Combinational Circuits

2

Given a digital electronic circuit specification and a set of available components, how can the designer translate this initial specification to a circuit? The answer is the central topic of this course. In this chapter, an answer is given in the particular case of the combinational circuits.

2.1 Definitions

A switching function is a binary function of binary variables. In other words, an n-variable switching function associates a binary value, 0 or 1, to any n-component binary vector. As an example, in Fig. 1.29, z_2, z_1, and z_0 are three 2-variable switching functions.

A digital circuit that implements a set of switching functions in such a way that at any time the output signal values only depend on the input signal values at the same moment is called a combinational circuit. The important point of this definition is "at the same moment." A combinational circuit with n inputs and m outputs is shown in Fig. 2.1. It implements m switching functions

$$f_i: \{0, 1\}^n \rightarrow \{0, 1\},\ i = 0, 1, \ldots, m-1.$$

To understand the condition "at the same moment" an example of circuit that is not combinational is now given.

Example 1.2 (A Non-combinational Circuit) Consider the temperature controller of Example 1.3 defined by Table 1.1 and substitute *ON* by 1 and *OFF* by 0. The obtained Table 2.1 does not define a combinational circuit: the knowledge that the current temperature is 20° does not permit to decide whether the output signal must be 0 or 1. To decide, it is necessary to know the previous value of the temperature. In other words, this circuit must have some kind of memory.

Example 2.2 Consider the 4-bit adder of Fig. 2.2. Input bits x_3, x_2, x_1, and x_0 represent an integer X in binary numeration (Appendix C); input bits y_3, y_2, y_1, and y_0 represent another integer Y, input bit c_i is an incoming carry; and output bits z_4, z_3, z_2, z_1, and z_0 represent an integer Z. The relation between inputs and outputs is

J.-P. Deschamps et al., *Digital Systems*, DOI 10.1007/978-3-319-41198-9_2

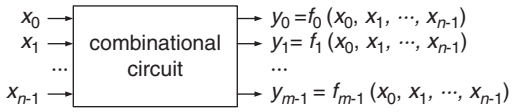
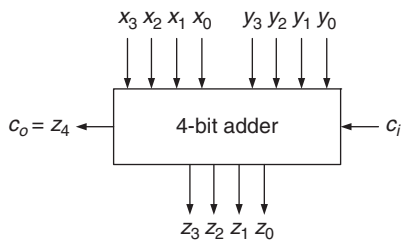


Fig. 2.1 n-Input m-output combinational circuit

Table 2.1 Specification of a non-combinational circuit

Temp	Onoff
0	1
1	1
...	...
18	1
19	1
20	*Unchanged*
21	0
22	0
...	...
49	0
50	0

Fig. 2.2 4-Bit adder



$$Z = X + Y + c_i.$$

Observe that X and Y are 4-bit integers included within the range of 0–15, so that the maximum value of Z is $15 + 15 + 1 = 31$ that is a 5-bit number. Output z_4 could also be used as an outgoing carry c_o.

In this example, the value of the output bits only depends on the value of the input bits at the same time; it is a combinational circuit.

2.2 Synthesis from a Table

A completely explicit specification of a 4-bit adder (Fig. 2.2) is a table that defines five switching functions z_4, z_3, z_2, z_1, and z_0 of nine variables x_3, x_2, x_1, x_0, y_3, y_2, y_1, y_0, and c_i (Table 2.2).

A straightforward implementation method consists in storing the Table 2.2 contents in a read-only memory (Fig. 2.3). The address bits are the input signals x_3, x_2, x_1, x_0, y_3, y_2, y_1, y_0, and c_i and the stored words define the value of the output signals z_4, z_3, z_2, z_1, and z_0. As an example, if the address bits are 100111001, so that $x_3x_2x_1x_0 = 1001$, $y_3y_2y_1y_0 = 1100$, and $c_i = 1$, then $X = 9$, $Y = 12$, and $Z = 9 + 12 + 1 = 22$, and the stored word is 10110 that is the binary representation of 22.

Obviously this is a universal synthesis method: it can be used to implement any combinational circuit. The generic circuit of Fig. 2.1 can be implemented by the ROM of Fig. 2.4. However, in many

Table 2.2 Explicit specification of a 4-bit adder

$x_3x_2x_1x_0$	$y_3y_2y_1y_0$	c_i	$z_4z_3z_2z_1z_0$
0000	0000	0	00000
0000	0000	1	00001
0000	0001	0	00001
0000	0001	1	00010
0000	0010	0	00010
0000	0010	1	00011
...	...	...	...
1001	1100	1	10110
...	...	...	...
1111	1111	0	11110
1111	1111	1	11111

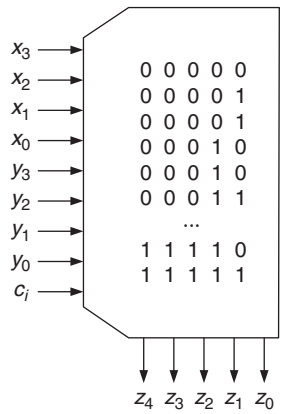


Fig. 2.3 ROM implementation of a 4-bit adder

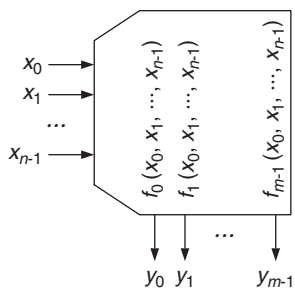


Fig. 2.4 ROM implementation of a combinational circuit

cases this is a very inefficient implementation method: the ROM of Fig. 2.4 must store $m \cdot 2^n$ bits, generally a (too) big number. As an example the ROM of Fig. 2.3 stores $5 \cdot 2^9 = 2{,}560$ bits.

Instead of using a universal, but inefficient, synthesis method, a better option is to take advantage of the peculiarities of the system under development. In the case of the preceding Example 2.2 (Fig. 2.2), a first step is to divide the 4-bit adder into four 1-bit adders (Fig. 2.5).

Each 1-bit adder is a combinational circuit that implements two switching functions z and d of three variables x, y, and c. Each 1-bit adder executes the operations that correspond to a particular step of the binary addition method (Appendix C). A completely explicit specification is given in Table 2.3.

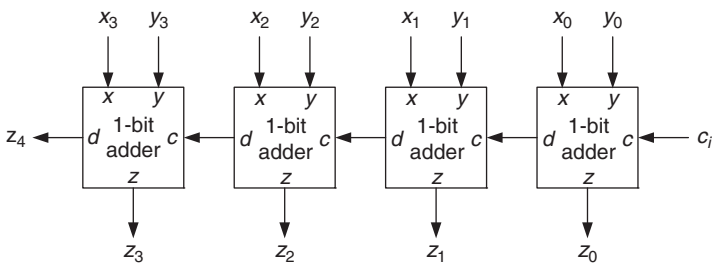


Fig. 2.5 Structure of a 4-bit adder

Table 2.3 Explicit specification of a 1-bit adder

x	y	c	d	z
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

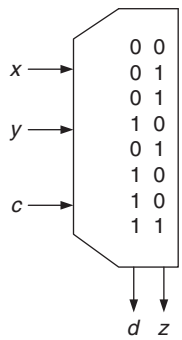


Fig. 2.6 ROM implementation of a 1-bit adder

In this case, a ROM implementation could be considered (Fig. 2.6). This type of small ROM (eight 2-bit words in this example) is often called lookup table (LUT) and it is the method used in field programmable gate arrays (FPGA) to implement switching functions of a few variables (Chap. 7).

Instead of a ROM, a table can also be implemented by means of logic gates (Sect. 1.3.2), for example AND gates, OR gates, and Inverters (or NOT gates). Remember that

- The output of an n-input AND gate is equal to 1 if, and only if, its n inputs are equal to 1.
- The output of an n-input OR gate is equal to 1 if, and only if, at least one of its n inputs is equal to 1.
- The output of an inverter is equal to 1 if, and only if, its input is equal to 0.

Define now a 3-input switching function $p(x, y, c)$ as follows: $p = 1$ if, and only if, $x = 1$, $y = 0$, and $c = 1$ (Table 2.4).

Table 2.4 Explicit specification of p

x	y	c	p
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

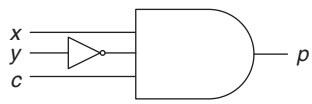


Fig. 2.7 Implementation of p

Table 2.5 Explicit specification of p_1, p_2, p_3, and p_4

x	y	c	p_1	p_2	p_3	p_4
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	1	0	0	0
1	0	0	0	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

This function p is implemented by the circuit of Fig. 2.7: the output of the AND3 gate is equal to 1 if, and only if, $x = 1$, $c = 1$ and the inverter output is equal to 1, that is, if $y = 0$.

The function d of Table 2.3 can be defined as follows: d is equal to 1 if, and only if, one of the following conditions is true:

$$x = 0, y = 1, c = 1,$$
$$x = 1, y = 0, c = 1,$$
$$x = 1, y = 1, c = 0,$$
$$x = 1, y = 1, c = 1.$$

The switching functions p_1, p_2, p_3, and p_4 can be associated to those conditions (Table 2.5). Actually, the function p of Table 2.4 is the function p_2 of Table 2.5.

Each function p_i can be implemented in the same way as p (Fig. 2.7) as shown in Fig. 2.8.

Finally, the function d can be defined as follows: d is equal to 1 if, and only if, one of the functions p_i is equal to 1. The corresponding circuit is a simple OR4 gate (Fig. 2.9) and the complete circuit is shown in Fig. 2.10.

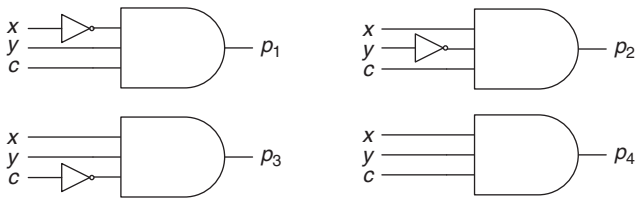


Fig. 2.8 Implementation of p_1, p_2, p_3, and p_4

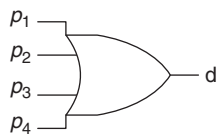


Fig. 2.9 Implementation of d

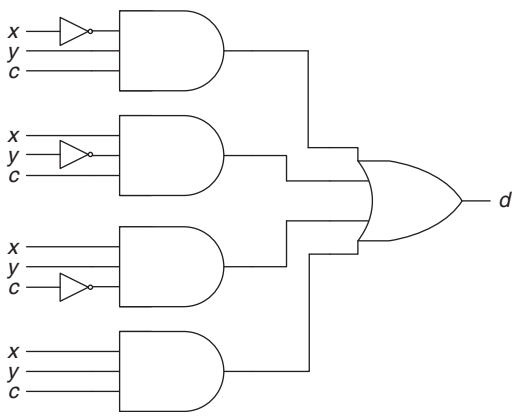


Fig. 2.10 Complete circuit

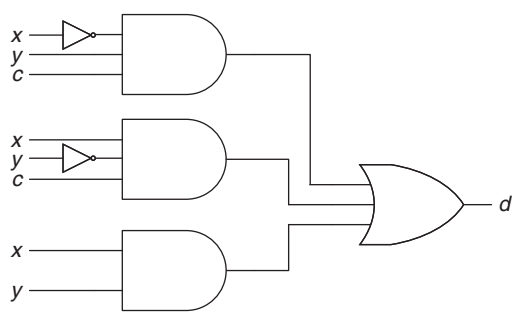


Fig. 2.11 Simplified circuit

Comment 2.1

The conditions implemented by functions p_3 and p_4 are $x = 1, y = 1, c = 0$ and $x = 1, y = 1, c = 1$, and can obviously be substituted by the simple condition $x = 1$ and $y = 1$, whatever the value of c. Thus, in Fig. 2.10 two of the four AND3 gates can be replaced by a single AND2 gate (Fig. 2.11).

The synthesis with logic gates of z (Table 2.3) is left as an exercise.

In conclusion, given a combinational circuit whose initial specification is a table, two possible options are:

- To store the table contents in a ROM
- To translate the table to a circuit made up of logic gates

Furthermore, in the second case, some optimization must be considered: if the inverters are not taken into account, the circuit of Fig. 2.11 contains 4 logic gates and 11 logic gate inputs, while the circuit of Fig. 2.10 contains 5 logic gates and 16 logic gate inputs. In CMOS technology (Sect. 1.3.2) the number of transistors is equal to twice the number of gate inputs so that the latter could be used as a measure of the circuit complexity.

A conclusion is that a tool that helps to minimize the number of gates and the number of gate inputs is necessary. It is the topic of the next section.

2.3 Boolean Algebra

Boolean algebra is a mathematical support used to specify and to implement switching functions. Only finite Boolean algebras are considered in this course.

2.3.1 Definition

A Boolean algebra B is a finite set over which two binary operations are defined:

- The Boolean sum $+$
- The Boolean product

Those operations must satisfy six rules (postulates).

The Boolean sum and the Boolean product are internal operations:

$$\forall a \text{ and } b \in B : a + b \in B \text{ and } a \cdot b \in B. \tag{2.1}$$

Actually, this postulate only emphasizes the fact that $+$ and $\cdot$ are operations over B.

The set B includes two particular (and different) elements 0 and 1 that satisfy the following conditions:

$$\forall a \in B : a + 0 = a \text{ and } a \cdot 1 = a. \tag{2.2}$$

In other words, 0 and 1 are neutral elements with respect to the sum (0) and with respect to the product (1).

Every element of B has an inverse in B:

$$\forall\, a \in B,\ \exists\, \overline{a} \in B \text{ such that } a + \overline{a} = 1 \text{ and } a \cdot \overline{a} = 0. \tag{2.3}$$

Both operations are commutative:

$$\forall a \text{ and } b \in B : a + b = b + a \text{ and } a \cdot b = b \cdot a. \tag{2.4}$$

Both operations are associative:

$$\forall a, b \text{ and } c \in B, a \cdot (b \cdot c) = (a \cdot b) \cdot c \text{ and } a + (b + c) = (a + b) + c. \tag{2.5}$$

The product is distributive over the sum and the sum is distributive over the product:

$$\forall a, b \text{ and } c \in B, a \cdot (b + c) = a \cdot b + a \cdot c \text{ and } a + b \cdot c = (a + b) \cdot (a + c). \tag{2.6}$$

Comment 2.2
Rules (2.1)–(2.6) constitute a set of symmetric postulates: given a rule, by interchanging sum and product, and 0 and 1, another rule is obtained: for example the fact that $a + 0 = a$ implies that $a \cdot 1 = a$, or the fact that $a \cdot (b + c) = a \cdot b + a \cdot c$ implies that $a + b \cdot c = (a + b) \cdot (a + c)$. This property is called duality principle.

The simplest example of Boolean algebra is the set $B_2 = \{0, 1\}$ with the following operations (Table 2.6):

- $a + b = 1$ if, and only if, $a = 1$ or $b = 1$, the OR function of a and b
- $a + b = 1$ if, and only if, $a = 1$ and $b = 1$, the AND function of a and b
- The inverse of a is $1 - a$

It can easily be checked that all postulates are satisfied. As an example (Table 2.7) check that the product is distributive over the sum, that is, $a \cdot (b + c) = a \cdot b + a \cdot c$.

The relation between B_2 and the logic gates defined in Sect. 1.3.2 is obvious: an AND gate implements a Boolean product, an OR gate implements a Boolean sum, and an inverter (NOT gate) implements an invert function (Fig. 2.12).

Table 2.6 Operations over B_2

a	b	$a + b$	$a \cdot b$	$\bar{a}$
0	0	0	0	1
0	1	1	0	1
1	0	1	0	0
1	1	1	1	0

Table 2.7 $a \cdot (b + c) = a \cdot b + a \cdot c$

a	b	c	$b + c$	$a \cdot (b + c)$	$a \cdot b$	$a \cdot c$	$a \cdot b + b \cdot c$
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

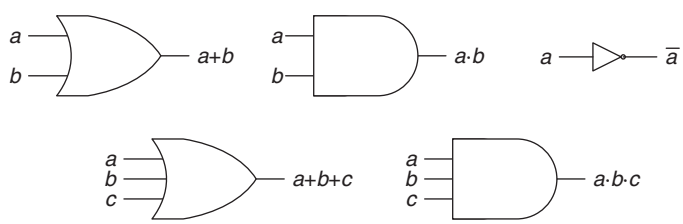


Fig. 2.12 Logic gates and Boolean functions

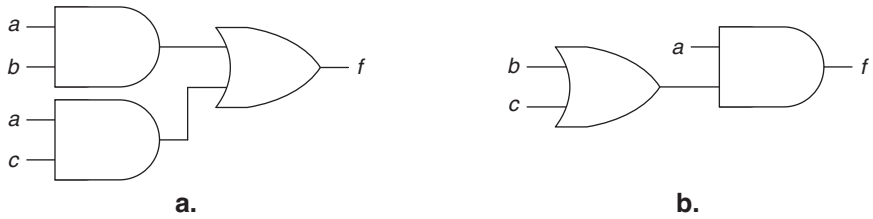


Fig. 2.13 Two equivalent circuits

In fact, there is a direct relation between Boolean expressions and circuits. As an example, a consequence of the distributive property $a \cdot b + a \cdot c = a \cdot (b + c)$ is that the circuits of Fig. 2.13 implement the same switching function, say f. However, the circuit that corresponds to the Boolean expression $a \cdot b + a \cdot c$ (Fig. 2.13a) has three gates and six gate inputs while the other (Fig. 2.13b) includes only two gates and three gate inputs. This is a first (and simple) example of how Boolean algebra helps the designer to optimize circuits.

Other finite Boolean algebras can be defined. Consider the set $B_2{}^n = \{0, 1\}^n$, that is, the set of all $2^n n$-component binary vectors. It is a Boolean algebra in which product, sum, and inversion are component-wise operations:

$$\begin{aligned} \forall a = (a_0, a_1, \ldots, a_{n-1}) \text{ and } b = (b_0, b_1, \ldots, b_{n-1}) \in B_2{}^n : \\ a + b &= (a_0 + b_0, a_1 + b_1, \ldots, a_{n-1} + b_{n-1}), \\ a \cdot b &= (a_0 \cdot b_0, a_1 \cdot b_1, \ldots, a_{n-1} \cdot b_{n-1}), \\ \overline{a} &= (\overline{a_0}, \overline{a_1}, \ldots, \overline{a_{n-1}}). \end{aligned}$$

The neutral elements are $0 = (0, 0, \ldots, 0)$ and $1 = (1, 1, \ldots, 1)$.

Another example is the set of all subsets of a finite set S. Given two subsets S_1 and S_2, their sum is $S_1 \cup S_2$ (union), their product is $S_1 \cap S_2$ (intersection), and the inverse of S_1 is $S \backslash S_1$ (complement of S_1 with respect to S). The neutral elements are the empty set Ø and S. If S has n elements the number of subsets of S is 2^n.

A third example, the most important within the context of this course, is the set of all n-variable switching functions. Given two switching functions f and g, functions $f + g, f \cdot g$, and $\overline{f}$ are defined as follows:

$$\begin{aligned} &\forall (x_0, x_1, \ldots, x_{n-1}) \in B_2{}^n : \\ &(f + g)(x_0, x_1, \ldots, x_{n-1}) = f(x_0, x_1, \ldots, x_{n-1}) + g(x_0, x_1, \ldots, x_{n-1}), \\ &(f \cdot g)(x_0, x_1, \ldots, x_{n-1}) = f(x_0, x_1, \ldots, x_{n-1}) \cdot g(x_0, x_1, \ldots, x_{n-1}), \\ &\overline{f}(x_0, x_1, \ldots, x_{n-1}) = \overline{f(x_0, x_1, \ldots, x_{n-1})}. \end{aligned}$$

The neutral elements are the constant functions 0 and 1.

Comment 2.3
Mathematicians have demonstrated that any finite Boolean algebra is isomorphic to $B_2{}^m$ for some $m > 0$. In particular, the number of elements of any finite Boolean algebra is a power of 2. Consider the previous examples.

1. The set of subsets of a finite set $S = \{s_1, s_2, \ldots, s_n\}$ is a Boolean algebra isomorphic to $B_2{}^n$: associate to every subset S_1 of S an n-component binary vector whose component i is equal to 1 if, and only if, $s_i \in S_1$, and check that the vectors that correspond to the union of two subsets, the

intersection of two subsets, and the complement of a subset are obtained by executing the component-wise addition, the component-wise product, and the component-wise inversion of the associated n-component binary vectors.

2. The set of all n-variable switching functions is a Boolean algebra isomorphic to ${B_2}^m$ with $m = 2^n$: associate a number i to each of the 2^n elements of $\{0, 1\}^n$ (for example the natural number represented in binary numeration by this vector); then, associate to any n-variable switching function f a 2^n-component vector whose component number i is the value of f at point i. In the case of functions d and z of Table 2.3, $n = 3$, $2^n = 8$, and the 8-component vectors that define d and z are (00010111) and (01101001), respectively.

2.3.2 Some Additional Properties

Apart from rules (2.1–2.6) several additional properties can be demonstrated and can be used to minimize Boolean expressions and to optimize the corresponding circuits.

Properties 2.1

1. $\overline{0} = 1$ and $\overline{1} = 0$. (2.7)

2. Idempotence : $\forall a \in B : a + a = a$ and $a \cdot a = a$. (2.8)

3. $\forall a \in B : a + 1 = 1$ and $a \cdot 0 = 0$. (2.9)

4. Inverse uniqueness : if $a \cdot b = 0, a + b = 1, a \cdot c = 0$ and $a + c = 1$ then $b = c$. (2.10)

5. Involution : $\forall a \in B : \overline{\overline{a}} = a$. (2.11)

6. Absorption law : $\forall a$ and $b \in B, a + a \cdot b = a$ and $a \cdot (a + b) = a$. (2.12)

7. $\forall a$ and $b \in B,\ a + \overline{a} \cdot b = a + b$ and $a \cdot (\overline{a} + b) = a \cdot b$. (2.13)

8. de Morgan laws : $\forall a$ and $b \in B, \overline{a + b} = \overline{a} \cdot \overline{b}$ and $\overline{a \cdot b} = \overline{a} + \overline{b}$. (2.14)

9. Generalized de Morgan laws : $\forall a_1, a_2, \ldots, a_n \in B$, (2.15)

$$\overline{a_1 + a_2 + \ \ldots\ + a_n} = \overline{a_1} \cdot \overline{a_2} \cdot \ \ldots\ \cdot \overline{a_n} \text{ and } \overline{a_1 \cdot a_2 \cdot \ \ldots\ \cdot a_n} = \overline{a_1} + \overline{a_2} + \ \ldots\ + \overline{a_n}.$$

Proof

1. $\overline{0} = 0 + \overline{0} = 1$ and $\overline{1} = 1 \cdot \overline{1} = 0$.

2. $a = a + 0 = a + (a \cdot \overline{a}) = (a + a) \cdot (a + \overline{a}) = (a + a) \cdot 1 = a + a$; $a = a \cdot 1 = a \cdot (a + \overline{a})$ $= (a \cdot a) + (a \cdot \overline{a}) = (a \cdot a) + 0 = a \cdot a$.

3. $a + 1 = a + a + \overline{a} = a + \overline{a} = 1; a \cdot 0 = a \cdot a \cdot \overline{a} = a \cdot \overline{a} = 0.$

4. $b = b \cdot (a + c) = a \cdot b + b \cdot c = 0 + b \cdot c = a \cdot c + b \cdot c = (a + b) \cdot c = 1 \cdot c = c.$

5. Direct consequence of (4).

6. $a + a \cdot b = a \cdot 1 + a \cdot b = a \cdot (1 + b) = a \cdot 1 = a; a \cdot (a + b) = a \cdot a + a \cdot b = a + a \cdot b = a.$

7. $a + \overline{a} \cdot b = (a + \overline{a}) \cdot (a + b) = 1 \cdot (a + b) = a + b; a \cdot (\overline{a} + b) = (a \cdot \overline{a}) + (a \cdot b) = 0 + (a \cdot b) = a \cdot b.$

8. $(a + b) \cdot \overline{a} \cdot \overline{b} = a \cdot \overline{a} \cdot \overline{b} + b \cdot \overline{a} \cdot \overline{b} = 0 \cdot \overline{b} + 0 \cdot \overline{a} = 0 + 0 = 0;$
$(a + b) + \overline{a} \cdot \overline{b} = a + \left(b + \overline{a} \cdot \overline{b}\right) = a + b + \overline{a} = b + 1 = 1.$

9. By induction.

2.3.3 Boolean Functions and Truth Tables

Tables such as Table 2.3 that defines two switching functions d and z are called truth tables. If f is an n-variable switching function then its truth table has 2^n rows, that is, the number of different n-component vectors.

In this section the relation between Boolean expressions, truth tables, and gate implementation of combinational circuits is analyzed.

Given a Boolean expression, that is a well-constructed expression using variables and Boolean operations (sum, product, and inversion), a truth table can be defined. For that the value of the expression must be computed for every combination of variable values, in total 2^n different combinations if there are n variables.

Example 2.3 Consider the following Boolean expression that defines a 3-variable switching function f:

$$f(a, b, c) = b \cdot \overline{c} + \overline{a} \cdot b.$$

Define a table with as many rows as the number of combinations of values of a, b, and c, that is, $2^3 = 8$ rows, and compute the value of f that corresponds to each of them (Table 2.8).

Table 2.8 $f(a, b, c) = b \cdot \overline{c} + \overline{a} \cdot b.$

abc	$\overline{c}$	$b \cdot \overline{c}$	$\bar{a}$	$\overline{a} \cdot b$	$f = b \cdot \overline{c} + \overline{a} \cdot b$
000	1	0	1	0	0
001	0	0	1	0	0
010	1	1	1	1	1
011	0	0	1	1	1
100	1	0	0	0	0
101	0	0	0	0	0
110	1	1	0	0	1
111	0	0	0	0	0

Conversely, a Boolean expression can be associated to any truth table. For that, first define some new concepts.

Definitions 2.1

1. A literal is a variable or the inverse of a variable. For example a, $\overline{a}$, b, $\overline{b}$, ... are literals.
2. An n-variable minterm is a product of n literals such that each variable appears only once.

For example, if $n = 3$ then there are eight different minterms:

$$\begin{aligned} &m_0 = \overline{a} \cdot \overline{b} \cdot \overline{c}, m_1 = \overline{a} \cdot \overline{b} \cdot c, m_2 = \overline{a} \cdot b \cdot \overline{c},\ m_3 = \overline{a} \cdot b \cdot c, \\ &m_4 = a \cdot \overline{b} \cdot \overline{c}, m_5 = a \cdot \overline{b} \cdot c, m_6 = a \cdot b \cdot \overline{c}, m_7 = a \cdot b \cdot c. \end{aligned} \tag{2.16}$$

Their corresponding truth tables are shown in Table 2.9. Their main property is that to each minterm m_i is associated one, and only one, combination of values of a, b, and c such that $m_i = 1$:

m_0 is equal to 1 if. and only if, $abc = 000$,
m_1 is equal to 1 if. and only if, $abc = 001$,
m_2 is equal to 1 if. and only if, $abc = 010$,
m_3 is equal to 1 if. and only if, $abc = 011$,
m_4 is equal to 1 if. and only if, $abc = 100$,
m_5 is equal to 1 if. and only if, $abc = 101$,
m_6 is equal to 1 if. and only if, $abc = 110$,
m_7 is equal to 1 if. and only if, $abc = 111$.

In other words, $m_i = 1$ if, and only if, abc is equal to the binary representation of i.

Consider now a 3-variable function f defined by its truth table (Table 2.10). From Table 2.9 it can be deduced that $f = m_2 + m_3 + m_6$, and thus (2.16)

$$f = \overline{a} \cdot b \cdot \overline{c} + \overline{a} \cdot b \cdot c + a \cdot b \cdot \overline{c}. \tag{2.17}$$

More generally, the n-variable minterm $m_i(x_{n-1}, x_{n-2}, \ldots, x_0)$ is equal to 1 if, and only if, the value of $x_{n-1}x_{n-2} \ldots x_0$ is the binary representation of i. Given a truth table that defines an n-variable switching function $f(x_{n-1}, x_{n-2}, \ldots, x_0)$, this function is the sum of all minterms m_i such that $i_{n-1}i_{n-2} \ldots i_0$ is the binary representation of i and $f(i_{n-1}, i_{n-2}, \ldots, i_0) = 1$. This type of representation of a switching function under the form of a Boolean sum of minterms (like (2.17)) is called canonical representation.

Table 2.9 3-Variable minterms

abc	m_0	m_1	m_2	m_3	m_4	m_5	m_6	m_7
000	1	0	0	0	0	0	0	0
001	0	1	0	0	0	0	0	0
010	0	0	1	0	0	0	0	0
011	0	0	0	1	0	0	0	0
100	0	0	0	0	1	0	0	0
101	0	0	0	0	0	1	0	0
110	0	0	0	0	0	0	1	0
111	0	0	0	0	0	0	0	1

Table 2.10 Truth table of f

abc	f
000	0
001	0
010	1
011	1
100	0
101	0
110	1
111	0

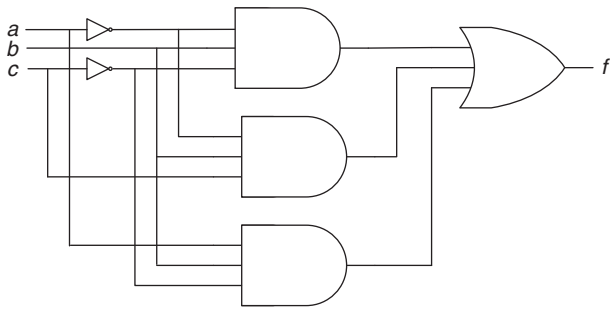


Fig. 2.14 $f = \overline{a} \cdot b \cdot \overline{c} + \overline{a} \cdot b \cdot c + a \cdot b \cdot \overline{c}$

The relation between truth table and Boolean expression, namely canonical representation, has been established. From a Boolean expression, for example (2.17), a circuit made up of logical gates can be deduced (Fig. 2.14).

Another example: the functions p_1, p_2, p_3, and p_4 of Table 2.5 are minterms of variables x, y, and c, and the circuit of Fig. 2.10 corresponds to the canonical representation of d.

Assume that a combinational system has been specified by some functional description, for example an algorithm (an implicit functional description). The following steps permit to generate a logic circuit that implements the function.

- Translate the algorithm to a table (an explicit functional description); for that, execute the algorithm for all combinations of the input variable values.
- Generate the canonical representation that corresponds to the table.
- Optimize the expression using properties of the Boolean algebras.
- Generate the corresponding circuit made up of logic gates.

As an example, consider the following algorithm that defines a 3-variable switching function.

Algorithm 2.1 Specification of $f(a, b, c)$

```
if (a=1 and b=1 and c=0) or (a=0 and b=1) then f = 1;
else f = 0;
end if;
```

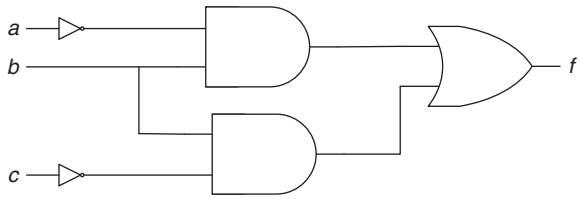


Fig. 2.15 Optimized circuit

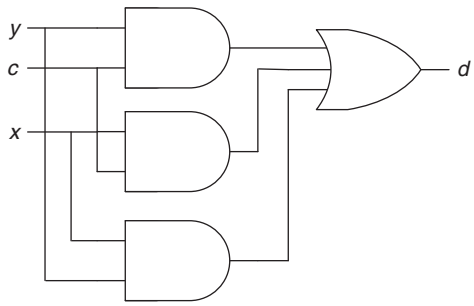


Fig. 2.16 Implementation of (2.20)

By executing this algorithm for each of the eight combinations of values of a, b, and c, Table 2.10 is obtained. The corresponding canonical expression is (2.17). This expression can be simplified using Boolean algebra properties:

$$\overline{a} \cdot b \cdot \overline{c} + \overline{a} \cdot b \cdot c + a \cdot b \cdot \overline{c} = \overline{a} \cdot b \cdot (\overline{c} + c) + (\overline{a} + a) \cdot b \cdot \overline{c} = \overline{a} \cdot b + b \cdot \overline{c}.$$

The corresponding circuit is shown in Fig. 2.15. It implements the same function as the circuit of Fig. 2.14, with fewer gates and fewer gate inputs. This is an example of the kind of circuit optimization that Boolean algebras permit to execute.

2.3.4 Example

The 4-bit adder of Sect. 2.2 is now revisited and completed. A first step is to divide the 4-bit adder into four 1-bit adders (Fig. 2.5). Each 1-bit adder implements two switching functions d and z defined by their truth tables (Table 2.3). The canonical expressions that correspond to the truth tables of d and z are the following:

$$d = \overline{x} \cdot y \cdot c + x \cdot \overline{y} \cdot c + x \cdot y \cdot \overline{c} + x \cdot y \cdot c, \tag{2.18}$$

$$z = \overline{x} \cdot \overline{y} \cdot c + \overline{x} \cdot y \cdot \overline{c} + x \cdot \overline{y} \cdot \overline{c} + x \cdot y \cdot c. \tag{2.19}$$

The next step is to optimize the Boolean expressions. Equation 2.18 can be optimized as follows:

$$d = (\overline{x} + x) \cdot y \cdot c + x \cdot (y + \overline{y}) \cdot c + x \cdot y \cdot (c + \overline{c}) = y \cdot c + x \cdot c + x \cdot y. \tag{2.20}$$

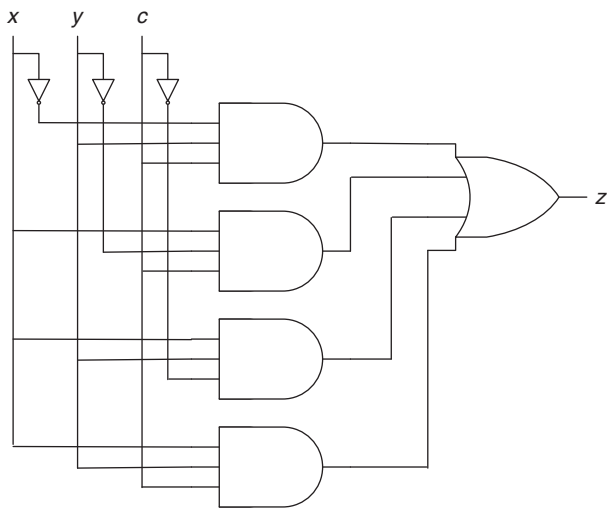


Fig. 2.17 Implementation of (2.19)

The corresponding circuit is shown in Fig. 2.16. It implements the same function d as the circuit of Fig. 2.11, with fewer gates, fewer gate inputs, and without inverters.

Equation 2.19 cannot be simplified. The corresponding circuit is shown in Fig. 2.17.

2.4 Logic Gates

In Sects. 2.2 and 2.3 a first approach to the implementation of switching functions has been proposed. It is based on the translation of the initial specification to Boolean expressions. Then, circuits made up of AND gates, OR gates, and inverters can easily be defined. However, there exist other components (Sect. 1.3.2) that can be considered to implement switching functions.

2.4.1 NAND and NOR

NAND gates and NOR gates have been defined in Sect. 1.3.2. They can be considered as simple extensions of the CMOS inverter and are relatively easy to implement in CMOS technology.

A NAND gate is equivalent to an AND gate and an inverter, and a NOR gate is equivalent to an OR gate and an inverter (Fig. 2.18).

The truth tables of a 2-input NAND function and of a 2-input NOR function are shown in Figs. 1.21a and 1.22b, respectively. More generally, the output of a k-input NAND gate is equal to 0 if, and only if, the k inputs are equal to 1, and the output of a k-input NOR gate is equal to 1 if, and only if, the k inputs are equal to 0. Thus,

$$\mathrm{NAND}(x_1, x_2, \ldots, x_n) = \overline{x_1 \cdot x_2 \cdot \ \ldots \cdot x_n} = \overline{x_1} + \overline{x_2} + \ldots + \overline{x_n}\,, \tag{2.21}$$

$$\mathrm{NOR}(x_1, x_2, \ldots, x_n) = \overline{x_1 + x_2 + \ldots + x_n} = \overline{x_1} \cdot \overline{x_2} \cdot \ldots \cdot \overline{x_n}\,. \tag{2.22}$$

Sometimes, the following algebraic symbols are used:

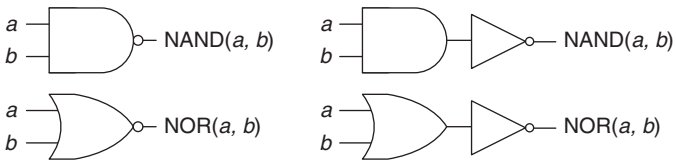


Fig. 2.18 NAND2 and NOR2 symbols and equivalent circuits

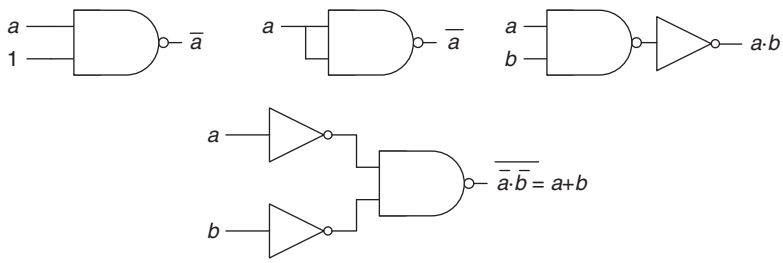


Fig. 2.19 NOT, AND2, and OR2 gates implemented with NAND2 gates and inverters

$$a \uparrow b = \text{NAND}(a, b) \text{ and } a \downarrow b = \text{NOR}(a, b).$$

NAND and NOR gates are universal modules. That means that any switching function can be implemented only with NAND gates or only with NOR gates. It has been seen in Sect. 2.3 that any switching function can be implemented with AND gates, OR gate, and inverters (NOT gates). To demonstrate that NAND gates are universal modules, it is sufficient to observe that the AND function, the OR function, and the inversion can be implemented with NAND functions. According to (2.21)

$$x_1 \cdot x_2 \cdot \ldots \cdot x_n = \overline{\overline{x_1 \cdot x_2 \cdot \ldots \cdot x_n}} = \overline{\text{NAND}(x_1, x_2, \ldots, x_n)}, \tag{2.23}$$

$$x_1 + x_2 + \ldots + x_n = \text{NAND}(\overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}), \tag{2.24}$$

$$\overline{x} = \overline{x \cdot 1} = \text{NAND}(x, 1) = \overline{x \cdot x} = \text{NAND}(x, x). \tag{2.25}$$

As an example NOT, AND2, and NOR2 gates implemented with NAND2 gates are shown in Fig. 2.19.

Similarly, to demonstrate that NOR gates are universal modules, it is sufficient to observe that the AND function, the OR function, and the inversion can be implemented with NOR functions. According to (2.22)

$$x_1 + x_2 + \ldots + x_n = \overline{\overline{x_1 + x_2 + \ldots + x_n}} = \overline{\text{NOR}(x_1, x_2, \ldots, x_n)}, \tag{2.26}$$

$$x_1 \cdot x_2 \cdot \ldots \cdot x_n = \text{NOR}(\overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}), \tag{2.27}$$

$$\overline{x} = \overline{x + 0} = \text{NOR}(x, 0) = \overline{x + x} = \text{NOR}(x, x). \tag{2.28}$$

Example 2.4 Consider the circuit of Fig. 2.11. According to (2.23) and (2.24), the AND gates and the OR gate can be substituted by NAND gates. The result is shown in Fig. 2.20a. Furthermore, two serially connected inverters can be substituted by a simple connection (Fig. 2.20b).

Comments 2.4

1. Neither the 2-variable NAND function (NAND2) nor the 2-variable NOR function (NOR2) are associative operations. For example

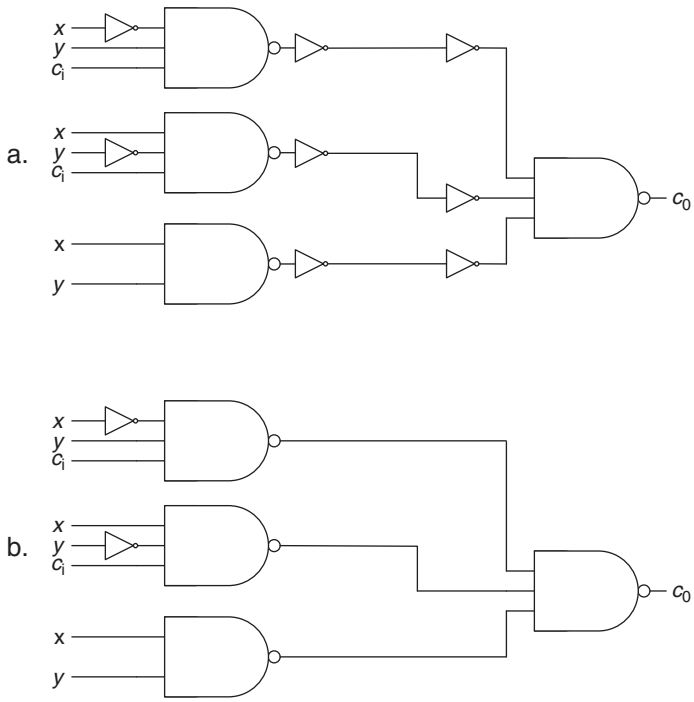


Fig. 2.20 Circuits equivalent to Fig. 2.11

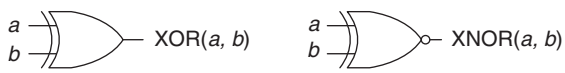


Fig. 2.21 XOR gate and XNOR gate symbols

$$\text{NAND}(a, \text{NAND}(b, c)) = \overline{a} + \overline{\text{NAND}(b, c)} = \overline{a} + b \cdot c,$$
$$\text{NAND}(\text{NAND}(a, b), c) = a \cdot b + \overline{c},$$

and none of the previous functions is equal to $NAND(a, b, c) = \overline{a} + \overline{b} + \overline{c}$.

2. As already mentioned above, NAND gates and NOR gates are easy to implement in CMOS technology. On the contrary, AND gates and OR gates must be implemented by connecting a NAND gate and an inverter or a NOR gate and an inverter, respectively. Thus, within a CMOS integrated circuit, NAND gates and NOR gates use less silicon area than AND gates and OR gates.

2.4.2 XOR and XNOR

XOR gates, where XOR stands for eXclusive OR, and XNOR gates are other commonly used components, especially in arithmetic circuits.

The 2-variable XOR switching function is defined as follows:

Table 2.11 XOR and XNOR truth tables

$a\ b$	XOR(a, b)	XNOR(a, b)
00	0	1
01	1	0
10	1	0
11	0	1

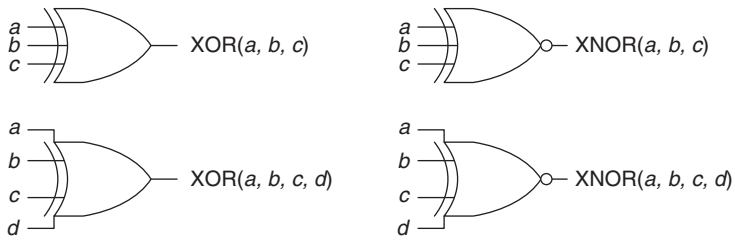


Fig. 2.22 3-Input and 4-input XOR gates and XNOR gates

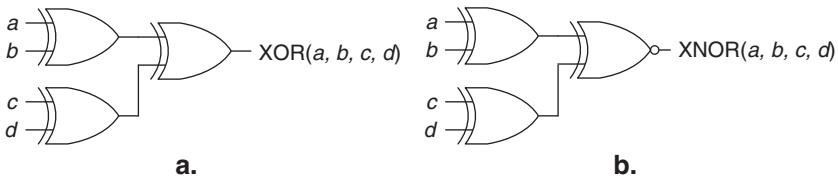


Fig. 2.23 4-Input XOR and XNOR gates implemented with 2-input gates

$$\text{XOR}(a,b) = 1 \text{ if, and only if}, a \neq b,$$

and the 2-variable XNOR switching function is the inverse of the XOR function, so that

$$\text{XNOR}(a,b) = 1 \text{ if, and only if}, a = b.$$

Their symbols are shown in Fig. 2.21 and their truth tables are defined in Table 2.11.
The following algebraic symbols are used:

$$a \oplus b = \text{XOR}(a,b), a \equiv b = \text{XNOR}(a,b).$$

An equivalent definition of the XOR function is

$$\text{XOR}(a,b) = (a+b)\text{mod}2 = a \oplus b.$$

With this equivalent definition an n-variable XOR switching function can be defined for any $n > 2$:

$$\text{XOR}(a_1, a_2, \ldots, a_n) = (a_1 + a_2 + \ldots + a_n)\text{mod}2 = a_1 \oplus a_2 \oplus \ldots \oplus a_n,$$

and the n-variable XNOR switching function is the inverse of the XOR function:

$$\text{XNOR}(a_1, a_2, \ldots, a_n) = \overline{\text{XOR}(a_1, a_2, \ldots, a_n)}.$$

Examples of XOR gate and XNOR gate symbols are shown in Fig. 2.22.

Mod 2 sum is an associative operation, so that n-input XOR gates can be implemented with 2-input XOR gates. As an example, in Fig. 2.23a a 4-input XOR gate is implemented with three 2-input XOR gates.

An n-input XNOR gate is implemented by the same circuit as an n-input XOR gate in which the XOR gate that generates the output is substituted by an XNOR gate. In Fig. 2.23b a 4-input XNOR gate is implemented with two 2-input XOR gates and a 2-input XNOR gate.

XOR gates and XNOR gates are not universal modules. However they are very useful to implement arithmetic functions.

Example 2.5 As a first example consider a 4-bit magnitude comparator: given two 4-bit numbers $a = a_3a_2a_1a_0$ and $b = b_3b_2b_1b_0$ generate a switching function *comp* equal to 1 if, and only if, $a = b$. The following trivial algorithm is used:

```
if (a3 = b3) and (a2 = b2) and (a1 = b1) and (a0 = b0)
then comp = 1;
else comp = 0;
end if;
```

The corresponding circuit is shown in Fig. 2.24: $comp = 1$ if, and only if, the four inputs of the NOR4 gate are equal to 0, that is, if $a_i = b_i$ and thus XOR(a_i, b_i) = 0, $\forall i = 0$, 1, 2, and 3.

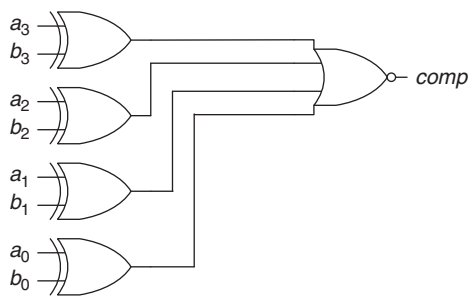


Fig. 2.24 4-Bit magnitude comparator

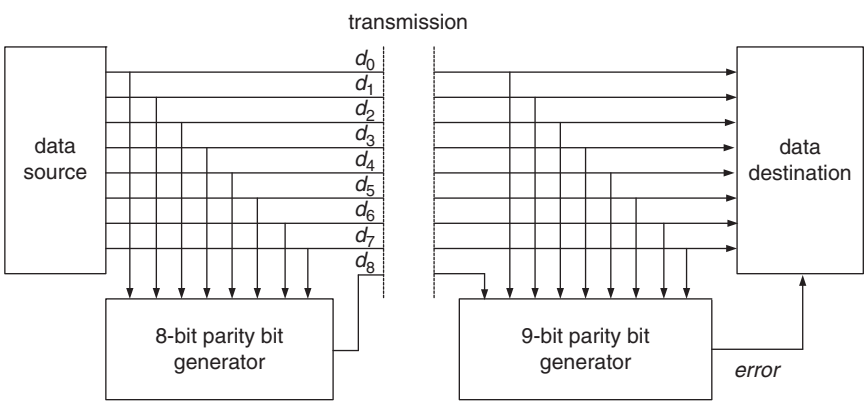


Fig. 2.25 Transmission of 8-bit data

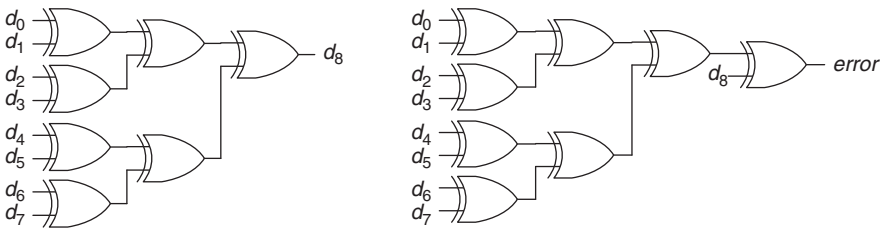
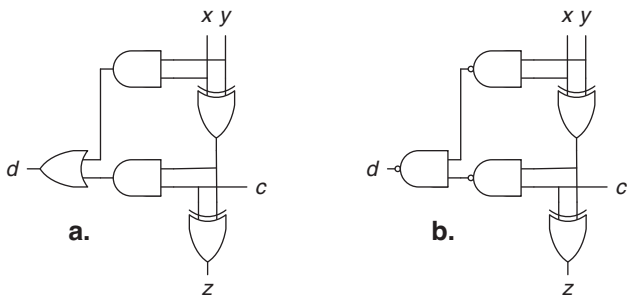


Fig. 2.26 Parity bit generation and parity check

Fig. 2.27 1-Bit adder



Example 2.6 The second example is a parity bit generator. It implements an n-variable switching function $parity(a_0, a_1, \ldots, a_{n-1}) = 1$ if, and only if, there is an odd number of 1s among variables a_0, $a_1, \ldots, a_{n-1}$. In other words,

$$parity(a_0, a_1, \ldots, a_{n-1}) = (a_0 + a_1 + \ldots + a_{n-1}) \bmod 2 = a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1}.$$

Consider a communication system (Fig. 2.25) that must transmit 8-bit data $d = d_0 d_1 \ldots d_7$ from a data source circuit to a data destination circuit. On the source size, an 8-bit parity generator generates an additional bit $d_8 = d_0\ d_1\ \ldots\ d_7$, and the nine bits $d_0 d_1 \ldots d_7 d_8$ are transmitted. Thus, the number of 1s among the transmitted bits d_0, d_1, …, d_8 is always even. On the destination side, a 9-bit parity generator checks whether the number of 1s among d_0, d_1, …, d_8 is even, or not. If even, the parity generator output is equal to 0; if odd, the output is equal to 1. If it is assumed that during the transmission at most one bit could have been modified, due to the noise on the transmission lines, the 9-bit parity generator output is an *error* signal equal to 0 if no error has happened and equal to 1 in the contrary case.

An 8-bit parity generator and a 9-bit parity generator implemented with XOR2 gates are shown in Fig. 2.26.

Example 2.7 The most common use of XOR gates is within adders. A 1-bit adder implements two switching functions z and d defined by Table 2.3 and by (2.19) and (2.20). According to Table 2.3, z can also be expressed as follows:

$$z = (x + y + c) \bmod 2 = x \oplus y \oplus c. \tag{2.29}$$

On the other hand, d is equal to 1 if, and only if, $x + y + c \geq 2$. This condition can be expressed in the following way: either $x = y = 1$ or $c = 1$ and $x \neq y$. The corresponding Boolean expression is

Fig. 2.28 Tristate buffer and tristate inverter symbols

Table 2.12 Definition of tristate buffer and tristate inverter

$c\ x$	3-State buffer output y	3-State inverter output y
0 0	Z	Z
0 1	Z	Z
1 0	0	1
1 1	1	0

Fig. 2.29 Symbols of tristate components with active-low control input

Table 2.13 Definition of tristate components with active-low control input

$c\ x$	3-State buffer output y	3-State inverter output y
0 0	0	1
0 1	1	0
1 0	Z	Z
1 1	Z	Z

$$d = x \cdot y + c \cdot (x \oplus y). \tag{2.30}$$

The circuit that corresponds to (2.29) and (2.30) is shown in Fig. 2.27a. As mentioned above (Fig. 2.20), AND gates and OR gates can be implemented with NAND gates (Fig. 2.27b).

2.4.3 Tristate Buffers and Tristate Inverters

Tristate buffers and tristate inverters are components whose output can be in three different states: 0 (low voltage), 1 (high voltage), or Z (disconnected). A tristate buffer CMOS implementation is shown in Fig. 1.28a: when the control input $c = 0$, the output is disconnected from the input, so that the output impedance is very high (infinite if leakage currents are not considered); if $c = 1$, the output is connected to the input through a CMOS switch.

A tristate inverter is equivalent to an inverter whose output is connected to a tristate buffer. It works as follows: when the control input $c = 0$, the output is disconnected from the input; if $c = 1$, the output is equal to the inverse of the input.

The symbols of a tristate buffer and of a tristate inverter are shown in Fig. 2.28 and their working is defined in Table 2.12.

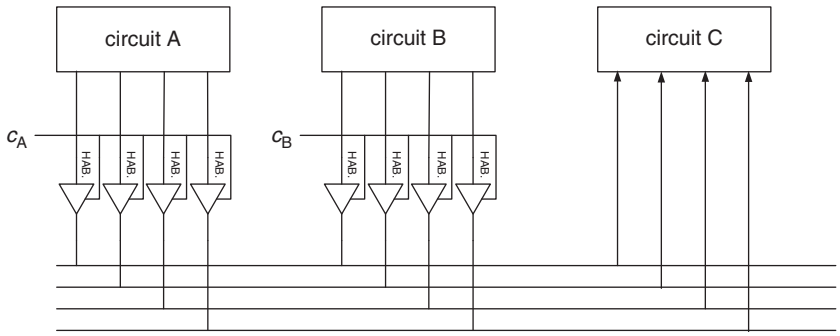


Fig. 2.30 4-Bit bus

Table 2.14 4-Bit bus definition

$c_A c_B$	Data transmission
0 0	None
0 1	B → C
1 0	A → C
1 1	Not allowed

In some tristate components the control signal c is active at low level. The corresponding symbols and definitions are shown in Fig. 2.29 and Table 2.13.

A typical application of tristate components is shown in Fig. 2.30. It is a 4-bit bus that permits to send 4-bit data either from circuit A to circuit C or from circuit B to circuit C. As an example, A could be a memory, B an input interface, and C a processor. Both circuits A and B must be able to send data to C but cannot be directly connected to C. To avoid collisions, 3-state buffers are inserted between A and B outputs and the set of wires connected to the circuit C inputs. To transmit data from A to C, $c_A = 1$ and $c_B = 0$, and to transmit data from B to C, $c_A = 0$ and $c_B = 1$ (Table 2.14).

2.5 Synthesis Tools

In order to efficiently implement combinational circuits, synthesis tools are necessary. In this section, some of the principles used to optimize combinational circuits are described.

2.5.1 Redundant Terms

When defining a switching function it might be that for some combinations of input variable values the corresponding output value is not defined because either those input value combinations never happen or because the function value does not matter. In the truth table, the corresponding entries are named “don’t care” (instead of 0 or 1). When defining a Boolean expression that describes the switching function to be implemented, the minterms that correspond to those don’t care entries can be used, or not, in order to optimize the final circuit.

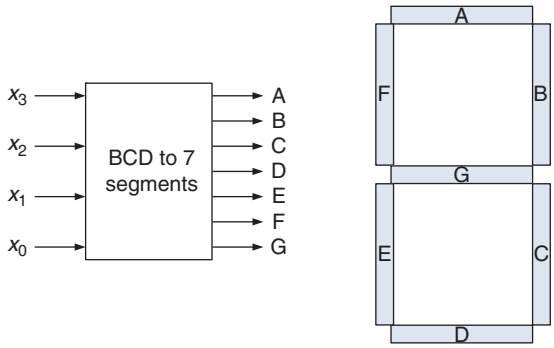


Fig. 2.31 BCD to 7-segment decoder

Table 2.15 BCD to 7-segment decoder definition

Digit	$x_3x_2x_1x_0$	A	B	C	D	E	F	G
0	0000	1	1	1	1	1	1	0
1	0001	0	1	1	0	0	0	0
2	0010	1	1	0	1	1	0	1
3	0011	1	1	1	1	0	0	1
4	0100	0	1	1	0	0	1	1
5	0101	1	0	1	1	0	1	1
6	0110	1	0	1	1	1	1	1
7	0111	1	1	1	0	0	0	0
8	1000	1	1	1	1	1	1	1
9	1001	1	1	1	0	0	1	1
	1010	–	–	–	–	–	–	–
	1011	–	–	–	–	–	–	–
	1100	–	–	–	–	–	–	–
	1101	–	–	–	–	–	–	–
	1110	–	–	–	–	–	–	–
	1111	–	–	–	–	–	–	–

Example 2.8 A BCD to 7-segment decoder (Fig. 2.31) is a combinational circuit with four inputs x_3, x_2, x_1, and x_0 that are the binary representation of a decimal digit (BCD means binary coded decimal) and seven outputs that control the seven segments of a display.

Among the 16 combinations of x_3, x_2, x_1, and x_0 values, only 10 are used: those that correspond to digits 0–9. Thus, the values of outputs A to G that correspond to inputs 1010 to 1111 are unspecified (don't care). The BCD to 7-segment decoder is defined by Table 2.15.

If all don't care entries are substituted by 0s, the following set of Boolean expressions is obtained:

$$\mathrm{A} = \overline{x}_3.x_1 + \overline{x}_3.x_2.x_0 + x_3.\overline{x}_2.\overline{x}_1, \tag{2.31a}$$

$$\mathrm{B} = \overline{x}_3.\overline{x}_2 + \overline{x}_2.\overline{x}_1 + \overline{x}_3.\overline{x}_1.\overline{x}_0 + \overline{x}_3.x_1.x_0, \tag{2.31b}$$

$$\mathrm{C} = \overline{x}_2.\overline{x}_1 + \overline{x}_3.x_0 + \overline{x}_3.\,x_2, \tag{2.31c}$$

$$\mathrm{D} = \overline{x}_2.\,\overline{x}_1.\overline{x}_0 + \overline{x}_3.\overline{x}_2.x_1 + \overline{x}_3.x_1.\overline{x}_0 + \,\overline{x}_3.x_2.\overline{x}_1.x_0, \tag{2.31d}$$

Table 2.16 Another definition of function B

$x_3x_2x_1x_0$	B
0000	1
0001	1
0010	1
0011	1
0100	1
0101	0
0110	0
0111	1
1000	1
1001	1
1010	1
1011	1
1100	1
1101	0
1110	0
1111	1

$$\mathrm{E} = \overline{x}_2.\overline{x}_1.\overline{x}_0 + \overline{x}_3.x_1.\overline{x}_0, \tag{2.31e}$$

$$\mathrm{F} = \overline{x}_3.\overline{x}_1.\overline{x}_0 + \overline{x}_3.x_2.\overline{x}_1 + \overline{x}_3.x_2.\overline{x}_0 + x_3.\overline{x}_2.\overline{x}_1, \tag{2.31f}$$

$$\mathrm{G} = \overline{x}_3.\overline{x}_2.x_1 + \overline{x}_3.x_2.\overline{x}_1 + \overline{x}_3.x_2.\overline{x}_0 + x_3.\overline{x}_2.\overline{x}_1. \tag{2.31g}$$

For example, B can be expressed as the sum of minterms m_0, m_1, m_2, m_3, m_4, m_7, m_8, and m_9:

$$\begin{aligned}\mathrm{B} \;= &\overline{x_3}.\overline{x_2}.\overline{x_1}.\overline{x_0} + \overline{x_3}.\overline{x_2}.\overline{x_1}.x_0 + \overline{x_3}.\overline{x_2}.x_1.\overline{x_0} + \overline{x_3}.\overline{x_2}.x_1.x_0 + \\ &\overline{x_3}.x_2.\overline{x_1}.\overline{x_0} + \overline{x_3}.x_2.x_1.x_0 + x_3.\overline{x_2}.\overline{x_1}.\overline{x_0} + x_3.\overline{x_2}.\overline{x_1}.x_0.\end{aligned}$$

Then, the previous expression can be minimized:

$$\begin{aligned}\mathrm{B} &= \overline{x_3}.\overline{x_2}.(\overline{x_1}.\overline{x_0} + \overline{x_1}.x_0 + x_1.\overline{x_0} + \overline{x_1}.\overline{x_0}) + \overline{x_3}.x_2.\overline{x_1}.\overline{x_0} + \\ &\quad \overline{x_3}.x_2.x_1.x_0 + x_3.\overline{x_2}.\overline{x_1}.(\overline{x_0} + x_0) \\ &= \overline{x_3}.\overline{x_2} + \overline{x_3}.x_2.\overline{x_1}.\overline{x_0} + \overline{x_3}.x_2.x_1.x_0 + x_3.\overline{x_2}.\overline{x_1} \\ &= \overline{x_3}.\overline{x_2} + \overline{x_3}.\overline{x_2}.\overline{x_1}.\overline{x_0} + \overline{x_3}.\overline{x_2}.x_1.x_0 + \overline{x_3}.\overline{x_2}.\overline{x_1} + \overline{x_3}.x_2.\overline{x_1}.\overline{x_0} + \\ &\quad \overline{x_3}.x_2.x_1.x_0 + x_3.\overline{x_2}.\overline{x_1} \\ &= \overline{x_3}.\overline{x_2} + \overline{x_3}.(\overline{x_2} + x_2).\overline{x_1}.\overline{x_0} + \overline{x_3}.(\overline{x_2} + x_2).x_1.x_0 + (\overline{x_3} + x_3).\overline{x_2}.\overline{x_1} \\ &= \overline{x_3}.\overline{x_2} + \overline{x_3}.\overline{x_1}.\overline{x_0} + \overline{x_3}.x_1.x_0 + \overline{x_2}.\overline{x_1}.\end{aligned} \tag{2.32}$$

By performing the same type of optimization for all other functions, the set of (2.31) has been obtained.

In Table 2.16 the don't care entries of function B have been defined in another way and a different Boolean expression is obtained: according to Table 2.16, B = 1 if, and only if, $x_2 = 0$ or $x_1x_0 = 00$ or 11; thus

$$\mathrm{B} = \overline{x}_2 + \overline{x}_1.\overline{x}_0 + x_1.x_0. \tag{2.33}$$

Equations 2.32 and 2.33 are compatible with the initial specification (Table 2.15). They generate different values of B when $x_3x_2x_1x_0 = 1010$, 1011, 1100, or 1111, but in those cases the value of B

Table 2.17 Comparison between (2.31) and (2.34)

Gate type	Number of gates (2.31)	Number of gates (2.34)
AND2	6	14
AND3	17	1
AND4	1	-
OR2	1	1
OR3	2	2
OR4	4	4
NOT	4	4

does not matter. On the other hand (2.33) is simpler than (2.32) and would correspond to a better implementation.

By performing the same type of optimization for all other functions, the following set of expressions has been obtained:

$$\mathrm{A} = x_1 + x_2.x_0 + x_3, \tag{2.34a}$$

$$\mathrm{B} = \overline{x}_2 + \overline{x}_1.\overline{x}_0 + x_1.x_0, \tag{2.34b}$$

$$\mathrm{C} = \overline{x}_1 + x_0 + x_2, \tag{2.34c}$$

$$\mathrm{D} = \overline{x}_2.\overline{x}_0 + \overline{x}_2.x_1 + x_1.\overline{x}_0 + x_2.\overline{x}_1.x_0, \tag{2.34d}$$

$$\mathrm{E} = \overline{x}_2.\overline{x}_0 + x_1.\overline{x}_0, \tag{2.34e}$$

$$\mathrm{F} = \overline{x}_1.\overline{x}_0 + x_2.\overline{x}_1 + x_2.\overline{x}_0 + x_3, \tag{2.34f}$$

$$\mathrm{G} = \overline{x}_2.\overline{x}_0 + x_2.\overline{x}_1 + x_1.\overline{x}_0 + x_3. \tag{2.34g}$$

To summarize:

- If the "don't care" of Table 2.15 are replaced by 0s, the set of (2.31) is obtained.
- If they are replaced by either 0 or 1, according to some optimization method (not described in this course), the set of (2.34) would have been obtained.

In Table 2.17 the numbers of AND, OR, and NOT gates necessary to implement (2.31) and (2.34) are shown.

The circuit that implements (2.31) has $6 \cdot 2 + 17 \cdot 3 + 1 \cdot 4 + 1 \cdot 2 + 2 \cdot 3 + 4 \cdot 4 + 4 \cdot 1 = 95$ gate inputs and the circuit that implements (2.34) has $14 \cdot 2 + 1 \cdot 3 + 1 \cdot 2 + 2 \cdot 3 + 4 \cdot 4 + 4 \cdot 1 = 59$ gate inputs. Obviously, the second circuit is better.

2.5.2 Cube Representation

A combinational circuit synthesis tool is a set of programs that generates optimized circuits, according to some criteria (cost, delay, power) starting either from logic expressions or from tables. The cube representation of combinational functions is an easy way to define Boolean expressions within a computer programming environment.

The set of n-component binary vectors ${B_2}^n$ can be considered as a cube (actually a hypercube) of dimension n. For example, if $n = 3$, the set ${B_2}^3$ of 3-component binary vectors is represented by the cube of Fig. 2.32a.

Fig. 2.32 Cubes

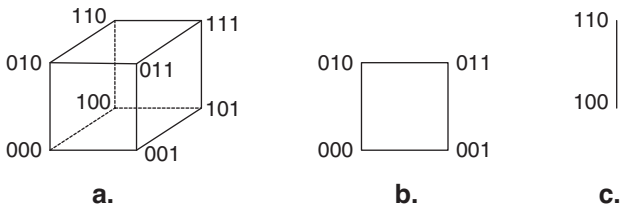
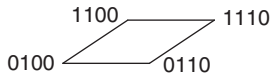


Fig. 2.33 Solutions of $x_2 \cdot \overline{x_0} = 1$



A subset of ${B_2}^n$ defined by giving a particular value to m vector components is a subcube ${B_2}^{n-m}$ of dimension n-m. As an example, the subset of vectors of ${B_2}^3$ whose first coordinate is equal to 0 (Fig. 2.32b) is a cube of dimension 2 (actually a square). Another example: the subset of vectors of ${B_2}^3$ whose first coordinate is 1 and whose third coordinate is 0 (Fig. 2.32c) is a cube of dimension 1 (actually a straight line).

Consider a 4-variable function f defined by the following Boolean expression:

$$f(x_3, x_2, x_1, x_0) = x_2 \cdot \overline{x_0}.$$

This function is equal to 1 if, and only if, $x_2 = 1$ and $x_0 = 0$, that is

$$f = 1 \text{ iff } (x_3, x_2, x_1, x_0) \in \left\{ x \in {B_2}^4 \middle| x_2 = 1 \text{ and } x_0 = 0 \right\}.$$

In other words, $f = 1$ if, and only if, (x_3, x_2, x_1, x_0) belongs to the 2-dimensional cube of Fig. 2.33.

This example suggests another definition.

Definition 2.2 A cube is a set of elements of ${B_2}^n$ where a product of literals (Definitions 2.1) is equal to 1.

In this chapter switching functions have been expressed under the form of sums of products of literals (e.g., (2.19) and (2.20)), and to those expressions correspond implementations by means of logic gates (e.g., Figs. 2.17 and 2.16). According to Definition 2.2, a set of elements of ${B_2}^n$ where a product of literals is equal to 1 is a cube. Thus, a sum of product of literals can also be defined as a union of cubes that defines the set of points of ${B_2}^n$ where $f = 1$. In what follows cube and product of literals are considered as synonymous.

How can a product of literals be represented within a computer programming environment? For that an order of the variables must be defined, for example (as above) x_{n-1}, x_{n-2}, . . ., x_1, x_0. Then, consider a product p of literals. It is represented by an n-component ternary vector ($p_{n-1}, p_{n-2}, \ldots, p_1$, p_0) where

- $p_i = 0$ if x_i is in p under inverted form ($\overline{x_i}$).
- $p_i = 1$ if x_i is in p under non-inverted form (x_i).
- $p_i = \mathrm{X}$ if x_i is not in p.

Example 2.9 (with $n = 4$) The set of cubes that describes (2.31d) is {X000, 001X, 0X10, 0101}, and the set of cubes that corresponds to (2.34g) is {X0X0, X10X, XX10, 1XXX}. Conversely, the product of literals represented by 1X01 is $x_3 \cdot \overline{x_1} \cdot x_0$ and the product of literals represented by X1X0 is $x_2 \cdot \overline{x_0}$.

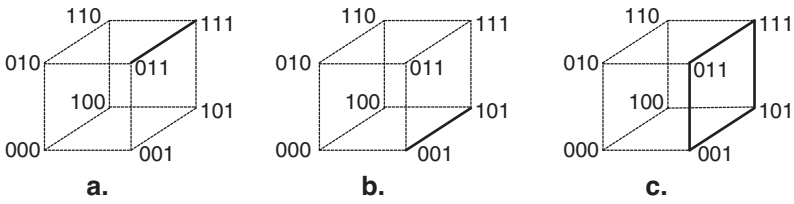


Fig. 2.34 Union of cubes

2.5.3 Adjacency

Adjacency is the basic concept that permits to optimize Boolean expressions. Two m-dimensional cubes are adjacent if their associated ternary vectors differ in only one position. As an example ($n = 3$), the 1-dimensional cubes X11 (Fig. 2.34a) and X01 (Fig. 2.34b) are adjacent and their union is a 2-dimensional cube XX1 = X11 ∪ X01 (Fig. 2.34c).

The corresponding products of literals are the following: X11 represents $x_1 \cdot x_0$, X01 represents $\overline{x_1} \cdot x_0$, and their union XX1 represents x_0. In terms of products of literals, the union of the two adjacent cubes is the sum of the corresponding products:

$$x_1 \cdot x_0 + \overline{x_1} \cdot x_0 = (x_1 + \overline{x_1}) \cdot x_0 = 1 \cdot x_0 = x_0.$$

Thus, if a function f is defined by a union of cubes and if two cubes are adjacent, then they can be replaced by their union. The result, in terms of products of literals, is that two products of n-m literals are replaced by a single product of $n - m - 1$ literals.

Example 2.10 A function f of four variables a, b, c, and d is defined by its minterms (Definition 2.1):

$$f(a,b,c,d) = \overline{a} \cdot \overline{b} \cdot c \cdot \overline{d} + \overline{a} \cdot \overline{b} \cdot c \cdot d + \overline{a} \cdot b \cdot \overline{c} \cdot d + \overline{a} \cdot b \cdot c \cdot \overline{d} + \overline{a} \cdot b \cdot c \cdot d + a \cdot \overline{b} \cdot \overline{c} \cdot \overline{d}.$$

The corresponding set of cubes is

$$\{0010,\ 0011,\ 0101,\ 0110,\ 0111,\ 1000\}.$$

The following adjacencies permit to simplify the representation of f:

$$0010 \cup 0011 = 001\text{X},$$
$$0110 \cup 0111 = 011\text{X},$$
$$0101 \cup 0111 = 01\text{X}1.$$

Thanks to the idempotence property (2.9) the same cube (0111 in this example) can be used several times. The simplified set of cubes is

$$\{001\text{X},\ 011\text{X},\ 01\text{X}1,\ 1000\}.$$

There remains an adjacency:

$$001\text{X} \cup 011\text{X} = 0\text{X}1\text{X}.$$

The final result is

$$\{0X1X,\ 01X1,\ 1000\}$$

and the corresponding Boolean expression is

$$f = \overline{a} \cdot c + \overline{a} \cdot b \cdot d + a \cdot \overline{b} \cdot \overline{c} \cdot \overline{d}.$$

To conclude, a repeated use of the fact that two adjacent cubes can be replaced by a single cube permits to generate new Boolean expressions, equivalent to the initial one and with fewer terms. Furthermore the new terms have fewer literals. This is the basis of most automatic optimization tools.

All commercial synthesis tools include programs that automatically generate optimal circuits according to some criteria such as cost, delay, or power consumption, and starting from several types of specification. For education purpose open-source tools are available, for example C. Burch (2005).

2.5.4 Karnaugh Map

In the case of switching functions of a few variables, a graphical method can be used to detect adjacencies and to optimize Boolean expressions. Consider the function $f(a, b, c, d)$ of Example 2.10. It can be represented by the Karnaugh map (Karnaugh 1953) of Fig. 2.35a. Observe the enumeration ordering of rows and columns (00, 01, 11, 10): the variable values that correspond to a row (a column) and to the next row (the next column) differ in only one position.

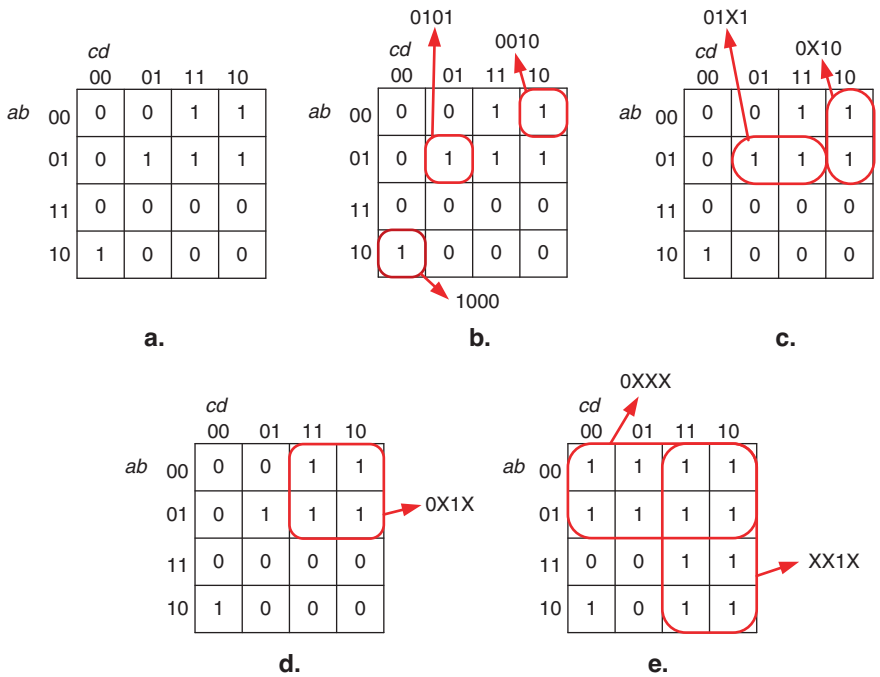


Fig. 2.35 Karnaugh maps

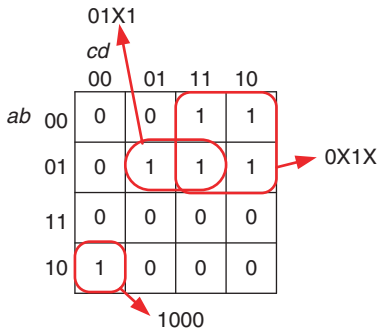


Fig. 2.36 Optimization of f

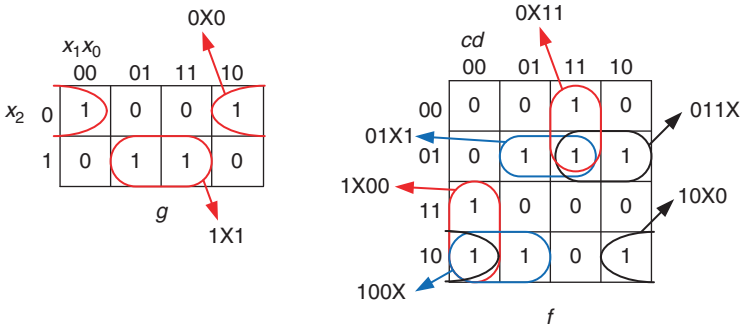


Fig. 2.37 Functions g and h

To each one of this graphical representation is associated a minterm of the function (a -0-dimensional cube). Several examples are shown in Fig. 2.35b. Thanks to the chosen enumeration ordering, to groups of two adjacent 1s like those of Fig. 2.35c are associated 1-dimensional cubes. To a group of four adjacent 1s like the one of Fig. 2.35d is associated a 2-dimensional cube. To groups of eight adjacent 1s like those of Fig. 2.35e (another switching function) are associated 3-dimensional cubes.

Thus (Fig. 2.36) the function $f(a, b, c, d)$ of Example 2.10 can be expressed as the Boolean sum of three cubes 0X1X, 01X1, and 1000 so that

$$f = \overline{a} \cdot c + \overline{a} \cdot b \cdot d + a \cdot \overline{b} \cdot \overline{c} \cdot \overline{d}.$$

It is important to observe that the rightmost cells and the leftmost cells are adjacent, and so are also the uppermost cells and the downmost cells (as if the map were drawn on the surface of a torus).

Two additional examples are given in Fig. 2.37. Function g of Fig. 2.37a can be expressed as the Boolean sum of two 1-dimensional cubes 0X0 and 1X1, so that

$$g = \overline{x_2} \cdot \overline{x_0} + x_2 \cdot x_0,$$

and function h of Fig. 2.37b can be expressed as the Boolean sum of six 1-dimensional cubes 01X1, 011X, 0X11, 10X0, 100X, and 1X00, so that

$$h = \overline{a} \cdot b \cdot d + \overline{a} \cdot b \cdot c + \overline{a} \cdot c \cdot d + a \cdot \overline{b} \cdot \overline{d} + a \cdot \overline{b} \cdot \overline{c} + a \cdot \overline{c} \cdot \overline{d}.$$

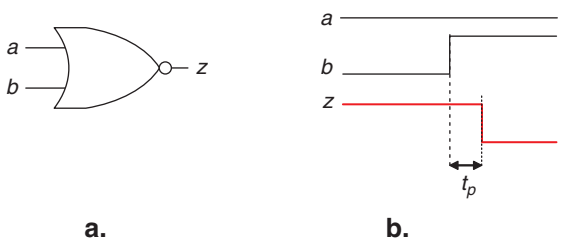


Fig. 2.38 Propagation time t_p

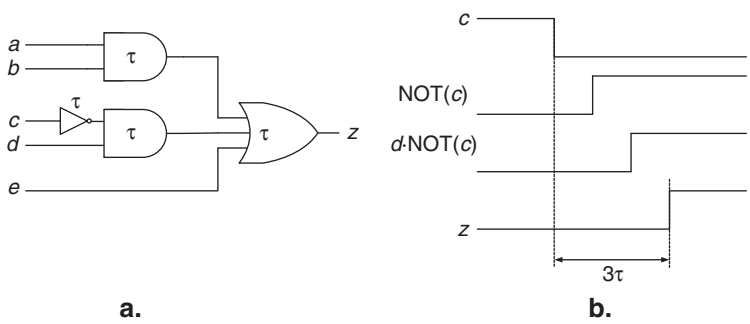


Fig. 2.39 Example of propagation time computation

2.6 Propagation Time

Logic components such as gates are physical systems. Any change of their state, for example the output voltage transition from some level to another level, needs some quantity of energy and therefore some time (zero delay would mean infinite power). Thus, apart from their function (AND2, OR3, NAND4, and so on), logic gates are also characterized by their propagation time (delay) between inputs and outputs.

Consider a simple NOR2 gate (Fig. 2.38a). Assume that initially $a = b = 0$. Then $z = \text{NOR}(0, 0) = 1$ (Fig. 2.38b). When b rises from 0 to 1, then $\text{NOR}(0, 1) = 0$ and z must fall from 1 to 0. However the output state change is not immediate; there is a small delay t_p generally expressed in nanoseconds (ns) or picoseconds (ps).

Example 2.11 The circuit of Fig. 2.39a implements a 5-variable switching function $z = a \cdot b + \overline{c} \cdot d + e$. Assume that all components (AND2, NOT, OR3) have the same propagation time τ ns. Initially $a = 0$, b is either 0 or 1, $c = 1$, $d = 1$, and $e = 0$. Thus $z = 0 \cdot b + \overline{1} \cdot 1 + 0 = 0$. If c falls from 1 down to 0 then the new value of z must be $z = 0 \cdot b + \overline{0} \cdot 1 + 0 = 1$. However this output state change takes some time: the inverter output $\overline{c}$ changes after τ ns; the AND2 output $\overline{c} \cdot d$ changes after 2τ ns, and the OR3 output z changes after 3τ ns (Fig. 2.39b).

Thus, the propagation time of a circuit depends on the component propagation times but also on the circuit itself. Two different circuits could implement the same switching circuit but with different propagation times.

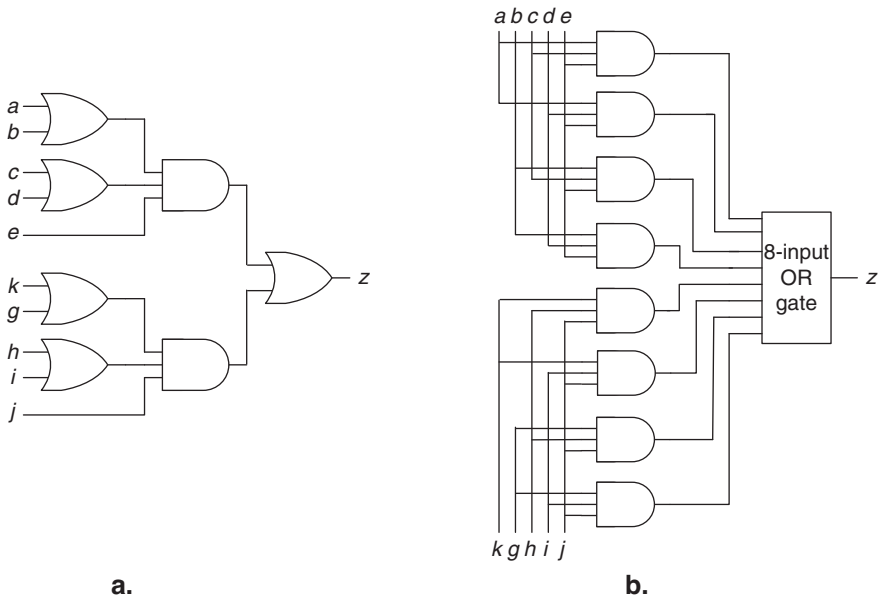
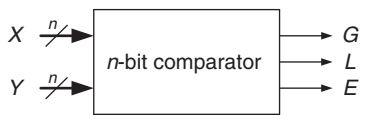


Fig. 2.40 Two circuits that implement the same function f

Fig. 2.41 n-Bit comparator



Example 2.12 The two following expressions define the same switching function z:

$$z = (a + b) \cdot (c + d) \cdot e + (k + g) \cdot (h + i) \cdot j,$$
$$z = a \cdot c \cdot e + a \cdot d \cdot e + b \cdot c \cdot e + b \cdot d \cdot e + k \cdot h \cdot j + k \cdot i \cdot j + g \cdot h \cdot j + g \cdot i \cdot j.$$

The corresponding circuits are shown in Fig. 2.40a, b. The circuit of Fig. 2.40a has 7 gates and 16 gate inputs while the circuit of Fig. 2.40b has 9 gates and 32 gate inputs. On the other hand, if all gates are assumed to have the same propagation time τ ns, then the circuit of Fig. 2.40a has a propagation time equal to 3τ ns while the circuit of Fig. 2.40b has a propagation time equal to 2τ ns. Thus, the circuit of Fig. 2.40a could be less expensive in terms of number of transistors but with a longer propagation time than the circuit of Fig. 2.40b. In function of the system specification, the designer will have to choose between a faster but more expensive implementation or a slower and cheaper implementation (speed vs. cost balance).

A more realistic example is now presented. An n-bit comparator (Fig. 2.41) is a circuit with two n-bit inputs $X = x_{n-1}x_{n-2} \ldots x_0$ and $Y = y_{n-1}y_{n-2} \ldots y_0$ that represent two naturals and three 1-bit outputs G (greater), L (lower), and E (equal). It works as follows: $G = 1$ if $X > Y$, otherwise $G = 0$; $L = 1$ if $X < Y$, otherwise $L = 0$; $E = 1$ if $X = Y$, otherwise $E = 0$.

A step-by-step algorithm can be used. For that, the pairs of bits (x_i, y_i) are sequentially explored starting from the most significant bits (x_{n-1}, y_{n-1}). Initially $G = 0$, $L = 0$, and $E = 1$. As long as $x_i = y_i$, the values of G, L, and E do not change. When for the first time $x_i \neq y_i$, there are two possibilities: if $x_i > y_i$ then $G = 1, L = 0$, and $E = 0$, and if if $x_i < y_i$ then $G = 0, L = 1$, and $E = 0$. From this step, the values of G, L, and E do not change any more.

Table 2.18 Magnitude comparison

X	1	0	1	1	0 or 1	0 or 1	0 or 1	0 or 1
Y	1	0	1	0	0 or 1	0 or 1	0 or 1	0 or 1
G	0	0	0	1	1	1	1	1
L	0	0	0	0	0	0	0	0
E	1	1	1	0	0	0	0	0

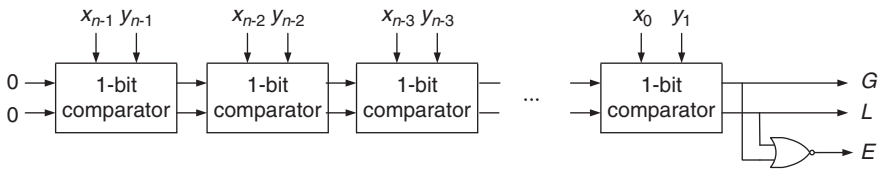


Fig. 2.42 Comparator structure

Algorithm 2.2 Magnitude Comparison

```
G = 0; L = 0; E = 1;
for i in n-1 downto 0 loop
  if E = 1 and x_i > y_i then
     G = 1; L = 0; E = 0;
  elsif E = 1 and x_i < y_i then
     G = 0; L = 1; E = 0;
  end if;
end loop;
```

This method is correct because in binary the weight of bits x_i and y_i is 2^i and is greater than $2^{i-1} + 2^{i-2} + \ldots + 2^0 = 2^i - 1$. An example of computation is given in Table 2.18 with $n = 8$, $X = 1011$---- and $Y = 1010$----.

The corresponding circuit structure is shown in Fig. 2.42. Obviously $E = 1$ if $G = 0$ and $L = 0$ so that $E = \text{NOR}\ (G, L)$.

Every block (Fig. 2.43) executes the loop body of Algorithm 2.2 and is defined by the following Boolean expressions where $E_i = \overline{G_i} \cdot \overline{L_i}$:

$$G_{i-1} = E_i \cdot x_i \cdot \overline{y_i} + \overline{E_i} \cdot G_i = \overline{G_i} \cdot \overline{L_i} \cdot x_i \cdot \overline{y_i} + (G_i + L_i) \cdot G_i = \overline{L_i} \cdot x_i \cdot \overline{y_i} + G_i, \tag{2.35}$$

$$L_{i-1} = E_i \cdot \overline{x_i} \cdot y_i + \overline{E_i} \cdot L_i = \overline{G_i} \cdot \overline{L_i} \cdot \overline{x_i} \cdot y_i + (G_i + L_i) \cdot L_i = \overline{G_i} \cdot \overline{x_i} \cdot y_i + L_i \tag{2.36}$$

The circuit that implements (2.35) and (2.36) is shown in Fig. 2.44. It contains 8 gates (including the inverters) and 14 gate inputs, and the propagation time is 3τ ns assuming as before that all components (NOT, AND3, and OR2) have the same delay τ ns.

The complete n-bit comparator (Fig. 2.42) contains $8n + 1$ gates and $14n + 2$ gate inputs and has a propagation time equal to $(3n + 1)\tau$ ns.

Instead of reading the bits of X and Y one at a time, consider an algorithm that reads two bits of X and Y at each step. Assume that $n = 2\ m$. Then the following Algorithm 2.3 is similar to Algorithm 2.2. The difference is that two successive bits x_{2j+1} and x_{2j} of X and two successive bits y_{2j+1} and y_{2j} of Y are considered. Those pairs of bits can be interpreted as quaternary digits (base-4 digits).

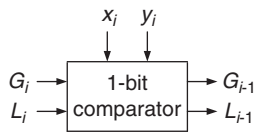


Fig. 2.43 1-Bit comparator

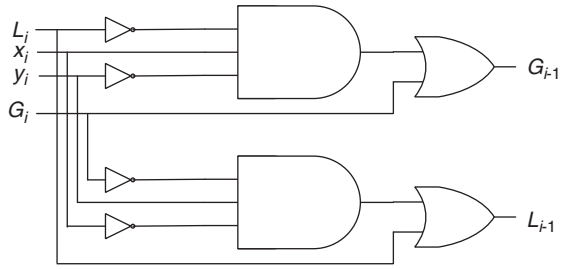


Fig. 2.44 1-Bit comparator implementation

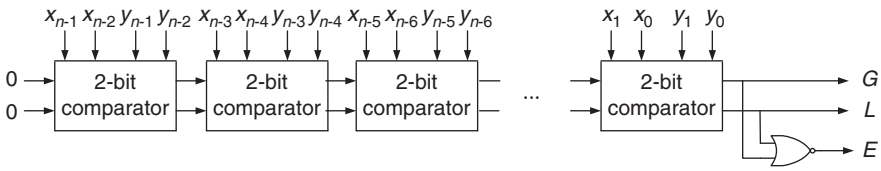


Fig. 2.45 Comparator structure (version 2)

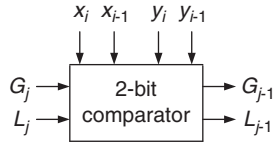


Fig. 2.46 2-Bit comparator

Algorithm 2.3 Magnitude Comparison, Version 2

```
G = 0; L = 0; E = 1;
for j in m-1 downto 0 loop
  if E = 1 and x2j+1 2j > y2j+1 2jthen
     G = 1; L = 0; E = 0;
  elsif E = 1 and x2j+1 2j < y2j+1 2jthen
     G = 0; L = 1; E = 0;
  end if;
end loop;
```

The corresponding circuit structure is shown in Fig. 2.45.

Every block (Fig. 2.46) executes the loop body of Algorithm 2.3 and is defined by Table 2.19 to which correspond the following equations:

$$G_{j-1} = \overline{L_j} \cdot x_{i-1} \cdot \overline{y_i} \cdot \overline{y_{i-1}} + \overline{L_j} \cdot x_i \cdot \overline{y_i} + \overline{L_j} \cdot x_i \cdot x_{i-1} \cdot \overline{y_{i-1}} + G_j, \tag{2.37}$$

$$L_{j-1} = \overline{G_j} \cdot y_{i-1} \cdot \overline{x_i} \cdot \overline{x_{i-1}} + \overline{G_j} \cdot y_i \cdot \overline{x_i} + \overline{G_j} \cdot y_i \cdot y_{i-1} \cdot \overline{x_{i-1}} + L_j, \tag{2.38}$$

where $i = 2j + 1$.

Table 2.19 2-Bit comparator definition

G_j	L_j	x_i	x_{i-1}	y_i	y_{i-1}	G_{j-1}	L_{j-1}
0	0	0	0	0	0	0	0
0	0	0	0	1	–	0	1
0	0	0	0	–	1	0	1
0	0	0	1	0	0	1	0
0	0	0	1	0	1	0	0
0	0	0	1	1	–	0	1
0	0	1	0	0	–	1	0
0	0	1	0	1	0	0	0
0	0	1	0	1	1	0	1
0	0	1	1	0	–	1	0
0	0	1	1	–	0	1	0
0	0	1	1	1	1	0	0
0	1	–	–	–	–	0	1
1	0	–	–	–	–	1	0
1	1	–	–	–	–	–	–

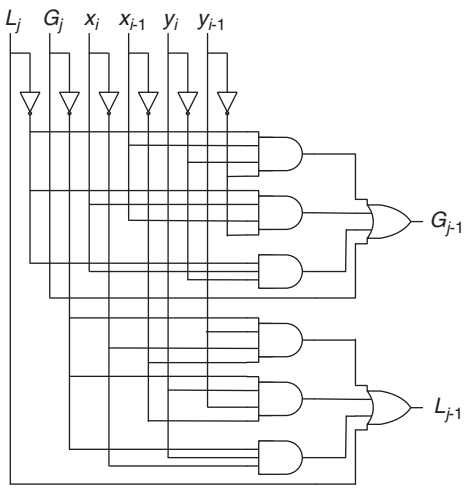


Fig. 2.47 2-Bit comparator implementation

Table 2.20 Comparison between the circuits of Figs. 2.42 and 2.45

Circuit	Gates	Gate inputs	Propagation time
Figure 2.42	$8n + 1$	$14n + 2$	$(3n + 1)\tau$
Figure 2.45	$7n + 1$	$18n + 2$	$(1.5n + 1)\tau$

The circuit that implements (2.37) and (2.38) is shown in Fig. 2.47. It contains 14 gates (including the inverters) and 36 gate inputs, and the propagation time is 3τ ns assuming as before that all components (NOT, AND3, AND4, and OR4) have the same delay τ ns.

The complete n-bit comparator (Fig. 2.45), with $n = 2m$, contains $14m + 1 = 7n + 1$ gates and $36m + 2 = 18n + 2$ gate inputs and has a propagation time equal to $(3m + 1)\tau = (1.5n + 1)\tau$ ns.

To summarize (Table 2.20) the circuit of Fig. 2.45 has fewer gates, more gate inputs, and a shorter propagation time than the circuit of Fig. 2.42 (roughly half the propagation time).

2.7 Other Logic Blocks

Apart from the logic gates, some other components are available and can be used to implement combinational circuits.

2.7.1 Multiplexers

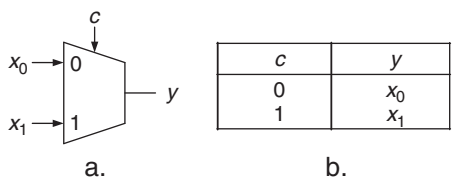
The circuit of Fig. 2.48a is a 1-bit 2-to-1 multiplexer (MUX2-1). It has two data inputs x_0 and x_1, a control input c, and a data output y. It works as follows (Fig. 2.48b): when $c = 0$ the data output y is connected to the data input x_0 and when $c = 1$ the data output y is connected to the data input x_1. So, the main function of a multiplexer is to implement controllable connections.

A typical application is shown in Fig. 2.49: the input of *circuit_C* can be connected to the output of either *circuit_A* or *circuit_B* under the control of signal *control*:

- If *control* = 0, *circuit_C* input = *circuit_A* output.
- If *control* = 1, *circuit_C* input = *circuit_B* output.

More complex multiplexers can be defined. An m-bit 2^n-to-1 multiplexer has $2^n m$-bit data inputs $x_0, x_1, \ldots, x_{2^n-1}$, an n-bit control input c, and an m-bit data output y. It works as follows: if c is equal to the binary representation of natural i, then $y = x_i$.

Two examples are given in Fig. 2.50: in Fig. 2.50a the symbol of an m-bit 2-to-1 multiplexer is shown, and in Fig. 2.50b the symbol and the truth table of a 1-bit 4-to-1 multiplexer (MUX4-1) are shown.



c	y
0	x_0
1	x_1

Fig. 2.48 1-Bit 2-to-1 multiplexer

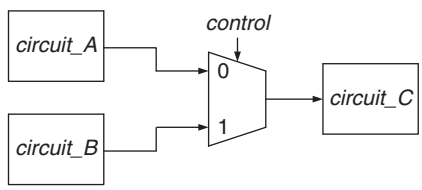
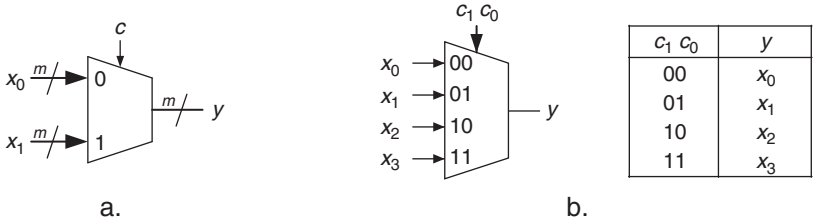


Fig. 2.49 Example of controllable connection



c_1 c_0	y
00	x_0
01	x_1
10	x_2
11	x_3

Fig. 2.50 Examples of multiplexers

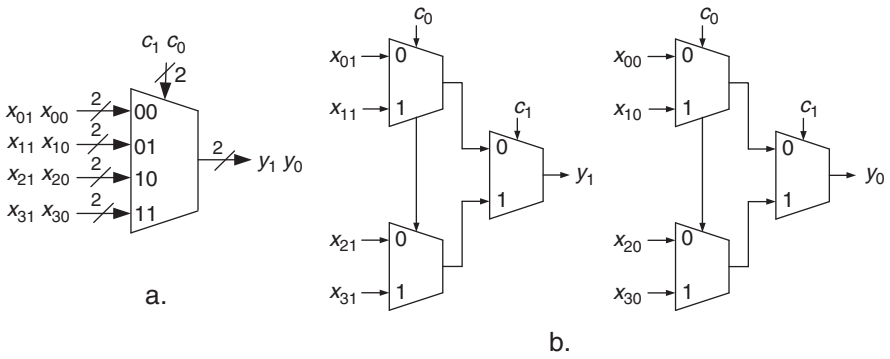
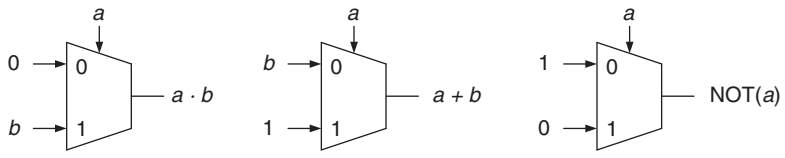


Fig. 2.51 2-Bit MUX4-1 implemented with six 1-bit MUX2-1

Fig. 2.52 MUX2-1 is a universal module



x_2	x_1	x_0	y_1	y_0
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

a.

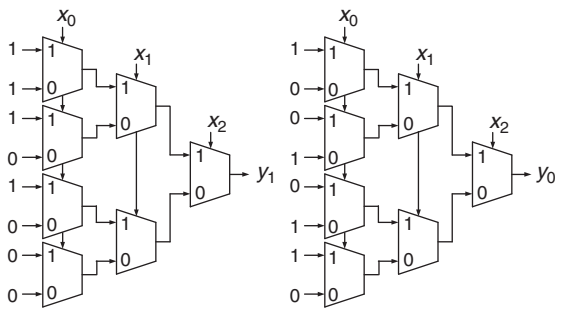


Fig. 2.53 Implementation of two 3-variable switching functions

In fact, any multiplexer can be built with 1-bit 2-to-1 multiplexers. For example, Fig. 2.51a is the symbol of a 2-bit MUX4-1 and Fig. 2.51b is an implementation consisting of six 1-bit MUX2-1.

Multiplexers can also be used to implement switching functions. The function executed by the 1-bit MUX2-1 of Fig. 2.48 is

$$y = \overline{c} \cdot x_0 + c \cdot x_1. \tag{2.39}$$

In particular, MUX2-1 is a universal module (Fig. 2.52):

- If $c = a$, $x_0 = 0$ and $x_1 = b$, then $y = a \cdot b$.
- If $c = a$, $x_0 = b$ and $x_1 = 1$, then $y = \overline{a} \cdot b + a = a + b$.
- If $c = a$, $x_0 = 1$ and $x_1 = 0$, then $y = \overline{a}$.

Furthermore, any switching function of n variables can be implemented by a 2^n-to-1 multiplexer. As an example, consider the 3-variable switching functions y_1 and y_0 of Fig. 2.53a. Each of them can be implemented by a MUX8-1 that in turn can be synthesized with seven MUX2-1 (Fig. 2.53b). The

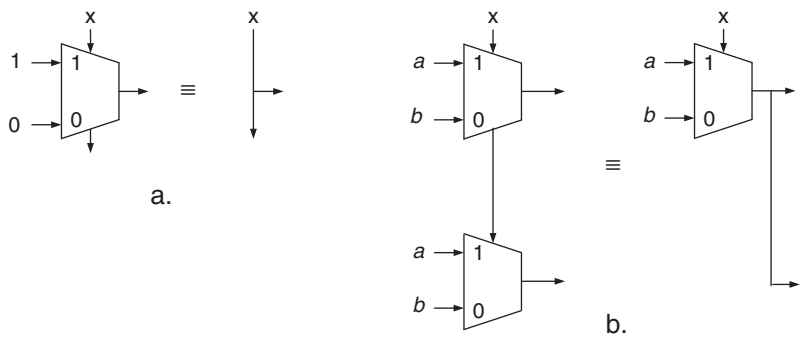


Fig. 2.54 Optimization rules

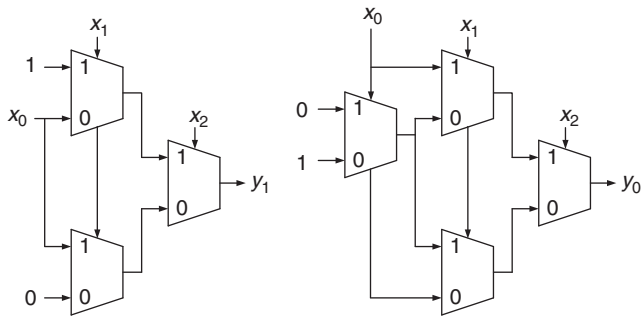


Fig. 2.55 Optimized circuits

three variables x_2, x_1, and x_0 are used to control the connection of the output (y_1 or y_0) to a constant value as defined in the function truth table.

In many cases the circuit can be simplified using simple and obvious rules. Two optimization rules are shown in Fig. 2.54. In Fig. 2.54a if $x = 0$ then the multiplexer output is equal to 0 and if $x = 1$ then the multiplexer output is equal to 1. Thus the multiplexer output is equal to x. In Fig. 2.54b two multiplexers controlled by the same variable x and with the same data inputs can be replaced by a unique multiplexer.

An optimized version of the circuits of Fig. 2.53b is shown in Fig. 2.55.

Two switching function synthesis methods using multiplexers have been described. The first is to use multiplexers to implement the basic Boolean operations (AND, OR, NOT), which is generally not a good idea, rather a way to demonstrate that MUX2-1 is a universal module. The second is the use of an m-bit 2^n-to-1 multiplexer to implement m functions of n variables. In fact, an m-bit 2^n-to-1 multiplexer with all its data inputs connected to constant values implements the same function as a ROM storing $2^n m$-bit words. Then the 2^n-to-1 multiplexers can be synthesized with MUX2-1 and the circuits can be optimized using rules such as those of Fig. 2.54.

A more general switching function synthesis method with MUX2-1 components is based on (2.39) and on the fact that any n-variable switching function $f(x_0, x_1, \ldots, x_{n-1})$ can be expressed under the form

$$f(x_0, x_1, \ldots, x_{n-1}) = \overline{x_0} \cdot f_0(x_1, \ldots, x_{n-1}) + x_0 \cdot f_1(x_1, \ldots, x_{n-1}) \tag{2.40}$$

where

$$f_0(x_1, \ldots, x_{n-1}) = f(0, x_1, \ldots, x_{n-1}) \text{ and } f_1(x_1, \ldots, x_{n-1}) = f(1, x_1, \ldots, x_{n-1}) \tag{2.41}$$

are functions of $n - 1$ variables. The circuit of Fig. 2.56 is a direct consequence of (2.40) and (2.41). In this way variable x_0 has been extracted.

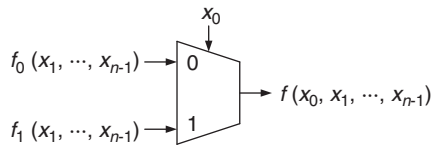


Fig. 2.56 Extraction of variable x_0

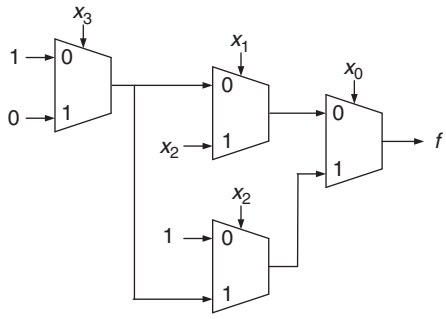


Fig. 2.57 MUX2-1 implementation of f

Then a similar variable extraction can be performed with functions f_0 and f_1 (not necessarily the same variable) so that functions of $n-2$ variables are obtained, and so on. Thus, an iterative extraction of variables finally generates constants (0-variable functions), variables, or already generated functions.

Example 2.13 Use the variable extraction method to implement the following 4-variable function:

$$f = \overline{x_0} \cdot \overline{x_1} \cdot \overline{x_3} + \overline{x_0} \cdot x_1 \cdot x_2 + x_0 \cdot \overline{x_2} + x_0 \cdot \overline{x_3}.$$

First extract x_0: $f_0 = \overline{x_1} \cdot \overline{x_3} + x_1 \cdot x_2$ and $f_1 = \overline{x_2} + \overline{x_3}$.
Then extract x_1 from f_0: $f_{00} = \overline{x_3}$ and $f_{01} = x_2$.
Extract x_2 from f_1: $f_{10} = 1$ and $f_{11} = \overline{x_3}$.
It remains to synthesize $\overline{x_3} = \overline{x_3} \cdot 1 + x_3 \cdot 0$.
The circuit is shown in Fig. 2.57.

2.7.2 Multiplexers and Memory Blocks

ROM blocks can be used to implement switching functions defined by their truth table (Sect. 2.2) but in most cases it is a very inefficient method. However the combined use of small ROM blocks, generally called LUT, and of multiplexers permits to define efficient circuits. This is a commonly used technique in field programmable devices such as FPGAs (Chap. 7).

Assume that 6-input LUTs (LUT6) are available. Then the variable extraction method of Fig. 2.56 can be iteratively applied up to the step where all obtained functions depend on at most six variables. As an example, the circuit of Fig. 2.58 implements any function of eight variables.

Observe that the rightmost part of the circuit of Fig. 2.58 synthesizes a function of six variables: x_6, x_7 and the four LUT6 outputs. An alternative circuit consisting of five LUT6 is shown in Fig. 2.59.

Figure 2.59 suggests a variable extraction method in which two variables are extracted at each step. It uses the following relation:

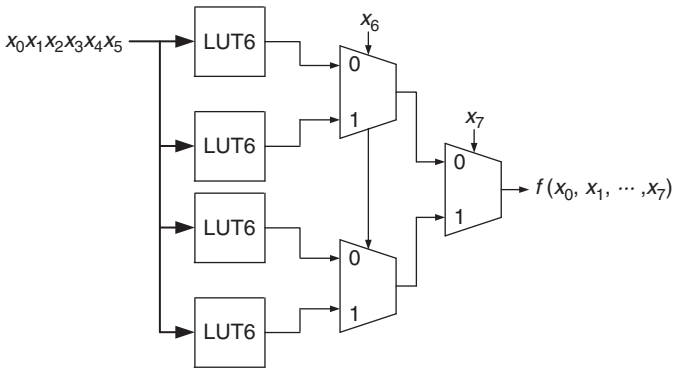


Fig. 2.58 Implementation of an 8-variable switching function

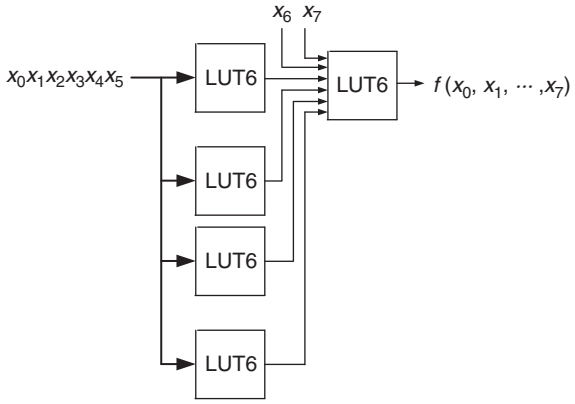


Fig. 2.59 Alternative circuit

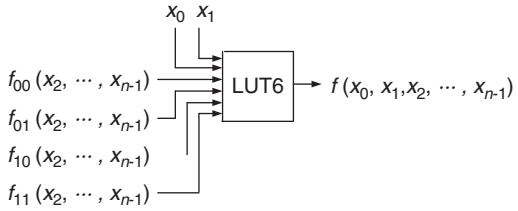


Fig. 2.60 Extraction of variables x_0 and x_1

$$f(x_0, x_1, \ \ldots, x_{n-1}) = \ \overline{x_0} \cdot \overline{x_1} \cdot f_{00}(x_2, \ldots, x_{n-1}) + \overline{x_0} \cdot x_1 \cdot f_{01}(x_2, \ \ldots, x_{n-1}) \\ + \ x_0 \cdot \overline{x_1} \cdot f_{10}(x_2, \ldots, x_{n-1}) + \ x_0 \cdot \ x_1 \cdot f_{11}(x_2, \ \ldots, x_{n-1}). \tag{2.42}$$

where

$$f_{00}(x_2, \ldots, x_{n-1}) = f(0, \ 0, x_2, \ \ldots, x_{n-1}), f_{01}(x_2, \ldots, x_{n-1}) = f(0, \ 1, x_2, \ \ldots, x_{n-1}),$$
$$f_{10}(x_2, \ldots, x_{n-1}) = f(1, \ 0, x_2, \ \ldots, x_{n-1}), f_{11}(x_2, \ldots, x_{n-1}) = f(1, \ 1, x_2, \ \ldots, x_{n-1})$$

are functions of $n - 2$ variables. The corresponding variable extraction circuit (Fig. 2.60) is a LUT6 that implements a function of six variables x_0, x_1, f_{00}, f_{01}, f_{10}, and f_{11} equal to

$$\overline{x_0} \cdot \overline{x_1} \cdot f_{00} + \overline{x_0} \cdot \ x_1 \cdot f_{01} + \ x_0 \cdot \ \overline{x_1} \cdot f_{10} + \ x_0 \cdot \ x_1 \cdot f_{11}.$$

Then a similar variable extraction can be performed with functions f_{00}, f_{01}, f_{10}, and f_{11} so that functions of $n - 4$ variables are obtained, and so on.

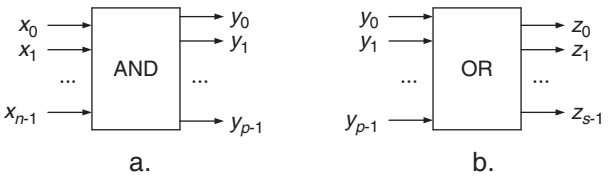


Fig. 2.61 AND plane and OR plane

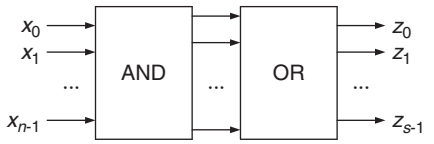
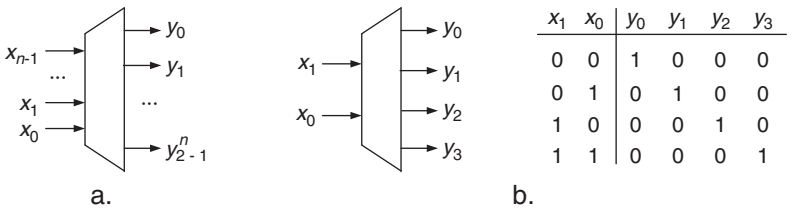


Fig. 2.62 Switching function implementation with two planes



x_1	x_0	y_0	y_1	y_2	y_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Fig. 2.63 Address decoders

2.7.3 Planes

Sometimes AND planes and OR planes are used to implement switching functions. An (n, p) AND plane (Fig. 2.61a) implements p functions y_j of n variables, where y_j is a product of literals (variable or inverse of a variable):

$$y_j = w_{j,0} w_{j,1} \ldots w_{j,n-1} \text{ where } w_{j,i} \in \{1, x_i, \overline{x}_i\}.$$

An (p, s) OR plane (Fig. 2.61b) implements s functions z_j of p variables, where z_j is a Boolean sum of variables:

$$z_j = w_{j,0} + w_{j,1} + \ldots + w_{j,p-1} \text{ where } w_{j,i} \in \{0, y_i\}.$$

Those planes can be configured when the corresponding integrated circuit (IC) is manufactured, or can be programmed by the user in which case they are called field programmable devices.

Any set of s switching functions that are expressed as Boolean sums of at most p products of at most n literals can be implemented by a circuit made up of an (n, p) AND plane and an (p, s) OR plane (Fig. 2.62): the AND plane generates p products of at most n literals and the OR plane generates s sums of at most p terms.

Depending on the manufacturing technology and on the manufacturer those AND-OR plane circuits receive different names such as programmable array of logic (PAL), Programmable Logic Array (PLA), Programmable Logic Device (PLD), and others.

2.7.4 Address Decoder and Tristate Buffers

Another type of useful component is the address decoder. An n-to-2^n address decoder (Fig. 2.63a) has n inputs and 2^n outputs and its function is defined as follows: if $x_{n-1}x_{n-2} \ldots x_0$ is the binary

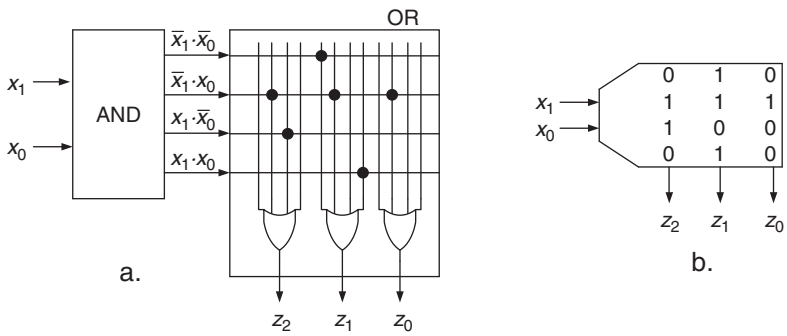


Fig. 2.64 AND-OR plane implementation of a ROM

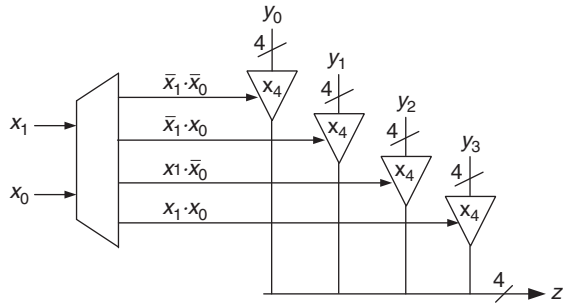


Fig. 2.65 4-Bit MUX4-1 implemented with an address decoder and four tristate buffers

representation of natural i, then $y_i = 1$ and all other outputs $y_j = 0$. As an example, a 2-to-4 address decoder and its truth table are shown in Fig. 2.63b.

In fact, an n-to-2^n address decoder implements the same function as an $(n, 2^n)$ AND plane that generates all n-variable minterms:

$$m_j = w_{j,0} w_{j,1} \ldots w_{j,n-1} \quad \text{where} \quad w_{j,i} \in \{x_i, \overline{x}_i\}.$$

By connecting an n-to-2^n address decoder to an $(2^n, s)$ OR plane, the obtained circuit implements the same function as a ROM storing $2^n s$-bit words. An example is given in Fig. 2.64a: the AND plane synthesizes the functions of a 2-to-4 address decoder and the complete circuit implements the same functions as the ROM of Fig. 2.64b.

The other common application of address decoders is the control of data buses. An example is given in Fig. 2.65: a 2-to-4 address decoder generates four signals that control four 4-bit tristate buffers. This circuit permits to connect a 4-bit output z to one among four 4-bit inputs y_0, y_1, y_2, or y_3 under the control of two address bits x_1 and x_0. Actually, the circuit of Fig. 2.65 realizes the same function as a 4-bit MUX4-1.

In Fig. 2.66a the circuit of Fig. 2.65 is used to connect one among four data sources (circuits A, B, C, and D) to a data destination (circuit E) under the control of two address bits. It executes the following algorithm:

```
case x1 x0 is
  when 00 => circuit_E = circuit_A;
  when 01 => circuit_E = circuit_B;
  when 00 => circuit_E = circuit_C;
  when 00 => circuit_E = circuit_D;
end case;
```

The usual symbol of this bus is shown in Fig. 2.66b.

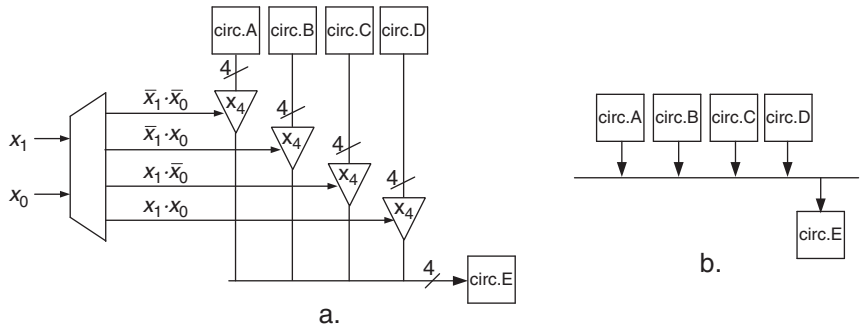


Fig. 2.66 A 4-bit data bus with four data sources

2.8 Programming Language Structures

The specification of digital systems by means of algorithms (Sect. 1.2.1) is a central aspect of this course. In this section the relation between some programming language instructions and digital circuits is analyzed. This relation justifies the use of hardware description languages (HDL) similar to programming languages, as well as the generation of synthesis tools able to translate HDL descriptions to circuits.

2.8.1 If Then Else

A first example of instruction that can be translated to a circuit is the conditional branch:

```
if a_condition then some_actions else other_actions;
```

As an example, consider the following binary decision algorithm. It computes the value of a switching function f of six variables x_0, x_1, y_0, y_1, y_2, and y_3.

Algorithm 2.4

```
if x1 = 0 then
  if x0 = 0 then f = y0; else f = y1; end if;
else
  if x0 = 0 then f = y2; else f = y3; end if;
end if;
```

This function can be implemented by the circuit of Fig. 2.67 in which the external conditional branch is implemented by the rightmost MUX2-1 and the two internal conditional branches are implemented by the leftmost MUX2-1s.

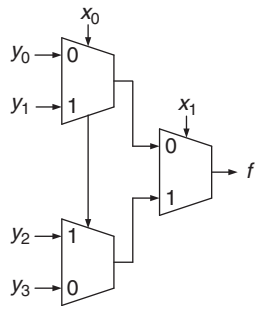


Fig. 2.67 Binary decision algorithm implementation

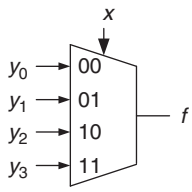


Fig. 2.68 Case instruction implementation

2.8.2 Case

A second example of instruction that can be translated to a circuit is the conditional switch:

```
case variable_identifier is
  when variable_value1 => actions1;
  when variable_value2 => actions2;
  ...
end case;
```

As an example, the preceding binary decision algorithm (Algorithm 2.4) is equivalent to the following, assuming that x has been previously defined as a 2-bit vector (x_0, x_1).

Algorithm 2.5

```
case x is
  when 00 => f = y0;
  when 01 => f = y1;
  when 10 => f = y2;
  when 11 => f = y3;
end case;
```

Function f can be implemented by a MUX4-1 (Fig. 2.68).

2.8.3 Loops

For-loops are a third example of easily translatable construct:

```
for variable_identifier in variable_range loop
  operations using the variable value
end loop;
```

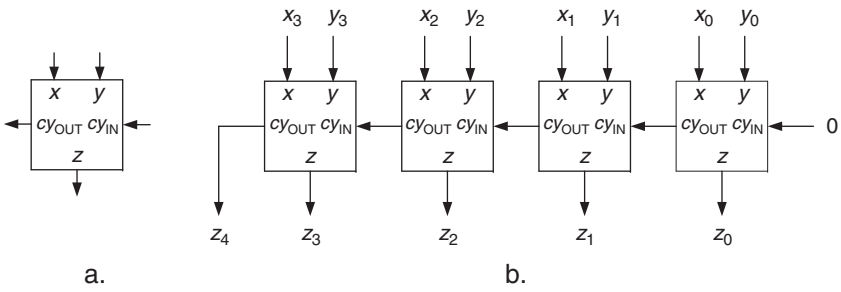


Fig. 2.69 4-Digit decimal adder

To this type of instruction can often be associated an iterative circuit. As an example, consider the following addition algorithm that computes $z = x + y$ where x and y are 4-digit decimal numbers, so that z is a 5-digit decimal number.

Algorithm 2.6 Addition of Two 4-Digit Naturals

```
cy0= 0;
for i in 0 to 3 loop
    --------- loop body:
    si= xi+ yi+ cyi;
    if si> 9 then zi= si- 10; cyi+1= 1;
    else zi= si; cyi+1= 0;
    end if;
    --------- end of loop body:
end loop;
z4= cy4;
```

The corresponding circuit is shown in Fig. 2.69b. It is an iterative circuit that consists of four identical blocks. Each of them is a 1-digit adder (Fig. 2.69a) that implements the loop body of Algorithm 2.6.

Comments 2.5

- Other (not combinational but sequential) loop implementation methods will be studied in Chap. 4.

- Not any loop can be implemented by means of an iterative combinational circuit. Consider a while-loop:

```
while a_condition loop operations end loop;
```

The loop body is executed as long as some condition (that can be modified by the operations) is true. If the maximum number of times that the condition will be true is either unknown or is a too large number, a sequential implementation (Chap. 4) must be considered.

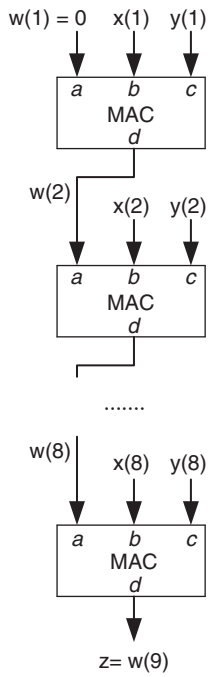


Fig. 2.70 Implementation of procedure calls

2.8.4 Procedure Calls

Procedure (or function) calls constitute a fundamental aspect of well-structured programs and can be associated to hierarchical circuit descriptions.

The following algorithm computes $z = x_1 \cdot y_1 + x_2 \cdot y_2 + \ldots + x_8 \cdot y_8$. For that it makes several calls to a previously defined procedure multiply and accumulate (MAC) to which it passes four parameters a, b, c, and d. The procedure call MAC(a, b, c, d) executes $d = a + b \cdot c$.

Algorithm 2.7 $z = x_1 \cdot y_1 + x_2 \cdot y_2 + \ldots + x_8 \cdot y_8$

```
w(1) = 0;
for i in 1 to 8 loop
  MAC(w(i), x(i), y(i), w(i+1));
end loop;
z = w(9);
```

Thus

$$w_2 = 0 + x_1 \cdot y_1 = x_1 \cdot y_1, w_2 = x_1 \cdot y_1 + x_2 \cdot y_2, w_3 = x_1 \cdot y_1 + x_2 \cdot y_2 + x_3 \cdot y_3, \ldots,$$
$$z = w_9 = x_1 \cdot y_1 + x_2 \cdot y_2 + x_3 \cdot y_3 + \ldots + x_8 \cdot y_8.$$

The corresponding circuit is shown in Fig. 2.70. Algorithm 2.7 is a for-loop to which is associated an iterative circuit. The loop body is a procedure call to which corresponds a component MAC whose functional specification is $d = a + b \cdot c$. This is an example of top-down hierarchical description: an iterative circuit structure (the top level) whose components are defined by their function and afterwards must be implemented (the down level).

2.8.5 Conclusion

There are several programming language constructs that can easily be translated to circuits. This fact justifies the use of formal languages to specify digital circuits, either classical programming languages such as C/C++ or specific HDL such as VHDL or Verilog. In this course VHDL will be used (Appendix A). The relation between programming language instructions and circuits also explains why it has been possible to develop software packages able to synthesize circuits starting from functional descriptions in some formal language.

2.9 Exercises

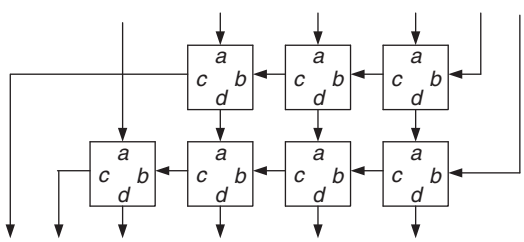
1. Synthesize with logic gates the function z of Table 2.3.
2. Generate Boolean expressions of functions f, g, and h of three variables x_2, x_1, and x_0 defined by the following table:

$x_2x_1x_0$	f	g	h
000	1	0	1
001	–	–	1
010	0	0	–
011	1	1	–
100	–	1	0
101	1	1	0
110	0	–	1
111	0	0	–

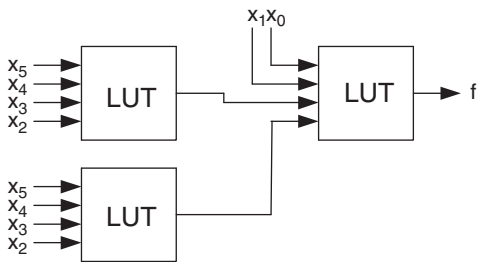
3. Simplify the following sets of cubes ($n = 4$):

   {0000, 0010, 01x1, 0110, 1000. 1010},
   {0001, 0011, 0100, 0101, 1100, 1110, 1011, 1010},
   {0000, 0010, 1000, 1010, 0101, 1101, 1111}.

4. The following circuit consists of seven identical components with two inputs a and b, and two outputs c and d. The maximum propagation time from inputs a or b to outputs c or d is equal to 0.5 ns. What is the maximum propagation time from any input to any output (in ns)?



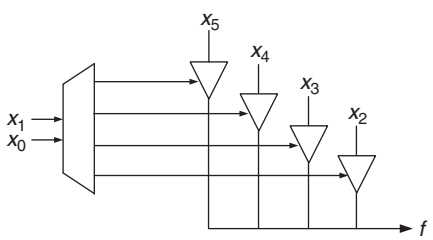
5. Compute an upper bound N_{max} and a lower bound N_{min} of the number N of functions that can be implemented by the following circuit.



6. Implement with MUX2-1 components the switching functions of three variables x_2, x_1, and x_0 defined by the following set of cubes:

   {11x, 101, 011},
   {111, 100, 010, 001},
   {1x1, 0x1}.

7. What set of cubes defines the function $f(x_5, x_4, x_3, x_2, x_1, x_0)$ implemented by the following circuit?



8. Minimize the following Boolean expression:

$$f(a,b,c,d) = a.b.c.d + \overline{a}.\overline{b} + a.b.\overline{c} + a.\overline{b} + a.\overline{c}.$$

9. Implement the circuits of Figs. 2.14, 2.16, and 2.17 with NAND gates.
10. Implement (2.34) with NAND gates.

References

Burch C (2005) Logisim. http://www.cburch.com/logisim/es/index.html
Karnaugh M (1953) The map method for synthesis of combinational logic circuits. Trans Inst Electr Eng (AIEE) Part I 72(9):593–599

http://www.springer.com/978-3-319-41197-2

Digital Systems
From Logic Gates to Processors
Deschamps, J.-P.; Valderrama, E.; Terés, L.
2017, XV, 241 p. 250 illus., 33 illus. in color., Hardcover
ISBN: 978-3-319-41197-2