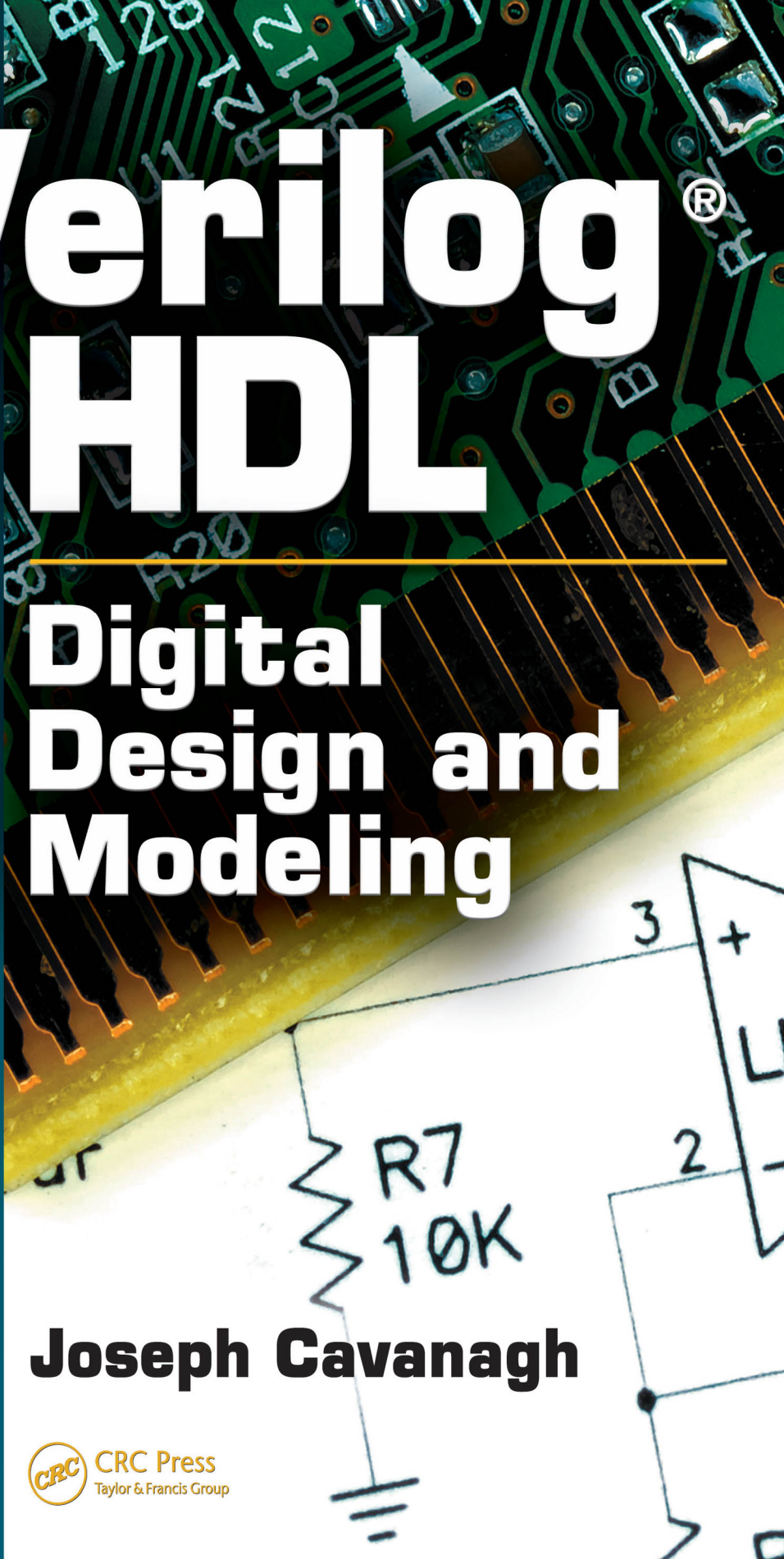
Verilog® HDL

Digital Design and Modeling



Joseph Cavanagh

CRC Press
Taylor & Francis Group

Verilog® HDL

Digital Design and Modeling

Verilog® HDL

Digital Design and Modeling

Joseph Cavanagh
Santa Clara University

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 in the United States of America on acid-free paper
10 9 8 7 6 5 4 3

International Standard Book Number-10: 1-4200-5154-7 (Hardcover)
International Standard Book Number-13: 978-1-4200-5154-4 (Hardcover)

Library of Congress Cataloging-in-Publication Data

Verilog HDL : digital design and modeling / Joseph Cavanagh.
p. cm.
Includes bibliographical references and index.
ISBN 1-4200-5154-7 (alk. paper)
1. Digital electronics. 2. Logic circuits--Computer-aided design. 3. Verilog (Computer hardware description language) I. Title.

TK7868.D5C395 2007
621.39'2--dc22 2006052725

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

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

By the same author:

DIGITAL COMPUTER ARITHMETIC: Design and Implementation

SEQUENTIAL LOGIC: Analysis and Synthesis

To my children,
Brad, Janice, and Valerie

CONTENTS

PREFACE

There are two dominant hardware description languages (HDLs): Verilog HDL and Very High Speed Integrated Circuit (VHSIC) HDL (VHDL). Both are Institute of Electrical and Electronics Engineers (IEEE) standards: Verilog IEEE standard 1364-1995 and VHDL IEEE standard 1076-1993. Of the two hardware description languages, Verilog HDL is the most widely used. 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 complete 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, including both combinational and sequential logic.

The examples include 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, a Hamming code error detection and correction circuit, a binary-coded decimal (BCD) adder/subtractor, arithmetic and logic units (ALUs), and the complete design of a pipelined reduced instruction set computer (RISC) processor. 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, the outputs obtained from the simulator, and the waveforms 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 provides a short history of HDLs and introduces Verilog HDL. Different modeling constructs are presented as well as different ways to indicate the active level (or assertion level) of a signal.

Chapter 2 presents an overview of Verilog HDL 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 is also a section that incorporates more than one modeling construct into a mixed-design model. Later chapters present these modeling constructs in more detail. The concept of ports and modules is introduced in conjunction with the use of test benches for module design verification.

Chapter 3 presents the language elements used in Verilog. These consist of comments, identifiers, keywords, data types, parameters, and a set of values that determine the logic state of a net. Comments are placed in the Verilog code to explain the function of a line of code or a block of code. Identifiers are names given to an object or variable so that they can be referenced elsewhere in the module. Verilog provides a list of predefined keywords that are used to define the language constructs. There are two predefined data types: nets and registers. Nets connect logical elements; registers provide storage elements. Compiler directives are used to induce changes during the compilation of a Verilog module.

Chapter 4 covers the expressions used in Verilog. Expressions consist of operands and operators, which are the basis of the language. Operands can be any of the following data types: constant, parameter, net, register, bit-select, part-select, or a memory element. Verilog contains a large set of operators that perform various operations on different types of data. The following categories of operators are available in Verilog: arithmetic, logical, relational, equality, bitwise, reduction, shift, conditional, concatenation, and replication.

Chapter 5 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. Gate delays are also introduced in this chapter. All gates have a propagation delay, which is the time necessary for a signal to propagate from the input terminals, through the internal circuitry, to the output terminal. Examples of iterative networks and a priority encoder are presented as design examples using built-in primitives.

Chapter 6 covers user-defined primitives (UDPs), which are primitive logic functions that are designed according to user specifications. These primitive functions are usually at a higher level of abstraction than the built-in primitives. They are independent primitives and do not instantiate other primitives or modules. They can be used in the design of both combinational and sequential logic circuits. Sequential primitives include level-sensitive and edge-sensitive circuits. Several design examples are included in this chapter, including a binary-to-Gray code converter, a full adder designed from two half adders, multiplexers, a level-sensitive latch, an edge-sensitive flip-flop, a modulo-8 counter, and a Moore finite-state synchronous sequential machine, among others.

Chapter 7 presents dataflow modeling, which is the first of three primary modeling constructs. Dataflow modeling is at a higher level of abstraction than either built-in primitives or UDPs. Dataflow modeling uses the continuous assignment statement to design combinational logic without employing logic gates and interconnecting wires. Design examples presented in this chapter include the use of

reduction operators, an octal-to-binary encoder, a multiplexer design using the conditional operator, a 4-bit adder, a high-speed carry lookahead adder, asynchronous sequential machines, and a Moore pulse-mode asynchronous sequential machine. All examples include test benches to test the design modules for correct functional operation, outputs from simulation, and waveforms.

Chapter 8 covers the concepts of behavioral modeling, which describe the behavior of a digital system and is not concerned with the direct implementation of logic gates, but rather the architecture of the system. Behavioral modeling represents a higher level of abstraction than previous modeling methods. A Verilog module that is designed using behavioral modeling contains no internal structural details; it simply defines the behavior of the hardware in an abstract, algorithmic description. Verilog contains two structured procedure statements or behaviors: **initial** and **always**. This chapter introduces procedural assignments and different delay techniques. Procedural assignments include blocking and nonblocking assignments. Conditional statements, which alter the flow within a behavior based upon certain conditions, are addressed. An alternative to conditional statements is the **case** statement, which is a multiple-way conditional branch. Looping statements are also presented. Many complete design examples are illustrated, which include a carry lookahead adder, an add-shift unit, an odd parity generator, a parallel-in, serial-out shift register, counters that count in different sequences, ALUs, various Moore and Mealy synchronous sequential machines, and asynchronous sequential machines.

Chapter 9 covers the third main modeling technique, structural modeling. Structural modeling consists of instantiating one or more of the following objects into a design module: built-in primitives, UDPs, or other design modules. This chapter presents several complete design examples using structural modeling constructs. The examples include a Gray-to-binary code converter, a BCD-to-decimal decoder, a modulo-10 counter, an adder-subtractor, an adder and high-speed shifter, an array multiplier, Moore and Mealy synchronous and asynchronous sequential machines, and a Moore pulse-mode asynchronous sequential machine. As in other chapters, the examples contain the design module, the test bench module, the outputs, and the waveforms.

Chapter 10 presents tasks and functions, which are similar to procedures or subroutines found in other programming languages. 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 and then called when required. Examples are given for both tasks and functions.

Chapter 11 presents several design examples utilizing the modeling methods covered in previous chapters. The designs are usually more complex than those previously given. As in other chapters, the designs are complete and include the design module, the test bench module, the outputs, and the waveforms. The examples include a Johnson counter, a counter shifter module, a universal shift register, a Hamming code error detection and correction circuit with accompanying theory, the Booth multiply algorithm with the Booth method described in detail, various Moore and Mealy sequential machines, including a Mealy one-hot machine, a BCD adder/subtractor, and the complete design of a pipelined RISC processor.

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 Dr. Ivan Pesic, CEO of Silvaco International, 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, Chair, Department of Computer Engineering, Santa Clara University who supported me in all my endeavors; Dr. Geri Lamble; Steve Midford, who reviewed the entire manuscript and offered many helpful suggestions and comments; and Ron Lewerenz. Thanks also to Nora Konopka and the staff at Taylor & Francis for their support.

Joseph Cavanagh

1

Introduction

This book covers the design of combinational and sequential logic using the Verilog hardware description language (HDL). An HDL is a method of designing digital hardware by means of software. A considerable saving of time can be realized when designing systems using an HDL. This offers a competitive advantage by reducing the time-to-market for a system. Another advantage is that the design can be simulated and tested for correct functional operation before implementing the system in hardware. Any errors that are found during simulation can be corrected before committing the design to expensive hardware implementation.

1.1 History of HDL

HDLs became popular in the 1980s and were used to describe large digital systems using a textual format rather than a schematic format such as logic diagrams. With the advent of application-specific integrated circuits (ASICs), field programmable gate arrays (FPGAs), and complex programmable logic devices (CPLDs), computer-aided engineering techniques became necessary. This allowed the engineers to use a programming language to design the logic of the system. Using this technique, test benches could be designed to simulate the entire system and obtain binary outputs

and waveforms. Many of these languages were proprietary and not placed in the public domain. Verilog HDL is one of two primary hardware description languages. The other main HDL is the Very High Speed Integrated Circuit (VHSIC) hardware description language (VHDL). VHDL was developed for the United States Department of Defense and was created jointly by IBM, Texas Instruments, and Intermetrics. Although VHDL has not achieved the widespread acceptance of Verilog, both are IEEE standards.

HDLs allow the designer to easily and quickly express architectural concepts in a precise notation without the aid of logic diagrams. All HDLs express the same fundamental concepts, but in slightly different notations.

1.2 Verilog HDL

The Verilog hardware description language is the state-of-the-art method for designing digital and computer systems. Verilog HDL is a C-like language — with some Pascal syntax — used to model a digital system at many levels of abstraction from a logic gate to a complex digital system to a mainframe computer. The combination of C and Pascal syntax makes Verilog easy to learn. The completed design is then simulated to verify correct functional operation. Verilog HDL is the most widely used HDL in the industry.

The Verilog HDL is able to describe both combinational and sequential logic, including level-sensitive and edge-triggered storage devices. Verilog provides a clear relationship between the language syntax and the physical hardware.

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 International for use in the design and verification of digital systems. The SILOS simulation environment is a method to quickly prototype and debug any ASIC, FPGA, or CPLD design. It is an intuitive environment that displays every variable and port from a module to a logic gate. SILOS allows single stepping through the Verilog source code, as well as drag-and-drop ability from the source code to a data analyzer for waveform generation and analysis.

Verilog HDL supports a top-down design approach of hierarchical decomposition as well as a bottom-up approach. In a top-down design method, the top-level block is defined, then each sub-block that is used to build the top-level is defined. These second-level blocks are then further subdivided until the lowest level is defined. In a bottom-up method, the building blocks (modules) are first defined. These modules are then used to build larger modules, which are then instantiated into a structural module. Verilog can be used to model algorithms, Boolean equations, and individual logic gates. Simulation occurs at different levels. The low-level modules are first designed and tested for correct functional operation by the simulator using a test bench. These modules are then instantiated into the top-level (structural) module, which is then simulated by means of a test bench.

1.2.1 IEEE Standard

Verilog HDL was developed by Phillip Moorby in 1984 as a proprietary HDL for Gateway Design Automation. Gateway was later acquired by Cadence Design Systems, which placed the language in the public domain in 1990. The Open Verilog International was then formed to promote the Verilog HDL language. In 1995, Verilog was made an IEEE standard HDL (IEEE Standard 1364-1995) and is described in the Verilog Hardware Description Language Reference Manual.

1.2.2 Features

Logic primitives such as AND, OR, NAND, and NOR gates are part of the Verilog language. These are built-in primitives that can be instantiated into a module. The designer also has the option of creating a user-defined primitive (UDP), which can then be instantiated into a module in the same way as a built-in primitive. UDPs can be any logic function such as a multiplexer, decoder, encoder, or flip-flop.

Different types of delays can be introduced into a logic circuit including: interstatement, intrastatement, inertial, and transport delays. These will be defined later in the appropriate sections.

Designs can be modeled in three different modeling constructs: (1) dataflow, (2) behavioral, and (3) structural. Module design can also be done in a mixed-design style, which incorporates the above constructs as well as built-in and user-defined primitives. Structural modeling can be described for any number of module instantiations.

Verilog does not impose a limit to the size of the system; therefore, SILOS can be used to design any size system. Verilog can be used not only to design all the modules of a system, but also to design the test bench that is used for simulation.

Verilog also has available bitwise logic functions such as bitwise AND (&) and bitwise OR (|). High-level programming language constructs such as multiway branching (case statements), conditional statements, and loops are also available.

1.3 Assertion Levels

There are different ways to indicate the active level (or assertion) of a signal. Table 1.1 lists various methods used by companies and textbooks. This book will use the +A and –A method. The AND function can be represented three ways, as shown in Figure 1.1, using an AND gate, a NAND gate, and a NOR gate. Although only two inputs are shown, both AND and OR circuits can have three or more inputs. The plus (+) and minus (–) symbols that are placed to the left of the variables indicate a high or low voltage level, respectively. This indicates the asserted (or active) voltage

level for the variables; that is, the *logical 1* (or true) state, in contrast to the *logical 0* (or false) state.

Table 1.1 Assertion Levels

Active high assertion	+A	A	A(H)	A	A	A
Active low assertion	–A	¬A	A(L)	*A	$\overline{A}$	A'

Thus, a signal can be asserted either plus or minus, depending upon the active condition of the signal at that point. For example, Figure 1.1(a) specifies that the AND function will be realized when both input *A* and input *B* are at their more positive potential, thus generating an output at its more positive potential. The word *positive* as used here does not necessarily mean a positive voltage level, but merely the more positive of two voltage levels. Therefore, the output of the AND gate of Figure 1.1(a) can be written as +(*A* & *B*).

To illustrate that a plus level does not necessarily mean a positive voltage level, consider two logic families: transistor-transistor logic (TTL) and emitter-coupled logic (ECL). The TTL family uses a +5 volt power supply. A plus level is approximately +3.5 volts and above; a minus level is approximately +0.2 volts. The ECL family uses a –5.2 volt power supply. A plus level is approximately –0.95 volts; a minus level is approximately –1.7 volts. Although –0.95 volts is a negative voltage, it is the more positive of the two ECL voltages.

The logic symbol of Figure 1.1(b) is a NAND gate in which inputs *A* and *B* must both be at their more positive potential for the output to be at its more negative potential. A small circle (or wedge symbol for IEEE std 91-1984 logic functions) at the input or output of a logic gate indicates a more negative potential. The output of the NAND gate can be written as –(*A* & *B*).

Figure 1.1(c) illustrates a NOR gate used for the AND function. In this case, inputs *A* and *B* must be active (or asserted) at their more negative potential in order for the output to be at its more positive potential. The output can be written as +(*A* & *B*). A variable can be active (or asserted) at a high and a low level at the same time, as shown in Figure 1.2.

Figure 1.1(d) shows a NAND gate used for the OR function. Either input *A* or *B* (or both) must be at its more negative potential to assert the output at its more positive potential. The output can be written as +(*A* | *B*), where the symbol (|) indicates the OR operation in Verilog.

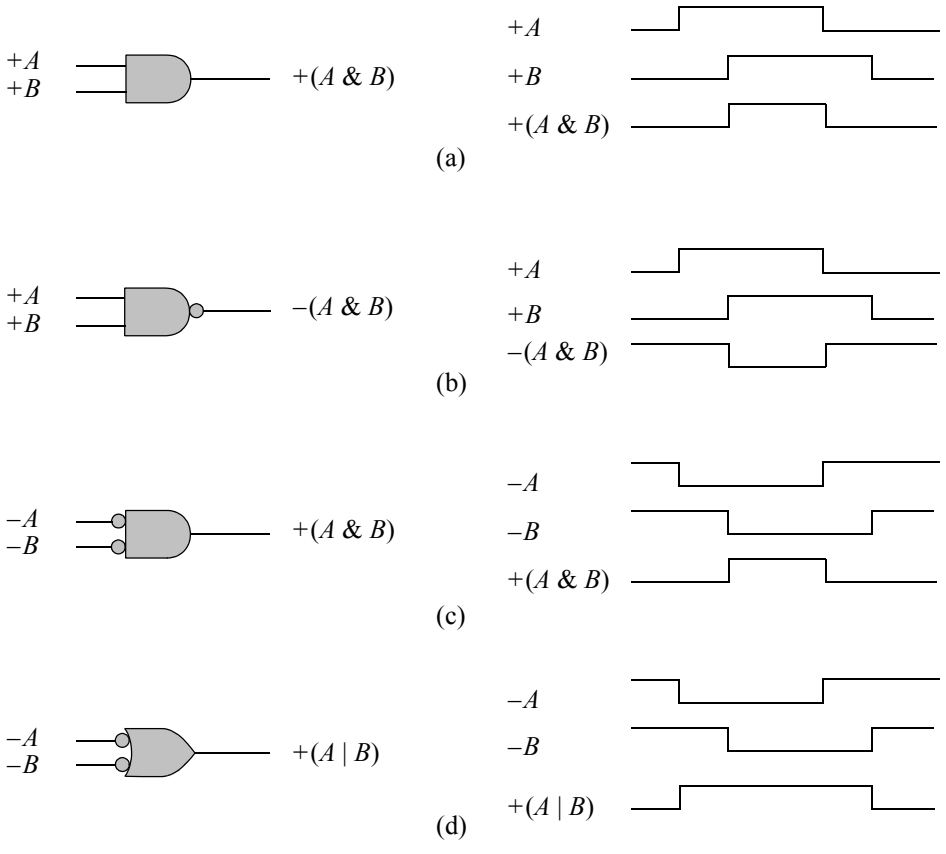


Figure 1.1 Logic symbols and waveforms for AND, NAND, NOR, and NAND (negative-input OR).

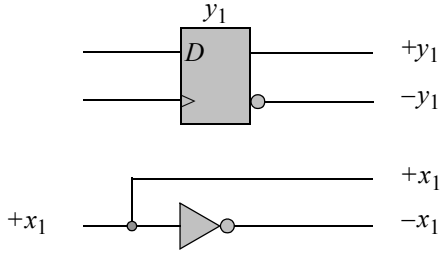


Figure 1.2 Signals can be active high and low at the same time.

2

Overview

This chapter provides a brief introduction to the design methodologies and modeling constructs of the Verilog hardware description language (HDL). Modules and ports will be presented. Modules are the basic units that describe the design of the Verilog hardware. Ports allow the modules to communicate with the external environment; that is, other modules and input/output signals. Different methods will be presented for designing test benches. Test benches are used to apply input vectors to the module in order to test the functional operation of the module in a simulation environment.

Three module constructs will be described together with applications of each type of module. The different modules are: dataflow modeling, behavioral modeling, and structural modeling. Examples will be shown for each type of modeling. There will also be an introduction to mixed-design modeling, which uses a combination of the three main modeling techniques.

2.1 Design Methodologies

There are two main types of design methodologies: *top-down* design and *bottom-up* design. Figure 2.1 shows the layout for a top-down design. In a top-down design, the top-level block is identified, then the blocks in the next lower level are defined. This process continues until all levels in the structure have been defined. The bottom level of the structure contains blocks that cannot be further divided and can be considered as leaf cells in the tree structure. Figure 2.2 shows the layout for a bottom-up design. In a bottom-up design, the lowest level — containing the leaf cells — is defined first.

These become the building blocks with which to design the next higher level. The blocks from each level become the building blocks for the next higher level. This process continues until the top level is reached.

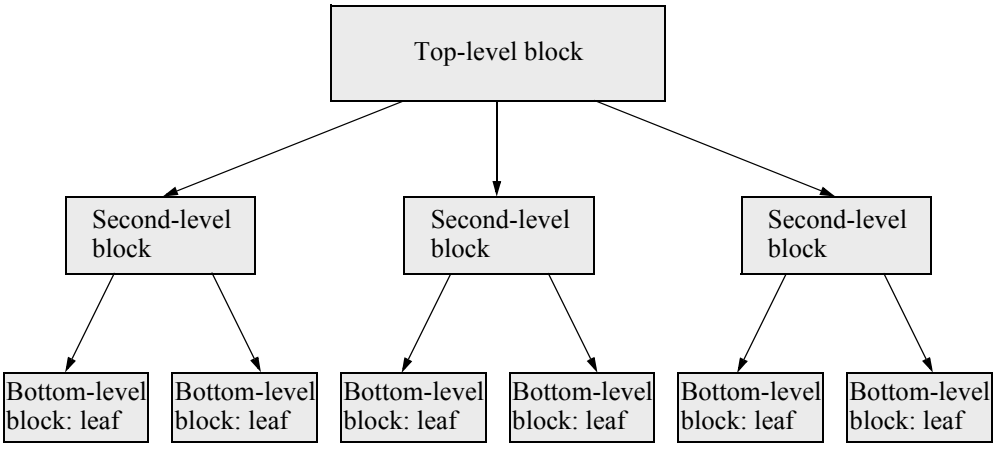


Figure 2.1 Top-down design methodology.

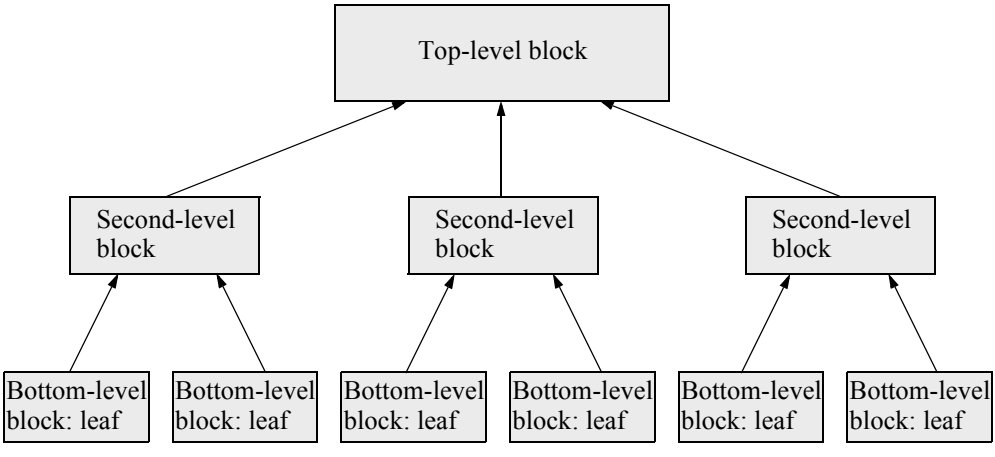


Figure 2.2 Bottom-up design methodology.

The system architect defines the specifications for the machine in the top-level block; for example, a pipelined reduced instruction set computer (RISC). The second-level blocks would consist of the different units in the computer such as instruction cache, instruction unit, decode unit, execution unit, register file, and data cache. The bottom level consists of the hardware in each of the six units.

For example, the instruction cache contains a program counter and an incrementer. The decode unit contains hardware to decode the instruction into its constituent parts such as operation code, source address, and destination address. The execution unit contains the arithmetic and logic unit and associated registers and multiplexers.

2.2 Modulo-16 Synchronous Counter

The top-down approach will now be illustrated for a modulo-16 synchronous counter using *D* flip-flops. The counting sequence is: $y_3y_2y_1y_0$ = 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111, 0000, . . . , where y_i is the name of the flip-flop. The low-order flip-flop is y_0. Figure 2.3 shows the tree structure for top-down design.

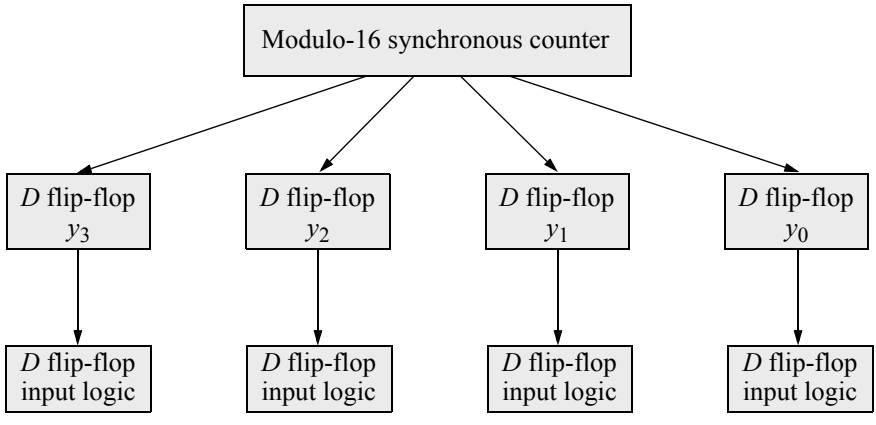


Figure 2.3 Top-down design for a modulo-16 synchronous counter.

The *D* flip-flop input logic blocks will now be expanded to show more detail. These blocks represent the logic for the input equations of the *D* flip-flops. The equations will be obtained using Karnaugh maps in the traditional manner. Using the

counting sequence shown above, the Karnaugh maps are illustrated in Figure 2.4. The equations for the *D* flip-flops are shown in Equation 2.1. The logic diagram, obtained from the *D* flip-flop input equations, is shown in Figure 2.5.

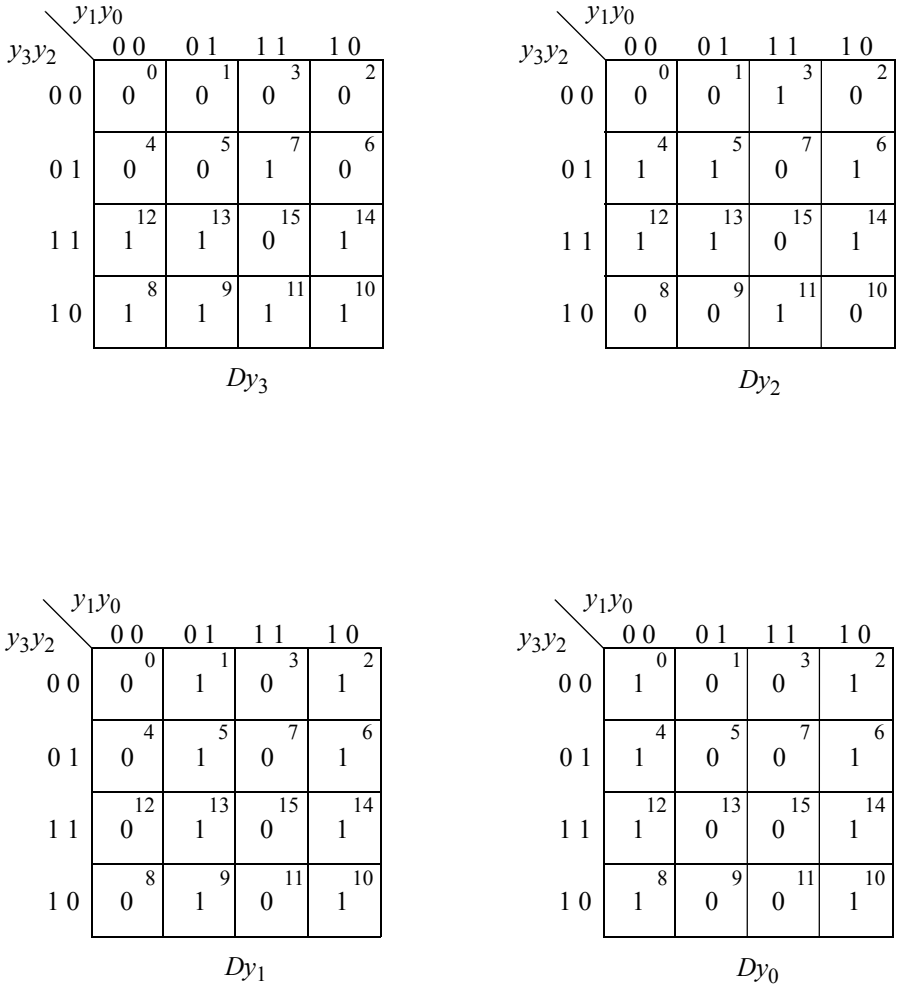


Figure 2.4 Karnaugh maps for the modulo-16 synchronous counter.

The reconfigured top-down design is illustrated in Figure 2.6 showing more detail for the bottom-level leaf nodes. The leaf nodes represent the input logic for the *D* flip-flops. The input logic is obtained by designing the appropriate gates as separate

modules and then instantiating the modules into the structural module that represents the modulo-16 synchronous counter. Modules for the 2-input, 3-input, and 4-input AND gates are labeled AND2, AND3, and AND4, respectively. Modules for the 3-input and 4-input OR gates are labeled OR3 and OR4, respectively. There is one exclusive-OR gate module labeled XOR2.

$$
\begin{aligned}
Dy_3 &= y_3y_2' + y_3y_1' + y_3y_0' + y_3'y_2y_1y_0 \\
Dy_2 &= y_2y_1' + y_2y_0' + y_2'y_1y_0 \\
Dy_1 &= y_1'y_0 + y_1y_0' \\
Dy_0 &= y_0'
\end{aligned} \tag{2.1}
$$

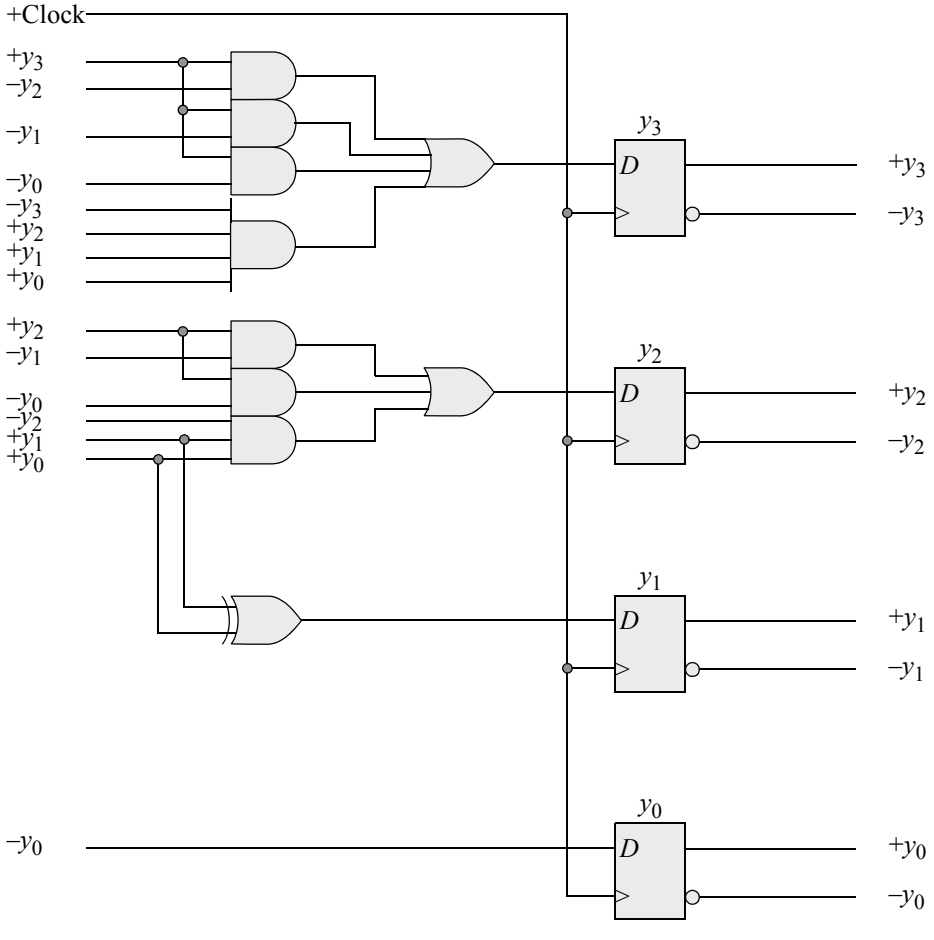


Figure 2.5 Logic diagram for the modulo-16 synchronous counter.

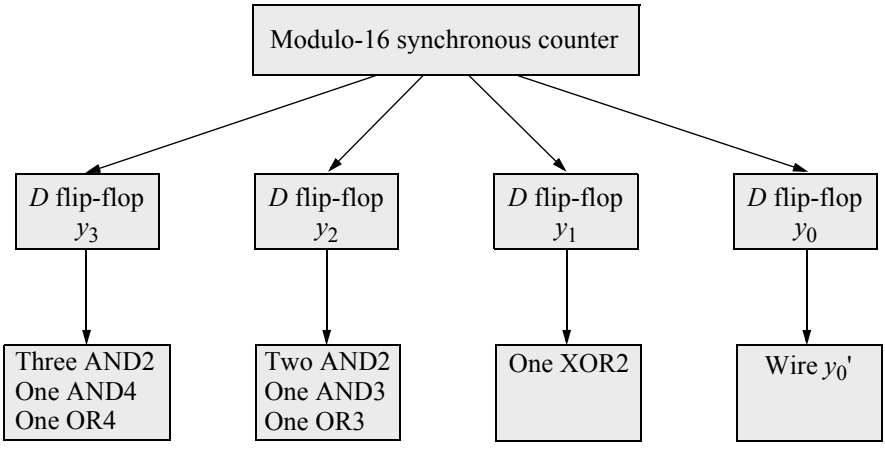


Figure 2.6 Reconfigured top-down design for the modulo-16 synchronous counter showing expanded detail for the bottom-level leaf nodes.

In the top-down design methodology, the functionality of the counter is defined in the top-level block as a modulo-16 synchronous counter. In the next level, the storage elements are established as *D* flip-flops. The bottom level defines the input logic for the *D* flip-flops. No further levels are required for this design, because the leaf nodes define the logic gates, which are the lowest level. In a bottom-up methodology, the flow is in the opposite direction by combining small building blocks into larger building blocks.

2.3 Four-Bit Ripple Adder

Another example of top-down methodology is shown in Figure 2.7. This is a ripple adder, which is a relatively low-speed adder because there is no carry-lookahead feature. The tree structure consists of four levels, each successive level providing additional detail of the 4-bit adder.

The 4-bit adder consists of four full adders connected serially in which the carry-out of stage$_i$ is carry-in to stage$_{i+1}$. A full adder adds the two operand bits — augend and addend — plus the carry-in from the previous lower-order stage and produces two outputs: sum and carry. Each full adder is designed using two half adders and one OR gate. A half adder adds the two operand bits only, and produces two outputs: sum and carry. Each half adder is designed using one exclusive-OR gate and one AND gate.

The truth tables for a half adder and full adder are shown in Table 2.1 and Table 2.2, respectively. The corresponding equations are listed in Equation 2.2 and Equation 2.3. The logic diagram for the half adder is shown in Figure 2.8 and for the full adder in Figure 2.9.

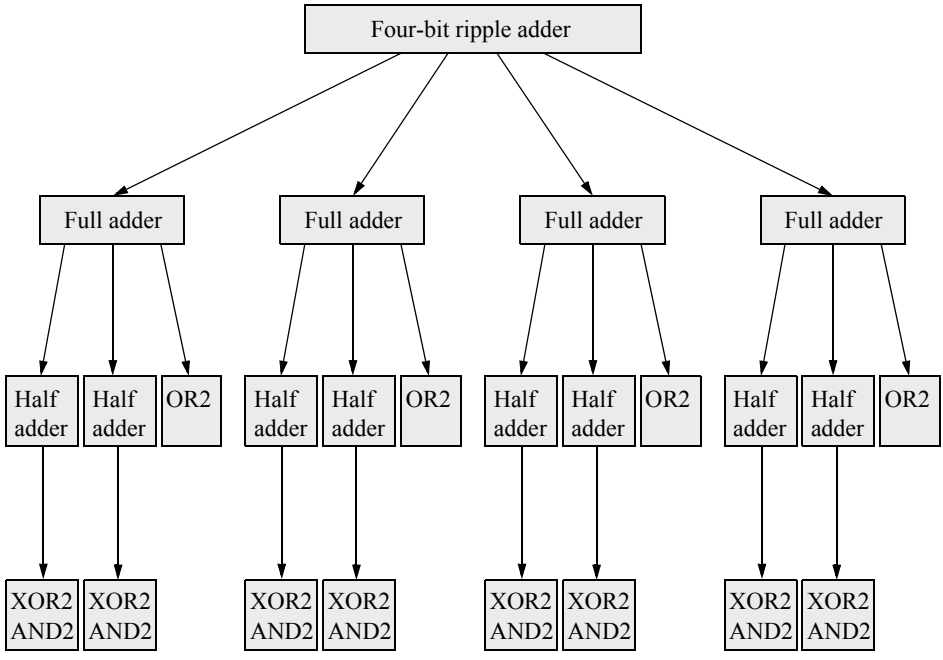


Figure 2.7 Top-down design for a 4-bit ripple adder.

Table 2.1 Truth Table for a Half Adder

a	*b*	*sum*	*cout*
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 2.2 Truth Table for a Full Adder

a	*b*	*cin*	*sum*	*cout*
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$
\begin{aligned}
sum &= a'b + ab' \\
&= a \oplus b \\
cout &= ab
\end{aligned}
\tag{2.2}
$$

$$
\begin{aligned}
sum &= a'b'cin + a'bcin' + ab'cin' + abcin \\
&= cin\ (a \oplus b)' + cin'\ (a \oplus b) \\
&= a \oplus b \oplus cin \\
cout &= a'bcin + ab'cin + abcin' + abcin \\
&= cin\ (a \oplus b) + ab
\end{aligned}
\tag{2.3}
$$

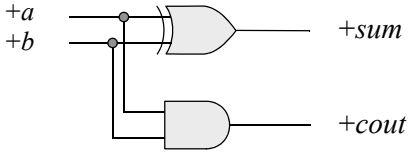


Figure 2.8 Logic diagram for a half adder.

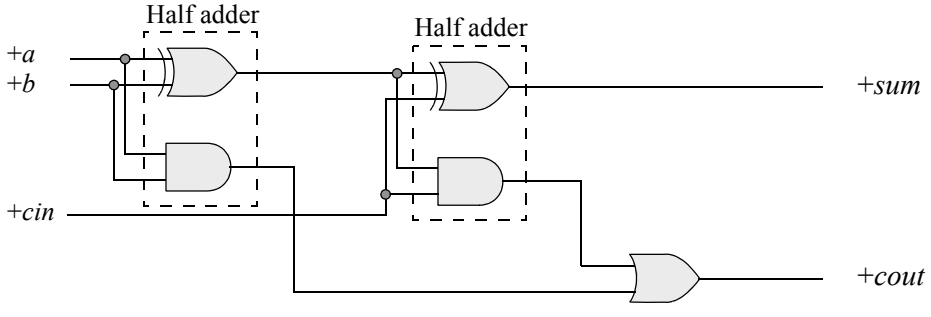


Figure 2.9 Logic diagram for a full adder.

2.4 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 Chapter 5.

Modules contain *ports* which allow communication with the external environment or other modules. For example, the full adder of Figure 2.9 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 2.10. An AND gate can be defined as shown in the module of Figure 2.11, where the input ports are x_1 and x_2 and the output port is z_1.

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

Figure 2.10 General structure of a Verilog module.

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 2.11, 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 Chapter 7. The keyword **assign** describes the behavior of the circuit. Output z_1 is assigned the value of x_1 ANDed (&) with x_2. This continuous assignment statement is discussed in detail in Chapter 7.

```
//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 2.11 Verilog module for an AND gate with two inputs.

2.4.1 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) as well as the waveforms. It is good practice to keep the design module and test bench module separate.

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 2.12 shows a simple test bench to test the 2-input AND gate of Figure 2.11. 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 7 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 11 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);

10 //apply input vectors
   initial
   begin
      #0    x1 = 1'b0;
            x2 = 1'b0;
15
      #10   x1 = 1'b0;
            x2 = 1'b1;

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

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

25    #10   $stop;

   end

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

   endmodule
```

Figure 2.12 Test bench for the 2-input AND gate of Figure 2.11.

Lines 13 and 14 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 16 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.

Line 29 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 30 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 2.13 and the waveforms in Figure 2.14. 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 waveforms show the values of the input and output variables at the specified simulation times. Notice that the inputs change value every 10 time units and that the duration of simulation is 40 time units. The time units can be specified for any duration.

The Verilog syntax will be covered in greater detail in subsequent chapters. 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 2.13 Binary outputs for the test bench of Figure 2.12 for a 2-input AND gate.

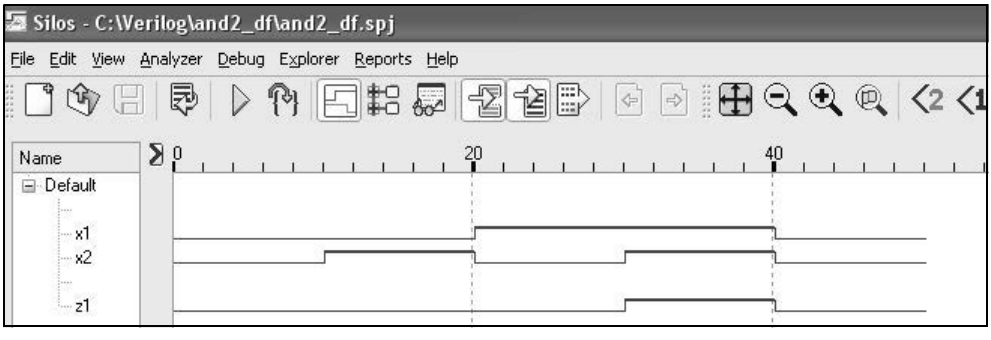


Figure 2.14 Waveforms for the test bench of Figure 2.12 for a 2-input AND gate.

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 2.15 illustrates the necessary statements to generate clock pulses that have a duty cycle of 20%. The clock pulse waveform is shown in Figure 2.16.

```
//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 2.15 Verilog code to generate clock pulses with a 20% duty cycle.

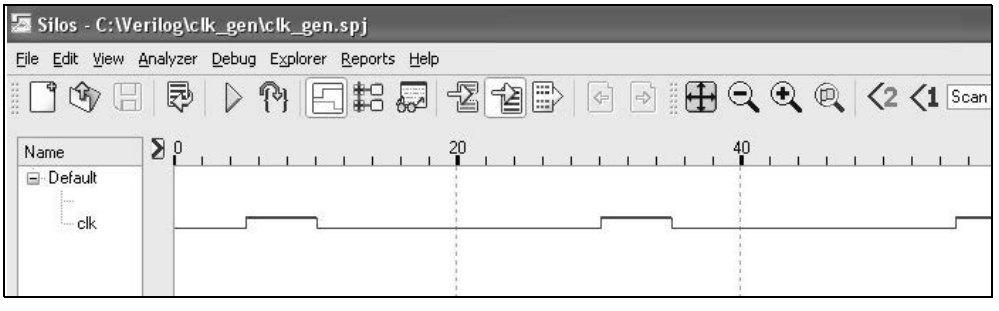


Figure 2.16 Waveform showing clock pulses with a 20% duty cycle.

In Figure 2.15, the clock begins at time 0 with a value of 0. Five time units later, the clock is assigned a value of 1; five time units later (at time 10), the clock goes low (0), 20 time units later (at time 30), the clock goes high (1); five time units later (at time 35), the clock is again assigned a value of 0. It is important to note that the times specified in the code are accumulative; that is, when the clock goes low five time units after the second pulse, this occurs at time 35. Simulation stops at time 70, 10 time units after the last pulse goes low.

Before introducing dataflow, behavioral, structural, and mixed modeling techniques, some definitions will be presented. These definitions fall into the general category of procedural constructs.

2.4.2 Construct Definitions

Continuous assignment The continuous assignment statement is used to describe combinational logic where the output of the circuit is evaluated whenever an input changes; that is, the value of the right-hand side expression is *continuously assigned* to the left-hand side net. Continuous assignments are similar to Boolean algebra, which is a systematic treatment of logical operations. Continuous assignments can be used only for nets, not for register variables. A continuous assignment, as shown below, establishes a relationship between a right-hand side expression and a left-hand side net. A continuous assignment occurs outside of an **initial** or an **always** statement. The syntax for a continuous assignment statement is:

assign [delay] lhs_net = rhs_expression

Whenever the value of a variable in the right-hand side expression changes, the right-hand side is evaluated and the value is assigned to the left-hand side after any specified delay. That is, the right-hand side creates a value that is assigned to the target variable. The delay specifies the time duration between the time a variable on the right-hand side changes and the time the new value is assigned to the left-hand side.

Procedural continuous assignment This is a procedural version of the continuous assignment statement and is made within a behavioral construct (**initial** or **always**) and creates a dynamic binding to a variable. This assignment overrides existing assignments to a net or register; for example, using **assign** . . . **deassign** or **force** . . . **release** procedural statements, which allows for continuous assignments to be made to registers for a specific period of time. A continuous assignment, used for combinational logic, is a static binding because its effect is retained for the duration of simulation. The dynamic binding of a procedural continuous assignment means that the assignment can be changed during execution of a behavioral procedure.

Procedure As in any programming language, a procedure is simply a sequence of operations that produce a result.

Procedural statement A procedural statement is a synonym for instruction.

Procedural assignment A procedural assignment statement represents a logic function that is derived from the right-hand side of an assignment statement and assigned to the left-hand side target. A procedural assignment statement can occur only within an **initial** or an **always** statement. The value assigned to a variable remains unchanged until another procedural assignment changes the value. There are two types of procedural assignment statements: blocking and nonblocking.

Blocking statement Blocking assignments are used only when describing combinational logic in a procedural block. Blocking assignment statements are executed in the order in which they are listed in a procedural block. In a blocking assignment statement, the assignment to the left-hand side variable takes place before the following statement in the sequential block is executed. The symbol that represents a blocking statement is (=).

Nonblocking statement Nonblocking assignments allow scheduling of assignments without blocking the following statements in a procedural block. Nonblocking assignments are used to describe only sequential elements in a procedural assignment. The simulator processes all blocking assignments first. Once the assignments have been made, all the nonblocking assignments are evaluated and placed on the event queue (the event queue is presented in Chapter 11).

All variables are evaluated and assigned in the current simulation time. Once all the variables in the blocking assignments have been evaluated, the variables using nonblocking assignments are evaluated and placed on the event queue. Nonblocking assignment statements are used to synchronize assignment statements so that they appear to happen simultaneously. They allow scheduling of assignments without blocking execution of the statements that follow in a sequential block. The symbol that represents a nonblocking statement is (<=).

2.5 Introduction to Dataflow Modeling

Dataflow modeling is used to design combinational logic only. It allows designers to implement a logical function at a higher level of abstraction than gate level modeling using built-in primitives. The fundamental method of designing in dataflow is the continuous assignment statement **assign**, an example of which is shown below.

assign *sum* = *a* ^ *b* ^ *cin*

For example, the 2-input AND gate shown in the Verilog code of Figure 2.11 uses the continuous assignment statement to assign a value to z_1 whenever x_1 or x_2 changes value (**assign** $z_1 = x_1 \ \& \ x_2$;). Since no delay is specified, the default value is zero delay.

2.5.1 Two-Input Exclusive-OR Gate

This section presents the design of a 2-input exclusive-OR gate using dataflow modeling. The circuit is shown in Figure 2.17 and the Verilog code is in Figure 2.18. Note that the symbol for the exclusive-OR function is the caret.

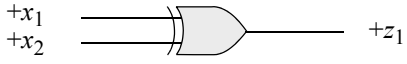


Figure 2.17 Two-input exclusive-OR gate to be designed using dataflow modeling.

```
//dataflow 2-input exclusive-or gate
module xor2 (x1, x2, z1);

input x1, x2;
output z1;

wire x1, x2;
wire z1;

assign z1 = x1 ^ x2;

endmodule
```

Figure 2.18 Verilog code for the 2-input exclusive-OR gate of Figure 2.17.

The test bench for the exclusive-OR gate will be different than the one shown in Figure 2.12 for the 2-input AND gate. Several new modeling constructs are shown in the exclusive-OR test bench of Figure 2.19. Since there are two inputs to the exclusive-OR gate, all four combinations of two 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.

Following the keyword **begin** is the name of the block: *apply_stimulus*. In this block, a 3-bit **reg** variable is declared called *invect*. This guarantees that all combinations of the two inputs will be tested by the **for** loop, which applies input vectors of $x_1x_2 = 00, 01, 10, 11$ to the circuit. The **for** loop stops when the pattern 100 is detected by the test segment (*invect* < 4). If only a 2-bit vector were applied, then the expression (*invect* < 4) 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 ($\{x_1, x_2\}$ = *invect [2:0]*) represents a concatenated target. The concatenation of inputs x_1 and x_2 is performed by positioning them within braces: $\{x_1, x_2\}$. A vector of three bits ([2:0]) is then assigned to the inputs. This will apply inputs of 00, 01, 10, 11, and stop when the vector is 100.

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 and x_2 are listed first; therefore, their values are obtained from the first argument to the right of the quotation marks: $\{x_1, x_2\}$. The value for the second variable z_1 is obtained from the second argument to the right of 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 2.20 and the waveforms in Figure 2.21.

```
//2-input exclusive-or gate test bench
module xor2_tb;

reg x1, x2;
wire z1;

//apply input vectors
initial
begin: apply_stimulus
   reg [2:0] invect;
   for (invect = 0; invect < 4; invect = invect + 1)
      begin
         {x1, x2} = invect [2:0];
         #10 $display ("{x1x2} = %b, z1 = %b",
                        {x1, x2}, z1);
      end
end

//continued on next page
```

Figure 2.19 Test bench for the 2-input exclusive-OR gate module of Figure 2.18.

```
//instantiate the module into the test bench
xor2 inst1 (
   .x1(x1),
   .x2(x2),
   .z1(z1)
   );
endmodule
```

Figure 2.19 (Continued)

```
{x1x2} = 00, z1 = 0
{x1x2} = 01, z1 = 1
{x1x2} = 10, z1 = 1
{x1x2} = 11, z1 = 0
```

Figure 2.20 Binary outputs for the test bench of Figure 2.19.

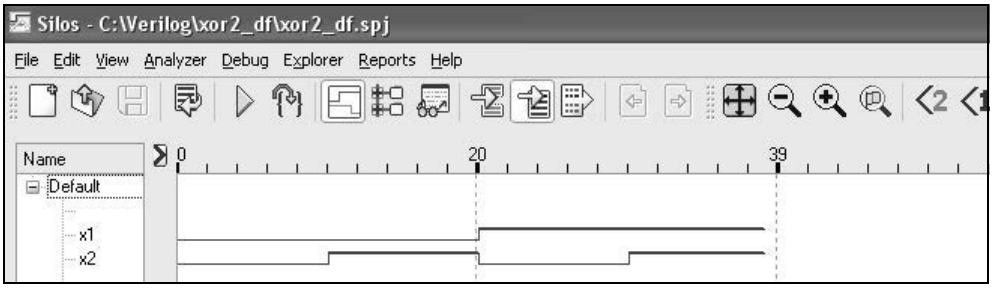


Figure 2.21 Waveforms for the 2-input exclusive-OR gate test bench of Figure 2.19.

2.5.2 Four 2-Input AND Gates with Delay

As a final example in the introduction to dataflow modeling, a circuit will be designed to illustrate a technique to introduce delays in logic gates. All delays in Verilog are specified in terms of time units. The delay value is indicated after the keyword **assign**. For example, a continuous assignment with delay is written as

$$\textbf{assign}\ \#5\ z_1 = x_1\ \&\ x_2;$$

The #5 indicates five time units. Whenever x_1 or x_2 changes value, the right-hand expression is evaluated and the value is assigned to the left-hand variable z_1 after five time units. The actual unit of time is specified by the **`timescale** compiler directive. For example, the statement

`timescale 10ns / 100ps

indicates that one time unit is specified as 10 nanoseconds (ns) with a precision of 100 picoseconds (ps). This property is called *inertial* delay.

Figure 2.22 illustrates a logic circuit with four 2-input AND gates. Two scalars — x_1 and x_2 — are specified as the inputs; the output of the circuit is a vector of four bits: z_1, where $z_1 = z_1[0], z_1[1], z_1[2]$, and $z_1[3]$. Figure 2.23 shows the Verilog code for Figure 2.22 using a time unit of 10 ns with a precision of 1 ns.

When input x_1 or x_2 changes value, the new value of the right-hand statement is assigned to the left-side after a delay of 20 ns. The test bench of Figure 2.24 lists six sets of vectors to be assigned to the inputs. The resulting waveforms are shown in Figure 2.25. Note that when x_1 or x_2 changes state, the output of the circuit is delayed by 20 ns.

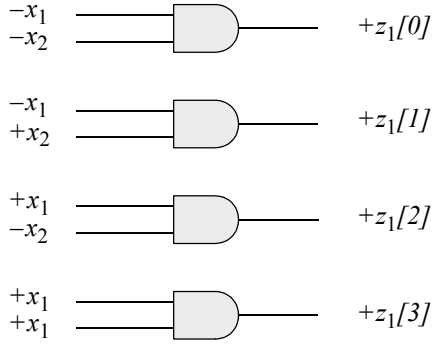


Figure 2.22 Four 2-input AND gates with scalar inputs and a vector output.

```
//dataflow with delay
`timescale 10ns / 1ns
module four_and_delay (x1, x2, z1);

input x1, x2;
output [3:0] z1;        //continued on next page
```

Figure 2.23 Verilog code for the logic circuit of Figure 2.22.

```
assign #2 z1[0] = ~x1 & ~x2;
assign #2 z1[1] = ~x1 & x2;
assign #2 z1[2] = x1 & ~x2;
assign #2 z1[3] = x1 & x2;

endmodule
```

Figure 2.23 (Continued)

```
//four_and_delay test bench
module four_and_delay_tb;

reg x1, x2;
wire [3:0] z1;

initial
$monitor ("x1 x2=%b, z1=%b",
          {x1, x2}, z1);

//apply input vectors
initial
begin
   #0 x1 = 1'b0;
      x2 = 1'b0;

   #5 x1 = 1'b1;
      x2 = 1'b0;

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

   #5 x1 = 1'b0;
      x2 = 1'b1;

   #5 x1 = 1'b0;
      x2 = 1'b1;

   #5 x1 = 1'b0;
      x2 = 1'b0;

   #5 $stop;
end

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

endmodule
```

Figure 2.24 Test bench for the Verilog code of Figure 2.23 for the circuit of Figure 2.22.

2.6 Introduction to Behavioral Modeling

Describing a module in *behavioral* modeling is an abstraction of the functional operation of the design. It does not describe the implementation of the design at the gate level. The outputs of the module are characterized by their relationship to the inputs. The behavior of the design is described using procedural constructs. These constructs are the **initial** statement and the **always** statement.

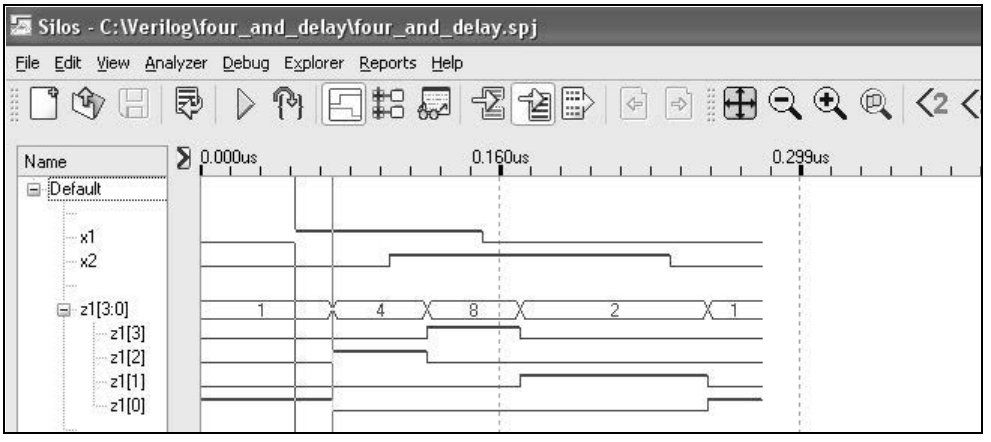


Figure 2.25 Waveforms for the four AND gates with delay using the test bench of Figure 2.24.

The **initial** statement is executed only once during a simulation — beginning at time 0 — and then suspends forever. The **always** statement also begins at time 0 and executes the statements in the **always** block repeatedly in a looping manner. Both statements use only register data types. Objects of the **reg** data types resemble the behavior of a hardware storage device because the data retains its value until a new value is assigned.

2.6.1 Three-Input OR Gate

Figure 2.26 shows a 3-input OR gate, which will be designed using behavioral modeling. The behavioral module is shown in Figure 2.27 using an **always** statement. The expression within the parentheses is called an *event control* or *sensitivity list*. Whenever a variable in the event control list changes value, the statements in the **begin** . . . **end** block will be executed; that is, if either x_1 or x_2 or x_3 changes value, the following statement will be executed:

$$z_1 = x_1 \mid x_2 \mid x_3;$$

where the symbol (|) signifies the logical OR operation.

If only a single statement appears after the **always** statement, then the keywords **begin** and **end** are not required. It is often useful, however, to include the **begin** and **end** keywords because additional statements may have to be added later. The **always** statement has a sequential block (**begin** . . . **end**) associated with an event control. The statements within a **begin** . . . **end** block execute sequentially and execution suspends when the last statement has been executed. When the sequential block completes execution, the **always** statement checks for another change of variables in the event control list.

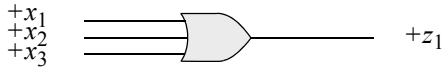


Figure 2.26 Three-input OR gate to be implemented using behavioral style modeling.

The test bench for the module is shown in Figure 2.28. As stated previously, the inputs for a test bench are of type **reg** because they retain their value until changed, and the outputs are of type **wire**. To ensure that all eight combinations of the inputs are tested, a register vector called *invect* is specified as four bits: 3 through 0, where bit 0 is the low-order bit. This guarantees that a vector of $x_1x_2x_3 = 111$ will be applied to the OR gate inputs. The inputs are applied in sequence, $x_1x_2x_3 = 000$ through 111. When an input vector of $x_1x_2x_3 = 1000$ is reached, the test in the **for** loop returns a value of false and the simulator exits the statements in the **begin** . . . **end** sequence. The binary outputs of the simulator are shown in Figure 2.29 listing the output value for z_1 for all combinations of inputs. The waveforms are shown in Figure 2.30.

```
//behavioral 3-input or gate
module or3 (x1, x2, x3, z1);

input x1, x2, x3;
output z1;

wire x1, x2, x3;
reg z1;

always @ (x1 or x2 or x3)
begin
   z1 = x1 | x2 | x3;
end

endmodule
```

Figure 2.27 Behavioral module for the 3-input OR gate of Figure 2.26.

The statements within the sequential block are called *blocking* statements because execution of the following statement is blocked until the current statement completes execution; that is, the statements within a sequential block execute sequentially. An

optional delay may be associated with a procedural assignment. There are two types of delays: interstatement and intrastatement.

```
//or3 test bench
module or3_tb;

reg x1, x2, x3;
wire z1;

//apply input vectors
initial
begin: apply_stimulus
   reg [3:0] invect;
   for (invect = 0; invect < 8; invect = invect + 1)
      begin
         {x1, x2, x3} = invect [3:0];
         #10 $display ("{x1x2x3} = %b, z1 = %b",
                  {x1, x2, x3}, z1);
      end
end

//instantiate the module into the test bench
or3 inst1 (
   .x1(x1),
   .x2(x2),
   .x3(x3),
   .z1(z1)
   );

endmodule
```

Figure 2.28 Test bench for the 3-input OR gate module of Figure 2.27.

```
{x1x2x3} = 000, z1 = 0
{x1x2x3} = 001, z1 = 1
{x1x2x3} = 010, z1 = 1
{x1x2x3} = 011, z1 = 1
{x1x2x3} = 100, z1 = 1
{x1x2x3} = 101, z1 = 1
{x1x2x3} = 110, z1 = 1
{x1x2x3} = 111, z1 = 1
```

Figure 2.29 Binary outputs for the test bench of Figure 2.28.

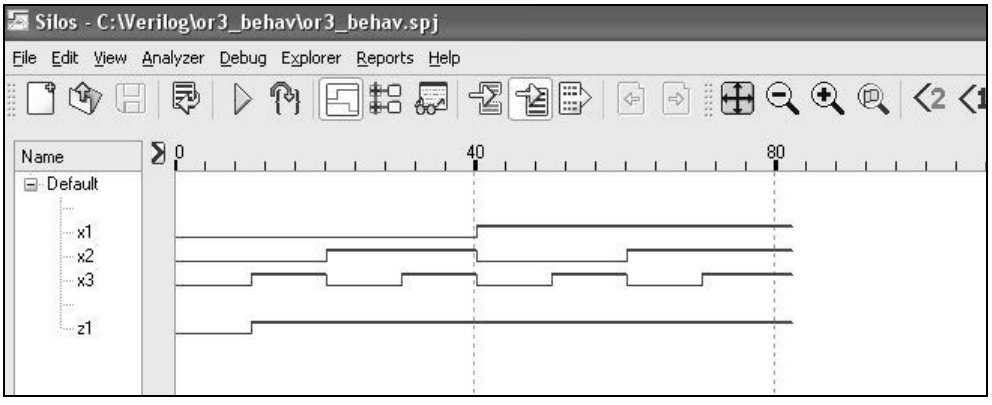


Figure 2.30 Waveforms for the test bench of Figure 2.28 for the 3-input OR gate module of Figure 2.26.

An *interstatement* delay is the delay by which a statement's execution is delayed; that is, it is the delay between statements. The following is an example of an interstatement delay:

$$z_1 = x_1 \mid x_2$$
$$\#5\ z_2 = x_1 \ \&\ x_2$$

When the first statement has completed execution, a delay of five time units will be taken before executing the second statement. An *intrastatement* delay is a delay on the right-hand side of the statement and indicates that the right-hand side is to be evaluated, wait the specified number of time units, and then assign the value to the left-hand side. This can be used to simulate logic gate delays. The following is an example of an intrastatement delay:

$$z_1 = \#5\ \ x_1 \ \&\ \ x_2$$

The statement evaluates the logical function x_1 AND x_2, waits five time units, then assigns the result to z_1. If no delay is specified in a procedural assignment, then zero delay is the default delay and the assignment occurs instantaneously.

Example 2.1 Following is an example of generating waveforms using intrastatement delays in a behavioral module. There are two inputs x_1 and x_2 that will be assigned values based on a timescale of 10 ns / 1 ns. The behavioral module is shown in Figure 2.31 and the waveforms in Figure 2.32.

```
//behavioral intrasegment delay example
`timescale 10ns / 1ns
module intra_stmt_dly_delay (x1,x2);

output x1, x2;
reg x1, x2;

initial
begin
   x1 = #0   1'b0;
   x2 = #0   1'b0;

   x1 = #1   1'b1;
   x2 = #0.5  1'b1;

   x1 = #1   1'b0;
   x2 = #2   1'b0;

   x1 = #1   1'b1;
   x2 = #2   1'b1;

   x1 = #2   1'b0;
   x2 = #1   1'b0;
end
endmodule
```

Figure 2.31 Behavioral module to generate waveforms using intrastatement delays.

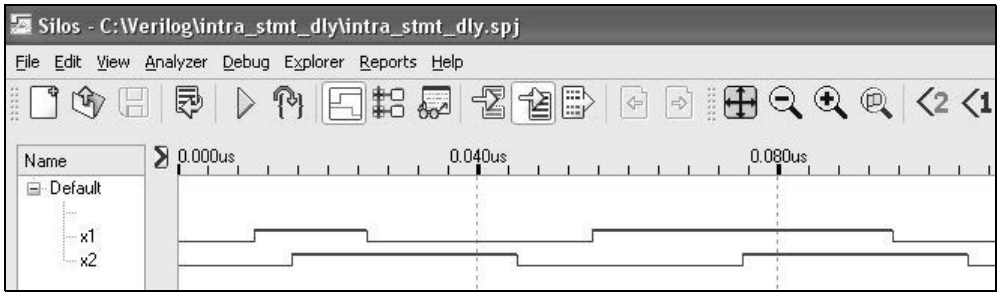


Figure 2.32 Waveforms showing intrastatement delays for Figure 2.31.

2.6.2 Four-Bit Adder

The top-down structural architecture of a 4-bit ripple adder was shown in Section 2.3. A similar 4-bit adder will now be designed using behavioral modeling to illustrate the differences between the two modeling techniques. Figure 2.33 depicts the block diagram of the adder, but does not indicate the internal logic. Figure 2.34 shows the behavioral module for the 4-bit adder. Note that only the *behavior* of the adder is specified, not the internal logic elements as in the structural architecture. The behavior is specified as:

$$sum = a + b + cin$$

The behavioral module does not indicate the specifics as to how to design the adder; the Verilog code simply specifies the functionality of the module. Also, the *sum* variable is specified as a 5-bit vector to include the carry-out of the adder. The augend *a[3:0]* and addend *b[3:0]* remain as 4-bit vectors with the carry-in (*cin*) as a scalar.

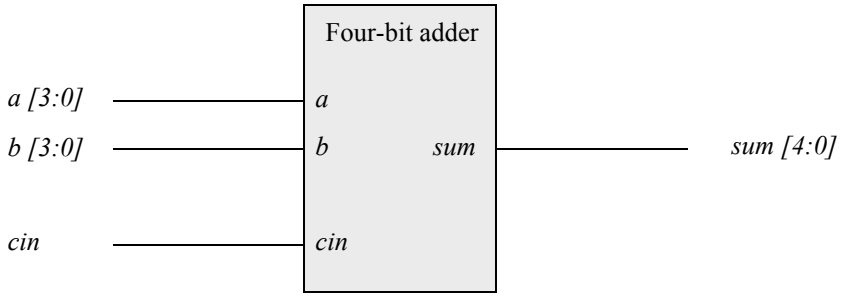


Figure 2.33 Block diagram of a 4-bit adder.

Inputs are declared as type **wire** and outputs as type **reg** in behavioral modeling. Operands *a* and *b* are specified as 4-bit vectors: *a[3]*, . . . , *a[0]* and *b[3]*, . . . , *b[0]*, where *a[0]* and *b[0]* are the low-order bits of *a* and *b*, respectively. The **always** statement continuously checks for a change of value to the operands and the carry-in. If *a* or *b* or *cin* change value, then the statement in the **begin** . . . **end** block is executed. The right-hand side of the expression is evaluated and then assigned to *sum*.

The test bench is shown in Figure 2.35, in which six sets of input vectors are applied to the operands. The binary outputs obtained from the test bench are shown in Figure 2.36 and the corresponding waveforms in Figure 2.37. The values of operands *a*, *b*, and *sum* are given in hexadecimal. By double-clicking the *sum[4:0]* entry in the SILOS Data Analyzer, the individual bits of the sum can be displayed.

```
//behavioral 4-bit adder
module adder_4_behav (a, b,
         cin, sum);

input [3:0] a, b;
input cin;

output [4:0] sum;

wire [3:0] a, b;
wire cin;
reg [4:0] sum;

always @ (a or b or cin)
begin
   sum = a + b + cin;
end

endmodule
```

Figure 2.34 Behavioral module for a 4-bit adder.

```
//behavioral 4-bit adder test
//bench
module adder_4_behav_tb;

reg [3:0] a, b;
reg cin;

wire [4:0] sum;

//display variables
initial
$monitor ("a b cin = %b_%b_%b,
sum = %b", a, b, cin, sum);

//apply input vectors
initial
begin
   #0    a = 4'b0011;
         b = 4'b0100;
         cin = 1'b0;

   #10   a = 4'b1100;
         b = 4'b0011;
         cin = 1'b0;

   #10   a = 4'b0111;
         b = 4'b0110;
         cin = 1'b1;

   #10   a = 4'b1001;
         b = 4'b0111;
         cin = 1'b1;

   #10   a = 4'b1101;
         b = 4'b0111;
         cin = 1'b1;

   #10   a = 4'b1111;
         b = 4'b0110;
         cin = 1'b1;

   #10   $stop;
end

//instantiate the module into
//the test bench
adder_4_behav inst1 (
   .a(a),
   .b(b),
   .cin(cin),
   .sum(sum)
   );

endmodule
```

Figure 2.35 Test bench for the 4-bit adder module of Figure 2.34.

```
a b cin = 0011_0100_0, sum = 00111
a b cin = 1100_0011_0, sum = 01111
a b cin = 0111_0110_1, sum = 01110
a b cin = 1001_0111_1, sum = 10001
a b cin = 1101_0111_1, sum = 10101
a b cin = 1111_0110_1, sum = 10110
```

Figure 2.36 Binary outputs for the 4-bit adder obtained from the test bench of Figure 2.35.

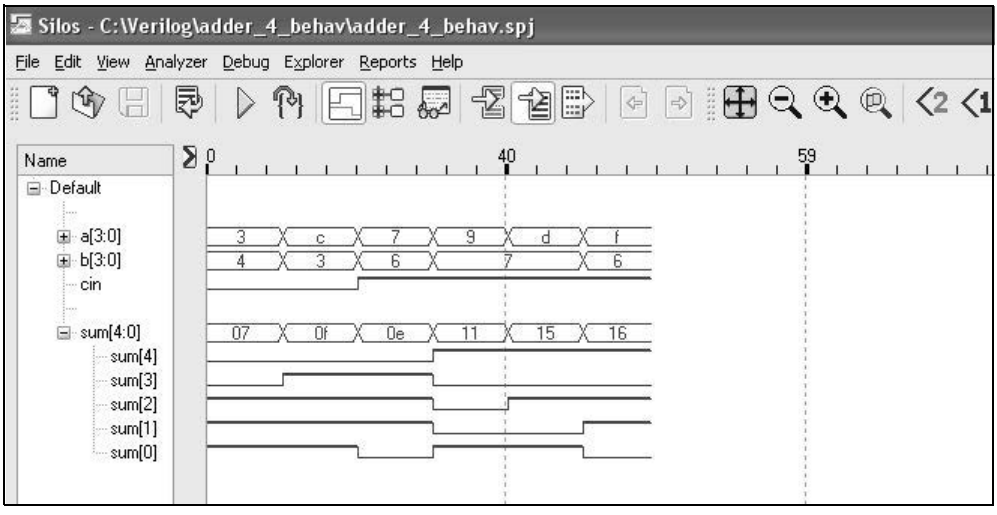


Figure 2.37 Waveforms for the 4-bit adder module of Figure 2.34.

2.6.3 Modulo-16 Synchronous Counter

Figure 2.5 showed the gate-level design of a modulo-16 synchronous counter. The same counter will now be designed using behavioral modeling. This is a much simpler approach because the counter is not designed using Karnaugh maps and equations to provide discrete logic gates and flip-flops. Also, the storage elements are not specified. Verilog is simply given the counting sequence; the counter is then designed by Verilog according to the specifications. Figure 2.38 shows the symbol for the modulo-16 counter.

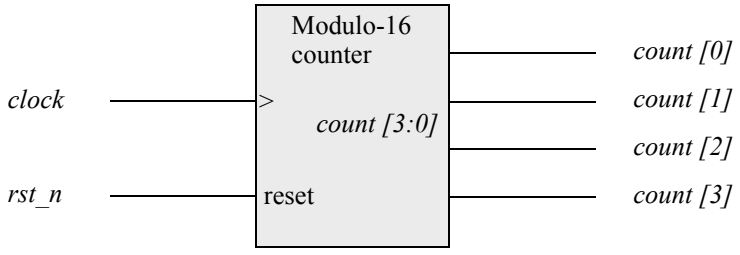


Figure 2.38 Symbol for a modulo-16 counter.

The Verilog module is shown in Figure 2.39 using nonblocking statements in the procedural block. There are three ports: *clk*, *rst_n*, and *count*, which is defined as a 4-bit vector, where *count [0]* is the low-order bit. The **always** statement contains two events in the sensitivity list: *posedge clk* and *negedge rst_n*. When the system clock transitions from a low level to a high level, the counter is incremented. The % symbol indicates a modulus or remainder such that the counter increments from 0 to 15, then begins again at a count of 0. The variable *rst_n* is the conventional way to indicate an active-low reset signal. If the reset signal is at a low voltage level, then the counter is reset to a value of *count* = 0000.

```
//behavioral modulo-16 counter
module ctr_mod_16 (clk, rst_n, count);

input clk, rst_n;
output [3:0] count;

wire clk, rst_n;
reg [3:0] count;

//define counting sequence
always @ (posedge clk or negedge rst_n)
begin
   if (rst_n == 0)
      count <= 4'b0000;
   else
      count <= (count + 1) % 16;
end
endmodule
```

Figure 2.39 Verilog code for a modulo-16 synchronous counter.

The test bench is shown in Figure 2.40. When a change occurs on *count*, the **&monitor** task will display the current count as a binary value. The reset signal is conditioned to an active-low level for five time units initially, then made inactive five time units later. The clock signal is initially set to 0, then alternates between positive and negative levels indefinitely. This is accomplished by the keyword **forever**, which assigns the clock the value of the complemented clock every 10 time units. The duration of simulation is defined to be 320 time units, which is sufficient duration to generate a counting sequence of 0000, ... , 1111, 0000 to show the modulo-16 counting sequence.

The binary outputs are shown in Figure 2.41 as obtained from the **$monitor** task in the test bench. The waveforms are shown in Figure 2.42 and indicate the same counting sequence, but in hexadecimal notation. By clicking the (+) sign to left of the *count [3:0]* signal name or by double-clicking the *count [3:0]* signal name, the individual bits of the counter are obtained.

```
//modulo-16 counter test bench
module ctr_mod_16_tb;

reg clk, rst_n;
wire [3:0] count;

initial
$monitor ("count=%b", count);

//define reset
initial
begin
   #0 rst_n = 1'b0;
   #5 rst_n = 1'b1;
end

//define clock
initial
begin
   #0 clk = 1'b0;
   forever
      #10clk = ~clk;
end
```

```
//define length of
//simulation
initial
begin
   #320 $stop;
end

//instantiate the module
//into the test bench
ctr_mod_16 inst1 (
   .clk(clk),
   .rst_n(rst_n),
   .count(count)
   );
endmodule
```

Figure 2.40 Test bench for the modulo-16 counter of Figure 2.39.

```
count = 0000                    count = 1001
count = 0001                    count = 1010
count = 0010                    count = 1011
count = 0011                    count = 1100
count = 0100                    count = 1101
count = 0101                    count = 1110
count = 0110                    count = 1111
count = 0111                    count = 0000
count = 1000
```

Figure 2.41 Binary outputs for the modulo-16 counter of Figure 2.39 obtained from the test bench of Figure 2.40.

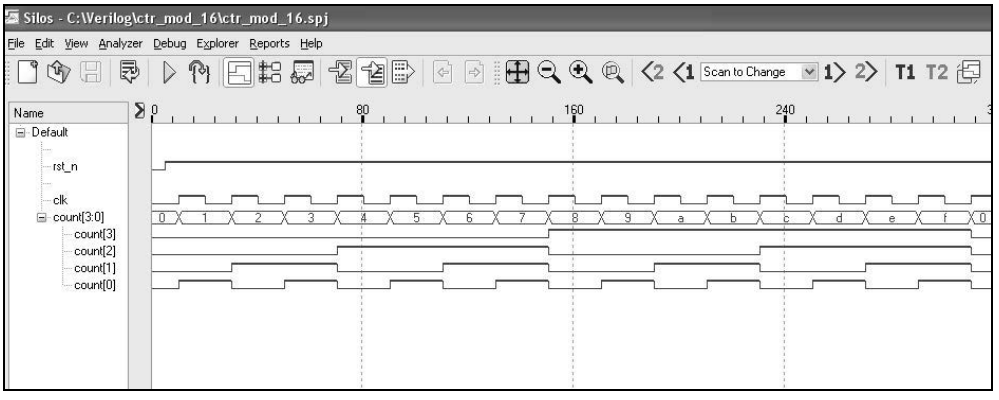


Figure 2.42 Waveforms for the modulo-16 counter of Figure 2.39 obtained from the test bench of Figure 2.40.

2.7 Introduction to Structural Modeling

Structural modeling is a top-level design that is synthesized by instantiating lower-level logic modules into a larger structural module. Each of the lower-level modules (submodules) must have previously been compiled and tested for correct functional operation. A structural module can be described using built-in gate primitives, user-defined primitives, or module instances in any combination. Interconnections between instances are specified using nets.

2.7.1 Sum-of-Products Implementation

This section will implement the sum-of-products expression of Equation 2.4 using structural modeling. The structural module will instantiate the following four sub-modules: *and2*, *and3*, *and4*, and *or3* as shown in Figure 2.43. The ports of the sub-modules are indicated by small black squares and correspond to the port names in the respective modules. The module name is shown outside the dashed line and the instance name associated with the module is shown within the dashed line. The structural module is called *sop*.

$$z_1 = x_1x_2 + x_2x_3'x_4 + x_1'x_2'x_3x_4 \tag{2.4}$$

The modules for *and2*, *and3*, *and4*, and *or3* are shown in Figure 2.44. The structural module is shown in Figure 2.45, which instantiates the four sub-modules. The instantiation modules and the instance names directly correspond to those shown in Figure 2.43. For example, in instance 2 of the structural module, port $.x_1$ of the *and3* module connects to x_2 $(+x_2)$ of the structural module; port $.x_2$ connects to $\sim x_3$ $(-x_3)$, port $.x_3$ connects to x_4 $(+x_4)$; and port $.z_1$ connects to *net2*. This represents the port connections and internal nets of the structural module. The structural module test bench is shown in Figure 2.46.

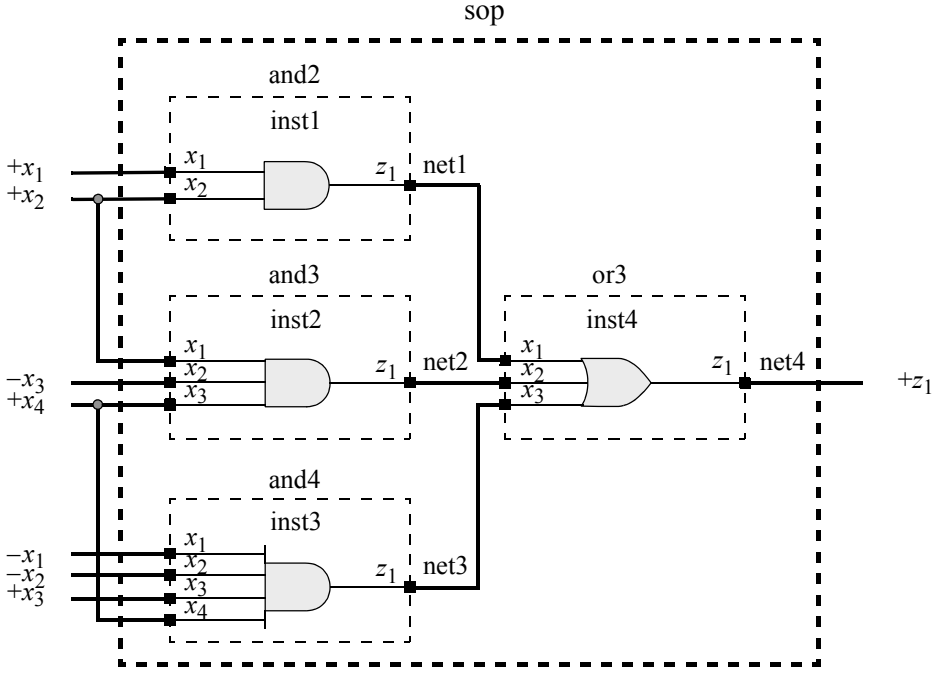


Figure 2.43 Structural module for the sum-of-products equation of Equation 2.4.

```
//dataflow 2-input and gate
module and2 (x1, x2, z1);

input x1, x2;
output z1;

wire x1, x2;
wire z1;

assign z1 = x1 & x2;

endmodule
```

```
//dataflow 4-input and gate
module and4 (x1, x2, x3, x4,
             z1);

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

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

assign z1 = x1 & x2 & x3 & x4;
endmodule
```

```
//dataflow 3-input and gate
module and3 (x1, x2, x3, z1);

input x1, x2, x3;
output z1;

wire x1, x2, x3;
wire z1;

assign z1 = x1 & x2 & x3;

endmodule
```

```
//behavioral 3-input or gate
module or3 (x1, x2, x3, z1);

input x1, x2, x3;
output z1;

wire x1, x2, x3;
reg z1;

always @ (x1 or x2 or x3)
begin
   z1 = x1 | x2 | x3;
end
endmodule
```

Figure 2.44 Modules for *and2*, *and3*, *and4*, and *or3* that will be instantiated into the sum-of-products structural module of Figure 2.45.

The binary values for output z_1 as obtained from the test bench are shown in Figure 2.47 for all combinations of the input variables. Output z_1 is a logical 1 for the conditions that satisfy Equation 2.4 and can be verified by the Karnaugh map of Figure 2.48. The resulting waveforms are shown in Figure 2.49.

```
//structural sum of products
module sop (x1, x2, x3, x4,
            z1);

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

```
//define internal nets
wire net1, net2, net3, net4;

wire z1;

assign z1 = net4;
//continued on next page
```

Figure 2.45 Structural module for the sum-of-products equation of Equation 2.4.

```
//instantiate the gate modules
//into the structural module
and2 inst1 (
   .x1(x1),
   .x2(x2),
   .z1(net1)
   );


and3 inst2 (
   .x1(x2),
   .x2(~x3),
   .x3(x4),
   .z1(net2)
   );
```

```
and4 inst3 (
   .x1(~x1),
   .x2(~x2),
   .x3(x3),
   .x4(x4),
   .z1(net3)
   );


or3 inst4 (
   .x1(net1),
   .x2(net2),
   .x3(net3),
   .z1(net4)
   );

endmodule
```

Figure 2.45 (Continued)

```
//structural sum of products test bench
module sop_tb;


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


//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, z1 = %b",
                      {x1, x2, x3, x4}, z1);
     end
end
//continued on next page
```

Figure 2.46 Test bench for the structural module of Figure 2.45.

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

Figure 2.46 (Continued)

```
{x1x2x3x4} = 0000, z1 = 0
{x1x2x3x4} = 0001, z1 = 0
{x1x2x3x4} = 0010, z1 = 0
{x1x2x3x4} = 0011, z1 = 1
{x1x2x3x4} = 0100, z1 = 0
{x1x2x3x4} = 0101, z1 = 1
{x1x2x3x4} = 0110, z1 = 0
{x1x2x3x4} = 0111, z1 = 0
```

```
{x1x2x3x4} = 1000, z1 = 0
{x1x2x3x4} = 1001, z1 = 0
{x1x2x3x4} = 1010, z1 = 0
{x1x2x3x4} = 1011, z1 = 0
{x1x2x3x4} = 1100, z1 = 1
{x1x2x3x4} = 1101, z1 = 1
{x1x2x3x4} = 1110, z1 = 1
{x1x2x3x4} = 1111, z1 = 1
```

Figure 2.47 Binary outputs for the test bench of Figure 2.46.

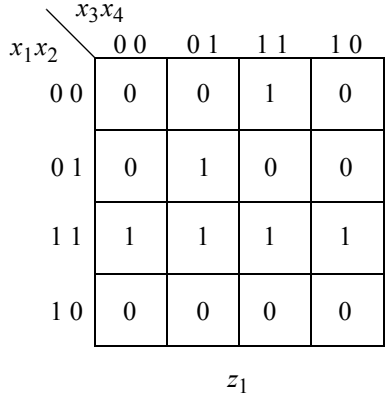


Figure 2.48 Karnaugh map for the sum-of-products expression of Equation 2.4.

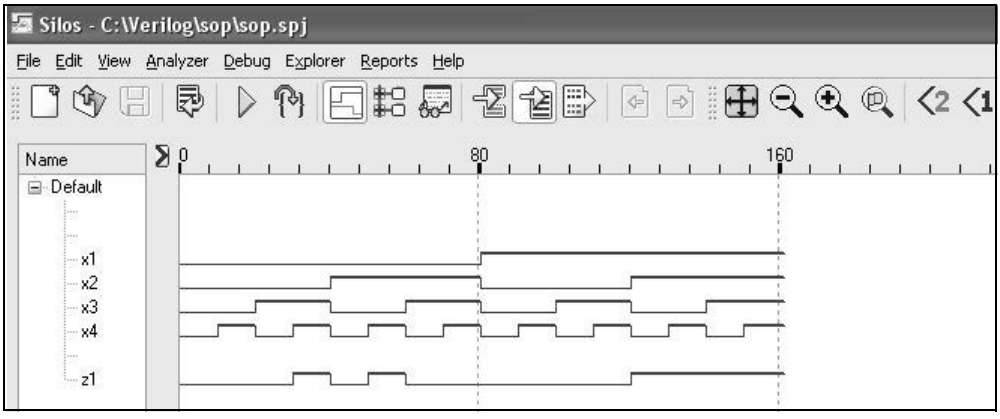


Figure 2.49 Waveforms for Figure 2.45 using the test bench of Figure 2.46.

2.7.2 Full Adder

A full adder will now be synthesized with two half adders and an OR gate using structural modeling. Recall that a full adder can be designed using a top-down approach as shown in Figure 2.50. The equations for a half adder and a full adder are shown in Equation 2.5 and 2.6, respectively, and the corresponding gate level designs in Figure 2.51 and Figure 2.52.

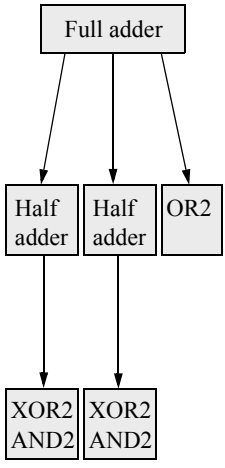


Figure 2.50 Top-down design for a full adder using two half adders.

$$sum = a \oplus b$$
$$cout = ab \tag{2.5}$$

$$sum = a \oplus b \oplus cin$$
$$cout = cin(a \oplus b) + ab \tag{2.6}$$

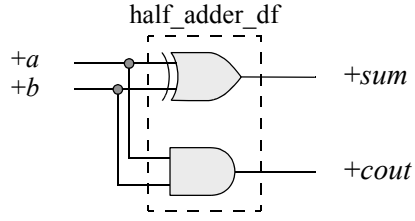


Figure 2.51 Logic diagram for a half adder to be instantiated into a full adder structural design.

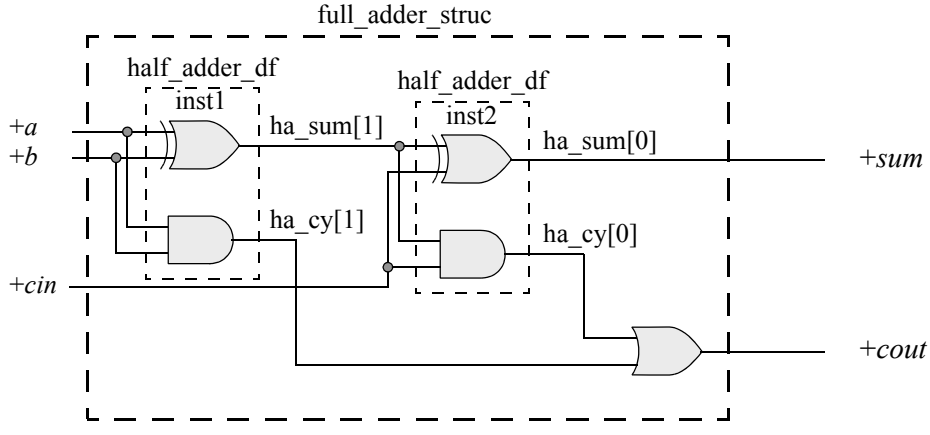


Figure 2.52 Logic diagram for a full adder using two half adders instantiated into a structural module.

The half adder module is named *half_adder_df*, which is then instantiated two times into the structural module *full_adder_struc*. In the full adder module, internal wires are specified for the sum and carry-out of the two half adders. The sum from the two half adders is a 2-bit vector with bits named *ha_sum[1]* and *ha_sum[0]*, where bit 0 is the low-order bit. The carry-out from the two half adders is a 2-bit vector with bits named *ha_cy[1]* and *ha_cy[0]*, where bit 0 is the low-order bit.

The Verilog code for the half adder is shown in Figure 2.53, where the symbol (^) is the exclusive-OR function and (&) is the AND function. As stated previously, all modules that are to be instantiated into a structural module should be tested for correct functional operation before they are instantiated. The test bench for the half adder is shown in Figure 2.54 in which all combinations of the two operands are applied to the inputs.

The **$monitor** system task continuously monitors the values of the variables specified in the parameter list. When any of the variables change value, all of the variables are displayed. The test bench displays the variables in the binary radix (%b). The resulting binary outputs are shown in Figure 2.55. The **$stop** task provides a stop during simulation. System tasks are covered in detail in Chapter 10. The waveforms shown in Figure 2.56 display the binary variables in a graphical representation.

The full adder structural module is shown in Figure 2.57. In instantiation *inst1*, input port *.a* of the half adder connects to input *a* (+*a*) of the full adder structural module and port *.b* connects to input *b* (+*b*). The output port *.sum* connects to net *ha_sum[1]* in the structural module, which is an output of the half adder and declared as a **wire**. The carry-out *cout* of the half adder connects to net *ha_cy[1]* in the structural module.

In instantiation *inst2*, input port *.a* of the half adder connects to net *ha_sum[1]* and port *.b* connects to input *cin* (+*cin*) of the full adder module. The *sum* and *cout* ports of the half adder connect to nets *ha_sum[0]* and *ha_cy[0]* of the full adder, respectively. The two continuous assignment statements assign output *sum* of the full adder the value of *ha_sum[0]* and *cout* the value of the logical OR of *ha_cy[0]* and *ha_cy[1]*.

```
//dataflow half_adder
module half_adder_df (a, b, sum, cout);

input a, b;              //list which are input
output sum, cout;        //list which are output

wire a, b, sum, cout;    //all are wire

assign sum = a ^ b;
assign cout = a & b;

endmodule
```

Figure 2.53 Verilog code for the half adder of Figure 2.51.

```
//dataflow half_adder test bench
module half_adder_df_tb;

reg a, b;             //inputs are reg for test bench
wire sum, cout;       //outputs are wire for test bench

initial
$monitor ("ab = %b, sum = %b, cout = %b",
         {a, b}, sum, cout);

initial
begin
   #0 a = 1'b0;
      b = 1'b0;

   #10a = 1'b0;
      b = 1'b1;

   #10a = 1'b1;
      b = 1'b0;

   #10a = 1'b1;
      b = 1'b1;

   #10   $stop;
end

//instantiate the dataflow module into the test bench
half_adder_df inst1 (
   .a(a),
   .b(b),
   .sum(sum),
   .cout(cout)
   );
endmodule
```

Figure 2.54 Test bench for the half adder module of Figure 2.53.

```
ab = 00, sum = 0, cout = 0
ab = 01, sum = 1, cout = 0
ab = 10, sum = 1, cout = 0
ab = 11, sum = 0, cout = 1
```

Figure 2.55 Binary outputs obtained from the test bench of Figure 2.54 for the half adder module of Figure 2.53.

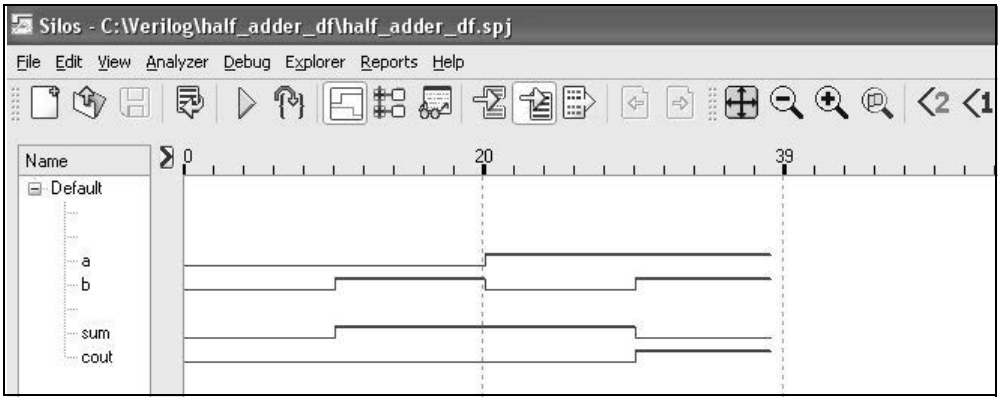


Figure 2.56 Waveforms for the half adder of Figure 2.53.

```
//structural full adder
module full_adder_struc (a, b, cin, sum, cout);

input a, b, cin;
output sum, cout;

wire [1:0] ha_sum, ha_cy;

//instantiate the half adder
half_adder_df inst1 (
  .a(a),
  .b(b),
  .sum(ha_sum[1]),
  .cout(ha_cy[1])
  );

half_adder_df inst2 (
  .a(ha_sum[1]),
  .b(cin),
  .sum(ha_sum[0]),
  .cout(ha_cy[0])
  );

assign sum = ha_sum[0];
assign cout = ha_cy[0] | ha_cy[1];

endmodule
```

Figure 2.57 Structural module for the full adder.

The test bench for the full adder structural module is shown in Figure 2.58 in which all possible combinations of three variables are applied to the inputs. The binary outputs are shown in Figure 2.59 and the waveforms in Figure 2.60.

```
//structural full adder test
//bench
module full_adder_struc_tb;

//inputs are reg for tb
reg a, b, cin;

//outputs are wire for tb
wire sum, cout;

initial
$monitor ("ab = %b, cin = %b,
          sum = %b, cout = %b",
          {a, b}, cin, sum,
          cout);

initial
begin
#0 a = 1'b0;
   b = 1'b0;
   cin = 1'b0;

#10a = 1'b0;
   b = 1'b0;
   cin = 1'b1;

#10a = 1'b0;
   b = 1'b1;
   cin = 1'b0;

#10a = 1'b0;
   b = 1'b1;
   cin = 1'b1;

#10a = 1'b1;
   b = 1'b0;
   cin = 1'b0;

#10a = 1'b1;
   b = 1'b0;
   cin = 1'b1;

#10a = 1'b1;
   b = 1'b1;
   cin = 1'b0;

#10a = 1'b1;
   b = 1'b1;
   cin = 1'b1;

#10 $stop;
end

//instantiate the module
//into the test bench
full_adder_struc inst1 (
.a(a),
.b(b),
.cin(cin),
.sum(sum),
.cout(cout)
);

endmodule
```

Figure 2.58 Test bench for the full adder module of Figure 2.57.

```
ab = 00, cin = 0, sum = 0, cout = 0
ab = 00, cin = 1, sum = 1, cout = 0
ab = 01, cin = 0, sum = 1, cout = 0
ab = 01, cin = 1, sum = 0, cout = 1
ab = 10, cin = 0, sum = 1, cout = 0
ab = 10, cin = 1, sum = 0, cout = 1
ab = 11, cin = 0, sum = 0, cout = 1
ab = 11, cin = 1, sum = 1, cout = 1
```

Figure 2.59 Binary outputs for the test bench of Figure 2.58.

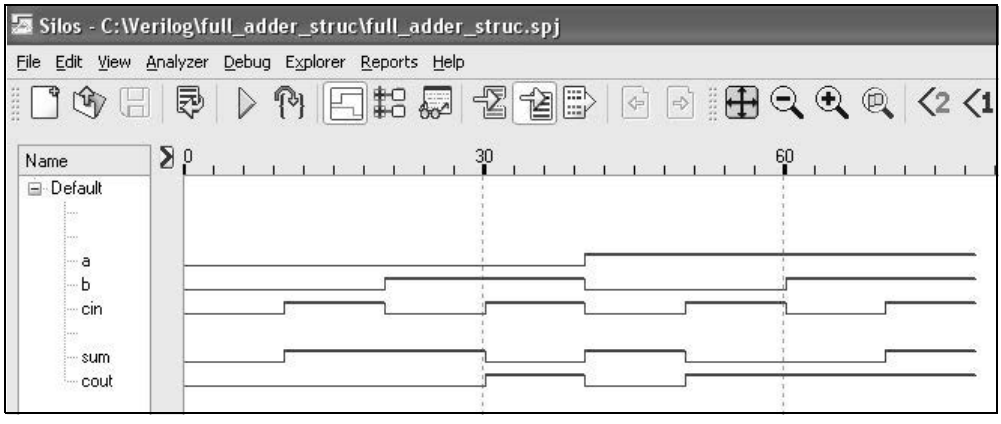


Figure 2.60 Waveforms for the test bench of Figure 2.58 for the structural full adder of Figure 2.57.

2.7.3 Four-Bit Ripple Adder

The 4-bit ripple adder of Section 2.3 will now be designed using structural modeling. The full adder of Section 2.7.2 will be instantiated four times into a structural module to produce a 4-bit ripple adder. Recall that a full adder adds the two operand bits plus the carry-out from the previous lower-order stage. Figure 2.61 shows the logic symbol for a full adder. In a 4-bit ripple adder, the carry-out from stage$_i$ becomes the carry-in to stage$_{i+1}$. The 4-bit ripple adder will be designed using 4-bit vectors for the two operands — augend and addend — and a 4-bit vector for the sum.

The full adder will be verified for correct operation by means of a test bench, then instantiated into the structural module, which will also be verified for correct

functionality. The binary outputs and waveforms will be obtained for several combinations of the augend and addend.

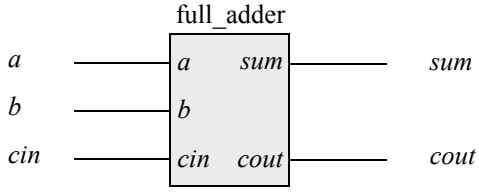


Figure 2.61 Logic symbol for a full adder.

Figure 2.62 shows the Verilog code for the full adder. The module is called *full_adder* and has five ports: *a*, *b*, *cin*, *sum*, and *cout*. Figure 2.63 shows the module for the test bench. The test bench provides all combinations of the three bits assigned to augend *a*, addend *b*, and the carry-in *cin*. The binary outputs and waveforms are shown in Figure 2.64 and Figure 2.65, respectively.

```
//dataflow full adder
module full_adder (a, b, cin, sum, cout);

//list inputs and outputs
input a, b, cin;
output sum, cout;

//define wires
wire a, b, cin;
wire sum, cout;

//continuous assign
assign sum = (a ^ b) ^ cin;
assign cout = cin & (a ^ b) | (a & b);

endmodule
```

Figure 2.62 Verilog code for a full adder.

When the full adder has been verified for correct functional operation, it is then instantiated into the 4-bit ripple adder structural model. When the operation of the ripple adder has been verified, it can then be used as a 4-bit-slice module to design a 16-bit or larger adder. For a 16-bit adder, the 4-bit slice is instantiated four times into a structural module to produce a 16-bit adder. Because Verilog supports a mixture of modeling styles within a module, the 4-bit adder and 16-bit adder can be designed using built-in primitives and continuous assignment statements.

In a later chapter, the design of a carry-lookahead adder will be presented. A carry-lookahead adder negates the drawback of the slow carry propagation delay in a ripple adder in which the carry is propagated serially between stages. The carry-lookahead method adds additional logic in each adder stage such that the carry-in to any stage is a function only of the two operand bits of that stage and the low-order carry-in to the adder. The carry-in to a stage is no longer the result of the carry-out from the previous lower-order stage.

```
//full adder test bench
module full_adder_tb;

reg a, b, cin;
wire sum, cout;

//apply input vectors
initial
begin: apply_stimulus
   reg [3:0] invect;
   for (invect = 0; invect < 8; invect = invect + 1)
      begin
         {a, b, cin} = invect [3:0];
         #10 $display ("{abcin} = %b, sum = %b, cout = %b",
                  {a, b, cin}, sum, cout);
      end
end

//instantiate the module into the test bench
full_adder inst1 (
   .a(a),
   .b(b),
   .cin(cin),
   .sum(sum),
   .cout(cout)
   );

endmodule
```

Figure 2.63 Test bench for the full adder module of Figure 2.62.

```
{abcin} = 000, sum = 0, cout = 0
{abcin} = 001, sum = 1, cout = 0
{abcin} = 010, sum = 1, cout = 0
{abcin} = 011, sum = 0, cout = 1
{abcin} = 100, sum = 1, cout = 0
{abcin} = 101, sum = 0, cout = 1
{abcin} = 110, sum = 0, cout = 1
{abcin} = 111, sum = 1, cout = 1
```

Figure 2.64 Binary outputs for the full adder of Figure 2.62 obtained from the test bench of Figure 2.63.

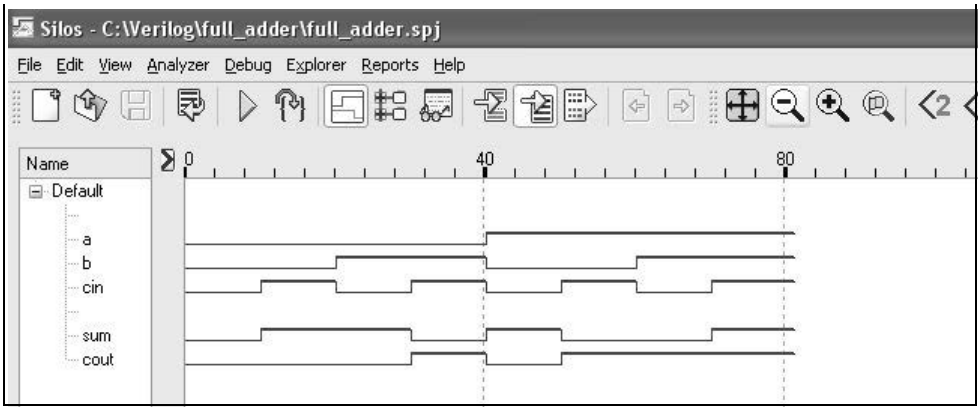


Figure 2.65 Waveforms for the full adder of Figure 2.62 obtained from the test bench of Figure 2.63.

The logic diagram for the 4-bit ripple adder is shown in Figure 2.66. The module for the 4-bit ripple adder is shown in Figure 2.67 using four instantiations of the Verilog code of Figure 2.62. The name of the module is *ripple_adder_4*; the ports are *a*, *b*, *cin*, *sum*, and *cout*. The keywords **input** and **output** tell Verilog which ports are used for external communication. Input ports are defaulted to **wire** in Verilog; however, the keyword **wire** is listed for all ports for completeness. The carry-out (*cout*) of the 4-bit adder is assigned the value of the internal carry *c[3]*.

The full adder is instantiated four times into the structural module of the ripple adder. In all instantiations, the name of the full adder (*full_adder*) is used. This is followed by the instance name; for example, *inst1* and *inst2*. The form *.a*(*a[0]*) indicates a connection between port *a* in the instance module (*full_adder*) to the expression *a[0]*

in the test bench. This method is *instantiation by name* and eliminates the need to know the order of the ports in the instantiated module. This is particularly useful when a large number of ports must be instantiated. The other method is *instantiation by position*, which is prone to errors when many ports are to be instantiated in the same order as indicated in the instance module.

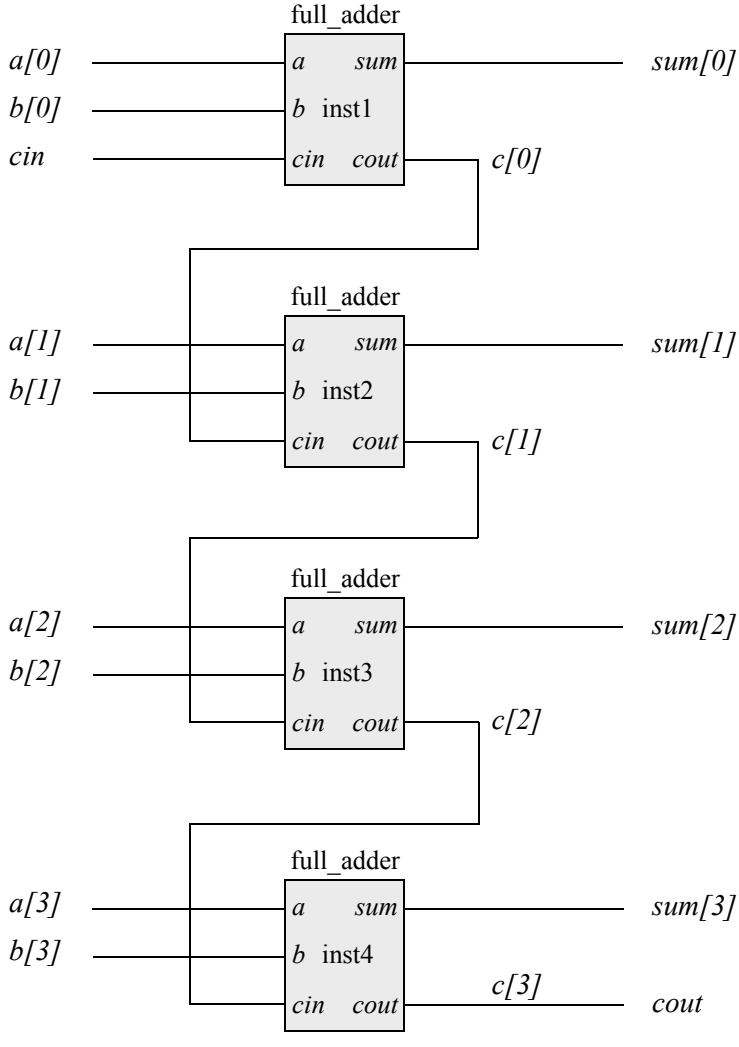


Figure 2.66 Logic diagram for a 4-bit ripple adder.

In instantiation *inst2* of the *ripple_adder_4* module, note that port *.a* of the full adder connects to *a[1]* of the ripple adder module. This is also shown in the logic

diagram of Figure 2.66. In a similar manner, in *inst3*, the *.a* port of the instance module connects to *a[2]* of the ripple adder module. The carry-out (*.cout*) of *inst3* connects to *c[2]* of the structural module. The final carry-out (*cout*) of the full adder in *inst4* connects to the carry-out of the 4-bit ripple adder.

The test bench for the 4-bit ripple adder module is shown in Figure 2.68. The name of the module under test is normally used as the name of the test bench module (with *_tb* appended). This allows for ease of cross-referencing. All Verilog modules are saved with the .v extension. Inputs for test benches are declared as type **reg** because they retain their values during simulation. Outputs are declared as **wire** for test benches.

As explained previously, Verilog provides a mechanism to monitor a variable when it changes state. This facility is provided by the **$monitor** task. All parameters within the quotation marks are displayed in the radix indicated. The parameter values are obtained from the variables to the right of the quotation marks, in the same positional relationship. The **$monitor** continuously monitors the values of the variables specified in the parameter list (within the quotation marks) and displays all the parameters in the list whenever any variable changes value.

```
//dataflow 4-bit ripple adder
module ripple_adder_4 (a, b, cin, sum, cout);

input [3:0]a, b;
input cin;

output [3:0] sum;
output cout;

wire [3:0]a, b;
wire cin;
wire [3:0] sum;
wire cout;
wire [3:0]c;//define internal carries

assign cout = c[3];
//instantiate the full adder
full_adder inst1 (
   .a(a[0]),
   .b(b[0]),
   .cin(cin),
   .sum(sum[0]),
   .cout(c[0])
   );

//continued on next page
```

Figure 2.67 Structural module for a 4-bit ripple adder.

```
full_adder inst2 (
   .a(a[1]),
   .b(b[1]),
   .cin(c[0]),
   .sum(sum[1]),
   .cout(c[1])
   );

full_adder inst3 (
   .a(a[2]),
   .b(b[2]),
   .cin(c[1]),
   .sum(sum[2]),
   .cout(c[2])
   );

full_adder inst4 (
   .a(a[3]),
   .b(b[3]),
   .cin(c[2]),
   .sum(sum[3]),
   .cout(c[3])
   );

endmodule
```

Figure 2.67 (Continued)

Following the **$monitor** task is a section of code that applies a sequence of input vectors to the module under test. Six sets of input vectors are shown. The first set of vectors is applied at time 0, where the augend *a* is assigned a value of 3. The statement *a* = 4'b0011 specifies that the field for the value of *a* is four bits wide, the radix is binary, and the value is 3. The addend *b* is assigned a value of 4; the carry-in *c* is 0. The result of the first set of vectors is: *sum* = 7, *cout* = 0 as shown in the binary outputs of Figure 2.69. In the last set of input vectors, *a* = 15, *b* = 6, and *cin* = 1. This generates the following results: *sum* = 6, *cout* = 1, where *cout* has a weight of 2^4.

Simulation stops 10 time units after the last set of inputs has been applied. The unit under test is instantiated by name, not by position. Thus, port *.a* in the ripple adder module connects to *a* of the test bench. The waveforms are shown in Figure 2.70.

```
//4-bit ripple adder test
//bench
module ripple_adder_4_tb;

reg [3:0]a, b;
reg cin;

wire [3:0] sum;
wire cout;

//display variables
initial
$monitor ("a b cin = %b_%b_%b,
          sum = %b, cout = %b",
      a, b, cin, sum, cout);

//apply input vectors
initial
begin
   #0    a = 4'b0011;
         b = 4'b0100;
         cin = 1'b0;

   #10   a = 4'b1100;
         b = 4'b0011;
         cin  = 1'b0;

   #10   a = 4'b0111;
         b = 4'b0110;
         cin = 1'b1;

   #10   a = 4'b1001;
         b = 4'b0111;
         cin = 1'b1;

   #10   a = 4'b1101;
         b = 4'b0111;
         cin = 1'b1;

   #10   a = 4'b1111;
         b = 4'b0110;
         cin = 1'b1;

   #10   $stop;
end

//instantiate the module into
//the test bench
ripple_adder_4 inst1 (
   .a(a),
   .b(b),
   .cin(cin),
   .sum(sum),
   .cout(cout)
   );

endmodule
```

Figure 2.68 Test bench module for the 4-bit ripple adder of Figure 2.67.

```
a b cin = 0011_0100_0, sum = 0111, cout = 0
a b cin = 1100_0011_0, sum = 1111, cout = 0
a b cin = 0111_0110_1, sum = 1110, cout = 0
a b cin = 1001_0111_1, sum = 0001, cout = 1
a b cin = 1101_0111_1, sum = 0101, cout = 1
a b cin = 1111_0110_1, sum = 0110, cout = 1
```

Figure 2.69 Binary outputs obtained from the adder test bench of Figure 2.68.

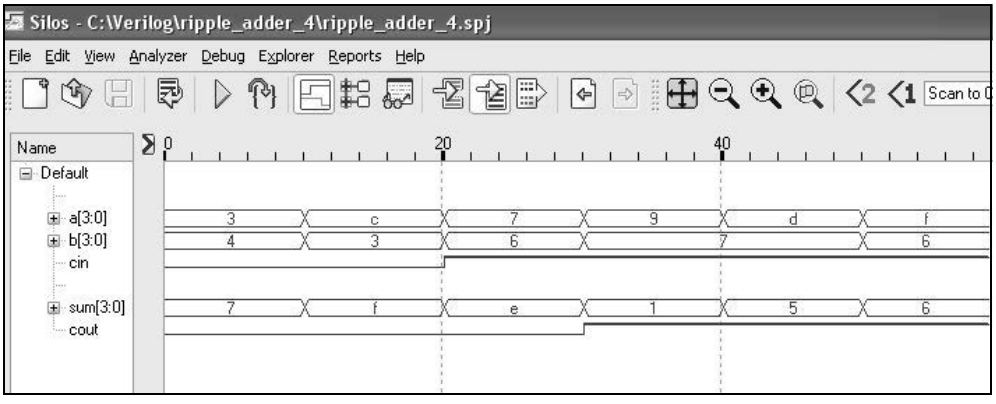


Figure 2.70 Waveforms for the 4-bit ripple adder test bench of Figure 2.68. The values for the variables are shown in hexadecimal.

2.8 Introduction to Mixed-Design Modeling

Mixed-design modeling incorporates different modeling styles in the same module. This includes gate and module instantiations, as well as continuous assignments and behavioral constructs. For example, a full adder can be designed that incorporates built-in primitives, dataflow, and behavioral modeling. When using behavioral modeling, variables within **always** and **initial** statements must be a register data type

2.8.1 Full Adder

In Section 2.7.2, a full adder was designed using structural modeling. The same adder will now be designed using mixed-design modeling. The equations for a full adder are restated below for convenience, where *a* and *b* are the augend and addend, respectively, *cin* is the carry-in to the adder, *sum* is the result, and *cout* is the carry-out.

$$sum = (a \oplus b) \oplus cin$$

$$cout = cin\ (a \oplus b) + ab$$

The full adder of Figure 2.52 is redrawn in Figure 2.71. The Verilog module for the full adder is shown in Figure 2.72. The variable *cout* is used in an **always** statement; therefore, it must be defined as type **reg**. There is an internal net declared as *net1*. Gate instantiation using a built-in primitive is used for the exclusive-OR function that generates *net1*. Behavioral modeling uses the **always** statement to define *cout* and dataflow modeling uses continuous assignment to define the *sum*.

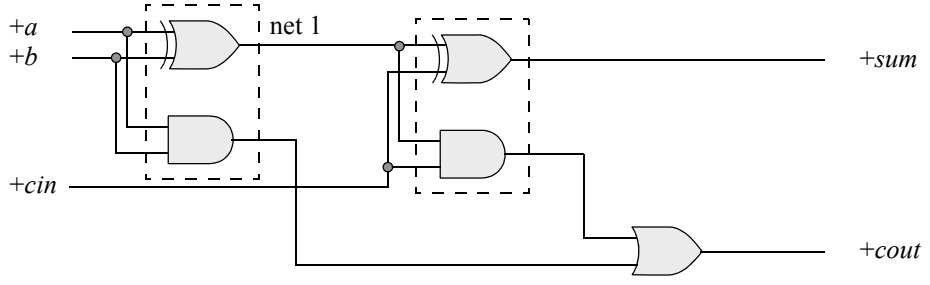


Figure 2.71 Full adder used in mixed-design modeling.

```
//mixed-design full adder
module full_adder_mixed (a, b, cin, sum, cout);

//list inputs and outputs
input a, b, cin;
output sum, cout;

//define reg and wires
reg cout;
wire a, b, cin;
wire sum;
wire net1;

//built-in primitive
xor (net1, a, b);

//continued on next page
```

Figure 2.72 Verilog module for the full adder of Figure 2.71 using mixed-design modeling.

```
//behavioral
always @ (a or b or cin)
begin
   cout = cin & (a ^ b) | (a & b);
end

//dataflow
assign sum = net1 ^ cin;

endmodule
```

Figure 2.72 (Continued)

The test bench is shown in Figure 2.73 in a slightly different version. The input vectors for each time unit are listed on the same line — a semicolon separates the individual bits. The complete set of vectors generates all combinations of the three inputs. The binary outputs are listed in Figure 2.74 and the data analyzer waveforms are shown in Figure 2.75. The binary outputs and waveforms are identical to those of the full adder of Section 2.7.2.

```
//mixed-design full adder test bench
module full_adder_mixed_tb;

reg a, b, cin;
wire sum, cout;

//display variables
initial
$monitor ("a b cin = %b %b %b, sum = %b, cout = %b",
      a, b, cin, sum, cout);

//apply input vectors
initial
begin
   #0    a = 1'b0; b = 1'b0; cin = 1'b0;
   #10   a = 1'b0; b = 1'b0; cin = 1'b1;
   #10   a = 1'b0; b = 1'b1; cin = 1'b0;
   #10   a = 1'b0; b = 1'b1; cin = 1'b1;
   #10   a = 1'b1; b = 1'b0; cin = 1'b0;
   #10   a = 1'b1; b = 1'b0; cin = 1'b1;
//continued next page
```

Figure 2.73 Test bench for the mixed-design full adder of Figure 2.72.

```
    #10    a = 1'b1; b = 1'b1; cin = 1'b0;
    #10    a = 1'b1; b = 1'b1; cin = 1'b1;
    #10    $stop;
end

//instantiate the module into the test bench
full_adder_mixed inst1 (
    .a(a),
    .b(b),
    .cin(cin),
    .sum(sum),
    .cout(cout)
    );
endmodule
```

Figure 2.73 (Continued)

```
a b cin = 0 0 0, sum = 0, cout = 0
a b cin = 0 0 1, sum = 1, cout = 0
a b cin = 0 1 0, sum = 1, cout = 0
a b cin = 0 1 1, sum = 0, cout = 1
a b cin = 1 0 0, sum = 1, cout = 0
a b cin = 1 0 1, sum = 0, cout = 1
a b cin = 1 1 0, sum = 0, cout = 1
a b cin = 1 1 1, sum = 1, cout = 1
```

Figure 2.74 Binary outputs for the mixed-design full adder of Figure 2.72.

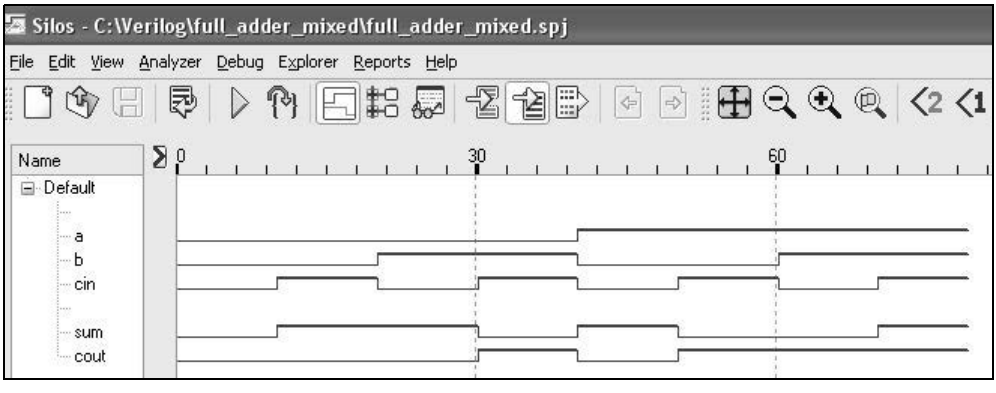


Figure 2.75 Waveforms for the mixed-design full adder of Figure 2.72.

2.9 Problems

2.1 Design a 4-input AND gate using dataflow modeling. Design a test bench and obtain the binary outputs and waveforms.

2.2 Design a 4-input AND gate using behavioral modeling. Design a test bench and obtain the binary outputs and waveforms.

2.3 Design a 2-input NAND gate using dataflow modeling. Design a test bench and obtain the binary outputs and waveforms.

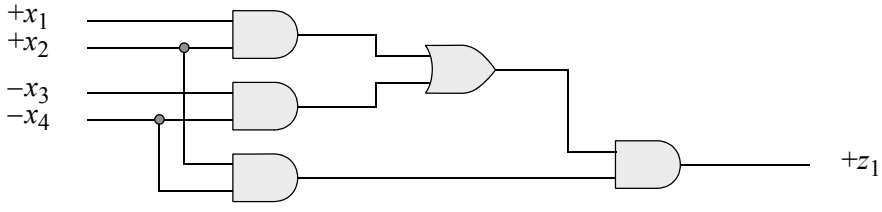
2.4 Design a 2-input NAND gate using behavioral modeling. Design a test bench and obtain the binary outputs and waveforms.

2.5 Design a 2-input NOR gate using dataflow modeling. Design a test bench and obtain the binary outputs and waveforms.

2.6 Design a 2-input NOR gate using behavioral modeling. Design a test bench and obtain the binary outputs and waveforms.

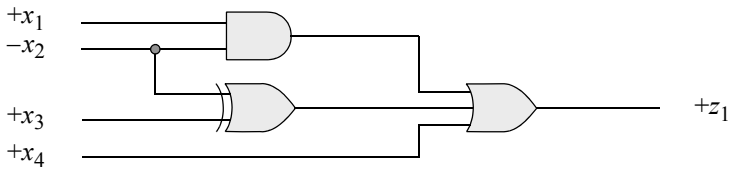
2.7 Design a 3-input exclusive-NOR gate using behavioral modeling. Design a test bench and obtain the binary outputs and waveforms.

2.8 Implement the logic diagram shown below using dataflow modeling. Design a test bench and obtain the binary outputs and waveforms.

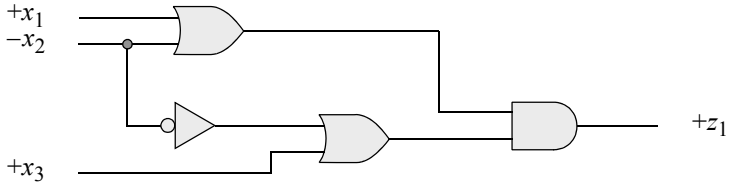


2.9 Implement the logic diagram shown in the previous problem using structural modeling. Design a test bench and obtain the binary outputs and waveforms.

2.10 Implement the logic diagram shown below using dataflow modeling. Design a test bench and obtain the binary outputs and waveforms.



2.11 Implement the logic diagram shown below using dataflow modeling. Design a test bench and obtain the binary outputs and waveforms.

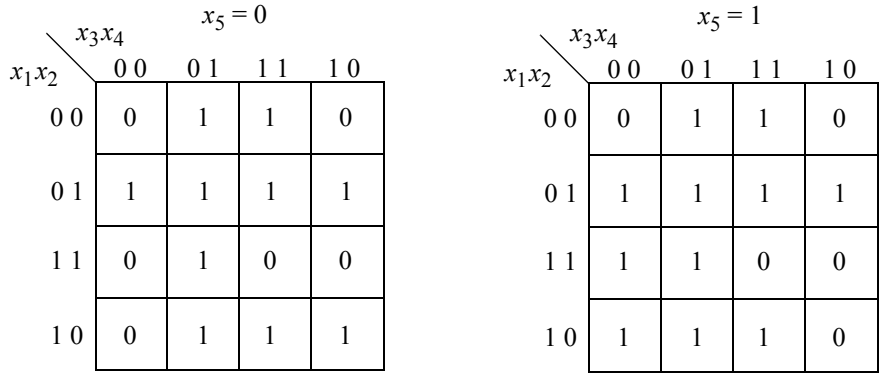


2.12 Given the Karnaugh map shown below, obtain the minimum sum-of-products expression and the minimum product-of-sums expression. Then implement both expressions using behavioral modeling. Design two test benches and compare the binary outputs and waveforms.

x_1x_2 \ x_3x_4	0 0	0 1	1 1	1 0
0 0	0	0	1	0
0 1	0	1	1	0
1 1	0	1	1	0
1 0	1	1	1	1

z_1

2.13 Obtain the minimized product-of-sums expression for function z_1 represented by the Karnaugh map shown below, then implement the equation using dataflow modeling.



z_1

2.14 Minimize the following equation, then implement the result using structural modeling:

$$z_1 = x_1'x_3'x_4' + x_1'x_3x_4' + x_1x_3'x_4' + x_2x_3x_4 + x_1x_3x_4'$$

2.15 Minimize the following equation, then implement the result using structural modeling:

$$z_1 = x_1'x_2x_3'x_4' + x_3'x_4 + x_1x_2x_3'x_4' + x_3'x_4 + x_1x_2'x_3'x_4$$

2.16 Obtain the equation for a logic circuit that generates an output when the 4-bit unsigned binary number $z_1 = x_1x_2x_3x_4$ is greater than 5, but less than 10, where x_4 is the low-order bit. The equation is to be in a minimal *sum-of-product* form. Then implement the equation using behavioral modeling.

2.17 Plot the expression shown below on a Karnaugh map using x_4 as a map-entered variable and obtain the minimized equation in a sum-of-products form. Then implement the equation using structural modeling.

$$z_1 = x_1'x_2'x_3'(x_4) + x_1'x_2x_3'(x_4) + x_1x_2x_3'(x_4) + x_1x_2'x_3(x_4) + x_1x_2'x_3(x_4')$$

2.18 Given the Karnaugh map shown below, obtain the minimized expression for z_1. Then implement the result using structural modeling.

x_1 \ x_2x_3	0 0	0 1	1 1	1 0
0	0: x_4	1: 0	3: 0	2: $x_4 + x_4'$
1	4: x_4'	5: $x_4 + x_4'$	7: x_4'	6: 0

z_1

2.19 Obtain the minimized product-of-sums expression for the function z_1 represented by the Karnaugh map shown below. Then implement the result using dataflow, behavioral, structural, and mixed-design modeling.

$x_5 = 0$

x_1x_2 \ x_3x_4	0 0	0 1	1 1	1 0
0 0	0: 0	2: 0	6: 0	4: 0
0 1	8: 0	10: 0	14: 0	12: 0
1 1	24: 0	26: 1	30: 1	28: 0
1 0	16: 1	18: 1	22: 1	20: 0

$x_5 = 1$

x_1x_2 \ x_3x_4	0 0	0 1	1 1	1 0
0 0	1: 1	3: 1	7: 1	5: 1
0 1	9: 0	11: 1	15: 1	13: 0
1 1	25: 0	27: 1	31: 1	29: 0
1 0	17: 1	19: 1	23: 1	21: 1

z_1

3

Language Elements

Language elements are the constituent parts of the Verilog language. They consist of comments, identifiers, keywords, data types, parameters, and a set of values which determines the logic value of a net. Also included in this chapter are compiler directives and an introduction to system tasks and functions, which are covered in more detail in Chapter 10.

3.1 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. */
```

3.2 Identifiers

An identifier is a name given to an object or variable so that it can be referenced elsewhere in the design. Identifier names are used for modules, registers, ports, wires, or module instance names. Also, **begin** . . . **end** blocks may include an identifier. An identifier consists of a sequence of characters that can be letters, digits, $, or underscore (_). The first character of an identifier must be a letter or an underscore. The $ character is reserved for system tasks. Identifiers are case sensitive. For example, *Clock* and *clock* are different identifiers. An identifier refers to a unique object in the module in which it is defined.

When a module is instantiated into another module, the module instance name is considered to be an identifier. An identifier can contain up to 1024 characters; however, the first character cannot be a digit. Examples of identifiers are shown below, where **input**, **output**, and **reg** are keywords.

```
input a, b, cin;      //a, b, and cin are identifiers
output sum, cout;     //sum and cout are identifiers
reg z1;               //z1 is an identifier
```

Escaped identifiers *Escaped identifiers* begin with a backslash (\) and end with a white space (space, tab, or new line) and provide a means to include any printable ASCII character in an identifier. The backslash and whitespace are not part of the identifier. An escaped identifier that contains a keyword is different than the keyword. For example, \assign is different than the keyword **assign**. Examples of escaped identifiers are shown below.

```
\2005
\~$~
\*****
```

3.3 Keywords

Verilog HDL reserves a list of special predefined, nonescaped identifiers called *keywords* which are used to define the language constructs. Only lower case keywords are used for reserved words. A list of keywords is shown by category in Table 3.1.

Table 3.1 Verilog HDL Keywords

Category	Keywords		
Bidirectional Gates	rtran	rtranif0	rtranif1
	tran	tranif0	tranif1
Charge Storage Strengths	large	medium	small
CMOS Gates	cmos	rcmos	
Combinational Logic Gates	and	buf	nand
	nor	not	or
	xnor	xor	
Continuous Assignment	assign		
Data Types	integer	real	realtime
	reg	scalared	time
	tri	tri0	tri1
	triand	trior	trireg
	vectored	wand	wire
	wor		
Module Declaration	endmodule	module	
MOS Gates	nmos	pmos	rnmos
	rpmos		
Multiway Branching	case	casex	casez
	default	endcase	
Named Event	event		
Parameters	defparam	parameter	specparam
Port Declaration	inout	input	output
Procedural Constructs	always	initial	
Procedural Continuous Assignment	assign	deassign	force
	release		
Procedural Flow Control	begin	disable	else
	end	for	forever
	fork	if	join
	repeat	wait	while
Pull Gates	pulldown	pullup	

Table 3.1 Verilog HDL Keywords

Category	Keywords		
Signal Strengths	highz0	highz1	pull0
	pull1	strong0	strong1
	supply0	supply1	weak0
	weak1		
Specify Block	endspecify	specify	
Tasks and Functions	endfunction	endtask	function
	task		
Three-State Gates	bufif0	bufif1	notif0
	notif1		
Timing Control	edge	negedge	posedge
User-Defined Primitives	endprimitive	endtable	primitive
	table		

3.3.1 Bidirectional Gates

tran, tranif0, tranif1, rtran, rtranif0, rtranif1 These are bidirectional primitive gates. The signals on either side of the gates can be specified as inputs or outputs; that is, either signal can be the driver. The inputs and outputs are classified as scalar signals. The **tran** gate (switch) acts as a buffer between the two signals. One terminal can be declared as **input** or **inout**, the other declared as **output** or **inout**. The **tran** primitive is instantiated as shown below, where the instance name *inst1* is optional.

tran inst1 (inout1, inout2);

The primitives **tranif0** and **tranif1** also have two bidirectional terminals plus a control input. The **tranif0** gate connects the two signals only if the control input is a logical 0; otherwise, the output of the gate is a high impedance. The **tranif1** gate transmits only if the control input is a logical 1; otherwise, the output of the gate is a high impedance. That is, the output logic value is the same as the input logic value or a high impedance. The **tranif0** and **tranif1** primitives are instantiated as shown below, where the control input is listed last.

tranif0 inst1 (inout1, inout2, control);
tranif1 inst1 (inout1, inout2, control);

There are also three bidirectional gates — called resistive gates — that operate the same as the previous gates, but have a higher source-to-drain impedance. This

reduces the signal strength. The resistive gates are declared with the same keywords, but prefixed with an **r** — for example, **rtran**, **rtranif0**, and **rtranif1**. The resistive gates have the same syntax as the non-resistive gates. Figure 3.1 shows the logic diagrams for the six types of bidirectional gates.

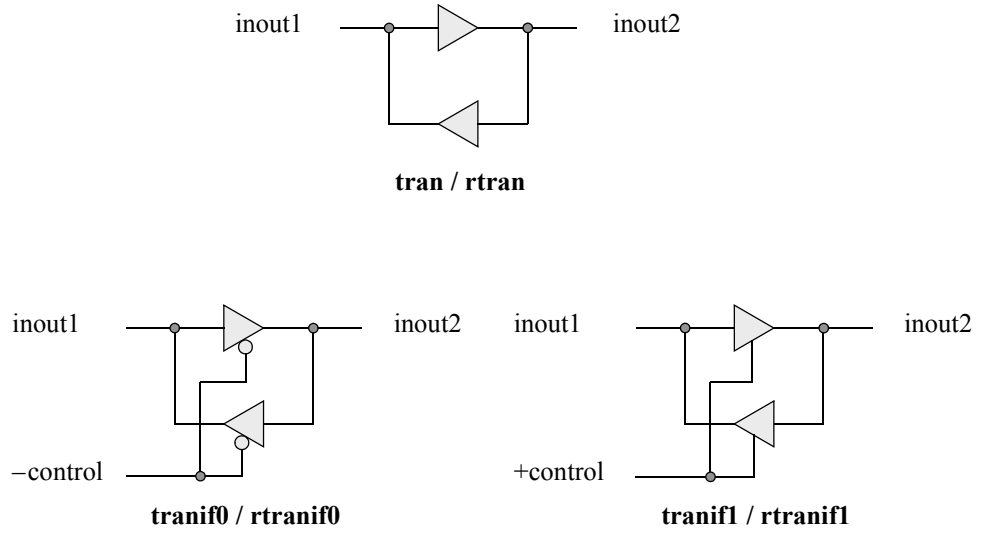


Figure 3.1 Bidirectional gates.

3.3.2 Charge Storage Strengths

small, medium, large There are three charge storage strengths that relate to the capacitance of a net: **small**, **medium**, and **large**. A **trireg** net, which models the charge stored on a net, can have a charge strength associated with the net when the drivers are in a high impedance state. The amount of charge stored determines the size of the capacitance associated with the net. A driver for a **trireg** net will generate three states: logic 0, logic 1, or **x**. If all drivers on the net generate a high impedance output, then the net retains the last value driven onto the net. The default strength is **medium**.

Optional delay times can also be specified for a **trireg** net, indicating the rise delay, the fall delay, and the charge decay time. The charge decays when all the drivers of the net are in a high impedance state. If no delay is specified, then the charge on the **trireg** net does not decay. A **trireg** net can be forced to a high impedance state by a **force** procedural continuous assignment.

3.3.3 CMOS Gates

cmos The **cmos** gate can be modeled with an **nmos** and a **pmos** device to implement a **cmos** transmission gate. There are two enable inputs: one for the n-channel device and one for the p-channel device. A **cmos** gate transmits data from input to output when its two control signals are in complementary states; that is, the n-channel control is a logic 1 and the p-channel control is a logic 0. If the states of the control signals are reversed, then the output is in a high impedance state. These control inputs allow both transistors to be either on (transmit) or off (high impedance). A **cmos** gate can be instantiated as shown below, where the instance *inst1* name is optional.

```
cmos  inst1 (output, data_input, n_enable, p_enable);
```

rcmos The **rcmos** gate is a high resistive version of the **cmos** gate.

3.3.4 Combinational Logic Gates

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_1n),
           inst2 (output_2, input_21, input_22, . . . , input_2n),
                                .
                                .
                                .
            instm (output_m, input_m1, input_m2, . . . , input_mn);
```

and This is a built-in primitive gate that operates according to the truth table shown in Table 3.2 for a 2-input AND gate.

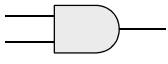
Table 3.2 Truth Table for the Logical AND Built-In Primitive

Inputs		Output
x_1	x_2	z_1
0	0	0
0	1	0
1	0	0
1	1	1
0	x	0
0	z	0
1	x	x
1	z	x

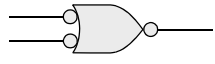
Inputs		Output
x_1	x_2	z_1
x	0	0
x	1	x
x	x	x
x	z	x
z	0	0
z	1	x
z	x	x
z	z	x

The **x** entry in Table 3.2 represents an unknown logic value. The **z** entry represents a high impedance state. Simulators use the **z** value to indicate when 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
```

The truth table for a **buf** gate is shown in Table 3.3 for one output.

Table 3.3 Truth Table for a buf Gate

Input	Output
0	0
1	1
x	x
z	x

nand This is a built-in primitive gate that operates according to the truth table shown in Table 3.4 for a 2-input NAND gate.

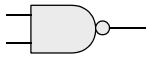
Table 3.4 Truth Table for the Logical NAND Built-In Primitive

Inputs		Output
x_1	x_2	z_1
0	0	1
0	1	1
1	0	1
1	1	0
0	x	1
0	z	1
1	x	x
1	z	x

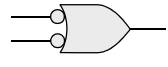
Inputs		Output
x_1	x_2	z_1
x	0	1
x	1	x
x	x	x
x	z	x
z	0	1
z	1	x
z	x	x
z	z	x

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 built-in primitive gate that operates according to the truth table shown in Table 3.5 for a 2-input NOR gate.

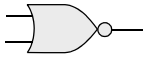
Table 3.5 Truth Table for the Logical NOR Built-In Primitive

Inputs		Output
x_1	x_2	z_1
0	0	1
0	1	0
1	0	0
1	1	0
0	x	x
0	z	x
1	x	0
1	z	0

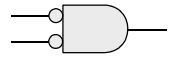
Inputs		Output
x_1	x_2	z_1
x	0	x
x	1	0
x	x	x
x	z	x
z	0	x
z	1	0
z	x	x
z	z	x

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 truth table for a **not** gate is shown in Table 3.6 for one output.