

E.1 BOOLEAN ALGEBRA

Boolean algebra deals with variables and constants that take only one of two values: 1 or 0. This algebra is a suitable way to represent information in a computer, which is made of a collection of signals that can be only in one of the two states: on or off.

E.1.1 Constants, variables, and operators

We use constants, variables, and operators in Boolean algebra.

Constants

There are only two constants: 1 and 0. The value of 1 is associated with the logical value *true*; the value 0 is associated with the logical value *false*.

Variables

We use letters such as x , y , and z to represent variables. Boolean variables can take only the values 0 or 1.

Operators

We use three basic operators: NOT, AND, and OR. We use a prime to represent NOT, a dot to represent AND, and a plus sign to represent OR, as shown below:

$$x' \rightarrow \text{NOT } x$$

$$x \cdot y \rightarrow x \text{ AND } y$$

$$x + y \rightarrow x \text{ OR } y$$

An operator takes one or two values and creates one output value. The first operator, NOT, is a unary operator that takes only one value; the other two, AND and OR, are binary operators that take two values. Note that the choice of operators are arbitrary. We can construct all gates from the NAND gate (explained later).

E.1.2 Expressions

An expression is a combination of Boolean operators, constants, and variables. The following shows some Boolean expressions.

$$\begin{array}{c} 0 \\ x + 1 + y \end{array}$$

$$\begin{array}{c} x \\ x \cdot (y + z) \end{array}$$

$$\begin{array}{c} x \cdot 1 \\ x + y + z \end{array}$$


$$\begin{array}{c} x + 0 \\ x \cdot y \cdot z \cdot t \end{array}$$

E.1.3 Logic gates

A logic gate is an electronic device that normally takes 1 to N inputs and creates one output. In this appendix, however, we use gates with only one or two inputs for simplicity. The logical value of the output is determined by the expression representing the gate and the input values. A variety of logic gates are commonly used in digital computers. Figure E.1 shows the symbols for the eight most common gates, their truth tables (See Chapter 4), and the expressions that can be used to find the output when the input or inputs are given.


Figure E.1 Symbols and truth table for common gates

Buffer




x	x
0	0
1	1

NOT




x	x'
0	1
1	0

AND




x	y	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

OR




x	y	$x + y$
0	0	0
0	1	1
1	0	1
1	1	1

NAND




x	y	$(x \cdot y)'$
0	0	1
0	1	1
1	0	1
1	1	0

NOR




x	y	$(x + y)'$
0	0	1
0	1	0
1	0	0
1	1	0

XOR



x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

XNOR



x	y	$(x \oplus y)'$
0	0	1
0	1	0
1	0	0
1	1	1

- **Buffer.** The first gate is just a buffer, in which the input and the output are the same. If the input is 0, the output is 0; if the input is 1, the output is 1. The buffer only amplifies the input signal.

- ❑ **NOT.** The NOT gate is the implementation of the NOT operator. The output of this gate is the complement of the input. If the input is 1, the output is 0; if the input is 0, the output is 1.
- ❑ **AND.** The AND gate is the implementation of the AND operator. It takes two inputs and creates one output. The output is 1 if both inputs are 1s, otherwise it is 0. Sometimes the AND operator is referred to as *product*.
- ❑ **OR.** The OR gate is the implementation of the OR operator. It takes two inputs and creates one output. The output is 1 if any of the inputs, or both of them, is 1, otherwise it is 0. Sometimes the OR gate is referred to as *sum*.
- ❑ **NAND.** The NAND gate is a logical combination of an AND gate followed by a NOT gate. The reason for its existence can be explained when we discuss the actual implementation of these gates. The output of a NAND gate is the complement of the corresponding AND gate if the inputs to two gates are the same.
- ❑ **NOR.** The NOR gate is a logical combination of an OR gate followed by a NOT gate. The reason for its existence can also be explained when we discuss the actual implementation of these gates. The output of a NOR gate is the complement of the corresponding OR gate if the inputs to two gates are the same.
- ❑ **XOR.** The XOR (exclusive-OR) gate is defined by the expression $(x \cdot y' + x' \cdot y)$, which is normally represented as $(x \oplus y)$. The output of this gate is 1 when the two inputs are different and is 0 when the inputs are the same. One can say that this is a more restricted OR gate. The output of an XOR gate is the same as the OR gate except that, if the two inputs are 1s, the output is 0.
- ❑ **XNOR.** The XNOR (exclusive-NOR) gate is defined by the expression $(x \cdot y' + x' \cdot y)'$ which is normally represented as $(x \oplus y)'$. It is the complement of the XOR gate. The output of this gate is 1 when the two inputs are the same and 0 when the inputs are different. One can say that this represents the logical idea of equivalence: only if the two inputs are equal is the output 1.

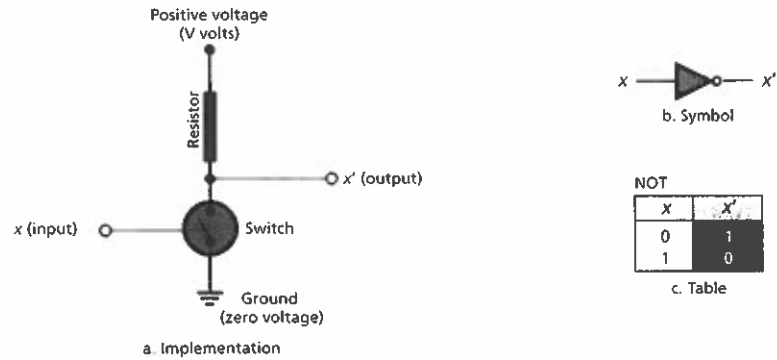
Implementation of gates

The logic gates discussed in the previous section can be physically implemented using electronic switches (transistors). The most common implementation uses only three gates: NOT, NAND, and NOR. A NAND gate uses less components than an AND gate. This is also true for the NOR gate versus the OR gate. As a result, NAND and NOR gates have become the common standard in the industry. We only discuss these three implementations. Although we show simple switches in this discussion, we need to know that, in practice, switches are replaced by transistors. A transistor, when used in gates, behaves like a switch. The switch can be opened or closed by applying the appropriate voltage to the input. Several different technologies are used to implement these transistors, but we leave this discussion to books on electronics.

Implementation of the NOT gate

The NOT gate can be implemented with an electronic switch, a voltage source, and a resistor as shown in Figure E.2.

Figure E.2 Implementation of the NOT gate



The input to the gate is a control signal that holds the switch open or closed. An input signal of 0 holds the switch open, while an input signal of 1 closes the switch. The output is the voltage at the point before the switch (output terminal). If the value of this voltage is positive (V volts), the output is interpreted as 1: if the voltage is 0 (or below a threshold), the output is interpreted as 0. When the switch is open, there is no current through the resistor, and therefore no voltage drop. The output voltage is V (interpreted as logic 1). Closing the switch grounds the output terminal and makes its voltage 0 (or almost 0), which is interpreted as logic 0. Note that the behavior of the circuit matches the values shown in the table.

To implement a NOT gate, we need only one electronic switch.

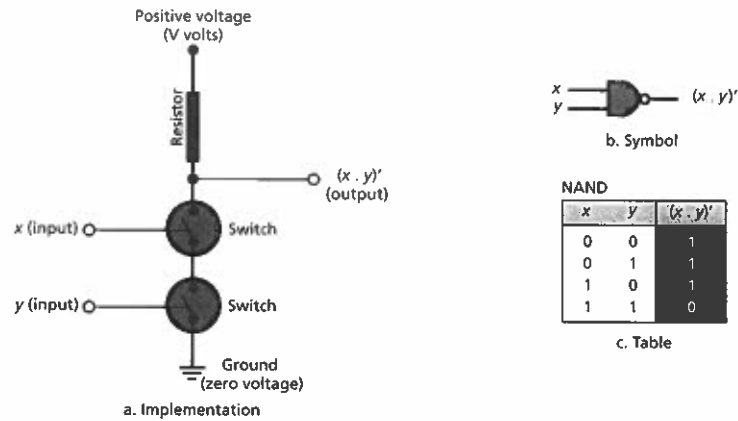
Implementation of the NAND gate

The NAND gate can be implemented using two switches in series (two inputs). For the current to flow through the circuit from the positive terminal to the ground, both switches must be closed—that is, both inputs must be 1s. In this case, the voltage of the output terminal is zero because it is grounded (logic 0). If one of the switches or both switches are open—that is, where the inputs are 00, 01, or 10—no current flows through the resistor. There is thus no voltage drop across the resistor and the voltage at the output terminal is V (logic 1).

Figure E.3 shows the implementation of the NAND gate. The behavior of the circuit matches the values shown in the table. Note that if an AND gate is needed, it can be made from a NAND gate followed by a NOT gate.

To implement a NAND gate, we need two electronic switches that are connected in series.

Figure E.3 Implementation of the NAND gate

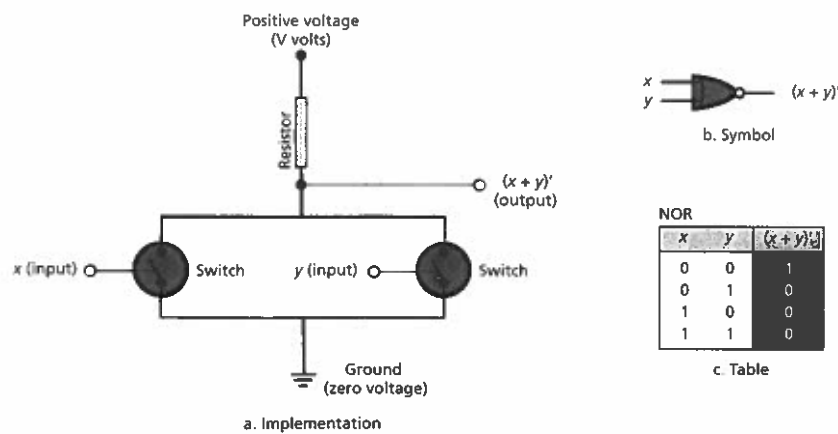
**Implementation of the NOR gate**

The NOR gate can also be implemented using two switches in parallel (two inputs). If both switches are open, then the current does not flow through the resistor. In this case, there is no voltage drop across the resistor, which means the output terminal holds the voltage V (logic 1). If either or both of the switches are closed, the output terminal is grounded and the output voltage is zero (logic 0).

Figure E.4 shows the implementation of the NOR gate. The behavior of the circuit matches the values in the table. Note that if an OR gate is needed, it can be simulated using a NOR gate followed by a NOT gate.

To implement a NOR gate, we need two electronic switches that are connected in parallel.

Figure E.4 Implementation of the NOR gate



E.1.4 Axioms, theorems, and Identities

To be able to work with Boolean algebra, we need to have some rules. The rules in Boolean algebra are divided into three broad categories: *axioms*, *theorems*, and *identities*.

Axioms

Boolean algebra, like any other algebra, uses some rules, called **axioms**: they cannot be proved. Table E.1 shows the axioms for Boolean algebra.

Table E.1 Axioms for Boolean algebra

	Related to NOT	Related to AND	Related to OR
1	$x = 0 \rightarrow x' = 1$		
2	$x = 1 \rightarrow x' = 0$		
3		$0 \bullet 0 = 0$	$0 + 0 = 0$
4		$1 \bullet 1 = 1$	$1 + 1 = 1$
5		$1 \bullet 0 = 0 \bullet 1 = 0$	$1 + 0 = 0 + 1 = 1$

Theorems

Theorems are rules that we prove using the axioms, although we must leave the proofs to textbooks on Boolean algebra. Table E.2 shows some theorems used in Boolean algebra.

Table E.2 Basic theorems for Boolean algebra

	Related to NOT	Related to AND	Related to OR
1	$(x')' = x$		
2		$0 \bullet x = 0$	$0 + x = x$
3		$1 \bullet x = x$	$1 + x = 1$
4		$x \bullet x = x$	$x + x = x$
5		$x \bullet x' = 0$	$x + x' = 1$

Identities

We can also derive many identities using the axioms and the theorems. We list only the most common in Table E.3, although we must leave the proofs to textbooks on Boolean algebra.

Table E.3 Basic Identities related to OR and AND operators

	Description	Related to AND	Related to OR
1	Commutativity	$x \cdot y = y \cdot x$	$x + y = y + x$
2	Associativity	$x \cdot (y \cdot z) = (x \cdot y) \cdot z$	$x + (y + z) = (x + y) + z$
3	Distributivity	$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$	$x + (y \cdot z) = (x + y) \cdot (x + z)$
4	De Morgan's Rules	$(x \cdot y)' = x' + y'$	$(x + y)' = x' \cdot y'$
5	Absorption	$x \cdot (x' + y) = x \cdot y$	$x + (x' \cdot y) = x + y$

De Morgan's Rules play a very important role in logic design, as we will see shortly. They can be extended to more than one variables. For example, we can have the following two identities for three variables:

$$(x + y + z)' = x' \cdot y' \cdot z'$$

$$(x \cdot y \cdot z)' = x' + y' + z'$$

E.1.5 Boolean functions

We define a **Boolean function** as a function with n Boolean input variables and one Boolean output variable, as shown in Figure E.5.

Figure E.5 A Boolean function



A function can be represented either by a truth table or an expression. The truth table for a function has 2^n rows and $n + 1$ columns, in which the first n columns define the possible values of the variables and the last column defines the value of the function's output for the combination of the values defined in the first n columns.

Figure E.6 shows the truth tables and expression representation for two functions F_1 and F_2 . Although the truth table representation is unique, a function can be represented by different expressions. We have shown two of the expressions for each function. Note that the second expressions are shorter and simpler. Later we show that we need to simplify the expressions to make the implementation more efficient.

Figure E.6 Examples of table-to-expression transformation

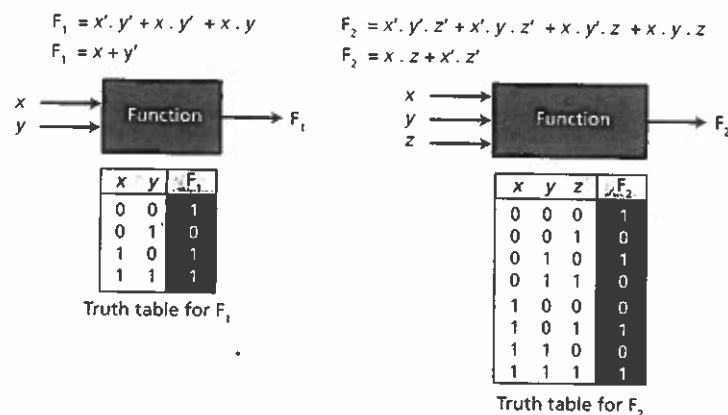


Table-to-expression transformation

The specification of a function is normally given by a truth table (see Chapter 4). To implement the function using logic gates (as discussed earlier), we need to find an expression for the truth table. This can be done in two ways.

Sum of products

The first method of changing a truth table into an expression is referred to as the *sum of products* method. A sum of products representation of a function is made of up to 2^n terms in which each term is called a **minterm**. A minterm is a product (ANDing) of all variables in a function in which each variable appears only once. For example, in a three-variable function, we can have eight minterms, such as $x' \cdot y' \cdot z'$ or $x \cdot y' \cdot z'$. Each term represents one row in the truth value. If the value of a variable is 0, the complement of the variable appears in the term: if the value of the variable is 1, the variable itself appears in the term. To transform a truth table to a sum of product representation, we use the following strategy:

1. Find the minterms for each row for which the function has a value of 1.
2. Use the sum (ORing) of the terms in step 1.

Product of sums

The second method of changing a truth table to an expression is referred to as the *product of sums* method. A product of sum representation of a function is made of up to 2^n terms in which each term is called a **maxterm**. A maxterm is a sum (ORing) of all variables in a function in which each variable appears only once. For example, in a three-variable function, we can have eight maxterms such as $x' + y' + z'$ or $x + y' + z'$. To transform a truth table to a product of sum representation, we use the following strategy:

1. Find the minterms for each row for which the function has a 0 value.
2. Find the complement of the sum of the terms in step 1.
3. Use De Morgan's rules to change minterms to maxterms.

Example E.1

Figure E.7 shows how we create the sum of products and product of sums for the function F_1 and F_2 in Figure E.6.

Figure E.7 Example E.1

x	y	F_1	
0	0	1	$x' \cdot y'$
0	1	0	$x' \cdot y$
1	0	1	$x \cdot y'$
1	1	1	$x \cdot y$

Truth table for F_1

Sum of products

$$F_1 = x' \cdot y' + x \cdot y' + x \cdot y$$

Product of sums

$$F_1 = (x' + y)' = (x + y)$$

x	y	z	F_2	
0	0	0	1	$x' \cdot y' \cdot z'$
0	0	1	0	$x' \cdot y' \cdot z$
0	1	0	1	$x' \cdot y \cdot z'$
0	1	1	0	$x' \cdot y \cdot z$
1	0	0	0	$x \cdot y' \cdot z'$
1	0	1	1	$x \cdot y' \cdot z$
1	1	0	0	$x \cdot y \cdot z'$
1	1	1	1	$x \cdot y \cdot z$

Truth table for F_2

Sum of products

$$F_2 = x' \cdot y' \cdot z' + x' \cdot y \cdot z' + x \cdot y' \cdot z + x \cdot y \cdot z$$

Product of sums

$$\begin{aligned} F_2 &= (x' \cdot y' \cdot z' + x' \cdot y \cdot z' + x \cdot y' \cdot z + x \cdot y \cdot z)' \\ &= (x' \cdot y' \cdot z)' \cdot (x' \cdot y \cdot z)' \cdot (x \cdot y' \cdot z')' \cdot (x \cdot y \cdot z)' \\ &= (x + y + z) \cdot (x + y' + z') \cdot (x' + y + z) \cdot (x' + y' + z) \end{aligned}$$

The sum of products is directly made from the table, but the product of sums needs the use of De Morgan's rules. Note that sometimes the first method gives the shorter expression and sometimes the second one.

E.1.6 Function simplification

Although we can implement a Boolean function using the logic gates discussed before, it is normally not efficient. The direct implementation of a function requires more gates. The number of gates could be reduced if we can carry out simplification. Traditionally one uses two methods of simplification: the algebraic method using Karnaugh maps, and the Quine-McCluskey method.

Algebraic method

We can simplify a function using the axioms, theorems, and identities discussed before. For example, we can simplify the first function (F_1) in Figure E.7, as shown below:

$$\begin{aligned} F_1 &= x' \cdot y' + x \cdot y' + x \cdot y \\ &= (x' + x) \cdot y' + x \cdot y \\ &= 1 \cdot y' + x \cdot y \\ &= y' + x \cdot y \\ &= y' + y \cdot x \\ &= y' + x \\ &= x + y' \end{aligned}$$

Identity 3 (distributivity) for AND

Theorem 5 for OR

Theorem 3 for AND

Theorem 1 (commutativity) for AND

Identity 5 (absorption)

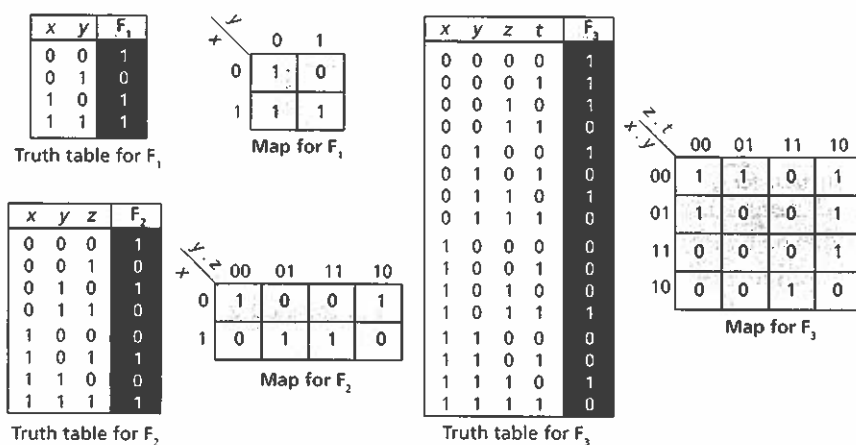
Theorem 1 (commutativity) for OR

This means that if the non-simplified version needs eight gates, the simplified version needs only two gates, one NOT and one OR.

Karnaugh map method

Another simplification method involves the use of a **Karnaugh Map**. This method can normally be used for functions of up to four variables. A map is a matrix of 2^n cells in which each cell represents one of the values of the function. The first point that deserves attention is to fill up the map correctly. Contrary to expectations, the map is not always filled up row by row or column by column: it is filled up according to the value of variables as shown on the map. Figure E.8 shows an example where $n = 2, 3$, or 4.

Figure E.8 Construction of Karnaugh Maps



In the truth table, we use the function values from the top to the bottom of the truth table. The map is filled up one by one, but the order of rows are 1, 2, 4, 3. In each row, the columns are filled up one by one, but the order of the columns are 1, 2, 4, 3. The fourth row comes before the third row: the fourth column comes before the third. This arrangement is needed to allow the maximum of simplification.

Sum of products

The simplification can be done to create sum of products terms. When we simplify a function in this way we use minterms with value of 1. To create an efficient expression, we first combine adjacent minterm cells. Note that adjacency can also include wrap-around of bits.

Example E.2

Figure E.9 shows the sum of products simplification for our first function. The 1s in the second row is the entire x domain. The 1s in the first column are the entire y' domain. The resulting simplified function is $F_1 = (x) + (y')$. The figure also shows the implementing using one OR gate and one NOT gate.

Figure E.9 Example E.2

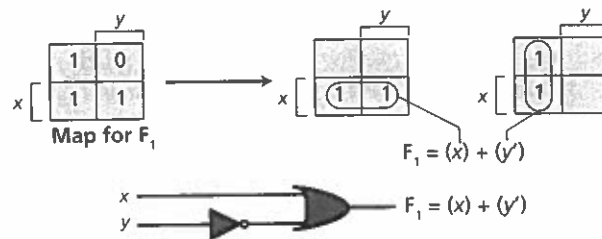
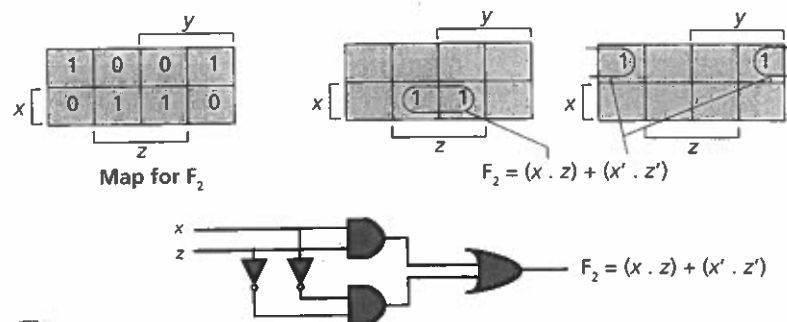
**Example E.3**

Figure E.10 shows the sum of products simplification for our second function. The 1s in the second row are the intersection of x and z domains, which is represented as $(x \cdot z)$. The 1s in the first row are the intersection of x' and z' domains, which is represented as $(x' \cdot z')$. The resulting simplified function is $F_2 = (x \cdot z) + (x' \cdot z')$. The figure also shows the implementing using one OR gate, two AND gates, and two NOT gates.

Figure E.10 Example E.3

**Product of sums**

The simplification can be done using the product of sums methods. When we simplify a function in this way, we need to use maxterms. To create an efficient expression, we first combine the adjacent minterm cells. However, the function obtained in this way is the complement of the function we are looking for: we need to use the De Morgan's rules to find our function.

Example E.4

Figure E.11 shows a product of sums simplification for our first function. Note that in this case the implementation is exactly the same as Figure E.11, but this is not always the case. Also note that our function has only one term: we need no AND gate.

Figure E.11 Example E.4

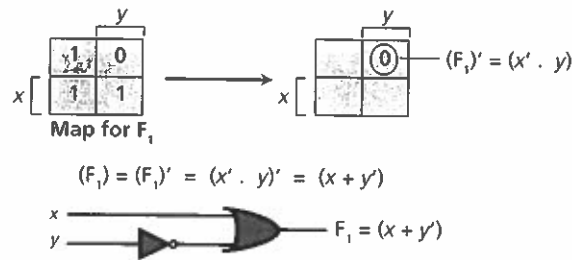
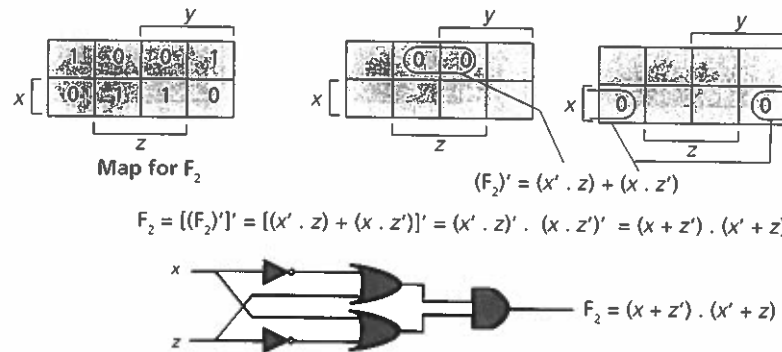
**Example E.5**

Figure E.12 shows the product-of-sums simplification for our second function. Note that the process gives us $(F_2)'$, so we need to apply the De Morgan's rules to find F_2 . The figure also shows the implementation using two NOT gates, two OR gates, and one AND gate. This implementation is less efficient than the one we found with minterms. We should always use the one which is more efficient.

Figure E.12 Example E.5

**E.2 LOGIC CIRCUITS**

A computer is normally built out of standard components that we collectively refer to as **logic circuits**. Logic circuits are divided into two broad categories, known as *combinational circuits* and *sequential circuits*. We briefly discuss each category here and give some examples.

E.2.1 Combinational circuits

A **combinational circuit** is a circuit made up of combination of logic gates with n inputs and m outputs. Each output at any time entirely depends on all given inputs.

In a combinational circuit, each output at any time depends entirely on all inputs.

Figure E.13 shows the block diagram of a combinational circuit with n inputs and m outputs. Comparing Figure E.13 and Figure E.5, we can say that a combinational circuit with m outputs can be thought as m functions, a function for each output.

Figure E.13 A combinational circuit

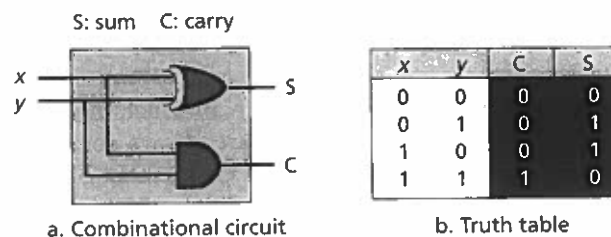


The outputs of a combinational circuit is normally defined by a truth table. However, the truth table needs to have m outputs.

Half adder

A simple example of a combinational circuit is a **half adder**, an adder that can only add two bits. A half-adder is a combinational circuit with two inputs and two outputs. The two inputs define the two bits to be added. The first output is the sum of the two bits, while the second output is the carry bit that needs to be propagated to the next adder. Figure E.14 shows a half-adder with its truth table and the logic gates used to make the circuit.

Figure E.14 Half adder



The sum of two bits can be achieved using an XOR gate; the carry can be achieved using an AND gate.

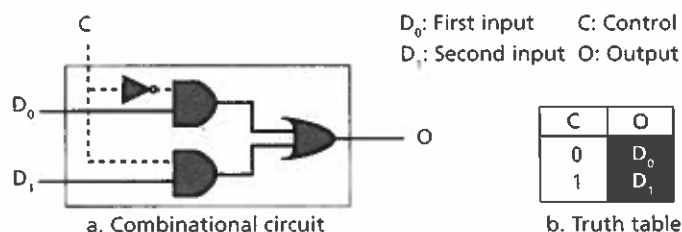
Multiplexer

A **multiplexer** is a combinational circuit with n inputs and only one output. The n inputs are made up of D data inputs and C control inputs ($n = D + C$). At any time, the multiplexer routes one of its D data inputs to its single data output. The selection is based on the value of control bits. To select one of the D data inputs, we need $C = \log_2 D$ control bits. If $D = 2$, at any time only one of the data inputs is routed to the output. The control input is only one

bit. If the control input is 0, the first data input is directed to the output: if the control input is 1, the second input is routed to the output.

Figure E.15 shows the truth table and the circuit for a 2×1 multiplexer. Note that the circuit actually has three inputs and one output: the control input is considered one of the inputs.

Figure E.15 Multiplexer



Note that the truth table here is very simplified: the output depends only on the control input but the value of the output, however, is one of the two data inputs.

E.2.2 Sequential circuits

A combinational circuit is memoryless: it does not remember its previous output. At any moment the output depends on the current input. A **sequential circuit**, on the other hand, includes the concept of memory in the logic. The memory enables the circuit to remember its current state to be used in the future; the future state can be dependent on the current state.

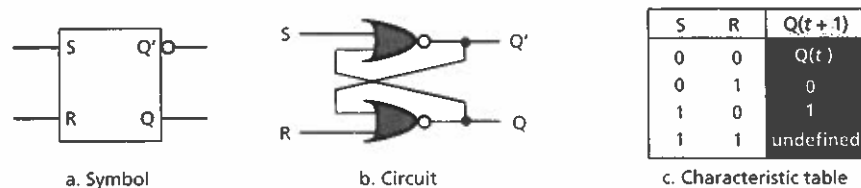
Flip-flops

To add the idea of memory to the combinational circuit, a storage element called a **flip-flop** was invented that can hold one bit of information. A set of flip-flops can be used to hold a set of bits.

SR flip-flops

The simplest type of flip-flop is called an **SR flip-flop**, in which there are two inputs S (set) and R (reset) and two outputs Q and Q' , which are always complements of each other. Figure E.16 shows the symbol, the circuit, and the characteristic table of an SR flip-flop. Note that the characteristic table is different from the truth tables we have used for combinational circuits. The characteristic table shows the next output, $Q(t+1)$ based on the current output, $Q(t)$ and the input.

Figure E.16 SR flip-flop



The characteristic table shows that if both S and R are zero, $Q(t+1) = Q(t)$. The next output will be the same as the current output. If S is 0 and R is 1, $Q(t+1) = 0$, which means the output will be reset ($R = 1$). If S is 1 and R is 0, $Q(t+1) = 1$, which means that the output will be set. However, if both S and R are 1s, the next output is unpredictable (undefined). Note that we have not shown the value of Q' in the characteristic table, because it is always the complement of Q .

An SR flip flop can be used as a set-reset device. For example, if the output is connected to an electric sounder, the alarm can be set by letting $R = 0$ and $S = 1$. After setting, the alarm continues sounding until it is reset by setting $R = 1$ and $S = 0$. The only flaw in this design is that R and S should not simultaneously be 1s.

To understand the behavior of the SR flip flop we need to create its truth table. However, note that we now have three inputs and one output (Q and Q' are independent). Table E.4 shows the truth table for this flip flop.

D flip-flop

The SR flip flop cannot be used as a 1-bit memory, as it needs two inputs instead of one. A small modification to the SR flip flop can create a **D flop** (D stands for data). Figure E.17 shows the symbol and characteristics of a D flip flop.

Note that the output of D flip-flop is the same as its input. However, the output remains as it is until the new input is given. This means that it memorizes its input states.

JK flip-flop

To remove the undefined state from the SR flip flop, the **JK flip flop** was invented (JK stands for Jack Kilby, who invented integrated circuits). Adding two AND gates to an SR flip flop creates a JK flip flop that has no undefined state. Figure E.18 shows the JK flip flop and its characteristic table.

Table E.4 Truth table for an SR flip-flop

S	R	$Q(t)$	$Q(t+1)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	undefined
1	1	1	undefined

Figure E.17 D flip-flop

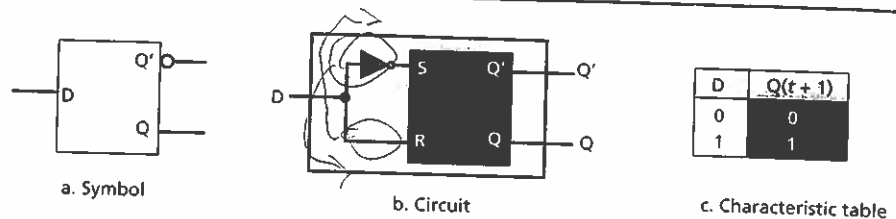
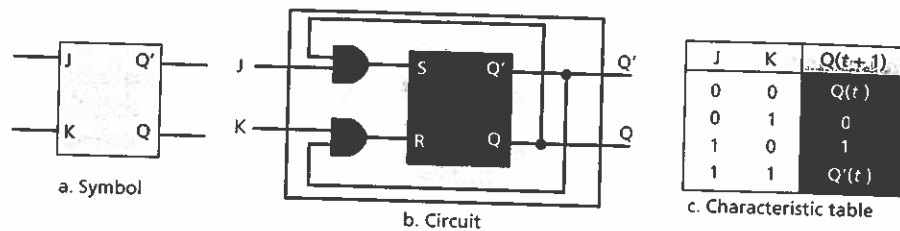


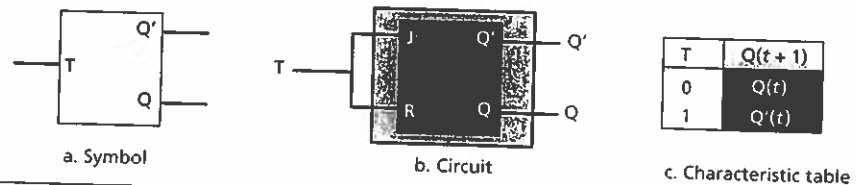
Figure E.18 JK flip-flop



T flip-flop

Another common type of flip-flop is the **T flip-flop** (T stands for *toggle*). This flip-flop can be made by connecting the two inputs of an JK flip-flop together and calling it the T input. This input toggles the state of the flip-flop: if the input is 0, the next state is the same as the current state. If the input is 1, the next state is the complement of current state. Figure E.19 shows the symbol, circuit, and characteristic table of the T flip-flop.

Figure E.19 T flip-flop



Synchronous versus Asynchronous

The flip-flops we have discussed so far are all referred to as **asynchronous devices**: the transition from one state to another can happen only when there is a change in the input. Digital computers, on the other hand, are **synchronous devices**. A central clock in the computer controls the timing of all logic circuits. The clock creates a signal—a series of pulses with an

exact pulse width—that coordinates all events. A simple event takes place only at the ‘tick’ of this clock signal.

Figure E.20 shows an abstract idea of a clock signal. We call it *abstract* because in reality no electronic circuit can generate a signal with perfectly sharp impulses, but the signal shown here is sufficient for our discussion.

Figure E.20 Clock pulses

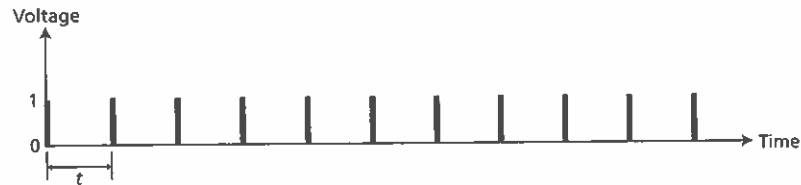
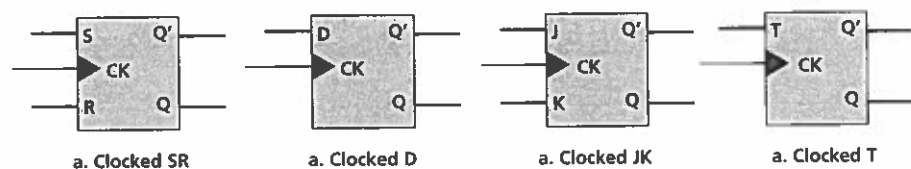


Figure E.21 Clocked flip-flops



A flip-flop can be synchronous if we add one more input to the circuit: the clock input. The clock input can be ANDed with every input to gate the input so that it is effective only when the clock pulse is present. Figure E.21 shows the symbols for the clocked versions of all four flip-flops types we discussed. Figure E.22 shows the circuit of an SR flip-flop with a clock signal. The other flip-flops have the same additional circuitry.

Register

As the first application of a synchronous (clocked) sequential circuit, we will introduce a simplified version of a **register**. A register is an n -bit storage device that stores its data between consecutive clock pulses. At the trigger of the clock, the old data is discarded and replaced by the new data.

Figure E.23 shows a 4-bit register in which each cell is composed of a D flip-flop. Note that the clock input is common for all cells. We have rotated our previous symbols to make the connections simpler.

Figure E.22 Circuit of clocked SR flip-flop

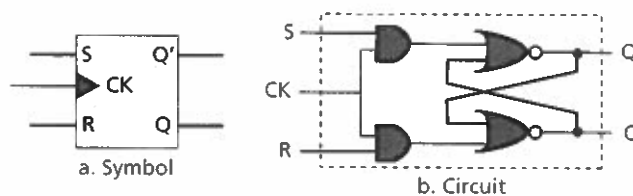
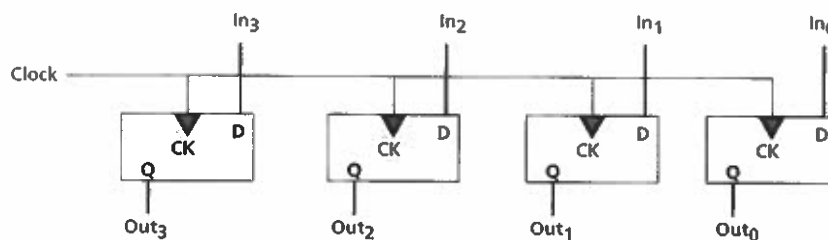


Figure E.23 A 4-bit register



Digital counter

An n -bit **digital counter** counts from 0 to $2^n - 1$. For example, when $n = 4$, the output of the counter is 0000, 0001, 0010, 0011, ..., 1111, so it counts from 0 to 15. An n -bit counter can be made out of n T flip-flops. At the start, the counter represents 0000. The count enable line—see Figure E.24—carries a sequence of 1s: the data (pulse) to be counted. Looking at the sequence of events, we can see that the rightmost bit is complemented with each positive transition of the count enable connection, simulating the arrival of a data item. When the rightmost bit changes from 1 to 0, the next leftmost bit is complemented. The process is repeated for all bits. This observation gives us a clue to the use of a T flip-flop. The characteristic table of this flip-flop shows that each input of value 1 complements the output. Note that this counter can count only up to 15 or $(1111)_2$. The arrival of the sixteenth data item resets the counter back to $(0000)_2$. Figure E.24 shows the circuit of a 4-bit counter.

Figure E.24 A 4-bit counter

