Logic and Computer Design Fundamentals M. Morris Mano Charles Kime

Fourth Edition

**Pearson Education Limited**

Edinburgh Gate

Harlow

Essex CM20 2JE

England and Associated Companies throughout the world

*Visit us on the World Wide Web at:* www.pearsoned.co.uk

© Pearson Education Limited 2014

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without either the prior written permission of the publisher or a licence permitting restricted copying in the United Kingdom issued by the Copyright Licensing Agency Ltd, Saffron House, 6–10 Kirby Street, London EC1N 8TS.

All trademarks used herein are the property of their respective owners. The use of any trademark in this text does not vest in the author or publisher any trademark ownership rights in such trademarks, nor does the use of such trademarks imply any affiliation with or endorsement of this book by such owners.

ISBN 10: 1-292-02468-2

ISBN 13: 978-1-292-02468-4

**British Library Cataloguing-in-Publication Data**

A catalogue record for this book is available from the British Library

Printed in the United States of America

PEARSON C U S T O M LIBRA R Y Table of Contents

1. Digital Systems and Information

M. Morris Mano, Charles R. Kime **1**

2. Combinational Logic Circuits

M. Morris Mano, Charles R. Kime **35**

3. Combinational Logic Design

M. Morris Mano, Charles R. Kime **99**

4. Arithmetic Functions and HDLs

M. Morris Mano, Charles R. Kime **155**

5. Sequential Circuits

M. Morris Mano, Charles R. Kime **215**

6. Selected Design Topics

M. Morris Mano, Charles R. Kime **305**

7. Registers and Register Transfers

M. Morris Mano, Charles R. Kime **347**

8. Memory Basics

M. Morris Mano, Charles R. Kime **427**

9. Instruction Set Architecture

M. Morris Mano, Charles R. Kime **459**

10. Computer Design Basics

M. Morris Mano, Charles R. Kime **507**

11. Input–Output and Communication

M. Morris Mano, Charles R. Kime **563**

12. Memory Systems

M. Morris Mano, Charles R. Kime **601**

13. RISC and CISC Central Processing Units

M. Morris Mano, Charles R. Kime **633**

I

Index **689**

II

DIGITAL SYSTEMS

AND INFORMATION

From Chapter 1 of *Logic and Computer Design Fundamentals*, Fourth Edition. M. Morris Mano, Charles R. Kime. Copyright © 2008 by Pearson Education, Inc. Published by Pearson Prentice Hall. All rights reserved.

1

Keyboard

LCD

Screen

Hard Drive

Drive ControllerBus Interface

Graphics Adapter

FPU

Internal Cache

CPU MMU

Processor

External

Cache

Generic Computer

RAM

Note: The companion website for this text is http://www.prenhall.com/mano 2

T

DIGITAL SYSTEMS AND INFORMATION

his text deals with logic circuits and digital computers. Early computers were used for computations with discrete numeric elements called *digits* (the Latin word for fingers)—hence the term *digital computer*. The use of “digital” spread

from the computer to logic circuits and other systems that use discrete elements of information, giving us the terms *digital circuits* and *digital systems*. The term *logic* is applied to circuits that operate on a set of just two elements with values True (1) and False (0). Since computers are based on logic circuits, they operate on patterns of

elements from these two-valued sets, which are used to represent, among other

things, the decimal digits. Today, the term “digital circuits” is viewed as synonymous with the term “logic circuits.”

The *general-purpose digital computer* is a digital system that can follow a stored

sequence of instructions, called a *program*, that operates on data. The user can

specify and change the program or the data according to specific needs. As a result of this flexibility, general-purpose digital computers can perform a variety of information processing tasks, ranging over a very wide spectrum of applications. This makes the digital computer a highly general and very flexible digital system. Also, due to its

generality, complexity, and widespread use, the computer provides an ideal vehicle for learning the concepts, methods, and tools of digital system design. To this end, we

use the exploded pictorial diagram of a computer of the class commonly referred to as a PC (personal computer) given on the opposite page. We employ this generic

computer to highlight the significance of the material covered and its relationship to the overall system. A bit later in this chapter, we will discuss the various major

components of the generic computer and see how they relate to a block diagram

commonly used to represent a computer. Otherwise, the remainder of the chapter

focuses on the digital systems in our daily lives and introduces approaches for

representing information in digital circuits and systems.

3

**DIGITAL SYSTEMS AND INFORMATION**

**1 INFORMATION REPRESENTATION**

Digital systems store, move, and process information. The information represents a broad range of phenomena from the physical and man-made world. The physical world is characterized by parameters such as weight, temperature, pressure, veloc ity, flow, and sound intensity and frequency. Most physical parameters are *continu ous*, typically capable of taking on all possible values over a defined range. In contrast, in the man-made world, parameters can be discrete in nature, such as business records using words, quantities, and currencies, taking on values from an alphabet, the integers, or units of currency, respectively. In general, information systems must be able to represent both continuous and discrete information. Sup pose that temperature, which is continuous, is measured by a sensor and converted to an electrical voltage, which is likewise continuous. We refer to such a continuous voltage as an *analog signal*, which is one possible way to represent temperature. But, it is also possible to represent temperature by an electrical voltage that takes on *discrete* values that occupy only a finite number of values over a range, e.g., cor responding to integer degrees centigrade between 40 and +119. We refer to such a voltage as a *digital signal*. Alternatively, we can represent the discrete values by multiple voltage signals, each taking on a discrete value. At the extreme, each sig nal can be viewed as having only two discrete values, with multiple signals repre senting a large number of discrete values. For example, each of the 160 values just mentioned for temperature can be represented by a particular combination of eight two-valued signals. The signals in most present-day electronic digital systems use just two discrete values and are therefore said to be *binary*. The two discrete values used are often called 0 and 1, the digits for the binary number system.

We typically represent the two discrete values by ranges of voltage values called HIGH and LOW. Output and input voltage ranges are illustrated in Figure 1(a). The HIGH output voltage value ranges between 0.9 and 1.1 volts, and the LOW out put voltage value between 0.1 and 0.1 volts. The high input range allows 0.6 to 1.1 volts to be recognized as a HIGH, and the low input range allows 0.1 to 0.4 volts to be recognized as a LOW. The fact that the input ranges are wider than the

Voltage (Volts)

HIGH

OUTPUT INPUT

1.0

0.9

0.6

HIGH

1.0 0.5 0.0

Time

LOW

0.4

0.1

0.0

Volts

LOW

(b) Time-dependent voltage 1

0

Time

(a) Example voltage ranges **FIGURE 1**

(c) Binary model of time-dependent voltage

Examples of Voltage Ranges and Waveforms for Binary Signals

4

**DIGITAL SYSTEMS AND INFORMATION**

output ranges allows the circuits to function correctly in spite of variations in their behavior and undesirable “noise” voltages that may be added to or subtracted from the outputs.

We give the output and input voltage ranges a number of different names.

Among these are HIGH (H) and LOW (L), TRUE (T) and FALSE (F), and 1 and 0. It is natural to associate the higher voltage ranges with HIGH or H, and the lower voltage ranges with LOW or L. For TRUE and 1 and FALSE and 0, how ever, there is a choice. TRUE and 1 can be associated with either the higher or lower voltage range and FALSE and 0 with the other range. Unless otherwise indi cated, we assume that TRUE and 1 are associated with the higher of the voltage ranges, H, and that FALSE and 0 are associated with the lower of the voltage ranges, L. This particular convention is called *positive logic*.

It is interesting to note that the values of voltages for a digital circuit in

Figure 1(a) are still continuous, ranging from 0.1 to +1.1 volts. Thus, the volt age is actually analog! The actual voltages values for the output of a very high speed digital circuit are plotted versus time in Figure 1(b). Such a plot is referred to as a *waveform*. The interpretation of the voltage as binary is based on a model using voltage ranges to represent discrete values 0 and 1 on the inputs and the outputs. The application of such a model, which redefines all voltage above 0.5 V as 1 and below 0.5 V as 0 in Figure 1(b), gives the wave form in Figure 1(c). The output has now been interpreted as binary, having only discrete values 1 and 0, with the actual voltage values removed. We note that digital circuits, made up of electronic devices called transistors, are designed to cause the outputs to occupy the two distinct output voltage ranges for 1 (H) and 0 (L) in Figure 1, whenever the outputs are not changing. In contrast, ana log circuits are designed to have their outputs take on continuous values over their range, whether changing or not.

Since 0 and 1 are associated with the binary number system, they are the pre

ferred names for the signal ranges. A binary digit is called a *bit*. Information is rep resented in digital computers by groups of bits. By using various coding techniques, groups of bits can be made to represent not only binary numbers, but also other groups of discrete symbols. Groups of bits, properly arranged, can even specify to the computer the program instructions to be executed and the data to be processed. Why is binary used? In contrast to the situation in Figure 1, consider a sys

tem with 10 values representing the decimal digits. In such a system, the voltages available—say, 0 to 1.0 volts—could be divided into 10 ranges, each of length 0.1 volt. A circuit would provide an output voltage within each of these 10 ranges. An input of a circuit would need to determine in which of the 10 ranges an applied voltage lies. If we wish to allow for noise on the voltages, then output voltage might be permitted to range over less than 0.05 volt for a given digit representa tion, and boundaries between inputs could vary by less than 0.05 volt. This would require complex and costly electronic circuits, and the output still could be dis turbed by small “noise” voltages or small variations in the circuits occurring during their manufacture or use. As a consequence, the use of such multivalued circuits is very limited. Instead, binary circuits are used in which correct circuit operation can be achieved with significant variations in values of the two output voltages and the

5

**DIGITAL SYSTEMS AND INFORMATION** Memory

Control

CPU

unit Datapath Input/Output

**FIGURE 2**

Block Diagram of a Digital Computer

two input ranges. The resulting transistor circuit with an output that is either HIGH or LOW is simple, easy to design, and extremely reliable. In addition, this use of binary values makes the results calculated repeatable in the sense that the same set of input values to a calculation always gives exactly the same set of out puts. This is not necessarily the case for multivalued or analog circuits, in which noise voltages and small variations due to manufacture or circuit aging can cause results to differ at different times.

**The Digital Computer**

A block diagram of a digital computer is shown in Figure 2. The memory stores programs as well as input, output, and intermediate data. The datapath performs arithmetic and other data-processing operations as specified by the program. The control unit supervises the flow of information between the various units. A data path, when combined with the control unit, forms a component referred to as a *central processing unit*, or CPU.

The program and data prepared by the user are transferred into memory by

means of an input device such as a keyboard. An output device, such as an LCD (liquid crystal display), displays the results of the computations and presents them to the user. A digital computer can accommodate many different input and output devices, such as CD-ROM and DVD drives, scanners, and printers. These devices use digital logic circuits, but often include analog electronic circuits, optical sensors, LCDs (liquid crystal displays), and electromechanical components.

The control unit in the CPU retrieves the instructions, one by one, from the

program stored in the memory. For each instruction, the control unit manipulates the datapath to execute the operation specified by the instruction. Both program and data are stored in memory. A digital computer can perform arithmetic compu tations, manipulate strings of alphabetic characters, and be programmed to make decisions based on internal and external conditions.

6

**DIGITAL SYSTEMS AND INFORMATION**

**Beyond the Computer**

In terms of world impact, computers, such as the PC, are not the end of the story. Smaller, often less powerful, single-chip computers called *microcomputers* or *microcontrollers*, or special-purpose computers called *digital signal processors* (DSPs) actually are more prevalent in our lives. These computers are parts of everyday products and their presence is often not apparent. As a consequence of being integral parts of other products and often enclosed within them, they are called *embedded systems.* A generic block diagram of an embedded system is shown in Figure 3. Central to the system is the microcomputer (or its equivalent). It has many of the characteristics of the PC, but differs in the sense that its soft ware programs are often permanently stored to provide only the functions required for the product. This software, which is critical to the operation of the product, is an integral part of the embedded system and referred to as *embedded software*. Also, the human interface of the microcomputer can be very limited or nonexistent. The larger information-storage components such as a hard drive and compact disk or DVD drive frequently are not present. The microcomputer con tains some memory; if additional memory is needed, it can be added externally.

With the exception of the external memory, the hardware connected to the embedded microcomputer in Figure 3 interfaces with the product and/or the out side world. The input devices transform inputs from the product or outside world into electrical signals, and the output devices transform electrical signals into out puts to the product or outside world. The input and output devices are of two types, those which use analog signals and those which use digital signals. Examples of digital input devices include a limit switch which is closed or open depending on whether a force is applied to it and a keypad having ten decimal integer buttons. Examples of analog input devices include a thermistor which changes its electrical

Analog

Input Devices and Signal

Conditioning

A-to-D

Converters

Digital

Input Devices and Signal

Conditioning

Microcomputer, Microcontroller, or Digital Signal Processor

External

Memory

D-to-A

Converters

Signal

Conditioning and Digital

Output Devices

Signal

Conditioning and Digital

Output Devices

**FIGURE 3**

Block Diagram of an Embedded System

7

**DIGITAL SYSTEMS AND INFORMATION**

resistance in response to the temperature and a crystal which produces a charge (and a corresponding voltage) in response to the pressure applied. Typically, elec trical or electronic circuitry is required to “condition” the signal so that it can be read by the embedded system. Examples of digital output devices include relays (switches that are opened or closed by applied voltages), a stepper motor that responds to applied voltage pulses, or an LED digital display. Examples of analog output devices include a loudspeaker and a panel meter with a dial. The dial posi tion is controlled by the interaction of the magnetic fields of a permanent magnet and an electromagnet driven by the voltage applied to the meter.

Next, we illustrate embedded systems by considering a temperature measure

ment performed by using a wireless weather station. In addition, this example also illustrates analog and digital signals, including conversion between the signal types.

**EXAMPLE 1 Temperature Measurement and Display**

A wireless weather station measures a number of weather parameters at an out door site and transmits them for display to an indoor base station. Its operation can be illustrated by considering the temperature measurement illustrated in Fig ure 4 with reference to the block diagram in Figure 3. Two embedded microproces sors are used, one in the outdoor site and the other in the indoor base station.

The temperature at the outdoor site ranges continuously from 40°F to

+115°F. Temperature values over one 24-hour period are plotted as a function of time in Figure 4(a). This temperature is measured by a sensor consisting of a ther mistor (a resistance that varies with temperature) with a fixed current applied by an electronic circuit. This sensor provides an analog voltage that is proportional to the temperature. Using signal conditioning, this voltage is changed to a continuous voltage ranging between 0 and 15 volts, as shown in Figure 4(b).

The analog voltage is sampled at a rate of once per hour (a very slow sam

pling rate used just for illustration), as shown by the dots in Figure 4(b). Each value sampled is applied to an analog-to-digital (A/D) converter, as in Figure 3, which replaces the value with a digital number written in binary and having deci mal values between 0 and 15, as shown in Figure 4(c). A binary number can be interpreted in decimal by multiplying the bits from left to right times the respective weights, 8, 4, 2, and 1, and adding the resulting values. For example, 0101 can be interpreted as . In the process of conversion, the

08140211 5

value of the temperature is quantized from an infinite number of values to just 16 values. Examining the correspondence between the temperature in Figure 4(a) and the voltage in Figure 4(b), we find that the typical digital value of temperature rep resents an actual temperature range up to 5 degrees above or below the digital value. For example, the analog temperature range between 25 and 15 degrees is represented by the digital temperature value of 20 degrees. This discrepancy between the actual temperature and the digital temperature is called the *quantiza tion error*. In order to obtain greater precision, we would need to increase the num ber of bits beyond four in the output of the A/D converter. The hardware

8

**DIGITAL SYSTEMS AND INFORMATION**

Temperature (degrees F)

120

80

40

0

40

0 4 8 2 12 16 20 4 (a) Analog temperature

Time (hours)

Sensor and

Voltage (Volts) 16

12

8

4

0

Sampling point

Signal Conditioning Time (Hours)

0 4 8 2 12 16 20 4 (b) Continuous (analog) voltage

Digital numbers (binary)

16

12

011101110111 0111

Analog-to-Digital (A/D) Conversion

8

00110011

0100010101100111011001000101010000110011 00110101

0011 001101000100

4 0

0011

0011

Time (hours)

0 4 8 2 12 16 20 4 (c) Digital voltage

Voltage (volts)

16

12

8

4

0

0 4 8 2 12 16 20 4 (d) Discrete (digital) voltage

Voltage (volts)

16

12

8

4

0

0 4 8 2 12 16 20 4 (e) Continuous (analog) voltage

Digital-to-Analog (D/A) Conversion

Time (hours)

Signal Conditioning

Time (hours)

Output

20 40 60

20 40 60

20 40 60

20 40 60

20 40 60

Temp Temp Temp

Temp Temp

0 80

0 80

0 80

0 80

0 80

20

40

100

120

20

40

100

120

20

40

100

120

20

40

100

120

20

40

100

120

(f) Continuous (analog) readout

**FIGURE 4**

Temperature Measurement and Display

9

**DIGITAL SYSTEMS AND INFORMATION**

components for sensing, signal conditioning, and A/D conversion are shown in the upper left corner of Figure 3.

Next, the digital value passes through the microcomputer to a wireless trans

mitter as a digital output device in the lower right corner of Figure 3. The digital value is transmitted to a wireless receiver, which is a digital input device in the base station. The digital value enters the microcomputer at the base station, where calculations may be performed to adjust its value based on thermistor properties. The resulting value is to be displayed with an analog meter shown in Figure 4(f) as the output device. In order to support this display, the digital value is converted to an analog value by a digital-to-analog converter, giving the quantized, discrete voltage levels shown in Figure 4(d). Signal conditioning, such as processing of the output by a low-pass analog filter, is applied to give the continuous signal in Fig ure 4(e). This signal is applied to the analog voltage display, which has been labeled with the corresponding temperature values shown for five selected points over the 24-hour period in Figure 4(f). ■

**TABLE 1**

**Embedded System Examples**

**Application Area Product**

Banking, commerce and manufacturing Copiers, FAX machines, UPC scanners, vend

ing machines, automatic teller machines,

automated warehouses, industrial robots

Communication Cell phones, routers, satellites

Games and toys Video games, handheld games, talking stuffed

toys

Home appliances Digital alarm clocks, conventional and micro

wave ovens, dishwashers

Media CD players, DVD players, flat panel TVs,

Digital cameras, digital video cameras

Medical equipment Pacemakers, incubators, magnetic resonance

imaging

Personal Digital watches, MP3 players, personal digital

assistants

Transportation and navigation Electronic engine controls, traffic light con

trollers, aircraft flight controls, global posi

tioning systems

10

**DIGITAL SYSTEMS AND INFORMATION**

You might ask: “How many embedded systems are there in my current living

environment?” Do you have a cell phone? An iPodTM? An XboxTM? A digital cam era? A microwave oven? An automobile? All of these are embedded systems! In fact, a late-model automobile can contain more than 50 microcontrollers, each con trolling a distinct embedded system, such as the engine control unit (ECU), auto matic braking system (ABS), and stability control unit (SCU). Further, a significant proportion of these embedded systems communicate with each other through a CAN (controller area network). A new automotive network called FlexRay promises to provide high-speed, reliable communication for safety-critical tasks such as braking-by-wire and steering-by-wire, eliminating primary depen dence on mechanical and hydraulic linkages and enhancing the potential for addi tional safety features such as collision avoidance. Table 1 lists examples of embedded systems classified by application area.

Considering the widespread use of personal computers and embedded sys

tems, the impact of digital systems on our lives is truly mind boggling! Digital systems play central roles in our medical diagnosis and treatment, our educa tional institutions and workplaces, in moving from place to place, in our homes, in interacting with others, and in just having fun! Considering the complexity of many of these systems, it is a wonder that they work at all. Thanks to the inven tion of the transistor and the integrated circuit and to the ingenuity and persever ance of millions of engineers and programmers, they indeed work and usually work well.

**More on the Generic Computer**

At this point, we will briefly discuss the generic computer and relate its various parts to the block diagram in Figure 2. At the lower left of the diagram at the beginning of this chapter is the heart of the computer, an integrated circuit called the *processor.* Modern processors such as this one are quite complex and consist of tens to hundreds of millions of transistors. The processor contains four functional modules: the CPU, the FPU, the MMU, and the internal cache.

We have already discussed the CPU. The FPU (*floating-point unit*) is some

what like the CPU, except that its datapath and control unit are specifically designed to perform floating-point operations. In essence, these operations pro 1.234 10 7

cess information represented in the form of scientific notation (e.g., ), permitting the generic computer to handle very large and very small numbers. The CPU and the FPU, in relation to Figure 2, each contain a datapath and a control unit.

The MMU is the *memory management unit*. The MMU plus the internal cache

and the separate blocks near the bottom of the computer labeled “External Cache” and “RAM” (*random-access memory*) are all part of the memory in Figure 2. The two caches are special kinds of memory that allow the CPU and FPU to get at the data to be processed much faster than with RAM alone. RAM is what is most com monly referred to as memory. As its main function, the MMU causes the memory

11

**DIGITAL SYSTEMS AND INFORMATION**

that appears to be available to the CPU to be much, much larger than the actual size of the RAM. This is accomplished by data transfers between the RAM and the hard drive shown at the top of the drawing of the generic computer. So the hard drive, which we discuss later as an input/output device, conceptually appears as a part of both the memory and input/output.

The connection paths shown between the processor, memory, and external

cache are the pathways between integrated circuits. These are typically imple mented as fine copper conductors on a printed circuit board. The connection paths below the bus interface are referred to as the processor bus. The connections above the bus interface are the input/output (I/O) bus. The processor bus and the I/O bus attached to the bus interface carry data having different numbers of bits and have different ways of controlling the movement of data. They may also operate at dif ferent speeds. The bus interface hardware handles these differences so that data can be communicated between the two buses.

All of the remaining structures in the generic computer are considered part of

I/O in Figure 2. In terms of sheer physical volume, these structures dominate. In order to enter information into the computer, a keyboard is provided. In order to view output in the form of text or graphics, a graphics adapter card and LCD (*liquid crystal display*) screen are provided. The hard drive, discussed previously, is an elec tromechanical magnetic storage device. It stores large quantities of information in the form of magnetic flux on spinning disks coated with magnetic materials. In order to control the hard drive and transfer information to and from it, a drive controller is used. The keyboard, graphics adapter card, and drive controller card are all attached to the I/O bus. This allows these devices to communicate through the bus interface with the CPU and other circuitry connected to the processor buses.

The generic computer consists mainly of an interconnection of digital modules.

To understand the operation of each module, we need a basic knowledge of digital systems and their general behavior.

Earlier, we mentioned that a digital computer manipulates discrete elements

of information and that all information in the computer is represented in binary 12

**DIGITAL SYSTEMS AND INFORMATION**

form. Operands used for calculations may be expressed in the binary number sys tem or in the decimal system by means of a binary code. The letters of the alpha bet are also converted into a binary code. The remainder of this chapter introduces the binary number system, binary arithmetic, and selected binary codes as a basis for further study. In relation to the generic computer, this material is very important and spans all of the components, excepting some in I/O that involve mechanical operations and analog (as contrasted with digital) electronics.

**2 NUMBER SYSTEMS**

The decimal number system is employed in everyday arithmetic to represent numbers by strings of digits. Depending on its position in the string, each digit has an associated value of an integer raised to the power of 10. For example, the decimal number 724.5 is interpreted to represent 7 hundreds plus 2 tens plus 4 units plus 5 tenths. The hundreds, tens, units, and tenths are powers of 10 implied by the position of the digits. The value of the number is computed as follows:

724.5 7 10 2 2 10 1 4 10 0 5 10  1

The convention is to write only the digits and infer the corresponding powers of 10 from their positions. In general, a decimal number with *n* digits to the left of the decimal point and *m* digits to the right of the decimal point is represented by a string of coefficients:

*An*  1*An*  2…*A*1*A*0.*A* 1*A* 2…*A**m*  1*A**m*

Each coefficient *Ai* is one of 10 digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9). The subscript value *i* gives the position of the coefficient and, hence, the weight 10*i* by which the coefficient must be multiplied.

The decimal number system is said to be of *base* or *radix* 10, because the

coefficients are multiplied by powers of 10 and the system uses 10 distinct digits. In general, a number in base *r* contains *r* digits, 0, 1, 2, ..., *r* − 1, and is expressed as a power series in *r* with the general form

*An*  1*rn*  1 *An*  2*rn*  2 … *A*1*r*1 *A*0*r*0

*A* 1*r* 1 *A* 2*r* 2 … *A**m*  1*r**m*  1 *A**mr**m*

When the number is expressed in positional notation, only the coefficients and the radix point are written down:

*An*  1*An*  2…*A*1*A*0.*A* 1*A* 2 …*A**m*  1*A**m*

In general, the “.” is called the *radix point*. *An – 1* is referred to as the *most signifi cant digit* (*msd*) and *A**m* as the *least significant digit* (*lsd*) of the number. Note that 13

**DIGITAL SYSTEMS AND INFORMATION**

if *m*  0, the lsd is *A* 0  *A*0. To distinguish between numbers of different bases, it is customary to enclose the coefficients in parentheses and place a subscript after the right parenthesis to indicate the base of the number. However, when the con text makes the base obvious, it is not necessary to use parentheses. The following illustrates a base 5 number with *n*  3 and *m*  1 and its conversion to decimal:

( ) 312.4 5 3 5 2 1 5 1 2 5 0 4 5  1

75 5 2 0.8 ( ) 82.8 10

Note that for all the numbers without the base designated, the arithmetic is per formed with decimal numbers. Note also that the base 5 system uses only five dig its, and, therefore, the values of the coefficients in a number can be only 0, 1, 2, 3, and 4 when expressed in that system.

An alternative method for conversion to base 10 that reduces the number of

operations is based on a factored form of the power series:

… *An*  1*r A*  *n*  2 ( )*r A*  *n*  3 ( )*r* … *A*1 ( )*r A*  0

(*A* 1  (*A* 2 (*A* 3 … (*A**m*  2 (*A**m*  1 *A**mr* 1)*r* 1)*r* 1  …)*r* 1)*r* 1)*r* 1 For the example above,

( ) 312.4 5 (( ) 351 5) 245  1

16 5 2 0.8 ( ) 82.8 10

In addition to decimal, three number systems are used in computer work:

binary, octal, and hexadecimal. These are base 2, base 8, and base 16 number sys tems, respectively.

**Binary Numbers**

The binary number system is a base 2 system with two digits: 0 and 1. A binary number such as 11010.11 is expressed with a string of 1s and 0s and, possibly, a binary point. The decimal equivalent of a binary number can be found by expand ing the number into a power series with a base of 2. For example,

( ) 11010 2 1 2 4 1 2 3 0 2 2 1 2 1 0 2 0 ( ) 26 10

As noted earlier, the digits in a binary number are called bits. When a bit is equal to 0, it does not contribute to the sum during the conversion. Therefore, the con version to decimal can be obtained by adding the numbers with powers of two cor responding to the bits that are equal to 1. For example,

( ) 110101.11 2 32 16 4 1 0.5 0.25 53.75 ( ) 10

14

**DIGITAL SYSTEMS AND INFORMATION**

**TABLE 2**

**Powers of Two**

***n* 2*n n* 2*n n* 2*n***

0 1 8 256 16 65,536

1 2 9 512 17 131,072

2 4 10 1,024 18 262,144

3 8 11 2,048 19 524,288

4 16 12 4,096 20 1,048,576

5 32 13 8,192 21 2,097,152

6 64 14 16,384 22 4,194,304

7 128 15 32,768 23 8,388,608

The first 24 numbers obtained from 2 to the power of *n* are listed in Table 2. In

digital systems, we refer to 210 as K (kilo), 220 as M (mega), 230 as G (giga), and 240 as T (tera). Thus,

4K 22 ⋅ 210 = 212 = 4096 and 16M 24 ⋅ 220 = 224 = 16,777,216

This convention does not necessarily apply in all cases, with more conventional us age of K, M, G, and T as 103, 106, 109 and 1012, respectively, sometimes applied as well. So caution is necessary in interpreting and using this notation.

The conversion of a decimal number to binary can be easily achieved by a

method that successively subtracts powers of two from the decimal number. To convert the decimal number *N* to binary, first find the greatest number that is a power of two (see Table 2) and that, subtracted from *N*, produces a positive differ ence. Let the difference be designated *N*1. Now find the greatest number that is a power of two and that, subtracted from *N*1, produces a positive difference *N*2. Continue this procedure until the difference is zero. In this way, the decimal num ber is converted to its powers-of-two components. The equivalent binary number is obtained from the coefficients of a power series that forms the sum of the compo nents. 1s appear in the binary number in the positions for which terms appear in the power series, and 0s appear in all other positions. This method is demonstrated by the conversion of decimal 625 to binary as follows:

625 512 113 *N*1 512 2 9

113 64 49 *N*2 64 2 6

49 32 17 *N*3 32 2 5

17 16 1 *N*4 16 2 4

1 1 0 *N*5 1 2 0

( ) 625 10 29 26 25 24 20 ( ) 1001110001 2

15

**DIGITAL SYSTEMS AND INFORMATION**

**Octal and Hexadecimal Numbers**

As previously mentioned, all computers and digital systems use the binary repre sentation. The octal (base 8) and hexadecimal (base 16) systems are useful for rep resenting binary quantities indirectly because their bases are powers of two. Since 2*3*  8 and 2*4*  16, each octal digit corresponds to three binary digits and each hexadecimal digit corresponds to four binary digits.

The more compact representation of binary numbers in either octal or

hexadecimal is much more convenient for people than using bit strings in binary that are three or four times as long. Thus, most computer manuals use either octal or hexadecimal numbers to specify binary quantities. A group of 15 bits, for example, can be represented in the octal system with only five digits. A group of 16 bits can be represented in hexadecimal with four digits. The choice between an octal and a hexadecimal representation of binary numbers is arbitrary, although hexadecimal tends to win out, since bits often appear in strings of size divisible by four.

The octal number system is the base 8 system with digits 0, 1, 2, 3, 4, 5, 6, and

7. An example of an octal number is 127.4. To determine its equivalent decimal value, we expand the number in a power series with a base of 8:

( ) 127.4 8 1 8 2 2 8 1 7 8 0 4 8  1 ( ) 87.5 10

Note that the digits 8 and 9 cannot appear in an octal number.

It is customary to use the first *r* digits from the decimal system, starting with

0, to represent the coefficients in a base *r* system when *r* is less than 10. The letters of the alphabet are used to supplement the digits when *r* is 10 or more. The hexa decimal number system is a base 16 system with the first 10 digits borrowed from the decimal system and the letters A, B, C, D, E, and F used for the values 10, 11, 12, 13, 14, and 15, respectively. An example of a hexadecimal number is

(B65F)16 11 16 3 6 16 2 5 161 15 16 0 ( ) 46687 10

The first 16 numbers in the decimal, binary, octal, and hexadecimal number sys tems are listed in Table 3. Note that the sequence of binary numbers follows a pre scribed pattern. The least significant bit alternates between 0 and 1, the second significant bit between two 0s and two 1s, the third significant bit between four 0s and four 1s, and the most significant bit between eight 0s and eight 1s.

The conversion from binary to octal is easily accomplished by partitioning

the binary number into groups of three bits each, starting from the binary point and proceeding to the left and to the right. The corresponding octal digit is then assigned to each group. The following example illustrates the procedure:

010 110 001 101 011. 111 100 000 110)2 ( ( ) 26153.7406 8

The corresponding octal digit for each group of three bits is obtained from the first eight entries in Table 3. To make the total count of bits a multiple of three, 0s can be added on the left of the string of bits to the left of the binary point. More importantly, 0s must

16

**DIGITAL SYSTEMS AND INFORMATION**

**TABLE 3**

**Numbers with Different Bases**

**Decimal (base 10)**

00

01

02

03

04

05

06

07

08

09

10

11

12

13

14

15

**Binary (base 2)**

0000

0001

0010

0011

0100

0101

0110

0111

1000

1001

1010

1011

1100

1101

1110

1111

**Octal**

**(base 8)**

00

01

02

03

04

05

06

07

10

11

12

13

14

15

16

17

**Hexadecimal (base 16)**

0

1

2

3

4

5

6

7

8

9

A

B

C

D

E

F

be added on the right of the string of bits to the right of the binary point to make the number of bits a multiple of three and obtain the correct octal result.

Conversion from binary to hexadecimal is similar, except that the binary

number is divided into groups of four digits, starting at the binary point. The previ ous binary number is converted to hexadecimal as follows:

(0010 1100 0110 1011. 1111 0000 0110)2 ( ) 2C6B.F06 16

The corresponding hexadecimal digit for each group of four bits is obtained by ref erence to Table 3.

Conversion from octal or hexadecimal to binary is done by reversing the pro

cedure just performed. Each octal digit is converted to a 3-bit binary equivalent, and extra 0s are deleted. Similarly, each hexadecimal digit is converted to its 4-bit binary equivalent. This is illustrated in the following examples:

(673.12)8 = 110 111 011. 001 010 = (110111011.00101)2

(3A6.C)16 = 0011 1010 0110. 1100 = (1110100110.11)2

**Number Ranges**

In digital computers, the range of numbers that can be represented is based on the number of bits available in the hardware structures that store and process informa tion. The number of bits in these structures is most frequently a power of two, such as 8, 16, 32, and 64. Since the numbers of bits is fixed by the structures, the addition of leading or trailing zeros to represent numbers is necessary, and the range of numbers that can be represented is also fixed.

17

**DIGITAL SYSTEMS AND INFORMATION**

For example, for a computer processing 16-bit unsigned integers, the num

ber 537 is represented as 0000001000011001. The range of integers that can be handled by this representation is from 0 to 216  1, that is, from 0 to 65,535. If the same computer is processing 16-bit unsigned fractions with the binary point to the left of the most significant digit, then the number 0.375 is represented by 0.0110000000000000. The range of fractions that can be represented is from 0 to (216  1)/216, or from 0.0 to 0.9999847412.

We will deal with fixed-bit representations and ranges for binary signed

numbers and floating-point numbers. In both of these cases, some bits are used to represent information other than simple integer or fraction values.

**3 ARITHMETIC OPERATIONS**

Arithmetic operations with numbers in base *r* follow the same rules as for decimal numbers. However, when a base other than the familiar base 10 is used, one must be careful to use only *r* allowable digits and perform all computations with base *r* digits. Examples of the addition of two binary numbers are as follows (note the names of the operands for addition):

Carries: 00000 101100

Augend: 01100 10110

Addend: 10001 10111

Sum: 11101 101101

The sum of two binary numbers is calculated following the same rules as for deci mal numbers, except that the sum digit in any position can be only 1 or 0. Also, a carry in binary occurs if the sum in any bit position is greater than 1. (A carry in decimal occurs if the sum in any digit position is greater than 9.) Any carry obtained in a given position is added to the bits in the column one significant posi tion higher. In the first example, since all of the carries are 0, the sum bits are sim ply the sum of the augend and addend bits. In the second example, the sum of the bits in the second column from the right is 2, giving a sum bit of 0 and a carry bit of 1 (2 2 0). The carry bit is added with the 1s in the third position, giving a sum of 3, which produces a sum bit of 1 and a carry of 1 (3 2 1).

The following are examples of the subtraction of two binary numbers; as with

addition, note the names of the operands:

Borrows: 00000 00110 00110

Minuend: 10110 10110 10011 11110

Subtrahend: 10010 10011 11110 10011

Difference: 00100 00011 01011

18

**DIGITAL SYSTEMS AND INFORMATION**

The rules for subtraction are the same as in decimal, except that a borrow into a given column adds 2 to the minuend bit. (A borrow in the decimal system adds 10 to the minuend digit.) In the first example shown, no borrows occur, so the differ ence bits are simply the minuend bits minus the subtrahend bits. In the second example, in the right position, the subtrahend bit is 1 with the minuend bit 0, so it is necessary to borrow from the second position as shown. This gives a difference bit in the first position of 1 (2 0 1 1). In the second position, the borrow is subtracted, so a borrow is again necessary. Recall that, in the event that the subtra hend is larger than the minuend, we subtract the minuend from the subtrahend and give the result a minus sign. This is the case in the third example, in which this interchange of the two operands is shown.

The final operation to be illustrated is binary multiplication, which is quite simple. The multiplier digits are always 1 or 0. Therefore, the partial products are equal either to the multiplicand or to 0. Multiplication is illustrated by the follow

ing example:

Multiplicand: 1011 Multiplier: ⋅ 101

1011

00000

101100

Product: 110111

Arithmetic operations with octal, hexadecimal, or any other base *r* system will normally require the formulation of tables from which one obtains sums and products of two digits in that base. An easier alternative for adding two numbers in base *r* is to convert each pair of digits in a column to decimal, add the digits in decimal, and then convert the result to the corresponding sum and carry in the base *r* system. Since addition is done in decimal, we can rely on our memories for obtaining the entries from the familiar decimal addition table. The sequence of steps for adding the two hexadecimal numbers 59F and E46 is shown in Example 2.

**EXAMPLE 2 Hexadecimal Addition**

Perform the addition

( ) 59F 16 ( ) E46 16  :

**Hexadecimal Equivalent Decimal Calculation**

1 1

5 9 F 5 Carry 9 15 Carry

E 4 6 14 4 6

1 3 E 5 1 19 16 3 14 E 21 16 5

19

**DIGITAL SYSTEMS AND INFORMATION**

The equivalent decimal calculation columns on the right show the mental reason ing that must be carried out to produce each digit of the hexadecimal sum. Instead of adding F 6 in hexadecimal, we add the equivalent decimals, 15 6 21. We then convert back to hexadecimal by noting that 21 16 5. This gives a sum digit of 5 and a carry of 1 to the next higher-order column of digits. The other two columns are added in a similar fashion. ■

In general, the multiplication of two base *r* numbers can be accomplished by doing all the arithmetic operations in decimal and converting intermediate results one at a time. This is illustrated in the multiplication of two octal numbers shown in Example 3.

**EXAMPLE 3 Octal Multiplication**

Perform the multiplication

( ) 762 8 ( ) 45 8:

**Octal Octal Decimal Octal**

7 6 2 5 ⋅ 2 10 8 2 12

4 5 5 ⋅ 6 1 31 24 7 37

4 6 7 2 5 ⋅ 7 3 38 32 6 46

3 7 1 0 0 4 ⋅ 2 8 8 0 10

4 3 7 7 2 4 ⋅ 6 1 25 24 1 31

4 ⋅ 7 3 31 24 7 37

Shown on the right are the mental calculations for each pair of octal digits. The octal digits 0 through 7 have the same value as their corresponding decimal digits. The multiplication of two octal digits plus a carry, derived from the calculation on the previous line, is done in decimal, and the result is then converted back to octal. The left digit of the two-digit octal result gives the carry that must be added to the digit product on the next line. The blue digits from the octal results of the decimal calcu lations are copied to the octal partial products on the left. For example, ( ) 5 2 8 ( ) 12 8  . ( ) 5 6 8,

The left digit, 1, is the carry to be added to the product

and the blue least significant digit, 2, is the corresponding digit of the octal partial product. When there is no digit product to which the carry can be added, the carry is written directly into the octal partial product, as in the case of the 4 in 46. ■

**Conversion from Decimal to Other Bases**

We convert a number in base *r* to decimal by expanding it in a power series and adding all the terms, as shown previously. We now present a general procedure for the operation of converting a decimal number to a number in base *r* that is the reverse of the alternative expansion to base 10 as discussed earlier. If the number includes a radix point, we need to separate the number into an integer part and a fraction part, since the two parts must be converted differently. The conversion of a decimal integer to a number in base *r* is done by dividing the number and all successive

20

**DIGITAL SYSTEMS AND INFORMATION**

quotients by *r* and accumulating the remainders. This procedure is best explained by example.

**EXAMPLE 4 Conversion of Decimal Integers to Octal**

Convert decimal 153 to octal:

The conversion is to base 8. First, 153 is divided by 8 to give a quotient of 19 and a remainder of 1, as shown in blue. Then 19 is divided by 8 to give a quotient of 2 and a remainder of 3. Finally, 2 is divided by 8 to give a quotient of 0 and a remainder of 2. The coefficients of the desired octal number are obtained from the remainders:

153/8 19 1/8 Remainder 1 Least significant digit

19/8 29 3/8 3

2/8 0 2/8 2 Most significant digit

(153)10  (231)8 ■

Note in Example 4 that the remainders are read from last to first, as indicated

by the arrow, to obtain the converted number. The quotients are divided by *r* until the result is 0. We also can use this procedure to convert decimal integers to binary, as shown in Example 5. In this case, the base of the converted number is 2, and therefore, all the divisions must be done by 2.

**EXAMPLE 5 Conversion of Decimal Integers to Binary**

Convert decimal 41 to binary:

41/2 20 1/2 Remainder 1 Least significant digit

20/2 10 0

10/2 5 0

5/2 2 1/2 1

2/2 1 0

1/2 0 1/2 1 Most significant digit

(4l)10  (l0l001)2

Of course, the decimal number could be converted by the sum of powers of two: (41)10  32 8 1 (101001)2 ■

The conversion of a decimal fraction to base *r* is accomplished by a method

similar to that used for integers, except that multiplication by *r* is used instead of 21

**DIGITAL SYSTEMS AND INFORMATION**

division, and integers are accumulated instead of remainders. Again, the method is best explained by example.

**EXAMPLE 6 Conversion of Decimal Fractions to Binary**

Convert decimal 0.6875 to binary:

First, 0.6875 is multiplied by 2 to give an integer and a fraction. The new fraction is multiplied by 2 to give a new integer and a new fraction. This process is continued until the fractional part equals 0 or until there are enough digits to give sufficient accuracy. The coefficients of the binary number are obtained from the integers as follows:

0.6875 ⋅ 2 1.3750 Integer 1 Most significant digit

0.3750 ⋅ 2 0.7500 0

0.7500 ⋅ 2 1.5000 1

0.5000 ⋅ 2 1.0000 1 Least significant digit

(0.6875)10  (0.1011)2 ■

Note in the foregoing example that the integers are read from first to last, as

indicated by the arrow, to obtain the converted number. In the example, a finite number of digits appear in the converted number. The process of multiplying frac tions by *r* does not necessarily end with zero, so we must decide how many digits of the fraction to use from the conversion. Also, remember that the multiplications are by number *r*. Therefore, to convert a decimal fraction to octal, we must multi ply the fractions by 8, as shown in Example 7.

**EXAMPLE 7 Conversion of Decimal Fractions to Octal**

Convert decimal 0.513 to a three-digit octal fraction:

0.513 ⋅ 8 4.104 Integer 4 Most significant digit

0.104 ⋅ 8 0.832 0

0.832 ⋅ 8 6.656 6

0.656 ⋅ 8 5.248 5 Least significant digit

The answer, to three significant figures, is obtained from the integer digits. Note that the last integer digit, 5, is used for rounding in base 8 of the second-to-the-last digit, 6, to obtain

(0.513)10  (0.407)8 ■

The conversion of decimal numbers with both integer and fractional parts is

done by converting each part separately and then combining the two answers. Using the results of Example 4 and Example 7, we obtain

(153.513)10  (23l.407)8

22

**DIGITAL SYSTEMS AND INFORMATION**

**4 DECIMAL CODES**

The binary number system is the most natural one for a computer, but people are accustomed to the decimal system. One way to resolve this difference is to convert decimal numbers to binary, perform all arithmetic calculations in binary, and then convert the binary results back to decimal. This method requires that we store the decimal numbers in the computer in such a way that they can be converted to binary. Since the computer can accept only binary values, we must represent the decimal digits by a code that contains 1s and 0s. It is also possible to perform the arithmetic operations directly with decimal numbers when they are stored in the computer in coded form.

An *n*-bit *binary code* is a group of *n* bits that assume up to 2*n* distinct combi

nations of 1s and 0s, with each combination representing one element of the set being coded. A set of four elements can be coded with a 2-bit binary code, with each element assigned one of the following bit combinations: 00, 01, 10, 11. A set of 8 elements requires a 3-bit code, and a set of 16 elements requires a 4-bit code. The bit combinations of an *n*-bit code can be determined from the count in binary from 0 to 2*n*  1. Each element must be assigned a unique binary bit combination, and no two elements can have the same value; otherwise, the code assignment is ambiguous.

A binary code will have some unassigned bit combinations if the number of

elements in the set is not a power of 2. The ten decimal digits form such a set. A binary code that distinguishes among ten elements must contain at least four bits, but six out of the 16 possible combinations will remain unassigned. Numerous dif ferent binary codes can be obtained by arranging four bits into 10 distinct combi nations. The code most commonly used for the decimal digits is the straightforward binary assignment listed in Table 3. This is called *binary-coded decimal* and is com monly referred to as BCD. Other decimal codes are possible.

Table 4 gives a 4-bit code for each decimal digit. A number with *n* decimal

digits will require 4*n* bits in BCD. Thus, decimal 396 is represented in BCD with 12 bits as

0011 1001 0110

with each group of four bits representing one decimal digit. A decimal number in BCD is the same as its equivalent binary number only when the number is between 0 and 9, inclusive. A BCD number greater than 10 has a representation different from its equivalent binary number, even though both contain 1s and 0s. Moreover, the binary combinations 1010 through 1111 are not used and have no meaning in the BCD code.

Consider decimal 185 and its corresponding value in BCD and binary:

(185)10  (0001 1000 0101)BCD  (10111001)2

23

**DIGITAL SYSTEMS AND INFORMATION**

**TABLE 4**

**Binary-Coded Decimal (BCD)**

**Decimal Symbol**

0

1

2

3

4

5

6

7

8

9

**BCD Digit**

0000 0001 0010 0011 0100 0101 0110 0111 1000 1001

The BCD value has 12 bits, but the equivalent binary number needs only 8 bits. It is obvious that a BCD number needs more bits than its equivalent binary value. However, BCD representation of decimal numbers is still important, because com puter input and output data used by most people needs to be in the decimal sys tem. BCD numbers are decimal numbers and not binary numbers, even though they are represented using bits. The only difference between a decimal and a BCD number is that decimals are written with the symbols 0, 1, 2, ..., 9, and BCD num bers use the binary codes 0000, 0001, 0010, ..., 1001.

**BCD Addition**

Consider the addition of two decimal digits in BCD, together with a possible carry of 1 from a previous less significant pair of digits. Since each digit does not exceed 9, the sum cannot be greater than 9 9 1 19, the 1 being a carry. Suppose we add the BCD digits as if they were binary numbers. Then the binary sum will produce a result in the range from 0 to 19. In binary, this will be from 0000 to 10011, but in BCD, it should be from 0000 to 1 1001, the first 1 being a carry and the next four bits being the BCD digit sum. When the binary sum is less than 1010 (without a carry), the corresponding BCD digit is correct. But when the binary sum is greater than or equal to 1010, the result is an invalid BCD digit. The addition of binary 6, (0110)2, to the sum converts it to the correct digit and also produces a decimal carry as required. The reason is that the differ ence between a carry from the most significant bit position of the binary sum and a decimal carry is 16 10 6. Thus, the decimal carry and the correct BCD sum

24

**DIGITAL SYSTEMS AND INFORMATION**

digit are forced by adding 6 in binary. Consider the next three-digit BCD addi tion example.

**EXAMPLE 8 BCD Addition**

110 BCD carry 1 1

448 0100 0100 1000

489 0100 1000 1001

937 Binary sum 1001 1101 1 0001

Add 6 0110 0110

BCD sum 1 0011 1 0111

BCD result 1001 0011 0111

In each position, the two BCD digits are added as if they were two binary numbers. If the binary sum is greater than 1001, we add 0110 to obtain the correct BCD digit sum and a carry. In the right column, the binary sum is equal to 17. The presence of the carry indicates that the sum is greater than 16 (certainly greater than 9), so a correction is needed. The addition of 0110 produces the correct BCD digit sum, 0111 (7), and a carry of 1. In the next column, the binary sum is 1101 (13), an invalid BCD digit. Addition of 0110 produces the correct BCD digit sum, 0011 (3), and a carry of 1. In the final column, the binary sum is equal to 1001 (9) and is the correct BCD digit. ■

**5 ALPHANUMERIC CODES**

Many applications of digital computers require the handling of data consisting not only of numbers, but also of letters. For instance, an insurance company with thou sands of policyholders uses a computer to process its files. To represent the names and other pertinent information, it is necessary to formulate a binary code for the letters of the alphabet. In addition, the same binary code must represent numerals and special characters such as $. Any alphanumeric character set for English is a set of elements that includes the ten decimal digits, the 26 letters of the alphabet, and several (more than three) special characters. If only capital letters are included, we need a binary code of at least six bits, and if both uppercase letters and lowercase letters are included, we need a binary code of at least seven bits. Binary codes play an important role in digital computers. The codes must be in binary because computers can handle only 1s and 0s. Note that binary encoding merely changes the symbols, not the meaning of the elements of information being encoded.

25

**DIGITAL SYSTEMS AND INFORMATION**

**ASCII Character Code**

The standard binary code for the alphanumeric characters is called ASCII (American Standard Code for Information Interchange). It uses seven bits to code 128 characters, as shown in Table 5. The seven bits of the code are desig nated by *B*1 through *B*7 , with *B*7 being the most significant bit. Note that the most significant three bits of the code determine the column of the table and the least significant four bits the row of the table. The letter A, for example, is represented in ASCII as 1000001 (column 100, row 0001). The ASCII code con tains 94 characters that can be printed and 34 nonprinting characters used for various control functions. The printing characters consist of the 26 uppercase letters, the 26 lowercase letters, the 10 numerals, and 32 special printable char acters such as %, @, and $.

The 34 control characters are designated in the ASCII table with abbreviated

names. They are listed again below the table with their full functional names. The control characters are used for routing data and arranging the printed text into a prescribed format. There are three types of control characters: format effectors, information separators, and communication control characters. Format effectors are characters that control the layout of printing. They include the familiar type writer controls such as backspace (BS), horizontal tabulation (HT), and carriage return (CR). Information separators are used to separate the data into divisions— for example, paragraphs and pages. They include characters such as record separa tor (RS) and file separator (FS). The communication control characters are used during the transmission of text from one location to the other. Examples of com munication control characters are STX (start of text) and ETX (end of text), which are used to frame a text message transmitted via communication wires.

ASCII is a 7-bit code, but most computers manipulate an 8-bit quantity as a

single unit called a *byte*. Therefore, ASCII characters most often are stored one per byte, with the most significant bit set to 0. The extra bit is sometimes used for spe cific purposes, depending on the application. For example, some printers recognize an additional 128 8-bit characters, with the most significant bit set to 1. These char acters enable the printer to produce additional symbols, such as those from the Greek alphabet or characters with accent marks as used in languages other than English.

**UNICODE** This supplement on Unicode, a 16-bit standard code for representing the symbols and ideographs for the world’s languages, is available on the Companion Website (http://www.prenhall.com/mano) for the text.

**Parity Bit**

To detect errors in data communication and processing, an additional bit is some times added to a binary code word to define its parity. A *parity bit* is the extra bit

26

**DIGITAL SYSTEMS AND INFORMATION**

**TABLE 5**

**American Standard Code for Information Interchange (ASCII)**

**B**7**B**6**B**5

**B**4**B**3**B**2**B**1 **000 001 010 011 100 101 110 111**

0000 NULL DLE SP 0 @ P ` p 0001 SOH DC1 ! 1 A Q a q 0010 STX DC2 " 2 B R b r 0011 ETX DC3 # 3 C S c s 0100 EOT DC4 $ 4 D T d t 0101 ENQ NAK % 5 E U e u 0110 ACK SYN & 6 F V f v

0111 BEL ETB 7 G W g w 1000 BS CAN ( 8 H X h x 1001 HT EM ) 9 I Y i y 1010 LF SUB \* : J Z j z 1011 VT ESC + ; K [ k { 1100 FF FS , < L \ l |

1101 CR GS - = M ] m } 1110 SO RS . > N ^ n ~ 1111 SI US / ? O \_ o DEL

**Control Characters**

NULL NULL DLE Data link escape

SOH Start of heading DC1 Device control 1

STX Start of text DC2 Device control 2

ETX End of text DC3 Device control 3

EOT End of transmission DC4 Device control 4

ENQ Enquiry NAK Negative acknowledge ACK Acknowledge SYN Synchronous idle

BEL Bell ETB End of transmission block BS Backspace CAN Cancel

HT Horizontal tab EM End of medium

LF Line feed SUB Substitute

VT Vertical tab ESC Escape

FF Form feed FS File separator

CR Carriage return GS Group separator

SO Shift out RS Record separator

SI Shift in US Unit separator

SP Space DEL Delete

27

**DIGITAL SYSTEMS AND INFORMATION**

included to make the total number of 1s in the resulting code word either even or odd. Consider the following two characters and their even and odd parity:

**With Even Parity With Odd Parity**

1000001 01000001 11000001

1010100 11010100 01010100

In each case, we use the extra bit in the most significant position of the code to pro duce an even number of 1s in the code for *even parity* or an odd number of 1s in the code for *odd parity*. In general, one parity or the other is adopted, with even parity being more common. Parity may be used with binary numbers as well as with codes, including ASCII for characters, and the parity bit may be placed in any fixed position in the code.

**EXAMPLE 9 Error Detection and Correction for ASCII Transmission**

The parity bit is helpful in detecting errors during the transmission of information from one location to another. Assuming that even parity is used, the simplest case is handled as follows: An even (or odd) parity bit is generated at the sending end for all 7-bit ASCII characters; the 8-bit characters that include parity bits are trans mitted to their destination. The parity of each character is then checked at the receiving end; if the parity of the received character is not even (odd), it means that at least one bit has changed its value during the transmission. This method detects one, three, or any odd number of errors in each character transmitted. An even number of errors is undetected. Other error-detection codes, some of which are based on additional parity bits, may be needed to take care of an even number of errors. What is done after an error is detected depends on the particular applica tion. One possibility is to request retransmission of the message on the assumption that the error was random and will not occur again. Thus, if the receiver detects a parity error, it sends back a NAK (negative acknowledge) control character con sisting of the even-parity eight bits, 10010101, from Table 5. If no error is detected, the receiver sends back an ACK (acknowledge) control character, 00000110. The sending end will respond to a NAK by transmitting the message again, until the correct parity is received. If, after a number of attempts, the transmission is still in error, an indication of a malfunction in the transmission path is given. ■

**6 GRAY CODES**

As we count up or down using binary codes, the number of bits that change from one binary value to the next varies. This is illustrated by the binary code for the octal digits on the left in Table 6. As we count from 000 up to 111 and “roll over”

28

**DIGITAL SYSTEMS AND INFORMATION**

**TABLE 6**

**Gray Code**

**Binary Code**

000

001

010

011

100

101

110

111

000

**Bit**

**Changes**

1

2

1

3

1

2

1

3

**Gray Code**

000

001

011

010

110

111

101

100

000

**Bit**

**Changes**

1

1

1

1

1

1

1

1

to 000, the number of bits that change between the binary values ranges from 1 to 3.

For many applications, multiple bit changes as the circuit counts is not a problem. There are applications, however, in which a change of more than one bit when counting up or down can cause serious problems. One such problem is illustrated by an optical shaft-angle encoder shown in Figure 5(a). The encoder is a disk attached to a rotating shaft for measurement of the rotational position of the shaft. The disk contains areas that are clear for binary 1 and opaque for binary 0. An illumination source is placed on one side of the disk, and optical sensors, one for each of the bits to be encoded, are placed on the other side of the disk. When a clear region lies between the source and a sensor, the sensor responds to the light with a binary 1 output. When an opaque region lies between the source and the sensor, the sensor responds to the dark with a binary 0.

The rotating shaft, however, can be in any angular position. For example, suppose that the shaft and disk are positioned so that the sensors lie right at the

111

110

B0

B1

B2

000

001

101

100 000

001

101

010

100 011

111

G0G1G2

110 010

011

(a) Binary code for positions 0 through 7 **FIGURE 5**

(b) Gray code for positions 0 through 7

Optical Shaft-Angle Encoder

29

**DIGITAL SYSTEMS AND INFORMATION**

boundary between 011 and 100. In this case, sensors in positions B2, B1 and B0 have the light partially blocked. In such a situation, it is unclear whether the three sensors will see light or dark. As a consequence, each sensor may produce either a 1 or a 0. Thus, the resulting encoded binary number for a value between 3 and 4 may be 000, 001, 010, 011, 100, 101, 110, or 111. Either 011 or 100 will be satisfac tory in this case, but the other six values are clearly erroneous!

To see the solution to this problem, notice that in those cases in which only

a single bit changes when going from one value to the next or previous value, this problem cannot occur. For example, if the sensors lie on the boundary between 2 and 3, the resulting code is either 010 or 011, either of which is satis factory. If we change the encoding of the values 0 through 7 such that only one bit value changes as we count up or down (including rollover from 7 to 0), then the encoding will be satisfactory for all positions. A code having the property that only one bit at a time changes between codes during counting is a *Gray code* named for Frank Gray, who patented its use for shaft encoders in 1953. There are multiple Gray codes for any set of *n* consecutive integers, with *n* even.

A specific Gray code for the octal digits, called a *binary reflected Gray code*,

appears on the right in Table 6. Note that the counting order for binary codes is now 000, 001, 011, 010, 110, 111, 101, 100, and 000. If we want binary codes for pro cessing, then we can build a digital circuit or use software that converts these codes to binary before they are used in further processing of the information.

Figure 5(b) shows the optical shaft-angle encoder using the Gray code from

Table 6. Note that any two segments on the disk adjacent to each other have only one region that is clear for one and opaque for the other.

The optical shaft encoder illustrates one use of the Gray code concept. There

are many other similar uses in which a physical variable, such as position or volt age, has a continuous range of values that is converted to a digital representation. A quite different use of Gray codes appears in low-power CMOS (Complementary Metal Oxide Semiconductor) logic circuits that count up or down. In CMOS, power is consumed only when a bit changes. For the example codes given in Table 6 with continuous counting (either up or down), there are 14 bit changes for binary counting for every eight bit changes for Gray code counting. Thus, the power con sumed at the counter outputs for the Gray code counter is only 57 percent of that consumed at the binary counter outputs.

A Gray code for a counting sequence of *n* binary code words (*n* must be

even) can be constructed by replacing each of the first *n*/2 numbers in the sequence with a code word consisting of 0 followed by the even parity for each bit of the binary code word and the bit to its left. For example, for the binary code word 0100, the Gray code word is 0, parity(0, 1), parity(1, 0), parity(0, 0) = 0110. Next, take the sequence of numbers formed and copy it in reverse order with the left most 0 replaced by a 1. This new sequence provides the Gray code words for the second *n*/2 of the original *n* code words. For example, for BCD codes, the first five Gray code words are 0000, 0001, 0011, 0010, and 0110. Reversing the order of these codes and replacing the leftmost 0 with a 1, we obtain 1110, 1010, 1011, 1001, and 1000 for the last five Gray codes. For the special cases in which the original binary codes are 0 through 2*n*  1, each Gray code word may be formed directly from the

30

**DIGITAL SYSTEMS AND INFORMATION**

corresponding binary code word by copying its leftmost bit and then replacing each of the remaining bits with the even parity of the bit of the number and the bit to its left.

**7 CHAPTER SUMMARY**

In this chapter, we introduced digital systems and digital computers and showed why such systems use signals having only two values. We briefly introduced the structure of the stored-program digital computer and showed how computers can be applied to a broad range of specialized applications by using embedded systems. We then related the computer structure to a representative example of a personal computer (PC).

Number-system concepts, including base (radix) and radix point, were pre

sented. Because of their correspondence to two-valued signals, binary numbers were discussed in detail. Octal (base 8) and hexadecimal (base 16) were also emphasized, since they are useful as shorthand notation for binary. Arithmetic operations in bases other than base 10 and the conversion of numbers from one base to another were covered. Because of the predominance of decimal in normal use, Binary-Coded Dec imal (BCD) was treated. The representation of information in the form of characters instead of numbers by means of the ASCII code for the English alphabet was pre sented. The parity bit was presented as a technique for error detection, and the Gray code, which is critical to selected applications, was defined.

**REFERENCES**

**1.** GRAY, F. *Pulse Code Communication.* U. S. Patent 2 632 058, March 17, 1953. **2.** PATTERSON, D. A., AND J. L. HENNESSY, *Computer Organization and Design: The Hardware/Software Interface,* 3rd ed*.* San Francisco: Morgan Kaufmann,

2004.

**3.** WHITE, R. *How Computers Work: Millennium Edition,* 5th ed. Indianapolis: Que, 1999.

**PROBLEMS**

The plus (+) indicates a more advanced problem, and the asterisk (\*) indicates that a solution is available on the Companion Website for the text.

**1.** This problem concerns wind measurements made by the wireless weather station illustrated in Example 1. The wind-speed measurement uses a

31

**DIGITAL SYSTEMS AND INFORMATION**

rotating anemometer connected by a shaft to an enclosed disk that is one

half clear and one-half black. There is a light above and a photodiode below

the disk in the enclosure. The photodiode produces a 3 V signal when

exposed to light and a 0 V signal when not exposed to light. **(a)** Sketch the

*relative* appearance of voltage waveforms produced by this sensor (1) when

the wind is calm, (2) when the wind is 10 mph, and (3) when the wind is 100

mph. **(b)** Explain verbally what information the microcomputer must have

available and the tasks it must perform to convert the voltage waveforms

produced into a binary number representing wind speed in miles per hour.

**2.** Using the scheme in Example 1, find the discrete, quantized value of voltage and the binary code for each of the following Fahrenheit tempera-tures: −34,

+31, +77, and +108.

**3.** \*List the binary, octal, and hexadecimal numbers from 16 to 31.

**4.** What is the exact number of bits in a memory that contains **(a)** 96K bits; **(b)** 640M bits; **(c)** 4G bits?

**5.** How many bits are in 1 Tb? [*Hint*: Depending on the tool used to calculate this, you may need to use a trick to get the *exact* result. Note that 220 =

1,000,00010 + *d*, where *d* is the difference between 220 and 1,000,00010, and

that 1T = (1,000,00010 + *d*)2. Expand the equation for 1T into a sum-of products form, insert the value of *d*, find the three products, and then find

their sum.]

**6.** What is the decimal equivalent of the largest binary integer that can be obtained with **(a)** 11 bits and **(b)** 25 bits?

**7.** \*Convert the following binary numbers to decimal: 1001101, 1010011.101, and 10101110.1001.

**8.** Convert the following decimal numbers to binary: 193, 751, 2007, and 19450.

**9.** \*Convert the following numbers from the given base to the other three bases listed in the table:

**Decimal Binary Octal Hexadecimal**

369.3125 ? ? ?

? 10111101.101 ? ?

? ? 326.5 ?

? ? ? F3C7.A

**10.** \*Convert the following decimal numbers to the indicated bases, using the methods of Examples 4 and 7:

**(a)** 7562.45 to octal **(b)** 1938.257 to hexadecimal **(c)** 175.175 to binary.

**11.** \*Perform the following conversion by using base 2 instead of base 10 as the intermediate base for the conversion:

**(a)** (673.6)8 to hexadecimal **(b)** (E7C.B)16 to octal **(c)** (310.2)4 to octal

32

**DIGITAL SYSTEMS AND INFORMATION**

**12.** Perform the following binary multiplications:

**(a)** 1101 ⋅ 1011 **(b)** 0101 ⋅ 1010 **(c)** 100111 ⋅ 011011

**13.** +Division is composed of multiplications and subtractions. Perform the binary division 1010110 ⎟ 101 to obtain a quotient and remainder.

**14.** A limited number system uses base 12. There are at most four integer digits. The weights of the digits are 123, 122, 12, and 1. Special names are given to

the weights as follows: 12 = 1 dozen, 122 = 1 gross, and 123 = 1 great gross.

**(a)** How many beverage cans are in 6 great gross + 8 gross + 7 dozen + 4?

**(b)** Find the representation in base 12 for 756910 beverage cans.

**15.** Considerable evidence suggests that base 20 has historically been used for number systems in a number of cultures.

**(a)** Write the digits for a base 20 system, using an extension of the same digit

representation scheme employed for hexadecimal.

**(b)** Convert (2007)10 to base 20. **(c)** Convert (BCI.G)20 to decimal.

**16.** \*In each of the following cases, determine the radix *r*:

**(a)** (BEE)*r*  (2699)10 **(b)** (365)*r*  (194)10

**17.** The following calculation was performed by a particular breed of unusually intelligent chicken. If the radix *r* used by the chicken corresponds to its total

number of toes, how many toes does the chicken have on each foot?

((34)*r* + (24)*r*) ⋅ (21)*r* = (1480)*r*

**18.** \*Find the binary representations for each of the following BCD numbers: **(a)** 0100 1000 0110 0111 **(b)** 0011 0111 1000.0111 0101

**19.** \*Represent the decimal numbers 694 and 835 in BCD, and then show the steps necessary to form their sum.

**20.** \*Internally in the computer, with few exceptions, all numerical computation is done using binary numbers. Input, however, often uses ASCII, which is

formed by appending 011 to the left of a BCD code. Thus, an algorithm that

directly converts a BCD integer to a binary integer is very useful. Here is

one such algorithm:

1. Draw lines between the 4-bit decades in the BCD number.

2. Move the BCD number one bit to the right.

3. Subtract 0011 from each BCD decade containing a binary value > 0111.

4. Repeat steps 2 and 3 until the leftmost 1 in the BCD number has been

moved out of the least significant decade position.

5. Read the binary result to the right of the least significant BCD decade.

**(a)** Execute the algorithm for the BCD number 0111 1000.

**(b)** Execute the algorithm for the BCD number 0011 1001 0111.

33

**DIGITAL SYSTEMS AND INFORMATION**

**21.** Internally in a computer, with few exceptions, all computation is done using binary numbers. Output, however, often uses ASCII, which is formed by

appending 011 to the left of a BCD code. Thus, an algorithm that directly

converts a binary integer to a BCD integer is very useful. Here is one such

algorithm:

1. Draw lines to bound the expected BCD decades to the left of the binary

number.

2. Move the binary number one bit to the left.

3. Add 0011 to each BCD decade containing a binary value > 0100.

4. Repeat steps 2 and 3 until the last bit in the binary number has been

moved into the least significant BCD decade position.

5. Read the BCD result.

**(a)** Execute the algorithm for the binary number 1111000.

**(b)** Execute the algorithm for the binary number 01110010111.

**22.** What bit position in an ASCII code must be complemented to change the ASCII letter represented from uppercase to lowercase and vice versa?

**23.** Write your full name in ASCII, using an 8-bit code **(a)** with the leftmost bit always 0 and **(b)** with the leftmost bit selected to produce even parity.

Include a space between names and a period after the middle initial.

**24.** Decode the following ASCII code: 1000111 1101111 0100000 1000010 1100001 1100100 1100111 1100101 1110010 1110011 0100001.

**25.** \*Show the bit configuration that represents the decimal number 255 in **(a)** binary, **(b)** BCD, **(c)** ASCII, **(d)** ASCII with odd parity.

**26. (a)** List the 6-bit binary number equivalents for 32 through 47 with a parity bit added in the rightmost position, giving odd parity to the overall 7-bit

numbers. **(b)** Repeat for even parity.

**27.** Using the procedure given in Section 5, find the hexadecimal Gray code.

**28.** This problem concerns wind measurements made by the wireless weather station in Example 1. The wind direction is to be measured with a disk

encoder like the one shown in Figure 5(b). **(a)** Assuming that the code 000

corresponds to N, list the Gray code values for each of the directions, S, E,

W, NW, NE, SW, and SE. **(b)** Explain why the Gray code you have assigned

avoids the reporting of major errors in wind direction.

**29.** +What is the percentage of power consumed for continuous counting (either up or down but not both) at the outputs of a binary Gray code counter (with

all 2*n* code words used) compared to a binary counter as a function of the

number of bits, *n*, in the two counters?

34

COMBINATIONAL LOGIC

CIRCUITS

From Chapter 2 of *Logic and Computer Design Fundamentals*, Fourth Edition. M. Morris Mano, Charles R. Kime. Copyright © 2008 by Pearson Education, Inc. Published by Pearson Prentice Hall. All rights reserved.

35

Keyboard

LCD

Screen

Hard Drive

Drive ControllerBus Interface

Graphics Adapter

FPU

Internal Cache

CPU MMU

Processor

External

Cache

Generic Computer

RAM

Note: The companion website for this text is http://www.prenhall.com/mano 36

COMBINATIONAL

LOGIC CIRCUITS

I

n this chapter we will learn about gates, the most primitive logic elements used in digital systems. In addition, we will learn the mathematical techniques for designing circuits from these gates and learn how to design cost-effective circuits.

These techniques, which are fundamental to the design of almost all digital circuits,

are based on Boolean algebra. One aspect of design is to avoid unnecessary circuitry and excess cost, a goal accomplished by a technique called optimization. Karnaugh

maps provide a graphical method for enhancing understanding of logic design and

optimization and solving small optimization problems for “two-level” logic circuits.

More general optimization methods for circuits with two or more levels are introduced. Types of logic gates characteristic of contemporary integrated-circuit implementation

are discussed. Exclusive-OR and exclusive-NOR gates are introduced, along with

associated algebraic techniques.

Concepts from this chapter apply to most of the generic computer. Exceptions are

circuits that are largely memory, such as caches and RAM, and analog electronic

circuits in the monitor and hard disk controller. Nevertheless, with its use throughout

the design of most of the computer, what we study in this chapter is fundamental to

an in-depth understanding of computers and digital systems and how they are

designed.

**1 BINARY LOGIC AND GATES**

Digital circuits are hardware components that manipulate binary information. The circuits are implemented using transistors and interconnections in complex semi conductor devices called *integrated circuits*. Each basic circuit is referred to as a *logic gate*. For simplicity in design, we model the transistor-based electronic circuits as logic gates. Thus, the designer need not be concerned with the internal

37

**COMBINATIONAL LOGIC CIRCUITS**

electronics of the individual gates, but only with their external logic properties. Each gate performs a specific logical operation. The outputs of gates are applied to the inputs of other gates to form a digital circuit.

In order to describe the operational properties of digital circuits, we need to

introduce a mathematical notation that specifies the operation of each gate and that can be used to analyze and design circuits. This binary logic system is one of a class of mathematical systems generally called *Boolean algebras* (after the English mathematician George Boole, who in 1854 published a book introducing the math ematical theory of logic). The specific Boolean algebra we will study is used to describe the interconnection of digital gates and to design logic circuits through the manipulation of Boolean expressions. We first introduce the concept of binary logic and show its relationship to digital gates and binary signals. We then present the properties of the Boolean algebra, together with other concepts and methods use ful in designing logic circuits.

**Binary Logic**

Binary logic deals with binary variables, which take on two discrete values, and with the operations of mathematical logic applied to these variables. The two val ues the variables take may be called by different names, but for our purpose, it is convenient to think in terms of binary values and assign 1 or 0 to each variable. In this chapter, variables are designated by letters of the alphabet, such as *A*, *B*, *C*, *X*, *Y*, and *Z*. Later this notation is expanded to include strings of letters, numbers, and special characters. Associated with the binary variables are three basic logical operations called AND, OR, and NOT:

**1.** AND. This operation is represented by a dot or by the absence of an opera tor. For example, *Z*  *X · Y* or *Z*  *XY* is read “*Z* is equal to *X* AND *Y.*” The

logical operation AND is interpreted to mean that *Z*  1 if and only if *X*  1

and *Y*  1; otherwise *Z*  0. (Remember that *X*, *Y*, and *Z* are binary vari

ables and can be equal to only 1 or 0.)

**2.** OR. This operation is represented by a plus symbol. For example, *Z*  *X*  *Y* is read “*Z* is equal to *X* OR *Y*,” meaning that *Z*  1 if *X*  1 or if *Y*  1, or if

both *X*  l and *Y*  1. *Z*  0 if and only if *X*  0 and *Y*  0.

**3.** NOT. This operation is represented by a bar over the variable. For example, *Z*  is read “*Z* is equal to NOT *X*,” meaning that *Z* is what *X* is not. In

*X*

other words, if *X*  1, then *Z*  0; but if *X*  0, then *Z*  1. The NOT opera

tion is also referred to as the *complement* operation, since it changes a 1 to 0

and a 0 to 1.

Binary logic resembles binary arithmetic, and the operations AND and OR

have similarities to multiplication and addition, respectively. This is why the sym bols used for AND and OR are the same as those used for multiplication and addi tion. However, binary logic should not be confused with binary arithmetic. One should realize that an arithmetic variable designates a number that may consist of

38

**COMBINATIONAL LOGIC CIRCUITS**

many digits, whereas a logic variable is always either a 1 or a 0. The following equa tions define the logical OR operation:

0 0 0

0 1 1

1 0 1

1 1 1

These resemble binary addition, except for the last operation. In binary logic, we have 1 1 1 (read “one OR one is equal to one”), but in binary arithmetic, we have 1 1 10 (read “one plus one is equal to two”). To avoid ambiguity, the ∨

symbol is sometimes used for the OR operation instead of the symbol. But as long as arithmetic and logic operations are not mixed, each can use the symbol with its own independent meaning.

The next equations define the logical AND operation:

0 · 0 0

0 · 1 0

1 · 0 0

1 · 1 1

This operation is identical to binary multiplication, provided that we use only a sin ∧ ∨

gle bit. Alternative symbols to the for AND and + for OR, are symbols and , respectively, that represent conjunctive and disjunctive operations in propositional calculus.

For each combination of the values of binary variables such as *X* and *Y*,

there is a value of *Z* specified by the definition of the logical operation. The defini tions may be listed in compact form in a truth table. A *truth table* for an operation is a table of combinations of the binary variables showing the relationship between the values that the variables take on and the values of the result of the operation. The truth tables for the operations AND, OR, and NOT are shown in Table 1. The tables list all possible combinations of values for two variables and the results of the operation. They clearly demonstrate the definition of the three operations.

**TABLE 1**

**Truth Tables for the Three Basic Logical Operations**

**AND OR NOT**

**Z** = **X Y** **Z** = **X** + **Y Z** = **X**

**X Y X Y X**

00 0 00 0 0 1

01 0 01 1 1 0

10 0 10 1

11 1 11 1

39

**COMBINATIONAL LOGIC CIRCUITS**

**Logic Gates**

Logic gates are electronic circuits that operate on one or more input signals to pro duce an output signal. Electrical signals such as voltages or currents exist through out a digital system in either of two recognizable values. Voltage-operated circuits respond to two separate voltage ranges that represent a binary variable equal to logic 1 or logic 0. The input terminals of logic gates accept binary signals within the allowable range and respond at the output terminals with binary signals that fall within a specified range. The intermediate regions between the allowed ranges in the figure are crossed only during changes from 1 to 0 or from 0 to 1. These changes are called *transitions*, and the intermediate regions are called the *transition regions*.

The graphics symbols used to designate the three types of gates—AND, OR, and NOT—are shown in Figure 1(a). The gates are electronic circuits that produce the equivalents of logic-1 and logic-0 output signals in accordance with their respective truth tables if the equivalents of logic-1 and logic-0 input signals are applied. The two input signals *X* and *Y* to the AND and OR gates take on one of four possible combinations: 00, 01, 10, or 11. These input signals are shown as tim ing diagrams in Figure 1(b), together with the timing diagrams for the corre sponding output signal for each type of gate. The horizontal axis of a *timing diagram* represents time, and the vertical axis shows a signal as it changes between

X

X

Z X YZ X YZ X X Y

Y

OR gate

(a) Graphic symbols

X 0 011

Y0 1 0 1

(AND) 0 0 0 1 X Y

(OR) X Y 0 1 1 1

(NOT) X 1 100

(b) Timing diagram

tG

(AND) 0 0 0 1 X Y

(c) AND timing diagram with gate delay tG

**FIGURE 1**

Digital Logic Gates

40

AND gate

NOT gate or inverter

**COMBINATIONAL LOGIC CIRCUITS**

the two possible voltage levels. The low level represents logic 0 and the high level represents logic 1. The AND gate responds with a logic-1 output signal when both input signals are logic 1. The OR gate responds with a logic-1 output signal if either input signal is logic 1. The NOT gate is more commonly referred to as an *inverter*. The reason for this name is apparent from the response in the timing diagram. The output logic signal is an inverted version of input logic signal *X*.

In addition to its function, each gate has another very important property called *gate delay*, the length of time it takes for an input change to result in the corresponding output change. Depending on the technology used to implement the gate, the length of time may depend on which of the inputs are changing. For example, for the AND gate shown in Figure 1(a), with both inputs equal to 1, the gate delay when input B changes to 0 may be longer than the gate delay when the input A changes to 0. Also, the gate delay when the output is changing from 0 to 1 may be longer than when the output is changing from 1 to 0, or vice versa. In the simplified model introduced here, these variations are ignored and the gate delay is assumed to have a single value, *tG*. This value may be different for each gate type, number of inputs, and the underlying technology and circuit design of the gate. In Figure 1(c), the output of the AND gate is shown taking into consid eration the AND gate delay, *tG*. A change in the output waveform is shifted *tG* time units later compared to the change in input *X* or *Y* that causes the it. When gates are attached together to form logic circuits, the delays down each path from an input to an output add together.

AND and OR gates may have more than two inputs. An AND gate with three inputs and an OR gate with six inputs are shown in Figure 2. The three input AND gate responds with a logic-l output if all three inputs are logic l. The output is logic 0 if any input is logic 0. The six-input OR gate responds with a logic 1 if any input is logic 1; its output becomes a logic 0 only when all inputs are logic 0.

**2 BOOLEAN ALGEBRA**

The Boolean algebra we present is an algebra dealing with binary variables and logic operations. The variables are designated by letters of the alphabet, and the three basic logic operations are AND, OR, and NOT (complementation). A *Bool ean expression* is an algebraic expression formed by using binary variables, the con stants 0 and 1, the logic operation symbols, and parentheses. A *Boolean function* can be described by a Boolean equation consisting of a binary variable identifying

ABCF ABC

ABCG A B C D E F DEF

(a) Three-input AND gate (b) Six-input OR gate

**FIGURE 2**

Gates with More than Two Inputs

41

**COMBINATIONAL LOGIC CIRCUITS**

the function followed by an equals sign and a Boolean expression. Optionally, the function identifier is followed by parentheses enclosing a list of the function vari ables separated by commas. A *single-output Boolean function* is a mapping from each of the possible combinations of values 0 and 1 on the function variables to value 0 or 1. A *multiple-output Boolean function* is a mapping from each of the possible combinations of values 0 and 1 on the function variables to combinations of 0 and 1 on the function outputs.

**EXAMPLE 1 Boolean Function Example - Power Windows**

Consider an example Boolean equation representing electrical or electronic logic for control of the lowering of the driver’s power window in a car.

*LDXA* ( ) , , *DX A*

The window is raised or lowered by a motor driving a lever mechanism connected to the window. The function *L* = 1 means that the window motor is powered up to turn in the direction that lowers the window. *L* = 0 means the window motor is not powered up to turn in this direction. *D* is an output produced by pushing a panel switch on the inside of the driver’s door. With *D =* 1, the lowering of the driver’s window is requested, and with *D* = 0, this action is not requested. *X* is the output of a mechanical limit switch. *X* = 1 if the window is at a limit—in this case, in the fully down position. *X* = 0 if the window is not at its limit—i.e., not in the fully down position. *A* = 1 indicates automatic lowering of the window until it is in the fully down position. *A* is a signal generated by timing logic from *D* and *X*. Whenever *D* has been 1 for at least one-half second, *A* becomes 1 and remains at 1 until *X* = 1. If *D =* 1 for less than one-half second, *A* = 0. Thus, if the driver requests that the window be lowered for one-half second or longer, the window is to be lowered automatically to the fully down position.

The two parts of the expression, *DX* and *A*, are called *terms* of the expression

for *L*. The function *L* is equal to 1 if term *DX* is equal to 1 or if term *A* is equal to 1. Otherwise, *L* is equal to 0. The complement operation dictates that if *X* = 1, then *X* = 0. Therefore, we can say that *L*  1 if *D*  1, and *X*  0 or if *A*  1. So what does the equation for *L* say if interpreted in words? It says that the window will be lowered if the window is not fully lowered (*X* = 0) and the switch *D* is being pushed (*D* = 1) or if the window is to be lowered automatically to fully down posi tion (*A* = 1). ■

A Boolean equation expresses the logical relationship between binary vari

ables. It is evaluated by determining the binary value of the expression for all pos sible combinations of values for the variables. A Boolean function can be represented by a truth table. A *truth table* for a function is a list of all combinations of 1s and 0s that can be assigned to the binary variables and a list that shows the value of the function for each binary combination. The truth tables for the logic operations given in Table 1 are special cases of truth tables for functions. The

42

**COMBINATIONAL LOGIC CIRCUITS**

**TABLE 2**

**Truth Table**

**for the Function L**  **DX**  **A DXA L**

0 0 0 0 1 1 1 1

0 0 1 1 0 0 1 1

0 1 0 1 0 1 0 1

0 1 0 1 1 1 0 1

number of rows in a truth table is 2*n*, where *n* is the number of variables in the function. The binary combinations for the truth table are the *n*-bit binary numbers that correspond to counting in decimal from 0 through 2*n*  1. Table 2 shows the truth table for the function *L*  *DX*  *A*. There are eight possible binary combina tions that assign bits to the three variables *D*, *X*, and *A*. The column labeled *L* con tains either 0 or 1 for each of these combinations. The table shows that the function *L* is equal to 1 if *D*  1 and *X*  0 or if *A*  1. Otherwise, the function *L* is equal to 0.

An algebraic expression for a Boolean function can be transformed into a cir cuit diagram composed of logic gates that implements the function. The logic cir cuit diagram for function *L* is shown in Figure 3. An inverter on input *X* generates the complement, *X*. An AND gate operates on *X* and *D*, and an OR gate combines *DX* and *A*. In logic circuit diagrams, the variables of the function *F* are taken as the inputs of the circuit, and the binary variable *F* is taken as the output of the circuit. If the circuit has a single output, *F* is a single output function. If the circuit has mul tiple outputs, function *F* is a multiple output function with multiple variables and equations required to represent its outputs. Circuit gates are interconnected by wires that carry logic signals. Logic circuits of this type are called *combinational logic circuits*, since the variables are “combined” by the logical operations. This is in contrast to sequential logic, in which variables are stored over time as well as being combined.

There is only one way that a Boolean function can be represented in a truth table. However, when the function is in algebraic equation form, it can be

D

L

X

A

**FIGURE 3**

Logic Circuit Diagram for *L*  *DX* *A*

43

**COMBINATIONAL LOGIC CIRCUITS**

expressed in a variety of ways. The particular expression used to represent the function dictates the interconnection of gates in the logic circuit diagram. By manipulating a Boolean expression according to Boolean algebraic rules, it is often possible to obtain a simpler expression for the same function. This simpler expres sion reduces both the number of gates in the circuit and the numbers of inputs to the gates. To see how this is done, we must first study the basic rules of Boolean algebra.

**Basic Identities of Boolean Algebra**

Table 3 lists the most basic identities of Boolean algebra. The notation is simplified by omitting the symbol for AND whenever doing so does not lead to confusion. The first nine identities show the relationship between a single variable *X*, its com plement *X*, and the binary constants 0 and 1. The next five identities, 10 through 14, have counterparts in ordinary algebra. The last three, 15 through 17, do not apply in ordinary algebra, but are useful in manipulating Boolean expressions.

The basic rules listed in the table have been arranged into two columns that demonstrate the property of duality of Boolean algebra. The *dual* of an algebraic expression is obtained by interchanging OR and AND operations and replacing 1s by 0s and 0s by 1s. An equation in one column of the table can be obtained from the corresponding equation in the other column by taking the dual of the expres sions on both sides of the equals sign. For example, relation 2 is the dual of relation 1 because the OR has been replaced by an AND and the 0 by 1. It is important to note that most of the time the dual of an expression is not equal to the original expression, so that an expression usually cannot be replaced by its dual.

The nine identities involving a single variable can be easily verified by sub stituting each of the two possible values for *X*. For example, to show that *X*  0 *X*, let *X*  0 to obtain 0 0 0, and then let *X*  1 to obtain 1 0 1. Both

**TABLE 3**

**Basic Identities of Boolean Algebra**

*X*  0 *X X* 1 *X*

1. 2.

3. 4.

*X*  1 1 *X* 0 0

5. 6.

*X X*  *X X* *X*  *X*

7. 8.

*X*  *X*  1 *X* *X*  0

9.

*X*  *X*

10. 11. Commutative

*X*  *Y Y X*  *XY YX*

12. 13. Associative

*X*  ( ) *Y Z*  ( ) *X Y*  *Z X*( ) *YZ*  ( ) *XY Z*

*XY Z* ( ) *XY XZ*  *X YZ*  ( ) *X Y*  ( ) *X Z*

14. 15. Distributive 16. 17. DeMorgan’s

*X Y*  *X Y* *X Y* *X Y*

44

**COMBINATIONAL LOGIC CIRCUITS**

equations are true according to the definition of the OR logic operation. Any expression can be substituted for the variable *X* in all the Boolean equations listed in the table. Thus, by identity 3 and with *X*  *AB*  *C*, we obtain

*AB C*  1 1

Note that identity 9 states that double complementation restores the variable to its original value. Thus, if *X*  0, then 1 and 0 *X*.

*X X*

Identities 10 and 11, the commutative laws, state that the order in which the variables are written will not affect the result when using the OR and AND opera tions. Identities 12 and 13, the associative laws, state that the result of applying an operation over three variables is independent of the order that is taken, and there fore, the parentheses can be removed altogether, as follows:

*X YZ*  ( ) ( ) *X Y*  *Z*  *XYZ*

*X YZ* ( ) ( ) *XY Z XYZ*

These two laws and the first distributive law, identity 14, are well known from ordi nary algebra, so they should not pose any difficulty. The second distributive law, given by identity 15, is the dual of the ordinary distributive law and does not hold in ordinary algebra. As illustrated previously, each variable in an identity can be replaced by a Boolean expression, and the identity still holds. Thus, consider the expression (*A*  *B*) (*A*  *CD*). Letting *X*  *A*, *Y*  *B*, and *Z*  *CD*, and applying the second distributive law, we obtain

( ) *A B*  ( ) *A CD*  *A BCD*

The last two identities in Table 3,

and

*X Y*  *X Y* *X Y* *X*  *Y*

are referred to as DeMorgan’s theorem. This is a very important theorem and is used to obtain the complement of an expression and of the corresponding function. DeMorgan’s theorem can be illustrated by means of truth tables that assign all the possible binary values to *X* and *Y*. Table 4 shows two truth tables that verify the *X Y*

first part of DeMorgan’s theorem. In (a), we evaluate for all possible values of *X* and *Y*. This is done by first evaluating *X*  *Y* and then taking its complement. *X Y*

In (b), we evaluate and and then AND them together. The result is the same

**TABLE 4**

**Truth Tables to Verify DeMorgan’s Theorem**

**(a) X Y X**  **Y (b) X Y**

**X Y**  **X Y *X*** **Y**

0 0 1 1

0 1 0 1

0 1 1 1

1 0 0 0

0 0 1 1

0 1 0 1

1 1 0 0

1 0 1 0

1 0 0 0

45

**COMBINATIONAL LOGIC CIRCUITS**

for the four binary combinations of *X* and *Y*, which verifies the identity of the equation.

Note the order in which the operations are performed when evaluating an expression. In part (b) of the table, the complement over a single variable is evalu ated first, followed by the AND operation, just as in ordinary algebra with multipli cation and addition. In part (a), the OR operation is evaluated first. Then, noting that the complement over an expression such as *X*  *Y* is considered as specifying NOT (*X*  *Y*), we evaluate the expression within the parentheses and take the complement of the result. It is customary to exclude the parentheses when comple menting an expression, since a bar over the entire expression joins it together. ( ) *X Y*  *X Y*

Thus, is expressed as when designating the complement of *X*  *Y*. DeMorgan’s theorem can be extended to three or more variables. The gen eral DeMorgan’s theorem can be expressed as

*X*1  *X*2 … *Xn*  *X*1*X*2…*Xn*

*X*1*X*2…*Xn*  *X*1  *X*2 … *Xn*

Observe that the logic operation changes from OR to AND or from AND to OR. In addition, the complement is removed from the entire expression and placed instead over each variable. For example,

*ABCD*  *ABCD*

**Algebraic Manipulation**

Boolean algebra is a useful tool for simplifying digital circuits. Consider, for exam ple, the Boolean function represented by

*F XYZ XYZ XZ*

The implementation of this equation with logic gates is shown in Figure 4(a). Input variables *X* and *Z* are complemented with inverters to obtain and . The three

*X Z*

terms in the expression are implemented with three AND gates. The OR gate forms the logical OR of the terms. Now consider a simplification of the expression for *F* by applying some of the identities listed in Table 3:

*F*

*XYZ XYZ XZ*

by identity 14

*XY Z Z* ( ) *XZ*

*XY* 1 *XZ*

by identity 7

by identity 2

*XY XZ*

The expression is reduced to only two terms and can be implemented with

gates as shown in Figure 4(b). It is obvious that the circuit in (b) is simpler than the one in (a), yet, both implement the same function. It is possible to use a truth table to verify that the two implementations are equivalent. This is shown in Table 5. As expressed in Figure 4(a), the function is equal to 1 if *X*  0, *Y*  1, and *Z*  1; if *X*  0, *Y*  1, and *Z*  0; or if *X* and *Z* are both 1. This produces the four 1s

46

**COMBINATIONAL LOGIC CIRCUITS**

X

Y

F

Z

(a) F XYZ XYZ XZ

X

Y

F

Z

(b) F XY XZ

**FIGURE 4**

Implementation of Boolean Function with Gates

**TABLE 5**

**Truth Table for Boolean Function**

**X Y Z (a) F (b) F**

0 0 0 0 1 1 1 1

0 0 1 1 0 0 1 1

0 1 0 1 0 1 0 1

0 0 1 1 0 1 0 1

0 0 1 1 0 1 0 1

for *F* in part (a) of the table. As expressed in Figure 4(b), the function is equal to 1 if *X*  0 and *Y*  1 or if *X*  1 and *Z*  1. This produces the same four 1s in part (b) of the table. Since both expressions produce the same truth table, they are equivalent. Therefore, the two circuits have the same output for all possible binary combinations of the three input variables. Each circuit implements the same func tion, but the one with fewer gates and/or fewer gate inputs is preferable because it requires fewer components.

When a Boolean equation is implemented with logic gates, each term

requires a gate, and each variable within the term designates an input to the gate. We define a *literal* as a single variable within a term that may or may not be com plemented. The expression for the function in Figure 4(a) has three terms and eight literals; the one in Figure 4(b) has two terms and four literals. By

47

**COMBINATIONAL LOGIC CIRCUITS**

reducing the number of terms, the number of literals, or both in a Boolean expression, it is often possible to obtain a simpler circuit. Boolean algebra is applied to reduce an expression for the purpose of obtaining a simpler circuit. For highly complex functions, finding the best expression based on counts of terms and literals is very difficult, even by the use of computer programs. Certain methods, however, for reducing expressions are often included in computer tools for synthesizing logic circuits. These methods can obtain good, if not the best, solutions. The only manual method for the general case is a cut-and-try proce dure employing the basic relations and other manipulations that become familiar with use. The following examples use identities from Table 3 to illustrate a few of the possibilities:

**1.**

*X XY*  *X* 1 *XY*  *X*( ) 1 *Y X* 1 *X* **2.**

*XY XY*  *XY Y* ( ) *X* 1 *X*

*X XY*  ( ) *X X*  ( ) *X Y*  1 ( ) *X Y*  *X Y*  **3.**

*X X* 1 *X* 1 *X*

Note that the intermediate steps and are often omitted 1 *Y*  1

because of their rudimentary nature. The relationship is useful for elim inating redundant terms, as is done with the term *XY* in this same equation. The *Y*  *Y*  1

relation is useful for combining two terms, as is done in equation 2. The two terms being combined must be identical except for one variable, and that vari able must be complemented in one term and not complemented in the other. Equation 3 is simplified by means of the second distributive law (identity 15 in Table 3). The following are three more examples of simplifying Boolean expres sions:

*XX Y* ( ) *X X* *X Y* *X XY*  *X*( ) 1 *Y X* 1 *X*

**4.**

( ) *X Y*  ( ) *X Y*  *X YY*  *X*  0 *X*

**5.**

*XX Y* ( ) *XX XY*  0 *XY XY*

**6.**

The six equalities represented by the initial and final expressions are theorems of Boolean algebra proved by the application of the identities from Table 3. These theorems can be used along with the identities in Table 3 to prove additional results and to assist in performing simplification.

Theorems 4 through 6 are the duals of equations 1 through 3. Remember that

the dual of an expression is obtained by changing AND to OR and OR to AND throughout (and 1s to 0s and 0s to 1s if they appear in the expression). The *duality principle* of Boolean algebra states that a Boolean equation remains valid if we take the dual of the expressions on both sides of the equals sign. Therefore, equa tions 4, 5, and 6 can be obtained by taking the dual of equations 1, 2, and 3, respec tively.

Along with the results just given in equations 1 through 6, the following *con*

*sensus theorem* is useful when simplifying Boolean expressions:

*XY XZ YZ*  *XY XZ*

48

**COMBINATIONAL LOGIC CIRCUITS**

The theorem shows that the third term, *YZ*, is redundant and can be eliminated. Note that *Y* and *Z* are associated with *X* and in the first two terms and appear

*X*

together in the term that is eliminated. The proof of the consensus theorem is *X*

obtained by first ANDing *YZ* with (*X*  ) 1 and proceeds as follows:

*XY XZ YZ*  *XY XZ YZ X X*  ( )

The dual of the consensus theorem is

*XY XZ XYZ XYZ*  *XY XYZ XZ XYZ*  *XY*( ) 1 *Z*  *XZ*( ) 1 *Y XY XZ*

( ) *X Y*  ( ) *X Z*  ( ) *Y Z*  ( ) *X Y*  ( ) *X Z*

The following example shows how the consensus theorem can be applied in manipulating a Boolean expression:

( ) *A B*  ( ) *A C*  *AA AC AB BC*

*AA*  0 0 *AC*  *AC*.

*AC AB BC*  *AC AB*

Note that and The redundant term eliminated in the last step by the consensus theorem is *BC*.

**Complement of a Function**

The complement representation for a function *F*, is obtained from an inter

*F*,

change of 1s to 0s and 0s to 1s for the values of *F* in the truth table. The comple ment of a function can be derived algebraically by applying DeMorgan’s theorem. The generalized form of this theorem states that the complement of an expression is obtained by interchanging AND and OR operations and complementing each variable and constant, as shown in Example 2.

**EXAMPLE 2 Complementing Functions**

Find the complement of each of the functions represented by the equations *F*1 *XYZ XYZ*  *F*2  *X YZ YZ* ( ) .

and Applying DeMorgan's theorem as many times as necessary, we obtain the complements as follows:

*F*1 *XYZ XYZ*  ( ) *XYZ*  ( ) *XYZ*

( ) *XYZ*  ( ) *XYZ*

49

**COMBINATIONAL LOGIC CIRCUITS**

*F*2 *X YZ YZ* ( ) *X YZ YZ*  ( )

*X YZ YZ*  ( )

■

*X YZ*  ( ) ( ) *Y Z*

A simpler method for deriving the complement of a function is to take the dual of the function equation and complement each literal. This method follows from the generalization of DeMorgan’s theorem. Remember that the dual of an expression is obtained by interchanging AND and OR operations and 1s and 0s. To avoid confusion in handling complex functions, adding parentheses around terms before taking the dual is helpful, as illustrated in the next example.

**EXAMPLE 3 Complementing Functions by Using Duals**

Find the complements of the functions in Example 2 by taking the duals of their equations and complementing each literal.

We begin with

*F*1 *XYZ XYZ*  ( ) *XYZ*  ( ) *XYZ*

The dual of *F*1 is

( ) *XYZ*  ( ) *XYZ*

Complementing each literal, we have

( ) *XYZ*  ( ) *XYZ*  *F*1

Now,

*F*2  *X YZ YZ* ( ) *X YZ* ( ) ( ) ( ) *YZ*

The dual of *F*2 is

*X YZ*  ( ) ( ) *Y Z*

Complementing each literal yields

■

*X YZ*  ( ) ( ) *Y Z*  *F*2

**3 STANDARD FORMS**

A Boolean function expressed algebraically can be written in a variety of ways. There are, however, specific ways of writing algebraic equations that are consid ered to be standard forms. The standard forms facilitate the simplification proce dures for Boolean expressions and, in some cases, may result in more desirable expressions for implementing logic circuits.

50

**COMBINATIONAL LOGIC CIRCUITS**

The standard forms contain *product terms* and *sum terms*. An example of a product term is *XYZ*. This is a logical product consisting of an AND operation among three literals. An example of a sum term is *X + Y + Z*. This is a logical sum consisting of an OR operation among the literals. In Boolean algebra, the words “product” and “sum” do not imply arithmetic operations; instead, they specify the logical operations AND and OR, respectively.

**Minterms and Maxterms**

A truth table defines a Boolean function. An algebraic expression for the function can be derived from the table by finding a logical sum of product terms for which the function assumes the binary value 1. A product term in which all the variables appear exactly once, either complemented or uncomplemented, is called a *min term*. Its characteristic property is that it represents exactly one combination of binary variable values in the truth table. It has the value 1 for that combination and 0 for all others. There are 2*n* distinct minterms for *n* variables. The four minterms for the two variables *X* and *Y* are *X Y*, *XY*, *XY*, and *XY*. The eight minterms for the three variables *X*, *Y*, and *Z* are listed in Table 6. The binary numbers from 000 to 111 are listed under the variables. For each binary combination, there is a related minterm. Each minterm is a product term of exactly *n* literals, where *n* is the num ber of variables. In this example, *n* = 3. A literal is a complemented variable if the corresponding bit of the related binary combination is 0 and is an uncomple mented variable if it is 1. A symbol *mj* for each minterm is also shown in the table, where the subscript *j* denotes the decimal equivalent of the binary combination corresponding to the minterm. This list of minterms for any given *n* variables can be formed in a similar manner from a list of the binary numbers from 0 through 2*n* – 1. In addition, the truth table for each minterm is given in the right half of the table. These truth tables clearly show that each minterm is 1 for the corresponding binary combination and 0 for all other combinations. Such truth tables will be help ful later in using minterms to form Boolean expressions.

**TABLE 6**

**Minterms for Three Variables**

**Product**

**XYZ**

**Term Symbol m0 m1 m2 m3 m4 m5 m6 m7**

0 0 0 0 1 1 1 1

0 0 1 1 0 0 1 1

0

0

0

0

m0

0

0

0

0

*XYZ*

1

0

0

0

0

0

1

0

1

0

*XYZ*

m1

0

0

0

0

0

0

1

0

0

*XYZ*

m2

0

0

0

0

1

1

0

0

0

*XYZ*

m3

0

0

0

1

0

0

0

0

0

*XYZ*

m4

0

0

1

0

0

1

0

0

0

*XYZ*

m5

0

1

0

0

0

0

0

0

0

*XYZ*

m6

1

0

0

0

0

1

0

0

0

*XYZ*

m7

51

**COMBINATIONAL LOGIC CIRCUITS**

**TABLE 7**

**Maxterms for Three Variables**

**X Y Z Sum Term Symbol M0 M1 M2 M3 M4 M5 M6 M7**

0 0 0 0 1 1 1 1

0 0 1 1 0 0 1 1

0 1 0 1 0 1 0 1

*XYZ*  *XYZ*  *XYZ*  *XYZ*  *XYZ*  *XYZ*  *XYZ*  *XYZ*

M0 M1 M2 M3 M4 M5 M6 M7

0 1 1 1 1 1 1 1

1 0 1 1 1 1 1 1

1 1 0 1 1 1 1 1

1 1 1 0 1 1 1 1

1 1 1 1 0 1 1 1

1 1 1 1 1 0 1 1

1 1 1 1 1 1 0 1

1 1 1 1 1 1 1 0

A sum term that contains all the variables in complemented or uncomple mented form is called a *maxterm*. Again, it is possible to formulate 2*n* maxterms with *n* variables. The eight maxterms for three variables are listed in Table 7. Each maxterm is a logical sum of the three variables, with each variable being comple mented if the corresponding bit of the binary number is 1 and uncomplemented if it is 0. The symbol for a maxterm is *Mj*, where *j* denotes the decimal equivalent of the binary combination corresponding to the maxterm. In the right half of the table, the truth table for each maxterm is given. Note that the value of the max term is 0 for the corresponding combination and 1 for all other combinations. It is now clear where the terms “minterm” and “maxterm” come from: a minterm is a function, not equal to 0, having the minimum number of 1s in its truth table; a maxterm is a function, not equal to 1, having the maximum of 1s in its truth table. Note from Table 6 and Table 7 that a minterm and maxterm with the same sub script are the complements of each other; that is, and . For exam

ple, for *j*  3, we have

*Mj*  *mj mj*  *Mj*

*M*3  *XYZ*  *XYZ m*  3

A Boolean function can be represented algebraically from a given truth table

by forming the logical sum of all the minterms that produce a 1 in the function. This expression is called a *sum of minterms*. Consider the Boolean function *F* in Table 8(a). The function is equal to 1 for each of the following binary combinations of the variables *X*, *Y*, and *Z*: 000, 010, 101 and 111. These combinations correspond to minterms 0, 2, 5, and 7. By examining Table 8 and the truth tables for these min terms in Table 6, it is evident that the function *F* can be expressed algebraically as the logical sum of the stated minterms:

*F XYZ XYZ XYZ XYZ*  *m*0  *m*2 *m*5 *m*7

This can be further abbreviated by listing only the decimal subscripts of the min terms:

*FXYZ* ( ) , , *m*( ) 0257 ,,,

52

**COMBINATIONAL LOGIC CIRCUITS**

**TABLE 8**

**Boolean Functions of Three Variables**

**F**

**(a) X Y Z F (b) X Y Z E**

0 0 0 0 1 1 1 1

0 0 1 1 0 0 1 1

0 1 0 1 0 1 0 1

1 0 1 0 0 1 0 1

0 1 0 1 1 0 1 0

0 0 0 0 1 1 1 1

0 0 1 1 0 0 1 1

0 1 0 1 0 1 0 1

1 1 1 0 1 1 0 0

The symbol stands for the logical sum (Boolean OR) of the minterms. The num

bers following it represent the minterms of the function. The letters in parentheses following *F* form a list of the variables in the order taken when the minterms are converted to product terms.

Now consider the complement of a Boolean function. The binary values of *F* in Table 8(a) are obtained by changing 1s to 0s and 0s to 1s in the values of *F*. Taking the logical sum of minterms of *F*, we obtain

*FXYZ* ( ) , , *XYZ*  *XYZ XYZ XYZ*  *m*1  *m*3 *m*4 *m*6 or, in abbreviated form,

*FXYZ* ( ) , , *m*( ) 1346 ,,,

Note that the minterm numbers for *F* are the ones missing from the list of the min term numbers of *F*. We now take the complement of *F* to obtain *F*:

*F*

*m*1  *m*3 *m*4 *m*6 *m*1 *m*3 *m*4 *m*6

(since ) *M*1 *M*3 *M*4 *M*6  *mj*  *Mj*

( ) *XYZ*  ( ) *XYZ*  ( ) *XYZ*  ( ) *XYZ*

This shows the procedure for expressing a Boolean function as a *product of max terms*. The abbreviated form for this product is

*FXYZ* ( ) , , *M*( ) 1346 ,,,

where the symbol denotes the logical product (Boolean AND) of the maxterms whose numbers are listed in parentheses. Note that the decimal numbers included in the product of maxterms will always be the same as the minterm list of the com plemented function, such as (1, 3, 4, 6) in the foregoing example. Maxterms are sel dom used directly when dealing with Boolean functions, since we can always replace them with the minterm list of *F*.

53

**COMBINATIONAL LOGIC CIRCUITS**

The following is a summary of the most important properties of minterms:

**1.** There are 2*n* minterms for *n* Boolean variables. These minterms can be gener ated from the binary numbers from 0 to 2*n*  1.

**2.** Any Boolean function can be expressed as a logical sum of minterms.

**3.** The complement of a function contains those minterms not included in the original function.

**4.** A function that includes all the 2*n* minterms is equal to logic 1.

A function that is not in the sum-of-minterms form can be converted to that form by means of a truth table, since the truth table always specifies the minterms of the function. Consider, for example, the Boolean function

*E Y XZ*

The expression is not in sum-of-minterms form, because each term does not con tain all three variables *X*, *Y*, and *Z*. The truth table for this function is listed in Table 8(b). From the table, we obtain the minterms of the function:

*EXYZ* ( ) , , *m*( ) 01245 ,,,,

The minterms for the complement of E are given by

*EXYZ* ( ) , , *m*( ) 367 , ,

*E*

Note that the total number of minterms in *E* and is equal to eight, since the func tion has three variables, and three variables produce a total of eight minterms. With four variables, there will be a total of 16 minterms, and for two variables, there will be four minterms. An example of a function that includes all the minterms is

*GXY* ( ) , *m*( ) 0123 ,,, 1

Since *G* is a function of two variables and contains all four minterms, it is always equal to logic 1.

**Sum of Products**

The sum-of-minterms form is a standard algebraic expression that is obtained directly from a truth table. The expression so obtained contains the maximum number of literals in each term and usually has more product terms than necessary. This is because, by definition, each minterm must include all the variables of the function, complemented or uncomplemented. Once the sum of minterms is obtained from the truth table, the next step is to try to simplify the expression to see whether it is possible to reduce the number of product terms and the number of literals in the terms. The result is a simplified expression in *sum-of-products* form. This is an alternative standard form of expression that contains product terms with up to *n* literals. An example of a Boolean function expressed as a sum of products is

54

**COMBINATIONAL LOGIC CIRCUITS** Y

XY

F

Z

X

Y

**FIGURE 5**

Sum-of-Products Implementation *F Y XYZ XY*

The expression has three product terms, the first with one literal, the second with three literals, and the third with two literals.

The logic diagram for a sum-of-products form consists of a group of AND

gates followed by a single OR gate, as shown in Figure 5. Each product term requires an AND gate, except for a term with a single literal. The logical sum is formed with an OR gate that has single literals and the outputs of the AND gates as inputs. Often, we assumed that the input variables are directly available in their complemented and uncomplemented forms, so inverters are not included in the diagram. The AND gates followed by the OR gate form a circuit configuration referred to as a *two-level implementation* or *two-level circuit*.

If an expression is not in sum-of-products-form, it can be converted to the

standard form by means of the distributive laws. Consider the expression

*F AB C D E*  ( )

This is not in sum-of-products form, because the term D E is part of a product, not a single literal. The expression can be converted to a sum of products by apply ing the appropriate distributive law as follows:

*F AB C D E*  ( ) *AB CD CE*

The function *F* is implemented in a nonstandard form in Figure 6(a). This requires two AND gates and two OR gates. There are three levels of gating in the circuit. *F* is implemented in sum-of-products form in Figure 6(b). This circuit requires three AND gates and an OR gate and uses two levels of gating. The decision as to whether to use a two-level or multiple-level (three levels or more) implementation is complex. Among the issues involved are the number of gates, number of gate inputs, and the amount of delay between the time the input values are set and the time the resulting output values appear. Two-level implementations are the natural form for certain implementation technologies.

55

**COMBINATIONAL LOGIC CIRCUITS**

A

B

A

B

C

D

C

D

C

E

E

(a) AB C(D E) **FIGURE 6**

(b) AB CD CE

Three-Level and Two-Level Implementation

**Product of Sums**

Another standard form of expressing Boolean functions algebraically is the *prod uct of sums*. This form is obtained by forming a logical product of sum terms. Each logical sum term may have any number of distinct literals. An example of a func tion expressed in product-of-sums form is

*F XY Z*  ( ) ( ) *XYZ*

This expression has sum terms of one, two, and three literals. The sum terms per form an OR operation, and the product is an AND operation.

The gate structure of the product-of-sums expression consists of a group of

OR gates for the sum terms (except for a single literal term), followed by an AND gate. This is shown in Figure 7 for the preceding function *F*. As with the sum of products, this standard type of expression results in a two-level gating structure.

**4 TWO-LEVEL CIRCUIT OPTIMIZATION**

The complexity of a logic circuit that implements a Boolean function is directly related to the algebraic expression from which the function is implemented. Although the truth-table representation of a function is unique, when expressed algebraically, the function appears in many different forms. Boolean expressions may be simplified by algebraic manipulation, as discussed in Section 2. How ever, this procedure of simplification is awkward, because it lacks specific rules to

X

Y F

Z

X

Y

Z

**FIGURE 7**

Product-of-Sums Implementation

56

**COMBINATIONAL LOGIC CIRCUITS**

predict each succeeding step in the manipulative process and it is difficult to determine whether the simplest expression has been achieved. By contrast, the map method provides a straightforward procedure for optimizing Boolean func tions of up to four variables. Maps for five and six variables can be drawn as well, but are more cumbersome to use. The map is also known as the *Karnaugh map*, or *K-map*. The map is a diagram made up of squares, with each square representing one row of a truth table, or correspondingly, one minterm of a single output func tion. Since any Boolean function can be expressed as a sum of minterms, it follows that a Boolean function is recognized graphically in the map by those squares for which the function has value 1, or correspondingly, whose minterms are included in the function. From a more complex view, the map presents a visual diagram of all possible ways a function may be expressed in a standard form. Among these ways are the optimum sum-of-products standard forms for the function. The opti mized expressions produced by the map are always in sum-of-products or prod uct-of-sums form. Thus, maps handle optimization for two-level implementations, but do not apply directly to possible simpler implementations for the general case with three or more levels. Initially, this section covers sum-of-products optimiza tion and, later, applies it to performing product-of-sums optimization.

**Cost Criteria**

In the prior section, counting literals and terms was mentioned as a way of measur ing the simplicity of a logic circuit. We introduce two cost criteria to formalize this concept.

The first criterion is *literal cost*, the number of literal appearances in a Bool

ean expression corresponding exactly to the logic diagram. For example, for the circuits in Figure 6, the corresponding Boolean expressions are

*F AB C D E*  ( ) *F AB CD CE*

and

There are five literal appearances in the first equation and six in the second, so the first equation is the simplest in terms of literal cost. Literal cost has the advantage that it is very simple to evaluate by counting literal appearances. It does not, how ever, represent circuit complexity accurately in all cases, even for the comparison of different implementations of the same logic function. The following Boolean equations, both for function *G*, illustrate this situation:

*G ABCD ABCD*  *G AB*  ( ) ( ) *B C*  ( ) *C D* ( ) *D A*

and

The implementations represented by these equations both have a literal cost of eight. But, the first equation has two terms and the second has four. This suggests that the first equation has a lower cost than the second.

To capture the difference illustrated, we define *gate-input cost* as the number

of inputs to the gates in the implementation corresponding exactly to the given equation or equations. This cost can be determined easily from the logic diagram by simply counting the total number of inputs to the gates in the logic diagram. For

57

**COMBINATIONAL LOGIC CIRCUITS**

sum-of-products or product-of-sums equations, it can be found from the equation by finding the sum of

(1) all literal appearances,

(2) the number of terms excluding terms that consist only of a single literal, and, optionally,

(3) the number of distinct complemented single literals.

In (1), all gate inputs from outside the circuit are represented. In (2), all gate inputs within the circuit, except for those to inverters, are represented and in (3), invert ers needed to complement the input variables are counted in the event that com plemented input variables are not provided. For the two preceding equations, excluding the count from (3), the respective gate-input counts are 8 + 2 = 10 and 8 + 4 = 12. Including the count from (3), that of input inverters, the respective counts are 14 and 16. So the first equation for *G* has a lower gate-input cost, even though the literal costs are equal.

Gate-input cost is currently a good measure for contemporary logic imple

mentations, since it is proportional to the number of transistors and wires used in implementing a logic circuit. Representation of gate inputs becomes particularly important in measuring cost for circuits with more than two levels. Typically, as the number of levels increases, literal cost represents a smaller proportion of the actual circuit cost, since more and more gates have no inputs from outside the circuit itself. Later, in Figure 23, we introduce complex gate types for which evaluation of the gate-input cost from an equation is invalid, since the correspondence between the AND, OR and NOT operations in the equation and the gates in the circuit can no longer be established. In such cases, as well as for equation forms more complex than sum-of-products and product-of-sums, the gate-input count must be deter mined directly from the implementation.

Regardless of the cost criteria used, we see later that the simplest expression

is not necessarily unique. It is sometimes possible to find two or more expressions that satisfy the cost criterion applied. In that case, either solution is satisfactory from the cost standpoint.

**Map Structures**

We will consider maps for two, three, and four variables as shown in Figure 8. The number of squares in each map is equal to the number of minterms in the corre sponding function. In our discussion of minterms, we defined a minterm *mi* to go with the row of the truth table with *i* in binary as the variable values. This use of *i* to represent the minterm *mi* is carried over to the cells of the maps, each of which corresponds to a minterm. For two, three, and four variables, there are 4, 8, and 16 squares, respectively. Each of the maps is labeled in two ways: 1) with variables at the upper left for the columns and the rows and with a binary combination of those variables for each column and each row, and 2) with single variable labels at the edges of the map applied by a bracket to single or double rows and columns. Each location of a variable label aligns with the region of the map for which the variable

58

**COMBINATIONAL LOGIC CIRCUITS**

Y

X

0

0 1 0 1

Y

X

0

Y

0 1

X Y X Y

1

2 3 X

1

X Y X Y

YZ

X

(a) (b)

Y

00 01 11 10

13

0

0132

X Z

7 5

2

0

X

457 6 1

Z

X Z

6 4

YZ

(c) (d) Y

WX

00 01

11 10

X Z

00

0

132

0 2

W

01

11

10

X Z

4576 12 13 1415 8 9 10 11 Z

X

8 10 1 3 911

(e) (f)

**FIGURE 8**

Map Structures

has value 1. The region for which the variable has value 0 is implicitly labeled with the complement of the variable. Only one of these two schemes is required to com pletely label a map, but both are shown to allow selection of the one that works best for a given user.

Beginning with the binary combination scheme, we note that the binary

combinations across the top and down the left side of a map take the form of a Gray code. The use of the Gray code is appropriate because it represents the adjacency of binary combinations and of the corresponding minterms that is the foundation of K-maps. Two binary combinations are said to be *adjacent* if they differ in the value of exactly one variable. Two product terms (including minterms) are *adjacent* if they differ in one and only one literal which appears uncomplemented in one and complemented in the other. For example, the combinations (*X*, *Y*, *Z*) = 011 and 010 are adjacent, since they differ only in the

59

**COMBINATIONAL LOGIC CIRCUITS**

value of variable *Z*. Further, the minterms *XYZ* and *XYZ* are adjacent, since they have identical literal appearances except for *Z,* which appears uncomplemented and complemented. The reason for the use of a Gray code on K-maps is that any two squares which share a common edge correspond to a pair of adjacent binary combinations and adjacent minterms. This correspondence can be used to perform simplification of product terms for a given function on a K-map. This simplification is based on the Boolean algebraic theorem:

*AB AB*  *A*

Applying this to the example with *A* = *XY* and *B* = *Z*,

( ) *XY Z XY*  ( )*Z*  *XY*

Looking at the K-map in Figure 8(c), we see that the two corresponding squares are located at (*X*, *Y*, *Z*) = 011 (3) and 010 (2), which are in row 0 and columns 11 and 10, respectively. Note that these two squares are adjacent (share an edge) and can be combined, as indicated by the black rectangle in Figure 8(c). This rectangle on the K-map contains both 0 and 1 for *Z*, and so no longer depends on *Z*, and can be read off as *XY*. This demonstrates that whenever we have two squares sharing edges that are minterms of a function, these squares can be combined to form a product term with one less variable.

For the 3- and 4-variable K-maps, there is one more issue to be addressed

with respect to the adjacency concept. For a 3-variable K-map, suppose we con sider the minterms 0 and 2 in Figure 8(c). These two minterms do not share an edge, and hence do not appear to be adjacent. However, these two minterms are *X Y Z* and *X Y Z*, which by definition are adjacent. In order to recognize this adja cency on the K-map, we need to consider the left and right borders of the map to be a shared edge. Geometrically, this can be accomplished by forming a cylinder from the map so that the squares touching the left and right borders actually have a shared edge! A view of this cylinder appears in Figure 8(d). Here minterms *m*0 and *m*2 share an edge and, from the K-map, are adjacent. Likewise, *m*4 and *m*6 share an edge on the K-map and are adjacent. The two rectangles resulting from these adjacencies are shown in Figure 8(c) and 8(d) in blue.

The 4-variable K-map in Figure 8(e) can likewise be formed into a cylinder.

This demonstrates four adjacencies, *m*0 and *m*2, *m*4 and *m*6, *m*12 and *m*14, and *m*8 and *m*10. The minterms *m*0 and *m*8, *W X Y Z* and *W X Y Z*, are adjacent, suggesting that the top border of the map should be a shared edge with the bottom border. This can be accomplished by taking the cylinder formed from the map and bending it, joining these two borders. This results in the torus (doughnut shape) in Figure 8(f). The additional resulting adjacencies identifiable on the map are *m*1 and *m*9, *m*3 and *m*11, and *m*2 and *m*10.

Unfortunately, the cylinder and the torus are not convenient to use, but they

can help us remember the locations of shared edges. These edges are at the left and right border pair for the flat 3-variable map and at the left and right border pair and the top and bottom border pair for 4-variable K-maps, respectively. The use of

60

**COMBINATIONAL LOGIC CIRCUITS**

flat maps will require the use of pairs of split rectangles lying across the border pairs.

One final detail is the placing of a given function *F* on a map. Suppose that the function *F* is given as a truth table with the row designated by decimal *i* corre sponding to the binary input values equivalent to *i*. Based on the binary combina tions on the left and top edges of the K-map combined in order, we can designate each cell of the map by the same *i*. This will permit easy transfer of the 0 and 1 val ues of *F* from the truth table onto the K-map. The values of *i* for this purpose are shown on the three maps in Figure 8. It is a good idea to determine how to fill in the values of *i* quickly by noting the order of the values of *i* in a row depends on the Gray code value order for the columns and the ordering of the rows of *i* values depends on the Gray code value order for the rows. For example, for the 4-variable map, the rows-of-columns order of the *i* values is: 0, 1, 3, 2, 4, 5, 7, 6, 12, 13, 15, 14, 8, 9, 11, 10. The rows-of-columns order of the *i* values for 2-variable and 3-variable maps are the first four values and the first eight values from this sequence. These values can also be used for sum of minterm expressions defined using the abbrevi

ated notation. Note that the positioning of the *i* values is dependent upon the placement of the variables in order from lower left side to middle right side to right top and middle bottom for a 4-variable map. For 2- and 3-variable maps, the order is the same with the nonexistent “middle” positions skipped. Any variation from this ordering will give a different map structure.

**Two-Variable Maps**

There are four basic steps for using a K-map. Initially, we present each of these steps using a 2-variable function *F*(*A*, *B*) as an example.

The first step is to enter the function on the K-map. The function may be in the form of a truth table, the shorthand notation for a sum of minterms, or a

*m*

sum-of-products expression. The truth table for *F*(*A*, *B*) is given in Table 9. For

**TABLE 9**

**Two-Variable Function F(A, B)**

**ABF**

0 0 1 1

0 1 0 1

1 1 0 1

each row in which the function *F* has value 1, the values of *A* and *B* can be read to determine where to place a 1 on the map. For example, the function has value 1 for the combination *A* = 0 and *B* = 0. Thus, a 1 is placed in the upper left square of the K-map in Figure 9(a) corresponding to *A* = 0 and *B* = 0. This operation is

61

**COMBINATIONAL LOGIC CIRCUITS**

B

A

0

A

B

0 1

0 1 11

2 3

B

A

0

A

B

0 1

0 1 1

2 3

1 1 1

1 1 11

(a) (b)

**FIGURE 9**

Two-Variable K-map Examples

repeated for rows (0, 1) and (1, 1) in the truth table to complete the entry of *F* in the map.

If the decimal subscripts for the minterms have been added to the truth table

and entered on the map as discussed previously, a much faster approach to enter ing the function on the map is available. The subscripts for the minterms of the function are those corresponding to the rows for which the function is a 1. So a 1 is simply entered in squares 0, 1, and 3 of the K-map. For these two entry methods, as well as others, we assume that each remaining square contains a 0, but do not actu ally enter 0s in the K-map.

*m FAB* ( ) , *m*( ) 013 , ,

The notation for *F* in the truth table is , which can

be entered on the K-map simply by placing 1 in each of the squares 0, 1, and 3. Alternatively, a sum-of-products expression such as *F* = *A* + *AB* can be given as a specification. This can be converted to minterms and entered on the K-map. More simply, the region of the K-map corresponding to each of the product terms can be identified and filled with 1s. Since *AB* is a minterm, we can simply place a 1 in square 3. For *A*, we note that the region is that identified as “not” *A* on the K-map and consists of squares 0 and 1. So *A* can be entered by placing a 1 in each of these two squares. In general, this last process becomes easier once we have mastered the concept of rectangles on a K-map, as discussed next.

The second step is to identify collections of squares on the map representing

product terms to be considered for the simplified expression. We call such objects *rectangles*, since their shape is that of a rectangle (including, of course, a square). Rectangles that correspond to product terms are restricted to contain numbers of squares that are powers of 2, such as 1, 2, 4, and 8. Also, this implies that the length of a side of any rectangle is a power of 2. Our goal is to find the fewest such rectan gles that include or cover all of the squares marked with 1s. This will give the few est product terms and the least input cost for summing the product terms. Any rectangle we are planning to use should be as large as possible in order to include as many 1s as possible. Also, a larger rectangle gives a lower input cost for the cor responding product term.

For the example, there are two largest rectangles. One consists of squares 1

and 0, the other of squares 3 and 1. Squares 1 and 0 correspond to minterms *AB* and *A B*, which can be combined to form rectangle *A*. Squares 3 and 1 correspond to minterms *AB* and *AB*, which can be combined to form rectangle *B*.

62

**COMBINATIONAL LOGIC CIRCUITS**

The third step is to determine if any of the rectangles we have generated is not needed to cover all of the 1s on the K-map. In example, we can see that rectan gle *A* is required to cover minterm 0 and rectangle *B* is required to cover minterm 3. In general, a rectangle is not required if it can be deleted and all of the 1s on the map are covered by the remaining rectangles. If there are choices as to which rect angle of two having unequal size to remove, the largest one should remain.

The final step is to read off the sum-of-products expression, determining the corresponding product terms for the required rectangles in the map. In the exam ple, we can read off the corresponding product terms by using the rectangles shown and the variable labels on the map boundary as *A* and *B*, respectively. This gives a sum-of-products expression for *F* as:

*F AB*

**EXAMPLE 4 Another 2-Variable Map Example**

The function *G*(*A*, *B*) = is shown on the 2-variable K-map in Figure 9(b).

*m*( ) 1 2,

Looking at the map, we find the two rectangles are simply the minterms 1 and 2. From the map,

*GAB* ( ) , *AB AB*

■

From Figure 9(a) and 9(b), we find that 2-variable maps contain: (1) 1 ⋅ 1

rectangles which correspond to minterms and (2) 2 ⋅ 1 rectangles consisting of a pair of adjacent minterms. A 1 ⋅ 1 rectangle can appear on any square of the map and a 2 ⋅ 1 rectangle can appear either horizontally or vertically on the map, each in one of two positions. Note that a 2 ⋅ 2 rectangle covers the entire map and cor responds to the function *F* = 1.

**Three-Variable Maps**

We introduce simplification on 3-variable maps by using two examples followed by a discussion of the new concepts involved beyond those required for 2-variable maps.

**EXAMPLE 5 Three-Variable Map Simplification 1**

Simplify the Boolean function

*FABC* ( ) , , *m*( ) 012345 ,,,,,

This function has been entered on the K-map shown in Figure 10(a), where squares 0 through 5 are marked with 1s. In the map, the two largest rectangles each enclose four squares containing 1s. Note that two squares, 0 and 1, lie in both of the rectangles. Since these two rectangles include all of the 1s in the map and neither

63

**COMBINATIONAL LOGIC CIRCUITS**

BC

A

B

00 01 11 10 1 0

BC

A

B

00 01 11 10

BC

A

B

00 01 11 10

231

1 0

231

1 0

1

11 0

1

1

0

0

231

1

A 1

4 5 1

C

7 6

1

A 1

4 5 1

C

7 6 1

1

A 1

4 5 1

C

7 6 1

(a) (b) (c)

**FIGURE 10**

Three-Variable K-Maps for Example 5 through 7

can be removed, the logical sum of the corresponding two product terms gives the optimized expression for *F*:

*F AB*

To illustrate algebraically how a 4 ⋅ 4 rectangle such as *B* arises, consider the two adjacent black rectangles *AB* and *A B* connected by two pairs of adjacent min terms. These can be combined based on the theorem *XY + XY*= *X* with *X* = *B* and *Y* = *A* to obtain *B*. ■

**EXAMPLE 6 Three-Variable Map Simplification 2**

Simplify the Boolean function

*GABC* ( ) , , *m*( ) 02456 ,,,,

This function has been enter on the K-map shown in Figure 10(b), where squares listed are marked with 1s. In some cases, two squares in the map are adjacent and form a rectangle of size two, even though they do not touch each other. For exam ple, in Figure 10(b) and 8(d), *m*0 is adjacent to *m*2 because the minterms differ by one variable. This can be readily verified algebraically:

*m*0  *m*2  *ABC ABC*  *AC B B* ( ) *A C*

This rectangle is represented in black in Figure 10(b) and in blue in Figure 8(d) on a cylinder where the adjacency relationship is apparent. Likewise, a rectangle is shown in both figures for squares 4 and 6 which corresponds to *AC*. From the prior example, it is apparent that these two rectangles can be combined to give a larger rectangle *C* which covers squares 0, 2, 4, and 6. An additional rectangle is required to cover square 5. The largest such rectangle covers squares 4 and 5. It can be read from the K-map as *AB*. The resulting simplified function is

*GAB* ( ) , *AB C*

■

From Figure 10(a) and 10(b), we find that 3-variable maps can contain all

of the rectangles contained in a 2-variable map plus: (1) 2 ⋅ 2 rectangles, (2) 1 ⋅ 64

**COMBINATIONAL LOGIC CIRCUITS**

4 rectangles, (3) 2 ⋅ 1 “split rectangles” at the left and right edges, and a 2 ⋅ 2 split rectangle at the left and right edges. Note that a 2 ⋅ 4 rectangle covers the entire map and corresponds to the function *G* = 1.

**EXAMPLE 7 Three-Variable Map Simplification 3**

Simplify the Boolean function

*HABC* ( ) , , *m*( ) 13456 ,,,,

This function has been entered on the K-map shown in Figure 10(c), where squares listed are marked with 1s. In this example, we intentionally set the goal of finding all of the largest rectangles in order to emphasize step 3 of simplification, which has not been a significant step in earlier examples. Progressing from the upper center, we find the rectangles corresponding to the following pairs of squares: (3, 1), (1, 5), (5, 4), (4, 6). Can any of these rectangles be removed and still have all squares cov ered? Since only (3, 1) covers 3, it cannot be removed. The same holds for (4, 6) which covers square 6. After these are included, the only square that remains uncovered is 5, which permits either (1, 5) or (5, 4), but not both, to be removed. Assuming that (5, 4) remains, the result can be read from the map as

■

*HABC* ( ) , , *AC AB AC*

**EXAMPLE 8 Four-Variable Map Simplification 1**

Simplify the Boolean function

*FABCD* ( ) ,,, *m*( ) 0 1 2 4 5 6 8 9 10 12 13 ,,,,,,,, , ,

The minterms of the function are marked with 1s in the K-map shown in Figure 11. Eight squares in the two left columns are combined to form a rectangle for the one literal term, *C*. The remaining three 1s cannot be combined to give a single simplified product term; rather, they must be combined as two split 2 ⋅ 2 rectangles. The top

CD

AB

00

00 01 0

1 1

C

11 10

1 3 2 1

4 5 7 6

01

1 1

1

12 13 14

B

11

1 1

15

A

8 9 11 10

10

1 1

D

1

**FIGURE 11**

Four-Variable K-map for Example 8

65

**COMBINATIONAL LOGIC CIRCUITS**

two 1s on the right are combined with the top two 1s on the left to give the term *A D*. Note again that it is permissible to use the same square more than once. We are now left with a square marked with a 1 in the fourth row and fourth column (minterm 1010). Instead of taking this square alone, which will give a term with four literals, we combine it with squares already used to form a rectangle of four squares on the four corners, giving the term *B D*. This rectangle is represented in Figure 11 and in Figure 8(e) on a torus, where the adjacency relationships between the four squares is apparent. The optimized expression is the logical sum of the three terms:

■

*F C AD BD*

**EXAMPLE 9 Four-Variable Map Simplification 2**

Simplify the Boolean function

*GABCD* ( ) ,,, *ACD AD BC CD*  *ABD*

This function has four variables: *A*, *B*, *C*, and *D*. It is expressed in a fairly complex sum-of-products form. In order to enter *G* on a K-map, we will actually enter the regions corresponding to the product terms onto the map, fill the regions with 1s, and then copy the 1s onto a new map for solution. The area in the map covered by the function is shown in Figure 12(a). *A C D* places 1s on squares 0 and 4. *AD* adds 1s to squares 1, 3, 5, and 7. *BC* adds new 1s to squares 2, 10, and 11. *CD* adds a new 1 to square 15 and *A B D* adds the final 1 to square 8. The resulting function

*GABCD* ( ) ,,, *m*( ) 0 1 2 3 4 5 7 8 10 11 15 ,,,,,,,, , ,

is placed on the map in Figure 12(b). It is a good idea to check to see if the four corner rectangle *B D* is present and required. It is present, is required to cover

CD

AB

00 01 0

C

11 10

1 3 2

CD

AB

00 01 0

C

11 10

1 3 2

00

1

1

1

1

00

1

1 1 1

4 5 7 6

4 5 7 6

01

11

1

01

1

1

1

12 13 14

B

12 13 14

B

11

1

15

11

1

15

A

10

8 9 11 10 1 1

1

D

A

10

8 9 11 10 1 1

1

D

(a) K-map for original function G (b) K-map for simplified function G

**FIGURE 12**

Four-Variable K-map for Example 9

66

**COMBINATIONAL LOGIC CIRCUITS**

square 8, and also covers squares 0, 2, and 10. With these squares covered, it is easy to see that just two rectangles, *A C* and *CD*, cover all of the remaining uncovered squares. We can read off the resulting function as:

*G BD AC CD*

Note that this function is much simpler than the original sum-of-products given. ■

**5 MAP MANIPULATION**

When combining squares in a map, it is necessary to ensure that all the minterms of the function are included. At the same time, we need to minimize the number of terms in the optimized function by avoiding any redundant terms whose minterms are already included in other terms. In this section, we consider a procedure that assists in the recognition of useful patterns in the map. Other topics to be covered are the optimization of products of sums and the optimization of incompletely specified functions.

**Essential Prime Implicants**

The procedure for combining squares in a map may be made more systematic if we introduce the terms “implicant,” “prime implicant,” and “essential prime impli cant.” A product term is an *implicant* of a function if the function has the value 1 for all minterms of the product term. Clearly, all rectangles on a map made up of squares containing 1s correspond to implicants. If the removal of any literal from an implicant *P* results in a product term that is not an implicant of the function, then *P* is a *prime implicant*. On a map for an *n*-variable function, the set of prime implicants corresponds to the set of all rectangles made up of 2*m* squares contain ing 1s (*m*  0, 1, …, *n*), with each rectangle containing as many squares as possible. If a minterm of a function is included in only one prime implicant, that prime

implicant is said to be *essential*. Thus, if a square containing a 1 is in only one rect angle representing a prime implicant, then that prime implicant is essential. In Figure 10(c), the terms and are essential prime implicants, and the

*AC AC*

terms and are *nonessential prime implicants*.

*AB BC*

The prime implicants of a function can be obtained from a map of the func

tion as all possible maximum collections of 2*m* squares containing 1s (*m*  0, 1, …, *n*) that constitute rectangles. This means that a single 1 on a map represents a prime implicant if it is not adjacent to any other 1s. Two adjacent 1s form a rectan gle representing a prime implicant, provided that they are not within a rectangle of four or more squares containing 1s. Four 1s form a rectangle representing a prime implicant if they are not within a rectangle of eight or more squares containing 1s, and so on. Each essential prime implicant contains at least one square that is not contained in any other prime implicant.

67

**COMBINATIONAL LOGIC CIRCUITS**

CD

AB

00 01

C

11 10

00

0

1

231 1

01

4 5

7 6

1 1 1 1

12

141513

B

11

1 1

A

**FIGURE 13**

10

8 9 D

11 10

Prime Implicants for Example 10: *AD BD , ,* and *AB*

The systematic procedure for finding the optimized expression from the map requires that we first determine all prime implicants. Then, the optimized expres sion is obtained from the logical sum of all the essential prime implicants, plus other prime implicants needed to include remaining minterms not included in the essential prime implicants. This procedure will be clarified by examples.

**EXAMPLE 10 Simplification Using Prime Implicants**

Consider the map of Figure 13. There are three ways that we can combine four squares into rectangles. The product terms obtained from these combinations are *A D A A D*

the prime implicants of the function, *D*, *B* and *B*. The terms *D* and *B A*

are essential prime implicants, but *B* is not essential. This is because minterms 1 *A*

and 3 are included only in the term *D*, and minterms 12 and 14 are included only in the term *B* . But minterms 4, 5, 6, and 7 are each included in two prime impli

*D*

cants, one of which is *B*, so the term *B* is not an essential prime implicant. In

*A A*

fact, once the essential prime implicants are chosen, the term *AB* is not needed, because all the minterms are already included in the two essential prime impli cants. The optimized expression for the function of Figure 13 is

*F AD BD*

■

**EXAMPLE 11 Simplification Via Essential and Nonessential Prime Implicants**

A second example is shown in Figure 14. The function plotted in part (a) has seven minterms. If we try to combine squares, we will find that there are six prime implicants. In order to obtain a minimum number of terms for the function, we must first determine the prime implicants that are essential. As shown in blue in part (b) of the figure, the function has four essential prime implicants. The product term is essential because it is the only prime implicant that includes

*ABCD*

minterm 0. Similarly, the product terms , , and are essential

*BCD ABC ABC*

prime implicants because they are the only ones that include minterms 5, 12, and 68

**COMBINATIONAL LOGIC CIRCUITS**

CD

AB

00

01

11

A

00 01 1

1

1 1

C

11 10 1

B

CD

AB

00

01

11

A

00 01 1

1

1 1

C

11 10 1

B

1

1

10

D

1

1

10

D

(a) Plotting the minterms **FIGURE 14**

(b) Essential prime implicants

Simplification with Prime Implicants in Example 11

10, respectively. Minterm 15 is included in two nonessential prime implicants. The optimized expression for the function consists of the logical sum of the four essen tial prime implicants and one prime implicant that includes minterm 15:

⎜⎜⎛

*F ABCD BCD ABC ABC*

*ACD* or

⎝ *ABD*

■

The identification of essential prime implicants in the map provides an addi

tional tool which shows the terms that must absolutely appear in every sum-of products expression for a function and provides a partial structure for a more systematic method for choosing patterns of prime implicants.

**Nonessential Prime Implicants**

Beyond using all essential prime implicants, the following rule can be applied to include the remaining minterms of the function in nonessential prime implicants: **Selection Rule:** Minimize the overlap among prime implicants as much as pos

sible. In particular, in the final solution, make sure that each prime implicant selected includes at least one minterm not included in any other prime implicant selected. In most cases, this results in a simplified, although not necessarily optimum,

sum-of-products expression. The use of the selection rule is illustrated in the next example.

**EXAMPLE 12 Simplifying a Function Using the Selection Rule**

Find a simplified sum-of-products form for (0, 1, 2, 4, 5, 10, 11,

*FABCD* ( ) *,,,* = *m*

13, 15).

The map for *F* is given in Figure 15, with all prime implicants shown. is

*A C*

the only essential prime implicant. Using the preceding selection rule, we can choose the remaining prime implicants for the sum-of-products form in the order

69

**COMBINATIONAL LOGIC CIRCUITS**

CD

AB

00 01

C

11 10

3

A

00 01 11

1 1

1 1 1

1

1

B

10

D

11

1 2

**FIGURE 15**

Map for Example 12

indicated by the numbers. Note how the prime implicants 1 and 2 are selected in ( ) *ABD*

order to include minterms without overlapping. Prime implicant 3 and prime implicant both include the one remaining minterm 0010, and prime

*BCD*

implicant 3 is arbitrarily selected to include the minterm and complete the sum-of products expression:

*FABCD* ( ) ,,, *AC ABD ABC ABD*

The prime implicants not used are shown in black in Figure 15. ■

**Product-of-Sums Optimization**

The optimized Boolean functions derived from the maps in all of the previous examples were expressed in sum-of-products form. With only minor modification, the product-of-sums form can be obtained.

The procedure for obtaining an optimized expression in product-of-sums

form follows from the properties of Boolean functions. The 1s placed in the squares of the map represent the minterms of the function. The minterms not included in the function belong to the complement of the function. From this, we see that the complement of a function is represented in the map by the squares not marked by 1s. If we mark the empty squares with 0s and combine them into valid rectangles, we obtain an optimized expression of the complement of the function, *F F*

. We then take the complement of to obtain *F* as a product of sums. This is done by taking the dual and complementing each literal, as in Example 3.

**EXAMPLE 13 Simplifying a Product-of-Sums Form**

Simplify the following Boolean function in product-of-sums form:

*FABCD* ( ) ,,, *m*( ) 0 1 2 5 8 9 10 ,,,,,,

70

**COMBINATIONAL LOGIC CIRCUITS**

CD

AB

00 01

C

11 10

00

1 1

01

0

1 1

0

0 0

B

A

11 10

0 1

0 0 0

1 1 0

D

**FIGURE 16**

Map for Example 13

The 1s marked in the map of Figure 16 represent the minterms of the function. The squares marked with 0s represent the minterms not included in *F* and therefore denote the complement of *F*. Combining the squares marked with 0s, we obtain the optimized complemented function

*F AB CD BD*

*F*

Taking the dual and complementing each literal gives the complement of . This is *F* in product-of-sums form:

*F AB*  ( ) ( ) *C D* ( ) *B D*

■

The previous example shows the procedure for obtaining the product-of

sums optimization when the function is originally expressed as a sum of minterms. The procedure is also valid when the function is originally expressed as a product of maxterms or a product of sums. Remember that the maxterm numbers are the same as the minterm numbers of the complemented function, so 0s are entered in the map for the maxterms or for the complement of the function. To enter a func tion expressed as a product of sums into the map, we take the complement of the function and, from it, find the squares to be marked with 0s. For example, the function

*F ABC*  ( ) ( ) *B D*

can be plotted in the map by first obtaining its complement,

*F ABC BD*

*F*

and then marking 0s in the squares representing the minterms of . The remaining squares are marked with 1s. Then, combining the 1s gives the optimized expression in sum-of-products form. Combining the 0s and then complementing gives the optimized expression in product-of-sums form. Thus, for any function plotted on

71

**COMBINATIONAL LOGIC CIRCUITS**

the map, we can derive the optimized function in either one of the two standard forms.

**Don’t-Care Conditions**

The minterms of a Boolean function specify all combinations of variable values for which the function is equal to 1. The function is assumed to be equal to 0 for the rest of the minterms. This assumption, however, is not always valid, since there are applications in which the function is not specified for certain variable value combi nations. There are two cases in which this occurs. In the first case, the input combi nations never occur. As an example, the four-bit binary code for the decimal digits has six combinations that are not used and not expected to occur. In the second case, the input combinations are expected to occur, but we do not care what the outputs are in response to these combinations. In both cases, the outputs are said to be unspecified for the input combinations. Functions that have unspecified out puts for some input combinations are called *incompletely specified functions*. In most applications, we simply do not care what value is assumed by the function for the unspecified minterms. For this reason, it is customary to call the unspecified minterms of a function *don’t-care conditions*. These conditions can be used on a map to provide further simplification of the function.

It should be realized that a don’t-care minterm cannot be marked with a 1 on

the map, because that would require that the function always be a 1 for such a min term. Likewise, putting a 0 in the square requires the function to be 0. To distin guish the don’t-care condition from 1s and 0s, an X is used. Thus, an X inside a square in the map indicates that we do not care whether the value of 0 or 1 is assigned to the function for the particular minterm.

In choosing adjacent squares to simplify the function in a map, the don’t-care

minterms may be used. When simplifying function *F* using the 1s, we can choose to include those don’t-care minterms that give the simplest prime implicants for *F*. When simplifying function using the 0s, we can choose to include those don’t

*F*

care minterms that give the simplest prime implicants for , irrespective of those

*F*

included in the prime implicants for *F*. In both cases, whether or not the don’t-care minterms are included in the terms in the final expression is irrelevant. The han dling of don’t-care conditions is illustrated in the next example.

**EXAMPLE 14 Simplification with Don’t-Care Conditions**

To clarify the procedure for handling the don’t-care conditions, consider the fol lowing incompletely specified function *F* that has three don’t-care minterms *d*:

*FABCD* ( ) ,,, *m*( ) 1 3 7 11 15 ,,, ,

*dABCD* ( ) ,,, *m*( ) 025 , ,

The minterms of *F* are the variable combinations that make the function equal to 1. The minterms of *d* are the don’t-care minterms. The map optimization is shown

72

**COMBINATIONAL LOGIC CIRCUITS**

CD

AB

00 01

C

11 10

CD

AB

00 01

C

11 10

00

X X 1 1

00 X

1 1 X

01

0 X 1 0 B

01

0 X 1 0 B

A

11 10

00 0 1 00 0 1

D

B

**FIGURE 17**

A

11 10

001 001 D

0 0

Example with Don’t-Care Conditions

in Figure 17. The minterms of *F* are marked by 1s, those of *d* are marked by X’s, and the remaining squares are filled with 0s. To get the simplified function in sum of-products form, we must include all five 1s in the map, but we may or may not include any of the X’s, depending on what yields the simplest expression for the function. The term *CD* includes the four minterms in the third column. The remaining minterm in square 0001 can be combined with square 0011 to give a three-literal term. However, by including one or two adjacent X’s, we can combine four squares into a rectangle to give a two-literal term. In part (a) of the figure, don’t-care minterms 0 and 2 are included with the 1s, which results in the simplified function

*F CD AB*

In part (b), don’t-care minterm 5 is included with the 1s, and the simplified func tion now is

*F CD AD*

The two expressions represent two functions that are algebraically unequal. Both include the specified minterms of the original incompletely specified function, but each includes different don’t-care minterms. As far as the incompletely specified function is concerned, both expressions are acceptable. The only difference is in the value of *F* for the unspecified minterms.

It is also possible to obtain an optimized product-of-sums expression for the

function of Figure 17. In this case, the way to combine the 0s is to include don’t care minterms 0 and 2 with the 0s, giving the optimized complemented function

*F D AC*

*F*

Taking the complement of gives the optimized expression in product-of-sums form:

73

**COMBINATIONAL LOGIC CIRCUITS**

*F DA C*  ( )

■

The foregoing example shows that the don’t-care minterms in the map are

initially considered as representing both 0 and 1. The 0 or 1 value that is eventually assigned depends on the optimization process. Due to this process, the optimized function will have a 0 or 1 value for each minterm of the original function, includ ing those that were initially don’t cares. Thus, although the outputs in the initial specification may contain Xs, the outputs in a particular implementation of the specification are only 0s and 1s.

**MORE OPTIMIZATION** This supplement gives a procedure for selecting prime impli cants that guarantees an optimum solution. In addition, it presents a symbolic method for performing prime-implicant generation and a tabular method for prime-implicant selection.

**6 PRAGMATIC TWO-LEVEL OPTIMIZATION**

The two-level optimization procedure that achieves a true optimum solution requires: 1) use of the minterms of the function, 2) the generation of all prime implicants, and 3) a selection process that potentially involves a huge number of alternative prime-implicant selection solutions. A computer algorithm for this pro cedure when applied to many realistic problems is impractical due to the number of minterms or prime implicants involved or the number of solutions that must be examined. As a consequence, algorithms have been developed that do not 1) depend upon the enumeration of minterms, 2) require generation of all prime implicants, or 3) require enumeration of alternate prime-implicant selections. The best known and most widely used of these algorithms is Espresso II. Using the vehicle of K-maps, we will illustrate the Espresso II algorithm. For simplicity, product-of-sums specifications, multiple output functions, and don’t cares are not considered. Further, we will not illustrate the complex underlying details that con tribute significantly to the efficiency and effectiveness of the algorithm. Finally, we use gate-input count as the cost measure rather than the complex multidimensional measure used in Espresso. The resulting simplified form of the algorithm appears in Figure 18. The five routines providing core operations for execution of Espresso are: EXPAND, ESSENTIAL\_PRIMES, IRREDUNDANT\_COVER, REDUCE, and LAST\_GASP. The function of each of these routines is outlined next, followed by an example that illustrates the execution of Espresso. In these discussions, we will deal with various sets of implicants that cover all of the minterms of *F*. Such a set is called a cover of *F*, denoted by ***F***.

EXPAND replaces each of the implicants of the current ***F*** with prime impli

cants and insures that the cover is reduced in the sense that no implicants remain that are covered by any single implicant. EXPAND depends on the order in which the original implicants are processed. The order selects the largest (in terms of size on the K-map) remaining unprocessed implicant first. If an implicant can be expanded into multiple prime implicants, the prime implicant chosen is the one

74

**COMBINATIONAL LOGIC CIRCUITS**

Input Function F and its initial cover **F**

Initialize Cost = Gate Input Cost of **F**

Loop 1: Execute EXPAND

On first pass only, execute ESSENTIAL\_PRIMES

Execute IRREDUNDANT\_COVER

If Cost not improved, goto OUT,

Else update Cost

Loop 2: Execute REDUCE

Goto Loop 1

Out: Execute LAST GASP

If Cost not improved, goto QUIT

Else goto Loop 2

Quit: Place Essential Primes back in **F**

Return Final **F** and Final Cost

**FIGURE 18**

Simplified Espresso Algorithm

which 1) covers the maximum number of other current implicants of *F*, and 2) in the case of ties, is the largest implicant.

ESSENTIAL\_PRIMES evaluates each of the current implicants to deter

mine if it is an essential prime implicant, a concept defined in the first part of Sec tion 5. An implicant is *essential prime* if it contains a minterm that is surrounded either by other minterms of the prime or by 0 values in all *n* directions when *n* is the number of variables in the function. This test is applied to each of the current implicants in **F**. Since the essential primes are required in all solutions, they are removed temporarily from the solution space. Also, since they are guaranteed to be covered, their minterms are changed to don’t cares in the solution space. We denote the changed function as *F*−E, and a cover of *F*−E without the essential primes as **F**−E.

IRREDUNDANT COVER is used on the implicants in **F**−E. First, it removes

implicants that are totally redundant in the sense that all can be removed without exposing any uncovered minterms (squares). Second, it takes the remaining impli cants and performs a selection process that resembles a formalization of the selec tion rule in Section 5.

REDUCE is used to move away from a solution called a *local minimum*. The

solution is irredundant but, based on the possibility that all prime implicants have not been found, may not be a minimum-cost solution. In REDUCE, each of the implicants is reduced to the smallest implicant possible while still maintaining the cover of the function *F*−E. REDUCE is performed sequentially on each of the implicants. This process is order dependent, since the reduction of one implicant potentially affects squares involved in the reduction of a subsequent implicant. The ordering is described as follows: (1) choose the largest implicant first, and (2) place the remaining implicants in the order of smallest number of positions in which the given implicant differs from the largest one.

75