a word length of 8 bits (including sign) and indicate if an overflow occurs.

(a) $(-12)_{10} + 13_{10}$
(b) $(-11)_{10} + (-21)_{10}$
(c) $(-110)_{10} + (+84)_{10}$

7. Repeat (a), (b) using 1's complement to represent negative numbers.

8. Determine the 1's complement of each binary number

(a) 100 (b) 111
(c) 1100 (d) 10111011
(e) 10111011 (f) 10101010

9. Determine the 2's complement of each binary number using either method

(a) 11 (b) 110 (c) 1010
(d) 101010 (e) 11001100 (f) 11000111

10. Express each decimal number as an 8-bit number in the 1's complement form

(a) -34_{10} (b) $+57_{10}$ (c) -99_{10} (d) $+115_{10}$

11. Express each decimal number as an 8-bit number in the 2's complement form

(a) $+12_{10}$ (b) -68_{10} (c) $+101_{10}$ (d) -125_{10}

12. Determine the decimal value of each signed binary number in the sign-magnitude form

(a) 10011001_2 (b) $01110100_{\ 2}$ (c) $10111111_{\ 2}$

13. Determine the decimal value of each signed binary number in the 1's complement form

(a) 10011001_2 (b) $01110100_{\ 2}$ (c) $10111111_{\ 2}$

14. Determine the decimal value of each signed binary number in the 2's complement form

(a) 10011001_2 (b) $01110100_{\ 2}$ (c) $10111111_{\ 2}$

15. Convert each of the following decimal numbers to BCD

(a) 98_{10} (b) 170_{10} (c) 2469_{10} (d) 9673_{10}

16. Convert each of the following BCD codes to decimal

(a) 001101010001_{BCD} (b) $1000010001001110110_{BCD}$

17. Add the following BCD numbers

(a) 1001 + 0100 (b) 00010110 + 00010101

(c) 01001000 + 00110100 (d) 1001000001000011 + 0000100100100101

18. Convert binary 101101_2 to Gray code.

19. Convert Gray code 100111_{gray} to binary.

اسم الطالب: .....

رقم الطالب: .....

اسم المقرر: .....

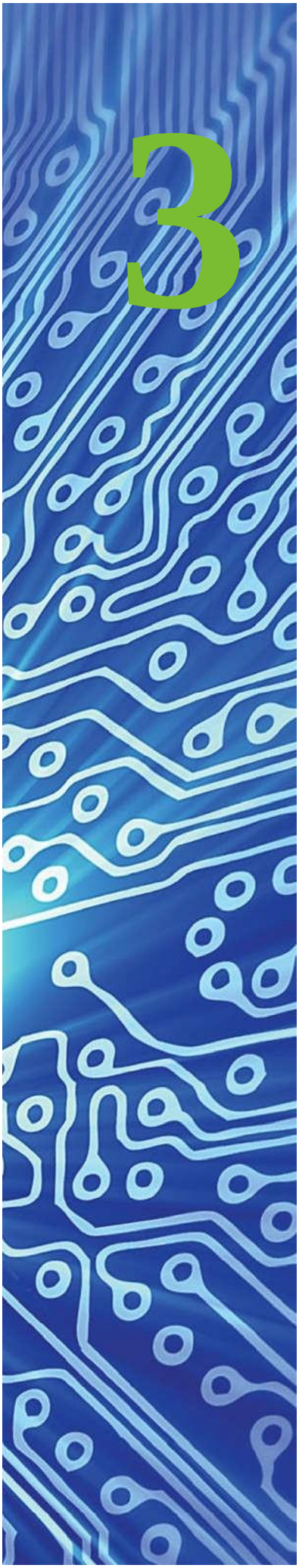
رقم الشيت: .....

الرقم السري: .....

استمارة التقويم المستمر

Question No.	Degree	Question No.	Degree	Signature
1.		2.		
3.		4.		
5.		6.		
7.		8.		
9.		10.		
11.		12.		
13.		14.		
15.		16.		
17.		18.		
19.		20.		

20



3 Basic Logic Gates

OUTLINE

3-1 AND Gate.

3-2 OR Gate.

3-3 Inverter Gate.

3-4 NAND Gate.

3-5 NOR Gate.

3.1 INTRODUCTION

Logic gates are the basic building blocks for forming digital electronic circuitry. A logic gate has one output terminal and one or more input terminals. Its output will be HIGH (1) or LOW (0) depending on the digital level(s) at the input terminal(s). Using logic gates, we can design digital systems that will evaluate digital input levels and produce a specific output response based on that logic circuit design. The five basic logic gates are the AND, OR, NAND, NOR, and inverter.

3.2 The AND Gate

An *AND* gate can have *two* or *more* inputs and performs what is known as logical *multiplication*. The operation of the AND gate is simple and is defined as follows: The output, X, is HIGH if input A AND input B are both HIGH. In other words, if $A = 1$ AND $B = 1$, then $X = 1$. If either A or B or both are LOW, the output will be LOW.

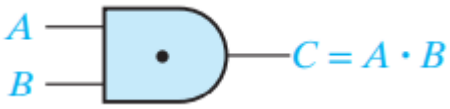


Figure 3–1 Two-input AND gate symbol.

TABLE 3–1 Truth Table for a Two-Input AND Gate

Inputs		Output
A	B	$X = AB$
0	0	0
0	1	0
1	0	0
1	1	1

The best way to illustrate how the output level of a gate responds to all the possible input-level combinations is with a ***truth table***.

From the truth table, we can see that the output at *X* is HIGH only when both *A* AND *B* are HIGH. Although this looks like *binary multiplication*, it is NOT, because 0 and 1 are Boolean constants rather than binary numbers.

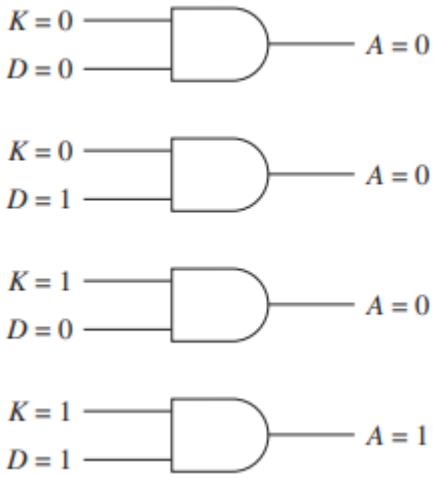


Figure 3–2 All combinations of two-input AND gate symbol.

The *Boolean equation* for the AND function can more simply be written as:

$$X = A \cdot B$$

or just *X* =*AB* (which is read as *X* equals *A* AND *B*). Boolean equations will be used throughout the rest of the book to depict algebraically the operation of a logic gate or a combination of logic gates.

AND gates can have more than two inputs. Figure 3–3 shows a four-input and three-input.

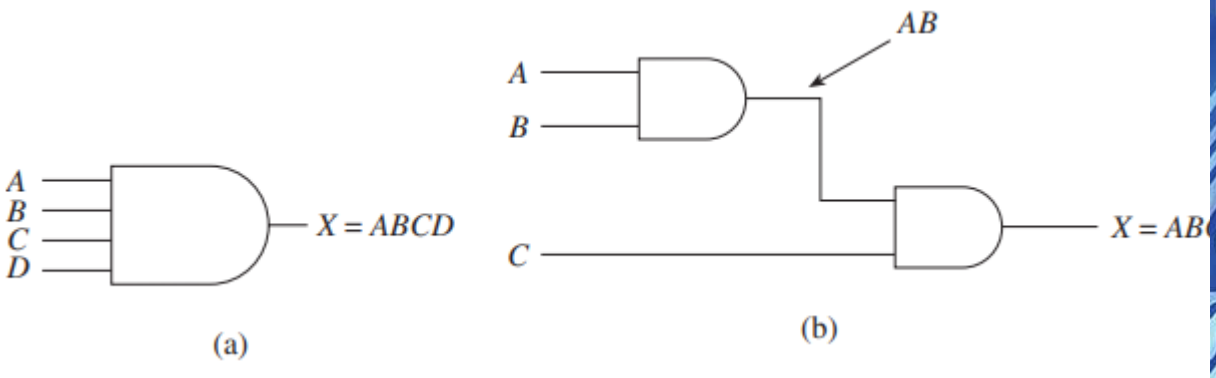


Figure 3–3 Multiple-input AND gate symbols.

TABLE 3–2 Truth Table for a Four-Input AND Gate

A	*B*	*C*	*D*	*X*
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

The output at *X* is HIGH only if *all* inputs are HIGH.

3.3 The OR Gate

The OR gate also has *two* or *more* inputs and a *single* output. The symbol for a two input OR gate is shown in Figure 3–4. The operation of the two-input OR gate is defined as follows: The output at *X* will be HIGH whenever input *A* OR input *B* is HIGH or both are HIGH. As a Boolean equation, this can be written:

$$X = A + B$$

Notice the use of the symbol to represent the OR function.

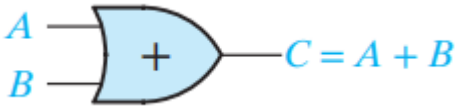


Figure 3–4 Two-input OR gate symbol.

TABLE 3–3 Truth Table for a Two-Input OR Gate

Inputs		Output
A	*B*	$X = A + B$
0	0	0
0	1	1
1	0	1
1	1	1

From the truth table you can see that *X* is 1 whenever *A* OR *B* is 1 or if both *A* and *B* are 1.

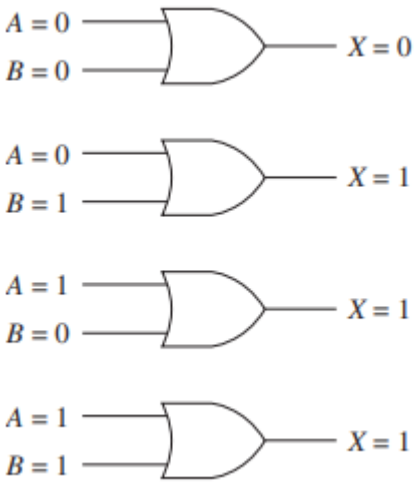


Figure 3–5 All combinations of two-input OR gate symbol.

OR gates can have more than two inputs. Figure 3–6 shows a three-input.

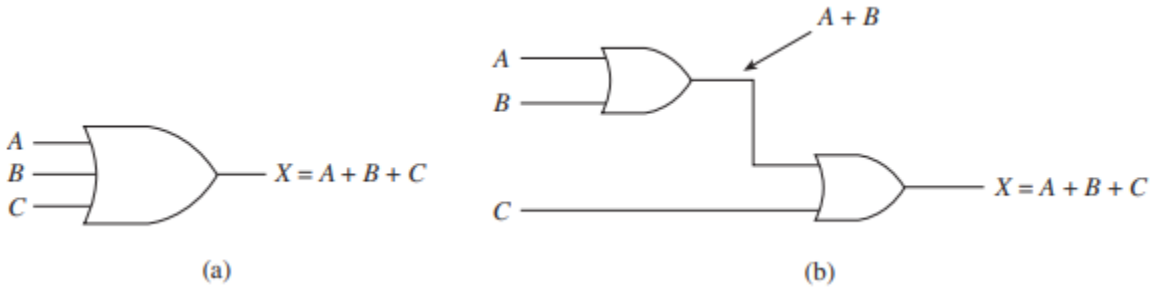


Figure 3–6 Multiple-input OR gate symbols.

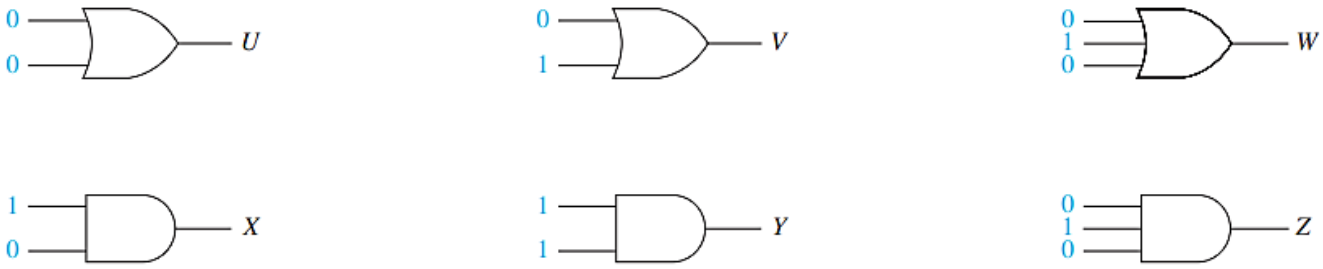
TABLE 3–4 Truth Table for a Three-Input OR Gate

A	*B*	*C*	*X*
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

The output at *X* is HIGH if *any* input is HIGH.

EXAMPLE 3–1

Determine the output at *U*, *V*, *W*, *X*, *Y*, and *Z* in Figure 3–9.



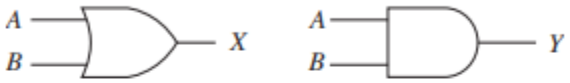
Solution:

$U = 0$ (0 OR 0 = 0)
$V = 1$ (0 OR 1 = 1)
$W = 1$ (0 OR 1 OR 0 = 1)

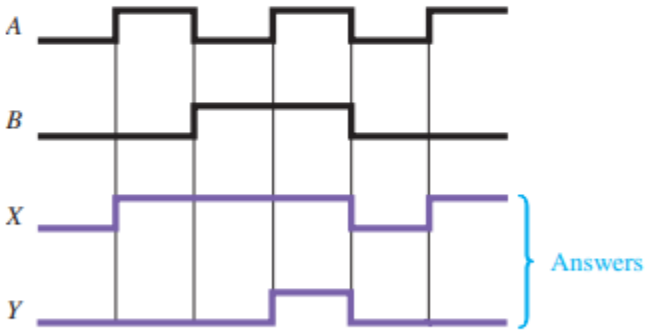
$X = 0$ (1 AND 0 = 0)
$Y = 1$ (1 AND 1 = 1)
$Z = 0$ (0 AND 1 AND 0 = 0)

EXAMPLE 3–2

Sketch the output waveform at *X* and *Y* for the two-input OR gate and AND gate shown in Figure , with the given *A* and *B* input waveforms in



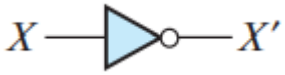
Solution:



(b)

3.4 The Inverter Gate

The *inverter* is used to *complement*, or invert, a digital signal. It has a *single* input and a *single* output. If a HIGH level (1) comes in, it produces a Low-level (0) output. If a LOW level (0) comes in, it produces a High-level (1) output. The symbol and truth table for the inverter gate are shown in Figure 3–7. (Note: The circle is the part of the symbol that indicates inversion. The inversion circle will be used on other gates in upcoming sections.)



Input A	Output X
0	1
1	0

Figure 3–7 Inverter symbol and truth table.

When the input is HIGH, the output is LOW, and when the input is LOW, the output is HIGH. The Boolean equation for an inverter is written:

$$\boldsymbol{X = A'} \text{ or } \boldsymbol{X = \overline{A}}$$

The bar over the *A* is an *inversion* bar, used to signify the complement. The *inverter* is sometimes referred to as the NOT gate.

3.5 The NAND Gate

The operation of the NAND gate is the same as the AND gate except that its output is inverted. You can think of a NAND gate as an AND gate with an inverter at its output. The symbol for a NAND gate is made from an AND gate with the inversion circle (bubble) at its output, as shown in Figure 3–8.

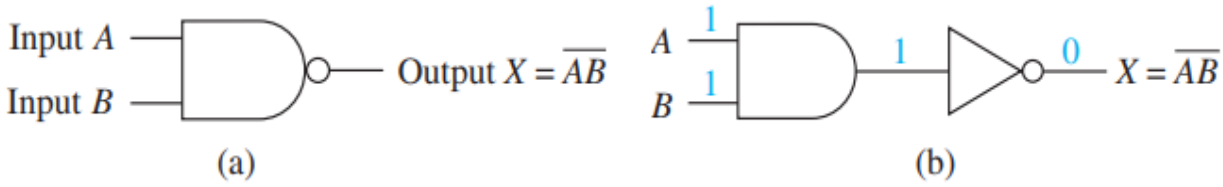


Figure 3–8 Two-input NAND gate symbol.

From the truth table you can see that *X* is 0 when *A* AND *B* is 0. Figure 3–9 shows the output results for all possible input combinations applied to a 7400 quad NAND.

TABLE 3–6 Two-Input NAND Gate Truth Table

A	*B*	$X = \overline{AB}$
0	0	1
0	1	1
1	0	1
1	1	0

Output is always HIGH unless both inputs are HIGH.

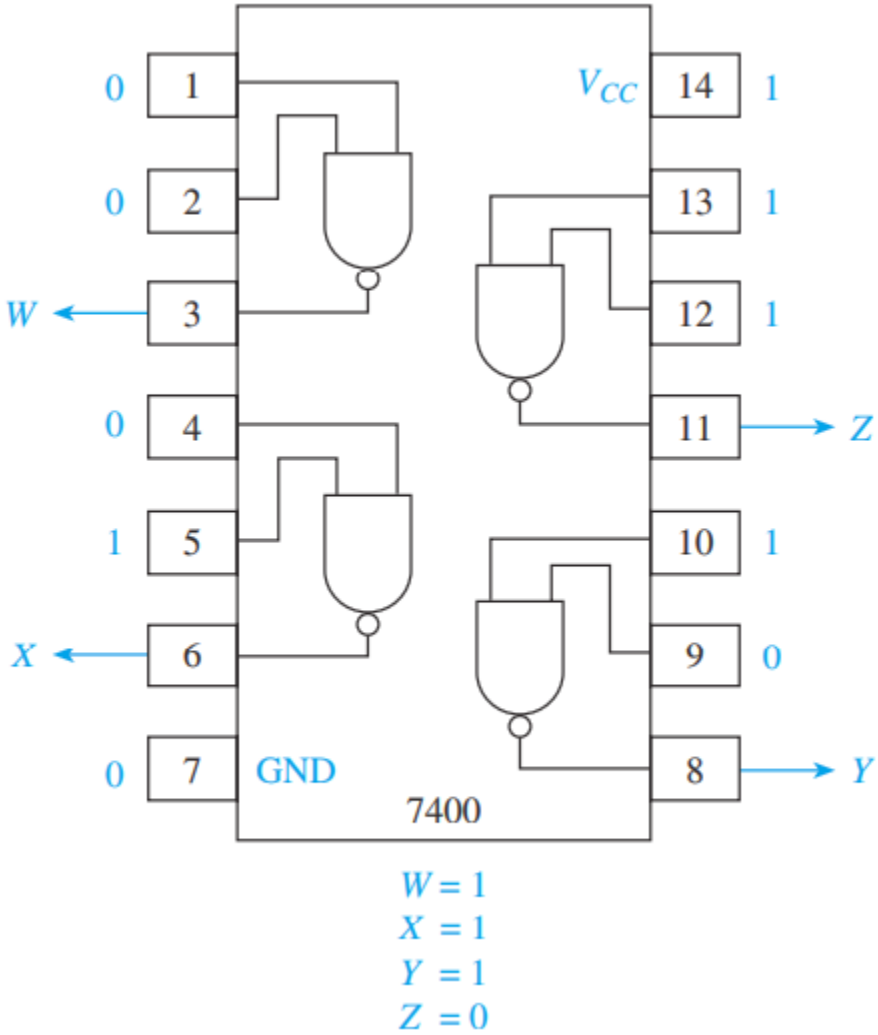


Figure 3–9 Inputs and outputs of a 7400 quad NAND IC.

The *Boolean equation* for the NAND gate is written $X = \overline{AB}$. The inversion bar is drawn over (*A* and *B*), meaning that the output of the NAND is the complement of (*A* and *B*) [NOT (*A* and *B*)]. Because we are inverting the

output, the truth table outputs in Table 3–6 will be the complement of the AND gate truth table outputs. The easy way to construct the truth table is to think of how an AND gate would respond to the inputs and then invert your answer. From Table 3–6, we can see that the output is LOW when both inputs *A* and *B* are HIGH (just the opposite of an AND gate). Also, the output is HIGH whenever either input is LOW.

TABLE 3–7 Truth Table for a Three-Input NAND Gate

A	*B*	*C*	*X*
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

3.6 The NOR Gate

The operation of the NOR gate is the same as that of the OR gate except that its output is inverted. You can think of a NOR gate as an OR gate with an inverter at its output. The symbol for a NOR gate and its equivalent OR–INVERT symbol are shown in Figure 3–10.

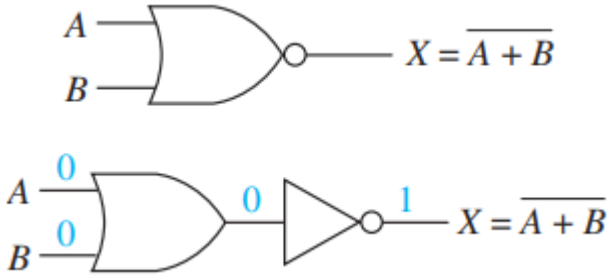


Figure 3–10 Two-input NOR gate symbol

TABLE 3–8 Truth Table for a NOR Gate

A	*B*	$X = \overline{A + B}$
0	0	1
0	1	0
1	0	0
1	1	0

Output is always LOW unless both inputs are LOW.

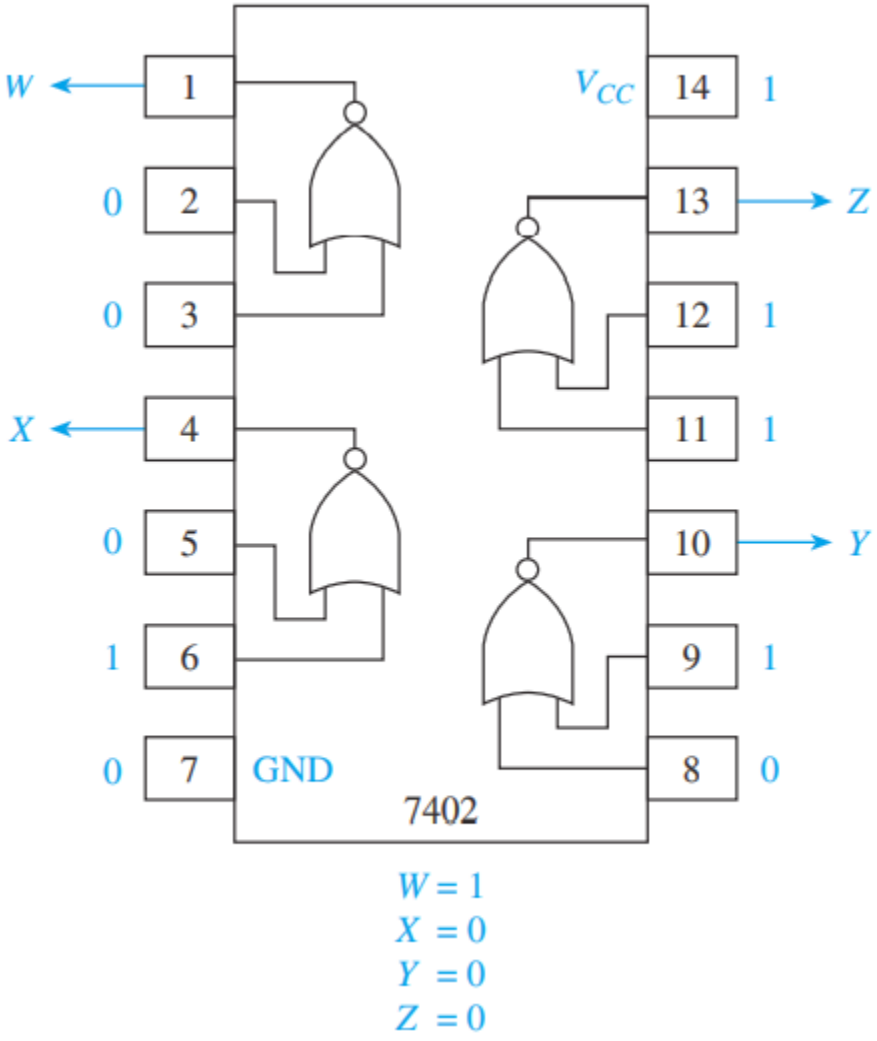


Figure 3–11 Inputs and outputs of a 7402 quad NOR IC.

The *Boolean equation* for the NOR function is $X = \overline{A + B}$. The equation is stated *X* equals not (*A* or *B*). In other words, *X* is LOW if *A* or *B* is HIGH. The truth table for a NOR gate is given in Table 3–8. Notice that the output column is the complement of the OR gate truth table output column.

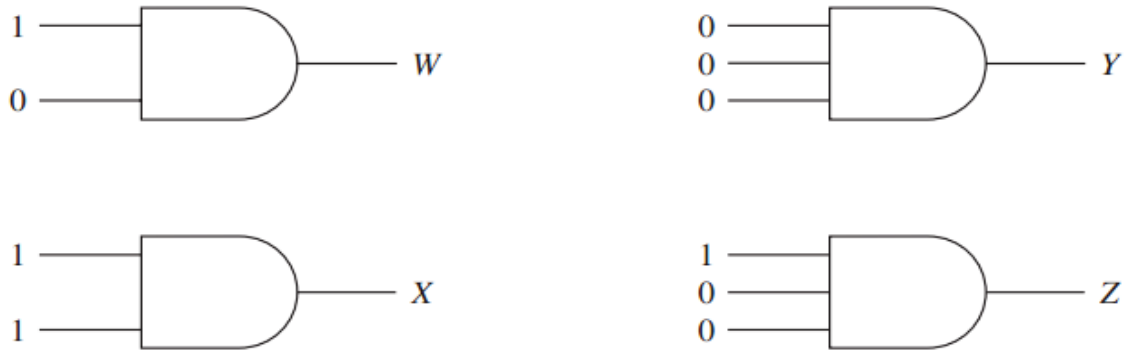
Problems

1. **Build the truth table for:**

   (a) a three-input AND gate.

   (b) a four-input AND gate.

2. **Determine the logic level at *W*, *X*, *Y* and *Z* in the following figure:**



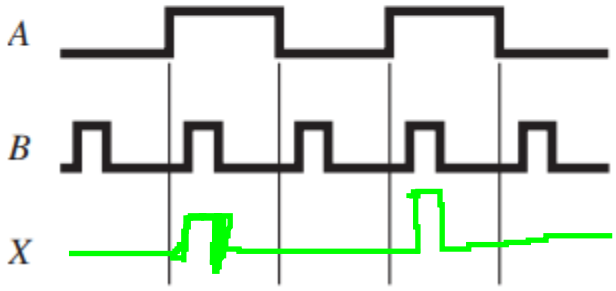
3. **Write the Boolean equation for:**

   (a) A three-input AND gate.

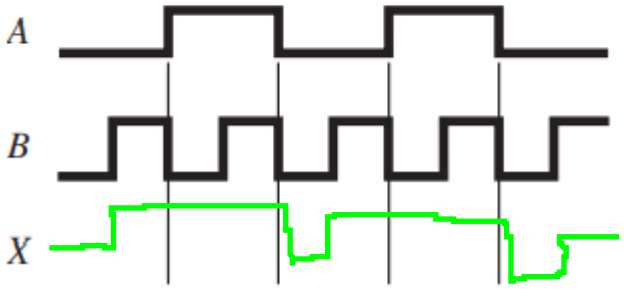
   (b) A four-input AND gate.

   (c) A three-input OR gate.

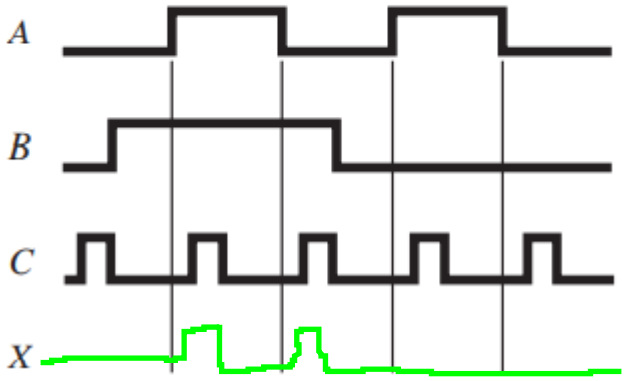
4. **Sketch the output waveform at *X* for the two-input AND gates shown in the following figure:**



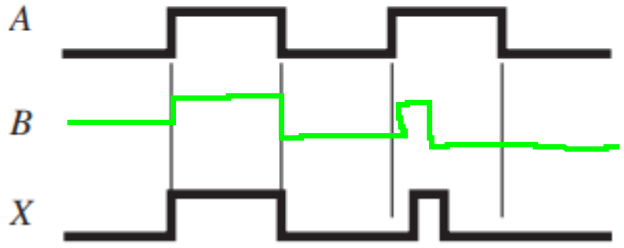
5. Sketch the output waveform at *X* for the two-input OR gates shown in the following figure:



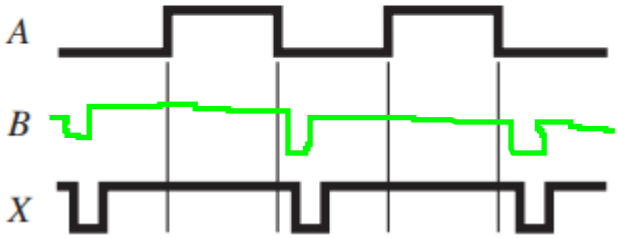
6. Sketch the output waveform at *X* for the three-input AND gates shown in the following figure:



7. The input waveform at *A* is given for the two-input AND gates shown in the following figure. Sketch the input waveform at *B* that will produce the output at *X*.

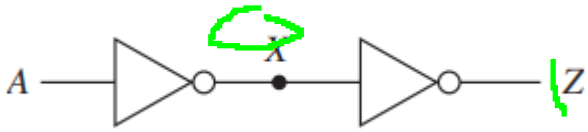


(a)



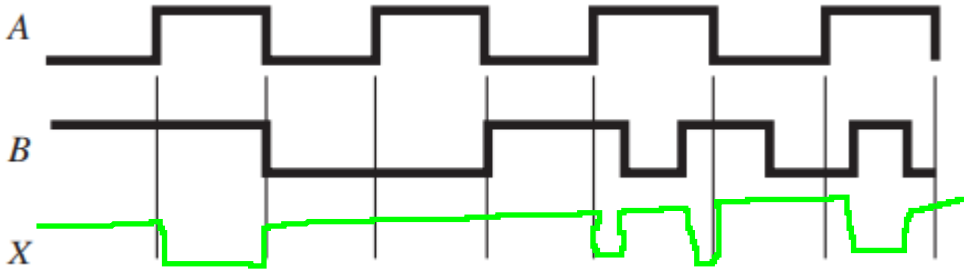
(b)

8. Write the Boolean equation at *X* and *Z*. If *A* = 1, what is the value of *X*? What is the value of *Z*?

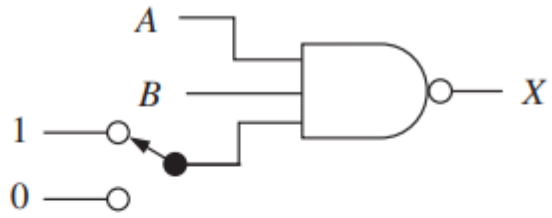


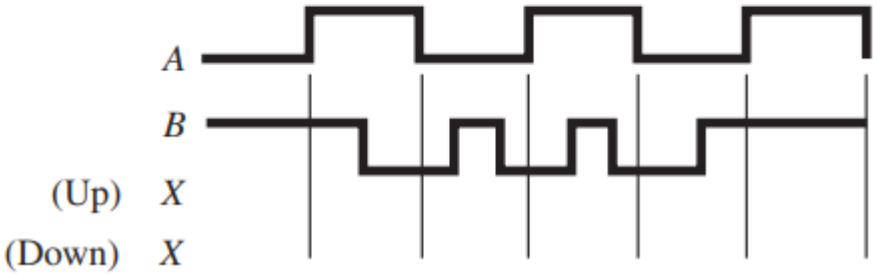
9. Write the Boolean equation at *Y* and build a truth table for it.

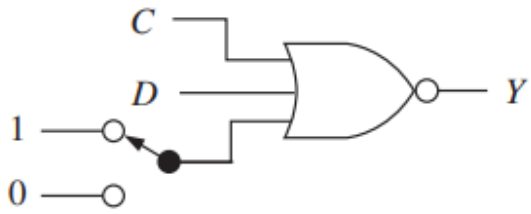
10. Sketch the output waveforms for *X*, given the input waveforms shown in the following figure. ($X = \overline{AB}$)



11. Sketch the waveforms at *X* and *Y* with the switches in the down (0) position. Repeat with the switches in the up (1) position.







اسم الطالب: .....
رقم الطالب: .....
اسم المقرر: .....
رقم الشيت: .....

الرقم السري: .....

استمارة التقويم المستمر

Question No.	Degree	Question No.	Degree	Signature
1.		2.		
3.		4.		
5.		6.		
7.		8.		
9.		10.		
11.				

11

4

Boolean Algebra and Reduction Techniques

OUTLINE

4-1 Combinational Logic.

4-2 Boolean Algebra Laws and Rules.

4-3 Simplification of Combinational Logic Circuits Using Boolean Algebra.

4-4 De Morgan's Theorem.

4-5 Simplification Theorems.

4.1 INTRODUCTION

Generally, we will find that the simple gate functions AND, OR, NAND, NOR, and INVERT are not enough by themselves to implement the complex requirements of digital systems. The basic gates will be used as the building blocks for the more complex logic that is implemented by using combinations of gates called *combinational logic*.

4.2 Combinational Logic

Combinational logic employs the use of two or more of the basic logic gates to form a more useful, *complex function*. For example, let us design the logic for an *automobile warning buzzer* using combinational logic. The criterion for the activation of the warning buzzer is as follows: The buzzer activates if the headlights are on, *and* the driver's door is opened *or* if the key is in the ignition *and* the door is opened.

The logic function for the automobile warning buzzer is illustrated symbolically in Figure 4–1. The figure illustrates a combination of logic functions that can be written as a Boolean equation in the form:

$$B = K \text{ and } D \text{ or } H \text{ and } D$$

which is also written as:

$$B = KD + HD$$

This equation can be stated as B is HIGH if K and D are HIGH or if H and D are HIGH.

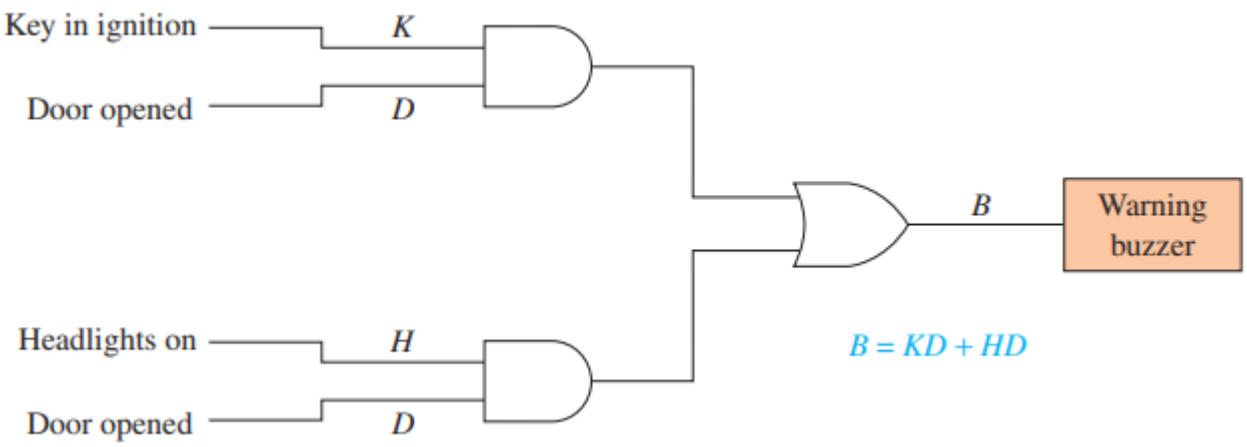


Figure 4–1 Combinational logic requirements for an automobile warning buzzer.

When we think about the operation of the warning buzzer, we may realize that it is activated whenever the door is opened and either the key is in the ignition, or the headlights are on. If we can realize that we have just performed our first ***Boolean reduction*** using Boolean algebra. (The systematic reduction of logic circuits is performed using Boolean algebra, named after the nineteenth-century mathematician George Boole.)

The new Boolean equation becomes $B = D$ and (K or H), also written as $B = D\ (K + H)$ (Notice the use of parentheses. Without them, the equation would imply that the buzzer activates if the door is opened with the key in the ignition or any time the headlights are on, which is invalid $B \neq DK + H$. Parentheses are always required when an OR gate is input to an AND gate.)

The new equation represents the same logic operation, but is a simplified implementation, because it requires only two logic gates, as shown in Figure 4–2.

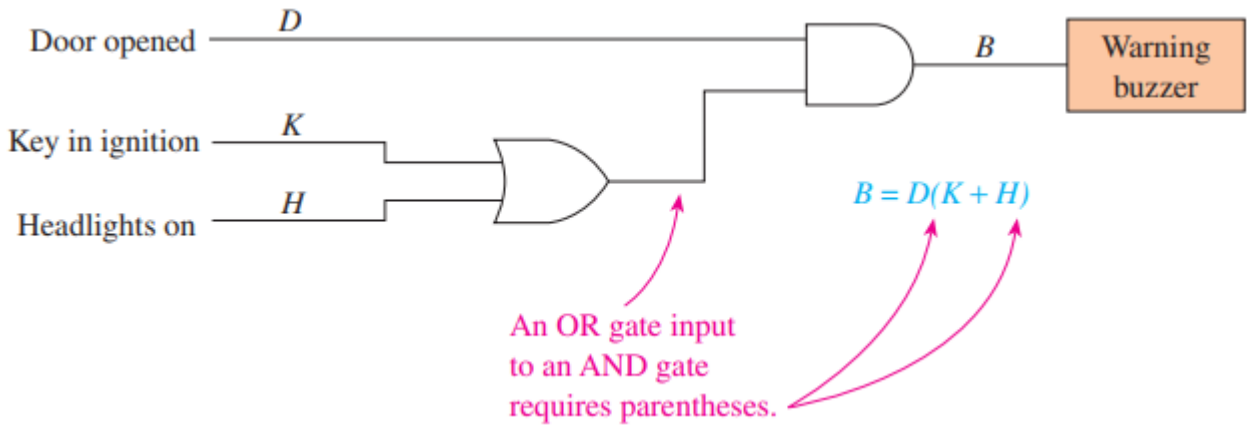
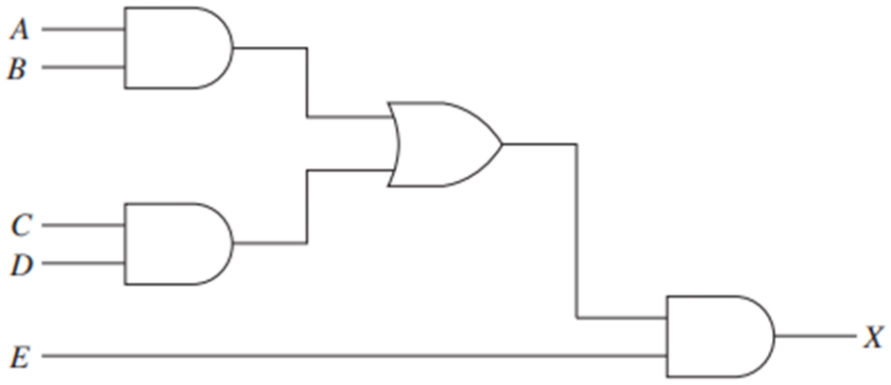


Figure 4–2 Reduced logic circuit for the automobile buzzer.

EXAMPLE 4 - 1

Write the Boolean equation for the logic circuit shown in the following figure:



Solution

$$\boldsymbol{X = (AB + CD)\,E}$$

4.3 Boolean Algebra Laws and Rules

Boolean algebra uses many of the same laws as those of ordinary algebra. The **OR** function ($X = A + B$) is Boolean *addition*, and the **AND** function ($X = AB$) is Boolean *multiplication*. The following *three laws* are the same for Boolean algebra as they are for ordinary algebra:

1. *Commutative law of addition and multiplication*:

$$A + B = B + A$$

$$AB = BA$$

These laws mean that the order of ORing or ANDing does not matter.

2. *Associative law of addition and multiplication:*

$$A + (B + C) = (A + B) + C$$

$$A\,(BC) = (AB)\,C$$

These laws mean that the grouping of several variables ORed or ANDed together does not matter.

3. *Distributive law:*

$$A\,(B + C) = AB + AC$$

$$(A + B)\,(C + D) = AC + AD + BC + BD$$

These laws show methods for expanding an equation containing ORs and ANDs.

You may wonder when you will need to use one of the laws. Later in this chapter, you will see that by using these laws to rearrange Boolean equations, you will be able to change some combinational logic circuits to simpler equivalent circuits using fewer gates. You can gain a better understanding of the application of these laws by studying Figures 4-3 to 4-8.

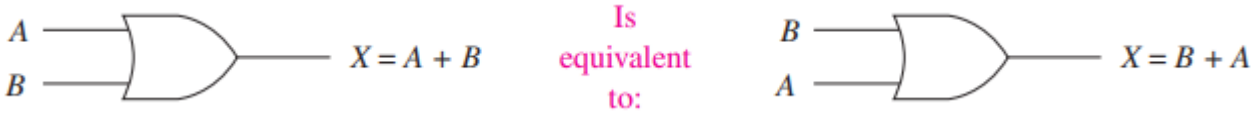


Figure 4–3 Using the commutative law of addition to rearrange an OR gate

Figure 4–4 Using the commutative law of multiplication to rearrange an AND gate.

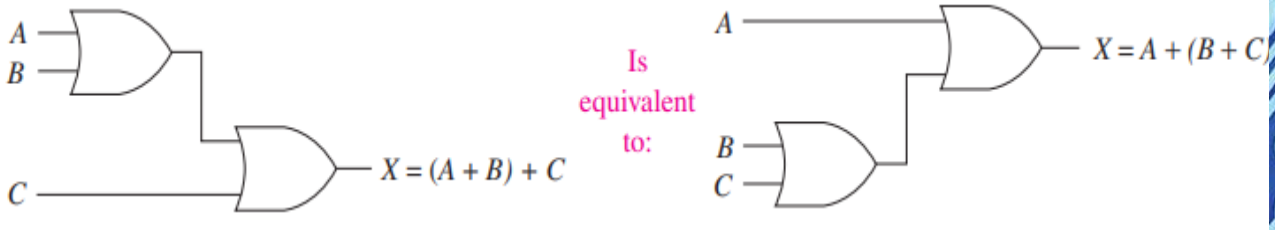


Figure 4–5 Using the associative law of addition to rearrange the grouping of OR gates.

Figure 4–6 Using the associative law of multiplication to rearrange the grouping of AND gates.

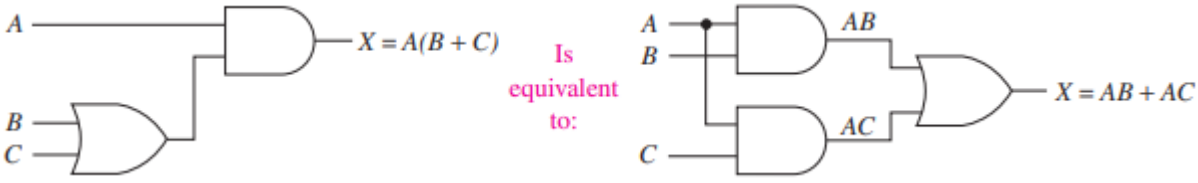


Figure 4–7 Using the distributive law to form an equivalent circuit.

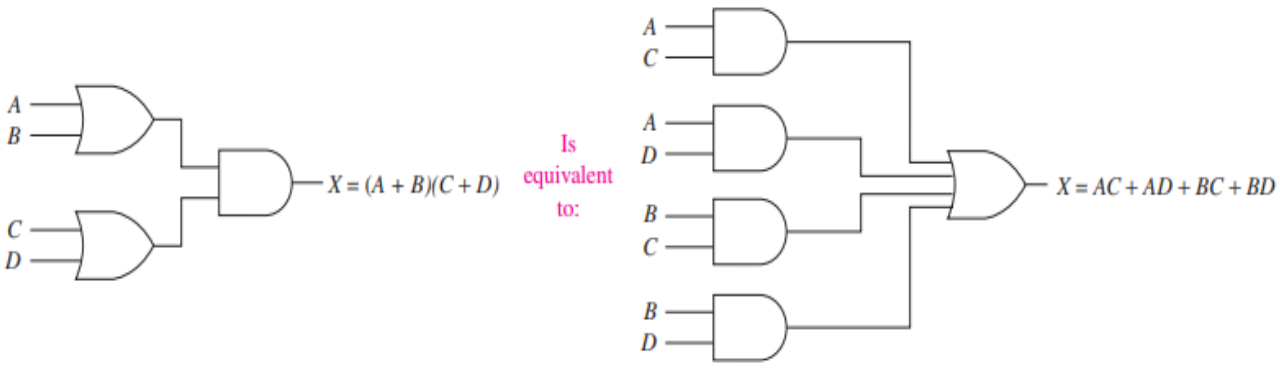
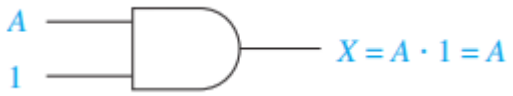


Figure 4–8 Using the distributive law to form an equivalent circuit (FOIL method).

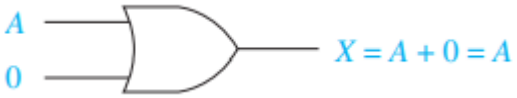
The other *ten rules* can be derived using common sense and knowing basic gate operation.

Rule (1): $A \cdot 0 = 0$

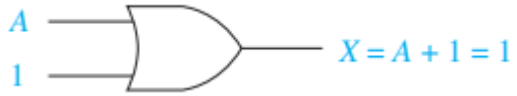
Rule (2): $A \cdot 1 = A$



Rule (3): $A + 0 = A$



Rule (4): $A + 1 = 1$

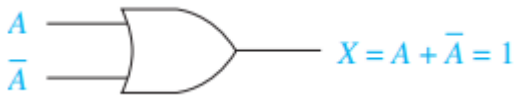


Rule (5): $A \cdot A = A$

Rule (6): $A + A = A$

Rule (7): $A.\overline{A} = 0$

Rule (8): $A + \overline{A} = 1$



Rule (9): $\overline{\overline{A}} = A$

Rule (10): $A + \overline{A}B = A + B$

$\overline{A} + AB = \overline{A} + B$

TABLE 1 Using Truth Tables to Prove the Equations in Rule 10

A	B	$A + \overline{A}B$	$A + B$	A	B	$\overline{A} + AB$	$\overline{A} + B$
0	0	0	0	0	0	1	1
0	1	1	1	0	1	1	1
1	0	1	1	1	0	0	0
1	1	1	1	1	1	1	1
		↑	↑			↑	↑
		Equivalent outputs				Equivalent outputs	

TABLE 2 Boolean Laws and Rules for the Reduction of Combinational Logic Circuits

Laws

1 $A + B = B + A$
$AB = BA$

2 $A + (B + C) = (A + B) + C$
$A(BC) = (AB)\,C$

3 $A(B + C) = AB + AC$
$(A + B)(C + D) = AC + AD + BC + BD$

Rules

1 $A \cdot 0 = 0$

2 $A \cdot 1 = A$

3 $A + 0 = A$

4 $A + 1 = 1$

5 $A \cdot A = A$

6 $A + A = A$

7 $A \cdot \overline{A} = 0$

8 $A + \overline{A} = 1$

9 $\overline{\overline{A}} = A$

10 (a) $A + \overline{A}B = A + B$
(b) $\overline{A} + AB = \overline{A} + B$

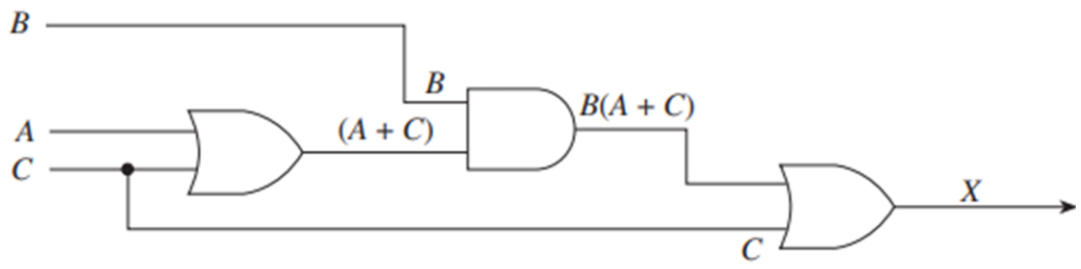
4.4 Simplification of Combinational Logic Circuits Using Boolean Algebra

A designer will start with simple logic gate requirements but add more and more complex gating, making the final design a complex combination of several gates, with some having the same inputs. At that point, the designer must step back and review the combinational logic circuit that has been developed and see if there are ways of reducing the number of gates without changing the function of the circuit.

If an equivalent circuit can be formed with fewer gates or fewer inputs, the cost of the circuit is reduced, and its reliability is improved. This process is called the *reduction* or *simplification* of combinational logic circuits and is performed by using the laws and rules of Boolean algebra.

EXAMPLE 4 - 2

Write the equation of the circuit in the following figure, then simplify the equation, and draw the logic circuit of the simplified equation.



Solution The Boolean equation for X is

$$\boldsymbol{X = B\,(A + C) + C}$$

To simplify, first apply $Law\,3$: $[B\,(A + C) = BA + BC]$

$$\boldsymbol{X = BA + BC + C}$$

Solution

Next, factor a C from terms 2 and 3:

$$\boldsymbol{X = BA + C(B + 1)}$$

Apply *Rule 4* ($B + 1 = B$):

$$\boldsymbol{X = BA + C.1}$$

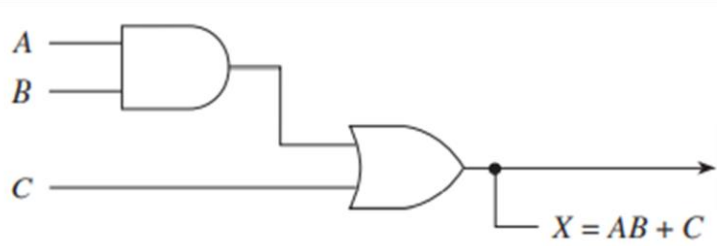
Apply *Rule 2* ($C.1 = C$):

$$\boldsymbol{X = BA + C}$$

Apply *Law 1* ($BA = AB$):

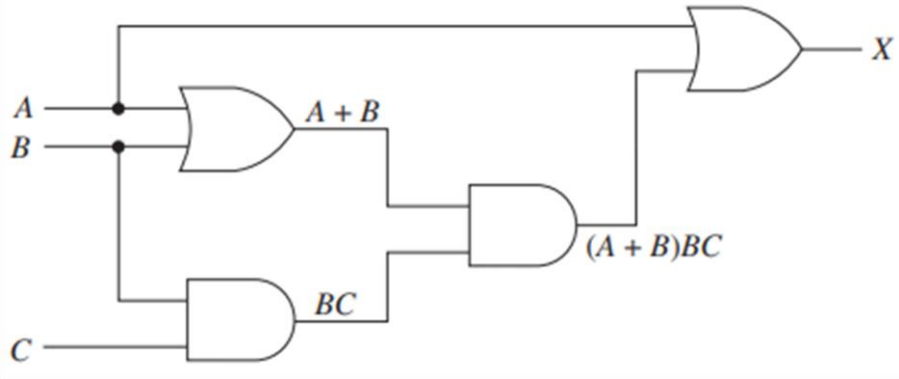
$$\boldsymbol{X = AB + C}$$

The logic circuit of the simplified equation is shown in the following figure:



EXAMPLE 4 - 3

Write the equation of the circuit in the following figure, then simplify the equation, and draw the logic circuit of the simplified equation.



Solution The Boolean equation for X is

$$\boldsymbol{X = (A + B)BC + A}$$

Solution

To simplify, first apply $Law\,3\,[(A + B)\,BC = ABC + BBC]$:

$$\boldsymbol{X = ABC + BBC + A}$$

Apply $Rule\,5\,(BB = B)$:

$$\boldsymbol{X = ABC + BC + A}$$

Factor a BC from terms 1 and 2:

$$\boldsymbol{X = BC\,(A + 1) + A}$$

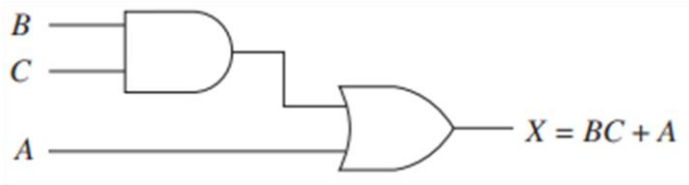
Apply $Rule\,4\,(A + 1 = 1)$:

$$\boldsymbol{X = BC\,.\,1 + A}$$

Apply $Rule\,2\,(BC\,.\,1 = BC)$:

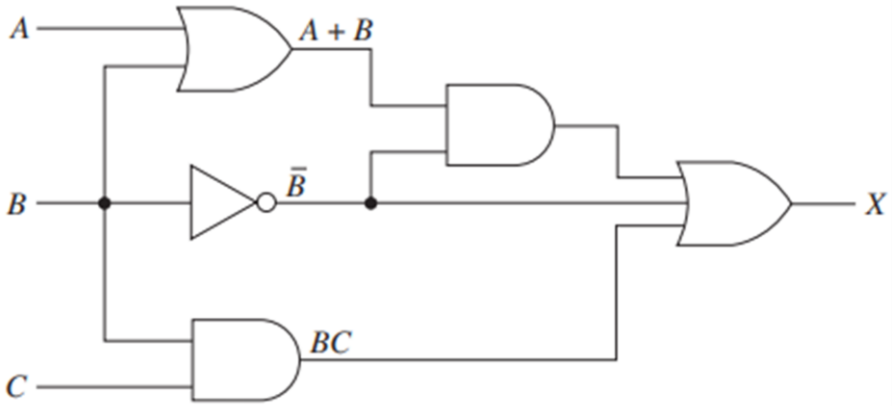
$$\boldsymbol{X = BC + A}$$

The logic circuit of the simplified equation is shown in the following figure:



EXAMPLE 4 - 4

Write the equation of the circuit in the following figure, then simplify the equation, and draw the logic circuit of the simplified equation.



Solution

The Boolean equation for X is

$$\boldsymbol{X = (A + B)\bar{B} + \bar{B} + BC}$$

To simplify, first apply *Law 3* $[(A + B)\,\bar{B} = A\bar{B} + B\bar{B}]$:

$$\boldsymbol{X = A\bar{B} + B\bar{B} + \bar{B} + BC}$$

Apply *Rule 7* $(B\bar{B} = 0)$:

$$\boldsymbol{X = A\bar{B} + 0 + \bar{B} + BC}$$

Apply *Rule 3* $(A\bar{B} + 0 = A\bar{B})$:

$$\boldsymbol{X = A\bar{B} + \bar{B} + BC}$$

Factor a $\bar{B}$ from terms 1 and 2:

$$\boldsymbol{X = \bar{B}(A + 1) + BC}$$

Apply *Rule 4* $(A + 1 = 1)$:

$$\boldsymbol{X = \bar{B}.1 + BC}$$

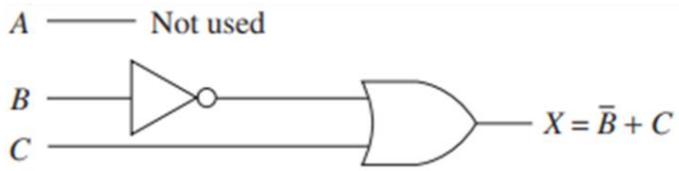
Apply *Rule 2* $(\bar{B}.1 = \bar{B})$:

$$\boldsymbol{X = \bar{B} + BC}$$

Apply *Rule 10* $(\bar{B} + BC = \bar{B} + C)$:

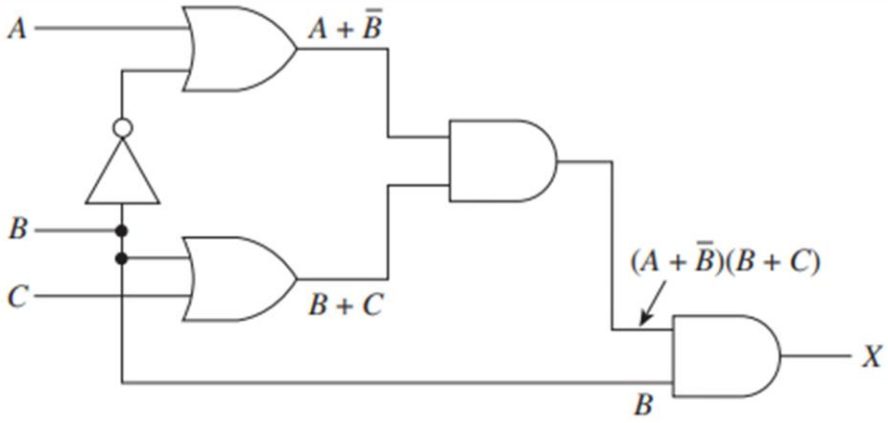
$$\boldsymbol{X = \bar{B} + C}$$

The logic circuit of the simplified equation is shown in the following figure:



EXAMPLE 4 - 5

Write the equation of the circuit in the following figure, then simplify the equation, and draw the logic circuit of the simplified equation.



Solution

The Boolean equation for X is

$$\boldsymbol{X = [(A + \bar{B})(B + C)]B}$$

To simplify, first apply *Law 3* $[(A + \bar{B})(B + C) = AB + AC + \bar{B}B + \bar{B}C]$:

$$\boldsymbol{X = (AB + AC + \bar{B}B + \bar{B}C)B}$$

Apply *Rule 7* $(B\bar{B} = 0)$:

$$\boldsymbol{X = (AB + AC + 0 + \bar{B}C)B}$$

Apply *Rule 3* $(AC + 0 = AC)$:

$$\boldsymbol{X = (AB + AC + \bar{B}C)B}$$

Apply *Law 3*:

$$\boldsymbol{X = ABB + ACB + \bar{B}CB}$$

Apply *Law 1*:

$$\boldsymbol{X = ABB + ABC + \bar{B}BC}$$

Apply *Rules 5* and *7*:

$$\boldsymbol{X = AB + ABC + 0.C}$$

Apply *Rule 1*:

$$\boldsymbol{X = AB + ABC}$$

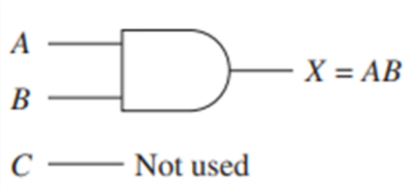
Factor an *AB* from both terms:

$$\boldsymbol{X = AB(1 + C)}$$

Apply *Rule 4* and then *Rule 2*:

$$\boldsymbol{X = AB}$$

The logic circuit of the simplified equation is shown in the following figure:



4.5 De Morgan's Theorem

You may have noticed that we did not use NANDs or NORs in any of the logic circuits in Chapter 3. To simplify circuits containing NANDs and NORs, we need to use a theorem developed by the mathematician Augustus *De Morgan*. This theorem allows us to convert an expression having an inversion bar over two or more variables into an expression having inversion bars over single variables only. This allows us to use the rules presented in the preceding section for the simplification of the equation. In the form of an equation, De Morgan's theorem is stated as follows:

$$\overline{\boldsymbol{A.B}} = \overline{\boldsymbol{A}} + \overline{\boldsymbol{B}}$$

$$\overline{\boldsymbol{A + B}} = \overline{\boldsymbol{A}}.\overline{\boldsymbol{B}}$$

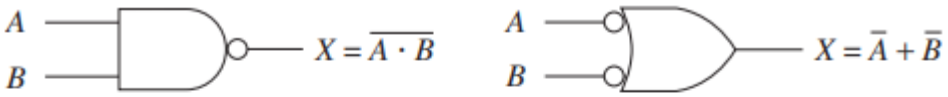
Also, for three or more variables,

$$\overline{\boldsymbol{A.B.C}} = \overline{\boldsymbol{A}} + \overline{\boldsymbol{B}} + \overline{\boldsymbol{C}}$$

$$\overline{A + B + C} = \overline{A}.\overline{B}.\overline{C}$$

Basically, to use the theorem, you break the bar over the variables and either change the AND to an OR or change the OR to an AND.

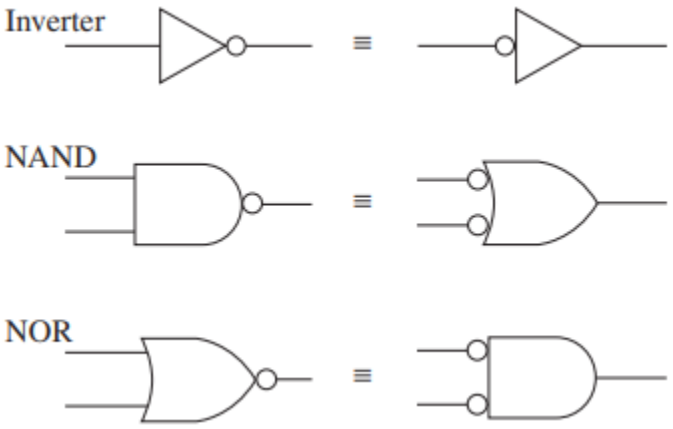
To prove to ourselves that this works, let us apply the theorem to a NAND gate and then compare the truth table of the equivalent circuit to that of the original NAND gate. As you can see in Figure 4-9, to use De Morgan's theorem on a NAND gate, first break the bar over the A.B, then change the AND symbol to an OR. The new equation becomes $X = \overline{A} + \overline{B}$. Notice that **inversion bubbles** are used on the OR gate instead of inverters. By observing the truth tables of the two equations, we can see that the result in the X column is the same for both, which proves that they provide an equivalent output result.



A	B	$X = \overline{AB}$
0	0	1
0	1	1
1	0	1
1	1	0

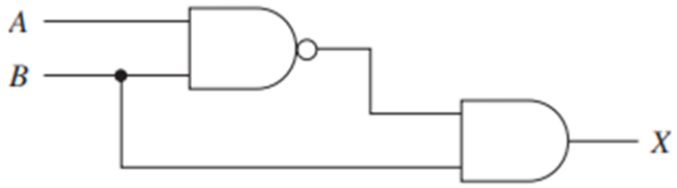
A	B	$X = \overline{A} + \overline{B}$
0	0	1
0	1	1
1	0	1
1	1	0

Figure 4-9 De Morgan's theorem applied to NAND gate produces two identical truth tables.



EXAMPLE 4 - 6

Write the Boolean equation for the circuit shown in the following figure. Then, use De Morgan's theorem and then Boolean algebra rules to simplify the equation. Draw the simplified circuit.



Solution

The Boolean equation for X is

$$\boldsymbol{X = \overline{A}\overline{B}.B}$$

Applying De Morgan's theorem produces:

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

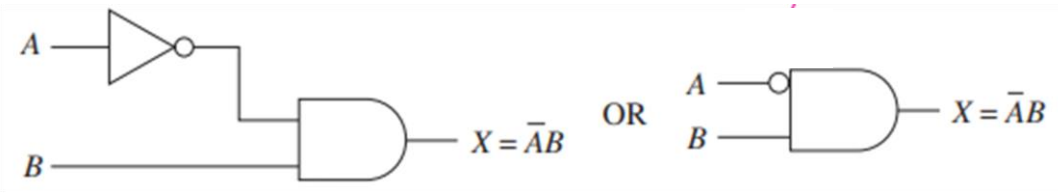
Using Boolean algebra rules produces:

$$\boldsymbol{X = \overline{A}B + B\overline{B}}$$

$$\boldsymbol{X = \overline{A}B + 0}$$

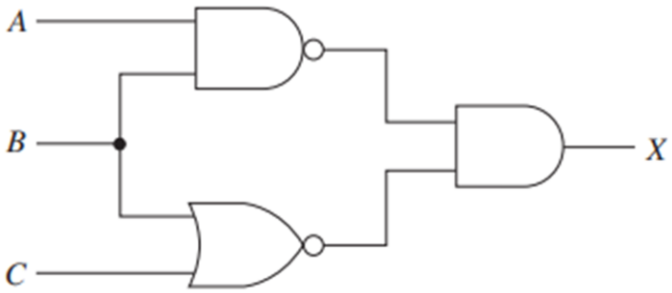
$$\boldsymbol{X = \overline{A}B}$$

The simplified circuit is shown in the following figure:



EXAMPLE 4 - 7

Write the Boolean equation for the circuit shown in the following figure. Then, use De Morgan's theorem and then Boolean algebra rules to simplify the equation. Draw the simplified circuit.



Solution

The Boolean equation for X is

$$\boldsymbol{X = \overline{AB}.\overline{B + C}}$$

Applying De Morgan's theorem produces:

$$\boldsymbol{X = (\overline{A} + \overline{B}).\overline{B}\,\overline{C}}$$

Using Boolean algebra rules produces:

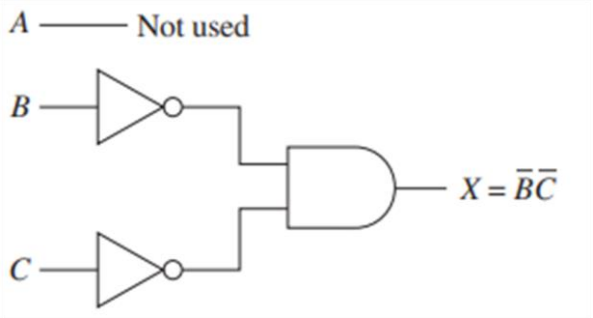
$$\boldsymbol{X = \overline{A}\overline{B}\overline{C} + \overline{B}\overline{B}\overline{C}}$$

$$\boldsymbol{X = \overline{A}\overline{B}\overline{C} + \overline{B}\overline{C}}$$

$$\boldsymbol{X = \overline{B}\overline{C}(\overline{A} + 1)}$$

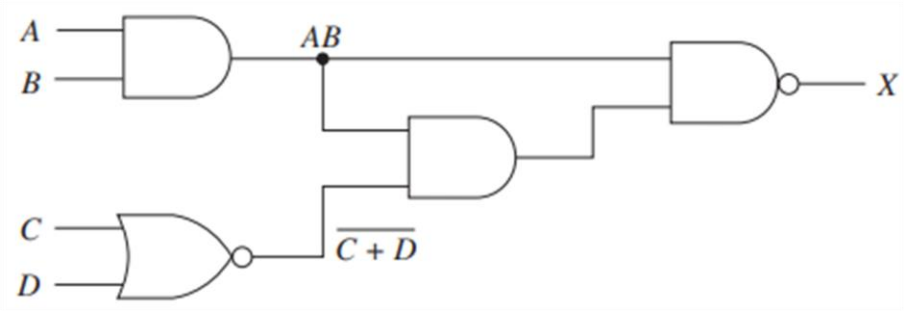
$$\boldsymbol{X = \overline{B}\overline{C}}$$

The simplified circuit is shown in the following figure:



EXAMPLE 4 - 8

Write the Boolean equation for the circuit shown in the following figure. Then, use De Morgan's theorem and then Boolean algebra rules to simplify the equation. Draw the simplified circuit.



Solution

The Boolean equation for *X* is

$$\boldsymbol{X = \overline{(AB.\overline{C+D})AB}}$$

Applying De Morgan's theorem produces:

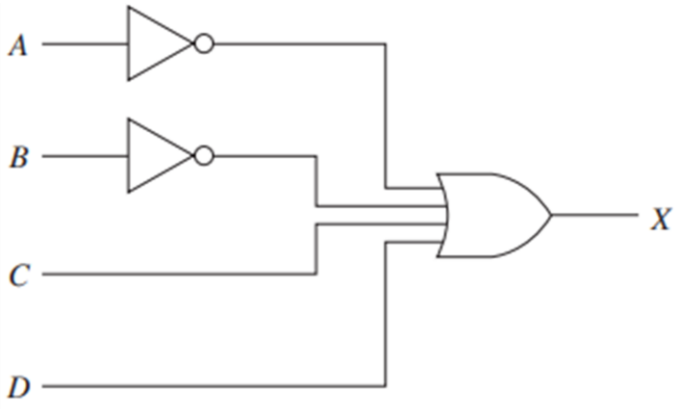
$$\boldsymbol{X = \overline{(AB.\overline{C+D})} + \overline{AB}}$$

$$\boldsymbol{X = \overline{AB} + \overline{\overline{C+D}} + \overline{AB}}$$

$$\boldsymbol{X = \overline{A} + \overline{B} + C + D + \overline{A} + \overline{B}}$$

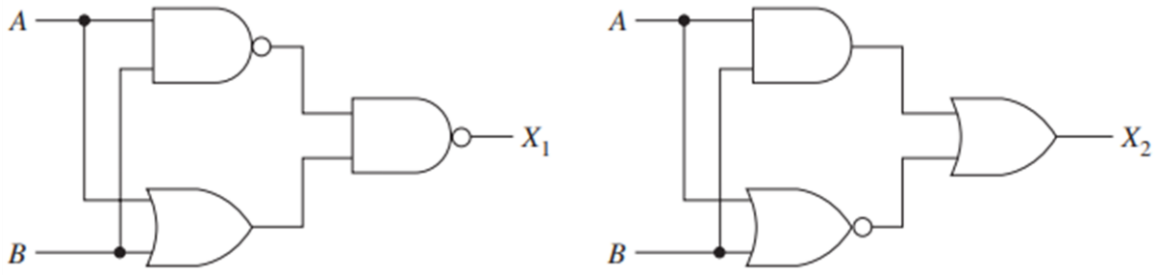
$$\boldsymbol{X = \overline{A} + \overline{B} + C + D}$$

The simplified circuit is shown in the following figure:



EXAMPLE 4 - 9

Using De Morgan's theorem and Boolean algebra, prove that the two circuits shown in the following are equivalent.



Solution

They can be proved to be equivalent if their simplified equations match.

$$X_1 = \overline{\overline{AB}.(A+B)}$$

$$X_1 = \overline{\overline{AB}} + \overline{A+B}$$

$$\boxed{X_1 = AB + \overline{A}.\overline{B}}$$

$$X_2 = AB + \overline{A+B}$$

$$\boxed{X_2 = AB + \overline{A}.\overline{B}}$$

4.6 Simplification Theorems

The following theorems are useful in simplifying Boolean expressions:

Uniting:	**(1)** $XY + X\overline{Y} = X$	$(X+Y)(X+\overline{Y}) = X$
Absorption:	**(2)** $X + XY = X$	$X(X+Y) = X$
Elimination:	**(3)** $X + \overline{X}Y = X + Y$	$X(\overline{X}+Y) = XY$
Consensus:	**(4)** $XY + \overline{X}Z + YZ = XY + \overline{X}Z$	

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

In switching algebra, each of the above theorems can be proved by using a truth table. In a general Boolean algebra, they must be proved algebraically starting with the basic theorems.

Proof of (1): $XY + X\bar{Y} = X(Y + \bar{Y}) = X(1) = X$

$$(X + Y)(X + \bar{Y}) = XX + X\bar{Y} + XY + Y\bar{Y}$$
$$= X + XY + X\bar{Y} = X + X = X$$

Proof of (2): $X + XY = X(1 + Y) = X(1) = X$

$$X(X + Y) = XX + XY = X + XY = X$$

Proof of (3): $X + \bar{X}Y = (X + \bar{X})(X + Y) = 1.(X + Y) = X + Y$

$$X(\bar{X} + Y) = X\bar{X} + XY = 0 + XY = XY$$

Proof of (4): $XY + \bar{X}Z + YZ = XY + \bar{X}Z + 1.YZ$

$$= XY + \bar{X}Z + (X + \bar{X}).YZ$$
$$= XY + \bar{X}Z + XYZ + \bar{X}YZ$$
$$= XY + \bar{X}Z + XYZ + \bar{X}YZ$$
$$= XY + \bar{X}Z \quad \textbf{(Using absorption twice)}$$

EXAMPLE 4 - 10

Simplify:

$$Z = [A + \bar{B}C + D + EF]\left[A + \bar{B}C + \overline{(D + EF)}\right]$$

Solution

Substituting $X = A + \bar{B}C$ and $Y = D + EF$.

$$Z = [X + Y][X + \bar{Y}] = X$$

$$\boldsymbol{Z = A + \bar{B}C}$$

EXAMPLE 4 - 11

Simplify:

$$Z = (AB + C)(\bar{B}D + \bar{C}\bar{E}) + \overline{(AB + C)}$$

Solution

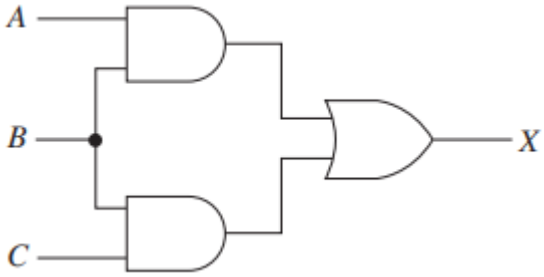
Substituting $\bar{X} = AB + C$ and $Y = \bar{B}D + \bar{C}\bar{E}$.

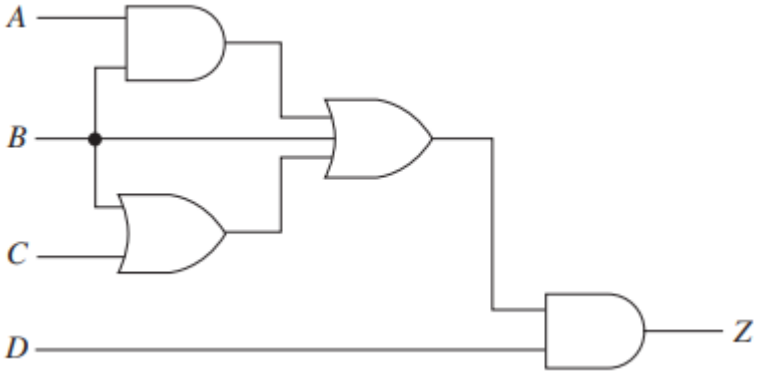
$$Z = \bar{X}Y + X = X + Y$$

$$\boldsymbol{Z = \overline{(AB + C)} + \bar{B}D + \bar{C}\bar{E}}$$

Problems

1. **Write the Boolean equation for each of the logic circuits shown in the following figures:**





2. **The gray water reclamation tank. Write the Boolean equation and draw the logic circuit to implement the following functions:**

   a. Turn on the red light (R) if there is a HIGH opacity (C) and pressure (P) when the level is full (F).

   b. Turn on the green light (G) if there is a HIGH opacity (C) and pressure (P) when the level is mid (M) or full (F).

   c. Turn on the blue light (B) when the tank level is full and any of the sensors for PH (H), opacity (C), or pressure (P) are HIGH.

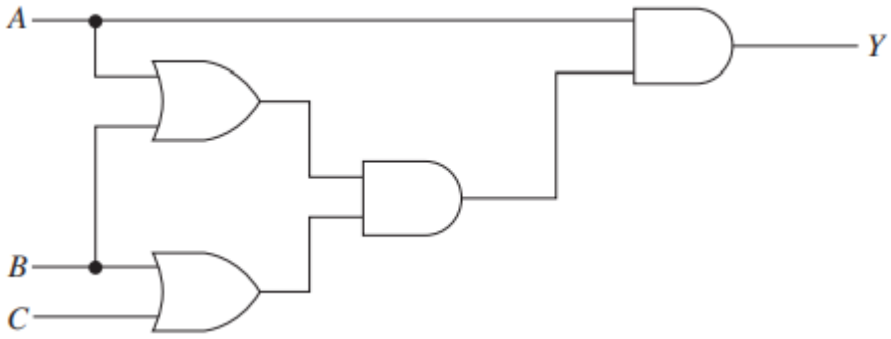
3. **Draw the logic circuit that would be used to implement the following Boolean equations. Also, construct a truth table for each of the equations. Simplify the equation for this problem and then draw the simplification circuit:**

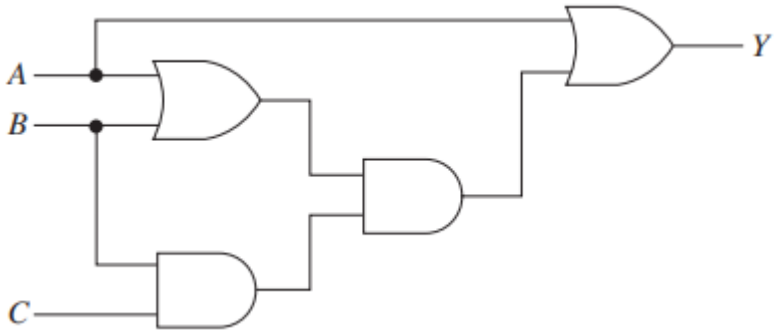
a. $P = (AC + BC)(A + C)$

b. $Q = (A + B)BCD$

c. $R = BC + D + AD$

4. Write the Boolean equation for the circuits of the following figures. Then, simplify the equations, and draw the simplified logic circuit.





5. Draw the logic circuit for the following equations. Simplify the equations and draw the simplified logic circuit.

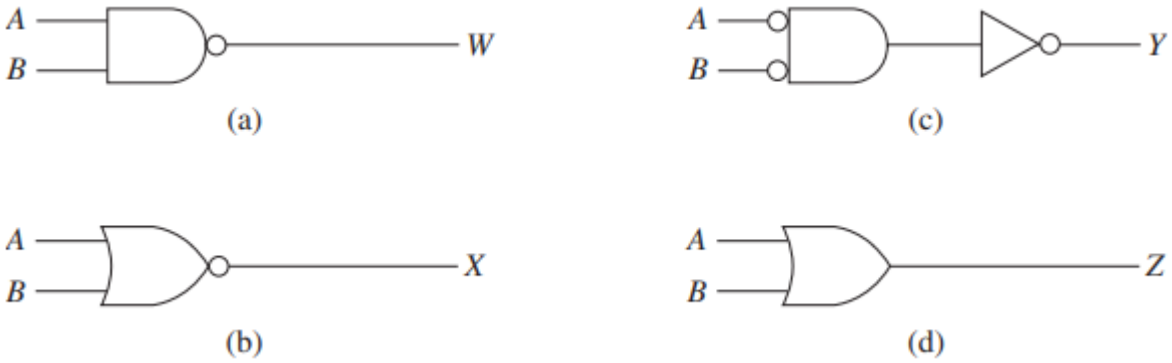
a) $V = AC + ACD + CD.$

b) $W = (BCD + C)CD.$

c) $X = (B + D)(A + C) + ABD.$

d) $Z = ABC + CD + CDE.$

6. **Which two circuits in the following figures produce equivalent output equations?**



7. **Draw the logic circuit for the following equations. Apply De Morgan's theorem and Boolean algebra rules to reduce them to equations having inversion bars over single variables only. Draw the simplified circuit.**

a) $W = \overline{AB} + \overline{A + C}$

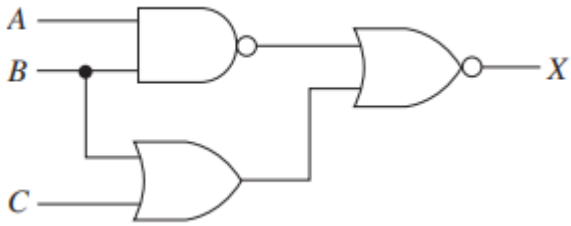
b) $Y = \overline{(AB) + C} + B\bar{C}$

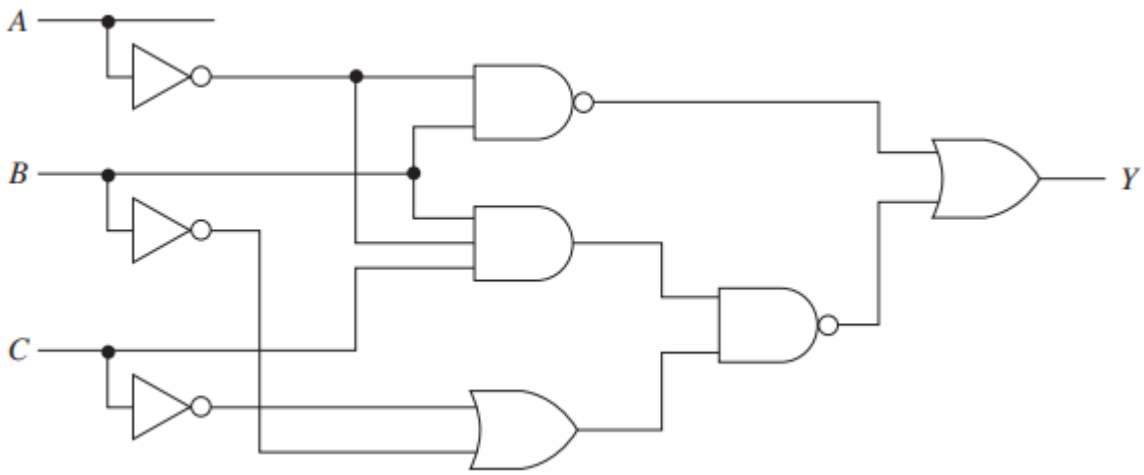
c) $Z = \overline{AB + (\bar{A} + C)}$

d) $X = \overline{\overline{AB} + CD} + \overline{AC\bar{D}}$

e) $R = \overline{(C + D)\overline{A\bar{C}\bar{D}}(\bar{A}C + \bar{D})}$

8. **Write the Boolean equation for the circuits of the following figure. Use De Morgan's theorem and Boolean algebra rules to simplify the equation. Draw the simplified circuit**





9. **Design a logic circuit that will output a 1 (HIGH) only if A and B are both 1 while either C or D is 1.**

10. **Design a logic circuit that will output a HIGH if only one of the inputs A, B, or C is LOW.**

اسم الطالب: .....
رقم الطالب: .....
اسم المقرر: .....
رقم الشيت: .....

الرقم السري: .....

استمارة التقويم المستمر

Question No.	Degree	Signature
1.		
2.		
3.		
4.		
5.		
6.		
7.		
8.		
9.		
10.		

10

5 Standard Forms of Boolean Expressions

OUTLINE

5-1 The Sum-of-Products (SOP) Form.

5-2 The Product-of-Sums (POS) Form.

5-3 Converting Standard SOP to Standard POS.

5-4 Boolean Expressions and Truth Tables.

5-5 Minterms and Maxterms.

5.1 INTRODUCTION

All *Boolean expressions*, regardless of their form, can be converted into either of two standard forms: the *sum-of-products* form or the *product-of-sums* form. Standardization makes the evaluation, simplification, and implementation of Boolean expressions much more systematic and easier.

5.2 The Sum-of-Products (SOP) Form

When two or more product terms are summed by Boolean addition, the resulting expression is a sum-of-products (**SOP**). Some examples are:

$$AB + ABC$$

$$A\overline{B} + \overline{A}\overline{B}\overline{C}$$

The ***domain*** of a general Boolean expression is the set of variables contained in the expression in either complemented or uncomplemented form. For example, the domain of the expression $AB + ABC$ is the set of variables A, B, C and the domain of the expression $A\overline{B}C + C\overline{D}E + BC\overline{D}$ is the set of variables A, B, C, D, E.

5.3 AND/OR Implementation of an SOP Expression

Implementing an **SOP** expression simply requires ORing the *outputs* of two or more AND gates. A product term is produced by an AND operation, and the sum (addition) of two or more product terms is produced by an OR operation. Therefore, an **SOP** expression can be implemented by **AND-OR** logic in which the outputs of a number (equal to the number of product terms in the expression) of AND gates connect to the inputs of an OR gate, as shown in Figure 5–1 for the expression $AB + BCD + AC$. The output X of the OR

gate equals the SOP expression.

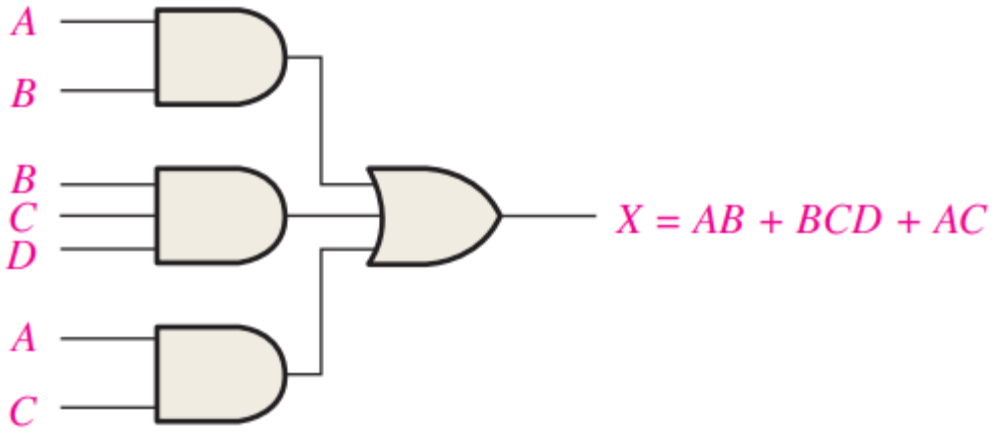


Figure 5–1 Implementation of the SOP expression $AB + BCD + AC$.

5.4 NAND/NAND Implementation of an SOP Expression

NAND gates can be used to implement an **SOP** expression, as illustrated in Figure 5–3. The first level of NAND gates feed into a NAND gate that acts as a negative-OR gate. The NAND and negative-OR inversions cancel, and the result is effectively an AND/OR circuit.

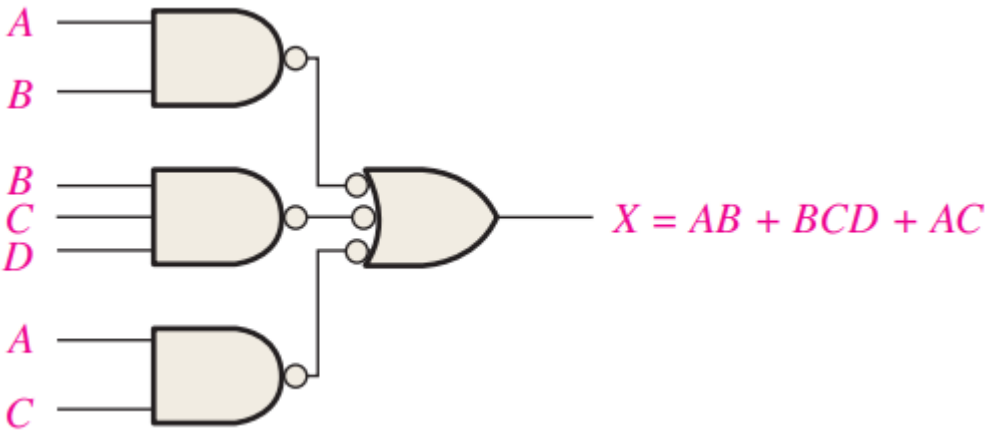


Figure 5–2 Implementation of the SOP expression $AB + BCD + AC$.

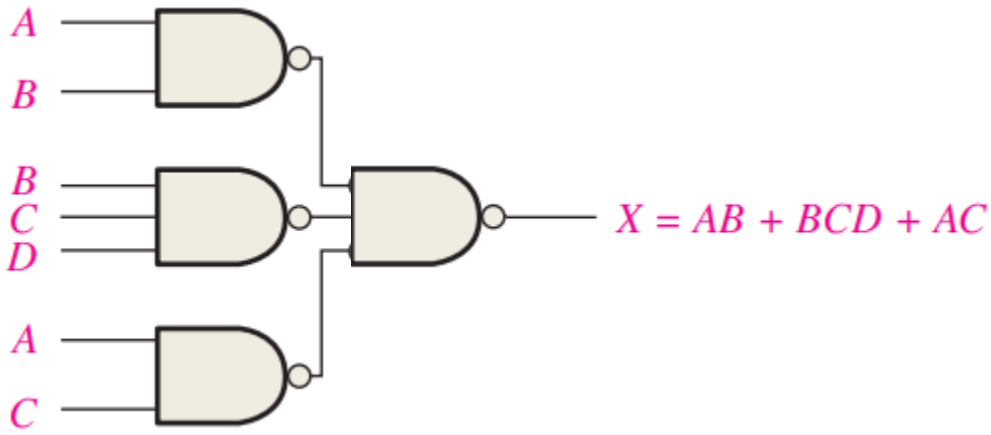


Figure 5–3 NAND/NAND implementation is equivalent to AND/OR in Figure 5–1.

5.5 Conversion of a General Expression to SOP Form

Any logic expression can be changed into SOP form by applying Boolean algebra techniques. For example, let's work with the following equation:

$$X = \overline{A\bar{B} + \bar{C}D}$$

Using De Morgan's theorem again puts it into a POS format:

$$X = \overline{A\bar{B}}.\overline{\bar{C}D}$$

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

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

Using the distributive law produces an equation in the SOP format:

$$X = \bar{A}C + \bar{A}\bar{D} + BC + B\bar{D}$$

5.6 The Standard SOP Form

So far, you have seen SOP expressions in which some of the product terms do not contain all the variables in the *domain* of the expression. For example, the expression $\bar{A}B\bar{C} + A\bar{B}D + \bar{A}B\bar{C}D$ has a domain made up of the variables

$A, B, C,$ and D. However, notice that the complete set of variables in the domain is not represented in the first two terms of the expression; that is, D or $\overline{D}$ is missing from the first term and C or $\bar{C}$ is missing from the second term. A **standard SOP** expression is one in which all the variables in the *domain* appear in each product term in the expression. For example, $\bar{A}B\bar{C}D + A\overline{B}CD + \bar{A}B\bar{C}D$ is a **standard SOP** expression.

Standard SOP expressions are important in constructing truth tables, and in the Karnaugh map simplification method. Any nonstandard SOP expression (referred to simply as SOP) can be converted to the standard form using Boolean algebra.

Converting Product Terms to Standard SOP

Each product term in an **SOP** expression that does not contain all the variables in the domain can be expanded to standard form to include all variables in the domain and their complements. As stated in the following steps, a nonstandard SOP expression is converted into standard form using Boolean algebra rule 8 ($A + \bar{A} = 1$).

Step 1: Multiply each nonstandard product term by a term made up of the sum of a missing variable and its complement. This results in two product terms. As you know, you can multiply anything by 1 without changing its value.

Step 2: Repeat Step 1 until all resulting product terms contain all variables in the domain in either complemented or uncomplemented form. In converting a product term to standard form, the number of product terms is doubled for each missing variable.

EXAMPLE 5 - 1

Convert the following Boolean expression into standard SOP form:

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

Solution The *domain* of this **SOP** expression is A, B, C, D. Take one term at a time. The first term, $A\bar{B}C$, is missing variable D or $\bar{D}$, so multiply the first term by $D + \bar{D}$ as follows:

$$A\bar{B}C = A\bar{B}C(D + \bar{D}) = A\bar{B}CD + A\bar{B}C\bar{D}$$

In this case, two standard product terms are the result.

The second term, $\bar{A}\bar{B}$, is missing variables C or $\bar{C}$ and D or $\bar{D}$, so first multiply the second term by $C + \bar{C}$ as follows:

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

The two resulting terms are missing variable D or $\bar{D}$, so multiply both terms by $D + \bar{D}$ as follows:

$$\bar{A}\bar{B} = \bar{A}\bar{B}C(D + \bar{D}) + \bar{A}\bar{B}\bar{C}(D + \bar{D}) = \bar{A}\bar{B}CD + \bar{A}\bar{B}C\bar{D} + \bar{A}\bar{B}\bar{C}D + \bar{A}\bar{B}\bar{C}\bar{D}$$

In this case, four standard product terms are the result.

The third term, $A\bar{B}C\bar{D}$, is already in standard form. The complete standard SOP form of the original expression is as follows:

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

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

5.7 The Product-of-Sums (POS) Form

When two or more sum terms are multiplied, the resulting expression is a *product-of-sums* (**POS**). Some examples are:

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

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

In a POS expression, a single overbar cannot extend over more than one variable; however, more than one variable in a term can have an overbar. For example, a POS expression can have the term $\bar{A} + \bar{B} + \bar{C}$ but not $\overline{A + B + C}$.

5.8 OR/ AND Implementation of an POS Expression

Implementing a **POS** expression simply requires ANDing the outputs of two or more OR gates. A sum term is produced by an OR operation, and the product of two or more sum terms is produced by an AND operation. Therefore, a **POS** expression can be implemented by logic in which the outputs of a number (equal to the number of sum terms in the expression) of OR gates connect to the inputs of an AND gate, as Figure 5–4.

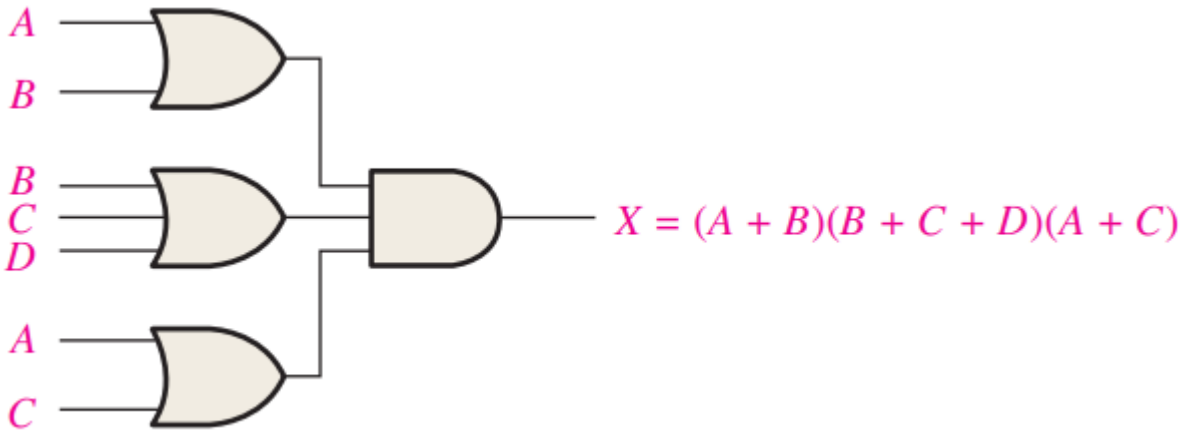


Figure 5–4 Implementation of the POS expression.

5.9 The Standard POS Form

So far, you have seen **POS** expressions in which some of the sum terms do not contain all the variables in the domain of the expression. For example, the expression $(A + \bar{B} + C)(A + B + \bar{D})(A + \bar{B} + \bar{C} + D)$ has a *domain* made up of the variables A, B, C, and D. Notice that the complete set of variables in the domain is not represented in the first two terms of the expression; that is, D or $\bar{D}$ is missing from the first term and C or $\bar{C}$ is missing from the second term.

A standard **POS** expression is one in which all the variables in the domain appear in each sum term in the expression. For example, $(A + \bar{B} + C + D)(A + B + C + \bar{D})(A + \bar{B} + \bar{C} + D)$ is a standard POS expression. Any nonstandard POS expression (referred to simply as POS) can be converted to the standard form using Boolean algebra.

Converting Sum Terms to Standard POS

Each sum term in a POS expression that does not contain all the variables in the domain can be expanded to standard form to include all variables in the domain and their complements. As stated in the following steps, a nonstandard POS expression is converted into standard form using Boolean algebra rule 6 $(A.\bar{A} = 0)$.

Step 1: Add to each nonstandard product term a term made up of the product of the missing variable and its complement. This results in two sum terms. As you know, you can add 0 to anything without changing its value.

Step 2: Apply the rule: $A + BC = (A + B)(A + C)$.

Step 3: Repeat Step 1 until all resulting sum terms contain all variables in the domain in either complemented or uncomplemented form.

EXAMPLE 5 - 2

Convert the following Boolean expression into standard POS form:

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

Solution

The *domain* of this POS expression is A, B, C, D. Take one term at a time. The first term, $A + \bar{B} + C$, is missing variable D or $\bar{D}$, so add $D\bar{D}$:

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

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

The second term, $\bar{B} + C + \bar{D}$, is missing variable A or $\bar{A}$, so add $A\bar{A}$:

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

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

The third term, $A + \bar{B} + \bar{C} + D$, is already in standard form. The standard POS form of the original expression is as follows:

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

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

5.10 Converting Standard SOP to Standard POS

The binary values of the product terms in a given standard SOP expression are not present in the equivalent standard POS expression. Also, the binary values that are not represented in the SOP expression are present in the equivalent POS expression. Therefore, to convert from standard SOP to standard POS, the following steps are taken:

Step 1: Evaluate each product term in the SOP expression. That is, determine the binary numbers that represent the product terms.

Step 2: Determine all the binary numbers not included in the evaluation in Step 1.

Step 3: Write the equivalent sum term for each binary number from Step 2 and express in POS form.

EXAMPLE 5 - 3

Convert the following SOP expression to an equivalent POS expression:

$$\bar{A}\bar{B}\bar{C} + \bar{A}B\bar{C} + \bar{A}BC + A\bar{B}C + ABC$$

Solution

The evaluation is as follows:

$$000 + 010 + 011 + 101 + 111$$

Since there are *three* variables in the domain of this expression, there are a total of eight (2^3) possible combinations. The **SOP** expression contains *five* of these combinations, so the **POS** must contain the other *three* which are **001**, **100**, and **110**. The equivalent **POS** expression is:

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

5.11 Boolean Expressions and Truth Tables

All standard Boolean expressions can be easily converted into truth table format using binary values for each term in the expression. The truth table is a common way of presenting, in a concise format, the logical operation of a circuit. Also, standard SOP or POS expressions can be determined from a truth table. You will find truth tables in data sheets and other literature related to the operation of digital circuits.

5.11.1 Converting SOP Expressions to Truth Table Format

SOP expression is equal to **1** only if at least one of the product terms is equal to 1. A *truth table* is simply a list of the possible combinations of input variable values and the corresponding output values (1 or 0). For an expression with a domain of two variables, there are four different combinations of those variables ($2^2 = 4$). For an expression with a domain of three variables, there are eight different combinations of those variables ($2^3 = 8$). For an expression with a domain of four variables, there are sixteen different combinations of those variables ($2^4 = 16$), and so on.

The **first** step in constructing a *truth table* is to list all possible combinations of binary values of the variables in the expression. **Next**, convert the SOP expression to standard form if it is not already. **Finally**, place a 1 in the output column (X) for each binary value that makes the standard SOP expression a **1** and place a **0** for all the remaining binary values.

EXAMPLE 5 - 4

Develop a truth table for the standard SOP expression $\bar{A}\bar{B}C + A\bar{B}\bar{C} + ABC$.

Solution

There are *three* variables in the *domain*, so there are *eight* possible combinations of binary values of the variables as listed in the following table. The binary values that make the product terms in the expressions equal to 1 are 001, 100, and 111. For each of these binary values, place a 1 in the output column as shown in the table. For each of the remaining binary combinations, place a 0 in the output column.

A	B	C	Output	Product Term
0	0	0	0	
0	0	1	1	$\bar{A}\bar{B}C$
0	1	0	0	
0	1	1	0	
1	0	0	1	$A\bar{B}\bar{C}$
1	0	1	0	
1	1	0	0	
1	1	1	1	ABC

5.11.2 Converting POS Expressions to Truth Table Format

Recall that a **POS** expression is equal to **0** only if at least one of the sum terms is equal to 0. To construct a truth table from a POS expression, list all the possible combinations of binary values of the variables just as was done for the SOP expression. Next, convert the POS expression to standard form if it is not already. Finally, place a 0 in the output column (X) for each binary value that makes the

expression a 0 and place a 1 for all the remaining binary values.

EXAMPLE 5 - 5

Determine the truth table for the following standard POS expression:

$$(A + B + C)(A + \overline{B} + C)(A + \overline{B} + \overline{C})(\overline{A} + B + \overline{C})(\overline{A} + \overline{B} + C)$$

Solution

There are three variables in the domain and the eight possible binary values are listed in the following table. The binary values that make the sum terms in the expression equal to 0 are 000; 010; 011; 101; and 110. For each of these binary values, place a 0 in the output column as shown in the table. For each of the remaining binary combinations, place a 1 in the output column.

A	B	C	Output	Product Term
0	0	0	0	$A + B + C$
0	0	1	1	
0	1	0	0	$A + \overline{B} + C$
0	1	1	0	$A + \overline{B} + \overline{C}$
1	0	0	1	
1	0	1	0	$\overline{A} + B + \overline{C}$
1	1	0	0	$\overline{A} + \overline{B} + C$
1	1	1	1	

EXAMPLE 5 - 6

From the following truth table, determine the standard SOP expression and the equivalent standard POS expression.

A	B	C	Output
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	**1**
1	0	0	**1**
1	0	1	0
1	1	0	**1**
1	1	1	**1**

Solution

There are four 1s in the output column and the corresponding binary values are 011, 100, 110, and 111. Convert these binary values to product terms as follows:

$$011 \rightarrow \bar{A}BC$$

$$100 \rightarrow A\bar{B}\bar{C}$$

$$110 \rightarrow AB\bar{C}$$

$$111 \rightarrow ABC$$

The resulting standard **SOP** expression for the output $X$is

$$X = \bar{A}BC + A\bar{B}\bar{C} + AB\bar{C} + ABC$$

For the **POS** expression, the output is 0 for binary values 000, 001, 010, and 101. Convert these binary values to sum terms as follows:

$$000 \rightarrow A + B + C$$

$$001 \rightarrow A + B + \bar{C}$$

$$010 \rightarrow A + \bar{B} + C$$

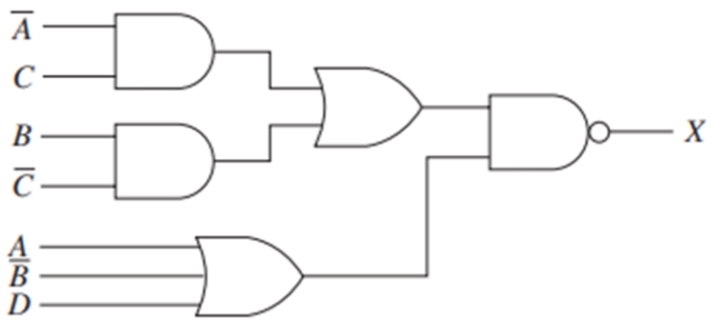
$$101 \rightarrow \bar{A} + B + \bar{C}$$

The resulting standard **POS** expression for the output X is

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

EXAMPLE 5 - 7

Simplify the circuit shown in the following figure down to its SOP form.



Solution

They can be proved to be equivalent if their simplified equations match.

$$\boldsymbol{X = \overline{(\bar{A}C + B\bar{C}).(A + \bar{B} + D)}}$$

$$\boldsymbol{X = \overline{(\bar{A}C + B\bar{C})} + \overline{(A + \bar{B} + D)}}$$

$$\boldsymbol{X = \bar{\bar{A}}C.\overline{B\bar{C}} + \bar{A}B\bar{D}}$$

$$\boldsymbol{X = (\bar{\bar{A}} + \bar{C}).(\bar{B} + \bar{\bar{C}}) + \bar{A}B\bar{D}}$$

$$\boldsymbol{X = (A + \bar{C}).(\bar{B} + C) + \bar{A}B\bar{D}}$$

$$\boldsymbol{X = A\bar{B} + AC + \bar{B}\bar{C} + \bar{C}C + \bar{A}B\bar{D}}$$

$$\boldsymbol{X = A\bar{B} + AC + \bar{B}\bar{C} + \bar{A}B\bar{D}}$$

5.12 Minterms and Maxterms

A binary variable may appear either in its normal form (A) or in its complement form ($\overline{A}$). Now consider two binary variables A and B combined with an AND operation. Since each variable may appear in either form, there are *four* possible combinations: AB, $\overline{A}B$, $A\overline{B}$, and $\overline{A}\overline{B}$. 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 2^n minterms. The 2^n different minterms may be determined by a method similar to the one shown in Table 5.1 for three variables. The binary numbers from 0 to 2^n - 1 are listed under the n 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 m_j, where the subscript j denotes the decimal equivalent of the binary number of the minterm designated.

Table 5.1 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

In a similar fashion, *n* variables forming an OR term, with each variable being primed or unprimed, provide 2^n possible combinations, called ***maxterms***, or ***standard sums***. The eight maxterms for three variables, together with their symbolic designations, are listed in Table 5.1. Any 2^n 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.

(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 $\boldsymbol{f_1}$ in Table 5.2 is determined by expressing the combinations **001**, **100**, and **111** as $\boldsymbol{\bar{x}\bar{y}z}$, $\boldsymbol{x\bar{y}\bar{z}}$, and $\boldsymbol{xyz}$, respectively. Since each one of these *minterms* results in $f_1 = 1$, we have:

$$\boldsymbol{f_1 = \bar{x}\bar{y}z + x\bar{y}\bar{z} + xyz = m_1 + m_4 + m_7}$$

Table 5.2 Functions of Three Variables

x	*y*	*z*	Function f_1	Function f_2
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Similarly, it may be easily verified that:

$$f_2 = \bar{x}yz + x\bar{y}z + xy\bar{z} + xyz = m_3 + m_5 + m_6 + m_7$$

Now consider the *complement* of a Boolean function. It may be read from the truth table by forming a *minterm* for each combination that produces a 0 in the function and then ORing those terms. The *complement* of f_1 is read as:

$$\overline{f_1} = \bar{x}\bar{y}\bar{z} + \bar{x}y\bar{z} + \bar{x}yz + x\bar{y}z + xy\bar{z}$$

If we take the complement of $\overline{f_1}$, we obtain the function f_1:

$$f_1 = (x + y + z)(x + \bar{y} + z)(x + \bar{y} + \bar{z})(\bar{x} + y + \bar{z})(\bar{x} + \bar{y} + z)$$

$$= M_0.M_2.M_3.M_5.M_6$$

Similarly, it is possible to read the expression for f_2 from the table:

$$f_2 = (x + y + z)(x + y + \bar{z})(x + \bar{y} + z)(\bar{x} + y + z)$$

$$= M_0.M_1.M_2.M_4$$

Boolean functions expressed as a sum of minterms or product of maxterms are said to be in *canonical form*.

EXAMPLE 5 - 8

Express the Boolean function $F = A + \bar{B}C$ as a sum of minterms.

Solution

The function has *three* variables: A, B, and C. The first term A is missing two variables, therefore,

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

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

$$A = ABC + AB\bar{C} + A\bar{B}C + A\bar{B}\bar{C}$$

The second term $\bar{B}C$ is missing one variable; hence,

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

Combining all terms, we have:

$$F = A + \bar{B}C$$
$$= ABC + AB\bar{C} + A\bar{B}C + A\bar{B}\bar{C} + \bar{A}\bar{B}C$$

Rearranging the *minterms* in ascending order, we finally obtain:

$$F = \bar{A}\bar{B}C + A\bar{B}\bar{C} + A\bar{B}C + AB\bar{C} + ABC$$

001 100 101 110 111

$$F = m_1 + m_4 + m_5 + m_6 + m_7$$

When a Boolean function is in its sum-of-minterms form, it is sometimes convenient to express the function in the following brief notation:

$$F(A, B, C) = \Sigma(1, 4, 5, 6, 7)$$

Table 5.3 Truth Table for $F = A + \bar{B}C$

A	B	C	F
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

EXAMPLE 5 - 9

Express the Boolean function $F = xy + \bar{x}z$ as a product of maxterms.

Solution

First, convert the function into OR terms by using the *distributive* law:

$$F = xy + \bar{x}z = (xy + \bar{x})(xy + z)$$

$$= (x + \bar{x})(\bar{x} + y)(x + z)(y + z)$$

$$= (\bar{x} + y)(x + z)(y + z)$$

The function has three variables: x, y, and z. Each OR term is missing one variable; therefore,

$$\bar{x} + y = \bar{x} + y + z\bar{z} = (\bar{x} + y + z)(\bar{x} + y + \bar{z})$$

$$x + z = x + z + y\bar{y} = (x + y + z)(x + \bar{y} + z)$$

$$y + z = y + z + x\bar{x} = (x + y + z)(\bar{x} + y + z)$$

Combining all the terms and removing those which appear more than once, we finally obtain:

$$F = (\bar{x} + y + z)(\bar{x} + y + \bar{z})(x + y + z)(x + \bar{y} + z)$$

Rearranging the maxterms, we finally obtain:

$$F = (x + y + z)(x + \bar{y} + z)(\bar{x} + y + z)(\bar{x} + y + \bar{z})$$

$$000 \qquad 010 \qquad 100 \qquad 101$$

$$F = M_0 M_2 M_4 M_5$$

A convenient way to express this function is as follows:

$$F = \Pi(0, 2, 4, 5)$$

EXAMPLE 5 - 10

Show that $\bar{a}c + \bar{b}\bar{c} + ab = \bar{a}\bar{b} + bc + a\bar{c}$.

Solution

We will find the minterm expansion of each side by supplying the missing variables. For the left side,

$$\bar{a}c + \bar{b}\bar{c} + ab = \bar{a}c(b + \bar{b}) + \bar{b}\bar{c}(a + \bar{a}) + ab(c + \bar{c})$$

$$= \bar{a}bc + \bar{a}\bar{b}c + a\bar{b}\bar{c} + \bar{a}\bar{b}\bar{c} + abc + ab\bar{c}$$

$$011 \quad 001 \quad 100 \quad 000 \quad 111 \quad 110$$

$$= m_3 + m_1 + m_4 + m_0 + m_7 + m_6$$

$$= m_0 + m_1 + m_3 + m_4 + m_6 + m_7$$

For the right side,

$$\bar{a}\bar{b} + bc + a\bar{c} = \bar{a}\bar{b}(c + \bar{c}) + bc(a + \bar{a}) + a\bar{c}(b + \bar{b})$$

$$= \bar{a}\bar{b}c + \bar{a}\bar{b}\bar{c} + abc + \bar{a}bc + ab\bar{c} + a\bar{b}\bar{c}$$

$$001 \quad 000 \quad 111 \quad 011 \quad 110 \quad 100$$

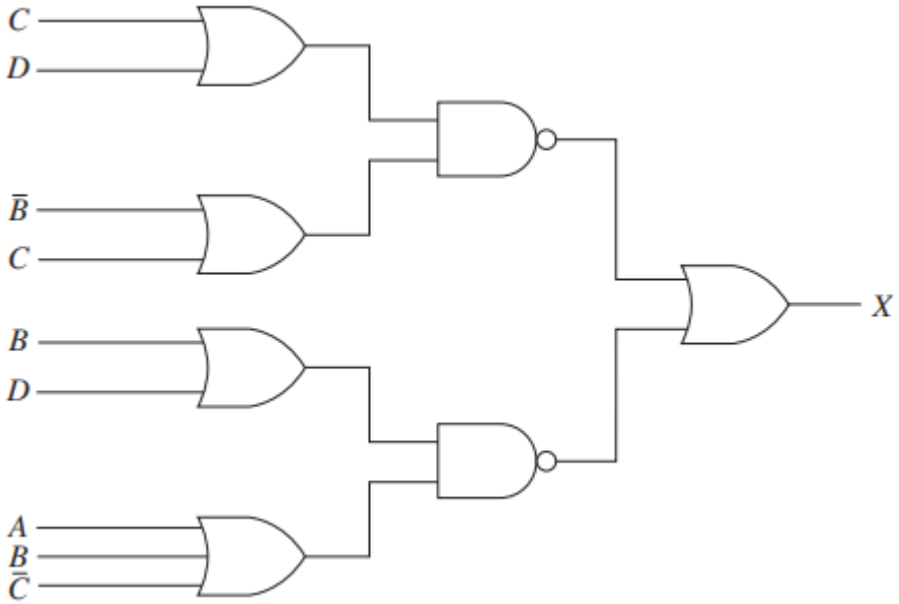
$$= m_1 + m_0 + m_7 + m_3 + m_6 + m_4$$

$$= m_0 + m_1 + m_3 + m_4 + m_6 + m_7$$

Because the two minterm expansions are the same, the equation is valid.

Problems

1. **Simplify the circuit to its SOP form, then draw the logic circuit of the simplified form.**



2. **Convert the following expressions to sum-of-product (SOP) forms:**

   a. $(A + C)(CD + AC)$.

   b. $BC + DE(B\bar{C} + DE)$.

   c. $B + C[BD + (C + \bar{D})E]$

3. **Define the domain of each SOP expression in Problem 2 and convert the expression to standard SOP form.**

4. **Determine the binary value of each term in the standard SOP expressions from Problem 3.**

5. **Convert each standard SOP expression in Problem 3 to standard POS form.**

6. Develop a truth table for each of the following standard SOP expressions:

a) $ABC + \bar{A}\bar{B}C + AB\bar{C}$.

b) $WXYZ + \bar{W}X\bar{Y}Z + W\bar{X}Y\bar{Z} + \bar{W}\bar{X}YZ + WX\bar{Y}\bar{Z}$.

c) $\bar{A}B + AB\bar{C} + \bar{A}\bar{C} + A\bar{B}C$.

7. Develop a truth table for each of the standard POS expressions:

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

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

8. Each of three coins has two sides, heads, and tails. Represent the heads or tails status of each coin by a logical variable (A for the first coin, B for the second coin, and C for the third) where the logical variable is 1 for heads and 0 for tails.

(a) Write a logic function $F\ (A, B, C)$ which is 1 iff exactly one of the coins is heads after a toss of the coins.

(b) Express F as a minterm expansion and as a maxterm expansion.

9. Given: $F(a, b, c) = ab\bar{c} + \bar{b}$.

(a) Express F as a minterm expansion. (Use m-notation.)

(b) Express F as a maxterm expansion. (Use M-notation.)

(c) Express F′ as a minterm expansion. (Use m-notation.)

(d) Express F′ as a maxterm expansion. (Use M-notation.)

10. For each truth table, derive a standard SOP and a standard POS expression.

(a)

A B C	*X*
0 0 0	0
0 0 1	0
0 1 0	0
0 1 1	0
1 0 0	0
1 0 1	1
1 1 0	1
1 1 1	1

(b)

A B C D	*X*
0 0 0 0	0
0 0 0 1	0
0 0 1 0	1
0 0 1 1	0
0 1 0 0	1
0 1 0 1	1
0 1 1 0	0
0 1 1 1	1
1 0 0 0	0
1 0 0 1	0
1 0 1 0	0
1 0 1 1	1
1 1 0 0	1
1 1 0 1	0
1 1 1 0	0
1 1 1 1	1

11. A flow rate sensing device used on a liquid transport pipeline functions as follows. The device provides a 5-bit output where all five bits are zero if the flow rate is less than 10 gallons per minute. The first bit is 1 if the flow rate is at least 10 gallons per minute; the first and second bits are 1 if the flow rate is at least 20 gallons per minute; the first, second, and third bits are 1 if the flow rate is at least 30 gallons per minute; and so on. The five bits, represented by the logical variables A, B, C, D, and E, are used as inputs to a device that provides two outputs Y and Z.

(a) Write an equation for the output Y if we want Y to be 1 iff the flow rate is less than 30 gallons per minute.

(b) Write an equation for the output Z if we want Z to be 1 iff the flow rate is at least 20 gallons per minute but less than 50 gallons per minute.

12. Given: $F(a, b, c) = (a + b + d)(\bar{a} + c)(\bar{a} + \bar{b} + \bar{c})(a + b + \bar{c} + \bar{d})$.

(a) Express F as a minterm expansion. (Use m-notation.)

(b) Express F as a maxterm expansion. (Use M-notation.)

(c) Express F′ as a minterm expansion. (Use m-notation.)

(d) Express F′ as a maxterm expansion. (Use M-notation.)

13. A combinational logic circuit has four inputs (A, B, C, and D) and one output Z. The output is 1 iff the input has three consecutive 0's or three consecutive 1's. For example, if A = 1, B = 0, C = 0, and D = 0, then Z = 1, but if A = 0, B = 1, C = 0, and D = 0, then Z = 0. Design the circuit using one four-input OR gate and four three-input AND gates.

اسم الطالب: .....
رقم الطالب: .....
اسم المقرر: .....
رقم الشيت: .....

الرقم السري: .....

استمارة التقويم المستمر

Question No.	Degree	Question No.	Degree	Signature
1.		2.		
3.		4.		
5.		6.		
7.		8.		
9.		10.		
11.		12.		
13.				

13

6 Karnaugh Map

OUTLINE

6.1 Introduction

We learned in the previous chapters that by using Boolean algebra and De Morgan's theorem, we can minimize the number of gates that are required to implement a particular logic function. This is very important for the reduction of circuit cost, physical size, and gate failures. You may have found that some of the steps in the Boolean reduction process require ingenuity on your part and a lot of practice.

Karnaugh mapping was named for its originator, Maurice Karnaugh, who in 1953 developed another method of simplifying logic circuit. A *Karnaugh map* provides a systematic method for simplifying Boolean expressions and, if properly used, will produce the simplest SOP or POS expression possible, known as the *minimum expression*. As you have seen, the effectiveness of algebraic simplification depends on your familiarity with all the laws, rules, and theorems of Boolean algebra and on your ability to apply them. The Karnaugh map provides a method for simplification. Other simplification techniques include the *Quine-McCluskey* method and the *Espresso* algorithm.

6.2 Karnaugh Map SOP Minimization

A *Karnaugh map* (*K*-map) is like a truth table in that it graphically shows the output level of a Boolean equation for each of the possible input variable combinations. Each output level is placed in a separate cell of the *K*-map. *K*-maps can be used to simplify equations having two, three, four, five, or six different input variables. Solving five- and six-variable *K*-maps is extremely cumbersome; they can be more practically solved using advanced computer techniques. In this book, we solve two-, three-, and four-variable *K*-maps.

Determining the number of cells in a *K*-map is the same as finding the number

of combinations or entries in a truth table. A *two*-variable map requires 2^2=4 cells. A *three*-variable map requires 2^3=8 cells. A *four*-variable map requires 2^4=16 cells. The three different *K*-maps are shown in Figure 6-1.

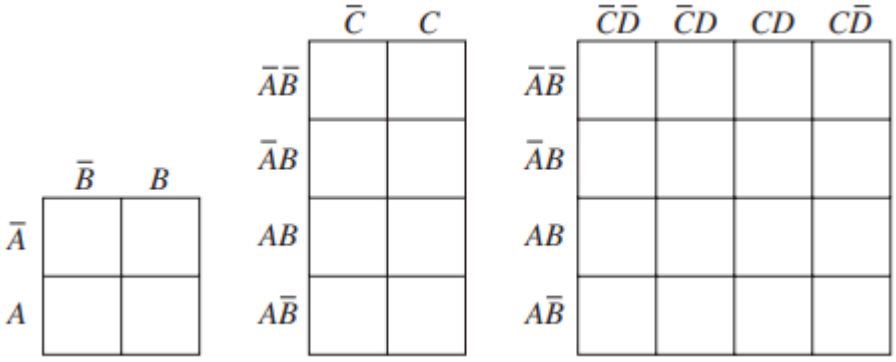


Figure 6-1 Two-, three-, and four-variable Karnaugh maps.

Each cell within the K-map corresponds to a particular combination of the input variables. For example, in the two-variable K-map, the upper left cell corresponds to $\bar{A}\bar{B}$, the lower left cell is $A\bar{B}$, the upper right cell is $\bar{A}B$ and the lower right cell is AB. Also notice that when moving from one cell to an adjacent cell, only one variable changes.

To use the *K*-map reduction procedure, you must perform the following steps:

1. Transform the Boolean equation to be reduced into an SOP expression.
2. Fill in the appropriate cells of the *K*-map.
3. Encircle adjacent cells in groups of two, four, or eight. (The more adjacent cells encircled, the simpler the final equation is; adjacent means a side is touching, not diagonal.)
4. Find each term of the final SOP equation by determining which variables remain constant within each circle.

EXAMPLE 6 - 1

Simplify the following SOP equation using the Karnaugh mapping technique.

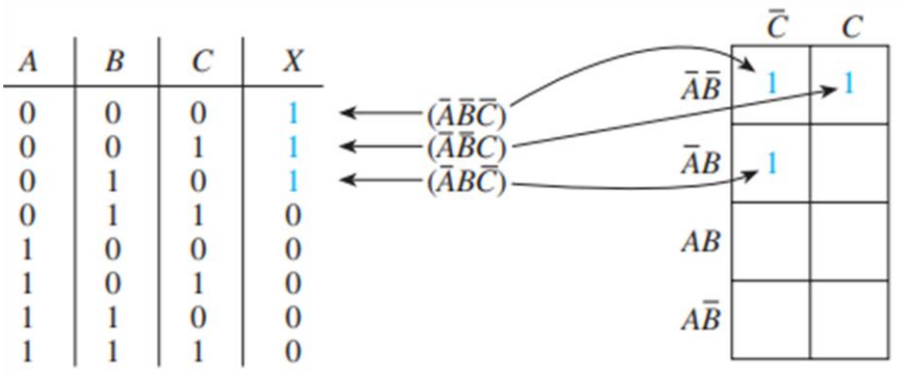
$$X = \overline{A}(\overline{B}C + \overline{B}\overline{C}) + \overline{A}B\overline{C}$$

Solution

First, transform the equation to an SOP expression:

$$X = \overline{A}\overline{B}C + \overline{A}\overline{B}\overline{C} + \overline{A}B\overline{C}$$

The terms of that SOP expression can be put into a *truth table* and then transferred to a *K*-map. Working with the *K*-map, we now encircle adjacent 1's in groups of two, four, or eight. We end up with two circles of two cells each. The first circle surrounds the two 1's at the top of the K-map, and the second circle surrounds the two 1's in the left column of the K-map.

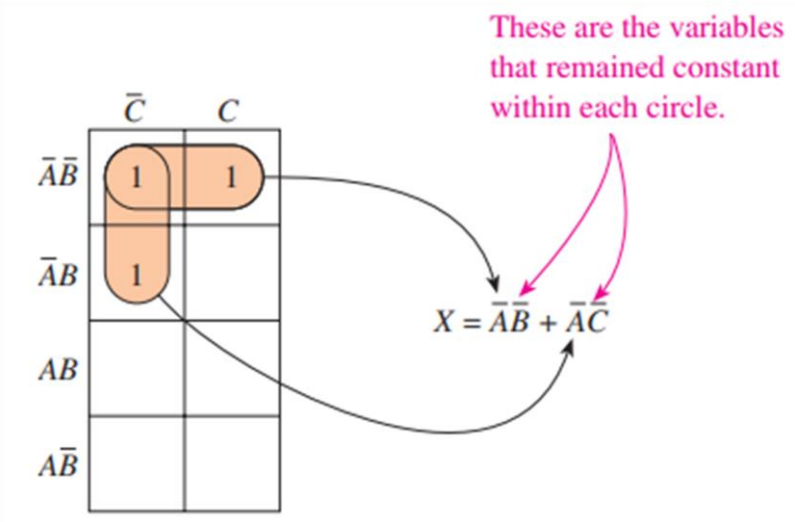


A	B	C	X
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Once the circles have been drawn encompassing all the 1's in the map, the final simplified equation is obtained by determining which variables remain the same within each circle. Well, the first circle (across the top) encompasses $\overline{A}\overline{B}\overline{C}$ and $\overline{A}\overline{B}C$.

The variables that remain the same within the circle are $\overline{A}\overline{B}$. Therefore, $\overline{A}\overline{B}$ becomes one of the terms in the final SOP equation. The second circle (left column) encompasses $\overline{A}\overline{B}\overline{C}$ and $\overline{A}B\overline{C}$.

The variables that remain the same within that circle are $\bar{A}\bar{C}$. Therefore, the second term in the final equation is $\bar{A}\bar{C}$.



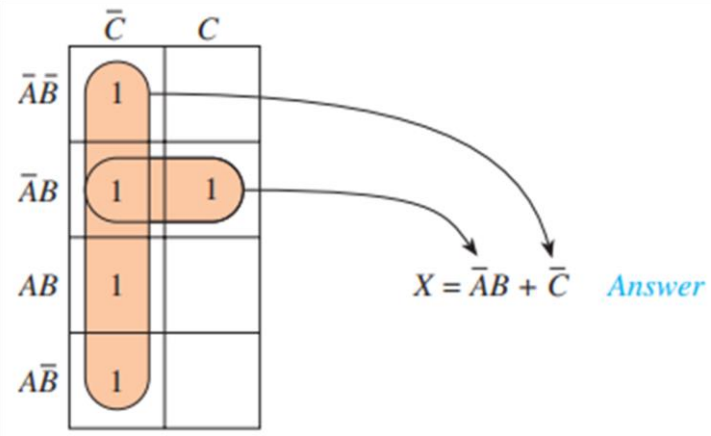
EXAMPLE 6 - 2

Simplify the following SOP equation using the Karnaugh mapping technique.

$$X = \bar{A}B + \bar{A}\bar{B}\bar{C} + AB\bar{C} + A\bar{B}\bar{C}$$

Solution

1. Construct an *eight*-cell K-map and fill in a 1 in each cell that corresponds to a term in the original equation. (Notice that $\bar{A}B$ has no C variable in it. Therefore, $\bar{A}B$ is satisfied whether C is HIGH or LOW, so will fill in two cells: $\bar{A}BC + \bar{A}B\bar{C}$).



2. Encircle adjacent cells in the largest group of two or four or eight.

3. Identify the variables that remain the same within each circle and write the final simplified SOP equation by ORing them together.

second circle surrounds the two 1's in the left column of the K-map.

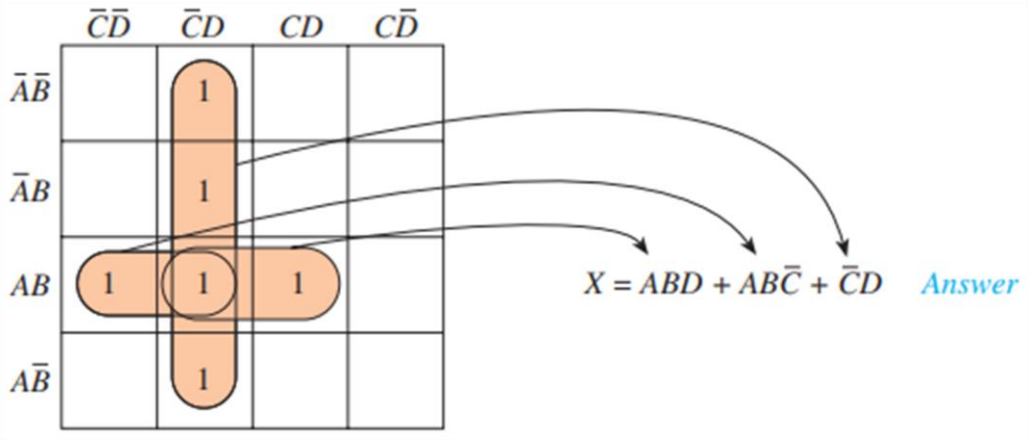
EXAMPLE 6 - 3

Simplify the following SOP equation using the Karnaugh mapping technique.

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

Solution

Because there are *four* different variables in the equation, we need a 16-cell map.

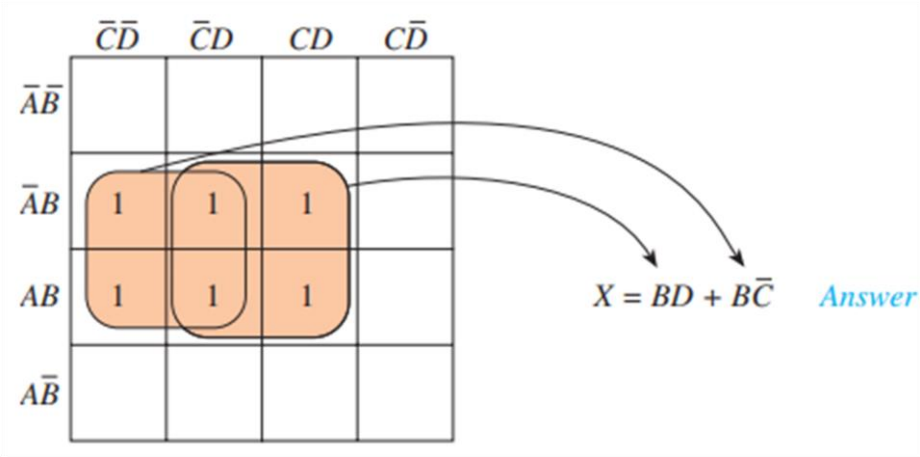


EXAMPLE 6 - 4

Simplify the following SOP equation using the Karnaugh mapping technique.

$$X = B\bar{C}\bar{D} + \bar{A}B\bar{C}D + AB\bar{C}D + \bar{A}BCD + ABCD$$

Solution



EXAMPLE 6 - 5

Simplify the following SOP equation using the Karnaugh mapping technique.

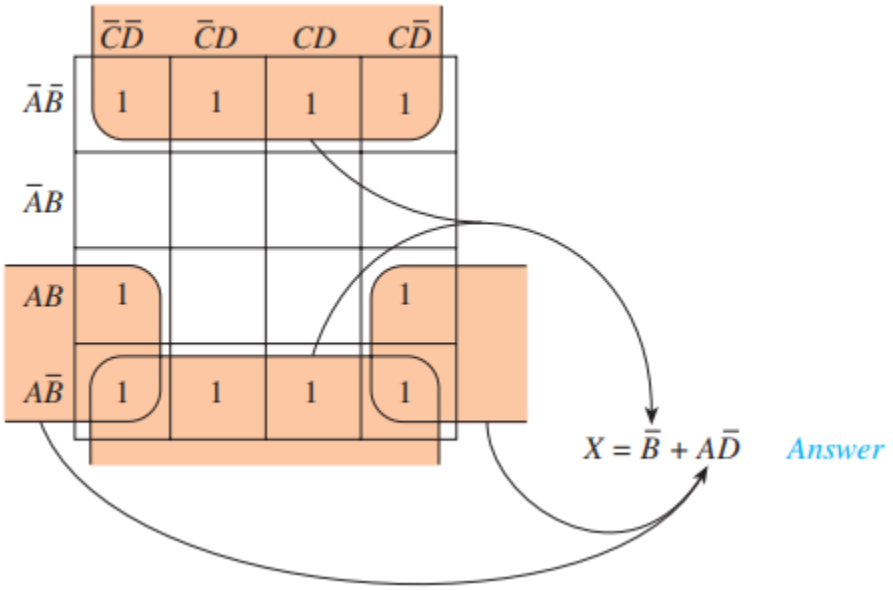
$$X = \bar{A}\bar{B}\bar{C} + A\bar{C}\bar{D} + A\bar{B} + ABC\bar{D} + \bar{A}\bar{B}C$$

Solution

Notice in the following figure that a new technique called **wraparound** is introduced. You must think of the K-map as a continuous cylinder in the horizontal direction, like the label on a soup can. This makes the left row of cells adjacent to the right row of cells. Also, in the vertical direction, a continuous cylinder like a soup can lying on its side makes the top row of cells adjacent to the bottom row of cells. In the following figure, for example, the four top cells are adjacent to the four bottom cells, to combine as eight cells having the variable $\bar{B}$ in common.

Another circle of four is formed by the wraparound adjacencies of the lower left and lower right pairs combining to have $A\bar{D}$ in common.

The final equation becomes $X = \bar{B} + A\bar{D}$. Compare that simple equation with the original equation that had five terms in it.

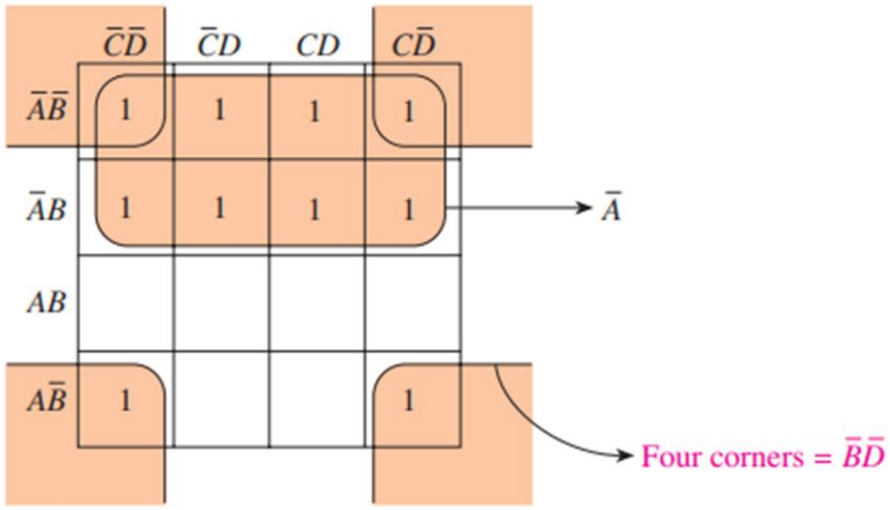


EXAMPLE 6 - 6

Simplify the following SOP equation using the Karnaugh mapping technique.

$$X = \bar{A}\bar{D} + A\bar{B}\bar{D} + \bar{A}\bar{C}D + \bar{A}CD$$

Solution



The final simplified equation is

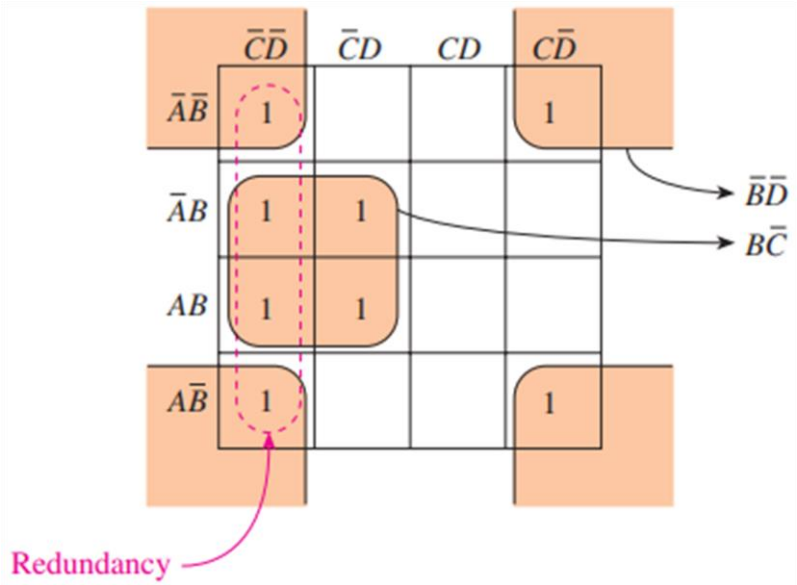
$$X = \bar{A} + \bar{B}\bar{D}$$

EXAMPLE 6 - 7

Simplify the following SOP equation using the Karnaugh mapping technique.

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

Solution

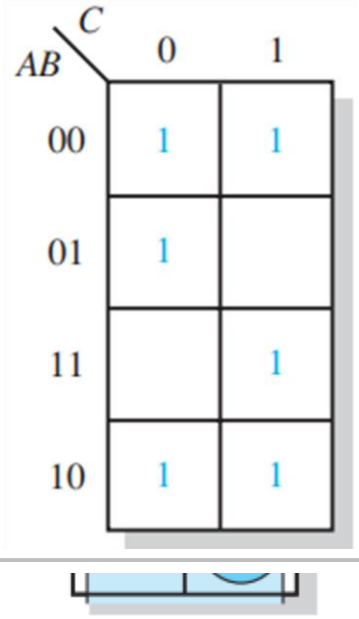


The final simplified equation is

$$X = B\bar{C} + \bar{B}\bar{D}$$

EXAMPLE 6 - 8

Determine the product terms for each of the Karnaugh maps in the following figure and write the resulting minimum SOP expression.

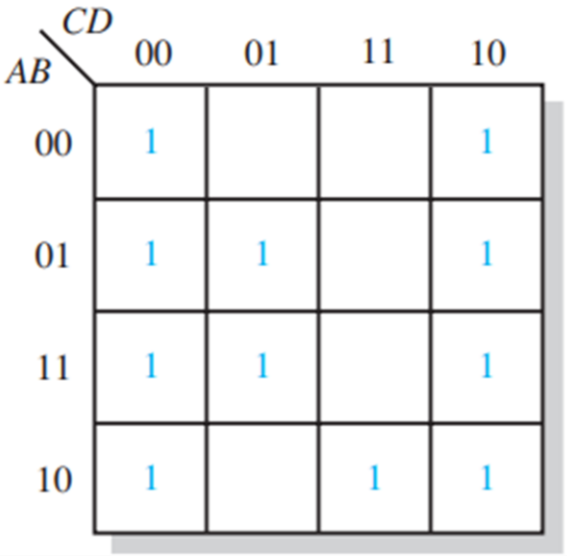


2. Identify the variables that remain the same within each circle and write the final simplified SOP equation.

$$\boldsymbol{X = \overline{A}\overline{C} + AC + \overline{B}}$$

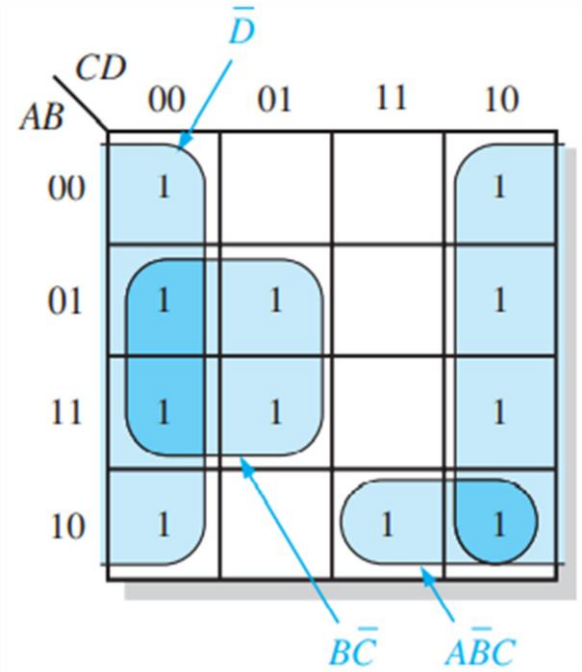
EXAMPLE 6 - 9

Determine the product terms for each of the Karnaugh maps in the following figure and write the resulting minimum SOP expression.



Solution

1. Encircle adjacent cells in the largest group of two or four or eight.



2. Identify the variables that remain the same within each circle and write the final simplified SOP equation.

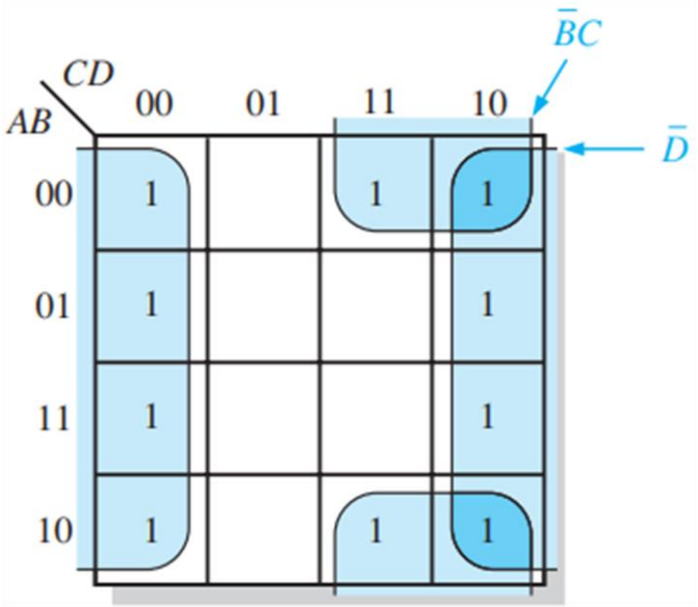
$$X = A\overline{B}C + B\overline{C} + \overline{D}$$

EXAMPLE 6 - 10

Use a Karnaugh map to minimize the following SOP expression:

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

Solution



$$X = \bar{B}C + \bar{D}$$

6.3 Mapping Directly from a Truth Table

You have seen how to map a Boolean expression; now you will learn how to go directly from a truth table to a Karnaugh map. Recall that a truth table gives the output of a Boolean expression for all possible input variable combinations. An example of a Boolean expression and its truth table representation is shown in Figure 6-2. Notice in the truth table that the output ***X*** is **1** for four different input variable combinations. The **1**s in the output column of the truth table are mapped directly onto a Karnaugh map into the cells corresponding to the values of the associated input variable combinations, as shown in Figure 6-2. In the figure you

can see that the Boolean expression, the truth table, and the Karnaugh map are simply different ways to represent a logic function.

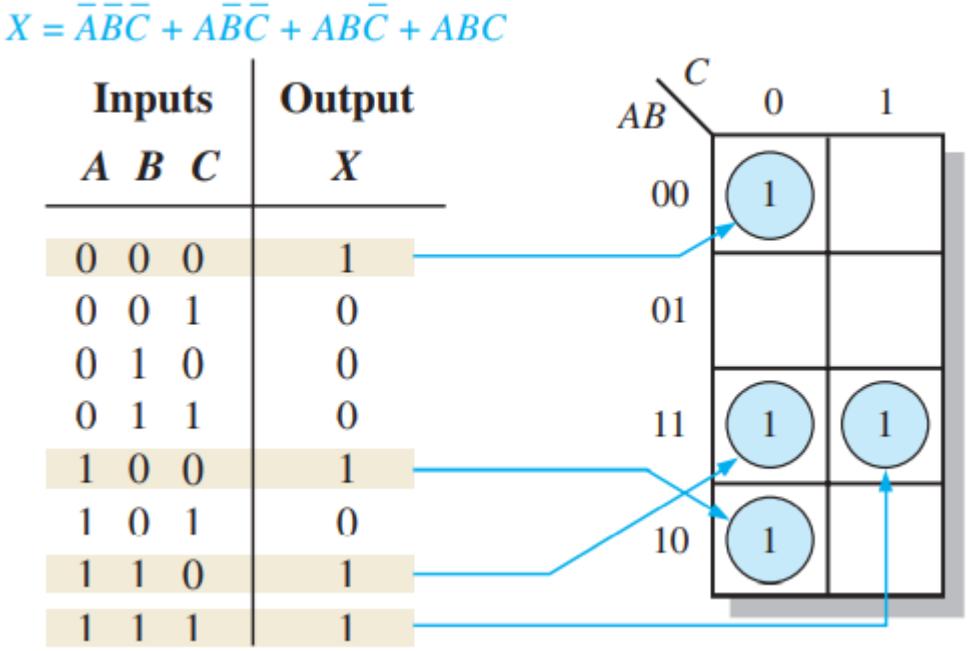


Figure 6-2 Example of mapping directly from a truth table to a Karnaugh map.

6.4 Karnaugh Map POS Minimization

In this section, we focus on POS expressions. The approaches are much the same except that with POS expressions, 0s representing the standard sum terms are placed on the Karnaugh map instead of 1s.

For a POS expression in standard form, a 0 is placed on the Karnaugh map for each sum term in the expression. Each 0 is placed in a cell corresponding to the value of a sum term. For example, for the sum term $\boldsymbol{A + \overline{B} + C}$, a 0 goes in the 010 cell on a 3-variable map.

When a POS expression is completely mapped, there will be a number of 0s on the Karnaugh map equal to the number of sum terms in the standard POS expression. The cells that do not have a 0 are the cells for which the expression

is 1. Usually, when working with POS expressions, the 1s are left off. The following steps and the illustration in Figure 6-3 show the mapping process.

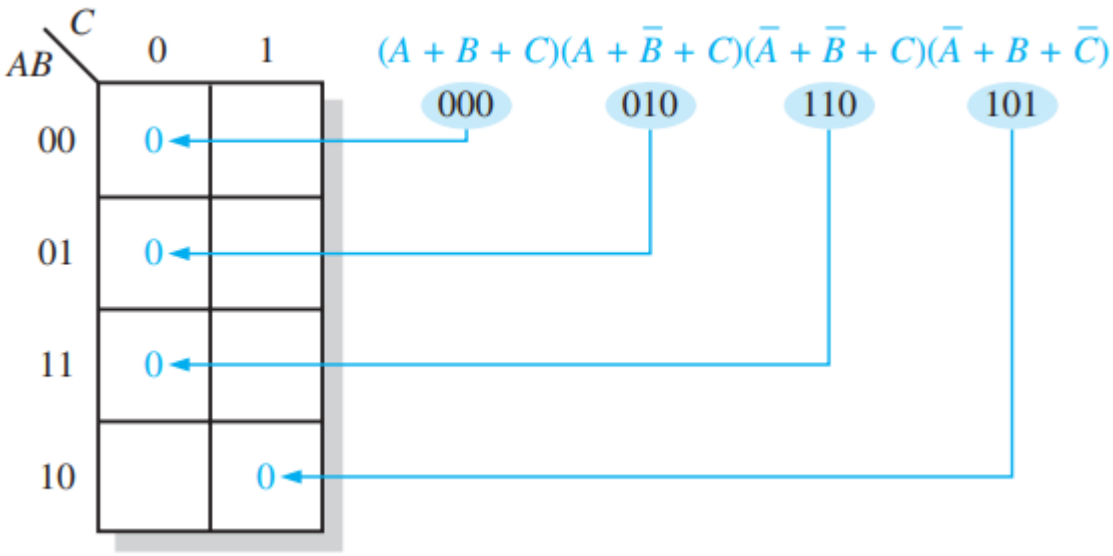


Figure 6-3 Example of mapping a standard POS expression.

The process for minimizing a POS expression is basically the same as for an SOP expression except that you group 0s to produce minimum sum terms instead of grouping 1s to produce minimum product terms. The rules for grouping the 0s are the same as those for grouping the 1s.

EXAMPLE 6 - 11

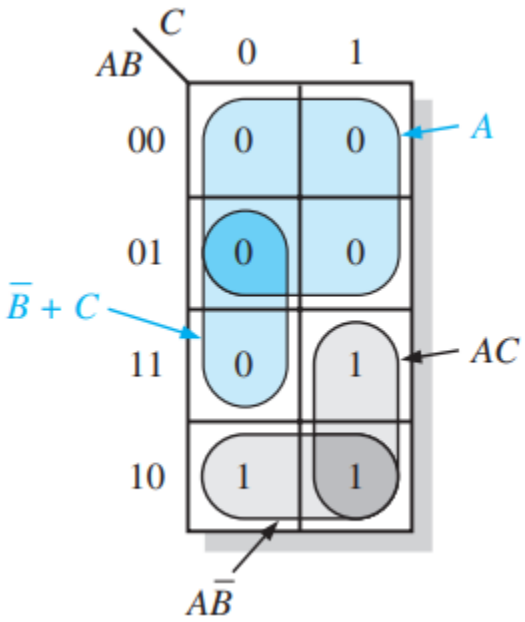
Use a Karnaugh map to minimize the following standard POS expression:

$$\boldsymbol{X = (A + B + C)(A + B + \overline{C})(A + \overline{B} + C)(A + \overline{B} + \overline{C})(\overline{A} + \overline{B} + C)}$$

Also, derive the equivalent SOP expression.

Solution

Map the standard POS expression and group the cells as shown in the following figure.



The sum term for each blue group is shown in the figure and the resulting minimum POS expression is:

$$\boldsymbol{A(\overline{B} + C)}$$

Grouping the 1s as shown by the gray areas yields an SOP expression that is equivalent to grouping the 0s. ($\boldsymbol{A\overline{B} + AC}$)

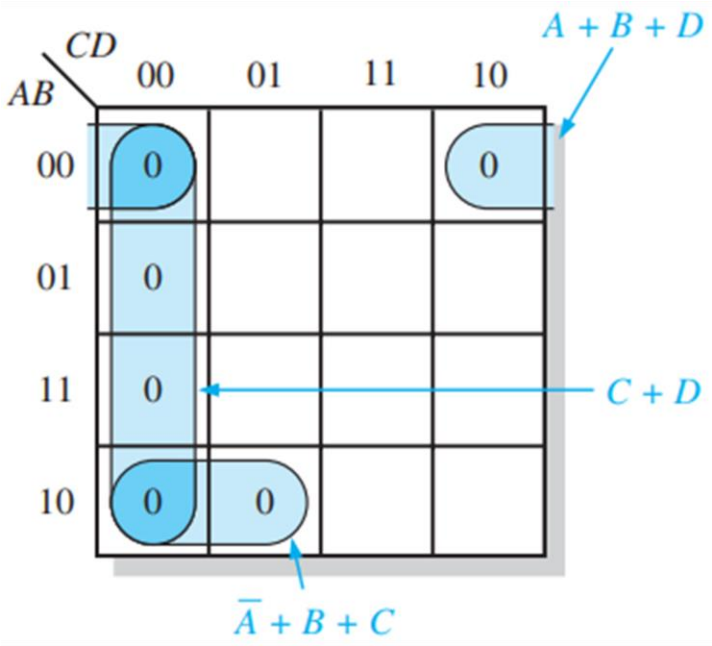
EXAMPLE 6 - 12

Use a Karnaugh map to minimize the following POS expression:

$$(\boldsymbol{B}+\boldsymbol{C}+\boldsymbol{D})(\boldsymbol{A}+\boldsymbol{B}+\overline{\boldsymbol{C}}+\boldsymbol{D})(\overline{\boldsymbol{A}}+\boldsymbol{B}+\boldsymbol{C}+\overline{\boldsymbol{D}})(\boldsymbol{A}+\overline{\boldsymbol{B}}+\boldsymbol{C}+\boldsymbol{D})(\overline{\boldsymbol{A}}+\overline{\boldsymbol{B}}+\boldsymbol{C}+\boldsymbol{D})$$

Solution

The first term must be expanded into $A+B+C+D$ and $\bar{A}+B+C+D$ to get a standard POS expression, which is then mapped; and the cells are grouped as shown in the following.



The sum term for each group is shown and the resulting minimum POS expression is:

$$(\boldsymbol{A}+\boldsymbol{B}+\boldsymbol{D})(\boldsymbol{C}+\boldsymbol{D})(\overline{\boldsymbol{A}}+\boldsymbol{B}+\boldsymbol{C})$$

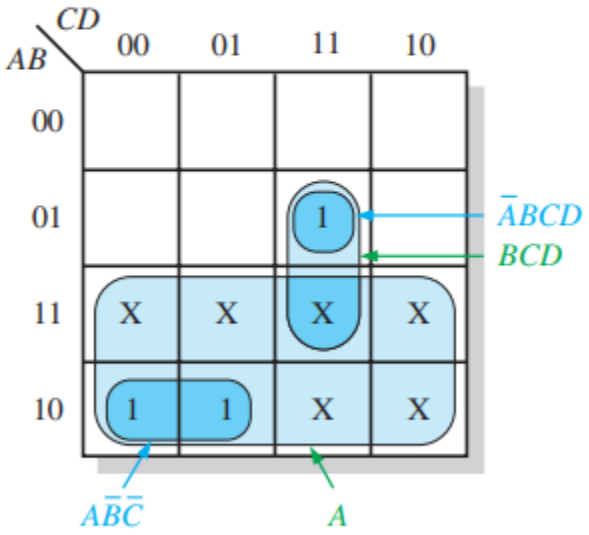
6.5 Don't Care Conditions

Sometimes a situation arises in which some input variable combinations are not allowed. For example, recall that in the BCD code covered in Chapter 1, there are six invalid combinations: 1010, 1011, 1100, 1101, 1110, and 1111. Since these unallowed states will never occur in an application involving the BCD code, they can be treated as "*don't care*" terms with respect to their effect on the output. That is, for these "*don't care*" terms either a 1 or a 0 may be assigned to the output; it really does not matter since they will never occur. The "*don't care*" terms can be used to advantage on the Karnaugh map. Figures 6-4 and 6-5 show that for each "*don't care*" term, an X is placed in the cell. When grouping the 1s, the Xs can be treated as 1s to make a larger grouping or as 0s if they cannot be used to advantage.

Inputs				Output
A	*B*	*C*	*D*	*Y*
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X

(a) Truth table

Don't cares



(b) Without "don't cares" $Y = A\bar{B}\bar{C} + \bar{A}BCD$
With "don't cares" $Y = A + BCD$

Figure 6-4 Example of the use of "don't care" conditions to simplify an expression.

The larger a group, the simpler the resulting term will be:

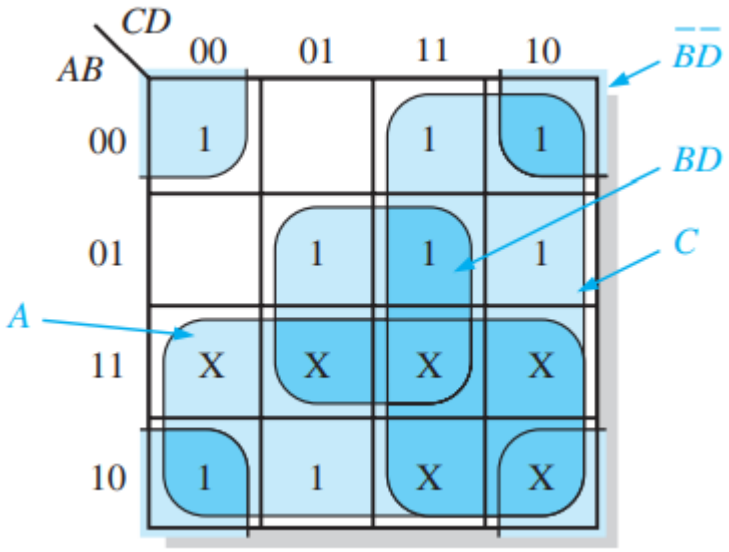


Figure 6-5 Example of the use of "don't care" conditions to simplify an expression.

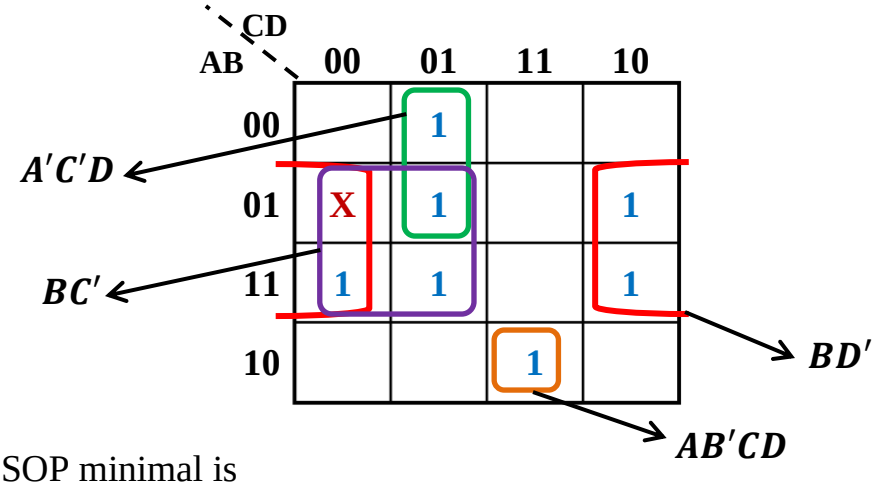
$$a = A + C + BD + \overline{B}\overline{D}$$

EXAMPLE 6 - 13

Minimize the following function in SOP minimal form using K-Maps:

$$f = m(1,5,6,11,12,13,14) + d(4)$$

Solution



Therefore, SOP minimal is

$$f = BC' + BD' + A'C'D + AB'CD$$

EXAMPLE 6 - 14

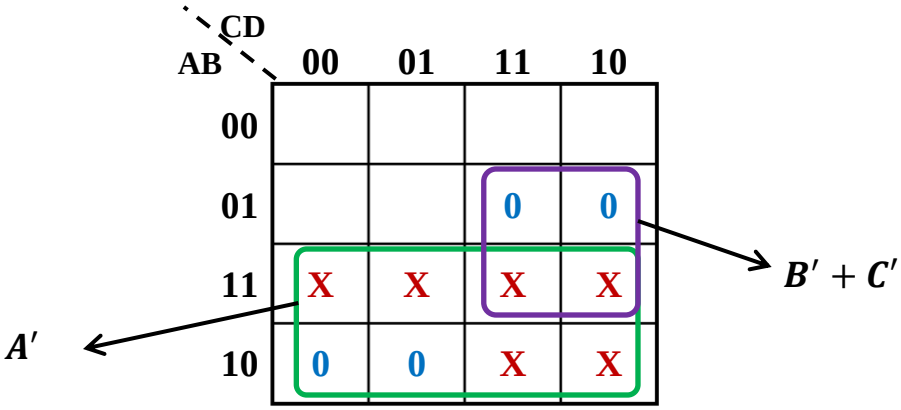
Minimize the following function in POS minimal form using K-Maps:

$$\boldsymbol{F(A,B,C,D) = m(0,1,2,3,4,5) + d(10,11,12,13,14,15)}$$

Solution

Writing the given expression in POS form:

$$F(A,B,C,D) = M(6,7,8,9) + d(10,11,12,13,14,15)$$



Therefore, POS minimal is,

$$\boldsymbol{F = A'(B' + C')}$$

Problems

1. **Using a Karnaugh map, reduce the following equations to a minimum form.**

   a) $X = AB\bar{C} + \bar{A}B + \bar{A}\bar{B}$.

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

   c) $W = \bar{B}(C\bar{D} + \bar{A}D) + \bar{B}\bar{C}(A + \bar{A}\bar{D})$.

   d) $Z = \bar{B}\bar{C}D + B\bar{C}D + \bar{C}\bar{D} + C\bar{D}(B + \bar{A}\bar{B})$.

2. **Use a Karnaugh map to find the minimum SOP form for each expression:**

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

   b) $AC(\bar{B} + C)$.

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

   d) $\bar{A}\bar{B}\bar{C} + A\bar{B}C + \bar{A}BC + AB\bar{C}$.

   e) $AC[\bar{B} + B(B + \bar{C})]$.

3. **Expand each expression to a standard SOP form and minimize each expression with a Karnaugh map:**

   a. $A + BC$.

   b. $A\bar{B}\bar{C}D + AC\bar{D} + B\bar{C}D + \bar{A}BC\bar{D}$.

   c. $A\bar{B} + A\bar{B}\bar{C}D + CD + B\bar{C}D + ABCD$.

4. **Use a Karnaugh map to find the minimum POS for each expression:**

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

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

(c) $(X + \bar{Y})(W + \bar{Z})(\bar{X} + \bar{Y} + \bar{Z})(W + X + Y + Z)$.

5. Work parts (a) through (d) with the given truth table.

A	B	C	F_1	F_2	F_3	F_4
0	0	0	1	1	0	1
0	0	1	X	0	0	0
0	1	0	0	1	X	0
0	1	1	0	0	1	1
1	0	0	0	1	1	1
1	0	1	X	0	1	0
1	1	0	0	X	X	X
1	1	1	1	X	1	X

a) Find the simplest expression for F_1, and specify the values for the don't-cares that lead to this expression.

b) Repeat for F_2.

c) Repeat for F_3.

d) Repeat for F_4.

6. For the following map,

CD \ AB	00	01	11	10
00	1	1	1	
01	1	1	1	
11	1			1
10	1	1	1	

a) Why is A′B′ not essential?

b) Why is BD′ essential? because it's covered only by one prime implicant

c) Is A′D′ essential? Why? not essential, because it's covered by more than one prime implicant

d) Is BC′ essential? Why? essential, because it's covered only by one prime implicant

e) Is B′CD essential? Why? essential, because it's covered only by one prime implicant

f) Find the minimum sum of products.

7. Minimize the following function in SOP minimal form using K-Maps:

$$F(A,B,C,D) = m(1,2,6,7,8,13,14,15) + d(3,5,12)$$

8. Minimize the following function in POS minimal form using K-Maps:

$$F(A,B,C,D) = m(1,2,6,7,8,13,14,15) + d(3,5,12)$$

اسم الطالب: .....
رقم الطالب: .....
اسم المقرر: .....
رقم الشيت: .....

الرقم السري: .....

استمارة التقويم المستمر

Question No.	Degree	Signature
1.		
2.		
3.		
4.		
5.		
6.		
7.		
8.		

8

7 Multi-Level Gate Circuits NAND and NOR Gates

OUTLINE

7.1 INTRODUCTION

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. Because of the prominence of NAND and NOR gates in the design of digital circuits, 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.

7.2 Multi-Level Gate Circuits

The maximum number of gates cascaded in series between a circuit input and the output is referred to as the *number of levels of gates*. Thus, a function written in sum-of-products form or in product-of-sums form corresponds directly to a *two-level gate* circuit.

We will not normally count inverters which are connected directly to the input variables when determining the number of levels in a circuit. In this chapter, we will use the following terminology:

1. **AND-OR** circuit means a *two-level* circuit composed of a level of AND gates followed by an OR gate at the output.
2. **OR-AND** circuit means a *two-level* circuit composed of a level of OR gates followed by an AND gate at the output.
3. **OR-AND-OR** circuit means a *three-level* circuit composed of a level of OR gates followed by a level of AND gates followed by an OR gate at the output.
4. Circuit of AND and OR gates implies no ordering of the gates; the output gate may be either AND or OR.

The number of gates, gate inputs, and levels in a circuit can be determined by inspection of the corresponding expression. In the example of Figure 7-1(a), the tree diagram drawn below the expression for Z indicates that the corresponding circuit will have *four* levels, *six* gates, and *13* gate inputs, as verified in Figure 7-1(b).

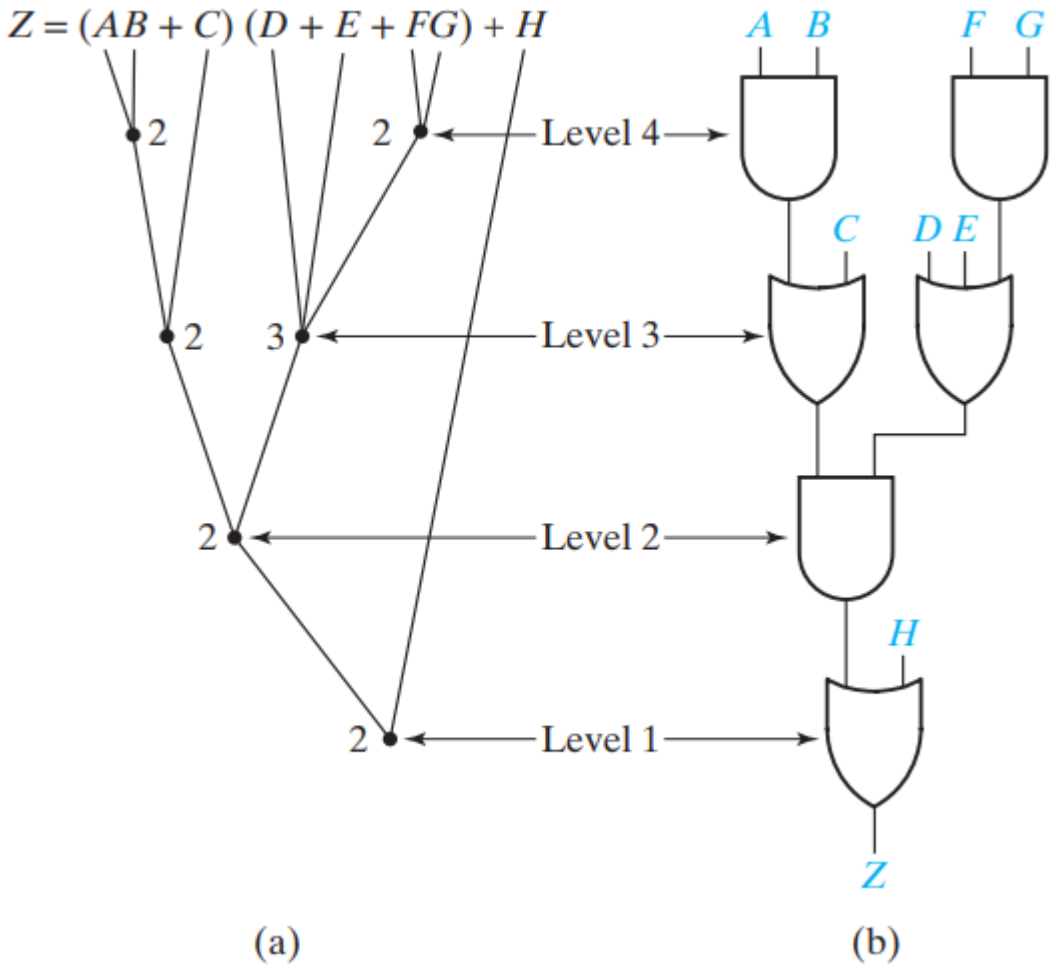


Figure 7.1 Four-Level Realization of Z

We can change the expression for Z to three levels by partially multiplying it: $Z = (AB + C)\,[(D + E) + FG] + H = AB\,(D + E) + C\,(D + E) + ABFG + CFG + H$ As shown in Figure 7-2, the resulting circuit requires *three* levels, *six* gates, and *19* gate inputs.

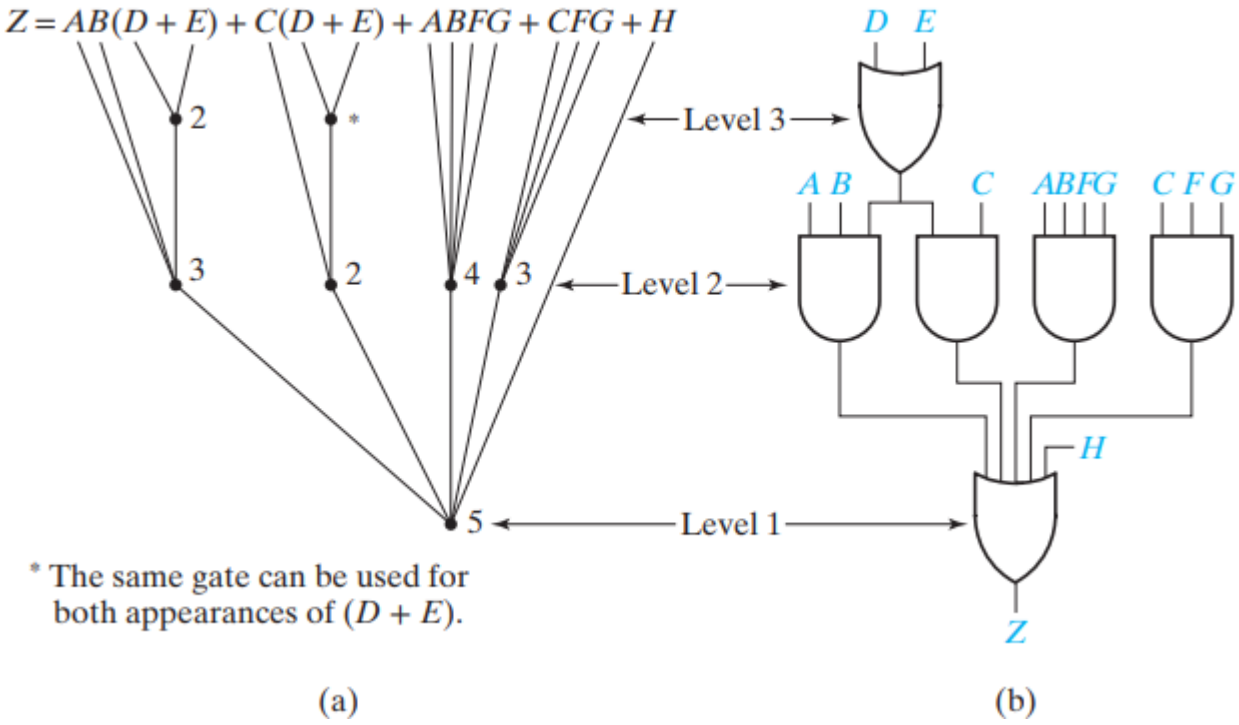


Figure 7.2 Three-Level Realization of Z

EXAMPLE 7 - 1

Find a circuit of AND and OR gates to realize:

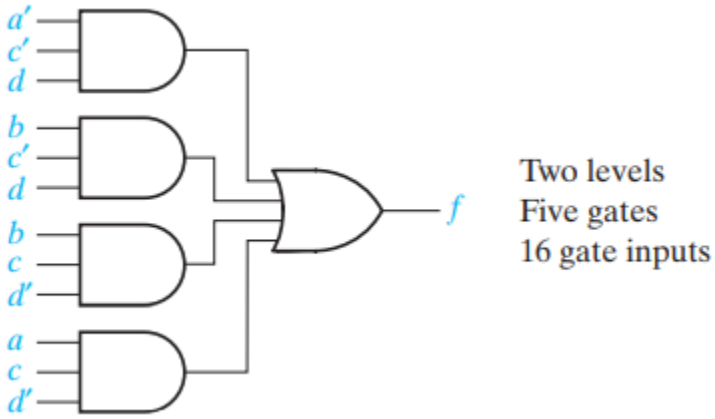
$$f(a,b,c,d) = \Sigma m(1,5,6,10,13,14)$$

Consider solutions with two levels of gates and three levels of gates. Try to minimize the number of gates and the total number of gate inputs.

Solution First, simplify f by using a Karnaugh map:

cd \ ab	00	01	11	10
00	0	0	0	0
01	1	1	1	0
11	0	0	0	0
10	0	1	1	1

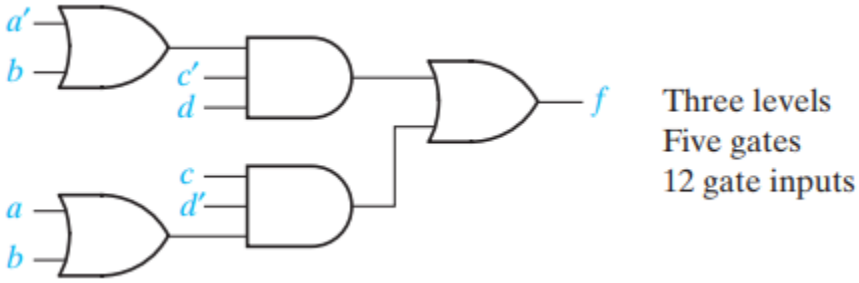
$$f = a'c'd + bc'd + bcd' + acd'$$



Factoring Equation yields:

$$f = c'd\,(a' + b) + cd'\,(a + b)$$

which leads to the following three-level OR-AND-OR gate circuit:



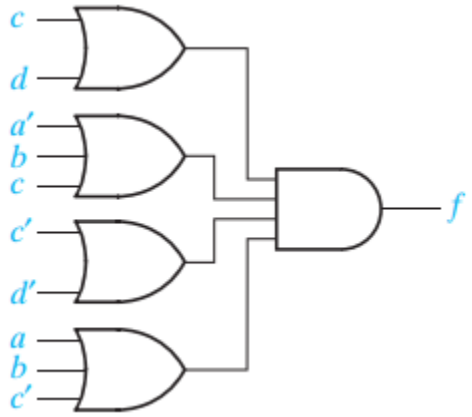
Both solutions have an OR gate at the output. A solution with an AND gate at the output might have fewer gates or gate inputs. A two-level OR-AND circuit corresponds to a product-of-sums expression for the function.

This can be obtained from the 0's on the Karnaugh map as follows:

$$f' = c'd' + ab'c' + cd + a'b'c$$

$$f = (c + d)\,(a' + b + c)\,(c' + d')\,(a + b + c')$$

This equation leads directly to a two-level OR-AND circuit:



Two levels
Five gates
14 gate inputs

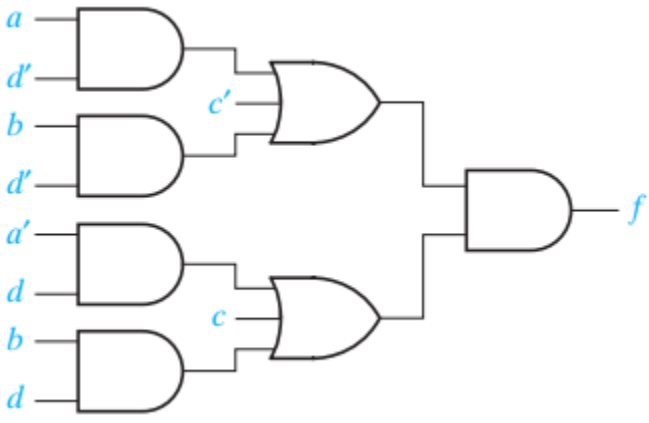
To get a *three-level* circuit with an AND-gate output, we partially multiply out the last equation using $(X + Y)(X + Z) = X + YZ$:

$$f = [c + d(a' + b)][c' + d'(a + b)]$$

This equation would require *four levels* of gates to realize; however, if we multiply out $d'(a + b)$ and $d(a' + b)$, we get:

$$f = (c + a'd + bd)(c' + ad' + bd')$$

which leads directly to a three-level AND-OR-AND circuit:

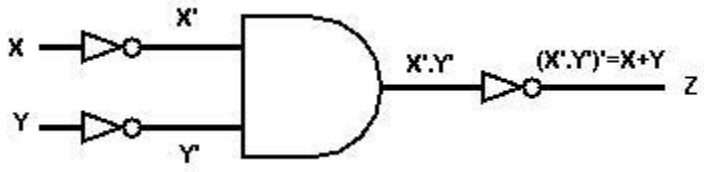


Three levels
Seven gates
16 gate inputs

7.3 Functionally Complete Set of Gates

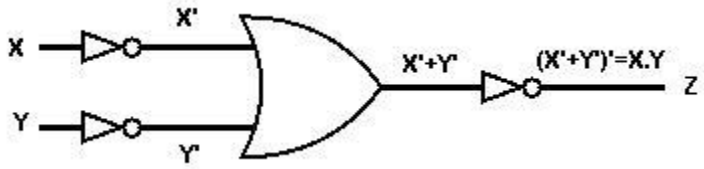
AND and NOT are a *functionally complete set* of gates because OR can be realized using AND and NOT using De-Morgan theorem as the following:

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



Similarly, OR and NOT are a *functionally complete set* of gates because AND can be realized using OR and NOT as the following:

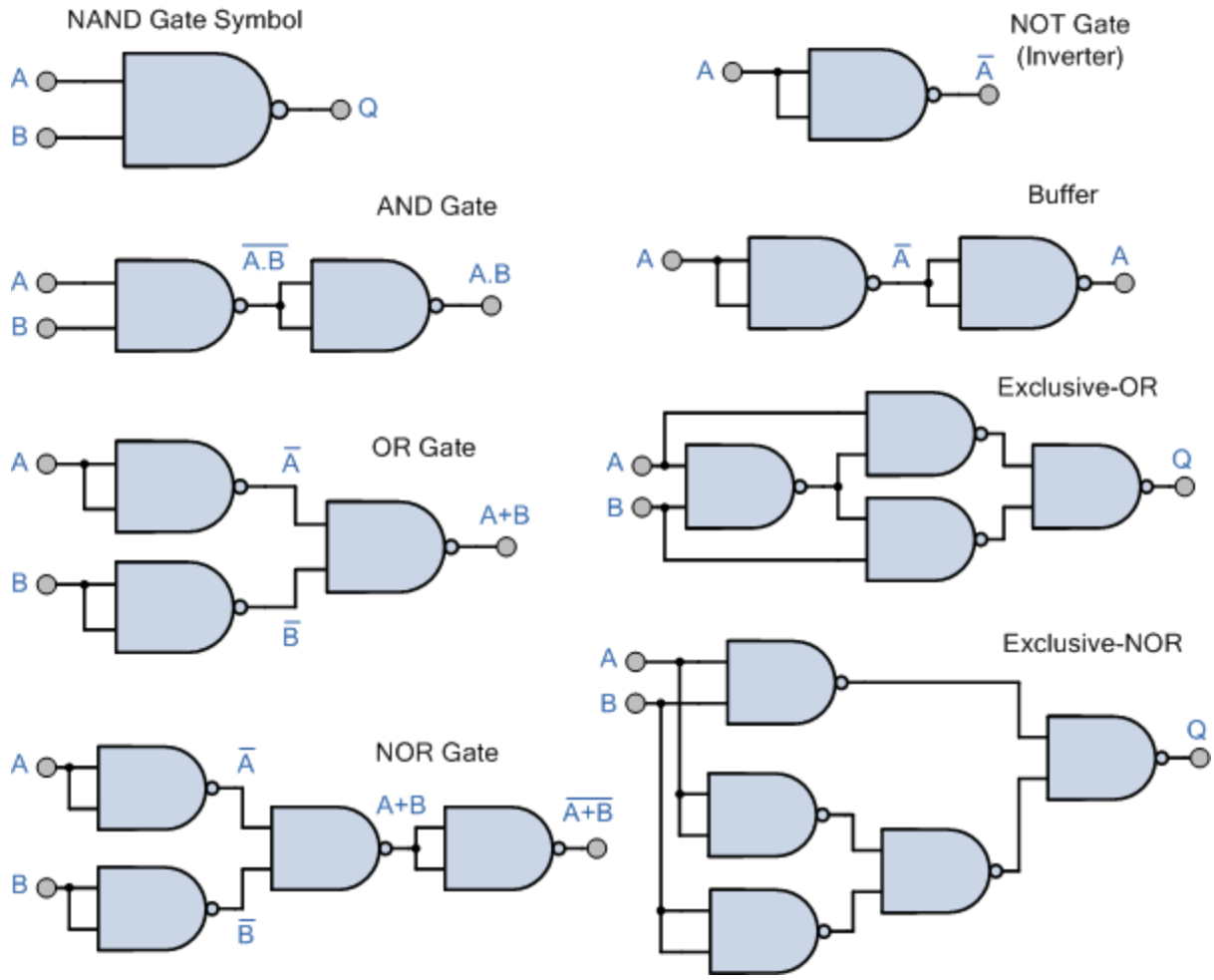
$$X.Y = \overline{\overline{(X.Y)}} = \overline{(\bar{X} + \bar{Y})}$$



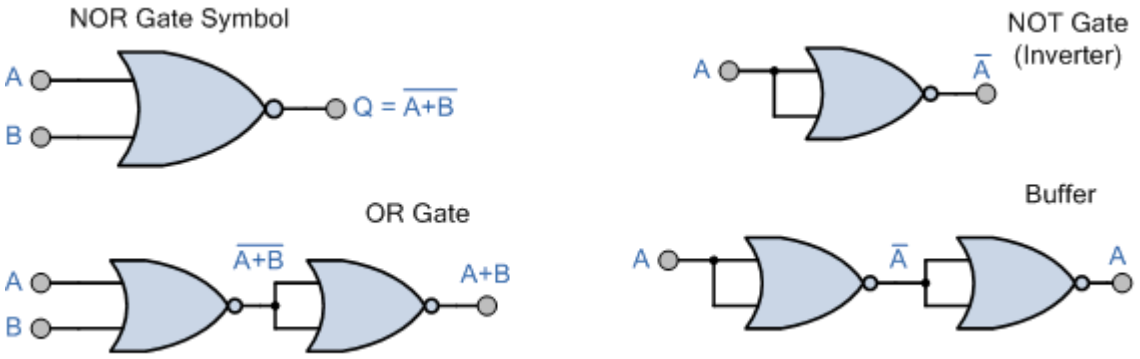
One of the main disadvantages of using the complete sets of AND, OR and NOT gates is that to produce any equivalent logic gate or function we require two (or more) different types of logic gate, AND and NOT, or OR and NOT, or all three as shown above. However, we can realize all the other Boolean functions and gates by using just one single type of universal logic gate, the NAND (NOT AND) or the NOR (NOT OR) gate, thereby reducing the number of different types of logic gates required, and the cost.

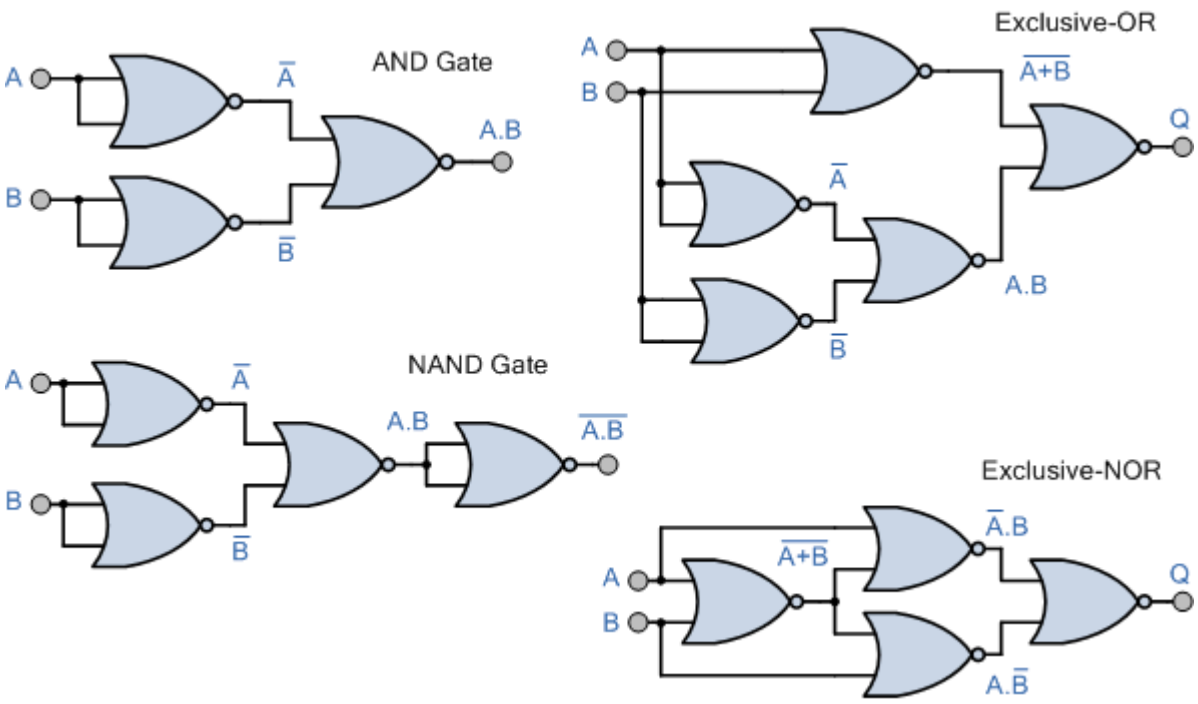
The NAND and NOR gates are the complements of the previous AND and OR functions respectively and are individually a complete set of logic as they can be used to implement any other Boolean function or gate. But as we can construct other logic switching functions using just these gates on their own, they are both called a minimal set of gates. Thus, the NAND and the NOR

gates are commonly referred to as **Universal Logic Gates**. Any function can be realized using only NAND gates:



Universal Logic Gates using only NOR Gates





7.4 Design of Two-Level NAND- and NOR-Gate Circuits

A two-level circuit realizing a function F using various combinations of NAND, NOR, AND, and OR gates are obtained by converting the switching algebra expression for F into the form matching the desired gate circuit.

It is difficult to extend this approach to multiple-level circuits because it requires repeated complementation of parts of the expression for F. The circuit with AND and OR gates is converted to one containing NAND or NOR gates by inserting inverters in pairs to convert each AND and OR gate to a NAND or NOR gate. This approach avoids manipulation of the expression for F and is less error prone.

A *two-level* circuit composed of AND and OR gates is easily converted to a circuit composed of NAND gates or NOR gates. This conversion is carried out by using $F = (F')'$ and then applying DeMorgan's laws:

$$(X_1 + X_2 + \cdots + X_n)' = X_1' X_2' \cdots X_n'$$

$$(X_1X_2 \cdots X_n)' = X_1' + X_2' + \cdots + X_n'$$

The following example illustrates conversion of a minimum sum-of-products form to several other two-level forms:

$$F = \boldsymbol{A} + \boldsymbol{BC'} + \boldsymbol{B'CD} = [(A + BC' + B'CD)']' \quad (7-1)$$

$$= [A' \cdot (BC')' \cdot (B'CD)']' \quad (7-2)$$

$$= [A' \cdot (B' + C) \cdot (B + C' + D')]' \quad (7-3)$$

$$= \boldsymbol{A} + (\boldsymbol{B'} + \boldsymbol{C})' + (\boldsymbol{B} + \boldsymbol{C'} + \boldsymbol{D'})' \quad (7-4)$$

Equations (7-1), (7-2), (7-3), and (7-4) represent the AND-OR, NAND-NAND, OR-NAND, and NOR-OR forms, respectively, as shown in Figure 7-3. Rewriting Equation (7-4) in the form:

$$F = ([A + (B' + C)' + (B + C' + D')']')' \quad (7-5)$$

leads to a *three-level* NOR-NOR-INVERT circuit. However, if we want a two-level circuit containing only NOR gates, we should start with the minimum product of sums form for F instead of the minimum SOP. After obtaining the minimum POS from a Karnaugh map, F can be written in the following two-level forms:

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

$$= \{[(A + B + C)(A + B' + C')(A + C' + D)]'\}' \quad (7-6)$$

$$= [(A + B + C)' + (A + B' + C')' + (A + C' + D)']' \quad (7-7)$$

$$= (A'B'C' + A'BC + A'CD')' \quad (7-8)$$

$$= (A'B'C')' \cdot (A'BC)' \cdot (A'CD')' \quad (7-9)$$

Equations (7-6), (7-7), (7-8), and (7-9) represent the OR-AND, NOR-NOR, AND-NOR, and NAND-AND forms, respectively, as shown in Figure 7-3. Two level AND-NOR (AND-OR-INVERT) circuits are available in integrated-circuit form. Some types of NAND gates can also realize AND-NOR circuits when the so-called wired OR connection is used.

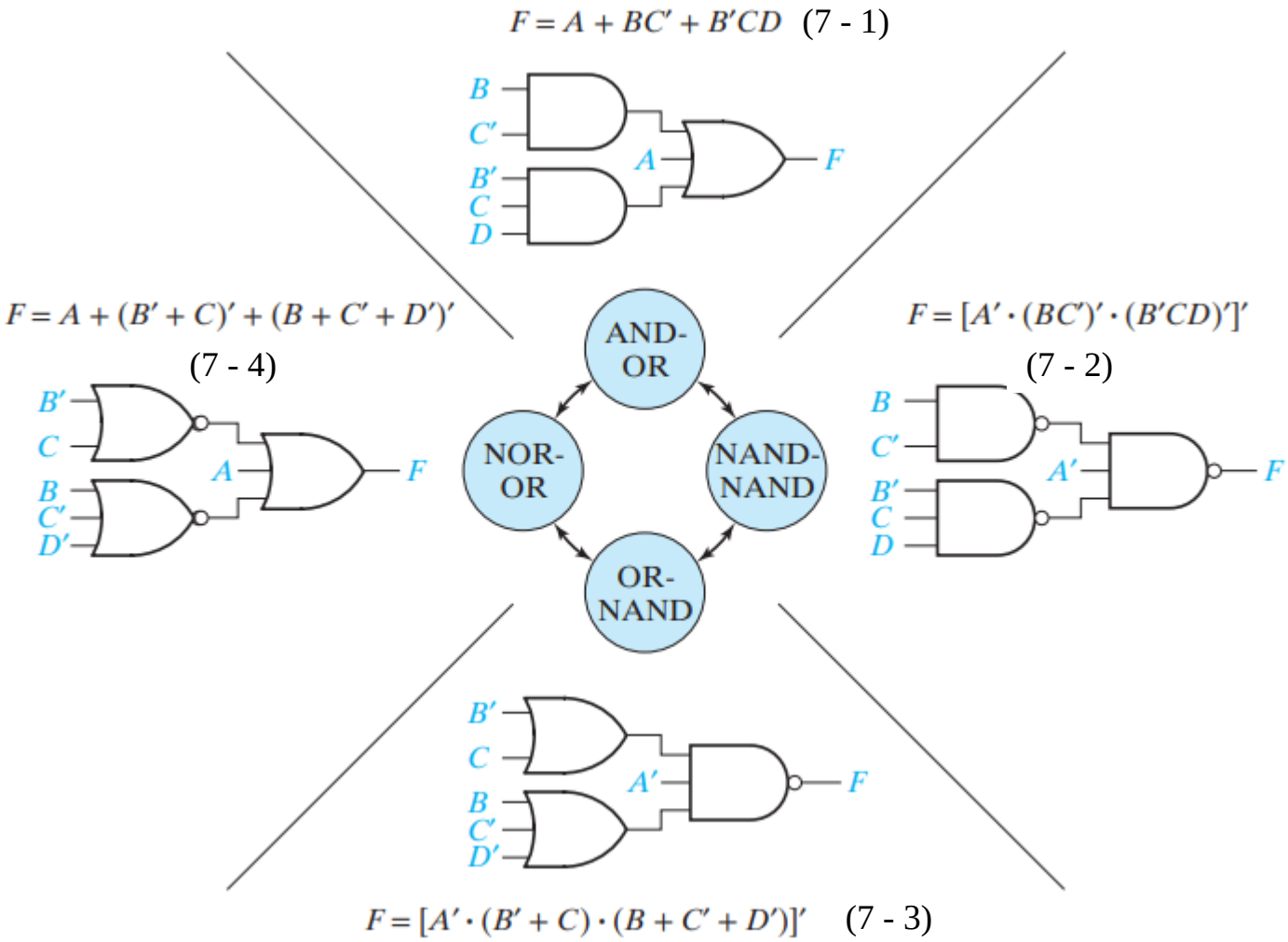
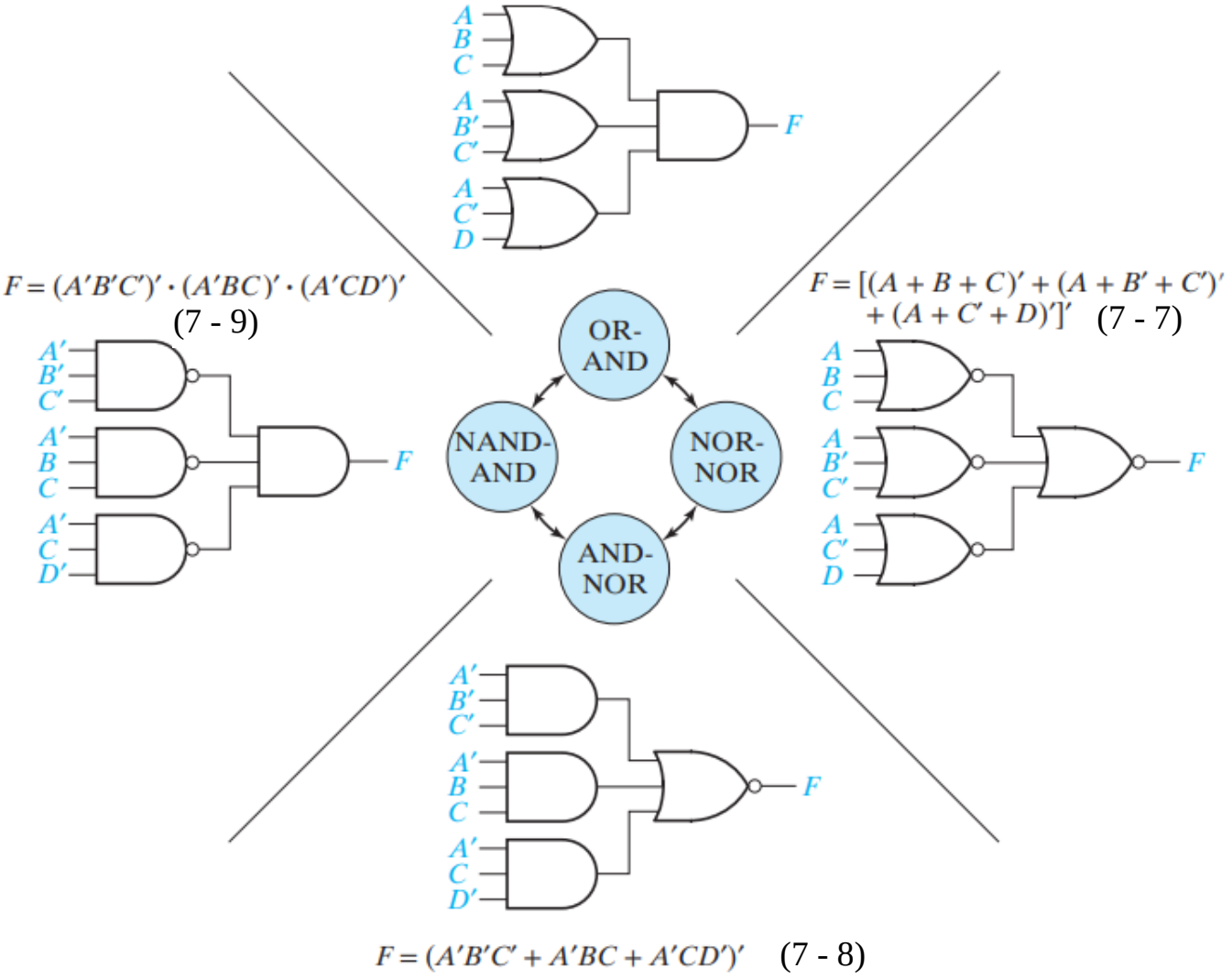


Figure 7.3 Four Basic Forms for Two-Level Circuits

$$F = (A + B + C)(A + B' + C')(A + C' + D) \quad (7 - 6)$$



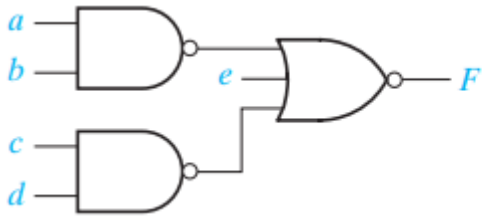
$$F = (A'B'C')' \cdot (A'BC)' \cdot (A'CD')' \quad (7 - 9)$$

$$F = [(A + B + C)' + (A + B' + C')' + (A + C' + D)']' \quad (7 - 7)$$

$$F = (A'B'C' + A'BC + A'CD')' \quad (7 - 8)$$

Figure 7.4 Four Basic Forms for Two-Level Circuits

The other eight possible two-level forms (AND-AND, OR-OR, OR-NOR, AND-NAND, NAND-NOR, NOR-NAND, etc.) are degenerate in the sense that they cannot realize all switching functions. Consider, for example, the following NAND-NOR circuit:



$$F = [(ab)' + (cd)' + e]' = abcde'$$

From this example, it is clear that the NAND-NOR form can realize only a product of literals and not a sum of products. Because NAND and NOR gates are readily available in integrated circuit form, two of the most used circuit forms are the NAND-NAND and the NOR-NOR. Assuming that all variables and their complements are available as inputs, the following method can be used to realize *F* with NAND gates:

Procedure for designing a minimum two-level NAND-NAND circuit:

1. Find a minimum sum-of-products expression for *F*.
2. Draw the corresponding two-level AND-OR circuit.
3. Replace all gates with NAND gates leaving the gate interconnections unchanged. If the output gate has any single literals as inputs, complement these literals.

Assuming that all variables and their complements are available as inputs, the following method can be used to realize *F* with NOR gates:

Procedure for designing a minimum two-level NOR-NOR circuit:

1. Find a minimum product-of-sums expression for *F*.
2. Draw the corresponding two-level OR-AND circuit.
3. Replace all gates with NOR gates leaving the gate interconnections unchanged. If the output gate has any single literals as inputs, complement these literals.

7.5 Design of Multi-Level NAND- and NOR-Gate Circuits

The following procedure may be used to design multi-level NAND-gate circuits:

1) Simplify the switching function to be realized.

2) Design a multi-level circuit of AND and OR gates. The output gate must be OR. AND-gate outputs cannot be used as AND-gate inputs; OR-gate outputs cannot be used as OR-gate inputs.

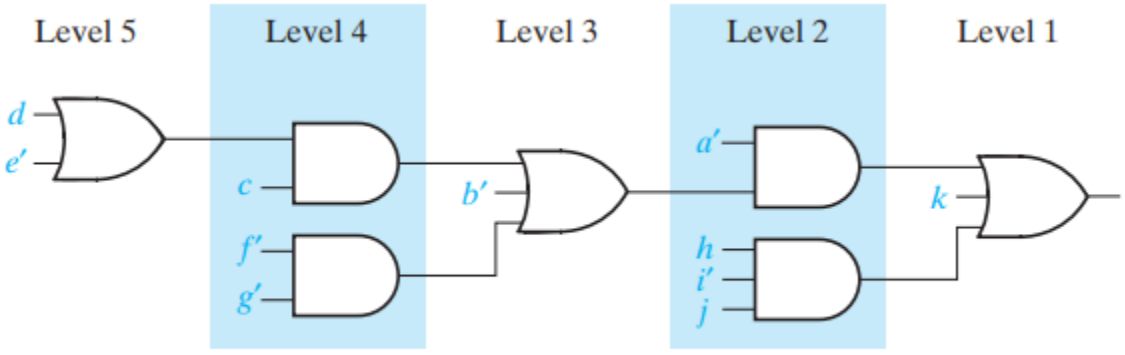
3) Number the levels starting with the output gate as level 1. Replace all gates with NAND gates, leaving all interconnections between gates unchanged. Leave the inputs to levels 2, 4, 6, . . . unchanged. Invert any literals which appear as inputs to levels 1, 3, 5, ...

The validity of this procedure is easily proven by dividing the multi-level circuit into two-level subcircuits and applying the previous results for two-level circuits to each of the two-level subcircuits. The example of the following figure illustrates the procedure. Note that if step 2 is performed correctly, each level of the circuit will contain only AND gates or only OR gates.

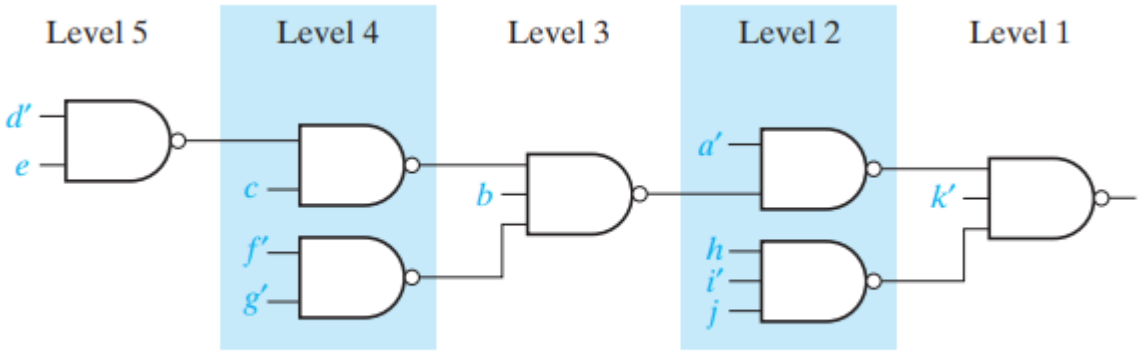
The procedure for the design of multi-level NOR-gate circuits is the same as for NAND-gate circuits except the output gate of the circuit of AND and OR gates must be an AND gate, and all gates are replaced with NOR gates. Suppose:

$$\boldsymbol{F = a'[b' + c(d + e') + f'g'] + hi'j + k}$$

The following figure shows how the AND-OR circuit for *F* is converted to the corresponding NAND circuit.



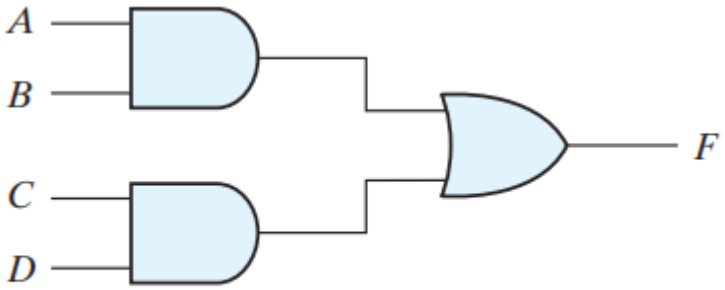
(a) AND-OR network



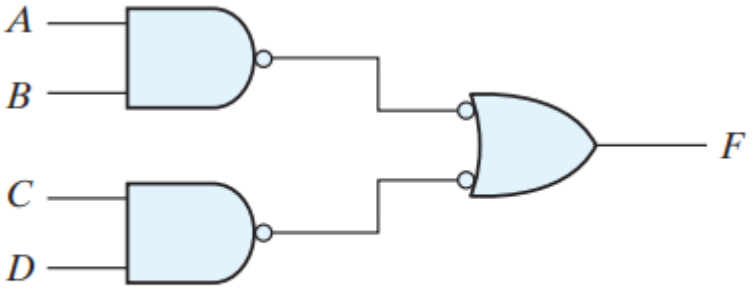
(b) NAND network

7.6 Summary

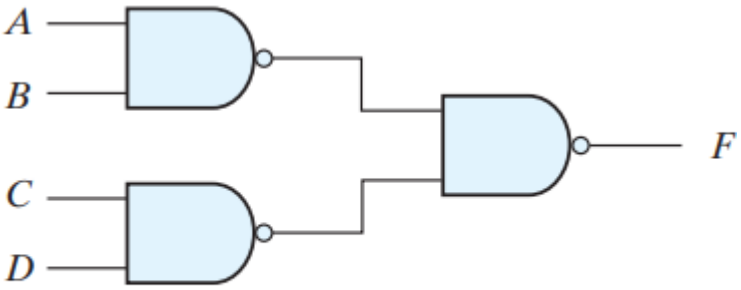
1. $F = AB + CD$



(a) **(AND-OR)**



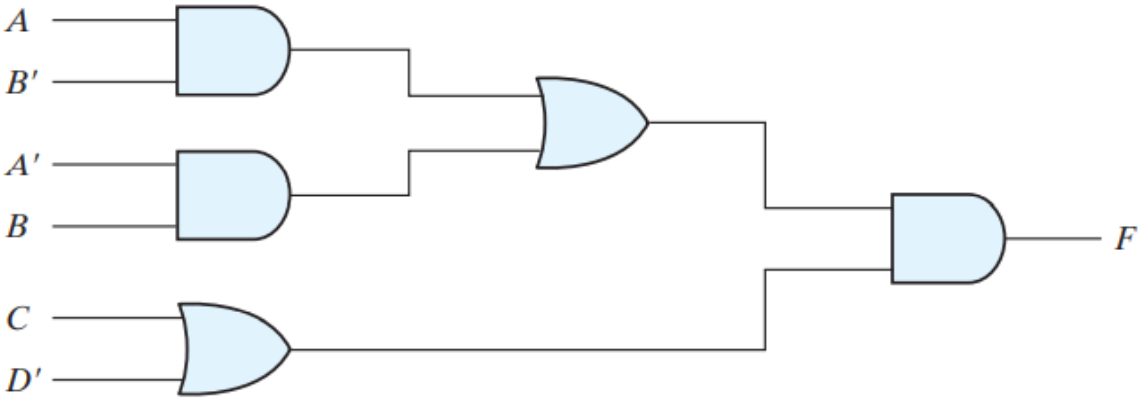
(b) **(NAND-OR)**



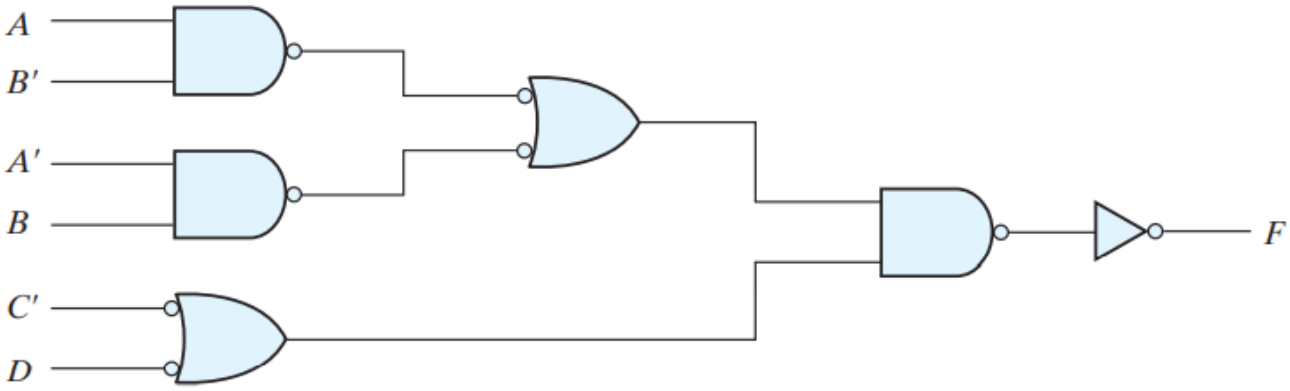
(c) **(NAND-NAND)**

Figure 7.5 Three ways to implement *F = AB + CD*

2. $\boldsymbol{F = (AB' + A'B)(C + D')}$



(a) AND–OR gates



(b) NAND gates

Figure 7.6 Implementing *F* = (*AB′* + *A′B*) (*C* + *D′*)

EXAMPLE 7 - 2

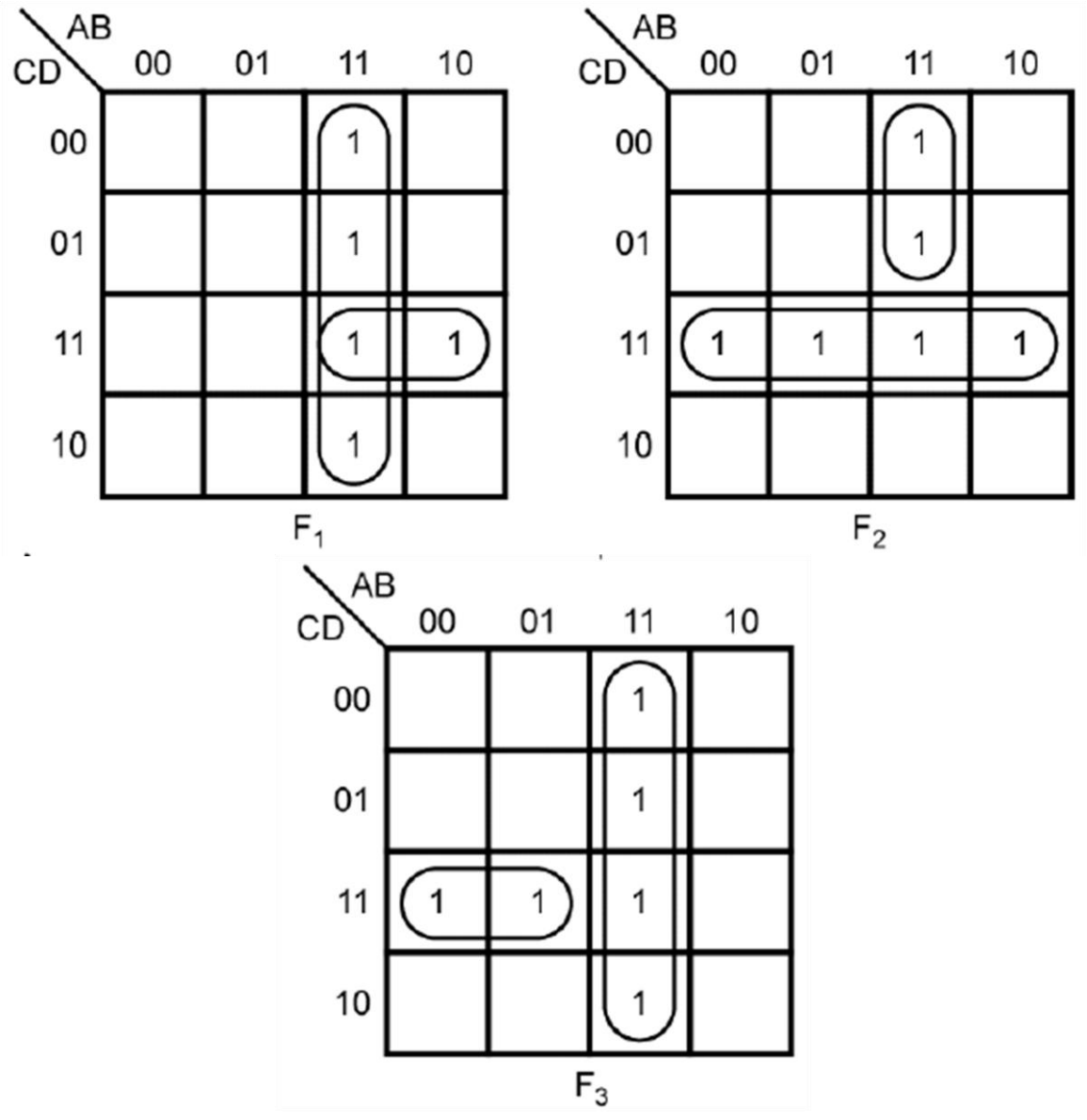
Design a circuit with four inputs and three outputs which realizes the functions:

$$F_1(A,B,C,D) = \Sigma\, m(11,12,13,14,15)$$

$$F_2(A,B,C,D) = \Sigma\, m(3,7,11,12,13,15)$$

$$F_3(A,B,C,D) = \Sigma\, m(3,7,12,13,14,15)$$

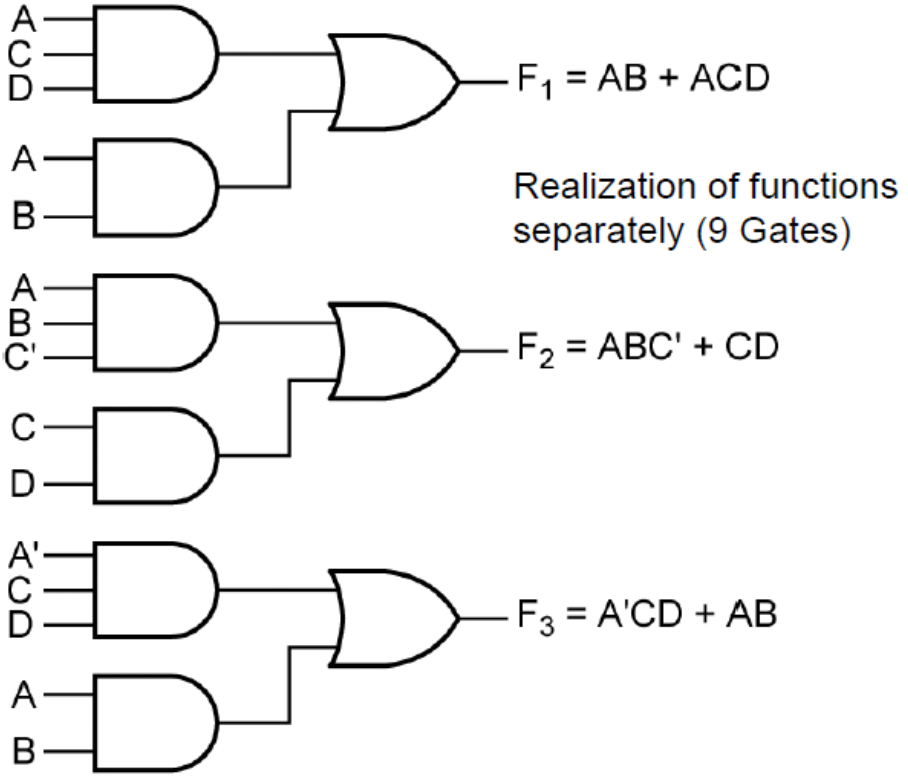
Solution First, simplify F_1, F_2, F_3 by using a Karnaugh map:

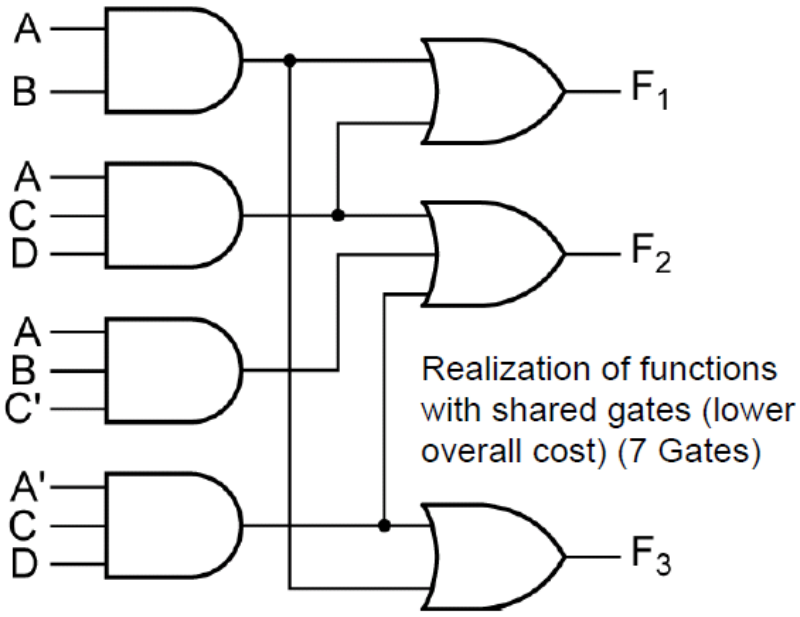


$$F_1 = AB + ACD$$

$$F_2 = ABC' + CD$$

$$F_3 = A'CD + AB$$





Problems

1. Draw and explain the connections required to construct:

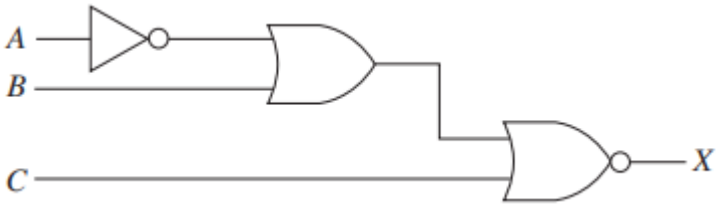
a. An OR gate from two NOR gates.

b. An AND gate from two NAND gates.

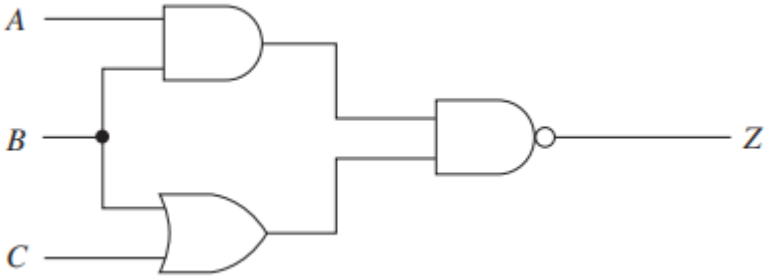
c. An AND gate from several NOR gates.

d. A NOR gate from several NAND gates.

2. Redraw the logic circuits of the following figures to their equivalents using only NOR gates:



(A)



(B)

3. Using AND and OR gates, find a minimum circuit to realize:

$$f(a,b,c,d) = m_4 + m_6 + m_7 + m_8 + m_9 + m_{10}$$

(a) using two-level logic.

(b) using three-level logic (12 gate inputs minimum).

4. **Realize the following functions using AND and OR gates. Assume that there are no restrictions on the number of gates which can be cascaded and minimize the number of gate inputs.**

   $(a)\ AC'D + ADE' + BE' + BC' + A'D'E'.$

   $(b)\ AE + BDE + BCE + BCFG + BDFG + AFG.$

5. **Find a minimum three-level NAND-gate circuit to realize F (A, B, C, D) = Σ m (5, 10, 11, 12, 13) (four gates).**

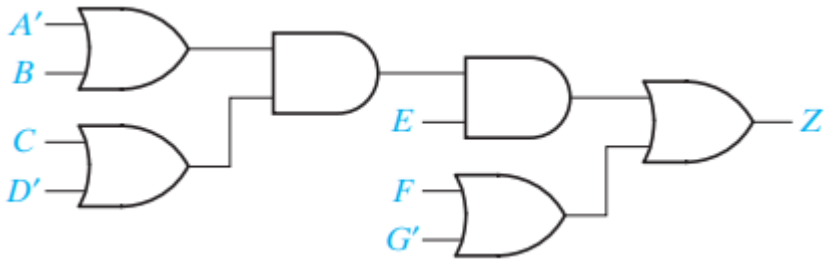
6. **Realize Z = A'D + A'C + AB'C'D' using four NOR gates.**

7. **Realize Z = ABC + AD + C'D' using only two-input NAND gates. Use as few gates as possible.**

8. **Realize Z = AE + BDE + BCEF using only two-input NOR gates. Use as few gates as possible.**

9. **(a) Convert the following circuit to all NAND gates, by adding bubbles and inverters where necessary.**

   (b)Convert to all NOR gates (an inverter at the output is allowed).



10. **Using AND and OR gates, find a minimum two-level circuit to realize:**

    (a) $F = a'c + bc'd + ac'd.$

(b) $F = (b' + c)(a + b' + d)(a + b + c' + d)$.

11. Implement f (x, y, z) = Σ m (0, 1, 3, 4, 7) as a two-level gate circuit, using a minimum number of gates.

(a) Use AND gates and NAND gates.

(b) Use NAND gates only.

12. Implement f (a, b, c, d) = Σ m (3, 4, 5, 6, 7, 11, 15) as a two-level gate circuit, using a minimum number of gates.

(a) Use OR gates and NOR gates.

(b) Use NOR gates only.

(c) Use AND gates and NAND gates.

(d) Use OR gates and NAND gates.

(e) Use NAND gates only.

اسم الطالب: .....
رقم الطالب: .....
اسم المقرر: .....
رقم الشيت: .....

الرقم السري: .....

استمارة التقويم المستمر

Question No.	Degree	Question No.	Degree	Signature
1.		2.		
3.		4.		
5.		6.		
7.		8.		
9.		10.		
11.		12.		

12

8

The Exclusive-OR Gate and Exclusive-NOR Gate

Binary Adder–Subtractor

OUTLINE

8.1 INTRODUCTION

We have seen in the previous chapters that by using various combinations of the basic gates, we can form almost any logic function that we need. Often, a particular combination of logic gates provides a function that is especially useful for a wide variety of tasks. In this chapter, we learn about and design systems using two new combinational logic gates: the *exclusive-OR* and the *exclusive-NOR*.

8.2 The Exclusive-OR Gate

Remember, a *two*-input OR gate provides a HIGH output if one input or the other input is HIGH or if both inputs are HIGH. The *exclusive-OR*, however, provides a HIGH output if one input or the other input is HIGH, but not both. This point is made clearer by comparing the truth tables for a two-input OR gate versus an exclusive-OR gate, as shown in Table 8–1.

TABLE Truth Tables for an OR Gate versus an Exclusive-OR Gate

A	*B*	*X*	*A*	*B*	*X*
0	0	0	0	0	0
0	1	1	0	1	1
1	0	1	1	0	1
1	1	1	1	1	0
	(OR)			(Exclusive-OR)	

The Boolean equation for the **XOR** function is written $X = \bar{A}B + A\bar{B}$ and can be constructed using the combinational logic shown in Figure 8–1. By experimenting and using Boolean reduction, we can find several other combinations of the basic gates that provide the XOR function. For example,

the combination of AND, OR, and NAND gates shown in Figure 8–2 will reduce to the one or the other but not both (XOR) function.

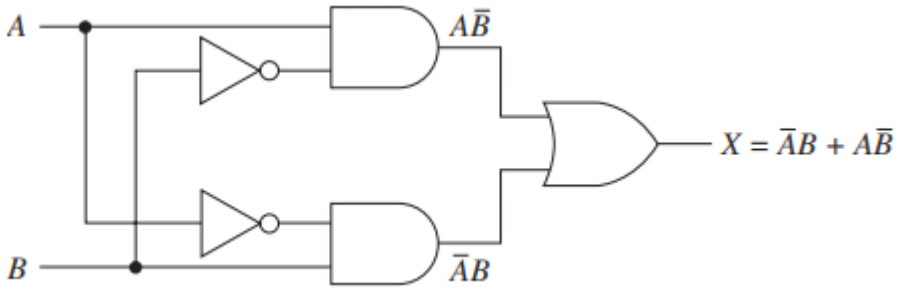


Figure 8-1 Logic circuit for providing the exclusive-OR function.

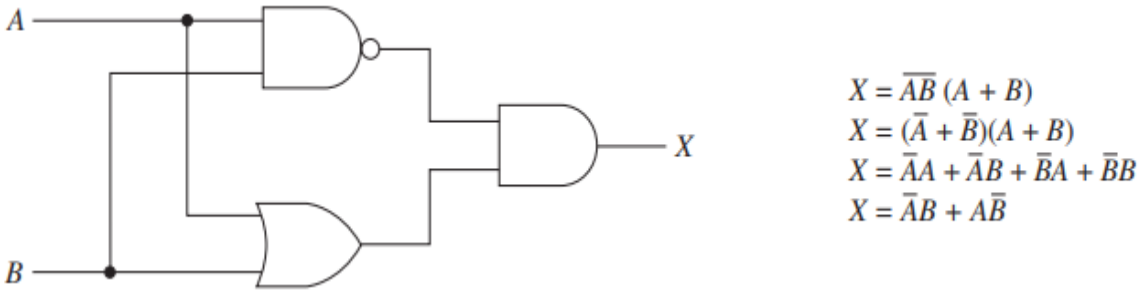


Figure 8-2 Exclusive-OR built with an AND-OR-NAND combination.

The **XOR** gate is common enough to deserve its own logic symbol and equation, as shown in Figure 8–3. (Note the shorthand method of writing the Boolean equation is to use a plus sign with a circle around it.)

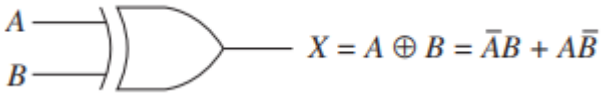


Figure 8-3 Logic symbol and equation for the Exclusive-OR.

8.3 The Exclusive-NOR Gate

The exclusive-NOR (XNOR) is the complement of the exclusive-OR (XOR). A comparison of the truth tables in Table 8–2 illustrates this point. The truth table for the **XNOR** shows a HIGH output for both inputs LOW or both inputs HIGH. The XNOR is sometimes called the equality gate because both inputs

must be equal to get a HIGH output. The basic logic circuit and symbol for the XNOR are shown in Figure 8–4.

TABLE Truth Tables of the Exclusive-NOR versus the Exclusive-OR

$X = AB + \overline{A}\,\overline{B}$ (Exclusive-NOR)

A	B	X
0	0	1
0	1	0
1	0	0
1	1	1

$X = \overline{A}B + A\overline{B}$ (Exclusive-OR)

A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

← One or the other, but not both

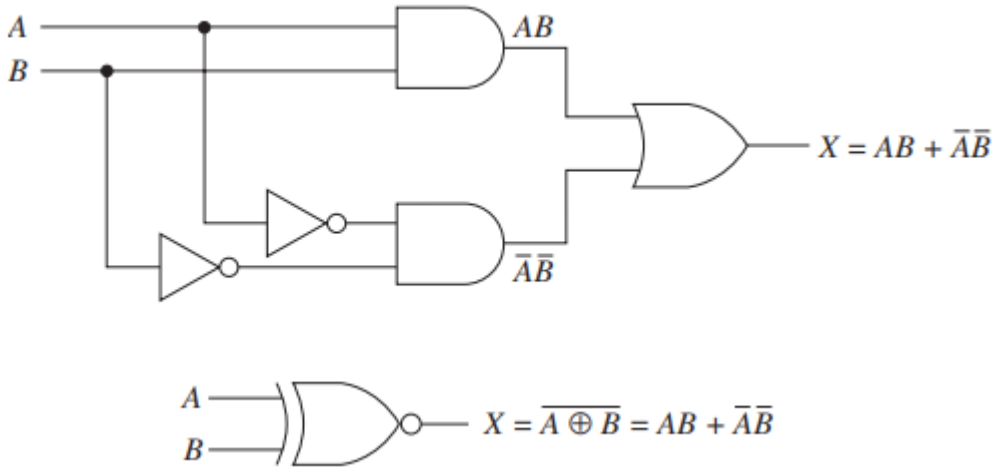
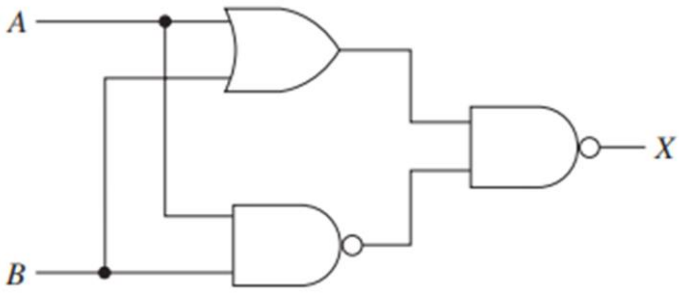


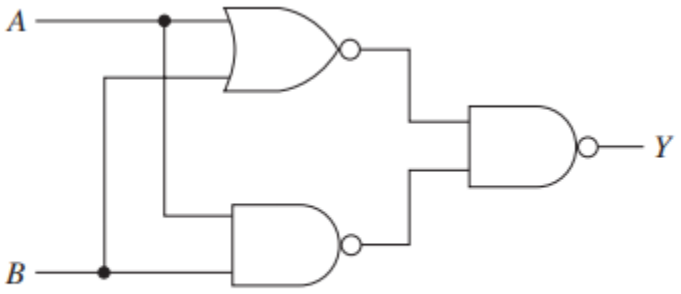
Figure 8-4 Exclusive-NOR logic circuit and logic symbol.

EXAMPLE 8 - 1

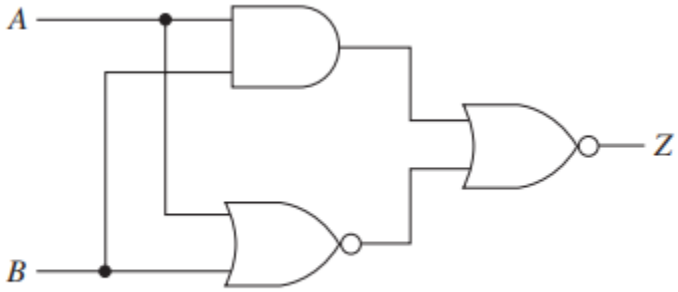
Determine for each circuit shown in the following figures if its output provides the XOR function, the XNOR function, or neither.



(a)



(b)



(c)

Solution

(a) $X = \overline{(A + B)\overline{AB}}$

$= \overline{\overline{A} + \overline{B}} + \overline{\overline{AB}}$

$= \overline{A}\,\overline{B} + AB$ ← Ex-NOR

(b) $Y = \overline{\overline{A + B}\,\overline{AB}}$

$= \overline{\overline{A + B}} + \overline{\overline{AB}}$

$= A + B + AB$

$= A + B(1 + A)$

$= A + B$ ← neither (OR function)

(c) $Z = \overline{AB + \overline{A + B}}$

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

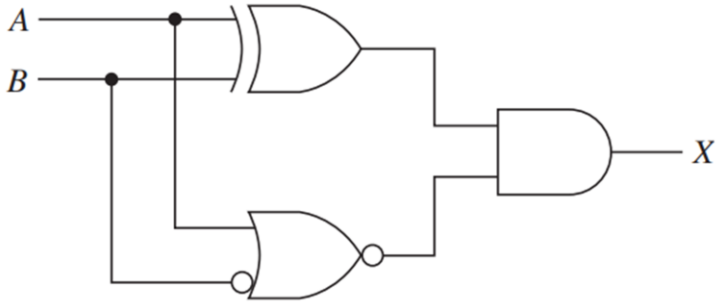
$= (\overline{A} + \overline{B})(A + B)$

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

$= \overline{A}B + A\overline{B}$ ← Ex-OR

EXAMPLE 8 - 2

Write the Boolean equation for the circuit shown in the following figure and simplify.

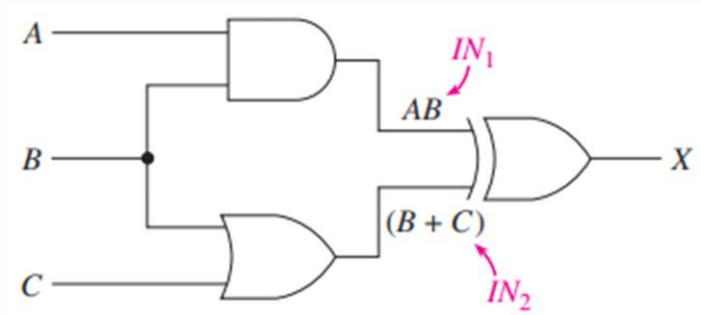


Solution

$$\begin{aligned} X &= (\overline{A}B + A\overline{B})\overline{A + \overline{B}} \\ &= (\overline{A}B + A\overline{B})\overline{A}\,\overline{\overline{B}} \\ &= \overline{A}B\overline{A}B + A\overline{B}\,\overline{A}B \\ &= \overline{A}B \end{aligned}$$

EXAMPLE 8 - 3

Write the Boolean equation for the circuit shown in the following figure and simplify.



Solution

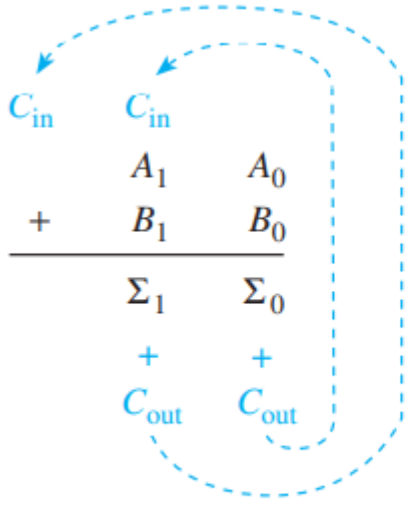
$$\begin{aligned} X &= \overline{A}\overline{B}(B + C) + AB(\overline{B + C}) \\ &= (\overline{A} + \overline{B})(B + C) + AB\overline{B}\,\overline{C} \\ &= \overline{A}B + \overline{A}C + \overline{B}B + \overline{B}C \\ &= \overline{A}B + \overline{A}C + \overline{B}C \end{aligned}$$

8.4 INTRODUCTION

Adders are important in computers and in other types of digital systems in which numerical data are processed. An understanding of the basic adder operation is fundamental to the study of digital systems. In this chapter, the half-adder and the full adder are introduced.

8.5 Basic Adder Circuit

By reviewing the truth table in Figure 8–5, we can determine the input conditions that produce each combination of *sum* and *carry* output bits. Figure 8–5 shows the addition of two 2-bit numbers. This could easily be expanded to cover 4-, 8-, or 16-bit addition. Notice that addition in the least-significant-bit (LSB) column requires analyzing only two inputs ($A_0 + B_0$) to determine the output *sum* (Σ_0) and *carry* (C_{out}), but any more significant columns (2^1 column and up) require the inclusion of a third input, which is the *carry-in* (C_{in}) from the column to its right.



(a)

2 inputs 2 outputs

A_0	B_0	Σ_0	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

(b)

3 inputs			2 outputs	
A_1	B_1	C_{in}	Σ_1	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(c)

Figure 8–5 (a) Addition of two 2-bit binary numbers; (b) Truth table for the LSB addition; (c) Truth table for the more significant column.

For example, the *carry-out* (C_{out}) of the 2^0 column becomes the *carry-in* ($_{Cin}$) to the 2^1 column. Figure 8–5(c) shows the inclusion of a third input for the truth table of the more significant column additions.

8.6 Half-Adder

Designing logic circuits to automatically implement the desired outputs for these truth tables is simple. Look at the LSB truth table; for what input conditions is the bit HIGH? The answer is *A* or *B* HIGH but not both (*exclusive-OR function*).

For what input condition is the C_{out} bit HIGH? The answer is *A* and *B* HIGH (*AND function*). Therefore, the circuit design to perform addition in the LSB column can be implemented using an *exclusive-OR* and an *AND* gate. That circuit is called a ***half-adder*** and is shown in Figure 8–6. A ***half-adder*** adds two bits and produces a sum and an output carry.

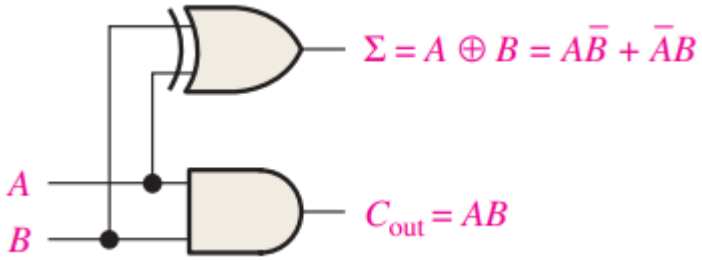


Figure 8–6 Half-adder circuit for addition in the LSB column

If the *exclusive-OR* function in Figure 8–6 is implemented using an *AND–NOR–NOR* configuration, we can tap off the AND gate for the carry, as shown in Figure 8–7.

$$\begin{aligned}
\Sigma_0 &= \overline{A_0}B_0 + A_0\overline{B_0} \\
&= \overline{\left(\overline{\overline{A_0}B_0 + A_0\overline{B_0}}\right)} \\
&= \overline{\overline{\overline{A_0}B_0}.\overline{A_0\overline{B_0}}} \\
&= \overline{(A_0 + \overline{B_0})(\overline{A_0} + B_0)} \\
&= \overline{(A_0B_0 + \overline{A_0}.\overline{B_0})} \\
&= \overline{\left(\boldsymbol{A_0B_0} + \overline{(\boldsymbol{A_0} + \boldsymbol{B_0})}\right)}
\end{aligned}$$

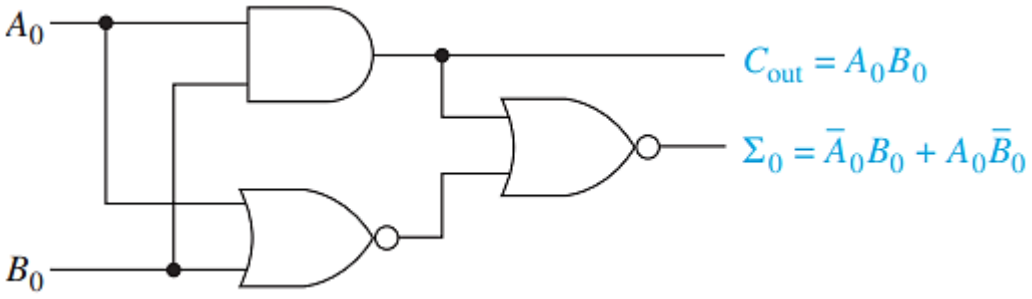


Figure 8–7 Alternative half-adder circuit built from an AND–NOR–NOR configuration.

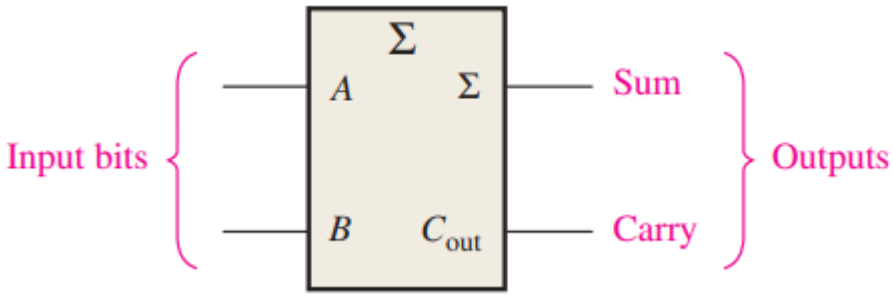


Figure 8–8 Logic symbol for a half-adder.

8.7 Full-Adder

As you can see in Figure 8–5, addition in the 2^1 (or higher) column requires three inputs to produce the *sum* (Σ_0) and *carry* (C_{out}) outputs. Look at the truth table [Figure 8–5(c)]; for what input conditions is the *sum* output HIGH? The answer is that the bit is HIGH whenever the three inputs (A_1, B_1, *0*) are odd. The *full adder* must add the two input bits and the input carry. From the *half-adder*, you know that the *sum* of the input bits *A* and *B* is the *exclusive-OR* of those two variables, $A \oplus B$. For the input carry (C_{in}) to be added to the input bits, it must be *exclusive-ORed* with $A \oplus B$, yielding the equation for the sum output of the full-adder.

$$\Sigma = (A \oplus B) \oplus C_{in}$$

This means that to implement the *full-adder* sum function, two 2-input exclusive-OR gates can be used. The first must generate the term $A \oplus B$, and the second has as its inputs the output of the first XOR gate and the input carry, as illustrated in Figure 8–9.

The output carry is a 1 when both inputs to the first XOR gate are 1s or when both inputs to the second XOR gate are 1s. You can verify this fact by studying Figure 8–5(c). The output carry of the full-adder is therefore produced by input *A* ANDed with input *B* and $A \oplus B$ ANDed with C_{in}. These two terms are

ORed, as expressed in the following Equation. This function is implemented and combined with the sum logic to form a complete full-adder circuit, as shown in Figure 8–10.

$$C_{out} = AB + (A \oplus B)\, C_{in}$$

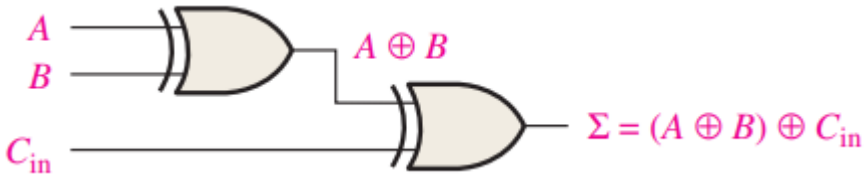


Figure 8–9 Logic required to form the sum of three bits

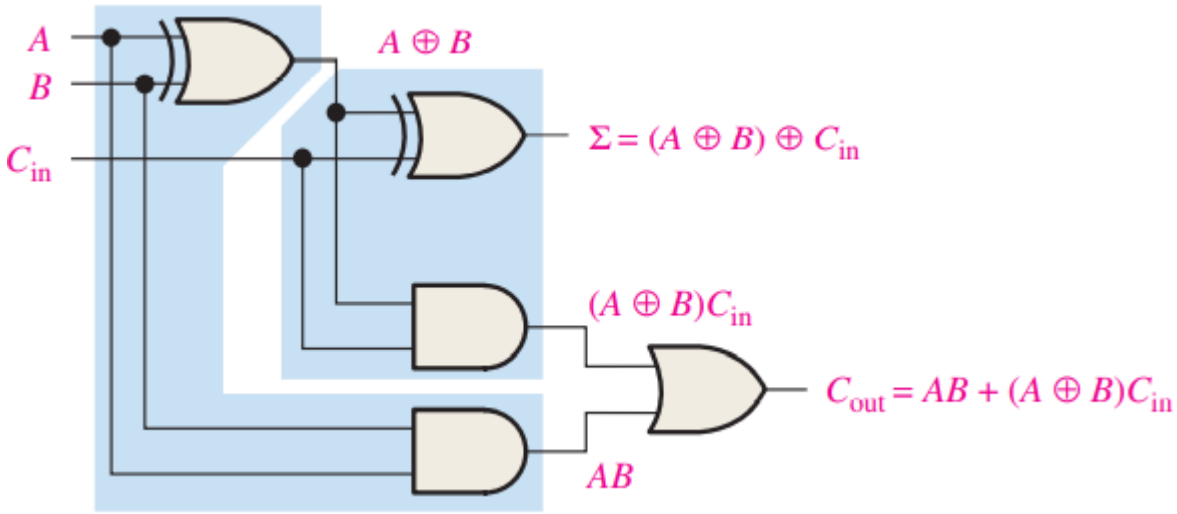


Figure 8–10 Full-adder logic

Notice in Figure 8–10 there are two half-adders, connected as shown in the block diagram of Figure 8–11, with their output carries ORed. The logic symbol shown in Figure 8–12 will normally be used to represent the full-adder. The basic difference between a full-adder and a half-adder is that the full-adder accepts an input carry.

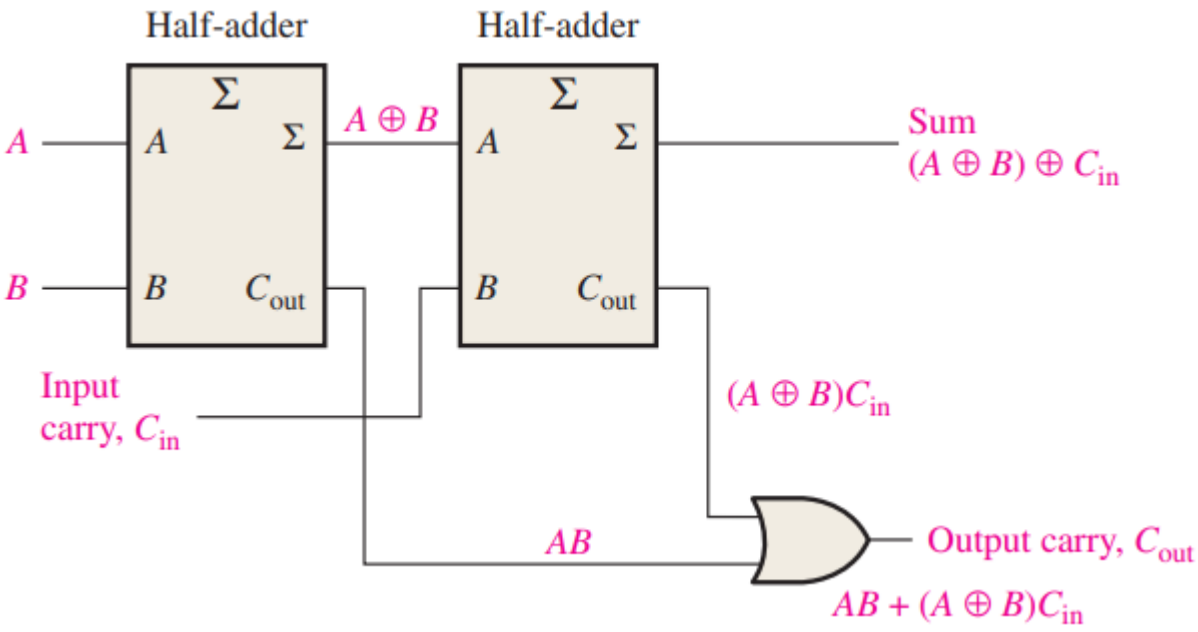


Figure 8–11 Arrangement of two half-adders to form a full-adder.

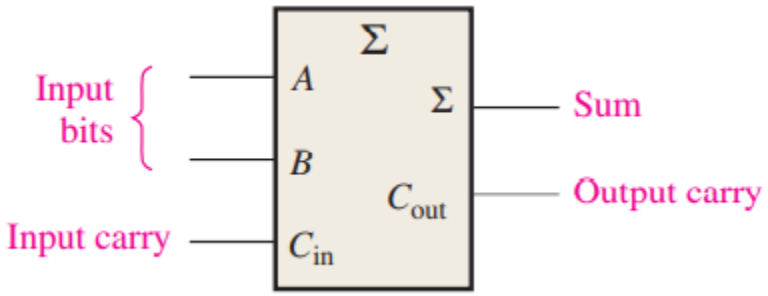
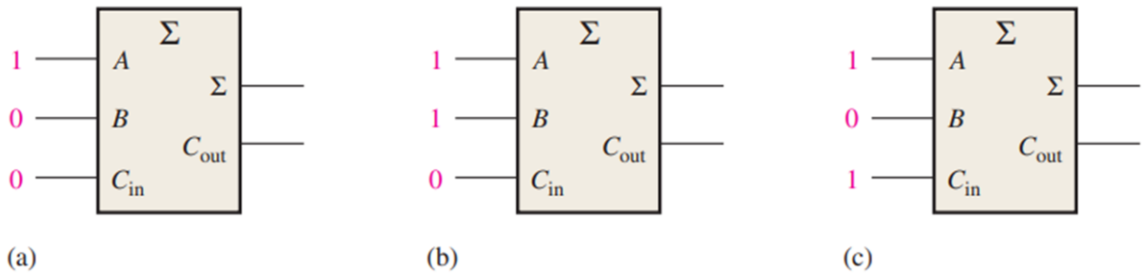


Figure 8–12 Logic symbol for a full-adder.

EXAMPLE 8 - 4

For each of the three full adders in the following Figure, determine the outputs for the inputs shown.



Solution

(a) The input bits are $A = 1$, $B = 0$, and $C_{in} = 0$.

$$1 + 0 + 0 = 1 \text{ with no carry}$$

Therefore, $\Sigma = 1$ and $C_{out} = 0$.

(b) The input bits are $A = 1$, $B = 1$, and $C_{in} = 0$.

$$1 + 1 + 0 = 0 \text{ with a carry of } 1$$

Therefore, $\Sigma = 0$ and $C_{out} = 1$.

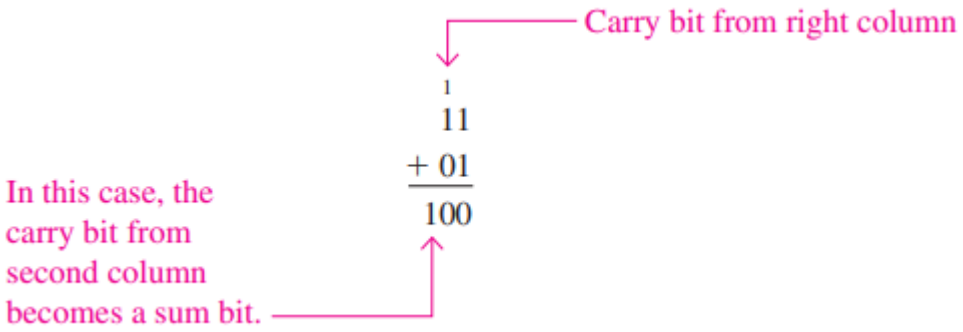
(c) The input bits are $A = 1$, $B = 0$, and $C_{in} = 1$.

$$1 + 0 + 1 = 0 \text{ with a carry of } 1$$

Therefore, $\Sigma = 0$ and $C_{out} = 1$.

8.8 Parallel Binary Adders

Two or more full adders are connected to form *parallel binary adders*. In this section, you will learn the basic operation of this type of adder and its associated input and output functions. A *single full-adder* is capable of adding two 1-bit numbers and an input carry. To add binary numbers with more than one bit, you must use additional full-adders. When one binary number is added to another, each column generates a sum bit and a 1 or 0 carry bit to the next column to the left, as illustrated here with 2-bit numbers.



To add two binary numbers, a *full-adder* (FA) is required for each bit in the numbers. So, for 2-bit numbers, two adders are needed; for 4-bit numbers, four adders are used; and so on. The carry output of each adder is connected to the carry input of the next higher-order adder, as shown in Figure 8–13 for a 2-bit adder. Notice that either a *half-adder* can be used for the least significant position (LSP) or the carry input of a full-adder can be made 0 (grounded) because there is no carry input to the least significant bit position.

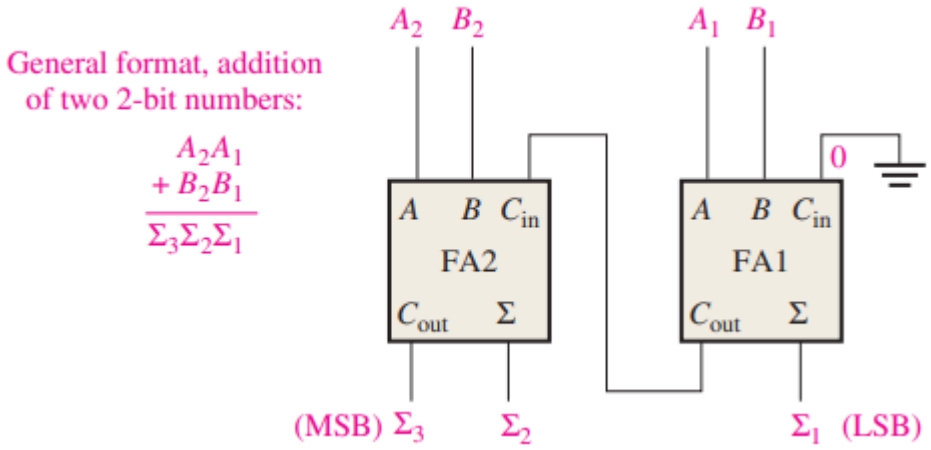


Figure 8–13 Block diagram of a basic 2-bit parallel adder using two full-adders.

A group of four bits is called a *nibble*. A basic *4-bit parallel adder* is implemented with four full-adder stages or three full-adder and one half-adder stages as shown in Figure 8–14 and Figure 8–15, respectively. Again, the LSBs (A_1 and B_1) in each number being added go into the right-most full-adder; the higher-order bits are applied as shown to the successively higher-

order adders, with the MSBs (A_4 and B_4) in each number being applied to the left-most full-adder. The carry output of each adder is connected to the carry input of the next higher-order adder as indicated. These are called *internal carries*.

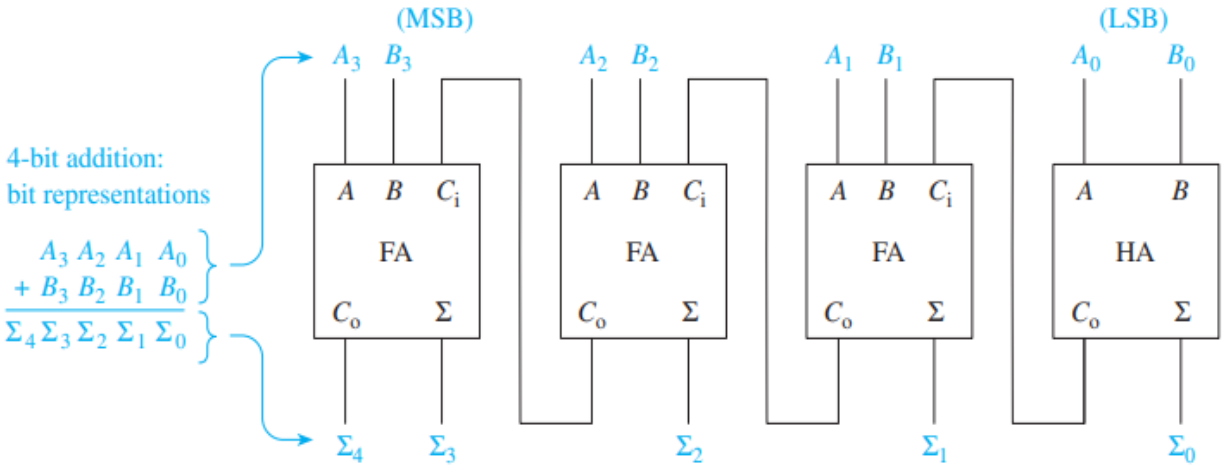


Figure 8–14 A 4-bit parallel adder.

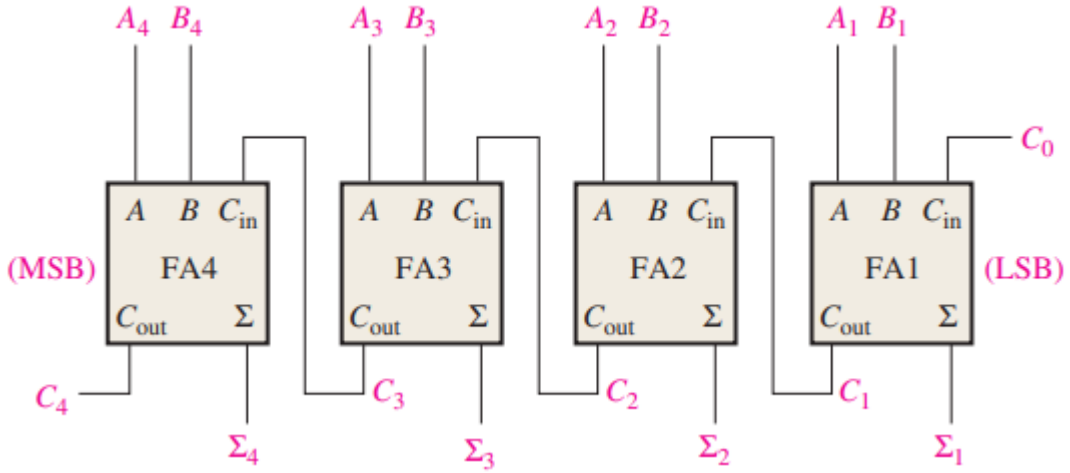


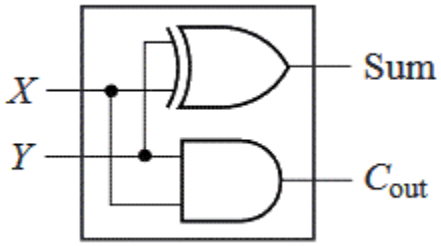
Figure 8–15 Block diagram of a 4-bit binary adder.

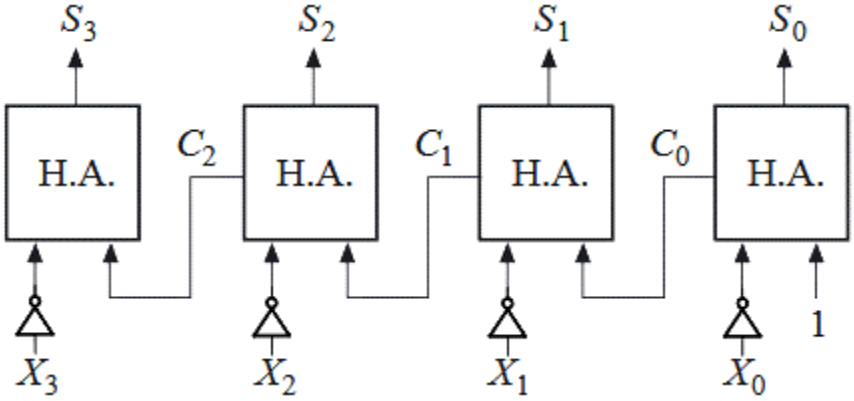
EXAMPLE 8 - 5

A half adder is a circuit that adds two bits to give a sum and a carry. Give the truth table for a half adder and design the circuit using only two gates. Then design a circuit which will find the 2's complement of a 4-bit binary number. Use four half adders and any additional gates. (*Hint:* Recall that one way to find the 2's complement of a binary number is to complement *all* bits, and then add 1).

Solution

XY	Sum	Cout
0 0	0	0
0 1	1	0
1 0	1	0
1 1	0	1

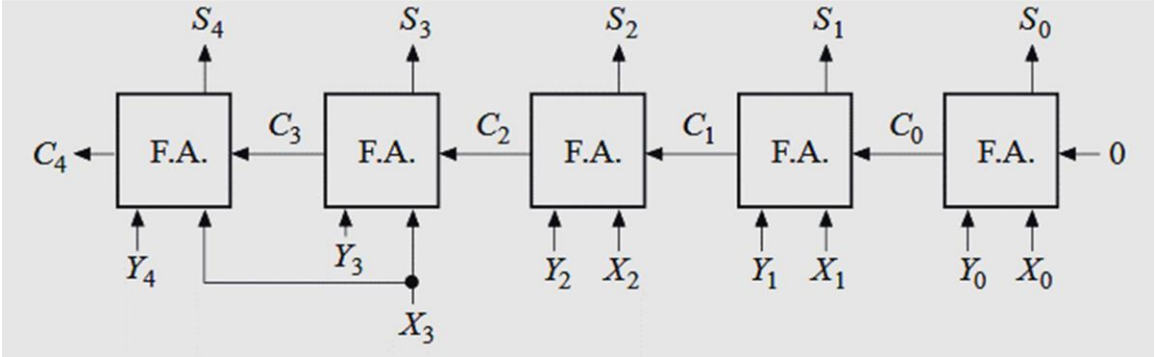




EXAMPLE 8 - 6

Design a circuit which will add a 4-bit binary number to a 5-bit binary number. Use five full adders. Assume negative numbers are represented in 2's complement. (Hint: How do you make a 4-bit binary number into a 5-bit binary number, without making a negative number positive or a positive number negative? Try writing Represent each of the following sentences by a Boolean equation down the representation for -3 as a 3-bit 2's complement number, a 4-bit 2's complement number, and a 5-bit 2's complement number. Recall that one way to find the 2's complement of a binary number is to complement all bits to the left of the first 1).

Solution



8.9 Half Subtractor

Half subtractor is a combination circuit with two inputs and two outputs which is difference and borrow. It produces the difference between the two binary bits at the input and also produces an output (Borrow) to indicate if a **1** has been borrowed. In the subtraction $(A - B)$, A is called a Minuend bit and B is called as Subtrahend bit.

Truth Table:

Inputs		Outputs	
A	B	Difference	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

The SOP form of the *Difference* and *Borrow* is as follows:

$$\text{Difference} = \bar{A}B + A\bar{B} = A \oplus B$$

$$\text{Borrow} = \bar{A}B$$

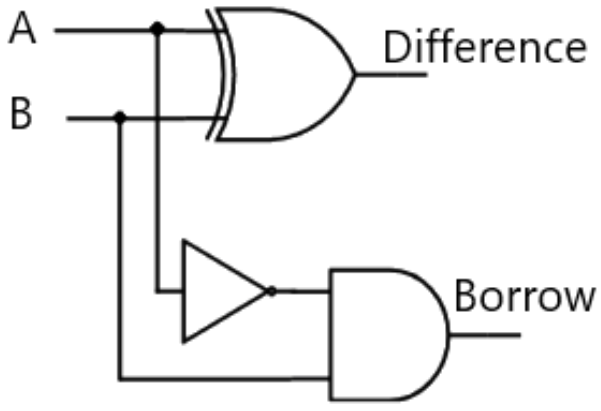


Figure 8–16 Half-subtractor logic.

8.10 Full Subtractor

A *full subtractor* is a combinational circuit that performs subtraction of two bits, one is minuend and other is subtrahend, taking into account borrow of the previous adjacent lower minuend bit. This circuit has *three* inputs and *two* outputs. The three inputs A, B and B_{in}, denote the minuend, subtrahend, and previous borrow, respectively. The two outputs, D and B_{out} represent the difference and output borrow, respectively.

Inputs			Outputs	
A	B	$Borrow_{in}$	Diff	Borrow
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

From above table, we can deduce the logical expression for *difference* and *borrow*.

Logical expression for ***difference***:

$$\boldsymbol{D} = \bar{A}\bar{B}B_{in} + \bar{A}B\overline{B_{in}} + A\bar{B}\overline{B_{in}} + AB\,B_{in}$$

$$= B_{in}(\bar{A}\bar{B} + AB) + \overline{B_{in}}(\bar{A}B + A\bar{B})$$

$$= B_{in}(\bar{A}\bar{B} + AB) + \overline{B_{in}}(\bar{A}B + A\bar{B})$$

$$= B_{in}\overline{(A \oplus B)} + \overline{B_{in}}(A \oplus B)$$

$$= (\boldsymbol{A} \oplus \boldsymbol{B}) \oplus \boldsymbol{B_{in}}$$

Logical expression for ***borrow***:

$$\boldsymbol{B_{out}} = \bar{A}\bar{B}B_{in} + \bar{A}B\overline{B_{in}} + \bar{A}BB_{in} + ABB_{in}$$

$$= (\bar{A}\bar{B} + AB)B_{in} + \bar{A}B(\overline{B_{in}} + B_{in})$$

$$= (\boldsymbol{\bar{A}\bar{B} + AB})\boldsymbol{B_{in}} + \boldsymbol{\bar{A}B}$$

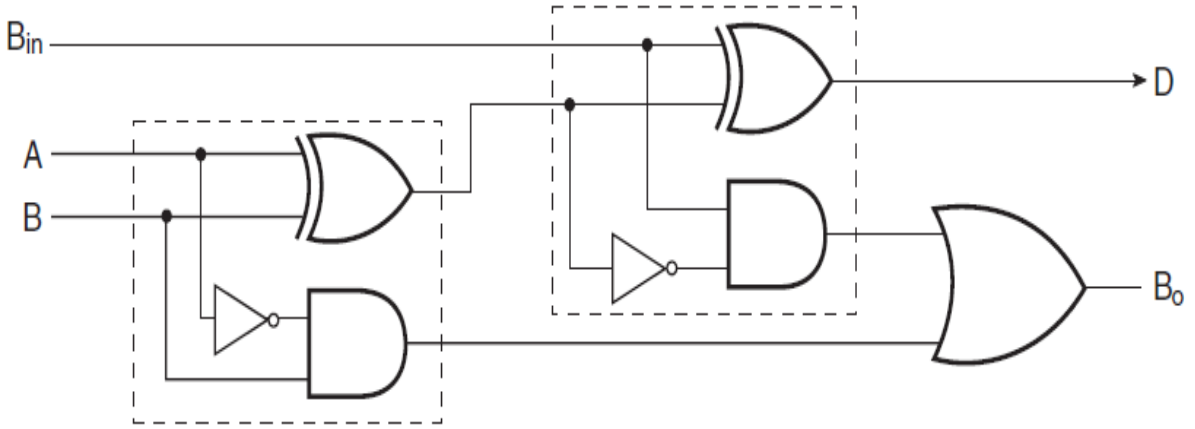


Figure 8–17 Full-subtractor logic.

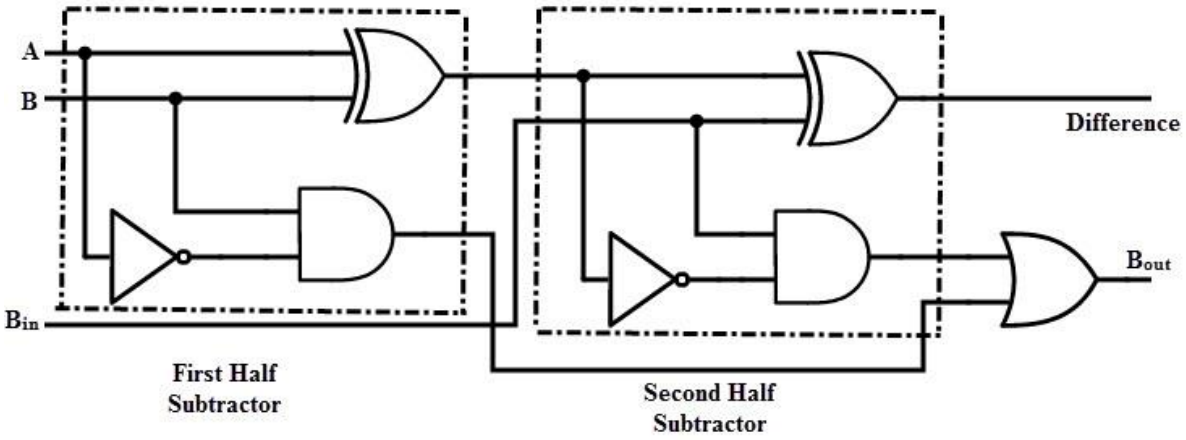


Figure 8–18 Implementation of Full Subtractor using 2 Half Subtractors and an OR gate.

Exercise 8-1

A. Implement Half Adder and Half subtractor using:

1. NAND gates.

2. NOR gates.

B. Implement Full Adder and Full subtractor using:

1. NAND gates.

2. NOR gates.

8.11 Binary Subtractor

The subtraction of unsigned binary numbers can be done most conveniently by means of complements. Remember that the subtraction A - B can be done by taking the 2's complement of B and adding it to A. The 2's complement can be obtained by taking the 1's complement and adding 1 to the least significant pair of bits. The 1's complement can be implemented with inverters, and a 1 can be added to the sum through the input carry.

The circuit for subtracting A - B consists of an adder with inverters placed between each data input B and the corresponding input of the full adder. The input carry C_0 must be equal to 1 when subtraction is performed. The operation thus performed becomes A, plus the 1's complement of B, plus 1. This is equal to A plus the 2's complement of B. For unsigned numbers, that gives A - B if $A \geq B$ or the 2's complement of $(B - A)$ if $A < B$. For signed numbers, the result is A - B, provided that there is no overflow.

The addition and subtraction operations can be combined into one circuit with one common binary adder by including an *exclusive-OR* gate with each full

adder. A four-bit adder–subtractor circuit is shown in Figure 8-12. The mode input M controls the operation. When $M = 0$, the circuit is an *adder*, and when $M = 1$, the circuit becomes a *subtractor*. Each *exclusive-OR* gate receives input M and one of the inputs of B. When $M = 0$, we have $B \oplus 0 = B$. The full adders receive the value of B, the input carry is 0, and the circuit performs A plus B. When $M = 1$, we have $B \oplus 1 = \overline{B}$ and $C_0 = 1$. The B inputs are all complemented and a 1 is added through the input carry. The circuit performs the operation A plus the 2's complement of B. (The *exclusive-OR* with output V is for detecting an overflow.)

It is worth noting that binary numbers in the signed-complement system are added and subtracted by the same basic addition and subtraction rules as are unsigned numbers. Therefore, computers need only one common hardware circuit to handle both types of arithmetic. The user or programmer must interpret the results of such addition or subtraction differently, depending on whether it is assumed that the numbers are signed or unsigned.

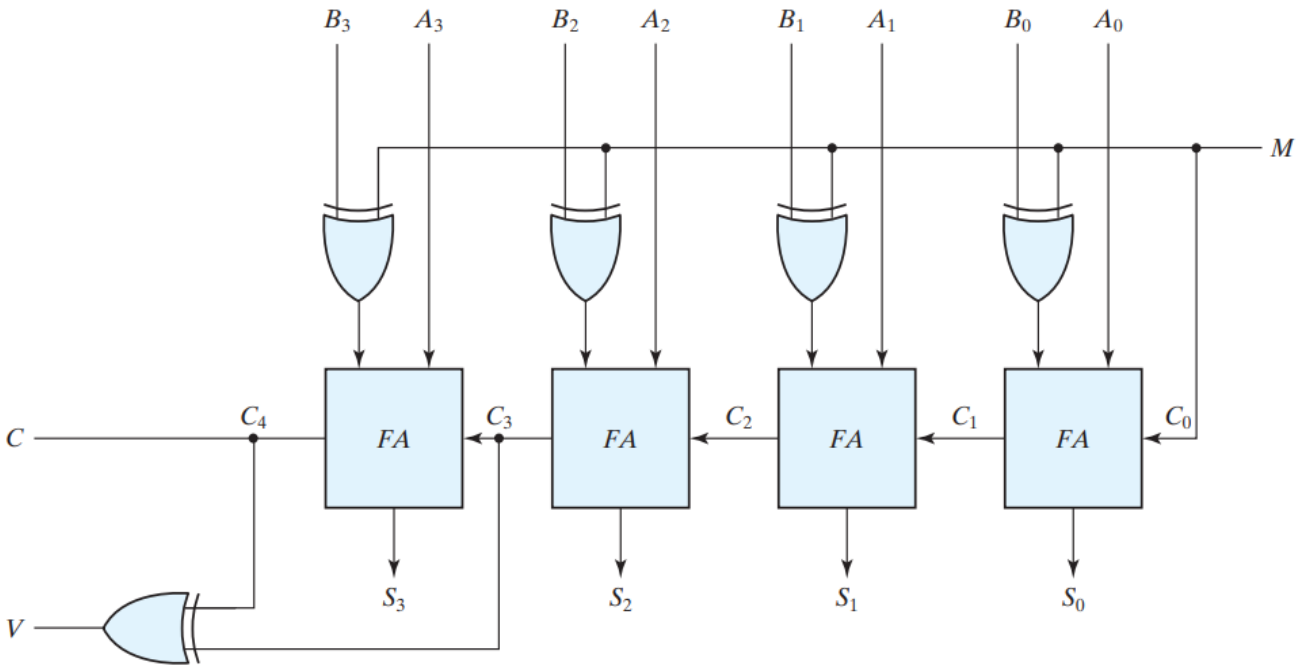
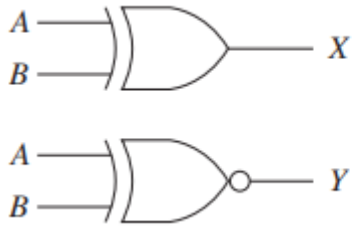
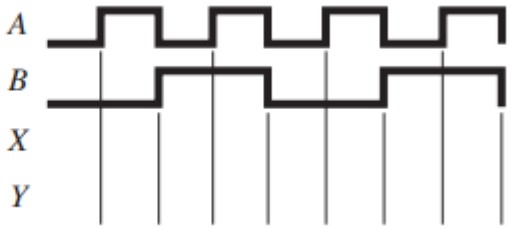


Figure 8–19 Four-bit adder–subtractor (with overflow detection)

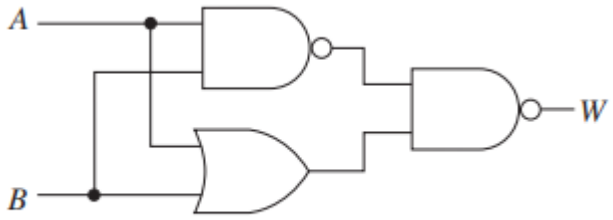
Problems

1. **Complete the timing diagram for the exclusive-OR and the exclusive-NOR:**

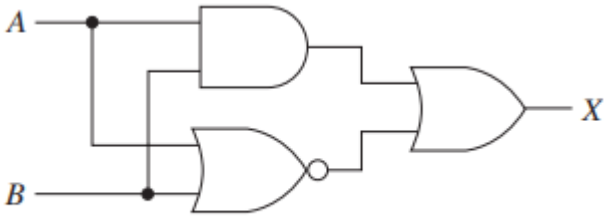




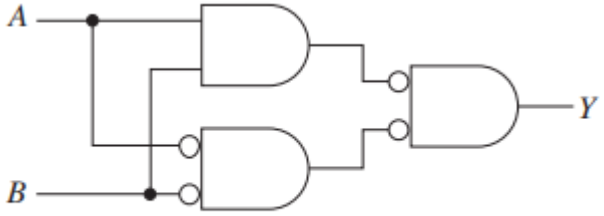
2. **Write the Boolean equations for the following circuits. Simplify the equations and determine if they function as an Ex-OR, Ex-NOR, or neither.**



(a)



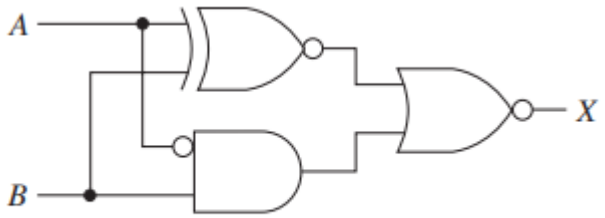
(b)



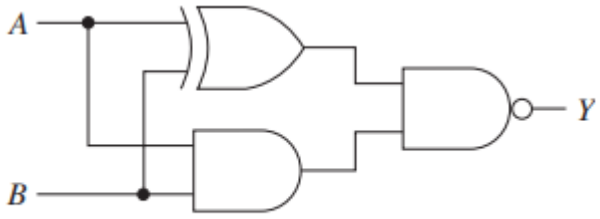
(c)

3. **Design an exclusive-OR gate constructed from all NOR gates.**

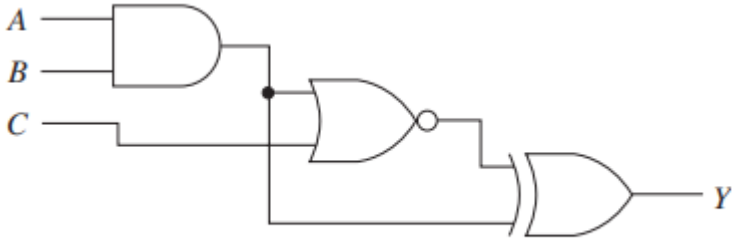
4. **Write the Boolean equations for the following circuits. Reduce the equations to their simplest form.**



(a)



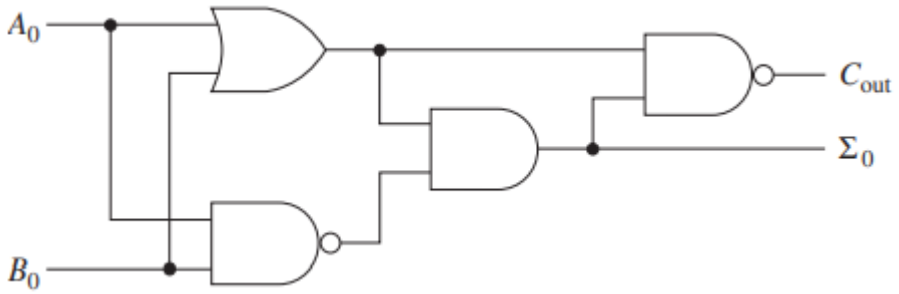
(b)



(b)

5. **Under what circumstances would you use a half-adder instead of a full-adder?**

6. **The circuit in the following Figure is an attempt to build a half-adder. Will the C_{out} and function properly? (Hint: Write the Boolean equation at C_{out} and Σ_0).**



7. Draw the block diagram of a 4-bit full-adder using four full-adders.

8. What changes would have to be made to the adder/subtractor circuit if exclusive-NORs are to be used instead of exclusive ORs?

9. Determine the outputs of a full-adder for each of the following inputs:

   (a) $A = 0$, $B = 1$, $C_{in} = 0$.

   (b) $A = 1$, $B = 0$, $C_{in} = 1$.

   (c) $A = 0$, $B = 0$, $C_{in} = 0$

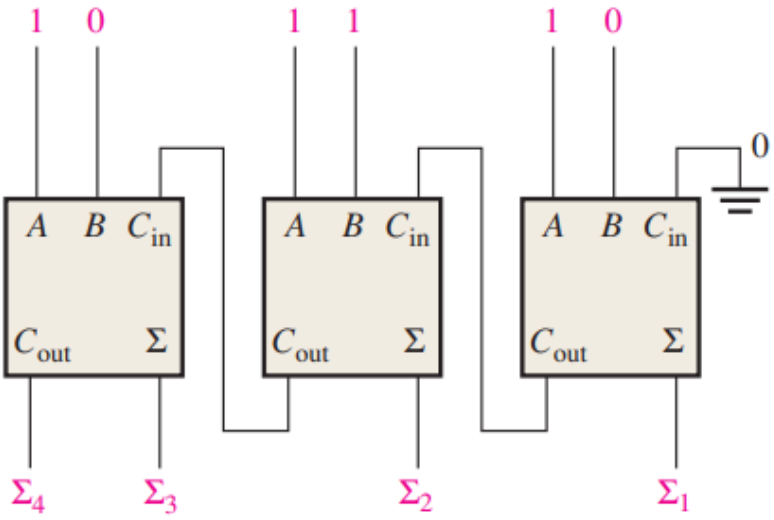
10. What are the half-adder inputs that will produce the following outputs:

    (a) $\Sigma = 0$, $C_{out} = 0$.

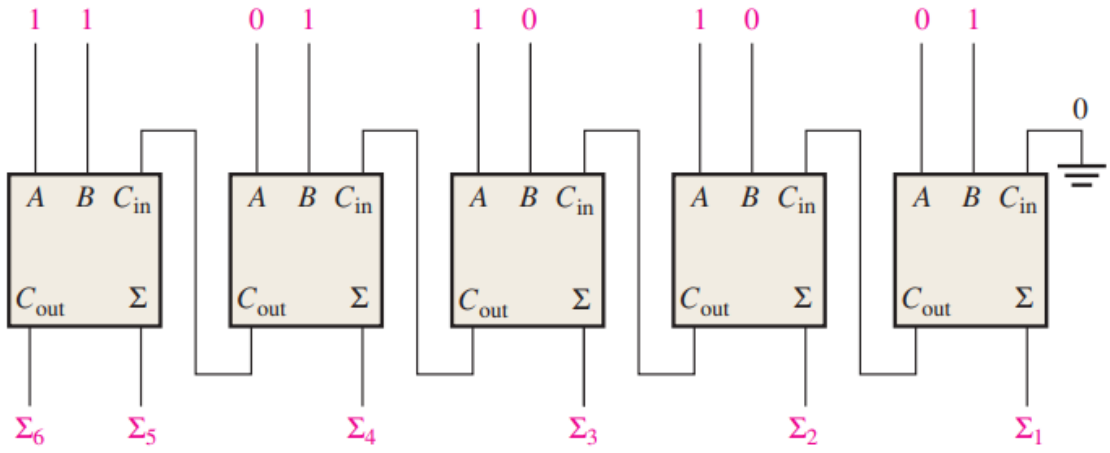
    (b) $\Sigma = 1$, $C_{out} = 0$.

    (c) $\Sigma = 0$, $C_{out} = 1$.

11. For the parallel adder in the following Figure, determine the complete sum by analysis of the logical operation of the circuit. Verify your result by longhand addition of the two input numbers.



12. Repeat Problem 11 for the circuit and input conditions in the following Figure.



اسم الطالب: .....
رقم الطالب: .....
اسم المقرر: .....
رقم الشيت: .....

الرقم السري: .....

استمارة التقويم المستمر

Question No.	Degree	Question No.	Degree	Signature
1.		2.		
3.		4.		
5.		6.		
7.		8.		
9.		10.		
11.		12.		

12

Notes

9

Comparator, Code Converters, Multiplexers, and Demultiplexers

OUTLINE

9.1 INTRODUCTION

The digital system consists of *two* types of circuits, namely:

(a) Combinational circuits and

(b) Sequential circuits.

A *combinational* circuit consists of logic gates, where outputs are at any instant and are determined only by the present combination of inputs without regard to previous inputs or previous state of outputs. A combinational circuit performs a specific information-processing operation assigned logically by a set of Boolean functions.

Sequential circuits contain logic gates as well as memory cells. Their outputs depend on the present inputs and also on the states of memory elements. Since the outputs of sequential circuits depend not only on the present inputs but also on past inputs, the circuit behavior must be specified by a time sequence of inputs and memory states.

In the previous chapters we have discussed binary numbers, codes, Boolean algebra and simplification of Boolean function, logic gates, and economical gate implementation. Binary numbers and codes bear discrete quantities of information, and the binary variables are the representation of electric voltages or some other signals. In this chapter, formulation and analysis of various systematic design of combinational circuits and application of information processing hardware will be discussed.

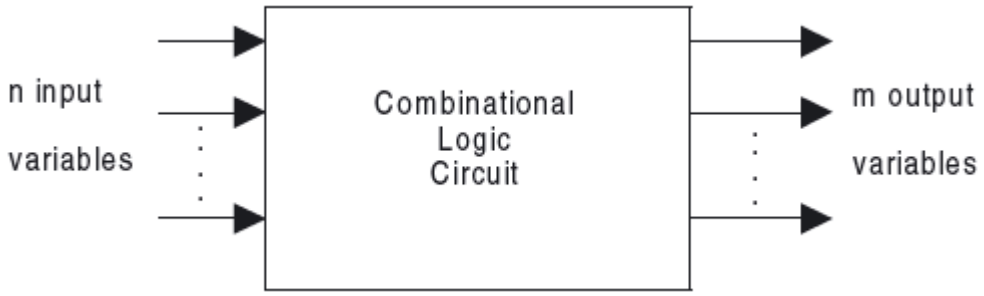


Figure 9.1 Combinational Logic Circuit

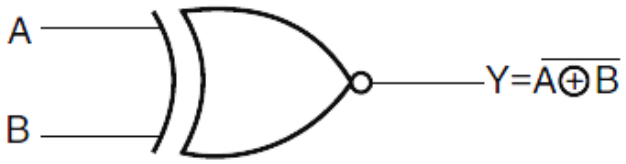
A *combinational* circuit consists of *input variables*, *logic gates*, and *output variables*. The logic gates accept signals from inputs and output signals are generated according to the logic circuits employed in it. Binary information from the given data transforms to desired output data in this process. Both input and output are obviously the binary signals, *i.e.*, both the input and output signals are of two possible states, logic **1** and logic **0**. Figure 9.1 shows a block diagram of a combinational logic circuit. There are *n* number of input variables coming from an electric source and *m* number of output signals go to an external destination. The source and/or destination may consist of memory elements or sequential logic circuit or shift registers, located either in the vicinity of the combinational logic circuit or in a remote external location. But the external circuit does not interfere in the behavior of the combinational circuit.

9.2 Comparators

Often in the evaluation of digital information, it is important to *compare* two binary strings (or binary words) to determine if they are exactly *equal*. This comparison process is performed by a ***digital comparator***.

The basic *comparator* evaluates two binary strings bit by bit and outputs a **1** if they are exactly equal. An *exclusive-NOR* gate is the easiest way to compare

the *equality* of bits. If both bits are equal, the ex-NOR puts out a **1**. To compare more than just **2** bits, we need additional ex-NORs, and the output of all of them must be **1**. For example, to design a comparator to evaluate two 4-bit numbers, we need four ex-NORs. To determine total equality, connect all four outputs into an AND gate. That way, if all four outputs are **1**s, the AND gate puts out a **1**. Figure 9–3.



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

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

Figure 9.2 Binary comparator for comparing two bits.

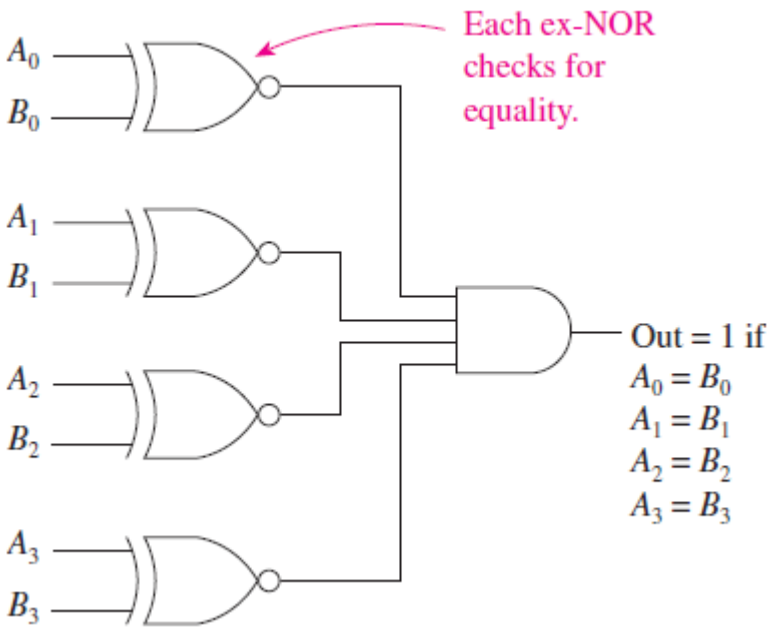


Figure 9.3 Binary comparator for comparing two 4-bit binary strings.

EXAMPLE 9 - 1

Referring to Figure 9–3, determine if the following pairs of input binary numbers will output a 1.

(a) $A_3A_2A_1A_0 = 1011$

$B_3B_2B_1B_0 = 1011$

(b) $A_3A_2A_1A_0 = 0110$

$B_3B_2B_1B_0 = 0111$

Solution

(a) When the A and B numbers are applied to the inputs, each of the four XNORs will output **1**s, so the output of the AND gate will be **1** (**equality**).

(b) For this case, the first three XNORs will output **1**s, but the last XNOR will output a **0** because its inputs are *not equal*. The AND gate will output a **0** (**inequality**).

Integrated-circuit *magnitude comparators* are available in both the TTL and CMOS families. A *magnitude comparator* not only determines if A equals B but also if A is *greater than* B or A is *less than* B.

The 7485 is a TTL 4-bit magnitude comparator. The pin configuration and logic symbol for the 7485 are given in Figure 9-4. The 7485 can be used just like the basic comparator of Figure 9-3 by using the A inputs, B inputs, and the equality output $(A = B)$. The 7485 has the additional feature of telling you which number is larger if the equality is not met. The $(A > B)$ output is **1** if A is larger than B, and the output $(A < B)$ is **1** if B is larger than A.

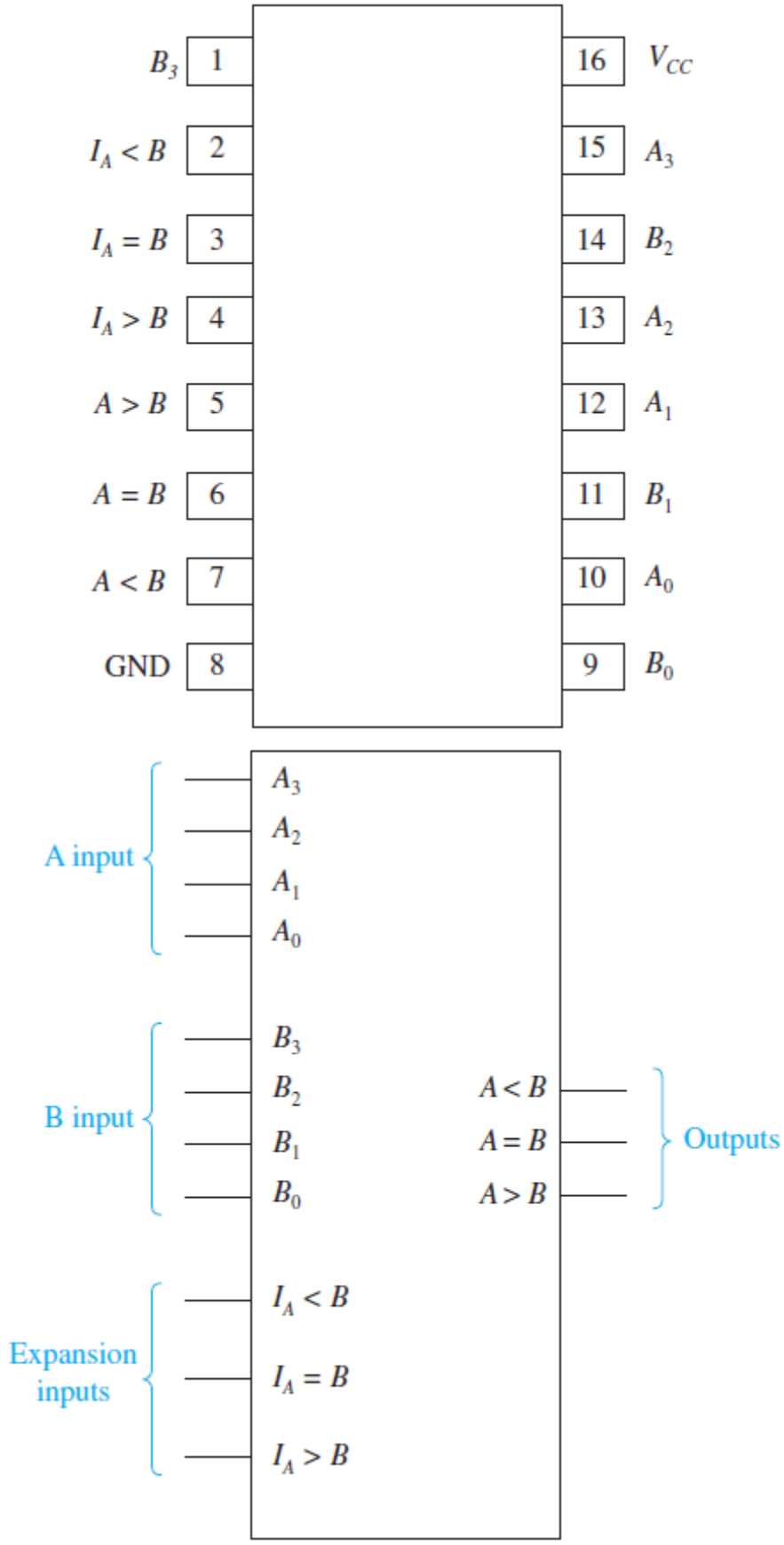


Figure 9-4 The 7485 4-bit magnitude comparator: (a) pin configuration and

(b) logic symbol.

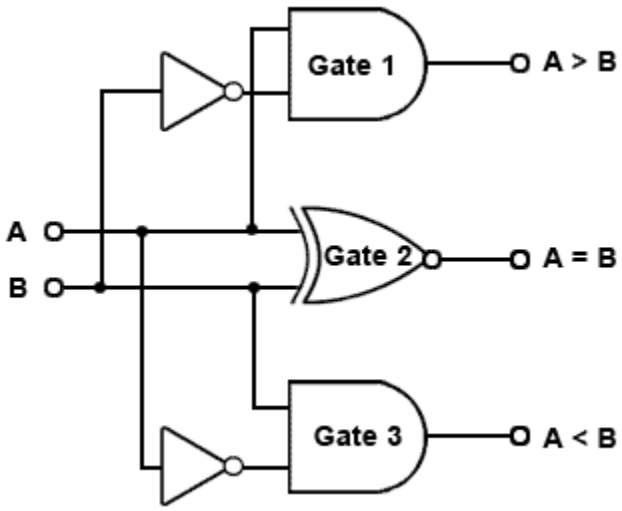
A *comparator* used to compare two bits is called a *single-bit comparator*. It consists of *two* inputs each for two single-bit numbers and *three* outputs to generate *less than*, *equal to*, and *greater than* between two binary numbers. The truth table for a 1-bit comparator is given below:

Inputs		Outputs		
B	A	A > B	A = B	A < B
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

From the above truth table logical expressions for each output can be expressed as follows:

$$(A < B) \rightarrow \bar{A}.B$$

$$(A > B) \rightarrow A.\overline{B}$$



$$(A < B) \rightarrow A.B + \bar{A}.\bar{B}$$

Figure 9-5 One-bit magnitude comparator logic gates diagram.

From the above expressions, we can derive the following formula:

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

Taking complement both sides:

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

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

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

$$= A\bar{A} + \bar{A}\bar{B} + AB + B\bar{B}$$

$$= \bar{A}\bar{B} + AB$$

By using these Boolean expressions, we can implement a logic circuit for this *comparator* as given below:

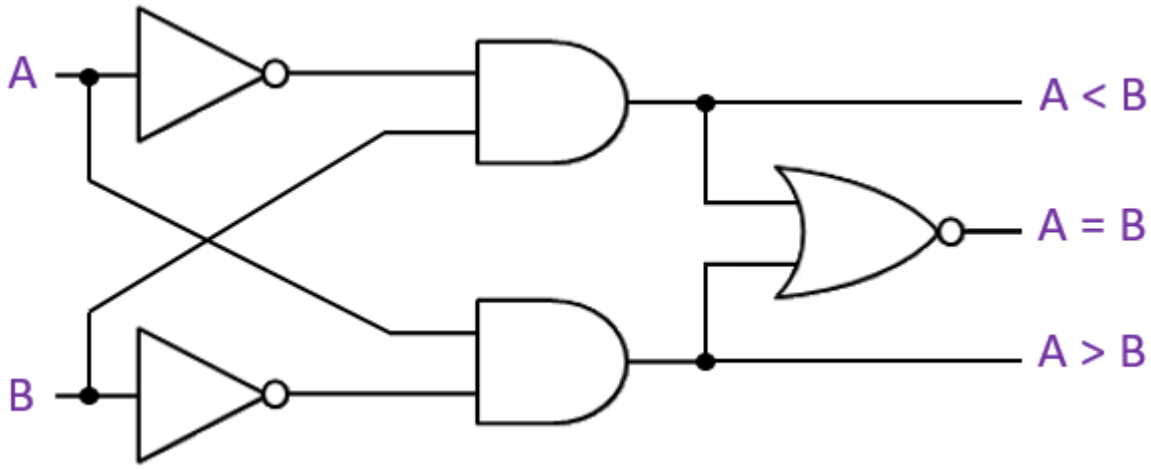
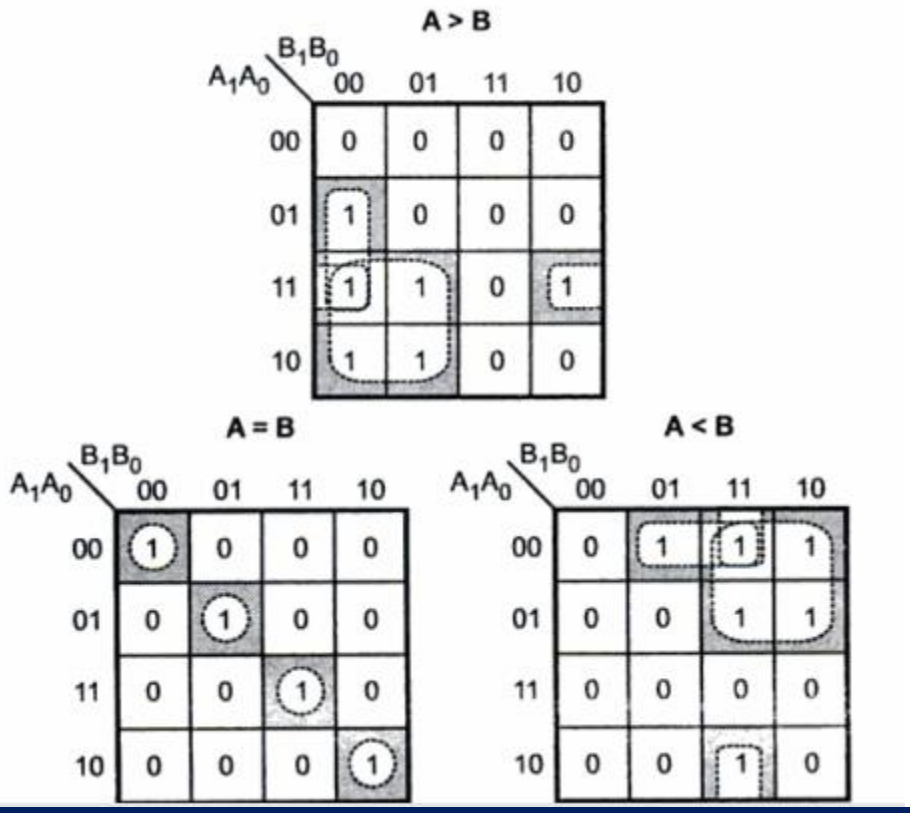


Figure 9-7 One-bit magnitude comparator logic gates diagram.

A comparator used to compare *two* binary numbers each of two bits is called a *2-bit Magnitude comparator*. It consists of *four* inputs and *three* outputs to generate *less than*, *equal to*, and *greater than* between two binary numbers. The truth table for a 2-bit comparator is given below:

Inputs				Outputs		
A_1	A_0	B_1	B_0	A>B	A=B	A<B
0	0	0	0	0	1	0
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	1	0	0
0	1	0	1	0	1	0
0	1	1	0	0	0	1
0	1	1	1	0	0	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	0	1	0
1	0	1	1	0	0	1
1	1	0	0	1	0	0
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	0	1	0

From the above truth table K-map for each output can be drawn as follows:



From the above K-maps logical expressions for each output can be expressed as follows:

$$(A > B) \rightarrow A_1\overline{B_1} + A_0\overline{B_1}.\overline{B_0} + A_1A_0\overline{B_0}$$

$$= A_1\overline{B_1} + \overline{B_0}(A_1A_0 + A_0\overline{B_1})$$

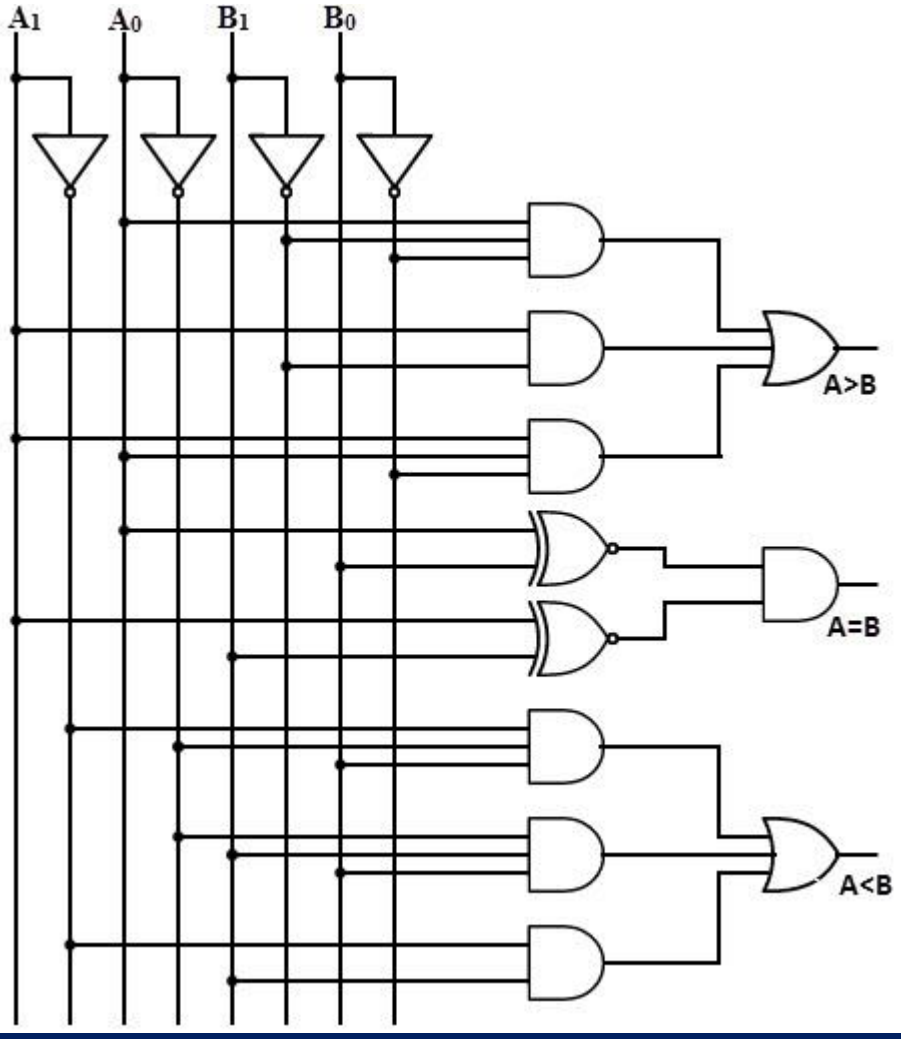
$$(A = B) \rightarrow \overline{A_1}.\overline{A_0}.\overline{B_1}.\overline{B_0} + \overline{A_1}A_0\overline{B_1}B_0 + A_1\overline{A_0}B_1\overline{B_0} + A_1A_0B_1B_0$$

$$= \overline{(A_1 \oplus B_1)}.\overline{(A_0 \oplus B_0)}$$

$$(A < B) \rightarrow \overline{A_1}B_1 + \overline{A_0}B_1B_0 + \overline{A_1}.\overline{A_0}B_0$$

$$= \overline{A_1}B_1 + \overline{A_0}(B_1B_0 + \overline{A_1}B_0)$$

By using these Boolean expressions, we can implement a logic circuit for this comparator as given below:



Applications of Comparators:

1. Comparators are used in central processing units (CPUs) and microcontrollers (MCUs).
2. These are used in control applications in which the binary numbers representing physical variables such as temperature, position, etc. are compared with a reference value.
3. Comparators are also used as process controllers and for Servo motor control.
4. Used in password verification and biometric applications.

9.3 Decoding

Decoding is the process of converting some code (such as *binary*, *BCD*, or *hexadecimal*) into a *singular* active output representing its numeric value. Take, for example, a system that reads a 4-bit BCD code and converts it to its appropriate decimal number by turning on a decimal indicating lamp. Figure 9–8 illustrates such a system.

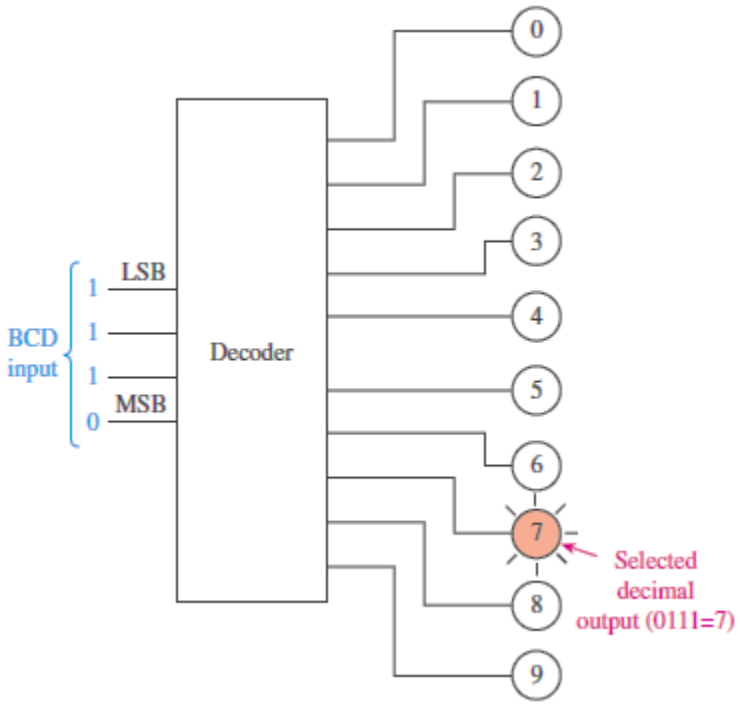


Figure 9-8 A BCD decoder selects the correct decimal-indicating lamp.

This ***decoder*** is made up of a combination of logic gates that produces a HIGH at one of the 10 outputs, based on the levels at the four inputs. In this section, we learn how to use decoder ICs by first looking at the combinational logic that makes them work and then by selecting the actual decoder IC and making the appropriate pin connections.

Binary Decoder is another combinational logic circuit constructed from individual logic gates and is the exact opposite to an *Encoder*. Binary Decoder is the process of converting the binary number into a singular active output representing its numeric value using $\boldsymbol{2^n}$ outputs.

Binary Decoders are another type of digital logic device that has inputs of 2-bit, 3-bit or 4-bit codes depending upon the number of data input lines. So, a decoder that has a set of two or more bits will be defined as having an $\boldsymbol{n}$-bit code, and therefore it will be possible to represent $\boldsymbol{2^n}$ possible values. Thus, a decoder generally decodes a binary value into a non-binary one by setting exactly one at the output that representing the binary number.

9.3.1 2-Bit Binary to 4 Decoding

Let 2 to 4 Decoder has two inputs A_1, A_0 and four outputs Y_3, Y_2, Y_1, Y_0. The **block diagram** of 2 to 4 decoder is shown in Figure 9-9.

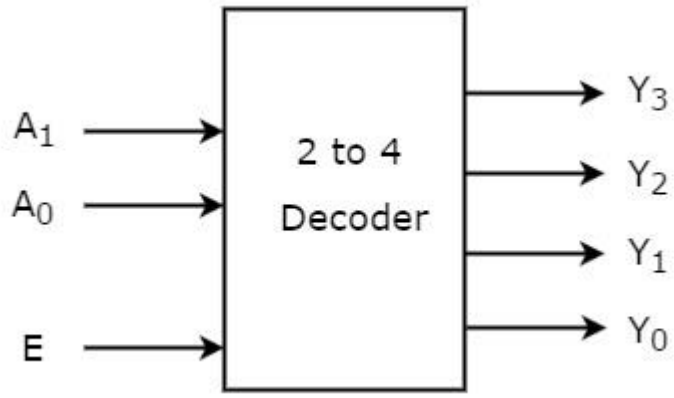


Figure 9-9 Block diagram of 2 to 4 decoder.

One of these four outputs will be '**1**' for each combination of inputs when enable, ***E*** is '**1**'. The Truth table of 2 to 4 decoder is shown below.

Enable	INPUTS		OUTPUTS			
E	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	X	X	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

From Truth table, we can write the **Boolean functions** for each output as:

$$Y_0 = \overline{A_1}.\overline{A_0}.E$$

$$Y_1 = \overline{A_1}.A_0.E$$

$$Y_2 = A_1.\overline{A_0}.E$$

$$Y_3 = A_1.A_0.E$$

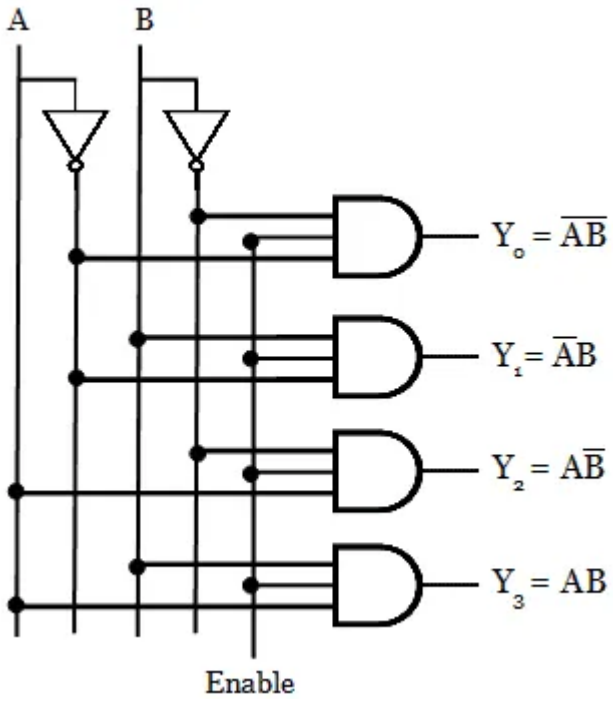


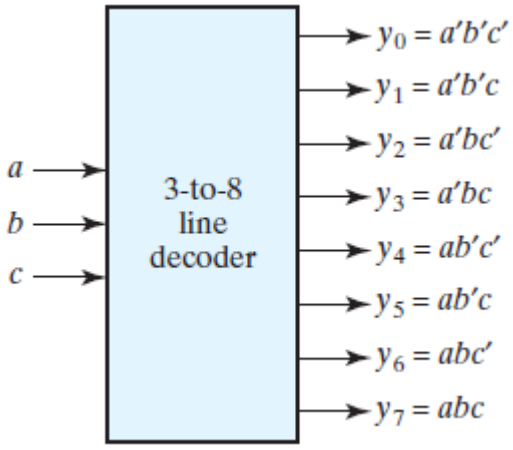
Figure 9-10 Circuit diagram of 2 to 4 decoder.

Each output is having one product term. So, there are four product terms in total. We can implement these four product terms by using four AND gates having three inputs each. The circuit diagram of 2 to 4 decoder is shown in Figure 9-10.

9.3.2 3-Bit Binary-to-Octal Decoding

To design a *decoder*, it is useful first to make a truth table of all possible input/output combinations. An *octal decoder* must provide *eight* outputs, one for each of the eight different combinations of inputs, as shown in the following table.

Inputs				Outputs							
EN	A	B	C	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	×	×	×	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0



9.3.3 Decoder Expansions

Larger decoders can be constructed using several smaller ones. For example, a 3 × 8 decoder can be built using 2 × 4 decoders and a 4 × 16 decoder can be built using 3 × 8 decoders. let us implement 3 × 8 decoder using 2 × 4 decoders. We know that 2 × 4 Decoder has *two* inputs, A_1, A_0 and *four* outputs, Y_3 to Y_0. Whereas, 3 × 8 Decoder has *three* inputs A_2, A_1, A_0 and *eight* outputs, Y_7 to Y_0. So, we need to find the *number of lower order decoders required* for implementing higher order decoder using the following formula.

$$\text{Required number of lower order decoders} = \frac{m_2}{m_1}$$

where,

m_1 is the number of outputs of lower order decoder.

m_2 is the number of outputs of higher order decoder.

Here, $m_1 = 4$ and $m_2 = 8$. Substitute, these two values in the above formula.

$$\text{Required number of lower order decoders} = \frac{8}{4} = \mathbf{2}$$

Therefore, we require *two* 2 × 4 decoders for implementing one 3 × 8 decoder. The *block diagram* of 3 × 8 decoder using 2 × 4 decoders is shown in Figure 9-11.

The parallel inputs A_1, A_0 are applied to each 2 × 4 decoder. The complement of input A_2 is connected to *Enable*, E of lower 2 × 4 decoder in order to get the outputs, Y_3 to Y_0. These are the **lower four min terms**. The input, A_2 is directly connected to *Enable*, E of upper 2 × 4 decoder in order to get the outputs, Y_7 to Y_4. These are the **higher four min terms**.

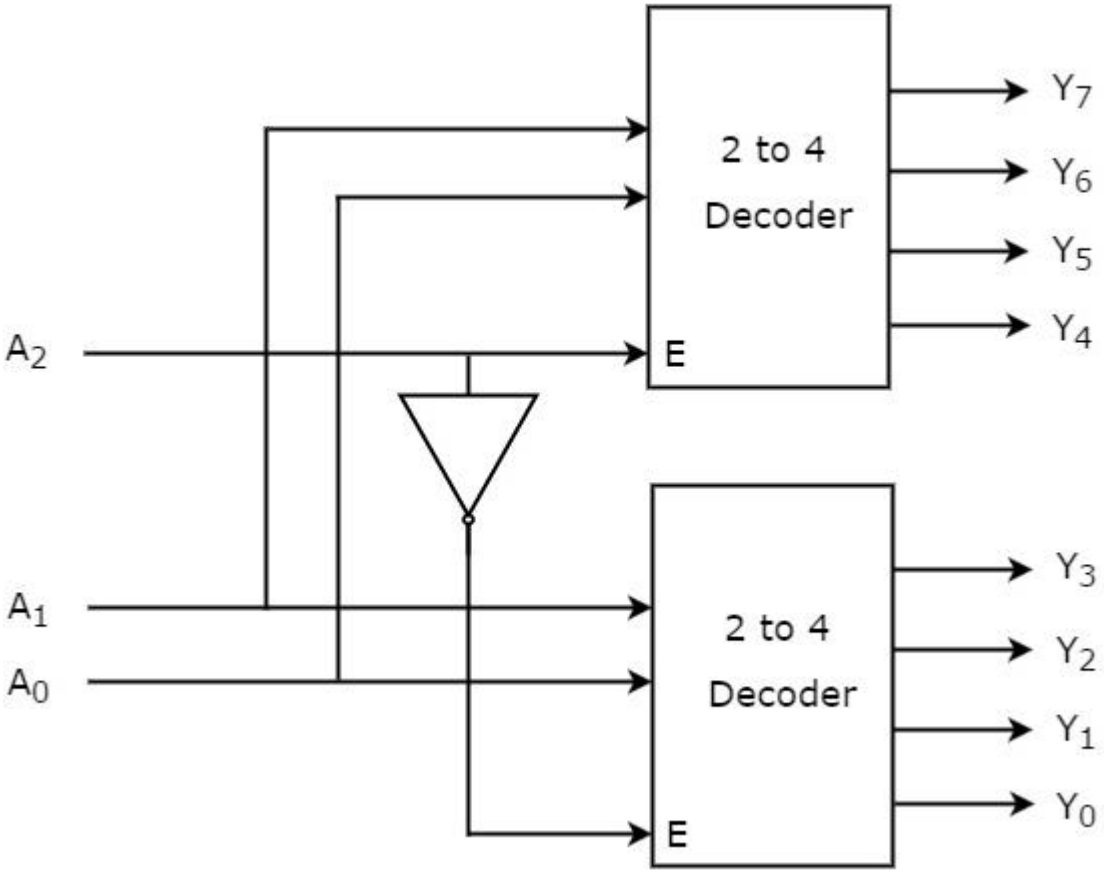


Figure 9-11 A 3 × 8 decoder constructed with two 2 × 4 decoders.

9.4 Encoding

Encoding is the opposite process from *decoding*. *Encoding* is used to generate a coded output (such as *BCD* or *binary*) from a singular active numeric input line. For example, Figure 9–12 shows a typical block diagram for a *decimal-to-BCD* encoder and an *octal-to-binary* encoder. An *encoder* has $\mathbf{2^n}$ input lines and n output lines. The output lines generate the binary equivalent of the input line whose value is **1**.

we can see that the A output (2^0) is HIGH for all *odd* decimal input numbers $(1, 3, 5, 7, \text{and } 9)$. The B output (2^1) is HIGH for decimal inputs $2, 3, 6$, and 7. The C output (2^2) is HIGH for decimal inputs $4, 5, 6$, and 7, and the D

output (2^3) is HIGH for decimal inputs 8 and 9.

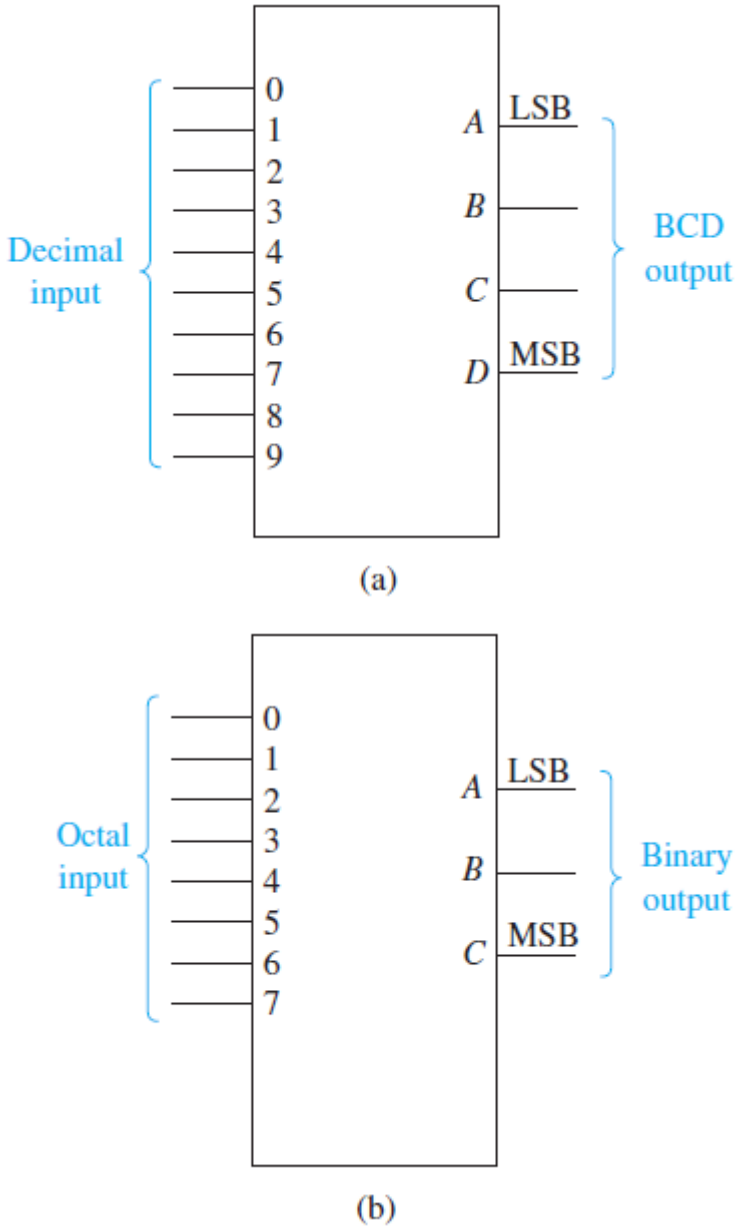


Figure 9-12 Typical block diagrams for encoders: (a) decimal-to-BCD encoder and (b) octal-to-binary encoder.

The design of *encoders* using combinational logic can be done by reviewing Table 9-1 for the operation to determine the relationship each output has with the inputs. For example, by studying this table for a decimal-to-BCD encoder, we can see that the *A* output (2^0) is HIGH for all *odd* decimal input numbers

(1, 3, 5, 7, and 9). The B output (2^1) is HIGH for decimal inputs 2, 3, 6, and 7. The C output (2^2) is HIGH for decimal inputs 4, 5, 6, and 7, and the D output (2^3) is HIGH for decimal inputs 8 and 9.

TABLE Decimal-to-BCD Encoder Truth Table

Decimal Input	BCD Output			
	D	*C*	*B*	*A*
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1

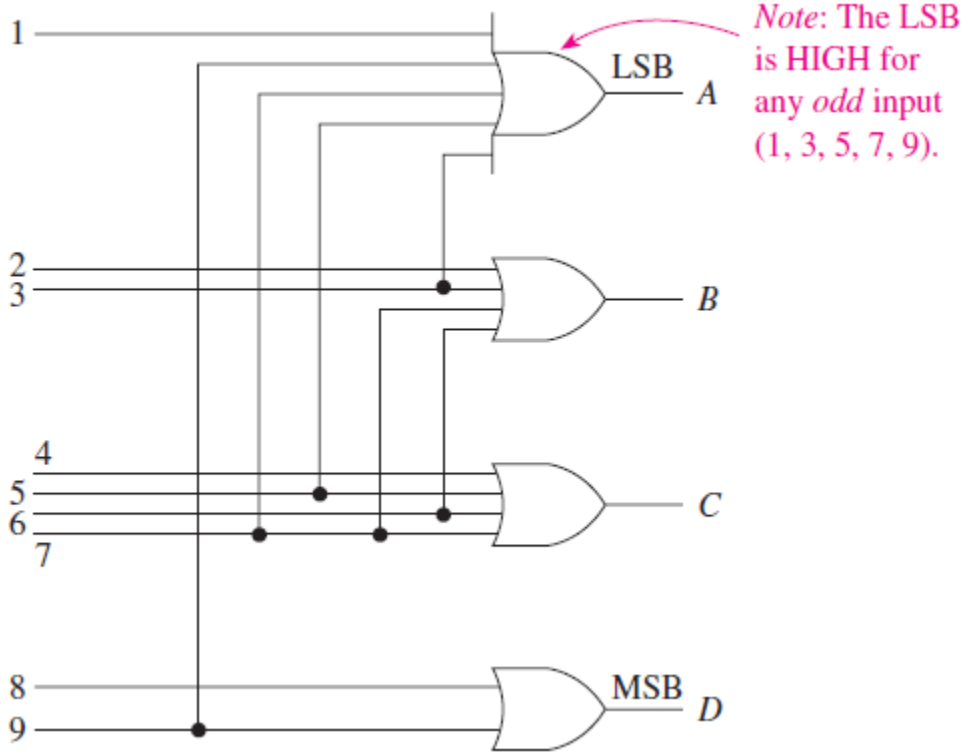


Figure 9-13 Basic decimal-to-BCD encoder.

Now, from what we have just observed, it seems that we can design a decimal-to-BCD encoder with just *four* OR gates; the *A* output OR gate goes HIGH for any *odd* decimal input, the *B* output goes HIGH for 2 or 3 or 6 or 7, and so on, for the *C* output and *D* output. The complete design of a basic decimal-to-BCD encoder is given in Figure 9–13.

What happens if more than one input is active (set to HIGH) at the same time? For example, D_3 and D_6?

Answer

- **If D_3 and D_6 are active simultaneously, the output would be $(111)_2 = (7)_{10}$, because all three outputs would be equal to 1.**
- **But this does not reflect the actual input which should have resulted in an output of $(011)_2 = (3)_{10}$ for D_3 or $(110)_2 = (6)_{10}$ for D_6.**
- **To overcome this problem, we use priority encoders.**
- **If we establish a higher priority for inputs with higher subscript numbers, and if both D_3 and D_6 are active at the same time, the output would be $(110)_2$ because D_6 has higher priority than D_3 .**

What happens if all inputs are equal to 0?

Answer

- **The encoder output would be $(000)_2$, but in fact this is the output when D_0 is equal to 1.**

- **This problem can be solved by providing an extra output to indicate whether at least one input is equal to 1. (v is the valid output)**

9.4.1 Priority Encoder

In Digital Electronics, the *binary encoders* are the multi-input combinational logic circuits, which consider all the input lines simultaneously and then convert them into an equivalent *single encoded* output. An n-bit digital encoder contains 2^n input lines and n output lines. To overcome the disadvantages of binary encoders, *priority encoders* were developed that work based on the highest priority input. This section gives a brief description of a priority encoder along with its applications.

The *priority encoder* is a combinational logic circuit that contains 2^n input lines and n output lines and represents the *highest priority* input among all the input lines. When multiple input lines are active high at the same time, then the input that has the highest priority is considered first to generate the output.

It is used to solve the issues in *binary encoders*, which generate wrong output when more than one input line is active high. If more than one input line is active high (**1**) at the same time, then this encoder prioritizes every input level and allocates the priority level to each input.

The output of this encoder corresponds to the input that has the highest priority. To obtain the output, only the input with the highest priority is considered by ignoring all other input lines. This is a type of binary encoder

or an ordinary encoder with a priority function. The input that has the larger magnitude or highest priority is encoded first rather than other input lines. Hence, the generated output is based on the priority assigned to the inputs.

In most digital applications, these *encoders* are used to select the inputs, which have the highest priority level. This process of selecting the input is called ***arbitration***. For example, when multiple devices transmit the data over the computer systems, then this encoder enables the device that has the highest priority and allows access to the computer among all other devices, which have lower priority.

These encoders are designed with **4** inputs and **8**-inputs. The **4**-bit *priority encoder* contains **4** inputs and **2** outputs along with one valid output. The **8**-bit priority encoder contains **8** inputs and **3** outputs. *Priority encoder* circuit with truth table for 8-bit and 4-bit are explained in the below section.

9.4.2 8 to 3 Priority Encoder

This kind of *encoder* is also named an ***8-bit*** or ***Octal*** to Binary priority encoder. This type of encoder consists of **8** inputs and **3** outputs. When multiple inputs are *active high* at the same time, the input with the highest priority is considered to represent the output.

For example, if $D_1, D_2,$ and D_3 inputs are *active high* or logic **1** regardless of other input bits, then the encoded output of the priority encoder will be D_2 i.e., 111. Here, the D_1 and D_2 input bits are either irrelevant or don't care conditions. The 8 to 3 priority encoder truth table is shown below. From the truth table, we can observe that $D_0, D_1, D_2, D_3, D_4, D_5, D_6, D_7$ are the inputs, and Y_0, Y_1, Y_2 are the outputs of an 8 to 3 priority encoder.

Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	Y_2	Y_1	Y_0
1	0	0	0	0	0	0	0	0	0	0
×	1	0	0	0	0	0	0	0	0	1
×	×	1	0	0	0	0	0	0	1	0
×	×	×	1	0	0	0	0	0	1	1
×	×	×	×	1	0	0	0	1	0	0
×	×	×	×	×	1	0	0	1	0	1
×	×	×	×	×	×	1	0	1	1	0
×	×	×	×	×	×	×	1	1	1	1

The output Y_2 of a *priority encoder* is represented as *active high* or logic '**1**' only when the inputs D_4, D_5, D_6, and D_7 are active high. The output Y_1 of an encoder is at logic **1** only when the inputs D_2, D_3, D_6, and D_7 are *active high*. Similarly, the output Y_0 is represented as logic '**1**' only when the inputs D_1, D_3, D_5, and D_7 are *active high*.

The output expressions are obtained as shown below:

$$Y_2 = D_4 + D_5 + D_6 + D_7$$

$$Y_1 = D_2 + D_3 + D_6 + D_7$$

$$Y_0 = D_1 + D_3 + D_5 + D_7$$

From these simplified expressions, the *8 to 3 priority encoder* circuit diagram is drawn as illustrated with logic gates as shown in Figure 9-14.

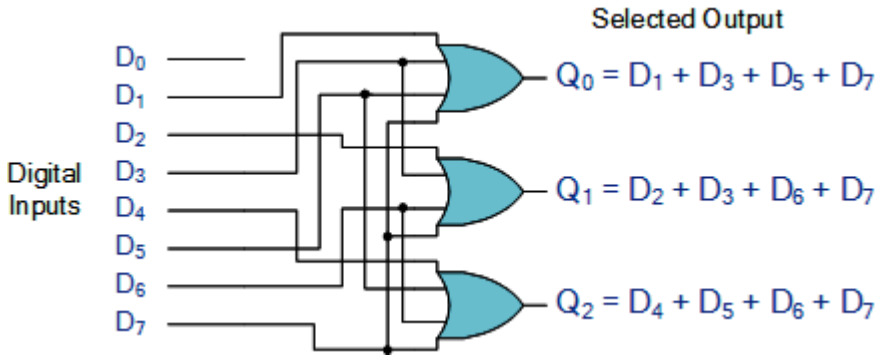


Figure 9-14 Digital Encoder using Logic Gates.

9.4.3 4 to 2 Priority Encoder

This is also referred to as 4- bit priority, which consists of 4 inputs and 2 output lines. Since an encoder contains 2^n input lines and n output lines. The third output is '*V*', which is considered as a *valid*, but indicator and it is set to **1** when more than one input line is *high* or *active* (**1**). If the *valid* bit is equal to '**0**', then all the inputs are '**0**'. In this case, the other 2 output lines are considered as don't care conditions denoted by X. The truth table of a 4 to 2 priority encoder is shown below.

Inputs				Outputs		
D_0	D_1	D_2	D_3	Y_1	Y_0	V
0	0	0	0	×	×	0
1	0	0	0	0	0	1
×	1	0	0	0	1	1
×	×	1	0	1	0	1
×	×	×	1	1	1	1

From the above truth table, we can observe that D_3, D_2, D_1, D_0 are the *inputs*; Y_1, Y_0 are the *outputs* and V is the *valid bit indicator*. Here D_3 input is the

highest priority input and D_0 is the lowest priority input.

When the input D_3 is active *high* (**1**), which has the highest priority irrespective of all other input lines, then the output of the 4-bit priority encoder is 11. When the D_3 input is active *low* and the D_2 is active *high* that has the next highest priority irrespective of all other input lines, then the output is 10. When D_3, D_2 inputs are active *low*, and the D_1 is active *high* and has the next highest priority regardless of the remaining input line, then the output will be 01. The *output expression* can be determined for a 4-bit encoder as shown below.

$$Y_0 = D_3 + D_1\overline{D_2}$$

$$Y_1 = D_2 + D_3$$

$$V = D_0 + D_1 + D_2 + D_3$$

From these output expressions, the 4 to 2 priority encoder circuit diagram is illustrated with logic gates as shown in Figure 9-15.

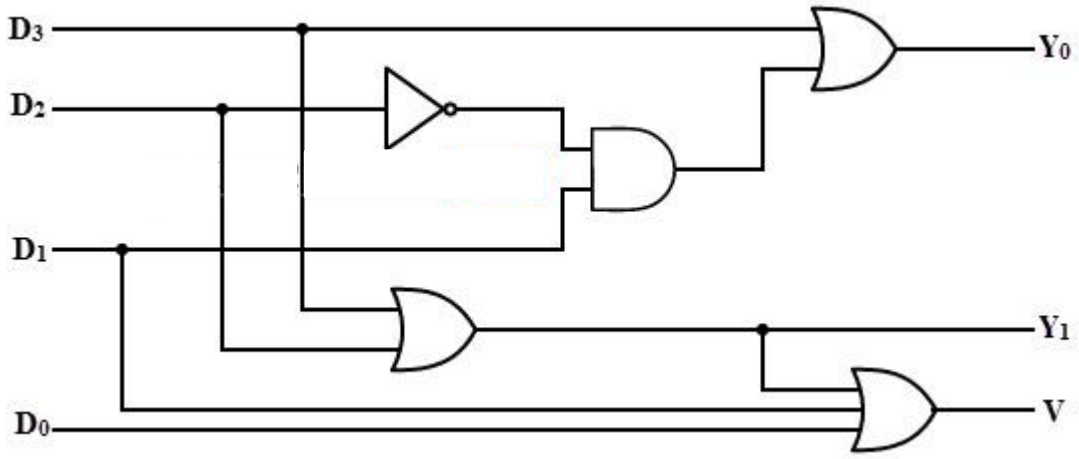


Figure 9-15 Digital Encoder using Logic Gates.

9.4.4 Difference Between Encoder and Priority Encoder

The main difference between *encoder* and *priority encoder* is the *encoder* generates an error output when more than one input is high. But they are used in applications compressing data. Hence, *priority encoders* are introduced to overcome the issues of binary encoders.

The *priority encoder* generates the accurate output by considering the highest priority input among the multiple input lines. These can handle the interrupt requests of a microprocessor by detecting the highest priority interrupt. If a circuit contains multiple inputs, then it is used to reduce the number of wires required during designing.

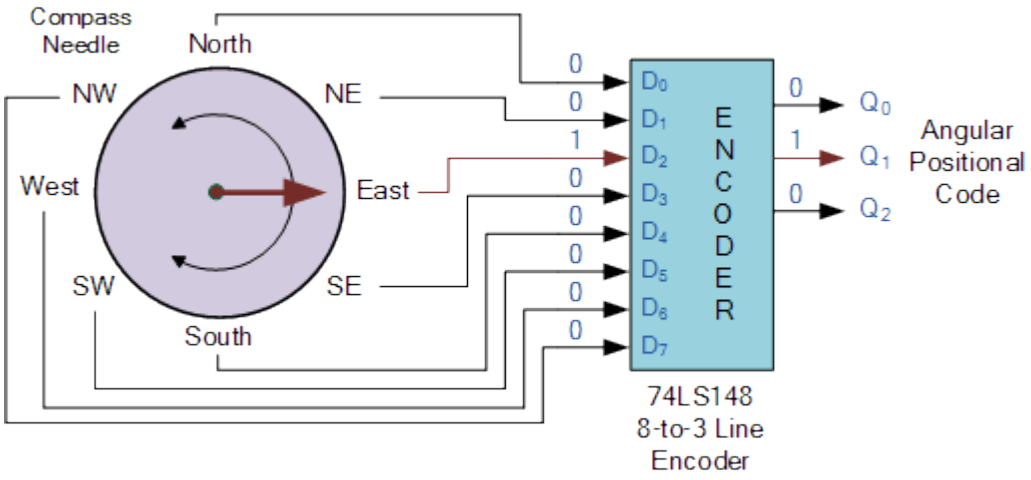
9.4.5 Applications

Some of the applications of priority encoder are:

- It is used to reduce the number of wires and connections required for electronic circuit designing that have multiple input lines. Example: keypads and keyboards.
- Used in controlling the position in the ship's navigation and robotics arm position.
- Used in the detection of highest priority input in various applications of microprocessor interrupt controllers.
- Used to protect the entire network from hackers by transmitting the binary code over the network.
- Used to encode the analog to digital converter's output.
- Used in synchronization of the speed of motors in industries.

- Used robotic vehicles.
- Used in applications of home automation systems with RF.
- Used in hospitals for health monitoring systems.

Priority Encoder Navigation



Compass Direction	Binary Output		
	Q_0	Q_1	Q_2
North	0	0	0
North-East	0	0	1
East	0	1	0
South-East	0	1	1
South	1	0	0
South-West	1	0	1
West	1	1	0
North-West	1	1	1

9.5 Multiplexers

A *multiplexer* is one of the important combinational circuits and has a wide range of applications. The term multiplex means *many into one*. Multiplexers transmit large numbers of information channels to a smaller number of channels.

A *digital multiplexer* is a combinational circuit that selects binary information from one of the many input channels and transmits to a single output line. That is why the *multiplexers* are also called *data selectors*. The selection of the particular input channel is controlled by a set of select inputs. A *digital multiplexer* of 2^n input channels can be controlled by n numbers of select lines and an input line is selected according to the bit combinations of select lines.

Multiplexers are used as one method of *reducing the number of logic gates required* in a circuit design or when a single data line or data bus is required to carry two or more different digital signals. For example, a single 8-channel multiplexer.

Generally, the selection of each input line in a *multiplexer* (MUX) is controlled by an additional set of inputs called *control lines* and according to the binary condition of these control inputs, either "HIGH" or "LOW" the appropriate data input is connected directly to the output. Normally, a multiplexer has an even number of 2^n data input lines and a number of *control* inputs that correspond with the number of data inputs.

Note that *multiplexers* are different in operation to *Encoders*. Encoders are able to switch an n-bit input pattern to multiple output lines that represent the binary coded (BCD) output equivalent of the active input.

9.5.1 2-input Multiplexer Design

A two-to-one-line multiplexer connects one of two 1-bit sources to a common destination, as shown in Figure 9-16. The circuit has two data input lines, one output line, and one selection line A. When the control input A is 0, the switch is in the *upper* position and the MUX output is $Z = I_0$; when A is 1, the switch is in the *lower* position and the MUX output is $Z = I_1$. In other words, a MUX acts like a *switch* that selects one of the data inputs (I_0 or I_1) and transmits it to the output. The *logic equation* for the 2-to-1 MUX is:

$$Z = \bar{A}I_0 + AI_1$$

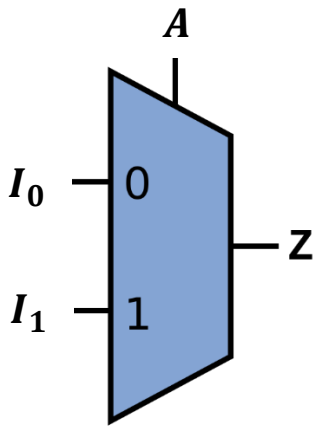


Figure 9-16 Block diagram of 2-to-1 Multiplexer

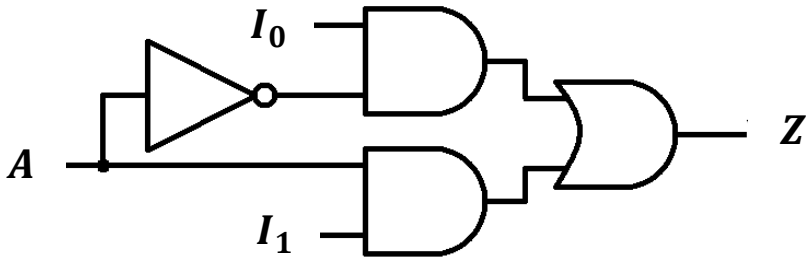


Figure 9-17 Logic gate diagram of 2-to-1 Multiplexer

Figure 9-18 shows diagrams for a 4-to-1 multiplexer, 8-to-1 multiplexer, and 2^n-to-1 multiplexer. The 4-to-1 MUX acts like a four-position switch that

transmits one of the four inputs to the output. Two control inputs (A and B) are needed to select one of the four inputs. If the control inputs are $AB = 00$, the output is I_0; similarly, the control inputs $01, 10$, and 11 give outputs of I_1, I_2 and I_3, respectively. The 4-to-1 multiplexer is described by the equation:

$$Z = \bar{A}\bar{B}I_0 + \bar{A}BI_1 + A\bar{B}I_2 + ABI_3$$

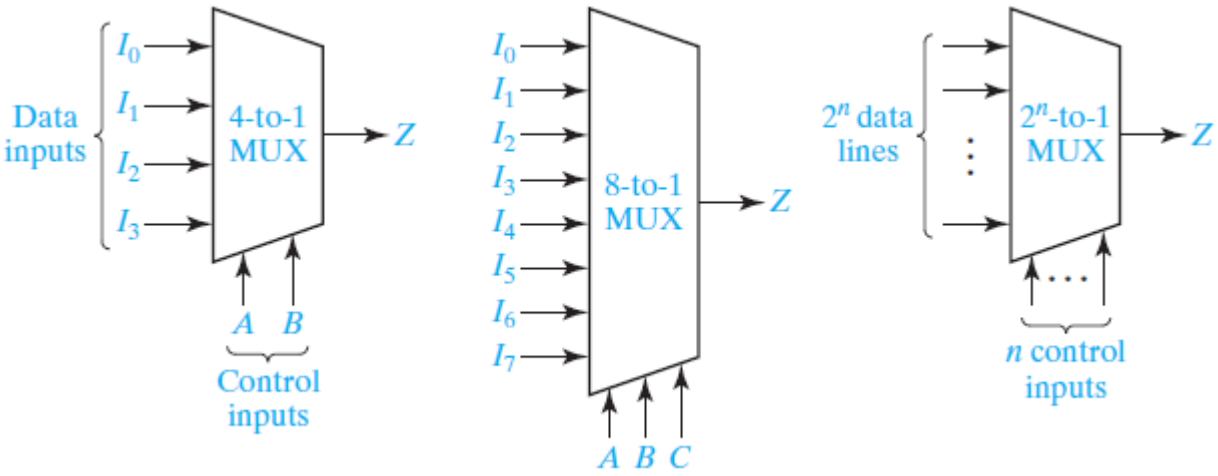


Figure 9-18 Multiplexers

Similarly, the 8-to-1 MUX selects one of eight data inputs using three control inputs. It is described by the equation:

$$Z = \bar{A}\bar{B}\bar{C}I_0 + \bar{A}\bar{B}CI_1 + \bar{A}B\bar{C}I_2 + \bar{A}BCI_3 + A\bar{B}\bar{C}I_4$$

$$+A\bar{B}CI_5 + AB\bar{C}I_6 + ABCI_7$$

Table 9-2 shows that the *data select control inputs* (S_1, S_0) are responsible for determining which data input (D_0 to D_3) is selected to be transmitted to the data-output line (Y). The S_1, S_0 inputs will be a binary code that corresponds to the data-input line that you want to select. If $S_1 = 0$, $S_0 = 0$, then D_0 is selected; if $S_1 = 0$, $S_0 = 1$, then D_1 is selected; and so on.

TABLE Data Select Input Codes for Figure 8–44

Data Select Control Inputs		
S_1	S_0	Data Input Selected
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

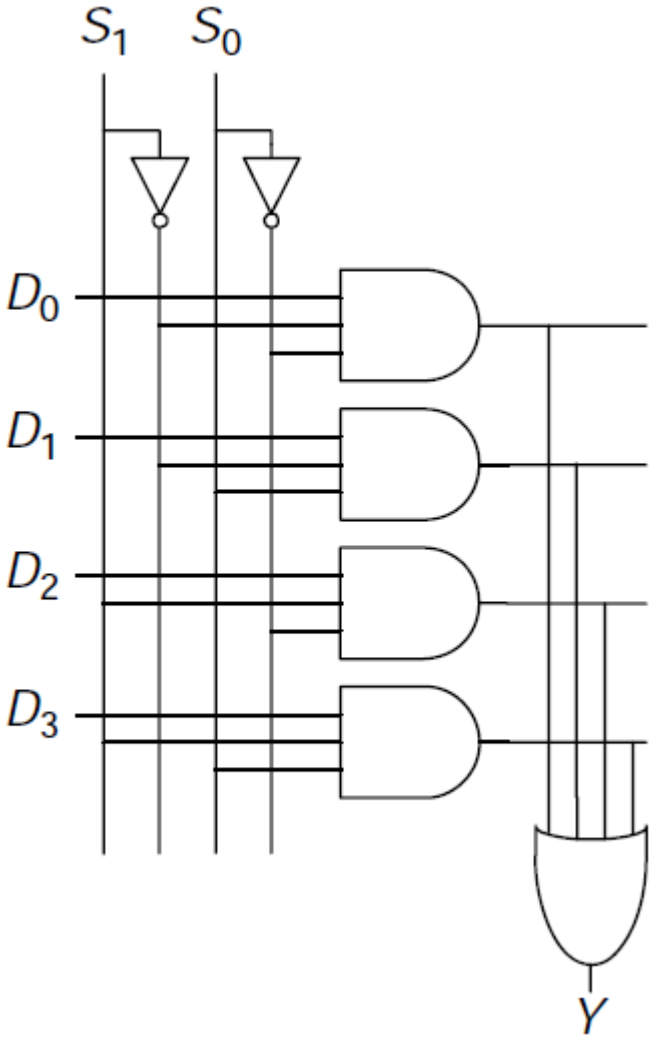


Figure 9-19 The circuit logic diagram for a 4 to 1 Multiplexer

A sample four-line multiplexer built from SSI logic gates is shown in Figure 9-20. The control inputs (S_1, S_0) take care of enabling the correct AND gate to pass just one of the data inputs through to the output. In Figure 9 – 20, 1s and 0s were placed on the diagram to show the levels that occur when selecting data input D_1. Notice that AND gate 1 is enabled, passing D_1 to the output, whereas all other AND gates are disabled.

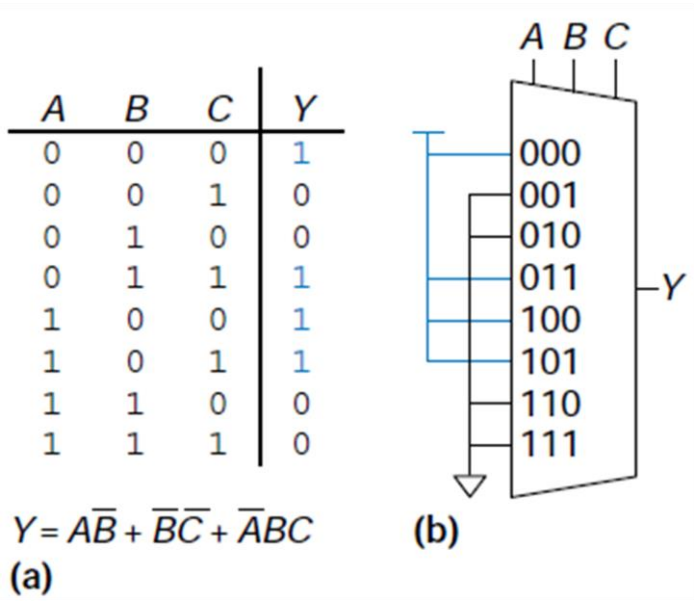
EXAMPLE 9 - 2

Alyssa P. Hacker needs to implement the function:

$$Y = A\overline{B} + \overline{B}\,\overline{C} + \overline{A}BC$$

to finish her senior project, but when she looks in her lab kit, the only part she has left is an 8:1 multiplexer. How does she implement the function?

Solution



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

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

(a)

Figure 9-20 Alyssa's circuit: (a) truth table, (b) 8:1 multiplexer implementation

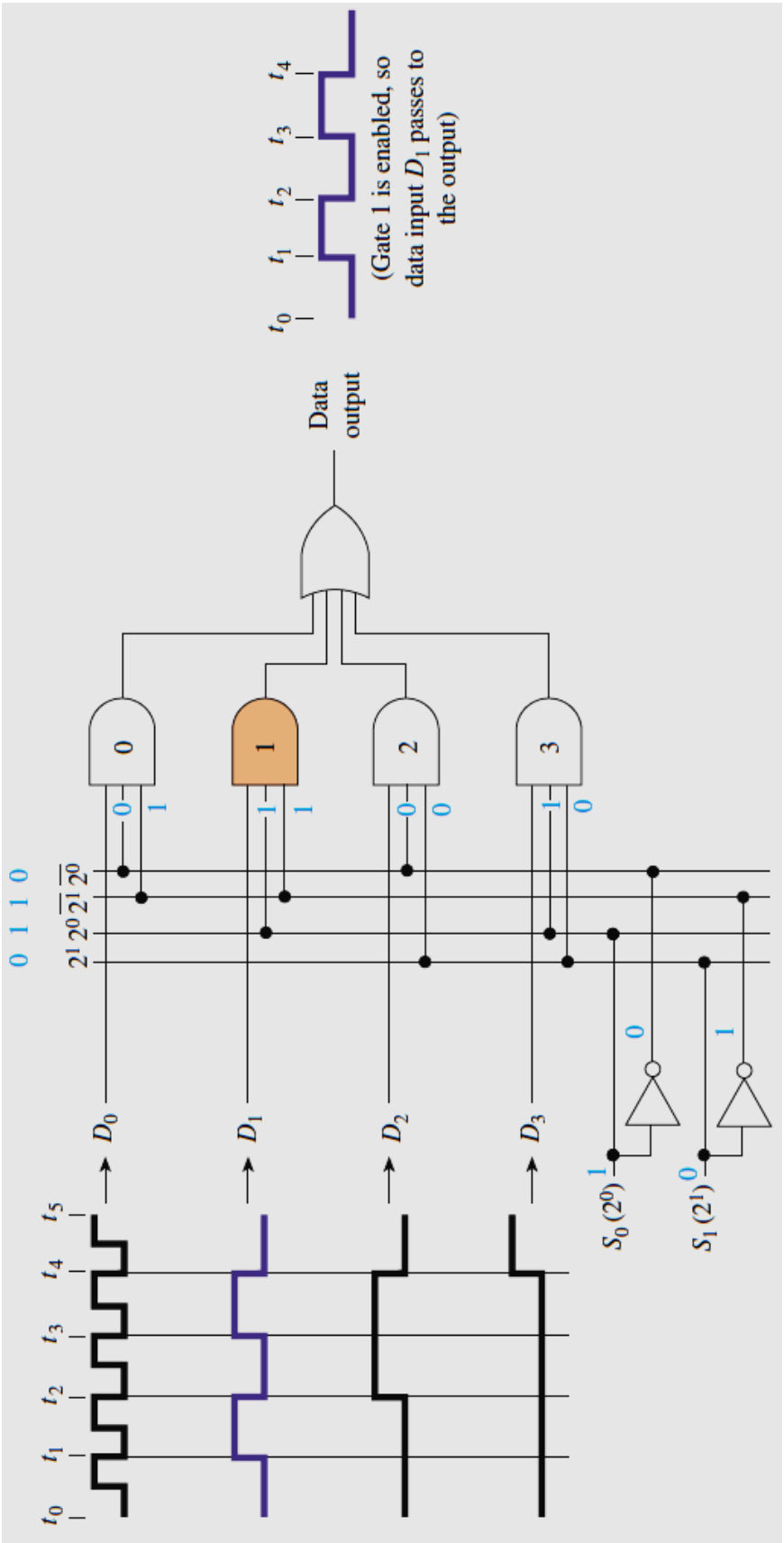


Figure 9–21 Logic diagram for a four-line multiplexer.

EXAMPLE 9 - 3

Use a multiplexer to implement the function:

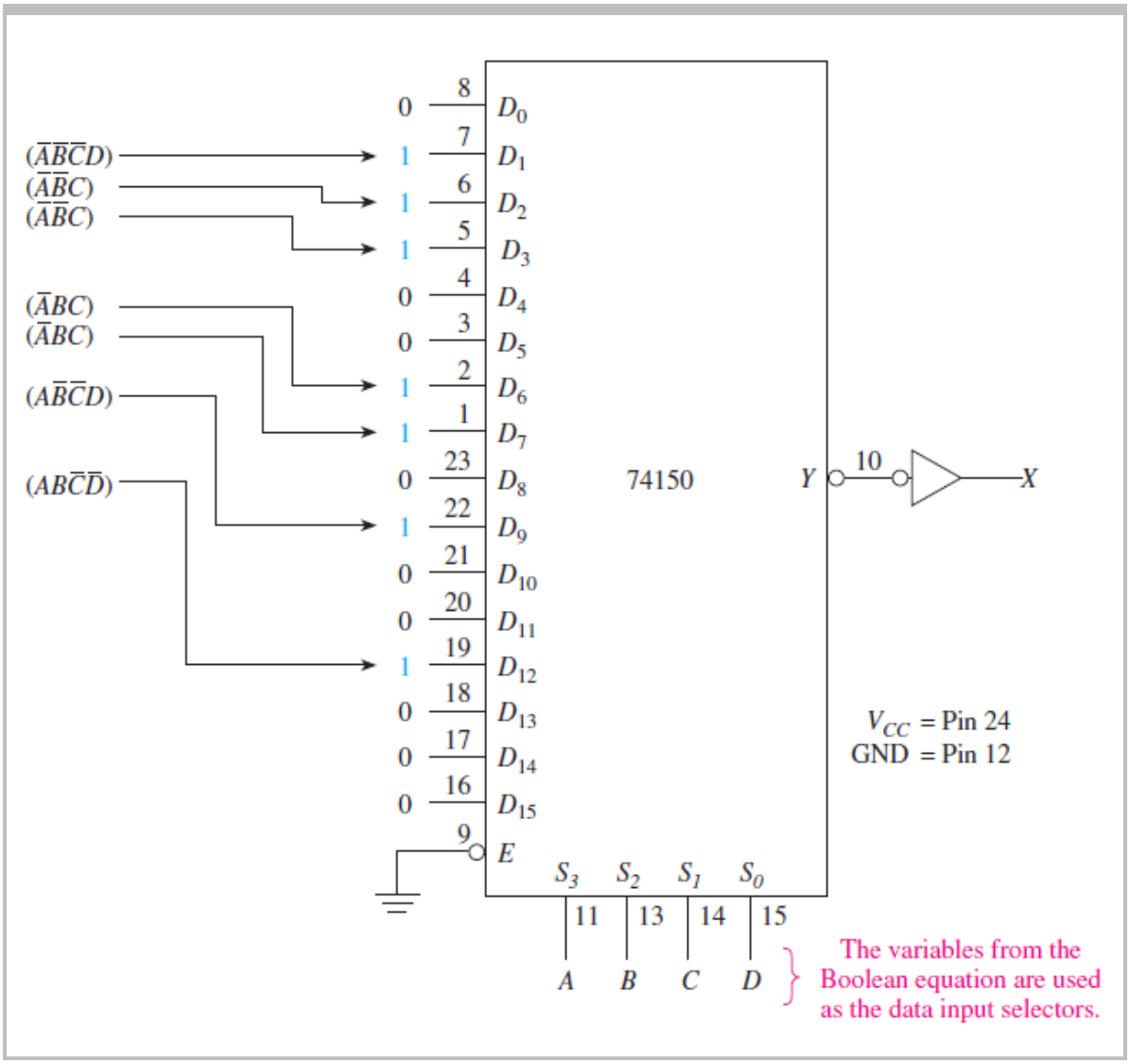
$$X = \overline{A}\overline{B}\overline{C}D + A\overline{B}\overline{C}D + AB\overline{C}\overline{D} + \overline{A}BC + \overline{A}\overline{B}C$$

Solution

If the A, B, C, and D variables are used as the data *input selectors* of a 16-line multiplexer (*four* input variables can have **16** possible combinations) and the appropriate digital levels are placed at the multiplexer data inputs, we can implement the function for X.

A **1** must be placed at each data input that satisfies any term in the Boolean equation. The truth table is:

A	B	C	D	X
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0



9.5.2 Designing Large Multiplexer Using Smaller Multiplexers

By combining small MUXs, then it is able to generate a bigger MUX; an 8×1 MUX can be constructed by *two* 4×1 multiplexers and one OR gate as shown in Figure 9-22. In this figure, S_1, S_0, and E are select lines. When $E = 0$, the *top* MUX is *enabled*, and when $E = 1$, the *lower* MUX is *enabled*. Also, the 8×1 MUX can be implemented using *two* 4×1 and *one* 2×1 MUX as shown in Figure 9.23. In this figure, when $S_2 = 0$, the output $O = 0$, and when $S_2 = 1$, the output $O = 1$.

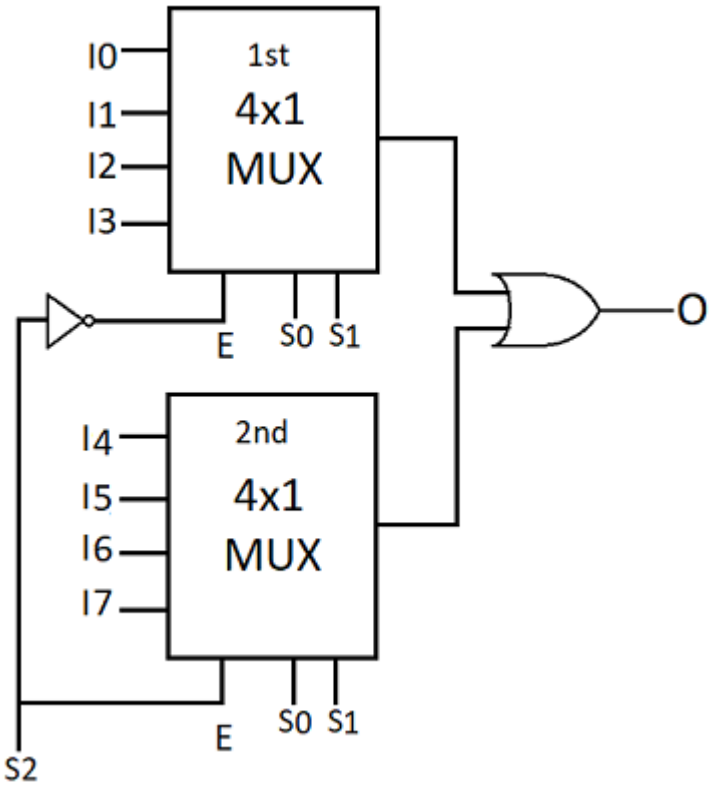


Figure 9 – 22 8×1 MUX using *two* 4×1 MUX and OR gate.

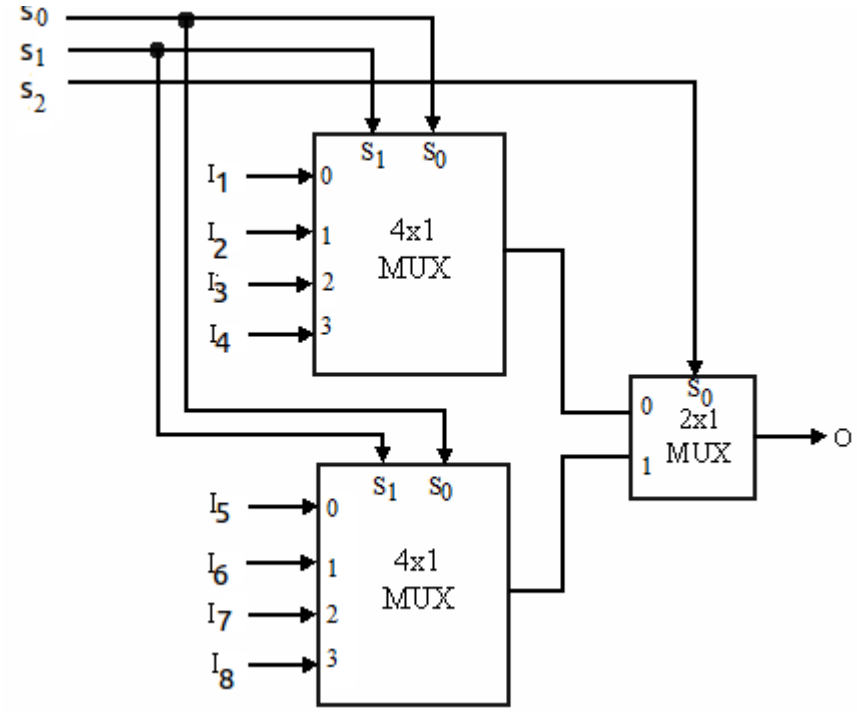


Figure 9 – 23 8×1 MUX using *two* 4×1 and *one* 2×1 MUXs.

A sample four-line multiplexer built from SSI logic gates is shown in Figure

9-20. The control inputs (S_1, S_0) take care of enabling the correct AND gate to pass just one of the data inputs through to the output. In Figure 9 – 20, 1s and 0s were placed on the diagram to show the levels that occur when selecting data input D_1. Notice that AND gate 1 is enabled, passing D_1 to the output, whereas all other AND gates are disabled.

Problems

1. **Sketch a 4 × 16 decoder using a number of 2 × 4 decoders?**

2. **Build a 4 × 16 decoder using ONLY TWO 2 × 4 decoders?**

3. **Draw the logic diagram of a two-to-four- line decoder using NOR gates only. (Include an enable input.)**

4. **A combinational circuit is specified by the following three Boolean functions:**

$$F_1(A, B, C) = \Sigma(2,\ 4,\ 7)$$

$$F_2(A, B, C) = \Sigma(0,\ 3)$$

$$F_3(A, B, C) = \Sigma(0,\ 2,\ 3,\ 4,\ 7)$$

Implement the circuit with a decoder constructed with NAND gates.

5. **Use a decoder and external gates to design the combinational circuit defined by the following three Boolean functions:**

(a)

$$F_1 = X'Y'Z' + XZ$$

$$F_2 = XY'Z' + X'Y$$

$$F_3 = X'Y'Z + XY$$

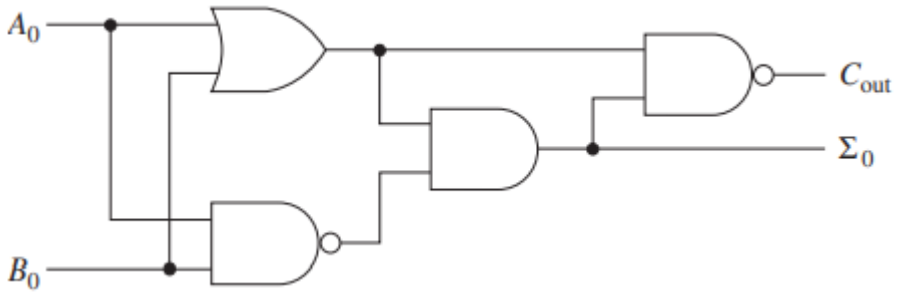
(b)

$$F_1 = (Y' + X)Z$$

$$F_2 = Y'^{Z'} + XY' + YZ'$$

$$F_3 = (X' + Y)Z$$

6. The circuit in the following Figure is an attempt to build a half-adder. Will the C_{out} and function properly? (Hint: Write the Boolean equation at C_{out} and Σ_0).



7. Draw the block diagram of a 4-bit full-adder using four full-adders.

8. What changes would have to be made to the adder/subtractor circuit if exclusive-NORs are to be used instead of exclusive ORs?

9. Determine the outputs of a full-adder for each of the following inputs:

(a) $A = 0$, $B = 1$, $C_{in} = 0$.

(b) $A = 1$, $B = 0$, $C_{in} = 1$.

(c) $A = 0$, $B = 0$, $C_{in} = 0$

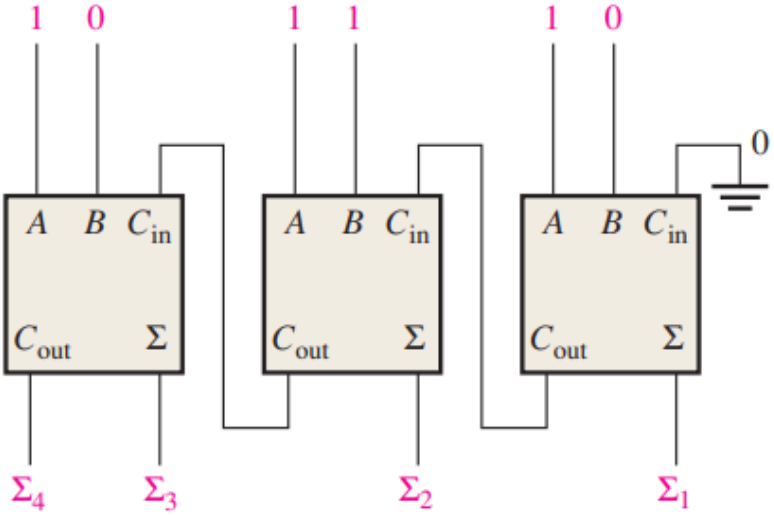
10. What are the half-adder inputs that will produce the following outputs:

(a) $\Sigma = 0$, $C_{out} = 0$.

(b) $\Sigma = 1$, $C_{out} = 0$.

(c) $\Sigma = 0$, $C_{out} = 1$.

11. For the parallel adder in the following Figure, determine the complete sum by analysis of the logical operation of the circuit. Verify your result by longhand addition of the two input numbers.



Scientific References

1. M . Morris Mano, Michael D. Ciletti, Digital Design, fourth edition, Pearson College, **2007**.
2. Saha, N. Manna, Digital Principles and Logic Design, Infinity Science Press LLC, **2007**.
3. William Kleitz, Digital Electronics: A Practical Approach with VHDL, Ninth Edition, Pearson Education, **2012**.
4. Charles H. Roth, Jr. and Larry L. Kinney, Fundamentals of Logic Design, seventh edition, Printed in the United States of America, **2014**.
5. Ata Elahi, Computer Systems: Digital Design, Fundamentals of Computer Architecture and Assembly Language, Springer, Cham, **2018**.