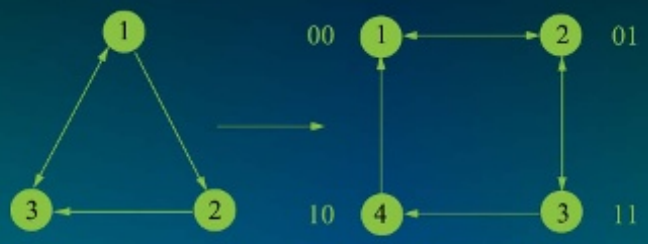
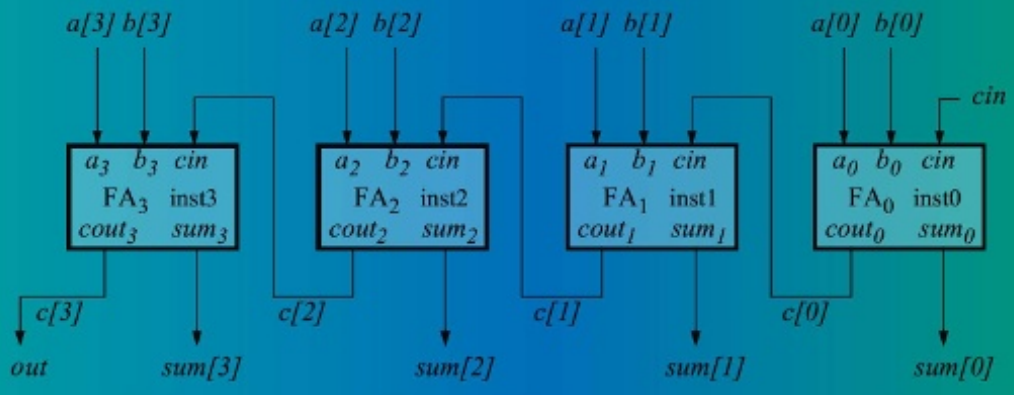
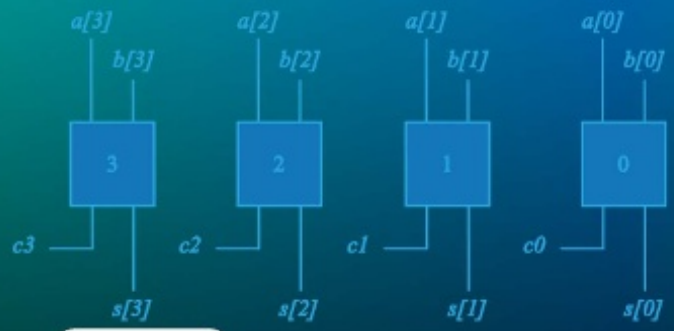


Verilog HDL Design Examples



Joseph Cavanagh



CRC Press
Taylor & Francis Group

Verilog HDL
Design Examples

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Verilog HDL
Design Examples

Joseph Cavanagh

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed on acid-free paper
Version Date: 20170918

International Standard Book Number-13: 978-1-138-09995-1 (Hardback)

Library of Congress Cataloging-in-Publication Data

Names: Cavanagh, Joseph, author.
Title: Verilog HDL design examples / Joseph Cavanagh.
Description: Boca Raton, FL : CRC Press, 2017. | Includes index.
Identifiers: LCCN 2017022734| ISBN 9781138099951 (hardback : acid-free paper)
| ISBN 9781315103846 (ebook)
Subjects: LCSH: Digital electronics--Computer-aided design. | Logic design. |
Verilog (Computer hardware description language)
Classification: LCC TK7868.D5 C3948 2017 | DDC 621.381--dc23
LC record available at https://lccn.loc.gov/2017022734

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

To David Dutton
CEO, Silvaco, Inc.
for generously providing the SILOS Simulation Environment software
for all of my books that use Verilog HDL and for his continued support

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

CONTENTS

PREFACE

The Verilog language provides a means to model a digital system at many levels of abstraction from a logic gate, to a complex digital system, to a mainframe computer. The purpose of this book is to present the Verilog language together with a wide variety of examples so that the reader can gain a firm foundation in the design of digital systems using Verilog HDL. The different modeling constructs supported by Verilog are described in detail.

Numerous examples are designed in each chapter. The examples include logical operations, counters of different moduli, half adders, full adders, a carry lookahead adder, array multipliers, the Booth multiply algorithm, different types of Moore and Mealy machines, including sequence detectors, arithmetic and logic units (ALUs). Also included are synchronous sequential machines and asynchronous sequential machines, including pulse-mode asynchronous sequential machines.

Emphasis is placed on the detailed design of various Verilog projects. The projects include the design module, the test bench module, and the outputs obtained from the simulator that illustrate the complete functional operation of the design. Where applicable, a detailed review of the theory of the topic is presented together with the logic design principles. This includes state diagrams, Karnaugh maps, equations, and the logic diagram.

The book is intended to be tutorial, and as such, is comprehensive and self-contained. All designs are carried through to completion — nothing is left unfinished or partially designed. Each chapter includes numerous problems of varying complexity to be designed by the reader.

Chapter 1 presents an overview of the Verilog HDL language and discusses the different design methodologies used in designing a project. The chapter is intended to introduce the reader to the basic concepts of Verilog modeling techniques, including dataflow modeling, behavioral modeling, and structural modeling. Examples are presented to illustrate the different modeling techniques. There are also sections that incorporate more than one modeling construct in a mixed-design model. The concept of ports and modules is introduced in conjunction with the use of test benches for module design verification.

The chapter introduces gate-level modeling using built-in primitive gates. Verilog has a profuse set of built-in primitive gates that are used to model nets, including **and**, **nand**, **or**, **nor**, **xor**, **xnor**, and **not**, among others. This chapter presents a design methodology that is characterized by a low level of abstraction, in which the logic hardware is described in terms of gates. This is similar to designing logic by drawing logic gate symbols.

The chapter also describes different techniques used to design logic circuits using dataflow modeling. These techniques include the continuous assignment statement, reduction operators, the conditional operator, relational operators, logical operators, bitwise operators, and shift operators.

This chapter also presents behavioral modeling, which describes the *behavior* of a digital system and is not concerned with the direct implementation of logic gates, but more on the architecture of the system. This is an algorithmic approach to hardware implementation and represents a higher level of abstraction than previous modeling methods.

Also included in this chapter is structural modeling, which consists of instantiating one or more of the following design objects into the module:

- Built-in primitives
- User-defined primitives (UDPs)
- Design modules

Instantiation means to use one or more lower-level modules — including logic primitives — that are interconnected in the construction of a higher-level structural module.

Chapter 2 presents combinational logic design using Verilog HDL. Verilog is used to design multiplexers, comparators, programmable logic devices, and a variety of logic equations in this chapter. A combinational logic circuit is one in which the outputs are a function of the present inputs only. This chapter also includes number systems and Boolean algebra. The number systems are binary, octal, decimal, and hexadecimal. Boolean algebra is a systematic treatment of the logic operations AND, OR, NOT, exclusive-OR, and exclusive-NOR. The axioms and theorems of Boolean algebra are also presented. The programmable logic devices include programmable read-only memories, programmable array logic devices, and programmable logic array devices.

Chapter 3 presents the design of sequential logic using Verilog HDL. The examples include both Moore and Mealy sequential machines. Moore machines are synchronous sequential machines in which the output function produces an output vector which is determined by the present state only, and is not a function of the present inputs. This is in contrast to Mealy synchronous sequential machines in which the output function produces an output vector which is determined by both the present input vector and the present state of the machine.

This chapter describes three types of sequential machines: synchronous sequential machines which use a system clock and generally require a state diagram or a state table for its precise description; asynchronous sequential machines in which there is no system clock — state changes occur on the application of input signals only; and pulse-mode asynchronous sequential machines in which state changes occur on the application of input pulses which trigger the storage elements, rather than on a system clock signal.

Chapter 4 presents arithmetic operations for the three primary number representations: fixed-point, binary-coded decimal (BCD), and floating-point. For fixed-point, the radix point is placed to the immediate right of the number for integers or to

the immediate left of the number for fractions. For binary-coded decimal, each decimal digit can be encoded into a corresponding binary number; however, only ten decimal digits are valid. For floating-point, the numbers consist of the following three fields: a sign bit, an exponent *e*, and a fraction *f*, as shown below for radix *r*. Addition, subtraction, multiplication, and division will be applied to all three number representations.

$$A = f \times r^e$$

For fixed-point addition, the two operands are the augend and the addend. The addend is added to the augend to produce the sum. Addition of two binary operands treats both signed and unsigned operands the same — there is no distinction between the two types of numbers during the add operation. If the numbers are signed, then the sign bit can be extended to the left indefinitely without changing the value of the number.

For fixed-point subtraction, the two operands are the minuend and the subtrahend. The subtrahend is subtracted from the minuend to produce the difference. Subtraction can be performed in all three number representations: sign magnitude, diminished-radix complement, and radix complement; however, radix complement is the easiest and most widely used method for subtraction in any radix.

For fixed-point multiplication, the two operands are the multiplicand and the multiplier. The *n*-bit multiplicand is multiplied by the *n*-bit multiplier to generate the 2*n*-bit product. In all methods of multiplication the product is usually 2*n* bits in length. The operands can be either unsigned or signed numbers in 2s complement representation.

For fixed-point division, the two operands are the dividend and the divisor. The 2*n*-bit dividend is divided by the *n*-bit divisor to produce an *n*-bit quotient and an *n*-bit remainder, as shown below.

2*n*-bit dividend = (*n*-bit divisor × *n*-bit quotient) + *n*-bit remainder

For binary-coded decimal addition, and other BCD calculations, the highest-valued decimal digit is 9, which requires four bits in the binary representation (1001). Therefore, each operand is represented by a 4-bit BCD code. Since four binary bits have sixteen combinations (0000 – 1111) and the range for a single decimal digit is 0 – 9, six of the sixteen combinations (1010 – 1111) are invalid for BCD. These invalid BCD digits must be converted to valid digits by adding six to the digit. This is the concept for addition with sum correction. The adder must include correction logic for intermediate sums that are greater than or equal to 1010 in radix 2.

For binary-coded decimal subtraction, the BCD code is not self-complementing as is the radix 2 fixed-point number representation; that is, the $r - 1$ complement cannot be acquired by inverting each bit of the 4-bit BCD digit. Therefore, a 9s complementer must be designed that provides the same function as the diminished-radix complement for the fixed-point number representation. Thus, subtraction in BCD is essentially the same as in fixed-point binary.

For binary-coded decimal multiplication, the algorithms for BCD multiplication are more complex than those for fixed-point multiplication. This is because decimal digits consist of four binary bits and have values in the range of 0 to 9, whereas fixed-point digits have values of 0 or 1. One method that is commonly used is to perform the multiplication in the fixed-point number representation; then convert the product to the BCD number representation. This is accomplished by utilizing a binary-to-decimal converter, which is used to convert a fixed-point multiplication product to the decimal number representation.

For binary-coded decimal division, the division process is first reviewed by using examples of the restoring division method. Then a mixed-design (behavioral/dataflow) module is presented. The dividend is an 8-bit vector, *a[7:0]*; the divisor is a 4-bit vector, *b[3:0]*; and the result is an 8-bit quotient/remainder vector, *rslt[7:0]*.

For floating-point addition, the material presented is based on the Institute of Electrical and Electronics Engineers (IEEE) Standard for Binary Floating-Point Arithmetic IEEE Std 754-1985 (Reaffirmed 1990). Floating-point numbers consist of the following three fields: a sign bit *s*, an exponent *e*, and a fraction *f*. Unbiased and biased exponents are explained. Numerical examples are given that clarify the technique for adding floating-point numbers. The floating-point addition algorithm is given in a step-by-step procedure. A floating-point adder is implemented using behavioral modeling.

For floating-point subtraction, several numerical examples are presented that graphically portray the steps required for true addition and true subtraction for floating-point operands. True addition produces a result that is the sum of the two operands disregarding the signs; true subtraction produces a result that is the difference of the two operands disregarding the signs. A behavioral module is presented that illustrates subtraction operations which yield results that are either true addition or true subtraction.

For floating-point multiplication, numerical examples are presented that illustrate the operation of floating-point multiplication. In floating-point multiplication, the fractions are multiplied and the exponents are added. The fractions are multiplied by any of the methods previously used in fixed-point multiplication. The operands are two normalized floating-point operands. Fraction multiplication and exponent addition are two independent operations and can be done in parallel. Floating-point multiplication is defined as follows:

$$A \times B = (f_A \times f_B) \times r^{(e\mathrm{A} + e\mathrm{B})}$$

For floating-point division, the operation is accomplished by dividing the fractions and subtracting the exponents. The fractions are divided by any of the methods presented in the section on fixed-point division and overflow is checked in the same manner. Fraction division and exponent subtraction are two independent operations and can be done in parallel. Floating-point division is defined as follows:

$$A / B = (f_A / f_B) \times r^{(e\mathrm{A} - e\mathrm{B})}$$

Appendix A presents a brief discussion on event handling using the event queue. Operations that occur in a Verilog module are typically handled by an event queue.

Appendix B presents a procedure to implement a Verilog project.

Appendix C contains the solutions to selected problems in each chapter.

The material presented in this book represents more than two decades of computer equipment design by the author. The book is not intended as a text on logic design, although this subject is reviewed where applicable. It is assumed that the reader has an adequate background in combinational and sequential logic design. The book presents the Verilog HDL with numerous design examples to help the reader thoroughly understand this popular HDL.

This book is designed for practicing electrical engineers, computer engineers, and computer scientists; for graduate students in electrical engineering, computer engineering, and computer science; and for senior-level undergraduate students.

A special thanks to David Dutton, CEO of Silvaco Incorporated, for allowing use of the SILOS Simulation Environment software for the examples in this book. SILOS is an intuitive, easy-to-use, yet powerful Verilog HDL simulator for logic verification.

I would like to express my appreciation and thanks to the following people who gave generously of their time and expertise to review the manuscript and submit comments: Professor Daniel W. Lewis, Department of Computer Engineering, Santa Clara University who supported me in all my endeavors; Geri Lamble; and Steve Midford. Thanks also to Nora Konopka and the staff at Taylor & Francis for their support.

Joseph Cavanagh

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

By the Same Author

SEQUENTIAL LOGIC and VERILOG HDL FUNDAMENTALS

X86 ASSEMBLY LANGUAGE and C FUNDAMENTALS

COMPUTER ARITHMETIC and Verilog HDL Fundamentals

DIGITAL DESIGN and Verilog HDL Fundamentals

VERILOG HDL: Digital Design and Modeling

SEQUENTIAL LOGIC: Analysis and Synthesis

DIGITAL COMPUTER ARITHMETIC: Design and Implementation

THE COMPUTER CONSPIRACY
A novel

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

1

Introduction to Logic Design Using Verilog HDL

This chapter provides an introduction to the design methodologies and modeling constructs of the Verilog hardware description language (HDL). Modules, ports, and test benches will be presented. This chapter introduces Verilog in conjunction with combinational logic and sequential logic. The Verilog simulator used in this book is easy to learn and use, yet powerful enough for any application. It is a logic simulator — called SILOS — developed by Silvaco Incorporated for use in the design and verification of digital systems. The SILOS simulation environment is a method to quickly prototype and debug any application-specific integrated circuit (ASIC), field-programmable gate array (FPGA), or complex programmable logic device (CPLD) design.

Language elements will be described, which consist of comments, logic gates, logic macro functions, parameters, procedural control statements which modify the flow of control in a program, and data types. Also presented will be expressions consisting of operands and operators. Built-in primitives are discussed which are used to describe a net. In addition to built-in primitives, user-defined primitives (UDPs) are presented which are at a higher-level logic function than built-in primitives.

This chapter also presents dataflow modeling which is at a higher level of abstraction than built-in primitives or user-defined primitives. Dataflow modeling corresponds one-to-one with conventional logic design at the gate level. Also introduced is behavioral modeling which describes the behavior of the system and is not concerned with the direct implementation of the logic gates but more on the architecture of the machine. Structural modeling is presented which instantiates one or more lower-level modules into the design. The objects that are instantiated are called *instances*. A

module can be a logic gate, an adder, a multiplexer, a counter, or some other logical function. Structural modeling is described by the interconnection of these lower-level logic primitives of modules.

Tasks and functions are also included in this chapter. These constructs allow a behavioral module to be partitioned into smaller segments. Tasks and functions permit modules to execute common code segments that are written once then called when required, thus reducing the amount of code needed.

1.1 Logic Elements

Logic elements are the constituent parts of the Verilog language. They consist of comments, logic gates, parameters, procedural control statements which modify the flow of control in a behavior, and data types.

1.1.1 Comments

Comments can be inserted into a Verilog module to explain the function of a particular block of code or a line of code. There are two types of comments: single line and multiple lines. A single-line comment is indicated by a double forward slash (//) and may be placed on a separate line or at the end of a line of code, as shown below.

```
//This is a single-line comment on a dedicated line
assign z1 = x1 | x2   //This is a comment on a line of code
```

A single-line comment usually explains the function of the following block of code. A comment on a line of code explains the function of that particular line of code. All characters that follow the forward slashes are ignored by the compiler.

A multiple-line comment begins with a forward slash followed by an asterisk (/*) and ends with an asterisk followed by a forward slash (*/), as shown below. Multiple-line comments cannot be nested. All characters within a multiple-line comment are ignored by the compiler.

```
/*This is a multiple-line comment.
   More comments go here.
   More comments. */
```

1.1.2 Logic Gates

Figure 1.1 shows the logic gate distinctive-shape symbols. The polarity symbol "○" indicates an active-low assertion on either an input or an output of a logic symbol.

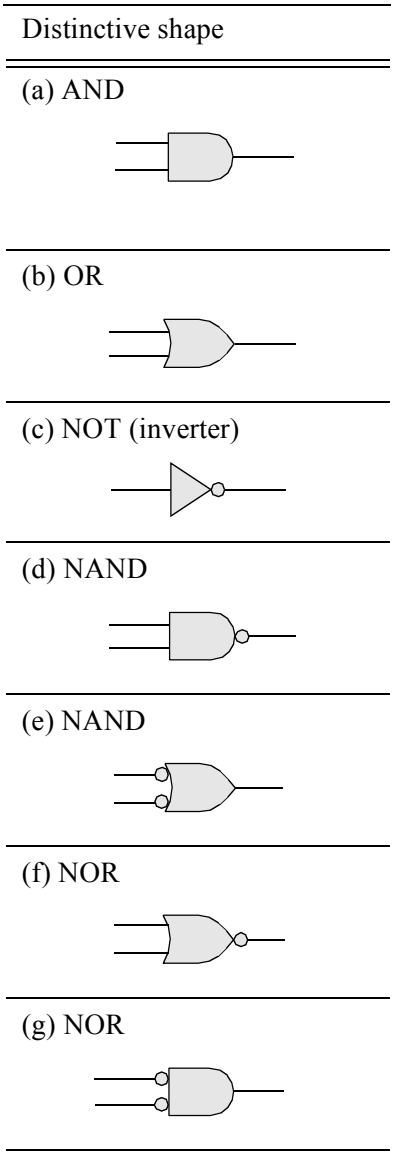
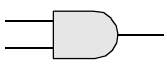


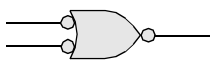
Figure 1.1 Logic gate symbols for logic design: (a) AND gate, (b) OR gate, (c) NOT function (inverter), (d) NAND gate, (e) NAND gate for the OR function, (f) NOR gate, (g) NOR gate for the AND function.

The AND gate can also be used for the OR function, as shown below.

AND gate for the AND function

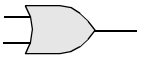


AND gate for the OR function



The OR gate can also be used for the AND function, as shown below.

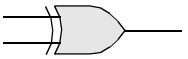
OR gate for the OR function



OR gate for the AND function

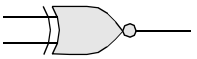
An exclusive-OR gate is shown below. The output of an exclusive-OR gate is a logical 1 whenever the two inputs are different.

Exclusive-OR gate



An exclusive-NOR gate is shown below. An exclusive-NOR gate is also called an equality function because the output is a logical 1 whenever the two inputs are equal.

Exclusive-NOR gate



Truth tables for the logic elements are shown in Table 1.1, Table 1.2, Table 1.3, Table 1.4, Table 1.5, and Table 1.6.

Table 1.1 Truth Table for the AND Gate

x_1	x_2	z_1
0	0	0
0	1	0
1	0	0
1	1	1

Table 1.2 Truth Table for the NAND Gate

x_1	x_2	z_1
0	0	1
0	1	1
1	0	1
1	1	0

Table 1.3 Truth Table for the OR Gate

x_1	x_2	z_1
0	0	0
0	1	1
1	0	1
1	1	1

Table 1.4 Truth Table for the NOR Gate

x_1	x_2	z_1
0	0	1
0	1	0
1	0	0
1	1	0

Table 1.5 Truth Table for the Exclusive-OR Function

x_1	x_2	z_1
0	0	0
0	1	1
1	0	1
1	1	0

Table 1.6 Truth Table for the Exclusive-NOR Function

x_1	x_2	z_1
0	0	1
0	1	0
1	0	0
1	1	1

Fan-In Logic gates for the AND and OR functions can be extended to accommodate more than two variables; that is, more than two inputs. The number of inputs available at a logic gate is called the *fan-in*.

Fan-Out The *fan-out* of a logic gate is the maximum number of inputs that the gate can drive and still maintain acceptable voltage and current levels. That is, the fan-out defines the maximum load that the gate can handle.

1.1.3 Logic Macro Functions

Logic macro functions are those circuits that consist of several logic primitives to form larger more complex functions. Combinational logic macros include circuits such as multiplexers, decoders, encoders, comparators, adders, subtractors, array multipliers, array dividers, and error detection and correction circuits. Sequential logic macros include circuits such as: *SR* latches; *D* and *JK* flip-flops; counters of various moduli, including count-up and count-down counters; registers, including shift registers; and sequential multipliers and dividers. This section will present the functional operation of multiplexers, decoders, encoders, priority encoders, and comparators.

Multiplexers A multiplexer is a logic macro device that allows digital information from two or more data inputs to be directed to a single output. Data input selection is controlled by a set of select inputs that determine which data input is gated to the output. The select inputs are labeled $s_0, s_1, s_2, \cdots, s_i, \cdots, s_{n-1}$, where s_0 is the low-order select input with a binary weight of 2^0 and s_{n-1} is the high-order select input with a binary weight of 2^{n-1}. The data inputs are labeled $d_0, d_1, d_2, \cdots, d_j, \cdots, d_{n-1}$. Thus, if a multiplexer has n select inputs, then the number of data inputs will be 2^n and will be labeled d_0 through d_{n-1}. For example, if $n = 2$, then the multiplexer has two select inputs s_0 and s_1 and four data inputs d_0, d_1, d_2, and d_3.

The logic diagram for a 4:1 multiplexer is shown in Figure 1.2. There can also be an *enable* input which gates the selected data input to the output. Each of the four data inputs x_0, x_1, x_2, and x_3 is connected to a separate 3-input AND gate. The select inputs

s_0 and s_1 are decoded to select a particular AND gate. The output of each AND gate is applied to a 4-input OR gate that provides the single output z_1. Input lines that are not selected cannot be transferred to the output and are treated as "don't cares."

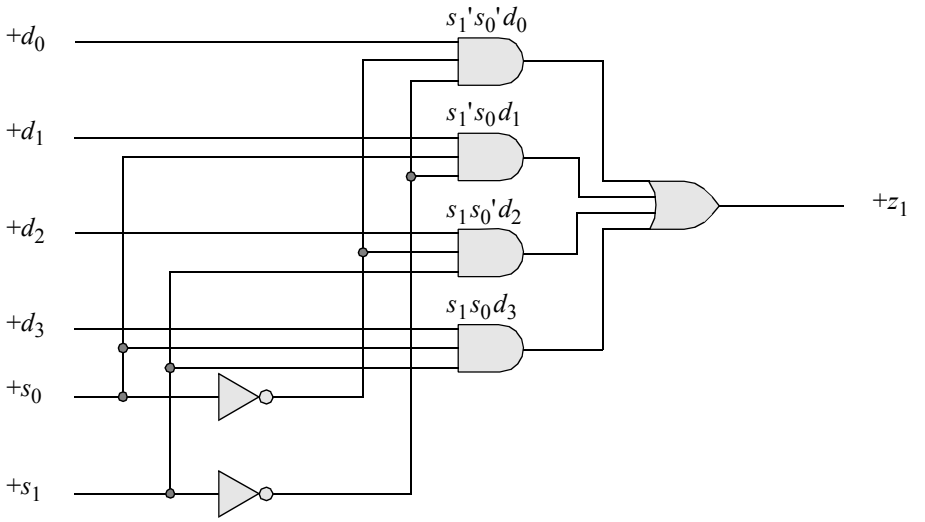


Figure 1.2 Logic diagram for a 4:1 multiplexer.

Figure 1.3 shows a typical multiplexer drawn in the ANSI/IEEE Std. 91-1984 format. Consider the 4:1 multiplexer in Figure 1.3. If $s_1 s_0 = 00$, then data input d_0 is selected and its value is propagated to the multiplexer output z_1. Similarly, if $s_1 s_0 = 01$, then data input d_1 is selected and its value is directed to the multiplexer output.

The equation that represents output z_1 in the 4:1 multiplexer is shown in Equation 1.1. Output z_1 assumes the value of d_0 if $s_1 s_0 = 00$, as indicated by the term $s_1's_0'd_0$. Likewise, z_1 assumes the value of d_1 when $s_1 s_0 = 01$, as indicated by the term $s_1's_0d_1$.

$$z_1 = s_1's_0'd_0 + s_1's_0d_1 + s_1s_0'd_2 + s_1s_0d_3 \tag{1.1}$$

There is a one-to-one correspondence between the data input numbers d_i of a multiplexer and the minterm locations in a Karnaugh map. Equation 1.2 is plotted on the Karnaugh map shown in Figure 1.3(a) using x_3 as a map-entered variable. Minterm location 0 corresponds to data input d_0 of the multiplexer; minterm location 1 corresponds to data input d_1; minterm location 2 corresponds to data input d_2; and minterm location 3 corresponds to data input d_3. The Karnaugh map and the multiplexer implement Equation 1.2, where x_2 is the low-order variable in the Karnaugh map. Figure 1.3(b) shows the implementation using a 4:1 multiplexer.

$$z_1 = x_1 x_2 (x_3') + x_1 x_2' (x_3) + x_1' x_2 \tag{1.2}$$

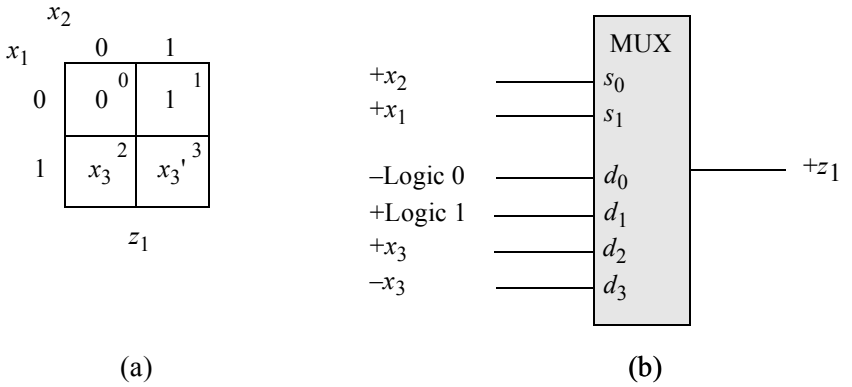


Figure 1.3 Multiplexer using a map-entered variable: (a) Karnaugh map and (b) a 4:1 multiplexer.

Linear-select multiplexers The multiplexer examples described thus far have been classified as *linear-select multiplexers*, because all of the variables of the Karnaugh map coordinates have been utilized as the select inputs for the multiplexer. Since there is a one-to-one correspondence between the minterms of a Karnaugh map and the data inputs of a multiplexer, designing the input logic is relatively straightforward. Simply assign the values of the minterms in the Karnaugh map to the corresponding multiplexer data inputs with the same subscript.

Nonlinear-select multiplexers Although the logic functions correctly according to the equation using a linear-select multiplexer, the design may demonstrate an inefficient use of the 2^p:1 multiplexers. Smaller multiplexers with fewer data inputs could be effectively utilized with a corresponding reduction in machine cost.

For example, the Karnaugh map shown in Figure 1.4 can be implemented with a 4:1 nonlinear-select multiplexer for the function z_1 instead of an 8:1 linear-select multiplexer. Variables x_2 and x_3 will connect to select inputs s_1 and s_0, respectively. When select inputs $s_1 s_0 = x_2 x_3 = 00$, data input d_0 is selected; therefore, $d_0 = 0$. When select inputs $s_1 s_0 = x_2 x_3 = 01$, data input d_1 is selected and d_1 contains the complement of x_1; therefore, $d_1 = x_1'$. When select inputs $s_1 s_0 = x_2 x_3 = 10$, data input d_2 is selected; therefore, $d_2 = 1$. When $s_1 s_0 = x_2 x_3 = 11$, data input d_3 is selected and contains the same value as x_1; therefore, $d_3 = x_1$. The logic diagram is shown in Figure 1.5

The multiplexer of Figure 1.5 can be checked to verify that it operates according to the Karnaugh map of Figure 1.4; that is, for every value of $x_1 x_2 x_3$, output z_1 should generate the same value as in the corresponding minterm location.

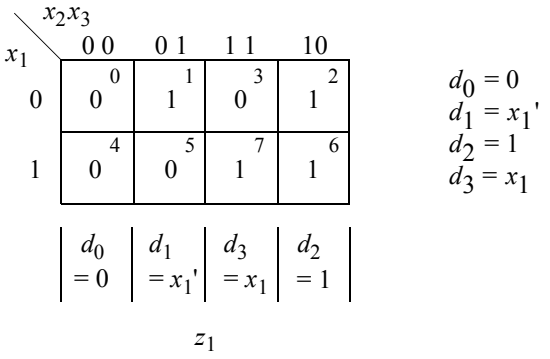


Figure 1.4 Karnaugh map for an example which will be implemented by a 4:1 nonlinear-select multiplexer.

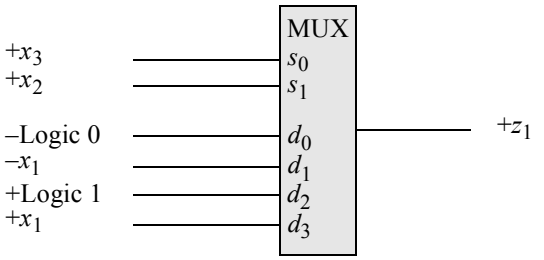


Figure 1.5 A 4:1 nonlinear-select multiplexer to implement the Karnaugh map of Figure 1.4.

Decoders A decoder is a combinational logic macro that is characterized by the following property: For every valid combination of inputs, a unique output is generated. In general, a decoder has n binary inputs and m mutually exclusive outputs, where $2^n \geq m$. An n:m (n-to-m) decoder is shown in Figure 1.6, where the label DX specifies a demultiplexer. Each output represents a minterm that corresponds to the binary representation of the input vector. Thus, $z_i = m_i$, where m_i is the ith minterm of the n input variables.

For example, if $n = 3$ and $x_1x_2x_3 = 101$, then output z_5 is asserted. A decoder with n inputs, therefore, has a maximum of 2^n outputs. Because the outputs are mutually exclusive, only one output is active for each different combination of the inputs. The decoder outputs may be asserted high or low. Decoders have many applications in digital engineering, ranging from instruction decoding to memory addressing to code conversion.

Figure 1.7 illustrates the logic symbol for a 2:4 decoder, where x_1 and x_2 are the binary input variables and z_0, z_1, z_2, and z_3 are the output variables. Input x_2 is the low-order variable. Since there are two inputs, each output corresponds to a different minterm of two variables.

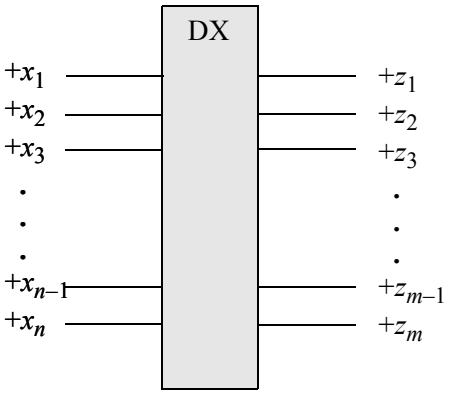


Figure 1.6 An n:m decoder.

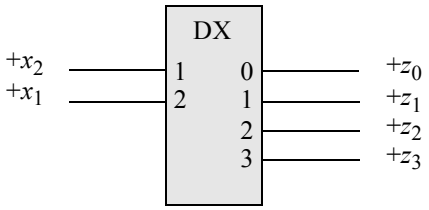


Figure 1.7 Logic symbol for a 2:4 decoder.

A 3:8 decoder is shown in Figure 1.8 which decodes a binary number into the corresponding octal number. The three inputs are x_1, x_2, and x_3 with binary weights of 2^2, 2^1, and 2^0, respectively. The decoder generates an output that corresponds to the decimal value of the binary inputs. For example, if $x_1x_2x_3 = 110$, then output z_6 is asserted high. A decoder may also have an enable function which allows the selected output to be asserted.

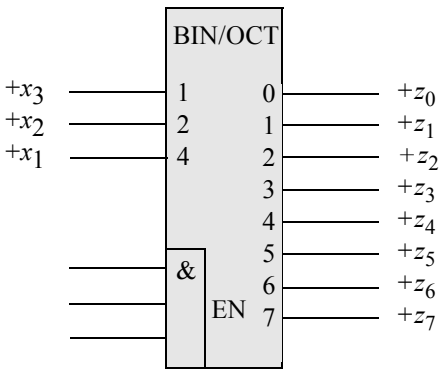


Figure 1.8 A binary-to-octal decoder.

The internal logic for the binary-to-octal decoder of Figure 1.8 is shown in Figure 1.9. The *Enable* gate allows for additional logic functions to control the assertion of the active-high outputs.

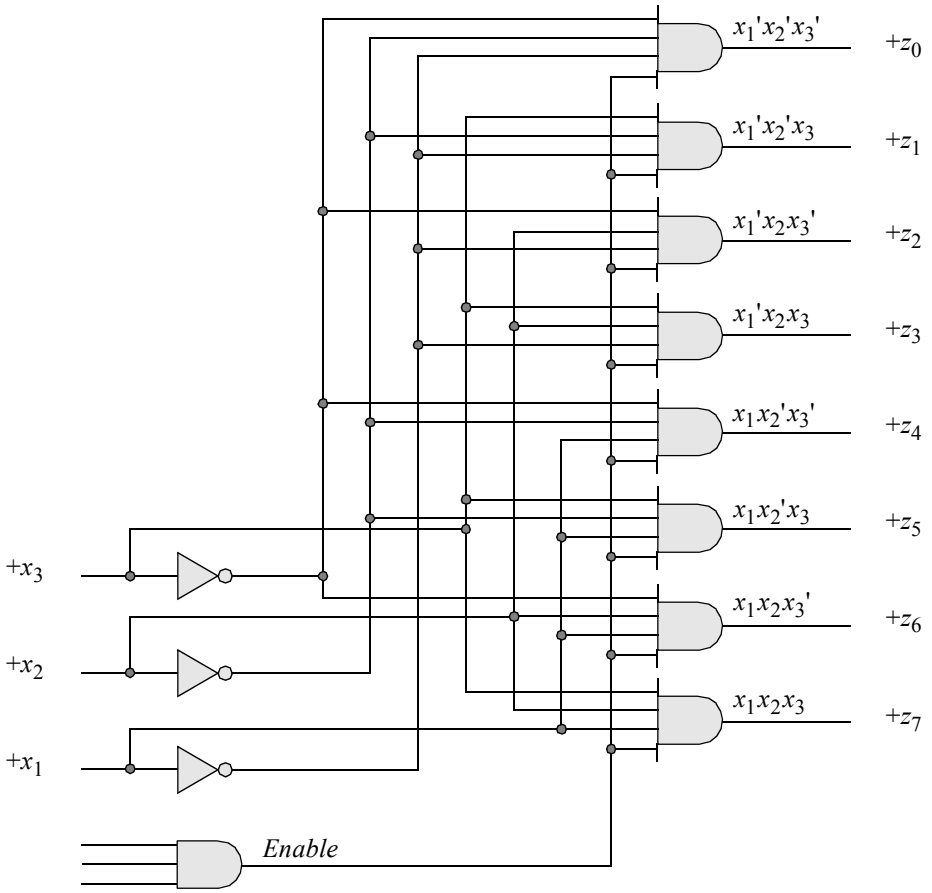


Figure 1.9 Internal logic for the binary-to-octal decoder of Figure 1.8.

Encoders An encoder is a macro logic circuit with n mutually exclusive inputs and m binary outputs, where $n \leq 2^m$. The inputs are mutually exclusive to prevent errors from appearing on the outputs. The outputs generate a binary code that corresponds to the active input value. The function of an encoder can be considered to be the inverse of a decoder; that is, the mutually exclusive inputs are encoded into a corresponding binary number.

A general block diagram for an n:m encoder is shown in Figure 1.10. An encoder is also referred to as a code converter. In the label of Figure 1.10, X corresponds to the

input code and Y corresponds to the output code. The general qualifying label X/Y is replaced by the input and output codes, respectively, such as, OCT/BIN for an octal-to-binary code converter. Only one input x_i is asserted at a time. The decimal value of x_i is encoded as a binary number which is specified by the m outputs.

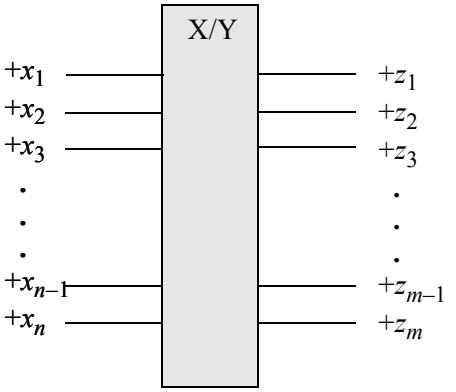


Figure 1.10 An n:m encoder or code converter.

An 8:3 octal-to-binary encoder is shown in Figure 1.11. Although there are 2^8 possible input combinations of eight variables, only eight combinations are valid. The eight inputs each generate a unique octal code word in binary. If the outputs are to be enabled, then the gating can occur at the output gates.

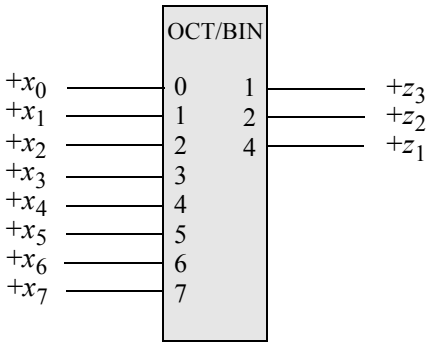


Figure 1.11 An octal-to-binary encoder.

The low-order output z_3 is asserted when one of the following inputs are active: x_1, x_3, x_5, or x_7. Output z_2 is asserted when one of the following inputs are active: x_2, x_3, x_6,

or x_7. Output z_1 is asserted when one of the following inputs are active: x_4, x_5, x_6, or x_7. The encoder can be implemented with OR gates whose inputs are established from Equation 1.3 and Figure 1.12.

$$\begin{aligned} z_3 &= x_1 + x_3 + x_5 + x_7 \\ z_2 &= x_2 + x_3 + x_6 + x_7 \\ z_1 &= x_4 + x_5 + x_6 + x_7 \end{aligned} \tag{1.3}$$

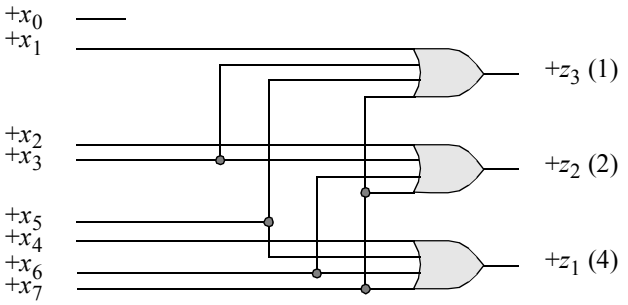


Figure 1.12 Logic diagram for an 8:3 encoder.

Priority encoder It was stated previously that encoder inputs are mutually exclusive. There may be situations, however, where more than one input can be active at a time. Then a priority must be established to select and encode a particular input. This is referred to as a *priority encoder*.

Usually the input with the highest valued subscript is selected as highest priority for encoding. Thus, if x_i and x_j are active simultaneously and $i < j$, then x_j has priority over x_i. The truth table for an octal-to-binary priority encoder is shown in Table 1.7. The outputs $z_1z_2z_3$ generate a binary number that is equivalent to the highest priority input. If $x_3 = 1$, the state of x_0, x_1, and x_2 is irrelevant ("don't care") and the output is the binary number 011.

Comparators A comparator is a logic macro circuit that compares the magnitude of two n-bit binary numbers X_1 and X_2. Therefore, there are $2n$ inputs and three outputs that indicate the relative magnitude of the two numbers. The outputs are mutually exclusive, specifying $X_1 < X_2$, $X_1 = X_2$, or $X_1 > X_2$. Figure 1.13 shows a general block diagram of a comparator.

If two or more comparators are connected in cascade, then three additional inputs are required for each comparator. These additional inputs indicate the relative magnitude of the previous lower-order comparator inputs and specify $X_1 < X_2$, $X_1 = X_2$, or $X_1 > X_2$ for the previous stage. Cascading comparators usually apply only to commercially available comparator integrated circuits.

Table 1.7 Octal-to-Binary Priority Encoder

Inputs								Outputs		
x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	z_1	z_2	z_3
1	0	0	0	0	0	0	0	0	0	0
–	**1**	0	0	0	0	0	0	0	0	1
–	–	**1**	0	0	0	0	0	0	1	0
–	–	–	**1**	0	0	0	0	0	1	1
–	–	–	–	**1**	0	0	0	1	0	0
–	–	–	–	–	**1**	0	0	1	0	1
–	–	–	–	–	–	**1**	0	1	1	0
–	–	–	–	–	–	–	**1**	1	1	1

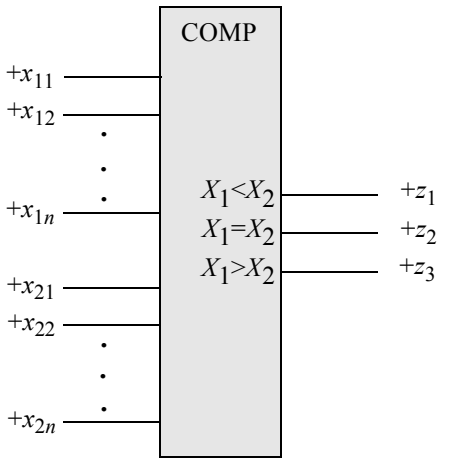


Figure 1.13 General block diagram of a comparator.

Designing the hardware for a comparator is relatively straightforward — it consists of AND gates, OR gates, and exclusive-NOR circuits as shown in Equation 1.4. An alternative approach which may be used to minimize the amount of hardware is to eliminate the equation for $X_1 = X_2$ and replace it with Equation 1.5. That is, if X_1 is neither less nor greater than X_2, then X_1 must equal X_2.

$$(X_1 < X_2) = x_{11}'x_{21} + (x_{11} \oplus x_{21})'x_{12}'x_{22} + (x_{11} \oplus x_{21})'(x_{12} \oplus x_{22})'x_{13}'x_{23}$$

$$(X_1 = X_2) = (x_{11} \oplus x_{21})'(x_{12} \oplus x_{22})'(x_{13} \oplus x_{23})'$$

$$(X_1 > X_2) = x_{11}x_{21}' + (x_{11} \oplus x_{21})'x_{12}x_{22}' + (x_{11} \oplus x_{21})'(x_{12} \oplus x_{22})'x_{13}x_{23}' \qquad (1.4)$$

$$(X_1 = X_2) \text{ if } (X_1 < X_2)' \text{ AND } (X_1 > X_2)' \tag{1.5}$$

1.1.4 Procedural Flow Control

Procedural flow control statements modify the flow in a behavior by selecting branch options, repeating certain activities, selecting a parallel activity, or terminating an activity. The activity can occur in sequential blocks or in parallel blocks.

begin . . . end The **begin** . . . **end** keywords are used to group multiple statements into sequential blocks. The statements in a sequential block execute in sequence; that is, a statement does not execute until the preceding statement has executed, except for nonblocking statements. If there is only one procedural statement in the block, then the **begin** . . . **end** keywords may be omitted.

disable The **disable** statement terminates a named block of procedural statements or a task and transfers control to the statement immediately following the block or task. The **disable** statement can also be used to exit a loop.

for The keyword **for** is used to specify a loop. The **for** loop repeats the execution of a procedural statement or a block of procedural statements a specified number of times. The **for** loop is used when there is a specified beginning and end to the loop. The format and function of a **for** loop is similar to the **for** loop used in the C programming language. The parentheses following the keyword **for** contain three expressions separated by semicolons, as shown below.

```
for (register initialization; test condition; update register control variable)
    procedural statement or block of procedural statements
```

forever The **forever** loop statement executes the procedural statements continuously. The loop is primarily used for timing control constructs, such as clock pulse generation. The **forever** procedural statement must be contained within an **initial** or an **always** block. In order to exit the loop, the **disable** statement may be used to prematurely terminate the procedural statements. An **always** statement executes at the beginning of simulation; the **forever** statement executes only when it is encountered in a procedural block.

if . . . else These keywords are used as conditional statements to alter the flow of activity through a behavioral module. They permit a choice of alternative paths based upon a Boolean value obtained from a condition. The syntax is shown below.

```
if (condition)
    {procedural statement 1}
else
    {procedural statement 2}
```

If the result of the *condition* is true, then procedural statement 1 is executed; otherwise, procedural statement 2 is executed. The procedural statement following the **if** and **else** statements can be a single procedural statement or a block of procedural statements. Two uses for the **if . . . else** statement are to model a multiplexer or decode an instruction register operation code to select alternative paths depending on the instruction. The **if** statement can be nested to provide several alternative paths to execute procedural statements as shown in the syntax below for nested **if** statements.

if (condition 1)
    {procedural statement 1}
else if (condition 2)
    {procedural statement 2}
else if (condition 3)
    {procedural statement 3}
else
    {procedural statement 4)

repeat The **repeat** keyword is used to execute a loop a fixed number of times as specified by a constant contained within parentheses following the **repeat** keyword. The loop can be a single statement or a block of statements contained within **begin . . . end** keywords. The syntax is shown below.

repeat (expression)
    statement or block of statements

When the activity flow reaches the **repeat** construct, the expression in parentheses is evaluated to determine the number of times that the loop is to be executed. The *expression* can be a constant, a variable, or a signal value. If the expression evaluates to **x** or **z**, then the value is treated as 0 and the loop is not executed.

while The **while** statement executes a statement or a block of statements while an expression is true. The syntax is shown below.

while (expression) statement

The expression is evaluated and a Boolean value, either true (a logical 1) or false (a logical 0) is returned. If the expression is true, then the procedural statement or block of statements is executed. The **while** loop executes until the expression becomes false, at which time the loop is exited and the next sequential statement is executed. If the expression is false when the loop is entered, then the procedural statement is not executed. If the value returned is **x** or **z**, then the value is treated as false. An example of the **while** statement is shown below where the initial count = 0.

```
while (count < 16)
begin
      count = count + 1;
end
```

1.1.5 Net Data Types

Verilog defines two data types: nets and registers. These predefined data types are used to connect logical elements and to provide storage. A net is a physical wire or group of wires connecting hardware elements in a module or between modules.

An example of net data types is shown in Figure 1.14, where five internal nets are defined: *net1*, *net2*, *net3*, *net4*, and *net5*. The value of *net1* is determined by the inputs to the *and1* gate represented by the term x_1x_2', where x_2 is active low; the value of *net2* is determined by the inputs to the *and2* gate represented by the term $x_1'x_2$, where x_1 is active low; the value of *net3* is determined by the input to the inverter represented by the term x_3', where x_3 is active low. The equations for outputs z_1 and z_2 are listed in Equation 1.6.

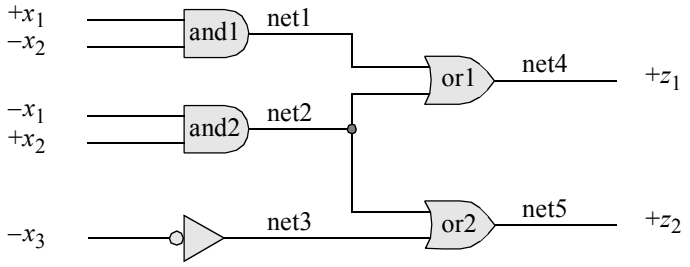


Figure 1.14 A logic diagram showing single-wire nets and one multiple-wire net.

$$z_1 = x_1x_2' + x_1'x_2$$
$$z_2 = x_1'x_2 + x_3 \tag{1.6}$$

1.1.6 Register Data Types

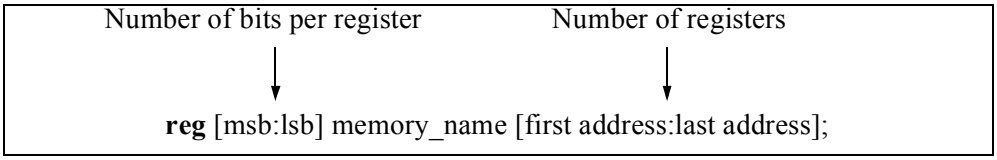
A register data type represents a variable that can retain a value. Verilog registers are similar in function to hardware registers, but are conceptually different. Hardware registers are synthesized with storage elements such as *D* flip-flops, *JK* flip-flops, and *SR* latches. Verilog registers are an abstract representation of hardware registers and are declared as **reg**.

The default size of a register is 1-bit; however, a larger width can be specified in the declaration. The general syntax to declare a width of more than 1-bit is as follows:

reg [most significant bit:least significant bit] register_name.

To declare a one-byte register called data_register is **reg** [7:0] data_register.

Memories Memories can be represented in Verilog by an array of registers and are declared using a **reg** data type as follows:



A 32-word register with one byte per word would be declared as follows:

reg [7:0] memory_name [0:31];

An array can have only two dimensions. Memories must be declared as **reg** data types, not as **wire** data types. A register can be assigned a value using one statement, as shown below. Register *buff_reg* is assigned the 16-bit hexadecimal value of 7ab5, which equates to the binary value of $0111\ \ 1010\ \ 1011\ \ 0101_2$.

```
reg [15:0] buff_reg;
buff_reg = 16'h7ab5;
```

Values can also be stored in memories by assigning a value to each word individually, as shown below for an instruction cache of eight registers with eight bits per register.

reg [7:0] instr_cache [0:7];

```
instr_cache [0] =   8'h08;
instr_cache [1] =   8'h09;
instr_cache [2] =   8'h0a;
instr_cache [3] =   8'h0b;
instr_cache [4] =   8'h0c;
instr_cache [5] =   8'h0d;
instr_cache [6] =   8'h0e;
instr_cache [7] =   8'h0f;
```

1.2 Expressions

Expressions consist of operands and operators, which are the basis of Verilog HDL. The result of a right-hand side expression can be assigned to a left-hand side net variable or register variable using the keyword **assign**. The value of an expression is determined from the combined operations on the operands. An expression can consist of a single operand or two or more operands in conjunction with one or more operators. The result of an expression is represented by one or more bits. Examples of expressions are as follows, where the symbol & indicates an AND operation and the symbol | indicates an OR operation:

```
assign z1 = x1 & x2 & x3;
assign z1 = x1 | x2 | x3;
assign cout = (a & cin) | (b & cin) | (a & b);
```

1.2.1 Operands

Operands can be any of the data types listed in Table 1.8.

Table 1.8 Operands

Operands	Comments
Constant	Signed or unsigned
Parameter	Similar to a constant
Net	Scalar or vector
Register	Scalar or vector
Bit-select	One bit from a vector
Part-select	Contiguous bits of a vector
Memory element	One word of a memory

Constant Constants can be signed or unsigned. A decimal integer is treated as a signed number. An integer that is specified by a base is interpreted as an unsigned number. Examples of both types are shown in Table 1.9.

Table 1.9 Signed and Unsigned Constants

Constant	Comments
127	Signed decimal: Value = 8-bit binary vector: 0111_1111
–1	Signed decimal: Value = 8-bit binary vector: 1111_1111
–128	Signed decimal: Value = 8-bit binary vector: 1000_0000
4'b1110	Binary base: Value = unsigned decimal 14
8'b0011_1010	Binary base: Value = unsigned decimal 58
16'h1A3C	Hexadecimal base: Value = unsigned decimal 6716
16'hBCDE	Hexadecimal base: Value = unsigned decimal 48,350
9'o536	Octal base: Value = unsigned decimal 350
–22	Signed decimal: Value = 8-bit binary vector: 1110_1010
–9'o352	Octal base: Value = 8-bit binary vector: 1110_1010 = unsigned decimal 234

The last two entries in Table 1.9 both evaluate to the same bit configuration, but represent different decimal values. The number -22_{10} is a signed decimal value; the number –9'o352 is treated as an unsigned number with a decimal value of 234_{10}.

Parameter A parameter is similar to a constant and is declared by the keyword **parameter**. Parameter statements assign values to constants; the values cannot be changed during simulation. Examples of parameters are shown in Table 1.10.

Table 1.10 Examples of Parameters

Examples	Comments
parameter width = 8	Defines a bus width of 8 bits
parameter width = 16, depth = 512	Defines a memory with two bytes per word and 512 words
parameter out_port = 8	Defines an output port with an address of 8

Parameters are useful in defining the width of a bus. For example, the adder shown in Figure 1.15 contains two 8-bit vector inputs *a* and *b* and one scalar input *cin*. There is also one 9-bit vector output *sum* comprised of an 8-bit result and a scalar carry-out. The Verilog line of code shown below defines a bus width of eight bits. Wherever *width* appears in the code, it is replaced by the value eight.

```
parameter  width = 8;
```

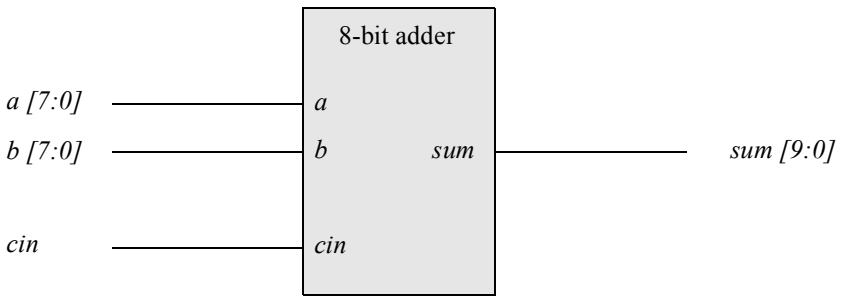


Figure 1.15 Eight-bit adder to illustrate the use of a **parameter** statement.

1.2.2 Operators

Verilog HDL contains a profuse set of operators that perform various operations on different types of data to yield results on nets and registers. Some operators are similar to those used in the C programming language. Table 1.11 lists the categories of operators in order of precedence, from highest to lowest.

Table 1.11 Verilog HDL Operators and Symbols

Operator type	Operator Symbol	Operation	Number of Operands
Arithmetic	+	Add	Two or one
	-	Subtract	Two or one
	*	Multiply	Two
	/	Divide	Two
	%	Modulus	Two
Logical	&&	Logical AND	Two
	||	Logical OR	Two
	!	Logical negation	One
Relational	>	Greater than	Two
	<	Less than	Two
	>=	Greater than or equal	Two
	<=	Less than or equal	Two
Equality	= =	Logical equality	Two
	! =	Logical inequality	Two
	= = =	Case equality	Two
	! = =	Case inequality	Two
Bitwise	&	AND	Two
	|	OR	Two
	~	Negation	One
	^	Exclusive-OR	Two
	^ ~ or ~ ^	Exclusive-NOR	Two
Reduction	&	AND	One
	~ &	NAND	One
	|	OR	One
	~ |	NOR	One
	^	Exclusive-OR	One
	~ ^ or ^ ~	Exclusive-NOR	One
Shift	<<	Left shift	One
	>>	Right shift	One
Conditional	? :	Conditional	Three
Concatenation	{ }	Concatenation	Two or more
Replication	{{ }}	Replication	Two or more

Arithmetic Arithmetic operations are performed on one (unary) operand or two (binary) operands in the following radices: binary, octal, decimal, or hexadecimal. The result of an arithmetic operation is interpreted as an unsigned value or as a signed value in 2s complement representation on both scalar and vector nets and registers. The operands shown in Table 1.12 are used for the operations of addition, subtraction, multiplication, and division.

Table 1.12 Operands Used for Arithmetic Operations

	Addition		Subtraction		Multiplication		Division
	Augend		Minuend		Multiplicand		Dividend
+)	Addend	–)	Subtrahend	×	Multiplier	÷)	Divisor
	Sum		Difference		Product		Quotient, Remainder

The unary + and – operators change the sign of the operand and have higher precedence than the binary + and – operators. Examples of unary operators are shown below.

$+45$ (Positive 45_{10})
-72 (Negative 72_{10})

Unary operators treat net and register operands as unsigned values, and treat real and integer operands as signed values.

The binary add operator performs unsigned and signed addition on two operands. Register and net operands are treated as unsigned operands; thus, a value of

$$1111\_1111\_1111\_1111_2$$

stored in a register has a value of $65{,}535_{10}$ unsigned, not -1_{10} signed. Real and integer operands are treated as signed operands; thus, a value of

$$1111\_1110\_1010\_0111_2$$

stored in an integer register has a value of -345_{10} signed, not $65{,}191_{10}$ unsigned. The width of the result of an arithmetic operation is determined by the width of the largest operand.

Logical There are three logical operators: the binary logical AND operator (&&), the binary logical OR operator (||), and the unary logical negation operator (!). Logical operators evaluate to a logical 1 (true), a logical 0 (false), or an **x** (ambiguous). If a logical operation returns a nonzero value, then it is treated as a logical 1 (true); if a bit in an operand is **x** or **z**, then it is ambiguous and is normally treated as a false condition.

Let a and b be two 4-bit operands, where $a = 0110$ and $b = 1100$. Let z_1, z_2, and z_3 be the outputs of the logical operations shown below.

$z_1 = a \;\&\&\; b$
$z_2 = a \;||\; b$
$z_3 = !\, a$

Therefore, the operation $z_1 = a \;\&\&\; b$ yields a value of $z_1 = 1$ because both a and b are nonzero. If a vector operand is nonzero, then it treated as a 1 (true). Output z_2 is also equal to 1 for the expression $z_2 = a \;||\; b$. Output z_3 is equal to 0 because a is true.

Now let $a = 0101$ and $b = 0000$. Thus, $z_1 = a \;\&\&\; b = 1 \;\&\&\; 0 = 0$ because a is true and b is false. Output z_2, however, is equal to 1 because $z_2 = a \;||\; b = 1 \;||\; 0 = 1$. In a similar manner, $z_3 = !a = !1 = 0$, because a is true.

As a final example, let $a = 0000$ and $b = 0000$; that is, both variables are false. Therefore, $z_1 = a \;\&\&\; b = 0 \;\&\&\; 0 = 0$; $z_2 = a \;||\; b = 0 \;||\; 0 = 0$; $z_3 = !a = !0 = 1$. If a bit in either operand is **x**, then the result of a logical operation is **x**. Also, ! **x** is **x**.

Relational Relational operators compare operands and return a Boolean result, either 1 (true) or 0 (false) indicating the relationship between the two operands. There are four relational operators as follows: greater than (>), less than (<), greater than or equal (>=), and less than or equal (<=). These operators function the same as identical operators in the C programming language.

If the relationship is true, then the result is 1; if the relationship is false, then the result is 0. Net or register operands are treated as unsigned values; real or integer operands are treated as signed values. An **x** or **z** in any operand returns a result of **x**. When the operands are of unequal size, the smaller operand is zero-extended to the left. Examples are shown below of relational operators, where the identifier *gt* means greater than, *lt* means less than, *gte* means greater than or equal, and *lte* means less than or equal when comparing operand *a* to operand *b*.

$a = 0110,\ b = 1100,$ $gt = 0,\ lt = 1,\ gte = 0,\ lte = 1$
a $= 0101,\ b = 0000,$ $gt = 1,\ lt = 0,\ gte = 1,\ lte = 0$
$a = 1000,\ b = 1001,$ $gt = 0,\ lt = 1,\ gte = 0,\ lte = 1$
$a = 0000,\ b = 0000,$ $gt = 0,\ lt = 0,\ gte = 1,\ lte = 1$
$a = 1111,\ b = 1111,$ $gt = 0,\ lt = 0,\ gte = 1,\ lte = 1$

Equality There are four equality operators: logical equality (==), logical inequality (!=), case equality (===), and case inequality (!==).

Logical equality is used in expressions to determine if two values are identical. The result of the comparison is 1 if the two operands are equal, and 0 if they are not equal. The *logical inequality* operator is used to determine if two operands are unequal. A 1 is returned if the operands are unequal; otherwise a 0 is returned. If the result of the comparison is ambiguous for logical equality or logical inequality, then a value of **x** is returned. An **x** or **z** in either operand will return a value of **x**. If the operands are nets or registers, they are treated as unsigned values; real or integer operands are treated as signed values, but are compared as though they were unsigned operands.

The *case equality* operator compares both operands on a bit-by-bit basis, including **x** and **z**. The result is 1 if both operands are identical in the same bit positions, including those bit positions containing an **x** or a **z**. The *case inequality* operator is used to determine if two operands are unequal by comparing them on a bit-by-bit basis, including those bit positions that contain **x** or **z**.

Examples of the equality operators are shown below, where the 4-bit variables are x_1, x_2, x_3, x_4, and x_5. The outputs are z_1 (logical equality), z_2 (logical inequality), z_3 (case equality), and z_4 (case inequality).

$x_1 = 1000,\ x_2 = 1101,\ x_3 = 01xz,\ x_4 = 01xz,\ x_5 = x1xx$
$z_1 = 0,\ z_2 = 1,\ z_3 = 1,\ z_4 = 1$

$x_1 = 1011,\ x_2 = 1011,\ x_3 = x1xz,\ x_4 = x1xz,\ x_5 = 11xx$
$z_1 = 1,\ z_2 = 1,\ z_3 = 1,\ z_4 = 1$

$x_1 = 1100,\ x_2 = 0101,\ x_3 = x01z,\ x_4 = 11xz,\ x_5 = 11xx$
$z_1 = 0,\ z_2 = 1,\ z_3 = 0,\ z_4 = 1$

Referring to the above outputs for the first set of inputs, the logical equality (z_1) of x_1 and x_2 is false because the operands are unequal. The logical inequality (z_2) of x_2 and x_3 is true. The case equality (z_3) of inputs x_3 and x_4 is 1 because both operands are identical in all bit positions, including the **x** and **z** bits. The case inequality (z_4) of inputs x_4 and x_5 is also 1 because the operands differ in the high-order and low-order bit positions.

Bitwise The bitwise operators are: AND (&), OR (|), negation (~), exclusive-OR (^), and exclusive-NOR (^~ or ~^). The bitwise operators perform logical operations on the operands on a bit-by-bit basis and produce a vector result. Except for negation, each bit in one operand is associated with the corresponding bit in the other operand. If one operand is shorter, then it is zero-extended to the left to match the length of the longer operand.

The *bitwise AND* operator performs the AND function on two operands on a bit-by-bit basis as shown in the following example:

```
    1 0 1 1 0 1 1 0
&)  1 1 0 1 0 1 0 1
    ---------------
    1 0 0 1 0 1 0 0
```

The *bitwise OR* operator performs the OR function on the two operands on a bit-by-bit basis as shown in the following example:

```
    1 0 1 1 0 1 1 0
|)  1 1 0 1 0 1 0 1
    ---------------
    1 1 1 1 0 1 1 1
```

The *bitwise negation* operator performs the negation function on one operand on a bit-by-bit basis. Each bit in the operand is inverted as shown in the following example:

```
~)  1 1 0 1 0 1 0 1
    ---------------
    0 0 1 0 1 0 1 0
```

The *bitwise exclusive-OR* operator performs the exclusive-OR function on two operands on a bit-by-bit basis as shown in the following example:

```
    1 0 1 1 0 1 1 0
^ ) 1 1 0 1 0 1 0 1
    ---------------
    0 1 1 0 0 0 1 1
```

The *bitwise exclusive-NOR* operator performs the exclusive-NOR function on two operands on a bit-by-bit basis as shown in the following example:

```
     1 0 1 1 0 1 1 0
^~ ) 1 1 0 1 0 1 0 1
     ---------------
     1 0 0 1 1 1 0 0
```

Bitwise operators perform operations on operands on a bit-by-bit basis and produce a vector result. This is in contrast to logical operators, which perform operations on operands in such a way that the truth or falsity of the result is determined by the truth or falsity of the operands.

The logical AND operator returns a value of 1 (true) only if both operands are nonzero (true); otherwise, it returns a value of 0 (false). If the result is ambiguous, it returns a value of **x**. The logical OR operator returns a value of 1 (true) if either or both operands are true; otherwise, it returns a value of 0. The logical negation operator returns a value of 1 (true) if the operand has a value of zero and a value of 0 (false) if the operand is nonzero. Examples of the five bitwise operators are shown below. The logical negation operator performs the operation on operand *a*.

```
a = 11000011,
b = 10011001,

and_rslt  = 10000001,
or_rslt   = 11011011,
neg_rslt  = 00111100,
xor_rslt  = 01011010,
xnor_rslt = 10100101
```

```
a = 01001111,
b = 11011001,

and_rslt  = 01001001,
or_rslt   = 11011111,
neg_rslt  = 10110000,
xor_rslt  = 10010110,
xnor_rslt = 01101001
```

```
a = 10010011,
b = 11011001,

and_rslt  = 10010001,
or_rslt   = 11011011,
neg_rslt  = 01101100,
xor_rslt  = 01001010,
xnor_rslt =10110101
```

```
a  = 11001111,
b  = 11011001,

and_rslt  = 11001001,
or_rslt   = 11011111,
neg_rslt  = 00110000,
xor_rslt  = 00010110,
xnor_rslt = 11101001
```

Reduction The reduction operators are: AND (&), NAND (~&), OR (|), NOR (~|), exclusive-OR (^), and exclusive-NOR (^~ or ~^). Reduction operators are unary operators; that is, they operate on a single vector and produce a single-bit result. If any bit of the operand is **x** or **z**, the result is **x**. Reduction operators perform their respective operations on a bit-by-bit basis.

For the *reduction AND* operator, if any bit in the operand is 0, then the result is 0; otherwise, the result is 1. For example, let x_1 be the vector shown below.

1	1	1	0	1	0	1	1

The reduction AND ($\&x_1$) operation is equivalent to the following operation:

1 & 1 & 1 & 0 & 1 & 0 & 1 & 1

which returns a result of 1'b0.

For the *reduction NAND* operator, if any bit in the operand is 0, then the result is 1; otherwise, the result is 0. For a vector x_1, the reduction NAND ($\sim\&x_1$) is the inverse of the reduction AND operator.

For the *reduction OR* operator, if any bit in the operand is 1, then the result is 1; otherwise, the result is 0. For example, let x_1 be the vector shown below.

1	1	1	0	1	0	1	1

The reduction OR ($|x_1$) operation is equivalent to the following operation:

1 | 1 | 1 | 0 | 1 | 0 | 1 | 1

which returns a result of 1'b1.

For the *reduction NOR* operator, if any bit in the operand is 1, then the result is 0; otherwise, the result is 1. For a vector x_1, the reduction NOR ($\sim|x_1$) is the inverse of the reduction OR operator.

For the *exclusive-OR* operator, if there are an even number of 1s in the operand, then the result is 0; otherwise, the result is 1. For example, let x_1 be the vector shown below.

1	1	1	0	1	0	1	1

The reduction exclusive-OR ($\wedge x_1$) operation is equivalent to the following operation:

1 ^ 1 ^ 1 ^ 0 ^ 1 ^ 0 ^ 1 ^ 1

which returns a result of 1'b0. The reduction exclusive-OR operator can be used as an even parity generator.

For the *exclusive-NOR* operator, if there are an odd number of 1s in the operand, then the result is 0; otherwise, the result is 1. For a vector x_1, the reduction exclusive-NOR ($\wedge\sim x_1$) is the inverse of the reduction exclusive-OR operator. The reduction exclusive-NOR operator can be used as an odd parity generator.

Shift The shift operators shift a single vector operand left or right a specified number of bit positions. These are logical shift operations, not algebraic; that is, as bits are shifted left or right, zeroes fill in the vacated bit positions. The bits shifted out of the operand are lost; they do not rotate to the high-order or low-order bit positions of the shifted operand. If the shift amount evaluates to **x** or **z**, then the result of the operation is **x**. There are two shift operators, as shown below. The value in parentheses is the number of bits that the operand is shifted.

<< (Left-shift amount)
>> (Right-shift amount)

When an operand is shifted left, this is equivalent to a multiply-by-two operation for each bit position shifted. When an operand is shifted right, this is equivalent to a divide-by-two operation for each bit position shifted. The shift operators are useful to model the sequential add-shift multiplication algorithm and the sequential shift-subtract division algorithm. Examples of shift left and shift right operations are shown below for 8-bit operands. Operand *a_reg* is shifted left three bits with the low-order bits filled with zeroes. Operand *b_reg* is shifted right two bits with the high-order bits filled with zeroes.

```
 a_reg = 00000010,   b_reg = 00001000,          //shift a_reg left 3
rslt_a = 00010000,   rslt_b = 00000010          //shift b_reg right 2

a_reg = 00000110,   b_reg = 00011000,          //shift a_reg left 3
rslt_a = 00110000,   rslt_b = 00000110          //shift b_reg right 2

a_reg = 00001111,   b_reg = 00111000,          //shift a_reg left 3
rslt_a = 01111000,   rslt_b = 00001110          //shift b_reg right 2

a_reg = 11100000,   b_reg = 00000011,          //shift a_reg left 3
rslt_a = 00000000,   rslt_b = 00000000          //shift b_reg right 2
```

Conditional The conditional operator (**? :**) has three operands, as shown in the syntax below. The *conditional_expression* is evaluated. If the result is true (1), then the *true_expression* is evaluated; if the result is false (0), then the *false_expression* is evaluated.

conditional_expression **?** true_expression **:** false_expression;

The conditional operator can be used when one of two expressions is to be selected. For example, in the statement below, if x_1 is greater than or equal to x_2, then z_1 is assigned the value of x_3; if x_1 is less than x_2, then z_1 is assigned the value of x_4.

```
z1 = (x1 >= x2) ? x3 : x4;
```

If the operands have different lengths, then the shorter operand is zero-extended on the left. Since the conditional operator selects one of two values, depending on the result of the conditional_expression evaluation, the operator can be used in place of the **if** . . . **else** construct. The conditional operator is ideally suited to model a 2:1 multiplexer. Conditional operators can be nested; that is, each true_expression and false_expression can be a conditional operation. This is useful for modeling a 4:1 multiplexer.

```
conditional_expression ? (cond_expr1 ? true_expr1 : false_expr1)
                       : (cond_expr2 ? true_expr2 : false_expr2);
```

Concatenation The concatenation operator ({ }) forms a single operand from two or more operands by joining the different operands in sequence separated by commas. The operands to be appended are contained within braces. The size of the operands must be known before concatenation takes place.

The examples below show the concatenation of scalars and vectors of different sizes. Outputs z_1, z_2, z_3, and z_4 are ten bits in length.

```
z1, z2, z3, and z4 are 10 bits in length.

a = 11, b = 001, c = 1100, d = 1

z1 = 0000_11_1100          //z1 = {a, c}
z2 = 00000_001_11          //z2 = {b, a}
z3 = 0_1100_001_11         //z3 = {c, b, a}
z4 = 11_001_1100_1         //z4 = {a, b, c, d}
```

Replication Replication is a means of performing repetitive concatenation. Replication specifies the number of times to duplicate the expressions within the innermost braces. The syntax is shown below together with examples of replication.

```
{number_ of_ repetitions {expression_1, expression_2, . . . , expression_n}};
```

```
a = 11, b = 010, c = 0011,

z1 = 11_0011_11_0011,                    //z1 = {2{a, c}}
z2 = 010_0011_0111_010_0011_0111         //z2 = {2{b, c, 4'b0111}}
```

1.3 Modules and Ports

A *module* is the basic unit of design in Verilog. It describes the functional operation of some logical entity and can be a stand-alone module or a collection of modules that are instantiated into a structural module. *Instantiation* means to use one or more lower-level modules in the construction of a higher-level structural module. A module can be a logic gate, an adder, a multiplexer, a counter, or some other logical function.

A module consists of declarative text which specifies the function of the module using Verilog constructs; that is, a Verilog module is a software representation of the physical hardware structure and behavior. The declaration of a module is indicated by the keyword **module** and is always terminated by the keyword **endmodule**.

Verilog has predefined logical elements called *primitives*. These built-in primitives are structural elements that can be instantiated into a larger design to form a more complex structure. Examples are: **and**, **or**, **xor**, and **not**. Built-in primitives are discussed in more detail in Section 1.4.

Modules contain *ports* which allow communication with the external environment or other modules. For example, the logic diagram for the full adder of Figure 1.16 has input ports *a*, *b*, and *cin* and output ports *sum* and *cout*. The general structure and syntax of a module is shown in Figure 1.17. An AND gate can be defined as shown in the module of Figure 1.18, where the input ports are x_1 and x_2 and the output port is z_1.

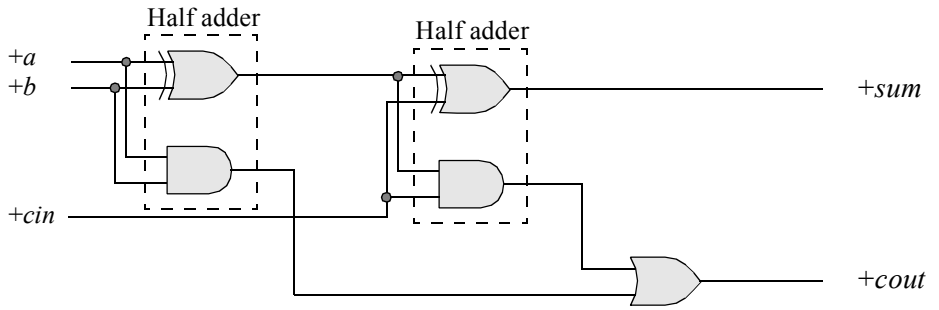


Figure 1.16 Logic diagram for a full adder.

```
module <module name> (port list);
   declarations
      reg, wire, parameter,
      input, output, . . .
      . . .
   <module internals>
      statements
      initial, always, module instantiation, . . .
      . . .
endmodule
```

Figure 1.17 General structure of a Verilog module.

```
//dataflow and gate with two inputs
module and2 (x1, x2, z1);

input x1, x2;
output z1;

wire x1, x2;
wire z1;

assign z1 = x1 & x2;

endmodule
```

Figure 1.18 Verilog module for an AND gate with two inputs.

A Verilog module defines the information that describes the relationship between the inputs and outputs of a logic circuit. A structural module will have one or more instantiations of other modules or logic primitives. In Figure 1.18, the first line is a comment, indicated by (//). In the second line, *and2* is the module name; this is followed by left and right parentheses containing the module ports, which is followed by a semicolon. The inputs and outputs are defined by the keywords **input** and **output**. The ports are declared as **wire** in this dataflow module. Dataflow modeling is covered in detail in Section 1.6. The keyword **assign** describes the behavior of the circuit. Output z_1 is *assigned* the value of x_1 ANDed (&) with x_2.

1.3.1 Designing a Test Bench for Simulation

This section describes the techniques for writing test benches in Verilog HDL. When a Verilog module is finished, it must be tested to ensure that it operates according to the machine specifications. The functionality of the module can be tested by applying stimulus to the inputs and checking the outputs. The test bench will display the inputs and outputs in a radix (binary, octal, hexadecimal, or decimal).

The test bench contains an instantiation of the unit under test and Verilog code to generate input stimulus and to monitor and display the response to the stimulus. Figure 1.19 shows a simple test bench to test the 2-input AND gate of Figure 1.18. Line 1 is a comment indicating that the module is a test bench for a 2-input AND gate. Line 2 contains the keyword **module** followed by the module name, which includes *tb* indicating a test bench module. The name of the module and the name of the module under test are the same for ease of cross-referencing.

Line 4 specifies that the inputs are **reg** type variables; that is, they contain their values until they are assigned new values. Outputs are assigned as type **wire** in test benches. Output nets are driven by the output ports of the module under test. Line 8 contains an **initial** statement, which executes only once.

Verilog provides a means to monitor a signal when its value changes. This is accomplished by the **$monitor** task. The **$monitor** continuously monitors the values of the variables indicated in the parameter list that is enclosed in parentheses. It will display the value of the variables whenever a variable changes state. The quoted string within the task is printed and specifies that the variables are to be shown in binary (%b). The **$monitor** is invoked only once. Line 12 is a second **initial** statement that allows the procedural code between the **begin** . . . **end** block statements to be executed only once.

```
 1 //and2 test bench
   module and2_tb;

   reg x1, x2;
 5 wire z1;

   //display variables
   initial
   $monitor ("x1 = %b, x2 = %b, z1 = %b", x1, x2, z1);

11 //apply input vectors
   initial
   begin
      #0    x1 = 1'b0;
            x2 = 1'b0;
16
      #10   x1 = 1'b0;
            x2 = 1'b1;

20    #10   x1 = 1'b1;
            x2 = 1'b0;

      #10   x1 = 1'b1;
            x2 = 1'b1;

26    #10   $stop;
   end

   //instantiate the module into the test bench
30 and2 inst1 (
      .x1(x1),
      .x2(x2),
      .z1(z1)
      );

   endmodule
```

Figure 1.19 Test bench for the 2-input AND gate of Figure 1.18.

Lines 14 and 15 specify that at time 0 (#0), inputs x_1 and x_2 are assigned values of 0, where 1 is the width of the value (one bit), ' is a separator, b indicates binary, and 0 is the value. Line 17 specifies that 10 time units later, the inputs change to: $x_1 = 0$ and $x_2 = 1$. This process continues until all possible values of two variables have been applied to the inputs. Simulation stops at 10 time units after the last input vector has been applied (**$stop**). The total time for simulation is 40 time units — the sum of all the time units. The time units can be specified for any duration.

Line 30 begins the instantiation of the module into the test bench. The name of the instantiation must be the same as the module under test, in this case, *and2*. This is followed by an instance name (*inst1*) followed by a left parenthesis. The $.x_1$ variable in line 31 refers to a port in the module that corresponds to a port (x_1) in the test bench. All the ports in the module under test must be listed. The keyword **endmodule** is the last line in the test bench.

The binary outputs for this test bench are shown in Figure 1.20. The output can be presented in binary (b or B), in octal (o or O), in hexadecimal (h or H), or in decimal (d or D).

The Verilog syntax will be covered in greater detail in subsequent sections. It is important at this point to concentrate on how the module under test is simulated and instantiated into the test bench.

```
x1 = 0, x2 = 0, z1 = 0
x1 = 0, x2 = 1, z1 = 0
x1 = 1, x2 = 0, z1 = 0
x1 = 1, x2 = 1, z1 = 1
```

Figure 1.20 Binary outputs for the test bench of Figure 1.19 for a 2-input AND gate.

Several different methods to generate test benches will be shown in subsequent sections. Each design in the book will be tested for correct operation by means of a test bench. Test benches provide clock pulses that are used to control the operation of a synchronous sequential machine. An **initial** statement is an ideal method to generate a waveform at discrete intervals of time for a clock pulse. The Verilog code in Figure 1.21 illustrates the necessary statements to generate clock pulses that have a duty cycle of 20%.

1.4 Built-In Primitives

Logic primitives such as **and**, **nand**, **or**, **nor**, and **not** gates, as well as **xor** (exclusive-OR), and **xnor** (exclusive_NOR) functions are part of the Verilog language and are classified as multiple-input gates. These are built-in primitives that can be instantiated into a module.

```
//generate clock pulses of 20% duty cycle
module clk_gen (clk);
output clk;
reg clk;

initial
begin
   #0    clk = 0;
   #5    clk = 1;
   #5    clk = 0;
   #20   clk = 1;
   #5    clk = 0;
   #20   clk = 1;
   #5    clk = 0;
   #10   $stop;
end
endmodule
```

Figure 1.21 Verilog code to generate clock pulses with a 20% duty cycle.

These are built-in primitive gates used to describe a net and have one or more scalar inputs, but only one scalar output. The output signal is listed first, followed by the inputs in any order. The outputs are declared as **wire;** the inputs can be declared as either **wire** or **reg**. The gates represent a combinational logic function and can be instantiated into a module, as follows, where the instance name is optional:

gate_type inst1 (output, input_1, input_2, . . . , input_*n*);

Two or more instances of the same type of gate can be specified in the same construct, as follows:

gate_type inst1 (output_1, input_11, input_12, . . . , input_1*n*),
inst2 (output_2, input_21, input_22, . . . , input_2*n*),
.
.
.
inst*m* (output_*m*, input_*m*1, input_*m*2, . . . , input_*mn*);

and This is a multiple-input built-in primitive gate that performs the AND function for a multiple-input AND gate. If any input is an **x**, then this represents an unknown logic value. If and entry is a **z**, then this represents a high impedance state, which indicates that the driver of a net is disabled or not connected. AND gates can be represented by two symbols as shown below for the AND function and the OR function.

AND gate for the AND function

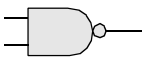
AND gate for the OR function

buf A **buf** gate is a noninverting primitive with one scalar input and one or more scalar outputs. The output terminals are listed first when instantiated; the input is listed last, as shown below. The instance name is optional.

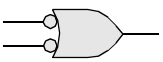
```
buf  inst1 (output, input);                              //one output
buf  inst2 (output_1, output_2, . . . , output_n, input); //multiple outputs
```

nand This is a multiple-input built-in primitive gate that operates as an AND function with a negative output. NAND gates can be represented by two symbols as shown below for the AND function and the OR function.

NAND gate for the AND function



NAND gate for the OR function



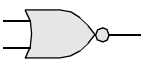
DeMorgan's theorems are associated with NAND and NOR gates and convert the complement of a sum term or a product term into a corresponding product or sum term, respectively. For every $x_1, x_2 \in B$,

(a) $(x_1 \bullet x_2)' = x_1' + x_2'$ Nand gate
(b) $(x_1 + x_2)' = x_1' \bullet x_2'$ NOR gate

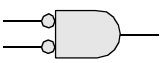
DeMorgan's laws can be generalized for any number of variables.

nor This is a multiple-input built-in primitive gate that operates as an OR function with a negative output. NOR gates can be represented by two symbols as shown below for the OR function and the AND function.

NOR gate for the OR function



NOR gate for the AND function

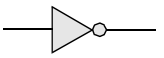


not A **not** gate is an inverting built-in primitive with one scalar input and one or more scalar outputs. The output terminals are listed first when instantiated; the input is listed last, as shown below. The instance name is optional.

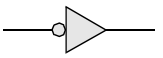
```
not  inst1 (output, input);                              //one output
not  inst2 (output_1, output_2, . . . , output_n, input); //multiple outputs
```

The NOT function can be represented by two symbols as shown below depending on the assertion levels required. The function of the inverters is identical; the low assertion is placed at the input or output for readability with associated logic.

NOT (inverter) function with low assertion output

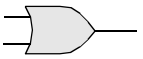


NOT (inverter) function with low assertion input



or This is a multiple-input built-in primitive gate that operates as an OR function. OR gates can be represented by two symbols as shown below for the OR function and the AND function.

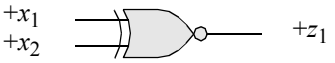
OR gate for the OR function



OR gate for the AND function

xnor This is a built-in primitive gate that functions as an exclusive-OR gate with a negative output. Exclusive-NOR gates can be represented by the symbol shown below. An exclusive-NOR gate is also called an equality function because the output is a logical 1 whenever the two inputs are equal.

Exclusive-NOR gate

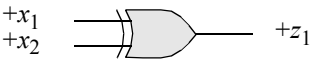


The equation for the exclusive-NOR gate shown above is

$$z_1 = (x_1 x_2) + (x_1' x_2')$$

xor This is a built-in primitive gate that functions as an exclusive-OR circuit. Exclusive-OR gates can be represented by the symbol shown below. The output of an exclusive-OR gate is a logical 1 whenever the two inputs are different.

Exclusive-OR gate



The equation for the exclusive-OR gate shown above is

$$z_1 = (x_1 x_2') + (x_1' x_2)$$

1.4.1 Built-In Primitive Design Examples

The best way to learn design methodologies using built-in primitives is by examples. Therefore, examples will be presented ranging from very simple to moderately complex. When necessary, the theory for the examples will be presented prior to the Verilog design. All examples are carried through to completion at the gate level. Nothing is left unfinished or partially designed.

Example 1.1 The Karnaugh map of Figure 1.22 will be implemented using only NOR gates in a product-of-sums format. Equation 1.7 shown the product-of-sums expression obtained from the Karnaugh map. The logic diagram is shown in Figure 1.23 which indicates the instantiation names and net names.

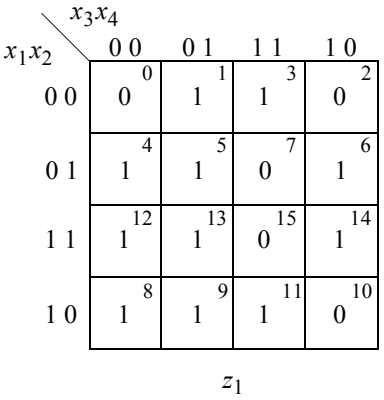


Figure 1.22 Karnaugh map for Example 1.1.

$$z_1 = (x_1 + x_2 + x_4)\,(x_2 + x_3' + x_4)\,(x_2' + x_3' + x_4') \tag{1.7}$$

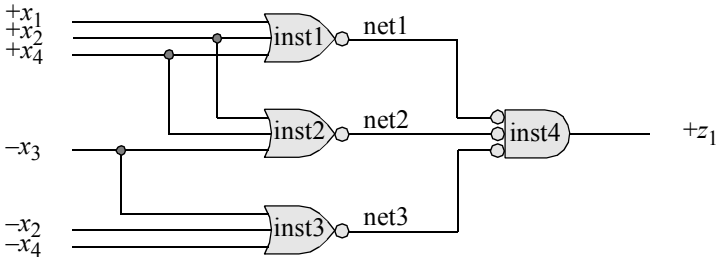


Figure 1.23 Logic diagram for Example 1.1.

The design module is shown in Figure 1.24 using NOR gate built-in primitives. The test bench is shown in Figure 1.25 using a different approach to generate all 16 combinations of the four inputs. Several new modeling constructs are shown in the test bench. Since there are four inputs to the circuit, all 16 combinations of four variables must be applied to the circuit. This is accomplished by a **for** loop statement, which is similar in construction to a **for** loop in the C programming language.

```
//logic diagram using built-in primitives
module log_eqn_pos5 (x1, x2, x3, x4, z1);

input x1, x2, x3, x4;
output z1;

//instantiate the nor built-in primitives
nor   inst1 (net1, x1, x2, x4);
nor   inst2 (net2, x2, x4, ~x3);
nor   inst3 (net3, ~x3, ~x2, ~x4);
nor   inst4 (z1, net1, net2, net3);

endmodule
```

Figure 1.24 Module for the product-of-sums logic diagram of Figure 1.23.

```
//test bench for log_eqn_pos5
module log_eqn_pos5_tb;

reg x1, x2, x3, x4;
wire z1;

//apply input vectors
initial
begin: apply_stimulus
   reg [4:0] invect;     //invect[4] terminates the loop
   for (invect = 0; invect < 16; invect = invect + 1)
      begin
         {x1, x2, x3, x4} = invect[4:0];
         #10 $display ("x1x2x3x4 = %b, z1 = %b",
                        {x1, x2, x3, x4}, z1);
      end
end
//continued on next page
```

Figure 1.25 Test bench for the design module of Figure 1.24.

```
//instantiate the module into the test bench
log_eqn_pos5 inst1 (
   .x1(x1),
   .x2(x2),
   .x3(x3),
   .x4(x4),
   .z1(z1)
   );
endmodule
```

Figure 1.25 (Continued)

Referring to the test bench of Figure 1.25, following the keyword **begin** is the name of the block: *apply_stimulus*. In this block, a 5-bit **reg** variable is declared called *invect*. This guarantees that all combinations of the four inputs will be tested by the **for** loop, which applies input vectors of $x_1x_2x_3x_4$ = 0000, 0001, 0010, 0011 . . . 1111 to the circuit. The **for** loop stops when the pattern 10000 is detected by the test segment (*invect* < 16). If only a 4-bit vector were applied, then the expression (*invect* < 16) would always be true and the loop would never terminate. The increment segment of the **for** loop does not support an increment designated as *invect*++; therefore, the long notation must be used: *invect* = *invect* + *1*.

The target of the first assignment within the **for** loop ({x_1, x_2, x_3, x_4} = *invect [4:0]*) represents a concatenated target. The concatenation of inputs x_1, x_2, x_3, and x_4 is performed by positioning them within braces: $\{x_1, x_2, x_3, x_4\}$. A vector of five bits ([4:0]) is then assigned to the inputs. This will apply inputs of 0000, 0001, 0010, 0011, . . . 1111 and stop when the vector is 10000.

The **initial** statement also contains a system task (**$display**) which prints the argument values — within the quotation marks — in binary. The concatenated variables x_1, x_2, x_3, and x_4 are listed first; therefore, their values are obtained from the first argument to the right of the quotation marks: $\{x_1, x_2, x_3, x_4\}$. The value for the second variable z_1 is obtained from the second argument to the right of the quotation marks. The variables to the right of the quotation marks are listed in the same order as the variables within the quotation marks.

The delay time (#10) in the system task specifies that the task is to be executed after 10 time units; that is, the delay between the application of a vector and the response of the module. This delay represents the propagation delay of the logic. The simulation results are shown in binary format in Figure 1.26.

```
x1x2x3x4 = 0000, z1 = 0
x1x2x3x4 = 0001, z1 = 1
x1x2x3x4 = 0010, z1 = 0
x1x2x3x4 = 0011, z1 = 1              //continued on next page
```

Figure 1.26 Outputs generated by the test bench of Figure 1.25.

```
x1x2x3x4 = 0100, z1 = 1
x1x2x3x4 = 0101, z1 = 1
x1x2x3x4 = 0110, z1 = 1
x1x2x3x4 = 0111, z1 = 0
x1x2x3x4 = 1000, z1 = 1
x1x2x3x4 = 1001, z1 = 1
x1x2x3x4 = 1010, z1 = 0
x1x2x3x4 = 1011, z1 = 1
x1x2x3x4 = 1100, z1 = 1
x1x2x3x4 = 1101, z1 = 1
x1x2x3x4 = 1110, z1 = 1
x1x2x3x4 = 1111, z1 = 0
```

Figure 1.26 (Continued)

Example 1.2 Equation 1.8 will be minimized as a sum-of-products form and then implemented using built-in primitives of AND and OR with x_4 and x_5 as map-entered variables. Variables may be entered in a Karnaugh map as map-entered variables, together with 1s and 0s. A map of this type is more compact than a standard Karnaugh map, but contains the same information. A map containing map-entered variables is particularly useful in analyzing and synthesizing synchronous sequential machines. When variables are entered in a Karnaugh map, two or more squares can be combined only if the squares are adjacent and contain the same variable(s).

$$z_1 = x_1'x_2'x_3'x_4x_5' + x_1'x_2 + x_1'x_2'x_3'x_4x_5 + x_1x_2'x_3'x_4x_5 + x_1x_2'x_3 + x_1x_2'x_3'x_4' + x_1x_2'x_3'x_5' \quad (1.8)$$

The Karnaugh map is shown in Figure 1.27 in which the following minterm locations combine:

Minterm location 0 = $x_4x_5' + x_4x_5 = x_4$
Minterm location 2 = $1 + x_4$
Combine minterm locations 0 and 2 to yield the sum term $x_1'x_3'x_4$

Combine minterm locations 2 and 3 to yield $x_1'x_2$

Minterm location 4 = $x_4x_5 + x_4' + x_5' = 1$
Minterm location 5 = 1
Combine minterm locations 4 and 5 to yield x_1x_2'

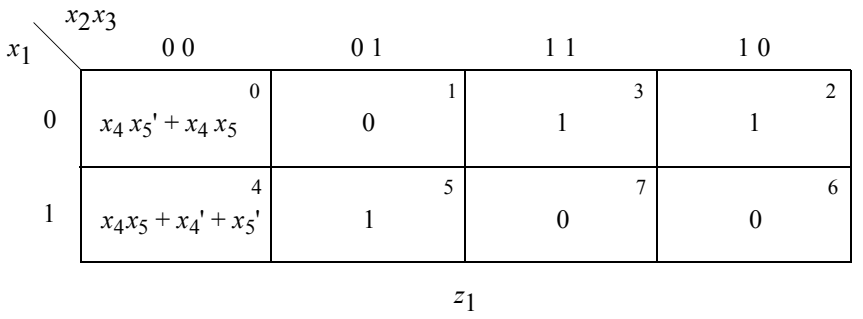


Figure 1.27 Karnaugh map for Example 1.2.

The minimized sum-of-products equation from the Karnaugh map is shown in Equation 1.9. The logic diagram is shown in Figure 1.28. The design module is shown in Figure 1.29 and the test bench is shown in Figure 1.30. Figure 1.31 lists the outputs obtained from the test bench.

$$z_1 = x_1'x_3'x_4 + x_1'x_2 + x_1x_2' \tag{1.9}$$

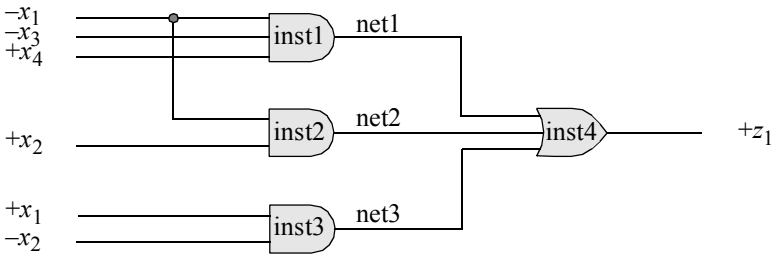


Figure 1.28 Logic diagram for Equation 1.9.

```
//logic equation using map-entered variables
module mev (x1, x2, x3, x4, z1);

input x1, x2, x3, x4;
output z1;

and   inst1 (net1, ~x1, ~x3, x4);
and   inst2 (net2, ~x1, x2);
and   inst3 (net3, x1, ~x2);
or    inst4 (z1, net1, net2, net3);
endmodule
```

Figure 1.29 Design module to implement Equation 1.9 using built-in primitives.

```
//test bench for logic equation using map-entered variables
module mev_tb;

reg x1, x2, x3, x4;
wire z1;

initial        //apply input vectors
begin: apply_stimulus
   reg [4:0] invect;
   for (invect=0; invect<16; invect=invect+1)
      begin
         {x1, x2, x3, x4} = invect [4:0];
         #10 $display ("x1x2x3x4 = %b, z1 = %b",
                        {x1, x2, x3, x4}, z1);
   end
end

//instantiate the module into the test bench
mev inst1 (
   .x1(x1),
   .x2(x2),
   .x3(x3),
   .x4(x4),
   .z1(z1)
   );
endmodule
```

Figure 1.30 Test bench for the design module of Figure 1.29.

```
x1x2x3x4 = 0000, z1 = 0
x1x2x3x4 = 0001, z1 = 1
x1x2x3x4 = 0010, z1 = 0
x1x2x3x4 = 0011, z1 = 0
x1x2x3x4 = 0100, z1 = 1
x1x2x3x4 = 0101, z1 = 1
x1x2x3x4 = 0110, z1 = 1
x1x2x3x4 = 0111, z1 = 1
x1x2x3x4 = 1000, z1 = 1
x1x2x3x4 = 1001, z1 = 1
x1x2x3x4 = 1010, z1 = 1
x1x2x3x4 = 1011, z1 = 1
x1x2x3x4 = 1100, z1 = 0
x1x2x3x4 = 1101, z1 = 0
x1x2x3x4 = 1110, z1 = 0
x1x2x3x4 = 1111, z1 = 0
```

Figure 1.31 Outputs for the test bench of Figure 1.30.

Example 1.3 A 4:1 multiplexer will be designed using built-in logic primitives. The 4:1 multiplexer of Figure 1.32 will be designed using built-in primitives of AND, OR, and NOT. The design is simpler and takes less code if a continuous assignment statement is used, but this section presents gate-level modeling only — continuous assignment statements are used in dataflow modeling.

The multiplexer has four data inputs: d_3, d_2, d_1, and d_0, which are specified as a 4-bit vector *d[3:0]*, two select inputs: s_1 and s_0, specified as a 2-bit vector *s[1:0]*, one scalar input *Enable*, and one scalar output z_1, as shown in the logic diagram of Figure 1.32.. Also, the system function **$time** will be used in the test bench to return the current simulation time measured in nanoseconds (ns). The design module is shown in Figure 1.33, the test bench in Figure 1.34, and the outputs in Figure 1.35.

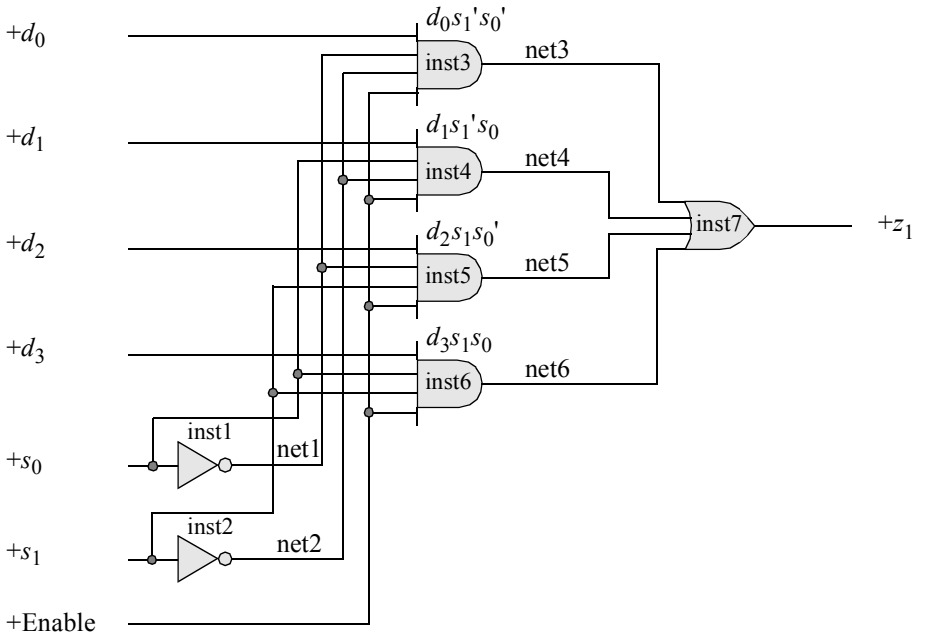


Figure 1.32 Logic diagram of a 4:1 multiplexer to be designed using built-in primitives.

```
//a 4:1 multiplexer using built-in primitives
module mux_4to1 (d, s, enbl, z1);

input [3:0] d;
input [1:0] s;
input enbl;
output z1;
                                    //continued on next page
```

Figure 1.33 Module for a 4:1 multiplexer with *Enable* using built-in primitives.

```
not   inst1 (net1, s[0]),
      inst2 (net2, s[1]);

and   inst3 (net3, d[0], net1, net2, enbl),
      inst4 (net4, d[1], s[0], net2, enbl),
      inst5 (net5, d[2], net1, s[1], enbl),
      inst6 (net6, d[3], s[0], s[1], enbl);

or    inst7 (z1, net3, net4, net5, net6);

endmodule
```

Figure 1.33 (Continued)

```
//test bench for 4:1 multiplexer
module mux_4to1_tb;

reg [3:0] d;
reg [1:0] s;
reg enbl;
wire z1;

initial
$monitor ($time,"ns, select:s=%b, inputs:d=%b, output:z1=%b",
         s, d, z1);
initial
begin
   #0    s[0]=1'b0;  s[1]=1'b0;
         d[0]=1'b0;  d[1]=1'b1;  d[2]=1'b0;  d[3]=1'b1;
         enbl=1'b1;  //d[0]=0; z1=0

   #10   s[0]=1'b0;  s[1]=1'b0;
         d[0]=1'b1;  d[1]=1'b1;  d[2]=1'b0;  d[3]=1'b1;
         enbl=1'b1;  //d[0]=1; z1=1

   #10   s[0]=1'b1;  s[1]=1'b0;
         d[0]=1'b1;  d[1]=1'b1;  d[2]=1'b0;  d[3]=1'b1;
         enbl=1'b1;  //d[1]=1; z1=1

   #10   s[0]=1'b0;  s[1]=1'b1;
         d[0]=1'b1;  d[1]=1'b1;  d[2]=1'b0;  d[3]=1'b1;
         enbl=1'b1;  //d[2]=0; z1=0
                                  //continued on next page
```

Figure 1.34 Test bench for the 4:1 multiplexer of Figure 1.33.

```
    #10   s[0]=1'b1;  s[1]=1'b0;
          d[0]=1'b1;  d[1]=1'b0;  d[2]=1'b0;  d[3]=1'b1;
          enbl=1'b1;  //d[1]=1; z1=0

    #10   s[0]=1'b1;  s[1]=1'b1;
          d[0]=1'b1;  d[1]=1'b1;  d[2]=1'b0;  d[3]=1'b1;
          enbl=1'b1;  //d[3]=1; z1=1

    #10   s[0]=1'b1;  s[1]=1'b1;
          d[0]=1'b1;  d[1]=1'b1;  d[2]=1'b0;  d[3]=1'b0;
          enbl=1'b1;  //d[3]=0; z1=0

    #10   s[0]=1'b1;  s[1]=1'b1;
          d[0]=1'b1;  d[1]=1'b1;  d[2]=1'b0;  d[3]=1'b0;
          enbl=1'b0;  //d[3]=0; z1=0

    #10   $stop;
end

//instantiate the module into the test bench
mux_4to1 inst1 (
    .d(d),
    .s(s),
    .z1(z1),
    .enbl(enbl)
    );
endmodule
```

Figure 1.34 (Continued)

```
0  ns, select:s=00, inputs:d=1010, output:z1=0
10 ns, select:s=00, inputs:d=1011, output:z1=1
20 ns, select:s=01, inputs:d=1011, output:z1=1
30 ns, select:s=10, inputs:d=1011, output:z1=0
40 ns, select:s=01, inputs:d=1001, output:z1=0
50 ns, select:s=11, inputs:d=1011, output:z1=1
60 ns, select:s=11, inputs:d=0011, output:z1=0
```

Figure 1.35 Outputs for the 4:1 multiplexer test bench of Figure 1.34.

Example 1.4 This example illustrates the design of a majority circuit using built-in primitives. The output of a majority circuit is a logic 1 if the majority of the inputs is a logic 1; otherwise, the output is a logic 0. Therefore, a majority circuit must have an odd number of inputs in order to have a majority of the inputs at the same logic level.

A 5-input majority circuit will be designed using the Karnaugh map of Figure 1.36, where a 1 entry indicates that the majority of the inputs is a logic 1.

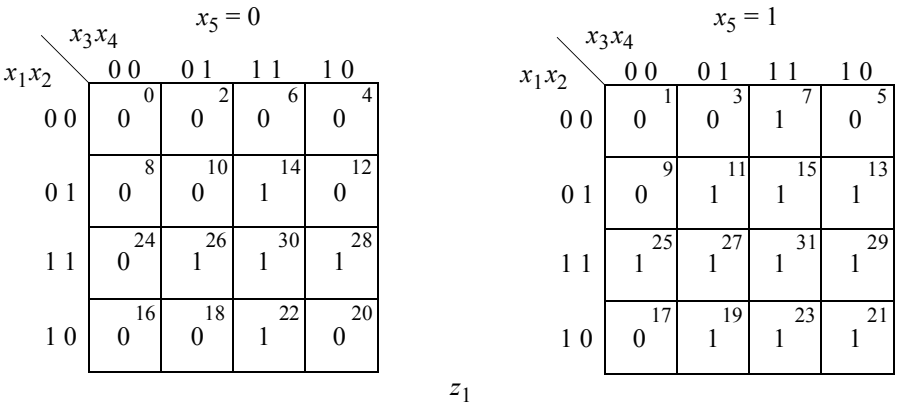


Figure 1.36 Karnaugh map for the majority circuit of Example 1.4.

Equation 1.10 represents the logic for output z_1 in a sum-of-products form. The design module is shown in Figure 1.37, which is designed directly from Equation 1.10 without the use of a logic diagram. The test bench is shown in Figure 1.38, and the outputs are shown in Figure 1.39.

$$z_1 = x_3x_4x_5 + x_2x_3x_5 + x_1x_3x_5 + x_2x_4x_5 + x_1x_4x_5 + x_1x_2x_5 + x_1x_2x_4 + x_2x_3x_4 + x_1x_3x_4 \quad (1.10)$$

```
//5-input majority circuit
module majority (x1, x2, x3, x4, x5, z1);
input x1, x2, x3, x4, x5;
output z1;

and   inst1  (net1, x3, x4, x5),
      inst2  (net2, x2, x3, x5),
      inst3  (net3, x1, x3, x5),
      inst4  (net4, x2, x4, x5),    //continued on next page
```

Figure 1.37 Design module for the majority circuit of Figure 1.36.

```
      inst5  (net5, x1, x4, x5),
      inst6  (net6, x1, x2, x5),
      inst7  (net7, x1, x2, x4),
      inst8  (net8, x2, x3, x4),
      inst9  (net9, x1, x3, x4);
or    inst10 (z1, net1, net2, net3, net4, net5,
                  net6, net7, net8, net9);
endmodule
```

Figure 1.37 (Continued)

```
//test bench for 5-input majority circuit
module majority_tb;
reg x1, x2, x3, x4, x5;
wire z1;

//apply input vectors
initial
begin: apply_stimulus
      reg [6:0] invect;
      for (invect=0; invect<32; invect=invect+1)
            begin
                  {x1, x2, x3, x4, x5} = invect [6:0];
                  #10 $display ("x1x2x3x4x5 = %b, z1 = %b",
                              {x1, x2, x3, x4, x5}, z1);
            end
end

//instantiate the module into the test bench
majority inst1 (
      .x1(x1),
      .x2(x2),
      .x3(x3),
      .x4(x4),
      .x5(x5),
      .z1(z1)
      );

endmodule
```

Figure 1.38 Test bench for the majority circuit module of Figure 1.37.

```
x1x2x3x4x5 = 00000, z1 = 0      x1x2x3x4x5 = 10000, z1 = 0
x1x2x3x4x5 = 00001, z1 = 0      x1x2x3x4x5 = 10001, z1 = 0
x1x2x3x4x5 = 00010, z1 = 0      x1x2x3x4x5 = 10010, z1 = 0
x1x2x3x4x5 = 00011, z1 = 0      x1x2x3x4x5 = 10011, z1 = 1
x1x2x3x4x5 = 00100, z1 = 0      x1x2x3x4x5 = 10100, z1 = 0
x1x2x3x4x5 = 00101, z1 = 0      x1x2x3x4x5 = 10101, z1 = 1
x1x2x3x4x5 = 00110, z1 = 0      x1x2x3x4x5 = 10110, z1 = 1
x1x2x3x4x5 = 00111, z1 = 1      x1x2x3x4x5 = 10111, z1 = 1
x1x2x3x4x5 = 01000, z1 = 0      x1x2x3x4x5 = 11000, z1 = 0
x1x2x3x4x5 = 01001, z1 = 0      x1x2x3x4x5 = 11001, z1 = 1
x1x2x3x4x5 = 01010, z1 = 0      x1x2x3x4x5 = 11010, z1 = 1
x1x2x3x4x5 = 01011, z1 = 1      x1x2x3x4x5 = 11011, z1 = 1
x1x2x3x4x5 = 01100, z1 = 0      x1x2x3x4x5 = 11100, z1 = 1
x1x2x3x4x5 = 01101, z1 = 1      x1x2x3x4x5 = 11101, z1 = 1
x1x2x3x4x5 = 01110, z1 = 1      x1x2x3x4x5 = 11110, z1 = 1
x1x2x3x4x5 = 01111, z1 = 1      x1x2x3x4x5 = 11111, z1 = 1
```

Figure 1.39 Outputs for the majority circuit of Figure 1.37.

Example 1.5 A code converter will be designed to convert a 4-bit binary number to the corresponding Gray code number. The inputs of the binary number $x_1x_2x_3x_4$ are available in both high and low assertion, where x_4 is the low-order bit. The outputs for the Gray code $z_1z_2z_3z_4$ are asserted high, where z_4 is the low-order bit. The binary-to-Gray code conversion table is shown in Table 1.13.

Table 1.13 Binary-to-Gray Code Conversion

Binary Code				Gray Code			
x_1	x_2	x_3	x_4	z_1	z_2	z_3	z_4
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0

//continued on next page

Table 1.13 Binary-to-Gray Code Conversion

Binary Code				Gray Code			
x_1	x_2	x_3	x_4	z_1	z_2	z_3	z_4
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

There are four Karnaugh maps shown in Figure 1.40, one map for each of the Gray code outputs. The equations obtained from the Karnaugh maps are shown in Equation 1.11. The logic diagram is shown in Figure 1.41. The design module is shown in Figure 1.42, the test bench module is shown in Figure 1.43, and the outputs are shown in Figure 1.44.

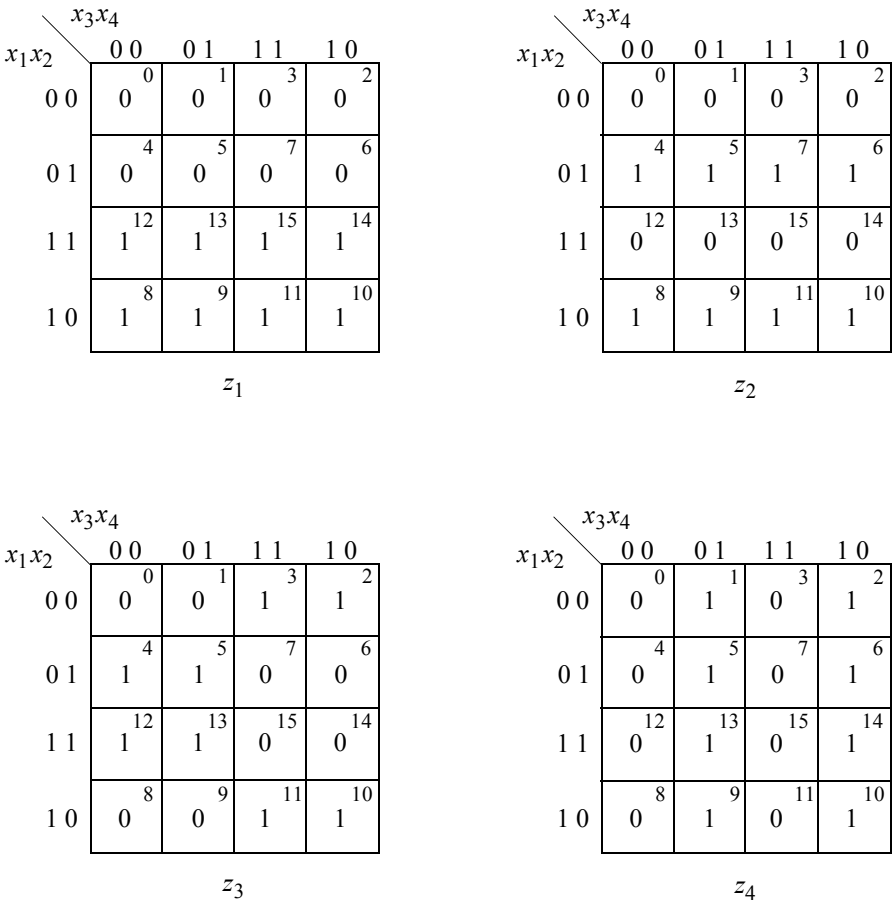


Figure 1.40 Karnaugh maps for the binary-to-Gray code converter.

$$
\begin{aligned}
z_1 &= x_1 \\
z_2 &= x_1'x_2 + x_1x_2' = x_1 \oplus x_2 \\
z_3 &= x_2x_3' + x_2'x_3 = x_2 \oplus x_3 \\
z_4 &= x_3'x_4 + x_3x_4' = x_3 \oplus x_4
\end{aligned} \tag{1.11}
$$

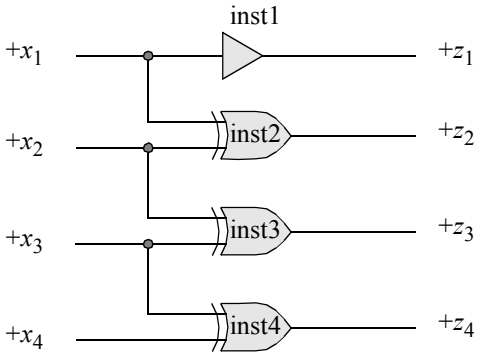


Figure 1.41 Logic diagram for the binary-to-Gray code converter.

```
//binary-to-gray code converter
module bin_to_gray (x1, x2, x3, x4, z1, z2, z3, z4);

input x1, x2, x3, x4;
output z1, z2, z3, z4;

buf inst1 (z1, x1);
xor inst2 (z2, x1, x2);
xor inst3 (z3, x2, x3);
xor inst4 (z4, x3, x4);
endmodule
```

Figure 1.42 Design module for the binary-to-Gray code converter.

```
//test bench for binary-to-gray code converter
module bin_to_gray_tb;

reg x1, x2, x3, x4;
wire z1, z2, z3, z4;
                                    //continued on next page
```

Figure 1.43 Test bench for the binary-to-Gray code converter.

```
//apply input vectors
initial
begin: apply_stimulus
   reg [4:0] invect;
   for (invect=0; invect<16; invect=invect+1)
      begin
         {x1, x2, x3, x4} = invect [4:0];
         #10 $display ("{x1x2x3x4}=%b, {z1z2z3z4}=%b",
                        {x1, x2, x3, x4}, {z1, z2, z3, z4});
      end
end

//instantiate the module into the test bench
bin_to_gray inst1 (
   .x1(x1),
   .x2(x2),
   .x3(x3),
   .x4(x4),
   .z1(z1),
   .z2(z2),
   .z3(z3),
   .z4(z4)
   );

endmodule
```

Figure 1.43 (Continued)

```
{x1x2x3x4}=0000, {z1z2z3z4}=0000
{x1x2x3x4}=0001, {z1z2z3z4}=0001
{x1x2x3x4}=0010, {z1z2z3z4}=0011
{x1x2x3x4}=0011, {z1z2z3z4}=0010
{x1x2x3x4}=0100, {z1z2z3z4}=0110
{x1x2x3x4}=0101, {z1z2z3z4}=0111
{x1x2x3x4}=0110, {z1z2z3z4}=0101
{x1x2x3x4}=0111, {z1z2z3z4}=0100
{x1x2x3x4}=1000, {z1z2z3z4}=1100
{x1x2x3x4}=1001, {z1z2z3z4}=1101
{x1x2x3x4}=1010, {z1z2z3z4}=1111
{x1x2x3x4}=1011, {z1z2z3z4}=1110
{x1x2x3x4}=1100, {z1z2z3z4}=1010
{x1x2x3x4}=1101, {z1z2z3z4}=1011
{x1x2x3x4}=1110, {z1z2z3z4}=1001
{x1x2x3x4}=1111, {z1z2z3z4}=1000
```

Figure 1.44 Outputs for the binary-to-Gray code converter.