CHAPTER 9

Boolean Algebras

9.1. Combinatorial Circuits

9.1.1. Introduction. At their lowest level digital computers handle only binary signals, represented with the symbols 0 and 1. The most elementary circuits that combine those signals are called *gates*. Figure 9.1 shows three gates: OR, AND and NOT.

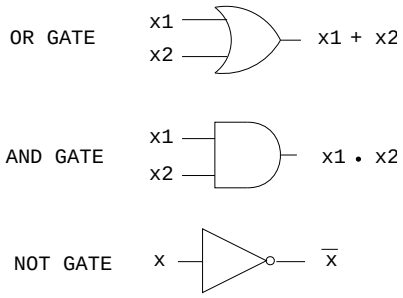

FIGURE 9.1. Gates.

Their outputs can be expressed as a function of their inputs by the following *logic tables*:

x_1	x_2	$x_1 + x_2$
1	1	1
1	0	1
0	1	1
0	0	0
OR GATE		

x_1	x_2	$x_1 \cdot x_2$
1	1	1
1	0	0
0	1	0
0	0	0
AND GATE		

x	$\overline{x}$
1	0
0	1
NOT GATE	

These are examples of *combinatorial circuits*. A combinatorial circuit is a circuit whose output is uniquely defined by its inputs. They do not have memory, previous inputs do not affect their outputs. Some combinations of gates can be used to make more complicated combinatorial circuits. For instance figure 9.2 is combinatorial circuit with the logic table shown below, representing the values of the *Boolean expression* $y = \overline{(x_1 + x_2) \cdot x_3}$.

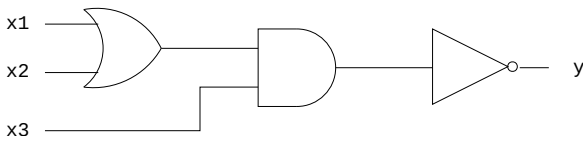

FIGURE 9.2. A combinatorial circuit.

x_1	x_2	x_3	$y = \overline{(x_1 + x_2) \cdot x_3}$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	1
0	1	1	0
0	1	0	1
0	0	1	1
0	0	0	1

However the circuit in figure 9.3 is *not* a combinatorial circuit. If $x_1 = 1$ and $x_2 = 0$ then y can be 0 or 1. Assume that at a given time $y = 0$. If we input a signal $x_2 = 1$, the output becomes $y = 1$, and

stays so even after x_2 goes back to its original value 0. That way we can store a bit. We can "delete" it by switching input x_1 to 0.

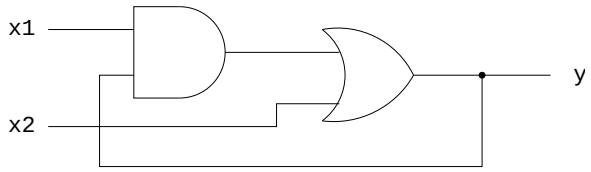

FIGURE 9.3. Not a combinatorial circuit.

9.1.2. Properties of Combinatorial Circuits. Here $\mathbb{Z}_2 = \{0, 1\}$ represents the set of signals handled by combinatorial circuits, and the operations performed on those signals by AND, OR and NOT gates are represented by the symbols $\cdot$, $+$ and $\overline{\ }$ respectively. Then their properties are the following (a, b, c are elements of $\mathbb{Z}_2$, i.e., each represents either 0 or 1):

1. Associative
$$(a + b) + c = a + (b + c)$$
$$(a \cdot b) \cdot c = a \cdot (b \cdot c)$$
2. Commutative
$$a + b = b + a$$
$$a \cdot b = b \cdot a$$
3. Distributive
$$a \cdot (b + c) = (a \cdot b) + (a \cdot c)$$
$$a + (b \cdot c) = (a + b) \cdot (a + c)$$
4. Identity
$$a + 0 = a$$
$$a \cdot 1 = a$$
5. Complement
$$a + \overline{a} = 1$$
$$a \cdot \overline{a} = 0$$

A system satisfying those properties is called a *Boolean algebra*.

Two Boolean expressions are defined to be *equal* is they have the same values for all possible assignments of values to their literals. *Example*: $\overline{x + y} = \overline{x} \cdot \overline{y}$, as shown in the following table:

x	y	$\overline{x+y}$	$\overline{x}\cdot\overline{y}$
1	1	0	0
1	0	0	0
0	1	0	0
0	0	1	1

9.1.3. Abstract Boolean Algebras. Here we deal with general Boolean algebras; combinatorial circuits are an example, but there are others.

A Boolean algebra $B = (S, \vee, \wedge, \bar{}, 0, 1)$ is a set S containing two distinguished elements 0 and 1, two binary operators $\vee$ and $\wedge$ on S, and a unary operator $\bar{}$ on S, satisfying the following properties (x, y, z are elements of S):

1. Associative
$$(x \vee y) \vee z = x \vee (y \vee z)$$
$$(x \wedge y) \vee z = x \wedge (y \wedge z)$$
2. Commutative
$$x \vee y = y \vee x$$
$$x \wedge y = y \wedge x$$
3. Distributive
$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$$
$$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$$
4. Identity
$$x \vee 0 = x$$
$$x \wedge 1 = x$$
5. Complement
$$x \vee \overline{x} = 1$$
$$x \wedge \overline{x} = 0$$

Example: $(\mathbb{Z}_2, +, \cdot, \bar{}, 0, 1)$ is a Boolean algebra.

Example: If U is a universal set and $\mathcal{P}(U)$= the power set of S (collection of subsets of S) then $(\mathcal{P}(U), \cup, \cap, \bar{}, \emptyset, U)$. is a Boolean algebra.

9.1.4. Other Properties of Boolean Algebras. The properties mentioned above define a Boolean algebra, but Boolean algebras also have other properties:

1. Idempotent
$$x \vee x = x$$
$$x \wedge x = x$$
2. Bound
$$x \vee 1 = 1$$
$$x \wedge 0 = 0$$
3. Absorption
$$x \vee xy = x$$
$$x \wedge (x \vee y) = x$$
4. Involution
$$\overline{\overline{x}} = x$$
5. 0 and 1
$$\overline{0} = 1$$
$$\overline{1} = 0$$
6. De Morgan's
$$\overline{x \vee y} = \overline{x} \wedge \overline{y}$$
$$\overline{x \wedge y} = \overline{x} \vee \overline{y}$$

For instance the first idempotent law can be proved like this: $x = x \vee 0 = x \vee x \wedge \overline{x} = (x \vee x) \wedge (x \vee \overline{x}) = (x \vee x) \wedge 1 = x \vee x$.

9.2. Boolean Functions, Applications

9.2.1. Introduction. A *Boolean function* is a function from $\mathbb{Z}_2^n$ to $\mathbb{Z}_2$. For instance, consider the *exclusive-or* function, defined by the following table:

x_1	x_2	$x_1 \oplus x_2$
1	1	0
1	0	1
0	1	1
0	0	0

The exclusive-or function can interpreted as a function $\mathbb{Z}_2^2 \to \mathbb{Z}_2$ that assigns $(1,1) \mapsto 0$, $(1,0) \mapsto 1$, $(0,1) \mapsto 1$, $(0,0) \mapsto 0$. It can also be written as a Boolean expression in the following way:

$$x_1 \oplus x_2 = (x_1 \cdot \overline{x}_2) + (\overline{x}_1 \cdot x_2)$$

Every Boolean function can be written as a Boolean expression as we are going to see next.

9.2.2. Disjunctive Normal Form. We start with a definition. A *minterm* in the symbols $x_1, x_2, \ldots, x_n$ is a Boolean expression of the form $y_1 \cdot y_2 \cdot \cdots \cdot y_n$, where each y_i is either x_i or $\overline{x}_i$.

Given any Boolean function $f : \mathbb{Z}_2^n \to \mathbb{Z}_2$ that is not identically zero, it can be represented

$$f(x_1, \ldots, x_n) = m_1 + m_2 + \cdots + m_k \,,$$

where $m_1, m_2, \ldots, m_k$ are all the minterms $m_i = y_1 \cdot y_2 \cdot \cdots \cdot y_n$ such that $f(a_1, a_2, \ldots, a_n) = 1$, where $y_j = x_j$ if $a_j = 1$ and $y_j = \overline{x}_j$ if $a_j = 0$. That representation is called *disjunctive normal form* of the Boolean function f.

Example: We have seen that the exclusive-or can be represented $x_1 \oplus x_2 = (x_1 \cdot \overline{x_2}) + (\overline{x_1} \cdot x_2)$. This provides a way to implement the exclusive-or with a combinatorial circuit as shown in figure 9.4.

9.2.3. Conjunctive Normal Form. A *maxterm* in the symbols $x_1, x_2, \ldots, x_n$ is a Boolean expression of the form $y_1 + y_2 + \cdots + y_n$, where each y_i is either x_i or $\overline{x}_i$.

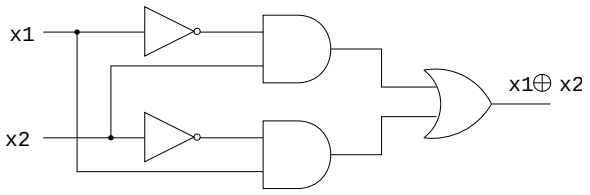

FIGURE 9.4. Exclusive-Or.

Given any Boolean function $f : \mathbb{Z}_2^n \to \mathbb{Z}_2$ that is not identically one, it can be represented

$$f(x_1, \ldots, x_n) = M_1 \cdot M_2 \cdot \cdots \cdot M_k \,,$$

where $M_1, M_2, \ldots, M_k$ are all the maxterms $M_i = y_1 + y_2 + \cdots + y_n$ such that $f(a_1, a_2, \ldots, a_n) = 0$, where $y_j = x_j$ if $a_j = 0$ and $y_j = \overline{x}_j$ if $a_j = 1$. That representation is called *conjunctive normal form* of the Boolean function f.

Example: The conjunctive normal form of the exclusive-or is

$$x_1 \oplus x_2 = (x_1 + x_2) \cdot (\overline{x}_1 + \overline{x}_2) \,.$$

9.2.4. Functionally Complete Sets of Gates. We have seen how to design combinatorial circuits using AND, OR and NOT gates. Here we will see how to do the same with other kinds of gates. In the following gates will be considered as functions from $\mathbb{Z}_2^n$ into $\mathbb{Z}_2$ intended to serve as building blocks of arbitrary boolean functions.

A set of gates $\{g_1, g_2, \ldots, g_k\}$ is said to be *functionally complete* if for any integer n and any function $f : \mathbb{Z}_2^n \to \mathbb{Z}_2$ it is possible to construct a combinatorial circuit that computes f using only the gates $g_1, g_2, \ldots, g_k$. *Example*: The result about the existence of a disjunctive normal form for any Boolean function proves that the set of gates $\{\text{AND}, \text{OR}, \text{NOT}\}$ is functionally complete. Next we show other sets of gates that are also functionally complete.

1. The set of gates $\{\text{AND}, \text{NOT}\}$ is functionally complete. Proof: Since we already know that $\{\text{AND}, \text{OR}, \text{NOT}\}$ is functionally complete, all we need to do is to show that we can compute $x + y$ using only AND and NOT gates. In fact:

   $$x + y = \overline{\overline{x} \cdot \overline{y}} \,,$$

   hence the combinatorial circuit of figure 9.5 computes $x + y$.

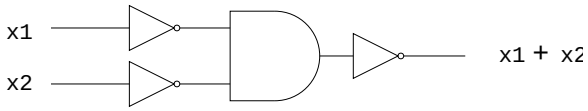

FIGURE 9.5. OR with AND and NOT.

2. The set of gates $\{\text{OR}, \text{NOT}\}$ is functionally complete. The proof is similar:
$$x \cdot y = \overline{\overline{x} + \overline{y}},$$
hence the combinatorial circuit of figure 9.6 computes $x+y$.

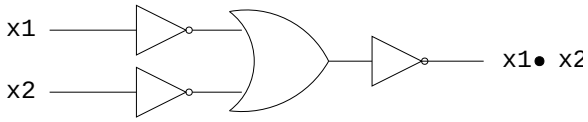

FIGURE 9.6. AND with OR and NOT.

3. The gate NAND, denoted $\uparrow$ and defined as
$$x_1 \uparrow x_2 = \begin{cases} 0 & \text{if } x_1 = 1 \text{ and } x_2 = 1 \\ 1 & \text{otherwise,} \end{cases}$$
is functionally complete.

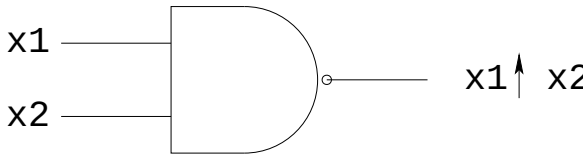

FIGURE 9.7. NAND gate.

Proof: Note that $x \uparrow y = \overline{x \cdot y}$. Hence $\overline{x} = \overline{x \cdot x} = x \uparrow x$, so the NOT gate can be implemented with a NAND gate. Also the OR gate can be implemented with NAND gates: $x+y = \overline{\overline{x} \cdot \overline{y}} = (x \uparrow x) \uparrow (y \uparrow y)$. Since the set $\{\text{OR}, \text{NOT}\}$ is functionally complete and each of its elements can be implemented with NAND gates, the NAND gate is functionally complete.

9.2.5. Minimization of Combinatorial Circuits. Here we address the problems of finding a combinatorial circuit that computes a given Boolean function with the minimum number of gates. The idea

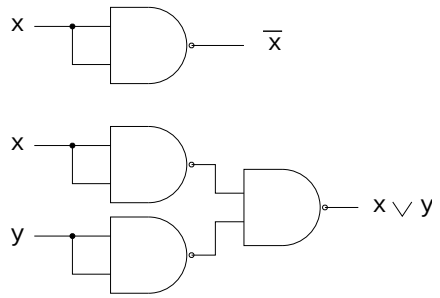

FIGURE 9.8. NOT and OR functions implemented with NAND gates.

is to simplify the corresponding Boolean expression by using algebraic properties such as $(E \cdot a) + (E \cdot \overline{a}) = E$ and $E + (E \cdot a) = E$, where E is any Boolean expression. For simplicity in the following we will represent $a \cdot b$ as ab, so for instance the expressions above will look like this: $Ea + E\overline{a} = E$ and $E + Ea = E$.

Example: Let $F(x, y, z)$ the Boolean function defined by the following table:

x	y	z	f(x,y,z)
1	1	1	1
1	1	0	1
1	0	1	0
1	0	0	1
0	1	1	0
0	1	0	0
0	0	1	0
0	0	0	0

Its disjunctive normal form is $f(x, y, z) = xyz + xy\overline{z} + x\overline{yz}$. This function can be implemented with the combinatorial circuit of figure 9.9.

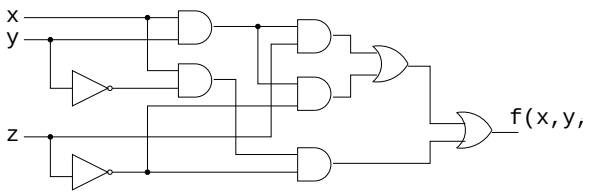

FIGURE 9.9. A circuit that computes $f(x, y, z) = xyz + xy\overline{z} + x\overline{yz}$.

But we can do better if we simplify the expression in the following way:

$$\begin{aligned} f(x, y, z) &= \overbrace{xyz + xy\overline{z}}^{xy} + x\overline{yz} \\ &= xy + x\overline{yz} \\ &= x(y + \overline{yz}) \\ &= x(y + \overline{y})(y + \overline{z}) \\ &= x(y + \overline{z}) \,, \end{aligned}$$

which corresponds to the circuit of figure 9.10.

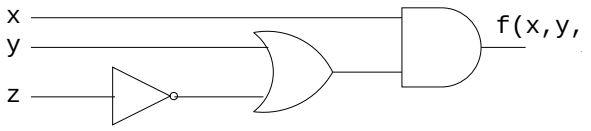

FIGURE 9.10. A simpler circuit that computes $f(x, y, z) = xyz + xy\overline{z} + x\overline{yz}$.

9.2.6. Multi-Output Combinatorial Circuits. *Example*: *Half-Adder*. A half-adder is a combinatorial circuit with two inputs x and y and two outputs s and c, where s represents the sum of x and y and c is the carry bit. Its table is as follows:

x	y	s	c
1	1	0	1
1	0	1	0
0	1	1	0
0	0	0	0

So the sum is $s = x \oplus y$ (exclusive-or) and the carry bit is $c = x \cdot y$. Figure 9.11 shows a half-adder circuit.

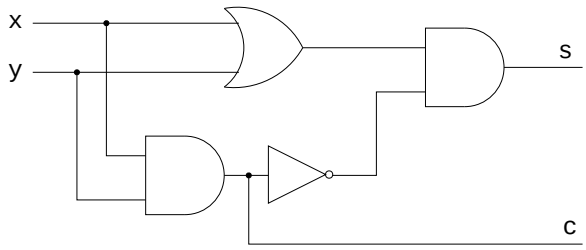

FIGURE 9.11. Half-adder circuit.