CHAPTER 3

Boolean Algebra and Digital Logic

3.1 Introduction 137
3.2 Boolean Algebra 138
   3.2.1 Boolean Expressions 139
   3.2.2 Boolean Identities 140
   3.2.3 Simplification of Boolean Expressions 142
   3.2.4 Complements 144
   3.2.5 Representing Boolean Functions 145
3.3 Logic Gates 147
   3.3.1 Symbols for Logic Gates 147
   3.3.2 Universal Gates 148
   3.3.3 Multiple Input Gates 149
3.4 Digital Components 150
   3.4.1 Digital Circuits and Their Relationship to Boolean Algebra 150
   3.4.2 Integrated Circuits 151
   3.4.3 Putting It All Together: From Problem Description to Circuit 153
3.5 Combinational Circuits 155
   3.5.1 Basic Concepts 155
   3.5.2 Examples of Typical Combinational Circuits 155
3.6 Sequential Circuits 162
   3.6.1 Basic Concepts 163
   3.6.2 Clocks 163
   3.6.3 Flip-Flops 163
   3.6.4 Finite State Machines 167
   3.6.5 Examples of Sequential Circuits 173
   3.6.6 An Application of Sequential Logic: Convolutional Coding and Viterbi
       Detection 178
3.7 Designing Circuits 184
   Chapter Summary 185

Focus on Karnaugh Maps 199
   3A.1 Introduction 199
   3A.2 Description of Kmaps and Terminology 199
   3A.3 Kmap Simplification for Two Variables 201
   3A.4 Kmap Simplification for Three Variables 202
   3A.5 Kmap Simplification for Four Variables 205
   3A.6 Don’t Care Conditions 208
   3A.7 Summary 209
3.1 Introduction 137

- 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 138

- 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 139

- 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

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
<td>(y)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
<td>(\bar{x})</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**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**

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
<td>(y)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**TABLE 3.4 The Truth Table for \(F(x, y, z) = x + y'z\)**
3.2.2 Boolean Identities 140

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

<table>
<thead>
<tr>
<th>Identity Name</th>
<th>AND Form</th>
<th>OR Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identity Law</td>
<td>( x = x )</td>
<td>( 0 + x = x )</td>
</tr>
<tr>
<td>Null (or Dominance) Law</td>
<td>( 0 x = 0 )</td>
<td>( 1 + x = 1 )</td>
</tr>
<tr>
<td>Idempotent Law</td>
<td>( xx = x )</td>
<td>( x + x = x )</td>
</tr>
<tr>
<td>Inverse Law</td>
<td>( x\overline{x} = 0 )</td>
<td>( x + \overline{x} = 1 )</td>
</tr>
<tr>
<td>Commutative Law</td>
<td>( xy = yx )</td>
<td>( x + y = y + x )</td>
</tr>
<tr>
<td>Associative Law</td>
<td>((xy)z = x(yz))</td>
<td>((x+y)+z = x + (y+z))</td>
</tr>
<tr>
<td>Distributive Law</td>
<td>(x+yz = (x+y)(x+z))</td>
<td>(x(y+z) = xy + xz)</td>
</tr>
<tr>
<td>Absorption Law</td>
<td>(x(x+y) = x)</td>
<td>(x+xy = x)</td>
</tr>
<tr>
<td>DeMorgan’s Law</td>
<td>((\overline{xy}) = \overline{x} + \overline{y})</td>
<td>((\overline{x+y}) = \overline{x} \overline{y})</td>
</tr>
<tr>
<td>Double Complement Law</td>
<td></td>
<td>(\overline{\overline{x}} = x)</td>
</tr>
</tbody>
</table>

TABLE 3.5 Basic Identities of Boolean Algebra

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

\[
(xy) = \overline{x} + \overline{y} \quad \text{and} \quad (x+y) = \overline{x} \overline{y}
\]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>( (xy) )</th>
<th>( (\overline{xy}) )</th>
<th>( \overline{x} )</th>
<th>( \overline{y} )</th>
<th>( \overline{x} + \overline{y} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

TABLE 3.6 Truth Tables for the AND Form of DeMorgan’s Law
3.2.3 Simplification of Boolean Expressions 142

- 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 that 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.

<table>
<thead>
<tr>
<th>Proof</th>
<th>Identity Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>((x + y)(\bar{x} + y)) = x\bar{x} + xy + y\bar{x} + y\bar{y})</td>
<td>Distributive Law</td>
</tr>
<tr>
<td>= 0 + xy + y\bar{x} + y\bar{y})</td>
<td>Inverse Law</td>
</tr>
<tr>
<td>= 0 + xy + y\bar{x} + y\bar{y})</td>
<td>Idempotent Law</td>
</tr>
<tr>
<td>= xy + y\bar{x} + y\bar{y})</td>
<td>Identity Law</td>
</tr>
<tr>
<td>= y(x + \bar{x}) + y\bar{y})</td>
<td>Distributive Law (and Commutative Law)</td>
</tr>
<tr>
<td>= y(1) + y\bar{y})</td>
<td>Inverse Law</td>
</tr>
<tr>
<td>= y + y\bar{y})</td>
<td>Identity Law</td>
</tr>
<tr>
<td>= y\bar{y})</td>
<td>Idempotent Law</td>
</tr>
</tbody>
</table>

Example using Identities

3.2.4 Complements 144

- \(F(x, y, z) = x' + yz'\) and its complement, \(F'(x, y, z) = x(y' + z)\)

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>y\bar{z}</th>
<th>\bar{x} + y\bar{z}</th>
<th>\bar{y} + z</th>
<th>x(\bar{y} + z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

TABLE 3.7 Truth Table Representation for a Function and Its Complement
3.2.5 Representing Boolean Functions

- 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 3.9

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**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 147

- 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 147

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

![FIGURE 3.1 The Three Basic Gates](image)

- 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.

![FIGURE 3.2 The exclusive OR (XOR) Gate](image)
3.3.2 Universal Gates 148

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

\[
\begin{array}{|c|c|c|}
\hline
X & Y & X \text{ NAND } Y \\
\hline
0 & 0 & 1 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 0 \\
\hline
\end{array}
\]

**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.

\[
\begin{array}{|c|c|c|}
\hline
X & Y & X \text{ NOR } Y \\
\hline
0 & 0 & 1 \\
0 & 1 & 0 \\
1 & 0 & 0 \\
1 & 1 & 0 \\
\hline
\end{array}
\]

**FIGURE 3.5 Three Circuits Constructed Using Only NAND Gates**

3.3.3 Multiple Input Gates 149

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

**FIGURE 3.6, 3.7, and 3.8**
3.4 Digital Components 150

- 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 150

- 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.

![Logic Diagram](image1)

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

3.4.2 Integrated Circuits 151

- 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.

![Integrated Circuit](image2)

**FIGURE 3.10 A simple SSI Integrated Circuit**
- Example:
  - The Boolean circuit: \( F(x, y, z) = x' y \)

  \[
  \begin{aligned}
  x & \rightarrow \quad x' y \\
   y & \rightarrow
  \end{aligned}
  \]

  - Can be rendered using only NAND gates as:

  \[
  \begin{aligned}
  x & \rightarrow \quad x' \quad (x' y)' = x + y' \\
   y & \rightarrow \quad (x + y')' (x + y')' = (x + y')' = x' y
  \end{aligned}
  \]

  - So we can wire the pre-packaged circuit to implement our function: \( F(x, y, z) = x' y \)
3.4.3 Putting It All Together: From Problem Description to Circuit 153

- Boolean logic is used to solve practical problems.
- Expressed in terms of Boolean logic practical problems can be expressed by truth tables.
- Truth tables can be readily rendered into Boolean logic circuits.
- Example 3.10
  o Suppose we are to design a logic circuit to determine the best time to plant a garden. We consider three factors (inputs):
    - (1) Time, where 0 represents day and 1 represents evening;
    - (2) Moon phase, where 0 represents not full and 1 represents full; and
    - (3) Temperature, where 0 represents 45°F and below, and 1 represents over 45°F.
    - We determine that the best time to plant a garden is during the evening with a full moon.
  o This results in the following truth table:

    | Time (x) | Moon (y) | Temperature (z) | Plant? |
    |----------|----------|-----------------|--------|
    | 0        | 0        | 0               | 0      |
    | 0        | 0        | 1               | 0      |
    | 0        | 1        | 0               | 0      |
    | 0        | 1        | 1               | 0      |
    | 1        | 0        | 0               | 0      |
    | 1        | 0        | 1               | 0      |
    | 1        | 1        | 0               | 1      |
    | 1        | 1        | 1               | 1      |

  o From the truth table, we derive the circuit:

    \[ F(x, y, z) = xy'z + xyz = xy \]

    \[ xy = F \]
3.5 Combinational Circuits

- 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

- 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

**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

**FIGURE 3.12 The Truth Table and The Logic Diagram for a Full-Adder**

\[
\text{Sum} = X \oplus Y \oplus \text{Cin}; \quad \text{Cout} = XY + (X \oplus Y) \text{Cin}
\]
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.

EXAMPLE 3.13 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: 000X XXXX XXXX XXXX: Because chip 0 contains the address 0-8191. Similarly, all addresses on the chip 1 have the format: 001X 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 one, 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.

![Diagram of a multiplexer](image-url)

**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.

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>P</th>
<th>Error detected?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Table 3.10 Parity Generator**

**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.

![4-Bit Shifter Diagram]

**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 $O_3$).
- 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.

![Diagram of a simple two-bit ALU](image)

FIGURE 3.17 A Simple Two-Bit ALU
3.6 Sequential Circuits 162

- 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 163

- 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 163

- 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.

![A clock Signal Indicating Discrete Instances of Time](image-url)
3.6.3 Flip-Flops 163

- 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](image)

- 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](image)

- $Q(t)$ 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**.

![FIGURE 3.21 a) The Actual SR Flip-Flop; b) The Characteristic Table for the SR Flip-Flop](image)
• The SR flip-flop actually has three inputs: S, R, and its current output, Q.

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>S  R  Q(t)</td>
<td>Q(t+1)</td>
</tr>
<tr>
<td>0  0  0</td>
<td>0</td>
</tr>
<tr>
<td>0  0  1</td>
<td>1</td>
</tr>
<tr>
<td>0  1  0</td>
<td>0</td>
</tr>
<tr>
<td>0  1  1</td>
<td>0</td>
</tr>
<tr>
<td>1  0  0</td>
<td>1</td>
</tr>
<tr>
<td>1  0  1</td>
<td>1</td>
</tr>
<tr>
<td>1  1  0</td>
<td>undefined</td>
</tr>
<tr>
<td>1  1  1</td>
<td>undefined</td>
</tr>
</tbody>
</table>

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.

![JK Flip-Flop Diagram](image)

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 Flip-Flop Diagram](image)

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

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

![Diagram of a 4-bit register and a block diagram for a 4-bit register](image)

**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.

![Diagram of a 4-bit synchronous counter using JK flip-flops](image)

**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 In0, In1, In2 are lines used to store, or write, a 3-bit word to memory.
The lines S0 and S1 are the address lines used to select which word in memory is being referenced (Notice that S0 and S1 are the input lines to a 2-to-4 decoder that is responsible for selecting the correct memory word.)
The output lines (Out0, Out1, and Out2) 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 S0 and S1.
2) WE (write enable) is set to high
3) The decoder using S0 and S1 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 184

- 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 185

- 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 using **SR**, **D**, and **JK flip-flops**.
Focus on Karnaugh Maps 199

3A.1 Introduction 199

- **Minimizing** circuits helps reduce the number of components in the actual physical implementation.
- Reducing Boolean expressions can be done using 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 199

- 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.

\[ \bar{x}\bar{y}, \bar{x}y, x\bar{y}, \text{and } xy \]

<table>
<thead>
<tr>
<th>Minterm</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\bar{x}\bar{y})</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(\bar{x}y)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(x\bar{y})</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(xy)</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**FIGURE 3A.1 Minterms for Two Variables**

<table>
<thead>
<tr>
<th>Minterm</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\bar{x}\bar{y}\bar{z})</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(\bar{x}\bar{y}z)</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(\bar{y}z)</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(xy\bar{z})</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(xyz)</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(xyz)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**FIGURE 3A.2 Minterms for Three Variables**

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

\[
F(x, y) = xy
\]

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>XY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**FIGURE 3A.3 XY**
EXAMPLE 3A.2 \( F(x, y) = x + y \)

<table>
<thead>
<tr>
<th>( x )</th>
<th>( y )</th>
<th>( x+y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ F(x, y) = x + y = \bar{x}y + x\bar{y} + xy \]

3A.3 Kmap Simplification for Two Variables 201

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.

FIGURE 3A.3 Kmap for \( F(x, y) = x + y \)
3A.4 Kmap Simplification for Three Variables 202

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

- **EXAMPLE 3A.3**

  \[ F(x, y, z) = \overline{xyz} + \overline{xyz} + xyz + xyz \]

  ![Kmap for Example 3A.3](image1)

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

- **EXAMPLE 3A.4**

  \[ F(x, y, z) = \overline{xyz} + \overline{xyz} + \overline{xyz} + \overline{xyz} + \overline{xyz} + xyz \]

  ![Kmap for Example 3A.4](image2)

  It reduces to \( F(x, y, z) = x' + z' \)
3A.5 Kmap Simplification for Four Variables

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

- **EXAMPLE 3A.6**

\[
F(w, x, y, z) = \overline{wxy} + wxy' + wxyz + \overline{wxyz} + \overline{wxyz} + wxyz + wxyz
\]

- **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.

\[
F(w, x, y, z) = \overline{xy} + xz + \overline{wy}z
\]

\[
F(w, x, y, z) = F1 = y'z' + w'yz + w'xz' \quad F(w, x, y, z) = F2 = y'z' + w'yz + w'xy
\]
3A.6 Don’t Care Conditions 208

- 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

![Karnaugh Map Example](image)

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

3A.7 Summary 209

- Using Boolean identities for reduction is awkward and can be very difficult.
- Kmaps provide a precise step-by-step way 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.