CHAPTER 3

Boolean Algebra and Digital Logic

3.1 Introduction 121

- In 1854 George Boole introduced a systematic treatment of logic and developed for this purpose an algebraic system known as symbolic logic, or **Boolean algebra**.
- Boolean algebra is a branch of mathematics and it can be used to describe the manipulation and processing of **binary** information. The two-valued Boolean algebra has important application in the design of modern computing systems.
- This chapter contains a brief introduction the basics of logic design. It provides minimal coverage of **Boolean algebra** and this algebra's relationship to **logic gates** and **basic digital circuit**.

3.2 Boolean Algebra 122

- Boolean algebra is algebra for the manipulation of objects that can take on only **two** values, typically true and false.
- It is common to interpret the digital value **0** as false and the digital value **1** as true.

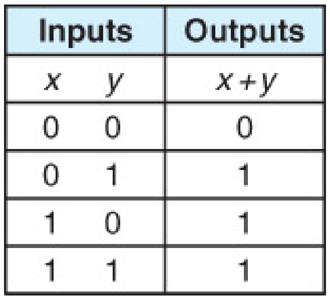
3.2.1 Boolean Expressions 123

- Boolean Expression: Combining the variables and operation yields Boolean expressions.
- Boolean Function: A Boolean function typically has one or more input values and yields **a result**, based on these input value, in the range {0, 1}.
- A Boolean operator can be completely described using a **table** that list inputs, all possible values for these inputs, and the resulting values of the operation.
- A **truth table** shows the **relationship**, in tabular form, between the input values and the result of a specific Boolean operator or function on the input variables.
- The AND operator is also known as a **Boolean product**. The Boolean expression xy is equivalent to the expression x * y and is read "x and y." The behavior of this operator is characterized by the truth table shown in Table 3.1

Inputs		Outputs
x	*y*	*xy*
0	0	0
0	1	0
1	0	0
1	1	1

TABLE 3.1 The Truth Table for AND

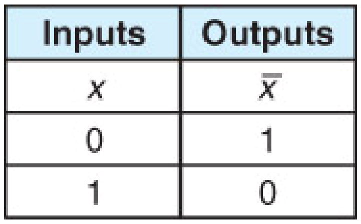
- The OR operator is often referred to as a **Boolean sum**. The expression x+y is read "x or y". The truth table for OR is shown in Table 3.2



Inputs		Outputs
x	y	$x+y$
0	0	0
0	1	1
1	0	1
1	1	1

TABLE 3.2 The Truth Table OR

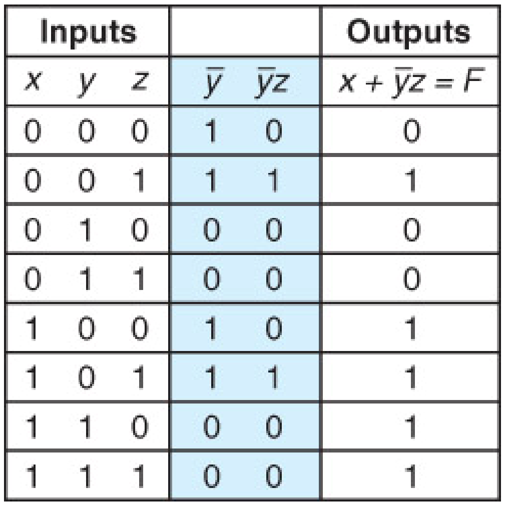
- Both $\bar{x}$ and x' are read as "NOT x." The truth table for NOT is shown in Table 3.3



Inputs	Outputs
x	$\bar{x}$
0	1
1	0

TABLE 3.3 The Truth Table for NOT

- The rule of precedence for Boolean operators give **NOT** top priority, followed by **AND**, and then **OR**



Inputs					Outputs
x	y	z	$\bar{y}$	$\bar{y}z$	$x + \bar{y}z = F$
0	0	0	1	0	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	1	0	1
1	0	1	1	1	1
1	1	0	0	0	1
1	1	1	0	0	1

TABLE 3.4 The Truth Table for F(x, y, z) = x + y'z

3.2.2 Boolean Identities 124

- Boolean expression can be simplified, but we need new **identities**, or **laws**, that apply to Boolean algebra instead of regular algebra.

Identity Name	AND Form	OR Form
Identity Law	$1x = x$	$0 + x = x$
Null (or Dominance) Law	$0x = 0$	$1 + x = 1$
Idempotent Law	$xx = x$	$x + x = x$
Inverse Law	$x\overline{x} = 0$	$x + \overline{x} = 1$
Commutative Law	$xy = yx$	$x + y = y + x$
Associative Law	$(xy)z = x(yz)$	$(x+y) + z = x + (y+z)$
Distributive Law	$x + yz = (x+y)\,(x+z)$	$x(y+z) = xy + xz$
Absorption Law	$x(x+y) = x$	$x + xy = x$
DeMorgan's Law	$\overline{(xy)} = \overline{x} + \overline{y}$	$\overline{(x+y)} = \overline{x}\,\overline{y}$
Double Complement Law	$\overline{\overline{x}} = x$	

TABLE 3.5 Basic Identities of Boolean Algebra

- **DeMorgan's** law provides an easy way of finding the complement of a Boolean function.

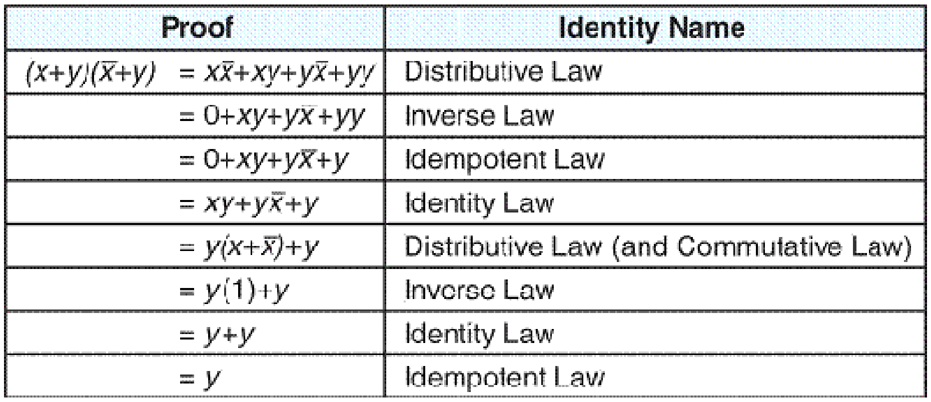
$$\overline{(xy)} = \overline{x} + \overline{y} \quad \text{and} \quad \overline{(x+y)} = \overline{x}\,\overline{y}$$

x	y	(xy)	$\overline{(xy)}$	$\overline{x}$	$\overline{y}$	$\overline{x}+\overline{y}$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

TABLE 3.6 Truth Tables for the AND Form of DeMorgan's Law

3.2.3 Simplification of Boolean Expressions 126

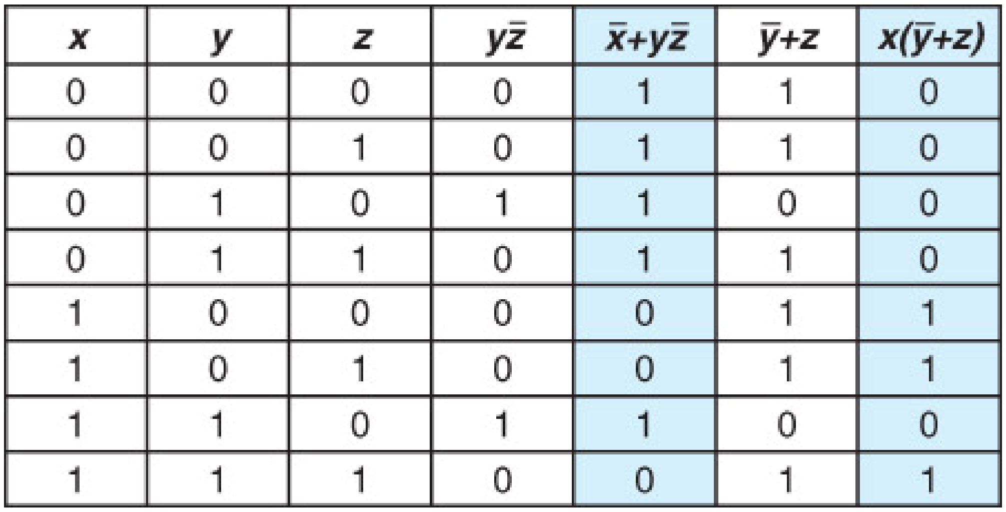
- The algebraic identities we studied in algebra class allow us to reduce algebraic expression to their **simplest** form.
- EXAMPLE 3.2
- EXAMPLE 3.3
- How did we know to insert additional terms to **simplify** the function? Unfortunately, there **no** defined set of rules for using these identities to minimize a Boolean expression: it is simply something tat comes with **experience**.
- To prove the equality of two Boolean expressions, you can also create the truth tables for each and compare. If the truth tables are **identical**, the expressions are **equal**.



Proof	Identity Name
$(x+y)(\bar{x}+y) = x\bar{x}+xy+y\bar{x}+yy$	Distributive Law
$= 0+xy+y\bar{x}+yy$	Inverse Law
$= 0+xy+y\bar{x}+y$	Idempotent Law
$= xy+y\bar{x}+y$	Identity Law
$= y(x+\bar{x})+y$	Distributive Law (and Commutative Law)
$= y(1)+y$	Inverse Law
$= y+y$	Identity Law
$= y$	Idempotent Law

Example using Identities

3.2.4 Complements 128



x	y	z	$y\bar{z}$	$\bar{x}+y\bar{z}$	$\bar{y}+z$	$x(\bar{y}+z)$
0	0	0	0	1	1	0
0	0	1	0	1	1	0
0	1	0	1	1	0	0
0	1	1	0	1	1	0
1	0	0	0	0	1	1
1	0	1	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	0	1	1

TABLE 3.7 Truth Table Representation for a Function and Its Complement

3.2.5 Representing Boolean Functions 130

- In fact, there are an **infinite** number of Boolean expressions that are **logically equivalent** to one another.
- Two expressions that can be represented by the **same** truth table are considered logically equivalent.
- EXAMPLE 3.4
- The two most common standardized forms are the **sum-of-products** form and the **product-of-sums** form.
- In the **sum-of-products form**, ANDed variables are ORed together. For example,

$$F(x,y,z) = xy + xz + yz$$

- In the **product-of-sums form**, ORed variables are ANDed together. For example,

$$F(x,y,z) = (x+y)(x+z)(y+z)$$

- The sum-of-products form is usually **easier** to work with and to simplify, so we use this form exclusively in the sections that follow.
- It is easy to convert a function to **sum-of-products** form using its truth table.
- We are interested in the values of the variables that make the function **true** (=1). Using the truth table, we list the values of the variables that result in a true function value.
- Each group of variables is then ORed together.
- EXAMPLE 3.5

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

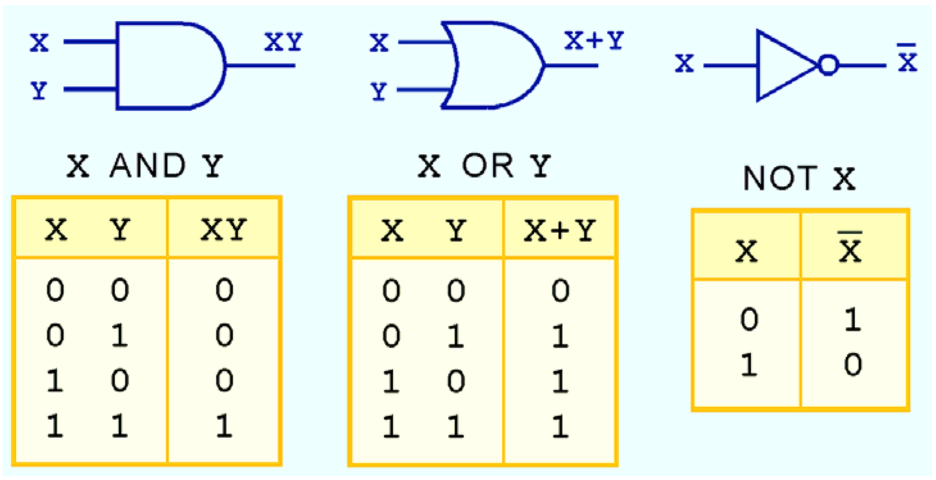
TABLE 3.8 Truth Table Representation for the Majority Function
sum-of-products: F(x, y, z) = x'yz + xy'z + xyz' + xyz

3.3 Logic Gates 131

- We see that Boolean functions are implemented in digital computer circuits called **gates**.
- A gate is an electronic device that produces **a result** based on two or more input values.
- In reality, gates consist of one to six **transistors**, but digital designers think of them as a single unit.
- Integrated circuits contain collections of gates suited to a particular purpose.

3.3.1 Symbols for Logic Gates 132

- The three simplest gates are the AND, OR, and NOT gates.



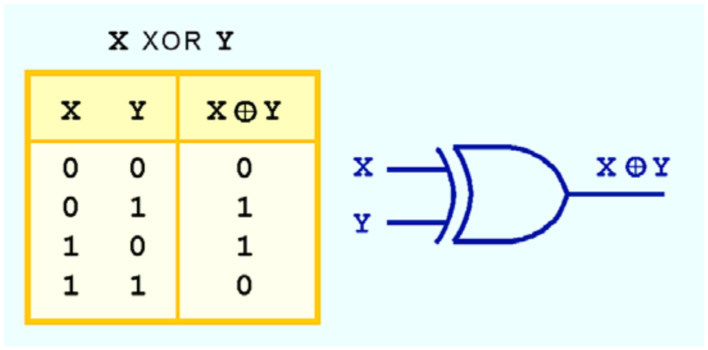
X	Y	XY
0	0	0
0	1	0
1	0	0
1	1	1

X	Y	X+Y
0	0	0
0	1	1
1	0	1
1	1	1

X	$\overline{X}$
0	1
1	0

FIGURE 3.1 The Three Basic Gates

- Another very useful gate is the exclusive OR (XOR) gate.
- The output of the XOR operation is true only when the values of the inputs differ.

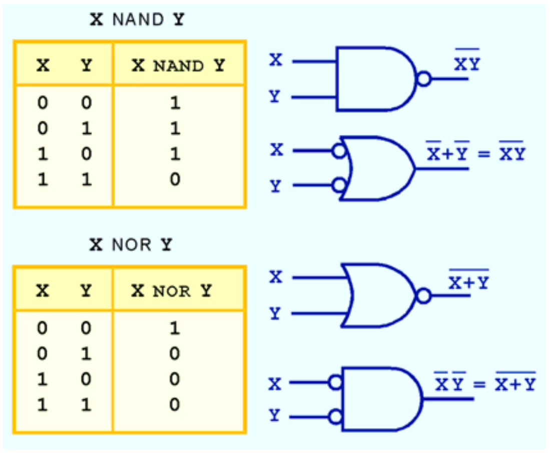


X	Y	$X \oplus Y$
0	0	0
0	1	1
1	0	1
1	1	0

FIGURE 3.2 The exclusive OR (XOR) Gate

3.3.2 Universal Gates 132

- Two other common gates are NAND and NOR, which produce **complementary** output to AND and OR.



X NAND Y

X	Y	X NAND Y
0	0	1
0	1	1
1	0	1
1	1	0

X NOR Y

X	Y	X NOR Y
0	0	1
0	1	0
1	0	0
1	1	0

X
Y
$\overline{XY}$
X
Y
$\overline{X}+\overline{Y} = \overline{XY}$
X
Y
$\overline{X+Y}$
X
Y
$\overline{X}\,\overline{Y} = \overline{X+Y}$

FIGURE 3.3 and 3.4 The Truth Table and Logic Symbols for NAND and NOR Gates

- NAND and NOR are known as **universal gates** because they are **inexpensive** to manufacture and **any** Boolean function can be constructed using **only** NAND or **only** NOR gates.

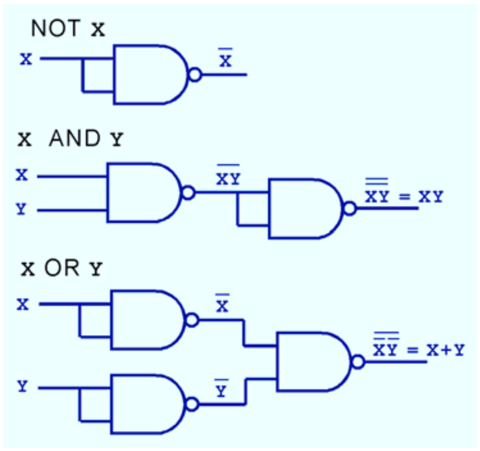


FIGURE 3.5 Three Circuits Constructed Using Only NAND Gates

3.3.3 Multiple Input Gates 133

- Gates can have multiple inputs and more than one output.

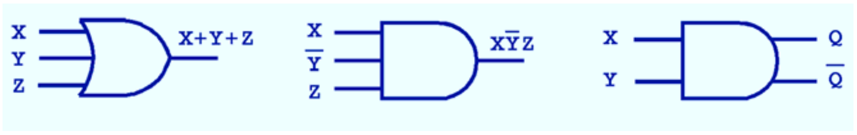


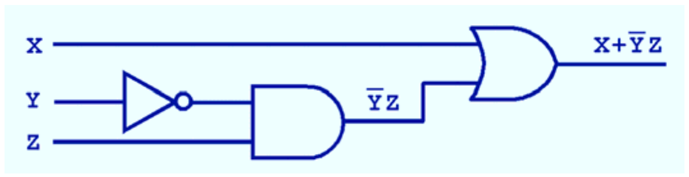
FIGURE 3.6, 3.7, and 3.8

3.4 Digital Components 134

- Every computer is built using collections of **gates** that are all connected by way of wires acting as signal gateway.

3.4.1 Digital Circuits and Their Relationship to Boolean Algebra 134

- More complex Boolean expressions can be represented as combinations of AND, OR, and NOT gates, resulting in a logic diagram that describes the entire expression.

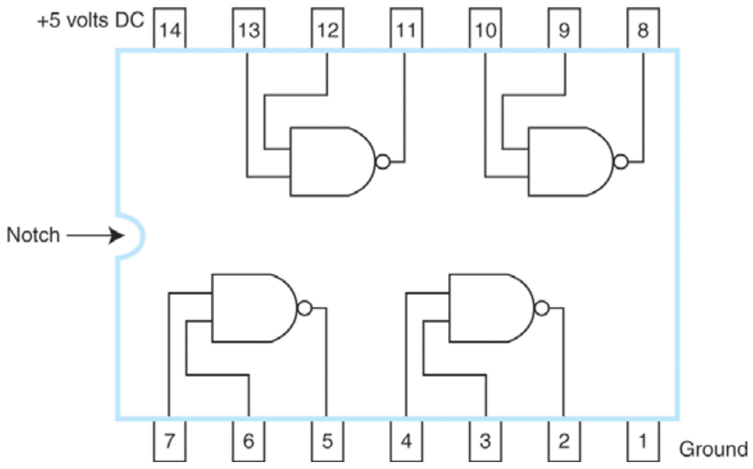


$F(X,Y,Z) = X+\bar{Y}Z$

FIGURE 3.9 A Logic Diagram for F(x, y, z) = x + y'z

3.4.2 Integrated Circuits 136

- Gates are not sold individually; they are sold in units called integrated circuits (ICs).
- A **chip** (a small silicon semiconductor crystal) is a small electronic device consisting of the necessary electronic components (transistors, resistors, and capacitors) to implement **various gates**.
- The first IC were called SSI chips and contained up to 100 electronic components per chip.
- We now have **ULSI** (ultra large-scale integration) with more than 1 million electronic components per chip.



FIUGRE 3.10 A simple SSI Integrated Circuit

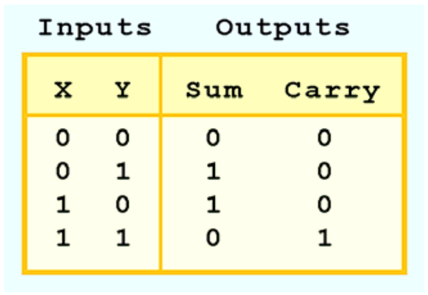
3.5 Combinational Circuits 138

- Digital logic chips are combined to give us useful circuits. These logic circuits can be categorized as either **combinational logic (Section 3.5)** or **sequential logic (Sec. 3.6)**.

3.5.1 Basic Concepts 138

- The **key** concept in recognizing a combinational circuit is that an output is always based on the given inputs.
- The output of a combinational circuit is a function of its inputs, and the output is uniquely determined by the values of the inputs at **any given moment**.
- A given combinational circuit may have several outputs. If so, each output represents a **different** Boolean function.

3.5.2 Examples of Typical Combinational Circuits 138



Inputs		Outputs	
X	Y	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

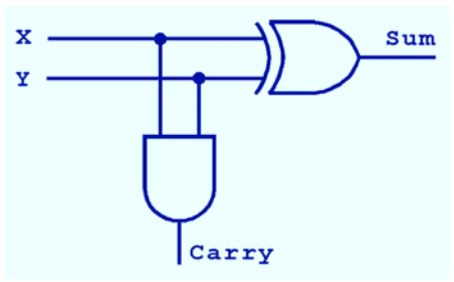
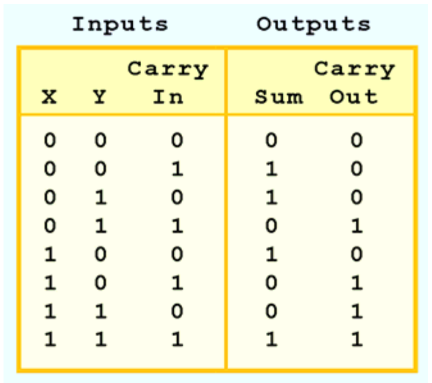


TABLE 3.9 and FIGURE 3.11 The Truth Table and The Logic Diagram for a Half-Adder

- Note that this full-adder is composed of **two** half-adder



Inputs			Outputs	
X	Y	Carry In	Sum	Carry Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

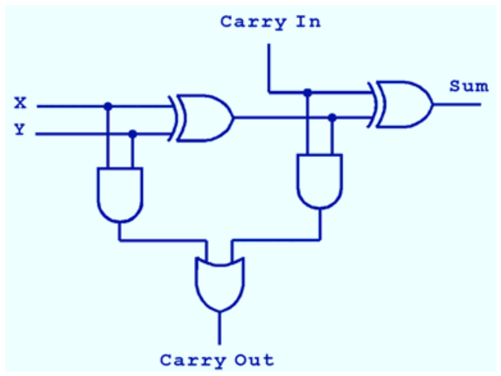


FIGURE 3.12 The Truth Table and The Logic Diagram for a Full-Adder
Sum = X xor Y xor Cin; Cout = XY + (X xor Y) Cin

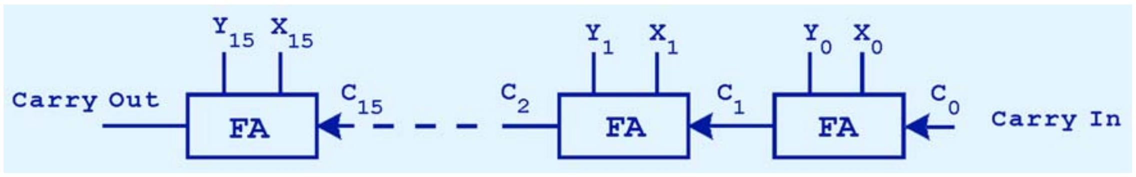


FIGURE 3.13 The Logic Diagram for a Ripple-Carry Adder

- Decoder: Decoding binary information from a set of **n** inputs to a maximum of **2^n** outputs.
- A decoder uses the inputs and their respective values to **select one** specific output line.
- One unique output line is set to **1**, while the other output lines are set to **0**.
- A decoder that has 3 inputs and 8 outputs is called a **3-to-8 decoder**.

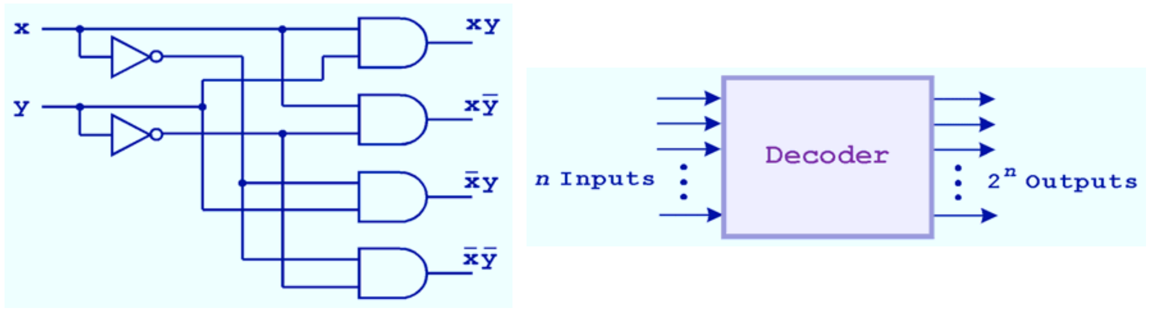
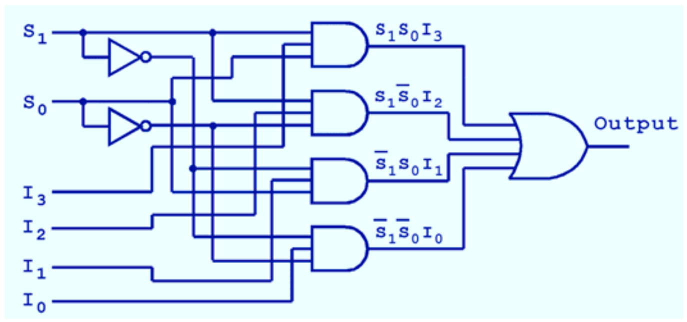


FIGURE 3.14 a) A Look Inside a Decoder; b) A Decoder Symbol

- EXAMPLE 3.6 A 3-to-8 decoder circuit
    - Imagine memory consisting of **8 chips**, each containing **8K bytes**.
    - We have a total of 8K * 8, or 64K (65,536) address available.
    - We need **16** bits to represent each address.
    - The leftmost 3 bits determine on which chip the address is actually located. All addresses on chip 0 have the format: **000**X XXXX XXXX XXXX: Because chip 0 contains the address 0-8191. Similarly, all addresses on the chip 1 have the format: **001**X XXXX XXXX XXXX.
    - The **3** high-order bits are actually used as the inputs to a **decoder** so the computer can determine which chip to **activate** for reading or writing.
    - The output of the decoder is used to **activate one, and only on**, chip as the addresses are decoded.

- The **Multiplexer** circuits binary information from **one** of many input lines and directs it to **a single output line**.
- Only one input (the one selected) is routed through the circuit to output line. All other inputs are "**cut off**."
- Can you think of some situations that require multiplexers? **Time-sharing** computers multiplex the input from user terminals. Modem pools multiplex the modem lines entering the computer.



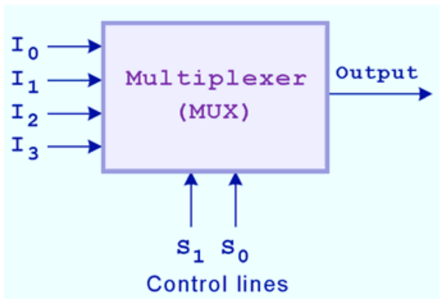


FIGURE 3.15 a) A Look Inside a Multiplexer; b) A Multiplexer Symbol

- A **parity generator** is a circuit that creates the necessary parity bit to add to a word.
- A **parity checker** checks to make sure proper parity (odd or even) is present in the word.
- Typically parity generators and parity checkers are constructed using **XOR** functions. Assuming we are using **odd** parity, the truth table for a parity generator for a 3-bit word is given in Table 3.11.
- The parity checker outputs a **1** if an **error** is detected and 0 otherwise.

x	*y*	*z*	Parity Bit
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Table 3.10 Parity Generator

x	*y*	*z*	*P*	Error detected?
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Table 3.11 Parity Checker

- Bit shifting, moving the bits of a word or byte one position to the left or right is a useful operation.
- When the bits of an unsigned integer are shifted to the **left by one position**, it has the same effect as **multiplying** that integer by **2**.
- The leftmost or rightmost bit is lost after a left or right shift (respectively).
- Left shifting the nibble, 1101, change it to 1010, and right shifting it produces 0110.

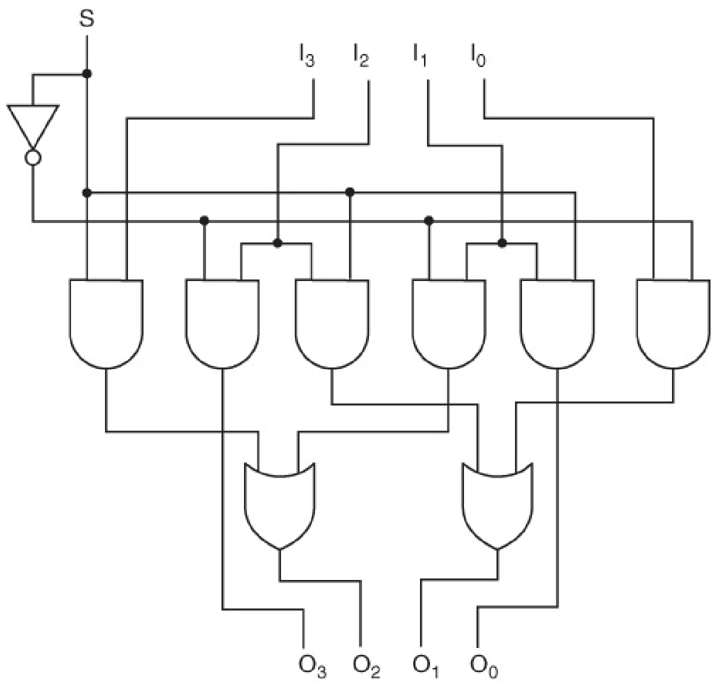


FIGURE 3.16 4-Bit Shifter

- When the control line, S, is **low**, each bit of the input (labeled I_0 to I_3) is shift **left** by one position into the outputs (Labeled O_0 through 0_3).
- When the control line, S, is **high**, each bit of the input (labeled I_0 to I_3) is shift **right** occurs.

- Figure 3.17 illustrates a very simple **ALU** with four basic operations – **AND, OR, NOT**, and addition – carried out on two machine words of 2 bits each.
- The control lines, f_0, and f_1, determine which operation is to be performed by the CPU.
- The signal **00** is used for addition (A + B); **01** for NOT A; **10** for A OR B, and **11** for A AND B.
- The input lines A_0 and A_1 indicate 2 bits of one word, while B_0 and B_1 indicate the second word, C_0 and C_1 represent the output lines.

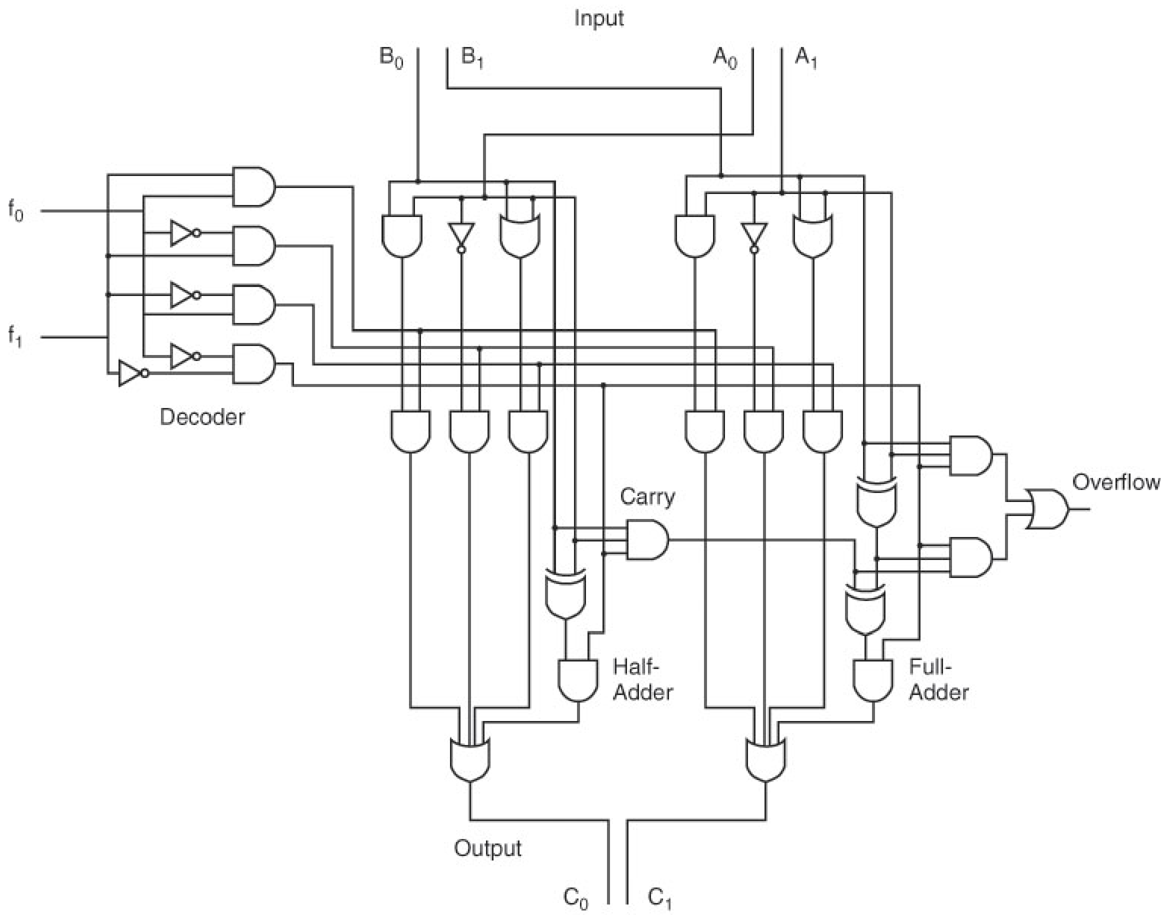


FIGURE 3.17 A Simple Two-Bit ALU

3.6 Sequential Circuits 145

- The major **weakness** of **combinational circuits** is that there is no concept of storage – they are **memoryless**. If we change an input value, this has a direct and **immediate** impact on the value of the output.

3.6.1 Basic Concepts 146

- A sequential circuit defines its output as a function of both its current inputs and its previous inputs. Therefore, the output depends on **past** inputs.
- We typically refer to this storage element as a **flip-flop**.
- Combinational circuits are generalizations of **gates;** sequential circuits are generalizations of **flip-flops**.

3.6.2 Clocks 146

- A sequential circuit uses **past inputs** to determine **present outputs** indicates we must have event ordering.
- A clock is a circuit that emits a series of pulses with a precise pulse width and a precise **interval** between consecutive pulses.
- This interval is called the **clock cycle time**. Clock speed is generally measured in megahertz (MHz), or millions of pulse per second.
- A clock is used by a **sequential circuit** to decide **when** to update the state of the circuit.
- Most sequential circuits are **edge-triggered** (as opposed to being level-triggered). It means they are allowed to change their state on either the **rising** or **falling** edge of the clock signal.

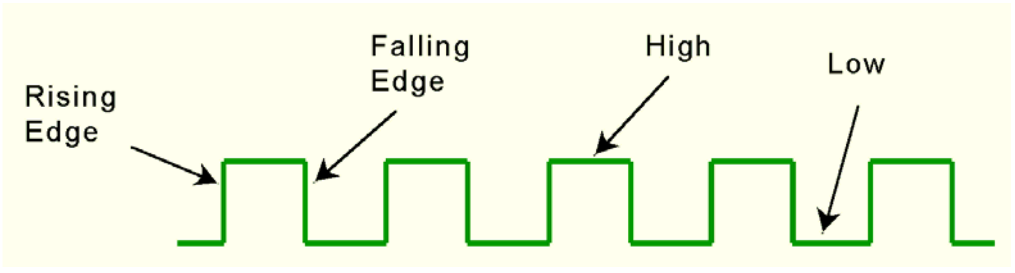


FIGURE 3.18 A clock Signal Indicating Discrete Instances of Time

3.6.3 Flip-Flops 146

- Many people use the terms **latch** and flip-flop interchangeably. Technically, a latch is level triggered, whereas a flip-flop is edge triggered.
- In order to "remember" a past state, sequential circuits rely on a concept called **feedback**. This simply means the output of a circuit is fed back as an input to the same circuit.

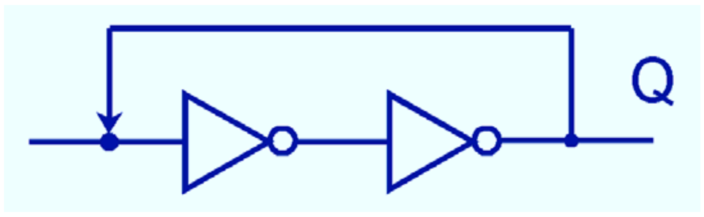


FIGURE 3.19 Example of Simple Feedback

- A more useful feedback circuit is composed of two NOR gates resulting in the most basic memory unit call an **SR flip-flop**. SR stands for "**set/reset**."

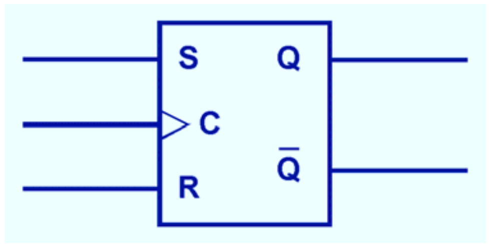
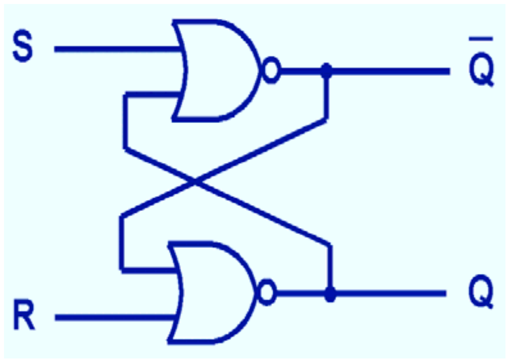
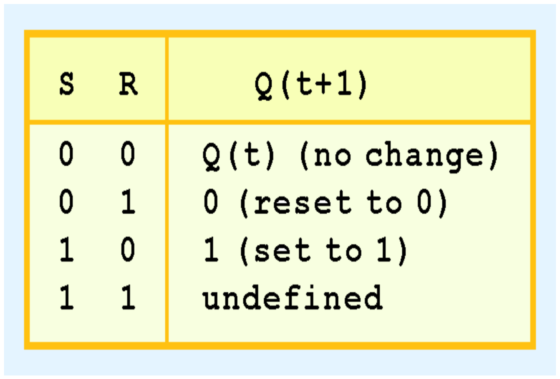


FIGURE 3.20 A SR Flip-Flop Logic Diagram

- Q(t) means the value of the output at time t. Q(t+1) is the value of Q after the **next** clock pulse.
- When both S and R are 1, the SR flip-flop is **unstable**.

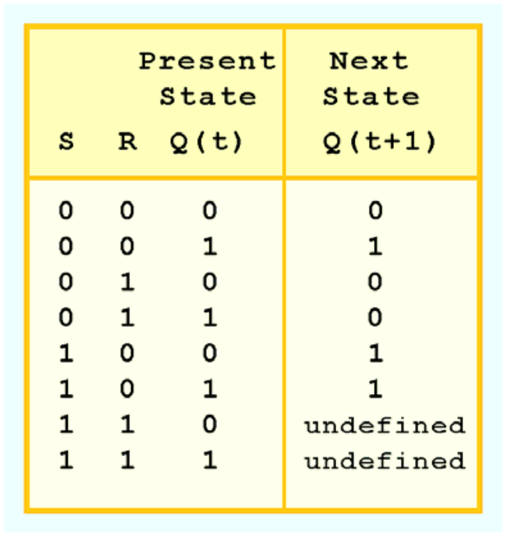




S	R	Q(t+1)
0	0	Q(t) (no change)
0	1	0 (reset to 0)
1	0	1 (set to 1)
1	1	undefined

FIGURE 3.21 a) The Actual SR Flip-Flop; b) The Characteristic Table for the SR Flip-Flop

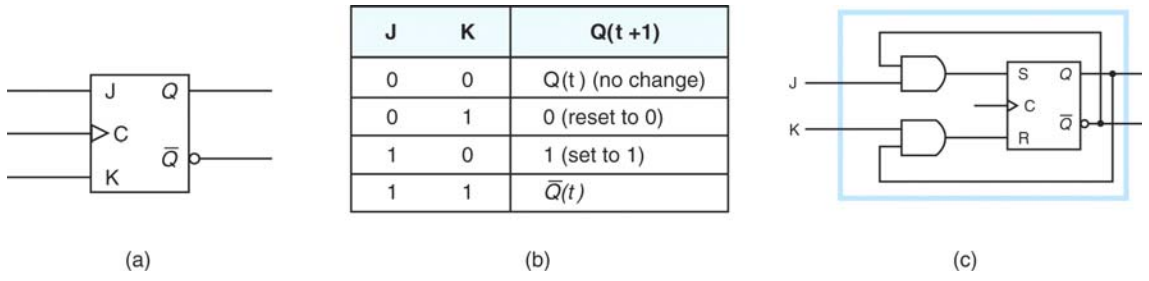
- The SR flip-flop actually has three inputs: S, R, and its current output, Q.



S	R	Present State Q(t)	Next State 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

TABLE 3.12 Truth Table for SR Flip-Flop

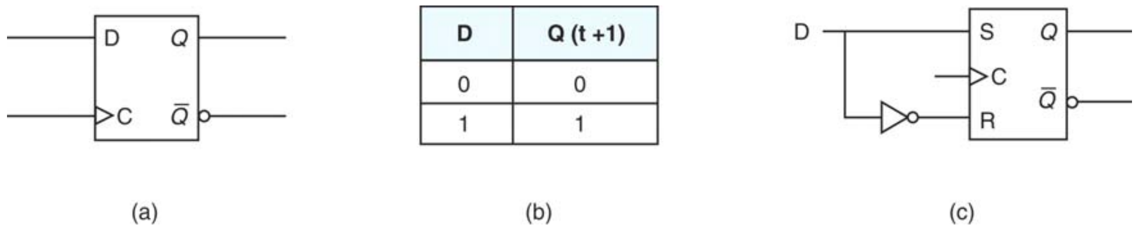
- A **JK flip-flop** simply modify the SR flip-flop to ensure that the **illegal state** (both S and R are 1) never arises.
- The inputs to an SR flip-flop will **never** both be 1
- We will never have an **unstable** circuit.



J	K	Q(t +1)
0	0	Q(t) (no change)
0	1	0 (reset to 0)
1	0	1 (set to 1)
1	1	$\overline{Q}(t)$

FIGURE 3.22 a) A JK Flip-Flop; b) The JK Characteristic Table c) A JK Flip-Flop as a Modified SR Flip-Flop

- A **D flip-flop** is a **true** representation of physical computer memory.
- An output value of 1 means the circuit is currently "**storing**" a value of 1.



D	Q (t +1)
0	0
1	1

FIGURE 3.23 a) A D Flip-Flop; b) The D Characteristic Table c) A D Flip-Flop as a Modified SR Flip-Flop

3.6.5 Examples of Sequential Circuits 155

- The registers must **all** accept their new input values and change their storage elements **at the same time**.

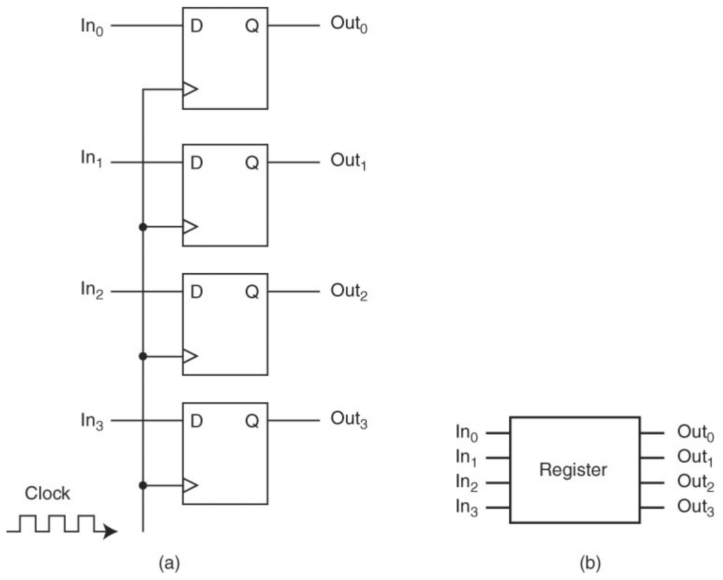


FIGURE 3.30 a) A 4-Bit Register; b) A Block Diagram for a 4-Bit Register

- If we begin counting in binary: 0000, 0001, 0010, 0011, ..., 1111, we can see that as the numbers increase, the low-order bit is complemented each time.
- When J and K are both equal to 1, the flip-flop **complements** the present state.
- The circuit counts only when the clock pulses and this count enable line is set to **1**.

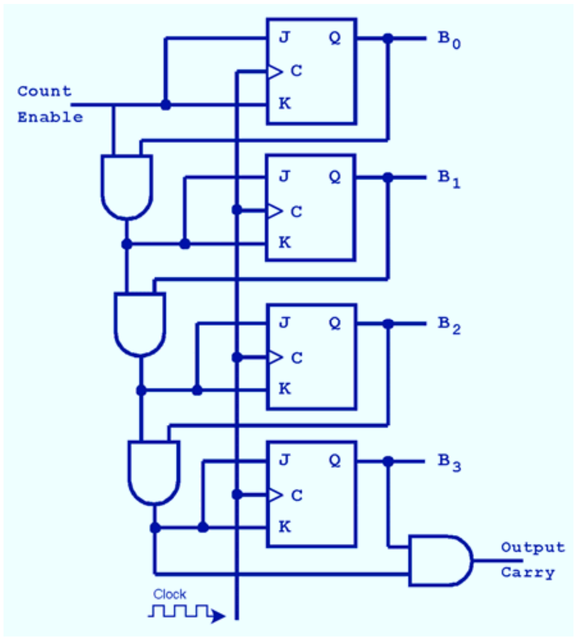


FIGURE 3.31 A 4-Bit Synchronous Counter Using JK Flip-Flops

- The memory depicted holds **four 3-bit** words (**4 X 3 memory**).
- A read or write operation always reads or writes a **complete** word.
- The input In_0, In_1, In_2 are lines used to **store, or write**, a 3-bit word to memory.
- The lines S_0 and S_1 are the **address** lines used to select which word in memory is being referenced (Notice that S_0 and S_1 are the input lines to a **2-to-4 decoder** that is responsible for selecting the correct memory word.)
- The output lines (Out_0, Out_1, and Out_2) are used when **reading** words from memory.
- The write enable control line indicates whether we are **reading or writing**.
- In practice, the input lines and output lines are the **same lines**.
- To summarize our discussion of this memory circuit, here are the steps necessary to write a word to memory:
    1) An address is asserted on S_0 and S_1.
    2) WE (write enable) is set to high
    3) The decoder using S_0 and S_1 enables only one AND gate, selecting a given word in memory
    4) The line selected in Step 3 combined with the clock and WE select only one word
    5) The write gate enabled in Step 4 drives the clock for the selected word.
    6) When clock pulses, the word on the input lines is loaded into the D flip-flops.

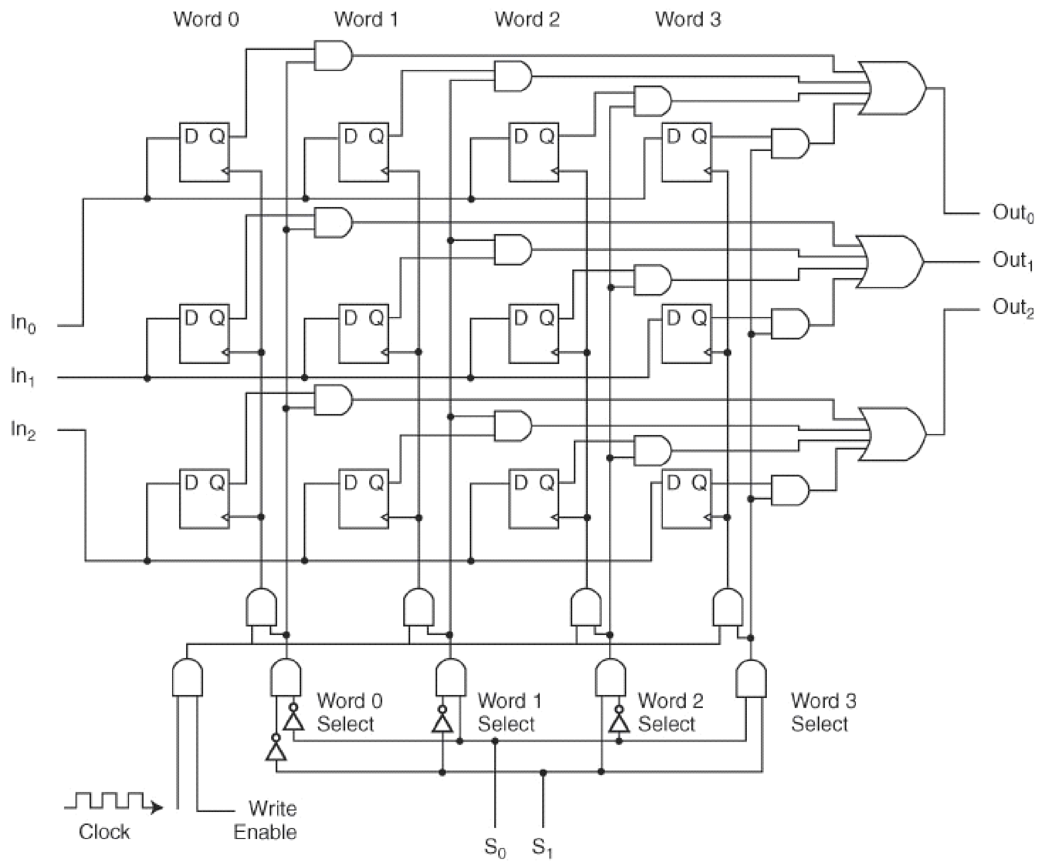


FIGURE 3.32 A 4 X 3 Memory

3.7 Designing Circuits 166

- Digital logic design requires someone not only familiar with **digital logic**, but also well versed in **digital analysis** (analyzing the relationship between **inputs and outputs**), **digital synthesis** (starting with a **truth table** and determining the **logic diagram** to implement the given logic function), and the use of **CAD** (computer-aided design) software.
- A circuit designer faces many problems, including:
    - finding **efficient** Boolean functions,
    - using the **smallest** number of gates,
    - using an **inexpensive** combination of gates,
    - **organizing** the gates of a circuit board to use the **smallest** surface area and minimal **power** requirements, and
    - attempting to do all of this using a **standard** set of modules for implementation.
- Digital systems designers must also be mindful of the physical behaviors of circuits to include minute **propagation delays** that occur between the time when a circuit's inputs are energized and when the output is **accurate and stable**.
- A circuit designer can implement any given algorithm in hardware (recall the Principle of **Equivalence** of Hardware and Software from chapter 1).
- When we need to implement a **simple, specialized** algorithm and its execution speed must be as **fast** as possible; a hardware solution is often preferred.
- This is the idea behind **embedded systems**, which are **small special-purpose** computers that we find in many everyday things. Your **microwave oven** and your **car** most likely contain embedded systems.
- Programming these embedded systems required design software that can **read input variables** and **send output signals** to perform such tasks as turning a light on or off, emitting a beep, sounding an alarm, or opening a door.
- Embedded systems require **special programming** that demands an understanding of the operation of digital circuits, the basics of which you have learned in this chapter.

Chapter Summary 167

- Computers are implementations of **Boolean logic**.
- Any Boolean functions can be represented as **truth tables**.
- Truth tables provide us with a means to express the characteristics of **Boolean functions** as well as **logic circuits**.
- There is a **one-to-one** correspondence between a Boolean function and its digital representation.
- From a chip designer's point of view, the two most important factors are speed and cost: **minimizing** the circuits helps to both lower the cost and increase performance.
- Computer circuits consist of **combinational logic circuits** and **sequential logic** circuits.
- Combinational circuits produce outputs (almost) **immediately** when their inputs change.
- Sequential circuits require **clocks** to control their changes of **state**.
- Combinational logic devices, such as **adders**, **decoders**, and **multiplexers**, produce outputs that are based **strictly** on the current inputs.
- The **AND, OR, and NOT** gates are the building blocks for combinational logic circuits, although **universal gates**, such as **NAND and NOR**, could also be used.
- Sequential logic devices, such as registers, counters, and memory, produce outputs based on the combination of **current inputs** and the **current state** of the circuits. These circuits are built suing **SR, D, and JK flip-flops**.

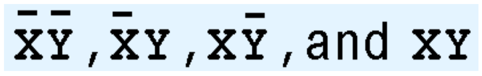
Focus on Karnaugh Maps 181

3A.1 Introduction 181

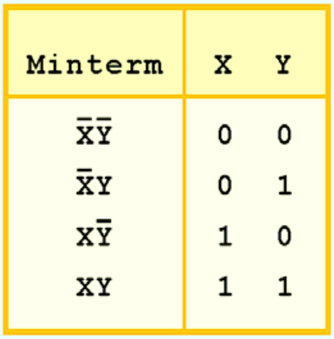
- **Minimizing** circuits helps reduce the number of components in the actual physical implementation.
- Reducing Boolean expressions can be done suing Boolean identities; however, using identities can be very difficult because **no rules** are given on how or when to use the identities.
- In this appendix, we introduce a systematic approach for reducing Boolean expression.

3A.2 Description of Kmaps and Terminology 181

- Karnaugh maps, or Kmaps, are **graphical** way to represent Boolean functions.
- For example, if there are two input values, x and y, there are four minterms.

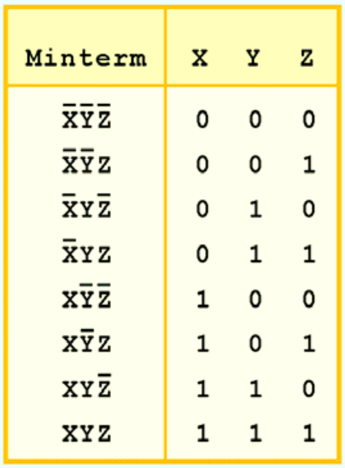


$\bar{X}\bar{Y}, \bar{X}Y, X\bar{Y}$, and XY



Minterm	X	Y
$\bar{X}\bar{Y}$	0	0
$\bar{X}Y$	0	1
$X\bar{Y}$	1	0
XY	1	1

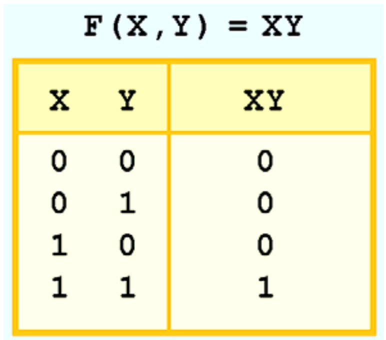
FIGURE 3A.1 Minterms for Two Variables



Minterm	X	Y	Z
$\bar{X}\bar{Y}\bar{Z}$	0	0	0
$\bar{X}\bar{Y}Z$	0	0	1
$\bar{X}Y\bar{Z}$	0	1	0
$\bar{X}YZ$	0	1	1
$X\bar{Y}\bar{Z}$	1	0	0
$X\bar{Y}Z$	1	0	1
$XY\bar{Z}$	1	1	0
XYZ	1	1	1

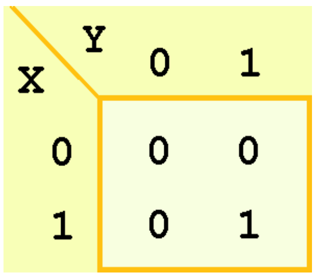
FIGURE 3A.2 Minterms for Three Variables

- EXAMPLE 3A.1 F(x, y) = xy



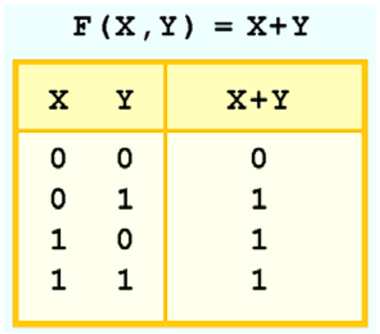
F(X,Y) = XY

X	Y	XY
0	0	0
0	1	0
1	0	0
1	1	1



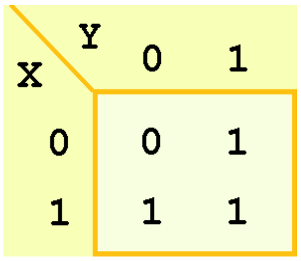
X \ Y	0	1
0	0	0
1	0	1

- EXAMPLE 3A.2 F(x, y) = x + y



F(X,Y) = X+Y

X	Y	X+Y
0	0	0
0	1	1
1	0	1
1	1	1



X \ Y	0	1
0	0	1
1	1	1

$$F(x,y) = X+Y = \bar{X}Y + X\bar{Y} + XY$$

3A.3 Kmap Simplification for Two Variables 183

- The rules of Kmap simplification are:
  1) **Groupings** can contain only **1**s; no 0s.
  2) Only 1s in **adjacent** cells can be grouped; diagonal grouping is not allowed.
  3) The number of 1s in a group must be a **power of 2**.
  4) The groups must be made as **large** as possible while still following all rules.
  5) All 1s must belong a group, even if it is a group of one.
  6) **Overlapping** groups are allowed.
  7) **Wrap** around is allowed.
  8) Use the **fewest** number of groups possible.

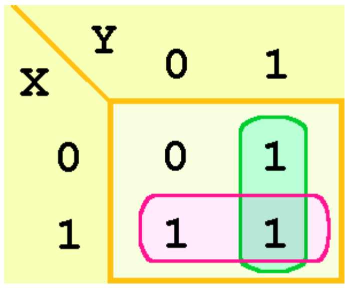


FIGURE 3A.3 Kmap for F(x, y) = x + y

3A.4 Kmap Simplification for Three Variables 185

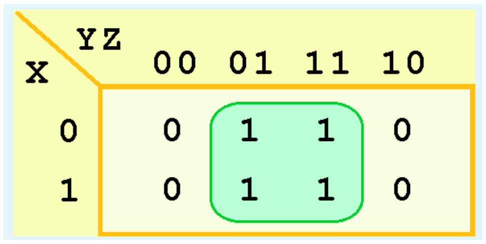
X \ YZ	00	01	11	10
0	$\overline{X}\overline{Y}\overline{Z}$	$\overline{X}\overline{Y}Z$	$\overline{X}YZ$	$\overline{X}Y\overline{Z}$
1	$X\overline{Y}\overline{Z}$	$X\overline{Y}Z$	XYZ	$XY\overline{Z}$

FIGURE 3A.8 Minterms and Kmap Format for Three Variables

- EXAMPLE 3A.3

$$F(X,Y,Z) = \overline{X}\overline{Y}Z + \overline{X}YZ + X\overline{Y}Z + XYZ$$

X \ YZ	00	01	11	10
0	0	1	1	0
1	0	1	1	0

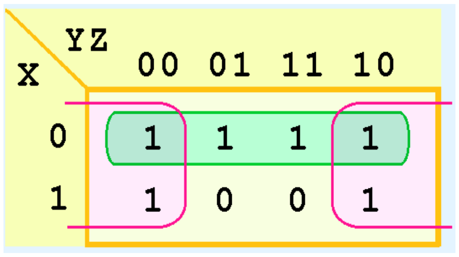


It reduces to **F(x, y, z) = z**

- EXAMPLE 3A.4

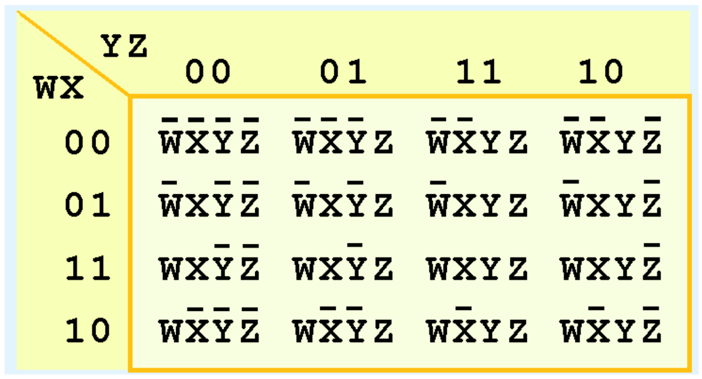
$$F(X,Y,Z) = \overline{X}\overline{Y}\overline{Z} + \overline{X}\overline{Y}Z + \overline{X}YZ + \overline{X}Y\overline{Z} + X\overline{Y}\overline{Z} + XY\overline{Z}$$

X \ YZ	00	01	11	10
0	1	1	1	1
1	1	0	0	1



It reduces to **F(x, y, z) = x' + z'**

3A.5 Kmap Simplification for Four Variables 187

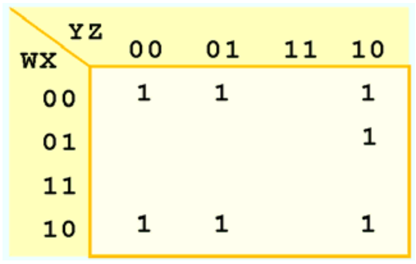


WX \ YZ	00	01	11	10
00	$\overline{W}\overline{X}\overline{Y}\overline{Z}$	$\overline{W}\overline{X}\overline{Y}Z$	$\overline{W}\overline{X}YZ$	$\overline{W}\overline{X}Y\overline{Z}$
01	$\overline{W}X\overline{Y}\overline{Z}$	$\overline{W}X\overline{Y}Z$	$\overline{W}XYZ$	$\overline{W}XY\overline{Z}$
11	$WX\overline{Y}\overline{Z}$	$WX\overline{Y}Z$	$WXYZ$	$WXY\overline{Z}$
10	$W\overline{X}\overline{Y}\overline{Z}$	$W\overline{X}\overline{Y}Z$	$W\overline{X}YZ$	$W\overline{X}Y\overline{Z}$

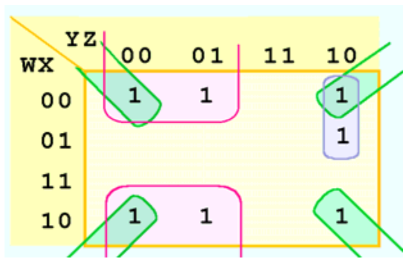
FIGURE 3A.9 Minterms and Kmap Format for Four Variables

- EXAMPLE 3A.6

$$F(W,X,Y,Z) = \overline{W}\overline{X}\overline{Y}\overline{Z} + \overline{W}\overline{X}\overline{Y}Z + \overline{W}\overline{X}Y\overline{Z} + \overline{W}XY\overline{Z} + W\overline{X}\overline{Y}\overline{Z} + W\overline{X}\overline{Y}Z + W\overline{X}Y\overline{Z}$$

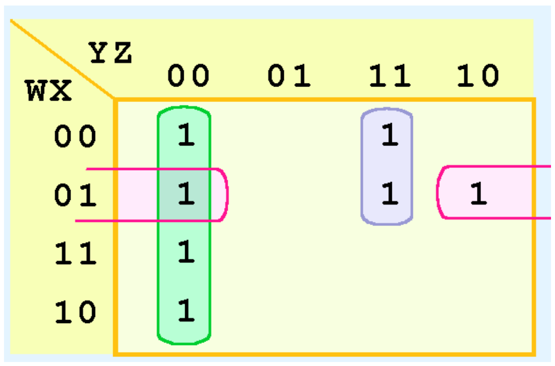


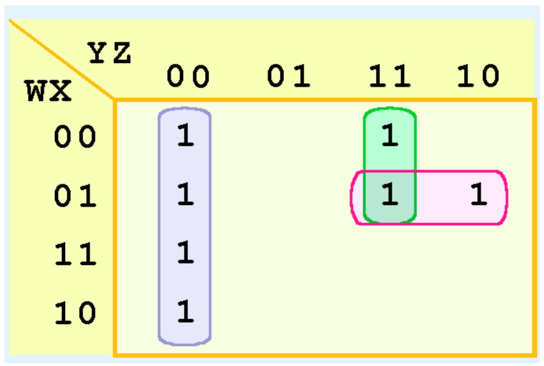
WX \ YZ	00	01	11	10
00	1	1		1
01				1
11				
10	1	1		1



$$F(W,X,Y,Z) = \overline{X}\overline{Y} + \overline{X}\overline{Z} + \overline{W}Y\overline{Z}$$

- EXAMPLE 3A.7
- The last terms are different. F1 and F2, however, are **equivalent**.
- If we follow the rules, Kmap minimization results in a minimized function (and thus a minimal circuit), but these minimized functions need **not be unique** in representation.

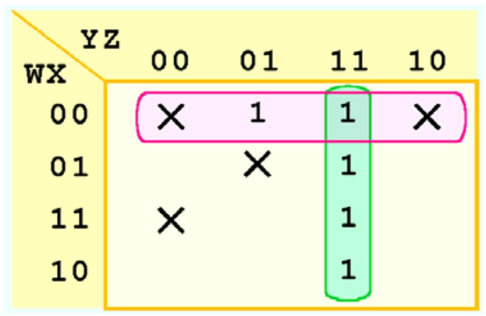




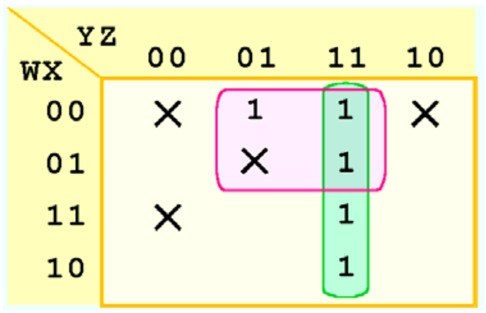
F(w, x, y, z) = **F1** = y'z' + w'yz + **w'xz'** F(w, x, y, z) = **F2** = y'z' + w'yz + **w'xy**

3A.6 Don’t Care Conditions 190

- There are certain situations where a function may not be completely specified, meaning there may be some inputs that are **undefined** for the function.
- **Real circuits don’t** always need to have an **output** defined for every possible input.
- If a circuit is designed so that a particular set of inputs can **never happen**, we call this set of inputs a **don’t care** condition.
- They are very **helpful** to us in Kmap circuit simplification. Because they are input values that should not matter (and should never occur), we can let them have values of **either 0 or 1**, depending on which helps us the most.
- Don’t care values are typically indicated with an “**X**” in the appropriate cell.
- EXAMPLE 3A.10



$$F(W,X,Y,Z) = \overline{W}\,\overline{X} + YZ$$



$$F(W,X,Y,Z) = \overline{W}Z + YZ$$

3A.7 Summary 191

- Using Boolean identities for reduction is awkward and can be very difficult.
- Kmaps provide a precise of steps to follow to find the minimal representation of a function, and thus the minimal circuit that function represents.
- The rules of Kmap simplification are:
  1) Groupings can contain only 1s; no 0s
  2) Only 1s in **adjacent** cells can be grouped; diagonal grouping is not allowed.
  3) The number of 1s in a group must be a **power of 2**.
  4) The groups must be made as **large** as possible while still following all rules.
  5) All 1s must belong a group, even if it is a group of one.
  6) **Overlapping** groups are allowed.
  7) **Wrap** around is allowed.
  8) Use the **fewest** number of groups possible.