DIGITAL LOGIC AND COMPUTER ORGANIZATION

DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING

LENDI INSTITUTE OF ENGINEERING AND TECHNOLOGY

(An Autonomous Institute, Approved by AICTE & Permanently Affiliated to JNTU-GV, Vizianagaram)

(Accredited By NAAC with A Grade and Accredited by NBA)

Jonnada (Village), Denkada (Mandal), Vizianagaram District – 535 005

Phone No. 08922-241111, 241112

E-Mail: lendi_2008@yahoo.com website: www.lendi.org

UNIT – I

Digital Logic Circuits-I: 2's Complement and 1's Complement, Subtraction of Unsigned Numbers, Signed Binary Numbers, Minimization of Logic expressions using K-Map Simplification up to 4 variable.

Combinational circuits-I: Half Subtractor, Full Subtractor, 4-bit Adder & Subtractor, Comparators.

INDEX

UNIT – I
DIGITAL LOGIC CIRCUITS-I

COMPLEMENTS

Complements are used in digital computers for simplifying the subtraction operation and for logical manipulation. There are two types of complements for each base r system: the r's complement and the (r - l)'s complement. When the value of the base r is substituted in the name, the two types are referred to as the 2's and 1's complement for binary numbers and the 10's and 9's complement for decimal numbers.

Diminished Radix complement or **(r -1)'s Complement**

9'S COMPLEMENT

Given a number N in base r having n digits, the (r-1)'s complement of N is defined as **$(r^n - 1) - N$**. For **decimal numbers r = 10 and r-1= 9**, the 9's complement of N is **$(10^n - 1) - N$**. Now, 10^n represents a number that consists of a single 1 followed by n 0's. $10^n - 1$ is a number represented by n 9's. For example, with n = 4 we have $10^4 = 1000$ and $10^4 - 1 = 9999$. It follows that the 9's complement of a decimal number is obtained by subtracting each digit from 9.

For example:

The 9's complement of 546700 is 999999 - 546700 = 453299 and
The 9's complement of 012389 is 999999 - 012389 = 987610.

1'S COMPLEMENT

For **binary numbers, r = 2 and r -1 = 1,** so the 1's complement of N is **$(2^n - 1) - N$.** Again, 2^n is represented by a binary number that consists of a 1 followed by n 0's. $2^n - 1$ is a binary number represented by n 1's. For example, with n = 4, we have $2^4 = (10000)_2$, and $2^4 – 1= (1111)_2$. Thus the 1's complement of a binary number is obtained by subtracting each digit from 1. However, the subtraction of a binary digit from 1, we can have either 1 - 0 = 1 or 1 -1 = 0, which causes the bit to change from 0 to 1 or from 1 to 0. Therefore, the 1's complement of a binary number is formed by changing 1's into 0's and 0's into 1's.

For example:

The 1's complement of 1011001 is 0100110 and
The 1's complement of 0001111 is 1110000.

The (r - 1)'s complement of octal or hexadecimal numbers are obtained by subtracting each digit from 7 or F (decimal 15) respectively.

Radix complement or **r's Complement**

10'S COMPLEMENT

The r's complement of an n-digit number N in base r is defined as **$r^n - N$ for $N \neq 0$** and **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 **$r^n - N = [(r^n - 1) - N] + 1$**. Thus the **10's complement of the decimal 2389 is 7610 + 1 = 7611** and is obtained by adding 1 to the 9's complement value. The **2's complement of binary 101100 is 010011 + 1 = 010100** and is obtained by adding 1 to the 1's complement value.

Since 10^n is a number represented by a 1 followed by n 0's, then 10^n - N, which is the 10's complement of N, can be formed also be leaving all least significant 0's unchanged, subtracting the first nonzero least significant digit from 10, and then subtracting all higher significant digits from 9.

For example:

The 10's complement of 012398 is 987602 and
The 10's complement of 246700 is 753300

The 10's complement of first number is obtained by subtracting 8 from 10 in the least significant position and subtracting all other digits from 9. The 10's complement of second number obtained by leaving the two zeros unchanged, subtracting 7 from 10, and subtracting the other three digits from 9.

2'S COMPLEMENT

The 2's complement can be formed by leaving all least significant 0's and the first 1 unchanged, and then replacing 1's by 0's and 0's by 1's in all other higher significant bits.

For example:

The 2's complement of 1101100 is 0010100 and
The 2's complement of 0110111 is 1001001.

The 2's complement of first number is obtained by leaving the two low-order 0's and the first 1 unchanged, and then replacing 1's by 0's and 0's by 1's in the other four most significant bits. The 2's complement of second number obtained by leaving the least significant 1 unchanged and complementing all other digits.

In the definitions above it was assumed that the numbers do not have a radix point. If the original number N contains a radix point, it should be removed temporarily to form the r's or (r - 1)'s complement. The radix point is then restored to the complemented number in the same relative position. It is also worth mentioning that the complement of the complement restores the number to its original value. The r's complement of N is r^n - N. The complement of the complement is $r^n - (r^n - N) = N$ giving back the original number.

SUBTRACTION OF UNSIGNED NUMBERS or
Subtraction with complements

The direct method of subtraction uses the borrow concept. In this method we borrow a 1 from a higher significant position when the **minuend** digit is smaller than the corresponding **subtrahend** digit. This seems to be easiest when we perform subtraction with paper and pencil. When subtraction is implemented with digital hardware, this method is found to be less efficient than the method that uses complements.

The subtraction of **two n-digit unsigned numbers** M - N ($N \neq 0$) in base r can be done as follows:

1. Add the minuend M to the r's complement of the subtrahend N. This performs $M + (r^n - N) = M - N + r^n$.

2. If $M \geq N$, the sum will produce an end carry r^n which is discarded, and what is left is the result M - N.

3. If $M < N$, the sum does not produce an end carry and is equal to r^n - (N - M), which is the r's complement of (N - M). To obtain the answer in a familiar form, take the r's complement of the sum and place a negative sign in front.

Example 1: Using 10's complement, Subtract 72532-3250
Solution: In general subtraction, 72532 - 3250 = 69282.
The 10's complement of 03250 is 96750.
Therefore:

M =	72532
10's complement of N =	+96750
Sum =	169282
Discard end carry 10^5 = -	100000
Answer =	**69282**

Note that M has 5 digits and N has only 4 digits. Both the numbers must have equal number of digits; so we can write N as 03250. Taking the 10's complement of N produces 9 in the most significant position. The occurrence of the end carry signifies that $M \geq N$ and the result is positive.

Example 2: Using 10's complement, Subtract 3250-72532
Solution: The subtraction 3250 – 72532 produces negative 69282. Using the procedure with complements, we have

M =	03250
10's complement of N =	+27468
Sum =	30718

There is no end carry
Answer is: - (10's complement of 30718) = **-69282**
Note that since 3250 < 72532, the result is negative. Since we are dealing with unsigned numbers, there is really no way to get an unsigned result for the second example. When working with paper and pencil, we recognize that the answer must be changed to a signed negative number. When **subtracting with complements**, the **negative answer is recognized by the absence of the end carry and the complemented result.**

Example 3: Given two binary numbers X = 1010100 and Y = 1000011, we perform the subtraction (a) X – Y and (b) Y - X using 2's complements:
Solution:

(a) For X-Y:

X =	1010100
2's complement of Y =	+0111101
Sum =	10010001
Discard end carry 2^7 = -	10000000
Answer: X - Y =	**0010001**

(b) For Y-X:

Y =	1000011
2's complement of X =	+0101100
Sum =	1101111

There is no end carry
Answer: Y-X = - (2's complement of 1101111) = **-0010001**

Subtraction of unsigned numbers can be done by means of (r-1)'s complement. Remember that the (r-1)'s complement is one less than r's complement. Because of this, the result of adding minuend to the complement of subtrahend produces a sum that is 1 less than the correct difference when the end carry occurs. **Removing the end carry and adding 1 to the sum is referred to as end-around carry.**

Example 4: Given two binary numbers X = 1010100 and Y = 1000011, we perform the subtraction (a) X – Y and (b) Y - X using 1's complement:
Solution:

(a) For X-Y:

X =	1010100
1's complement of Y =	+ 0111100
Sum =	10010000
End-around carry =	+1
Answer: X - Y =	**0010001**

(b) For Y-X:

Y =	1000011
2's complement of X =	+ 0101011
Sum =	1101110

There is no end carry
Answer: Y-X = - (1's complement of 1101111) = **-0010001**

Note that the negative result is obtained by taking the 1's complement of the sum since this is the type of complement used. The procedure with end-around carry is also applicable for subtracting unsigned decimal numbers with 9's complement.

SIGNED BINARY NUMBERS

Positive integers, including zero, can be represented as unsigned numbers. However, to represent negative integers, we need a notation for negative values. **In ordinary arithmetic**, a **negative number is indicated by a minus sign and a positive number by a plus sign**. Because of hardware limitations, computers must represent everything with binary digits, commonly referred to as bits, 1's and 0's, including the sign of a number. As a consequence, it is customary to represent the sign with a bit placed in the leftmost position of the number. The convention is to make the sign bit equal to **0 for positive and to 1 for negative.**

It is important to realize that both signed and unsigned binary numbers consist of a string of bits when represented in a computer. The user determines whether the number is **signed or unsigned**. If the **binary number is signed**, then the **leftmost bit** represents the **sign** and the **rest of the bits represent the number**. If the **binary number is assumed to be unsigned**, then the **leftmost bit is the most significant bit of the number**.

For example, the string of bits **01001** can be considered as **9 (unsigned binary**) or as **+9 (signed binary)** because **the leftmost bit is 0**. The string of bits **11001** represents the binary **equivalent of 25** when considered as an **unsigned number** and the binary equivalent of **-9** when considered **as a signed number**. This is because the **1 that is in the leftmost position**

designates a negative and the other four bits represent binary 9. Usually, there is no confusion in interpreting the bits if the type of representation for the number is known in advance.

The representation of the signed numbers in the last example is referred to as the **signed-magnitude** convention. In this notation, the number consists of a magnitude and a symbol (+ or -) or a bit (0 or 1) indicating the sign. This is the representation of signed numbers used in ordinary arithmetic.

When arithmetic operations are implemented in a computer, it is more convenient to use a different system, referred to as the **signed complement** system, **for representing negative numbers**. In this system, a negative number is indicated by its complement. Whereas the **signed-magnitude system negates a number by changing its sign**, **the signed-complement system negates a number by taking its complement.** Since positive numbers always start with 0 (plus) in the leftmost position, the complement will always start with a 1, indicating a negative number. The signed-complement system can use either the 1's or the 2's complement, but the **2's complement is the most common**.

As an example, consider the **number 9, represented in binary with eight bits**. +9 is represented with a sign bit of 0 in the leftmost position, followed by the binary equivalent of 9, which gives **00001001**. Note that all eight bits must have a value; therefore, 0's are inserted following the sign bit up to the first 1. Although there is only one way to represent +9, there are **three different ways to represent -9** with eight bits:

Unsigned representation of 9:	**00001001**
Signed-magnitude representation:	10001001
Signed-1's-complement representation:	11110110
Signed-2's-complement representation:	11110111

In signed-magnitude, -9 is obtained from +9 by changing only the sign bit in the leftmost position from 0 to 1. In signed-1's-complement, -9 is obtained by complementing all the bits of +9, including the sign bit. The signed-2's-complement representation of -9 is obtained by taking the 2's complement of the positive number, including the sign bit.

The table below lists **all possible four-bit signed binary numbers in the three representations**. The equivalent decimal number is also shown for reference. Note that the positive numbers in all three representations are identical and have 0 in the leftmost position. The **signed-2's-complement system has only one representation for 0, which is always positive.** The other two systems have either a positive 0 or a negative 0, something not encountered in ordinary arithmetic. Note that all negative numbers have a 1 in the leftmost bit position; that is the way we distinguish them from the positive numbers. With four bits, we can represent 16 binary numbers. In the signed-magnitude and the 1's-complement representations, there are eight positive numbers and eight negative numbers, including two zeros. **In the 2's-complement representation, there are eight positive numbers, including one zero, and eight negative numbers.**

Table: Signed Binary Numbers

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

The **signed-magnitude system** is used in ordinary arithmetic, but **is awkward** when employed in computer **arithmetic because of the separate handling of the sign and the magnitude**. Therefore, the **signed-complement system is normally used**. The **1's complement imposes some difficulties** and is seldom used for arithmetic operations. It is useful as a logical operation, since the change of 1 to 0 or 0 to 1 is equivalent to a logical complement operation. The discussion of **signed binary arithmetic** that follows deals exclusively **with the signed-2's-complement representation of negative numbers**. The same procedures can be applied to the signed-1's-complement system by including the end-around carry as is done with unsigned numbers.

Arithmetic Addition

The addition of two numbers in the **signed-magnitude system** follows the rules of ordinary arithmetic. If the **signs are the same**, we **add the two magnitudes and give the sum the common sign**. If the **signs are different**, we **subtract the smaller magnitude from the larger** and give the **result the sign of the larger magnitude**. For example, (+25) + (-37) = -(37 - 25) = - 12 and is done by subtracting the smaller magnitude 25 from the larger magnitude 37 and using the sign of 37 for the sign of the result. This is a process that requires the comparison of the signs and the magnitudes and then performing either addition or subtraction. The same procedure applies to binary numbers in signed-magnitude representation.

In contrast, the rule for adding numbers in the **signed-2's complement system** does not require a comparison or subtraction, only addition and complementation. The procedure is very simple and can be stated as follows: **The addition of two signed binary numbers with negative numbers represented in signed- 2's-complement form is obtained from the addition of the two numbers, including their sign bits. A carry out of the sign-bit position is discarded.**

Numerical examples for addition are shown below. Note that negative numbers must initially be in 2's complement and that if the sum obtained after the addition is negative, it is in 2's complement form.

+ 6	00000110	- 6	11111010	+ 6	00000110	- 6	11111010
+13	00001101	+13	00001101	-13	11110011	- 13	11110011
+19	00010011	+7	00000111	-7	11111001	-19	11101101

Note that negative numbers must be initially in 2's-complement form and that if the sum obtained after the addition is negative, it is in 2's-complement form. For example, -7 is represented as 11111001, which is the 2s complement of +7.

In each of the four cases, the operation performed is addition with the sign bit included. **Any carry out of the sign-bit position is discarded, and negative results are automatically in 2's-complement form.**

In order to obtain a correct answer, we must ensure that the result has a sufficient number of bits to accommodate the sum. If we start with two *n*-bit numbers and the sum occupies *n* + 1 bits, we say that an overflow occurs. **Overflow** is a problem in computers because the number of bits that hold a number is finite, and a result that exceeds the finite value by 1 cannot be accommodated.

The complement form of representing negative numbers is unfamiliar to those used to the signed-magnitude system. To determine the value of a negative number in signed-2's complement, it is necessary to convert the number to a positive number to place it in a more familiar form. For example, the signed binary number 11111001 is negative because the leftmost bit is 1. Its 2's complement is 00000111, which is the binary equivalent of +7. We therefore recognize the original negative number to be equal to -7.

Arithmetic Subtraction

Subtraction of two signed binary numbers when negative numbers are in 2's complement form is very simple and can be stated as follows: **Take the 2's complement of the subtrahend (including the sign bit) and add it to the minuend (including the sign bit). A carry out of the sign bit position is discarded.**

This procedure stems from the fact that a subtraction operation can be changed to an addition operation if the sign of the subtrahend is changed. This is demonstrated by the following relationship:

(±A) - (+ B) = (±A) + (- B)
(±A) - (- B) = (±A) + (+ B)

But changing a positive number to a negative number is easily done by taking its 2's complement. The reverse is also true because the complement of a negative number in complement form produces the equivalent positive number.

Consider the subtraction of (-6) - (-13) = +7. In binary with eight bits this is written as 11111010 - 11110011. The subtraction is changed to addition by taking the 2's complement of the subtrahend (-13) to give (+13). In binary this is 11111010 + 00001101 = 100000111. Removing the end carry, we obtain the correct answer 00000111 (+7).

It is worth noting that binary numbers in the signed-2's complement system are added and subtracted by the same basic addition and subtraction rules as unsigned numbers. Therefore, **computers need only one common hardware circuit to handle both types of arithmetic**. This consideration has resulted in the **signed-complement system being used in virtually all arithmetic units of computer systems**. 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.

Axiomatic definition of Boolean algebra

In 1854, George Boole developed an algebraic system now called **Boolean algebra**. In 1938, Claude E. Shannon introduced a two-valued Boolean algebra called **switching algebra** that represented the properties of bistable electrical switching circuits. For the formal definition of Boolean algebra, we shall employ the postulates formulated by E. V. Huntington in 1904

.

Boolean algebra is an algebraic structure defined by a set of elements, *B*, together with two binary operators, + and *, provided that the following (Huntington) postulates are satisfied:

1. (a) The structure is closed with respect to the operator +.
(b) The structure is closed with respect to the operator *.

2. (a) The element 0 is an identity element with respect to +; that is, $x + 0 = 0 + x = x$.
(b) The element 1 is an identity element with respect to *; that is, $x * 1 = 1 * x = x$.

3. (a) The structure is commutative with respect to +; that is, $x + y = y + x$.
(b) The structure is commutative with respect to *; that is, $x * y = y * x$.

4. (a) The operator # is distributive over +; that is, $x * (y + z) = (x * y) + (x * z)$.
(b) The operator + is distributive over *; that is, $x + (y * z) = (x + y) * (x + z)$.

5. For every element $x \epsilon B$, there exists an element $x^I \epsilon B$ (called the complement of *x*) such that (a) $x + x^I = 1$ and (b) $x * x^I = 0$.

6. There exist at least two elements $x, y \epsilon B$ such that $x \neq y$.

Basic theorems and properties of Boolean algebra

Duality

The Huntington postulates were listed in pairs and designated by part (a) and part (b). One part may be obtained from the other if the binary operators and the identity elements are interchanged. This important property of Boolean algebra is called the **duality principle** and states that every algebraic expression deducible from the postulates of Boolean algebra remains valid if the operators and identity elements are interchanged. In a two-valued Boolean algebra, the identity elements and the elements of the set *B* are the same: 1 and 0. The duality principle has many applications. **If the dual of an algebraic expression is desired, we simply interchange OR and AND operators and replace 1's by 0's and 0's by 1's**.

Basic Theorems

The list of six theorems of Boolean algebra and four of its postulates are given below.

Table: Postulates and Theorems of Boolean algebra

Postulate 2	(a) $x + 0 = x$	(b) $x * 1 = x$
Postulate 5	(a) $x + x' = 1$	(b) $x * x' = 0$
Theorem 1	(a) $x + x = x$	(b) $x * x = x$
Theorem 2	(a) $x + 1 = 1$	(b) $x * 0 = 0$
Theorem 3, involution	$(x')' = x$	
Postulate 3, commutative	(a) $x + y = y + x$	(b) $xy = yx$
Theorem 4, associative	(a) $x + (y + z) = (x + y) + z$	(b) $x(yz) = (xy)z$
Postulate 4, distributive	(a) $x(y + z) = xy + xz$	(b) $x + yz = (x + y)(x + z)$
Theorem 5, DeMorgan	(a) $(x + y)' = x' y'$	(b) $(xy)' = x' + y'$
Theorem 6, absorption	(a) $x + xy = x$	(b) $x(x + y) = x$

MINIMIZATION OF LOGIC EXPRESSIONS USING K-MAP SIMPLIFICATION UP TO 4 VARIABLE or **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. This task is well understood, but is difficult to execute by manual methods when the logic has more than a few inputs. Fortunately, computer-based logic synthesis tools can minimize a large set of Boolean equations efficiently and quickly. Nevertheless, it is important that a designer understand the underlying mathematical description and solution of the problem.

THE MAP METHOD

The complexity of the digital logic gates that implement a Boolean function is directly related to the complexity of the algebraic expression from which the function is implemented. Although the truth table representation of a function is unique, when it is expressed algebraically it can appear in many different, but equivalent, forms. The map method presented here provides a simple, straightforward procedure for minimizing Boolean functions. This method may be regarded as a pictorial form of a truth table. The map method is also known as the **Karnaugh map** or **K-map**.

A K-map is a diagram made up of squares, with **each square representing one min-term of the function that is to be minimized**. Since any Boolean function can be expressed as a **sum of min-terms**, it follows that a Boolean function is recognized graphically in the map from the area enclosed by those squares whose min-terms are included in the function. In fact, the map presents a visual diagram of all possible ways a function may be expressed in standard form. By recognizing various patterns, the user can derive alternative algebraic expressions for the same function, from which the simplest can be selected.

The simplified expressions produced by the map are always in one of the two standard forms: **sum of products** or **product of sums.** It will be assumed that the simplest algebraic expression is an algebraic expression with a minimum number of terms and with the smallest possible number of literals in each term. This expression produces a circuit diagram with a minimum number of gates and the minimum number of inputs to each gate. We will see subsequently that the simplest expression is not unique: It is sometimes possible to find two or

more expressions that satisfy the minimization criteria. In that case, either solution is satisfactory.

Two-Variable K-Map

The two-variable map is shown in figure (a). There are four minterms for two variables; hence, the map consists of four squares, one for each minterm. The map is redrawn in (b) to show the relationship between the squares and the two variables *x* and *y*. The 0 and 1 marked in each row and column designate the values of variables. **Variable *x* appears primed in row 0 and unprimed in row 1. Similarly, *y* appears primed in column 0 and unprimed in column 1.**

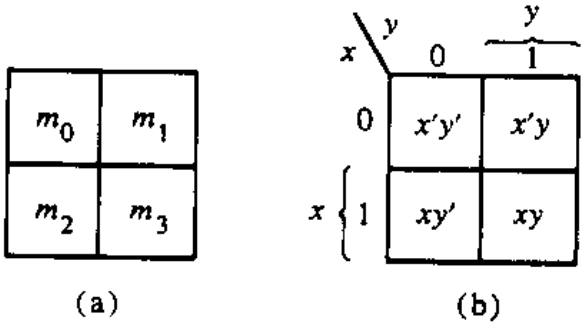


Figure: Two- variable map

If we mark the squares whose minterms belong to a given function, the two-variable map becomes another useful way to represent any one of the 16 Boolean functions of two variables. **As an example**, the function ***xy*** is shown in the below figure (a). Since ***xy* is equal to *m*3, a 1 is placed inside the square that belongs to *m*3.** Similarly, the function ***x* + *y* is** represented in the map of figure (b) by three squares marked with 1's. These squares are found from the minterms of the function: ***m*1 + *m*2 + *m*3 = *x*'*y* + *xy*' + *xy* = *x* + *y***

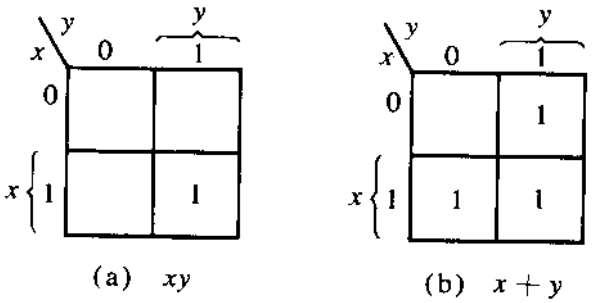


Figure: Representation of functions in the map

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

Three-Variable K-Map

A three-variable K-map is shown in the below figure. There are eight minterms for three binary variables; therefore, the map consists of eight squares. **Note that the minterms are not arranged in a binary sequence, but in a sequence similar to the Gray code.** The characteristic of this sequence is that **only one bit changes in value from one adjacent column to the next.** The map drawn in part (b) is marked with numbers in each row and each

column to show the relationship between the squares and the three variables. For example, the square assigned to *m*5 corresponds to row 1 and column 01. When these two numbers are concatenated, they give the binary number 101, whose decimal equivalent is 5. Each cell of the map corresponds to a unique minterm, so another way of looking at **square *m*5 = *xy*'*z*** is to consider it to be in **the row marked *x*** and the **column belonging to *y*'*z* (column 01**). Note that there are four squares in which each variable is equal to 1 and four in which each is equal to 0. The variable appears unprimed in the former four squares and primed in the latter. For convenience, we write the variable with its letter symbol under the four squares in which it is unprimed.

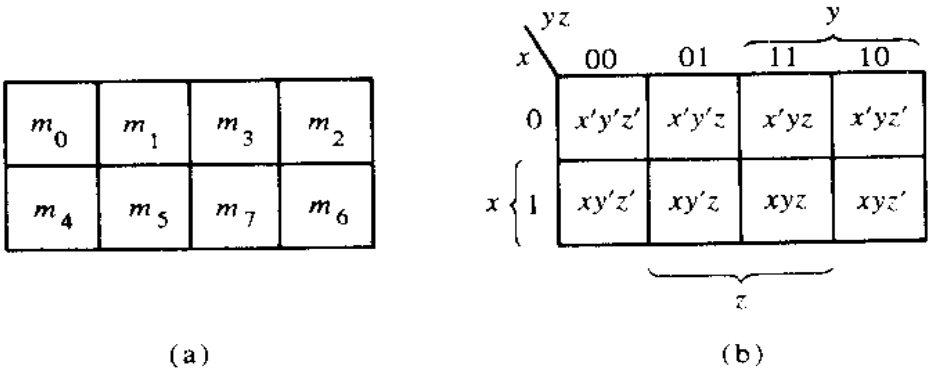


Figure: Three- variable map

To understand the usefulness of the map in simplifying Boolean functions, we must recognize the basic property possessed by adjacent squares: **Any two adjacent squares in the map differ by only one variable,** which is primed in one square and unprimed in the other. For example, *m*5 and *m*7 lie in two adjacent squares. Variable *y* is primed in *m*5 and unprimed in *m*7, whereas the other two variables are the same in both squares.

From the postulates of Boolean algebra, it follows that **the sum of two minterms in adjacent squares can be simplified to a single product term consisting of only two literals.** To clarify this concept, consider the sum of two adjacent squares such as ***m*5 and *m*7**:

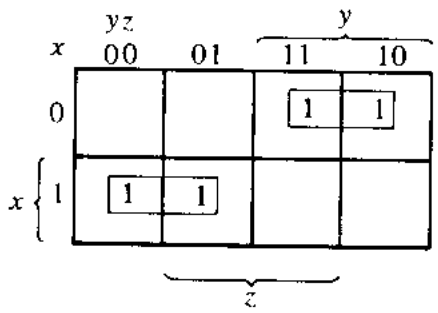
$$m5 + m7 = xy'z + xyz = xz\,(y' + y) = xz$$

Here, the two squares differ by the variable *y*, which can be removed when the sum of the two minterms is formed. Thus, any two minterms in adjacent squares (vertically or horizontally, but not diagonally, adjacent) that are ORed together will cause a removal of the dissimilar variable.

Example 1: Simplify the Boolean function $F(x, y, z) = \Sigma(2, 3, 4, 5)$

Solution:

First, a 1 is marked in each minterm square that represents the function as shown in figure, in which the squares for minterms 010, 011, 100, and 101 are marked with 1's.The next step is to find possible adjacent squares. These are indicated in the map by two rectangles, each enclosing two 1's.



The upper right rectangle represents the area enclosed by *x*'*y*. This area is determined by observing that the two-square area is in row 0, corresponding to *x*', and the last two columns, corresponding to *y*. Similarly, the lower left rectangle represents the product term *xy*'. (The second row represents *x* and the two left columns represent *y*'). The sum of four minterms can be replaced by a sum of only two product terms. The logical sum of these two product terms gives the simplified expression. $F = x'y + xy'$

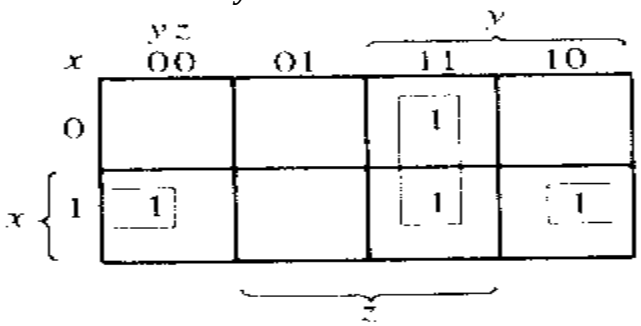
NOTE: In certain cases, **two squares in the map are considered to be adjacent even though they do not touch each other**. For example, ***m*0 is adjacent to *m*2** and ***m*4 is adjacent to *m*6** because their minterms differ by one variable. This difference can be readily verified algebraically:

$$m0 + m2 = x'y'z' + x'yz' = x'z' (y' + y) = x'z'$$
$$m4 + m6 = xy'z' + xyz' = x z' (y' + y) = xz'$$

Consequently, we must modify the **definition of adjacent squares to include this** and other similar cases. We do so by considering **the map as being drawn on a surface in which the right and left edges touch each other to form adjacent squares**.

Example 2: Simplify the Boolean function $F(x, y, z) = \Sigma(3, 4, 6, 7)$
Solution:

The map for this function is shown in the figure below. There are four squares marked with 1's, one for each minterm of the function. Two adjacent squares are combined in the third column to give a two-literal term *yz*. The remaining two squares with 1's are also adjacent by the new definition. These two squares, when combined, give the two-literal term *xz*'. The simplified function then becomes $F = yz + xz'$



NOTE: Consider now **any combination of four adjacent squares in the three-variable map**. Any such combination represents the logical sum of four minterms and **results in an expression with only one literal**. As an example, the logical sum of the four adjacent minterms 0, 2, 4, and 6 reduces to the single literal term *z*':

$$m0 + m2 + m4 + m6 = x'y'z' + x'yz' + xy'z' + xyz'$$
$$= x'z' (y' + y) + xz' (y' + y)$$
$$= x'z' + xz' = z' (x' + x) = z'$$

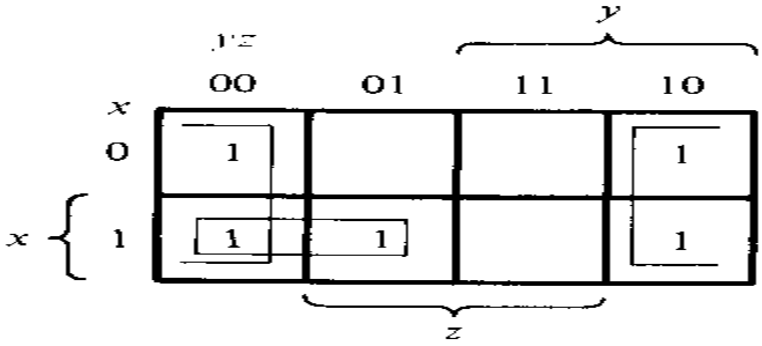
The number of adjacent squares that may be combined must always represent a number that is a power of two, such as 1, 2, 4, and 8. As more adjacent squares are combined, we obtain a product term with fewer literals.

- One square represents one minterm, giving a term with three literals.
- Two adjacent squares represent a term with two literals.
- Four adjacent squares represent a term with one literal.
- Eight adjacent squares encompass the entire map and produce a function that is always equal to 1.

Example 3: Simplify the Boolean function $F(x, y, z) = \Sigma(0, 2, 4, 5, 6)$

Solution:

The map for *F* is shown in below figure. First, we combine the four adjacent squares in the first and last columns to give the single literal term *z*'. The remaining single square, representing minterm 5, is combined with an adjacent square that has already been used once. This is not only permissible, but rather desirable, because the two adjacent squares give the two-literal term *xy*' and the single square represents the three-literal minterm *xy*'*z*. The simplified function is *F* = *z*'+ *xy*'



NOTE: If a function is not expressed in sum-of-minterms form, it is possible to use the map to obtain the minterms of the function and then simplify the function to an expression with a minimum number of terms. It is necessary, however, to make sure that the algebraic expression is in sum-of-products form. Each product term can be plotted in the map in one, two, or more squares. The minterms of the function are then read directly from the map.

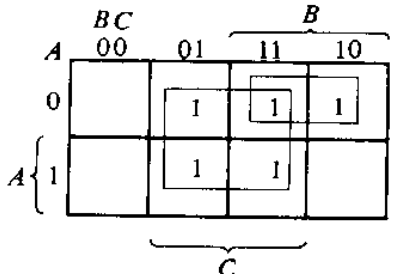
Example 4: For the Boolean function ***F = A'C + A'B + AB'C + BC***

(a) Express this function as a sum of minterms.

(b) Find the minimal sum-of-products expression.

Solution:

(a) Note that *F* is a sum of products. Three product terms in the expression have two literals and are represented in a three-variable map by two squares each. The two squares corresponding to the first term, ***A'C***, are found in figure from the coincidence of *A*' (first row) and *C* (two middle columns) to give squares **001 and 011**. Note that, in marking 1's in the squares, it is possible to find a 1 already placed there from a preceding term. This happens with the second term, ***A'B***, which has 1's in squares **011 and 010**. Square 011 is common with the first term, *A*'*C*, though, so only one 1 is marked in it. Continuing in this fashion, we determine that the term ***AB'C*** belongs in square **101**, corresponding to minterm 5, and the term ***BC*** has two 1's in squares **011 and 111.** The function has a total of five minterms, as indicated by the five 1's in the map of below figure.



The minterms are read directly from the map to be 1, 2, 3, 5, and 7. The function can be expressed in sum-of-minterms form as ***F* (*A, B, C*) = Σ (1, 2, 3, 5, 7)**

(b) The sum-of-products expression, as originally given, has too many terms. It can be simplified, as shown in the map, to an expression with only two terms:

$$F = C + A'B$$

FOUR-VARIABLE K-MAP

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

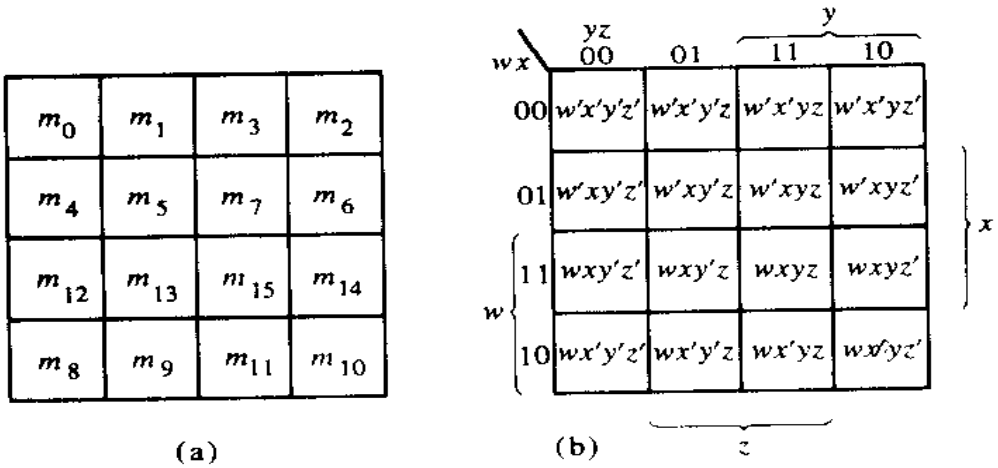


Figure: Four- variable map

The map minimization of four-variable Boolean functions is similar to the method used to minimize three-variable functions. Adjacent squares are defined to be squares next to each other. In addition, **the map is considered to lie on a surface with the top and bottom edges, as well as the right and left edges, touching each other to form adjacent squares.** For example, *m*0 and *m*2 form adjacent squares, as do *m*3 and *m*11. The combination of adjacent squares that is useful during the simplification process is easily determined from inspection of the four-variable map:

- One square represents one minterm, giving a term with four literals.
- Two adjacent squares represent a term with three literals.
- Four adjacent squares represent a term with two literals.
- Eight adjacent squares represent a term with one literal.
- Sixteen adjacent squares produce a function that is always equal to 1.

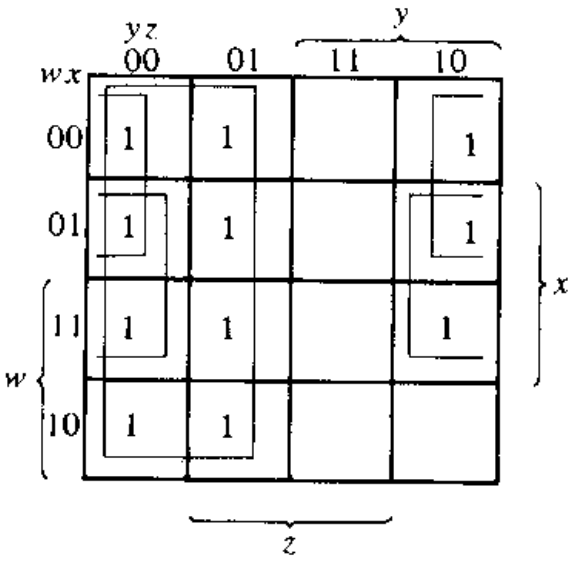
No other combination of squares can simplify the function.

Example 1: Simplify the Boolean function $F(w, x, y, z) = \Sigma(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)$
Solution:

Since the function has four variables, a four-variable map must be used. The minterms listed in the sum are marked by 1's in the map given below. Eight adjacent squares marked with 1's can be combined to form the one literal term *y*'. The remaining three 1's on the right cannot be combined to give a simplified term; they must be combined as two or four adjacent

squares. The larger the number of squares combined, the smaller is the number of literals in the term. In this example, the top two 1's on the right are combined with the top two 1's on the left to give the term *w*'*z*'. Note that it is permissible to use the same square more than once. We are now left with a square marked by 1 in the third row and fourth column (square 1110). Instead of taking this square alone (which will give a term with four literals), we combine it with squares already used to form an area of four adjacent squares. These squares make up the two middle rows and the two end columns, giving the term *xz*'. The simplified function is

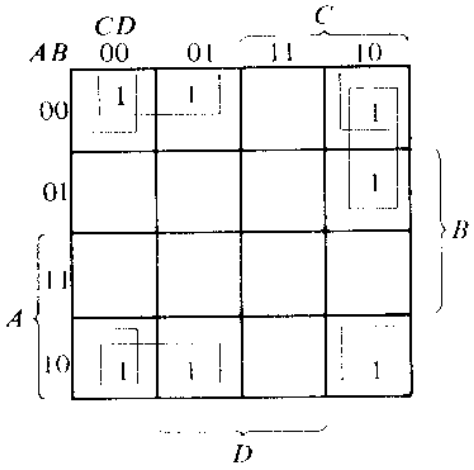
$$F = y' + w'z' + xz'$$



Example 2: Simplify the Boolean function $F = A'B'C' + B'CD' + A'BCD' + AB'C'$

Solution:

The area in the map covered by this function consists of the squares marked with 1's in figure below. The function has four variables and, as expressed, consists of three terms with three literals each and one term with four literals. Each term with three literals is represented in the map by two squares. For example, *A*'*B*'*C*' is represented in squares 0000 and 0001. The function can be simplified in the map by taking the 1's in the four corners to give the term *B'D*'. This is possible because these four squares are adjacent when the map is drawn in a surface with top and bottom edges, as well as left and right edges, touching one another. The two left-hand 1's in the top row are combined with the two 1's in the bottom row to give the term *B*'*C*'. The remaining 1 may be combined in a two square area to give the term *A*'*CD*'. The simplified function is $F = B'D' + B'C' + A'CD'$



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). Sometimes there may be two or more expressions that satisfy the simplification criteria. The procedure for combining squares in the map may be made more systematic if we understand the meaning of two special types of 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 essential prime implicants are found by looking at each square marked with a 1 and checking the number of prime implicants that cover it. The prime implicant is essential if it is the only prime implicant that covers the minterm.

Example:
Consider the following four-variable Boolean function:

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

The minterms of the function are marked with 1's in the maps of the figure below. The partial map **(Fig. (a))** shows **two essential prime implicants**, each formed by collapsing four cells into a term having only two literals. One term is essential because there is only one way to include minterm ***m*0** within four adjacent squares. These four squares define the term *B*'*D*'. Similarly, there is only one way that minterm ***m*5** can be combined with four adjacent squares, and this gives the second term *BD*. The two essential prime implicants cover eight minterms. The three minterms that were omitted from the partial map (*m*3, *m*9, and *m*11) must be considered next.

Figure (b) shows all possible ways that the three minterms can be covered with **prime implicants**. Minterm *m*3 can be covered with either prime implicant *CD* or prime implicant *B*'*C*. Minterm *m*9 can be covered with either *AD* or *AB*'. Minterm *m*11 is covered with any one of the four prime implicants. The simplified expression is obtained from the logical sum of the two essential prime implicants and any two prime implicants 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 above example has demonstrated that the identification of the prime implicants in the map helps in determining the alternatives that are available for obtaining a simplified expression.

The procedure for finding the simplified expression from the map requires that we first determine all the essential prime implicants. 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. Occasionally, there may be more than one way of combining squares, and each combination may produce an equally simplified expression.

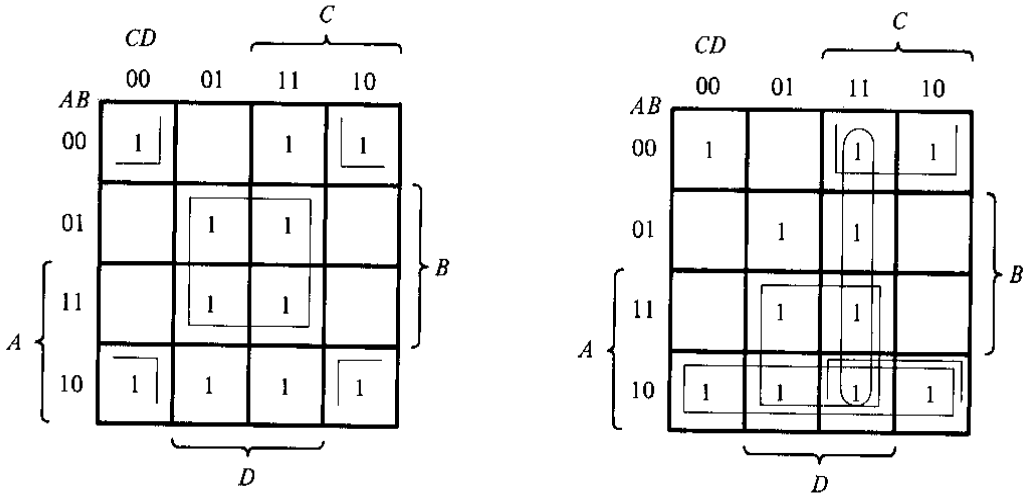


Figure: Simplification using prime implicants

PRODUCT-OF-SUMS SIMPLIFICATION

The minimized Boolean functions derived from the map in all previous examples were expressed in sum-of-products form. With a minor modification, the product-of-sums form can be obtained. The procedure for obtaining a minimized function in product-of-sums form follows from the basic properties of Boolean functions. **The 1's placed in the squares of the map represent the minterms of the function. The minterms not included in the standard sum-of-products form of a function denote the complement of the function**. From this observation, we see that the complement of a function is represented in the map by the squares not marked by 1's. 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 (i.e., of F^I). The complement of F^I gives us back the function F in product-of-sums form (a consequence of DeMorgan's theorem). Because of the generalized DeMorgan's theorem, the function so obtained is automatically in product-of-sums form. The best way to show this is by example.

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

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

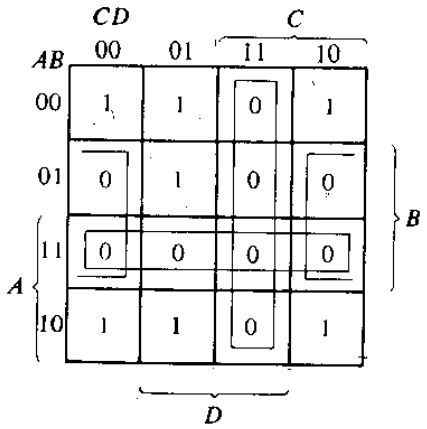
Solution:

The 1's marked in the map of below figure represent all the minterms of the function. The squares marked with 0's represent the minterms not included in F and therefore denote the complement of F. Combining the squares with 1's gives the simplified function in sum-of-products form: (a) $F = B'D' + B'C' + A'C'D$

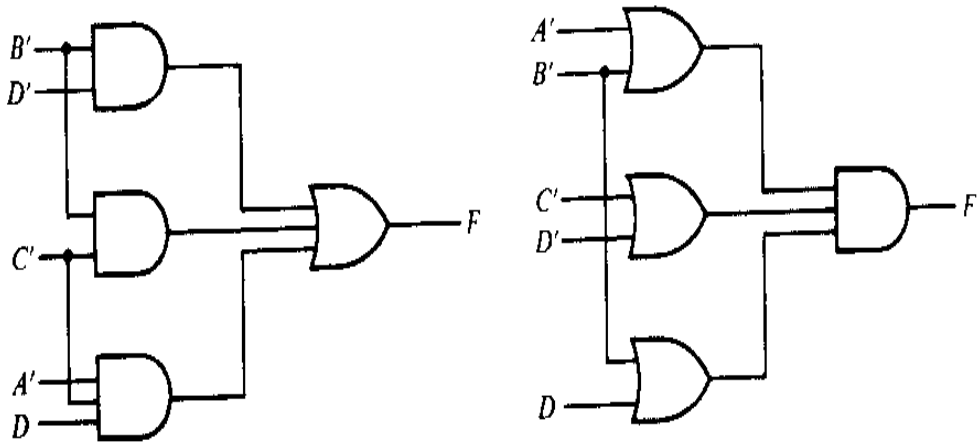
If the squares marked with 0's are combined, as shown in the diagram, we obtain the simplified complemented function:

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

Applying DeMorgan's theorem (by taking the dual and complementing each literal), we obtain the simplified function in product of-sums form: (b) $F = (A' + B')(C' + D')(B' + D)$



The gate-level implementation of the simplified expressions obtained is shown below. The sum-of-products expression is implemented in (a) with a group of AND gates, one for each AND term. The outputs of the AND gates are connected to the inputs of a single OR gate. The same function is implemented in (b) in its product-of-sums form with a group of OR gates, one for each OR term. The outputs of the OR gates are connected to the inputs of a single AND gate. In each case, it is assumed that the input variables are directly available in their complement, so inverters are not needed. The configuration pattern established in below figure is the general form by which any Boolean function is implemented when expressed in one of the standard forms. AND gates are connected to a single OR gate when in sum-of-products form; OR gates are connected to a single AND gate when in product-of-sums form. Either configuration forms two levels of gates. Thus, the implementation of a function in a standard form is said to be a **two-level implementation**. The two-level implementation may not be practical, depending on the number of inputs to the gates.

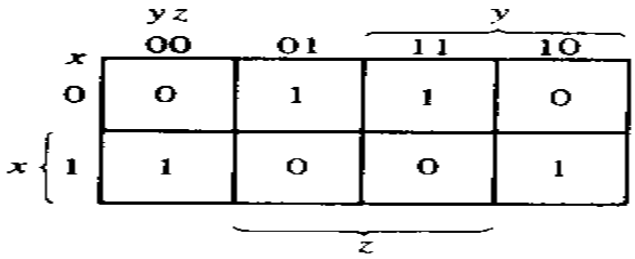


(a) $F = B'D' + B'C' + A'C'D$ (b) $F = (A' + B')(C' + D')(B' + D)$

Figure: Gate implementation of the $F(A, B, C, D) = \Sigma(0, 1, 2, 5, 8, 9, 10)$

The above example showed the procedure for obtaining the product-of-sums simplification when the function is originally expressed in the sum-of-minterms canonical form. The procedure is also valid when the function is originally expressed in the product of-maxterms canonical form.

Consider, for example, ***F* (*x, y, z*) = Σ (1, 3, 4, 6),** it is in sum of minterms form. In product-of-maxterms form, it is expressed as ***F* (*x, y, z*) = π (0, 2, 5, 7)**. In other words, the 1's of the function represent the minterms and the 0's represent the maxterms. The map for this function is shown below.



One can start simplifying the function by first marking the 1's for each minterm that the function is a 1. The remaining squares are marked by 0's. If, instead, the product of maxterms is initially given, one can start marking 0's in those squares listed in the function; the remaining squares are then marked by 1's. Once the 1's and 0's are marked, the function can be simplified in either one of the standard forms. For the **sum of products**, we combine the 1's to obtain

$$F = x'z + xz'.$$

For the **product of sums**, we combine the 0's to obtain the simplified complemented function

$$F' = xz + x'z'$$

which shows that the exclusive-OR function is the complement of the equivalence function. Taking the complement of *F*', we obtain the simplified function in **product-of-sums** form:

$$F = (x' + z')\,(x + z)$$

To enter a function expressed in product-of-sums form into the map, use the complement of the function to find the squares that are to be marked by 0's.

For example, the function $F = (A' + B' + C')\,(B + D)$ can be entered into the map by first taking its complement, namely, $F' = ABC + B'D'$ and then marking 0's in the squares representing the minterms of *F*'. The remaining squares are marked with 1's.

DON'T-CARE CONDITIONS

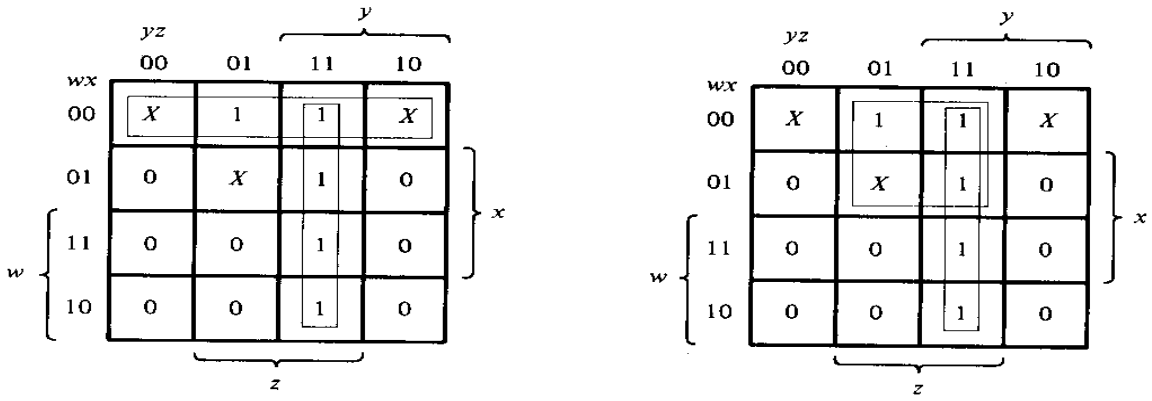
The logical sum of the minterms associated with a Boolean function specifies the conditions under which the function is equal to 1. The function is equal to 0 for the rest of the minterms. This pair of conditions assumes that all the combinations of the values for the variables of the function are valid. In practice, in some applications the function is not specified for certain combinations of the variables. As an example, the four-bit binary code for the decimal digits has six combinations that are not used and consequently are considered to be unspecified**. Functions that have unspecified outputs for some input combinations are called incompletely specified functions**. In most applications, we simply don't care what value is assumed by the function for the unspecified minterms. For this reason, it is customary to call the unspecified minterms of a function **don't-care conditions**. These don't-care conditions can be used on a map to provide further simplification of the Boolean expression.

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

In choosing adjacent squares to simplify the function in a map, the don't-care minterms may be assumed to be either 0 or 1. When simplifying the function, we can choose to include each don't-care minterm with either the 1's or the 0's, depending on which combination gives the simplest expression.

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

Solution: The minterms of *F* are the variable combinations that make the function equal to 1. The minterms of *d* are the don't-care minterms that may be assigned either 0 or 1. The map simplification is shown below.



The minterms of *F* are marked by 1's, those of *d* are marked by X's, and the remaining squares are filled with 0's. To get the simplified expression in sum-of-products form, we must include all five 1's in the map, but we may or may not include any of the X's, depending on the way the function is simplified. The term *yz* covers the four minterms in the third column. The remaining minterm, *m*1, can be combined with minterm *m*3 to give the three-literal term *w*'*x*'*z*. However, by including one or two adjacent X's we can combine four adjacent squares to give a two-literal term. In Fig. (a), don't-care minterms 0 and 2 are included with the 1's, resulting in the simplified function $F = yz + w'x$.

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

The above example has shown that the don't-care minterms in the map are initially marked with X's and are considered as being either 0 or 1. The choice between 0 and 1 is made depending on the way the incompletely specified function is simplified. Once the choice is made, the simplified function obtained will consist of a sum of minterms that includes those minterms which were initially unspecified and have been chosen to be included with the 1's. Consider the two simplified expressions obtained in the above example:

$$F(w, x, y, z) = yz + w'x' = \Sigma(0, 1, 2, 3, 7, 11, 15)$$
$$F(w, x, y, z) = yz + w'z = \Sigma(1, 3, 5, 7, 11, 15)$$

Both expressions include minterms 1, 3, 7, 11, and 15 that make the function *F* equal to 1. The don't-care minterms 0, 2, and 5 are treated differently in each expression. The first expression includes minterms 0 and 2 with the 1's and leaves minterm 5 with the 0's. The second expression includes minterm 5 with the 1's and leaves minterms 0 and 2 with the 0's. The two expressions represent two functions that are not algebraically equal. Both cover the specified minterms of the function, but each covers different don't-care minterms. As far as the incompletely specified function is concerned, either expression is acceptable because the only difference is in the value of *F* for the don't-care minterms.

It is also possible to **obtain a simplified product-of-sums expression** for the above given function. In this case, the only way to combine the 0's is to include don't-care minterms 0 and 2 with the 0's to give a simplified complemented function: $F' = z' + wy'$. Taking the complement of *F*' gives the simplified expression in product-of-sums form:

$$F(w, x, y, z) = z(w' + y) = \Sigma(1, 3, 5, 7, 11, 15)$$

In this case, we include minterms 0 and 2 with the 0's and minterm 5 with the 1's.

COMBINATIONAL CIRCUITS-I

INTRODUCTION

Logic circuits for digital systems may be combinational or sequential. A **combinational circuit** consists of logic gates **whose outputs at any time are determined from only the present combination of inputs**. A combinational circuit performs an operation that can be specified logically by a set of Boolean functions. In contrast, **sequential circuits employ storage elements in addition to logic gates.** Their outputs are a function of the inputs and the stage of the storage elements. Because the state of the storage elements is a function of previous inputs, the **outputs of a sequential circuit depend not only on present value of inputs, but also on past inputs,** and the circuit behavior must be specified by a time sequence of inputs and internal states.

COMBINATIONAL CIRCUITS

A **combinational circuit consists of an interconnection of logic gates.** Combinational logic gates react to the values of the signals at their inputs and produce the value of the output signal, transforming binary information from the given input data to a required output data. A block diagram of a combinational circuit is shown below. The ***n* input binary variables** come from an external source; the ***m* output variables are produced by the internal combinational logic circuit** and go to an external destination. Each input and output variable exists physically as an analog signal whose values are interpreted to be a binary signal that represents logic 1 and logic 0. (Note: Logic simulators show only 0's and 1's, not the actual analog signals.) In many applications, the source and destination are storage registers. If the registers are included with the combinational gates, then the total circuit must be considered to be a sequential circuit.

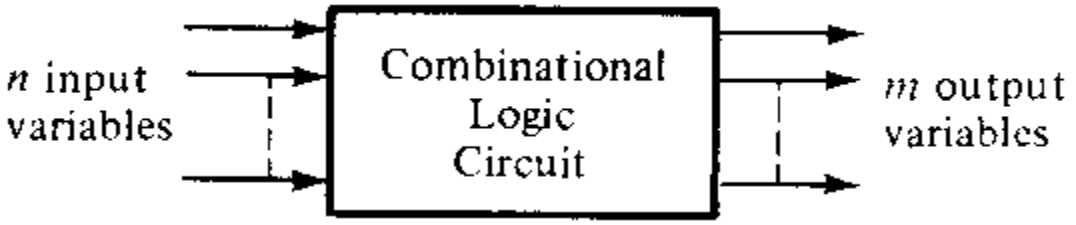


Figure: Block diagram of combinational circuit

For *n* input variables, there are 2^n possible combinations of the binary inputs. For each possible input combination, there is one possible value for each output variable. Thus, a combinational circuit can be specified with a truth table that lists the output values for each combination of input variables. A combinational circuit also can be described by *m* Boolean functions, one for each output variable. Each output function is expressed in terms of the *n* input variables.

DESIGN PROCEDURE

The design of combinational circuits starts from the specification of the design objective and culminates in a logic circuit diagram or a set of Boolean functions from which the logic diagram can be obtained. The procedure involves the following steps:

1. The problem is stated.
2. The number of available input variables and required output variables is determined.
3. The input and output variables are assigned letter symbols.
4. The truth table that defines the required relationship between inputs and outputs is derived.
5. The simplified Boolean functions for each output is obtained.
6. The logic diagram is drawn.

A truth table for a combinational circuit consists of input columns and output columns. The input columns are obtained from the 2^n binary numbers for the *n* input variables. The binary values for the outputs are determined from the stated specifications. The output functions specified in the truth table give the exact definition of the combinational circuit. It is important that the verbal specifications be interpreted correctly in the truth table.

BINARY ADDER–SUBTRACTOR

Digital computers perform a variety of information-processing tasks. Among the functions encountered are the various arithmetic operations. The most basic arithmetic operation is the addition of two binary digits. This simple addition consists of four possible elementary operations: 0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1, and 1 + 1 = 10. The first three operations produce a sum of one digit, but when **both augend and addend bits are equal to 1**, the binary sum consists of two digits. The **higher significant bit of this result is called a carry.** When the augend and addend numbers contain more significant digits, the carry obtained from the addition of two bits is added to the next higher order pair of significant bits. **A combinational circuit that performs the addition of two bits is called a half adder.** One that performs the **addition of three bits** (two significant bits and a previous carry**) is a full adder**. The names of the circuits stem from the fact that two half adders can be employed to implement a full adder.

A binary adder–subtractor is a combinational circuit that performs the arithmetic operations of addition and subtraction with binary numbers. We will develop this circuit by means of a hierarchical design. The half adder design is carried out first, from which we develop the full adder. Connecting *n* full adders in cascade produces a binary adder for two *n* -bit numbers. The subtraction circuit is included in a complementing circuit.

HALF ADDER

From the verbal explanation of a half adder, we find that this circuit needs two binary inputs and two binary outputs. The input variables designate the augend and addend bits; the output variables produce the sum and carry. We assign **symbols *x* and *y* to the two inputs** and ***S* (for sum) and *C* (for carry) to the outputs**. The truth table for the half adder is listed in

below table. The *C* output is 1 only when both inputs are 1. The *S* output represents the least significant bit of the sum.

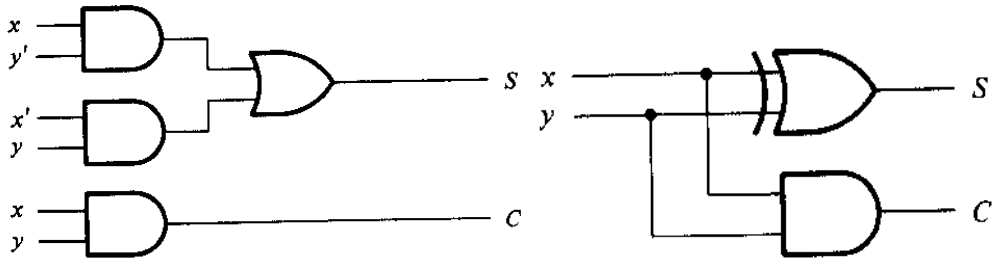
Table: Truth table for Half-adder

x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

The simplified Boolean functions for the two outputs can be obtained directly from the truth table. The simplified sum-of-products expressions are

$$S = x'y + xy'$$
$$C = xy$$

The logic diagram of the half adder implemented in sum of products is shown in Fig. (a). It can be also implemented with an exclusive-OR and an AND gate as shown in Fig. (b), this form is used to show that two half adders can be used to construct a full adder.



(a) S = x'y + xy' **(b) S = x ⊕ y**
C = xy **C = xy**

Figure: Implementation of Half adder

FULL ADDER

Addition of *n*-bit binary numbers requires the use of a full adder, and the process of addition proceeds on a bit-by-bit basis, right to left, beginning with the least significant bit. After the least significant bit, addition at each position adds not only the respective bits of the words, but must also consider a possible carry bit from addition at the previous position.

A full adder is a combinational circuit that forms the arithmetic sum of three bits. It consists of three inputs and two outputs. Two of the input variables, denoted by *x* and *y*, represent the two significant bits to be added. The third input, *z*, represents the carry from the previous lower significant position. Two outputs are necessary because the arithmetic sum of three binary digits ranges in value from 0 to 3, and binary representation of 2 or 3 needs two bits. The two outputs are designated by the symbols *S* for sum and *C* for carry.

The binary variable *S* gives the value of the least significant bit of the sum. The binary variable *C* gives the output carry formed by adding the input carry and the bits of the words. The truth table of the full adder is listed in below table. The eight rows under the input variables designate all possible combinations of the three variables. The output variables are determined from the arithmetic sum of the input bits. When all input bits are 0, the output is 0. The *S* output is equal to 1 when only one input is equal to 1 or when all three inputs are equal to 1. The *C* output has a carry of 1 if two or three inputs are equal to 1.

Table: Truth table for Full-adder

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

The input and output bits of the combinational circuit have different interpretations at various stages of the problem. On the one hand, physically, the binary signals of the inputs are considered binary digits to be added arithmetically to form a two-digit sum at the output. On the other hand, the same binary values are considered as variables of Boolean functions when expressed in the truth table or when the circuit is implemented with logic gates. The maps for the outputs of the full adder are shown below.

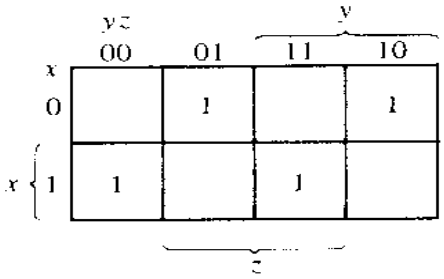


Figure: (a) For sum

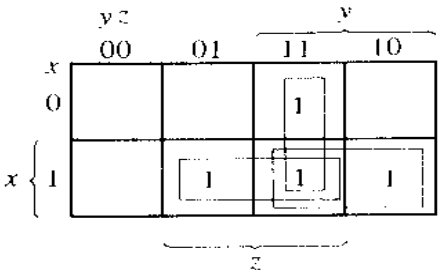


Figure: (b) For carry

The simplified expressions are: $S = x'y'z + x'yz' + xy'z' + xyz$ and $C = xy + xz + yz$

The logic diagram for the full adder implemented in sum-of-products form is shown in Fig. 1. It can also be implemented with two half adders and one OR gate, as shown in Fig. 2. The *S* output from the second half adder is the exclusive-OR of *z* and the output of the first half adder, giving

$$S = z \oplus (x \oplus y)$$
$$= z'(xy' + x'y) + z(xy' + x'y)'$$
$$= z'(xy' + x'y) + z(xy + x'y')$$
$$= xy'z' + x'yz' + xyz + x'y'z$$

The carry output is

$$C = z(xy' + x'y) + xy = xy'z + x'yz + xy$$

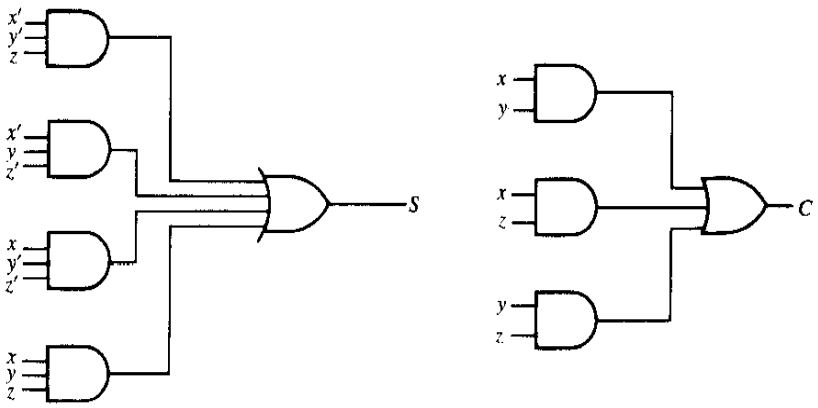


Figure 1: Implementation of full adder in sum-of-products form

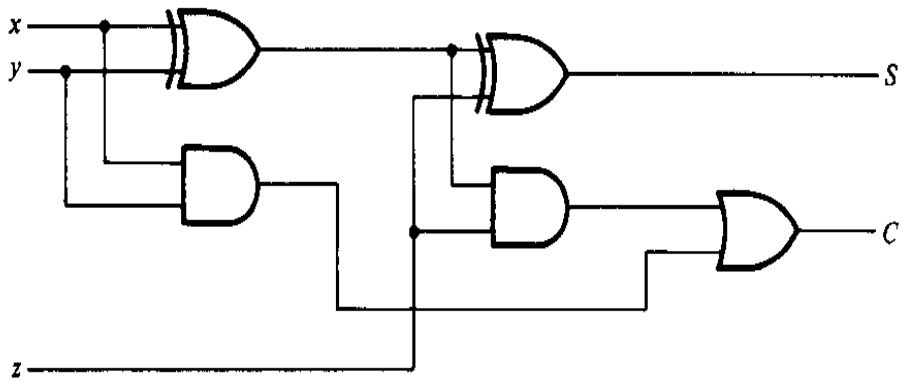


Figure 2: Implementation of full adder with two half adders and an OR gate

SUBTRACTORS

The subtraction of two binary numbers may be accomplished by taking the complement of the subtrahend and adding it to the minuend. By this method, the subtraction operation becomes an addition operation requiring full adders for its machine implementation.

HALF SUBTRACTORS

A half- subtractor is a combinational circuit that subtracts two bits and produces their difference. It also has an output to specify if a 1 has been borrowed. Designate the minuend bit by x and the subtrahend bit by y. To perform x-y, we have to check the relative magnitude of x and y. If **x >y,** we have three possibilities: **0 – 0 = 0, 1 – 0 = 0, and 1 – 1 = 0**. The result is called the **difference bit**. If **x <y,** we have 0 – 1, and it is necessary to borrow a 1 from the next higher stage. The 1 borrowed from the next higher stage adds 2 to the minuend digit. With the minuend equal to 2, the difference becomes 2 – 1 =1.

The half subtractor needs **two outputs**. One output generates the **difference** and will be designated by **symbol D**. The second output, designated **B for borrow**, generates the binary signal that informs the next stage that a 1 has been borrowed. The **truth table** for the input-output relationships of a half subtractor can be derived as follows:

x	*y*	*B*	*D*
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

The output borrow B is a 0 as long as x > y. It is a 1 for x = 0 and y =1. The D output is the result of the arithmetic operation 2B + x – y.

The Boolean functions for two outputs of the subtractor are derived directly from the truth table:

$$D = x'y + xy'$$

$$B = x'y$$

It is interesting to note that logic for D is exactly the same as the logic for output S in the half adder. The logic diagram for half subtractor is given below:

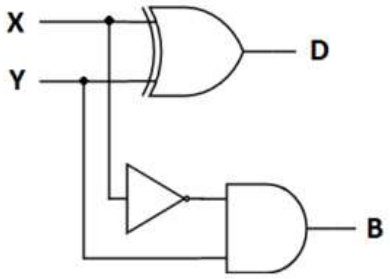


Figure: Implementation of Half subtractor

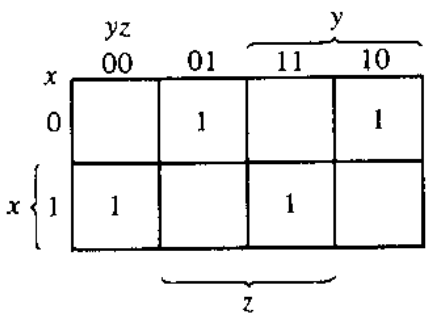
FULL SUBTRACTOR

A full subtractor is a combinational circuit that performs a subtraction between two bits, taking into account that a 1 may have been borrowed by a lower significant stage. This circuit has three inputs and two outputs. The **three inputs x, y, and z, denote the minuend, subtrahend, and previous borrow, respectively.** The two outputs, D and B, represent the difference and output borrow, respectively. The truth table for the circuit is

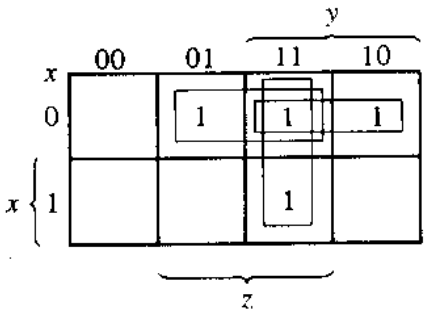
x	*y*	*z*	*B*	*D*
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

The eight rows under this input variables designate all possible combinations of 1's and 0's that binary variables may take. The 1's and 0's for the output variables are determined from the subtraction of **x – y – z**. The combination having input borrow z = 0 reduce to same four conditions of the half adder. For x = 0, y = 0, and z = 1, we have to borrow a 1 from the next stage, which makes B = 1 and adds 2 to x. Since 2 – 0 – 1 = 1, D = 1. For x = 0, yz = 11, we need to borrow again, making B = 1 and x = 2. Since 2 – 1 – 1 = 0, D = 0. For x = 1 and yz = 01, we have x – y – z = 0, which makes B = 0 and D = 0. Finally, for x = 1, y = 1, z = 1, we have to borrow 1, making B = 1 and x = 3, and 3 – 1 – 1 = 1, making D =1.

The simplified Boolean functions for the two outputs of the full subtractor are derived in the maps given below:



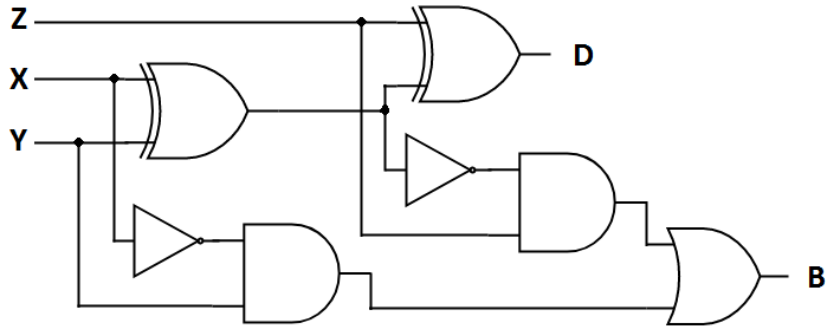
$D = x'y'z + x'yz + xy'z' + xyz$



$B = x'y + x'z + yz$

The simplified sum of products output functions are:

$$D = xy'z' + x'yz' + xyz + x'y'z = z \oplus (x \oplus y)$$
$$B = x'y + x'z + yz = x' (y \oplus z) + yz$$



4-BIT ADDER AND 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.

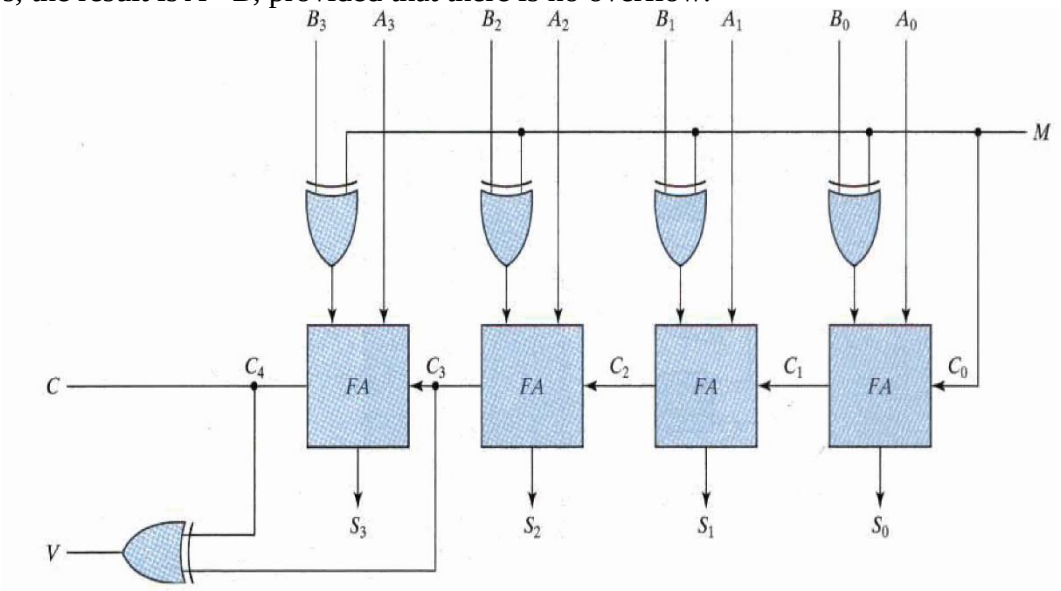


Figure: Four-bit adder–subtractor (with overflow detection)

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 above. 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 $B \oplus 1 = B'$ 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.

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**. This is true for binary or decimal numbers, signed or unsigned. When the addition is performed with paper and pencil, an overflow is not a problem, since there is no limit by the width of the page to write down the sum. Overflow is a problem in digital computers because the number of bits that hold the number is finite and a result that contains *n* + 1 bits cannot be accommodated by an *n* -bit word. For this reason, many computers detect the occurrence of an overflow, and when it occurs, a corresponding flip-flop is set that can then be checked by the user.

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, since adding a positive number to a negative number produces a result whose magnitude is smaller than the larger of the two original numbers. An overflow may occur if the two numbers added are either positive or negative. To see how this can happen, consider the following example: 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. This is also true for -70 and -80. The two additions in binary are shown next, together with the last two carries:

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. This is indicated in the examples in which the two carries are explicitly shown. 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. For this method to work correctly, the 2's complement of a negative number must be computed by taking the 1's complement and adding 1. This takes care of the condition when the maximum negative number is complemented.

The binary adder–subtractor circuit with outputs *C* and *V* is shown in above figure. If the two binary numbers are considered to be unsigned, then the *C* bit detects a carry after addition or borrow after subtraction. If the numbers are considered to be signed, then the *V* bit detects an overflow. If $V = 0$ after an addition or subtraction, then no overflow occurred and the n -bit result is correct. If $V = 1$, then the result of the operation contains $n + 1$ bits, but only the rightmost n bits of the number fit in the space available, so an overflow has occurred. The $(n + 1)$ th bit is the actual sign and has been shifted out of position.

COMPARATORS:

The comparison of two numbers is an operation that determines whether one number is greater than, less than, or equal to the other number. 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 $A < B$.

The circuit works by comparing the bits of the two numbers starting from the **most significant bit (MSB) and moving toward the least significant bit (LSB).** At each bit position, the two corresponding bits of the numbers are compared. If the **bit in the first number is greater than the corresponding bit in the second number**, **the A>B output is set to 1**, and the circuit immediately determines that the first number is greater than the second. Similarly, if the bit in the second number is greater than the corresponding bit in the first number, **the A<B output is set to 1**, and the circuit immediately determines that the first number is less than the second.

If the two corresponding bits are equal, the circuit moves to the next bit position and compares the next pair of bits. This process continues until all the bits have been compared. If at any point in the comparison, the circuit determines that the first number is greater or less than the second number, the comparison is terminated, and the appropriate output is generated. If all the bits are equal, the circuit generates an **A=B output**, indicating that the two numbers are equal.

There are different ways to implement a magnitude comparator, such as using a combination of XOR, AND, and OR gates, or by using a cascaded arrangement of full adders. The choice of implementation depends on factors such as speed, complexity, and power consumption.

1-Bit Magnitude Comparator

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.

A	B	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: AB'

A < B: A'B

A = B: A'B'+ AB

From the above expressions, we can derive the following formula.

(A<B)+(A>B) = A'B+AB'

Taking complement both sides

((A<B) + (A>B))' = (A'B + AB')'

((A<B) + (A>B))' = (A'B)' (AB')'

((A<B) + (A>B))' = (A + B') (A' +B)

((A<B) + (A>B))' =(AA' + AB + A'B' +BB')

" " = (AB + A'B')

Thus,

((A<B) + (A > B))' = (A = B)

Figure: Derivation of 1-Bit Magnitude Comparator

By using these Boolean expressions, we can implement a logic circuit for this comparator as given below.

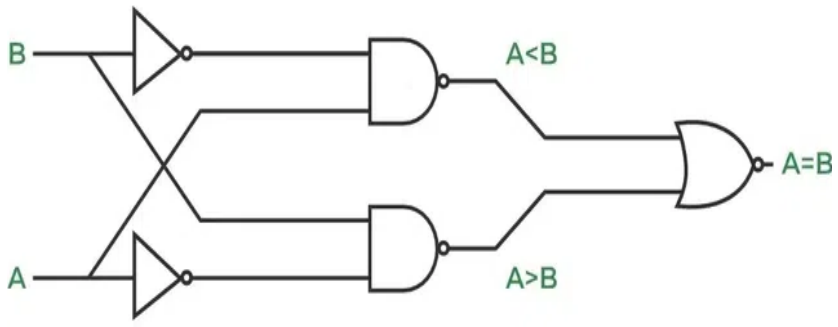


Figure: 1-bit Magnitude comparator

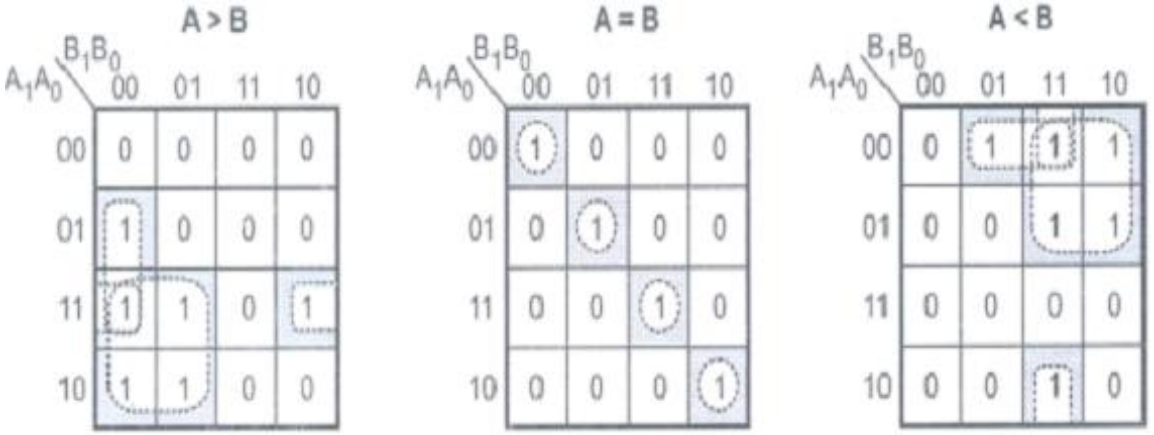
2-Bit Magnitude Comparator

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.

INPUT				OUTPUT		
A1	**A0**	**B1**	**B0**	**A<B**	**A=B**	**A>B**
0	0	0	0	0	1	0
0	0	0	1	1	0	0
0	0	1	0	1	0	0
0	0	1	1	1	0	0
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	1	0	0
0	1	1	1	1	0	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	0	1	0
1	0	1	1	1	0	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	0	1	0

From the above truth table, K-map for each output can be drawn as follows.



F (A > B) = $A_0B_1'B_0'$ + A_1B_1' + $A_1A_0B_0'$

F (A = B) = $A_1'A_0'B_1'B_0'$ + $A_1'A_0B_1'B_0$ + $A_1A_0B_1B_0$ + $A_1A_0'B_1B_0'$
= $A_1'B_1'(A_0'B_0' + A_0B_0)$ + $A_1B_1(A_0'B_0' + A_0B_0)$
= $(A_0 \odot B_0)(A_1 \odot B_1)$

F (A < B) = $A_1'A_0'B_0$ + $A_0'B_1B_0$ + $A_1'B_1$

By using these Boolean expressions, we can implement a logic circuit for this comparator as given below.

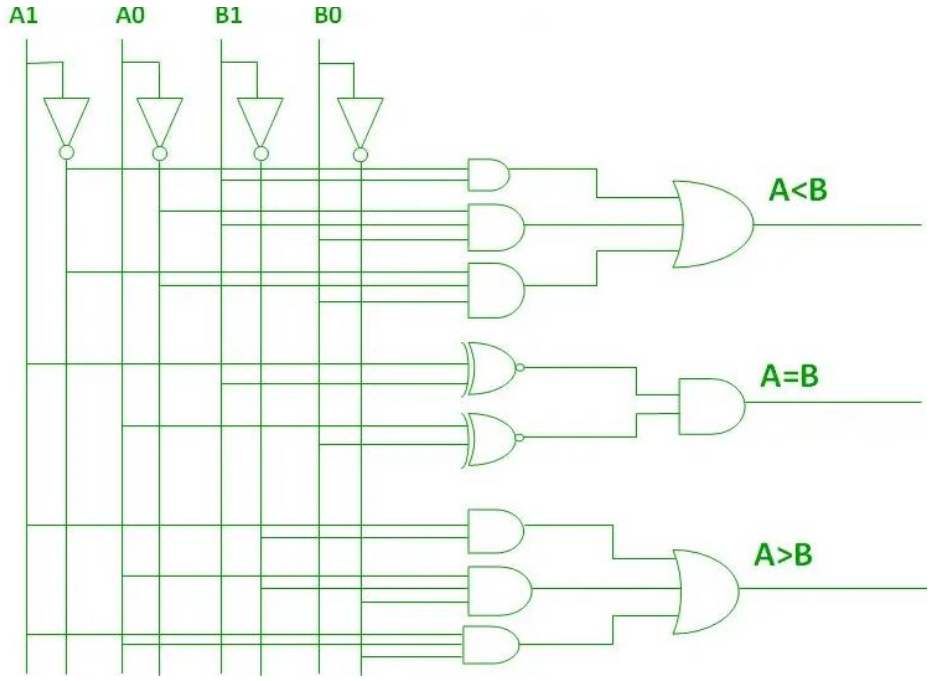


Figure: 2-bit Magnitude comparator

4-Bit Magnitude Comparator

The circuit for comparing two n -bit numbers has 2^{2n} entries in the truth table and becomes too cumbersome, even with n = 3. On the other hand, as one may suspect, a comparator circuit possesses a certain amount of regularity. Digital functions that possess an inherent well-defined regularity can usually be designed by means of an algorithm—a procedure which specifies a finite set of steps that, if followed, give the solution to a problem. We illustrate this method here by deriving an algorithm for the design of a four-bit magnitude comparator.

The algorithm is a direct application of the procedure a person uses to compare the relative magnitudes of two numbers. Consider two numbers, *A* and *B*, with four digits each. Write the coefficients of the numbers in descending order of significance:

$$A = A_3\ A_2\ A_1\ A_0$$
$$B = B_3\ B_2\ B_1\ B_0$$

Each subscripted letter represents one of the digits in the number. The two numbers are equal if all pairs of significant digits are equal: $\mathbf{A_3 = B_3, A_2 = B_2, A_1 = B_1,}$ **and** $\mathbf{A_0 = B_0}$**.** When the numbers are binary, the digits are either 1 or 0, and the equality of each pair of bits can be expressed logically with an exclusive-NOR function as

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

where X_i = 1 only if the pair of bits in position i are equal (i.e., if both are 1 or both are 0).

The equality of the two numbers *A* and *B* is displayed in a combinational circuit by an output binary variable that we designate by the symbol (*A* = *B*). This binary variable is equal to 1 if the input numbers, *A* and *B*, are equal, and is equal to 0 otherwise. For equality to exist, all X_i variables must be equal to 1, a condition that dictates an AND operation of all variables:

$$\mathbf{(A = B) = X_3X_2X_1X_0}$$

The binary variable (*A* = *B*) is equal to 1 only if all pairs of digits of the two numbers are equal.

To determine whether *A* is greater or less than *B*, we inspect the relative magnitudes of pairs of significant digits, starting from the most significant position. If the two digits of a pair are equal, we compare the next lower significant pair of digits. The comparison continues until a pair of unequal digits is reached. If the corresponding digit of *A* is 1 and that of *B* is 0, we conclude that $A > B$. If the corresponding digit of *A* is 0 and that of *B* is 1, we have $A < B$. The sequential comparison can be expressed logically by the two Boolean functions

$$(A > B) = A_3B'_3 + X_3 A_2B'_2 + X_3X_2A_1B'_1 + X_3X_2X_1A_0B'_0$$
$$(A < B) = A'_3B_3 + X_3A'_2B_2 + X_3X_2A'_1B_1 + X_3X_2X_1A'_0B_0$$

The symbols $(A > B)$ and $(A < B)$ are *binary* output variables that are equal to 1 when $A > B$ and $A < B$, respectively.

The gate implementation of the three output variables just derived is simpler than it seems because it involves a certain amount of repetition. The unequal outputs can use the same gates that are needed to generate the equal output. The logic diagram of the four-bit magnitude comparator is shown below. The four X outputs are generated with exclusive-NOR circuits and are applied to an AND gate to give the output binary variable $(A = B)$. The other two outputs use the X variables to generate the Boolean functions listed previously. This is a multilevel implementation and has a regular pattern. The procedure for obtaining magnitude comparator circuits for binary numbers with more than four bits is obvious from this example.

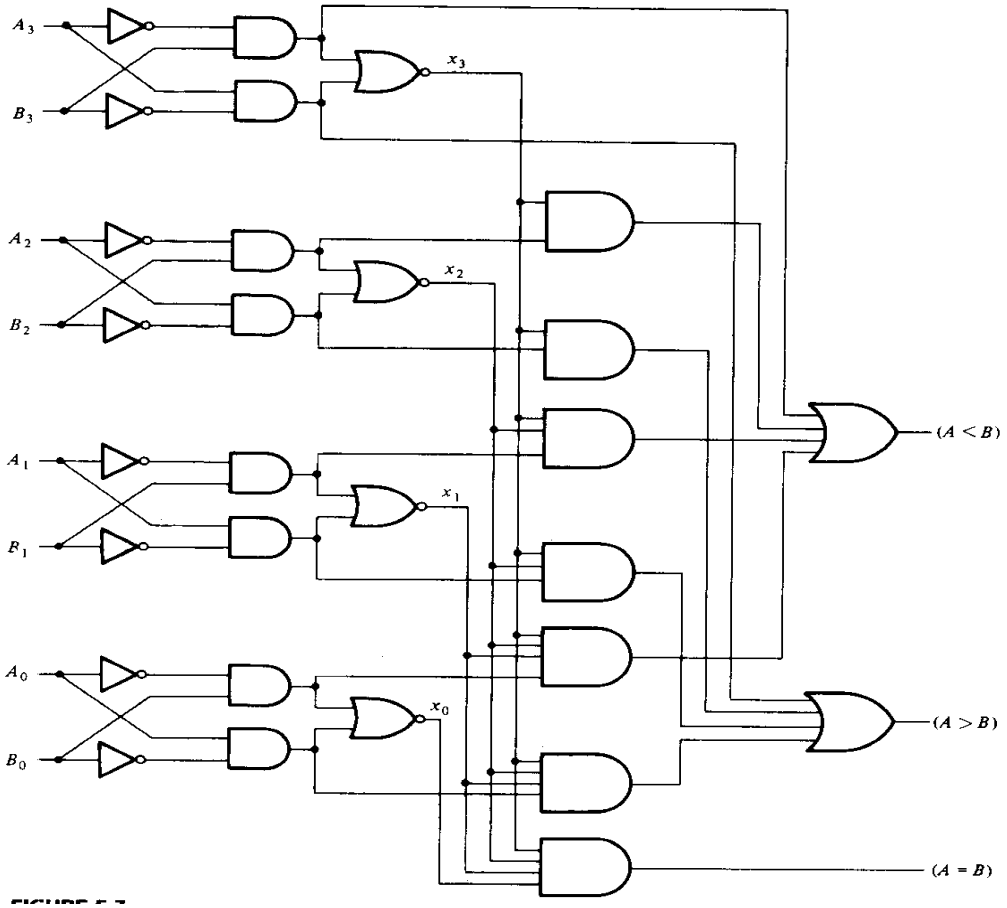


Figure: 4- bit Magnitude comparator

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.