Digital Electronics

Reference Books:

1. Digital Logic and Computer Design 5th edition, by M. Morris Mano

This PDF contains the notes from the standards books and are only meant for GATE CSE aspirants.

Notes Compiled By-
Manu Thakur
Mtech CSE, IIT Delhi
worstguymanu@gmail.com
https://www.facebook.com/Worstguymanu

Digital Electronics

Duality:
This important property of Boolean algebra is called the **duality principle** and states that every algebraic expression remains valid if the operators and identity elements are interchanged.

1. $x(x' + y) = xx' + xy = 0 + xy = xy.$
2. $x + x'y = (x + x')(x + y) = 1(x + y) = x + y.$

Functions 1 and 2 are the dual of each other and use dual expressions in corresponding steps.

BOOLEAN FUNCTIONS

A Boolean function described by an algebraic expression consists of binary variables, the constants 0 and 1, and the logic operation symbols. For a given value of the binary variables, the function can be equal to either 1 or 0.
A Boolean function can be represented in a truth table. The number of rows in the truth table is 2^n, where n is the number of variables in the function.
A Boolean function can be transformed from an algebraic expression into a circuit diagram composed of logic gates connected in a particular structure:

Table 2.2
Truth Tables for F_1 and F_2

x	y	z	F_1	F_2
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	0	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	0
1	1	1	1	0

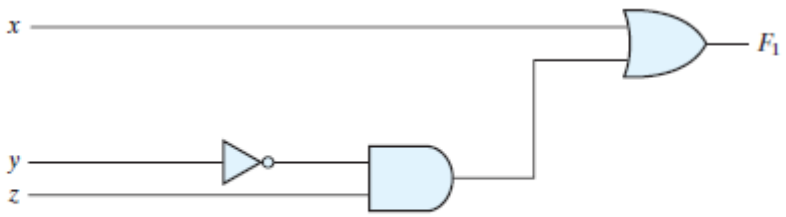


FIGURE 2.1
Gate implementation of $F_1 = x + y'z$

In logic-circuit diagrams, the variables of the function are taken as the **inputs** of the circuit and the **binary variable F1 is taken as the output** of the circuit.
Note: There is only **one way that a Boolean function can be represented in a truth table**. However, when the function is **in algebraic form, it can be expressed in a variety of ways, all of which have equivalent logic**. The particular expression used to represent the function will dictate the interconnection of gates in the logic-circuit diagram.

Consider, for example, the following Boolean function:

F2 = x'y'z + x'yz + xy'

The function is equal to 1 when xyz = 001 or 011 or when xy = 10 (irrespective of the value of z, contributes two 1's) and is equal to 0 otherwise. This set of conditions produces four 1's and four 0's for F2.

Complement of a Function:

Take duals of functions and complementing each literal:

1. $F_1 = x'yz' + x'y'z.$
   The dual of F_1 is $(x' + y + z')(x' + y' + z).$
   Complement each literal: $(x + y' + z)(x + y + z') = F_1'.$
2. $F_2 = x(y'z' + yz).$
   The dual of F_2 is $x + (y' + z')(y + z).$
   Complement each literal: $x' + (y + z)(y' + z') = F_2'.$

CANONICAL AND STANDARD FORMS

Minterms and Maxterms:

A binary variable may appear either in its normal form (x) or in its complement form (x'). Now consider two binary variables x and y combined with an AND operation. Since each variable may appear in either form, there are four possible combinations: x'y', x'y, xy', and xy. Each of these four AND terms is called a **minterm**, or a **standard product**. variables can be combined to **form 2^n minterms**.
In a similar fashion, n variables forming an OR term, with each variable being primed or unprimed, provide 2n possible combinations, called **maxterms**, or **standard sums**.

Table 2.3
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

$$f_1 = x'y'z + xy'z' + xyz = m_1 + m_4 + m_7$$

$$f_2 = x'yz + xy'z + xyz' + xyz = m_3 + m_5 + m_6 + m_7$$

Now consider the complement of a Boolean function

$$f_1' = x'y'z' + x'yz' + x'yz + xy'z + xyz'$$

If we take the complement of f_1', we obtain the function f_1:

$$\begin{aligned} f_1 &= (x + y + z)(x + y' + z)(x' + y + z')(x' + y' + z) \\ &= M_0 \cdot M_2 \cdot M_3 \cdot M_5 \cdot M_6 \end{aligned}$$

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

$$\begin{aligned} f_2 &= (x + y + z)(x + y + z')(x + y' + z)(x' + y + z) \\ &= M_0 M_1 M_2 M_4 \end{aligned}$$

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

Sum of Minterms:

for n binary variables, one can obtain 2^n distinct minterms and that any Boolean function can be expressed as a sum of minterms. **The minterms whose sum defines the Boolean function are those which give the 1's of the function in a truth table**.

Since the function can be either 1 or 0 for each minterm, and since there are 2^n minterms, **one can calculate all the functions that can be formed with n variables to be 2^(2^n)**.

Example:

EXAMPLE 2.4

Express the Boolean function $F = A + B'C$ as a sum of minterms. The function has three variables: A, B, and C. The first term A is missing two variables; therefore,

$$A = A(B + B') = AB + AB'$$

This function is still missing one variable, so

$$\begin{aligned} A &= AB(C + C') + AB'(C + C') \\ &= ABC + ABC' + AB'C + AB'C' \end{aligned}$$

The second term $B'C$ is missing one variable; hence,

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

Combining all terms, we have

$$\begin{aligned} F &= A + B'C \\ &= ABC + ABC' + AB'C + AB'C' + A'B'C \end{aligned}$$

But $AB'C$ appears twice, and according to theorem 1 ($x + x = x$), it is possible to remove one of those occurrences. Rearranging the minterms in ascending order, we finally obtain

$$\begin{aligned} F &= A'B'C + AB'C + AB'C + ABC' + ABC \\ &= m_1 + m_4 + m_5 + m_6 + m_7 \end{aligned}$$

■

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

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

An alternative procedure for deriving the minterms of a Boolean function is to obtain the truth table of the function directly from the algebraic expression and then read the minterms from the truth table.

F = A + B'C

Table 2.5
Truth Table for F = A + 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

1's under F for those combinations for which $A = 1$ and $BC = 01$. From the truth table, we can then read the five minterms of the function to be 1, 4, 5, 6, and 7.

Product of Maxterms

Each of the 2^(2^n) functions of n binary variables can be also expressed as a product of maxterms. This may be done by using the distributive law, x + yz = (x + y)(x + z). Then any missing variable x in each OR term is ORed with xx'.

EXAMPLE 2.5

Express the Boolean function $F = xy + x'z$ as a product of maxterms. First, convert the function into OR terms by using the distributive law:

$$\begin{aligned} F &= xy + x'z = (xy + x')(xy + z) \\ &= (x + x')(y + x')(x + z)(y + z) \\ &= (x' + y)(x + z)(y + z) \end{aligned}$$

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

$$\begin{aligned} x' + y &= x' + y + zz' = (x' + y + z)(x' + y + z') \\ x + z &= x + z + yy' = (x + y + z)(x + y' + z) \\ y + z &= y + z + xx' = (x + y + z)(x' + y + z) \end{aligned}$$

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

$$\begin{aligned} F &= (x + y + z)(x + y' + z)(x' + y + z)(x' + y + z') \\ &= M_0M_2M_4M_5 \end{aligned}$$

A convenient way to express this function is as follows:

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

Conversion between Canonical Forms

The **complement of a function** expressed as the sum of minterms equals the **sum of minterms missing from the original function**.
This is because the original function is expressed by those minterms **which make the function equal to 1,** whereas its complement is a 1 for those minterms for **which the function is a 0**.

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

This function has a complement that can be expressed as

$$F'(A, B, C) = \Sigma(0, 2, 3) = m_0 + m_2 + m_3$$

Now, if we take the complement of F' by DeMorgan's theorem, we obtain F in a different form:

$$F = (m_0 + m_2 + m_3)' = m_0' \cdot m_2' \cdot m_3' = M_0 M_2 M_3 = \Pi(0, 2, 3)$$

The last conversion follows from the definition of minterms and maxterms as shown in Table 2.3. From the table, it is clear that the following relation holds:

$$m_j' = M_j$$

That is, the **maxterm** with subscript j is a complement of the **minterm** with the same subscript j and vice versa.

Table 2.6
Truth Table for F = xy + x'z

x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Minterms

Maxterms

Since there is a total of eight minterms or maxterms in a function of three variables, we determine the missing terms to be 0, 2, 4, and 5. The function expressed as a product of maxterms is

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

"Note: each minterm or maxterm must contain, by definition, all the variables, either complemented or uncomplemented."

Standard Forms:

Another way to express Boolean functions is in standard form. In this configuration, the terms that form the function **may contain one, two, or any number of literals**. There are two types of standard forms: **the sum of products and products of sums**.
The **sum of products (SOP)** is a Boolean expression **containing AND terms**, called product terms, **with one or more literals each**. The **sum denotes the ORing** of these terms.

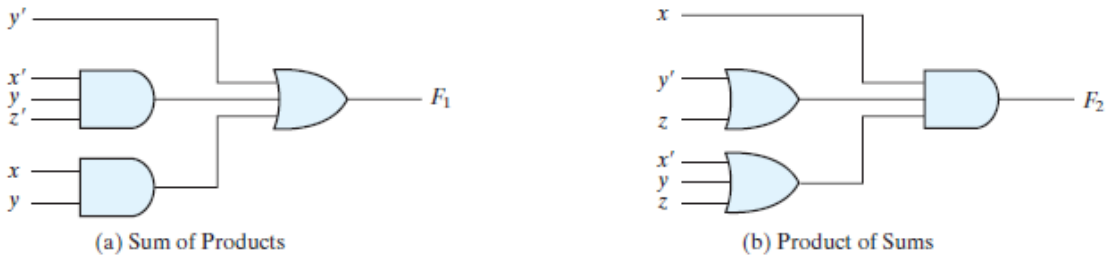
F1 = Y' + XY + X'YZ'

The expression has three product terms, with one, two, and three literals. Their sum is, in effect, an OR operation. The logic diagram of a sum-of-products expression consists **of a group of AND gates followed by a single OR** gate.

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

F2 = X(Y' + Z)(X'+Y+Z')

This expression has three sum terms**, with one, two, and three literals**. The product is an AND operation.



(a) Sum of Products (b) Product of Sums

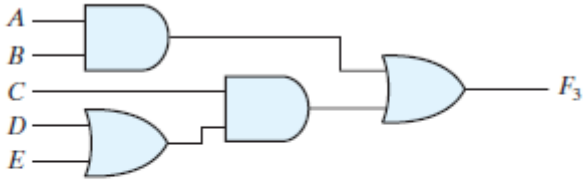
FIGURE 2.3
Two-level implementation

Note: This standard type of expression results in a two-level structure of gates.

A Boolean function may be expressed in a **nonstandard form**. For example, the function

F3 = AB + C(D+E)

Is neither SOP or POS, and there are three levels of gating in this circuit. Though it can be changed to a standard form.



(a) $AB + C(D + E)$

FIGURE 2.4
Three- and two-level implementation

Note: a two-level implementation is preferred because it produces the least amount of delay through the gates when the signal propagates from the inputs to the output.

Positive and Negative Logic:

Choosing the high – level H to represent logic 1 defines a positive logic system. Choosing the low – level L to represent logic 1 defines a negative logic system.

x	y	z
L	L	L
L	H	L
H	L	L
H	H	H

(a) Truth table with H and L

If we are considering positive logic, we'll get following corresponding truth table:

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

(c) Truth table for positive logic

If we are considering negative logic, we'll get following corresponding truth table:

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

(e) Truth table for negative logic

Note: We have replace H and L with 1 and 0 or 0 and 1 accordingly. Positive Logic and Negative Logic are dual of each other.

GATE – LEVEL MINIMIZATION

Gate-level minimization is the design task of finding an optimal gate-level implementation of the Boolean functions describing a digital circuit.

THE MAP METHOD

A K-map is a diagram made up of squares, with each square representing one minterm of the function that is to be minimized.

The simplified expressions produced by the map are always in one of the two standard forms: **sum of products** or **product of sums**.

Note: the simplest (Minimum) expression is not unique. Any two adjacent squares in the map differ by only one variable, which is primed in one square and unprimed in the other**.**

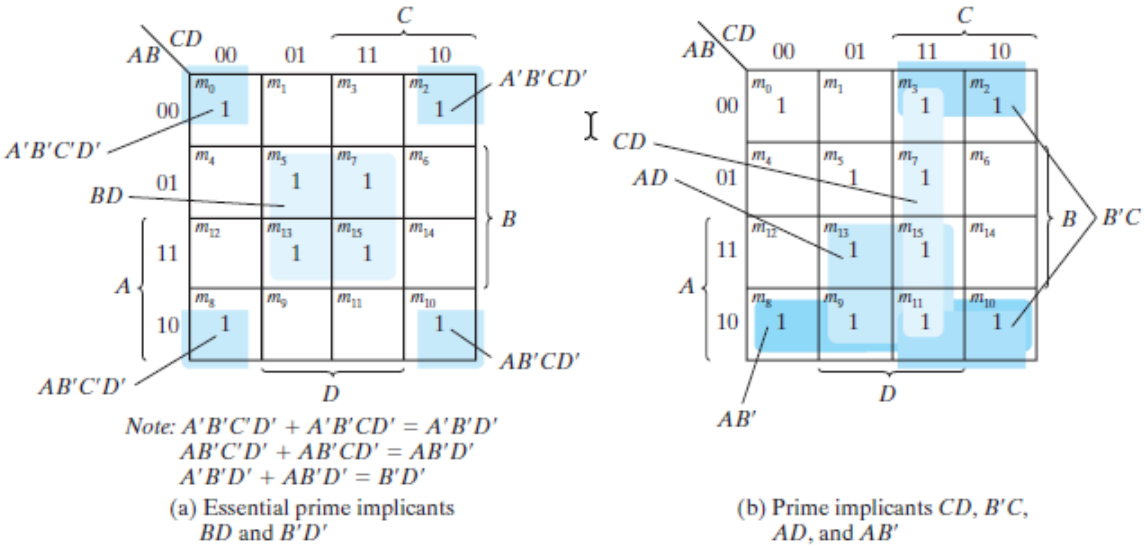
Prime Implicants

In choosing adjacent squares in a map, we must ensure that (1) all the minterms of the function are covered when we combine the squares, (2) the number of terms in the expression is minimized, and (3) there are no redundant terms (i.e., minterms already covered by other terms).
A prime implicant is a product term obtained by combining the maximum possible number of adjacent squares in the map.
If a minterm in a square is covered **by only one prime implicant, that prime implicant is said to be essential**.
The prime implicants of a function can be obtained from the map by combining all possible maximum numbers of squares.

This means that a single 1 on a map represents a prime implicant if it is not adjacent to any other 1's. Two adjacent 1's form a prime implicant, provided that they are not within a group of four adjacent squares. Four adjacent 1's form a prime implicant if they are not within a group of eight adjacent squares, and so on. The prime implicant is essential if it is the only prime implicant that covers the minterm.

Consider the following four-variable Boolean function:

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



Note: $A'B'C'D' + A'B'CD' = A'B'D'$
$AB'C'D' + AB'CD' = AB'D'$
$A'B'D' + AB'D' = B'D'$

(a) Essential prime implicants BD and $B'D'$

(b) Prime implicants CD, $B'C$, AD, and AB'

FIGURE 3.11
Simplification using prime implicants

There are total 6 Prime Implicants as BD, B'D', CD, B'C, AD and AB'. In which BD and B'D are essential Prime Implicants and rest four CD, B'C, AD and AB' are only prime implicants. The two essential prime implicants cover eight minterms.

that cover minterms m_3, m_9, and m_{11}. There are four possible ways that the function can be expressed with four product terms of two literals each:

$$\begin{aligned} F &= BD + B'D' + CD + AD \\ &= BD + B'D' + CD + AB' \\ &= BD + B'D' + B'C + AD \\ &= BD + B'D' + B'C + AB' \end{aligned}$$

The simplified expression is obtained from the logical **sum of all the essential prime implicants, plus other prime implicants that may be needed to cover any remaining minterms** not covered by the essential prime implicants.

PRODUCT-OF-SUMS SIMPLIFICATION:

If we mark the empty squares by 0's and combine them into valid adjacent squares, we obtain a simplified **sum-of-products** expression of the complement of the function. The complement of F' gives us back the function F in product-of-sums form.

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

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

The 1's marked in the map of Fig. 3.12 represent all the minterms of the function. The squares marked with 0's represent the minterms not included in F and therefore denote the complement of F. Combining the squares with 1's gives the simplified function in sum-of-products form:

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

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

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

Applying DeMorgan's theorem (by taking the dual and complementing each literal as described in Section 2.4), we obtain the simplified function in product-of-sums form:

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

Note: For option (a) consider those cells which contain 1's and find SOP and for option (b) consider those cells which contain 0's and find POS. **Both are equivalent**.

Map for Example 3.7, $F(A, B, C, D) = \Sigma(0,1, 2, 5, 8, 9,10) = B'D' + B'C' + A'C'D = (A' + B')(C' + D')(B' + D)$

DON'T-CARE CONDITIONS

As an example, the four-bit binary code for the decimal digits has six combinations that are not used and consequently are considered to be unspecified. Functions that have unspecified outputs for some input combinations are called **incompletely specified functions** . In most applications, we simply don't care what value is assumed by the function for the unspecified minterms. For this reason, it is customary to call the unspecified minterms of a function **don't-care conditions** . These don't-care conditions can be used on a map to provide further simplification of the Boolean expression.

don't-care minterm is a combination of variables whose logical value is not specified. Such a minterm **cannot be marked with a 1 in the map, because it would require that the function always be a 1 for such a combination. Likewise, putting a 0 on the square requires the function to be 0**. To distinguish the don't-care condition from 1's and 0's, an X is used. Thus, an X inside a square in the map indicates that we don't care whether the value of 0 or 1 is assigned to F for the particular minterm. In choosing adjacent squares to simplify the function in a map, the don't-care minterms

may be assumed to be either 0 or 1. When simplifying the function, we can choose to include each don't-care minterm with either the 1's or the 0's, depending on which combination gives the simplest expression.

NAND Circuits:

A convenient way to implement a Boolean function with NAND gates is to obtain the simplified Boolean function in terms of Boolean operators and then convert the function to NAND logic.

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

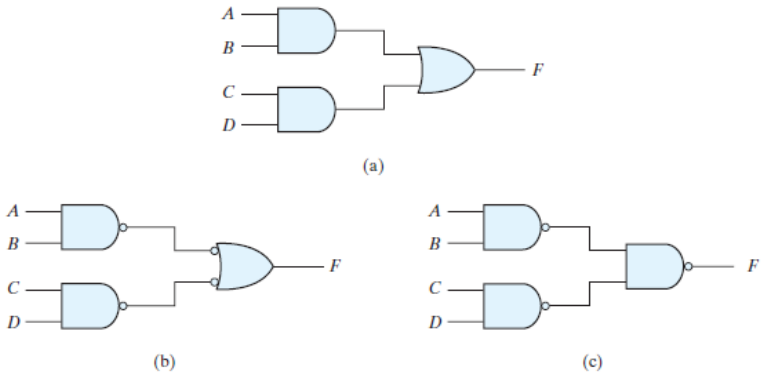


FIGURE 3.18
Three ways to implement $F = AB + CD$

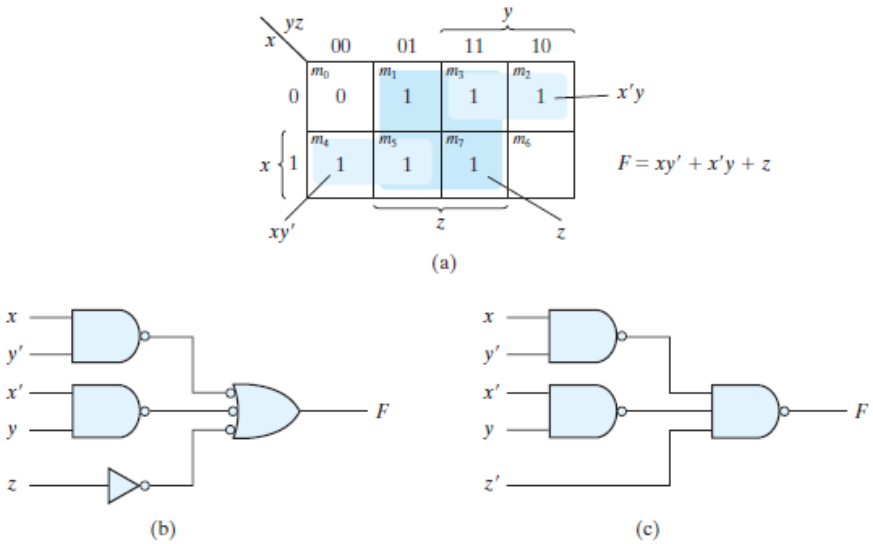
EXAMPLE 3.9

Implement the following Boolean function with NAND gates:

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

The first step is to simplify the function into sum-of-products form
F=XY' + X'Y + Z

The two-level NAND implementation is shown



NOR Implementation:

The NOR operation is the dual of the NAND operation. A two-level implementation with NOR gates requires that the function be simplified into **product-of-sums form**.

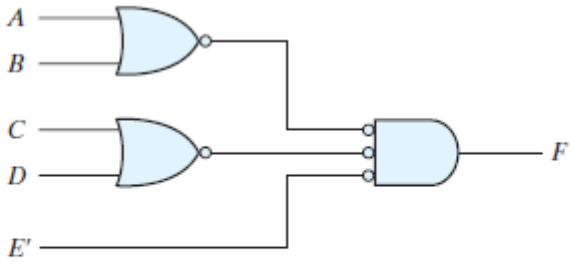


FIGURE 3.24
Implementing $F = (A + B)(C + D)E$

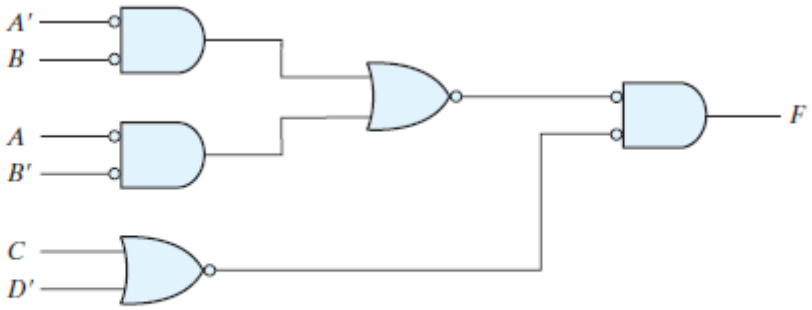


FIGURE 3.25
Implementing $F = (AB' + A'B)(C + D')$ with NOR gates

EXCLUSIVE-OR FUNCTION

It can be shown that the exclusive-OR operation is both commutative and associative; that is

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

and

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

We can evaluate a three or more variables can be expressed without parentheses. A two-input exclusive-OR function is constructed with conventional gates using two inverters, two AND gates, and an OR gate.

Odd Function

$$\begin{aligned} A \oplus B \oplus C &= (AB' + A'B)C' + (AB + A'B')C \\ &= AB'C' + A'BC' + ABC + A'B'C \\ &= \Sigma(1, 2, 4, 7) \end{aligned}$$

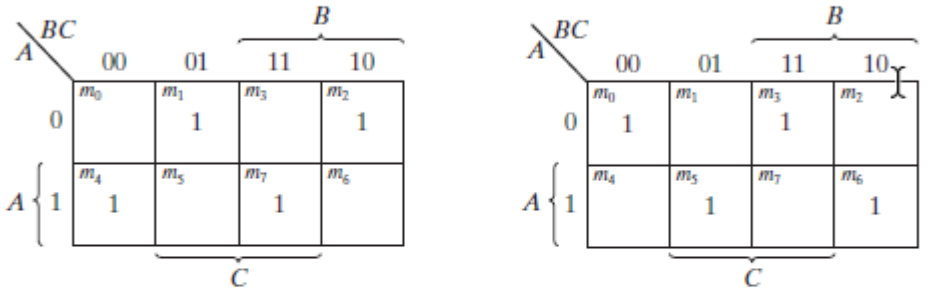
Here we can see that each minterm has Odd number of 1's either one or three 1's. The multiple-variable exclusive-OR operation is defined as an **odd function**.

The complement of an odd function is an even function.

The three-variable even function is equal to 1 when an even number of its variables is equal to 1.

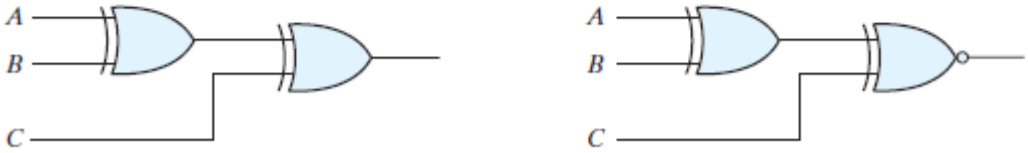
Consider now the four-variable exclusive-OR operation. By algebraic manipulation, we can obtain the sum of minterms for this function:

$$\begin{aligned} A \oplus B \oplus C \oplus D &= (AB' + A'B) \oplus (CD' + C'D) \\ &= (AB' + A'B)(CD + C'D') + (AB + A'B')(CD' + C'D) \\ &= \Sigma(1, 2, 4, 7, 8, 11, 13, 14) \end{aligned}$$



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

FIGURE 3.31
Map for a three-variable exclusive-OR function



(a) 3-input odd function (b) 3-input even function

FIGURE 3.32
Logic diagram of odd and even functions

This is an odd function because the binary values of all the minterms have an odd number of 1's. **The four minterms of the function are a unit distance apart from each other**.

Combinational Circuits:

Combinational Circuits:

If the registers are included with the combinational gates, then the total circuit must be considered to be a **sequential circuit**.
A combinational circuit also can be described by m Boolean functions, one for each output variable.
Note: The diagram of a combinational circuit has logic gates with no feedback paths or memory elements

Literal:

A literal to be a single variable within a term, in **complemented or uncomplemented** form.
Example:

i. **F2 = x'y'z + x'yz + xy'** this function has **3 terms** and **8 literals**.
ii. **F2 = xy' + x'z has** two terms and 4 literals.

Table 4.2
Truth Table for Code Conversion Example

Input BCD				Output Excess-3 Code			
A	B	C	D	w	x	y	z
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

With there are 16 entries possible with 4 variables but there are only 10 entries from 0 to 9 so in K-**Map we will keep don't care from 11 to 15**.

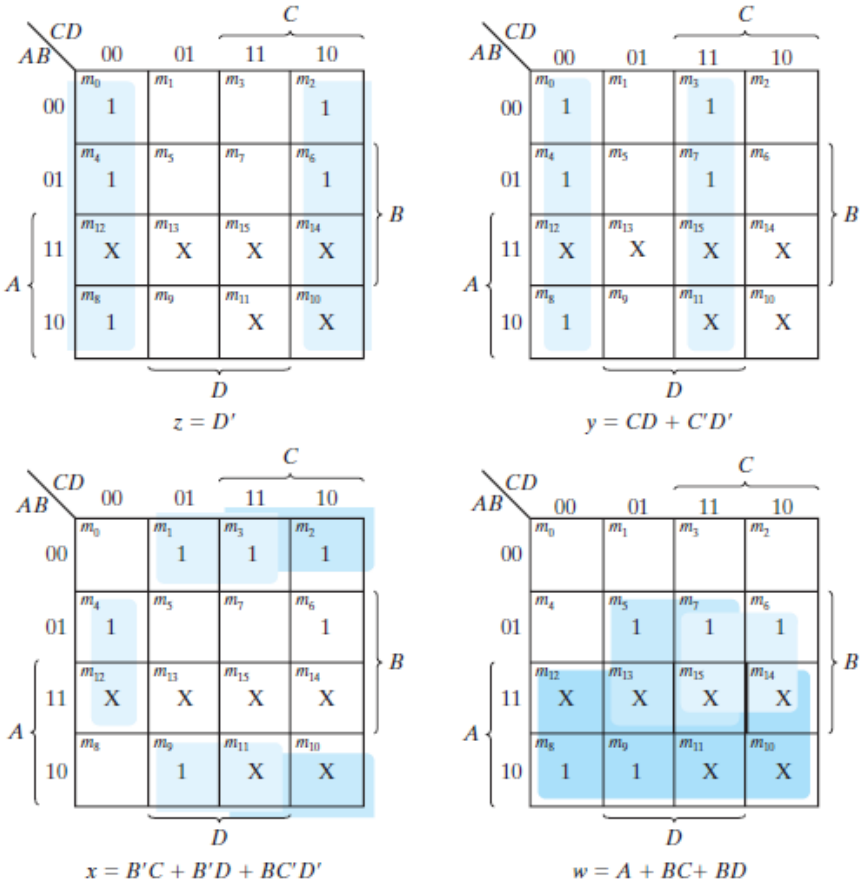


FIGURE 4.3
Maps for BCD-to-excess-3 code converter

implemented with three or more levels of gates:

$$z = D'$$
$$y = CD + C'D' = CD + (C + D)'$$
$$x = B'C + B'D + BC'D' = B'(C + D) + BC'D'$$
$$= B'(C + D) + B(C + D)'$$
$$w = A + BC + BD = A + B(C + D)$$

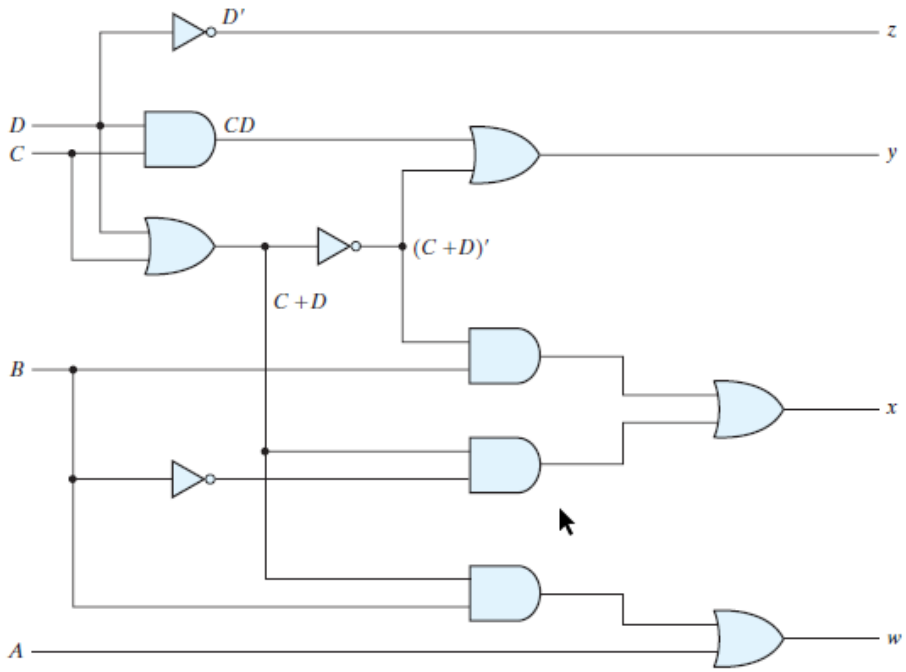


FIGURE 4.4
Logic diagram for BCD-to-excess-3 code converter

Binary Adder:

A binary adder is a digital circuit that produces the arithmetic sum of two binary numbers. It can be constructed with full adders connected in cascade, with the output carry from each full adder connected to the input carry of the next full adder in the chain.

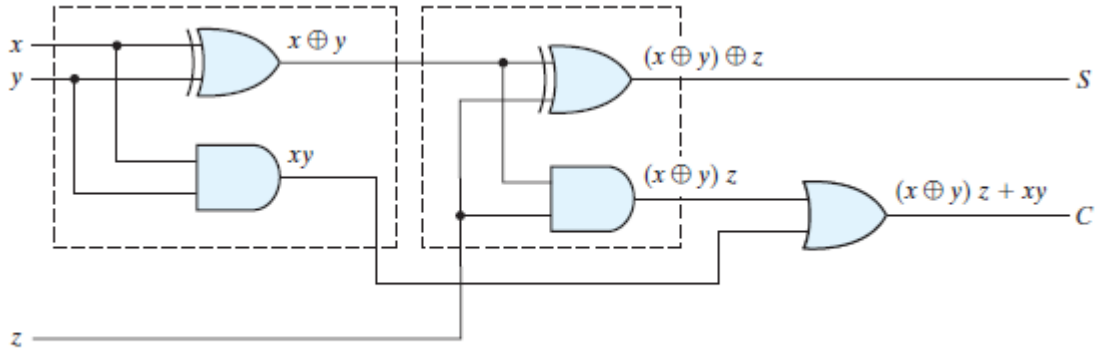


FIGURE 4.8
Implementation of full adder with two half adders and an OR gate

Addition of n-bit numbers requires a chain of n full adders or a chain of **one-half adder and n-1 full adders**. The input carry to the least significant position is fixed at 0. Interconnection of four full-adder (FA) circuits to provide a four-bit binary **Ripple Carry Adder**. The carries are connected in a chain through the full adders. The input carry to the adder is C0, and it ripples through the full adders to the output carry C4. The S outputs generate the required sum bits. **An n -bit adder requires n full adders, with each output carry connected to the input carry of the next higher order full adder.**

6 (Augend)
+3 (Addend)
---
9 (Sum)

Subscript *i*:	3	2	1	0	
Input carry	0	1	1	0	C_i
Augend	1	0	1	1	A_i
Addend	0	0	1	1	B_i
Sum	1	1	1	0	S_i
Output carry	0	0	1	1	C_{i+1}

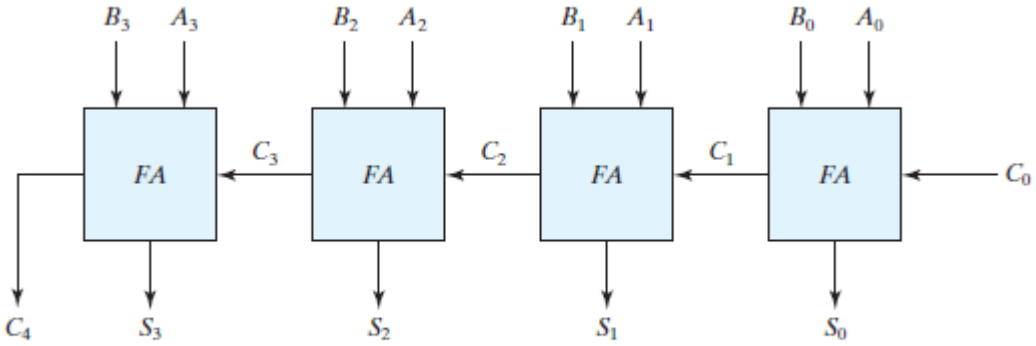


FIGURE 4.9
Four-bit adder

Carry Propagation:

The input and output variables use the subscript i to denote a typical stage of the adder. The signals at Pi and Gi settle to their steady-state values after they propagate through their respective gates. These two signals are common to all half adders and depend on only the input augend and addend bits. The signal from the input carry Ci to the output carry Ci+1 propagates through **an AND gate and an OR gate, which constitute two gate levels.** If there are four full adders in the adder, the output carry C4 would have 2 * 4 = 8 gate levels from C0 to C4. For an n -bit adder, there are 2 n gate levels for the carry to propagate from input to output.

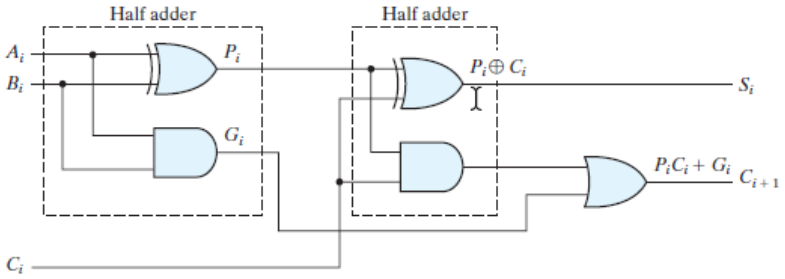


FIGURE 4.10
Full adder with *P* and *G* shown

Carry Look Ahead Adder:

If we define two new binary variables,
Pi = Ai EXOR Bi
Gi = Ai*Bi

the output sum and carry can respectively be expressed as
Si = Pi EXOR Ci
Ci+1 = Gi + PiCi
Gi is called a **carry generate**, and it produces a carry of 1 when both Ai and Bi are 1, regardless of the input carry Ci. Pi is called a **carry propagate**, because it determines whether a carry into stage i will propagate into stage i + 1.

$$C_0 = \text{input carry}$$
$$C_1 = G_0 + P_0C_0$$

$$C_2 = G_1 + P_1C_1 = G_1 + P_1(G_0 + P_0C_0) = G_1 + P_1G_0 + P_1P_0C_0$$
$$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 = P_2P_1P_0C_0$$

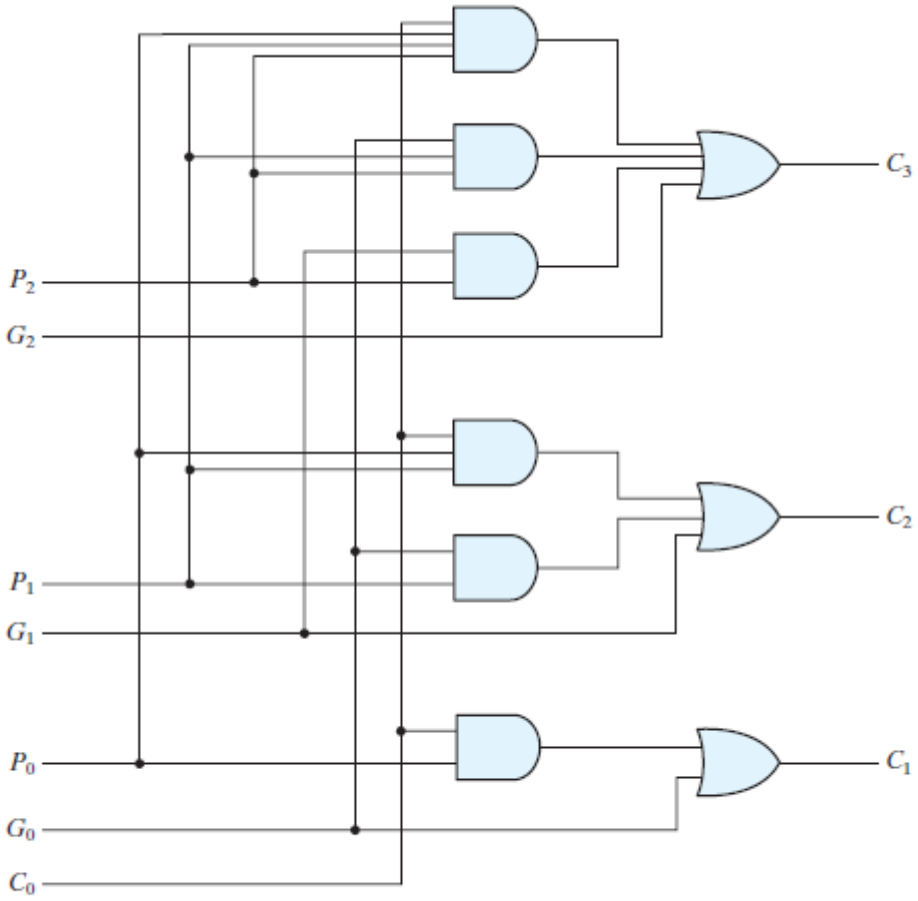


FIGURE 4.11
Logic diagram of carry lookahead generator

Each sum output requires two exclusive-OR gates. The output of the first exclusive-OR gate generates the Pi variable, and the AND gate generates the Gi variable. The carries are propagated through the carry lookahead generator.

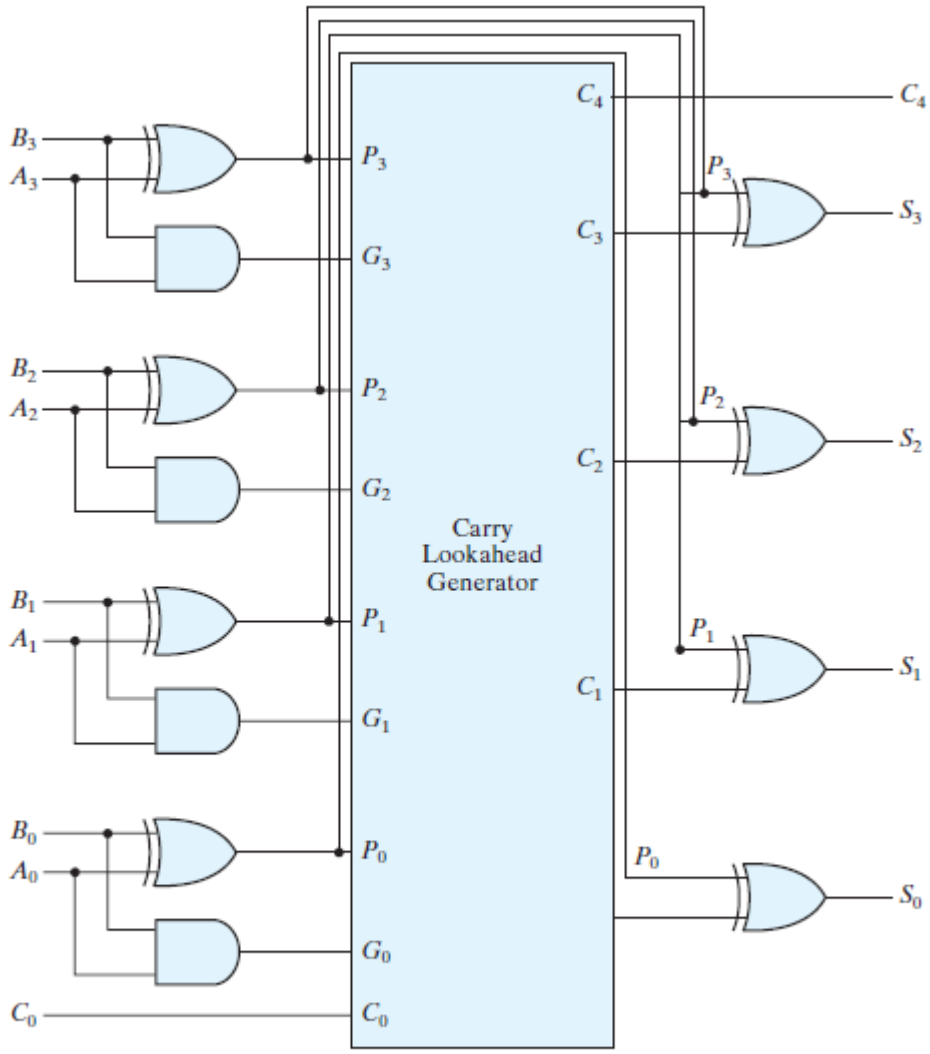


FIGURE 4.12
Four-bit adder with carry lookahead

NOTE: outputs S1 through S3 have equal propagation delay times. (The two-level circuit for the output carry C4 is not shown)

Binary Subtractor:

The subtraction of unsigned binary numbers can be done most conveniently by means of complements.

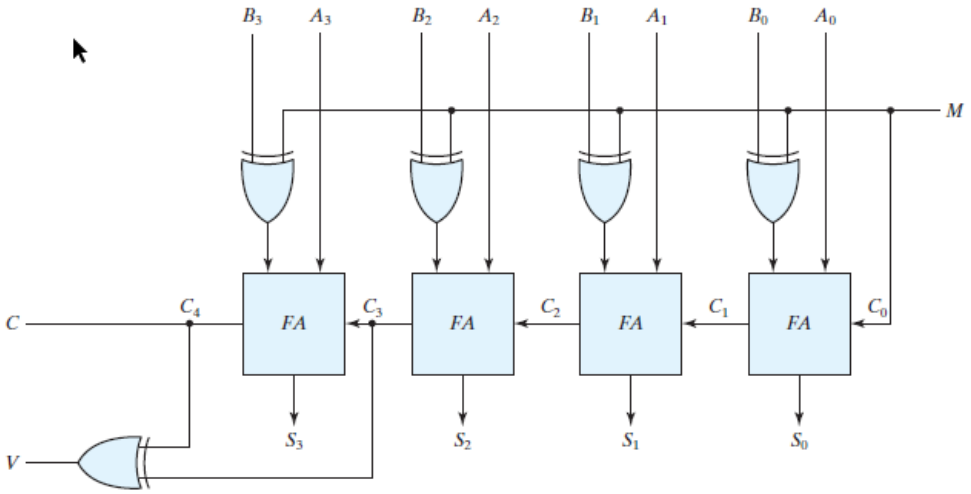


FIGURE 4.13
Four-bit adder–subtractor (with overflow detection)

The input carry C0 must be equal to 1 when subtraction is performed.
The mode input M controls the operation. When M = 0, the circuit is an adder, and when M = 1, the circuit becomes a subtractor.
Note: The exclusive-OR with output V is for detecting an overflow.

Overflow:

When two numbers with n digits each are added and the sum is a number occupying **n + 1 digits**, we say that an overflow occurred, result that contains n + 1 bits cannot be accommodated by an **n -bit word.** This is true for binary or decimal numbers, signed or unsigned.
The detection of an overflow after the addition of two binary numbers depends on whether the numbers are considered to be signed or unsigned.
When two **unsigned numbers** are added, an overflow is detected from the **end carry out of the most significant position**.
In the case of signed numbers, two details are important: the leftmost bit always represents the **sign**, and negative numbers are in 2's-complement form. When two signed numbers are added, the **sign bit is treated as part of the number and the end carry does not indicate an overflow**.
An overflow cannot occur after an addition if one number is positive and the other is negative. An overflow may occur if the two numbers added are both positive and both negative.

Two signed binary numbers, +70 and +80, are stored in two eight-bit registers. The range of numbers that each register can accommodate is from binary +127 to binary -128. Since the sum of the two numbers is +150, it exceeds the capacity of an eight-bit register.

carries:	0 1	carries:	1 0
+70	0 1000110	−70	1 0111010
+80	0 1010000	−80	1 0110000
+150	1 0010110	−150	0 1101010

Note that the eight-bit result that should have been positive has a negative sign bit (i.e., the eighth bit) and the eight-bit result that should have been negative has a positive sign bit. If, however, **the carry out of the sign bit position is taken as the sign bit of the result, then the nine-bit answer so obtained will be correct**. But since the answer cannot be accommodated within eight bits, we say that an overflow has occurred.
An overflow condition can be detected by observing the carry into the sign bit position and the carry out of the sign bit position. If these two carries are not equal, an overflow has occurred. If the two carries are **applied to an exclusive-OR gate**, an overflow is detected when the output of the gate is equal to 1.

DECIMAL ADDER:

A decimal adder requires a minimum of **nine inputs and five outputs**, since four bits are required to code each decimal digit and the circuit must have an input and output carry. Here we examine a **decimal adder for the BCD code**.

BCD Adder:

Note: Since each input digit does not exceed 9, the output sum cannot be greater than 9 + 9 + 1 = 19, the 1 in the sum being an input carry. When the binary sum is equal to or less than 1001, the corresponding BCD number is identical, and therefore no conversion is needed.
Suppose **we apply two BCD digits to a four-bit binary adder**, when the binary sum is greater than 1001, we obtain an invalid BCD representation. The **addition of binary 6 (0110)** to the binary sum converts it to the correct BCD representation and also **produces an output carry as required**.
When the output carry is equal to 0, nothing is added to the binary sum. When it is equal to 1, binary 0110 is added to the binary sum through the bottom four-bit adder. The output carry generated from the bottom adder can be ignored.

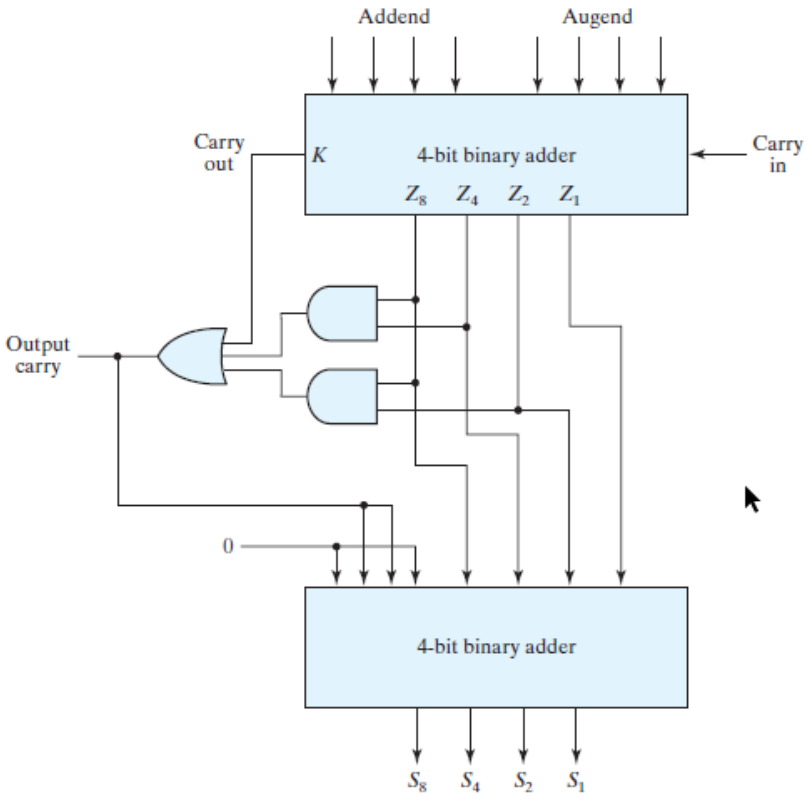


FIGURE 4.14
Block diagram of a BCD adder

Note: A decimal parallel adder that adds n decimal digits needs n BCD adder stages. The output carry from one stage must be connected to the input carry of the next higher order stage.

MAGNITUDE COMPARATOR:

A magnitude comparator is a combinational circuit that compares two numbers A and B and determines their relative magnitudes. The outcome of the comparison is specified by three binary variables that indicate whether A > B, A = B, or A < B.

$$x_i = A_i B_i + A_i' B_i' \quad \text{for } i = 0, 1, 2, 3$$

$$(A > B) = A_3 B_3' + x_3 A_2 B_2' + x_3 x_2 A_1 B_1' + x_3 x_2 x_1 A_0 B_0'$$
$$(A < B) = A_3' B_3 + x_3 A_2' B_2 + x_3 x_2 A_1' B_1' + x_3 x_2 x_1 A' n_0 B_0'$$

Table from notes:

Decoder:

A binary code of n bits is capable of representing up to 2^n distinct elements of coded Information A decoder is a combinational circuit that converts binary information from n input lines to a maximum of 2^n unique output lines. If the n-bit coded information has unused combinations, the decoder may have fewer than 2^n outputs..

The decoders presented here are called n -to- m -line decoders, where m <= 2^n

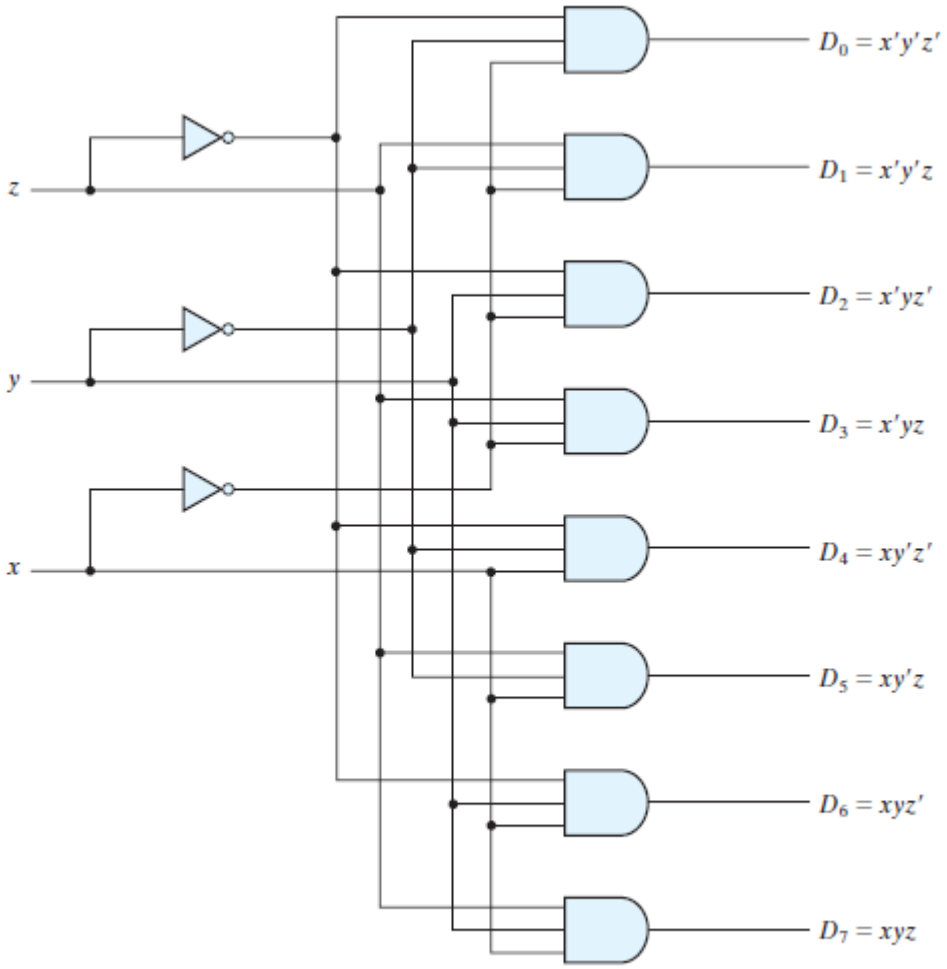


FIGURE 4.18
Three-to-eight-line decoder

Table 4.6
Truth Table of a Three-to-Eight-Line Decoder

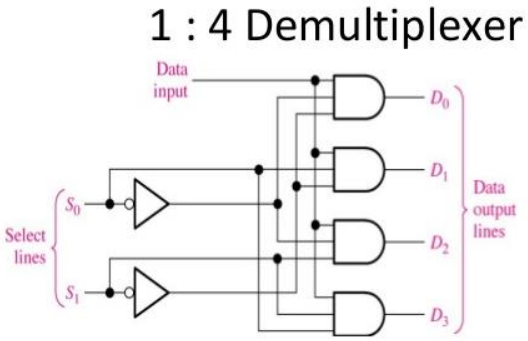
Inputs			Outputs							
x	y	z	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

For each possible input combination, there are seven outputs that are equal to 0 and only one that is equal to 1. **Observe that the output variables are mutually exclusive only one output can be equal to 1 at any point of time**. A particular application of this decoder is binary-to-octal conversion. The input variables represent a binary number, and the outputs represent the eight digits of a number in the octal number system.

Note: Any combinational circuit with n inputs and m outputs can be Implemented with n-to-2^n line decoder and m OR gates.

Demultiplexer:

A decoder with enable input can function as a **demultiplexer** circuit that receives information from a single line and directs it to **one of 2^n** possible output lines.
The selection of a specific output is controlled by the bit combination of n selection lines.



S_0	S_1	D_0	D_1	D_2	D_3
0	0	D	0	0	0
0	1	0	D	0	0
1	0	0	0	D	0
1	1	0	0	0	D

The single input variable E has a path to all four outputs, but the input information is directed to only one of the output lines, as specified by the binary combination of the two selection lines S0 and S1.

Decoders with enable inputs can be connected together to form a larger decoder circuit.
two 3-to-8-line decoders with enable inputs connected to form a 4-to-16-line decoder. When w = 0, the top decoder is enabled and the other is disabled

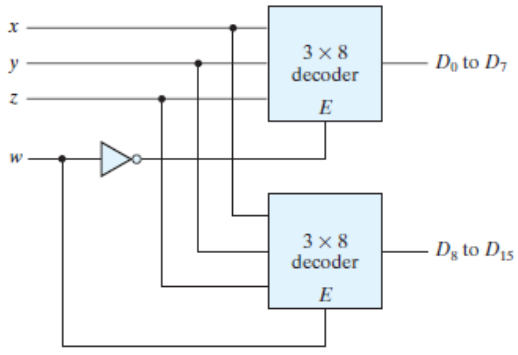


FIGURE 4.20
4 × 16 decoder constructed with two 3 × 8 decoders

Combinational Logic Implementation

A decoder provides the 2^n minterms of n input variables. **a decoder and OR gates requires that the Boolean function for the circuit be expressed as a sum of minterms**.

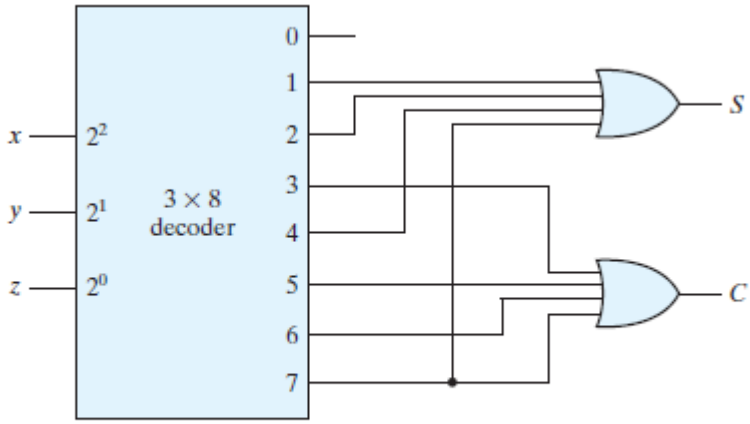


FIGURE 4.21
Implementation of a full adder with a decoder

A function with a long list of **minterms requires an OR gate with a large number of inputs**. A function having a list of k minterms can be expressed in its complemented form F' with 2^n - k minterms. If the number of minterms in the function is greater than 2^n/2, then F' can be expressed with fewer **minterms In such a case, it is advantageous to use a NOR gate to sum the minterms of F'**.

Encoders:

An encoder is a digital circuit that performs the inverse operation of a decoder. An encoder has **2^n (or fewer) input lines and n output lines**.
The output lines, generate the binary code corresponding to the input value. An example of an encoder is the **octal-to-binary encoder** It has eight inputs (**one for each of the octal digits**) and three outputs that generate the corresponding binary number. **It is assumed that only one input has a value of 1 at any given time**.

Table 4.7
Truth Table of an Octal-to-Binary Encoder

Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

z= D1 + D3 + D5 + D7
y = D2 + D3 + D6 + D7
x = D4 + D5 + D6 + D7
The encoder can be implemented with three OR gates.
Note: If two inputs are active simultaneously, the output produces an undefined combination. For example, if D3 and D6 are 1 simultaneously, the output of the encoder will be 111 because all three outputs are equal to 1. output 111 doesn't represent either binary 3 or binary 6. **Another ambiguity in** the octal-to-binary encoder is that **an output with all 0's is generated when all the inputs are 0**; but this output is the same as when D0 is equal to 1.

The discrepancy can be resolved by providing one more output to indicate whether at least one input is equal to 1.

Priority Encoder

https://www.youtube.com/watch?time_continue=385&v=kEj-m3YuGa4

if two or more inputs are equal to 1 at the same time, the input having the highest priority(higher subscript) will take precedence.

Table 4.8
Truth Table of a Priority Encoder

Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

The truth table of a four-input priority encoder is given In addition to the two outputs x and y , the circuit has a third output designated by V ; **this is a valid bit indicator that is set to 1 when one or more inputs are equal to 1.** If all inputs are 0, there is no valid input and V is equal to 0. The other two outputs are not inspected when V equals 0 and are specified as don't-care conditions.

Note: the truth table uses an X to represent either 1 or 0. For example, X 100 represents the two minterms 0100 and 1100.

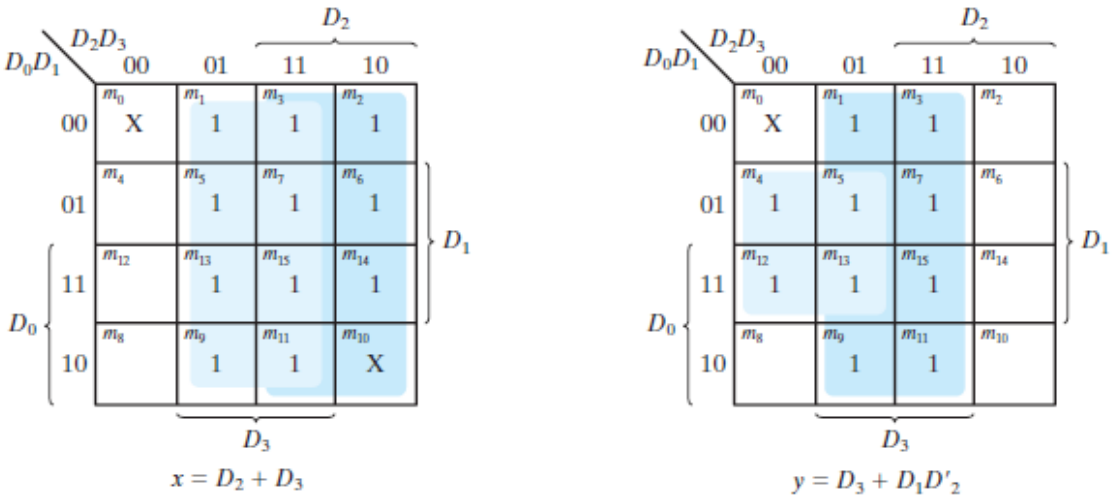


FIGURE 4.22
Maps for a priority encoder

Although the table has only five rows, when each X in a row is replaced first by 0 and then by 1, **we obtain all 16 possible input combinations.** For example, **the fourth row in the table, with inputs XX10, represents the four minterms 0010, 0110, 1010, and 1110**.

$$x = D_2 + D_3$$
$$y = D_3 + D_1 D_2'$$
$$V = D_0 + D_1 + D_2 + D_3$$

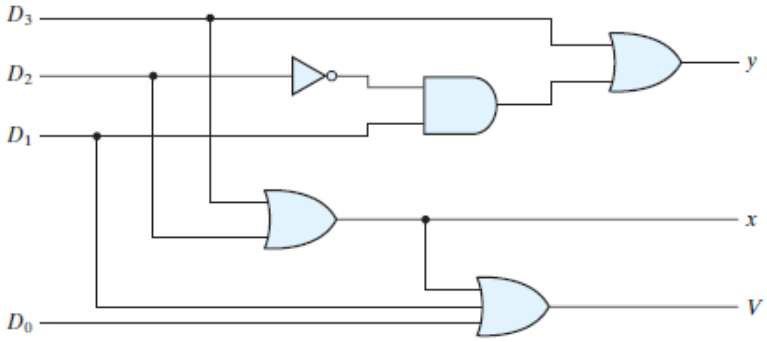
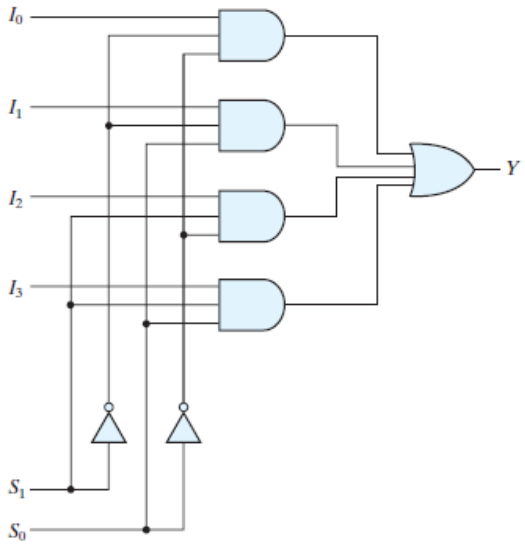


FIGURE 4.23
Four-input priority encoder

MULTIPLEXERS

A multiplexer is a combinational circuit that selects binary information from one of many input lines and directs it to a single output line. The selection of a particular input line is controlled by a set of selection lines. Normally, there **are 2^n input lines and n selection lines** whose bit combinations determine which input is selected. A multiplexer is also called a **data selector**.
In general, a 2^n -to-1-line multiplexer is constructed from an n -to-2^n decoder by adding 2^n input lines to it, **one to each AND gate.** The outputs of the AND gates are applied to **a single OR** gate.

Note: multiplexers may have an enable input to control the operation of the unit. When the enable input is in the inactive state, the outputs are disabled, and when it is in the active state, the circuit functions as a normal multiplexer.



(a) Logic diagram

S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

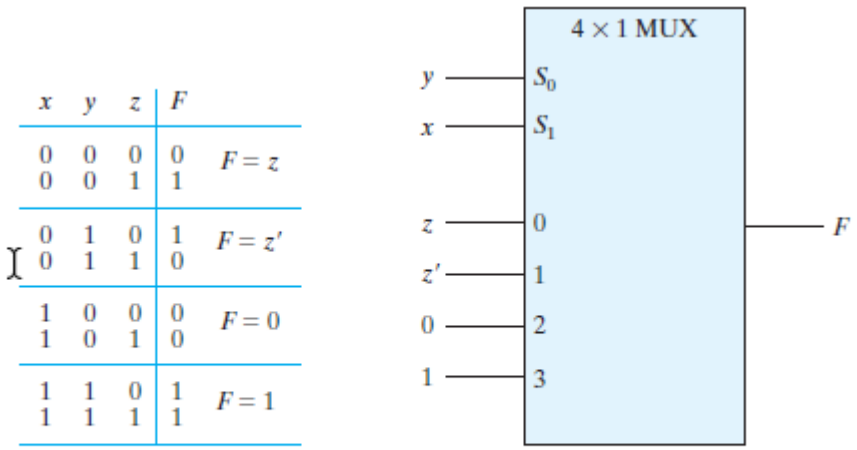
(b) Function table

FIGURE 4.25
Four-to-one-line multiplexer

Note: implementing a B**oolean function of n variables with a multiplexer that has n selection inputs and 2^n data inputs(total 2^n minterms pssible with n variable Boolean function), one for each minterm**.

But there is one **more efficient way** of doing it, we **can have n-1 selection lines and and 2^(n-1) data inputs**. The first n - 1 variables of the function are connected to the selection inputs of the multiplexer. The remaining **single variable of the function is used** for the data inputs.

F (x, y, z) = Σ (1, 2, 6, 7)



x	y	z	F	
0	0	0	0	$F = z$
0	0	1	1	
0	1	0	1	$F = z'$
0	1	1	0	
1	0	0	0	$F = 0$
1	0	1	0	
1	1	0	1	$F = 1$
1	1	1	1	

(a) Truth table

(b) Multiplexer implementation

FIGURE 4.27
Implementing a Boolean function with a multiplexer

****

Synchronous Sequential Logic

SEQUENTIAL CIRCUITS:

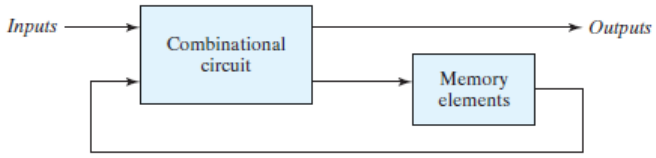


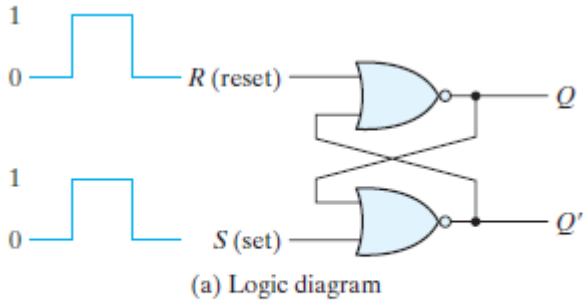
FIGURE 5.1
Block diagram of sequential circuit

It consists of a combinational circuit to which storage elements are connected to form a feedback path. The storage elements are devices capable of storing binary information. The binary information stored in these elements at any given time defines the state of the sequential circuit at that time.
A sequential circuit is specified by a time sequence of inputs, outputs, and internal states.
Note: Rest will be studied from my notes.

The outputs are formed by a combinational logic function of the inputs to the circuit or the values stored in the flip-flops (or both).

SR Latch

The SR latch is a circuit with two cross-coupled NOR gates or two cross-coupled NAND gates, and two inputs labelled S for set and R for reset. The SR latch constructed with two cross-coupled NOR gates is shown as below:



(a) Logic diagram

FIGURE 5.3
SR latch with NOR gates

When output Q = 1 and Q' = 0, the latch is said to be in the set state. When Q = 0 and Q' = 1, it is In the reset state. Outputs Q and Q' are normally the complement of each other. However, **when both inputs are equal to 1 at the same time**, a condition in which **both outputs are equal to 0** (rather than be mutually complementary) occurs.
If both inputs are then switched to 0 simultaneously, the device will enter an unpredictable or undefined state or a metastable state.
Under normal conditions, both inputs of the latch remain at 0 unless the state has to be Changed.

****

Registers and Counters

A clocked sequential circuit consists of a group of flip-flops and combinational gates. A circuit with flip-flops is considered a sequential circuit even in the absence of combinational gates. Circuits that include flip-flops are usually classified by the function they perform rather than by the name of the sequential circuit. Two such circuits are registers and counters.
A synchronous sequential circuit employs signals that affect the storage elements **at only discrete instants of time.** Synchronization is achieved by a timing device called a clock generator, which provides a clock signal having the form of a periodic train of clock pulses. **The storage elements (memory) used in clocked sequential circuits are called flipflops**. A flip-flop is a binary storage device capable of storing one bit of information. In a stable state, the output of a flip-flop is either 0 or 1. A sequential circuit may use many flip-flops to store as many bits as necessary.

Registers:

A register is a group of flip-flops, each one of which shares a common clock and is capable of storing one bit of information. An n -bit register consists of a group of n flip-flops capable of storing n bits of binary information. In addition to the flip-flops, a register may have combinational gates that perform certain data-processing tasks.

A **counter is essentially a register** that goes through a predetermined sequence of binary states. The gates in the counter are connected in such a way as to produce the prescribed sequence of states.

SHIFT REGISTERS:

A register capable of shifting the binary information held in each cell to its neighboring cell, in a selected direction, is called **a shift register**. The logical configuration of a shift register consists of a chain of flip-flops in cascade, with the output of one flip-flop connected to the input of the next flip-flop. All flip-flops receive common clock pulses, which activate the shift of data from one stage to the next.

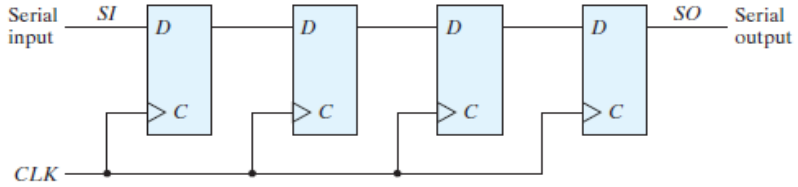


FIGURE 6.3
Four-bit shift register

RIPPLE COUNTERS

A register that goes through a prescribed sequence of states upon the application of input pulses is called a **counter**. The input pulses may be clock pulses, or they **may originate from some external** source and may occur at a fixed interval of time or at random.

The sequence of states may follow the binary number sequence or any other sequence of states. A counter that follows the binary number sequence is called a binary counter. An n -bit binary counter consists of n flip-flops and can count in binary from 0 through 2^n - 1.

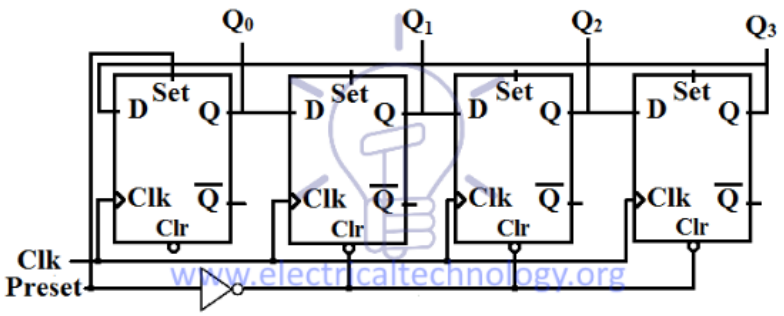
Counters are available in two categories: **ripple counters and synchronous counters**. In a ripple counter, a **flip-flop output transition** serves as a source for triggering other flip-flops.

Note: A ripple counter is an asynchronous sequential circuit.

Binary Ripple Counter

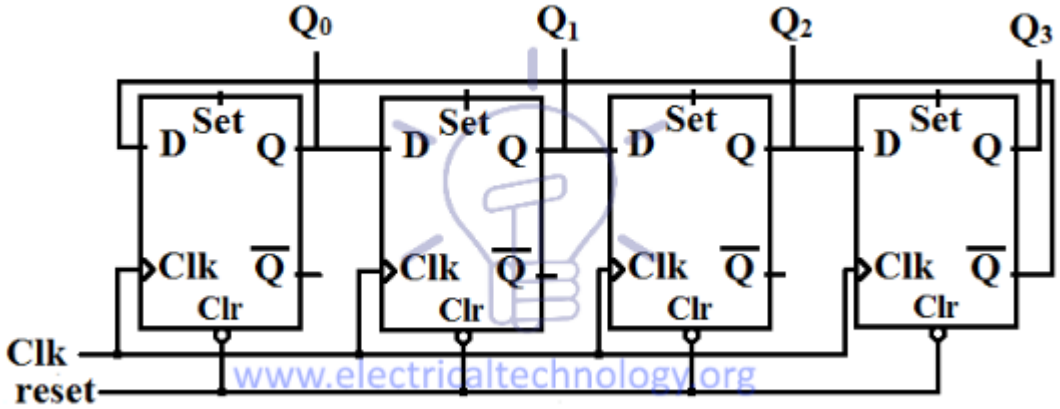
A binary ripple counter consists of **a series connection of complementing flip-flops**, with the output of each flip-flop connected to the C input of the next higher order flip-flop. **The flip-flop holding the least significant bit receives the incoming count pulses**. A complementing flip-flop can be obtained from a JK flip-flop with the J and K inputs tied together or from a T flip-flop. A third possibility is to use a D flip-flop with the complement output connected to the D input. The counter is constructed with complementing flip-flops of the T type in part (a) and D type in part (b). The output of each flip-flop is connected to the C input of the next flip-flop in sequence.

Ring Counter: (mod n) counter
In case of ring counter we need to set the initial state 1000 through **preset** input.



Note that first flip is set and others are reset (cleared). States 1000 -> 0100 -> 0010 -> 0001 -> 1000 there are 4 distinct state.

Johnson's Ring Counter:



States 0000->1000->1100->1110->1111->0111->0011->0001->0000 hence 2n states and mod 8 counter.

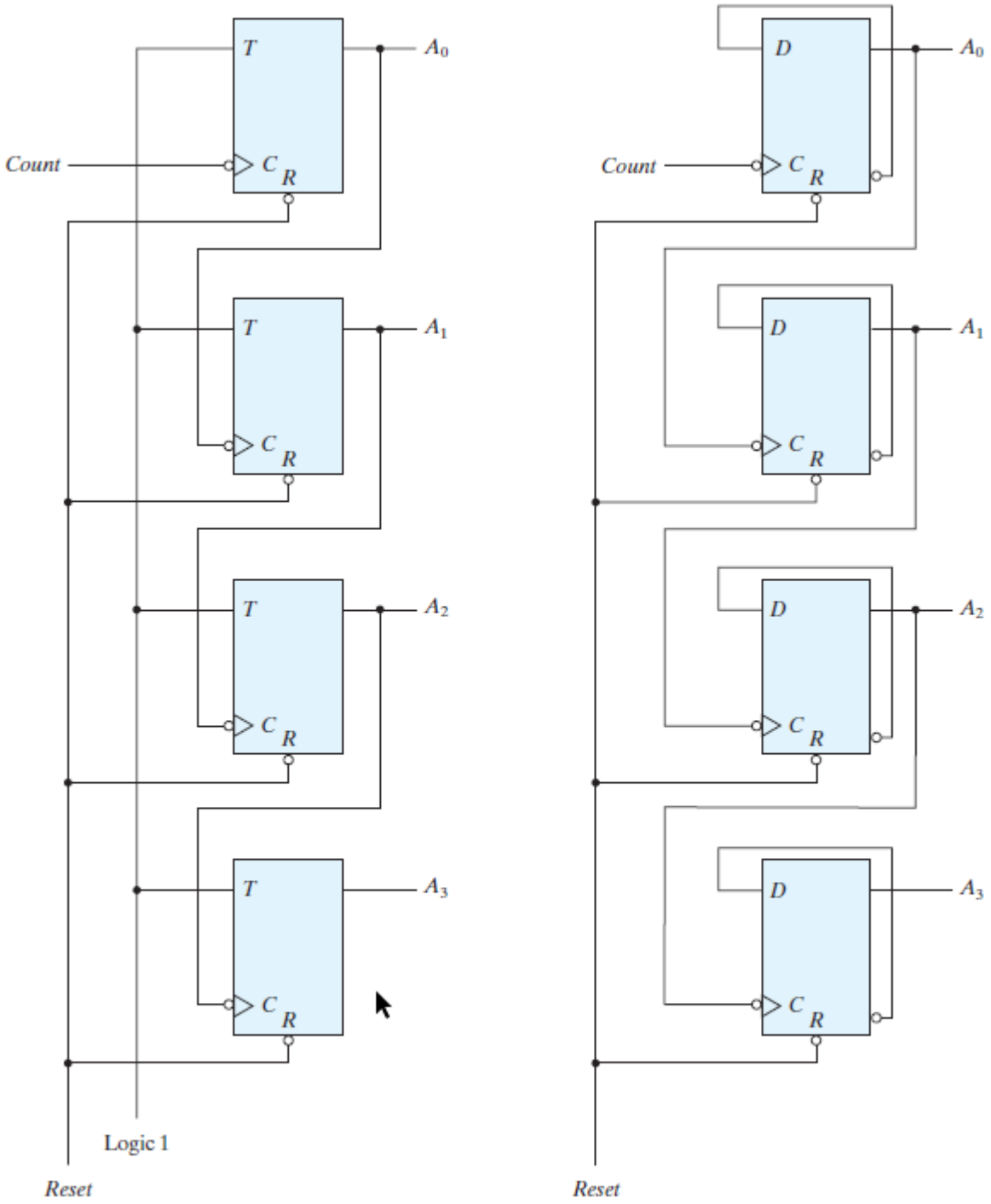


FIGURE 6.8

BCD Ripple Counter

A decimal counter follows a sequence of 10 states and returns to 0 after the count of 9. Such a counter must have at least four flip-flops to represent each decimal digit, since a decimal digit is represented by a binary code with at least four bits.

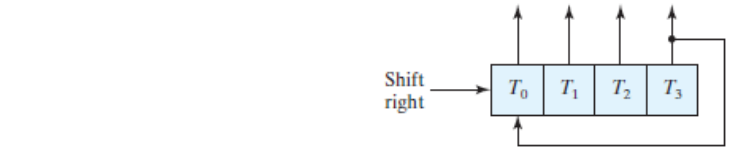
The BCD counter is a decade counter, since it counts from 0 to 9. To count in decimal from 0 to 99, we need a two-decade counter. To count from 0 to 999, we need a three-decade counter. Multiple decade counters can be constructed by connecting BCD counters in cascade, one for each decade.

SYNCHRONOUS COUNTERS:

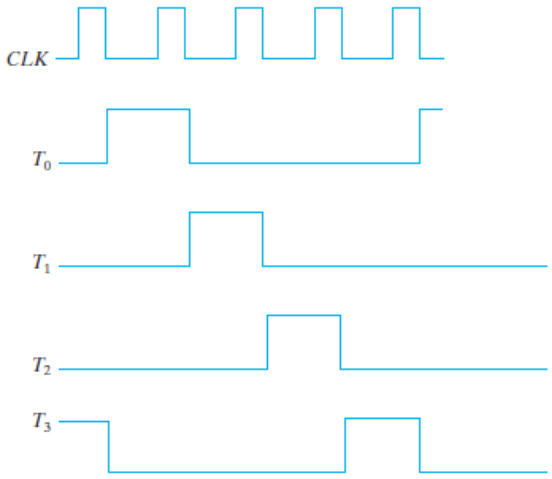
Synchronous counters are different from ripple counters in that clock pulses are applied to the inputs of all flip-flops. A common clock triggers all flip-flops simultaneously, rather than one at a time in succession as in a ripple counter.

Ring Counter

A ring counter is a circular shift register with only one flip-flop being set at any particular time; all others are cleared. The single bit is shifted from one flip-flop to the next to produce the sequence of timing signals.

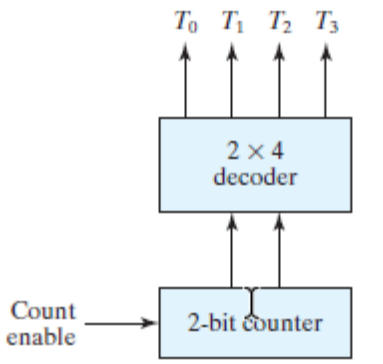


(a) Ring-counter (initial value = 1000)



(b) Sequence of four timing signals

The timing signals can be generated by a two-bit counter that goes through four distinct states. The decoder shown below decodes the four states of the counter and generates the required sequence of timing signals.

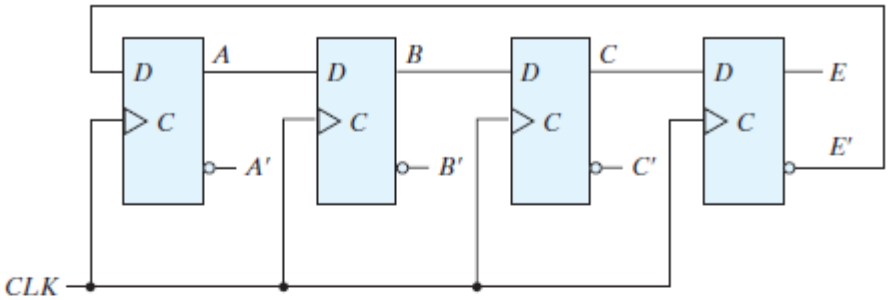


(c) Counter and decoder

To generate **2^n timing signals, we need either a shift register with 2^n flip-flops** or an n -bit binary counter together with an n -to-2^n -line decoder. **For example**, 16 timing signals can be generated with a 16-bit shift register connected as a ring counter or with a **4-bit binary counter and a 4-to-16-line decoder**.

Johnson Counter

A **k -bit ring counter** circulates a single bit among the flip-flops to provide **k distinguishable states**. The number of states can be doubled if the shift register is connected as a switch-tail ring counter. A switch-tail ring counter is a circular shift register with the complemented output of the last flip-flop connected to the input of the first flip-flop.



(a) Four-stage switch-tail ring counter

Sequence number	Flip-flop outputs A	B	C	E	AND gate required for output
1	0	0	0	0	$A'E'$
2	1	0	0	0	AB'
3	1	1	0	0	BC'
4	1	1	1	0	CE'
5	1	1	1	1	AE
6	0	1	1	1	$A'B$
7	0	0	1	1	$B'C$
8	0	0	0	1	$C'E$

(b) Count sequence and required decoding

FIGURE 6.18
Construction of a Johnson counter

Points to be Rememberd:

1. **Implicant:**

   $f(x,y,z) = x' + y'x + xz$

   An implicant of a function is a product term that is included in the function.

   so x', y'x and xz ,all are implicants of given function.

2. any binary function can be implemented using half Adders only but we need support of 1.
3. In Carry Look Ahead Adder, if we implement Carry Look Ahead Generator using 2-Input AND and OR gates, it will require O(Logn) while in Ripple Carry it needs O(n) time.

EXCESS – 3 to BCD Convertor:

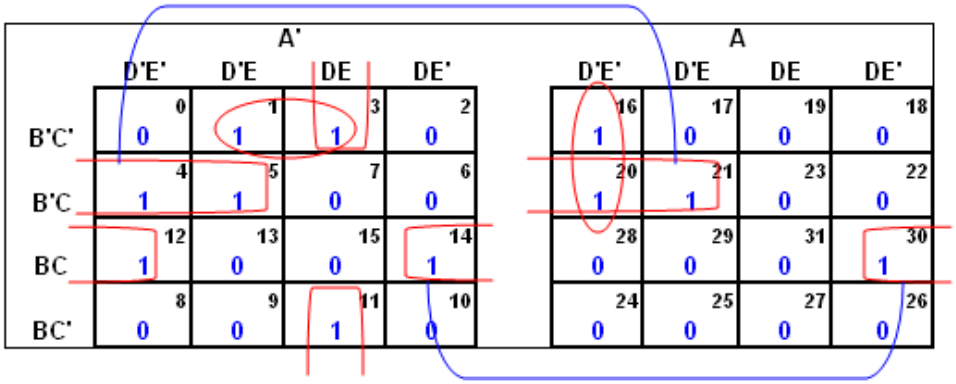
We know that, excess-3 code begins with the binary 0011(decimal 3) and it will continue up to binary 1100(decimal 12) where we get the output binary 1001(decimal 9) for input binary 1100(decimal 12). Since we don't use 0,1,2,13,14 and 15 hence we can use them as don't care values. Following is the truth table:

Problem 1: Design an Excess-3 to BCD code converter. 4-bit input words other than those listed in the following table are invalid. Design the system assuming that invalid Excess-3 code words will not be applied to the input. Use invertors and 2-input logic gates.

Decimal Digit	Input Excess-3 Code Word				Output BCD Code Word			
	A	B	C	D	W	X	Y	Z
×	0	0	0	0	×	×	×	×
×	0	0	0	1	×	×	×	×
×	0	0	1	0	×	×	×	×
0	0	0	1	1	0	0	0	0
1	0	1	0	0	0	0	0	1
2	0	1	0	1	0	0	1	0
3	0	1	1	0	0	0	1	1
4	0	1	1	1	0	1	0	0
5	1	0	0	0	0	1	0	1
6	1	0	0	1	0	1	1	0
7	1	0	1	0	0	1	1	1
8	1	0	1	1	1	0	0	0
9	1	1	0	0	1	0	0	1
×	1	1	0	1	×	×	×	×
×	1	1	1	0	×	×	×	×
×	1	1	1	1	×	×	×	×

5 – Variables K – Map

F = Σ (1, 3, 4, 5, 11, 12, 14, 16, 20, 21, 30)



1. (4, 5, 20, 21) – B'CD' (Since A & E are the changing variables, it is eliminated)
2. (12, 14) – A'BCE' (Since D is the changing variable, it is eliminated)
3. (14, 30) – BCDE' (Since A is the changing variable, it is eliminated)
4. (3, 11) – A'C'DE (Since B is the changing variable, it is eliminated)
5. (16, 20) – AB'D'E' (Since C is the changing variable, it is eliminated)
6. (1, 3) – A'B'C'E (Since D is the changing variable, it is eliminated)

Thus, **F = B'CD' + A'BCE' + BCDE' + A'C'DE + AB'D'E' + A'B'C'E**

Self Dual (GTAEBOOK)

If we apply dual to a function and if again get the same function, such functions are called **dual function**. (f=fd)

Self dual function

x	y	z	f_1	f2
0	0	0	1	1
0	0	1	0	1
0	1	0	1	1
0	1	1	1	1
1	0	0	0	0
1	0	1	1	0
1	1	0	1	0
1	1	1	1	0

f1 = x'y'z' + x'yz' + x'yz + xy'z + xyz' + zyz
Let's simplify it first before we apply dual operation to it.

It's simplified to x'z' + xy + xz + yz

f1d = (x' + z')(x+y)(x+z)(y+z) = x'yz + xyz' = 3,6 this function is not self-dual as we're not getting that function back when we applied dual operation.

How to calculate dual of the function

f'(x,y,z) = fd(x',y'z') //complement of a function
fd(x',y',z') = f'(x,y,z)
now replace x' with x, y' with y, and z' with z. it will be

fd(x,y,z) = f'(x',y',z')

If we want to calculate dual of a function, then write the complemented value in the function and take the complement of the whole function.

Calculate the dual of above f1 function.
f(x,y,z) = x'y'z' + x'yz' + x'yz + xy'z + xyz' + zyz = (0, 2, 3, 5, 6, 7)

First we calculate f(x',y',z') = xyz + xy'z + xy'z' + x'yz' + x'y'z + x'y'z' = (7, 5, 4, 2, 1, 0)

Note – if we look at the function f(x',y',z') we are subtracting each minterm value from 7 (2^n -1).

f(x',y',z') = xyz + xy'z + xy'z' + x'yz' + x'y'z + x'y'z' = (7, 5, 4, 2, 1, 0)
now we have to calculate f'(x',y,'z') = that will be (3, 6) this is same as we calculated above.

(0, 1, 2, 3, 4, 5, 6, 7) – (7, 5, 4, 2, 1, 0) = (3, 6) we didn't get back the actual function hence it's not self dual.

Calculate dual of f2.
f2(x,y,z) = (0, 1, 2, 3)
f2(x',y',z') = (7,6,5,4) // each minterm is subtracted from 7
calculate f'(x',y',z') = (0,1,2,3,4,5,6,7) – (7,6,5,4) = (0,1,2,3) = f2(x,y,z) we got the same function back after dual operation, hence it's a **dual function**.

Example f(x,y,z) = Sigma(1,2,4,7) is it a self-dual function?

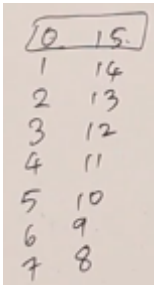
f(x',y',z') = sigma(6,5,3,0)

f'(x',y,'z') = sigma(1, 2, 4, 7) we got the original minterms back, it's aself dual function.

Note –
1. If a function is self-dual, then there should not be any 2 terms in our function whose sum is equal to 7. As, we shouldn't have the common minterms in f(x,y,z) and f(x',y',z').
2. If we have exactly half of the total minterms in our function f(x,y,z) then there is a possibility that it may be a self-dual function.

Example f(A,B,C,D) = Sigma(0,1,3,6,7,10,11,14)
i. is it self—dual function?
ii. The number of self-dual functions on 4 variables.
Solution
This is not a self-dual function because we have two terms 1 and 14 and their sum is 15. 1 and 14 will be in f(A',B',C',D') so when we calculate complement of f(A',B',C',D') we will not get them back.

We will write minterms in group



These terms shouldn't be together, for example if 0 and 15 are together present in our function f, that function is **not self-dual**.
Out of these two terms in a group, exactly one should be present. Note that there both terms can't be absent, because we need exactly half of all the minterms.

ii. number of self-dual functions, each member can be selected two ways from each group. There are 2^8 ways to select dual functions.
If there are n variables then there will be 2^(n-1) groups. So selecting one from each group, and there are 2^(n-1) groups hence
2x2x2x2x….. (2^(n-1)) times so this is **2^(2^(n-1))**

Find those functions whose dual is same as its complement

Suppose there is a function f(x,y,z) if after subtracting each minterm from (2^n-1), if we get the original minterms back then complement and self-dual of a function will be same.

It means if f(x,y,z) = f(x',y',z') then complement and self-dual will be same.

Example f(x,y,z) = sigma(2,5,6,1) =
F(x,y,z) = sigma(2,5,6,1)
F(x',y'z') = sigma(5,2,1,6).

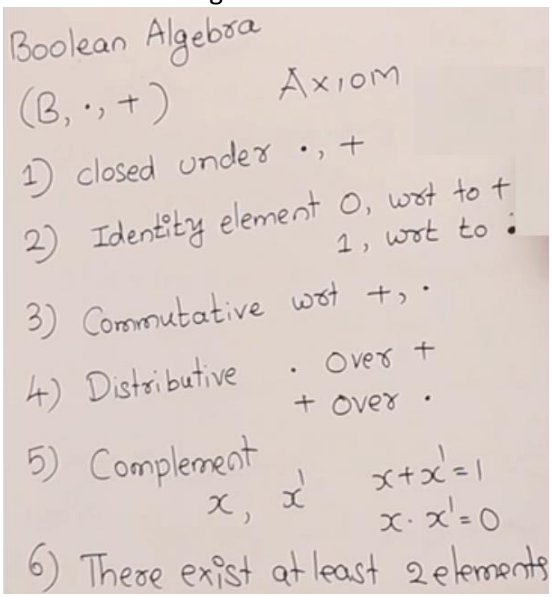
We have 2^(n-1) total groups and any subset of these pairs will give us dual and complement same.
For example f(x,y,z) = Sigma(5,2) or f(x,y,z) = (5,2,1,6) both will have dual and complement same.

There are total possible such functions. **2^(2^(n-1)) -1. As empty subset is not counted.**

GATEBOOK lectures

1. **Introduction to Boolean Algebra**
   Boolean algebra is a triplate in which B is a set and . and + are operators.

   Boolean algebra should follow the following axioms:

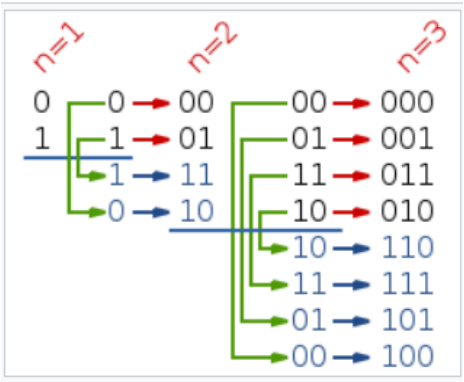


   Axiom is a statement without a proof. 2 elements that's why it's called Boolean algebra.

Binary Down Counter

A binary counter with a reverse count is called a *binary countdown counter*. In a countdown counter, the binary count is decremented by 1 with every input count pulse. The count of a four-bit countdown counter starts from binary 15 and continues to binary counts 14, 13, 12, . . . , 0 and then back to 15. A list of the count sequence of a binary countdown counter shows that the least significant bit is complemented with every count pulse. Any other bit in the sequence is complemented if its previous least significant bit

Gray Counter:



The first few steps of the reflect-and-prefix method.

Take previous numbers and do concatenation with the reverse of the previous number as shown above.