DIGITAL DESIGN

PRINCIPLES AND PRACTICES

FIFTH EDITION WITH VERILOG

JOHN F. WAKERLY

Pearson

DIGITAL DESIGN

Principles and Practices

DIGITAL DESIGN

Principles and Practices

Fifth Edition with Verilog

John F. Wakerly

Pearson 330 Hudson Street, NY NY 10013

Senior Vice President Courseware Portfolio Management: Marcia J. Horton
Director, Portfolio Management: Engineering, Computer Science & Global Editions: Julian Partridge
Portfolio Manager Assistant: Michelle Bayman
Field Marketing Manager: Demetrius Hall
Product Marketing Manager: Yvonne Vannatta
Marketing Assistant: Jon Bryant
Content Managing Producer, ECS and Math: Scott Disanno
Operations Specialist: Maura Zaldivar-Garcia
Manager, Rights and Permissions: Ben Ferrini
Cover Designer: Black Horse Designs
Cover Art: "Tuesday Matinee," by Peter Alan Crowell

Many of the designations by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed in initial caps or all caps. The author and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of theories and programs to determine their effectiveness.

The author and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages with, or arising out of, the furnishing, performance, or use of these programs.

Pearson Education Ltd., London
Pearson Education Singapore, Pte. Ltd
Pearson Education Canada, Inc.
Pearson Education Japan
Pearson Education Australia PTY, Ltd
Pearson Education North Asia, Ltd., Hong Kong
Pearson Education de Mexico, S.A. de C.V.
Pearson Education Malaysia, Pte. Ltd.
Pearson Education, Inc., Hoboken

Library of Congress Cataloging-in-Publication Data on File

ISBN-10: 013446009X
1 16 ISBN-13: 9780134460093

Pearson www.pearsonhighered.com

For Ralph and Carm, again

This page intentionally left blank

CONTENTS

PREFACE

This book is for everyone who wants to design and build real digital circuits. It is based on the idea that, in order to do this, you have to grasp the fundamentals, but at the same time you need to understand how things work in the real world. Hence, the “principles and practices” theme.

The practice of digital design has undergone a major transformation during the past 30 years, a direct result of the stunning increases in integrated-circuit speed and density over the same time period. In the past, when digital designers were building systems with thousands or at most tens of thousands of gates and flip-flops, academic courses emphasized minimization and efficient use of chip and board-level resources.

Today, a single chip can contain tens of millions of transistors and can be programmed to create a system-on-a-chip that, using the technology of the past, would have required hundreds of discrete chips containing millions of individual gates and flip-flops. Successful product development nowadays is limited more by the design team’s ability to correctly and completely specify the product’s detailed functions, than by the team’s ability to cram all the needed circuits into a single board or chip. Thus, a modern academic program must necessarily emphasize design methodologies and software tools, including hardware description languages (HDLs), that allow very large, hierarchical designs to be accomplished by teams of designers.

On one hand, with HDLs, we see the level of abstraction for typical designs moving higher, above the level of individual gates and flip-flops. But at the same time, the increased speed and density of digital circuits at both the chip and board level is forcing many digital designers to be more competent at a lower, electrical circuit level.

The most employable and ultimately successful digital designers are skilled, or at least conversant, at both levels of abstraction. This book gives you the opportunity to learn the basics at the high level (HDLs), at the low level (electrical circuits), and throughout the "vast middle" (gates, flip-flops, and higher-level digital-design building blocks).

Target Audience

introductory courses
electronics concepts

The material in this book is appropriate for introductory and second courses on digital logic design in electrical or computer engineering or computer science curricula. Computer science students who are unfamiliar with basic electronics concepts or who just aren't interested in the electrical behavior of digital devices may wish to skip Chapter 14; the rest of the book is written to be independent of this material, as long as you understand the basics in Chapter 1. On the other hand, *anyone* with a basic electronics background who wants to get up to speed on digital electronics can do so by reading Chapter 14. In addition, students with *no* electronics background can get the basics by reading a 20-page electronics tutorial at the author's website, `www.ddpp.com`.

optional sections
sidebars
boxed comments

Although this book's starting level is introductory, it goes beyond that and contains much more material than can be taught in a typical introductory course. I expect that typical courses will use no more than two-thirds of the material here, but each will use a *different* two thirds. Therefore, I've left it to the individual instructors and independent readers to tailor their reading to their own needs. To help these choices along, though, I've marked the headings of *optional sections* with an asterisk. In general, these sections can be skipped without any loss of continuity in the non-optional sections that follow. Also, the material in the sidebars (aka "boxed comments") is generally optional.

second courses
laboratory courses
fun stuff

Undoubtedly, some people will use this book in second courses and in laboratory courses. Advanced students will want to skip the basics and get right into the fun stuff. Once you know the basics, some of the most important and fun stuff is in the many sections and examples of digital design using Verilog.

marginal notes
marginal pun

All readers should make good use of the comprehensive index and of the *marginal notes* throughout the text that call attention to definitions and important topics. Maybe the highlighted topics in *this* section were more marginal than important, but I just wanted to show off my text formatting system.

Chapter Descriptions

What follows is a list of short descriptions of this book's fifteen chapters. This may remind you of the section in typical software guides, "For People Who Hate Reading Manuals." If you read this list, then maybe you don't have to read the rest of the book.

- Chapter 1 gives a few basic definitions and a preview of a few important topics. It also has a little bit on digital circuits, to enable readers to handle the rest of the book without Chapter 14's "deep dive."

NOT AS LONG AS IT SEEMS

A few reviewers have complained about the length of previous editions of this book. The present edition is a little shorter, but also please keep in mind:

- You don't have to read everything. The headings of sections and subsections that are optional for most readers are marked with an asterisk.
- Stuff written in these "boxed comments" (a.k.a. sidebars) is usually optional too.
- I asked the publisher to print this book in a larger font (11 point) than is typical for technical texts (10 point). This is easier on your eyes and mine, and it also allows me to put in more figures and tables while still keeping most of them on the same facing pages as the referring text. (I do the page layout myself and pay a lot of attention to this.)
- I write my books to be "reference quality," with comprehensive topic coverage and excellent indexing, so you can come back to them in later courses, or later in your career to refresh or even to learn new things. The cost of books being what they are these days, you may not keep this book, but the option is there.

- Chapter 2 is an introduction to binary number systems and codes. Readers who are already familiar with binary number systems from a software course should still read Sections 2.10–2.13 to get an idea of how binary codes are used by hardware. Advanced students can get a nice introduction to error-detecting codes by reading Sections 2.14 and 2.15. The material in Section 2.16.1 should be read by everyone; it is used in a lot of modern systems.
- Chapter 3 teaches combinational logic design principles, including switching algebra and combinational-circuit analysis, synthesis, and minimization.
- Chapter 4 introduces various digital-design practices, starting with documentation standards, probably the most important practice for aspiring designers to start practicing. Next, it introduces timing concepts, especially for combinational circuits, and it ends with a discussion of HDLs, design flow, and tools.
- Chapter 5 is a tutorial and reference on Verilog, the HDL that is used throughout the rest of the book. The first few sections should be read by all, but some readers may wish to skip the rest until it's needed, since new Verilog constructs are summarized in later chapters "on the fly" the first time they're used, mainly in Chapter 6.
- Chapter 6 describes two "universal" combinational building blocks, ROMs and PLDs. It then describes the two most commonly used functional building blocks, decoders and multiplexers; gate-level and Verilog-based designs are shown for each. It's possible for the reader to go from here directly to state machines in Chapter 9, and come back to 7 and 8 later.

- Chapter 7 continues the discussion of combinational building blocks, at both the gate level and in Verilog, for three-state devices, priority encoders, XOR and parity functions, and comparators, then concludes with a Verilog example design for a nontrivial "random logic" function.
- Chapter 8 covers combinational circuits for arithmetic functions, including adding and subtracting, shifting, multiplying, and dividing.
- Chapter 9 is a traditional introduction to state machines using D flip-flops, including analysis and synthesis using state tables, state diagrams, ASM charts, and Verilog.
- Chapter 10 introduces other sequential elements including latches, more edge-triggered devices, and their Verilog behavioral models. This chapter also describes the sequential elements in a typical FPGA and, for interested readers, has sections on sequential PLDs and feedback sequential circuits.
- Chapter 11 is focused on the two most commonly used sequential-circuit building blocks, counters and shift registers, and their applications. Both gate-level and Verilog-based examples are given.
- Chapter 12 gives a lot more details on how to model state machines using Verilog and gives many examples.
- Chapter 13 discusses important practical concepts for sequential-circuit design, including synchronous system structure, clocking and clock skew, asynchronous inputs and metastability, and a detailed two-clock synchronization example in Verilog.
- Chapter 14 describes digital circuit operation, placing primary emphasis on the external electrical characteristics of logic devices. The starting point is a basic electronics background including voltage, current, and Ohm's law. This chapter may be omitted by readers who aren't interested in how to make real circuits work, or who have the luxury of having someone else to do the dirty work.
- Chapter 15 is all about memory devices and FPGAs. Memory coverage includes read-only memory and static and dynamic read/write memories in terms of both internal circuitry and functional behavior. The last section gives more details of an FPGA architecture, the Xilinx 7 series.

Most of the chapters contain references, drill problems, and exercises. Drill problems are typically short-answer or "turn-the-crank" questions that can be answered directly based on the text material, while exercises typically require a little more thinking. The drill problems in Chapter 14 are particularly extensive and are designed to allow non-EEs to ease into this material.

Differences from the Fourth Edition

For readers and instructors who have used previous editions of this book, this fifth edition has several key differences in addition to general updates:

- This edition covers Verilog only; there's no VHDL. Bouncing between the languages is just too distracting. Moreover, Verilog and its successor SystemVerilog are now the HDLs of choice in non-government settings. See the excellent, well-reasoned and nicely documented paper by Steve Golson and Leah Clark, "Language Wars in the 21st Century: Verilog versus VHDL—Revisited" (2016 Synopsys Users Group Conference), and jump to the last section if you don't want to read the whole article.
- This edition has many more HDL examples and a much greater emphasis on design flow and on test benches, including purely stimulative as well as self-checking ones.
- To make the book more accessible to non-EE computer engineering students, detailed coverage of CMOS circuits has been moved to Chapter 14 and a minimal amount of electronics has been added to Chapter 1 so that the CMOS chapter can be skipped entirely if desired.
- TTL, SSI, MSI, 74-series logic, PLDs, and CPLDs have been deprecated.
- Karnaugh-map-based minimization has finally been, well, minimized.
- While the book still has a comprehensive Verilog tutorial and reference in Chapter 5, Verilog concepts are interspersed "just in time" in sidebars in Chapters 6 and 7 so students can go straight to "the good stuff" there.
- There is a greater emphasis on FPGA-based design, FPGA architectural features, and synthesis results and trade-offs.
- The chapter on combinational-logic elements has been split into three, to facilitate going straight to state machines after just the first if desired. This also allows more coverage of arithmetic circuits in the last.
- An entire chapter has been devoted to state-machine design in Verilog, including many examples.
- The chapter on synchronous design methodology now contains a detailed control-unit-plus-datapath example and a comprehensive example on crossing clocking domains using asynchronous FIFOs.
- The jokes aren't quite as bad, I hope.

Digital-Design Software Tools

All of the Verilog examples in this book have been compiled and tested using the Xilinx Vivado® suite, which includes tools for targeting Verilog, SystemVerilog, and VHDL designs to Xilinx 7-series FPGAs. However, in general there's no special requirement for the examples to be compiled and synthesized using Vivado or even to be targeted to Xilinx or any other FPGA. Also, this book does *not* contain a tutorial on Vivado; Xilinx has plenty of online materials for that. Thus, a reader will able to use this text with any Verilog tools, including the ones described below.

The free "Webpack" edition of Vivado can be downloaded from Xilinx; it supports smaller 7-series FPGAs, Zynq® SoC-capable FPGAs, and evaluation boards. It's a big download, over 10 gigabytes, but it's a comprehensive tool suite. Pre-7-series FPGAs as well as the smaller Zynq FPGAs are supported by the Xilinx ISE® (Integrated Software Environment) tool suite, also available in a free "Webpack" edition. Note that ISE is supported in "legacy" mode and has not been updated since 2013. For either suite, go to `www.xilinx.com` and search for "Webpack download."

If you're using Altera (now part of Intel) devices, they also have a good University Program and tools; search for "Altera university support" and then navigate to the "For Students" page. Free tools include their Quartus™ Prime Lite Edition for targeting Verilog, SystemVerilog, and VHDL designs to their entry-level FPGAs and CPLDs, and a starter edition of industry-standard ModelSim® software for simulating them.

Both Altera and Xilinx offer inexpensive evaluation boards suitable for implementing FPGA-based student projects, either directly or through third parties. Such boards may include switches and LEDs, analog/digital converters and motion sensors, and even USB and VGA interfaces, and may cost less than $100 through the manufacturers' university programs.

Another long-time source of professional digital design tools with good university support is Aldec, Inc. (`www.aldec.com`). They offer a student edition of their popular Active-HDL™ tool suite for design entry and simulation; besides the usual HDL tools, it also includes block-diagram and state-machine graphical editors, and its simulator also includes a waveform editor for creating stimuli interactively. The Active-HDL simulator can be installed as a plug-in with Vivado to use its features instead of the Vivado simulator.

All of the above tools, as well as most other engineering design tools, run on Windows PCs, so if you are a Mac fan, get used to it! Depending on the tools, you may or may not have some success running them on a Mac in a Windows emulation environment like VMware's. The most important thing you can do to make the tools "go fast" on your PC is to equip it with a solid-state disk drive (SSD), not a rotating one.

Even if you're not ready to do your own original designs, you can use any of the above tools to try out and modify the examples in the text, since the source code for all of them is available online, as discussed next.

Engineering Resources and www.ddpp.com

Abundant support materials for this book are available on the Web at Pearson's "Engineering Resources" site. At the time of publication, the Pearson link was `media.pearsoncmg.com/bc/abp/engineering-resources`, but you know how it goes with long links. It's easier just to go to the author's website, `www.ddpp.com`, which contains a link to Pearson's site. Also, the author's site will contain the latest errata and other materials that may be added or changed "on the fly," and perhaps even a blog someday.

Resources at the Pearson site include downloadable source-code files for all Verilog modules in the book, selected drill and exercise solutions, and supplementary materials, such as a 20-page introduction to basic electronics concepts for non-EEs.

For Instructors

Pearson maintains a website with a comprehensive set of additional materials for instructors only. Go to the Engineering Resources site mentioned above, navigate to this book, and click on the "Instructor Resources" link. Registration is required, and it may take a few days for your access to be approved. Resources include additional drill and exercise solutions, additional source code, more exercises, and line art and tables from the book for use in your lectures. Upon request, materials from previous editions may also be posted there to aid instructors who are transitioning their courses from older technology.

Other resources for instructors include the author's site, `www.ddpp.com`, and the university programs at Xilinx, Altera, and Aldec; go to `www.ddpp.com` for up-to-date links to them. The manufacturer sites offer a variety of product materials, course materials, and discounts on chips and boards you can use in digital-design lab courses, and in some cases "full-strength" tool packages that you can obtain at a steep discount for use in your advanced courses and research.

Errors

Warning: This book may contain errors. The author and the publisher assume no liability for any damage—incidental, brain, or otherwise—caused by errors.

There, that should make the lawyers happy. Now, to make *you* happy, let me assure you that a great deal of care has gone into the preparation of this book to make it as error free as possible. I am anxious to learn of the remaining errors so that they may be fixed in future printings, editions, and spin-offs. Therefore, I will pay $5 via PayPal to the first finder of each undiscovered error—technical, typographical, or otherwise—in the printed book. Please email your comments to me by using the appropriate link at `www.ddpp.com`.

An up-to-date list of discovered errors can always be obtained using the appropriate link at `www.ddpp.com`. It will be a very short file transfer, I hope.

Acknowledgements

Many people helped make this book possible. Most of them helped with the first four editions and are acknowledged there. For the ideas on the “principles” side of this book, I still owe great thanks to my teacher, research advisor, and friend, the late Ed McCluskey. On the “practices” side, I got good advice from my friend Jesse Jenkins, from Xilinx staffers Parimal Patel and Trevor Bauer, and from fellow McCluskey advisee Prof. Subhasish Mitra of Stanford.

Since the fourth edition was published, I have received many helpful comments from readers. In addition to suggesting or otherwise motivating many improvements, readers have spotted dozens of typographical and technical errors whose fixes are incorporated in this fifth edition.

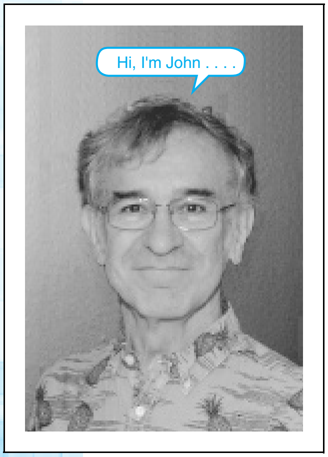
The most substantial influence and contribution to this edition came from ten anonymous (to me) academic reviewers, all of whom teach digital design courses using my fourth edition or one of its competitors. I did my best to incorporate their suggestions, which often meant deleting material that experienced designers like me (aka old-timers) are perhaps too attached to, while greatly enhancing the coverage of modern concepts in HDL-based design flow, test benches, synthesis, and more.

My sponsoring editor at Pearson, Julie Bai, deserves thanks for shepherding this project over the past couple of years; she’s my first editor who actually took a digital design course using a previous edition of this book. Unfortunately, she’s also the fourth or fifth editor who has changed jobs after almost completing one of my book projects, convincing me that working with me inevitably leads to an editor’s burnout or success or both. Special thanks go to her boss’s boss, Marcia Horton, who has kept an eye on my projects for a couple of decades, and to Scott Disanno and Michelle Bayman, who guided the production and launch processes for this edition.

Thanks also go to artist Peter Crowell, whose paintings I discovered on Ebay when editor Julie Bai suggested we do a cover based on Piet Mondrian’s work, some of which she said “almost looks like an abstract take on logic circuits.” Crowell’s “Tuesday Matinee” fits the bill beautifully. His painting is “tiled” on the cover and in the chapter-opening art in much the same way that logic blocks and interconnect are tiled in an FPGA. Our cover designer Marta Samsel took my engineering-ish concept and adapted it beautifully.

Finally, my wife Joanne Jacobs was very supportive of this project, letting me work in peace “upstairs” while she worked “downstairs” on her education blog. She didn’t even complain that the Christmas tree was still up in February.

John F. Wakerly
Los Altos, California



chapter 1

Introduction

Welcome to the world of digital design. Perhaps you're a computer science student who knows all about computer software and programming, but you're still trying to figure out how all that fancy hardware could possibly work. Or perhaps you're an electrical engineering student who already knows something about analog electronics and circuit design, but you wouldn't know a bit if it bit you. No matter. Starting from a fairly basic level, this book will show you how to design digital circuits and subsystems.

We'll give you the basic principles that you need to figure things out, and we'll give you lots of examples. Along with principles, we'll try to convey the flavor of real-world digital design by discussing practical matters whenever possible. And I, the author, will often refer to myself as "we" in the hope that you'll be drawn in and feel that we're walking through the learning process together.

1.1 About Digital Design

Some people call it "logic design." That's OK, but ultimately the goal of design is to build systems. To that end, we'll cover a whole lot more in this text than logic equations and theorems.

This book claims to be about principles and practices. Most of the principles that we present will continue to be important years from now;

some may be applied in ways that have not even been discovered yet. As for practices, they are sure to be a little different from what's presented here by the time you start working in the field, and they will continue to change throughout your career. So you should treat the "practices" material in this book as a way to reinforce principles, and as a way to learn design methods by example.

One of the book's goals is to present enough about basic principles for you to know what's happening when you use software tools to "turn the crank" for you. The same basic principles can help you get to the root of problems when the tools happen to get in your way.

Listed in the box below are several key points that you should learn through your studies with this text. Many of these items may not make sense to you right now, but you can come back and review them later.

Digital design is engineering, and engineering means "problem solving." My experience is that only 5% to 10% of digital design is "the fun stuff"—the creative part of design, the flash of insight, the invention of a new approach. Much of the rest is just "turning the crank." To be sure, turning the crank is much

IMPORTANT THEMES IN DIGITAL DESIGN

- Good tools do not guarantee good design, but they help a lot by taking the pain out of doing things right.
- Digital circuits have analog characteristics.
- Know when to worry and when not to worry about the analog aspects of digital design.
- Transistors and all the digital components built with them are cheap and plentiful; make sensible trade-offs between minimizing the size of your designs and your engineering time.
- Always document your designs to make them understandable to yourself and to others.
- Use consistent coding, organizational, and documentation styles in your HDL-based designs, following your company's guidelines.
- Understand and use standard functional building blocks.
- State-machine design is like programming; approach it that way.
- Design for minimum cost at the system level, including your own engineering effort as part of the cost.
- Design for testability and manufacturability.
- Use programmable logic to simplify designs, reduce cost, and accommodate last-minute modifications.
- Avoid asynchronous design. Practice synchronous design until a better methodology comes along (if ever).
- Pinpoint the unavoidable asynchronous interfaces between different subsystems and the outside world, and provide reliable synchronizers.

easier now than it was 25 or even 10 years ago, but you still can't spend 100% or even 50% of your time on the fun stuff.

Besides the fun stuff and turning the crank, there are many other areas in which a successful digital designer must be competent, including the following:

- *Debugging.* It's next to impossible to be a good designer without being a good troubleshooter. Successful debugging takes planning, a systematic approach, patience, and logic: if you can't discover where a problem *is*, find out where it is *not*!
- *Business requirements and practices.* A digital designer's work is affected by a lot of non-engineering factors, including documentation standards, component availability, feature definitions, target specifications, task scheduling, office politics, and going to lunch with vendors.
- *Risk-taking*. When you begin a design project, you must carefully balance risks against potential rewards and consequences, in areas ranging from component selection (Will it be available when I'm ready to build the first prototype?) to schedule commitments (Will I still have a job if I'm late?).
- *Communication.* Eventually, you'll hand off your successful designs to other engineers, other departments, and customers. Without good communication skills, you'll never complete this step successfully. Keep in mind that communication includes not just transmitting but also receiving—learn to be a good listener!

In the rest of this chapter, and throughout the text, I'll continue to state some opinions about what's important and what is not. I think I'm entitled to do so as a moderately successful practitioner of digital design.

1.2 Analog versus Digital

Analog devices and systems process time-varying signals that can take on any value across a continuous range of voltage, current, or other measurable physical quantity. So do *digital* circuits and systems; the difference is that we can pretend that they don't! A digital signal is modeled as taking on, at any time, only one of two discrete values, which we call *0* and *1* (or LOW and HIGH, FALSE and TRUE, negated and asserted, Frank and Teri, or whatever).

analog

digital

0

1

Digital computers have been around since the 1940s, and they've been in widespread commercial use since the 1960s. Yet only in the past few decades has the "digital revolution" spread to many other aspects of life. Examples of once-analog systems that have now "gone digital" include the following:

- *Still pictures*. Twenty years ago, the majority of cameras still used silver-halide film to record images. Today, inexpensive digital cameras and smartphones record a picture as a 1920×1080 or larger array of pixels, where each pixel stores the intensities of its red, green, and blue color components as 8 or more bits each. This data, almost 50 million bits in this example, is usually processed and compressed in JPEG format down to as few as 5% of the original number of bits. So, digital cameras rely on both digital storage and digital processing.

- *Video recordings*. "Films" are no longer stored on film. A Blu-ray disc (BD) stores video in a highly compressed digital format called MPEG-4. This standard compresses a small fraction of the individual video frames into a format similar to JPEG, and encodes each other frame as the difference between it and the previous one. The capacity of a dual-layer BD is about 400 billion bits, sufficient for about 2 hours of high-definition video.
- *Audio recordings*. Once made exclusively by impressing analog waveforms onto magnetic tape or vinyl, audio recordings are now made and delivered digitally, using a sequence of 16- to 24-bit values corresponding to samples of the original analog waveform, and up to 192,000 samples per second per audio channel. The number of bits, samples, and channels depends on the recording format; a compact disc (CD) stores two channels of 44,100 16-bit values for up to 73 minutes of stereo audio. Like a still picture or a video recording, an audio recording may be compressed for delivery to or storage on a device such as a smartphone, typically using a format called MP3.
- *Automobile carburetors*. Once controlled strictly by mechanical linkages (including clever "analog" mechanical devices that sensed temperature, pressure, etc.), automobile engines are now controlled by embedded microprocessors. Various electronic and electromechanical sensors convert engine conditions into numbers that the microprocessor can examine to determine how to control the flow of fuel and oxygen to the engine. The microprocessor's output is a time-varying sequence of numbers that operate electromechanical actuators which, in turn, control the engine.
- *The telephone system*. It started out over a hundred years ago with analog microphones and receivers connected to the ends of a pair of copper wires (or was it string?). Even today, many homes still use analog telephones, which transmit analog signals to the phone company's central office (CO). However, in the majority of COs, these analog signals are converted into a digital format before they are routed to their destinations, be they in the same CO or across the world. For many years, private branch exchanges (PBXs) used by businesses have carried the digital format all the way to the desktop. Now most businesses, COs, and traditional telephony service providers have converted to integrated systems that combine digital voice with data traffic over a single IP (Internet Protocol) network.
- *Traffic lights*. Stop lights used to be controlled by electromechanical timers that would give the green light to each direction for a predetermined amount of time. Later, relays were used in controllers that could activate

the lights according to the pattern of traffic detected by sensors embedded in the pavement. Today's controllers use microprocessors and can control the lights in ways that maximize vehicle throughput or, in Sunnyvale, California, frustrate drivers with all kinds of perverse behavior.

- *Movie effects.* Special effects used to be created exclusively with miniature clay models, stop action, trick photography, and numerous overlays of film on a frame-by-frame basis. Today, spaceships, cities, bugs, and monsters are synthesized entirely using digital computers. Even actors and actresses have been created or recreated using digital effects.

The electronics revolution has been going on for quite some time now, and the "solid-state" revolution began with analog devices and applications like transistors and transistor radios. So why has there now been a *digital* revolution? There are in fact many reasons to favor digital circuits over analog ones, including:

- *Reproducibility of results*. Given the same set of inputs (in both value and time sequence), a properly designed digital circuit always produces exactly the same results. The outputs of an analog circuit vary with temperature, power-supply voltage, component aging, and other factors.
- *Ease of design*. Digital design, often called "logic design," is logical. No special math skills are needed, and the behavior of small logic circuits can be mentally visualized without any special insights about the operation of capacitors, transistors, or other devices that require calculus to model.
- *Flexibility and functionality.* Once a problem has been reduced to digital form, it can be solved using a set of logical steps in space and time. For example, you can design a digital circuit that scrambles your recorded voice so it is absolutely indecipherable by anyone who does not have your "key" (password), but it can be heard virtually undistorted by anyone who does. Try doing that with an analog circuit.
- *Programmability.* You're probably already quite familiar with digital computers and the ease with which you can design, write, and debug programs for them. Well, guess what? Most of digital design is done today by writing "programs" too, in *hardware description languages (HDLs)*.

  hardware description language (HDL)

  While they're not "programming" languages in the sense of C++ or Java, HDLs allow both structure and function of a digital circuit to be specified or *modeled* with language-based constructs rather than a circuit diagram. Moreover, besides a compiler, an HDL also comes with simulation and synthesis programs that are used to test the hardware model's behavior before any real hardware is built, and then to synthesize the model into a circuit in a particular component technology. This saves a lot of work, because the synthesized circuit typically has a lot more detail than the model that generated it.

  hardware model

PROGRAMS, MODELS, MODULES, AND CODE

As you'll see throughout this text, Verilog HDL examples look a lot like "programs" and are even labeled as such. But generally they are *not* programs in the sense that C++ or Java programs execute a sequence of instructions to produce a result. Rather, they are *models* of hardware structures that receive input signals and produce output signals on wires, and that's something quite different. Since we'll show you hardware basics before we get into HDL models, you should be able to understand the difference when we get there. To help you, we will avoid calling an HDL model a "program."

Verilog can also be used to write procedural programs called "test benches" that do not model hardware. A test bench *exercises* a hardware model, applying a sequence of inputs to it and observing the resulting outputs, and we will actually sometimes call it a "program" and never a "model."

To model a piece of hardware, Verilog typically uses statements in a construct called a *module*, which may be stored in a single text file. We could call such a text file either a module or a model, and we will. However, a complex piece of hardware may be modeled hierarchically using *multiple* modules, so in that case, its model is a collection of modules.

If none of the above terms seems quite appropriate for describing a particular bit of Verilog, we may just call it Verilog "code," for lack of a better short term.

- *Speed.* Today's digital devices are very fast. Individual transistors in the fastest integrated circuits can switch in less than 10 picoseconds, and a complex circuit built from these transistors can examine its inputs and produce an output in less than a nanosecond. A device incorporating such circuits can produce a billion or more results per second.
- *Economy.* Digital circuits can provide a lot of functionality in a small space. Circuits that are used repetitively can be "integrated" into a single "chip" and mass-produced at very low cost, making possible throw-away items like calculators, digital watches, and singing birthday cards. (You may ask, "Is this such a good thing?" Never mind!)
- *Steadily advancing technology.* When you design a digital system, you almost always know that there will be a faster, cheaper, or otherwise better technology for it in a few years. Clever designers can accommodate these expected advances during the initial design of a system, to forestall system obsolescence and to add value for customers. For example, desktop computers often have "expansion sockets" to accommodate faster processors or larger memories than are available at the time of the computer's introduction.

So, that's enough of a sales pitch on digital design. The rest of this chapter will give you a bit more technical background to prepare you for the rest of the book.

SHORT TIMES A *millisecond (ms)* is 10^{-3} second, and a *microsecond (µs)* is 10^{-6} second. A *nanosecond (ns)* is just 10^{-9} second, and a *picosecond (ps)* is 10^{-12} second. In a vacuum, light travels about a foot in a nanosecond, and an inch in 85 picoseconds. With individual transistors in the fastest integrated circuits now switching in less than 10 picoseconds, the speed-of-light delay between these transistors across a half-inch-square silicon chip has become a limiting factor in circuit design.

1.3 Analog Signals

Marketing hype notwithstanding, we live in an analog world, not a digital one. Voltages, currents, and other physical quantities in real circuits take on values that are infinitely variable, depending on properties of the real devices that comprise the circuits. Because real values are continuously variable, we could use a physical quantity such as a signal voltage in a circuit to represent a real number (e.g., 3.14159265358979 volts represents the mathematical constant *pi* to 14 decimal digits of precision).

However, stability and accuracy in physical quantities are difficult to obtain in real circuits. They can be affected by manufacturing variations, temperature, power-supply voltage, cosmic rays, and noise created by other circuits, among other things. If we used an analog voltage to represent *pi*, we might find that instead of being an absolute mathematical constant, *pi* varied over a range of 10% or more.

Also, many mathematical and logical operations can be difficult or impossible to perform with analog quantities. While it is possible with some cleverness to build an analog circuit whose output voltage is the square root of its input voltage, no one has ever built a 100-input, 100-output analog circuit whose outputs are a set of voltages identical to the set of input voltages, but sorted arithmetically.

1.4 Digital Logic Signals

digital logic
digital signals
logic values

Digital logic hides the pitfalls of the analog world by using *digital signals*, where the infinite set of real values for a physical quantity are mapped into two subsets corresponding to just two possible numbers or *logic values*: 0 and 1. Thus, digital logic circuits can be analyzed and designed functionally, using switching algebra, tables, and other abstract means to describe the operation of well-behaved 0s and 1s in a circuit.

binary digit
bit

A logic value, 0 or 1, is often called a *binary digit*, or *bit*. If an application requires more than two discrete values, additional bits may be used, with a set of n bits representing 2^n different values.

Examples of the physical phenomena used to represent bits in some modern (and not-so-modern) digital technologies are given in Table 1-1. With

Table 1-1 Physical states representing bits in different logic and memory technologies.

	State Representing Bit	
Technology	***0***	***1***
Pneumatic logic	Fluid at low pressure	Fluid at high pressure
Relay logic	Circuit open	Circuit closed
Transistor-transistor logic (TTL)	0–0.8 V	2.0–5.0 V
Complementary metal-oxide semiconductor (CMOS) 2-volt logic	0–0.5 V	1.5–2.0 V
Dynamic memory	Capacitor discharged	Capacitor charged
Nonvolatile, erasable memory	Electrons trapped	Electrons released
On-chip nonvolatile security key	Fuse blown	Fuse intact
Polymer memory	Molecule in state A	Molecule in state B
Fiber optics	Light off	Light on
Magnetic disk or tape	Flux direction "north"	Flux direction "south"
Compact disc (CD), digital versatile disc (DVD), and Blu-ray disc (BD)	No pit	Pit
Writable compact disc (CD-R)	Dye in crystalline state	Dye in noncrystalline state

most phenomena, there is an undefined region between the 0 and 1 states (e.g., voltage = 1.0 V, dim light, capacitor slightly charged, etc.). This undefined region is needed so the 0 and 1 states can be unambiguously defined and reliably detected. Noise can more easily corrupt results if the boundaries separating the 0 and 1 states are too close to each other.

LOW
HIGH

When discussing electronic logic circuits like CMOS, digital designers often use the words "LOW" and "HIGH" in place of "0" and "1" to remind them that they are dealing with real circuits, not abstract quantities:

LOW A signal in the range of algebraically lower voltages, which is interpreted as a logic 0.

HIGH A signal in the range of algebraically higher voltages, which is interpreted as a logic 1.

STATE TRANSITIONS

The last four technologies in Table 1-1 don't actually use absolute states to represent bit values. Rather, they use transitions (or absence of transitions) between states to represent 0s and 1s using a code such as the Manchester code described on page 82.

THE DIGITAL ABSTRACTION Digital circuits are not exactly a binary version of alphabet soup—with all due respect to our forthcoming descriptions like Figure 1-3, digital circuits don't have little 0s and 1s floating around in them. As you'll see in Chapter 14, digital circuits deal with analog voltages and currents and are built with analog components. The "digital abstraction" allows analog behavior to be ignored in most cases, so circuits can be modeled as if they really did process 0s and 1s.

Note that the assignments of 0 and 1 to LOW and HIGH are somewhat arbitrary. Still, assigning 0 to LOW and 1 to HIGH seems natural and is called *positive logic*, and that's what we use in this book exclusively. The opposite assignment, 1 to LOW and 0 to HIGH, is not often used and is called *negative logic*.

positive logic
negative logic

Because a wide range of physical values represent the same binary value, digital logic is highly immune to component and power-supply variations and noise. Furthermore, *buffer* circuits can be used to regenerate (or amplify) "weak" values into "strong" ones, so that digital signals can be transmitted over arbitrary distances without loss of information. For example, using the voltage ranges in the fourth row of Table 1-1, a buffer for 2-volt CMOS logic converts any LOW input voltage into an output very close to 0.0 V, and any HIGH input voltage into an output very close to 2.0 V.

buffer

1.5 Logic Circuits and Gates

A logic circuit can be represented with a minimum amount of detail simply as a "black box" with a certain number of inputs and outputs. For example, Figure 1-1 shows a logic circuit with three inputs and one output. However, this representation does not describe how the circuit responds to input signals.

From the point of view of electronic circuit design, it takes a lot of information to describe the precise electrical behavior of a circuit. However, since the inputs of a digital logic circuit can be viewed as taking on only discrete 0 and 1 values, the circuit's "logical" operation can be described with a table that ignores electrical behavior and lists only discrete 0 and 1 values.

A logic circuit whose outputs depend only on its current inputs is called a *combinational circuit*. Its operation is fully described by a *truth table* that lists all combinations of input values and the output value(s) produced by each one.

combinational circuit
truth table

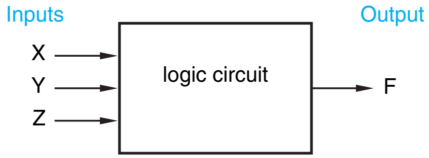


Figure 1-1 "Black-box" representation of a 3-input, 1-output logic circuit.

Table 1-2
Truth table for a combinational logic circuit.

X	Y	Z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Table 1-2 is the truth table for a logic circuit with three inputs X, Y, and Z and a single output F. This truth table lists all eight possible combinations of values of X, Y, and Z and the circuit's output value F for each combination.

sequential circuit
state table

A circuit with memory, whose outputs depend on the current input *and* the sequence of past inputs, is called a *sequential circuit*. Its behavior may be described by a *state table* that specifies its output and next state as functions of its current state and input. Sequential circuits will be introduced in Chapter 9.

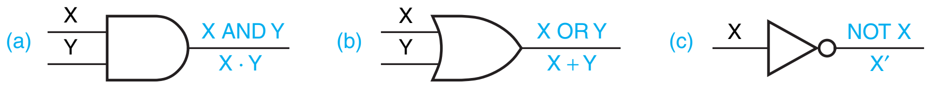
gate

The most basic digital devices are called *gates*, and no, they were not named after the founder of a large software company. Gates originally got their name from their function of allowing or retarding ("gating") the flow of digital information. In general, a gate has one or more inputs and produces an output that is a function of the current input value(s). While the inputs and outputs may be analog conditions such as voltage, current, even hydraulic pressure, they are modeled as taking on just two discrete values, 0 and 1.

As we'll show in Section 3.1, just three basic logic functions, AND, OR, and NOT, can be used to build any combinational digital logic circuit. The graphical symbols for these logic gates are shown in Figure 1-2 along with their corresponding truth tables. The gates' functions are easily defined in words:

AND gate
OR gate

- An *AND gate* produces a 1 output if and only if all of its inputs are 1.
- An *OR gate* produces a 1 if and only if one or more of its inputs is 1.



X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

X	Y	X OR Y
0	0	0
0	1	1
1	0	1
1	1	1

X	NOT X
0	1
1	0

Figure 1-2
Basic logic gates: (a) AND; (b) OR; (c) NOT (inverter).

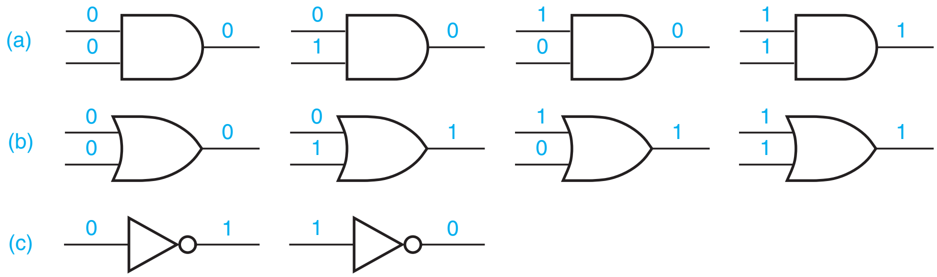


Figure 1-3
Logic gates with input values and outputs:
(a) AND; (b) OR;
(c) NOT or inverter.

- A *NOT gate,* usually called an *inverter*, produces an output value that is the opposite of its input value.

NOT gate
inverter

Notice that in the definitions of AND and OR functions, we only had to state the input conditions for which the output is 1, because there is only one possibility when the output is not 1—it must be 0. The symbols and truth tables for AND and OR may be extended to gates with any number of inputs, and the functional definition above already covers such cases.

Figure 1-2 also shows, in color, algebraic expressions for the output of each gate, using both words and mathematical symbols for the logic operations. The symbols are used in switching algebra, which we introduce in Chapter 3. Figure 1-3 shows the gates' graphical symbols again, each with all possible combinations of inputs that may be applied to it, and the resulting outputs.

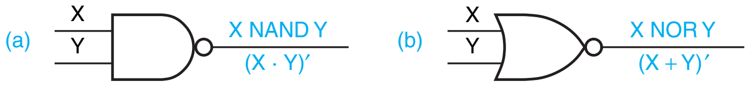
The circle on the inverter symbol's output is called an *inversion bubble* and is used in this and other gate symbols to denote "inverting" behavior. For example, two more logic functions are obtained inverting the outputs of AND and OR gates. Figure 1-4 shows the truth tables, graphical symbols, and algebraic expressions for the new gates. Their functions are also easily described in words:

inversion bubble

- A *NAND gate* produces the opposite of an AND gate's output, a 0 if and only if all of its inputs are 1.
- A *NOR gate* produces the opposite of an OR gate's output, a 0 if and only if one or more of its inputs are 1.

NAND gate
NOR gate

As with AND and OR gates, the symbols and truth tables for NAND and NOR may be extended to gates with any number of inputs.



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

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

Figure 1-4
Inverting gates:
(a) NAND; (b) NOR.

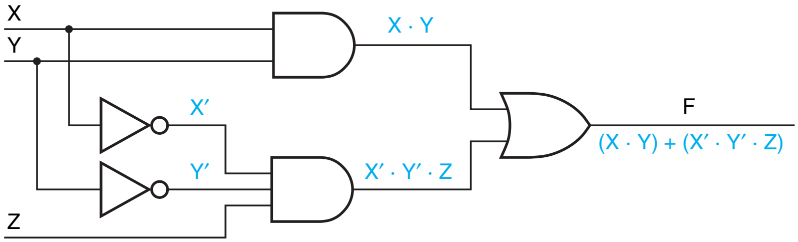


Figure 1-5
Logic circuit with the truth table of Table 1-2.

logic diagram
wire

A *logic diagram* shows the graphical symbols for multiple logic gates and other elements in a logic circuit, and their interconnections (*wires*). The output of each element may connect to inputs of one or more other elements. Signals traditionally flow from left to right, and inputs and outputs of the overall circuit are drawn on the left and right, respectively.

Figure 1-5 is the logic diagram of a circuit using AND, OR, and NOT gates that functions according to the truth table of Table 1-2. In Chapter 3, you'll learn how to go from a truth table to a logic circuit, and vice versa. You'll also learn how to derive an algebraic expression for the logic function on any signal wire (again shown in color in the figure). Most importantly, you'll learn how to create a logic circuit in any of several different structures to perform the operation defined by any given algebraic expression.

We mentioned at that outset that digital circuits called sequential circuits have memory, so their outputs depend on the sequence of past inputs as well as their current inputs. The simplest sequential circuits are latches and flip-flops, each of which stores a single 0 or 1. These devices are typically interconnected with gates or more complex combinational circuits to create larger sequential circuits. You'll learn about sequential circuits beginning in Chapter 9.

timing diagram

In this section's examples, the digital abstraction allowed us to ignore most of the analog aspects of logic signals, like voltage and current, but logic circuits function in another very important analog dimension—time. For example, Figure 1-6 is a *timing diagram* that graphically shows how the circuit of Figure 1-5 might respond to a time-varying pattern of input signals. Time is graphed horizontally, and logic values are graphed vertically. While each logic

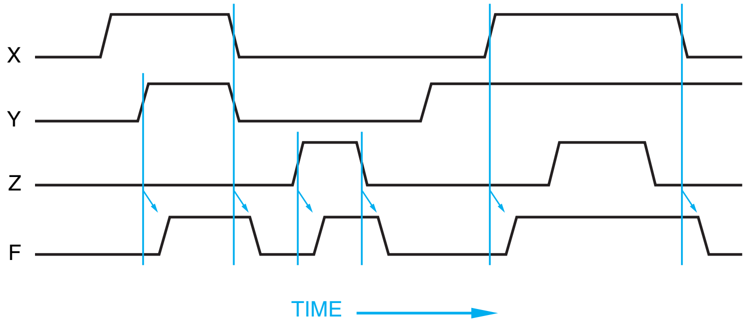


Figure 1-6
Timing diagram for a logic circuit.

signal has only two possible values in the steady state, this timing diagram is drawn with the signal transitions sloped, reminding us that the logic signals do not change instantaneously between the analog values corresponding to 0 and 1. Also, looking at the colored vertical reference lines and arrows, you can see there is a lag between an input change in X, Y, or Z and the beginning of the resulting change in the output F. In later chapters, you'll learn how the timing behavior of digital devices and circuits is specified and handled.

1.6 Software Aspects of Digital Design

Digital design need not involve any software tools. For example, Figure 1-7 shows the primary tool of what used to be the "old school" and could now be called the "prehistoric school" of digital design—a plastic template for drawing symbols in logic diagrams by hand (the designer's name was engraved into the plastic with a soldering iron).

Today, however, software tools are an essential part of digital design. Indeed, the availability and practicality of hardware description languages (HDLs) and accompanying circuit simulation and synthesis tools have changed the entire landscape of digital design over the past decades. We'll use the Verilog HDL extensively throughout this book.

electronic design automation (EDA)

Modern *electronic design automation (EDA)* tools improve the designer's productivity and help to improve the correctness and quality of designs. In a competitive world, the use of software tools is mandatory to obtain high-quality results on aggressive schedules. Important examples of EDA tools for digital design are listed below:

- *Schematic entry.* This is the digital designer's equivalent of a word processor. It allows schematic diagrams—fully detailed logic diagrams—to be drawn "online," instead of with paper and pencil. The more advanced schematic-entry programs also check for common, easy-to-spot errors, like shorted outputs, signals that don't go anywhere, and so on.
- *HDLs.* Hardware description languages, originally developed for circuit modeling and simulation, are now used extensively for hardware *design.*

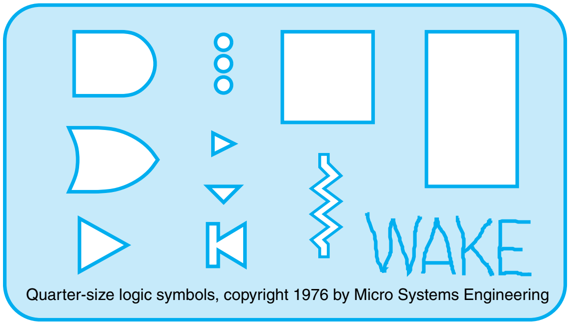

WAKE
Quarter-size logic symbols, copyright 1976 by Micro Systems Engineering

Figure 1-7
A logic-design template.

They can be used to design anything from small, individual function modules to large, multichip digital systems. We will introduce Verilog, one of two predominant HDLs, in Chapter 5. If you go on to practice digital design in industry, you'll be likely also to encounter and learn the other, VHDL. Both HDLs may be used in large, multi-module system designs, especially when the modules are supplied by different providers.

- *HDL text editors, compilers, and synthesizers.* A typical HDL software package has many components. The designer uses a text editor to write an HDL model, and an HDL compiler checks it for syntax and related errors. The designer then can hand over the model to a synthesizer that creates (or "synthesizes") a corresponding circuit targeted to a particular hardware technology. Most often though, before synthesis, the designer runs the HDL model on a "simulator" to verify the behavior of the design.
- *Simulators.* The design cycle for a customized, single-chip digital integrated circuit is long and expensive. Once the first chip is built, it's very difficult, often impossible, to debug it by probing internal connections (they are really tiny), or to change the gates and interconnections. Usually, changes must be made in the original design database, and a new chip must be manufactured to incorporate the required changes. Since this process can take months to complete and cost hundreds of thousands of dollars, chip designers are highly motivated to "get it right" on the first try. Simulators help designers predict the electrical and functional behavior of a chip without actually building it, allowing most if not all bugs to be found before the chip is fabricated.
- Simulators are also used in the overall design of systems that incorporate many individual components. They are somewhat less critical in this case because it's possible for the designer to make changes in components and interconnections on a printed-circuit board, though the process can take anywhere from hours (cutting connections and making new ones with tiny wires) to weeks (modifying a printed-circuit board's layout and fabricating a new one). So, even a little bit of simulation can save time and expense by catching mistakes early.
- Simulators are also used extensively to check for proper operation of designs that are implemented in programmable devices, introduced in Section 1.10. These devices can be easily reprogrammed if the design doesn't work correctly the first time, so why not just try out a new design on the real device instead of simulating, and reprogram it if it doesn't work? The answer is that a simulator is like a software debugger—besides showing you the device outputs that are produced by applying various inputs, the simulator lets you observe what's happening *inside* the design so you can more easily determine what's gone wrong and how to fix it.

- If you think you've already seen already too many bullets on "simulators," you're wrong—they are really important. In the design of chips, systems, and even HDL-based programmable logic devices, more engineering time may actually be devoted to simulation than to the original logic design!
- *Test benches*. HDL-based digital designs are simulated and tested using software called "test benches." The idea is to build a set of programs around the HDL models to automatically exercise them, checking both their functional and their timing behavior. This is especially handy when small design changes are made—the test bench can be run to ensure that bug fixes or "improvements" in one area do not break something else. Test-bench programs may be written in the same HDL as the design itself, in C or C++, or in a combination of languages including scripting languages like Perl.
- *Timing analyzers and verifiers*. The time dimension is very important in digital design. All digital circuits take time to produce a new output value in response to an input change, and much of a designer's effort is spent ensuring that such output changes occur quickly enough (or, in some cases, not *too* quickly). Timing analysis and verification is often a component of the simulation program and environment. In addition, specialized tools can automate the tedious task of specifying and verifying timing relationships between different signals in a complex system.
- *Word processors*. HDL-specific text editors are useful for writing source code, but word processors supporting fancy fonts and pretty graphics also have an important use in every design—to create documentation!

In addition to using the tools above, designers may sometimes write specialized programs in high-level languages such as C or C++, or scripts in languages like TCL and Perl, to solve particular design problems. For example, Section 6.1.2 gives an example where a program is needed to generate the truth table of a complex combinational logic function.

As you continue to study and use EDA tools, you will encounter other nomenclature and acronyms that refer to them. In particular, *computer-aided engineering (CAE)* usually refers to tools that are used at the "front end" of the design process, including the tools listed above. On the other hand, *computer-aided design (CAD)* usually refers to "back-end" tools, such as the ones that are used to place components or route their interconnections on a customized chip. Note that the term "CAD" is also used predominantly in nonelectronic physical design, such as mechanical design and architecture (as in the "AutoCAD" tool).

computer-aided engineering (CAE)

computer-aided design (CAD)

Although EDA tools are important, they don't make or break a digital designer. To take an analogy from another field, you couldn't consider yourself to be a great writer just because you're a fast typist or very handy with a word processor. During your study of digital design, be sure to learn and use all the tools that are available to you, such as HDL-specific text editors and compilers, simulators, and timing analyzers. But remember that learning to use tools is no guarantee that you'll be able to produce good results. Please pay attention to what you're producing with them!

1.7 Integrated Circuits

integrated circuit (IC)

A collection of one or more gates fabricated on a single silicon chip is called an *integrated circuit (IC)*. Large ICs with hundreds of millions of transistors may be 10 millimeters (mm) or more on a side, while small ICs may be less than 1 mm on a side.

wafer

die

pad

Regardless of its size, an IC is initially part of a much larger circular *wafer*, up to 300 mm in diameter, containing dozens to thousands of replicas of the same IC. All of the IC chips on the wafer are fabricated at the same time, like pizzas that are eventually sold by the slice. In the case of IC chips, the pieces are cut into rectangles and each piece is called a *die*. Each die has *pads* around its periphery—electrical contact points that are much larger than other chip features, so wires can be connected later. After the wafer is fabricated, the dice are tested in place on the wafer using tiny probing pins that temporarily contact the pads, and defective dice are marked. Then the wafer is sliced up to produce the individual dice, and the marked ones are discarded. (Compare with the pizza-maker who sells all the pieces, even the ones without enough pepperoni!) Each "good" die is mounted in a package, its pads are wired to the package pins, the packaged IC is subjected to a final test, and it is shipped to a customer.

IC

Some people use the term "IC" to refer to a silicon die. Some use "chip" to refer to the same thing. Still others use "IC" or "chip" to refer to the combination of a silicon die and its package. Digital designers tend to use the two terms interchangeably, and they really don't care what they're talking about. They don't require a precise definition, since they're only looking at the functional and electrical behavior of these things. In the balance of this text, we'll use the term *IC* to refer to a packaged die.

small-scale integration (SSI)

In the early days of integrated circuits, ICs were classified by size—small, medium, or large—according to how many gates they contained. The simplest type of commercially available ICs are still called *small-scale integration (SSI)* and contain the equivalent of 1 to 20 gates. SSI ICs typically contain a handful of gates or flip-flops, the basic building blocks of digital design.

PIZZA ROMA You might debate whether pizzas should be cut into wedge-shaped slices or into rectangles. I started my career at Pizza Roma, and like most Chicago pizzerias, we cut our pies into rectangles. As far as I'm concerned, that's the only way to do it!

NOT A DICEY DECISION

A reader of a previous edition wrote to me to collect a $5 reward for pointing out my “glaring” misuse of “dice” as the plural of “die.” According to the dictionary, she said, the plural form of “die” is “dice” *only* when describing those little cubes with dots on each side; otherwise it’s “dies,” and she produced the references to prove it.

I disagreed with her then, and being stubborn, I recently did a Google search to see which usage is more common. Yes, Google now reported about 50,000 Web pages containing the word sequence “integrated circuit dies” and only about 30,000 with “integrated circuit dice.” But “dice” still wins, as far as I’m concerned. Its first page of results had six integrated circuit patents, two administrative law citations, and a site devoted to IC skills. On the other hand, half the hits in the first page for “dies” were in headlines like “Jack Kilby, inventor of the integrated circuit, dies” (in 2005). So, out of respect for Jack, whom I once met, I’m sticking with “dice”!

The SSI ICs that you might encounter in an educational lab come in a 14-pin *dual inline-pin (DIP)* package. As shown in Figure 1-8(a), the space between pins in a column is 0.1 inch, and the space between columns is 0.3 inch. Larger DIP packages accommodate functions with more pins, as shown in (b) and (c). A *pin diagram* supplied by the manufacturer shows the assignment of device signals to package pins, or *pinout*. A 14-pin package might contain four 2-input AND or OR gates or six inverters. SSI ICs are rarely used in new designs except as “glue,” for example, to invert the polarity of a control signal between two otherwise compatible larger-scale devices.

dual inline-pin (DIP) package

pin diagram

pinout

The next larger commercially available ICs, which you also might use in an educational lab, are called *medium-scale integration (MSI)* and contain the equivalent of about 20 to 200 gates. An MSI IC typically contains a functional building block, like a decoder, register, or counter; we’ll describe such blocks in later chapters. Even though discrete MSI ICs are rarely used in new designs, equivalent building blocks are used frequently within larger ICs.

medium-scale integration (MSI)

Large-scale integration (LSI) ICs are bigger still; the term originated at a time when 1,000 gates seemed like a lot. LSI parts included small memories, the first microprocessors, programmable logic devices, and customized devices. As

large-scale integration (LSI)

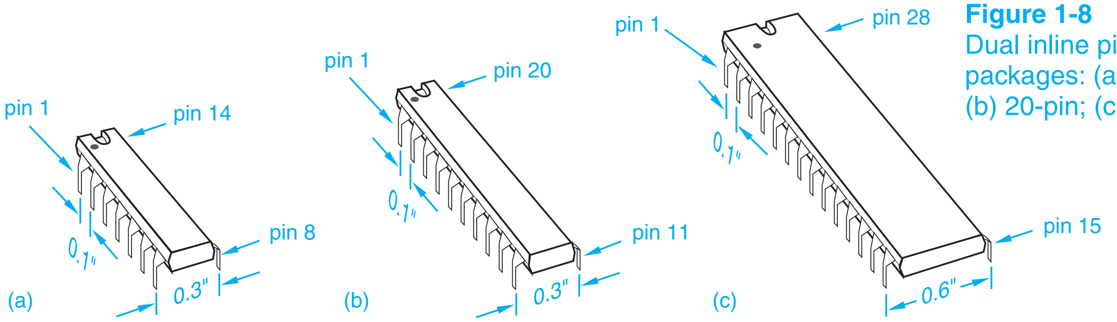


Figure 1-8
Dual inline pin (DIP) packages: (a) 14-pin; (b) 20-pin; (c) 28-pin.

very large-scale integration (VLSI)

chip densities continued to increase, the term *very large-scale integration (VLSI)* gradually came into use.

As LSI evolved into VLSI, and an increasing number of ICs combined both logic gates and memories, chip sizes came to be stated in terms of number of transistors rather than gates. This was more representative, independent of the logic and memory mix, because typical logic gates use two transistors per input, while different memories use one to six transistors per bit. In 2017, the largest commercially available VLSI devices contained over ten billion transistors.

At one time, marketers and engineers who love to classify things flirted with names like “ULSI” for even higher density chips beyond VLSI. But the huge number of transistors available nowadays on ordinary, inexpensive ICs makes such classifications irrelevant. Economics favors functional integration well beyond mere “LSI,” and most new digital ICs today are VLSI.

IC process

process node

There’s another set of nomenclature that you’ll encounter when working with VLSI chips. The main reason for ever-increasing transistor counts is the industry’s ever-improving ability to make smaller transistors, so the chips have higher density (transistors per unit area). An *IC process* is the set of technologies, fabrication steps, and other characteristics associated with manufacturing ICs of a given density. Different chip manufacturers have their own proprietary processes, but all of the processes that produce ICs of a similar density are said to belong to a particular *process node*. The node is identified by a number that roughly corresponds to the smallest linear dimensions of physical features on a chip, such as the widths of signal lines or transistors.

submicron process

deep submicron process

Intel’s first microprocessor, the 4004, was fabricated in 1971 using a 10 µm (micron, 10^{-6} m) process. In 1985, the Intel 80386, whose architecture is the basis of today’s personal computers, was launched using a 1 µm process. Processes with even smaller dimensions are called *submicron processes*. By 1999, most manufacturers were able to achieve 250 nm (nanometers, 10^{-9} m), and denser processes became known as *deep submicron processes*. By 2006, much of the industry had achieved 45 nm. In 2015, the microprocessors in many computers and smartphones used a 14 nm process. Major chip makers were preparing to fabricate chips at 10 nm in 2017.

So, between 1971 and 2017, the linear dimensions of chip features shrank by a factor of 1,000. Since transistors and wiring are laid out more or less in two dimensions, overall density has increased by a factor of a million or more in that time. Some transistor features are created nowadays by physical stacking, and wiring can overlay transistors, but we still don’t routinely stack multiple vertical layers of transistors on a single chip.

ULSI AND RLSI ULSI stands for “ultra-large-scale integration.” In a previous edition of this text, I suggested RLSI (ridiculously-large-scale integration), but it never caught on.

1.8 Logic Families and CMOS

There are many, many ways to design an electronic logic circuit. The first electrically controlled logic circuits, developed at Bell Laboratories in the 1930s, were based on relays. In the mid-1940s, the first electronic digital computer, the Eniac, used logic circuits based on vacuum tubes. The Eniac had about 18,000 tubes and a similar number of logic gates, not a lot by today's standards of microprocessor chips with billions of transistors. However, the Eniac could hurt you a lot more than a chip could if it fell on you—it was 100 feet long, 10 feet high, 3 feet deep, and consumed 140,000 watts of power!

The inventions of the *semiconductor diode* and the *bipolar junction transistor* allowed the development of smaller, faster, and more capable computers in the late 1950s. In the 1960s, the invention of the *integrated circuit (IC)* allowed multiple diodes, transistors, and other components to be fabricated on a single chip, and computers got still better.

semiconductor diode

bipolar junction transistor

integrated circuit (IC)

The 1960s also saw the introduction of the first integrated-circuit logic families. A *logic family* is a collection of different integrated-circuit chips that have similar input, output, and internal circuit characteristics, but that perform different logic functions. Chips from the same family can be interconnected to perform any desired logic function. Chips from different families might not be compatible; they may use different power-supply voltages or may use different input and output conditions to represent logic values.

logic family

The most successful *bipolar logic family* (one based on bipolar junction transistors) was *transistor-transistor logic (TTL)*. First introduced in the 1960s, TTL evolved into a family of logic families that were compatible with each other but differed in speed, power consumption, and cost. Digital systems could mix components from several different TTL families, according to design goals and constraints in different parts of the system.

bipolar logic family

transistor-transistor logic (TTL)

Ten years *before* the bipolar junction transistor was invented, the principles of operation were patented for another type of transistor, called the *metal-oxide semiconductor field-effect transistor (MOSFET)*, or simply *MOS transistor*. However, MOS transistors were difficult to fabricate in the early days, and it wasn't until the 1960s that a wave of developments made MOS-based logic and memory circuits practical. Even then, MOS circuits lagged bipolar circuits considerably in speed. They were attractive in only a few applications because of their lower power consumption and higher levels of integration.

metal-oxide semiconductor field-effect transistor (MOSFET)

MOS transistor

Beginning in the mid-1980s, advances in the design of MOS circuits, in particular *complementary MOS (CMOS)* circuits, tremendously increased their performance and popularity. Today, almost all large-scale integrated circuits, such as microprocessors, memories, and programmable logic devices, use CMOS. Even small- to medium-scale applications, for which engineers used to put together a customized collection of TTL devices, are now typically handled by a CMOS microprocessor or by one or a few CMOS programmable devices,

complementary MOS (CMOS)

GREEN STUFF Nowadays, the acronym “MOS” is usually spoken as “moss,” rather than spelled out. Hence in this book, we say “a MOS transistor,” not “an MOS transistor.” And “CMOS” has always been spoken as “sea moss.”

LEGACY LOGIC SSI and MSI device part numbers are written “*74FAMnn*” where *FAM* designates a family like LS, HC, or AC, and two or more digits *nn* designate the function; for example, a 74HC00 is a High-speed CMOS NAND gate. To specify only the function and not the family, we write “74x00” or simply “ ’00” without the “74x.”

achieving more functionality, higher speed, and lower power consumption. A few CMOS SSI and MSI parts might be used to tie up the loose ends. CMOS circuits account for the vast majority of the worldwide integrated-circuit market.

1.9 CMOS Logic Circuits

CMOS logic is both the most capable and the easiest to understand commercial digital logic technology. In Chapter 14, we will describe CMOS logic circuits in a fair amount of detail, from their basic structure to their electrical characteristics, and we will introduce some common variants of CMOS logic families. In this section, we’ll give you a small, only mildly “electronic” preview of CMOS operation which will serve you until then.

“off” transistor

“on” transistor

A MOS transistor can be modeled as a 3-terminal device that acts like a voltage-controlled resistance. In digital logic applications, a MOS transistor is operated so its resistance is always either very high (and the transistor is “off”) or very low (and the transistor is “on”).

n-channel MOS (NMOS) transistor

gate

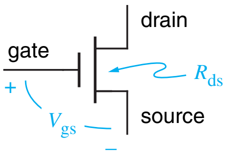
source

drain

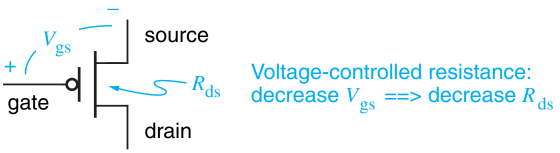
There are two types of MOS transistors, *n*-channel and *p*-channel: the names refer to the type of semiconductor material used in the controlled resistance. A simplified circuit symbol for an *n-channel MOS (NMOS) transistor* is shown in Figure 1-9. The terminals are called *gate, source,* and *drain*. Note the “gate” of a MOS transistor is not a “logic gate,” though it does “gate” the flow of current between the other two terminals. As you might guess from the orientation of the circuit symbol, the drain is normally at a higher voltage than the source.

The voltage from gate to source (V_{gs}) in an NMOS transistor controls R_{ds}, the resistance between the drain and the source. If V_{gs} is zero or negative, then

Figure 1-9 Simplified circuit symbol for an n-channel MOS (NMOS) transistor.



Voltage-controlled resistance: increase V_{gs} ==> decrease R_{ds}



Voltage-controlled resistance:
decrease V_{gs} ==> decrease R_{ds}

Figure 1-10
Simplified circuit symbol for a *p*-channel MOS (PMOS) transistor.

R_{ds} is very high, at least a megohm (10^6 ohms) or more. As we increase V_{gs} (i.e., increase the voltage on the gate), R_{ds} decreases to a very low value, 10 ohms or less in some devices. In digital applications, V_{gs} is always LOW or HIGH (except during transitions), and the connection between source and drain acts like a logic-controlled switch—open if V_{gs} is LOW, and closed if it is HIGH.

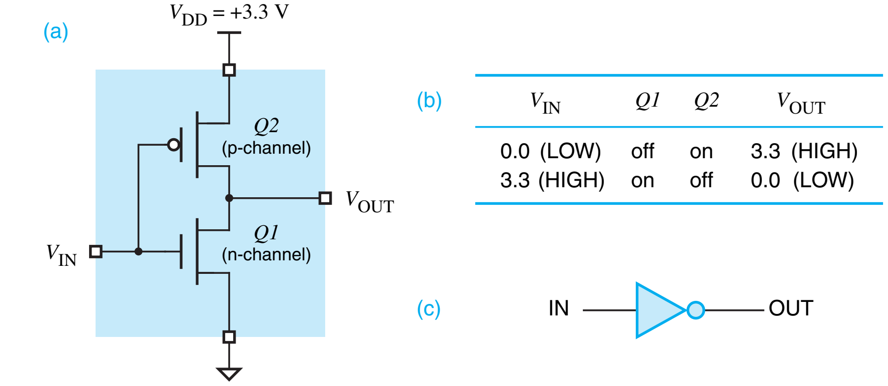
p-channel MOS (PMOS) transistor

A circuit symbol for a *p-channel MOS (PMOS) transistor* is shown in Figure 1-10. Operation is analogous but opposite to that of an NMOS transistor. If V_{gs} is zero or positive, then R_{ds} is very high. As we algebraically decrease V_{gs} (i.e., *decrease* the voltage on the gate), R_{ds} decreases to a very low value. The inversion bubble on the gate of the PMOS transistor's symbol in digital applications reminds us of this "opposite" behavior. Again the connection between source and drain acts like a logic-controlled switch—but it's open if V_{gs} is HIGH, and closed if V_{gs} is LOW.

CMOS logic

NMOS and PMOS transistors are used together in a complementary way to form *CMOS logic*. The simplest CMOS circuit, a logic inverter, requires only one of each type of transistor, connected as shown in Figure 1-11(a). The power-supply voltage, V_{DD}, may be in the range 1–6 V depending on the CMOS family, and is shown as 3.3 V in the figure.

The functional behavior of the CMOS inverter circuit can be characterized by just two cases tabulated shown in Figure 1-11(b). A LOW voltage on the input turns on the *p*-channel transistor *Q2*, and turns off the *n*-channel transistor *Q1*. So, the output is connected to V_{DD} through *Q2* and the output is HIGH. When the input voltage is HIGH, we have the opposite behavior and the output is connected to ground (0 volts) through *Q1* and is LOW. This is clearly an inverter function—the output's logic value is the opposite of the input's.



V_{IN}	*Q1*	*Q2*	V_{OUT}
0.0 (LOW)	off	on	3.3 (HIGH)
3.3 (HIGH)	on	off	0.0 (LOW)

Figure 1-11
CMOS inverter:
(a) circuit diagram;
(b) functional behavior;
(c) logic symbol.

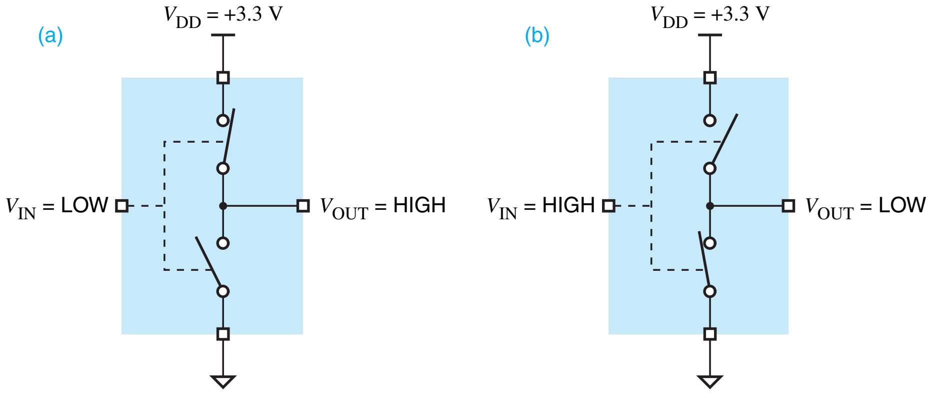
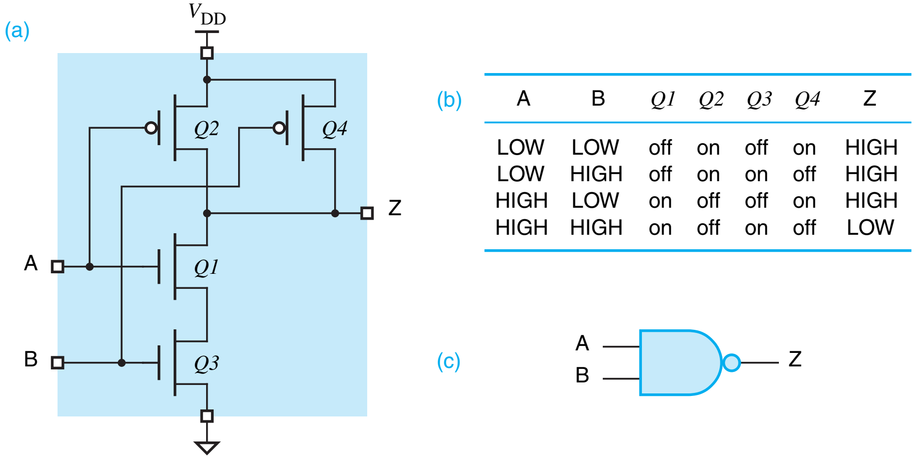


Figure 1-12
Switch model for CMOS inverter:
(a) LOW input;
(b) HIGH input.

The inverter's function can also be visualized using switches. As shown in Figure 1-12(a), the *n*-channel (bottom) transistor is modeled by a normally-open switch, and the *p*-channel (top) transistor by a normally-closed switch. Applying a HIGH voltage "pushes" each switch to the opposite of its normal state, as shown in (b).

Both NAND and NOR gates can be constructed in CMOS using *p*-channel and *n*-channel transistors in a series-parallel configuration. Figure 1-13 shows a 2-input CMOS NAND gate. If either input is LOW, the output Z is connected to V_{DD} through the corresponding "on" *p*-channel transistor, and the path to ground is blocked by the corresponding "off" *n*-channel transistor. If both inputs are HIGH, the path to V_{DD} is blocked, and Z is connected to ground. Figure 1-14 shows the switch model for the NAND gate's operation.

Figure 1-15 shows a CMOS NOR gate. If both inputs are LOW, then the output Z connects to V_{DD} through the "on" *p*-channel transistors, and the path to ground is blocked by the "off" *n*-channel transistors. If either input is HIGH, the path to V_{DD} is blocked, and Z connects to ground



A	B	*Q1*	*Q2*	*Q3*	*Q4*	Z
LOW	LOW	off	on	off	on	HIGH
LOW	HIGH	off	on	on	off	HIGH
HIGH	LOW	on	off	off	on	HIGH
HIGH	HIGH	on	off	on	off	LOW

Figure 1-13
CMOS 2-input NAND gate:
(a) circuit diagram;
(b) function table;
(c) logic symbol.

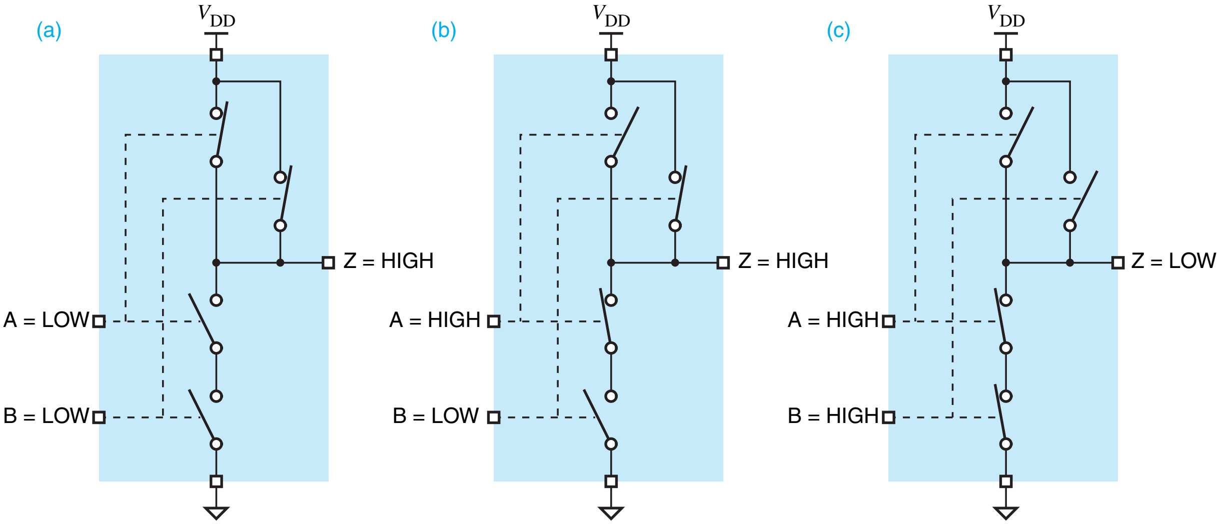
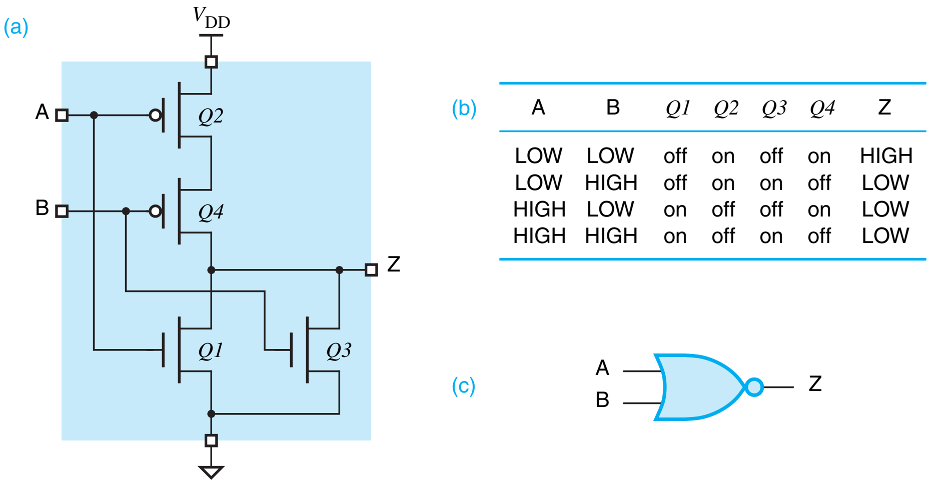


Figure 1-14 Switch model for CMOS 2-input NAND gate: (a) both inputs LOW; (b) one input HIGH; (c) both inputs HIGH.

By extending the series-parallel configurations, you can build a *k*-input CMOS NAND or NOR gate using 2*k* transistors, although there is a limit to *k* based on electrical performance. CMOS inverters, NAND gates, and NOR gates "naturally" perform a logical inversion using the minimal transistor-level circuits that we've shown. To build a noninverting buffer, an AND gate, or an OR gate, you must follow the inverting gate with an inverter, which uses another pair of transistors.

transmission gate

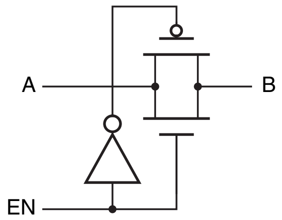
Another important CMOS circuit configuration is the *transmission gate*, which acts as a logic-controlled switch that passes or blocks a CMOS logic signal. A transmission gate is a *p*-channel and an *n*-channel transistor pair with a



A	B	*Q1*	*Q2*	*Q3*	*Q4*	Z
LOW	LOW	off	on	off	on	HIGH
LOW	HIGH	off	on	on	off	LOW
HIGH	LOW	on	off	off	on	LOW
HIGH	HIGH	on	off	on	off	LOW

Figure 1-15 CMOS 2-input NOR gate: (a) circuit diagram; (b) function table; (c) logic symbol.

Figure 1-16
CMOS transmission gate.



control signal and its complement applied as shown in Figure 1-16. When the control signal EN is HIGH, both transistors are on, and a logic signal can pass from A to B or vice versa. When EN is LOW, both transistors are off, and A and B are effectively disconnected. A pair of transistors is needed for analog reasons: when a HIGH logic signal is passed between A and B, the *p*-channel transistor has low resistance; when a LOW signal is passed, the *n*-channel transistor makes the connection. In later chapters, we'll see how transmission gates can be used in multiplexers, flip-flops, and other logic elements.

Besides the basics that we already covered, there are a few more things you should know about CMOS electrical characteristics to augment the digital topics that we will cover between now and Chapter 14:

- With higher power-supply voltages, CMOS circuits run faster and their noise immunity is better, and vice versa. However, with higher voltage they also consume more power.
- In fact, the major portion of CMOS power consumption, called "dynamic power," is proportional to CV^2f, where V is the supply voltage, C is the electrical capacitance of the signal lines that are being switched, and f is the switching frequency. Because of this formula's square term, halving the voltage reduces dynamic power by a factor of 4.

ANALOG STUFF

If you're not an electrical engineer and "analog stuff" bothers you, don't worry, at least for now. This book is written to be as independent of that stuff as possible while still recognizing its pitfalls and occasional opportunities. Even if you know little or nothing about analog electronics, you will be able to understand the logical behavior of digital circuits.

Almost every practicing digital logic designer eventually faces a time in design and debugging when the digital abstraction must be thrown out temporarily and the analog phenomena that limit or disrupt digital performance must be considered. Knowing something about the analog side of digital design will help you earn more of that other "green stuff."

If you *are* an electrical engineer, you may already have studied the equivalent of Chapter 14 in another course, but you still may enjoy reviewing that material in your copious "spare time."

- As a result of the above, there has been a big incentive to reduce supply voltage wherever possible in different CMOS components, and in many cases, in different parts of the same VLSI chip. This has given rise to so-called power-management ICs (PMICs), which supply and control the voltages used among different ICs in a digital system, including systems as small as smartphones and smartwatches.
- Usually the biggest contributor to a CMOS circuit's delay is the time to charge or discharge the electrical capacitance of the signal lines and inputs that are driven by each output. A longer signal line, or one that drives more inputs, means more capacitance and hence more delay. Physically larger CMOS devices can charge or discharge this capacitance faster, but they consume more power and more chip area. So there is a trade-off among speed, power, and chip area.

1.10 Programmable Devices

There are a wide variety of ICs that can have their logic function "programmed" into them after they are manufactured. Most of these devices use technology that also allows the function to be *re*programmed, which means that if you find a bug in your design, you may be able to fix it without physically replacing or rewiring the device. In this book, we'll generically call such chips *programmable devices*, and we'll pay a lot of attention to design methods that use them.

programmable device

Probably the earliest available programmable device for combinational logic was the *read-only memory (ROM)*. A ROM stores a two-dimensional array of bits, with 2^n rows and b columns; if n=16 and b=8, we would speak of a "64 Kbyte ROM." ROMs are typically used for storing programs and fixed data for microprocessors. However, a 64 Kbyte ROM can store the truth table for any combinational logic function with up to 16 inputs and 8 outputs, for example, one that compares two 8-bit numbers and outputs the larger one.

read-only memory (ROM)

Early ROMs were slow and expensive compared to SSI and MSI functions, so they weren't often considered for performing logic; in the example, you'd do better on both counts with an MSI-based design. However, ROMs were often used for the most complex, non-time-critical functions with up to about 20 inputs. More significantly, today's "FPGA" devices, introduced shortly, use a large collection of much smaller ROMs as their fundamental combinational-logic building blocks.

Historically, *programmable logic arrays (PLAs)* arrived next, offering a two-level array of AND and OR gates with user-programmable connections. Using this structure, a designer could accommodate any logic function up to a certain level of complexity using the well-known theory of two-level combinational logic synthesis that we'll present in Chapter 3.

programmable logic array (PLA)

PLA structure was enhanced and PLA costs were reduced with the introduction of *programmable array logic (PAL) devices*. Today, such devices

programmable array logic (PAL) device

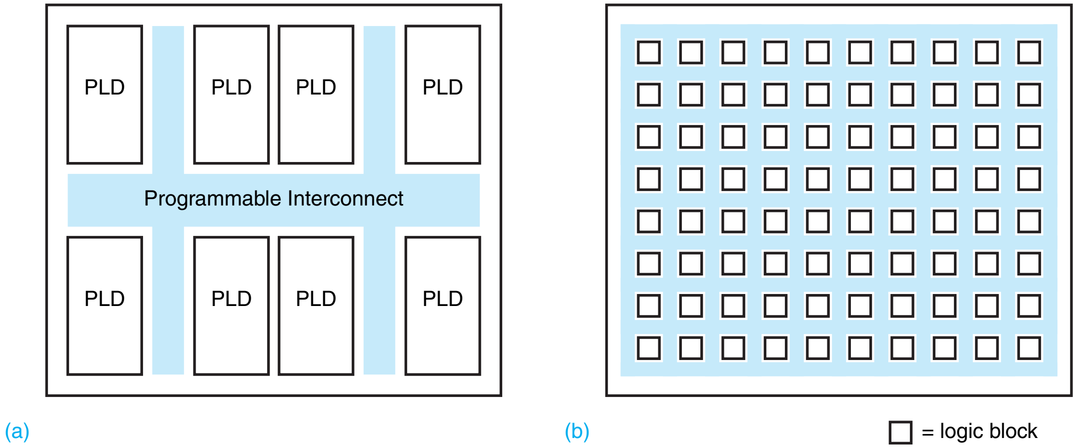


Figure 1-17 Large programmable-logic-device scaling approaches: (a) CPLD; (b) FPGA.

programmable logic device (PLD)

are generically called *programmable logic devices (PLDs)* and are the "SSI and MSI" of the programmable logic industry. Since their functionality and density are very low compared to newer programmable devices, they are seldom at the heart of a new design, but they are sometimes convenient to use as "glue" between larger chips whose interfaces are mismatched. We'll introduce basic PLD architecture and technology in Sections 6.2 and 10.6.

complex PLD (CPLD)

The ever-increasing capacity of integrated circuits created an opportunity for IC manufacturers to create larger PLDs for larger digital-design applications. However, for technical reasons, the basic two-level AND-OR structure of PLDs could not be scaled to larger sizes. Instead, IC manufacturers devised *complex PLD (CPLD)* architectures to achieve the required scale. A typical CPLD is just a collection of multiple PLDs and an interconnection structure, all on the same chip. The on-chip interconnection structure is programmed at the same time as the individual PLDs, providing a rich variety of design possibilities. CPLDs can be scaled to larger sizes by increasing the number of individual PLDs and the richness of the interconnection structure on the CPLD chip.

field-programmable gate array (FPGA)

configurable logic block (CLB)

At about the same time that CPLDs were invented, other IC manufacturers took a different approach to scaling the size of programmable logic chips. Compared to a CPLD, a *field-programmable gate array (FPGA)* contains a much larger number of smaller *configurable logic blocks (CLBs)* and provides a large, distributed and programmable interconnection structure that dominates the entire chip. Figure 1-17 shows the difference between the two chip-design approaches. We'll describe the basic architectural features of an example FPGA family's logic blocks in Sections 6.1.3 and 10.7, and its overall architecture including programmable interconnect and input/output blocks in Section 15.5.1.

Proponents of CPLDs and FPGAs used to get into "religious" arguments over which programmable-device architectural approach was better. At one time, leading manufacturers of both device types acknowledged that there is a place for both approaches, and they developed new versions of both types for different design requirements (including different density, speed, power, and cost levels). Both types of device are still actively marketed for new designs. However, it has been over five years since any manufacturer introduced a new CPLD architecture, and even the most prominent CPLD manufacturer (Altera, acquired by Intel in 2015) has moved to an FPGA architecture in their latest devices. With the industry's two decades of design and application experience with both device types, FPGAs have proven to be more able to take advantage of ever increasing IC density and performance.

Programmable devices support a style of design in which products can be moved from design concept to prototype and production in a very short time. Also important in achieving short "time to market" for these products is the use of HDLs in their design. Verilog and VHDL and their accompanying EDA tools enable a design to be compiled, synthesized, and downloaded into a CPLD or FPGA in minutes. These highly structured, hierarchical languages are essential for enabling designers to utilize the millions of gates provided in the largest programmable devices.

Programmable devices do have some downsides. Because of their programmability, they are almost always larger and slower than a customized chip would be for the same application, and they usually have a higher cost per chip. That brings us to the subject of "ASICs."

1.11 Application-Specific ICs

Chips that are designed for a particular, limited product or application are called *application-specific ICs (ASICs)*. ASICs generally reduce the total component and manufacturing cost of a product by reducing chip count, physical size, and power consumption, and they often provide higher performance.

application-specific IC (ASIC)

While the price per chip of an ASIC for a given application is typically much lower than that of a programmable device that performs the same function, the up-front costs are much higher. Compared to a programmable device, the basic engineering cost for an ASIC's logic design may be about the same, but additional *nonrecurring engineering (NRE) cost* for the ASIC can be $100,000 to $1,000,000 or more. NRE charges are paid to the IC manufacturer and others who are responsible for designing the internal structure of the chip, creating tooling such as the metal masks for manufacturing the chips, developing tests for the manufactured chips, and actually making the first few sample chips. So, an ASIC design normally makes sense only if NRE cost is offset by the per-unit savings over the expected sales volume of the product, or if there's no way to achieve the required performance in a programmable device.

nonrecurring engineering (NRE) cost

The NRE cost to design a *custom VLSI* chip—a chip whose functions, internal architecture, and detailed transistor-level design is tailored for a specific

custom VLSI

customer—is very high, $10,000,000 or more. Thus, full custom VLSI design is done only for chips that have general commercial application (e.g., microprocessors) or that will enjoy very high sales volume in a specific application (e.g., a digital watch chip, a network interface, or a sensor controller for smartphones).

standard cells

standard-cell design

To reduce NRE charges, IC manufacturers have developed libraries of *standard cells* including commonly used small building blocks like decoders, registers, and counters, and larger blocks like memories, microprocessors, and network interfaces. In a *standard-cell design*, the logic designer interconnects functions in much the same way as in a multichip board-level design. Custom cells are created (at added cost, of course) only if absolutely necessary. All of the cells are then laid out on the chip, optimizing the layout to reduce timing delays and minimize the size of the chip. Minimizing the chip size reduces the per-unit cost of the chip, since it increases the number of chips that can be fabricated on a single wafer. The NRE cost for a standard-cell design is typically on the order of $300,000 or more.

The basic digital design methods that you'll study throughout this book apply very well to the functional design of ASICs. However, there are additional opportunities, constraints, and steps in ASIC design which usually depend on the particular ASIC vendor and design environment.

1.12 Printed-Circuit Boards

printed-circuit board (PCB)

printed-wiring board (PWB)

An IC is normally mounted on a *printed-circuit board (PCB)* that connects it to other ICs in a system. The multilayer PCBs used in typical digital systems have copper wiring etched on multiple, thin layers of fiberglass that are laminated into a single board usually about 1/16 inch thick.

PCB traces

mil

fine-line

Individual wire connections, or *PCB traces*, are usually quite narrow, 5 to 25 mils in typical PCBs. (A *mil* is one-thousandth of an inch.) In *fine-line* PCB technology, the traces and spaces are extremely narrow, under 2 mils wide in high-density interconnect (HDI) PCBs. Thus, hundreds of connections may be routed in a one-inch-wide band on a single layer of the PCB. If higher connection density is needed, then more layers are used.

surface-mount technology (SMT)

Most of the components in modern PCBs use *surface-mount technology (SMT)*. Instead of having the long pins of DIP packages that poke through the board and are soldered to the underside, some SMT IC packages have pins that are bent to make flat contact with the top surface of the PCB. Instead of pins, others have "bumps" on the underside of the package, which in many cases occupy the entire under-surface of the package, not just the edges. Before such components are mounted on the PCB, a special "solder paste" is applied to contact pads on the PCB using a stencil with a hole pattern that matches the contact pads to be soldered. Then the SMT components are placed on the pads, usually by machine, where they are held in place by the solder paste (or in some cases, by glue). Finally, the entire assembly is passed through an oven to melt the solder paste, which then solidifies when cooled.

Surface-mount technology, coupled with fine-line PCB technology, allows extremely dense packing of integrated circuits and other components on a PCB. This dense packing does more than save space. For very high-speed circuits, dense packing helps to minimize certain adverse analog phenomena, such as transmission-line effects and speed-of-light limitations.

multichip module (MCM)

To satisfy the most stringent requirements for speed and density, *multichip modules (MCMs)* have been developed. In this technology, IC dice are not mounted in individual plastic or ceramic packages. Instead, the IC dice for a high-speed system (say, a processor, memory, and system interface) are bonded directly to a substrate that contains the required interconnections on multiple layers. The MCM is hermetically sealed and has its own external pins for power, ground, and just those signals that are required by the system that contains it.

1.13 Digital-Design Levels

Digital design can be carried out at several different levels of representation and abstraction. Although you may learn and practice design at a particular level, from time to time you'll need to go up or down a level or two to get the job done. Also, the industry itself and most designers have been steadily moving to higher levels of abstraction as circuit density and functionality have increased.

Moore's Law

The lowest level of digital design is device physics and IC manufacturing processes. This is the level that is primarily responsible for the breathtaking advances in IC speed and density that have occurred over the past decades. The effects of these advances are summarized in *Moore's Law*, first stated by Intel founder Gordon Moore in 1965: that the number of transistors per square inch in the newest ICs will double every year. In recent years, the rate of advance has slowed down to doubling about every 24 months, but it is important to note that with each doubling of density has also come a significant increase in speed.

This book does not reach down to the level of device physics and IC processes, but you need to recognize the importance of that level. Being aware of likely technology advances and other changes is important in system and product planning. For example, reductions in chip geometries in the past decades have forced a move to lower logic-power-supply voltages, causing major changes in the way designers plan and specify modular systems and upgrades.

In this book, we will discuss some digital-design topics at the transistor level (mostly in Chapter 14) and go all the way up to the level of logic design using HDLs. We stop short of the next level, which includes computer design and overall system design. The "center" of our discussion is at the level of functional building blocks.

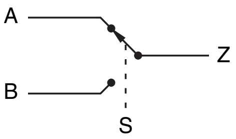


Figure 1-18
Switch model for multiplexer function.

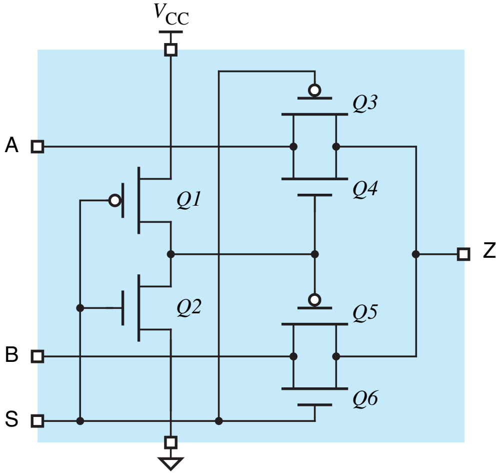


Figure 1-19
Multiplexer design using CMOS transmission gates.

To get a preview of the levels of design that we'll cover, consider a simple design example. Suppose you are to build a "multiplexer" with two data input bits, A and B, a control input bit S, and an output bit Z. Depending on the value of S, 0 or 1, the circuit is to transfer the value of either A or B to the output Z. This idea is illustrated in the "switch model" of Figure 1-18. Let us consider the design of this function at several different levels.

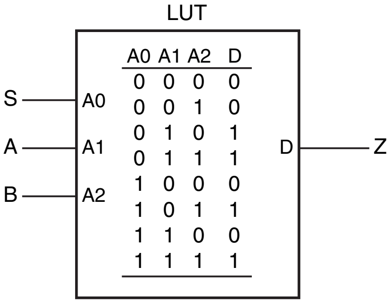
Although logic design is usually carried out at a higher level, for some functions it is advantageous to optimize them by designing at the transistor level. The multiplexer is such a function. Figure 1-19 shows how the multiplexer can be designed in CMOS technology using a pair of the transmission gates that we introduced in Section 1.9, along with an inverter for the control input. Using this approach, the multiplexer can be built with just six transistors. Any of the other approaches that we describe requires at least 14 transistors.

In the traditional study of logic design, we would use a truth table to describe the multiplexer's logic function. Since the multiplexer has three inputs, it has 2^3 or 8 possible input combinations, as shown in the truth table in Table 1-3. To create this function inside an FPGA, we could load the truth table into one of the FPGA's ROM lookup tables (LUTs), and connect A, B, and S to the ROM's address inputs and Z to its data output as shown in Figure 1-20.

Starting with the truth table, and using the traditional logic design methods described in Section 3.3.3, we could use switching algebra and well-understood

S	A	B	Z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Table 1-3
Truth table for the multiplexer function.



A0	A1	A2	D
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Figure 1-20
Multiplexer design using an FPGA lookup table (LUT).

minimization algorithms to derive an "optimal" two-level AND-OR equation for the multiplexer function:

$$Z = (S' \cdot A) + (S \cdot B)$$

This equation is read "Z equals not S and A, or S and B." Going one step further, we can convert the equation into a set of logic gates suitable for performing the specified logic function in an ASIC, as shown in Figure 1-21. This circuit requires 20 transistors if we use standard CMOS technology for the four gates shown, and 14 transistors if we replace the AND and OR gates with NAND gates, which is possible as we'll show in Section 3.2.

The gate-level structure of the multiplexer circuit in Figure 1-21 can also be specified using an HDL model rather than a logic diagram. A Verilog model corresponding to the logic diagram is shown in Program 1-1. The first four lines

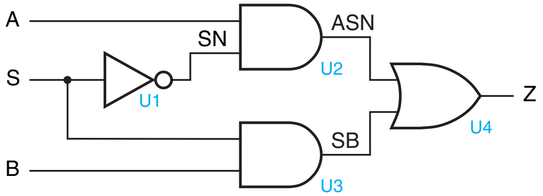


Figure 1-21
Gate-level logic diagram for multiplexer function.

Program 1-1 Structural Verilog model for a 2-input multiplexer circuit.

```
module Ch1mux_s( A, B, S, Z);     // 2-input multiplexer
  input A, B, S;
  output Z;
  wire SN, ASN, SB;

  not U1 (SN, S);
  and U2 (ASN, A, SN);
  and U3 (SB, B, S);
  or  U4 (Z, ASN, SB);
endmodule
```

define the circuit's inputs, outputs, and internal signals. The next four statements create the four gates, whose identifiers are shown in color in Figure 1-21, along with their output and input connections.

The Verilog "structural" model in Program 1-1 doesn't actually raise the level of the multiplexer design—it merely uses text to specify the same gate-level structure that the logic diagram does. We must use a different approach to capture the primary value of HDLs, which is to take design to a higher level by specifying logic functions behaviorally. Then, we can let a synthesis tool work out the details of implementing the specified behaviors in lookup tables, gate-level structures, or whatever other implementation technology is targeted.

Thus, Program 1-2 is a behavioral model that uses other Verilog language features to obtain the same multiplexer function. After defining the circuit's inputs and outputs, the model has just one high-level statement. The inputs A, B, and S are to be monitored continuously. If any of them change, then the Z output

Program 1-2 Behavioral Verilog model for a 2-input multiplexer circuit.

```
module Ch1mux_b( A, B, S, Z);     // 2-input multiplexer
  input A, B, S;
  output reg Z;

  always @ (A, B, S) if (S==1) Z = B; else Z = A;
endmodule
```

IMPLEMENTING REALIZATIONS

… or is it realizing implementations? In this book, we'll use the verbs *implement* and *realize*, as well as the nouns *implementation* and *realization*, pretty much interchangeably. Both verbs refer to the process of converting an abstract functional description, whether it be a truth table or an HDL model or anything in between, into a real circuit that performs the described function. The nouns refer to the results of that process. Note, however, that "implementation" has a more specific technical meaning in the context of Xilinx Vivado tools as described on page 173.

is to be updated so it always equals B if S is 1, and equals A otherwise. In Program 1-2, it's obviously a lot easier to see what's going on than in the original transistor-level multiplexer circuit, the truth table, the resulting logic equations and gate-level circuit, or the corresponding structural Verilog model that we showed. This ease of description and the automated implementation provided by synthesis tools are primary reasons for using HDLs.

1.14 The Name of the Game

board-level design

Given the functional and performance requirements for a digital system, the name of the game in practical digital design is to minimize cost. For *board-level designs*—systems that are packaged on a single PCB—this usually means minimizing the number of IC packages. If too many ICs are required, they won't all fit on the PCB. "Well, just use a bigger PCB," you say. Unfortunately, PCB sizes are usually constrained by factors like preexisting standards (e.g., add-in boards for PCs), packaging constraints (e.g., it has to fit in your pocket), or edicts from above (e.g., in order to get the project approved three months ago, you foolishly told your manager that it would all fit on a 3 × 5 inch PCB, and now you've got to deliver!). In each of these cases, the cost of using a larger PCB or multiple PCBs may be unacceptable.

ASIC design

In *ASIC design*, the name of the game is a little different, but the importance of structured, functional design techniques is the same. Although it's easy to burn hours and weeks creating custom macrocells and minimizing the total gate count of an ASIC, only rarely is this advisable. The per-unit cost reduction achieved by having a 10% smaller chip is negligible except in high-volume applications. In applications with low to medium volume (the majority), two other factors are more important: design time and NRE cost.

A shorter design time allows a product to reach the market sooner, increasing revenues over the lifetime of the product. A lower NRE cost also flows right to the "bottom line" and in small companies may be the only way the project can be completed before the company runs out of money (Believe me, I've been there!). If the product is successful, it's always possible and profitable to "tweak" the design later to reduce per-unit costs. The need to minimize design time and NRE cost argues in favor of a structured, as opposed to highly optimized, approach to ASIC design, using standard building blocks provided in the ASIC manufacturer's library.

design with programmable devices

The design considerations using programmable devices are a combination of the above. For example, the choice of a particular FPGA technology and device size is usually made fairly early in the design cycle. Later, as long as the design "fits" in the selected device, there's no point in trying to optimize gate count or board area—the device has already been committed. However, if new functions or bug fixes push the design beyond the capacity of the selected device, that's when you must work very hard to modify the design to make it fit!

1.15 Going Forward

This concludes the introductory chapter. As you continue reading this book, keep in mind two things. First, the ultimate goal of digital design is to build systems that solve problems for people. While this book will give you the basic tools for design, it's still your job to keep "the big picture" in the back of your mind. Second, cost is an important factor in every design decision; and you must consider not only the cost of digital components, but also the cost of the design activity itself.

Finally, as you get deeper into the text, if you encounter something that you think you've seen before but don't remember where, please consult the index. I've tried to make it as helpful and complete as possible.

Drill Problems

1.1 Give three different definitions for the word "bit" as used in this chapter.

1.2 Find the definitions in this chapter of the following acronyms: ASIC, BD, CAD, CAE, CD, CMOS, CO, CPLD, DIP, DVD, EDA, FPGA, HDL, IC, IP, LSI, LUT, MCM, MOS, MOSFET, MSI, NMOS, NRE, PBX, PCB, PLD, PMIC, PMOS, ROM, SMT, SSI, TTL, VHDL, VLSI.

1.3 Research the definitions of the following acronyms: DDPP, JPEG, MPEG, MP3, OK, PERL, TCL. (Are OK and PERL really acronyms?)

1.4 Excluding the topics in Section 1.2, list three once-analog systems that have "gone digital" since you were born.

1.5 Draw a digital circuit consisting of a 2-input AND gate and three inverters, where an inverter is connected to each of the AND gate's inputs and its output. For each of the four possible combinations of inputs applied to the two primary inputs of this circuit, determine the value produced at the primary output. Is there a simpler circuit that gives the same input/output behavior?

1.6 The ability to search the Web for specifications and other needed information is an important skill for digital designers and most engineers. With that in mind, draw a pin diagram showing the pinouts of a quadruple 2-input NAND gate in a 14-pin DIP package.

1.7 What is the relationship between "die" and "dice"?

1.8 Today there are actually some SSI parts that are tinier than the originals. Go online and find the part number for a single 2-input CMOS NAND gate. What are the number of pins and dimensions of its largest package? Its smallest?

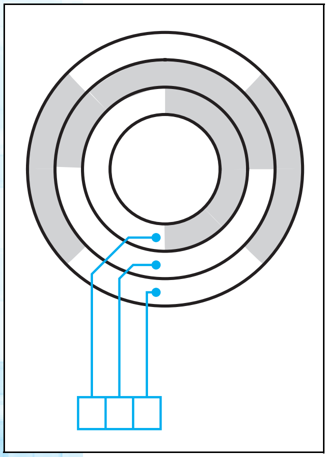
1.9 Draw a switch-level model for a CMOS NOR gate in the style of Figure 1-14, showing the same three input conditions as in that figure.

1.10 In Figure 1-19, which transistors form an inverter?

1.11 How many bits of memory are stored in the LUT in Figure 1-20?

1.12 How is an HDL different from an executable programming language like C or Java? How is it the same? (*Hint*: You won't find a comprehensive answer in this chapter. Look ahead, or do some Web research.)

chapter 2



Number Systems and Codes

Digital systems are built from circuits that process binary digits—0s and 1s—yet very few real-life problems are based on binary numbers, or any numbers at all. As a result, a digital system designer must establish some correspondence between the binary digits processed by digital circuits and real-life numbers, events, and conditions. The purpose of this chapter is to show you how familiar numeric quantities can be represented and manipulated in a digital system, and also to show you how nonnumeric data, events, and conditions can be represented.

The first nine sections will describe binary number systems and show how addition, subtraction, multiplication, and division are performed in binary number systems. Sections 2.10–2.13 will show how several other things, such as decimal numbers, text characters, mechanical positions, and arbitrary conditions, can be encoded using strings of binary digits.

Section 2.14 will introduce "*n*-cubes," which provide a way to visualize the relationship between different bit strings. The *n*-cubes are especially useful in the study of error-detecting and -correcting codes in Section 2.15. These codes are especially important for preserving the integrity of memory and storage systems, which have grown tremendously in size over the years. We will conclude the chapter in Section 2.16 with an introduction to "serial" codes that are used for transmitting and storing data one bit at a time.

2.1 Positional Number Systems

positional number system
weight

The traditional number system we learned in school and use every day in business is called a *positional number system*. In such a system, a number is represented by a string of digits, where each digit position has an associated *weight*. The value of a number is a weighted sum of the digits, for example:

$$1734 = 1\cdot 1000 + 7\cdot 100 + 3\cdot 10 + 4\cdot 1$$

Each weight is a power of 10 corresponding to the digit's position. A decimal point allows negative as well as positive powers of 10 to be used:

$$5185.68 = 5\cdot 1000 + 1\cdot 100 + 8\cdot 10 + 5\cdot 1 + 6\cdot 0.1 + 8\cdot 0.01$$

In general, a number D of the form $d_1 d_0 . d_{-1} d_{-2}$ has the following value:

$$D = d_1\cdot 10^1 + d_0\cdot 10^0 + d_{-1}\cdot 10^{-1} + d_{-2}\cdot 10^{-2}$$

base
radix

Here, 10 is called the *base* or *radix* of the number system. In a general positional number system, the radix may be any integer $r \geq 2$, and a digit in position i has weight r^i. The general form of a number in such a system is

$$d_{p-1} d_{p-2} \cdots d_1 d_0 \,.\, d_{-1} d_{-2} \cdots d_{-n}$$

radix point
expansion formula

where there are p digits to the left of the point and n digits to the right of the point, called the *radix point*. If the radix point is missing, it is assumed to be to the right of the rightmost digit. The value D of the number is given by an *expansion formula*, the sum of each digit multiplied by the corresponding power of the radix:

$$D = d_{p-1}\cdot r^{p-1} + d_{p-2}\cdot r^{p-2} + \cdots + d_1\cdot r + d_0 + d_{-1}\cdot r^{-1} + d_{-2}\cdot r^{-2} + \cdots + d_{-n}\cdot r^{-n}$$

high-order digit
most significant digit
low-order digit
least significant digit

Except for possible leading and trailing zeroes, the representation of a number in a positional number system is unique. (Obviously, 0185.6300 equals 185.63, and so on.) The leftmost digit in such a number is called the *high-order* or *most significant digit*; the rightmost is the *low-order* or *least significant digit*.

binary digit
bit
binary radix

Digital circuits have signals that are normally in one of only two states such as low or high, charged or discharged, off or on. The signals in these circuits are interpreted to represent *binary digits* (or *bits*) that have one of two values, 0 and 1. Thus, the *binary radix* is normally used to represent numbers in a digital system. The general form of a binary number is

$$b_{p-1} b_{p-2} \cdots b_1 b_0 \,.\, b_{-1} b_{-2} \cdots b_{-n}$$

and its value is

$$B = b_{p-1}\cdot 2^{p-1} + b_{p-2}\cdot 2^{p-2} + \cdots + b_1\cdot 2 + b_0 + b_{-1}\cdot 2^{-1} + b_{-2}\cdot 2^{-2} + \cdots + b_{-n}\cdot 2^{-n}$$

binary point

In a binary number, the radix point is called the *binary point*. When dealing with binary and other nondecimal numbers, we use a subscript to indicate the radix of each number, unless the radix is clear from the context. Examples of binary numbers and their decimal equivalents are given here:

$$\begin{aligned} 10011_2 &= 1\cdot16 + 0\cdot8 + 0\cdot4 + 1\cdot2 + 1\cdot1 = 19_{10} \\ 100010_2 &= 1\cdot32 + 0\cdot16 + 0\cdot8 + 0\cdot4 + 1\cdot2 + 0\cdot1 = 34_{10} \\ 101.001_2 &= 1\cdot4 + 0\cdot2 + 1\cdot1 + 0\cdot0.5 + 0\cdot0.25 + 1\cdot0.125 = 5.125_{10} \end{aligned}$$

The leftmost bit of a binary number is called the *high-order* or *most significant bit (MSB)*; the rightmost is the *low-order* or *least significant bit (LSB)*.

MSB
LSB

2.2 Binary, Octal, and Hexadecimal Numbers

Radix 10 is important because we use it in everyday business, and radix 2 is important because binary numbers can be processed directly by digital circuits. Numbers in other radices are not often processed directly but may be important for documentation or other purposes. In particular, the radices 8 and 16 provide convenient shorthand representations for multibit numbers in a digital system.

octal number system
hexadecimal (hex) number system

The *octal number system* uses radix 8, while the *hexadecimal (hex) number system* uses radix 16. You may have already encountered octal in permissions settings in file systems, and hexadecimal in Ethernet MAC addresses or in the MEID of your phone. Table 2-1 shows the binary integers from 0 to 1111 and their octal, decimal, and hexadecimal equivalents. The octal system needs 8 digits, so it uses digits 0–7 of the decimal system. The hexadecimal system needs 16 digits, so it supplements decimal digits 0–9 with the letters *A–F*.

hexadecimal digits A–F

The octal and hexadecimal number systems are useful for representing multibit numbers because their radices are powers of 2. Since a string of three bits can take on eight different combinations, it follows that each 3-bit string can be uniquely represented by one octal digit, according to the third and fourth columns of Table 2-1. Likewise, a 4-bit string can be represented by one hexadecimal digit according to the fifth and sixth columns of the table.

binary-to-octal conversion

Thus, it is very easy to convert a binary number to octal. Starting at the binary point and working left, we simply separate the bits into groups of three and replace each group with the corresponding octal digit as follows:

$$\begin{aligned} 100011001110_2 &= 100\ 011\ 001\ 110_2 = 4316_8 \\ 11101101110101001_2 &= 011\ 101\ 101\ 110\ 101\ 001_2 = 355651_8 \end{aligned}$$

binary-to-hexadecimal conversion

The procedure for binary-to-hexadecimal conversion is similar, except we use groups of four bits as follows:

$$\begin{aligned} 100011001110_2 &= 1000\ 1100\ 1110_2 = 8CE_{16} \\ 11101101110101001_2 &= 0001\ 1101\ 1011\ 1010\ 1001_2 = 1DBA9_{16} \end{aligned}$$

In these examples, we have freely added zero bits on the left to make the total number of bits a multiple of 3 or 4 as required.

Table 2-1 Binary, decimal, octal, and hexadecimal numbers.

Binary	*Decimal*	*Octal*	*3-Bit String*	*Hexadecimal*	*4-Bit String*
0	0	0	000	0	0000
1	1	1	001	1	0001
10	2	2	010	2	0010
11	3	3	011	3	0011
100	4	4	100	4	0100
101	5	5	101	5	0101
110	6	6	110	6	0110
111	7	7	111	7	0111
1000	8	10	—	8	1000
1001	9	11	—	9	1001
1010	10	12	—	A	1010
1011	11	13	—	B	1011
1100	12	14	—	C	1100
1101	13	15	—	D	1101
1110	14	16	—	E	1110
1111	15	17	—	F	1111

If a binary number contains digits to the right of the binary point, we can convert them to octal or hexadecimal by starting at the binary point and working right. Both the lefthand and righthand sides can be padded with zeroes to get multiples of three or four bits, as shown in the following example:

$$\begin{aligned} 10.1011001011_2 &= 010\,.\,101\ 100\ 101\ 100_2 = 2.5454_8 \\ &= 0010\,.\,1011\ 0010\ 1100_2 = 2.\mathrm{B2C}_{16} \end{aligned}$$

octal- or hexadecimal-to-binary conversion

Converting in the reverse direction, from octal or hexadecimal to binary, is very easy. We simply replace each octal or hexadecimal digit with the corresponding 3- or 4-bit string, as shown below:

$$\begin{aligned} 1357_8 &= 001\ 011\ 101\ 111_2 \\ 2046.17_8 &= 010\ 000\ 100\ 110\,.\,001\ 111_2 \\ \mathrm{BEAD}_{16} &= 1011\ 1110\ 1010\ 1101_2 \\ \mathrm{9F.46C}_{16} &= 1001\ 1111\,.\,0100\ 0110\ 1100_2 \end{aligned}$$

byte

Computers primarily process information in groups of 8-bit *bytes*. In the hexadecimal system, two hex digits represent an 8-bit byte, and $2n$ hex digits represent an n-byte word; each pair of digits constitutes exactly one byte. For example, the 32-bit hexadecimal number $\mathrm{5678ABCD}_{16}$ consists of four bytes with values 56_{16}, 78_{16}, AB_{16}, and CD_{16}.

ANCIENT MINIS The octal number system was quite popular 40 years ago because of certain minicomputers that had their front-panel lights and switches arranged in groups of three. In fact, Unix (the predecessor of Linux) was developed in part on such computers, which perhaps explains the use of octal in Linux filesystem permissions. However, the octal number system is not used for much else today. It is cumbersome to extract individual byte values in multibyte quantities in the octal representation; for example, what are the octal values of the four 8-bit bytes in the 32-bit number with octal representation 12345670123_8?

nibble

In this context, a 4-bit hexadecimal digit is sometimes called a *nibble*; so a 32-bit (4-byte) number has eight nibbles. Hexadecimal numbers are often used to describe a computer's memory address space. For example, a computer with 32-bit addresses might be described as having 1 gigabyte (GB) of read/write memory installed at addresses $0\text{–}3FFFFFFF_{16}$, reserved expansion space at addresses $40000000\text{–}FFEFFFFF_{16}$, and input/output ports at addresses $FFF00000\text{–}FFFFFFFF_{16}$. Many computer programming languages use the prefix "0x" to denote a hexadecimal number, for example, `0xBFC00000`.

0x prefix

2.3 Binary-Decimal Conversions

binary-to-decimal conversion

Binary numbers can be converted into decimal numbers pretty easily, using decimal (base-10) arithmetic. As we showed in examples at the end of Section 2.1, we simply substitute the value of each bit according to its position in the binary number, and solve using decimal arithmetic.

The same principle can be used for octal or hexadecimal conversion, substituting the decimal equivalents for hex digits A-F as needed, for example:

$$\begin{aligned}
1CE8_{16} &= 1\cdot16^3 + 12\cdot16^2 + 14\cdot16^1 + 8\cdot16^0 = 7400_{10}\\
F1A3_{16} &= 15\cdot16^3 + 1\cdot16^2 + 10\cdot16^1 + 3\cdot16^0 = 61859_{10}\\
436.5_8 &= 4\cdot8^2 + 3\cdot8^1 + 6\cdot8^0 + 5\cdot8^{-1} = 286.625_{10}
\end{aligned}$$

WHEN I'M 64 As you grow older, you'll find that the hexadecimal number system is useful for more than just computers. When I turned 40, I told friends that I had just turned 28_{16}. The "$_{16}$" was whispered under my breath, of course. At age 50, I was only 32_{16}.

People get all excited about decennial birthdays like 20, 30, 40, 50, …, but you should be able to convince your friends that the decimal system is of no fundamental significance. More significant life changes occur around birthdays 2, 4, 8, 16, 32, and 64, when you add a most significant bit to your age. Why do you think the Beatles sang "When I'm sixty-*four*"?

Even today, very few people expect their age to make it to eight bits.

decimal-to-binary conversion

nested expansion formula

To convert decimal numbers into binary, we take a different approach, based on rewriting the expansion formula for a binary number's decimal value in a nested fashion:

$$B = ((\cdots((b_{p-1})\cdot 2 + b_{p-2})\cdot 2 + \cdots)\cdot 2 + b_1)\cdot 2 + b_0$$

That is, we start with a sum of 0; beginning with the leftmost bit, we multiply the sum by 2 and add the next bit to the sum, repeating until all bits have been processed. For example, we can write

$$10110011_2 = (((((((((1)\cdot 2 + 0)\cdot 2 + 1)\cdot 2 + 1)\cdot 2 + 0)\cdot 2 + 0)\cdot 2 + 1)\cdot 2 + 1) = 179_{10}$$

Now, consider what happens if we divide the formula for B by 2. Since the parenthesized part of the formula is evenly divisible by 2, the quotient will be

$$Q = (\cdots((b_{p-1})\cdot 2 + b_{p-2})\cdot 2 + \cdots)\cdot 2 + b_1$$

and the remainder will be b_0. Thus, d_0 can be computed as the remainder of the long division of B by 2. Furthermore, the quotient Q has the same form as the original formula. Therefore, successive divisions by 2 yield successive digits of B from right to left, until all the digits of B have been derived. Thus, running the previous example in reverse, we get:

179 ÷ 2 = 89 remainder 1 (LSB)
÷ 2 = 44 remainder 1
÷ 2 = 22 remainder 0
÷ 2 = 11 remainder 0
÷ 2 = 5 remainder 1
÷ 2 = 2 remainder 1
÷ 2 = 1 remainder 0
÷ 2 = 0 remainder 1 (MSB)
$179_{10} = 10110011_2$

We can use the same approach to convert from decimal to other bases, for example, dividing by 8 or 16 to convert to octal or hexadecimal:

467 ÷ 8 = 58 remainder 3 (least significant digit)
÷ 8 = 7 remainder 2
÷ 8 = 0 remainder 7 (most significant digit)
$467_{10} = 723_8$

3417 ÷ 16 = 213 remainder 9 (least significant digit)
÷ 16 = 13 remainder 5
÷ 16 = 0 remainder 13 (most significant digit)
$3417_{10} = \mathrm{D59}_{16}$

Table 2-2 summarizes methods for converting among the most common radices.

Table 2-2 Conversion methods for common radices using decimal arithmetic.

Conversion	***Method***	***Example***
Binary to		
Octal	Substitution	$10111011001_2 = 10\ 111\ 011\ 001_2 = 2731_8$
Hexadecimal	Substitution	$10111011001_2 = 101\ 1101\ 1001_2 = 5D9_{16}$
Decimal	Summation	$10111011001_2 = 1 \cdot 1024 + 0 \cdot 512 + 1 \cdot 256 + 1 \cdot 128 + 1 \cdot 64$ $+ 0 \cdot 32 + 1 \cdot 16 + 1 \cdot 8 + 0 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = 1497_{10}$
Octal to		
Binary	Substitution	$1234_8 = 001\ 010\ 011\ 100_2$
Hexadecimal	Substitution	$1234_8 = 001\ 010\ 011\ 100_2 = 0010\ 1001\ 1100_2 = 29C_{16}$
Decimal	Summation	$1234_8 = 1 \cdot 512 + 2 \cdot 64 + 3 \cdot 8 + 4 \cdot 1 = 668_{10}$
Hexadecimal to		
Binary	Substitution	$C0DE_{16} = 1100\ 0000\ 1101\ 1110_2$
Octal	Substitution	$C0DE_{16} = 1100\ 0000\ 1101\ 1110_2 = 1\ 100\ 000\ 011\ 011\ 110_2 = 140336_8$
Decimal	Summation	$C0DE_{16} = 12 \cdot 4096 + 0 \cdot 256 + 13 \cdot 16 + 14 \cdot 1 = 49374_{10}$
Decimal to		
Binary	Division	$108_{10} \div 2 = 54$ remainder 0 (LSB) $\div 2 = 27$ remainder 0 $\div 2 = 13$ remainder 1 $\div 2 = 6$ remainder 1 $\div 2 = 3$ remainder 0 $\div 2 = 1$ remainder 1 $\div 2 = 0$ remainder 1 (MSB) $108_{10} = 1101100_2$
Octal	Division	$108_{10} \div 8 = 13$ remainder 4 (least significant digit) $\div 8 = 1$ remainder 5 $\div 8 = 0$ remainder 1 (most significant digit) $108_{10} = 154_8$
Hexadecimal	Division	$108_{10} \div 16 = 6$ remainder 12 (least significant digit) $\div 16 = 0$ remainder 6 (most significant digit) $108_{10} = 6C_{16}$

2.4 Addition and Subtraction of Binary Numbers

Addition and subtraction of a pair of nondecimal numbers by hand uses the same technique that your parents learned and maybe you even learned in grammar school for decimal numbers, before you got a calculator. For addition, we line up the numbers with their rightmost digits on the right, and starting at that end, we add the digits one column at a time. If a column sum has more than one digit, we propagate the extra digit or "carry" to the column to the left. Subtraction is similar, using "borrows." Compared to decimal operations, the only catch for binary operations is that the addition and subtraction tables are different.

binary addition

Table 2-3 is the addition and subtraction table for binary digits. To add two binary numbers X and Y, we add together the least significant bits with an initial carry (c_{in}) of 0, producing carry (c_{out}) and sum (s) bits according to the table. We continue processing bits from right to left, adding the carry out of each column into the next column's sum.

Two examples of decimal additions and the corresponding binary additions are shown in Figure 2-1, using a colored arrow to indicate a carry of 1. The same examples are repeated below along with two more, with the carries shown as a bit string C:

$$\begin{array}{crr} C & & 101111000 \\ X & 190 & 10111110 \\ Y & +141 & +\ \ 10001101 \\ \hline X+Y & 331 & 101001011 \end{array} \qquad \begin{array}{crr} C & & 001011000 \\ X & 173 & 10101101 \\ Y & +\ 44 & +\ 00101100 \\ \hline X+Y & 217 & 11011001 \end{array}$$

$$\begin{array}{crr} C & & 011111110 \\ X & 127 & 01111111 \\ Y & +\ 63 & +\ \ 00111111 \\ \hline X+Y & 190 & 10111110 \end{array} \qquad \begin{array}{crr} C & & 000000000 \\ X & 170 & 10101010 \\ Y & +\ 85 & +\ 01010101 \\ \hline X+Y & 255 & 11111111 \end{array}$$

Table 2-3
Binary addition and subtraction table.

c_{in} or b_{in}	x	y	c_{out}	s	b_{out}	d
0	0	0	0	0	0	0
0	0	1	0	1	1	1
0	1	0	0	1	0	1
0	1	1	1	0	0	0
1	0	0	0	1	1	1
1	0	1	1	0	1	0
1	1	0	1	0	0	0
1	1	1	1	1	1	1

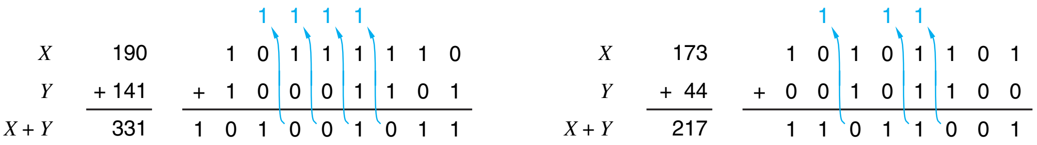


Figure 2-1 Examples of decimal and corresponding binary additions.

Binary subtraction is performed similarly, using borrows (b_{in} and b_{out}) instead of carries between steps, and producing a difference bit d. Two examples of decimal subtractions and the corresponding binary subtractions (*minuend* minus *subtrahend* yields *difference*) are shown in Figure 2-2. As in decimal subtraction, the binary minuend values in the columns are modified when borrows occur, as shown by the colored arrows and bits. The examples from the figure are repeated below along with two more, this time showing the borrows as a bit string B:

binary subtraction

minuend

subtrahend

difference

B		001111100	B		011011010
X	229	11100101	X	210	11010010
Y	− 46	− 00101110	Y	−109	− 01101101
$X - Y$	183	10110111	$X - Y$	101	01100101
B		010101010	B		000000000
X	170	10101010	X	221	11011101
Y	− 85	− 01010101	Y	− 76	− 01001100
$X - Y$	85	01010101	$X - Y$	145	10010001

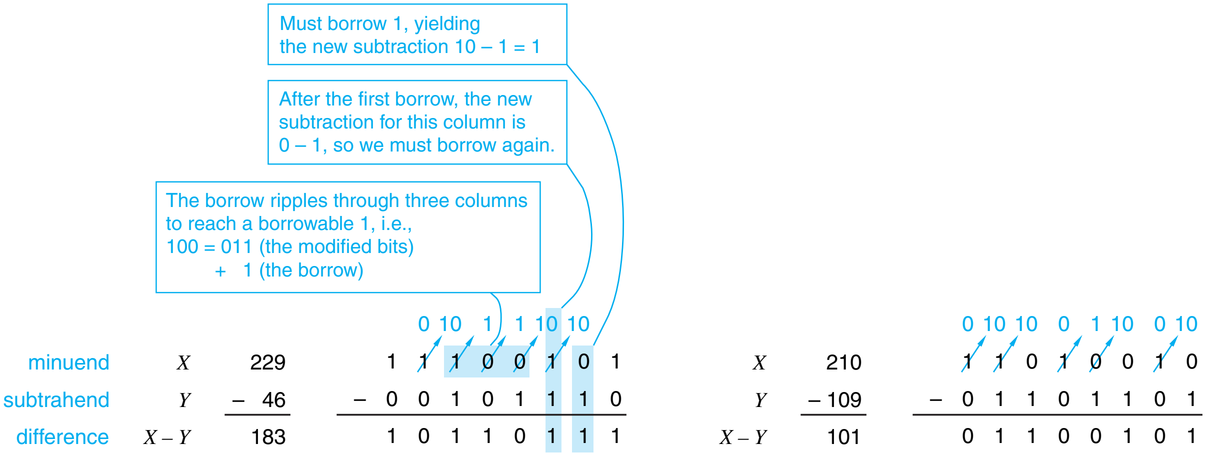

Must borrow 1, yielding the new subtraction 10 – 1 = 1
After the first borrow, the new subtraction for this column is 0 – 1, so we must borrow again.
The borrow ripples through three columns to reach a borrowable 1, i.e., 100 = 011 (the modified bits) + 1 (the borrow)
minuend
subtrahend
difference
0 10 1 1 10 10
X 229 1 1 1 0 0 1 0 1
Y – 46 – 0 0 1 0 1 1 1 0
X – Y 183 1 0 1 1 0 1 1 1
0 10 10 0 1 10 0 10
X 210 1 1 0 1 0 0 1 0
Y – 109 – 0 1 1 0 1 1 0 1
X – Y 101 0 1 1 0 0 1 0 1

Figure 2-2 Examples of decimal and corresponding binary subtractions.

comparing numbers

A very common use of subtraction in computers is to compare two numbers. For example, if the operation $X - Y$ produces a borrow out of the most significant bit position, then X is less than Y; otherwise, X is greater than or equal to Y. The relationship between carries and borrows in adders and subtractors will be explored in Section 8.1.3.

Addition and subtraction tables can be developed for octal and hexadecimal digits, or any other desired radix. However, few computer engineers bother to memorize these tables anymore—it's easier to install a programmer's "hex calculator" app on your computer or phone.

2.5 Representation of Negative Numbers

So far, we have dealt only with positive numbers, but there are many ways to represent negative numbers. In everyday business we use the signed-magnitude system, discussed next. However, most computers use the two's-complement number system that we will introduce right after that.

2.5.1 Signed-Magnitude Representation

signed-magnitude system

In the *signed-magnitude system*, a number consists of a magnitude and a symbol indicating whether the number is positive or negative. Thus, we interpret decimal numbers +98, –57, +123.5, and –13 in the usual way, and we also assume that the sign is "+" if no sign symbol is written. There are two possible representations of zero, "+0" and "–0", but both have the same value.

sign bit

The signed-magnitude system is applied to binary numbers by using an extra bit position to represent the sign (the *sign bit*). Traditionally, the most significant bit (MSB) of a bit string is used as the sign bit (0 = plus, 1 = minus), and the lower-order bits contain the magnitude. Thus, we can write several 8-bit signed-magnitude integers and their decimal equivalents:

$$
\begin{array}{lclclcl}
01010101_2 & = & +85_{10} & \qquad & 11010101_2 & = & -85_{10} \\
01111111_2 & = & +127_{10} & & 11111111_2 & = & -127_{10} \\
00000000_2 & = & +0_{10} & & 10000000_2 & = & -0_{10}
\end{array}
$$

The signed-magnitude system has an equal number of positive and negative integers. An n-bit signed-magnitude integer lies within the range $-(2^{n-1} - 1)$ through $+(2^{n-1} - 1)$, and there are two possible representations of zero.

signed-magnitude adder

Now suppose we wanted to build a digital logic circuit that adds signed-magnitude numbers. The circuit must examine the signs of the addends to determine what to do with the magnitudes. If the signs are the same, it must add the magnitudes and give the result the same sign. If the signs are different, it must compare the magnitudes, subtract the smaller from the larger, and give the result the sign of the larger. All of these "ifs," "adds," "subtracts," and "compares" translate into a lot of logic-circuit complexity. Adders for complement number systems are much simpler, as we'll show next.

THE SIMPLEST SUBTRACTOR

Perhaps the one redeeming feature of a signed-magnitude system is that, once we know how to build a signed-magnitude adder, a signed-magnitude subtractor is almost trivial to build—it need only change the sign of the subtrahend and pass it along with the minuend to the signed-magnitude adder.

2.5.2 Complement Number Systems

While the signed-magnitude system negates a number by changing its sign, a *complement number system* negates a number by taking its complement as defined by the system. Taking the complement is more difficult than changing the sign, but two numbers in a complement number system can be added or subtracted directly without the sign and magnitude checks that have to be done in the signed-magnitude system. We'll describe two such systems for binary numbers, called the "two's complement" and the "ones' complement."

complement number system

In two's and ones' complement number systems, we normally deal with a fixed number of bits, say n. However, we can increase the number of bits by "sign extension" as shown in Exercise 2.35, and decrease the number by truncating high-order bits as shown in Exercise 2.36. We assume the numbers have the following form:

$$B = b_{n-1}b_{n-2}\cdots b_1 b_0 .$$

The binary point is on the right, and so the number is an integer. In either system, if an operation produces a result that requires more than n bits, we throw away the extra high-order bit(s). If a number B is complemented twice, the result is B.

2.5.3 Two's-Complement Representation

In a *two's-complement system*, the complement of an n-bit number B is obtained by subtracting it from 2^n. If B is between 1 and $2^n - 1$, this subtraction produces another number between 1 and $2^n - 1$. If B is 0, the result of the subtraction is 2^n, which has the form $100 \cdots 00$, where there are a total of $n + 1$ bits. We throw away the extra high-order bit and get a result of $00 \cdots 00$ (n 0s). Thus, there is only one representation of zero in a two's-complement system.

two's-complement system

It seems from the definition that a subtraction operation is needed to calculate the two's complement of B. However, this subtraction can be avoided by rewriting 2^n as $(2^n - 1) + 1$ and $2^n - B$ as $((2^n - 1) - B) + 1$. The number $2^n - 1$ has the form $11 \cdots 11$, where there are n 1's. For example, for $n = 8$, 100000000_2

computing the two's complement

FUNNY PUNCTUATION

The punctuation for two's complement and ones' complement really *is* supposed to be different, for rather arcane reasons. See the first two citations in the references at the end of this chapter.

equals $11111111_2 + 1$. If we define the complement of a bit b to be the opposite value of the bit, then $(2^n - 1) - B$ is obtained by simply complementing the bits of B. Therefore, the two's complement of a number B is obtained by complementing the individual bits of B and adding 1. For example, again for $n = 8$, the 2's complement of 01110100 is 10001011 + 1, or 10001100.

weight of MSB

The MSB of a number in the two's-complement system serves as the sign bit; a number is negative if and only if its MSB is 1. The decimal equivalent for a two's-complement binary number is calculated in the same way as for an unsigned number, except that the weight of the MSB is -2^{n-1} instead of $+2^{n-1}$. The range of representable numbers is $-(2^{n-1})$ through $+(2^{n-1} - 1)$. Some 8-bit examples are shown below:

$$\begin{array}{rrl} 17_{10} = & 00010001_2 & \\ & \downarrow & \text{complement bits} \\ & 11101110 & \\ & +1 & \\ \hline & 11101111_2 & = -17_{10} \end{array} \qquad \begin{array}{rrl} -99_{10} = & 10011101_2 & \\ & \downarrow & \text{complement bits} \\ & 01100010 & \\ & +1 & \\ \hline & 01100011_2 & = 99_{10} \end{array}$$

$$\begin{array}{rrl} 119_{10} = & 01110111_2 & \\ & \downarrow & \text{complement bits} \\ & 10001000_2 & \\ & +1 & \\ \hline & 10001001_2 & = -119_{10} \end{array} \qquad \begin{array}{rrl} -127_{10} = & 10000001_2 & \\ & \downarrow & \text{complement bits} \\ & 01111110_2 & \\ & +1 & \\ \hline & 01111111_2 & = 127_{10} \end{array}$$

$$\begin{array}{rrl} 0_{10} = & 00000000_2 & \\ & \downarrow & \text{complement bits} \\ & 11111111 & \\ & +1 & \\ \hline & 1\ 00000000_2 & = 0_{10} \end{array} \qquad \begin{array}{rrl} -128_{10} = & 10000000_2 & \\ & \downarrow & \text{complement bits} \\ & 01111111 & \\ & +1 & \\ \hline & 10000000_2 & = -128_{10} \end{array}$$

A carry out of the MSB position occurs in one case, in the bottom left-hand example above. As in all two's-complement operations, this bit is ignored and only the low-order n bits of the result are used.

extra negative number

In the two's-complement number system, zero is considered positive because its sign bit is 0. Since the two's-complement system has only one representation of zero, we end up with one extra negative number, -2^{n-1}, that does not have a positive counterpart.

sign extension

We can convert an n-bit two's-complement number X into an m-bit one, but some care is needed. If $m > n$, we must append $m - n$ copies of X's sign bit to the left of X (see Exercise 2.35). That is, we pad a positive number with 0s and a negative one with 1s; this is called *sign extension*. If $m < n$, we discard X's $n - m$ leftmost bits; however, the result is valid only if all of the discarded bits are the same as the sign bit of the result (see Exercise 2.36).

Most computers and other digital systems use the two's-complement system to represent negative numbers. However, for completeness, we'll also describe two other representations that have some special uses.

*2.5.4 Ones'-Complement Representation

In a *ones'-complement system*, the complement of an n-bit number B is obtained by subtracting it from $2^n - 1$. This can be accomplished by complementing the individual digits of B, *without* adding 1 as in the two's-complement system. As in two's complement, the most significant bit is the sign, 0 if positive and 1 if negative. Thus, there are two representations of zero: positive zero $(00\cdots00)$ and negative zero $(11\cdots11)$. Positive-number representations are the same for both ones' and two's complements. However, negative-number representations differ by 1. A weight of $-(2^{n-1} - 1)$, rather than -2^{n-1}, is given to the most significant bit when computing the decimal equivalent of a ones'-complement number. The range of representable numbers is $-(2^{n-1} - 1)$ through $+(2^{n-1} - 1)$. Some 8-bit numbers and their ones' complements are shown below:

ones'-complement system

$$\begin{array}{rcll}
17_{10} & = & 00010001_2 & \\
 & & \downarrow & \\
 & & 11101110_2 & = -17_{10}
\end{array}
\qquad
\begin{array}{rcll}
-99_{10} & = & 10011100_2 & \\
 & & \downarrow & \\
 & & 01100011_2 & = 99_{10}
\end{array}$$

$$\begin{array}{rcll}
119_{10} & = & 01110111_2 & \\
 & & \downarrow & \\
 & & 10001000_2 & = -119_{10}
\end{array}
\qquad
\begin{array}{rcll}
-127_{10} & = & 10000000_2 & \\
 & & \downarrow & \\
 & & 01111111_2 & = 127_{10}
\end{array}$$

$$\begin{array}{rcll}
0_{10} & = & 00000000_2 \text{ (positive zero)} & \\
 & & \downarrow & \\
 & & 11111111_2 & = 0_{10} \text{ (negative zero)}
\end{array}$$

The main advantages of the ones'-complement system are its symmetry and the ease of complementation, which led to its use in a few early computers. However, the adder design for ones'-complement numbers is somewhat trickier than for two's-complement (see Exercise 10.47). Also, zero-detecting circuits in a ones'-complement system either must check for both representations of zero, or must always convert $11\cdots11$ to $00\cdots00$. Still, ones'-complement addition is everywhere since it's used in the header checksum of Internet packets.

*2.5.5 Excess Representations

Yes, the number of different systems for representing negative numbers may seem excessive, but there's just one more for us to cover. In *excess-B representation*, an m-bit string whose unsigned integer value is M $(0 \le M < 2^m)$ represents the signed integer $M - B$, where B is called the *bias* of the number system.

excess-B representation

bias

* Throughout this book, *optional sections* are marked with an asterisk.

excess-2^{m-1} system

For example, an *excess-2^{m-1} system* represents any number X in the range -2^{m-1} through $+2^{m-1} - 1$ by the m-bit binary representation of $X + 2^{m-1}$ (which is always nonnegative and less than 2^m). The range of this representation is exactly the same as that of m-bit two's-complement numbers. In fact, the representations of any number in the two systems are identical except for the sign bits, which are always opposite. (Note this is true only when the bias is 2^{m-1}.)

The most common use of excess representation is in exponents in floating-point number systems (see References).

2.6 Two's-Complement Addition and Subtraction

2.6.1 Addition Rules

Table 2-4, a table of decimal numbers and their equivalents in different number systems, reveals why the two's complement is preferred for arithmetic operations. If we start with the smallest (most negative) number 1000_2 (-8_{10}) and count up, we see that each successive two's-complement number all the way to 0111_2 ($+7_{10}$) can be obtained by adding 1 to the previous one, but ignoring any

Table 2-4 Decimal and 4-bit numbers.

Decimal	*Two's Complement*	*Ones' Complement*	*Signed Magnitude*	*Excess 2^{m-1}*
−8	1000	—	—	0000
−7	1001	1000	1111	0001
−6	1010	1001	1110	0010
−5	1011	1010	1101	0011
−4	1100	1011	1100	0100
−3	1101	1100	1011	0101
−2	1110	1101	1010	0110
−1	1111	1110	1001	0111
0	0000	1111 or 0000	1000 or 0000	1000
1	0001	0001	0001	1001
2	0010	0010	0010	1010
3	0011	0011	0011	1011
4	0100	0100	0100	1100
5	0101	0101	0101	1101
6	0110	0110	0110	1110
7	0111	0111	0111	1111

carries beyond the fourth bit position. The same thing cannot be said of signed-magnitude and ones'-complement numbers.

two's-complement addition

Because ordinary addition is just an extension of counting, two's-complement numbers can thus be added by ordinary binary addition, ignoring any carries beyond the MSB. The result will always be the correct sum as long as the range of the number system is not exceeded. Some examples of decimal addition and the corresponding 4-bit two's-complement additions confirm this:

$$\begin{array}{rr}
+3 & 0011 \\
+\ +4 & +\ 0100 \\
\hline
+7 & 0111
\end{array}
\qquad\qquad
\begin{array}{rr}
-2 & 1110 \\
+\ -6 & +\ 1010 \\
\hline
-8 & 1\,1000
\end{array}$$

$$\begin{array}{rr}
+6 & 0110 \\
+\ -3 & +\ 1101 \\
\hline
+3 & 1\,0011
\end{array}
\qquad\qquad
\begin{array}{rr}
+4 & 0100 \\
+\ -7 & +\ 1001 \\
\hline
-3 & 1101
\end{array}$$

2.6.2 A Graphical View

Another way to view the two's-complement system uses the 4-bit "counter wheel" shown in Figure 2-3. Here we have shown the numbers in a circular or "modular" representation. The operation of this counter wheel very closely mimics that of a real 4-bit up/down counter circuit, which we'll encounter in Section 11.1.5. Starting with the arrow pointing to any number, we can add $+n$ to that number by counting up n times, that is, by moving the arrow n positions clockwise. It is also evident that we can subtract n from a number by counting down n times, that is, by moving the arrow n positions counterclockwise. Of course, these operations give correct results only if n is small enough that we don't cross the discontinuity between −8 and +7.

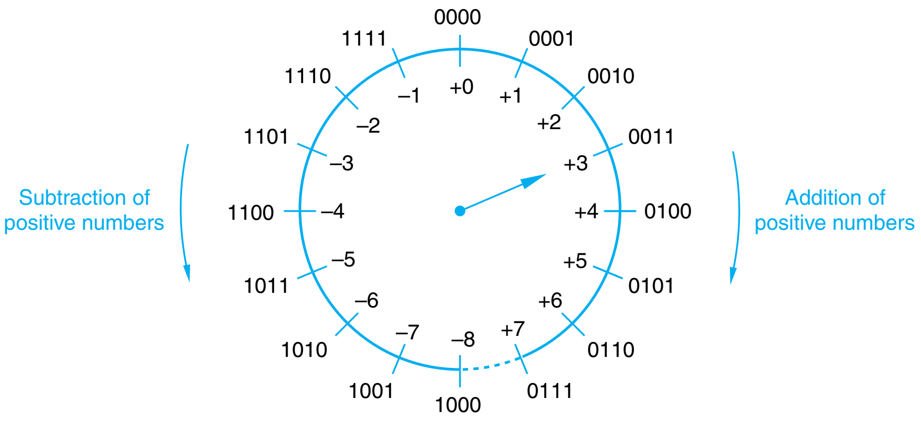


Figure 2-3 A counter wheel for adding and subtracting 4-bit two's-complement numbers.

What is most interesting is that we can also subtract n (or add $-n$) by moving the arrow $16 - n$ positions clockwise. Notice the quantity $16 - n$ is what we defined to be the 4-bit two's complement of n, that is, the two's-complement representation of $-n$. This graphically supports our earlier claim that a negative number in two's-complement representation may be added to another number simply by adding the 4-bit representations using ordinary binary addition. In Figure 2-3, adding a number is equivalent to moving the arrow a corresponding number of positions clockwise.

2.6.3 Overflow

overflow

If an addition operation produces a result that exceeds the range of the number system, *overflow* is said to occur. In the counter wheel of Figure 2-3, overflow occurs during addition of positive numbers when we count past +7. Addition of two numbers with different signs can never produce overflow, but addition of two numbers of like sign can, as shown by the following examples:

$$\begin{array}{rrl} -3 & 1101 & \\ + \;\; -6 & + \;\; 1010 & \\ \hline -9 & 10111 & = +7 \end{array} \qquad \begin{array}{rrl} +5 & 0101 & \\ + \;\; +6 & + \;\; 0110 & \\ \hline +11 & 1011 & = -5 \end{array}$$

$$\begin{array}{rrl} -8 & 1000 & \\ + \;\; -8 & + \;\; 1000 & \\ \hline -16 & 10000 & = +0 \end{array} \qquad \begin{array}{rrl} +7 & 0111 & \\ + \;\; +7 & + \;\; 0111 & \\ \hline +14 & 1110 & = -2 \end{array}$$

overflow rules

Fortunately, there is a simple rule for detecting overflow in addition: An addition overflows if the addends' signs are the same but the sum's sign is different from that of the addends. The overflow rule is sometimes stated in terms of carries generated during the addition operation: An addition overflows if the carry bits c_{in} into and c_{out} out of the sign position are different. Close examination of Table 2-3 on page 42 shows that the two rules are equivalent—there are only two cases where $c_{\text{in}} \neq c_{\text{out}}$, and these are the only two cases where $x = y$ and the sum bit is different.

2.6.4 Subtraction Rules

two's-complement subtraction

Two's-complement numbers may be subtracted as if they were ordinary unsigned binary numbers, and appropriate rules for detecting overflow may be formulated. However, most subtraction circuits for two's-complement numbers do not perform subtraction directly. Rather, they negate the subtrahend by taking its two's complement, and then add it to the minuend using the normal rules for addition.

Negating the subtrahend and adding the minuend can be accomplished with only one addition operation as follows: Perform a bit-by-bit complement of the subtrahend and add the complemented subtrahend to the minuend with an initial carry (c_{in}) of 1 instead of 0. Examples are given below:

$$\begin{array}{rrrl}
 & & 1 & \text{—}\; c_{\text{in}} \\
+4 & 0100 & 0100 & \\
-\;\;+3 & -\;\;0011 & +\;\;1100 & \\
\hline
+1 & & 10001 &
\end{array}
\qquad\qquad
\begin{array}{rrrl}
 & & 1 & \text{—}\; c_{\text{in}} \\
+3 & 0011 & 0011 & \\
-\;\;+4 & -\;\;0100 & +\;\;1011 & \\
\hline
-1 & & 1111 &
\end{array}$$

$$\begin{array}{rrrl}
 & & 1 & \text{—}\; c_{\text{in}} \\
+3 & 0011 & 0011 & \\
-\;\;-4 & -\;\;1100 & +\;\;0011 & \\
\hline
+7 & & 0111 &
\end{array}
\qquad\qquad
\begin{array}{rrrl}
 & & 1 & \text{—}\; c_{\text{in}} \\
-3 & 1101 & 1101 & \\
-\;\;-4 & -\;\;1100 & +\;\;0011 & \\
\hline
+1 & & 10001 &
\end{array}$$

Overflow in subtraction can be detected by examining the signs of the minuend and the *complemented* subtrahend, using the same rule as in addition. Or, using the technique in the preceding examples, the carries into and out of the sign position can be observed, and overflow can be detected irrespective of the signs of inputs and output, again using the same rule as in addition.

An attempt to negate the "extra" negative number results in overflow according to the rules above, when we add 1 in the complementation process:

$$\begin{array}{rrl}
-(-8) = -1000 = & 0111 & \\
 & +\;\;0001 & \\
\cline{2-2}
 & 1000 & = -8
\end{array}$$

However, this number can still be used in additions and subtractions, with results being valid as long as they do not exceed the number range:

$$\begin{array}{rr}
 & \\
+4 & 0100 \\
+\;\;-8 & +\;\;1000 \\
\hline
-4 & 1100
\end{array}
\qquad\qquad
\begin{array}{rrrl}
 & & 1 & \text{—}\; c_{\text{in}} \\
-3 & 1101 & 1101 & \\
-\;\;-8 & -\;\;1000 & +\;\;0111 & \\
\hline
+5 & & 10101 &
\end{array}$$

2.6.5 Two's-Complement and Unsigned Binary Numbers

Since two's-complement numbers are added and subtracted by the same basic binary addition and subtraction algorithms as unsigned numbers of the same length, a computer or other digital system can use the same adder circuit to deal with numbers of both types. However, the results must be interpreted differently, depending on whether the system is dealing with signed numbers (e.g., −8 through +7) or unsigned numbers (e.g., 0 through 15).

signed vs. unsigned numbers

We introduced a graphical representation of the 4-bit two's-complement system in Figure 2-3. We can relabel this figure as shown in Figure 2-4 to obtain

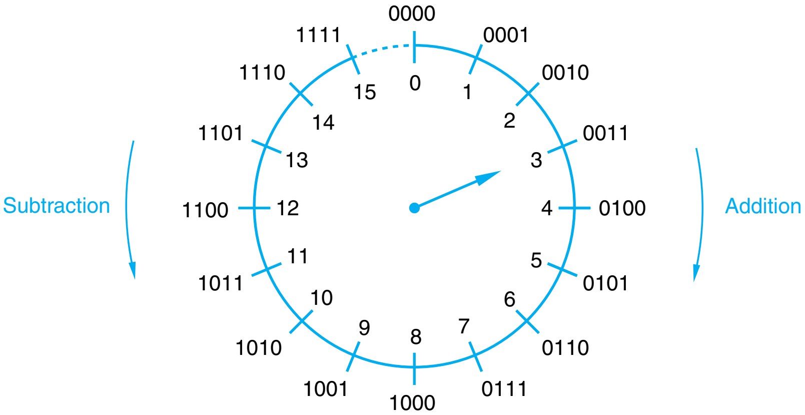


Figure 2-4 Modular counting representation of 4-bit unsigned numbers.

a representation of the 4-bit unsigned numbers. The binary combinations occupy the same positions on the wheel, and a number is still added by moving the arrow a corresponding number of positions clockwise, and subtracted by moving the arrow counterclockwise.

carry

An addition operation can be seen to exceed the range of the 4-bit unsigned-number system in Figure 2-4 if the arrow moves clockwise through the discontinuity between 0 and 15. In this case, a *carry* out of the most significant bit position is said to occur.

borrow

Likewise a subtraction operation exceeds the range of the number system if the arrow moves counterclockwise through the discontinuity. In this case, a *borrow* out of the most significant bit position is said to occur.

From Figure 2-4 it is also evident that we may subtract an unsigned number n by counting *clockwise* $16 - n$ positions. This is equivalent to *adding* the 4-bit two's-complement of n. The subtraction produces a borrow if the corresponding addition of the two's complement *does not* produce a carry.

In summary, in unsigned addition the carry or borrow in the most significant bit position indicates an out-of-range result. In signed, two's-complement addition the overflow condition defined earlier indicates an out-of-range result. The carry from the most significant bit position is irrelevant in signed addition in the sense that overflow may or may not occur independently of whether or not a carry occurs.

*2.7 Ones'-Complement Addition and Subtraction

Another look at Table 2-4 helps to explain the rule for adding ones'-complement numbers. If we start at 1000_2 (-7_{10}) and count up, we obtain each successive ones'-complement number by adding 1 to the previous one, *except* at the transition from 1111_2 (negative 0) to 0001_2 ($+1_{10}$). To maintain the proper count, we must add 2 instead of 1 whenever we count past 1111_2. This suggests a technique for adding ones'-complement numbers: Perform a standard binary addition, but add an extra 1 whenever we count past 1111_2.

Counting past 1111_2 during an addition can be detected by observing the carry out of the sign position. Thus, the rule for adding ones'-complement numbers can be stated quite simply:

ones'-complement addition

- Perform a standard binary addition; if there is a carry out of the sign position, add 1 to the result.

This rule is often called *end-around carry*. Examples of ones'-complement addition are given below; the last three include an end-around carry:

end-around carry

$$
\begin{array}{rr@{\qquad}rr@{\qquad}rr}
+3 & 0011 & +4 & 0100 & +5 & 0101 \\
+\ +4 & +\ 0100 & +\ -7 & +\ 1000 & +\ -5 & +\ 1010 \\
\hline
+7 & 0111 & -3 & 1100 & -0 & 1111
\end{array}
$$

$$
\begin{array}{rr@{\qquad}rr@{\qquad}rr}
-2 & 1101 & +6 & 0110 & -0 & 1111 \\
+\ -5 & +\ 1010 & +\ -3 & +\ 1100 & +\ -0 & +\ 1111 \\
\hline
-7 & 10111 & +3 & 10010 & -0 & 11110 \\
 & +\quad 1 & & +\quad 1 & & +\quad 1 \\
\cline{2-2}\cline{4-4}\cline{6-6}
 & 1000 & & 0011 & & 1111
\end{array}
$$

Following the two-step addition rule above, the addition of a number and its ones' complement produces negative 0. In fact, an addition operation using this rule can never produce positive 0 unless both addends are positive 0 (think about it!).

Just like two's-complement subtraction, ones'-complement subtraction is easiest to do by complementing the subtrahend and then adding. Overflow rules for ones'-complement addition and subtraction are also the same as for two's-complement.

ones'-complement subtraction

Today, ones'-complement addition is actually happening all around you, though you'd never know it. That's because the header of every Internet Protocol Version 4 (IPv4) packet contains a 16-bit ones'-complement sum of all of its other 16-bit words, that is transmitted along with header and checked for errors when the packet is received.

NUMBER SYSTEMS SUMMARY

Concluding our discussion of number systems, Table 2-5 summarizes the rules that we presented in the preceding four sections for addition, negation, and subtraction in binary number systems.

Table 2-5 Summary of addition and subtraction rules for binary numbers.

Number System	***Addition Rules***	***Negation Rules***	***Subtraction Rules***
Unsigned	Add the numbers. Result is out of range if a carry out of the MSB occurs.	Not applicable	Subtract the subtrahend from the minuend. Result is out of range if a borrow out of the MSB occurs.
Signed magnitude	(same sign) Add the magnitudes; overflow occurs if a carry out of the MSB occurs; result has the same sign. (opposite sign) Subtract the smaller magnitude from the larger; overflow is impossible; result has the sign of the larger.	Change the number's sign bit.	Change the sign bit of the subtrahend and proceed as in addition.
Two's complement	Add, ignoring any carry out of the MSB. Overflow occurs if the carries into and out of the MSB are different.	Complement all bits of the number; add 1 to the result.	Complement all bits of the subtrahend and add to the minuend with an initial carry of 1.
Ones' complement	Add; if there is a carry out of the MSB, add 1 to result. Overflow occurs if carries into and out of the MSB are different.	Complement all bits of the number.	Complement all bits of the subtrahend and proceed as in addition.

*2.8 Binary Multiplication

shift-and-add multiplication

unsigned binary multiplication

In grammar school we learned to multiply by adding a list of shifted multiplicands computed according to the digits of the multiplier. The same method can be used to obtain the product of two unsigned binary numbers. Forming the shifted multiplicands is trivial in binary multiplication, since the only possible values of the multiplier digits are 0 and 1. An example is shown below:

```
      11              1011    multiplicand
    × 13          ×   1101    multiplier
    ----          --------
      33              1011
     11              0000
    ----            1011      shifted multiplicands
     143           1011
                  --------
                  10001111    product
```

Instead of listing all the shifted multiplicands and then adding, in a digital system it is more convenient to add each shifted multiplicand as it is created to a partial product

partial product. Applying this technique to the previous example, four additions and partial products are used to multiply 4-bit numbers:

$$\begin{array}{r c r l}
11 & \qquad & 1011 & \text{multiplicand}\\
\times\ 13 & & \times\quad 1101 & \text{multiplier}\\ \cline{1-1}\cline{3-3}
 & & 0000 & \text{partial product}\\
 & & 1011 & \text{shifted multiplicand}\\ \cline{3-3}
 & & 01011 & \text{partial product}\\
 & & 0000{\downarrow} & \text{shifted multiplicand}\\ \cline{3-3}
 & & 001011 & \text{partial product}\\
 & & 1011{\downarrow}{\downarrow} & \text{shifted multiplicand}\\ \cline{3-3}
 & & 0110111 & \text{partial product}\\
 & & 1011{\downarrow}{\downarrow}{\downarrow} & \text{shifted multiplicand}\\ \cline{3-3}
 & & 10001111 & \text{product}
\end{array}$$

In general, when we multiply an n-bit number by an m-bit number, the resulting product requires at most $n + m$ bits to express. The shift-and-add algorithm requires m partial products and additions to obtain the result, but the first addition is trivial, since the first partial product is zero. Although the first partial product has only n significant bits, after each addition step the partial product gains one more significant bit, since each addition may produce a carry. At the same time, each step yields one more partial product bit, starting with the rightmost and working toward the left, that does not change. The shift-and-add algorithm can be performed by a digital circuit that includes a shift register, an adder, and control logic, as shown in Section 13.2.2.

signed multiplication

Multiplication of signed numbers can be accomplished using unsigned multiplication and the usual grammar-school rules: Perform an unsigned multiplication of the magnitudes and make the product positive if the operands had the same sign, negative if they had different signs. This is very convenient in signed-magnitude systems, since the sign and magnitude are separate.

two's-complement multiplication

In the two's-complement system, obtaining the magnitude of a negative number and negating the unsigned product are nontrivial operations. This leads us to seek a more efficient way of performing two's-complement multiplication, described next.

Conceptually, unsigned multiplication is accomplished by a sequence of unsigned additions of the shifted multiplicands; at each step, the shift of the multiplicand corresponds to the weight of the multiplier bit. The bits in a two's-complement number have the same weights as in an unsigned number, except for the MSB, which has a negative weight (see Section 2.5.3). Thus, we can perform two's-complement multiplication by a sequence of two's-complement additions of shifted multiplicands, except for the last step, in which the shifted multiplicand corresponding to the MSB of the multiplier must be negated before it is added to the partial product. Our previous example is repeated below, this time interpreting the multiplier and multiplicand as two's-complement numbers:

```
   −5            1011   multiplicand
 × −3      ×     1101   multiplier
 ────      ──────────
                00000   partial product
                11011   shifted multiplicand
           ──────────
               111011   partial product
               00000↓   shifted multiplicand
           ──────────
              1111011   partial product
              11011↓↓   shifted multiplicand
           ──────────
             11100111   partial product
             00101↓↓↓   shifted and negated multiplicand
           ──────────
             00001111   product
```

Handling the MSBs is a little tricky because we gain one significant bit at each step and we are working with signed numbers. Therefore, before adding each shifted multiplicand and *k*-bit partial product, we change them to *k* + 1 significant bits by sign extension, as shown in color above. Each resulting sum has *k* + 1 bits; any carry out of the MSB of the *k* + 1-bit sum is ignored.

*2.9 Binary Division

shift-and-subtract division

unsigned division

The simplest binary division algorithm is based on the shift-and-subtract method that we learned in grammar school. Table 2-6 gives examples of this method for unsigned decimal and binary numbers. In both cases, we mentally compare the reduced dividend with multiples of the divisor to determine which multiple of the shifted divisor to subtract. In the decimal case, we first pick 11 as the greatest multiple of 11 less than 21, and then pick 99 as the greatest multiple less than 107. The binary case is simpler, since there are only two choices—zero and the divisor itself.

division overflow

Division methods for binary numbers are somewhat complementary to binary multiplication methods. A typical division algorithm takes an $(n+m)$-bit dividend and an n-bit divisor, and produces an m-bit quotient and an n-bit remainder. A division *overflows* if the divisor is zero or the quotient would take more than m bits to express. In most computer division circuits, $n = m$.

signed division

Division of signed numbers can be accomplished using unsigned division and the usual grammar school rules: Perform an unsigned division of the magnitudes and make the quotient positive if the operands had the same sign, negative if they had different signs. The remainder should be given the same sign as the dividend. As in multiplication, there are special techniques for performing

19		10011	quotient
11)217	1011	)11011001	dividend
11		1011	shifted divisor
107		0101	reduced dividend
99		0000	shifted divisor
8		1010	reduced dividend
		0000	shifted divisor
		10100	reduced dividend
		1011	shifted divisor
		10011	reduced dividend
		1011	shifted divisor
		1000	remainder

Table 2-6 Example of long division.

division directly on two’s-complement numbers; these techniques are often implemented in computer division circuits (see References).

*2.10 Binary Codes for Decimal Numbers

Even though binary numbers are the most appropriate for the internal computations of a digital system, most people still prefer to deal with decimal numbers. As a result, the external interfaces of a digital system may read or display decimal numbers, and some digital devices actually process decimal numbers directly.

The human need to represent decimal numbers doesn’t change the basic nature of digital electronic circuits—they still process signals that take on one of only two states that we call 0 and 1. Therefore, a decimal number is represented in a digital system by a string of bits, where different combinations of bit values in the string represent different decimal numbers. For example, if we use a 4-bit string to represent a decimal number, we might assign bit combination 0000 to decimal digit 0, 0001 to 1, 0010 to 2, and so on.

code
code word

A set of n-bit strings in which different bit strings represent different numbers or other things is called a *code*. A particular combination of n 1-bit values is called a *code word*. As we’ll see in the examples of decimal codes in this section, there may or may not be an arithmetic relationship between the bit values in a code word and the thing that it represents. Furthermore, a code that uses n-bit strings need not contain 2^n valid code words.

At least four bits are needed to represent the ten decimal digits. There are billions of different ways to choose ten 4-bit code words, but some of the more common decimal codes are listed in Table 2-7.

Table 2-7
Decimal codes

Decimal digit	BCD (8421)	2421	Excess-3	Biquinary	1-out-of-10
0	0000	0000	0011	0100001	1000000000
1	0001	0001	0100	0100010	0100000000
2	0010	0010	0101	0100100	0010000000
3	0011	0011	0110	0101000	0001000000
4	0100	0100	0111	0110000	0000100000
5	0101	1011	1000	1000001	0000010000
6	0110	1100	1001	1000010	0000001000
7	0111	1101	1010	1000100	0000000100
8	1000	1110	1011	1001000	0000000010
9	1001	1111	1100	1010000	0000000001
Unused code words					
	1010	0101	0000	0000000	0000000000
	1011	0110	0001	0000001	0000000011
	1100	0111	0010	0000010	0000000101
	1101	1000	1101	0000011	0000000110
	1110	1001	1110	0000101	0000000111
	1111	1010	1111	. . .	. . .

binary-coded decimal (BCD)

packed-BCD representation

Probably the most “natural” decimal code is *binary-coded decimal (BCD)*, which encodes the digits 0 through 9 by their 4-bit unsigned binary representations, 0000 through 1001. The code words 1010 through 1111 are not used. Conversions between BCD and decimal representations are trivial, a direct substitution of four bits for each decimal digit. Some computer programs place two BCD digits in one 8-bit byte in *packed-BCD representation*; thus, one byte may represent the values from 0 to 99 as opposed to 0 to 255 for a normal unsigned 8-bit binary number. BCD numbers with any desired number of digits may be obtained by using one byte for each two digits.

BINOMIAL COEFFICIENTS

The number of different ways to choose m items from a set of n items is given by a *binomial coefficient*, denoted $\binom{n}{m}$, whose value is $\frac{n!}{m! \cdot (n-m)!}$. For a 4-bit decimal code, there are $\binom{16}{10}$ different ways to choose 10 out of 16 4-bit code words, and 10! ways to assign each different choice to the 10 digits. So there are $\frac{16!}{10! \cdot 6!} \cdot 10!$ or 29,059,430,400 different 4-bit decimal codes.

Converting a number between packed-BCD representation and binary requires a little work. An n-digit packed-BCD number D has the value

$$D \;=\; d_{n-1}\cdot 10^{n-1} + d_{n-2}\cdot 10^{n-2} + \cdots + d_1\cdot 10^1 + d_0\cdot 10^0$$

where $d_{n-1}d_{n-1}...d_1d_0$ are its BCD digits. This value can be rewritten as

$$D \;=\; ((\cdots((d_{n-1})\cdot 10 + d_{n-2})\cdot 10 + \cdots)\cdot 10 + d_1)\cdot 10 + d_0$$

Thus, given the n BCD digits, we can obtain the corresponding binary value with the following algorithm using binary arithmetic:

1. Set $i = n{-}1$ and set $D = 0$.
2. Multiply D by 10 and add d_i to D.
3. Set $i = i{-}1$ and go back to step 2 if $i \ge 0$.

The rewritten formula also leads to a way to convert a binary number into the corresponding set of BCD digits. If we divide the righthand side of the formula by 10, the remainder is d_0 and the quotient is

$$D \;/\; 10 = \;(\cdots((d_{n-1})\cdot 10 + d_{n-2})\cdot 10 + \cdots)\cdot 10 + d_1$$

which has the same form as before. Successive divisions by 10 yield successive digits of D, from right to left. Thus, the conversion can be performed as follows, using binary arithmetic:

1. Set $i = 0$.
2. Divide D by 10. Set D equal to the quotient and set d_i to the remainder.
3. Set $i = i{+}1$ and go back to step 2 if $i \le n{-}1$.

If the number of BCD digits needed to represent D is unknown at the outset, we can just start the conversion algorithm above and keep going until D is 0.

As with binary numbers, there are many possible representations of negative BCD numbers. Signed BCD numbers have one extra digit position for the sign. Both the signed-magnitude and 10's-complement (analogous to two's-complement) representations are used in BCD arithmetic. In signed-magnitude BCD, the encoding of the sign bit string is arbitrary; in 10's-complement, 0000 indicates plus and 1001 indicates minus.

BCD addition

Addition of BCD digits is similar to adding 4-bit unsigned binary numbers, except that a correction must be made if a result exceeds 1001. The result is corrected by adding 6; examples are shown below:

$$\begin{array}{rrl}
5 & 0101 & \\
+\;\;9 & +\;1001 & \\ \hline
14 & 1110 & \\
 & +\;0110 & \text{— correction} \\ \cline{2-2}
10+4 & 1\;0100 &
\end{array}
\qquad\qquad
\begin{array}{rr}
4 & 0100 \\
+\;\;5 & +\;0101 \\ \hline
9 & 1001
\end{array}$$

$$\begin{array}{rrl} 8 & 1000 & \\ +\ 8 & +\ 1000 & \\ \hline 16 & 1\ 0000 & \\ & +\ 0110 & \text{— correction} \\ \cline{2-2} 10+6 & 1\ 0110 & \end{array} \qquad \begin{array}{rrl} 9 & 1001 & \\ +\ 9 & +\ 1001 & \\ \hline 18 & 1\ 0010 & \\ & +\ 0110 & \text{— correction} \\ \cline{2-2} 10+8 & 1\ 1000 & \end{array}$$

Notice the addition of two BCD digits produces a carry into the next digit position if either the initial binary addition or the correction-factor addition produces a carry. Many computers perform packed-BCD arithmetic using special instructions that handle the carry correction automatically.

weighted code

8421 code

2421 code

self-complementing code

Binary-coded decimal is a *weighted code* because each decimal digit can be obtained from its code word by assigning a fixed weight to each code-word bit. The weights for the BCD bits are 8, 4, 2, and 1, and for this reason the code is sometimes called the *8421 code*. Another set of weights results in the *2421 code* shown in Table 2-7. This code has the advantage that it is *self-complementing*, that is, the code word for the 9s' complement of any digit may be obtained by complementing the individual bits of the digit's code word.

excess-3 code

Another self-complementing code shown in Table 2-7 is the *excess-3 code*. Although this code is not weighted, it has an arithmetic relationship with the BCD code—the code word for each decimal digit is the corresponding BCD code word plus 0011_2.

biquinary code

Decimal codes can have more than four bits; for example, the *biquinary code* in Table 2-7 uses seven. The first two bits in a code word indicate whether the number is in the range 0–4 or 5–9, and the last five bits indicate which of the five numbers in the selected range is represented. This code is used in an abacus.

One potential advantage of using more than the minimum number of bits in a code is an error-detecting property. In the biquinary code, if any one bit in a code word is accidentally changed to the opposite value, the resulting code word does not represent a decimal digit and can therefore be flagged as an error. Out of 128 possible 7-bit code words, only 10 are valid and recognized as decimal digits; the rest can be flagged as errors if they appear.

1-out-of-10 code

A *1-out-of-10 code*, like the one shown in the last column of Table 2-7, is the sparsest encoding for decimal digits, using 10 out of 1024 possible 10-bit code words.

2.11 Gray Code

In electromechanical applications of digital systems—such as machine tools, automotive braking systems, and copiers—it is sometimes necessary for an input sensor to produce a digital value that indicates a mechanical position. For example, Figure 2-5 is a conceptual sketch of an encoding disk and a set of contacts that produce one of eight 3-bit binary-coded values depending on the rotational position of the disk. The dark areas of the disk are connected to a

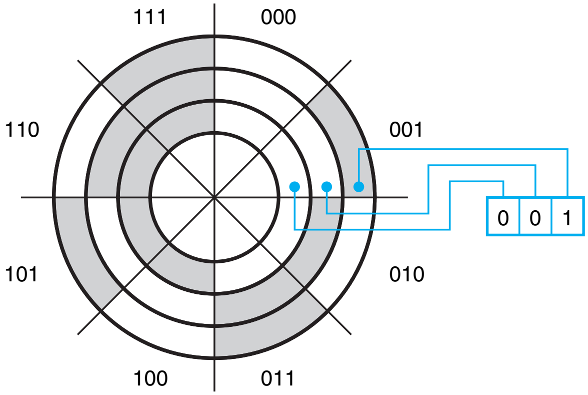


Figure 2-5 A mechanical encoding disk using a 3-bit binary code.

signal source corresponding to logic 1, and the light areas are unconnected, which the contacts interpret as logic 0.

The encoder in Figure 2-5 has a problem when the disk is positioned at certain boundaries between the regions. For example, consider the boundary between the 001 and 010 regions of the disk; two of the encoded bits change here. What value will the encoder produce if the disk is positioned right on the theoretical boundary? Since we're on the border, both 001 and 010 are acceptable. However, because the mechanical assembly is imperfect, the two righthand contacts may both touch a "1" region, giving an incorrect reading of 011. Likewise, a reading of 000 is possible. In general, this sort of problem can occur at any boundary where more than one bit changes. The worst problems occur when all three bits are changing, as at the 000–111 and 011–100 boundaries.

The encoding-disk problem can be solved by devising a digital code in which only one bit changes between each pair of successive code words, as in the redesigned disk in Figure 2-6. As you can see, only one bit of the new disk changes at each border, so borderline readings give us a value on one side or the other of the border. The new code is called a *Gray code*, and its code words are listed in Table 2-8.

Gray code

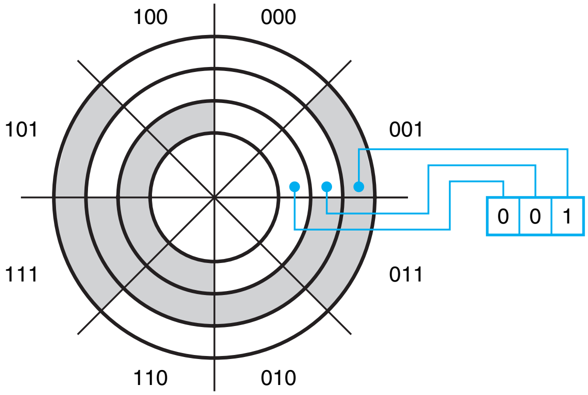


Figure 2-6 A mechanical encoding disk using a 3-bit Gray code.

Table 2-8
A comparison of 3-bit binary code and Gray code.

Decimal Number	Binary Code	Gray Code
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

reflected code

There are two convenient ways to construct a Gray code with any desired number of bits. The first method is based on the fact that Gray code is a *reflected code*; it can be defined (and constructed) recursively using the following rules:

1. A 1-bit Gray code has two code words: 0 and 1.
2. The first 2^n code words of an $(n+1)$-bit Gray code equal the code words of an n-bit Gray code, written in order with a leading 0 appended.
3. The last 2^n code words of an $(n+1)$-bit Gray code equal the code words of an n-bit Gray code, but written in reverse order with a leading 1 appended.

If we draw a line between rows 3 and 4 of Table 2-8, we can see that rules 2 and 3 are true for the 3-bit Gray code. Of course, to construct an n-bit Gray code for an arbitrary value of n with this method, we must also construct a Gray code of each length smaller than n.

The second method allows us to derive a code word in an n-bit Gray-code directly from the corresponding n-bit binary code word:

1. The bits of an n-bit binary or Gray-code code word are numbered from right to left, from 0 to $n-1$.
2. Bit i of a Gray-code code word is 0 if bits i and $i+1$ of the corresponding binary code word are the same, else bit i is 1. (When $i+1=n$, bit n of the binary code word is considered to be 0.)

Again, inspection of Table 2-8 shows that this is true for the 3-bit Gray code.

*2.12 Character Codes

text

As we showed in the preceding section, a string of bits need not represent a number. In fact, most of the information processed by computers is nonnumeric. The most common type of nonnumeric data is *text*, strings of characters from some character set. Each character is represented in the computer by a bit string according to an established convention.

Table 2-9 American Standard Code for Information Interchange (ASCII), Standard No. X3.4-1968 of the American National Standards Institute.

		$b_6b_5b_4$ (column)							
	Row	000	001	010	011	100	101	110	111
$b_3b_2b_1b_0$	(hex)	0	1	2	3	4	5	6	7
0000	0	NUL	DLE	SP	0	@	P	‘	p
0001	1	SOH	DC1	!	1	A	Q	a	q
0010	2	STX	DC2	"	2	B	R	b	r
0011	3	ETX	DC3	#	3	C	S	c	s
0100	4	EOT	DC4	$	4	D	T	d	t
0101	5	ENQ	NAK	%	5	E	U	e	u
0110	6	ACK	SYN	&	6	F	V	f	v
0111	7	BEL	ETB	’	7	G	W	g	w
1000	8	BS	CAN	(	8	H	X	h	x
1001	9	HT	EM	)	9	I	Y	i	y
1010	A	LF	SUB	*	:	J	Z	j	z
1011	B	VT	ESC	+	;	K	[	k	{
1100	C	FF	FS	,	<	L	\	l	|
1101	D	CR	GS	-	=	M	]	m	}
1110	E	SO	RS	.	>	N	ˆ	n	˜
1111	F	SI	US	/	?	O	_	o	DEL

Control codes

NUL	Null	DLE	Data link escape
SOH	Start of heading	DC1	Device control 1
STX	Start of text	DC2	Device control 2
ETX	End of text	DC3	Device control 3
EOT	End of transmission	DC4	Device control 4
ENQ	Enquiry	NAK	Negative acknowledge
ACK	Acknowledge	SYN	Synchronize
BEL	Bell	ETB	End transmitted block
BS	Backspace	CAN	Cancel
HT	Horizontal tab	EM	End of medium
LF	Line feed	SUB	Substitute
VT	Vertical tab	ESC	Escape
FF	Form feed	FS	File separator
CR	Carriage return	GS	Group separator
SO	Shift out	RS	Record separator
SI	Shift in	US	Unit separator
SP	Space	DEL	Delete or rubout

ASCII

The most commonly used character code is *ASCII* (pronounced *ASS key*), the American Standard Code for Information Interchange. ASCII represents each character with a 7-bit string, yielding a total of 128 different characters shown in Table 2-9. The code contains the uppercase and lowercase alphabet, numerals, punctuation, and various nonprinting control characters. Thus, the text string “D’oh!” is represented by the following list of five 7-bit numbers:

1000100 0100111 1101111 1101000 0100001

2.13 Codes for Actions, Conditions, and States

The codes that we’ve described so far are generally used to represent things that we would probably consider to be “data”—things like numbers, positions, and characters. Programmers know that dozens of different data types can be used in a single computer program.

In digital system design, we often encounter nondata applications where a string of bits must be used to control an action, to flag a condition, or to represent the current state of the hardware. Probably the most commonly used type of code for such an application is a simple binary code. If there are n different actions, conditions, or states, we can represent them with a b-bit binary code with $b = \lceil \log_2 n \rceil$ bits. (The brackets $\lceil\ \rceil$ denote the *ceiling function*—the smallest integer greater than or equal to the bracketed quantity. As defined above, b is the smallest integer such that $2^b \geq n$.)

$\lceil\ \rceil$
ceiling function

For example, consider a simple traffic-light controller. The signals at the intersection of a north-south (N-S) and an east-west (E-W) street might be in any of the six states listed in Table 2-10. These states can be encoded in three bits, as shown in the last column of the table. Only six of the eight possible 3-bit code words are needed, and the choice of six code words and the assignment of states to them is arbitrary, so many other encodings are possible. An experienced digi-

Table 2-10 States in a traffic-light controller.

	Lights						
State	***N-S Green***	***N-S Yellow***	***N-S Red***	***E-W Green***	***E-W Yellow***	***E-W Red***	***Code Word***
N-S go	ON	off	off	off	off	ON	000
N-S wait	off	ON	off	off	off	ON	001
N-S delay	off	off	ON	off	off	ON	010
E-W go	off	off	ON	ON	off	off	100
E-W wait	off	off	ON	off	ON	off	101
E-W delay	off	off	ON	off	off	ON	110

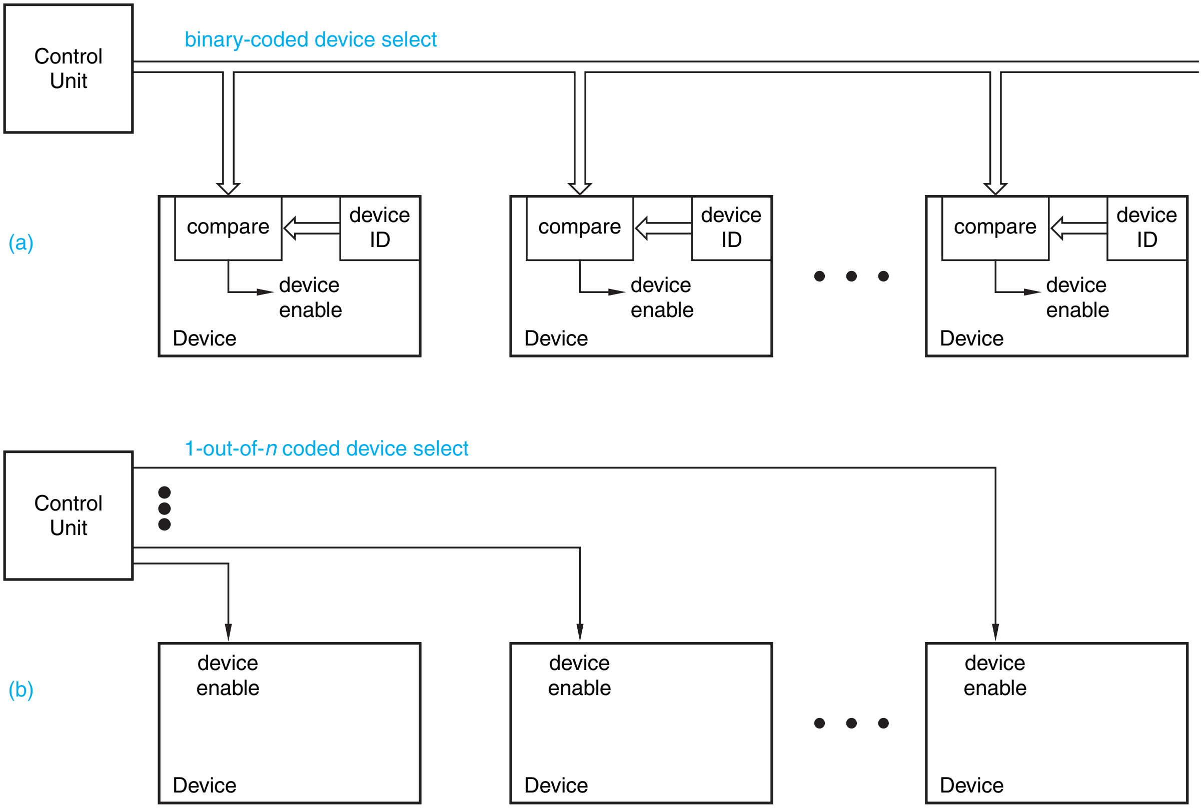


Figure 2-7 Control structure for a digital system with *n* devices: (a) using a binary code; (b) using a 1-out-of-*n* code.

tal designer will choose a particular encoding to minimize circuit cost, or to optimize some other parameter like design time—there's rarely any need to try dozens of possible encodings.

Another application of a binary code is illustrated in Figure 2-7(a). Here, we have a system with n devices, each of which can perform a certain action. The characteristics of the devices are such that they may be enabled to operate only one at a time. The control unit produces a binary-coded "device-select" word with $\lceil \log_2 n \rceil$ bits to indicate which device is enabled at any time. The "device-select" code word is applied to each device, which compares it with its own "device ID" to determine whether it is enabled.

Although a binary code has the smallest code words (fewest bits), it isn't always the best choice for encoding actions, conditions, or states. Figure 2-7(b) shows how to control n devices with a *1-out-of-n code*, an n-bit code in which valid code words have one bit equal to 1 and the rest of the bits equal to 0. Each bit of the 1-out-of-n code word is connected directly to the enable input of a

1-out-of-n code

corresponding device. This simplifies the design of the devices, since they no longer have device IDs; they need only a single "enable" input bit.

inverted 1-out-of-n code

The code words of a 1-out-of-10 code were listed in Table 2-7. Sometimes an all-0s word may also be included in a 1-out-of-n code, to indicate that no device is selected. Another common code is an *inverted 1-out-of-n code*, in which valid code words have one 0 bit and the rest of the bits equal to 1.

In complex systems, a combination of coding techniques may be used. For example, consider a system similar to Figure 2-7(b), in which each of the n devices contains up to s subdevices. The control unit could produce a device-select code word with a 1-out-of-n coded field to select a device, and a $\lceil \log_2 s \rceil$-bit binary-coded field to select one of the s subdevices of the selected device.

m-out-of-n code

An *m-out-of-n code* is a generalization of the 1-out-of-n code in which valid code words have m bits equal to 1 and the rest of the bits equal to 0. An m-out-of-n code word can be detected with an m-input AND gate, which produces a 1 output if all of its inputs are 1. This is fairly simple and inexpensive to do, yet for most values of m, an m-out-of-n code typically has far more valid code words than a 1-out-of-n code. The total number of code words is given by the binomial coefficient $\binom{n}{m}$, which has the value $\frac{n!}{m! \cdot (n-m)!}$. Thus, a 2-out-of-4 code has 6 valid code words, and a 3-out-of-10 code has 120.

8B10B code

An important variation of an m-out-of-n code is the *8B10B code* used in the 802.3z Gigabit Ethernet standard. This code uses 10 bits to represent 8 bits of data using 256 code words, most of which use a 5-out-of-10 coding. However, since $\binom{10}{5}$ is only 252, some 4- and 6-out-of-10 words are also used in the code in a very interesting way; we'll say more about this in Section 2.16.2.

*2.14 *n*-Cubes and Distance

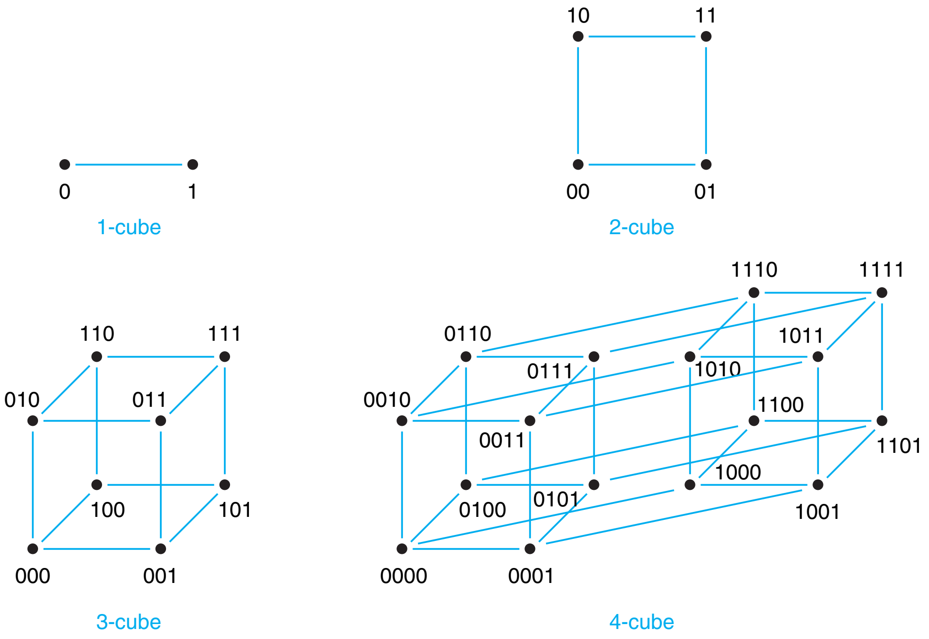
n-cube

An n-bit string can be visualized geometrically, as a vertex of an object called an *n-cube*. Figure 2-8 shows n-cubes for $n = 1, 2, 3, 4$. An n-cube has 2^n vertices, each of which is labeled with an n-bit string. Edges are drawn so that each vertex is adjacent to n other vertices whose labels differ from the given vertex in only one bit. Beyond $n = 4$, n-cubes are really tough to draw.

When n is small, n-cubes make it easy to visualize and understand certain coding and logic-minimization problems. For example, the problem of designing an n-bit Gray code is equivalent to finding a path along the edges of an n-cube, a path that visits each vertex exactly once. The paths for 3- and 4-bit Gray codes are shown in Figure 2-9.

distance
Hamming distance

Cubes also provide a geometrical interpretation for the concept of *distance*, also called *Hamming distance*. The distance between two n-bit strings is the number of bit positions in which they differ. In terms of an n-cube, the distance is the minimum length of a path between the two corresponding vertices.



1-cube

2-cube

3-cube

4-cube

Figure 2-8
n-cubes for *n* = 1, 2, 3, and 4.

Two adjacent vertices have distance 1; vertices 001 and 100 in the 3-cube have distance 2. The concept of distance is crucial in the design and understanding of error-detecting codes, discussed in the next section.

*2.15 Codes for Detecting and Correcting Errors

An *error* in a digital system is the corruption of data from its correct value to some other value. In this sense, an error is caused by a physical *failure*. Failures can be either temporary or permanent. For example, a cosmic ray or alpha particle can cause a temporary failure of a memory circuit, changing the value of a bit stored in it. Letting a circuit get too hot or zapping it with static electricity can cause a permanent failure, so that it never works correctly again.

error
failure
temporary failure
permanent failure

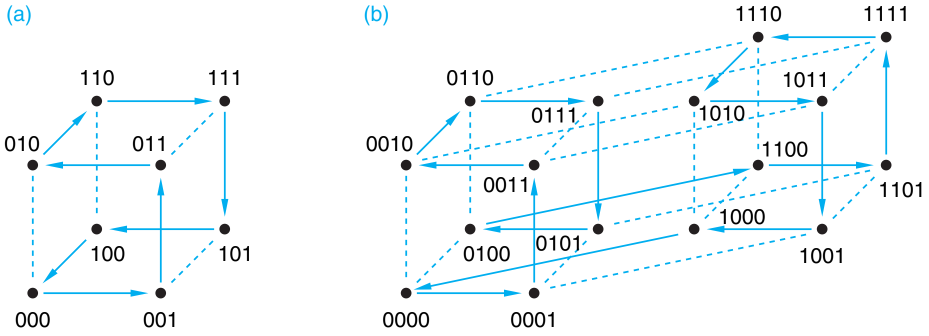


Figure 2-9
Traversing *n*-cubes in Gray-code order: (a) 3-cube; (b) 4-cube.

error model
independent error model
single error
multiple error

The effects of failures on data are predicted by *error models*. The simplest error model, which we consider here, is called the *independent error model*. In this model, a single physical failure is assumed to affect only a single bit of data; the corrupted data is said to contain a *single error*. Multiple failures may cause *multiple errors*—two or more bits in error—but multiple errors are normally assumed to be less likely than single errors.

*2.15.1 Error-Detecting Codes

error-detecting code
noncode word

Recall from our definitions in Section 2.10 that a code that uses n-bit strings need not contain 2^n valid code words; this is certainly the case for the codes that we now consider. An *error-detecting code* has the property that corrupting or garbling a code word will likely produce a bit string that is *not* a code word (a *noncode word*).

A system that uses an error-detecting code generates, transmits, and stores only code words. Thus, errors in a bit string can be detected by a simple rule—if the bit string is a code word, it is assumed to be correct; if it is a noncode word, it definitely contains at least one error.

An n-bit code and its error-detecting properties under the independent error model are easily explained in terms of an n-cube. A code is simply a subset of the vertices of the n-cube. In order for the code to detect all single errors, no code-word vertex can be immediately adjacent to another code-word vertex.

For example, Figure 2-10(a) shows a 3-bit code with five code words. Code word 111 is immediately adjacent to code words 110, 011, and 101. Since a single failure could change 111 to 110, 011, or 101, this code does not detect all single errors. If instead we specify 111 to be a noncode word, we obtain a code that does have the single-error-detecting property, as shown in (b). No single error can change one code word into another.

The ability of a code to detect single errors can be stated in terms of the concept of distance introduced in the preceding section:

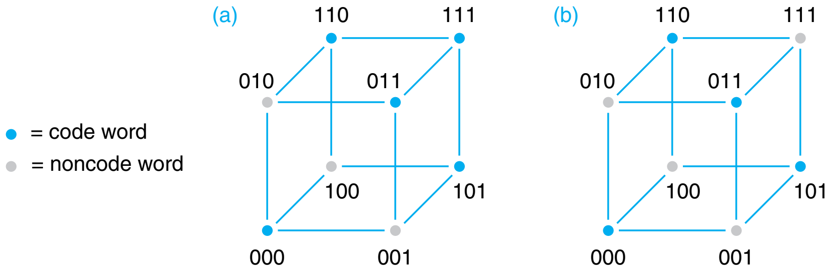
minimum distance

- A code detects all single errors if the *minimum distance* between all possible pairs of code words is 2.

information bit

In general, we need $n + 1$ bits to construct a single-error-detecting code with 2^n code words. The first n bits of a code word, called *information bits*, may

Figure 2-10
Code words in two different 3-bit codes: (a) minimum distance 1, does not detect all single errors; (b) minimum distance 2, detects all single errors.



Information Bits	***Even-parity Code***	***Odd-parity Code***
000	000 0	000 1
001	001 1	001 0
010	010 1	010 0
011	011 0	011 1
100	100 1	100 0
101	101 0	101 1
110	110 0	110 1
111	111 1	111 0

Table 2-11 Distance-2 codes with three information bits.

be any of the 2^n n-bit strings. To obtain a minimum-distance-2 code, we add one more bit, called a *parity bit*, that is set to 0 if there are an even number of 1s among the information bits, and to 1 otherwise. This is illustrated in the first two columns of Table 2-11 for a code with three information bits. A valid $(n+1)$-bit code word has an even number of 1s, and this code is called an *even-parity code*. We can also construct a code in which the total number of 1s in a valid $(n+1)$-bit code word is odd; this is called an *odd-parity code* and is shown in the third column of the table. These codes are also sometimes called *1-bit parity codes*, since they each use a single parity bit.

parity bit

even-parity code

odd-parity code

1-bit parity code

The 1-bit parity codes do not detect 2-bit errors, since changing two bits does not affect the parity. However, the codes can detect errors in any *odd* number of bits. For example, if three bits in a code word are changed, then the resulting word has the wrong parity and is a noncode word. This doesn't help us much, though. Under the independent error model, 3-bit errors are much less likely than 2-bit errors, which are not detectable. Thus, practically speaking, the 1-bit parity codes' error-detection capability stops after 1-bit errors. Other codes, with minimum distance greater than 2, can be used to detect multiple errors.

*2.15.2 Error-Correcting and Multiple-Error-Detecting Codes

By using more than one parity bit, or *check bits*, according to some well-chosen rules, we can create a code whose minimum distance is greater than 2. Before showing how this can be done, let's look at how such a code can be used either to correct single errors or to detect multiple errors.

check bits

Suppose that a code has a minimum distance of 3. Figure 2-11 shows a fragment of the n-cube for such a code. As shown, there are at least two noncode words between each pair of code words. Now suppose we transmit code words, and assume that failures affect at most one bit of each received code word. Then a received noncode word with a 1-bit error will be closer to the originally transmitted code word than to any other code word. Therefore, when we receive a

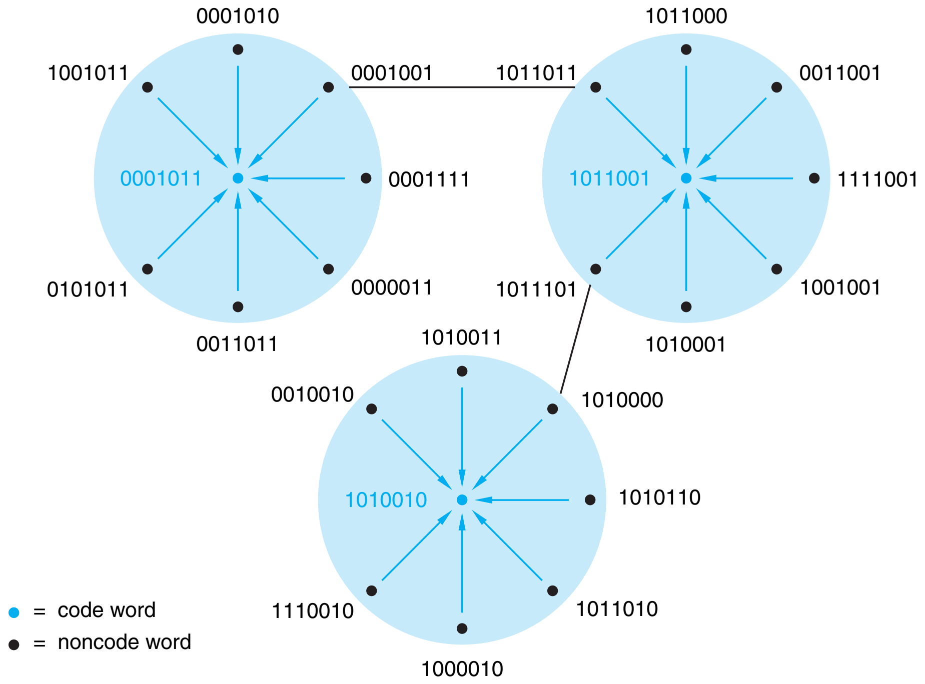


Figure 2-11 Some code words and noncode words in a 7-bit, distance-3 code.

error correction

decoding

decoder

error-correcting code

noncode word, we can *correct* the error by changing the received noncode word to the nearest code word, as indicated by the arrows in the figure. Deciding which code word was originally transmitted to produce a received word is called *decoding*, and the hardware that does this is an error-correcting *decoder*.

A code that is used to correct errors is called an *error-correcting code*. In general, if a code has minimum distance $2c + 1$, it can be used to correct errors that affect up to c bits ($c = 1$ in the preceding example). If a code's minimum distance is $2c + d + 1$, it can be used to correct errors in up to c bits and to detect errors in up to d additional bits.

For example, Figure 2-12(a) shows a fragment of the n-cube for a code with minimum distance 4 ($c = 1$, $d = 1$). Single-bit errors that produce noncode words 00101010 and 11010011 can be corrected. However, an error that produces 10100011 cannot be corrected, because no single-bit error can produce this noncode word, and either of two 2-bit errors could have produced it. So the code can detect a 2-bit error, but it cannot correct it.

With an error-correcting code, when a noncode word is received, we don't know which code word was originally transmitted; we only know which code

DECISIONS, DECISIONS The names *decoding* and *decoder* make sense, since they are just distance-1 perturbations of *deciding* and *decider*.

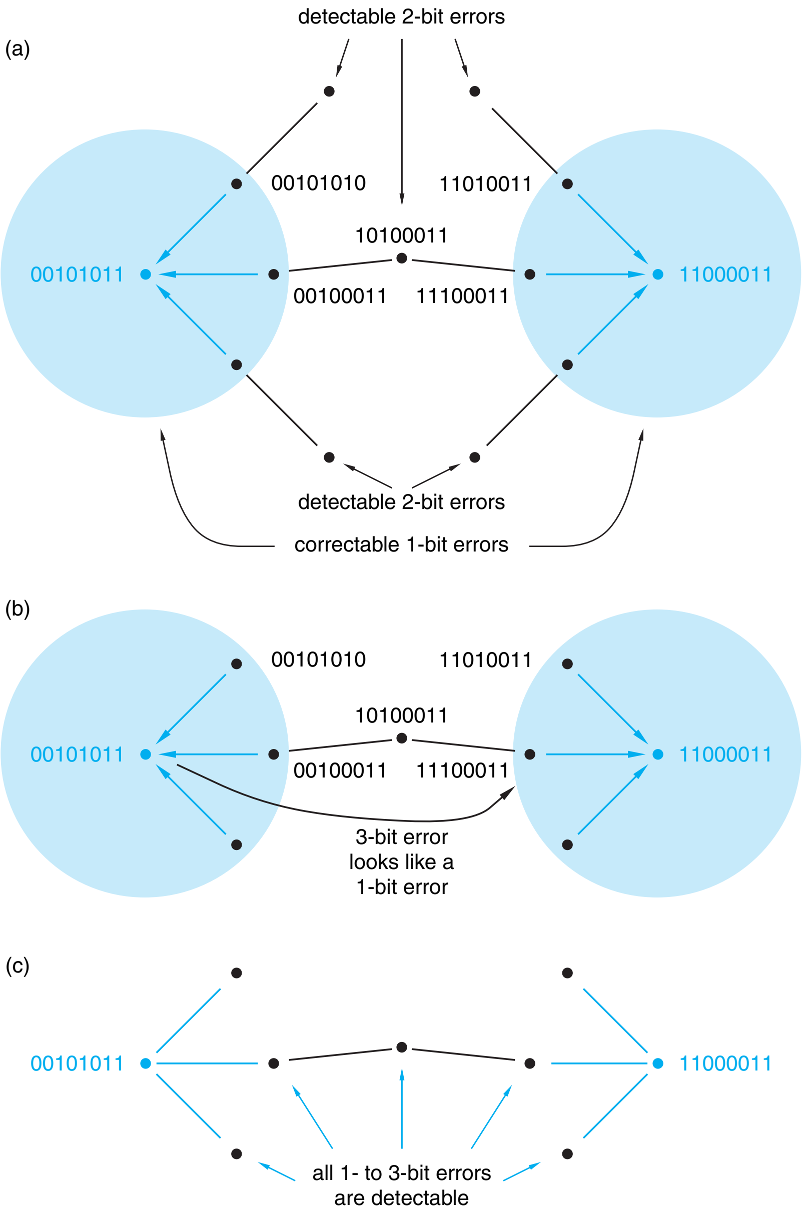


Figure 2-12
Some code words and noncode words in an 8-bit, distance-4 code: (a) correcting 1-bit and detecting 2-bit errors; (b) incorrectly "correcting" a 3-bit error; (c) correcting no errors but detecting up to 3-bit errors.

word is closest to what we've received. Thus, as shown in Figure 2-12(b), a 3-bit error may be "corrected" to the wrong value. The possibility of making this kind of mistake may be acceptable if 3-bit errors are very unlikely to occur. On the other hand, if we are concerned about 3-bit errors, we can change the decoding policy for the code. Instead of trying to correct errors, we just flag all noncode words as uncorrectable errors. Thus, as shown in (c), we can use the same distance-4 code to detect up to 3-bit errors but correct no errors ($c = 0$, $d = 3$).

*2.15.3 Hamming Codes

Hamming code

In 1950, R. W. Hamming described a general method for constructing codes with a minimum distance of 3, now called *Hamming codes*. For any value of i, his method yields a (2^i-1)-bit code with i check bits and 2^i-1-i information bits. Distance-3 codes with a smaller number of information bits are obtained by deleting information bits from a Hamming code with a larger number of bits.

parity-check matrix

The bit positions in a Hamming code word can be numbered from 1 through 2^i-1. In this case, any position whose number is a power of 2 is a check bit, and the remaining positions are information bits. Each check bit is grouped with a subset of the information bits, as specified by a *parity-check matrix*. As shown in Figure 2-13(a), each check bit is grouped with the information positions whose numbers have a 1 in the same bit when expressed in binary. For example, check bit 2 (010) is grouped with information bits 3 (011), 6 (110), and 7 (111). For a given combination of information-bit values, each check bit is chosen to produce even parity, that is, so the total number of 1s in its group is even.

Traditionally, the bit positions of a parity-check matrix and the resulting code words are rearranged so all of the check bits are on the righthand side, as in Figure 2-13(b). The first two columns of Table 2-12 list the resulting code words.

We can prove that the minimum distance of a Hamming code is 3 by proving that at least a 3-bit change must be made to a code word to obtain another code word. That is, we'll prove that a 1-bit or 2-bit change in a code word yields a noncode word.

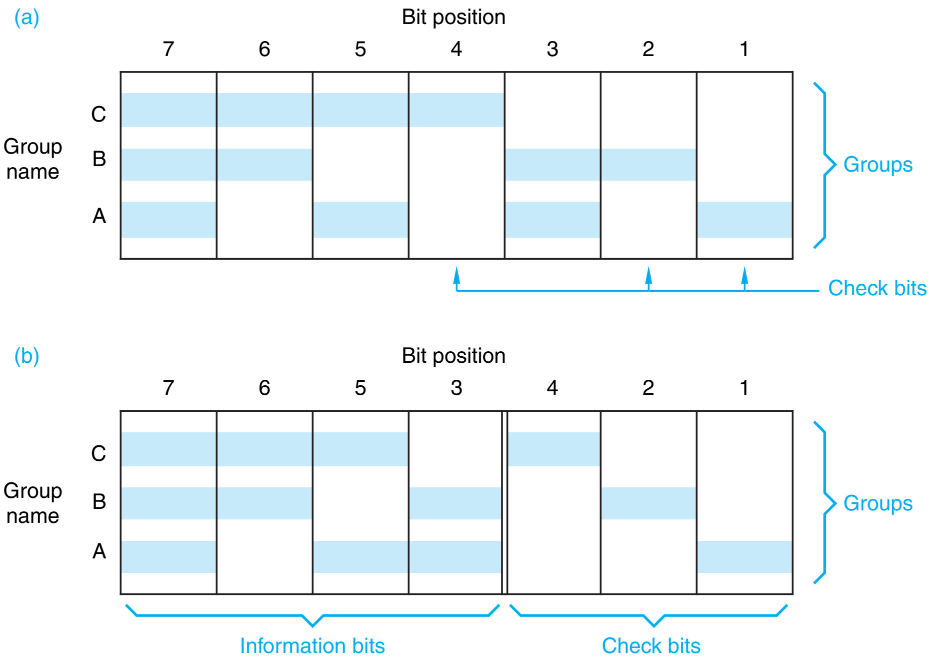


Figure 2-13 Parity-check matrices for 7-bit Hamming codes: (a) bit positions in numerical order; (b) check bits and information bits separated.

Minimum-Distance-3 Code		***Minimum-Distance-4 Code***	
Information Bits	***Parity Bits***	***Information Bits***	***Parity Bits***
0000	000	0000	0000
0001	011	0001	0111
0010	101	0010	1011
0011	110	0011	1100
0100	110	0100	1101
0101	101	0101	1010
0110	011	0110	0110
0111	000	0111	0001
1000	111	1000	1110
1001	100	1001	1001
1010	010	1010	0101
1011	001	1011	0010
1100	001	1100	0011
1101	010	1101	0100
1110	100	1110	1000
1111	111	1111	1111

Table 2-12
Code words in distance-3 and distance-4 Hamming codes with four information bits.

If we change one bit of a code word, in position *j*, then we change the parity of every group that contains position *j*. Since every information bit is contained in at least one group, at least one group has incorrect parity, and the result is a noncode word.

What happens if we change two bits, in positions *j* and *k*? Parity groups that contain both positions *j* and *k* will still have correct parity, since parity is not affected when an even number of bits are changed. However, since *j* and *k* are different, their binary representations differ in at least one bit, corresponding to one of the parity groups. This group has only one bit changed, resulting in incorrect parity and a noncode word.

If you understand this proof, you should also understand how the position-numbering rules for constructing a Hamming code are a simple consequence of the proof. For the first part of the proof (1-bit errors), we required that the position numbers be nonzero. And for the second part (2-bit errors), we required that no two positions have the same number. Thus, with an *i*-bit position number, you can construct a Hamming code with up to $2^i - 1$ bit positions.

Table 2-13
Word size of distance-3 and distance-4 extended Hamming codes

	Minimum-Distance-3 Codes		Minimum-Distance-4 Codes	
Information Bits	***Parity Bits***	***Total Bits***	***Parity Bits***	***Total Bits***
1	2	3	3	4
≤ 4	3	≤ 7	4	≤ 8
≤ 11	4	≤ 15	5	≤ 16
≤ 26	5	≤ 31	6	≤ 32
≤ 57	6	≤ 63	7	≤ 64
≤ 120	7	≤ 127	8	≤ 128

error-correcting decoder

syndrome

The proof also suggests how we can design an *error-correcting decoder* for a received Hamming code word. First, we check all of the parity groups; if all have even parity, then the received word is assumed to be correct. If one or more groups have odd parity, then a single error is assumed to have occurred. The pattern of groups that have odd parity (called the *syndrome*) must match one of the columns in the parity-check matrix; the corresponding bit position is assumed to contain the wrong value and is complemented. For example, using the code defined by Figure 2-13(b), suppose we receive the word 0101011. Groups B and C have odd parity, corresponding to position 6 of the parity-check matrix (the syndrome is 110, or 6). By complementing the bit in position 6 of the received word, we determine that the correct word is 0001011.

extended Hamming code

A distance-3 Hamming code can easily be extended to increase its minimum distance to 4. We simply add one more check bit, chosen so that the parity of all the bits, including the new one, is even. As in the 1-bit even-parity code, this bit ensures that all errors affecting an odd number of bits are detectable. In particular, any 3-bit error is detectable. We already showed that 1- and 2-bit errors are detected by the other parity bits, so the minimum distance of the extended code must be 4.

Distance-3 and distance-4 extended Hamming codes are commonly used to detect and correct errors in computer memory systems, especially in large servers where memory circuits account for the bulk of the system's electronics and hence failures. These codes are especially attractive for very wide memory words, since the required number of parity bits grows slowly with the width of the memory word, as shown in Table 2-13.

*2.15.4 CRC Codes

cyclic-redundancy-check (CRC) code

Beyond Hamming codes, many other error-detecting and -correcting codes have been developed. The most important codes, which happen to include Hamming codes, are the *cyclic-redundancy-check (CRC) codes*. A rich theory has been developed for these codes, focusing both on their error-detecting and error-correcting properties and on the design of inexpensive encoders and decoders for them (see References).

Two important applications of CRC codes are in disk drives and in data networks. In a disk drive, each block of data (typically 512 bytes) is protected by a CRC code, so that errors within a block can be detected and often corrected. In a data network, each packet of data has appended to it check bits in a CRC code. The CRC codes for both applications were selected because of their burst-error detecting properties. In addition to single-bit errors, they can detect multibit errors that are clustered together within the disk block or packet. Such errors are more likely than errors of randomly distributed bits, because of the likely physical causes of errors in the two applications—surface defects in disk drives and noise bursts in communication links.

*2.15.5 Two-Dimensional Codes

Another way to obtain a code with large minimum distance is to construct a *two-dimensional code*, as illustrated in Figure 2-14(a). The information bits are conceptually arranged in a two-dimensional array, and parity bits are provided to check both the rows and the columns. A code C_{row} with minimum distance d_{row} is used for the rows, and a possibly different code C_{col} with minimum distance d_{col} is used for the columns. That is, the row-parity bits are selected so that each row

two-dimensional code

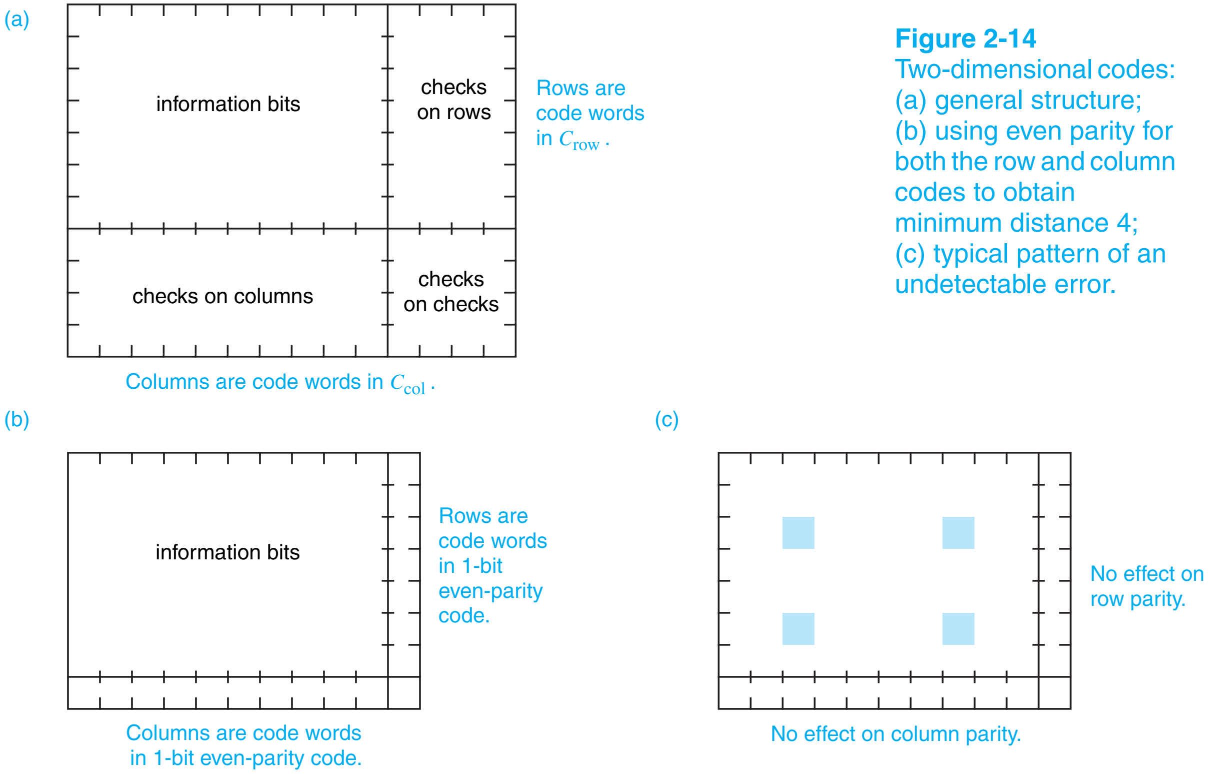

(a)
information bits
checks on rows
Rows are code words in C_{row}.
checks on columns
checks on checks
Columns are code words in C_{col}.
(b)
information bits
Rows are code words in 1-bit even-parity code.
Columns are code words in 1-bit even-parity code.
(c)
No effect on row parity.
No effect on column parity.

Figure 2-14
Two-dimensional codes: (a) general structure; (b) using even parity for both the row and column codes to obtain minimum distance 4; (c) typical pattern of an undetectable error.

is a code word in C_{row} and the column-parity bits are selected so that each column is a code word in C_{col}. (The "corner" parity bits can be chosen according to either code.) The minimum distance of the two-dimensional code is the product of d_{row} and d_{col}; in fact, two-dimensional codes are sometimes called *product codes*.

product code

As shown in Figure 2-14(b), the simplest two-dimensional code uses 1-bit even-parity codes for the rows and columns and has a minimum distance of $2 \cdot 2$, or 4. You can easily prove that the minimum distance is 4 by convincing yourself that any pattern of one, two, or three bits in error causes incorrect parity of a row or a column or both. In order to obtain an undetectable error, at least four bits must be changed in a rectangular pattern as in (c).

The error-detecting and -correcting procedures for this code are straightforward. Assume we are reading information one row at a time. As we read each row, we check its row code. If an error is detected in a row, we can't tell which bit is wrong from the row check alone. However, assuming only one row is bad, we can reconstruct it by forming the bit-by-bit Exclusive OR of the columns, omitting the bad row, but including the column-check row.

To obtain an even larger minimum distance, a distance-3 or -4 Hamming code can be used for the row or column code or both. It is also possible to construct a code in three or more dimensions, with minimum distance equal to the product of the minimum distances in each dimension.

RAID

An important application of two-dimensional codes is in some *RAID* storage systems. RAID stands for "redundant array of inexpensive disks." In the RAID scheme, $n+1$ identical disk drives may be used to store n disks worth of data. For example, four 2-terabyte drives could be used to store 8 terabytes of nonredundant data, and a fifth 2-terabyte drive would be used to store checking information. This setup could store about 200 high-definition movies in MPEG-2 format, and never lose one to a (single) hard-drive crash!

Figure 2-15 shows a simplified scheme for a two-dimensional code in a RAID system; each disk drive is considered to be a row in the code. Each drive

KILO-, MEGA-, GIGA-, TERA-

The prefixes k (kilo-), M (mega-), G (giga-), and T (tera-) mean 10^3, 10^6, 10^9, and 10^{12}, respectively, when referring to bps, hertz, ohms, watts, and most other engineering quantities. However, when referring to computer memory sizes, the prefixes mean 2^{10}, 2^{20}, 2^{30}, and 2^{40}. Historically, the prefixes were co-opted for this purpose because memory sizes are normally powers of 2, and 2^{10} (1024) is very close to 1000.

Perversely, when referring to the sizes of disk and removable-storage devices (including SD cards and the like), the prefixes go back to referring to powers of 10; the drive manufacturers undoubtedly did this originally to make their drives seem a little bit bigger. Percentage-wise, the size disparity between the nomenclatures has only grown with increasing storage capacities.

So, when somebody offers you 70 kilobucks a year for your first engineering job, it's up to you to negotiate what the prefix means!

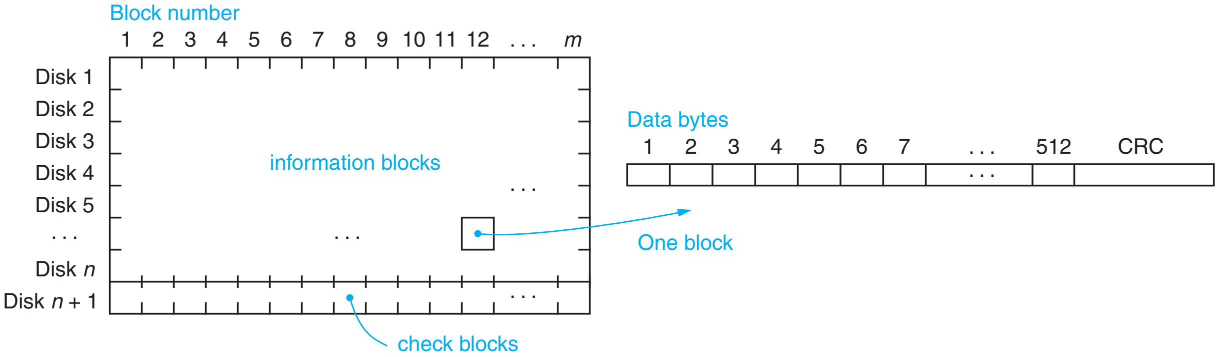


Figure 2-15 Structure of error-correcting code for a RAID-5 system.

stores m blocks of data, where a block typically contains 512 bytes. For example, a 2-terabyte drive would store about 4 billion blocks. As shown in the figure, each block includes its own check bits in a CRC code, to detect and possibly correct errors within that block. The first n drives store the nonredundant data. Each block in drive $n + 1$ stores parity bits for the corresponding blocks in the first n drives. That is, each bit i in drive $n + 1$, block b, is chosen so that there are an even number of 1s in block b, bit position i, across all the drives.

In operation, errors in the information blocks are detected by the CRC code. Whenever an error is detected on one of the drives and cannot be corrected using the local CRC code, the block's original contents can still be reconstructed by computing the parity of the corresponding blocks in all the other drives, including drive $n + 1$. This method still works even if you lose *all* of the data on a single drive.

Although RAID correction operations require n extra disk reads plus some computation, it's better than losing your data! Write operations also have extra disk accesses, to update the corresponding check block when an information block is written (see Exercise 2.62). Since disk writes are much less frequent than reads in typical applications, this overhead usually is not a problem.

*2.15.6 Checksum Codes

The parity-checking operation that we've used in the previous subsections is essentially modulo-2 addition of bits—the sum modulo 2 of a group of bits is 0 if the number of 1s in the group is even, and 1 if it is odd. This approach of modular addition can be extended to other bases besides 2 to form check digits.

For example, a computer stores information as a sequence of 8-bit bytes. Each byte may be considered to have a decimal value from 0 to 255. Therefore, we can use modulo-256 addition to check the bytes. We form a single check byte, called a *checksum*, that is the sum modulo 256 of all the information bytes. The resulting *checksum code* can detect any single *byte* error, since such an error will cause a recomputed sum of bytes to disagree with the checksum.

checksum

checksum code

ones'-complement checksum code

Checksum codes can also use a different modulus of addition. In particular, checksum codes using modulo-255 or -65535, ones'-complement addition are important because of their special computational and error-detecting properties, and because they are used to check the headers of IPv4 packets on the Internet.

*2.15.7 *m*-out-of-*n* Codes

The 1-out-of-*n* and *m*-out-of-*n* codes we introduced in Section 2.13 have a minimum distance of 2, since changing only one bit changes the total number of 1s in a code word and therefore produces a noncode word.

unidirectional error

These codes have another useful error-detecting property—they detect unidirectional multiple errors. In a *unidirectional error*, all of the erroneous bits change in the same direction (0s change to 1s, or vice versa). This property is very useful in systems where the predominant error mechanism tends to change all bits in the same direction.

2.16 Codes for Transmitting and Storing Serial Data

2.16.1 Parallel and Serial Data

parallel data

Most computers and other digital systems transmit and store data in a *parallel* format. In parallel data transmission, a separate signal line is provided for each bit of a data word. In parallel data storage, all of the bits of a data word can be written or read simultaneously.

serial data

Parallel formats are not cost effective for some applications. For example, parallel transmission of data bytes in an Ethernet cable would require eight signal wires in each direction, and parallel storage of data bytes on a magnetic disk would require a disk drive with read/write heads for eight separate tracks. *Serial* formats allow data to be transmitted or stored one bit at a time. Even in board-level design and in computer-peripheral interfacing, serial formats can reduce cost and simplify certain system-design problems. For example, the PCI Express serial interface evolved from the original, parallel PCI bus used by add-in modules in desktop computers.

bit rate, bps

Figure 2-16 illustrates some of the basic ideas in serial data transmission. A repetitive clock signal, named CLOCK in the figure, defines the rate at which bits are transmitted, one bit per clock cycle. Thus, the *bit rate* in bits per second (bps) numerically equals the clock frequency in cycles per second (hertz, or Hz).

bit time

bit cell

line code

Non-Return-to-Zero (NRZ)

The reciprocal of the bit rate is called the *bit time* and numerically equals the clock period in seconds (s). This amount of time is reserved on the serial data line (named SERDATA in Figure 2-16) for each bit that is transmitted. The time occupied by each bit is sometimes called a *bit cell*. The format of the actual signal that appears on the line during each bit cell depends on the *line code*. In the simplest line code, called *Non-Return-to-Zero (NRZ)*, a 1 is transmitted by placing a 1 on the line for the entire bit cell, and a 0 is transmitted as a 0. More complex line codes have other rules, as discussed in the next subsection.

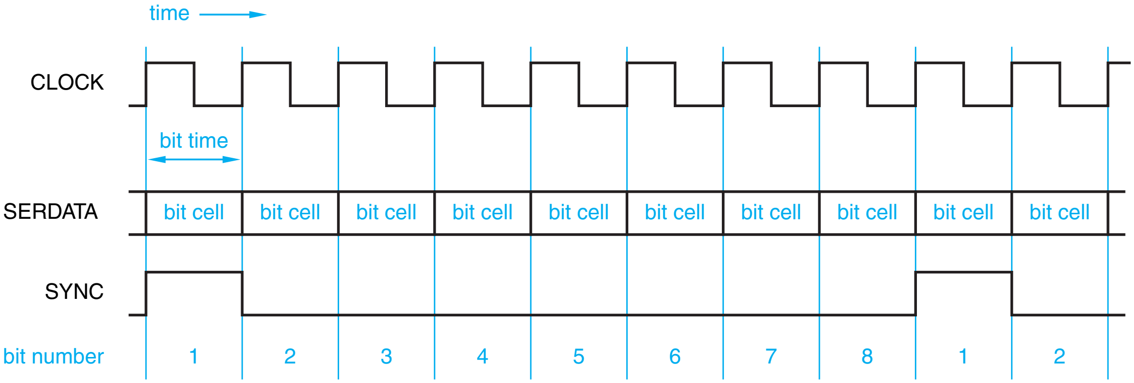


Figure 2-16 Basic concepts for serial data transmission.

Regardless of the line code, a serial data-transmission or storage system needs some way of identifying the significance of each bit in the serial stream. For example, suppose that 8-bit bytes are transmitted serially. How can we tell which bit is the first bit of each byte? In Figure 2-16, a *synchronization signal*, named SYNC provides the necessary information; it is 1 only for the first bit of each byte.

synchronization signal

Evidently, we need a minimum of three signals to recover a serial data stream: a clock to define the bit cells, a synchronization signal to define the word boundaries, and the serial data itself. In some applications, like the interconnection of modules in a computer or telecommunications system, a separate wire is used for each of these signals, since reducing the number of wires per connection from *n* to three is savings enough.

But in many applications, the cost of having three separate signals is still too high (e.g., three signals per direction for Ethernet, or using multiple radios in any kind of wireless system). Such systems typically combine all three signals into a single serial data stream and use sophisticated analog and digital circuits to recover clock and synchronization information from the data stream, as we'll discuss in the next subsection.

*2.16.2 Serial Line Codes

The most commonly used line codes for serial data are illustrated in Figure 2-17 on the next page. As we describer earlier, the NRZ code transmits each bit value for the entire bit cell. While this is the simplest coding scheme for short-distance transmission, it generally requires a clock signal to be sent along with the data to define the bit cells. Otherwise, it is not possible for the receiver to determine how many 0s or 1s are represented by a continuous 0 or 1 level. For example, without a clock signal to define the bit cells, the NRZ waveform in Figure 2-17 might be erroneously interpreted as 01010.

A *digital phase-locked loop (DPLL)* is an analog/digital circuit that can be used to recover a clock signal from a serial data stream. The DPLL works only if

digital phase-locked loop (DPLL)

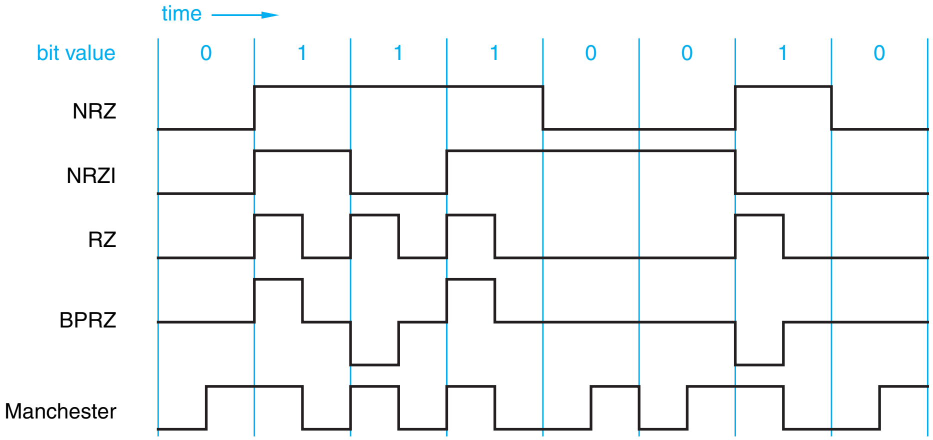


Figure 2-17 Commonly used line codes for serial data.

the serial data stream contains enough 0-to-1 and 1-to-0 transitions to give the DPLL "hints" about when the original clock transitions took place. With NRZ-coded data, the DPLL works only if the data does not contain any long, continuous streams of 1s or 0s.

transition-sensitive media

Some serial transmission and storage media are *transition sensitive*; they cannot transmit or store absolute 0 or 1 levels, only transitions between two discrete levels. For example, a magnetic disk or tape stores information by changing the polarity of the medium's magnetization in regions corresponding to the stored bits. When the information is recovered, it is not feasible to determine the absolute magnetization polarity of a region, only that the polarity changes between one region and the next.

Non-Return-to-Zero Invert-on-1s (NRZI)

Data stored in NRZ format on transition-sensitive media cannot be recovered unambiguously; the data in Figure 2-17 might be interpreted as 01110010 or 10001101. The *Non-Return-to-Zero Invert-on-1s (NRZI)* code overcomes this limitation by sending a 1 as the opposite of the level that was sent during the previous bit cell, and a 0 as the same level. A DPLL can recover the clock from NRZI-coded data as long as the data does not contain any long, continuous streams of 0s.

Return-to-Zero (RZ)

The *Return-to-Zero (RZ)* code is similar to NRZ except that, for a 1 bit, the 1 level is transmitted only for a fraction of the bit time, usually 1/2. With this code, data patterns that contain a lot of 1s create lots of transitions for a DPLL to use to recover the clock. However, as in the other line codes, a string of 0s has no transitions, and a long string of 0s makes clock recovery impossible.

DC balance

Another requirement of some transmission media, like high-speed fiber-optic links, is that the serial data stream be *DC balanced*. That is, it must have an equal number of 1s and 0s; any long-term DC component in the stream (created by having a lot more 1s than 0s or vice versa) creates a bias at the receiver that reduces its ability to distinguish reliably between 1s and 0s.

Ordinarily, NRZ, NRZI or RZ data has no guarantee of DC balance; there's nothing to prevent a user data stream from having a long string of words with more than 1s than 0s or vice versa. However, DC balance can still be achieved by using a few extra bits to code the user data in a *balanced code*, in which each code word has an equal number of 1s and 0s, and then sending these code words in NRZ format.

balanced code

For example, in Section 2.13 we introduced the 8B10B code, which codes 8 bits of user data into 10 bits in a mostly 5-out-of-10 code. Recall that there are only 252 5-out-of-10 code words, so at least four "extra" code words are needed (plus a few more to convey certain control information). But there are another $\binom{10}{4} = 210$ 4-out-of-10 code words, and an equal number of 6-out-of-10 code words. Of course, these code words aren't quite DC balanced. The 8B10B code solves this problem by associating with each "extra" 8-bit value to be encoded a *pair* of unbalanced code words, one 4-out-of-10 ("light") and the other 6-out-of-10 ("heavy"). The coder also keeps track of the *running disparity*, a single bit of information indicating whether the last unbalanced code word that it transmitted was heavy or light. When it comes time to transmit another unbalanced code word, the coder selects the one of the pair with the opposite weight. This simple trick makes available 252 + 210 = 462 code words, more than enough for the 8B10B to encode 8 bits of user data. Some of the "extra" code words are used to conveniently encode nondata conditions on the serial line, such as IDLE, SYNC, and ERROR. Not all the unbalanced code words are used. Also, some of the balanced code words, like 0000011111, are not used either, in favor of unbalanced pairs that contain more transitions.

running disparity

A DPLL can recover a clock signal, but not byte synchronization. Still, byte synchronization can be achieved in various clever ways by embedding special patterns into the long-term serial data stream, recognizing them digitally, and then "locking" onto them. For example, suppose that the IDLE code word in a 10-bit code is 1011011000, and IDLE is sent continuously at system startup. Then the beginning of the code word can be easily recognized as the bit after three 0s in a row. Successive code words, even if not IDLE, can be expected to begin at every tenth bit time thereafter. Of course, additional work is needed to recognize loss of synchronization due to noise, and to get the transmitter to send IDLE again, and this is an area of much cleverness and variety.

The preceding codes transmit or store only two signal levels, 0 and 1. The *Alternate-Mark-Inversion (AMI)* code transmits three signal levels: +1, 0, and −1 . The code is like RZ except that 1s are transmitted alternately as +1 and −1. The word "mark" in the code's name comes from old phone-company parlance, which called a 1 a "mark."

Alternate-Mark-Inversion (AMI)

The big advantage of AMI over RZ is that it's DC balanced. This makes it possible to send AMI streams over transmission media that cannot tolerate a DC component, such as transformer-coupled analog phone lines. In fact, the AMI code was used in T1 digital telephone links for decades, where analog speech signals are carried as streams of 8000 8-bit digital samples per second that are transmitted in AMI format on 64-Kbps serial channels.

As with RZ, it is possible to recover a clock signal from a AMI stream as long as there aren't too many 0s in a row. Although TPC (The Phone Company) has no control over what you say (at least, not yet), they still have a simple way of limiting runs of 0s. If one of the 8-bit bytes that results from sampling your analog speech pattern is all 0s, they simply change the second-least significant bit to 1! This is called *zero-code suppression* and I'll bet you never noticed it. And this is also why, in many data applications of T1 links, you get only 56 Kbps of usable data per 64-Kbps channel; in data applications, the LSB of each byte is always set to 1 to prevent zero-code suppression from changing the other bits.

zero-code suppression

The last code in Figure 2-17 is called *Manchester* or *diphase* code. The major strength of this code is that, regardless of the transmitted data pattern, it provides at least one transition per bit cell, making it very easy to recover the clock. As shown in the figure, a 0 is encoded as a 0-to-1 transition in the middle of the bit cell, and a 1 is encoded as a 1-to-0 transition. The Manchester code's major strength is also its major weakness. Since it has more transitions per bit cell than other codes, it also requires more media bandwidth to transmit a given bit rate. Bandwidth is not a problem in coaxial cable, however, which was used in the original Ethernet local area networks to carry Manchester-coded serial data at the rate of 10 Mbps (megabits per second).

Manchester
diphase

References

Precise, thorough, and entertaining discussions of topics in the first nine sections of this chapter can be found in Donald E. Knuth's *Seminumerical Algorithms*, third edition (Addison-Wesley, 1997). Mathematically inclined readers will find Knuth's analysis of the properties of number systems and arithmetic to be excellent, and all readers should enjoy the insights and history sprinkled throughout the text.

Descriptions of algorithms for arithmetic operations appear in *Digital Arithmetic* by Miloš Ercegovac and Tomas Láng (Morgan Kaufmann, 2003). A thorough discussion of arithmetic techniques and floating-point number systems can be found in *Introduction to Arithmetic for Digital Systems Designers* by Shlomo Waser and Michael J. Flynn (Oxford University Press, 1995).

CRC codes are based on the theory of *finite fields,* which was developed by French mathematician Évariste Galois (1811–1832) shortly before he was killed in a duel with a political opponent. The classic book on error-detecting and error-correcting codes is *Error-Correcting Codes* by W. W. Peterson and E. J. Weldon, Jr. (MIT Press, 1972, second edition); however, this book is recommended only for mathematically sophisticated readers. A more accessible

finite fields

introduction to coding can be found in *Error Correcting Codes: A Mathematical Introduction* by John Baylis (Chapman & Hall/CRC, 1997), despite its use of the word “mathematical” in the title. Another treatise on coding is *Fundamentals of Error-Correcting Codes* by W. C. Huffman and V. Pless (Cambridge University Press, 2010).

An introduction to coding techniques for serial data transmission, as well as very useful coverage of the higher layers of communication and networking, appears in *Data and Computer Communications* by William Stallings (Pearson, 2014, tenth edition).

The structure of the 8B10B code and the rationale behind it is explained nicely in the original IBM patent by Peter Franaszek and Albert Widmer, U.S. patent number 4,486,739 (1984). This and all U.S. patents issued after 1971 can be found on the Web at `www.uspto.gov`. or at `patents.google.com`.

Drill Problems

2.1 Perform the following number system conversions:

(a) $1011101_2 = ?_{16}$ (b) $137023_8 = ?_2$

(c) $10011011_2 = ?_{16}$ (d) $64.23_8 = ?_2$

(e) $11000.0111_2 = ?_{16}$ (f) $\mathrm{D3B6}_{16} = ?_2$

(g) $11110101_2 = ?_8$ (h) $\mathrm{ACBD}_{16} = ?_2$

(i) $101101.0111_2 = ?_8$ (j) $\mathrm{37E.73}_{16} = ?_2$

2.2 Convert the following octal numbers into binary and hexadecimal:

(a) $4321_8 = ?_2 = ?_{16}$ (b) $1772631_8 = ?_2 = ?_{16}$

(c) $533434_8 = ?_2 = ?_{16}$ (d) $245277 = ?_2 = ?_{16}$

(e) $7542.22_8 = ?_2 = ?_{16}$ (f) $63712.1515_8 = ?_2 = ?_{16}$

2.3 Convert the following hexadecimal numbers into binary and octal:

(a) $2047_{16} = ?_2 = ?_8$ (b) $\mathrm{6CBA}_{16} = ?_2 = ?_8$

(c) $\mathrm{FEAB}_{16} = ?_2 = ?_8$ (d) $\mathrm{C079}_{16} = ?_2 = ?_8$

(e) $\mathrm{79EF.3C}_{16} = ?_2 = ?_8$ (f) $\mathrm{BAD.DADD}_{16} = ?_2 = ?_8$

2.4 What are the octal values of the four 8-bit bytes in the 32-bit number with octal representation 34567654321_8?

2.5 Convert the following numbers into decimal:

(a) $1111011_2 = ?_{10}$ (b) $173016_8 = ?_{10}$

(c) $10110001_2 = ?_{10}$ (d) $66.27_8 = ?_{10}$

(e) $10101.1001_2 = ?_{10}$ (f) $\mathrm{FCB6}_{16} = ?_{10}$

(g) $12210_3 = ?_{10}$ (h) $\mathrm{FEED}_{16} = ?_{10}$

(i) $7716_8 = ?_{10}$ (j) $\mathrm{15C1.93}_{16} = ?_{10}$

2.6 Perform the following number-system conversions:

(a) $129_{10} = ?_2$ (b) $4398_{10} = ?_8$

(c) $207_{10} = ?_2$ (d) $4196_{10} = ?_8$

(e) $138_{10} = ?_2$ (f) $22439_{10} = ?_{16}$

(g) $797_{10} = ?_5$ (h) $52844_{10} = ?_{16}$

(i) $1333_{10} = ?_8$ (j) $64000_{10} = ?_{16}$

2.7 Add the following pairs of binary numbers, showing all carries:

$$\begin{array}{lrlrlrlr}
\text{(a)} & 110011 & \text{(b)} & 101110 & \text{(c)} & 11011101 & \text{(d)} & 1110011 \\
 & \underline{+\ \ \ 11001} & & \underline{+\ 100101} & & \underline{+\ \ \ 1100011} & & \underline{+\ 1101001}
\end{array}$$

2.8 Repeat Drill 2.7 using subtraction instead of addition, and showing borrows instead of carries.

2.9 Add the following pairs of octal numbers:

$$\begin{array}{lrlrlrlr}
\text{(a)} & 1362 & \text{(b)} & 47135 & \text{(c)} & 175314 & \text{(d)} & 110321 \\
 & \underline{+\ \ \ 4231} & & \underline{+\ \ \ 5145} & & \underline{+\ 152405} & & \underline{+\ 57573}
\end{array}$$

2.10 Add the following pairs of hexadecimal numbers:

$$\begin{array}{lrlrlrlr}
\text{(a)} & 1872 & \text{(b)} & \text{4F1A5} & \text{(c)} & \text{F32B} & \text{(d)} & \text{1B90F} \\
 & \underline{+\ \ \ 4737} & & \underline{+\ \ \ \text{B7D4}} & & \underline{+\ \ \text{2AE6}} & & \underline{+\ \ \text{A44E}}
\end{array}$$

2.11 Write the 8-bit signed-magnitude, two's-complement, and ones'-complement representations for each of these decimal numbers: +19, +105, +81, –47, –2, –112.

2.12 Indicate whether or not overflow occurs when adding the following 8-bit two's-complement numbers:

$$\begin{array}{lrlrlrlr}
\text{(a)} & 11010110 & \text{(b)} & 11011111 & \text{(c)} & 00011101 & \text{(d)} & 01110001 \\
 & \underline{+\ 11101001} & & \underline{+\ 10111111} & & \underline{+\ 01110001} & & \underline{+\ 00001111}
\end{array}$$

2.13 How many errors can be detected by a code with minimum distance d+1?

2.14 What is the minimum number of parity bits required to obtain a distance-4, two-dimensional code with n information bits?

2.15 Why is it that U.S. computer engineers sometimes confuse the dates of Christmas and Halloween?

2.16 What 60s rock group had the lucky number 64,180?

2.17 The author grew up in ZIP code 60453, and missed his calling by a power of 10. How?

2.18 Here's a problem that lets you have a ball. What is the hexadecimal equivalent of 724174_{10}?

2.19 Find a number that is a palindrome (reads the same forwards and backwards) in binary, octal, *and* hexadecimal.

2.20 List the code words of a 4-bit Gray code.

2.21 How many code words are there in a 2-out-of-5 code? List them.

2.22 Based on the formula for the value of the binomial coefficient that gives the number of code words in an *m*-out-of-*n* code, and you can see that the number of code words in an (*n*–*m*)-out-of-*n* code is exactly the same. But can you ignore the math and give a simple explanation in words of why this must be true?

2.23 Perform a Web search to determine where the "mark" in phone parlance and in the AMI code originated.

2.24 *The Magic Mind Reader*. Make a copy of Figure X2.24 and cut it into six individual slips. Ask a friend to pick one of the slips, secretly pick a number from it, and hand the slip to you. Then ask your friend to look at the remaining slips and hand you any of them that contain the chosen number. Quickly add up the numbers in the top left corners of all the slips you were handed and tell your friend the sum—it is the chosen number! Explain how the trick works.

1 3 5 7 9 11 13 15
17 19 21 23 25 27 29 31
33 35 37 39 41 43 45 47
49 51 53 55 57 59 61 63
	2 3 6 7 10 11 14 15
	18 19 22 23 26 27 30 31
	34 35 38 39 42 43 46 47
	50 51 54 55 58 59 62 63
		4 5 6 7 12 13 14 15
		20 21 22 23 28 29 30 31
		36 37 38 39 44 45 46 47
		52 53 54 55 60 61 62 63

8 9 10 11 12 13 14 15
24 25 26 27 28 29 30 31
40 41 42 43 44 45 46 47
56 57 58 59 60 61 62 63
	16 17 18 19 20 21 22 23
	24 25 26 27 28 29 30 31
	48 49 50 51 52 53 54 55
	56 57 58 59 60 61 62 63
		32 33 34 35 36 37 38 39
		40 41 42 43 44 45 46 47
		48 49 50 51 52 53 54 55
		56 57 58 59 60 61 62 63

Figure X2.24

Exercises

2.25 Make a new version of a Magic Mind Reader based on Drill 2.24. This version should have eight slips each containing *fewer* than 32 integers from 1 to 80 (but the same number of integers on each slip). You may wish to write a program in your favorite programming language to print out the slips.

2.26 Make a new version of a Magic Mind Reader based on Drill 2.24. This version should have nine slips each containing only 16 integers from 1 to 63. You may wish to write a program in your favorite programming language to print out the slips.

2.27 Find an 8-bit binary number that has the same negative value when interpreted as either a signed-magnitude or a two's-complement number.

2.28 The first manned expedition to Mars found only the ruins of a civilization. From the artifacts and pictures, the explorers deduced that the creatures who produced this civilization were four-legged beings with a tentacle that branched out at the end with a number of grasping "fingers." After much study, the explorers were able to translate Martian mathematics. They found the following equation:

$$5x^2 - 50x + 125 = 0$$

with the indicated solutions $x = 5$ and $x = 8$. The value $x = 5$ seemed legitimate enough, but $x = 8$ required some explanation. Then the explorers reflected on the way in which Earth's number system developed, and found evidence that the Martian system had a similar history. How many fingers would you say the Martians had? (From *The Bent of Tau Beta Pi*, February 1956.)

2.29 Each of the following arithmetic operations is correct in at least one number system. Determine possible radices of the numbers in each operation.

(a) $1234 + 4321 = 5555$

(b) $51 / 3 = 15$

(c) $44/4 = 11$

(d) $23 + 44 + 14 + 32 = 201$

(e) $315/24 = 10.2$

(f) $\sqrt{51} = 6$

2.30 Suppose a $4n$-bit number B is represented by an n-digit hexadecimal number H. Prove that the two's complement of B is represented by the 16's complement of H. Make and prove true a similar statement for octal representation.

2.31 Repeat Exercise 2.30 using the ones' complement of B and the 15s' complement of H.

2.32 Given an integer x in the range $-2^{n-1} \leq x \leq 2^{n-1} - 1$, we define $[x]$ to be the two's-complement representation of x, expressed as a positive number: $[x] = x$ if $x \geq 0$ and $[x] = 2^n - |x|$ if $x < 0$, where $|x|$ is the absolute value of x. Let y be another integer in the same range as x. Prove that the two's-complement addition rules given in Section 2.6 are correct by proving that the following equation is always true:

$$[x + y] = [x] + [y] \text{ modulo } 2^n$$

(*Hints:* Consider four cases based on the signs of x and y. Without loss of generality, you may assume that $|x| \geq |y|$.)

2.33 Repeat Exercise 2.32, this time using appropriate expressions and rules for ones'-complement addition.

2.34 State an overflow rule for addition of two's-complement numbers in terms of counting operations in the modular representation of Figure 2-3.

2.35 Show that a two's-complement number can be converted to a representation with more bits by *sign extension*. That is, given an n-bit two's-complement number X, show that the m-bit two's-complement representation of X, where $m > n$, can be obtained by appending $m - n$ copies of X's sign bit to the left of the n-bit representation of X.

2.36 Show that a two's-complement number can be converted to a representation with fewer bits by removing higher-order bits. That is, given an n-bit two's-complement number X, show that the m-bit two's-complement number Y obtained by discarding the d leftmost bits of X represents the same number as X if and only if the discarded bits all equal the sign bit of Y.

2.37 Why is the punctuation of "two's complement" and "ones' complement" inconsistent? (See the first citation in the References.)

2.38 A n-bit binary adder can be used to perform an n-bit unsigned subtraction operation $X - Y$, by performing the operation $X + Y + 1$, where X and Y are n-bit unsigned numbers and Y represents the bit-by-bit complement of Y. Demonstrate this fact as follows. First, prove that $(X - Y) = (X + Y + 1) - 2^n$. Second, prove that the carry out of the n-bit adder is the opposite of the borrow from the n-bit subtraction. That is, show that the operation $X - Y$ produces a borrow out of the MSB position if and only if the operation $X + \overline{Y} + 1$ *does not* produce a carry out of the MSB position.

2.39 In most cases, the product of two *n*-bit two's-complement numbers requires fewer than 2*n* bits to represent it. In fact, there is only one case in which 2*n* bits are needed—find it.

2.40 Prove that a two's-complement number can be multiplied by 2 by shifting it one bit position to the left, with a carry of 0 into the least significant bit position and disregarding any carry out of the most significant bit position, assuming no overflow. State the rule for detecting overflow.

2.41 State and prove correct a technique similar to the one described in Exercise 2.40, for multiplying a ones'-complement number by 2.

2.42 Show how to subtract BCD numbers, by stating the rules for generating borrows and applying a correction factor. Show how your rules apply to each of the following subtractions: 8 – 3, 4 – 8, 5 – 9, 2 – 7.

2.43 How many different 3-bit binary state encodings are possible in a controller with 4 states? How many are possible with 6 states? How many are possible with 8 states?

2.44 Your pointy-haired boss says every state encoding has to contain at least one 0, because it "saves power." So how many different management-approved 3-bit binary state encodings are possible for the traffic-light controller of Table 2-10? How many 4-bit encodings if two 0s are required in each state's encoding?

2.45 List all of the "bad" boundaries in the mechanical encoding disk of Figure 2-5, where an incorrect position may be sensed.

2.46 As a function of *n*, how many "bad" boundaries are there in a mechanical encoding disk that uses an *n*-bit binary code?

2.47 A manufacturer of mechanical encoders discovers the 2-bit Gray code and manufactures encoders with the decimal sequence 0, 1, 3, 2. Generalizing to *n*-bit encoders, they decide all they need to do is to reverse every other pair of values in the decimal sequence, resulting in a sequence of 0, 1, 3, 2, 4, 5, 7, 6, 8, 9, etc. But this proves to be less than perfect. As a function of *n*, how many "bad" boundaries are there? How much better is their code than an *n*-bit binary code?

2.48 On-board altitude transponders on commercial and private aircraft use Gray code to encode the altitude readings that are transmitted to air traffic controllers. Why?

2.49 An incandescent light bulb is stressed every time it is turned on, so in some applications the lifetime of the bulb is limited by the number of on/off cycles rather than the total time it is illuminated. Use your knowledge of codes to suggest a way to double the lifetime of 3-way bulbs in such applications.

2.50 The 5-byte sequence `0x44`, `0x27`, `0x6F`, `0x68`, `0x21` occurs repeatedly in a certain computer file. Why?

2.51 Find a way to draw a 3-cube on a sheet of paper (or other two-dimensional object) so none of the lines cross, or prove that it's impossible.

2.52 Find a way to construct a 4-cube so none of the lines cross, or prove that it's impossible.

2.53 Is the statement in the box on page 70 true in ASCII?

2.54 Define parity groups for a distance-3 Hamming code with 11 information bits.

2.55 Write the code words of a Hamming code with one information bit.

2.56 A certain 64-bit computer's memory system uses 72-bit-wide memory modules. Describe in some detail the extra feature that this memory system can provide.

2.57 Exhibit the pattern for a 3-bit error that is not detected if the "corner" parity bits are not included in the two-dimensional codes of Figure 2-14.

rate of a code

2.58 The *rate of a code* is the ratio of the number of information bits to the total number of bits in a code word. High rates, approaching 1, are desirable for efficient transmission of information. Construct a graph comparing the rates of distance-2 parity codes and distance-3 and -4 Hamming codes for up to 100 information bits.

2.59 Which type of distance-4 code has a higher rate: a two-dimensional code or a Hamming code? Support your answer with a table in the style of Table 2-13, including the rate as well as the number of parity and information bits of each code for up to 100 information bits.

2.60 Show how to construct a distance-6 code with eight information bits.

2.61 Show how to generalize your solution to Exercise 2.60 to create a distance-6 code with an arbitrarily large number of information bits. What is the maximum rate of this code as the number of information bits approaches infinity?

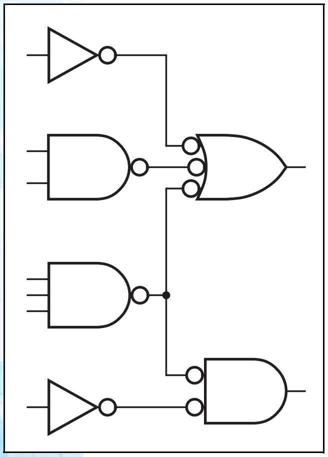
2.62 Describe the operations that must be performed in a RAID system to write new data into information block *b* in drive *d*, so the data can be recovered in the event of an error in block *b* in any drive. Minimize the number of disk accesses required.

2.63 The headers of IPv4 packets on the Internet contain a 16-bit ones'-complement sum of all of the 16-bit words in the header. This header checksum detects almost all of the possible errors in any single 16-bit word in the header. Describe the two errors that it does not detect.

2.64 The header checksum in IPv4 packets uses a 16-bit ones'-complement sum rather than a two's complement sum because the former can be computed in about half or a fourth as many ones'-complement additions as the latter on a processor with 32-bit or 64-bit arithmetic, respectively. Explain why this is so; you'll have to do some online research for the answer.

2.65 In the style of Figure 2-17, draw the waveforms for the bit pattern 01010001 when sent serially using the NRZ, NRZI, RZ, BPRZ, and Manchester codes, assuming that the bits are transmitted in order from left to right.

2.66 What serial line code is used in a single lane of the first two versions of the PCI Express serial interface (PCIe 1.0 and 2.0)? Search the Web for the answer.



chapter 3

Switching Algebra and Combinational Logic

You have undoubtedly already used various logic expressions when forming conditional statements in a programming language. There, variables and relations are combined in a somewhat ad-hoc manner in parenthesized expressions to make decisions and to control actions.

Digital logic design employs logic expressions that are often much more complex than those found in a typical program. Moreover, in logic design such expressions usually lead to a corresponding hardware implementation, a logic circuit whose output is the value obtained by evaluating and combining the inputs as specified by the expression.

Logic expressions are therefore often manipulated—by a human or more typically nowadays by an EDA tool—to achieve various circuit-design goals. Such goals may include adapting an expression to an available circuit structure or optimizing a circuit's size or its performance in metrics like speed and power consumption. To create, understand, and manipulate logic expressions and circuits, digital hardware designers use *switching algebra* as a fundamental tool.

Logic circuits are classified into two types, "combinational" and "sequential." A *combinational logic circuit* is one whose outputs depend only on its current inputs. The fan-speed selection knob in an older car is like a combinational circuit. Its "output" selects a speed based only on its current "input"—the position of the knob.

The outputs of a *sequential logic circuit* depend not only on the current inputs but also on the past sequence of inputs, possibly arbitrarily far back in time. The fan-speed circuit controlled by up and down buttons in a newer car is a sequential circuit—the current speed depends on an arbitrarily long sequence of up/down pushes, beginning when you first turned on the fan.

This chapter focuses on combinational logic circuits, exploring switching algebra, logic expressions, and the analysis and synthesis of combinational logic circuits at the gate level. Sequential circuits will be discussed in later chapters.

feedback loop

A combinational circuit may contain an arbitrary number of logic gates and inverters but no feedback loops. A *feedback loop* is a signal path of a circuit that allows the output of a gate to propagate back to the input of that same gate. Such a loop generally creates memory and results in sequential circuit behavior, as we'll show in the later chapters.

In combinational circuit *analysis*, we start with a gate-level logic diagram and proceed to a formal description of the function performed by that circuit, like a truth table or a logic expression. In *synthesis*, we do the reverse, starting with a formal description and proceeding to a logic diagram or other description that will allow us to build the circuit using available components.

Combinational circuits may have one or more outputs. In this chapter, we'll discuss methods that apply to single-output circuits. Most analysis and synthesis techniques can be extended in an obvious way from single-output to multiple-output circuits (e.g., "Repeat these steps for each output"). Some techniques can be extended to improve effectiveness for multiple outputs.

The purpose of this chapter is to give you a solid theoretical foundation for the analysis and synthesis of combinational logic circuits, a foundation that will be doubly important later when we move on to sequential circuits. Although most analysis and synthesis procedures are automated nowadays by EDA tools, a basic understanding of the fundamentals can help you use the tools as well as understand what's gone wrong when they give you undesirable results.

WHAT IS SYNTHESIS?

In Chapter 1, we introduced the concept of HDL-based digital design using EDA tools. In that approach, we can write an HDL model to precisely specify a combinational logic function in any of a variety of ways, and then use an EDA synthesis tool to realize the function in a selected implementation technology, as we'll describe in more detail in Section 4.3.

synthesis

In the present chapter, *synthesis* has a narrower meaning. We again start with a precise specification of a combinational logic function, but only in the form of a logic equation, truth table, or equivalent. And we target only one implementation technology, a gate-level circuit that realizes the logic function. This is traditional logic design, where minimizing the number of gates needed to perform the function is the key goal. For most other implementation technologies, experience has proven that's still a good starting point for subsequent technology-specific optimization.

DESIGN VS. SYNTHESIS

Digital logic *design* is a superset of synthesis, since in a real design problem we start out with an informal description of the circuit's function, typically using natural language and perhaps pseudo-code to describe its behavior. For a combinational circuit, an informal description must at least name all of the circuit's inputs and outputs, and indicate how each output is affected by the inputs.

To formalize the circuit description, we need something that precisely defines the circuit's behavior for all situations; for a combinational circuit, this means the output values produced for all possible input combinations. Formal descriptions for combinational circuits include truth tables, logic equations, and models created using an HDL.

Once we have a formal circuit description, we can follow a "turn-the-crank" synthesis procedure to obtain a circuit with the specified functional behavior. The circuit may be described in a logic diagram that shows its elements (such as gates) and their interconnections, in a net list which is a text file conveying the same information as the latter, or in one of a myriad of formats which specify a circuit's elements and interconnections in a particular implementation technology such as an ASIC or FPGA.

The material in the first three sections of this chapter is the basis for "turn-the-crank" synthesis procedures for creating combinational logic circuits using discrete gates, whether the crank is turned by hand or by a computer.

Before launching into a discussion of combinational logic circuits, we'll introduce switching algebra, the fundamental mathematical tool for analyzing and synthesizing logic circuits of all types.

3.1 Switching Algebra

Formal analysis techniques for digital circuits have their roots in the work of an English mathematician, George Boole. In 1854, he invented a two-valued algebraic system, now called *Boolean algebra*, to "give expression … to the fundamental laws of reasoning in the symbolic language of a Calculus." Using this system, a philosopher, logician, or inhabitant of the planet Vulcan can formulate propositions that are true or false, combine them to make new propositions, and determine the truth or falsehood of the new propositions. For example, if we agree that "People who haven't studied this material are either failures or not nerds," and "No computer designer is a failure," then we can answer questions like "If you're a nerdy computer designer, then have you already studied this?"

Boolean algebra

Long after Boole, in 1938, Bell Labs researcher Claude E. Shannon showed how to adapt Boolean algebra to analyze and describe the behavior of circuits built from relays, the most commonly used digital logic elements of that time. In Shannon's *switching algebra*, the condition of a relay contact, open or

switching algebra

closed, is represented by a variable X that can have one of two possible values, 0 or 1. In today's logic technologies, these values correspond to a wide variety of physical conditions—voltage HIGH or LOW, light off or on, capacitor discharged or charged, electrons trapped or released, and so on.

In the remainder of this section, we will develop switching algebra directly, using "first principles" and what you may already know about the behavior of logic elements (gates and inverters). For more historical and/or mathematical treatments of this material, consult the References section of this chapter.

3.1.1 Axioms

In switching algebra we use a symbolic variable, such as X, to represent the condition of a logic signal. A logic signal is in one of two possible conditions—low or high, off or on, and so on, depending on the technology. We say that X has the value "0" for one of these conditions and "1" for the other.

positive-logic convention

negative-logic convention

For example, with CMOS and most other logic circuits the *positive-logic convention* dictates that we associate the value "0" with a LOW voltage and "1" with a HIGH voltage. The *negative-logic convention*, which is rarely used, makes the opposite association: 0 = HIGH and 1 = LOW. However, the choice of positive or negative logic has no effect on our ability to develop a consistent algebraic description of circuit behavior; it only affects details of the physical-to-algebraic abstraction. Thus, we may generally ignore the physical realities of logic circuits and pretend that they operate directly on the logic symbols 0 and 1.

axiom

postulate

The *axioms* (or *postulates*) of a mathematical system are a minimal set of basic definitions that we assume to be true, from which all other information about the system can be derived. The first two axioms of switching algebra embody the "digital abstraction" by formally stating that a variable X can take on only one of two values:

$$\text{(A1)}\quad X = 0 \quad \text{if } X \neq 1 \qquad \text{(A1D)}\quad X = 1 \quad \text{if } X \neq 0$$

Notice that we stated these axioms as a pair, the only difference between A1 and A1D being the interchange of the symbols 0 and 1. This is a characteristic of all the axioms of switching algebra and is the basis of the "duality" principle that we'll study later.

inverter

complement

prime (′)

An *inverter* is a logic circuit whose output signal level is the opposite (or *complement*) of its input signal level. We use a *prime (′)* to denote an inverter's function. That is, if a variable X denotes an inverter's input signal, then X′ denotes the value of a signal on the inverter's output. This notation is formally specified in the second pair of axioms:

$$\text{(A2)}\quad \text{If } X = 0, \text{ then } X' = 1 \qquad \text{(A2D)}\quad \text{If } X = 1, \text{ then } X' = 0$$

Figure 3-1 shows the logic symbol for an inverter. with its input on the left and its output on the right. The input and output signals may have arbitrary names, say X and Y. Algebraically, however, we write Y = X′ to say "signal Y has

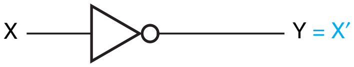


Figure 3-1 Signal naming and algebraic notation for an inverter.

the opposite value as signal X." The prime (′) is an *algebraic operator*, and X′ is an *expression*, which you can read as "X prime" or "NOT X." This usage is analogous to what you've learned in programming languages, where if J is an integer variable, then −J is an expression whose value is 0 − J. Although this may seem like a small point, you'll learn that the distinction between signal names (X, Y), expressions (X′), and equations (Y = X′) is very important when we study documentation standards and software tools for logic design. In the logic diagrams in this book, we maintain this distinction by writing signal names in black and expressions in color.

algebraic operator
expression
NOT operation

A 2-input AND gate is a circuit whose output is 1 if *both* of its inputs are 1, and it has the symbol in Figure 3-2(a). The function of a 2-input AND gate is sometimes called *logical multiplication* and is symbolized algebraically by a *multiplication dot (·)*. That is, an AND gate with inputs X and Y has an output signal whose value is X · Y, as shown in Figure 3-2(a). Some authors, especially mathematicians and logicians, denote logical multiplication with a wedge (X ∧ Y). We follow standard engineering practice here by using the dot (X · Y). Verilog uses an ampersand (&) to denote the same thing.

logical multiplication
multiplication dot (·)

A 2-input OR gate is a circuit whose output is 1 if *either* of its inputs is 1 and has the symbol in Figure 3-2(b). The function of a 2-input OR gate is sometimes called *logical addition* and is symbolized algebraically by a plus sign (+). An OR gate with inputs X and Y has an output signal whose value is X + Y, as shown in the figure. Some authors denote logical addition with a vee (X ∨ Y), but we follow the typical engineering practice of using the plus sign (X + Y). Once again, other notations are used in HDLs, like "|" in Verilog.

logical addition

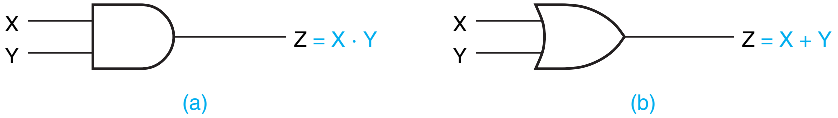


Figure 3-2 Signal naming and algebraic notation: (a) AND gate; (b) OR gate.

NOTE ON NOTATION

The notations $\overline{\mathrm{X}}$, ~X, and ¬X are also used by some authors to denote the complement of X. The overbar notation ($\overline{\mathrm{X}}$) is probably the most widely used and the best looking typographically. However, we use the prime notation to get you used to writing logic expressions on a single text line without the more graphical overbar, and to force you to parenthesize complex complemented subexpressions—because that's what you'll have to do when you use HDLs like Verilog and other tools.

precedence

By convention in this and most texts, as well as by definition in Verilog, multiplication has a higher *precedence* than addition in logical expressions, just as in arithmetic expressions in conventional programming languages. That is, the expression $\mathsf{W} \cdot \mathsf{X} + \mathsf{Y} \cdot \mathsf{Z}$ is equivalent to $(\mathsf{W} \cdot \mathsf{X}) + (\mathsf{Y} \cdot \mathsf{Z})$. But be careful if you ever use VHDL. There, "`and`" and "`or`" have the *same* precedence and are evaluated from left to right. So, "`W and X or Y and Z`" has the same meaning as "`((W and X) or Y) and Z`", not "`(W and X) or (Y and Z)`".

AND operation
OR operation

The last three pairs of axioms state the formal definitions of the AND and OR operations by listing the output produced by each gate for each possible input combination:

(A3)	$0 \cdot 0 = 0$	(A3D)	$1 + 1 = 1$
(A4)	$1 \cdot 1 = 1$	(A4D)	$0 + 0 = 0$
(A5)	$0 \cdot 1 = 1 \cdot 0 = 0$	(A5D)	$1 + 0 = 0 + 1 = 1$

The five pairs of axioms, A1–A5 and A1D–A5D, completely define switching algebra. All other facts about the system can be proved using these axioms as a starting point.

3.1.2 Single-Variable Theorems

theorem

During the analysis or synthesis of logic circuits, we can write algebraic expressions that characterize a circuit's actual or desired behavior. Switching-algebra *theorems* are statements, known to be always true, that allow us to manipulate algebraic expressions for simpler analysis or more efficient synthesis of the corresponding circuits. For example, the theorem $\mathsf{X} + 0 = \mathsf{X}$ allows us to substitute every occurrence of $\mathsf{X} + 0$ in an expression with X.

Table 3-1 lists switching-algebra theorems involving a single variable X. How do we know that these theorems are true? We can either prove them ourselves or take the word of someone who has. OK, we're in college now, let's learn how to prove them.

perfect induction

Most theorems in switching algebra are exceedingly simple to prove using a technique called *perfect induction*. Axiom A1 is the key to this technique—

JUXT A MINUTE... Older texts use simple *juxtaposition* (XY) to denote logical multiplication, but we don't. In general, juxtaposition is a clear notation only when signal names are limited to a single character. Otherwise, is XY a logical product or a two-character signal name? One-character variable names are common in algebra, but in real digital design problems we prefer to use multicharacter signal names that mean something. Thus, we need a separator between names, and the separator might just as well be a multiplication dot rather than a space. The HDL equivalent of the multiplication dot (like "`&`" in Verilog) is absolutely required when logic formulas are written in a hardware description language.

Table 3-1 Switching-algebra theorems with one variable.

(T1)	$X + 0 = X$	(T1D)	$X \cdot 1 = X$	(Identities)
(T2)	$X + 1 = 1$	(T2D)	$X \cdot 0 = 0$	(Null elements)
(T3)	$X + X = X$	(T3D	$X \cdot X = X$	(Idempotency)
(T4)	$(X')' = X$			(Involution)
(T5)	$X + X' = 1$	(T5′D)	$X \cdot X' = 0$	(Complements)

since a switching variable can take on only two different values, 0 and 1, we can prove a theorem involving a single variable X by proving that it is true for both $X = 0$ and $X = 1$. For example, to prove theorem T1, we make two substitutions:

$[X = 0]$	$0 + 0 = 0$	true, according to axiom A4D
$[X = 1]$	$1 + 0 = 1$	true, according to axiom A5D

All of the theorems in Table 3-1 can be proved using perfect induction, as you're asked to do in Drills 3.2 and 3.3.

3.1.3 Two- and Three-Variable Theorems

Switching-algebra theorems with two or three variables are listed in Table 3-2. Each of these theorems is easily proved by perfect induction, by evaluating the theorem statement for the four possible combinations of X and Y, or the eight possible combinations of X, Y, and Z.

The first two theorem pairs concern commutativity and associativity of logical addition and multiplication and are identical to the commutative and associative laws you already know for addition and multiplication of integer and real numbers. Taken together, they indicate that the parenthesization or order of terms in a logical sum or logical product is irrelevant. For example, from a strictly algebraic point of view, an expression such as $W \cdot X \cdot Y \cdot Z$ is ambiguous; it should be written as $(W \cdot (X \cdot (Y \cdot Z)))$ or $(((W \cdot X) \cdot Y) \cdot Z)$ or $(W \cdot X) \cdot (Y \cdot Z)$ (see Exercise 3.22). But the theorems tell us that the ambiguous form of the expression is OK because we get the same results in any case. We even could

Table 3-2 Switching-algebra theorems with two or three variables.

(T6)	$X + Y = Y + X$	(T6D)	$X \cdot Y = Y \cdot X$	(Commutativity)
(T7)	$(X + Y) + Z = X + (Y + Z)$	(T7D)	$(X \cdot Y) \cdot Z = X \cdot (Y \cdot Z)$	(Associativity)
(T8)	$X \cdot Y + X \cdot Z = X \cdot (Y + Z)$	(T8D)	$(X + Y) \cdot (X + Z) = X + Y \cdot Z$	(Distributivity)
(T9)	$X + X \cdot Y = X$	(T9D)	$X \cdot (X + Y) = X$	(Covering)
(T10)	$X \cdot Y + X \cdot Y' = X$	(T10D)	$(X + Y) \cdot (X + Y') = X$	(Combining)
(T11)	$X \cdot Y + X' \cdot Z + Y \cdot Z = X \cdot Y + X' \cdot Z$			(Consensus)
(T11′)	$(X + Y) \cdot (X' + Z) \cdot (Y + Z) = (X + Y) \cdot (X' + Z)$			

have rearranged the order of the variables (e.g., $X \cdot Z \cdot Y \cdot W$) and gotten the same results.

As trivial as this discussion may seem, it is very important, because it forms the mathematical basis for using logic gates with more than two inputs. We defined $\cdot$ and + as *binary operators*—operators that combine *two* variables. Yet we use 3-input and larger AND and OR gates in practice. The theorems tell us we can connect gate inputs in any order; in fact, printed-circuit-board and ASIC layout programs take advantage of this principle to optimize wiring. Also, we can use either one n-input gate or $(n-1)$ 2-input gates interchangeably, though cost and timing delay are likely to be higher with multiple 2 input gates.

binary operator

Theorem T8 is identical to the distributive law for integers and reals—that is, logical multiplication distributes over logical addition. As a result, we can "multiply out" an expression to convert it to a sum-of-products form, as in the example below:

$$V \cdot (W + X) \cdot (Y + Z) = V \cdot W \cdot Y + V \cdot W \cdot Z + V \cdot X \cdot Y + V \cdot X \cdot Z$$

However, switching algebra also has the unfamiliar property that the reverse is true—logical addition distributes over logical multiplication—as demonstrated by theorem T8D. Thus, we can also "add out" an expression to obtain a product-of-sums form:

$$(V \cdot W \cdot X) + (Y \cdot Z) = (V + Y) \cdot (V + Z) \cdot (W + Y) \cdot (W + Z) \cdot (X + Y) \cdot (X + Z)$$

Theorems T9 and T10 are used extensively to minimize the number of terms in logic expressions, which minimizes the number of gates or gate inputs in the corresponding logic circuits. For example, if the subexpression $X + X \cdot Y$ appears in a logic expression, the *covering theorem* T9 says that we need only include X in the expression; X is said to *cover* $X \cdot Y$. The *combining theorem* T10 says if the subexpression $X \cdot Y + X \cdot Y'$ appears in an expression, we can replace it with X. Since Y must be 0 or 1, either way the original subexpression is 1 if and only if X is 1.

covering theorem
cover
combining theorem

Although we could easily prove T9 by perfect induction, the truth of T9 may be more obvious if we prove it using the other theorems that we've proved so far:

$$\begin{aligned} X + X \cdot Y &= X \cdot 1 + X \cdot Y && \text{(according to T1D)} \\ &= X \cdot (1 + Y) && \text{(according to T8)} \\ &= X \cdot 1 && \text{(according to T2)} \\ &= X && \text{(according to T1D)} \end{aligned}$$

Likewise, the other theorems can be used to prove T10, where the key step is to use T8 to rewrite the lefthand side as $X \cdot (Y + Y')$.

Theorem T11 is known as the *consensus theorem*. The $Y \cdot Z$ term is called the *consensus* of $X \cdot Y$ and $X' \cdot Z$. The idea is that if $Y \cdot Z$ is 1, then either $X \cdot Y$ or $X' \cdot Z$ must also be 1, since Y and Z are both 1 and either X or X′ must be 1. Thus, the $Y \cdot Z$ term is redundant and may be dropped from the righthand side of T11.

consensus theorem
consensus

The consensus theorem has two important applications. It can be used to eliminate certain timing hazards in combinational logic circuits, as we'll see in Section 3.4. And it also forms the basis of the iterative-consensus method used in logic-minimization programs to find "prime implicants" (see References).

In all of the theorems, it is possible to replace each variable with an arbitrary logic expression. A simple replacement is to complement one or more variables:

$$(\mathsf{X} + \mathsf{Y}') + \mathsf{Z}' \;=\; \mathsf{X} + (\mathsf{Y}' + \mathsf{Z}') \qquad \text{(based on T7)}$$

But more complex expressions may be substituted as well:

$$(\mathsf{V}' + \mathsf{X}) \cdot (\mathsf{W} \cdot (\mathsf{Y}' + \mathsf{Z})) + (\mathsf{V}' + \mathsf{X}) \cdot (\mathsf{W} \cdot (\mathsf{Y}' + \mathsf{Z}))' \;=\; \mathsf{V}' + \mathsf{X} \qquad \text{(based on T10)}$$

3.1.4 *n*-Variable Theorems

Several important theorems, listed in Table 3-3, are true for an arbitrary number of variables, n. Most of these theorems can be proved using a two-step method called *finite induction*—first proving that the theorem is true for $n = 2$ (the *basis step*), and then proving that if the theorem is true for $n = i$, then it is also true for $n = i + 1$ (the *induction step*). For example, consider the generalized idempotency theorem T12. For $n = 2$, T12 is equivalent to T3 and is therefore true. If it is true for a logical sum of i X's, then it is also true for a sum of $i + 1$ X's, according to the following reasoning:

finite induction
basis step
induction step

$$\begin{aligned}
\mathsf{X} + \mathsf{X} + \mathsf{X} + \cdots + \mathsf{X} &= \mathsf{X} + (\mathsf{X} + \mathsf{X} + \cdots + \mathsf{X}) && (i + 1 \text{ X's on either side})\\
&= \mathsf{X} + (\mathsf{X}) && (\text{if T12 is true for } n = i)\\
&= \mathsf{X} && (\text{according to T3})
\end{aligned}$$

Thus, the theorem is true for all finite values of n.

DeMorgan's theorems (T13 and T13D) are probably the most commonly used of all the theorems of switching algebra. Theorem T13 says that an n-input AND gate whose output is complemented is equivalent to an n-input OR gate whose inputs are complemented. That is, the circuits of Figure 3-3(a) and (b) are equivalent.

DeMorgan's theorems

Table 3-3 Switching-algebra theorems with *n* variables.

(T12)	$\mathsf{X} + \mathsf{X} + \cdots + \mathsf{X} = \mathsf{X}$	(Generalized idempotency)
(T12D)	$\mathsf{X} \cdot \mathsf{X} \cdot \;\cdots\; \cdot \mathsf{X} = \mathsf{X}$	
(T13)	$(\mathsf{X}_1 \cdot \mathsf{X}_2 \cdot \;\cdots\; \cdot \mathsf{X}_n)' = \mathsf{X}_1' + \mathsf{X}_2' + \cdots + \mathsf{X}_n'$	(DeMorgan's theorems)
(T13D	$(\mathsf{X}_1 + \mathsf{X}_2 + \cdots + \mathsf{X}_n)' = \mathsf{X}_1' \cdot \mathsf{X}_2' \cdot \;\cdots\; \cdot \mathsf{X}_n'$	
(T14)	$[\mathsf{F}(\mathsf{X}_1, \mathsf{X}_2, \ldots, \mathsf{X}_n, +, \cdot)]' = \mathsf{F}(\mathsf{X}_1', \mathsf{X}_2', \ldots, \mathsf{X}_n', \cdot, +)$	(Generalized DeMorgan's theorem)
(T15)	$\mathsf{F}(\mathsf{X}_1, \mathsf{X}_2, \ldots, \mathsf{X}_n) = \mathsf{X}_1 \cdot \mathsf{F}(1, \mathsf{X}_2, \ldots, \mathsf{X}_n) + \mathsf{X}_1' \cdot \mathsf{F}(0, \mathsf{X}_2, \ldots, \mathsf{X}_n)$	(Shannon's expansion theorems)
(T15D)	$\mathsf{F}(\mathsf{X}_1, \mathsf{X}_2, \ldots, \mathsf{X}_n) = [\mathsf{X}_1 + \mathsf{F}(0, \mathsf{X}_2, \ldots, \mathsf{X}_n)] \cdot [\mathsf{X}_1' + \mathsf{F}(1, \mathsf{X}_2, \ldots, \mathsf{X}_n)]$	

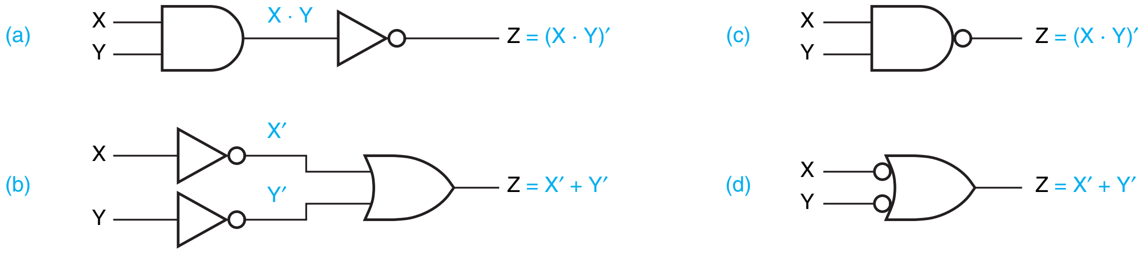


Figure 3-3 Equivalent circuits according to DeMorgan's theorem T13: (a) AND-NOT; (b) NOT-OR; (c) logic symbol for a NAND gate; (d) equivalent symbol for a NAND gate.

A NAND gate is like an AND gate, but with its output complemented, and thus can have the logic symbol in Figure 3-3(c). However, a typical CMOS NAND-gate circuit typically is not designed at the transistor level as an AND gate followed by a transistor inverter (NOT gate); it's just a collection of transistors that happens to perform the AND-NOT function. In fact, theorem T13 tells us that the logic symbol in Figure 3-3(d) denotes the same logic function (bubbles on the OR-gate inputs indicate logical inversion). That is, a NAND gate may be considered to perform a NOT-OR function.

By observing the inputs and output of a NAND gate, it is impossible to determine whether it has been built internally as an AND gate followed by an inverter, as inverters followed by an OR gate, or as a direct CMOS realization, because all NAND gates perform precisely the same logic function. Although the choice of symbol has no bearing on a gate's functionality, the proper symbol choice in documentation for a larger circuit incorporating the gate can make the larger circuit easier to understand, as we'll see in later chapters.

Another symbolic equivalence can be inferred from theorem T13D. As shown in Figure 3-4, a NOR gate may be realized as an OR gate followed by an inverter, or as inverters followed by an AND gate. Once again, the choice of one or the other of the equivalent logic symbols can make a big difference in the understandability of a larger circuit.

generalized DeMorgan's theorem

Theorems T13 and T13D happen to be just special cases of a *generalized DeMorgan's theorem*, T14, that applies to an arbitrary logic expression F. By

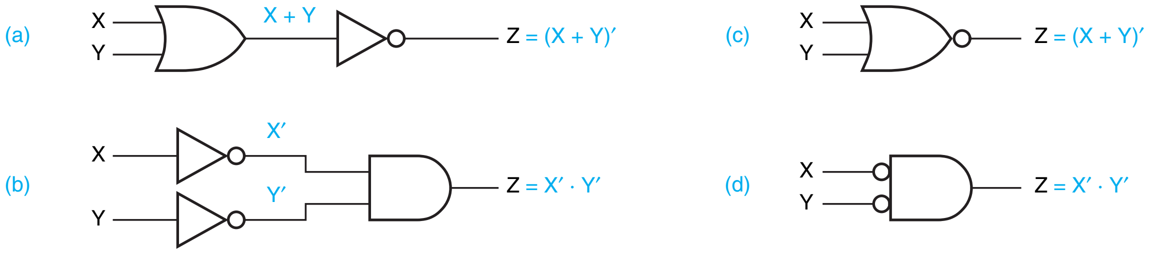


Figure 3-4 Equivalent circuits according to DeMorgan's theorem T13D: (a) OR-NOT; (b) NOT-AND; (c) logic symbol for a NOR gate; (d) equivalent symbol for a NOR gate.

definition, the *complement of a logic expression*, written as $(\mathsf{F})'$, is an expression whose value is the opposite of F's for all possible input combinations. Theorem T14 is very important because it gives us a way to manipulate and simplify the complement of an expression.

complement of a logic expression

Theorem T14 states that, given any *n*-variable logic expression, its complement can be obtained by swapping the + and · operators and complementing all variables. For example, suppose that we have

$$\begin{aligned} \mathsf{F(W,X,Y,Z)} &= \mathsf{(W'\cdot X)+(X\cdot Y)+(W\cdot(X'+Z'))} \\ &= \mathsf{((W)'\cdot X)+(X\cdot Y)+(W\cdot((X)'+(Z)'))} \end{aligned}$$

In the second line, we have enclosed complemented variables in parentheses to remind you that the ′ is an operator, not part of the variable name. By applying theorem T14, we obtain

$$\mathsf{[F(W,X,Y,Z)]'} = \mathsf{((W')'+X')\cdot(X'+Y')\cdot(W'+((X')'\cdot(Z')'))}$$

Using theorem T4, this can be simplified to

$$\mathsf{[F(W,X,Y,Z)]'} = \mathsf{(W+X')\cdot(X'+Y')\cdot(W'+(X\cdot Z))}$$

In general, we can use theorem T14 to complement a parenthesized expression by swapping + and ·, complementing all uncomplemented variables, and uncomplementing all complemented ones.

The generalized DeMorgan's theorem T14 can be proved by showing that all logic functions can be written as either a sum or a product of subfunctions, and then applying T13 and T13D recursively. Also, an enlightening and satisfying proof is presented in previous editions of this book, based on the principle of duality which we explain in the next subsection.

Shannon's expansion theorems T15 and T15D are very important for their use in FPGAs to implement arbitrary combinational logic functions. An FPGA contains many instances of a basic resource called a lookup table (LUT) that can realize any combinational logic function of up to a certain number of inputs, on the order of 6. What if you need a 7-input function? Shannon's theorems tell you how to combine the outputs of two 6-input LUTs to realize any 7-input function. Similarly, 8-input functions can be implemented by combining 7-input functions realized this way (with 4 LUTs total), and so on. Logic synthesizers for FPGAs do this automatically, as discussed in Section 6.1.3 on page 244.

3.1.5 Duality

We stated all of the axioms of switching algebra in pairs. The *dual* of each axiom (e.g., A5D) is obtained from the base axiom (e.g., A5) by simply swapping 0 and 1 and, if present, · and +. As a result, we can state the following metatheorem (a *metatheorem* is a theorem about theorems):

dual

metatheorem

Principle of Duality Any theorem or identity in switching algebra is also true if 0 and 1 are swapped and · and + are swapped throughout.

The metatheorem is true because the duals of all the axioms are true, so duals of all switching-algebra theorems can be proved using duals of the axioms.

After all, what's in a name, or in a symbol for that matter? If the software that was used to format this book had a bug, one that swapped $0 \leftrightarrow 1$ and $\cdot \leftrightarrow +$ throughout this chapter, you still would have learned exactly the same switching algebra; only the nomenclature would have been a little weird, using words like "product" to describe an operation that uses the symbol "+".

Duality is important because it doubles the usefulness of almost everything that you know about switching algebra and manipulation of switching functions. This statement applies not only to you, but also to automated tools that manipulate logic functions and synthesize circuits that perform them. For example, if a software tool can derive a sum-of-products expression from an arbitrary combinational logic function defined in an HDL model, and synthesize a corresponding two-stage AND-OR logic circuit from that expression, then with relatively little effort, it can be adapted also to derive a product-of-sums expression and synthesize a corresponding OR-AND circuit for the same logic function. We explore this idea in Exercise 3.41.

There is just one convention in switching algebra where we did not treat $\cdot$ and + identically, so duality does not necessarily hold true—can you remember what it is before reading the answer below? Consider the following statement of theorem T9 and its clearly absurd "dual":

$$\mathsf{X} + \mathsf{X} \cdot \mathsf{Y} = \mathsf{X} \qquad \text{(theorem T9)}$$

$$\mathsf{X} \cdot \mathsf{X} + \mathsf{Y} = \mathsf{X} \qquad \text{(after applying the principle of duality)}$$

$$\mathsf{X} + \mathsf{Y} = \mathsf{X} \qquad \text{(after applying theorem T3D)}$$

Obviously the last line above is false—where did we go wrong? The problem is in operator precedence. We were able to write the lefthand side of the first line without parentheses because of our convention that $\cdot$ has precedence. However, once we applied the principle of duality, we should have given precedence to + instead, or written the second line as $\mathsf{X} \cdot (\mathsf{X} + \mathsf{Y}) = \mathsf{X}$. The best way to avoid problems like this is to parenthesize an expression fully before taking its dual.

3.1.6 Standard Representations of Logic Functions

Before moving on to analysis and synthesis of combinational logic functions, we'll introduce some needed nomenclature and notation.

truth table

The most basic representation of a logic function is the *truth table*. Similar in approach to the perfect-induction proof method, this brute-force representation simply lists the output of the circuit for every possible input combination. Traditionally, the input combinations are arranged in rows in ascending binary counting order, and the corresponding output values are written in a column next to the rows. For example, the general structure of a 3-variable truth table is

Row	X	Y	Z	F
0	0	0	0	F(0,0,0)
1	0	0	1	F(0,0,1)
2	0	1	0	F(0,1,0)
3	0	1	1	F(0,1,1)
4	1	0	0	F(1,0,0)
5	1	0	1	F(1,0,1)
6	1	1	0	F(1,1,0)
7	1	1	1	F(1,1,1)

Table 3-4
General truth table structure for a 3-variable logic function, F(X,Y,Z).

shown in Table 3-4. The rows of the table are numbered 0–7, corresponding to the binary input combinations, but this numbering is not a necessary part of the truth table.

The truth table for a particular 3-variable logic function is shown in Table 3-5. Each distinct pattern of eight 0s and 1s in the output column yields a different logic function; there are 2^8 such patterns. Thus, the logic function in Table 3-5 is one of 2^8 different logic functions of three variables.

The truth table for an n-variable logic function has 2^n rows. Obviously, truth tables are practical to write only for logic functions with a small number of variables, say, 10 for students and about 4–5 for everyone else.

The information contained in a truth table can also be conveyed algebraically. To do so, we first need some definitions:

- A *literal* is a variable or the complement of a variable. Examples: X, Y, X′, Y′. *literal*
- A *product term* is a single literal or a logical product of two or more literals. Examples: Z', $W \cdot X \cdot Y$, $X \cdot Y' \cdot Z$, $W' \cdot Y' \cdot Z$. *product term*

Row	X	Y	Z	F
0	0	0	0	1
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	1
5	1	0	1	0
6	1	1	0	1
7	1	1	1	1

Table 3-5
Truth table for a particular 3-variable logic function, F(X,Y,Z).

sum-of-products expression

- A *sum-of-products expression* is a logical sum of product terms. Example: $Z' + W \cdot X \cdot Y + X \cdot Y' \cdot Z + W' \cdot Y' \cdot Z$.

sum term

- A *sum term* is a single literal or a logical sum of two or more literals. Examples: Z', $W + X + Y$, $X + Y' + Z$, $W' + Y' + Z$.

product-of-sums expression

- A *product-of-sums expression* is a logical product of sum terms. Example: $Z' \cdot (W + X + Y) \cdot (X + Y' + Z) \cdot (W' + Y' + Z)$.

normal term

- A *normal term* is a product or sum term in which no variable appears more than once. A nonnormal term can always be simplified to a constant or a normal term using one of theorems T3, T3′, T5, or T5′. Examples of nonnormal terms: $W \cdot X \cdot X \cdot Y'$, $W + W + X' + Y$, $X \cdot X' \cdot Y$. Examples of normal terms: $W \cdot X \cdot Y'$, $W + X' + Y$.

minterm

- An n-variable *minterm* is a normal product term with n literals. There are 2^n such product terms. Some examples of 4-variable minterms: $W' \cdot X' \cdot Y' \cdot Z'$, $W \cdot X \cdot Y' \cdot Z$, $W' \cdot X' \cdot Y \cdot Z'$.

maxterm

- An n-variable *maxterm* is a normal sum term with n literals. There are 2^n such sum terms. Examples of 4-variable maxterms: $W' + X' + Y' + Z'$, $W + X' + Y' + Z$, $W' + X' + Y + Z'$.

There is a close correspondence among the truth table and minterms and maxterms. A minterm can be defined as a product term that is 1 in exactly one row of the truth table. Similarly, a maxterm can be defined as a sum term that is 0 in exactly one row of the truth table. Table 3-6 shows this correspondence for a 3-variable truth table.

minterm number
minterm i

An n-variable minterm can be represented by an n-bit integer, the *minterm number*. We'll use the name *minterm i* to denote the minterm corresponding to row i of the truth table. In minterm i, a particular variable appears complemented if the corresponding bit in the binary representation of i is 0; otherwise, it is uncomplemented. For example, row 5 has binary representation 101, and the corresponding minterm is $X \cdot Y' \cdot Z$. As you might expect, the correspondence for maxterms is just the opposite: in *maxterm i*, a variable is complemented if the

maxterm i

Table 3-6
Minterms and maxterms for a 3-variable logic function, F(X, Y, Z).

Row	X	Y	Z	F	***Minterm***	***Maxterm***
0	0	0	0	F(0,0,0)	$X' \cdot Y' \cdot Z'$	$X + Y + Z$
1	0	0	1	F(0,0,1)	$X' \cdot Y' \cdot Z$	$X + Y + Z'$
2	0	1	0	F(0,1,0)	$X' \cdot Y \cdot Z'$	$X + Y' + Z$
3	0	1	1	F(0,1,1)	$X' \cdot Y \cdot Z$	$X + Y' + Z'$
4	1	0	0	F(1,0,0)	$X \cdot Y' \cdot Z'$	$X' + Y + Z$
5	1	0	1	F(1,0,1)	$X \cdot Y' \cdot Z$	$X' + Y + Z'$
6	1	1	0	F(1,1,0)	$X \cdot Y \cdot Z'$	$X' + Y' + Z$
7	1	1	1	F(1,1,1)	$X \cdot Y \cdot Z$	$X' + Y' + Z'$

corresponding bit in the binary representation of i is 1. Thus, maxterm 5 (101) is $X' + Y + Z'$. Note that all of this makes sense only if we have stated the number of variables, three in the examples.

Based on the correspondence between the truth table and minterms, we can easily create an algebraic representation of a logic function from its truth table. The *canonical sum* of a logic function is a sum of the minterms corresponding to truth-table rows (input combinations) for which the function produces a 1 output. For example, the canonical sum for the logic function in Table 3-5 on page 101 is

canonical sum

$$\begin{aligned} F &= \Sigma_{X,Y,Z}(0,3,4,6,7) \\ &= X' \cdot Y' \cdot Z' + X' \cdot Y \cdot Z + X \cdot Y' \cdot Z' + X \cdot Y \cdot Z' + X \cdot Y \cdot Z \end{aligned}$$

Here, the notation $\Sigma_{X,Y,Z}(0,3,4,6,7)$ is a *minterm list* and means “the sum of minterms 0, 3, 4, 6, and 7 with variables X, Y, and Z.” The minterm list is also known as the *on-set* for the logic function. You can visualize that each minterm “turns on” the output for exactly one input combination. Any logic function can be written as a canonical sum.

minterm list

on-set

The *canonical product* of a logic function is a product of the maxterms corresponding to input combinations for which the function produces a 0 output. For example, the canonical product for the logic function in Table 3-5 is

canonical product

$$\begin{aligned} F &= \Pi_{X,Y,Z}(1,2,5) \\ &= (X + Y + Z') \cdot (X + Y' + Z) \cdot (X' + Y + Z') \end{aligned}$$

Here, the notation $\Pi_{X,Y,Z}(1,2,5)$ is a *maxterm list* and means “the product of maxterms 1, 2, and 5 with variables X, Y, and Z.” The maxterm list is also known as the *off-set* for the logic function. You can visualize that each maxterm “turns off” the output for exactly one input combination. Any logic function can be written as a canonical product.

maxterm list

off-set

It’s easy to convert between a minterm list and a maxterm list. For a function of n variables, the possible minterm and maxterm numbers are in the set $\{0, 1, \ldots, 2^n - 1\}$; a minterm or maxterm list contains a subset of these numbers. To switch between list types, take the set complement, for example:

$$\begin{aligned} \Sigma_{A,B,C}(0,1,2,3) &= \Pi_{A,B,C}(4,5,6,7) \\ \Sigma_{X,Y}(1) &= \Pi_{X,Y}(0,2,3) \\ \Sigma_{W,X,Y,Z}(0,1,2,3,5,7,11,13) &= \Pi_{W,X,Y,Z}(4,6,8,9,10,12,14,15) \end{aligned}$$

A combinational logic function can also be described in many different ways by statements in an HDL. In Verilog, a `case` statement can be written that corresponds directly to the minterm list or maxterm list of a function. For the example logic function that we’ve been using, from Table 3-5 on page 101, we could write the following Verilog statement corresponding to the minterm list:

```
case ({X,Y,Z})
  0,3,4,6,7: F = 1;
  default:   F = 0;
endcase
```

Here, the braces {} convert the three 1-bit inputs into a 3-bit value used to select a case; the minterm numbers are listed for the cases where the function's value is 1; and the default value for unlisted cases is specified to be 0. We could also write Verilog for the corresponding maxterm list as follows:

```
case ({X,Y,Z})
  1,2,5:   F = 0;
  default: F = 1;
endcase
```

The Verilog statements above are of course just code fragments, but we'll give details of the language in Chapter 5.

We have now learned six equivalent representations for a combinational logic function:

1. A truth table.
2. An algebraic sum of minterms, the canonical sum.
3. A minterm list using the Σ notation.
4. An algebraic product of maxterms, the canonical product.
5. A maxterm list using the Π notation.
6. A Verilog `case` statement.

Each one of these representations specifies exactly the same information; given any one of them, we can derive any of the others using a simple process of selection and/or substitution. For example, to go from a minterm list to a canonical product, we create a truth table with a 1 in each row corresponding to a listed minterm number, and then write the algebraic product of the maxterms corresponding to each truth-table row that does not have a 1 in it.

3.2 Combinational-Circuit Analysis

We can analyze a combinational logic circuit by obtaining a formal description of its logic function. Once we have a description of the logic function, we can perform a number of other operations:

- Determine the behavior of the logic circuit for various input combinations. We can do this with paper and pencil, or use an EDA tool—a simulator.
- Manipulate an algebraic or equivalent graphical description to suggest different circuit elements or structures for the logic function. Some such manipulations are very straightforward and may make the circuit's function easier to understand.

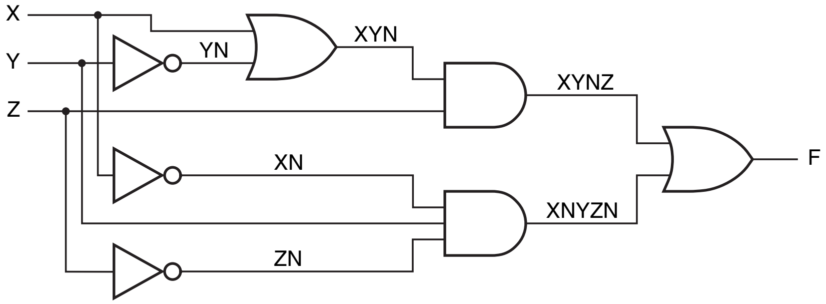


Figure 3-5 A 3-input,1-output logic circuit.

- Transform an algebraic description into a standard form corresponding to an available circuit structure; such an operation could be used by a software tool to "realize" (make real) a circuit that performs the logic function. For example, a truth table corresponds to the function "lookup-table" (LUT) memory used in FPGAs (field programmable gate arrays), and a sum-of-products expression corresponds directly to the circuit structure used in PLDs (programmable logic devices).
- Use an algebraic description of the circuit's functional behavior in the analysis of a larger system that contains the circuit.

In this subsection we'll focus on operations that you can carry out by hand for small circuits, but we'll also point out how equivalent operations can be done by logic-design software tools.

Given a logic diagram for a combinational circuit, like Figure 3-5, there are several ways to obtain a formal description of the circuit's function. The most basic functional description is the truth table.

Using only the basic axioms of switching algebra, we can obtain the truth table of an n-input circuit by working our way through all 2^n input combinations. For each input combination, we determine all of the gate outputs produced by that input, propagating information from the circuit inputs to the circuit outputs. Figure 3-6 applies this "exhaustive" technique to our example circuit. Written on each signal line in the circuit is a sequence of eight logic values, the values

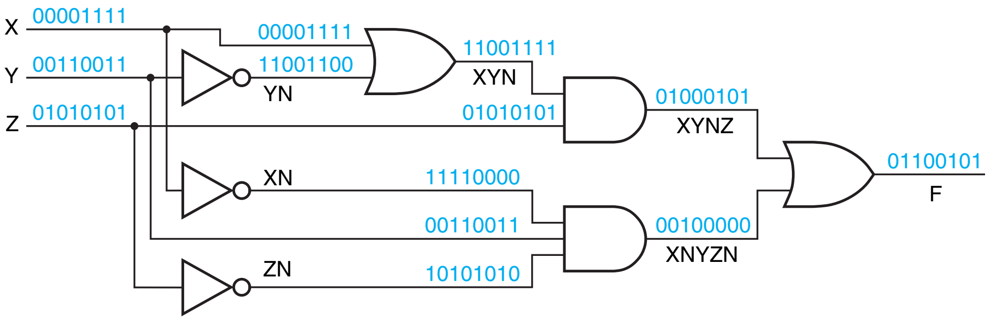


Figure 3-6 Gate outputs created by all input combinations.

Table 3-7 Truth table for the logic circuit of Figure 3-5.

Row	X	Y	Z	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	0
4	1	0	0	0
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

present on that line when the circuit inputs X Y Z are 000, 001, ..., 111. The output column F of the truth table can be filled in by transcribing the output sequence of the final OR gate, as shown in Table 3-7. Once we have the truth table for the circuit, we can also directly write a logic expression—the canonical sum or product—for F if we wish.

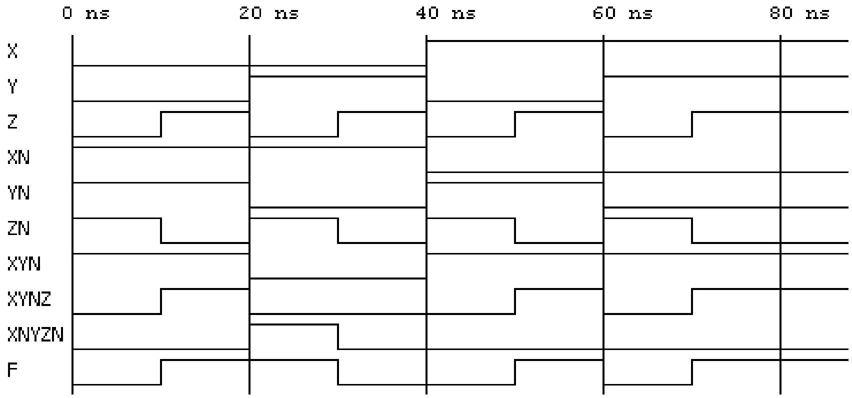


Figure 3-7 Simulator timing diagram for logic circuit.

A LESS EXHAUSTING WAY TO GO

You can easily obtain the results in Figure 3-6 with typical EDA tools that include a logic simulator. First, you draw the logic diagram or create an equivalent "structural" HDL model. Then, in the simulator, you apply 3-bit combinations to the circuit's X, Y, and Z inputs in binary counting order as in the figure. (Many simulators have counters built in for just this sort of exercise.) The simulator allows you to create a timing diagram of the resulting signal values at any point in the schematic, including the intermediate points as well as the output.

Figure 3-7 is a timing diagram produced by the simulator when a 3-bit counter was provided to step through the input combinations, one every 10 ns. The simulated output values on the signal lines correspond exactly to those shown in Figure 3-6.

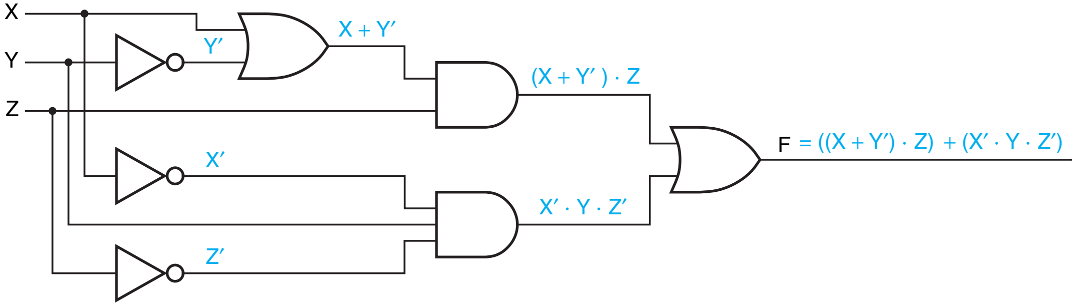


Figure 3-8 Logic expressions for signal lines.

The number of input combinations of a logic circuit grows exponentially with the number of inputs, so the exhaustive approach can quickly become exhausting. For many analysis problems, it may be better to use an algebraic approach whose complexity is more linearly proportional to the size of the circuit. The method is simple—we build up a parenthesized logic expression corresponding to the logic operators and structure of the circuit. We start at the circuit inputs and propagate expressions through gates toward the output. Using the theorems of switching algebra, we may simplify the expressions as we go, or we may defer all algebraic manipulations until an output expression is obtained.

Figure 3-8 applies the algebraic technique to our example circuit. The output function is given on the output of the final OR gate:

$$F = ((X+Y') \cdot Z) + (X' \cdot Y \cdot Z')$$

No switching-algebra theorems were used to obtain this expression. However, we can use theorems to transform this expression into another form. For example, a sum of products can be obtained by "multiplying out":

$$F = X \cdot Z + Y' \cdot Z + X' \cdot Y \cdot Z'$$

The new expression leads to a different circuit for the same logic function, as shown in Figure 3-9.

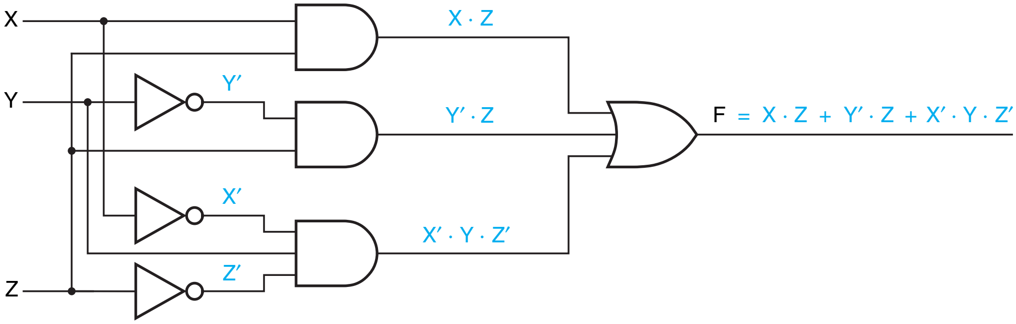


Figure 3-9 Two-level AND-OR circuit.

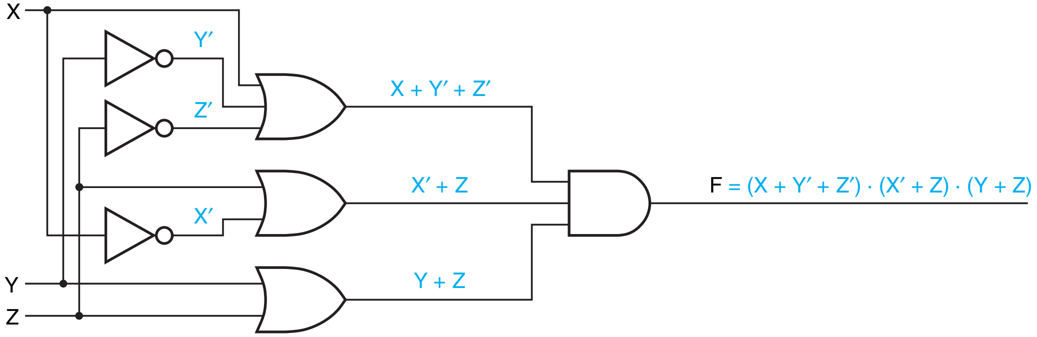


Figure 3-10 Two-level OR-AND circuit.

Similarly, we can "add out" the original expression to obtain a product of sums corresponding to the logic circuit in Figure 3-10:

$$\begin{aligned} F &= ((X + Y') \cdot Z) + (X' \cdot Y \cdot Z') \\ &= (X + Y' + X') \cdot (X + Y' + Y) \cdot (X + Y' + Z') \cdot (Z + X') \cdot (Z + Y) \cdot (Z + Z') \\ &= 1 \cdot 1 \cdot (X + Y' + Z') \cdot (X' + Z) \cdot (Y + Z) \cdot 1 \\ &= (X + Y' + Z') \cdot (X' + Z) \cdot (Y + Z) \end{aligned}$$

Our next example of algebraic analysis uses a circuit with NAND and NOR gates, shown in Figure 3-11. This analysis is a little messier than the previous example, because each gate produces a complemented subexpression, not just a simple sum or product. However, the output expression can be simplified by repeated application of the generalized DeMorgan's theorem:

$$\begin{aligned} F &= [((W \cdot X')' \cdot Y)' + (W' + X + Y')' + (W + Z)']' \\ &= ((W' + X)' + Y')' \cdot (W \cdot X' \cdot Y)' \cdot (W' \cdot Z')' \\ &= ((W \cdot X')' \cdot Y) \cdot (W' + X + Y') \cdot (W + Z) \\ &= ((W' + X) \cdot Y) \cdot (W' + X + Y') \cdot (W + Z) \end{aligned}$$

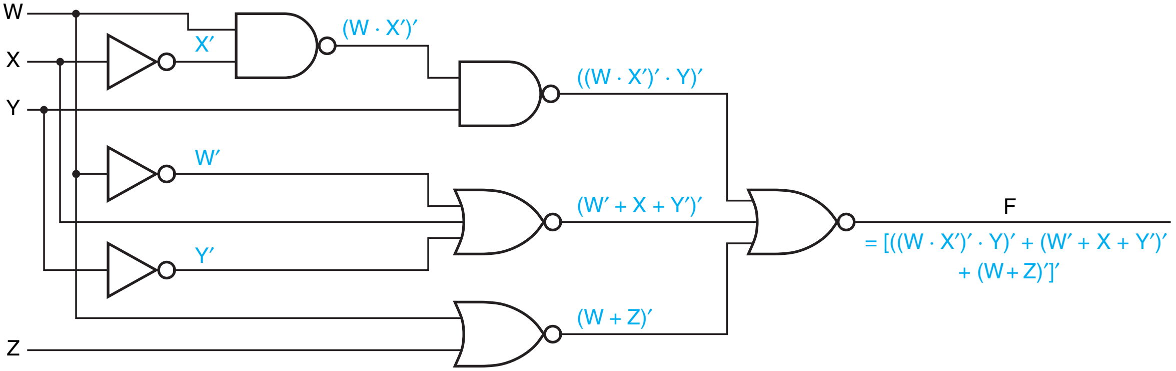


Figure 3-11 Algebraic analysis of a logic circuit with NAND and NOR gates.

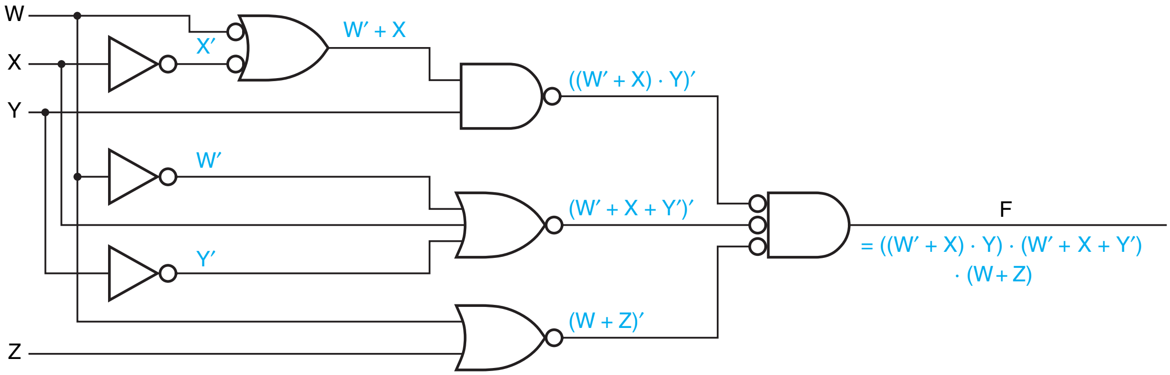


Figure 3-12 Algebraic analysis of previous circuit after substituting some NAND and NOR symbols.

Quite often, DeMorgan's theorem can be applied *graphically* to simplify algebraic analysis. Recall from Figures 3-3 and 3-4 that NAND and NOR gates each have two equivalent symbols. By judiciously redrawing Figure 3-11, we make it possible to cancel out some of the inversions during the analysis by using theorem T4 [(X′)′ = X], as shown in Figure 3-12. This manipulation leads us directly to a simplified output expression:

$$F = ((W' + X) \cdot Y) \cdot (W' + X + Y') \cdot (W + Z)$$

Figures 3-11 and 3-12 were just two different ways of drawing the same physical logic circuit. However, when we simplify a logic expression using the theorems of switching algebra, we get an expression corresponding to a different physical circuit. For example, the simplified expression above corresponds to the circuit of Figure 3-13, which is physically different from the one in the previous two figures. Furthermore, we could multiply out and add out the expression to obtain sum-of-products and product-of-sums expressions corresponding to two more physically different circuits for the same logic function.

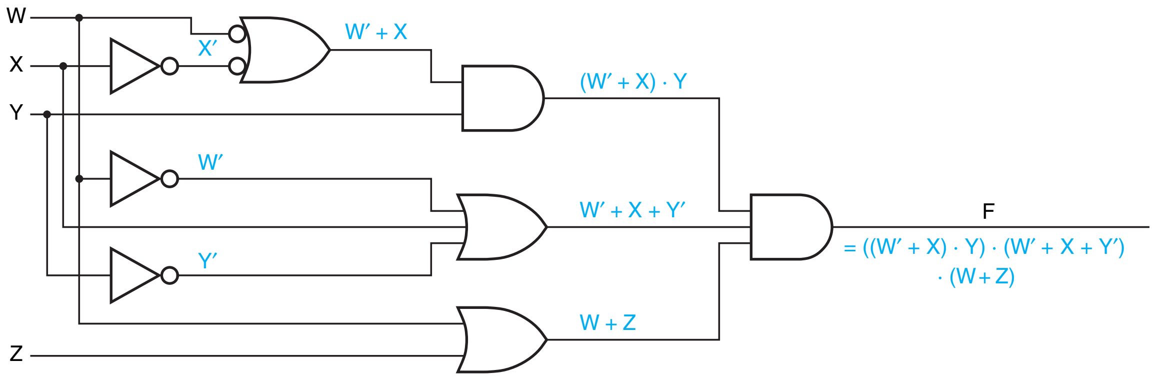


Figure 3-13 A different circuit for same logic function.

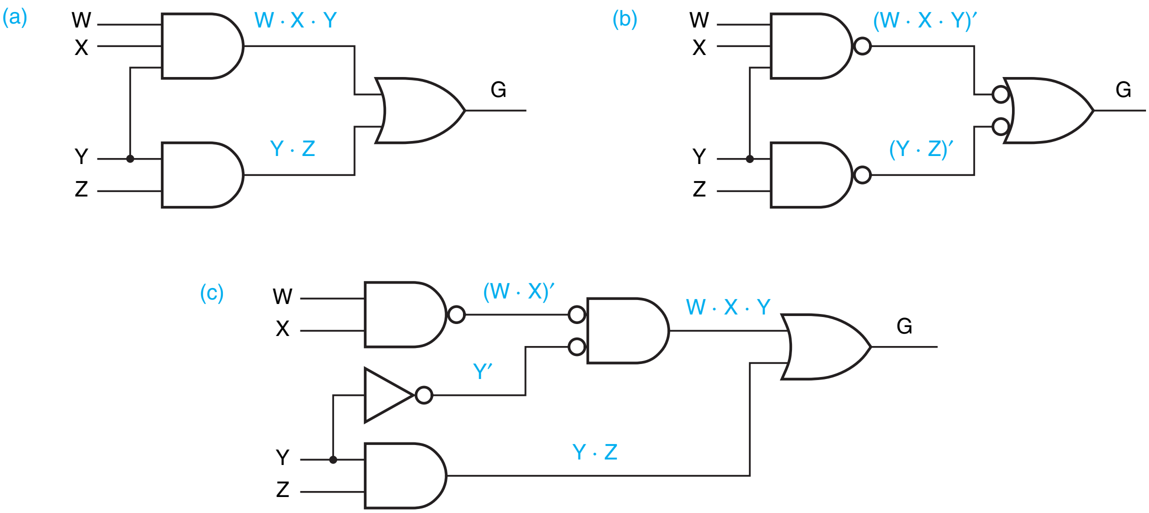


Figure 3-14 Three circuits for G(W, X, Y, Z) = W · X ·Y + Y · Z: (a) two-level AND-OR; (b) two-level NAND-NAND; (c) with 2-input gates only.

Although we used logic expressions above to convey information about the physical structure of a circuit, we don't always do this. For example, we might use the expression G(W, X, Y, Z) = W · X · Y + Y · Z to describe any of the circuits in Figure 3-14. Normally, the only sure way to determine a circuit's structure is to look at its logic diagram. However, for certain restricted classes of circuits, structural information can be inferred from logic expressions. For example, the circuit in (a) could be described without reference to the drawing as "a two-level AND-OR circuit for W · X · Y + Y · Z," while the circuit in (b) could be described as "a two-level NAND-NAND circuit for W · X · Y + Y · Z."

3.3 Combinational-Circuit Synthesis

We may use the words "digital design" to refer broadly to an entire process, from concept to physical design of a digital logic circuit or system. However, we use the word *synthesis* more narrowly, referring to the process that starts with a precise formal specification of the required function and creates details of an implementation—a physical logic circuit that performs the function.

What is the starting point for the *design* of combinational logic circuits? Usually, we are given a word description of a problem, or we develop one ourselves. Unless we're constrained to use a particular technology to realize the corresponding physical circuit (as in this chapter, where we're only looking at discrete gates), the next step would be to select the target technology, since different ones may have different synthesis tools. We should develop the circuit's formal specification in a format that is compatible with those tools.

WHY STUDY GATE-LEVEL SYNTHESIS?

Most digital design nowadays is carried out using building blocks that are larger (perhaps much larger) than discrete gates, or using HDLs and synthesizers that create the corresponding physical implementations. There's no need for the designer to get involved with synthesis at the level described in this section. To design a microprocessor with millions of gates, an HDL-based approach for the "routine" parts of the design is essential if the it's ever to be completed.

However, sometimes the synthesizer's results just aren't good enough. To achieve performance goals, it may still be necessary for critical blocks (such as adders, multipliers, multiplexers, and specialized high-speed control circuits) to be synthesized "by hand," with the designer playing an active role in the selection of gate-level structures and interconnections, and even guiding physical layout in the case of both ASIC and FPGA design.

There are also cases where the synthesizer may "run amok," creating a circuit that is much less efficient (in speed, size, or some other metric) than what is expected and required. In these cases, it is important for the designer to have a good feel for what *could* be achieved, and perhaps try a different style of HDL modeling or structuring to cajole the synthesizer into creating a result that is closer to what is desired. We'll see some examples of that in Chapters 6 and 8.

A basic understanding of combinational logic synthesis at the level presented in this section can help you develop such a "good feel."

In modern digital-design environments, the word description may be converted into a model in a hardware description language (HDL) like Verilog, and we'll see many examples of that beginning in Chapter 6. In the current chapter, which targets discrete gate-level designs, we'll look at synthesis methods that start with specifications using one of the tabular or algebraic representations that we introduced in Section 3.1.

3.3.1 Circuit Descriptions and Designs

Occasionally, a logic circuit description is just a list of input combinations for which a signal should be on or off, the verbal equivalent of a truth table or the Σ or $\prod$ notation introduced previously. For example, the description of a 4-bit prime-number detector might be, "Given a 4-bit input combination $\mathsf{N} = \mathsf{N}_3\mathsf{N}_2\mathsf{N}_1\mathsf{N}_0$, produce a 1 output for $\mathsf{N} = 1, 2, 3, 5, 7, 11, 13$, and 0 otherwise." A logic function described in this way can be designed directly from the canonical sum or product expression. For the prime-number detector, we have

$$
\begin{aligned}
\mathsf{F} &= \Sigma_{\mathsf{N}_3,\mathsf{N}_2,\mathsf{N}_1,\mathsf{N}_0}(1, 2, 3, 5, 7, 11, 13) \\
&= \mathsf{N}_3'\cdot\mathsf{N}_2'\cdot\mathsf{N}_1'\cdot\mathsf{N}_0 + \mathsf{N}_3'\cdot\mathsf{N}_2'\cdot\mathsf{N}_1\cdot\mathsf{N}_0' + \mathsf{N}_3'\cdot\mathsf{N}_2'\cdot\mathsf{N}_1\cdot\mathsf{N}_0 + \mathsf{N}_3'\cdot\mathsf{N}_2\cdot\mathsf{N}_1'\cdot\mathsf{N}_0 \\
&\quad + \mathsf{N}_3'\cdot\mathsf{N}_2\cdot\mathsf{N}_1\cdot\mathsf{N}_0 + \mathsf{N}_3\cdot\mathsf{N}_2'\cdot\mathsf{N}_1\cdot\mathsf{N}_0 + \mathsf{N}_3\cdot\mathsf{N}_2\cdot\mathsf{N}_1'\cdot\mathsf{N}_0
\end{aligned}
$$

The corresponding circuit is shown in Figure 3-15.

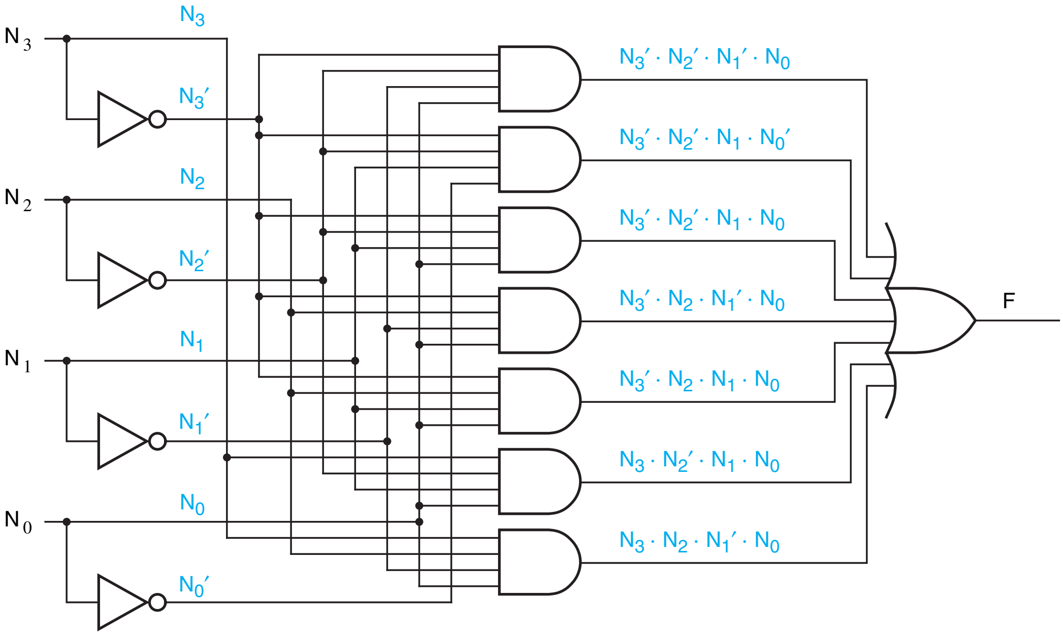

N_3
N_3
N_3'
N_2
N_2
N_2'
N_1
N_1
N_1'
N_0
N_0
N_0'
$N_3' \cdot N_2' \cdot N_1' \cdot N_0$
$N_3' \cdot N_2' \cdot N_1 \cdot N_0'$
$N_3' \cdot N_2' \cdot N_1 \cdot N_0$
$N_3' \cdot N_2 \cdot N_1' \cdot N_0$
$N_3' \cdot N_2 \cdot N_1 \cdot N_0$
$N_3 \cdot N_2' \cdot N_1 \cdot N_0$
$N_3 \cdot N_2 \cdot N_1' \cdot N_0$
F

Figure 3-15 Canonical-sum design for 4-bit prime-number detector.

More often, we describe a logic function using the natural-language connectives "and," "or," and "not." For example, we might describe an alarm circuit by saying, "The ALARM output is 1 if the PANIC input is 1, or if the ENABLE input is 1, the EXITING input is 0, and the house is not secure; the house is secure if the WINDOW, DOOR, and GARAGE inputs are all 1." Such a description can be translated directly into algebraic expressions:

$$\begin{aligned} \text{ALARM} &= \text{PANIC} + \text{ENABLE} \cdot \text{EXITING}' \cdot \text{SECURE}' \\ \text{SECURE} &= \text{WINDOW} \cdot \text{DOOR} \cdot \text{GARAGE} \\ \text{ALARM} &= \text{PANIC} + \text{ENABLE} \cdot \text{EXITING}' \cdot (\text{WINDOW} \cdot \text{DOOR} \cdot \text{GARAGE})' \end{aligned}$$

Notice that we used the same method in switching algebra as in ordinary algebra to formulate a complicated expression—we defined an auxiliary variable SECURE to simplify the first equation, developed an expression for SECURE, and used substitution to get the final expression. We can easily draw a circuit using AND, OR, and NOT gates that realizes the final expression, as shown in

PRIME TIME Mathematicians will tell you that "1" is not really a prime number. But our prime-number detector example is not nearly as interesting, from a logic-synthesis point of view, if "1" is not prime. So, please do Drill 3.11 if you want to be a mathematical purist.

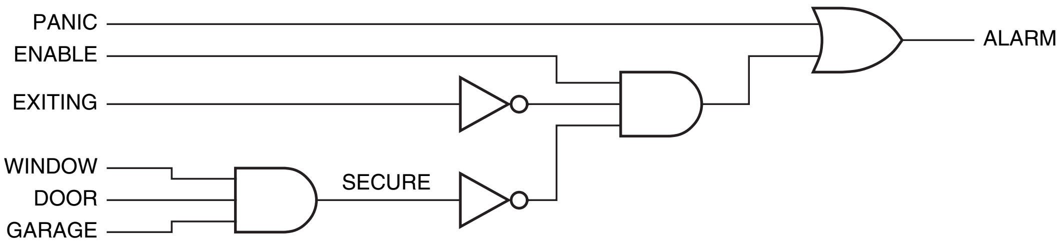


Figure 3-16 Alarm circuit derived from logic expression.

Figure 3-16. A circuit *realizes* ("makes real") an expression if its output function equals that expression, and the circuit is called a *realization* of the function. We can and will also call it an *implementation*; both terms are used in practice.

realize
realization
implementation

Once we have an expression, any expression, for a logic function, we can do other things besides building a circuit directly from the expression. We can manipulate the expression to get different circuits. For example, the ALARM expression above can be multiplied out to get the sum-of-products circuit in Figure 3-17. Or, if the number of variables is not too large, we can construct the truth table for the expression and use any of the synthesis methods that apply to truth tables, including the canonical sum or product method described earlier and the minimization methods to be described later.

In general, when we're designing a logic function for an application, it's easier to describe it in words using logical connectives and to write the corresponding logic expressions than it is to write a complete truth table, especially if the number of variables is large. However, sometimes we start with imprecise word descriptions of logic functions, for example, "The ERROR output should be 1 if the GEARUP, GEARDOWN, and GEARCHECK inputs are inconsistent." In this situation, the truth-table approach is best because it allows us to determine the output required for every input combination, based on our knowledge and understanding of the problem environment (e.g., the brakes cannot be applied unless the gear is down). Using a logic expression may make it difficult to notice so-called "corner cases" and handle them appropriately.

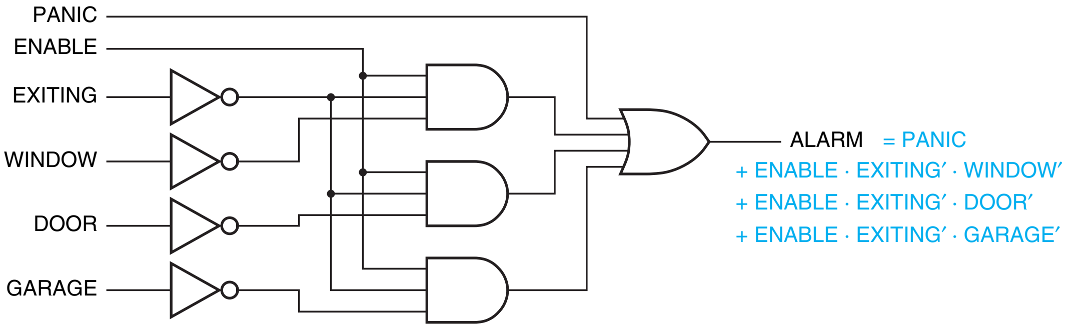

PANIC
ENABLE
EXITING
WINDOW
DOOR
GARAGE
ALARM = PANIC
+ ENABLE · EXITING′ · WINDOW′
+ ENABLE · EXITING′ · DOOR′
+ ENABLE · EXITING′ · GARAGE′

Figure 3-17 Sum-of-products version of alarm circuit.

3.3.2 Circuit Manipulations

The design methods that we've described so far use AND, OR, and NOT gates. We might like to use NAND and NOR gates, too—they're faster than ANDs and ORs in most technologies, including typical CMOS ASIC libraries. However, most people don't develop logical propositions in terms of NAND and NOR connectives. That is, you probably wouldn't say, "I won't date you if you're not clean or not wealthy and also you're not smart or not friendly." It would be more natural for you to say, "I'll date you if you're clean and wealthy, or if you're smart and friendly." So, given a "natural" logic expression, we need ways to translate it into other forms for efficient implementation.

We can translate any logic expression into an equivalent sum-of-products expression, simply by multiplying it out. As shown in Figure 3-18(a), a sum-of-products expression can be realized directly with AND and OR gates. The inverters needed for complemented inputs are not shown.

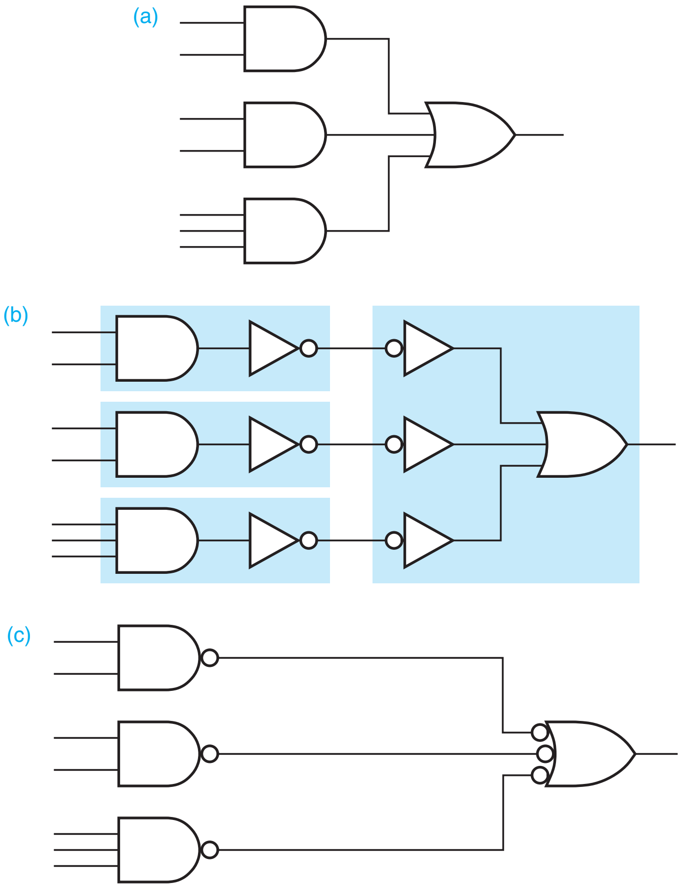


Figure 3-18
Alternative sum-of-products circuits:
(a) AND-OR;
(b) AND-OR with extra inverter pairs;
(c) NAND-NAND.

As shown in Figure 3-18(b), we can insert a pair of inverters between each AND-gate output and the corresponding OR-gate input in a two-level AND-OR circuit. According to theorem T4, these inverters have no effect on the output function of the circuit. In fact, we've drawn the second inverter of each pair with its inversion bubble on its input to provide a graphical reminder that the inverters cancel. However, if these inverters are absorbed into the AND and OR gates, we wind up with AND-NOT gates at the first level and a NOT-OR gate at the second level. These are just two different symbols for the same type of gate—a NAND gate. Thus, a two-level *AND-OR circuit* may be converted to a two-level *NAND-NAND circuit* simply by substituting gates.

AND-OR circuit

NAND-NAND circuit

If any product terms in the sum-of-products expression contain just one literal, then we may gain or lose inverters in the transformation from AND-OR to NAND-NAND. In the example of Figure 3-19, an inverter is no longer needed on the W input, but an inverter must be added to the Z input.

We have shown that any sum-of-products expression can be realized in either of two ways—as an AND-OR circuit or as a NAND-NAND circuit. The dual of this statement is also true: any product-of-sums expression can be real-

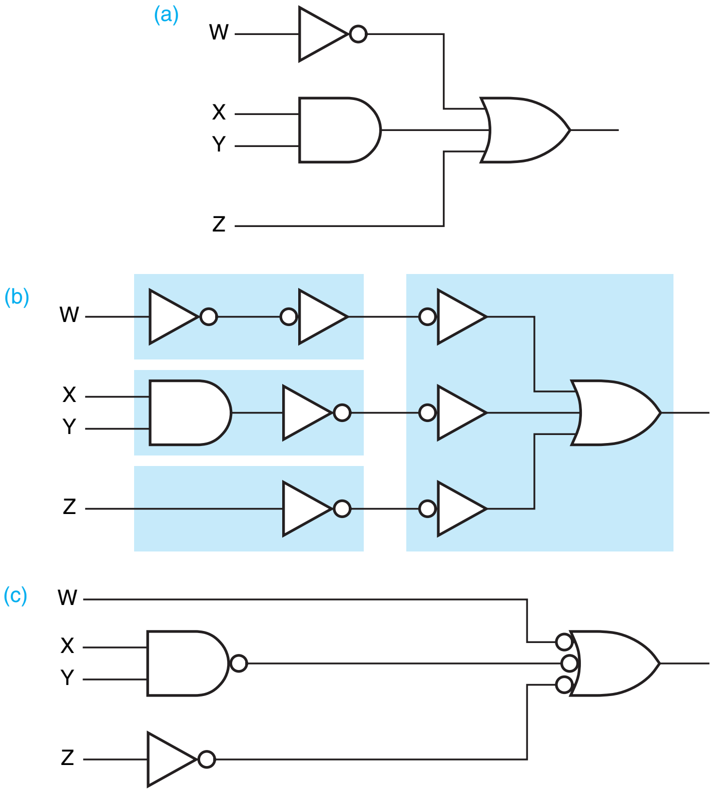


Figure 3-19 Another sum-of-products circuit: (a) AND-OR; (b) AND-OR with extra inverter pairs; (c) NAND-NAND.

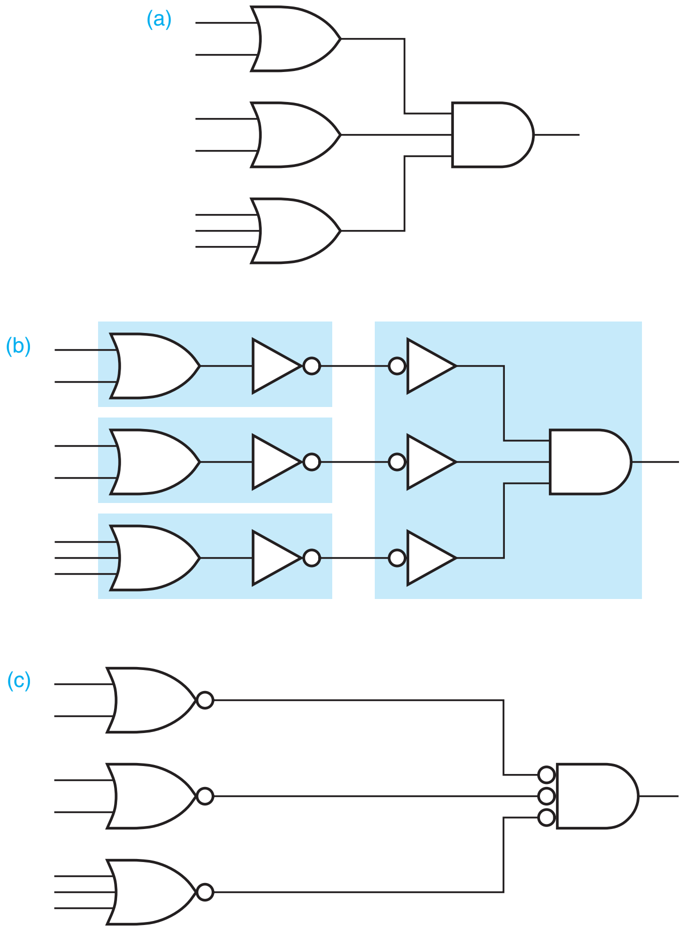


Figure 3-20
Realizations of a product-of-sums expression:
(a) OR-AND;
(b) OR-AND with extra inverter pairs;
(c) NOR-NOR.

OR-AND circuit
NOR-NOR circuit

ized as an *OR-AND circuit* or as a *NOR-NOR circuit*. Figure 3-20 shows an example. Any logic expression can be translated into an equivalent product-of-sums expression by adding it out, and hence has both OR-AND and NOR-NOR circuit realizations.

The same kind of manipulations can be applied to arbitrary logic circuits. For example, Figure 3-21(a) shows a circuit built from AND and OR gates. After adding pairs of inverters, we obtain the circuit in (b). However, one of the gates, a 2-input AND gate with a single inverted input, is not a standard type. We can use a discrete inverter as shown in (c) to obtain a circuit that uses only standard gate types—NAND, AND, and inverters. Actually, a better way to use the inverter is shown in (d); one level of gate delay is eliminated, and the bottom gate becomes a NOR instead of AND. Synthesis tools can perform such "inverter pushing" operations automatically. In CMOS logic technology, inverting gates like NAND and NOR are faster than noninverting gates like AND and OR.

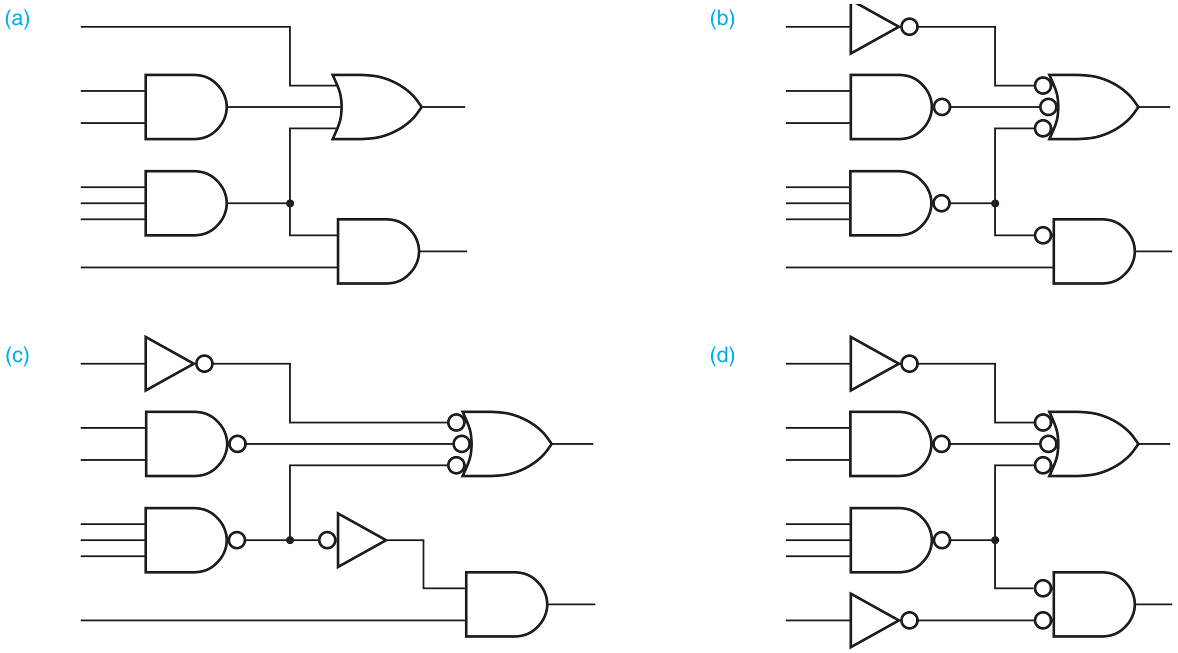


Figure 3-21 Logic-symbol manipulations: (a) original circuit; (b) transformation with a nonstandard gate; (c) inverter used to eliminate nonstandard gate; (d) preferred inverter placement.

3.3.3 Combinational-Circuit Minimization

It's often uneconomical or inefficient to realize a logic circuit directly from the first logic expression or other description that pops into your head. Canonical sum and product expressions are especially expensive because the number of possible minterms or maxterms (and hence gates) grows exponentially with the number of variables. We *minimize* a combinational circuit by reducing the number and size of gates that are needed to build it.

minimize

The traditional combinational-circuit-minimization methods that we'll study have as their starting point a truth table or, equivalently, a minterm list or maxterm list. If we are given a logic function that is not expressed in this form, then we must convert it to an appropriate form before using these methods. For example, if we are given an arbitrary logic expression, then we can evaluate it for every input combination to construct the truth table.

The minimization methods reduce the cost of a two-level AND-OR, OR-AND, NAND-NAND, or NOR-NOR circuit in three ways:

1. By minimizing the number of first-level gates.
2. By minimizing the number of inputs on each first-level gate.
3. By minimizing the number of inputs on the second-level gate. This is actually a side effect of the first reduction.

minimal sum
minimal product

However, the minimization methods do not consider the cost of input inverters; they assume that both true and complemented versions of all input variables are available, which is the case in some implementation technologies, in particular PLDs. A two-level realization that has the minimum possible number of first-level gates and gate inputs is called a *minimal sum* or *minimal product*. Some functions have multiple minimal sums or products.

Most minimization methods are based on a generalization of the combining theorems, T10 and T10D:

$$\begin{aligned} \text{given product term} \cdot \mathsf{Y} + \text{given product term} \cdot \mathsf{Y}' &= \text{given product term} \\ (\text{given sum term} + \mathsf{Y}) \cdot (\text{given sum term} + \mathsf{Y}') &= \text{given sum term} \end{aligned}$$

That is, if two product or sum terms differ only in the complementing or not of one variable, we can combine them into a single term with one less variable. So we eliminate one gate, and the remaining gate has one less input.

We can apply this algebraic method repeatedly to combine minterms 1, 3, 5, and 7 of the prime-number detector shown in Figure 3-15 on page 112:

$$\begin{aligned} \mathsf{F} &= \Sigma_{\mathsf{N_3,N_2,N_1,N_0}}(1, 3, 5, 7, 2, 11, 13) \\ &= \mathsf{N_3' \cdot N_2' \cdot N_1' \cdot N_0 + N_3' \cdot N_2' \cdot N_1 \cdot N_0 + N_3' \cdot N_2 \cdot N_1' \cdot N_0 + N_3' \cdot N_2 \cdot N_1 \cdot N_0} + \ldots \\ &= \mathsf{(N_3' \cdot N_2' \cdot N_1' \cdot N_0 + N_3' \cdot N_2' \cdot N_1 \cdot N_0) + (\cdot N_3' \cdot N_2 \cdot N_1' \cdot N_0 + N_3' \cdot N_2 \cdot N_1 \cdot N_0)} + \ldots \\ &= \mathsf{N_3' \cdot N_2' \cdot N_0 + N_3' \cdot N_2 \cdot N_0} + \ldots \\ &= \mathsf{N_3' \cdot N_0} + \ldots \end{aligned}$$

The resulting circuit is shown in Figure 3-22; it has three fewer gates, and one of the remaining gates has two fewer inputs.

If we had worked a little harder on the preceding expression, we could have saved a couple more first-level gate inputs, though not any more gates. But it's difficult to find terms that can be combined in a jumble of algebraic symbols. And we don't have to, as will be shown in the next subsection.

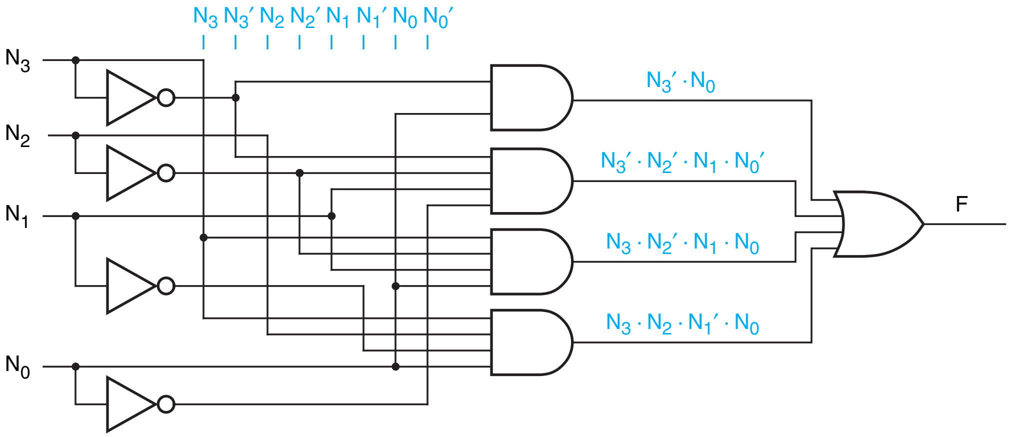

N_3 N_3' N_2 N_2' N_1 N_1' N_0 N_0'
N_3
N_2
N_1
N_0
$N_3' \cdot N_0$
$N_3' \cdot N_2' \cdot N_1 \cdot N_0'$
$N_3 \cdot N_2' \cdot N_1 \cdot N_0$
$N_3 \cdot N_2 \cdot N_1' \cdot N_0$
F

Figure 3-22 Simplified sum-of-products realization for 4-bit prime-number detector.

WHY MINIMIZE? FPGAs don’t have a programmable AND-OR structure. Instead, they use a lookup table that can realize any logic function of *n* variables, where *n* is typically 4 to 6. But their synthesis tools may still perform two-level minimization along the lines described here. For larger functions that don’t fit into one lookup table, experience has shown that a minimized two-level expression is a good place to start “factoring” to find a multi-level expressioin that *will* fit into a collection of smaller lookup tables. For the same reason, minimization is also important in ASIC synthesis using discrete gates, since the number of gate inputs is limited.

Programmable logic devices (PLDs) do use a programmable AND-OR structure. Since the number of gates in a PLD is fixed even if you don’t use them all, you might think that extra gates are free—and they are, until you run out of them and have to upgrade to a bigger, slower, and more expensive PLD. So, EDA tools for FPGA, ASIC and PLD design have a minimization program built in. The main purpose of Sections 3.3.3 and 3.3.4 is to give you a feel for how minimization works.

*3.3.4 Karnaugh Maps

Karnaugh map

Decades ago, digital designers used diagrams called *Karnaugh maps* to create graphical representations of logic functions, so that minimization opportunities could be identified by simple, visual pattern recognition. The key feature of a Karnaugh map is its cell layout: each pair of adjacent cells corresponds to a pair of minterms that differ in only one variable which is uncomplemented in one cell and complemented in the other. Such a minterm pair can be combined into one product term using a generalization of theorem T10, term · Y + term · Y′ = term. Thus, using a logic function’s Karnaugh map, we can combine product terms to reduce how many AND gates and gate inputs are needed to realize the function.

Figure 3-23 shows Karnaugh maps for 2-, 3-, and 4-variable functions. The rows and columns of a map are labeled so the input combination for a cell can be determined from its row and column headings, and the number inside each cell is the truth table row or minterm number corresponding to that cell. Also, the labeled brackets indicate the rows or columns where each variable is 1.

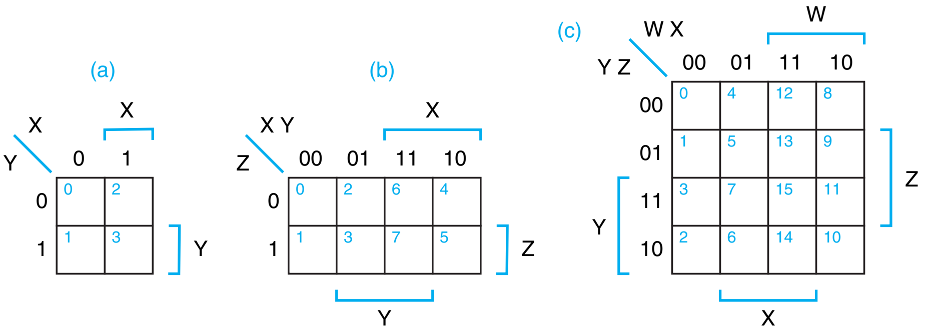


Figure 3-23 Karnaugh maps: (a) 2-variable; (b) 3-variable; (c) 4-variable.

* Throughout this book, optional sections are marked with an asterisk.

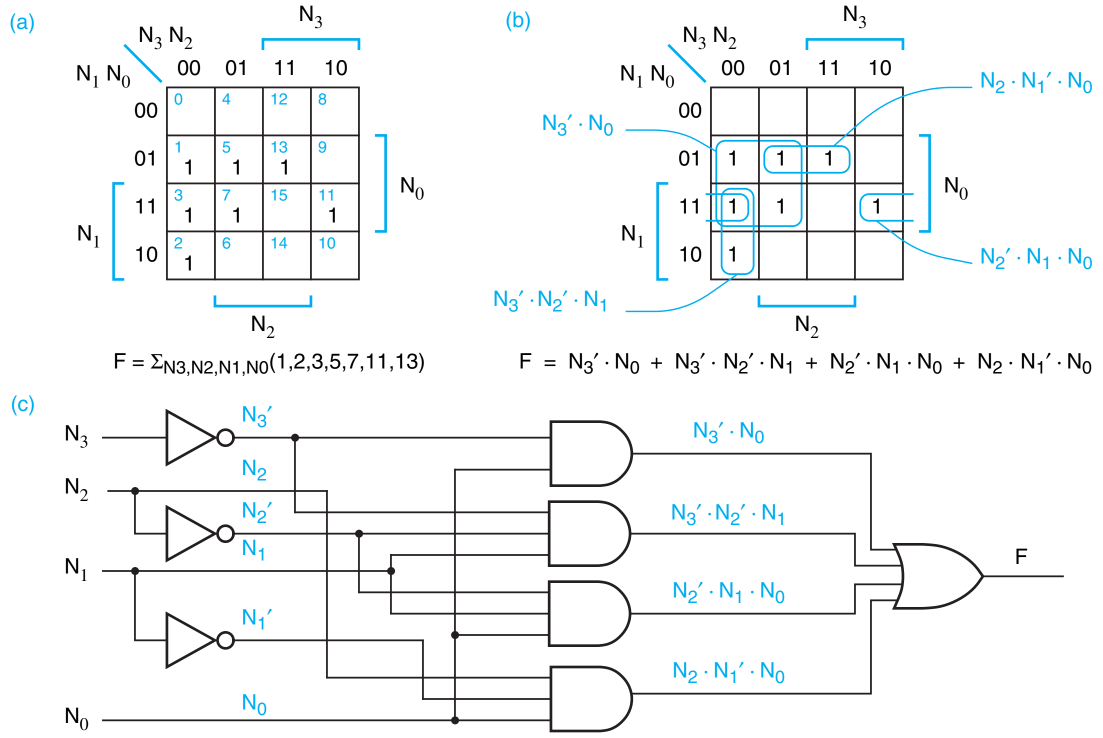

(a)
N3 N2
N1 N0
00 01 11 10
N3
N0
N1
N2
0 4 12 8
1 5 13 9
3 7 15 11
2 6 14 10
$F = \Sigma_{N3,N2,N1,N0}(1,2,3,5,7,11,13)$
(b)
$N_3' \cdot N_0$
$N_2 \cdot N_1' \cdot N_0$
$N_2' \cdot N_1 \cdot N_0$
$N_3' \cdot N_2' \cdot N_1$
$F = N_3' \cdot N_0 + N_3' \cdot N_2' \cdot N_1 + N_2' \cdot N_1 \cdot N_0 + N_2 \cdot N_1' \cdot N_0$
(c)
N_3
N_2
N_1
N_0
N_3'
N_2'
N_1'
$N_3' \cdot N_0$
$N_3' \cdot N_2' \cdot N_1$
$N_2' \cdot N_1 \cdot N_0$
$N_2 \cdot N_1' \cdot N_0$
F

Figure 3-24 Prime-number detector: (a) initial Karnaugh map; (b) circled product terms; (c) minimized circuit.

prime implicant

Figure 3-24 shows how a Karnaugh map can be used to minimize a logic function, our prime-number-detector example. In (a), we've copied the 1 outputs in the function's truth table and entered them in the numbered cells in the map for the corresponding input combinations (minterms). In (b), we have grouped adjacent 1 cells in ways that correspond to *prime implicants*: product terms that cover only input combinations for which the function has a 1 output, and that would cover at least one input combination with a 0 output if any variable were removed. These product terms are realized by "smallest possible" AND gates, whose outputs are then combined to obtain a minimized AND-OR circuit, as shown in (c). It has the same number of gates as the algebraically simplified circuit in Figure 3-22, but three of the gates have one fewer input each. See Exercise 3.48 for other interesting examples.

Like most other things involving truth tables, minterms, or maxterms, Karnaugh maps grow in size exponentially as the number of inputs is increased. The maximum practical size for Karnaugh-map minimization is only 6 inputs.

Karnaugh maps are also useful for visualizing the properties of small logic functions, as an aid to understanding the challenges in realizing certain larger ones. In particular, consider an *n*-input even-parity function, which produces a

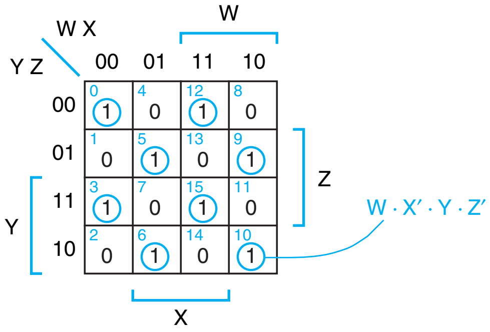


Figure 3-25
Karnaugh map for a 4-input even-parity function.

1 output if the number of 1 inputs is even. As we showed in Section 2.15, parity functions are used to encode and check data using error-detecting and -correcting codes. The map for a 4-input even-parity function is shown in Figure 3-25, and it looks like a checkerboard. It doesn't have *any* adjacent 1-cells that can be combined. Therefore, this function's minimal sum is its canonical sum, which is the sum of its eight minterms circled on the map. A corresponding two-level AND-OR circuit has eight 4-input AND gates to realize product terms like the one shown in the lower righthand corner of the map, and an 8-input OR gate.

Two-level circuits for larger even-parity function would be even bigger; for example, a 6-input function requires 32 AND gates and a 32-input OR gate, which is well beyond the limits of electronic circuit design using a single "level" of CMOS transistors. Instead, troublesome logic functions like this one may be implemented using more than two levels of logic. For example, a 2^n-input parity function may be implemented as an *n*-level "tree" of 2^n–1 2-input parity functions that each have two levels of logic, as we'll show in Section 7.3.

Karnaugh maps can also be used to visualize and understand the possibility of a combinational logic circuit producing a short, unwanted pulse when input signals change, as we will discuss in the next section.

TROUBLESOME FUNCTIONS AND EASY LOOKUPS

Another example of a 6-input logic function that requires a lot of first-level gates is the result bit S2 (third bit from the right) in the addition of a pair of 3-bit or larger numbers. Though not quite as bad as the 6-bit even-parity function, a minimal sum-of-products expression for this function has 18 AND gates. Higher-order bits are exponentially worse, necessitating other methods, both multilevel and hierarchical, to be used for addition, a very commonly used function, as we'll show in Section 8.1.

In an FPGA, these functions are not troublesome at all, up to a point. There, the basic resource for realizing combinational logic problem is a lookup table (LUT) that can store the truth table for any function with up to a certain number of inputs, on the order of six. Thus, the cost and performance of a 6-input parity function and a 6-input NAND gate in a LUT are exactly the same; while they would be much different in any gate-level realization.

*3.4 Timing Hazards

steady-state behavior

The analysis methods that we developed in Section 3.2 ignore circuit delay and predict only the *steady-state behavior* of combinational logic circuits. That is, they predict a circuit's output as a function of its inputs under the assumption that the inputs have been stable for a long time, relative to the delays in the circuit's electronics. However, the actual delay from an input change to the corresponding output change in a real logic circuit is nonzero and depends on many factors in the circuit's electronic design and physical implementation.

transient behavior

glitch

hazard

Because of circuit delays, the *transient behavior* of a combinational logic circuit may differ from what is predicted by a steady-state analysis. In particular, a circuit's output may produce a short pulse, often called a *glitch*, at a time when steady-state analysis predicts that the output should not change. A *hazard* is said to exist when a circuit has the possibility of producing such a glitch. Whether or not the glitch actually occurs depends on the exact delays and other electrical characteristics of the circuit.

Depending on how the circuit's output is used, a system's operation may or may not be adversely affected by a glitch. When we discuss sequential circuits in Chapters 9–13, you'll see situations where glitches may be harmful. In these situations, since exact delays and other electrical characteristics are difficult to control in production circuits, a logic designer must be prepared to eliminate hazards (the *possibility* of a glitch) even though a glitch may occur only under a worst-case combination of logical and electrical conditions. This section will introduce you to hazards and give you some tools to predict and eliminate hazards, allowing you to design glitch-free circuits in simple cases when required.

*3.4.1 Static Hazards

static-1 hazard

A *static-1 hazard* is the possibility of a circuit's output producing a 0 glitch when we would expect the output to remain at a nice steady 1 based on a static analysis of the circuit function. A formal definition is given as follows:

Definition: A static-1 hazard is a pair of input combinations that: (a) differ in only one input variable and (b) both give a 1 output; such that it is possible for a momentary 0 output to occur during a transition in the differing input variable.

For example, consider the logic circuit in Figure 3-26(a). Suppose that X and Y are both 1 and that Z is changing from 1 to 0. Then (b) shows the timing diagram, assuming that the timing delay through each gate or inverter is one unit time. Even though "static" analysis predicts that the output is 1 for both input combinations X,Y,Z = 111 and X,Y,Z = 110, the timing diagram shows that F goes to 0 for one unit time during a 1-0 transition on Z, because of the delay in the inverter that generates Z′.

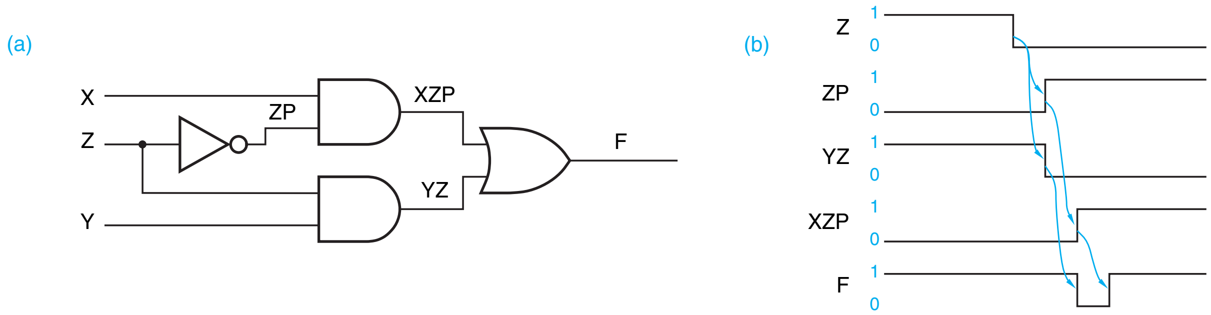


Figure 3-26 Circuit with a static-1 hazard: (a) logic diagram; (b) timing diagram.

static-0 hazard

A *static-0 hazard* is the possibility of a 1 glitch when we expect the circuit to have a steady 0 output:

Definition: A static-0 hazard is a pair of input combinations that: (a) differ in only one input variable and (b) both give a 0 output; such that it is possible for a momentary 1 output to occur during a transition in the differing input variable.

Since a static-0 hazard is just the dual of a static-1 hazard, an OR-AND circuit that is the dual of Figure 3-26(a) would have a static-0 hazard.

An OR-AND circuit with four static-0 hazards is shown in Figure 3-27(a). One of the hazards occurs when W,X,Y = 000 and Z is changed, as shown in (b). You should be able to find the other three hazards and eliminate all of them after studying the next subsection.

*3.4.2 Finding Static Hazards Using Maps

A Karnaugh map can be used to detect static hazards in a two-level sum-of-products or product-of-sums circuit. The existence or nonexistence of static hazards depends on the circuit design for a logic function.

A properly designed two-level sum-of-products (AND-OR) circuit has no static-0 hazards. A static-0 hazard would exist in such a circuit only if both a variable and its complement were connected to the same AND gate, which would

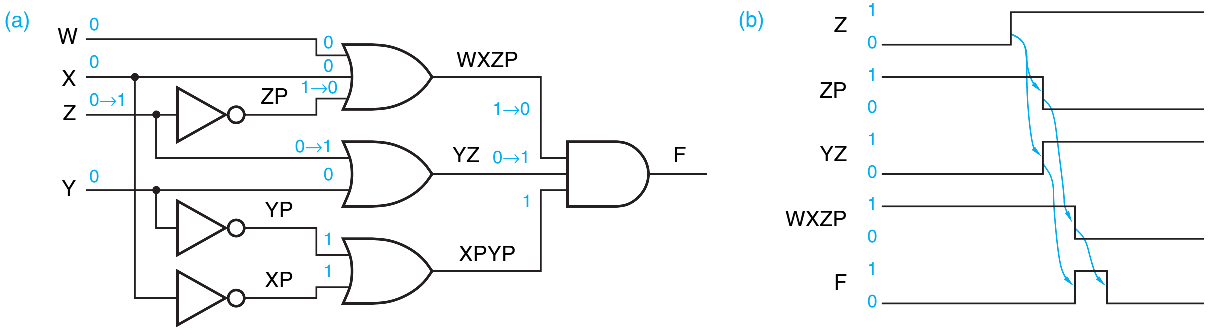


Figure 3-27 Circuit with static-0 hazards: (a) logic diagram; (b) timing diagram.

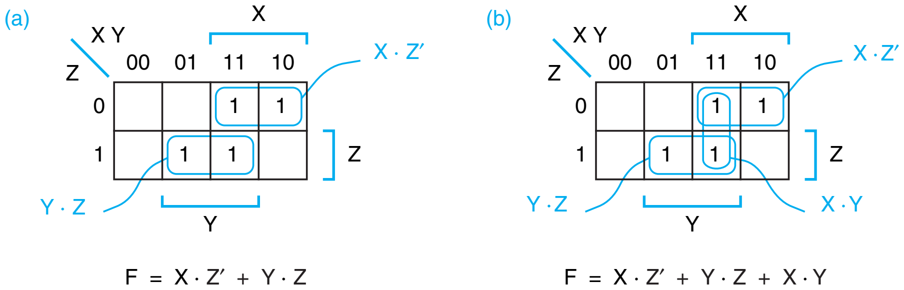


Figure 3-28 Karnaugh map for the circuit of Figure 3-26: (a) as originally designed; (b) with static-1 hazard eliminated.

usually be silly. However, the circuit *may* have static-1 hazards. Their existence can be predicted from a Karnaugh map.

Recall that a Karnaugh map is constructed so that each pair of immediately adjacent cells corresponds to a pair of minterms that differ in only one variable which is uncomplemented in one cell and complemented in the other. For static-1 hazard analysis, we circle the product terms corresponding to the AND gates in the circuit, and we search for adjacent 1 cells that are not covered by a single product term.

Figure 3-28(a) shows the Karnaugh map for the circuit of Figure 3-26. It is clear from the map that there is no single product term that covers both input combinations X,Y,Z = 111 and X,Y,Z = 110. Thus, intuitively, it is possible for the output to "glitch" momentarily to 0 if the AND gate output that covers one of the combinations goes to 0 before the AND gate output covering the other input combination goes to 1. The way to eliminate the hazard is also quite apparent: Simply include an extra product term (AND gate) to cover the hazardous input pair, as shown in Figure 3-28(b). The extra product term, as it turns out, is the *consensus* of the two original terms; in general, we must add consensus terms to eliminate hazards. The corresponding hazard-free circuit is shown in Figure 3-29.

consensus

Another example is shown in Figure 3-30. In this example, three product terms must be added to eliminate the static-1 hazards.

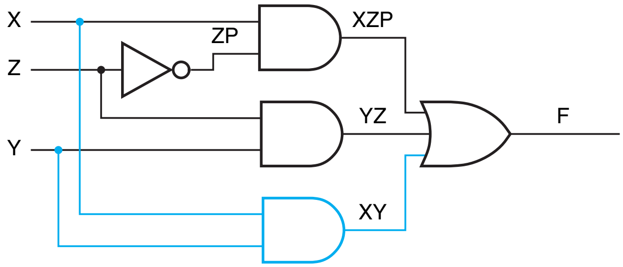
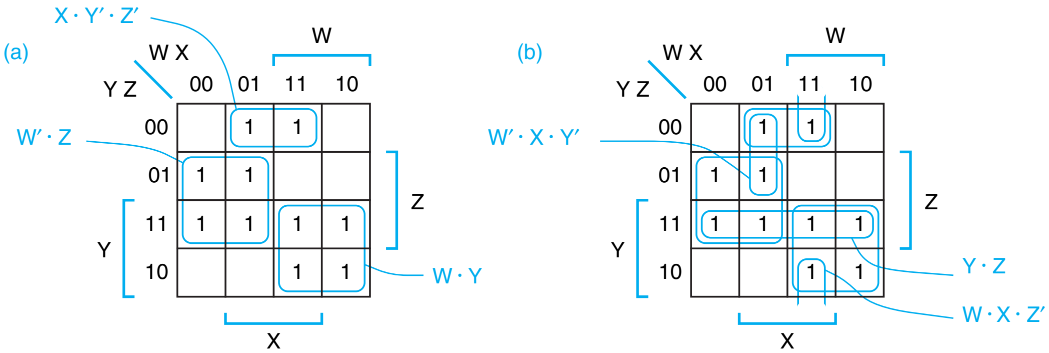


Figure 3-29 Circuit with static-1 hazard eliminated.



$$F = X \cdot Y' \cdot Z' + W' \cdot Z + W \cdot Y$$

$$F = X \cdot Y' \cdot Z' + W' \cdot Z + W \cdot Y + W' \cdot X \cdot Y' + Y \cdot Z + W \cdot X \cdot Z'$$

Figure 3-30 Karnaugh map for another sum-of-products circuit: (a) as originally designed; (b) with extra product terms to cover static-1 hazards.

A properly designed two-level product-of-sums (OR-AND) circuit has no static-1 hazards. It *may* have static-0 hazards, however. These hazards can be detected and eliminated by studying the adjacent 0s in the Karnaugh map, in a manner dual to the foregoing.

*3.4.3 Dynamic Hazards

A *dynamic hazard* is the possibility of an output changing more than once as the result of a single input transition. Multiple output transitions can occur if there are multiple paths with different delays from the input to the output.

dynamic hazard

For example, consider the circuit in Figure 3-31; it has three different paths from input X to the output F. One of the paths goes through a slow OR gate, and another goes through an OR gate that is even slower. If the input to the circuit is W, X, Y, Z = 0, 0, 0, 1, then the output will be 1, as shown. Now suppose we change the X input to 1. Assuming that all of the gates except the two marked "slow" and "slower" are very fast, the transitions shown in black occur next, and the output goes to 0. Eventually, the output of the "slow" OR gate changes, creating the transitions shown in nonitalic color, and the output goes to 1. Finally, the output

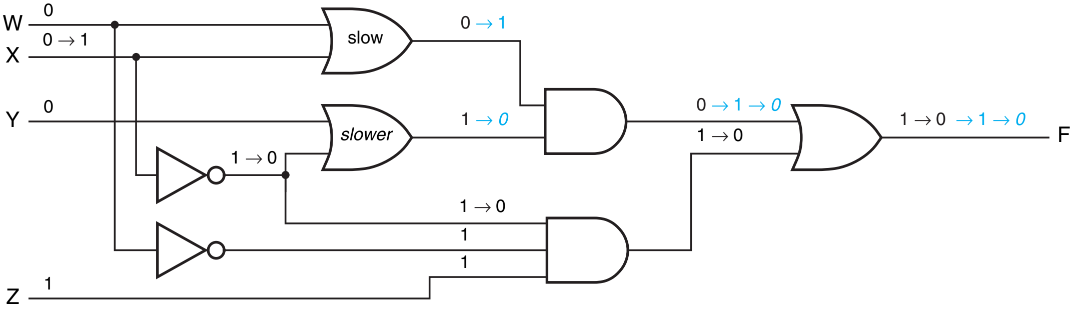


Figure 3-31 Circuit with a dynamic hazard.

MOST HAZARDS ARE NOT HAZARDOUS!

Any combinational circuit can be analyzed for the presence of hazards. However, a well-designed, *synchronous* digital system is structured so that hazard analysis is not needed for most of its circuits. In a synchronous system, all of the inputs to a combinational circuit are changed at a particular time, and the outputs are not "looked at" until they have had time to settle to a steady-state value. Hazard analysis and elimination are typically needed only in the design of asynchronous sequential circuits, like the feedback sequential circuits discussed in Section 10.8. You'll rarely need to design such a circuit, but if you do, an understanding of hazards will be essential for a reliable result.

of the "slower" OR gate changes, creating the transitions shown in italic color, and the output goes to its final state of 0.

Dynamic hazards do not occur in a properly designed two-level AND-OR or OR-AND circuit, that is, one in which no variable and its complement are connected to the same first-level gate.

*3.4.4 Designing Hazard-Free Circuits

Only a few situations, such as the design of feedback sequential circuits, require hazard-free combinational circuits. Methods for finding hazards in arbitrary circuits, described in the References, are rather difficult to use. So, when you need a hazard-free design, it's best to use a circuit structure that's easy to analyze.

complete sum

In particular, we have indicated that a properly designed two-level AND-OR circuit has no static-0 or dynamic hazards. Static-1 hazards may exist in such a circuit, but they can be found and eliminated using the map method described earlier. If cost is not a problem, then a brute-force method of obtaining a hazard-free realization is to use the *complete sum*—the sum of all of the prime implicants of the logic function (see previous editions of this book as well as Exercise 3.53). In a dual manner, a hazard-free two-level OR-AND circuit can be designed for any logic function. Finally, note that everything we've said about AND-OR circuits naturally applies to the corresponding NAND-NAND designs, and about OR-AND applies to NOR-NOR.

References

A historical description of Boole's development of "the science of Logic" appears in *The Computer from Pascal to von Neumann* by Herman H. Goldstine (Princeton University Press, 1972). Claude E. Shannon showed how Boole's work could be applied to logic circuits in "A Symbolic Analysis of Relay and Switching Circuits" (*Trans. AIEE,* Vol. 57, 1938, pp. 713–723).

Although the two-valued Boolean algebra is the basis for switching algebra, a Boolean algebra need not have only two values. Boolean algebras with 2^n values exist for every integer n; for example, see *Discrete Mathematical*

Structures and Their Applications by Harold S. Stone (SRA, 1973). Such algebras may be formally defined using the *Huntington postulates* devised by E. V. Huntington in 1907; for example, see *Digital Design* by M. Morris Mano and Michael D. Ciletti (Pearson, 2013, fifth edition). Our engineering-style, "direct" development of switching algebra follows that of Edward J. McCluskey in his *Introduction to the Theory of Switching Circuits* (McGraw-Hill, 1965) and *Logic Design Principles* (Prentice Hall, 1986).

Huntington postulates

A graphical method for simplifying Boolean functions was proposed by E. W. Veitch in "A Chart Method for Simplifying Boolean Functions" (*Proc. ACM*, May 1952, pp. 127–133). His *Veitch diagram* actually reinvented a chart proposed by an English archaeologist, A. Marquand ("On Logical Diagrams for *n* Terms," *Philosophical Magazine* XII, 1881, pp. 266–270). The Veitch diagram or Marquand chart uses "natural" binary counting order for its rows and columns, with the result that some adjacent rows and columns differ in more than one value, and product terms do not always cover adjacent cells.

The Karnaugh-map method for minimizing combinational logic functions was introduced by Maurice Karnaugh in "A Map Method for Synthesis of Combinational Logic Circuits" (*Trans. AIEE, Comm. and Electron.*, Vol. 72, Part I, November 1953, pp. 593–599). Previous editions of the book you're reading and other books describe the method in more detail.

Minimization can be performed on logic functions of an arbitrarily large number of variables (at least in principle) using a tabular method called the *Quine-McCluskey algorithm*. The method was originally developed as a paper-and-pencil tabular procedure, but like all algorithms, it can be translated into a computer program. Previous editions of this book give details on the operation of such a program. As discussed there, both the program's data structures and its execution time can become quite large for even modest logic functions, growing exponentially with the number of inputs. So, even with very fast computers and gigabytes of main memory, this algorithm is practical only for logic functions with a relatively small number of inputs (a dozen or so).

Quine-McCluskey algorithm

A different, heuristic minimization algorithm called Espresso was later developed by Robert K. Brayton and others, as described in *Logic Minimization Algorithms for VLSI Synthesis* (Kluwer Academic Publishers, 1984). While not guaranteed always to find the smallest two-level realization of a logic function, Espresso minimizes even large functions using a practical amount of memory and computation time, and can find close to minimal implementations. Therefore, Espresso or one of its derivatives is typically used at least as a first step to minimize logic functions in most of today's logic synthesis tools.

After minimization, modern synthesis tools perform additional steps, such as factoring, to address the limitations of various implementation technologies, like the maximum number of inputs in a single gate or other logic building block. Such steps are described in *Synthesis and Optimization of Digital Circuits* by Giovanni De Michelli (McGraw-Hill, 1994).

In this chapter we described a map method for finding static hazards in two-level AND-OR and OR-AND circuits, but any combinational circuit can be analyzed for hazards. In both his 1965 and 1986 books, McCluskey defines the *0-set* and *1-sets* of a circuit and shows how they can be used to find static hazards. He also defines *P-sets* and *S-sets* and shows how they can be used to find dynamic hazards.

0-set
1-set
P-set
S-set

Many deeper and varied aspects of switching theory have been omitted from this book but have been beaten to death in other books and literature. A good starting point for an academic study of classical switching theory is in *Switching and Finite Automata Theory*, by Zvi Kohavi and Niraj K. Jha (Cambridge University Press, 2010, third edition), which includes material on set theory, symmetric networks, functional decomposition, threshold logic, fault detection, and path sensitization. Another area of great academic interest is nonbinary *multiple-valued logic*, in which each signal line can take on more than two values, typically four, and new logic operations are defined to operate directly on multivalued variables. But the only practical use of multivalued logic so far has been in memories, such as MLC flash EPROMs, that use four discrete analog levels to store two bits of information in each physical memory cell.

multiple-valued logic

Drill Problems

3.1 Using variables ENGR, POET, and RHYME, write a logic expression that is 1 for poets who don't know how to rhyme and digital designers who like to come up with rhyming signal names.

3.2 Prove theorems T2–T5 using perfect induction.

3.3 Prove theorems T1D–T3D and T5D using perfect induction.

3.4 Prove theorems T6–T9 using perfect induction.

3.5 According to DeMorgan's theorem, the complement of $X + Y \cdot Z$ is $X' \cdot Y' + Z'$. Yet both functions are 1 for $XYZ = 110$. How can both a function and its complement be 1 for the same input combination? What's wrong here?

3.6 Use the theorems of switching algebra to simplify each of the following logic functions:

(a) $F = W \cdot X \cdot Y \cdot Z \cdot (W \cdot X \cdot Y \cdot Z' + W \cdot X' \cdot Y \cdot Z + W' \cdot X \cdot Y \cdot Z + W \cdot X \cdot Y' \cdot Z)$

(b) $F = A \cdot B + A \cdot B \cdot C' \cdot D + A \cdot B \cdot D \cdot E' + A \cdot B \cdot C' \cdot E + C' \cdot D \cdot E$

(c) $F = M \cdot N \cdot O + Q' \cdot P' \cdot N' + P \cdot R \cdot M + Q' \cdot O \cdot M \cdot P' + M \cdot R$

3.7 Write the truth table for each of the following logic functions:

(a) $F = X' \cdot Y + X' \cdot Y' \cdot Z$ (b) $F = W' \cdot X + Y' \cdot Z' + X' \cdot Z$

(c) $F = W + X' \cdot (Y' + Z)$ (d) $F = A \cdot B + B' \cdot C + C' \cdot D + D' \cdot A$

(e) $F = V \cdot W + X' \cdot Y' \cdot Z$ (f) $F = (A' + B' + C \cdot D) \cdot (B + C' + D' \cdot E')$

(g) $F = (W \cdot X)' \cdot (Y' + Z')'$ (h) $F = (((A + B)' + C')' + D)'$

(i) $F = (A' + B + C) \cdot (A + B' + D') \cdot (B + C' + D') \cdot (A + B + C + D)$

3.8 Write the truth table for each of the following logic functions:

(a) $F = X' \cdot Y' \cdot Z' + X \cdot Y \cdot Z + X \cdot Y' \cdot Z'$ (b) $F = M' \cdot N' + M \cdot P' + N \cdot P'$

(c) $F = A \cdot B + A \cdot B' \cdot C' + A' \cdot B \cdot C'$ (d) $F = A' \cdot B \cdot (C \cdot B \cdot A' + B' \cdot C')$

(e) $F = X \cdot Y \cdot (X' \cdot Y \cdot Z + X \cdot Y' \cdot Z + X' \cdot Y \cdot Z' + X \cdot Y \cdot Z)$ (f) $F = M \cdot N + M' \cdot N' \cdot P$

(g) $F = (A + A') \cdot B + B' \cdot A \cdot C + C' \cdot (A + B') \cdot (A' + B)$ (h) $F = X \cdot Y' + Y \cdot Z + Z' \cdot X'$

3.9 Write the canonical sum and product for each of the following logic functions:

(a) $F = \Sigma_{X,Y}(1,2)$ (b) $F = \prod_{A,B}(0,1,2)$

(c) $F = \Sigma_{A,B,C}(2,4,6,7)$ (d) $F = \prod_{W,X,Y}(0,1,3,4,5)$

(e) $F = X + Y' \cdot Z'$ (f) $F = V' + (W' \cdot X)'$

3.10 Write the canonical sum and product for each of the following logic functions:

(a) $F = \Sigma_{X,Y,Z}(0,1,3)$ (b) $F = \prod_{A,B,C}(0,2,4)$

(c) $F = \Sigma_{A,B,C,D}(1,2,6,7)$ (d) $F = \prod_{M,N,P}(0,2,3,6,7)$

(e) $F = X + Y' \cdot Z + Y \cdot Z'$ (f) $F = A' \cdot B + B \cdot C + A$

3.11 Mathematicians will tell you that “1” is not really a prime number. Rewrite the minterm list and the canonical sum and state how to modify the logic diagram of the prime-number-detector example on page 111, assuming that “1” is not prime.

3.12 If the canonical sum for an *n*-input logic function is also a minimal sum, how many literals are in each product term of the sum? Might there be any other minimal sums in this case?

3.13 Give two reasons why the cost of input inverters typically is not considered in logic minimization.

3.14 Re-do the prime-number-detector minimization example of Figure 3-24, assuming “1” is not a prime number. *Hint:* There are two correct answers.

3.15 Give another name for a 2-input even-parity function. *Hint*: The answer appears in this chapter’s exercises.

3.16 For each of the following logic expressions, use a Karnaugh map to find all of the static hazards in the corresponding two-level AND-OR circuit, and design a hazard-free circuit that realizes the same logic function:

(a) $F = W \cdot X + W' \cdot Y'$ (b) $F = W \cdot X' \cdot Y' + X \cdot Y' \cdot Z + X \cdot Y$

(c) $F = W \cdot Y + W' \cdot Z' + X \cdot Y' \cdot Z$ (d) $F = W' \cdot X' + Y' \cdot Z + W' \cdot X \cdot Y \cdot Z + W \cdot X \cdot Y \cdot Z'$

(e) $F = W' \cdot Y + X' \cdot Y' + W \cdot X \cdot Z$ (f) $F = W' \cdot X + Y' \cdot Z + W \cdot X \cdot Y \cdot Z + W \cdot X' \cdot Y \cdot Z'$

(g) $F = W \cdot X' \cdot Y' + X \cdot Y' \cdot Z + X \cdot Y$

Exercises

3.17 Design a non-trivial-looking logic circuit that contains a feedback loop but has an output that depends only on its current input.

3.18 Prove the combining theorem T10 without using perfect induction, but assuming that theorems T1–T9 and T1D–T9D are true.

3.19 Prove that $(X + Y') \cdot Y = X \cdot Y$ *without* using perfect induction. You may assume that theorems T1–T11 and T1D–T11D are true.

3.20 Prove that $(X + Y) \cdot (X' + Z) = X \cdot Z + X' \cdot Y$ *without* using perfect induction. You may assume that theorems T1–T11 and T1D–T11D are true.

3.21 Show that an n-input OR gate can be replaced by $(n-1)$ 2-input OR gates. Can the same statement be made for NOR gates? Justify your answer.

3.22 How many physically different ways are there to realize $V \cdot W \cdot X \cdot Y \cdot Z$ using four 2-input AND gates? Justify your answer.

3.23 Use switching algebra to prove that tying together two inputs of an $(n+1)$-input AND or OR gate gives it the functionality of an n-input gate.

3.24 Prove DeMorgan's theorems (T13 and T13D) using finite induction.

3.25 Use the theorems of switching algebra to rewrite the following expression using as few terms and inversions as possible (complemented parentheses are allowed):

$B' \cdot C + A \cdot C \cdot D' + A' \cdot C + D \cdot B' + E \cdot (A + C) \cdot (A' + D')$

3.26 Prove Shannon's expansion theorems. (*Hint:* Don't get carried away; it's easy.)

generalized Shannon-expansion theorems

3.27 The *generalized Shannon expansion theorems* "pull out" not just one but i variables so that a logic function can be expressed as a sum or product of 2^i terms. Figure out and state the generalized Shannon expansion theorems.

3.28 Show how the generalized Shannon expansion theorems lead to the canonical sum and canonical product representations of logic functions.

3.29 Prove or disprove the following propositions:

(a) Let A and B be switching-algebra *variables*. Then $A \cdot B = 0$ and $A + B = 1$ implies that $A = B'$.

(b) Let X and Y be switching-algebra *expressions*. Then $X \cdot Y = 0$ and $X + Y = 1$ implies that $X = Y'$.

Exclusive OR (XOR) gate

3.30 An *Exclusive OR (XOR) gate* is a 2-input gate whose output is 1 if and only if exactly one of its inputs is 1. Write a truth table, sum-of-products expression, and corresponding AND-OR circuit for the Exclusive OR function.

Exclusive NOR (XNOR) gate

3.31 An *Exclusive NOR (XNOR) gate* is a 2-input gate whose output is 1 if and only if both of its inputs are equal. Write a truth table, sum-of-products expression, and corresponding AND-OR circuit for the Exclusive NOR function.

3.32 From the point of view of switching algebra, what is the function of a 2-input XNOR gate whose inputs are tied together? How might the output behavior of a real XNOR gate differ?

complete set

3.33 Any set of logic-gate types that can realize any logic function is called a *complete set* of logic gates. For example, 2-input AND gates, 2-input OR gates, and inverters are a complete set, because any logic function can be expressed as a sum of products of variables and their complements, and AND and OR gates with any number of inputs can be made from 2-input gates. Do 2-input NOR gates form a complete set of logic gates? Prove your answer.

3.34 Do 2-input AND gates which have one input inverted form a complete set of logic gates? Prove your answer. Why might this type of gate be called an "inhibit" gate? Does this mean that a standard AND gate could be called "uninhibited"?

3.35 Do 2-input XOR gates form a complete set of logic gates? Prove your answer.

3.36 For each of the following descriptions of a combinational logic function, name the function's inputs and output and give their meanings. Then, fully specify the function using a truth table or logic equations. In the second approach, you may use intermediate variables if they are useful to simplify the problem.

(a) Specify the on/off control signal for the dome light in a typical car.

(b) Specify a signal that is 1 if and only if two 2-bit numbers *N* and *M* are equal.

(c) In a certain nerdy family, each person *P* is identified by their generation *PG* (0 being the parents) and their sex *PS* (for simplicity, just one bit, please); each child was also given a unique 2-bit identifier *PN* at birth, starting at 00. Specify a function that is 1 if and only if person *P* is a daughter of person *Q*.

(d) Repeat problem (c) for a function that is 1 if and only if person *P* is the father of person *Q*.

(e) Repeat problem (c) for a function that is 1 if and only if person *P* is a younger brother of person *Q*.

(f) Repeat problem (c) for a function that is 1 if and only if persons *P* and *Q* are the parents.

BUT

BUT gate

3.37 Some people think that there are *four* basic logic functions, AND, OR, NOT, and *BUT*. Figure X3.37 is a possible symbol for a 4-input, 2-output *BUT gate*. Invent a useful, nontrivial function for the BUT gate to perform. The function should have something to do with the name (BUT). Keep in mind that, due to the symmetry of the symbol, the function should be symmetric with respect to the A and B inputs of each section and with respect to sections 1 and 2. Describe your BUT's function and write its truth table.

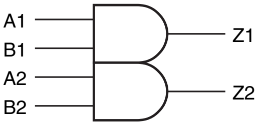


Figure X3.37

3.38 Write logic expressions for the Z1 and Z2 outputs of the BUT gate you designed in the preceding exercise, and draw a corresponding logic diagram using AND gates, OR gates, and inverters.

3.39 How many different nontrivial logic functions are there of *n* variables? Here, "nontrivial" means that all of the variables affect the output.

3.40 Most students have no problem using theorem T8 to "multiply out" logic expressions, but many develop a mental block if they try to use theorem T8D to "add out" a logic expression. How can duality be used to overcome this problem?

3.41 Describe how to adapt any software tool that synthesizes AND-OR logic instead to synthesize OR-AND logic.

3.42 Prove that $\mathrm{F}^{\mathrm{D}}(\mathrm{X}_1, \mathrm{X}_2, \ldots, \mathrm{X}_n) = [\mathrm{F}(\mathrm{X}_1', \mathrm{X}_2', \ldots, \mathrm{X}_n')']$.

self-dual logic function

⊕

3.43 A *self-dual logic function* is a function F such that $\mathrm{F} = \mathrm{F}^{\mathrm{D}}$. Which of the following functions are self-dual? (The symbol ⊕ denotes the Exclusive OR (XOR) operation.)

(a) $\mathrm{F} = \mathrm{X}$

(b) $\mathrm{F} = \Sigma_{\mathrm{X,Y,Z}}(0,3,5,6)$

(c) $\mathrm{F} = \mathrm{X} \cdot \mathrm{Y}' + \mathrm{X}' \cdot \mathrm{Y}$

(d) $\mathrm{F} = \mathrm{W} \cdot (\mathrm{X} \oplus \mathrm{Y} \oplus \mathrm{Z}) + \mathrm{W}' \cdot (\mathrm{X} \oplus \mathrm{Y} \oplus \mathrm{Z})'$

(e) A function F of 7 variables such that F = 1 if and only if 4 or more of the variables are 1

(f) A function F of 10 variables such that F = 1 if and only if 5 or more of the variables are 1

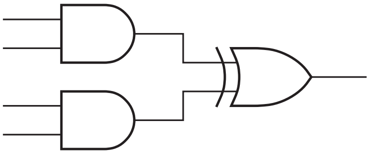
3.44 Assuming the signal delay through NAND gates and inverters is 5 ns, NOR gates is 6 ns, and noninverting gates is 9 ns, what is the total delay to the slowest output in each of the circuits in Figure 3-21(a), (c), and (d).

3.45 How many self-dual logic functions of n input variables are there? (*Hint:* Consider the structure of the truth table of a self-dual function.)

3.46 Prove that any n-input logic function $\mathsf{F}(\mathsf{X}_1,\ldots,\mathsf{X}_n)$ that can be written in the form $\mathsf{F} = \mathsf{X}_1 \cdot \mathsf{G}(\mathsf{X}_2,\ldots,\mathsf{X}_n) + \mathsf{X}_1' \cdot \mathsf{G}^{\mathsf{D}}(\mathsf{X}_2,\ldots,\mathsf{X}_n)$ is self-dual.

3.47 Assign variables to the inputs of the AND-XOR circuit in Figure X3.47 so that its output is $\mathsf{F} = \Sigma_{\mathsf{W,X,Y,Z}}(6,7,12,13)$. You may use a Karnaugh map if it helps you. What is the solution if the AND gates are changed to NAND gates?

Figure X3.47



distinguished 1-cell

essential prime implicant

3.48 A *distinguished 1-cell* in a logic function's Karnaugh map is a cell (and a corresponding input combination) that is covered by only one prime implicant. Such an *essential prime implicant* must be present in any minimal sum for the function. Thus, an efficient minimization algorithm looks first for essential prime implicants, and then selects additional prime implicants only as needed for 1-cells that are still uncovered, if any. The following logic functions all have one or more essential prime implicants; find their minimal sums:

(a) $\mathsf{F} = \Sigma_{\mathsf{X,Y,Z}}(1,3,5,6,7)$ (b) $\mathsf{F} = \Sigma_{\mathsf{W,X,Y,Z}}(1,4,5,6,7,9,14,15)$

(c) $\mathsf{F} = \prod_{\mathsf{W,X,Y}}(1,4,5,\ 6,\ 7)$ (d) $\mathsf{F} = \Sigma_{\mathsf{W,X,Y,Z}}(0,1,6,7,8,9,14,15)$

(e) $\mathsf{F} = \prod_{\mathsf{A,B,C,D}}(4,\ 5,\ 6,\ 13,15)$ (f) $\mathsf{F} = \Sigma_{\mathsf{A,B,C,D}}(4,5,6,\ 11,\ 13,14,15)$

3.49 A 3-bit "comparator" circuit receives two 3-bit numbers, $\mathsf{P} = \mathsf{P}_2\mathsf{P}_1\mathsf{P}_0$ and $\mathsf{Q} = \mathsf{Q}_2\mathsf{Q}_1\mathsf{Q}_0$. Design a minimal sum-of-products circuit that produces a 1 output if and only if $\mathsf{P} < \mathsf{Q}$.

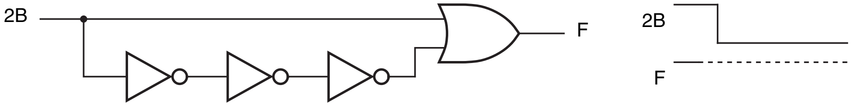
3.50 Algebraically prove whether or not the following expression is minimal. That is, can any product term be eliminated and if not, can any input be removed from any product term?

$$\mathsf{F} \;=\; \mathsf{C} \cdot \mathsf{D} \cdot \mathsf{E}' \cdot \mathsf{F}' \cdot \mathsf{G} + \mathsf{B} \cdot \mathsf{C} \cdot \mathsf{E} \cdot \mathsf{F}' \cdot \mathsf{G} + \mathsf{A} \cdot \mathsf{B} \cdot \mathsf{C} \cdot \mathsf{D} \cdot \mathsf{F}' \cdot \mathsf{G}$$

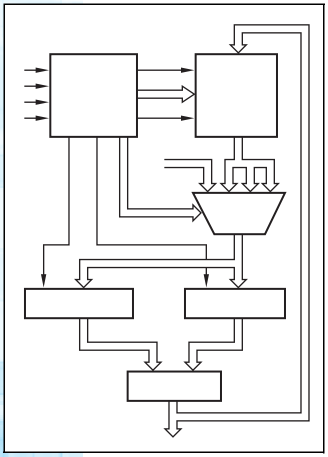
3.51 Exhibit a 4-input logic function, other than the one in Figure 3-25, whose Karnaugh map is a "checkerboard" with eight minterms. Does this logic function have a concise name?

3.52 (*Hamlet circuit.*) Complete the timing diagram and explain the function of the circuit in Figure X3.52. Where does the circuit get its name?

Figure X3.52



3.53 Prove that a two-level AND-OR circuit corresponding to the complete sum of a logic function is always hazard free.



chapter 4

Digital Design Practices

This book aims to show you both the theoretical principles and the practices used in modern digital design. This chapter focuses on general practices and a few that are particular to combinational circuits. We'll wait until Chapter 13 to discuss practices that are particular to sequential-circuit design.

The first topic we will discuss here is one that you don't very often see in engineering texts, namely the documentation practices that engineers use to ensure that their designs are correct, manufacturable, and maintainable. The next topic is circuit timing, a crucial element for successful digital design. Finally, we give an introduction to HDL-based digital design, and "design flow" in an HDL-based environment.

4.1 Documentation Standards

Good documentation is essential for correct design and efficient maintenance of digital systems. In addition to being accurate and complete, documentation must be somewhat instructive, so that a test engineer, maintenance technician, or even the original design engineer (six months after designing the circuit) can figure out how the system works just by reading the documentation.

Although the type of documentation depends on system complexity and the engineering and manufacturing environments, a documentation package should generally contain at least the following items:

specification (spec)

1. A *specification* (also known as a *spec*) describes exactly what the circuit or system is supposed to do, including a description of all inputs and outputs ("interfaces") and the functions that are to be performed. Note the spec doesn't have to specify *how* the system achieves its results, just *what* the results are supposed to be. However, in many companies it is a common practice also to incorporate one or more of the following documents into the spec to describe how the system works.

block diagram

2. A *block diagram* is an informal pictorial description of the system's major functional modules and their basic interconnections.

logic-device description

3. A *logic-device description* describes the functions of each "custom" logic device used in the system. ("Standard" devices are described by data sheets or user manuals provided by their manufacturers.) Custom devices include application-specific integrated circuits (ASICs), field-programmable gate arrays (FPGAs), and programmable logic devices (PLDs and CPLDs).

   At a high level, device descriptions are written in English, but the internals are often specified in an HDL like Verilog. Some internals may also be specified by logic diagrams, equations, state tables, or state diagrams. Sometimes, a conventional programming language like C may be used to model the operation of a circuit or to specify parts of its behavior.

schematic diagram

logic diagram

4. A *schematic diagram* is a formal specification of the electrical components of the system, their interconnections, and related details needed to build the system. We've been using the term *logic diagram* for a less formal drawing that does not have quite this level of detail.

   bill of materials (BOM)

   In board-level design, the schematic is usually created by the designer and should include IC types, reference designators, signal names, and the pin numbers where signals appear on the physical devices. Most schematic-drawing programs have the ability to generate a *bill of materials (BOM)* from the schematic; this tells the purchasing department what electronic components they have to acquire to build the system.

   In FPGA- and PLD-based design, the internals of the FPGA or PLD are usually specified with an HDL like Verilog and no schematic is needed for the internals. But the EDA tools may have the ability to create one after the language-specified design has been implemented. In addition to signal names, such a schematic would include the names, types, and possibly the on-chip locations of resources used by the implemented design. Such a schematic may be useful in optimization and debugging of the design.

timing diagram

5. A *timing diagram* shows the values of various logic signals as a function of time, including the cause-and-effect delays between critical signals.

DOCUMENTS ON-LINE Professional engineering documentation nowadays is carefully maintained on corporate intranets, so it's very useful to include pointers, like URLs, in circuit specifications and descriptions so that references can be easily located. Of course, URLs sometimes change as a result of network and server reconfiguration, so documents might be referenced instead by a permanent number assigned by the company's document-control system.

On-line documentation is so important and authoritative in one company that the footer on every page of every specification contains the warning that "A printed version of this document is an uncontrolled copy." That is, a printed copy could very well be obsolete.

6. A *circuit description* is a narrative text document that, in conjunction with the other documentation, explains how the circuit works internally. The circuit description should list any assumptions and potential pitfalls in the circuit's design and operation, and point out the use of any nonobvious design "tricks." A good circuit description also contains definitions of acronyms and other specialized terms and has references to related documents. Each "custom" logic device in the system should have its own circuit description.

circuit description

7. A *test plan* describes the methods and resources that will be needed to test the system for proper operation, both before and after it is physically built.

test plan

You've probably already seen block diagrams in many contexts. We'll first present a few rules for drawing them, and then in the rest of this section, we'll focus on schematics for combinational logic circuits. Section 4.2.1 introduces timing diagrams. Logic-device descriptions in the form of Verilog models will be covered in Chapter 5, with many examples to follow in later chapters. In Section 6.1.2, we'll show how a C program can be used to generate and specify the contents of a read-only memory that solves a design problem.

The circuit description is sometimes overlooked, but is very important in practice. Just as an experienced programmer creates a program design document before beginning to write code, an experienced logic designer starts writing the

DON'T FORGET TO WRITE! In order to create successful products, digital designers must develop their language and writing skills, especially in the area of *logical* outlining and organization. The most successful digital designers (and later, project leaders, system architects, and entrepreneurs) are the ones who communicate their ideas, proposals, and decisions effectively to others. Even though it's a lot of fun to tinker in the digital design lab, don't use that as an excuse to shortchange your writing and communications courses and projects!

circuit description before drawing a schematic or writing HDL code. Sadly, the circuit description is sometimes the last document to be created, and sometimes it's never written at all. A circuit without a description is difficult to debug, manufacture, test, maintain, modify, and enhance, even by the original designer six months after it's done.

Overall test plans are beyond the scope of this text, but we'll cover one aspect of them, specifically test benches for Verilog models, in quite a bit of detail in later chapters.

4.1.1 Block Diagrams

block diagram

A *block diagram* shows the inputs, outputs, functional modules, internal data paths, and important control signals of a system. In general, it should not be so detailed that it occupies more than one page, yet it must not be too vague. A small block diagram may have three to six blocks, while a large one may have 10 to 15 blocks, depending on system complexity. In any case, the block diagram must show the most important system elements and how they work together.

Large systems are designed and described hierarchically. At the top level, and in the corresponding block diagram, the system is partitioned into a small number of independent subsystems or blocks that interact with each other in well defined ways. Each of those subsystems or blocks is further partitioned as needed until reaching an appropriate lower level below which the details can be fully understood and designed with the available components and tools.

Figure 4-1 shows a sample block diagram. Each block is labeled with the function of the block, not the individual components that comprise it. As another example, Figure 4-2(a) shows the block-diagram symbol for a 32-bit register. If the register is to be built using four 8-bit-register components named "REG8," and this information might be important to someone reading the diagram, then it can be conveyed as shown in (b). However, splitting the block to show individual components as in (c) is generally incorrect.

bus

A *bus* is a collection of two or more related signal lines. In a block diagram, buses may be drawn with a double or heavy line, as in Figure 4-1. A slash and a number may indicate how many individual signal lines are contained in a bus. Alternatively, size may be denoted in the bus name (e.g., INBUS[31:0] or INBUS[31-0]). Active levels (defined later) and inversion bubbles may or may not appear in block diagrams; in most cases, they are unimportant at this level of detail. However, important control signals and buses should have names, usually the same names that appear in the more detailed schematic.

The flow of control and data in a block diagram should be indicated clearly. Schematic diagrams are generally drawn with signals flowing from left to right, but in block diagrams this ideal is more difficult to achieve. Inputs and outputs may be on any side of a block, and the direction of signal flow may be arbitrary. Arrowheads are often used on buses and ordinary signal lines to eliminate any ambiguity.

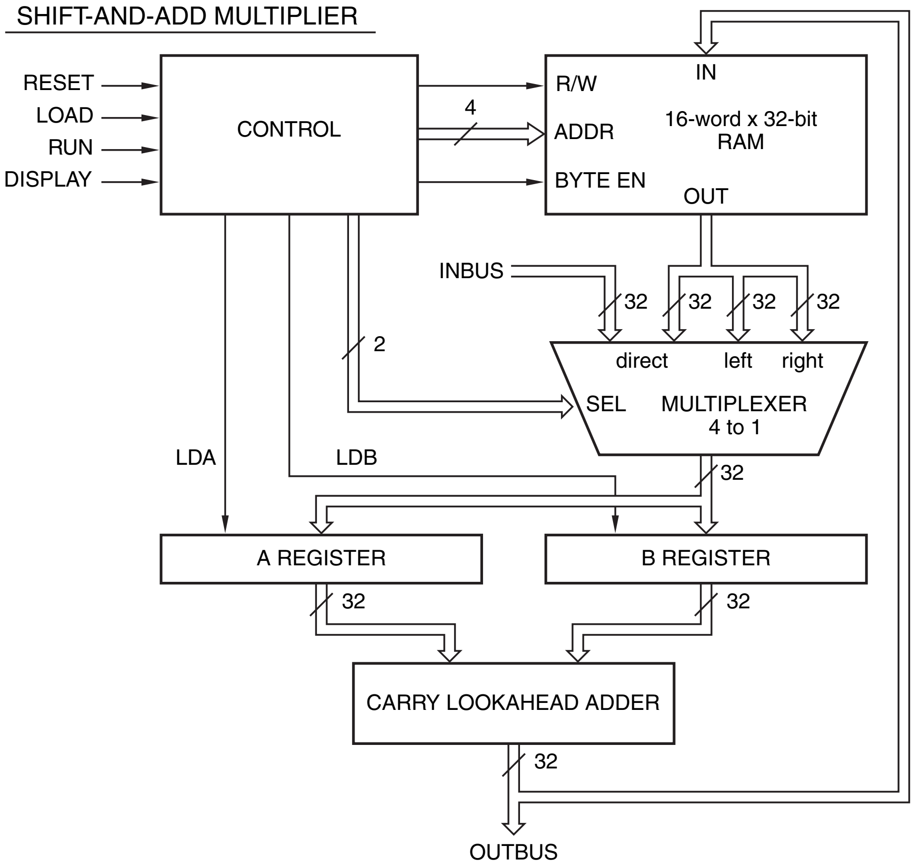


Figure 4-1
Block diagram for a digital design project.

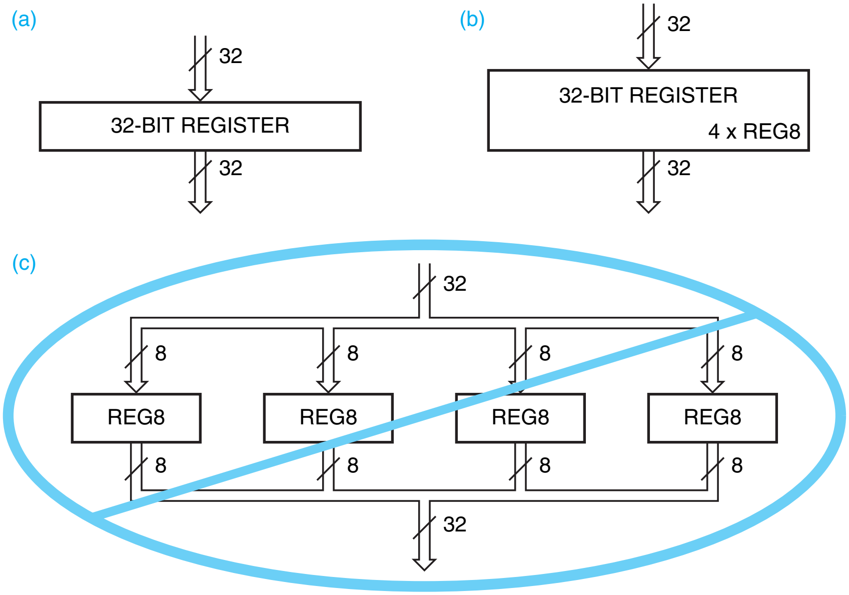
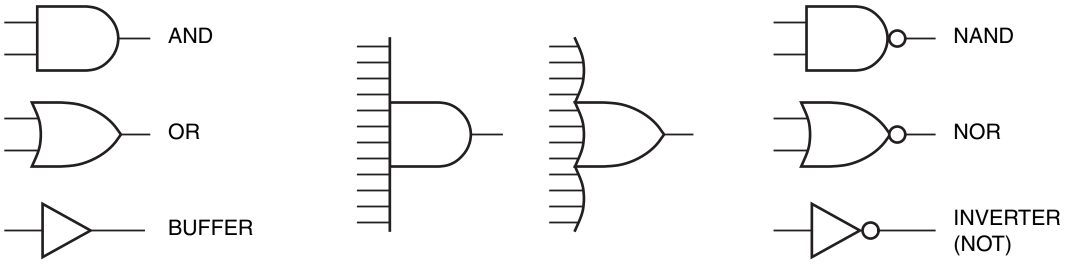


Figure 4-2
A 32-bit register block:
(a) realization unspecified;
(b) chips specified;
(c) too much detail.



(a) (b) (c)

Figure 4-3 Shapes for basic logic gates: (a) AND, OR, and buffers; (b) expansion of inputs; (c) inversion bubbles.

4.1.2 Gate Symbols

buffer

noninverting buffer

inversion bubble

We introduced logic gates in Chapters 1 and 3, and the symbol shapes for AND and OR gates are shown again in Figure 4-3(a). The figure also shows a *buffer*, sometimes called a *noninverting buffer*, which is a circuit that simply converts an electrically "weak" logic signal into a "strong" one with the same logic value. To draw logic gates with more than a few inputs, we expand the AND and OR symbols as shown in (b). A small circle, called an *inversion bubble*, denotes logical inversion or complementing and is used in the symbols for NAND and NOR gates and inverters in (c).

As shown in Section 3.1.4, we can use DeMorgan's theorem to manipulate the logic expressions for gates with complemented outputs. For example, if X and Y are the inputs of a NAND gate with output Z, then we can write

$$\begin{aligned} Z &= (X \cdot Y)' \\ &= X' + Y' \end{aligned}$$

This gives rise to two different but equally correct symbols for a NAND gate, as we demonstrated in Figure 3-3 on page 98. In fact, this sort of manipulation may be applied to gates with uncomplemented outputs as well. For example, consider the following equations for an AND gate:

$$\begin{aligned} Z &= X \cdot Y \\ &= ((X \cdot Y)')' \\ &= (X' + Y')' \end{aligned}$$

Thus, an AND gate may be symbolized as an OR gate with inversion bubbles on its inputs and output.

Equivalent symbols for standard gates that can be obtained by these manipulations are summarized in Figure 4-4. Even though both symbols in a pair represent the same logic function, the choice of one symbol or the other in a logic diagram is not arbitrary, at least not if we are adhering to good documentation standards. As we'll show in the next few subsections, proper choices of gate

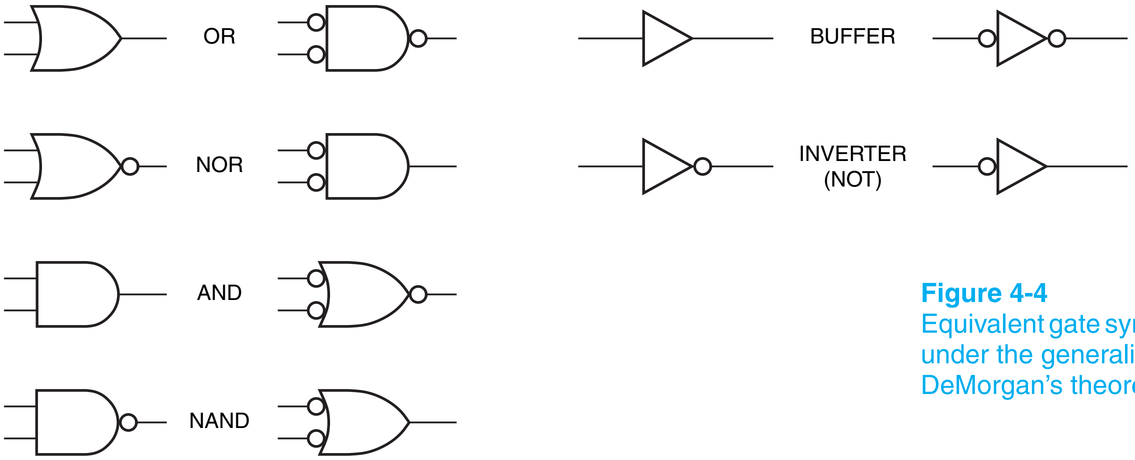


Figure 4-4
Equivalent gate symbols under the generalized DeMorgan's theorem.

symbols can make logic diagrams much easier to use and understand. In addition, corresponding choices of signal names can make both logic diagrams and HDL code more understandable.

IEEE STANDARD LOGIC SYMBOLS

Together with the American National Standards Institute (ANSI), the Institute of Electrical and Electronics Engineers (IEEE) has developed a standard set of logic symbols. The most recent revision of the standard is ANSI/IEEE Std 91-1984, *IEEE Standard Graphic Symbols for Logic Functions*, and it allows both rectangular- and distinctive-shape symbols for logic gates.

We have been using, and we'll continue to use, traditional distinctive-shape symbols in this book. The rectangular-shape symbols are described in this book's second edition, as well as at various sites on the Web.

4.1.3 Signal Names and Active Levels

Each input and output signal in a logic circuit should have a descriptive alphanumeric label, the signal's name. HDLs and most EDA programs for drawing logic circuits also allow certain special characters, such as *, _, and $, to be included in signal names. In the analysis and synthesis examples in Chapter 3, we used mostly single-character signal names (X, Y, etc.) because we were still thinking in terms of switching *algebra* and the circuits didn't do much. However, in a real system, well-chosen signal names convey information the same way that variable names in a software program do. A signal's name indicates an action that is controlled (GO, PAUSE), a condition that it detects (READY, ERROR), or data that it carries (INBUS[31:0]).

Each signal name should have an *active level* associated with it. A signal is *active high* if it performs the named action or denotes the named condition when

active level
active high

Table 4-1
Each line shows a different naming convention for active levels.

Active Low	*Active High*
READY–	READY+
ERROR.L	ERROR.H
ADDR15(L)	ADDR15(H)
RESET*	RESET
ENABLE~	ENABLE
~GO	GO
/RECEIVE	RECEIVE
TRANSMIT_L	TRANSMIT

active low
assert
negate
deassert

it is HIGH or 1. (Under the positive-logic convention, which we use throughout this book, "HIGH" and "1" are equivalent.) A signal is *active low* if it performs the named action or denotes the named condition when it is LOW or 0. A signal is said to be *asserted* when it is at its active level. A signal is said to be *negated* (or, sometimes, *deasserted*) when it is not at its active level.

active-level naming convention

The active level of each signal in a circuit is normally specified as part of its name, according to some convention. Examples of several different *active-level naming conventions* are shown in Table 4-1. The choice of one of these or other signal-naming conventions is sometimes just a matter of personal preference, but more often it is constrained by the engineering environment. Since the active-level designation is part of the signal name, the naming convention must be compatible with the input requirements of any EDA tools that will process the signal names, such as schematic editors, HDL compilers, and simulators.

_L suffix

In this text, we'll use the last convention in the table, which is compatible with modern HDLs: An active-low signal name has a suffix of _L, and an active-high signal has no suffix. The _L suffix may be read as if it were a prefix "not."

signal name
logic expression
logic equation

It's extremely important for you to understand the difference between signal names, expressions, and equations. A *signal name* is just a name—an alphanumeric label. A *logic expression* combines signal names using the operators of switching algebra—AND, OR, and NOT—as we explained and used throughout Chapter 3. A *logic equation* is an assignment of a logic expression to a signal name—it describes one signal's function in terms of other signals.

The distinction between signal names and logic expressions can be related to a concept used in computer programming languages: The lefthand side of an assignment statement contains a variable *name*, and the righthand side contains an *expression* whose value will be assigned to the named variable (for example, `Z = -(X+Y)`). In a programming language, you can't put an expression on the lefthand side of an assignment statement. In logic design, you can't use a logic expression as a signal name.

Logic signals may have names like X, READY, and GO_L. The "_L" in GO_L is just part of the signal's name, like an underscore in a variable name in a C program. There is *no* signal whose name is READY′—this is an expression, since ′ is an operator. However, there may be two signals named READY and READY_L such that READY_L = READY′ during normal operation of the circuit.

We are very careful in this book to distinguish between signal names, which are always printed in black, and logic expressions, which are always printed in color when they are written near the corresponding signal lines.

In HDL models, most signals are active-high. It's just easier to manage and understand a model and corresponding circuit when signals perform or denote their named operations or conditions when they are 1.

However, when chips or functions are interconnected on a printed circuit board or in a system, some signals, especially control signals, may be active low. This occurs because certain signals, including ones on larger-scale devices, are active low for compatibility with other devices with which they are frequently paired, or for better electrical performance in areas like noise immunity.

Examples of typical active-low signals include chip-select inputs on memories (compatibility) and reset inputs on all sorts of devices (compatibility, noise immunity, and safe operation during power-on and power-off). Thus, in HDL-based designs, you are mostly likely to see and use active-low signals only on the external pins of a targeted FPGA, ASIC, or PLD.

4.1.4 Active Levels for Pins

When we draw the outline of an AND or OR symbol, or a rectangle representing a larger-scale logic element, we think of the given logic function as occurring *inside* that symbolic outline. In Figure 4-5(a), we show the logic symbols for an AND and an OR gate and for a larger-scale element with an ENABLE input. The AND and OR gates have active-high inputs—they require 1s on their input pins (or other "wires," depending on the technology) to assert their outputs.

Likewise, the larger-scale element has an active-high ENABLE input, which must be 1 to enable the element to do its thing. In Figure 4-5(b), we show the same logic elements with active-low input and output pins. Exactly the same logic functions are performed *inside* the symbolic outlines, but the inversion bubbles indicate that 0s must now be applied to the input pins to activate the logic functions, and that the outputs are 0 when they are "doing their thing."

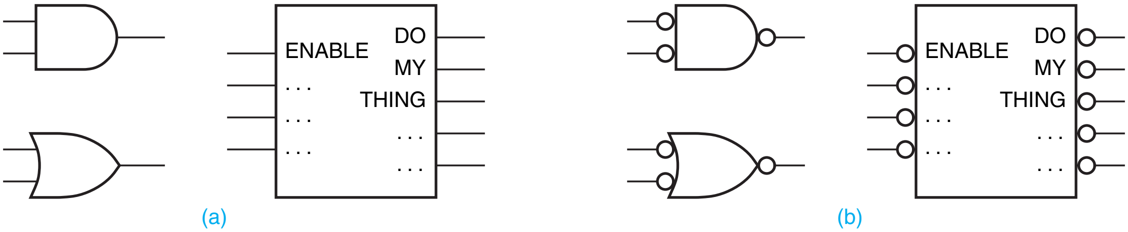


Figure 4-5 Logic symbols: (a) AND, OR, and a larger-scale logic element; (b) the same elements with active-low inputs and outputs.

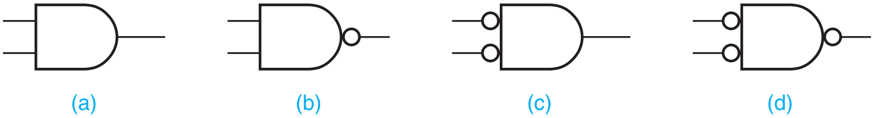


Figure 4-6 Four ways of obtaining an AND function: (a) AND gate; (b) NAND gate; (c) NOR gate; (d) OR gate.

Thus, we associate active levels with the input and output pins of gates and larger-scale logic elements. We use an inversion bubble to indicate an active-low pin and the absence of a bubble to indicate an active-high pin. For example, the AND gate in Figure 4-6(a) performs the logical AND of two active-high inputs and produces an active-high output: if both inputs are asserted (1), the output is asserted (1). The NAND gate in (b) also performs the AND function, but it produces an active-low output. Even a NOR or OR gate can be construed to perform the AND function using active-low inputs and perhaps output, as shown in (c) and (d). All four gates in the figure can be said to perform the same function: the output of each gate is asserted if both of its inputs are asserted.

Figure 4-7 shows the same idea for the OR function: The output of each gate is asserted if either of its inputs is asserted.

Sometimes a noninverting buffer is used simply to boost the fanout of a logic signal without changing its function. Figure 4-8 shows the possible logic symbols for both inverters and noninverting buffers. In terms of active levels, all of the symbols perform exactly the same function: Each asserts its output signal if and only if its input is asserted.

4.1.5 Constant Logic Signals

Sometimes, a logic signal with a constant-0 or constant-1 value is needed. For example, a larger-scale logic element may need to be "always enabled," or one or more inputs of discrete gate may be unused but the rest of the gate needs to function. A few examples are shown in Figure 4-9. Here, the little triangle

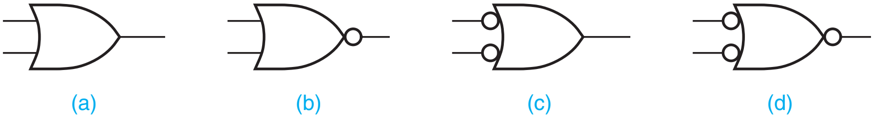


Figure 4-7 Four ways of obtaining an OR function: (a) OR gate; (b) NOR gate; (c) NAND gate; (d) AND gate.

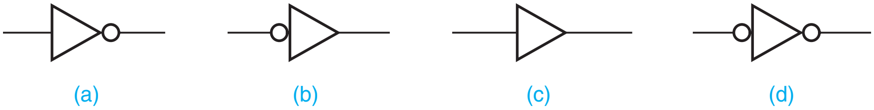


Figure 4-8 Alternate symbols: (a, b) inverters; (c, d) noninverting buffers.

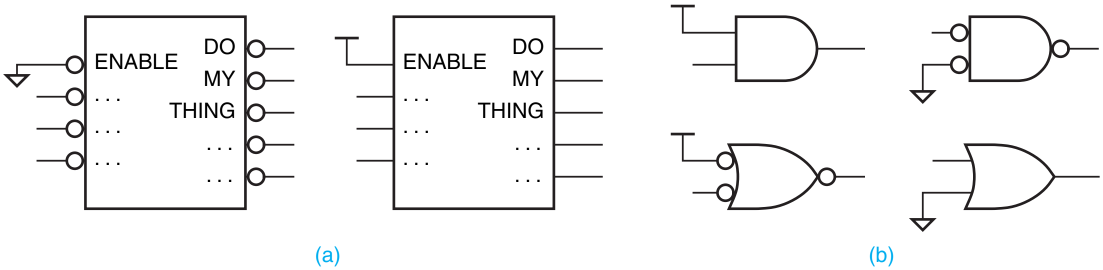


Figure 4-9 Constant 0 and 1 inputs for unused inputs: (a) with larger-scale logic element; (b) with individual gates.

pointing down is the traditional electronic symbol for "ground" or 0 volts, which is logic 0 for CMOS with a positive-logic convention. The horizontal bar is the traditional symbol for the power-supply voltage, which may vary depending on the logic family, but which is always a logic 1 for CMOS with a positive-logic convention. Ground and the power supply voltage are often called the *power-supply rails*. Note that some logic families and some design practices may require that connections to the rails be made through resistors instead of directly, for reliability or testing reasons.

power-supply rails

*4.1.6 Bubble-to-Bubble Logic Design

Experienced logic circuit designers formulate their circuits in terms of the logic functions performed *inside* the symbolic outlines. Whether you're designing with discrete gates or in an HDL, it's easiest to think of logic signals and their interactions using active-high names. However, once you're ready to realize your circuit, you may have to deal with active-low signals because of various requirements in the circuit's environment.

When you design with discrete gates, either at board or ASIC level, a key requirement is often speed. As we'll show in Section 14.1.6, inverting gates are typically faster than noninverting ones, so there's often a performance payoff in generating some signals in active-low form.

When you design with larger-scale elements, many of them may be off-the-shelf chips or other existing components that already have some inputs and outputs fixed in active-low form. The reasons that they use active-low signals may range from performance improvement to backwards compatibility to years of ingrained tradition, but in any case, you still have to deal with it.

bubble-to-bubble logic design

Bubble-to-bubble logic design is the practice of choosing logic symbols and signal names, including active-level designators, that make the function of a logic circuit easier to understand. Usually, this means choosing signal names and gate types and symbols so that most of the inversion bubbles "cancel out" and the design can be analyzed as if most of the signals were active high.

* Throughout this book, optional sections are marked with an asterisk.

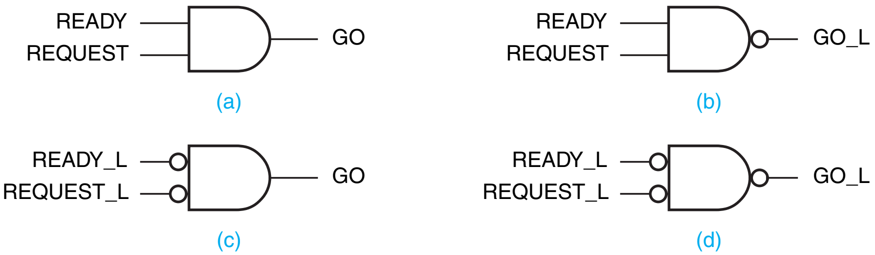


Figure 4-10 Many ways to GO: (a) active-high inputs and output; (b) active-high inputs, active-low output; (c) active-low inputs, active-high output; (d) active-low inputs and output.

For example, suppose we need to produce a signal that tells a device to "GO" when we are "READY" and we get a "REQUEST." Clearly from the problem statement, an AND function is required; in switching algebra, we would write GO = READY · REQUEST. However, we can use different gates to perform the AND function, depending on the active level required for the GO signal and the active levels of the available input signals.

Figure 4-10(a) shows the simplest case, where GO must be active-high and the available input signals are also active-high; we use an AND gate. If, on the other hand, the controlled device requires an active-low GO_L signal, we can use a NAND gate as shown in (b). If the available input signals are active-low, we can use a NOR or OR gate as shown in (c) and (d).

The active levels of available signals don't always match the active levels of available gates. For example, suppose we are given input signals READY_L (active-low) and REQUEST (active-high). Figure 4-11 shows two different ways to generate GO using an inverter to generate the active level needed for the AND function. The second way is generally preferred, since inverting gates like NOR are generally faster than noninverting ones like AND. We drew the inverter differently in each case to make the output's active level match its signal name.

To understand the benefits of bubble-to-bubble logic design, consider the circuit in Figure 4-12(a). What does it do? In Section 3.2 we showed several ways to analyze such a circuit, and we could certainly obtain a logic expression for the DATA output using these techniques. However, when the circuit is redrawn in Figure 4-12(b), the output function can be read directly from the logic diagram: the DATA output a copy of A if ASEL is asserted, else it is a copy of B. In more detail, the mental process in arriving at this result is as follows:

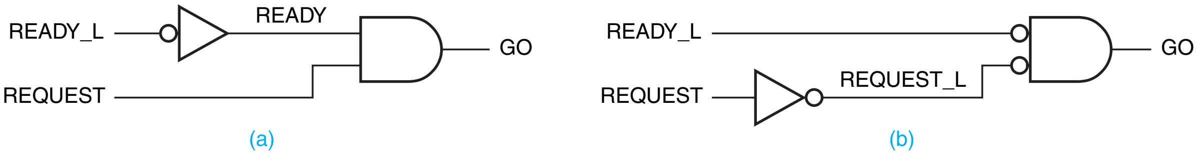


Figure 4-11 Two more ways to GO, with mixed input levels: (a) with an AND gate; (b) with a NOR gate.

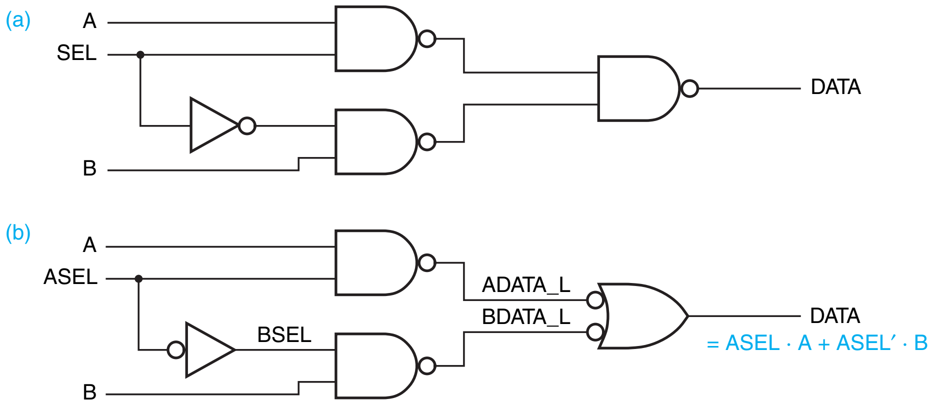


Figure 4-12 A 2-input multiplexer: (a) cryptic logic diagram; (b) proper logic diagram with named active levels and alternate logic symbols.

- If ASEL is asserted, then ADATA_L is asserted if and only if A is asserted; that is, ADATA_L is a copy of A.
- If ASEL is negated, BSEL is asserted and BDATA_L is a copy of B.
- The DATA output is asserted when either ADATA_L or BDATA_L is asserted.

Even though the logic diagram has five inversion bubbles, we mentally had to perform only one negation to understand the circuit—that BSEL is asserted if ASEL is not.

If we wish, we can write an algebraic expression for the DATA output. We use the technique of Section 3.2, simply propagating expressions through gates toward the output. In doing so, we can ignore pairs of inversion bubbles that cancel, and directly write the expression shown in color in the figure.

Another example is shown in Figure 4-13. Reading directly from the logic diagram, we see that ENABLE_L is asserted if READY_L and REQUEST_L are asserted or if TEST is asserted. The HALT output is asserted if READY_L and REQUEST_L are not both asserted or if LOCK_L is asserted. Once again, this example has only one place where a gate input's active level does not match the input signal level, and this is reflected in the verbal description of the circuit.

We can, if we wish, write algebraic equations for the ENABLE_L and HALT outputs. As we propagate expressions through gates toward the output, we obtain expressions like READY_L′ · REQUEST_L′. However, we can use our

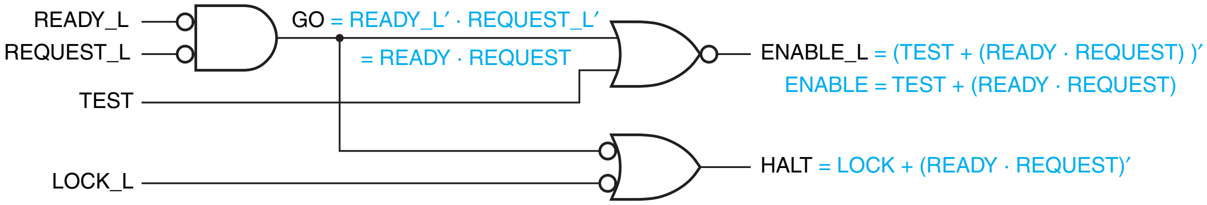

READY_L
REQUEST_L
TEST
LOCK_L
GO = READY_L′ · REQUEST_L′
= READY · REQUEST
ENABLE_L = (TEST + (READY · REQUEST))′
ENABLE = TEST + (READY · REQUEST)
HALT = LOCK + (READY · REQUEST)′

Figure 4-13 Another properly drawn logic diagram.

active-level naming convention to simplify terms like READY_L′. The circuit contains no signal with the name READY; but if it did, it would satisfy the relationship READY = READY_L′ according to the naming convention. This allows us to write the ENABLE_L and HALT equations as shown. Complementing both sides of the ENABLE_L equation, we obtain an equation that describes a hypothetical active-high ENABLE output in terms of hypothetical active-high inputs.

We'll see more examples of bubble-to-bubble logic design in Chapter 6, in both the internals of some larger-scale combinational-logic building blocks and the interconnection of multiple blocks.

BUBBLE-TO-BUBBLE LOGIC DESIGN RULES

The following rules are useful for performing bubble-to-bubble logic design:

- The signal name on a device's output should have the same active level as the device's output pin—that is, active-low if the device symbol has an inversion bubble on the output pin, and active-high if not.
- If the active level of an input signal is the same as that of the input pin to which it is connected, then the logic function inside the symbolic outline is activated when the signal is asserted. This is the most common case in a logic diagram.
- If the active level of an input signal is the opposite of the input pin's, then the logic function inside the symbolic outline is activated when the signal is negated. This case should be avoided if possible because it forces us to mentally keep track of a logical negation to understand the circuit.

4.1.7 Signal Naming in HDL Models

We have already emphasized two important aspects of proper signal naming—picking names that correspond to the function of a signal, and indicating the signal's active level. There are additional aspects to consider when signal names will be used in HDL models and with various EDA tools.

Probably the most important aspect to consider is compatibility of signal names among a collection of different EDA tools. Each tool has its own rules on what it accepts as legal identifiers, and what other characters it might interpret as giving special commands or information to the tool, such as macros, compiler directives, and so on. Therefore, it's important to construct your signal names only from a restricted, "least-common-denominator" set of characters, accepted by *all* tools. The safest such set is usually letters, digits, and the underscore "_", and that's what we use in this book.

Tools may also differ in what characters may be used to begin an identifier; for example, some may allow a leading digit, and others may not. Thus, it's also best to begin each signal name with a letter. Some or perhaps even all of the tools that you use may also allow a leading underscore. But such signal names may, by convention, have special meaning or significance in some environments; for

NAME THAT SIGNAL! Although it is absolutely necessary to name only a circuit's main inputs and outputs, most logic designers find it useful to name internal signals as well. During circuit debugging, it's nice to have a meaningful name to use when pointing to an internal signal that's behaving strangely.

Most EDA tools automatically generate labels for unnamed signals, but a user-chosen name is preferable to a computer-generated one like XSIG1057.

example, they may be used for signal names that are created by the compiler or synthesizer. So, it is still best to begin your own signal names with letters.

There is also the issue of case for letters—upper or lower. In some HDLs, including Verilog, case is significant—`sig`, `Sig`, and `SIG` are three different signals. In others (like VHDL), it is not. So, it's best not to define multiple signal names that differ only in case; the distinction may be lost on some of the people who need to understand your design.

There's another aspect to the use of case, as it affects the readability of programs. Historically, software programming languages have used various case conventions to distinguish different language elements. A popular convention in HDL code is to use `UPPERCASE` for constants and other definitions, `lowercase` for signal names, and `color` for reserved words. Using color is easy because typical modern, programming-language aware text editors recognize reserved words and automatically render them in color. In fact, they may also recognize the syntax for comments and render them in another color.

We've shown many examples of using the suffix "`_L`" to denote an active-low signal. But now if you consider the use of lowercase signal names, this suffix loses a bit of its appeal because the need either to shift when typing the suffix or to suffer eyestrain to distinguish between "`_l`" and "`_1`". So, some design environments may use a different suffix, like "`_n`", to denote an active-low signal.

Some design environments may have conventions for additional suffixes, before or after the active-level suffix, to convey additional information. For example, suffixes "`_1`", "`_2`", and so on might be used to name multiple copies of a signal that has been replicated for fanout purposes.

JUST IN CASE We use a couple of case conventions in this book, just to keep you flexible. In source-code listings, we use `lowercase color` for reserved words. We normally give signal names in `UPPERCASE` in small examples typically with accompanying logic diagrams or equations that also have signal names in UPPERCASE. Later in the book, we'll have some larger Verilog examples with signal names in `lowercase`—as is typical in industry for HDL models—and we may define constant names in `UPPERCASE`. So, overall in this book, you can't rely on case as denoting anything special.

Signal names that are used in an HDL model have limited “scope,” just like variables in a software programming language. So, it’s possible for the same signal name to be reused in multiple modules, and for it to denote completely independent signals. Just as in software programs, though, one must be careful.

In large projects with multiple hardware designers, it’s difficult to ensure that designers use unique names for commonly needed functions. For example, each module may have a reset input signal named “`reset`”. It’s true that because of scope rules, the tools can keep everything straight when different modules happen to use the same signal names for internal signals. But it may be difficult for designers to do the same when identical names are used on different modules’ inputs and outputs, especially if they are defined or used differently. Therefore, large projects may adopt a convention to guarantee signal-name uniqueness. Each high-level module is assigned a two- or three-letter designator corresponding to the module name (e.g., “`sam`” for “`ShiftAddMultiplier`”). Then, all signals connected to the module can use its designator as a prefix (e.g., “`sam_reset`”).

CHOOSING CONVENTIONS

The bottom line for all of this is that there are many good signal-naming conventions, and you should follow the particular ones that have been established in your current environment, so that your designs will be maintainable by you and by others in the long term.

4.1.8 Drawing Layout

Logic diagrams and schematics should be drawn with gates in their “normal” orientation with inputs on the left and outputs on the right. The logic symbols for larger-scale logic elements are also normally drawn with inputs on the left and outputs on the right.

A complete schematic page should be drawn with system inputs on the left and outputs on the right, and the general flow of signals should be from left to right. If an input or output appears in the middle of a page, it should be extended to the left or right edge, respectively. In this way, a reader can find all inputs and outputs by looking at the edges of the page only. All signal paths on the page should be connected when possible; paths may be broken if the drawing gets crowded, but breaks should be flagged in both directions, as described later.

Sometimes block diagrams are drawn without crossing lines for a neater appearance, but this is never done in logic diagrams. Instead, lines are allowed to cross, and connections are indicated clearly with a dot. Still, some EDA tools (and some designers) can’t draw legible connection dots. To distinguish between crossing lines and connected lines, they adopt the convention that only “T”-type connections are allowed, as shown in Figure 4-14. This is a good convention to follow in any case.

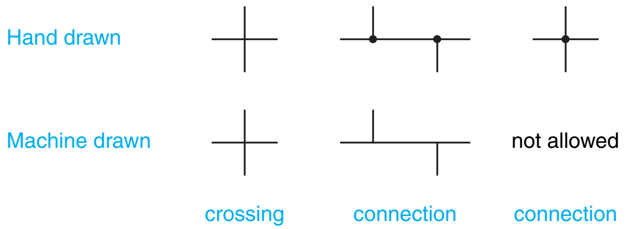


Figure 4-14 Line crossings and connections.

Schematics that fit on a single page are the easiest to work with. The largest practical size for a paper schematic might be E-size (44"×34"). Although its drawing capacity is great, such a large paper size is unwieldy. Probably the best compromise of drawing capacity and practicality is B-size (17"×11"). A displayed version fits well on a typical computer screen with a 16×9 aspect ratio, and a printed schematic can be easily folded for storage and quick reference in standard 3-ring notebooks (for example, for use in the lab during debugging). Regardless of paper size, schematics come out best when the page is used in landscape format, that is, with its long dimension oriented from left to right, the direction of most signal flow.

flat schematic structure

Schematics that don't fit on a single page should be broken up into individual pages in a way that minimizes the connections (and confusion) between pages, and may have a "flat" structure. As shown in Figure 4-15, each page is carved out from the complete schematic and can connect to any other page as if all the pages were on one large sheet. Each page may also use a 2-dimensional

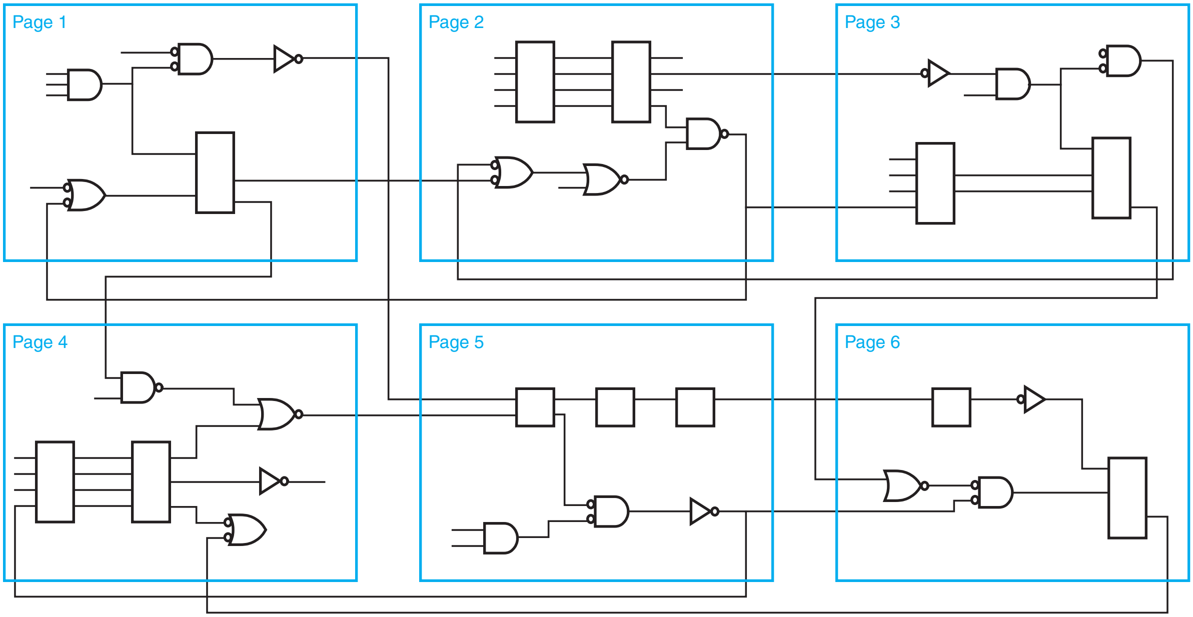


Figure 4-15 Flat schematic structure.

signal flags

coordinate system, like that of a paper road map (used one of those lately?), to flag the sources and destinations of signals that travel from one page to another. An outgoing signal should have flags referring to all of the destinations of that signal, while an incoming signal should have a flag referring to the source only. That is, an incoming signal should be flagged to the place where it is generated, not to a place somewhere in the middle of a chain of destinations that use the signal.

hierarchical schematic structure

Much like programs and block diagrams, schematics can also be constructed hierarchically, as illustrated in Figure 4-16. In this approach, the "top-level" schematic is just a single page that may even take the place of the top-level block diagram. Typically, the top-level schematic contains no gates or other logic elements; it only shows blocks corresponding to the major subsystems, and their interconnections. The blocks or subsystems are in turn defined on lower-level pages, which may contain ordinary gate-level descriptions, or which themselves may use blocks defined in lower-level hierarchies. If a particular lower-level

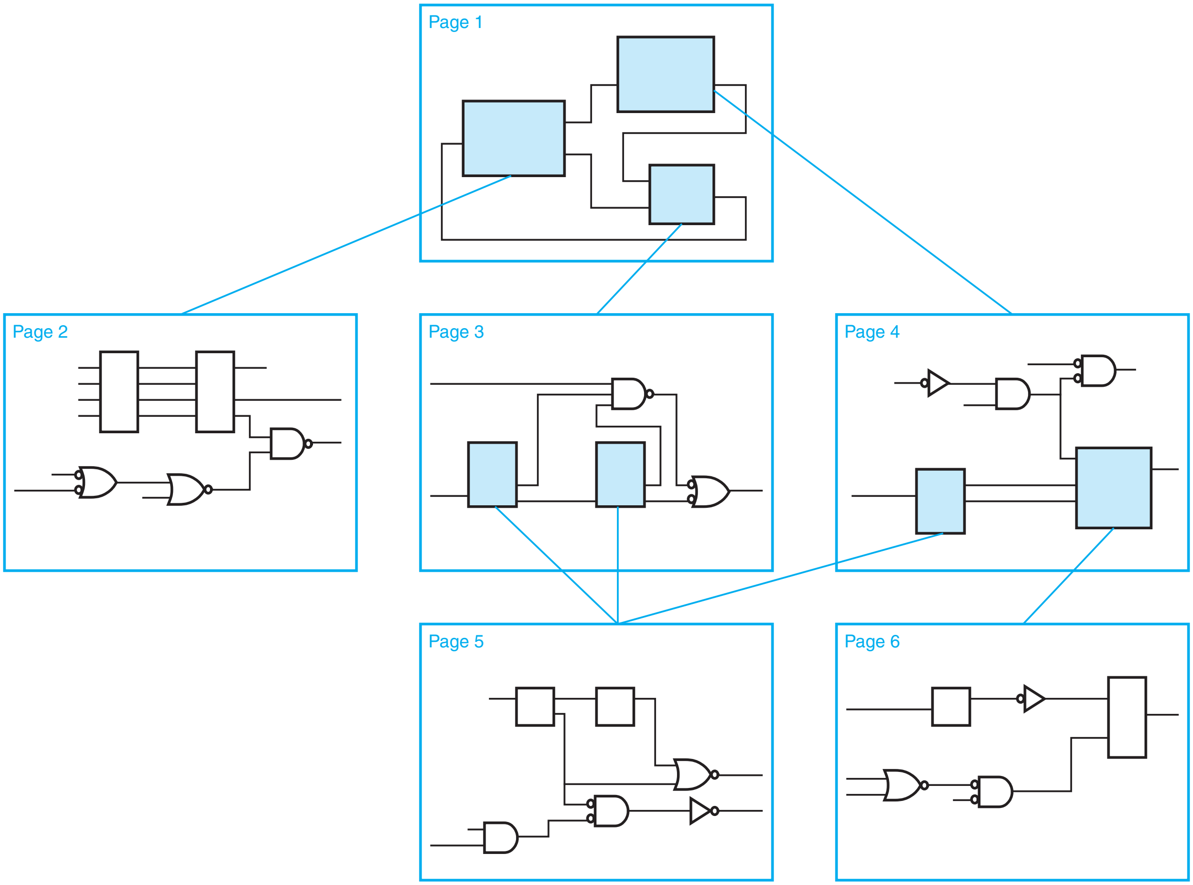


Figure 4-16 Hierarchical schematic structure.

hierarchy needs to be used more than once, it may be reused (instantiated, or "called" in the programming sense) multiple times by the higher-level pages.

The hierarchical approach is inherent in HDLs like Verilog; for example, a module can instantiate another module. In an overall hierarchical design, it's possible for some modules (or "hierarchical schematic pages") to be specified by gate-level logic diagrams, while others are specified by HDL models. In such a "mixed" environment, a given schematic page may contain gates, other off-the-shelf MSI and LSI hardware components, and blocks that represent HDL modules or other schematic pages.

Most EDA environments support both flat and hierarchical schematics. As in HDLs, proper signal naming is very important in both styles, since there are a number of common errors that can occur, such as:

- Like any other program, a schematic-entry program does what you say, not what you mean. If you use slightly different names for what you intend to be the same signal on different pages, they won't be connected.
- Conversely, if you inadvertently use the same name for different signals on different pages of a flat schematic, many programs will dutifully connect them together, even if you haven't connected them with an off-page flag. (In a hierarchical schematic, reusing a name at different places in the hierarchy is generally OK, because the program qualifies each name with its position in the hierarchy, that is, based on its scope.)
- In a hierarchical schematic, you have to be careful in naming the external interface signals on pages in the lower levels of the hierarchy. These are the names that will appear inside the blocks corresponding to these pages when they are used at higher levels of the hierarchy. It's very easy to transpose signal names or use a name with the wrong active level, yielding incorrect results when the block is used.
- This is not usually a naming problem, but many schematic programs have quirks in which signals that appear to be connected are not, or vice versa. Using the "T" convention in Figure 4-14 can help minimize this problem.

Fortunately, most schematic programs have error-checking facilities that can catch many of these errors, for example, by searching for signal names that have no inputs, no outputs, or multiple outputs associated with them. But most logic designers learn the importance of careful, manual schematic double-checking only through the bitter experience of building a printed-circuit board or an ASIC based on a schematic containing some silly error.

4.1.9 Buses

As defined previously, a bus is a collection of two or more related signal lines. For example, a microprocessor system might have an address bus with 16 lines, ADDR0–ADDR15, and a data bus with 8 lines, DATA0–DATA7. The signal names in a bus are not necessarily related or ordered as in these first examples.

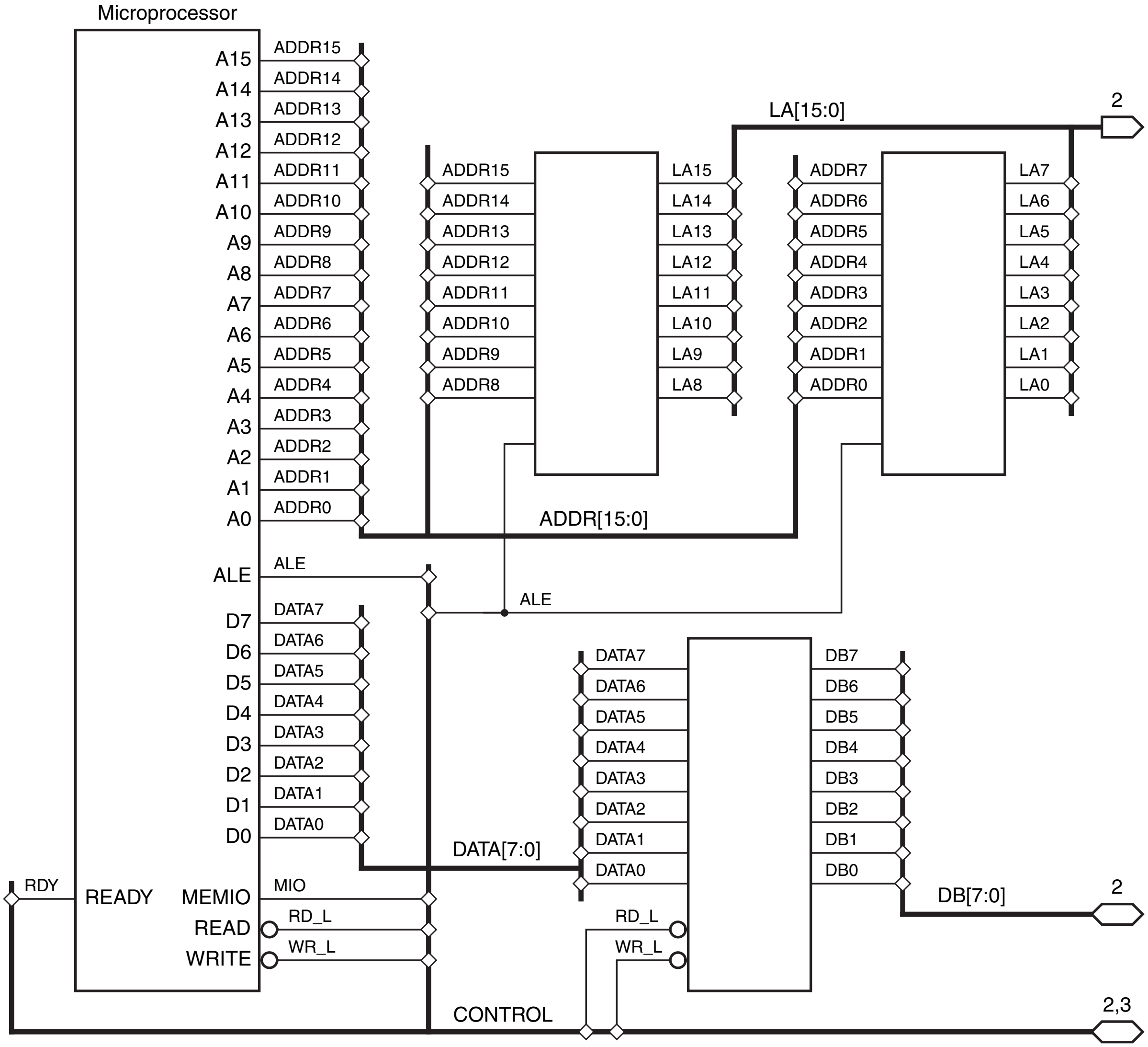


Figure 4-17 Examples of buses.

For example, a microprocessor system might have a control bus containing five signals, ALE, MIO, RD_L, WR_L, and RDY.

range in bus name

Logic diagrams use special notation for buses to reduce the amount of drawing and to improve readability. As shown in Figure 4-17, a bus has its own descriptive name, such as ADDR[15:0], DATA[7:0], or CONTROL. A bus name might use brackets and a hyphen or colon to denote a range. Buses may be drawn with different or thicker lines than ordinary signals. Individual signals are put into or pulled out of the bus by connecting an ordinary signal line to the bus and writing the signal name. Often a special connection dot is also used, as in the example. Different environments may use different conventions.

An EDA system keeps track of the individual signals in a bus. When it actually comes time to build a circuit from the schematic, signal lines in a bus are treated just as if they had all been drawn individually.

The symbols at the righthand edge of Figure 4-17 are interpage signal flags. They indicate that LA goes out to page 2, DB is bidirectional and connects to page 2, and CONTROL is bidirectional and connects to pages 2 and 3.

4.1.10 Additional Schematic Information

Complete schematic diagrams indicate IC types, reference designators, and pin numbers, as shown in Figure 4-18 for a very simple circuit that uses old-style SSI ICs. The *IC type* is a part number identifying the component that performs a given logic function. For example, a 2-input NAND gate might be identified as a 74HCT00 or a 74AC00.

IC type

The *reference designator* for an IC identifies a particular instance of that IC type installed in the system. In conjunction with the system's mechanical documentation, the reference designator allows a particular IC to be located during assembly, test, and maintenance of the system. Traditionally, reference designators for ICs begin with the letter U (for "unit"). Some ICs have multiple instances of a function in the same package, so a schematic might have multiple logic symbols with the same reference designator (like the NAND gates in ICs U1 and U2 in Figure 4-18).

reference designator

Once a particular IC is located, *pin numbers* are used to locate individual logic signals on its pins. The pin numbers are written near the corresponding inputs and outputs of the standard logic symbol, as shown in Figure 4-18.

pin number

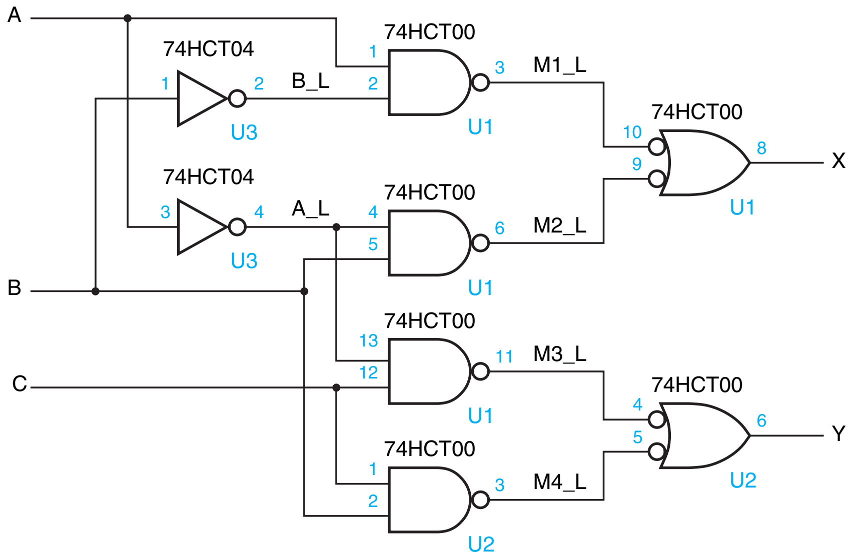


Figure 4-18
Schematic diagram for a circuit using several SSI parts.

In the rest of this book, we’ll omit reference designators and pin numbers in most examples. Our examples are typically targeted to be implemented inside a programmable device or an ASIC, so they wouldn’t have pin numbers anyway. Even in “real-world” board-level design, the components and pins have gotten to be so small that it is very difficult to probe and debug them except with very specialized tools. And when you do get out into the “real world” and prepare a schematic diagram for a board-level design using a schematic drawing program, the program automatically provides the pin numbers for the devices that you select from its component library.

Note also that elements that serve the same purposes as the schematic information just described also exist in HDL-based designs, as we’ll see later in many Verilog examples:

- The equivalent of an IC type is a Verilog component or module name.
- A unique reference designator for each component or module in a Verilog design is provided by the designer or generated by the tools.
- Logic signals are normally connected to inputs and outputs of components by pairing them with the components’ alphanumeric port names. (Or, they may be hooked up according to position in a list of connected signals, but using this error-prone alternative is discouraged.)

4.2 Circuit Timing

“Timing is everything”—in comedy, in investing, and yes, in digital design. As you’ll learn in Section 14.4, the outputs of real circuits take time to react to their inputs, and many of today’s circuits and systems are so fast that even the speed-of-light delay in propagating an output signal to an input on the other side of a board or chip is significant.

Most digital systems are sequential circuits that operate step-by-step under the control of a periodic clock signal, and the speed of the clock is limited by the worst-case time that it takes for the operations in one step to complete. Thus, digital designers need to be keenly aware of timing behavior in order to build fast circuits that operate correctly under all conditions.

The past decades have seen great advances in the number and quality of EDA tools for analyzing circuit timing. Still, quite often the greatest challenge in completing a board-level or especially an FPGA-based or ASIC design is to get the required timing performance. In this section, we start with the basics so you can understand what the tools are doing when you use them, and figure out how to improve your circuits when their timing isn’t quite making it.

4.2.1 Timing Diagrams

timing diagram

A *timing diagram* illustrates the logical behavior of signals in a digital circuit as a function of time. Timing diagrams are an important part of the documentation of any digital system. They can be used both to explain the timing relationships

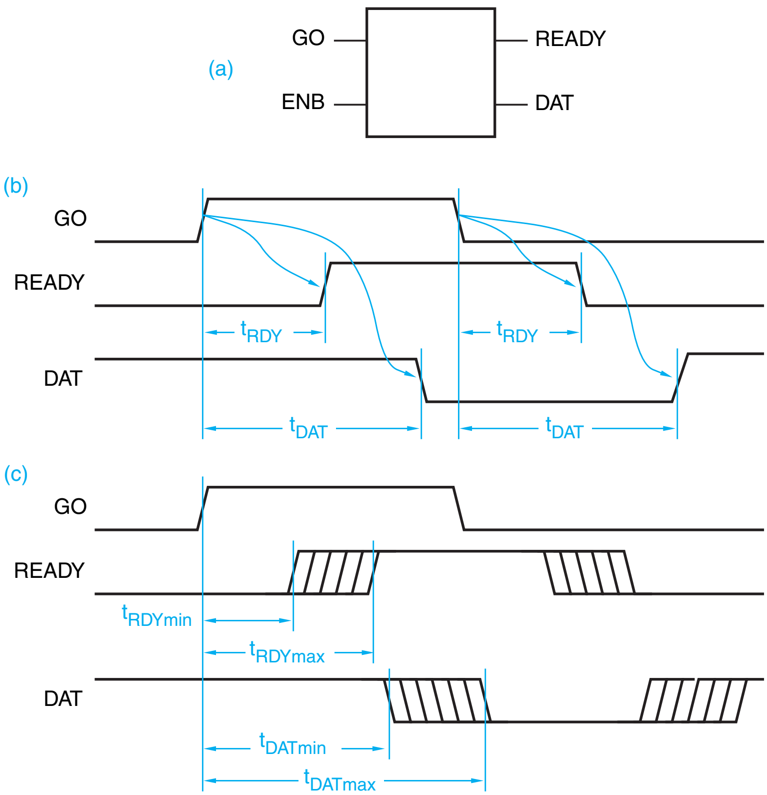


Figure 4-19
Timing diagrams for a combinational circuit: (a) block diagram of circuit; (b) causality, propagation delay; (c) minimum and maximum delays.

among signals within a system and to define the timing of external signals that are applied to and produced by a module (also known as *timing specifications*).

timing specifications

Figure 4-19(a) is the block diagram of a simple combinational circuit with two inputs and two outputs. Assuming that the ENB input is held at a constant value, (b) shows the delay of the two outputs with respect to the GO input. In each waveform, the upper line represents a logic 1 and the lower line a logic 0. Signal transitions are drawn as slanted lines to remind us that they do not occur in zero time in real circuits.

The time it takes for a signal to change from one state to the other is called the *transition time*; more specifically, the time to go from LOW to HIGH is called the *rise time*, and from HIGH to LOW the *fall time*. For simplicity's sake in many timing diagrams, including most of the ones in this book, time is measured from the centerpoints of transitions. We'll have more to say about signal transitions in Section 14.4.

transition time
rise time
fall time

Arrows are sometimes drawn, especially in complex timing diagrams, to show *causality*—which input transitions cause which output transitions. In any case, the most important information provided by a timing diagram is a specification of the *delay* between transitions.

causality

delay

Different paths through a circuit may have different delays. For example, Figure 4-19(b) shows that the delay from GO to READY is shorter than the delay from GO to DAT. Similarly, the delays from the ENB input to the outputs may vary, and could be shown in another timing diagram. Furthermore, as we'll show in Section 4.2.3, the delay through any given path may vary depending on whether the output is changing from LOW to HIGH or from HIGH to LOW (this phenomenon is not shown in the figure).

A single timing diagram may contain many different delay specifications. Each different delay is marked with a different identifier, like t_{RDY} and t_{DAT} in the figure. In large timing diagrams, the delay identifiers are often numbered for easier reference (e.g., t_1, t_2, …, t_{42}). In either case, the timing diagram would normally accompanied by a *timing table* that specifies each delay amount and the conditions under which it applies.

timing table

Since the delays of real digital components can vary depending on many factors, delay is seldom specified as a single number. Instead, a timing table may specify a range of values as discussed in the next subsection. The idea of a range of delays is sometimes carried over into the timing diagram itself by showing the transitions to occur at uncertain times, as in Figure 4-19(c).

For some signals, the timing diagram needn't show whether the signal changes from 1 to 0 or from 0 to 1 at a particular time, only that a transition occurs then. Any signal that carries a bit of "data" has this characteristic—the actual value of the data bit varies according to circumstances but, regardless of value, the bit is transferred, stored, or processed at a particular time relative to "control" signals in the system. Figure 4-20(a) is a timing diagram that illustrates this concept. The "data" signal is normally at a steady 0 or 1 value, and transitions occur only at the times indicated. The idea of an uncertain delay time can also be used with "data" signals, as shown for the DATAOUT signal.

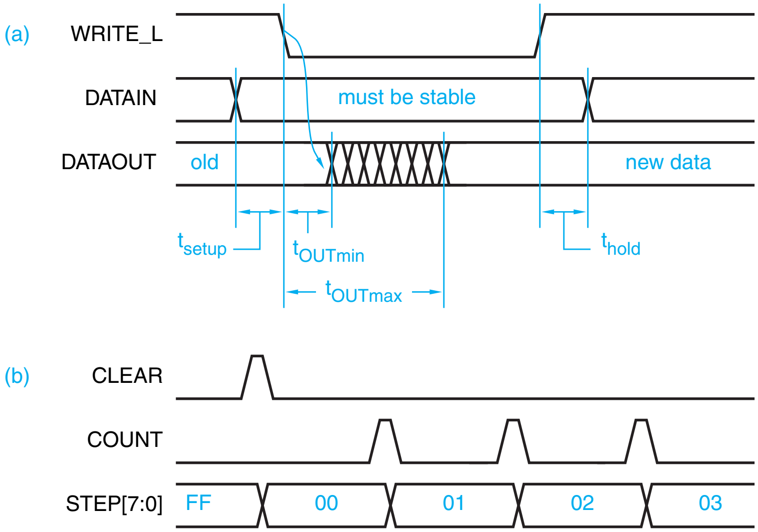


Figure 4-20
Timing diagrams for "data" signals: (a) certain and uncertain transitions; (b) sequence of values on an 8-bit bus.

Quite often in digital systems, a group of data signals in a bus is processed by identical circuits. In this case, all signals in the bus have the same timing, and they can be represented by a single line in the timing diagram and corresponding specifications in the timing table. If the bus bits are known to take on a particular combination at a particular time, this is sometimes shown in the timing diagram using binary, octal, or hexadecimal numbers, as in Figure 4-20(b).

4.2.2 Propagation Delay

propagation delay

The *propagation delay* of a signal path is the time that it takes for a change at the input of the path to produce a change at the output of the path; this may be designated by a symbol like t_{pX}, where the label "X" qualifies the path. Propagation delay depends on the internal, analog design of a circuit, as well as on many operational characteristics of the circuit, including the following:

- *Power-supply voltage*. Many CMOS circuits are designed to work over a range of supply voltages, and they usually run slower at lower voltages. Even at a particular, "nominal" supply voltage, the actual voltage at any time will vary due to component tolerances, noise, and other factors, and delay will increase or decrease with these variations.
- *Temperature*. The speed of a circuit varies with its operating temperature, which varies with the environment and with the heat generated by both the circuit itself and the larger system that contains it.
- *Output loading*. A circuit's output must continuously supply current to the other component inputs, from a small to a large amount, depending on the electrical characteristics of those inputs. It must also provide extra current during transitions to charge and discharge the electrical capacitance that is associated with inputs and wiring. This affects signal rise and fall times, which affect total delay even if we are just measuring from the centerpoints of the transitions.
- *Input rise and fall times*. Similarly, if input transition times are slow, it will take longer for resulting output transitions to "get started."
- *Transition direction*. The propagation delay when an output changes from LOW to HIGH (t_{pLH}) may be different from the delay when it changes from HIGH to LOW (t_{pHL}). This effect may occur because of circuit internals, direction-dependent driving capabilities of the circuit's output, or both.
- *Speed-of-light delays*. The propagation delay of electrical signals is about 5 ns/meter (or 50 ps/cm) in typical wiring, a significant number between ICs in board-level circuits that are physically large, and on-chip in circuits that are very fast.
- *Noise and crosstalk*. Signal voltages can be affected by electrical noise in general and by transitions occurring on nearby, adjacent signal lines. As a result, the input switching thresholds may be reached a little sooner or a little later, making the path delay correspondingly shorter or longer.

- *Manufacturing tolerances*. Although IC manufacturing processes are controlled to a high degree of precision, there is still some variation, and circuit speed will vary among different batches of a component, and even among different instances of a component taken from the same wafer.

With all these different sources of timing variation, it's not practical to calculate a circuit's exact timing in a particular application and environment, but luckily, we don't have to. Instead, we can make good engineering estimates based on "maximum," "typical," and "minimum" propagation delays specified by IC manufacturers, and throw in a little "engineering margin" just for good measure.

On top of all of the timing variations listed above for a given signal path, there is another whole dimension: a combinational circuit with many inputs and outputs may have many different internal signal paths, and each one may have a different propagation delay. Thus, an IC's manufacturer normally specifies a delay for each of its internal signal paths. We'll see this when we look at a few example MSI parts in the next subsection.

A logic designer who combines ICs in a larger circuit can use individual device specifications to analyze the overall circuit timing. In simple designs, this can be done by hand; in more complex designs, a timing analysis program can be used. In either case, the delay of a path through the overall circuit is computed as the sum of the delays through the individual devices on the path, plus speed-of-light delays if they are not negligible.

In ASIC-, FPGA-, and CPLD-based design, delay analysis can be a lot more involved, and the EDA tools normally handle most of it. While delays can be highly dependent on signal routing within these chips, the final routing is not known until the design has been completed, targeted to a chip, and actually laid out. So, timing is typically analyzed at two different stages in the design.

First, the timing can be estimated once the logic design is complete, using known timing for the individual logic elements and an estimate of the routing delays based on factors that are already known at that stage, such as approximate chip size and the number of inputs driven by each output. At this point, the designer can decide if the top-level design approach is likely to be capable of meeting timing goals, or whether more work must be done to develop a speedier approach, or to cut back on project goals, which takes a different kind of work!

Later, once the logic elements have been placed on the chip and their connections have been routed, it is possible to calculate the expected delays more precisely, based on details of the placed elements, wire lengths, and other factors. At that point, it is also possible to identify the worst-case paths within

GLITCHES

As we showed in Section 3.4, a combinational-circuit output can sometimes exhibit a short pulse at a time when steady-state analysis predicts that the output should not change, depending on actual propagation delays in an instance of the circuit.

the design, and it's possible to change the chip design and layout to do a little better. For example, one could instruct the tools to place critical-path elements closer to each other, to use stronger drivers on critical-path outputs, to use faster "wires" on the critical path, to make multiple copies of high-fanout signals so each copy has a lighter load, and so on. All of this may be possible without changing the basic logic design.

4.2.3 Timing Specifications

A manufacturer's timing specification for a device may give minimum, typical, and maximum values for each propagation-delay path and transition direction:

maximum delay

- *Maximum.* This specification is the one most often used by experienced designers, since a path "never" has a propagation delay longer than the maximum. However, the definition of "never" varies among logic families and manufacturers. For example, "maximum" propagation delays of Texas Instruments' old 74LS and 74S TTL bipolar devices are specified with supply voltage (V_{CC}) 5.0 V, ambient temperature (T_A) 25°C, and very little capacitive load (15 pF). If the voltage or temperature is different, or if the capacitive load is larger, the delay may be longer. On the other hand, a "maximum" propagation delay for 74AC devices at 5.0 V "nominal" is specified more conservatively with a supply-voltage range of 4.5–5.5 V, a temperature range of –25°C to 85°C, and a capacitive load of 50 pF.

typical delay

- *Typical.* This specification is the one most often used by designers who don't expect to be around when their product leaves the friendly environment of the engineering lab and is shipped to customers. The "typical" delay is what you see from a device that was manufactured on a good day and is operating under near-ideal conditions. Perhaps because of the danger of relying on "typical" specifications, manufacturers have stopped using them for many newer, advanced CMOS logic families.

minimum delay

- *Minimum.* This is the smallest propagation delay that a path will ever exhibit. Most well-designed circuits don't depend on this number; that is, they will work properly even if the delay is zero. That's good, because manufacturers don't specify minimum delay in some moderate-speed logic families. However, in very high-speed families, a nonzero minimum delay is specified to help the designer ensure that hold-time requirements of latches and flip-flops, to be discussed in Section 10.2, are met.

HOW TYPICAL IS TYPICAL?

Most ICs, perhaps 99%, really are manufactured on "good" days and exhibit delays near the "typical" specifications. However, if you design a system that works only if all of its 100 ICs meet the "typical" timing specs, probability theory suggests that 63% ($1 - .99^{100}$) of the systems won't work and you'll quickly detect the problem in the lab. But see the next box ….

A COROLLARY OF MURPHY'S LAW

Murphy's law states, "If something can go wrong, it will." A corollary to this is, "If you want something to go wrong, it won't."

In the previous boxed comment, you might think that you have a 63% chance of detecting the potential timing problems in the engineering lab. The problems aren't spread out evenly, though, since all ICs from a given batch tend to behave about the same. Murphy's corollary says that *all* of the engineering prototypes will be built with ICs from the same, "good" batches. Therefore, everything works fine for a while, just long enough for the system to get into volume production and for everyone to become complacent and self-congratulatory.

Then, unbeknownst to the manufacturing department, a "slow" batch of some IC type arrives from a supplier and gets used in every system that is built, so that *nothing* works. The manufacturing engineer scurries around trying to analyze the problem (not easy, because the designer is long gone and didn't bother to write a circuit description), and in the meantime, the company loses big bucks because it is unable to ship its product.

*4.2.4 Sample Timing Specifications

This subsection gives you some real delay numbers to think about and to use to work out sample delay calculations by hand, as in the Exercises. The purpose is to give you a "feel" for the complexity of delay calculations. But for simplicity, delays are given only for a selection of individual gates and MSI parts in older discrete CMOS logic families as published by device manufacturers.

Table 4-2 lists minimum, typical, and maximum delays of several 74-series CMOS gates. The 74AC CMOS family operates with a supply voltage of 1.5 V to 5.5 V, and delay specifications in the table are for nominal 5.0-V operation (V_{CC} = 4.5 V to 5.5 V) over a temperature range of –25°C to 85°C, with a capacitive load of 50 pF. Note that no typical delays are specified for 74AC parts, only minimums and maximums.

All inputs of a CMOS SSI gate have the same specification for propagation delay to the output (this is not necessarily true for gates in an ASIC). Also,

ASIC AND FPGA TIMING SPECS

The timing specifications for gates and larger blocks within an ASIC or FPGA are typically not published in tables like the ones in this subsection. Instead, they are embedded in an EDA tool for analyzing the timing in a device. Rather than using simple minimum and maximum delays, a typical EDA tool uses a more detailed electrical model to get a more precise delay range for each gate instance based on loading and other factors previously mentioned. It also accounts for capacitive and perhaps speed-of-light delays of wiring, plus the significant circuit delays through programmable interconnect in FPGAs. The overall delay calculation in these devices is so involved that you would never attempt do it by hand.

Table 4-2 Propagation delay in nanoseconds of selected CMOS SSI parts.

		74AC @ 5.0V				***74HC @ 2.0V***			***74HC @ 4.5V***		
		Minimum		***Maximum***		***Typ.***	***Maximum***		***Typ.***	***Maximum***	
Part Number	***Function***	t_{pLH}	t_{pHL}	t_{pLH}	t_{pHL}	25°C t_{pd}	25°C t_{pd}	85°C t_{pd}	25°C t_{pd}	25°C t_{pd}	85°C t_{pd}
’00	2-input NAND	1.9	1.9	6.6	6.6	45	90	115	9	18	23
’02	2-input NOR	3.0	3.0	10.4	10.4	45	90	115	9	18	23
’04	Inverter	1.7	1.7	5.9	5.9	45	95	120	9	19	24
’08	2-input AND	1.0	1.0	8.5	7.5	50	100	125	10	20	25
’10	3-input NAND	1.0	1.0	8.0	6.5	35	95	120	10	19	24
’11	3-input AND	1.0	1.0	8.5	7.5	35	100	125	10	20	25
’20	4-input NAND	1.5	1.5	8.0	7.0	45	110	140	14	22	28
’21	4-input AND	1.5	1.5	6.5	7.0	44	110	140	14	22	28
’27	3-input NOR	1.5	1.5	8.5	8.5	35	90	115	10	18	23
’30	8-input NAND	1.0	1.0	9.5	9.5	41	130	165	15	26	33
’32	2-input OR	1.5	1.0	10.0	9.0	50	100	125	10	20	25
’86	2-input XOR	1.0	1.0	9.0	9.5	40	100	125	12	20	25

CMOS outputs have a very symmetrical output driving capability, so the delays for LOW-to-HIGH and HIGH-to-LOW output transitions are usually the same. There are just a few cases in the 74AC columns of Table 4-2 where a difference may be seen between t_{pLH} and t_{pHL}.

The delays in the 74HC CMOS SSI family are specified a little differently. These devices can be operated with V_{CC} anywhere between 2.0 V and 6.0 V, and manufacturers specify delays at three possible voltages: 2.0, 4.5, and 6.0 V. In the table, we have given the delays for $V_{\mathrm{CC}} = 2.0$ V and 4.5 V. Note that unlike 74AC, 74HC specifies its maximum delays at the stated supply voltage, not over a range. Also, no minimum delays are given. Finally, the delays for rising and falling transitions on each device are equal, or near enough to equal, that the manufacturers specify only a single propagation delay t_{pd} that applies to both transition directions.

The first 74HC column of Table 4-2 gives typical delays, which are for a “typical” device operating with an ambient temperature of 25°C, a capacitive load of 50 pF, and the stated supply voltage. The worst-case maximum delay under those operating conditions, given in the second column, can be twice as long or more, depending on the device. As specified in the third column, the

ESTIMATING MINIMUM DELAYS

If the minimum delay of an IC is not specified, a conservative designer assumes that it has a minimum delay of zero.

Some circuits won't work if the propagation delay actually goes to zero, but the cost of modifying a circuit to handle the zero-delay case may be unreasonable, especially since this case is expected never to occur. To show that a design always works under "reasonable" conditions, logic designers may estimate that ICs have minimum delays of one-fifth to one-fourth of their published *typical* delays.

It is also possible to determine minimum delays by performing analog analysis of the circuit as it is actually used in a physical design, considering factors like load capacitance and wiring delay. Even if the IC delay is near zero, these external factors will contribute something to create a minimum delay.

delay can be even higher at other points in the full allowed temperature range of –45°C to 85°C, though normally this would occur at the high end of 85°C, so that's how the column is labeled. With a 4.5-V supply voltage, as shown in the next three columns, almost all of the delays are shorter by a factor of five.

Table 4-3 gives the delay specs for CMOS MSI versions of some of the combinational logic building blocks that will be introduced in Chapter 6. Here, the delay from an input transition to the resulting output transition depends on which input is causing a change in which output, as noted in the "From" and "To" columns. The delay may also depend on the internal path taken by the changing signal, but not for any of the devices listed in the table.

worst-case delay

To permit a simplified "worst-case" analysis, board-level designers often use a single *worst-case delay* specification that is the maximum of t_{pLH} and t_{pHL} specifications under worst-case conditions of voltage and temperature. The worst-case delay through a path is then computed as the sum of the worst-case delays through the individual components, independent of the transition direction and other circuit conditions. This may give an pessimistic view of the overall circuit delay, but it saves analysis time and it always works.

4.2.5 Timing Analysis Tools

To accurately analyze the timing of a circuit containing more than a few gates and other components, a designer may have to study its logical behavior in excruciating detail. A moderate-size circuit can have many different paths from a set of input signals to a set of output signals. To determine the minimum and maximum delays through the circuit, you must look at every possible path.

A combinational circuit potentially has a path between every input and every output. so the number of paths to examine may be at least the product of the number of inputs and the number of outputs. On some paths, a signal may fan out to multiple internal paths, only to have those paths come together at a single output (for example, as in Figure 3-5 on page 105), further increasing the total

Table 4-3 Propagation delay in nanoseconds of selected CMOS MSI parts.

				74AC @ 5.0V		***74HC @ 2.0V***			***74HC @ 4.5V***		
				Min.	***Max.***	***Typ.***	***Maximum***		***Typ.***	***Maximum***	
						25°C	25°C	85°C	25°C	25°C	85°C
Part	***Function***	***From***	***To***	t_{pd}	t_{pd}	t_{pd}	t_{pd}	t_{pd}	t_{pd}	t_{pd}	t_{pd}
’138	3-to-8 binary decoder	any select	output	2.8	10.0	67	180	225	18	36	45
		$\overline{\mathrm{G2A}}$, $\overline{\mathrm{G2B}}$	output	2.6	9.1	66	155	195	18	31	39
		G1	output	2.8	10.0	66	155	195	18	31	39
’139	dual 2-to-4 binary decoder	any select	output	2.8	9.5	47	175	220	14	35	44
		enable	output	2.8	9.5	39	175	220	11	35	44
’148	8-to-3 priority encoder	$\overline{\mathrm{I1}}$-$\overline{\mathrm{I7}}$	$\overline{\mathrm{A0}}$-$\overline{\mathrm{A2}}$			69	180	225	23	36	45
		$\overline{\mathrm{I0}}$-$\overline{\mathrm{I7}}$	$\overline{\mathrm{EO}}$			60	150	190	20	30	38
		$\overline{\mathrm{I0}}$-$\overline{\mathrm{I7}}$	$\overline{\mathrm{GS}}$			75	190	240	25	38	48
		$\overline{\mathrm{EI}}$	$\overline{\mathrm{A0}}$-$\overline{\mathrm{A2}}$			78	195	245	26	39	49
		$\overline{\mathrm{EI}}$	$\overline{\mathrm{GS}}$			57	145	180	19	29	36
		$\overline{\mathrm{EI}}$	$\overline{\mathrm{EO}}$			66	165	205	22	33	41
’151	8-to-1 multiplexer	any select	Y	4.7	16.5	94	250	312	30	50	63
		any select	$\overline{\mathrm{Y}}$	5.1	17.8	94	250	312	30	50	63
		any data	Y	3.5	12.3	74	195	244	23	39	49
		any data	$\overline{\mathrm{Y}}$	3.8	13.5	74	195	244	23	39	49
		enable	Y	3.1	11.1	49	127	159	15	25	32
		enable	$\overline{\mathrm{Y}}$	3.5	12.3	49	127	159	15	25	32
’157	2-to-4 multiplexer	select	output	3.8	13.2		145	180	12	29	36
		any data	output	2.2	7.7		125	155	10	25	31
		enable	output	3.6	12.3		135	170	11	27	34
’280	9-input parity circuit	any input	EVEN	5.2	18.2		200	250	17	40	50
		any input	ODD	5.4	19.1		200	250	17	40	50
’283	4-bit adder	C0	any Si	4.5	16.0		230	290	19	46	58
		any Ai, Bi	any Si	4.7	16.5		210	265	18	42	53
		any input	C4	4.5	16.0		195	245	16	39	49
’682	8-bit comp.	any input	output			130	275	344	26	55	69

5.0 V VS. 4.5 V The old bipolar logic family, TTL, uses a nominal 5-V ±10 % power supply, and several CMOS logic families, including 74AC and 74AC, were designed to provide some compatibility with TTL when operated with a 5-V supply.

So, it may seem odd that 74AC timing is specified with a 5.0-V ±0.5 V supply, while 74HC is specified only with a 4.5-V supply. However, since the maximum delay for CMOS occurs at the low end of a stated voltage range, 74HC's 4.5-V spec actually does yield an appropriate worst-case delay number when interfacing with nominal 5-V devices that specify a 4.5–5.5-V supply range, including 74AC as well as TTL.

Still, you need to be careful when using "typical" numbers. For 74HC parts, some manufacturers state their typical delays at 4.5 V and a 50-pF load, while others use 5.0 V and 15 pF, which makes them look faster.

number of paths to be examined. On the other hand, in larger circuits, multiple inputs or outputs may have similar functions and timing paths, allowing them to be "grouped" by function and analyzed together; for example, in Figure 3-15 on page 112, all of the "number" inputs have similar timing paths. In any case, analyzing all of the different delay paths in a large circuit is typically practical only with the assistance of automated tools.

EDA environments for board-level logic design have component libraries that typically contain not only logic symbols and functional models for various logic elements, but also their timing models. Likewise, EDA tools for ASICs and FPGAs have timing models for their internal elements.

Timing is important enough that it is a fundamental capability of the HDLs Verilog and VHDL. As we will show in Chapter 5 for Verilog, these HDLs have facilities for specifying expected delay at the component or module level. A simulator allows you to apply input sequences to an HDL model and observe how and when outputs are produced in response. You may be able to control whether minimum, typical, maximum, or some combination of delay values are used.

Even with a simulator, you're not off the hook, though. It's up to the designer to supply the input sequences for which the simulator should produce outputs, for example, using a test bench. Thus, you still need to have a good feel for what to look for and how to stimulate your circuit to produce and observe the worst-case delays.

timing-analysis program

timing analyzer

Instead of using a simulator and supplying your own input sequences, you can use a *timing analysis program* (or *timing analyzer*). Based on the topology of a synthesized circuit, such a program can automatically find all possible delay paths and print out a sorted list of them, starting with the slowest. These results may be overly pessimistic, however, as some paths may not actually be used in normal operation of the circuit; the designer must still use some intelligence and experience to interpret the results properly.

Also, timing may have to be examined during two or more stages of project development, especially if the design will be realized in an ASIC, FPGA, or CPLD. This is true whether the design is done using schematics at the gate and block level or using an HDL.

In the early stages of a design, it's easy enough for the timing analyzer to estimate worst-case path delays in a preliminary realization by finding all the signal paths and adding up the known delays of individual logic elements. However, the final circuit realization is not determined until later in the design, when the complete design is fitted into an FPGA or CPLD, or physically laid out in a ASIC. At that time, other elements of delay will be known, due to capacitive loads, larger buffers inserted to handle heavier-than-expected loads, speed-of-light delays on long wires, and other differences between the estimates made in the early stages and the actual realized circuit.

On the first try, the timing results for the "final" realized circuit may not meet the design's requirements—the circuit may be too slow, or parts of it may be too fast, such that flip-flops' hold-time requirements are not met (in which case, the circuit will not work even at slow speeds; see Section 10.2). When that happens, the designer must change parts of the circuit, adding or changing buffers and other components, reworking the internal design of individual modules to get better timing performance, changing signaling between modules, or even talking to the boss about relaxing the performance goals for the project (this is a last resort!). Then, the circuit must be resynthesized and the timing results must be checked again, and the process must be repeated until the performance goals are met. This is called *timing closure* and can take several months in large ASIC and FPGA projects.

timing closure

4.3 HDL-Based Digital Design

4.3.1 HDL History

Forty years ago, the primary tools of a digital designer included a logic-drawing template, like in Figure 1-7 on page 13, a ruler, and a pencil, all for drawing schematic diagrams. In the 1980s, schematics were still the primary means of describing digital circuits and systems, but at least schematic creation and maintenance had been simplified by the introduction of schematic editor tools. That decade also saw limited use of hardware description languages (HDLs), mainly to describe logic equations to be realized in the first generation of programmable logic devices.

In the 1990s, HDL usage by digital system designers accelerated as PLDs, CPLDs, and FPGAs became inexpensive and commonplace. At the same time, as ASIC densities continued to increase, it became increasingly more difficult to describe large circuits using schematics alone, and many ASIC designers turned to HDLs as a means to design individual modules within a system-on-a-chip. Today, HDLs are by far the most common way to describe both the top-level and the detailed module-level design of an ASIC, FPGA, or CPLD. Schematics are often used only to specify the board-level interconnections among these devices and other LSI components like memories and microprocessors, and SSI/MSI "glue" logic if any.

The first HDL to enjoy widespread commercial use was PALASM (PAL Assembler) from Monolithic Memories, Inc., inventors of the first-generation PLD, the so-called PAL device. Introduced in the early 1980s, PALASM was used to specify logic equations for realization in PAL devices. Comparing with computer programming languages, the first version of PALASM was like assembly language—it provided a text-based means to specify the information to be programmed (in PALASM's case, logic equations), but little else. Subsequent developments in PALASM and in competing languages, such as CUPL (Compiler Universal for Programmable Logic) and ABEL (Advanced Boolean Equation Language), yielded more capabilities. These included logic minimization, "high-level" statement constructs like "`if-then-else`" and "`case`", and the ability to derive logic equations from these high-level constructs. Previous editions of this book included the ABEL language and design examples.

The next important innovations in HDLs occurred in the mid-1980s, and were the developments of Verilog and VHDL. Both languages support modular, hierarchical coding, like C and other high-level computer programming languages, and both have a rich variety of high-level constructs, including arrays, procedure and function calls, and conditional and iterative statements.

Both Verilog and VHDL started out as *simulation* languages, allowing a digital system's hardware to be modeled and its operations to be simulated on a computer. Thus, many of these languages' features have roots in their original simulation-only application. However, later developments in the language tools allowed actual hardware designs, based on real components, to be synthesized from the language-based descriptions. You might even think of the "D" in "HDL" changing its meaning from "Description" to "Design." ABEL, on the other hand, started as a synthesizable design language specifically targeted to PAL devices, and simulation capability was added later.

4.3.2 Why HDLs?

As we stated in Chapter 1, digital design activity is moving to ever higher levels of abstraction. This move has been both enabled and necessitated by decreasing cost per function and the ever higher level of functionality and integration that can be achieved on a single chip.

In traditional software design, high-level programming languages like C, C++, and Java have raised the level of abstraction so that programmers can design larger, more complex systems, albeit with some sacrifice in performance compared to handcrafted assembly-language programs for the lowest-level functions. But you would have *no* performance in today's complex software systems if they had to be written completely in assembly language—they never could have been finished! Beyond the languages themselves, accompanying software libraries allow commonly used functions, like creating and managing interactive display windows, to be performed easily without requiring the programmer to build such functions from scratch.

The situation for hardware is now similar for the most complex and highest performance devices and systems. Verilog and VHDL allow designers to describe hardware at a high level, and then interconnect multiple modules in a hierarchy to perform a higher-level function. Moreover, commonly used functions and subsystems, from specialized register files and memories, to serial interfaces like USB and Ethernet, to memory and graphics interfaces, can be licensed from an IP (intellectual property) provider and combined with the designer's custom circuits that provide the "secret sauce" for a new application. This frees the designer from the need to recreate the commonly used functions, while still allowing everything to be integrated in a single ASIC or FPGA.

The circuit produced by a Verilog or VHDL synthesis tool may not be as small or fast as one designed and tweaked by hand by an experienced designer—or team of designers—but in the right hands these tools can support much larger system designs. This is, of course, a requirement if we're to continue to take advantage of the tens of millions of gates offered by most advanced FPGA and ASIC technologies, or even the tens of thousands offered by the inexpensive components of today.

4.3.3 EDA Tool Suites for HDLs

Typically, a single integrated tool suite handles several different aspects of an HDL's use. We might informally call this "the HDL compiler," but an EDA tool suite for design with HDLs really includes many different tools with their own names and purposes:

- A *text editor* allows you to write, edit, and save an HDL source file. Since the editor is coupled to the rest of the HDL development system, it often contains HDL-specific features such as recognizing specific filename extensions associated with the HDL, and recognizing HDL reserved words and comments and displaying them in different colors (as you'll see for reserved words in the Verilog models in this book).

text editor

- The *compiler* is responsible for parsing the HDL source file, finding syntax errors, and figuring out what the model really "says." A typical HDL compiler creates a file in an intermediate, technology-neutral digital-design description language typically called an RTL (register-transfer language). The RTL file is an unambiguous description of the interconnections and logic operations, both combinational and sequential, implied by the HDL model. However, this is still not a hardware version of the model.

compiler

register-transfer language (RTL)

REGISTER-TRANSFER LANGUAGE (RTL)

HDL usage took off with when synthesis tools became available in the late 1980s, but non-synthesizable hardware description languages were around a while before then. Most prominent are *register-transfer languages*, which have been used for decades to describe the operation of synchronous systems. Such a language combines the control-flow notation of a state-machine description language with a means for defining and operating on multibit registers. Register-transfer languages have been especially useful in computer design, where individual machine-language instructions are defined as a sequence of more primitive steps involving loading, storing, combining, and examining the contents of registers.

synthesizer
synthesis tool
libraries

- A *synthesizer* (or *synthesis tool*) targets the RTL design to a specific hardware technology, such as an ASIC, FPGA, or CPLD. In doing so, it refers to one or more *libraries* containing details of the targeted technology, such as the features and limitations of FPGA macrocells, or the kinds of gates and flip-flops available as basic building blocks in an ASIC. Libraries may also contain larger-scale components such as multibit adders, registers, and counters. By analyzing the RTL description, a sophisticated synthesizer can "infer" opportunities to convert portions of the design efficiently into available larger-scale library components. Synthesis typically has multiple phases, and these phases may be broken out into separate tools, or at least be visible and controllable by the user:

mapping

  - The first phase is *mapping* the RTL design into a set of hardware elements that are available in the target technology.

placement

  - The second phase is *placement* of the needed elements onto a physical substrate, usually a chip layout. In FPGA- and CPLD-based designs, this means assigning each needed element to a specific instance or set of instances of a programmable resource on the targeted chip. In ASIC design, this means creating and spatially packing together instances of gates, flip-flops, and other basic building blocks as needed.

routing

  - In FPGA- and ASIC-based design, the third phase is *routing*: finding or creating paths between the inputs and outputs of placed elements. In CPLD design, the interconnect is usually fixed, and resources were selected in the first place based on the available connections.

simulator
test bench
waveform editor

- The inputs to a *simulator* are the HDL model and a timed sequence of inputs for the hardware that it describes. The input sequence can be contained in or generated algorithmically by another program, called a *test bench*, usually written in the same HDL; or it can be described graphically, using another tool called a *waveform editor*. The simulator applies the specified input sequence to the modeled hardware, and then determines the values of the hardware's internal signals and its outputs over a specified period of time. The outputs of the simulator can include waveforms to be viewed using the waveform editor, text files that list signal values over simulated time, and error and warning messages that highlight unusual conditions or deviations of signal values from what's expected.

Several other useful programs and utilities may be found in a typical EDA tool suite for an HDL, including the following:

- A *template generator* creates a text file with the outline of a commonly used HDL structure, so the designer can "fill in the blanks" to create source code for a particular purpose. Templates may include input and output declarations; commonly used logic structures like decoders, adders, and registers; and test benches.

template generator

- A *schematic viewer* may create a schematic diagram corresponding to an HDL model, based on the RTL output of the compiler. Such a schematic is an accurate representation of the *function* performed by the final, synthesized circuit, but beware. If the compiler output has not yet been targeted to a particular technology and optimized, the depicted circuit structure may be quite different from the final synthesized result, especially after optimization. However, a schematic viewer may also be able to view a schematic based on the final, synthesized result, as we'll show for a few FPGA-based circuit realizations in Chapter 6 and later.

schematic viewer

- A *chip viewer* lets the designer see how the synthesis tool has physically placed and routed a design on the chip. This is important for devices like FPGAs and ASICs where layout can profoundly affect the electrical and timing performance of the final chip.

chip viewer

- A *constraints editor* lets the user define instructions and preferences to be used by the synthesizer and other tools as they do their jobs. Examples of constraints include placement and routing instructions, identification of important timing requirements, and selection among different top-level strategies available to the synthesis tool—should it optimize device speed, resource utilization, its own run time, or something else?

constraints editor

- A *timing analyzer* calculates the delays through some or all of the signal paths in the final chip, and produces a report showing the worst-case paths and their delays.

timing analyzer

- A *back annotator* inserts delay clauses or statements into the original HDL source code, corresponding to the delays calculated by the timing analyzer. This allows subsequent simulations to include expected timing, whether the source code is simulated by itself or as part of a larger system.

back annotator

The best way to learn about all these kinds of tools, and more, is to get some hands-on experience with an actual HDL tool suite, like the Vivado suite for Xilinx FPGAs which is available in a free student edition, and was used to create and debug all of the Verilog examples in this book.

4.3.4 HDL-Based Design Flow

design flow

It's useful to understand the overall HDL design environment before jumping into Verilog itself. There are several steps in an HDL-based design process, often called the *design flow*. These steps are applicable to any HDL-based design process, and are outlined in Figure 4-21.

specification, block diagram, and hierarchy

The so-called "front end" begins with a functional specification of what's to be designed, and figuring out the basic approach for achieving that function at the block-diagram level. Large logic designs, like software applications, are hierarchical, and Verilog provides a good framework for modeling hardware modules and their interfaces and filling in the details later.

coding

The next step is the actual writing of HDL code for the modules, their interfaces, and their internal details. Although you can use any text editor for this step, the editor included in the HDL's tool suite can make the job a little easier. HDL editor features may include highlighting of keywords, automatic indenting, templates for frequently used code structures, built-in syntax checking, and one-click access to the compiler.

compilation

Once you've written some code, you will want to compile it, of course. The HDL compiler analyzes your code for syntax errors and also checks it for compatibility with other modules on which it relies. It also creates the internal information that is needed for the simulator to process your design later. As in other programming endeavors, you probably shouldn't wait until the very end of coding to compile all of your code. Doing a piece at a time can prevent you from proliferating syntax errors, inconsistent names, and so on, and can certainly give you a much-needed sense of progress when the project end is far from sight!

simulation

Perhaps the most satisfying step comes next—*simulation*. The HDL simulator allows you to define and apply inputs to your design, and to observe its outputs, without ever having to build the physical circuit. In small projects, the kind you might do as homework in a digital-design class, you might generate inputs and observe outputs manually. But for larger projects, it is essential to create "test benches" that automatically apply inputs and compare them with expected outputs.

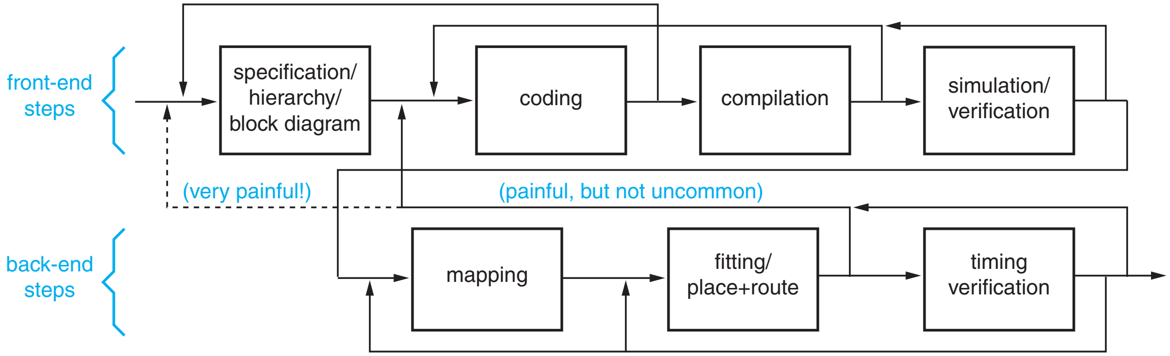


Figure 4-21 Steps in an HDL-based design flow.

Actually, simulation is just one piece of a larger step called *verification*. Sure, it is satisfying to watch your simulated circuit produce simulated outputs, but the purpose of simulation is larger—it is to *verify* that the circuit works as desired. In a typical large project, a substantial amount of effort is expended both during and after the coding stage to define test cases that exercise the circuit over a wide range of logical operating conditions. Finding bugs at this stage has a high value; otherwise, all of the "back-end" steps might have to be repeated.

verification

Note that there are at least two dimensions to verification. In *functional verification*, we study the circuit's logical operation independent of timing considerations; gate delays and other timing parameters are considered to be zero or otherwise ideal. In *timing verification*, we study the circuit's operation including estimated delays, and we verify that the setup, hold, and other timing requirements for sequential devices like flip-flops are met.

functional verification

timing verification

It is customary to perform thorough *functional* verification before starting the back-end steps. However, our ability to do accurate *timing* verification before doing the back end is often limited, since timing may be quite dependent on the results of synthesis of the complete design. Still, it can be useful to do preliminary synthesis and timing verification of a subset of the complete design, just to get a feel for whether that subset's timing performance will be adequate to support the overall timing requirements—it can only get worse later. This gives us a chance to rethink the overall design approach or the specifications if the subset's timing is unexpectedly poor at this early stage.

After verification, we are ready to move into the *back-end* stage. The nature of and tools for this stage vary depending on the target technology for the design, but there are three basic steps. As mentioned previously, the first phase in synthesis is *mapping* or converting the RTL description into a set of primitives or components that can be assembled in the target technology.

back end

mapping

For example, with PLDs or CPLDs, the synthesis tool may generate two-level sum-of-products equations for combinational logic. With FPGAs, the tool converts all combinational functions with more than a few inputs into an interconnected set of smaller functions that each fit within the FPGA's lookup tables. With ASICs, the tool may generate a list of gates and flip-flops and a *netlist* that specifies how they should be interconnected. The designer may "help" the synthesis tool by specifying certain technology-specific constraints, such as the maximum number of logic levels or the strength of logic buffers to use.

netlist

In the *fitting* step, a fitter assigns the mapped primitives or components onto available device resources. For a PLD or CPLD, this may mean assigning equations to available AND-OR elements. For an FPGA or ASIC, it may mean selecting macrocells or laying down individual gates in a pattern, then finding ways to connect them within the physical constraints of the FPGA or ASIC die; this is called the *place-and-route* process. The designer can specify additional constraints for this stage, such as the placement of modules within a chip and the pin assignments of external input and output pins.

fitting

place and route

IT WORKS!?

As a long-time logic designer and system builder, I always thought I knew what it meant when a circuit designer said, “It works!” It meant you could go into the lab, power-up a prototype without seeing smoke, and push a reset button and use an oscilloscope or logic analyzer to watch the prototype go through its paces.

But over the years, the meaning of “It works” has changed. When I took a new job in the late 1990s, I was very pleased to hear that several key ASICs for an important new product were all “working.” But just a short time later, I found out that the ASICs were working only in simulation, and that the design team and I still had to endure many arduous months of synthesizing, fitting, verifying timing, and repeating before they could even order any prototypes. “It works!”—sure. Just like my kids’ homework—“It’s done!”

post-fitting timing verification

The “final” step is *post-fitting timing verification* of the fitted circuit. It is only at this stage that the actual circuit delays due to wire lengths, electrical loading, and other factors can be calculated with reasonable precision. It is usual during this step to apply the same test cases that were used in functional verification, but in this step they are run against the circuit as it will actually be built.

As in any other creative process, you may occasionally take two steps forward and one step back (or worse!). As we suggested in Figure 4-21, during coding you may encounter problems that force you to go back and rethink your hierarchy, and you will almost certainly have compilation and simulation errors that force you to rewrite parts of the code. After timing verification, it is common to have to go back to the fitting and place-and-route processes after establishing physical constraints to help these processes achieve better results.

The most painful problems are the ones that you encounter in the back end of the design flow, because they can force you to take the most steps back. For example, if the synthesized design doesn’t fit into an available FPGA or doesn’t meet timing requirements, you may have to go back as far as rethinking your whole design approach. That’s worth remembering—excellent tools are still no substitute for careful thought at the outset of a design.

References

Digital designers who want to improve their writing should start by reading the classic *Elements of Style*, by William Strunk, Jr., E. B. White, and R. Angell (Pearson, 1999*,* fourth edition). Probably the most inexpensive and concise yet very useful guide to technical writing is *The Elements of Technical Writing*, by Gary Blake and Robert W. Bly (Pearson, 2000). For a more encyclopedic treatment, see *Handbook of Technical Writing*, by G. J. Alred, C. T. Brusaw, and W. E. Oliu (Bedford/St. Martin’s, 2015, 11th edition).

Real logic devices are described in data sheets and user manuals published by the manufacturers. Hardcopy editions of data-sheet compilations (“data

XILINX FPGA DESIGN FLOW

In Chapter 6 and beyond, we will give many examples of Verilog modules, and we often target them to Xilinx 7-series FPGAs using their Vivado tool suite. So, it is useful here to introduce the nomenclature used by Xilinx for their design flow:

elaboration

- In the *elaboration* step, the compiler reads the model's HDL file(s) and checks for syntax errors and the like. Finding none, it creates a corresponding, technology-independent RTL description of the model, using "generic" elements such as gates, multiplexers, latches, and flip-flops. Using Vivado, it is possible to view a schematic of the elaborated design.

synthesis

- The *synthesis* step converts the RTL description of the model into a hardware design using the specific hardware resources that are available in the targeted FPGA, including LUTs (lookup tables for combinational logic), specific latch and flip-flop types, and specialized elements like carry chains for adders. In Vivado, it is also possible to view a schematic of the synthesized design.

implementation

- The *implementation* step has three phases:

  optimization

  - The first is *optimization*, which checks for errors (like multiple outputs driving the same signal line) and then manipulates the synthesized logic in order to reduce resource requirements, for example by combining LUTs .

  placement

  - The second phase is *placement* , where the synthesized elements like LUTs and flip-flops are assigned to physical locations on the FPGA device.

  routing

  - The third phase is *routing*, where the placed elements' inputs and outputs are hooked up to each other using the device's programmable interconnect. Vivado cannot generate a schematic of the implemented design, but it *does* offer a view of the layout of placed elements and their interconnections.

program and debug

- The last step, which Xilinx calls *program and debug*, has utilities like "write bitstream" which generates a device programming pattern to be loaded into the FPGA for debugging in the lab or shipping the final product.

At any step along the way, it is possible to run the simulator. After elaboration, only a functional simulation is available, which assumes zero delay or otherwise ideal timing behavior. After synthesis or implementation, both functional and timing simulation are available. Both simulate the circuit's operation using the actual FPGA elements that were created in synthesis, including optimizations in the case of post-implementation simulation. When run after synthesis, timing simulation uses a rough estimate of expected delays. After implementation, it uses more accurate estimates based on the actual placement and routing results.

The tool suite also allows various "constraints" to be applied at any of the steps. In synthesis, for example, the overall strategy can be set to optimize area, performance, or the run time of the tool itself. The synthesizer can be set to "flatten" the design's hierarchy, which allows elements to be moved, shared, or otherwise optimized between the outputs of one module and the inputs of another, or to preserve the hierarchy for ease of debugging or other reasons. In implementation, various options for placement, routing, and power consumption can be enabled and disabled.

books") used to be published every few years, but they have been mostly eliminated in favor of always up-to-date information on the Web. Among the better sites for logic-family data sheets and application notes are `www.ti.com` (Texas Instruments) and `www.onsemi.com` (formerly Fairchild Semiconductor).

For a given logic family such as 74AHCT, all manufacturers list generally equivalent specifications, so you can usually get by with just one set of data sheets per family. Some specifications, especially timing, may vary slightly between manufacturers, so when timing is tight it's best to check a couple of different sources and use the worst case. That's a *lot* easier than convincing your manufacturing department to buy a component only from a single supplier.

Lots of textbooks cover digital design principles, but few cover practices. More useful for the active designer are articles written by other engineers in trade publications like *EDN*, and sometimes collected in anthologies like Clive Maxfield's books in *EDN*'s Series for Design Engineers.

Drill Problems

4.1 What documents contain reference designators? Pin numbers? Arrowheads?

4.2 Draw the DeMorgan equivalent symbol for an 8-input NAND gate.

4.3 Draw the DeMorgan equivalent symbol for a 3-input NOR gate.

4.4 What's wrong with the signal name READY′ ?

4.5 You may find it annoying to have to keep track of the active levels of all the signals in a logic circuit. Why not use only noninverting gates, so all signals are active high?

4.6 In bubble-to-bubble logic design, why would you connect a bubble output to a non-bubble input?

4.7 True or false: Either all inputs to a logic gate must have a bubble, or none may have a bubble. Justify your answer.

4.8 Redesign the alarm circuit of Figure 3-16, substituting inverting gates for the noninverting ones and adding or deleting inverters as needed. Draw a logic diagram for your circuit using the ideas of bubble-to-bubble logic design and name all the signals.

4.9 A digital communication system is being designed with twelve identical network ports. Which type of schematic structure is most appropriate for the design?

4.10 Search the Web to find Texas Instruments datasheets with information to create columns for Table 4-2 for 74AHC parts operating at 3.3 V with a 15-pF capacitive load. Provide the values needed for the first four rows in the new columns.

4.11 Determine the exact maximum propagation delay from IN to OUT of the circuit fragment in Figure X4.11 for both LOW-to-HIGH and HIGH-to-LOW transitions, using the timing information given in Table 4-2. Repeat, using a single worst-case delay number for each gate, and compare and comment on your results.

4.12 Repeat Drill 4.11, substituting 74AC00s operating at 4.5 V for the 74AC08s.

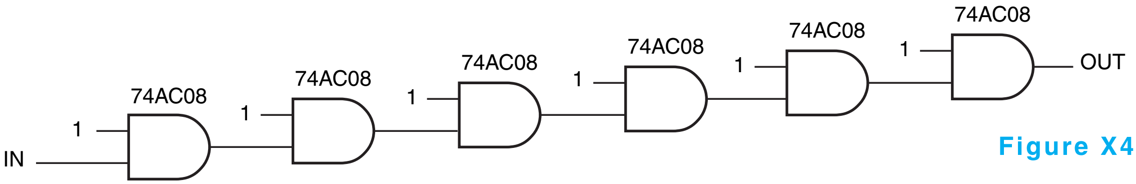


Figure X4.11

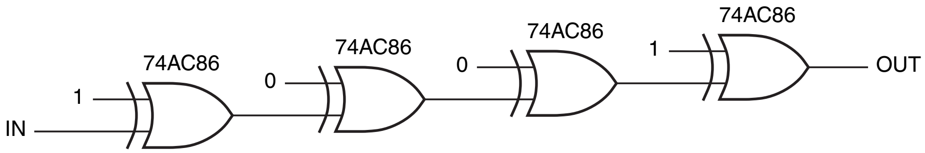


Figure X4.15

4.13 Repeat Drill 4.11, substituting 74AC21s (with three inputs at constant 1) for the 74AC08s.

4.14 Repeat Drill 4.11, substituting 74AC32s with constant 0 instead of 1 inputs.

4.15 Estimate the *minimum* propagation delay from IN to OUT for the circuit shown in Figure X4.15. Justify your answer.

4.16 Determine the exact maximum propagation delay from IN to OUT of the circuit in Figure X4.15 for both LOW-to-HIGH and HIGH-to-LOW transitions, using the timing information given in Table 4-2. Repeat, using a single worst-case delay number for each gate, and compare and comment on your results.

4.17 Repeat Drill 4.15, substituting 74HC86s operating at 4.5 V for the 74AC86s.

4.18 Estimate the *minimum* propagation delay from IN to OUT for the circuit shown in Figure X4.18. Justify your answer.

4.19 Determine the exact maximum propagation delay from IN to OUT of the circuit in Figure X4.18 for both LOW-to-HIGH and HIGH-to-LOW transitions, using the timing information given in Table 4-2. Repeat, using a single worst-case delay number for each gate, and compare and comment on your results.

4.20 Repeat Drill 4.19, substituting 74HC parts operating at 4.5 V.

4.21 What is the minimum number of different delay paths in a combinational circuit with *n* inputs and *m* outputs?

4.22 Timing specifications rarely give different specifications for LOW-to-HIGH vs. HIGH-to-LOW transitions (t_{pLH} vs. t_{pHL}) on outputs that are considered to be "data" outputs. Why?

4.23 Suppose that the microprocessor chip in your smartphone has a clock frequency of 2 Ghz, and the chip is 1 cm square. Assuming the on-chip wiring runs only on the X and Y axes, what fraction of the clock period is the speed-of-light delay for a signal transition to propagate between opposite corners of the chip?

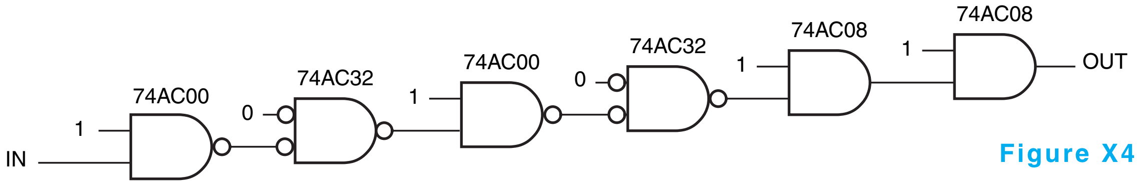


Figure X4.18

4.24 Which CMOS circuit would you expect to be faster, a 3-to-8 decoder with gate-level design similar to Figure 6-17 as shown with active-low outputs, or one with active-high outputs?

4.25 Using the information in Table 4-3 for the 74HC682 operating at 4.5 V, determine the maximum propagation delay from any input to any output in the 22-bit comparator circuit of Figure 7-27.

4.26 Repeat Drill 4.25 for the 64-bit comparator circuit of Figure 7-28.

Exercises

4.27 For what testing reasons do you think that a constant-0 or constant-1 input would be connected to the corresponding power-supply rail through a resistor rather than directly?

4.28 How many different input-to-output delay paths exist in the 5-to-32 decoder circuit of Figure 6-19? Based on the information in Table 4-3 for the 74AC138, how many paths would you actually have to analyze to determine the delays for these paths? *Hint*: Some input and output signals can be handled by group.

4.29 Using the information in Table 4-3 for the 74AC138, determine the maximum propagation delay from any input to any output in the 5-to-32 decoder circuit of Figure 6-19. Use the results of Exercise 4.28 to minimize your task.

4.30 Repeat Exercise 4.29, using 74AHCT timing with 15-pF loads, using a Texas Instruments datasheet obtained from the Web.

4.31 Using the information in Tables 4-2 and 4-3 for the 74AC139, 74AC151, and 74AC32 components, determine the maximum propagation delay from any input to any output in a 32-to-1 multiplexer circuit similar to Figure 6-33. To simplify the analysis, group inputs and outputs as appropriate.

4.32 Repeat Exercise 4.31, using the 74AC20 and the $\overline{\mathsf{Y}}$ output of the 74AC151s.

4.33 Using the information in Tables 4-2 and 4-3 for the 74HC20 and the 74HC148 operating at 2.0 V, determine the maximum propagation delay from any input to any output in a 32-to-5 priority encoder similar to Figure 7-13. A 74HC148 has active-low inputs and outputs and replaces each “cascadable priority encoder,” and you will also need to pick appropriate parts for the OR functions. Note that for this exercise you don’t need to understand how the circuit works; you just need to find and analyze all of the delay paths. *Hint*: You don’t have to compute the delay on every possible path, even after grouping the input and output signals. You should be able to “eyeball” the circuit structure to recognize only a handful of paths that may be able to produce the worst-case delay.

4.34 Using the information in Tables 4-2 and 4-3 for 74HC operating at 2.0 V, determine the maximum propagation delay from any input to any output in an error-correcting circuit similar to Figure 7-18. Note that the available 3-to-8 decoder has active-low outputs, and you will have to compensate by adding inverters. Show how to do this without increasing the overall maximum delay of the circuit.

4.35 Using the information in Table 4-3 for 74AC parts, determine the maximum propagation delay from any input to any output in the 16-bit adder of Figure 8-7.

```
module ButGate
  (A, B, C, D, Y, Z);
input A, B, C, D;
output reg Y, Z;

always @ (A, B, C, D)
  begin
    if ((A==B)&&(C!=D))
      Y = 1;
    else Y = 0;
    if ((C==D)&&(A!=B))
      Z = 1;
    else Z = 0;
  end
endmodule
```

chapter 5

Verilog Hardware Description Language

Verilog HDL, or simply *Verilog*, was introduced by Gateway Design Automation in 1984 as a proprietary hardware description and simulation language. The introduction of Verilog-based synthesis tools in 1988 by then-fledgling Synopsys, and the 1989 acquisition of Gateway by Cadence Design Systems, were important events that led to widespread use of the language.

Verilog synthesis tools can create logic-circuit structures directly from Verilog behavioral models and target them to a selected technology for realization. Using Verilog, you can design, simulate, and synthesize just about any digital circuit, from a handful of combinational logic gates to a complete microprocessor-based system on a chip.

One thing led to another, and in 1993 the IEEE was asked to formally standardize the language as it was being used at that time. So, the IEEE created a standards working group which produced IEEE 1364-1995, the official Verilog standard published in 1995 (*Verilog-1995*). By 1997, the Verilog community, including users and both simulator and synthesizer suppliers, wanted to make several enhancements to the language, and an IEEE standards group was reconvened. The result was a superset of Verilog-1995, IEEE standard 1364-2001 (*Verilog-2001*).

A few years later, the IEEE standards group made some spec corrections and clarifications and added a few new language features (not used in this book) in publishing standard 1364-2005, also known as *Verilog-2005*.

Language development then continued in a new working group which created IEEE standard 1800-2009, also known as *SystemVerilog*. The 2009 standard has Verilog-2001/2005 as a subset, and includes significant new functionality for specifying, designing, and verifying the correctness of larger systems. If you continue in digital system design, you will undoubtedly use SystemVerilog at some point.

Important features of Verilog-2001/2005 include the following:

- Designs may be decomposed hierarchically.
- Each design element has both a well-defined interface (for connecting it to other elements) and a precise functional specification (for simulating it).
- Functional specifications can use either a behavioral algorithm or an actual hardware structure to model an element's operation. For example, an element can be modeled initially by an algorithm to allow design verification of higher-level elements that use it. Later, the algorithmic model can be replaced by a preferred hardware structure.
- Concurrency, timing, and clocking can all be modeled. Verilog handles asynchronous as well as synchronous sequential-circuit structures.
- The logical operation and timing behavior of a design can be simulated.

In summary, Verilog started out as a documentation and modeling language, allowing the behavior of digital-system designs to be fairly accurately modeled and simulated. But the availability of synthesis tools that translate a Verilog model into an actual hardware realization are what led to its widespread use.

VERILOG AND VHDL

Today, Verilog and VHDL both enjoy widespread use and share the logic synthesis market perhaps 60/40. Verilog has its syntactic roots in C and is in some respects an easier language to learn and use, while VHDL is more like Ada (a DoD-sponsored software programming language). Verilog initially had fewer features than VHDL to support large project development, but with new features added in 2001, and especially with SystemVerilog, it has caught up and gone beyond VHDL.

Comparing the pros and cons of starting out with one language versus the other, David Pellerin and Douglas Taylor probably put it best in their book, *VHDL Made Easy!* (Prentice Hall, 1997):

> Both languages are easy to learn and hard to master. And once you have learned one of these languages, you will have no trouble transitioning to the other.

While writing an older Verilog/VHDL edition of this book, I found their advice to be generally true. But it *was* hard to go back and forth between the two on a daily or even weekly basis. Since you have this book in hand, my advice to you is to learn Verilog well, and to tackle VHDL later.

Today, virtually all commercial Verilog compilers and related tools support Verilog-2001, as opposed to only the Verilog-1995 feature subset, and that's what we use in this book.

In this chapter, we focus mainly on Verilog's general language structure and its use in combinational logic design. Towards the end, we will introduce the one additional feature that is needed to support sequential logic design, which we'll use for the first time towards the end of Chapter 9.

JUST-IN-TIME VERILOG

This chapter aims to be a well organized and complete reference as well as a concise tutorial on the most commonly used Verilog language elements. However, people have different learning styles, and it *is* a bit of a slog to try to learn Verilog or any language at one sitting, especially if you don't have to build anything with it yet.

Therefore, on the assumption that you may very well want to jump ahead to the "good stuff" soon, I've written boxed comments titled "Just-in-Time Verilog" in Chapters 6 and 7, whenever a Verilog concept or feature is used in an example for the first time. I've tried to give enough information there to let you work through the examples without having to come back here. In most cases, however, you'll eventually want to look up many of the features here for precise and complete definitions, especially as you begin to write your own models and things aren't quite working as you expect them to.

For now, I would recommend that you take the time to read at least the first two or three sections in this chapter, or if you're really impatient, just the first.

5.1 Verilog Models and Modules

module

The basic unit of design and programming in Verilog is a *module*—a text file containing declarations and statements, as shown in Figure 5-1(a). A typical Verilog module may correspond to a single piece of hardware, in much the same sense as a "module" in traditional hardware design. The hardware is said to be *modeled* by a single module or by a collection of modules working together.

hardware model

declarations

A Verilog module has *declarations* that describe the names and types of the module's inputs and outputs, as well as local signals, variables, constants, and functions that are used strictly internally to the module, and are not visible outside. The rest of the module contains *statements* that specify or "model" the operation of the module's outputs and internal signals.

statements

behavioral model

Verilog statements can specify a module's operation *behaviorally*; for example, by making assignments of new values to signals based on tests of logical conditions, using familiar constructs like `if` and `case`. They can also specify the module's operation *structurally*. In this formulation, the statements define instances of other modules and individual components to be used (like gates and flip-flops) and specify their interconnections, equivalent to a logic diagram.

structural model

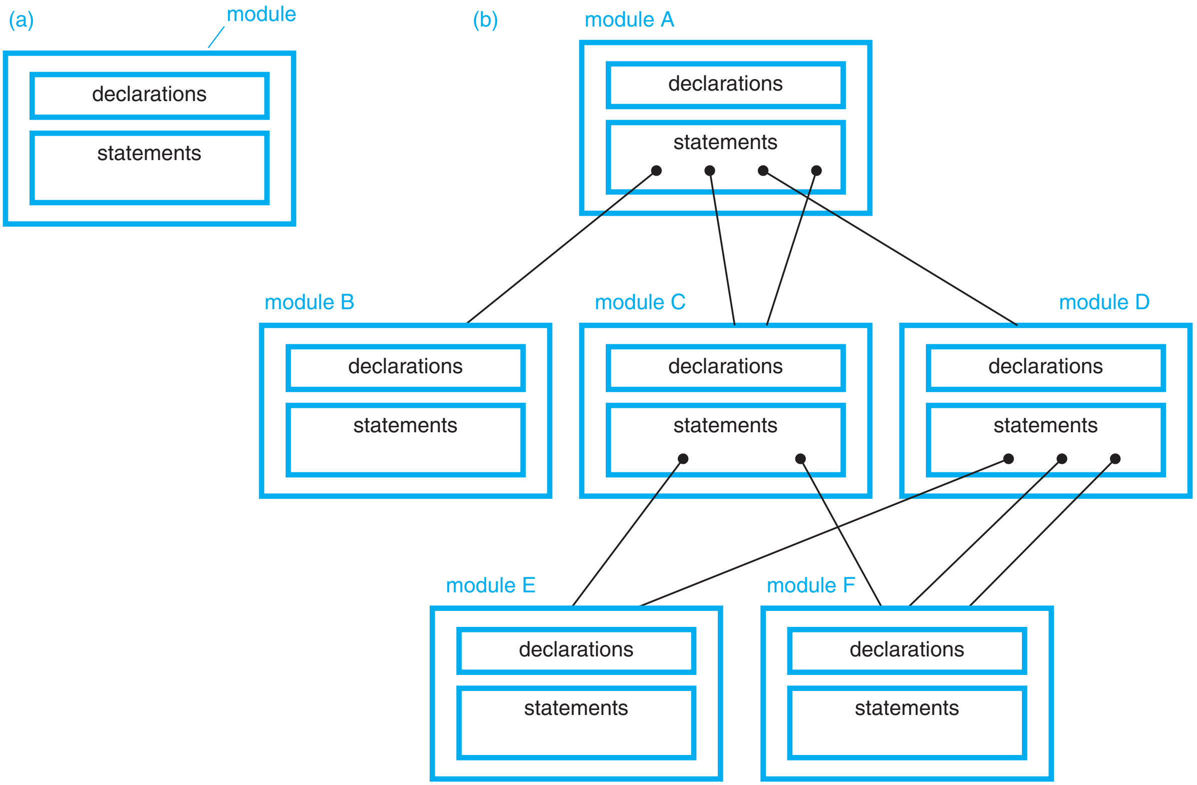


Figure 5-1 Verilog modules: (a) one module; (b) modules instantiating other modules hierarchically.

Verilog modules can use a mix of behavioral and structural models, and may do so hierarchically as shown in Figure 5-1(b). Just as procedures and functions in a high-level software programming language can "call" others, Verilog modules can "instantiate" other modules. A higher-level module may use a

INSTANCES AND INSTANTIATION

It's very important for you to understand what we mean by instantiation, especially if you have a software background. To put it simply, to instantiate is to create an instance, and an instance is a physical piece of hardware (or an emulation of it).

While a Verilog model will typically describe a module only once and create only one copy of the software code that emulates the module's operation, physical hardware is created in synthesis. In the synthesized design, each instance of a module is a separate piece of hardware that has the inputs and outputs specified in its instantiation, and that performs the operations specified in the module's definition. Each instance of a given module operates independently of and in parallel with other instances of the same module. So, as much as a module might remind you of a software procedure or subroutine, it's really quite different.

ONE MODULE PER FILE, PLEASE

The Verilog language specification allows multiple modules to be stored in a single text file, often named with a suffix of "`.v`". However, most designers put just one module in each file, with the filename based on the module name, for example, `adder.v` for module `adder`. This just makes it easier to keep track of things.

MODULE NAMES IN THIS BOOK

Most of the module names in this book begin with the letters "`Vr`". That's a practice that I started to conveniently distinguish them and their files from VHDL modules in an earlier edition of this book, and perhaps in a future edition. You don't have to use this prefix in your own modules, of course, but if you ever create HDL models in a commercial environment, you will undoubtedly be required to follow some other local naming practices.

lower-level module multiple times, and multiple top-level modules may use the same lower-level one. In the figure, modules B, E, and F stand alone; they do not instantiate any others. In Verilog, the *scope* of signal, constant, and other definitions remains local to each module; values can be passed between modules only by using declared input and output signals.

scope

Verilog's modular approach provides a great deal of flexibility in designing large systems, especially when multiple designers and design phases are involved. For example, a given module can be specified with a rough behavioral model during the initial phase of system design, so that overall system operation can be checked. Later, it can be replaced with a more intricate behavioral model for synthesis, or perhaps with a hand-tuned structural model that achieves higher performance than one synthesized from the behavioral model.

Now we're ready to talk about some of the details of Verilog syntax and model structure. A simple example module is shown in Program 5-1. Like other high-level languages, Verilog mostly ignores spaces and line breaks, which may be used as desired for readability. Short *comments* begin with two slashes (`//`) anywhere in a line and stop at the line's end. Verilog also allows C-style, multiline long comments that begin anywhere with `/*` and end anywhere with `*/`.

comments

Program 5-1 Verilog model for an "inhibit" gate.

```
module VrInhibit( X, Y, Z ); // also known as 'BUT-NOT'
  input X, Y;                // as in 'X but not Y'
  output Z;                  // (see [Klir, 1972])

  assign Z = X & ~Y;
endmodule
```

CONFIGURATION MANAGEMENT

When one Verilog module instantiates another, the compiler finds the other by searching the current workspace, as well as predefined libraries, for a module with the instantiated name. Verilog-2001 actually allows you to define multiple versions of each module, and it provides a separate configuration management facility that allows you to specify which one to use for each different instantiation during a particular compilation or synthesis run. This lets you try out different approaches without throwing away or renaming your other efforts. Still, we won't use or further discuss that facility in this text.

reserved words
keywords

Verilog defines many special character strings, called *reserved words* or *keywords*. Our example includes a few—`module`, `input`, `output`, `assign`, and `endmodule`. Following the practice of typical Verilog text editors, we use `color` for keywords in Verilog code in this book.

identifiers

User-defined *identifiers* begin with a letter or underscore and can contain letters, digits, underscores (`_`), and dollar signs (`$`). (Identifiers that start with a dollar sign refer to built-in system functions.) Identifiers in the example are `VrInhibit`, `X`, `Y`, and `Z`. Unlike VHDL, Verilog is sensitive to case for both keywords (lowercase only) and identifiers (`XY`, `xy`, and `Xy` are all different). Case sensitivity can create problems in projects containing modules in both languages, when the same identifier must be used in both Verilog and VHDL, but most compilers provide renaming facilities to deal with it in large projects. Still, it's best not to rely on case alone to distinguish two different identifiers.

module declaration
`module` keyword
input and output ports

The basic syntax for a Verilog *module declaration* is shown in Table 5-1. It begins with the keyword *`module`*, followed by an identifier for the module's name and a list of identifiers for the module's *input and output ports*. The input and output ports are signals that the modules uses to communicate with other modules. Think of them as wires, since that's what they usually are in the module's realization.

Next comes a set of optional declarations that we describe in this and later sections. These declarations can be made in any order. Besides the declarations shown in Table 5-1, there are a few more that we don't use in this book and have therefore omitted. Concurrent statements, introduced in Section 5.7, follow the declarations, and a module ends with the *`endmodule`* keyword.

`endmodule` keyword

OPTIONAL?

We said that the declarations in Table 5-1 are optional, and that's true even for input, output, and inout declarations if the module does not have the corresponding port types. For example, most modules do not have inout ports. A module that generates a clock signal would have an output port but might have no inputs. And a test-bench module, discussed later, has no inputs or outputs.

```
module module-name (port-name, port-name, ..., port-name);
  input declarations
  output declarations
  inout declarations
  net declarations
  variable declarations
  parameter declarations
  function declarations
  task declarations

  concurrent statements
endmodule
```

Table 5-1
Syntax of a Verilog module declaration.

```
input  identifier, identifier, ..., identifier;
output identifier, identifier, ..., identifier;
inout  identifier, identifier, ..., identifier;

input  [msb:lsb] identifier, identifier, ..., identifier;
output [msb:lsb] identifier, identifier, ..., identifier;
inout  [msb:lsb] identifier, identifier, ..., identifier;
```

Table 5-2
Syntax of Verilog input/output declarations.

input, output, and inout declarations

Each port that is named at the beginning of the module, in the input/output list, must have a corresponding *input*, *output*, or *inout declaration*. The simplest form of these declarations is shown in the first three lines of Table 5-2. The keyword `input`, `output`, or `inout` is followed by a comma-separated list of the identifiers for signals (ports) of the corresponding type. The keyword specifies the signal direction as follows:

`input` keyword

`input` The signal is an input to the module.

`output` keyword

`output` The signal is an output of the module. Note that the value of such a signal cannot necessarily be "read" inside the module architecture, only by other modules that use it. A "`reg`" declaration, shown in the next section, is needed to make it readable.

`inout` keyword

`inout` The signal can be used as a module input or output. This mode is typically used for three-state input/output pins.

range specification

vector

An input/output signal declared as described above is one bit wide. Multibit or "vector" signals can be declared by including a *range specification*, [*msb*:*lsb*], in the declaration as in the last three lines of Table 5-2. Here, *msb* and *lsb* are integers that indicate the starting and ending indexes of the individual bits within a *vector* of signals. The signals in a vector are ordered from left to right, with *msb* giving the index of the leftmost signal. A range can be ascending or descending; that is, `[7:0]`, `[0:7]`, and `[13:20]` are all valid 8-bit ranges. We'll have more to say about vectors in Section 5.3.

LESS ANTSY DEFINITIONS

Verilog-2001 introduced a second way for module ports to be defined, so-called "ANSI-style" declarations, similar to the function definition style used in ANSI C. In this style, the signal direction, optional range specification, and name of each port is given in the parenthesized, comma-separated list following the module name, instead of in separate declarations. As a simple example, the first three lines in Program 5-1 on page 181 become:

```
module VrInhibit( input X, Y,
                  output Z   );
```

One benefit of ANSI-style declarations is that they avoid the redundancy and error potential of writing each signal name twice, in the module's port-name list and in subsequent declarations. Another benefit is that if they are written one per line, as below, they provide room for and encourage the use of separate comments that explain the purpose of each signal:

```
module VrInhibit (    // also known as 'BUT-NOT'
  input X,            // noninverting input
  input Y,            // inverting input
  output Z            // output 'X but not Y'
);
```

That's a second declaration style that we'll use occasionally in this book. Note that if the port type is not mentioned, it still defaults to `wire`.

5.2 Logic System, Nets, Variables, and Constants

Verilog uses a simple, four-valued logic system. A 1-bit signal can take on one of only four possible values:

`0` Logical 0, or false

`1` Logical 1, or true

`x` An unknown logical value

`z` High impedance, as in three-state logic (see Section 7.1)

bitwise boolean operators

Verilog has built-in *bitwise boolean operators*, shown in Table 5-3. The AND, OR, and XOR operators combine a pair of 1-bit signals and produce the expected result, and the NOT operator complements a single bit. The XNOR operation can be viewed either as the complement of XOR, or as XOR with the second signal complemented, corresponding to the two different symbols shown in the table. (XOR and XNOR were introduced in Exercises 3.30 and 3.31.)

In Verilog's boolean operations, if one or both of the input signals is `x` or `z`, then the output is `x` unless another input dominates. That is, if at least one input of an OR operation is 1, then the output is always 1; if at least one input of an AND operation is 0, then the output is always 0. Verilog's boolean operations can also be applied to vector signals, as discussed in Section 5.3.

Operator	***Operation***
&	AND
|	OR
^	Exclusive OR (XOR)
~^, ^~	Exclusive NOR (XNOR)
~	NOT

Table 5-3
Bitwise boolean operators in Verilog's logic system.

Up until now, we've used the word "signal" somewhat loosely. Verilog actually has two classes of signals—nets and variables. A *net* corresponds roughly to a wire in a physical circuit, and provides connectivity between modules and other elements in a Verilog structural model. The signals in a Verilog module's input/output port list are often nets. We'll come back to variables later.

net

Verilog provides several kinds of nets, which can be specified by type name in net declarations. The default net type is *`wire`*—any signal name that appears in a module's input/output port list but not declared to be some other type is assumed to be type `wire`. A `wire` net provides basic connectivity, but no other functionality is implied.

`wire` keyword

Verilog also provides several other net types, shown in Table 5-4. The `supply0` and `supply1` net types are considered to be permanently wired to the corresponding power-supply rail, and provide a source of constant logic-0 and logic-1 signals. The remaining types allow modeling of three-state and wired-logic connections in a board-level system. They are seldom used inside an ASIC, FPGA, or CPLD design, except for modeling external-pin connections to other three-state devices. Note that the net types are all reserved words.

The syntax of Verilog *net declarations* is similar to an input/output declaration, as shown in Table 5-5 for `wire` and `tri` net types. A list of identifiers follows the keyword for the desired net type. For vector nets, a range specification precedes the list of identifiers.

net declaration

Keep in mind that net declarations have two uses: to specify the net type of a module's input/output ports, if not `wire`; and to declare signals (nets) that will be used to establish connectivity in structural descriptions inside a module. We'll see many examples of the latter in Sections 5.7 and 5.8.

`wire`	`trior`	`trireg`	`supply0`
`tri`	`tri0`	`wand`	`supply1`
`triand`	`tri1`	`wor`	

Table 5-4
Verilog net types.

```
wire identifier, identifier, ..., identifier;
wire [msb:lsb] identifier, identifier, ..., identifier;

tri identifier, identifier, ..., identifier;
tri [msb:lsb] identifier, identifier, ..., identifier;
```

Table 5-5
Syntax of Verilog `wire` and `tri` net declarations.

Table 5-6
Syntax of Verilog `reg` and `integer` variable declarations.

```
reg identifier, identifier, ..., identifier;
reg [msb:lsb] identifier, identifier, ..., identifier;

integer identifier, identifier, ..., identifier;
```

variable

Verilog *variables* store values during a Verilog model's execution, and they need not have physical significance in a circuit. They are used only in "procedural code," discussed in Section 5.9. A variable's value can be used in an expression and can be combined with and assigned to other variables, as in conventional software programming languages. The most commonly used Verilog variable types are `reg` and `integer`.

variable declaration

`reg` keyword

A ***`reg`*** variable is a single bit or a vector of bits, declared as shown in the first two lines of Table 5-6. The value of a 1-bit `reg` variable is always 0, 1, `x`, or `z`. The main use of `reg` variables is to store values of bits in procedural code.

`integer` keyword

An ***`integer`*** variable is declared as shown in the last line of Table 5-6. Its value is a signed, two's-complement number of 32 bits or more, depending on the word length used by the Verilog tools. An `integer` variable is typically used to control a repetitive statement, like a `for` loop, in Verilog procedural code. Integers in an actual circuit are normally modeled using multibit vector signals, as discussed in Section 5.3.

The difference between Verilog's nets and variables is subtle. A variable's value can be changed only by procedural code within a module; it cannot be changed from outside. Thus, input and inout ports cannot have a variable type; they must have a net type like `wire`. Output ports, however, can have either a net or a `reg` type, and can drive the input and inout ports of other modules.

Another important difference, as we'll see later, is that procedural code can assign values only to variables, not nets. If an output port is declared with type `reg`, the module's procedural code can use it like any other `reg` variable, but its value is always present on the output port for connection to other modules.

A reg IS NOT A FLIP-FLOP

The variable-type name "`reg`" in Verilog has nothing to do with the flip-flops and registers found in sequential circuits. If you already know what those are, please disassociate them with "`reg`" right now or you risk being confused for a long time!

Remember, Verilog was originally designed for simulation only, and when its designers came up with "`reg`," they were thinking in terms of storage registers, or *variables*, that are used during the simulator's program execution to keep track of modeled values. So, `reg` variables can be used to model sequential *or* combinational circuit outputs. A `reg` variable may contain a bit or a vector of bits. It's too bad that Verilog's designers didn't use a better keyword, like "`var`" or "`bitvar`".

Sequential-circuit latches, flip-flops, and registers are defined in Verilog by an entirely different mechanism, to be introduced in Section 10.3.2.

The result of all this is that if you want to write procedural Verilog code to specify the value of a module output, you have basically two ways to do it:

1. Declare the output port to have type `reg`, and use procedural code to assign values to it directly.
2. If for any reason the port must be declared as a net type (like `tri`), define an internal "output" `reg` variable and specify its value procedurally. Then assign the value of the internal `reg` variable to the module output net.

literals

Verilog has its own particular syntax for writing numeric *literals*, tailored to its use in describing digital logic circuits. Literals that are written as a sequence of decimal digits, with no other frills except an optional minus sign, are interpreted as signed decimal numbers, as you would expect. Verilog also gives you the ability to write numeric literals in a specific base and number of bits, using the format *n*'*Bdd...d*, where:

- *n* is a decimal number that specifies the size of the literal in bits. This is the number of bits represented, *not* the number of digits *dd...d*.
- *B* is a single letter specifying the base, one of the following: `b` or `B` (binary), `o` or `O` (octal), `h` or `H` (hexadecimal), or `d` or `D` (decimal).
- *dd...d* is a string of one or more digits in the specified base. Hexadecimal digits `a-f` may be typed in upper or lower case. If the digits provide more than *n* nonzero bits, unneeded bits are discarded on the left. If they provide fewer than *n* nonzero bits, zeroes are appended on the left as needed.

?, question mark in literal

Examples of literals are given in Table 5-7. A *question mark "?"* used in a literal is equivalent to "z". Sized literals are interpreted as multibit vectors, as shown in examples in the next section. Literals written without a size indicator default to 32 bits or the word width used by the simulator or compiler; this may cause errors or ambiguity, so you must be careful with unsized literals.

Literal	***Meaning***
`1'b0`	A single 0 bit
`1'b1`	A single 1 bit
`1'bx`	A single unknown bit
`8'b00000000`	An 8-bit vector of all 0 bits
`8'h07`	An 8-bit vector of five 0 and three 1 bits
`8'b111`	The same 8-bit vector (0-padded on left)
`16'hF00D`	A 16-bit vector that makes me hungry
`16'd61453`	The same 16-bit vector, with less hunger
`2'b1011`	Tricky or an error, leftmost "`10`" ignored
`4'b1?zz`	A 4-bit vector with three high-Z bits
`8'b01x11xx0`	An 8-bit vector with some unknown bits

Table 5-7
Examples of literals in Verilog.

TYPING WITH LESS TYPING

You'll sometimes see a separate declaration of the net or variable type of an input or output port (like the common case of an output port having type `reg`). In Verilog-1995, that was the only way to do it. Verilog-2001 lets you identify the type within the port declaration, as in the example below:

```
module Vr3to8deca( G1, G2, G3, A, Y );
  input wire G1, G2, G3;
  input wire [2:0] A;
  output reg [0:7] Y;
```

You can do the same in ANSI-style declarations.

Table 5-8 Syntax of Verilog parameter declarations.

```
parameter identifier = value;
parameter identifier = value,
          identifier = value,
          ...
          identifier = value;
```

parameter declaration
`parameter` keyword

Verilog provides a facility for defining named constants within a module, to improve the readability and maintainability of code. A *parameter declaration* has the syntax shown in Table 5-8. An identifier is assigned a constant value that will be used in place of the identifier throughout the current module. Multiple constants can be defined in a single parameter declaration by a comma-separated list of assignments. Some examples are shown below:

```
parameter BUS_SIZE = 32,              // width of bus
          MSB = BUS_SIZE-1, LSB = 0;  // range of indices
parameter ESC = 7'b0011011;           // ASCII escape character
```

constant expression

The *value* in a parameter declaration can be a simple constant, or it can be a *constant expression*—an expression involving multiple operators and constants, including other parameters, that yields a constant result at compile time. Note that the scope of a parameter is limited to the module in which it is defined.

NOTHING TO DECLARE?

Verilog allows you to use nets that have not been declared. In structural code, you can use an undeclared identifier in a context where the compiler would allow a net to be used. In such a case, the compiler will define the identifier to be a `wire`, local to the module in which it appears.

But to experienced programmers, using undeclared identifiers seems like a bad idea. In a large module, declaring all the identifiers in one place gives you a greater opportunity to document and ensure consistency among names. Whether or not you declare all identifiers, if you mistype an identifier, the compiler will usually notice and warn you that the accidental wire (or, in some cases, the intended wire) is not being driven by anything.

5.3 Vectors and Operators

As shown earlier, Verilog allows individual 1-bit signals to be grouped together in a *vector*. Nets, variables, and constants can all be vectors. Verilog provides a number of operations and conventions related to vectors. In general, Verilog does "the right thing" with vectors, but it's important to know the details.

vector

Table 5-9 gives some example definitions of vectors that are used in this section. In a vector definition, you should think of the first (left) index in the definition as corresponding to the bit on the left end of the vector, and the second (right) index as corresponding to the bit on the right. Thus, the rightmost bit in `byte1` has index 0, and the leftmost bit in `Zbus` has index 1. As shown in the examples, index numbers can be ascending or descending from left to right.

Verilog provides a natural *bit-select* syntax to select an individual bit in a vector, using square brackets and a constant (or constant expression). Thus, `byte1[7]` is the leftmost bit of `byte1`, and `Zbus[16]` is the rightmost bit of `Zbus`. The *part-select* syntax is also natural, using the same range-specification syntax that is used in declarations. Thus, `Zbus[1:8]` and `Zbus[9:16]` are the left and right bytes of `Zbus`, and `byte1[5:2]` is the middle four bits of `byte1`. Note that the indices in a part-select should be in the same order as the range specification in the original definition.

[], bit-select

[:], part-select

Just as bits and parts can be extracted from vectors, so can bits and parts be put together to create larger vectors. *Concatenation* uses curly brackets `{}` to combine two or more bits or vectors into a single vector. Thus, `{2'b00,2'b11}` is equivalent to `4'b0011`, and `{byte1,byte1,byte2,byte2}` is a 32-bit vector with two copies of `byte1` on the left, and two copies of `byte2` on the right. Verilog also has a *replication operator* *n*`{}` that can be used within a concatenation to replicate a bit or vector *n* times. Thus, `{2{byte1},2{byte2}}` is the same 32-bit vector that we defined two sentences ago. If `N` is a constant (like a parameter), then `{N{1'b1}}` is an `N`-bit vector of all 1s.

{}, concatenation operator

n{}, replication operator

The bitwise boolean operators listed in Table 5-3 also work (surprise, surprise) "bitwise" on vectors. For example, the expression `(byte1 & byte2)`

```
reg [7:0] byte1, byte2, byte3;
reg [15:0] word1, word2;
reg [1:16] Zbus;
```

Table 5-9 Examples of Verilog vectors.

"OOPS" VECTOR OPERATIONS

If you try to reference (read) part of a vector using an index in a bit-select or part-select that is outside of the vector's defined range, a value of "x" (unknown) is returned for any bits that are out of range. Conversely, if you try to write a value into part of a vector partially or completely outside of its defined range, the out-of-range portion of the assignment is ignored, but the rest of it works.

yields an 8-bit vector where each bit is the logical AND of the bits in the corresponding position of `byte1` and `byte2`; the value of `(4'b0011 & 4'b0101)` is `4'b0001`; and the value of `~3'b011` is `3'b100`.

padding

Vectors of different sizes also can be combined by the bitwise boolean operators. The vectors are aligned on their rightmost bits, and the shorter vector is padded on the left with 0s. Thus, the value of `2'b11 & 4'b1101` is equivalent to `4'b0011 & 4'b1101` and has the value `4'b0001`.

Zero-padding also applies in general to literals. Thus, `16'b0` is a 16-bit constant in which all the bits are zero. However, if the literal's leftmost specified bit is `x` or `z`, then the vector is padded with `x` or `z`. Thus, `8'bx` is an 8-bit vector of all `x`'s, and `8'bz00` is equivalent to `8'bzzzzzz00`.

Later, we'll see assignment statements where the value of a vector expression is assigned to a vector net or variable. If the vector expression's result size is smaller than the size of the net or variable, then it is padded on the left with 0s. If the result size is wider than the net or variable, then its rightmost bits are used. However, if the result of the expression is an *integer*, and the system's integer width is narrower than the net or variable, then the integer value is sign extended before being assigned to the vector net or variable, so be careful!

arithmetic operators

Verilog has built-in *arithmetic operators*, shown in the first six rows of Table 5-10, that treat vectors as unsigned integers by default; but they can also be treated as signed, two's-complement integers as described in the boxed comment. An unsigned integer value is associated with a vector in the "natural" way. The rightmost bit has a weight of 1, and bits to the left have weights of successive powers of two. This is true regardless of the index range of the vector. Thus, if the constant `4'b0101` is assigned to the `Zbus[1:16]` variable that we defined earlier, the integer value of `Zbus` is 5.

Addition and subtraction are the most often used arithmetic operators, and Verilog synthesis tools know how to synthesize adders and subtractors; for example, see Section 8.1.8. Unary plus and minus are also allowed. Multiplica-

Table 5-10 Arithmetic and shift operators in Verilog.

Operator	***Operation***
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`%`	Modulus (remainder)
`**`	Exponentiation
`<<`	(Logical) shift left
`>>`	(Logical) shift right
`<<<`	Arithmetic shift left
`>>>`	Arithmetic shift right

tion is also handled by most synthesis tools, although the multiplier's size and speed may not be as good as what you would get from a hand-tuned design (see Section 8.3.2). Depending on the tool, division and modulus might not be synthesizable unless the divisor is a constant power of two. In that case, the operation is equivalent to shifting the dividend to the right (division) or selecting the rightmost bits of the dividend (modulus). Exponentiation is used primarily in test benches.

Verilog also has explicit *shift operators*, for vectors, shown in the next two rows of Table 5-10. These are sometimes called *logical shift operators* to distin-

shift operators
logical shift operators

SIGNED ARITHMETIC

Verilog-2001 provides for signed as well as unsigned arithmetic. A `reg` variable, a net, or a function output can be declared as signed by including the keyword `signed` in the declaration, for example, "`reg signed [15:0] A`". Likewise, a module's ports may be declared as signed, for example, "`output reg signed [15:0] T`".

Integer variables, as well as plain integer literals, are always considered to be signed. Other numeric literals are signed if the letter "`s`" or "`S`" is included before the base, for example, `8'sb11111111` is an 8-bit two's-complement number, with an integer value of -1. The sign bit of a signed literal is its leftmost bit at its specified width. So, the sign bit of `4'sb1101` is 1 and its integer value is -3. But the sign bit of `5'sb1101` is 0 (after 0-padding one bit on the left) and its integer value is 13.

Signed operations and comparisons follow the rules for two's-complement arithmetic, in both simulation and synthesis. However, signed operations are used only if *all* of an expression's operands are signed. Otherwise, its signed operands are converted to unsigned *before* the expression is evaluated. For example, if you wrote "`4'sb1101 + 1'b1`", the integer value of the sum would be 14 (decimal $13 + 1$), not $-3 + 1 = -2$ as you might have intended, since `1'b1` is unsigned. To get a signed interpretation, you could write, for example, "`4'sb1101 + 1`," since integers are always considered to be signed.

In another example, suppose you wrote "`(4'sb1110 << 1) + 1`". This expression appears to shift the first operand left yielding 1100, the signed representation of -4, and add 1 to that for a final result of -3. But its actual integer result is 13 decimal—how can that be? The problem is that the shift operator << is a *logical* shift, so its result is interpreted as *un*signed, or +12 decimal. To preserve the signed type of the shifted result, you must use the *arithmetic* shift operator <<<.

In yet another seemingly simple but actually complicated scenario, suppose you assign the value of an integer variable `I` to a wide, unsigned vector `W`. In this special case, if `W` is wider than the system's integer width, then `I` is sign-extended to `W`'s width for the assignment. But suppose that `I` is first assigned to a vector `V` with the same width as integers, and then `V` is assigned to `W`. In that case, a standard vector assignment occurs; `V` is 0-extended regardless of its MSB's value.

As you've seen, signed operations can be error-prone, given Verilog's rules that often convert operands and results as explained above, often without warning. So, if you need to use signed operations, be careful!

guish them from the arithmetic shift operators discussed in a moment. The first operand is shifted by a number of positions given by the second operand, and vacated positions are filled with 0s. Thus, the value of `8'b11010011<<3` is `8'b10011000`. In a right shift, the leftmost bits are always filled with 0s as well; so the value of `8'b11010011>>3` is `8'b00011010`.

arithmetic shift operators

The results produced by the *arithmetic shift operators*, in the last two rows of the table, depend on the type of their first operand, unsigned or signed. If the first operand is unsigned, the result is the same as with the corresponding logical shift operator; 0s are shifted into the vacated positions and the result is considered to be unsigned. If the first operand of a right arithmetic shift is signed, then the sign (leftmost) bit of the operand is shifted into the vacated positions. If the first operand of a left arithmetic shift is signed, then the vacated positions are filled with 0s, which would appear to give the same result as a left logical shift. But there is a subtle difference, discussed in the box on signed arithmetic.

boolean reduction operators

Besides its frequently used bitwise boolean operators, Verilog also has infrequently used *boolean reduction operators*. These operations use the same operator symbols as in the first four rows of Table 5-3 on page 185, but they take

NOT MY TYPE

Unsigned operands in expressions can be converted to signed using Verilog's built-in type-casting function `$signed()`. So, if `A` is a signed vector and `B` is an unsigned vector, then "`A+$signed(B)`" is their signed sum. Similarly, signed operands may be converted to unsigned using the built-in function `$unsigned()`.

Note that the type-casting functions don't do anything if you're casting to a narrower bit width; the high-order bits of the wider operand are merely discarded in the usual way.

You must be careful when working with signed operands because there are several Verilog operations which, when applied to a signed operand, produce an *un*signed result:

- Logical shifts (<<, >>): Only the arithmetic shifts produce signed results.
- Part-selects: Even if a vector is signed, a selected part is unsigned, even if it includes the original sign bit, for example, writing `A[15:8]` to select the high-order byte of signed variable `A` in the example in the previous boxed comment.
- A single bit is always unsigned. Including a bit in an arithmetic expression involving other, signed operands still yields an unsigned result. For example, the result of "`A+1'b1`" is unsigned, as is the result of "`A+CIN`" if `CIN` is a 1-bit variable (carry-in).

You can convert a single unsigned bit to signed using `$signed`, but still you must be very careful. The integer value of `1'sb1` is −1 , not +1, since its leftmost bit is treated as the sign bit and is sign-extended. Changing the example in the last bullet to "`A+$signed(CIN)`" still yields an unexpected result. The carry-in must be at least two bits wide with a sign bit of 0 to get the expected signed result; for example, "`A+$signed({1'b0,CIN})`".

a single vector operand. They combine all of the bits in the vector using the corresponding operation, and return a 1-bit result. Thus, the value of `&Zbus` is `1'b1` if all the bits of `Zbus` are 1; else it is `1'b0`. Similarly, the value of `^byte1` is `1'b1` if `byte1` has an odd number of 1s; else it's `1'b0`. (To be precise, the results may be `1'bx` if any operand bit is `z` or `x`, unless another input dominates.)

5.4 Arrays

Verilog-1995 had a limited capability to define and use one-dimensional arrays of `reg` and `integer` variables. Verilog-2001 extended this capability to allow multidimensional arrays and to allow net-type array elements like `wire`.

array
array index

An *array* is an ordered set of variables of the same type, where each element is selected by an *array index*. The basic array declaration formats are shown in the first three rows of Table 5-11, and have been supported ever since Verilog-1995. Here, the `reg` or `integer` identifier is followed by an array-index range in square brackets. In the range, *start* and *end* are integer constants or constant expressions that define the possible range of the array index and hence the total number of array elements.

Array elements can be bits (first line of the table), vectors (second line), or integers (third line). In Verilog-2001, elements can also be `net` types like `wire` or vectors of them (fourth and fifth lines). Multiple variables of the same type and size, including arrays of different sizes, can be defined in a single declaration, for example:

```
reg [7:0] byte1, recent[1:5], mem1[0:255], cache[0:511];
```

Here, `byte1` is an 8-bit vector, and the other variables are arrays containing 5, 256, and 512 8-bit vectors, respectively.

Individual array elements are accessed using the array name followed by the index of the desired element, enclosed in square brackets. For example, `recent[1]` is the first 8-bit-vector element in the `recent` array, and `recent[i]` is the *i*th element, assuming that `i` is an `integer` variable and its value is in the range 1 to 5. Verilog-1995 did not provide a means to directly access individual bits of a vector array element; you had to first copy the array element to a like-size `reg` variable or net, and then access the desired bit(s) using a bit or part-select. For example, to read bit 5 of `mem1[117]`, you could copy `mem1[117]` to `byte1`, and then access `byte1[5]`. Verilog-2001 is more capable—you can use a part-select as a second index and write `mem1[117][5]` in the previous example, or write `mem1[i][3:0]` to access the low-order nibble of byte `i`.

```
reg identifier [start:end];
reg [msb:lsb] identifier [start:end];
integer identifier [start:end];
wire identifier [start:end];
wire [msb:lsb] identifier [start:end];
```

Table 5-11
Syntax of Verilog array declarations.

MULTI-DIMENSIONAL ARRAYS

Verilog-2001 supports multidimensional arrays. In the declaration, a [*start*:*end*] index range for each additional dimension is written after the array name; and to access an element, an index must be provided for each dimension. Thus, you could declare a two-dimensional array of bytes, "`reg [7:0] mem3 [1:10] [0:255]`" and access the lower nibble of the byte in row 5, column 7 as `mem3[5][7][3:0]`.

With the availability of multidimensional arrays, you have an alternate way to declare a one-dimensional array of vectors, namely, as a two-dimensional array of bits. For example, the `mem1` example in the text could instead be declared as

```
reg mem2[0:255][7:0];
```

This stores exactly the same information as in the `mem1` declaration, but the capabilities for accessing it are more limited. You might think that you could copy a row of `mem2` into the corresponding vector of `mem1` by writing

```
mem1[i] = mem2[i][7:0];
```

But that's not legal. I wrote `[7:0]` as if it were a part-select for a vector, but I actually need to select an 8-bit subarray of the two-dimensional array. And Verilog doesn't provide for that, even though the syntax above looks good. To copy a row of bits into a vector, you need to do it one bit at a time. So, the decision of whether to declare as a one-dimensional array of vectors as such, or as a two-dimensional array of bits, depends on how you expect to access it in your code.

We give an example that uses two-dimensional arrays of bits in Program 8-17 on page 420. Near the end of a later version, Program 8-20, we need to treat two array rows as vectors in order to add them, and to do that, we use a loop to copy the individual bits into a vector variable that was declared only for that purpose. An alternative, given in Exercise 8.46, is declare the arrays as one-dimensional arrays of vectors, so the array elements (vectors) can be added directly.

5.5 Logical Operators and Expressions

true

false

Verilog has several operators and statements that rely on the concept of true/false values ("truth values"). In Verilog, a 1-bit value of `1'b1` is considered to be *true*, and `1'b0` or any "unknown" value (`x` or `z`) is considered to be *false*. With multi-bit "known" values (no `x` or `z`), any nonzero value is considered true, and only a zero value is considered false; thus, `4'b0100` is just as true as `4'b1111`; among possible 4-bit "known" values , only `4'b0000` is false (but see Exercise 5.32).

logical operators

True and false values can be created and combined in expressions by the *logical operators*, shown in Table 5-12. A logical operation yields a value of `1'b1` or `1'b0`, depending on whether the result is true or false. If such a value is assigned to a wider variable or net, it is extended on the left with 0s.

Keep in mind that in the first three logical operations, the truth or falsehood of each operand is evaluated before the operands are logically combined. For example, the expression `4'b0100 && 4'b1011` evaluates as "true `&&` true" and yields the value true in a logical expression. But the corresponding bitwise

Operator	*Operation*
`&&`	logical AND
`||`	logical OR
`!`	logical NOT
`==`	logical equality
`!=`	logical inequality
`>`	greater than
`>=`	greater than or equal
`<`	less than
`<=`	less than or equal

Table 5-12
Verilog logical operators.

boolean operation `4'b0100 & 4'b1011` has the value `4'b0000`, which would be considered false in a logical expression.

The logical equality and inequality operators do a bit-by-bit comparison of their operands, and consider them equal only if corresponding bits are equal. The magnitude comparisons in the last four rows of Table 5-12 consider their operands to be unsigned numbers if either operand is unsigned.

In all six of the comparison operations, if the operand sizes are unequal and either one is unsigned, the shorter operand is extended on the left with 0s before the (unsigned) comparison is made. Thus, the expression `2'b11 < 4'b0100` is true, `8'h0a < 4'b1001` is false, and `8'h05==4'b0101` is true.

If both operands are signed, the shorter operand is signed extended—its leftmost bit is replicated on the left before the (signed) comparison is made. Thus, the expression `2'bs11 < 4'bs0100` is true, `8'hs08 < 4'bs1001` is false, and `2'bs11==4'bs1111` is true.

In synthesis, the last six operators can create comparators which may be expensive, as discussed in the box on page 213 and in Section 7.4.6.

LOGICAL VS. BOOLEAN

It is very important to understand the difference between logical operations and the corresponding bitwise boolean operations, and to use the one that is appropriate according to the circumstances. Logical operations normally should be used only when the result is used as a conditional expression with the conditional operator “`?:`” or a procedural statement (both introduced later). Bitwise boolean operations should be used to combine bits and vectors to produce a value in combinational logic.

Because of the way Verilog defines true and false, when operands are one bit wide, logical operators have the same effect as the corresponding bitwise boolean operators. Especially in the case of NOT, you sometimes see sloppy code that mixes `!` with `&` and `|`, when what is really meant is `~`. Although this may seem OK to C programmers, it’s a bad idea, because the `!` operation will produce unintended results if those 1-bit operands are ever changed to multibit vectors.

Table 5-13
Syntax and examples of the Verilog conditional operator.

logical-expression ? *true-expression* : *false-expression*

```
X ? Y : Z

(A>B) ? A : B;

(sel==1) ? op1 : (
  (sel==2) ? op2 : (
    (sel==3) ? op3 : (
      (sel==4) ? op4 : 8'bx )))
```

`?:`, conditional operator

Verilog's *conditional operator* *`?:`* selects one of two alternate expressions depending on the value of a logical expression: the first alternative if the logical expression is true, and the second if it is false.

The conditional operator's syntax and a few examples are given in Table 5-13. In the first example, the value of the expression is `Y` if `X` is true, else it's `Z`. The second example selects the maximum of two vector operands, `A` and `B`. The last example shows how conditional operations can be nested. For complex conditional operations, parenthesization is recommended for both readability and correctness. A design example is shown in Program 5-7 on page 204.

There's an important subtlety to understand when Verilog's comparison operators are used in test benches, that is, in simulation. There, it is possible for one or more of an operand's bits to have a value of "x" or "z"—unknown or high-impedance. If any bit of any operand is `x` or `z`, the operators in Table 5-12 return a value of `x`. And `x` is treated as false by the conditional operator above

EXPRESSIONS AND OPERATOR PRECEDENCE (OR, ALWAYS OBEY YOUR PARENS!)

So far, we've introduced a bunch of Verilog net and variable types, and operators that combine them; and there are a few more to come. All of these can be combined in *expressions* that yield a value.

As in other programming languages, each Verilog operator has a certain *precedence* that determines the order in which operators are applied in a non-parenthesized expression. For example, the NOT operator `~` has higher precedence (is applied sooner) than AND and OR (`&` and `|`), so `~X&Y` is the same as `(~X)&Y`. Also, `&` has higher precedence than `|` (but see the potential trap discussed in Section 3.1.5 on page 100). Therefore, `W&X|Y&Z` is the same as `(W&X)|(Y&Z)`. But `W|X & Y|Z` does not mean what you might think from the spacing—it's the same as `W|(X&Y)|Z`.

Verilog reference manuals can show you the detailed pecking order that is defined for operator precedence. However, it's not a good idea to rely on this. You can easily slip up, especially if you move frequently among different programming languages with different operators and precedence orders. Also, others who read your code may interpret your expressions incorrectly. So, although `W|X&Y|Z` is the same as `W|(X&Y)|Z`, if that's what you want, you should write it the second way.

The best policy is to parenthesize expressions fully, except for the very common case of negating a single variable. With parentheses, there can be no confusion.

Operator	***Operation***
`===`	case equality
`!==`	case inequality

Table 5-14
Verilog "case equality" operators.

and in statements like `if` and `for`, introduced later, whose behaviors depend on the value of a logical expression.

When comparisons are performed in test benches on operands that may have unknown values (likely in testing!), it is usually more appropriate to use one of Verilog's so-called *case equality* operators in Table 5-14. These operators compare their operators bit-by-bit and consider them equal only if 0, 1, `x`, and `z` bits match in every position. They always return a value of true or false (`1'b1` or `1'b0`), never `x`. However, these operators should *not* be used in the definition of a synthesizable module, since there are no corresponding circuit elements that distinguish `x` and `z` from "real" 0s and 1s.

case equality

5.6 Compiler Directives

A Verilog compiler provides several directives for controlling compilation; we'll introduce two of them here. All compiler directives begin with an accent grave (`` ` ``). First is the *`` `include ``* compiler directive, with the syntax below:

`` `include ``

`` `include `` *filename*

The named file is read immediately and processed as if its contents were part of the current file. This facility is typically used to read in definitions that are common to multiple modules in a project. Nesting is allowed; that is, an `include`'d file can contain `` `include `` directives of its own.

Next is the *`` `define ``* compiler directive, with the syntax below:

`` `define ``

`` `define `` *identifier text*

Notice that there is no ending semicolon. The compiler textually replaces each appearance of *identifier* in subsequent code with *text*. Keep in mind that the substitution is *textual*; no expression evaluation or other processing takes place. Also, it is important to know that the definition is in effect not only in the current file, but in subsequent files that are processed during a given compiler run (for example, in files that are `include`'d by this one).

A `` `timescale `` compiler directive will be described in Section 5.11.

`define VS. parameter

Although `` `define `` can be used to define constants, such as bus sizes and range starting and ending indexes, in general it's better to use `parameter` declarations for such definitions, unless the constant is truly a global one. With `` `define ``, you run the risk that unbeknownst to you, another module or `include`'d file will change the constant's definition. Parameter definitions are local to a module.

5.7 Structural Models

concurrent statement

We're finally ready to look at the part of a Verilog model that actually specifies digital-logic operation, from which a realization is synthesized. This is the set of *concurrent statements* in the module declaration of Program 5-1 on page 183. The most important of the concurrent-statement types are instance, continuous-assignment, and `always` statements. They lead to three distinct styles of circuit design and description, which we cover in this and the next two sections.

Statements of these three types, and corresponding design styles, can be freely intermixed within a Verilog module. In Section 5.13, we'll cover one more concurrent-statement type, `initial`, that's used in test benches.

structural model
structural description

In the *structural* style of circuit description or modeling, individual gates and other components are instantiated and connected to each other using nets. This is a language-based equivalent of a logic diagram, schematic, or net list.

built-in gate types

Verilog has several *built-in gate types*, shown in Table 5-15. The names of these gates are reserved words. The `and`, `or`, and `xor` gates and their complements may have any number of inputs. A `buf` gate is a 1-input noninverting buffer, and a `not` gate is an inverter.

The remaining four gates are 1-input buffers and inverters with three-state outputs. They drive the output with the data input (or its complement) if the enable input is 0 or 1, as in the gate's name; else the output is `z`. For example, `bufif0` drives its output with its data input if the enable input is 0.

A typical design environment has libraries that provide many other predefined components such as input/output buffers for external pins, flip-flops,

Table 5-15
Verilog built-in gates.

`and`	`xor`	`bufif0`
`nand`	`xnor`	`bufif1`
`or`	`buf`	`notif0`
`nor`	`not`	`notif1`

CONCURRENT STATEMENTS AND SIMULATION

Each concurrent statement in a Verilog module "executes" simultaneously with the other statements in the same module declaration. This behavior is markedly different from that of statements in conventional software programming languages, which execute sequentially. Concurrent operation is necessary to simulate the behavior of hardware, where connected elements affect each other continuously, not just at particular, ordered time steps.

Consider a module whose last concurrent statement updates a signal that is used by the first concurrent statement. When the module's operation is simulated, the simulator will go back to that first statement and update its results using the signal that just changed. In fact, it will keep propagating changes and updating results until the model stabilizes; we'll discuss this in more detail in Section 5.12. This behavior is needed to emulate real hardware, where outputs affect inputs continuously.

```
component-name  instance-identifier  ( expr, expr, ..., expr );

component-name  instance-identifier  ( .port-name(expr),
                                       .port-name(expr),
                                       ...
                                       .port-name(expr) );
```

Table 5-16 Syntax of Verilog instance statements.

and higher-complexity functions such as decoders and multiplexers. Each of these components has a corresponding module declaration in a library.

instance statement

Gates and other components are instantiated in an *instance statement* with two syntax variants shown in Table 5-16. The statement gives the name of the component, like `and`, followed by an optional identifier for this particular instance, followed by a parenthesized list that associates component ports (inputs and outputs) with an expression (*expr*). In the case of an inout or output port, the associated *expr* must be the name of a local net to which the port connects. In the case of an input port, the *expr* can be a net name, a `reg` variable, or an expression that evaluates to a value compatible with the input-port type.

Note that instance identifiers (like "`U1`") must be unique within a module, but may be reused in different modules. The compiler creates a longer, globally unique identifier for each instance based on its position in the overall design hierarchy, which identifies the instance in system-level simulation and synthesis. Also, if the instance identifier is completely omitted, the compiler creates one.

As shown in Table 5-16, two different formats are allowed for the port-association list. The first format depends on the order in which port names appear in the original component definition. The local expressions are listed in the same order as the ports to which they connect. Built-in gates can only be instantiated using the first format, and their defined port-name order is (output, input, input, ...). The order among the multiple inputs doesn't matter when all inputs affect a gate's output in the same way. For the built-in three-state buffers and inverters, the defined order is (output, data-input, enable-input).

Using the first format, Program 5-2 shows a module that uses structural code and built-in gates to define an "inhibit" gate—basically an AND gate with one inverted input. Note in the third line of the module that the type of the output port `out` is not declared, and is therefore defined to be a `wire` by default. We

Program 5-2 Structural Verilog model for an "inhibit" gate.

```
module VrInhibit( in, invin, out ); // also known as 'BUT-NOT'
  input in, invin;                  // as in 'in but not invin'
  output out;                       // (see [Klir, 1972])
  wire notinvin;

  not U1 (notinvin, invin);
  and U2 (out, in, notinvin);
endmodule
```

Program 5-3 Structural Verilog model for an alarm circuit.

```
module VrAlarmCkt (       // Note ANSI-style port declarations
  input panic, enable, exiting, window, door, garage,
  output alarm
);
  wire secure, notsecure, notexiting, otheralarm;

  or U1 (alarm, panic, otheralarm);
  and U2 (otheralarm, enable, notexiting, notsecure);
  not U3 (notexiting, exiting);
  not U4 (notsecure, secure);
  and U5 (secure, window, door, garage);
endmodule
```

could have instead written "`output wire out`" with exactly the same effect. In this book, we'll sometimes include the "`wire`" keyword just to make it perfectly clear that we really want the default `wire`, not a `reg`.

As another example, Program 5-3 defines a module for an alarm circuit with the same inputs, output, and function as the logic diagram in Figure 3-16 on page 113. Note how we defined local wires for internal signals, including three that weren't named in the logic diagram.

Library components and user-defined modules can be instantiated with either the first or the second format. In the second format, each item in the port-association list gives the port name preceded by a period and followed by a parenthesized expression. For example, Program 5-4 instantiates two inverters and three copies of the inhibit-gate module of Program 5-2 to create a 2-input XOR gate, albeit in a very roundabout way. A corresponding logic diagram is shown in Figure 5-2.

The best coding practices use the second format only, since simple errors like transposing inputs can occur easily in the first format and can be hard to find. In the second format, port associations can be listed in any order, since they are written out explicitly.

Program 5-4 Structural Verilog model for an XOR function.

```
module VrSillyXOR(in1, in2, out);
  input in1, in2;
  output out;
  wire inh1, inh2, notinh2, notout;

  VrInh U1 ( .out(inh1), .in(in1), .invin(in2) );
  VrInh U2 ( .out(inh2), .in(in2), .invin(in1) );
  not U3 ( notinh2, inh2 );
  VrInh U4 ( .out(notout), .in(notinh2), .invin(inh1) );
  not U5 ( out, notout );
endmodule
```

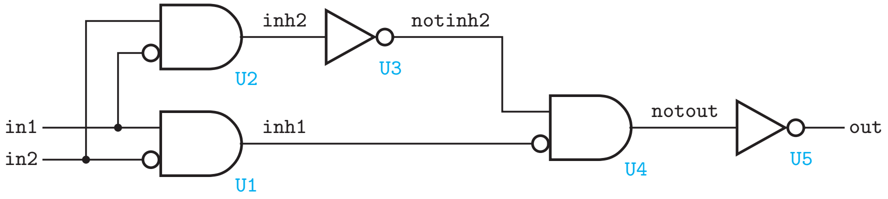


Figure 5-2 Logic diagram corresponding to the `VrSillyXOR` module.

SERIOUS SYNTHESIS A competent synthesizer can analyze the `VrSillyXOR` and `VrInh` modules together and reduce the circuit realization down to a single 2-input XOR gate, or equivalent realization in the target technology. Such a synthesizer typically also has an option to turn off global optimization and force synthesis of each module individually.

Remember that the instance statements in the example modules, Programs 5-2 through 5-4, execute *concurrently*. In each module, even if the statements were listed in a different order, the same circuit would be synthesized, and the simulated circuit operation would be the same.

parameterized module

majority function

Parameters, introduced at the end of Section 5.2, can be put to good use to *parameterize* structural modules that can handle inputs and outputs of any width. For example, consider a 3-input *majority function*, which produces a 1 output if at least two of its inputs are 1. That is, $\mathrm{OUT} = \mathrm{I0} \cdot \mathrm{I1} + \mathrm{I1} \cdot \mathrm{I2} + \mathrm{I0} \cdot \mathrm{I2}$. A module that performs a bitwise majority function on corresponding bits of input and output vectors of any width can be defined as shown in Program 5-5.

parameter substitution

#

When the `Maj` module is used with the instance-statement syntax in the previous examples, the parameter `WID` takes on its default value of 1, and the module works on 1-bit vectors (bits). However, instance-statement syntax has an option for overriding the instantiated module's parameter definitions. In the instance statement, the component name is followed by # and a parenthesized, comma-separated list of values that are substituted for the default parameter values given in the module definition. These values appear in the same order that the parameters are defined in the module. Thus, if `W`, `X`, `Y`, and `Z` are all 8-bit

Program 5-5 Parameterized Verilog module for a 3-input majority function.

```
module Maj(OUT, I0, I1, I2);
  parameter WID = 1;
  input [WID-1:0] I0, I1, I2;
  output [WID-1:0] OUT;

  assign OUT = I0 & I1 | I0 & I2 | I1 & I2 ;
endmodule
```

vectors, the following instance statement creates an 8-bit majority function for X, Y, and Z:

```
Maj #(8) U1 ( .OUT(W), .I0(X), .I1(Y), .I2(Z) );
```

The parameter substitution method above works alright when a module has just one parameter, but it's not so great for modules with more parameters. The need to list parameters in order obviously makes it error-prone. Worse, if you need to change only one of the parameters from its default value, you still have to provide values for all the ones before it in the list. Therefore, Verilog-2001 provides an ANSI-style mechanism to specify the values of any or all parameters, called *named parameter redefinition*. Instead of a list of parameter values, an instance statement may now contain a list that defines new values for one or more named parameters, in much the same style as the port-name/expression list later in the statement. With this method, the previous example becomes

named parameter redefinition

```
Maj #( .WID(8) ) U1 ( .OUT(W), .I0(X), .I1(Y), .I2(Z) );
```

Or, consider a more complex module Cmaj with the same inputs and output, but three parameters PARM1, PARM2, and PARM3. We could change the values of just two of them with an instance statement like the following:

```
Cmaj #( .PARM3(8), .PARM1(4) ) U2 ( .OUT(F), .I0(A), .I1(B), .I2(C) );
```

Any parameters that don't appear in the list retain their default values.

Note that parameters can be *defined* in modules that are coded in any style—included dataflow and procedural which will be discussed in the next two sections. But parameters can be *substituted* as shown above only when a module is instantiated structurally. Verilog provides another parameter definition and substitution mechanism, using the ***defparam*** keyword, but this mechanism is easily misused, leading to errors, and its use is not generally recommended.

defparam keyword

VERILOG'S GENERATE

In some applications, it is necessary to create multiple copies of a particular structure within a model, and Verilog-2001 addresses this need. Keywords generate and endgenerate begin and end a "generate block." Within a generate block, certain "behavioral" statements introduced later (if, case, and for) can be used to control whether or not instance and dataflow-style statements are executed. Instances can be generated in iterative loops (for), where the loop is controlled by a purpose-specific integer variable type (genvar).

The Verilog compiler takes care of generating unique component identifiers and, if necessary, net names for all instances and nets that are created within a for loop in a generate block, so they can be tracked during simulation and synthesis. We will give our first generate example in Program 7-5 on page 311, and more in Chapter 8 starting with Program 8-9 on page 393.

5.8 Dataflow Models

If Verilog had only instance statements, then it would be nothing more than a hierarchical net-list description language. “Continuous-assignment statements” allow Verilog to model a combinational circuit in terms of the flow of data and operations in the circuit. This style is called a *dataflow model* or *description*.

dataflow model
dataflow description
continuous-assignment statement
`assign` *keyword*

Dataflow models use *continuous-assignment statement*, with the basic syntax shown in the first line of Table 5-17. The keyword `assign` is followed by the name of a net, then an = sign, and finally an expression giving the value to be assigned. As shown in the remaining lines of the table, the lefthand side of the statement may also specify a bit or part of a net vector, or a concatenation using the standard concatenation syntax shown earlier. The syntax also has options that allow a drive strength and a delay value to be specified, but they aren’t often used in design for synthesis, and we don’t discuss or use them in this book.

A continuous-assignment statement evaluates the value of its righthand side and assigns it to the lefthand side, well, continuously. In simulation, the assignment occurs in zero simulated time, unless the delay option is used.

As with instance statements, the order of continuous assignment statements in a module doesn’t matter. If the last statement changes a net value used by the first statement, then the simulator will go back to that first statement and update its results according to the net that just changed, as discussed in more detail in Section 5.12. So, if a module contains two statements, “`assign X = Y`” and “`assign Y = ~X`”, then a simulation of it will loop “forever” (until the simulator times out). The corresponding synthesized circuit would be an inverter with its input connected to its output; if you actually built it, it would oscillate at a rate dependent on the signal propagation delay of the inverter.

Program 5-6 shows a Verilog module for a prime-number detector circuit (see Figure 3-24(c) on page 120) written in dataflow style. In this style, we don’t show the explicit gates and their connections; rather, we use Verilog’s bitwise-boolean operators to write the logic equation directly.

```
assign net-name = expression;
assign net-name[bit-index] = expression;
assign net-name[msb:lsb] = expression;
assign net-concatenation = expression;
```

Table 5-17 Syntax of continuous-assignment statements.

Program 5-6 Dataflow Verilog model for a prime-number detector.

```
module Vrprimed (N, F);
input [3:0] N;
output F;
  assign F = (~N[3] & N[0])  |  (~N[3] & ~N[2] &  N[1])
           |  (~N[2] &  N[1] & N[0])  |  (N[2] & ~N[1] & N[0]);
endmodule
```

WHERE DID THAT COME FROM?

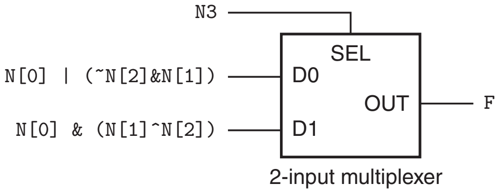
You may be wondering how I got the conditional expression in Program 5-7. It results from Shannon decomposition on variable N3. See Theorem T15 on page 97.

Program 5-7 Prime-number-detector code using a conditional operator.

```
module Vrprimec (N, F);
input [3:0] N;
output F;

assign F = N[3] ? (N[0] & (N[1]^N[2])) : (N[0] | (~N[2]&N[1]) ) ;
endmodule
```

Figure 5-3 Logic circuit corresponding to a conditional operator.



Verilog's continuous-assignment statement is unconditional, but different values can be assigned if the righthand side uses the conditional operator (?:). For example, Program 5-7 codes the same prime-number detection function using a completely different approach with a conditional operator. This operator corresponds very naturally to a 2-input multiplexer (introduced in Section 1.13 on page 29), a device that selects one of two possible data inputs based on the value of a select input. Thus, in an ASIC design, a synthesizer could realize the assignment in Program 5-7 using the circuit structure in Figure 5-3.

A dataflow-style example in which the conditional operator's use is more natural and intuitive is shown in Program 5-8. This module transfers one of three input bytes to its output depending on which of three corresponding select inputs is asserted. The order of the nested conditional operations determines which byte is transferred if multiple select inputs are asserted—input A has the highest priority, and C has the lowest. If no select input is asserted, the output is 0.

Program 5-8 Verilog module for selecting an input byte.

```
module Vrbytesel (A, B, C, selA, selB, selC, Z);
input [7:0] A, B, C;
input selA, selB, selC;
output [7:0] Z;

  assign Z = selA ? A : (
               selB ? B : (
                 selC ? C : 8'b0 )) ;
endmodule
```

5.9 Behavioral Models (Procedural Code)

As we saw in the last example, it may be possible to model a desired logic-circuit behavior directly using a continuous-assignment statement and a conditional operator. This is a good thing, because the ability to create a *behavioral model* or *description* is one of the key benefits of hardware description languages in general and of Verilog in particular. However, for most behavioral models, we need to use some additional language elements that allow us to write "procedural code," as described in this section.

behavioral model
behavioral description

5.9.1 Always Statements and Blocks

The key element of Verilog behavioral modeling is the `always` *statement*, with syntax options shown in Table 5-18. An `always` statement is followed by one or more "procedural statements," introduced shortly. The syntax in the table shows only one procedural statement. But as we'll show later, one type of procedural statement is a "`begin-end` block" that encloses a list of other procedural statements. That's what is used in all but the simplest `always` statements, and that's why we often call it an `always` *block*.

`always` statement
`always` keyword
`always` block

Procedural statements in an `always` block execute sequentially, as in software programming languages. However, the `always` block itself executes concurrently with other concurrent statements in the same module (instance, continuous-assignment, and `always`). Thus, if one of the procedural statements changes the value of a net or variable that is used in another concurrent statement, it may cause that statement to be re-executed (more on this to follow).

In the first three forms of an `always` statement, the `@` sign is followed by a parenthesized list of signal names, called a *sensitivity list*, which should specify all the signals whose values may affect results in the `always` block. The first form of `always` statement in Table 5-18 was the only one allowed in Verilog-1995, where individual signal names were separated by the keyword `or`. Given that this "`or`" has nothing to do with the logical operation, Verilog-2001 allows a comma to be used instead. In either case, a Verilog simulator reevaluates the procedural statement each time that any signal in the sensitivity list changes.

sensitivity list
`or` keyword

The third form of the sensitivity list (`*`), also introduced in Verilog-2001, is a shorthand for "every signal that might change a result," and puts the burden

sensitivity wildcard

```
always @ (signal-name or signal-name or ... or signal-name)
  procedural-statement
always @ (signal-name, signal-name, ... , signal-name)
  procedural-statement
always @ ( * ) procedural-statement

always @ (posedge signal-name) procedural-statement
always @ (negedge signal-name) procedural-statement

always procedural-statement
```

Table 5-18 Syntax of Verilog `always` blocks.

on the compiler to determine which signals should be in the list—basically all signals whose values may be read within the procedural statement. In this text, we often use this form (see the boxed comment below for why we do this).

The fourth and fifth forms of sensitivity list in Table 5-18 are used in sequential circuits, and will be discussed in Section 5.14. The last form of an `always` statement does not have a sensitivity list. Such an `always` statement starts running at time zero in simulation and keeps looping forever. This is not a good thing in synthesis, but can be very useful in a test bench. By coding explicit delays within the `always` statement, you can generate a repetitive waveform like a clock signal. See, for example, Program 12-6 on page 619.

An instance or continuous-assignment statement also has a sensitivity list, an implicit one. All of the input signals in an instantiated component or module are on the instance statement's implicit sensitivity list. Likewise, all of the signals on the righthand side of a continuous-assignment statement are on its implicit sensitivity list.

executing statement
suspended statement

In simulation, a Verilog concurrent statement such as an `always` block is always either *executing* or *suspended*. A concurrent statement initially is suspended; when any signal in its sensitivity list changes value, it resumes execution. A resumed `always` block starts with its first procedural statement and continues executing them sequentially all the way until its end. If any signal in the sensitivity list changes value as a result of executing the concurrent statement, it executes again. This continues until the statement executes without any

YOU SHOULD BE MORE SENSITIVE (*)

A Verilog simulator executes the procedural statements within an `always` block only when one or more of the signals in its sensitivity list changes. It is very easy to inadvertently write a "combinational" `always` block with an incomplete sensitivity list—one in which not all of the signals that affect the outcomes of the procedural statements are listed. Typically, you might forget to include one or more signals that appear on the righthand side of an assignment statement, especially after revising the model and adding new signals to it.

Faced with such an error, the simulator still follows the definition and does not execute the `always` block until one of the *listed* signals changes. Thus, the block's behavior will be partially sequential, rather than combinational as intended. A typical synthesizer, however, does not attempt to create logic with this weird behavior. Instead, it ignores your error and synthesizes your intended combinational logic.

No problem then, right? Wrong. Now the behaviors of the simulator and the synthesized logic don't match. The "incorrect" simulated behavior may mask other errors and give you the intended and expected results at the system level, while the synthesized circuit may not work properly in all cases.

One solution to this problem is always to pay close attention to warning messages from the synthesizer—most will flag this condition. A better solution is to use the wildcard "`*`" as the sensitivity list.

of these signals changing value at the current time. In simulation, all of this happens in zero simulated time.

Upon resumption, a properly written concurrent statement will suspend after one or a few executions. However, it is possible to write a statement that never suspends. For example, consider the instance statement "`not(X,~X)`". Since `X` changes on every execution pass, the statement will execute forever in zero simulated time—not very useful! In practice, simulators have safeguards that normally can detect such unwanted behavior, terminating the misbehaving statement after a thousand or so passes.

5.9.2 Procedural Statements

procedural statement

Verilog has several different *procedural statements* that are used within an `always` block. They are assignment, `begin-end` blocks, `if`, `case`, `while`, and `repeat`; we'll describe them soon. There are a few other seldom-used types; they are not synthesizable and we don't cover them in this book.

Procedural statements are written in a style similar to software programming languages like C. Every value assigned to a variable is preserved until it is changed in the current or in a subsequent execution of an `always` block. This behavior is natural in software programming languages, but with Verilog models you can get unwanted behavior in both simulation and synthesis if you stray from recommended coding guidelines, as discussed in the next subsection.

5.9.3 Inferred Latches

Consider an `always` block which is intended to create combinational logic and assigns a value to a variable `X`. As we'll soon see, besides unconditional assignments, Verilog has conditional procedural statements like `if` and `case` that can control whether other statements, including assignments, are executed. So, `X` might appear on the lefthand side of several different assignments, and in a given pass through the `always` block, zero, one, or more of them might actually be executed, depending on current condition values.

There's no problem if one or even if multiple values are assigned to `X` during a given pass through the `always` block. Since the block executes in zero simulated time, and procedural statements are executed sequentially, the last value assigned dominates. But suppose that *no* value is assigned to `X`. Since this is procedural code, the simulator "infers" that you don't want this pass to change the value of `X` from what it had during the previous pass. And so the synthesizer *infers a latch*—it creates a storage element to retain the previous value of `X` if conditions are such that no new value is assigned. This is rarely the designer's intent when modeling combinational logic.

latch inference

The solution to this problem is to ensure that a value is assigned to `X` (and to every other variable on the lefthand side of an assignment statement) in every possible execution path through an `always` block. Although nothing is foolproof, we'll show a pretty reliable way to do this in the box on page 216.

5.9.4 Assignment Statements

blocking assignment statement, =

nonblocking assignment statement, <=

The first two procedural statements we need are *blocking* and *nonblocking assignment*, with the syntax shown in Table 5-19. The lefthand side of either procedural assignment statement must be a variable, but the righthand side can be any expression that produces a compatible value and can include both nets and variables.

A blocking assignment looks and acts like an assignment statement in any other procedural language, like C. A nonblocking assignment looks and acts a little different—it evaluates its righthand side immediately, but it does not assign the resulting value to the lefthand side until an infinitesimal delay *after* the entire `always` block has been executed. Thus, the "old" value of the lefthand side continues to be available for the rest of the `always` block. You can read a nonblocking assignment as "*variable-name* eventually gets *expression*."

If you think too hard about the subtle differences between the two types of assignment statements, your head may hurt or you may get confused. But fortunately, if you follow a basic, consistent coding style for synthesis as practiced in this book, it's easy to know which one to use—just follow the simple rules at the bottom of the next page. Still, we'll give some examples later in this section and in Section 10.3 that shed more light on the reasons for the rules.

WHY "BLOCKING"?

Blocking assignments get their name because they *block* the execution of subsequent procedural statements in the same `always` block until the assignment has actually been made. Well, duh, that's what you'd expect in any procedural programming language like C or Java, right?

What you don't know yet is that Verilog also allows a procedural assignment statement to specify a delay. Although such delays can be used in modeling actual hardware delays, they are not synthesizable, and we don't describe or use them in this book. But if you did specify such a delay, it would block the execution of the rest of the `always` block until the delay had passed.

AND WHY "NONBLOCKING"?

Nonblocking assignments, whether a delay is specified or not, allow execution of the `always` block to continue. But they're still not the same as assignments in typical procedural programming languages like C or Java.

As noted in the main text above, a nonblocking-assignment statement evaluates its righthand side immediately, but even if no delay is specified (typical in design-for-synthesis) it does not assign this value to the lefthand side until an infinitesimal delay after the entire `always` block has completed execution.

Table 5-19 Procedural assignment statements.

```
variable-name = expression ;   // blocking assignment

variable-name <= expression ;  // nonblocking assignment
```

For instructional purposes, we've rewritten the dataflow Verilog module for the prime-number detector in Program 5-6 on page 203 using an `always` statement, in Program 5-9. There are a couple of things to notice about this code:

- The output signal `F` must be declared as a `reg` variable, since it appears on the lefthand side of an assignment statement in an `always` block.
- The assignment statement is a blocking one, as recommended by our coding guidelines.

If we want an `always` statement to perform two or more assignments or other operations when it executes, we need "`begin-end`" blocks, described next.

Program 5-9 Prime-number detector using an `always` block.

```
module Vrprimea (N, F);
input [3:0] N;
output reg F;

  always @ (*)
      F = ~N[3] & N[0] | ~N[3] & ~N[2] &  N[1]
        | ~N[2] &  N[1] & N[0] | N[2] & ~N[1] & N[0] ;
endmodule
```

LEARN THE RULES, YOU BLOCKHEAD!

The two rules below are so important, it's the only place that you'll find boldface roman font used in this book:

- Always use **blocking** assignments (=) in `always` blocks intended to create **combinational** logic.
- Always use **nonblocking** assignments (<=) in `always` blocks intended to create **sequential** logic. (See Section 10.3.2.)
- Do not mix blocking and nonblocking assignments in the same `always` block.
- Do not make assignments to the same variable in two different `always` blocks.

Once you've learned these rules, the only thing left is for you to remember is which assignment operator symbol is which. But that's easy, too. The < character in the nonblocking assignment operator is a mirror image of the dynamic-input indicator (>) used in an edge-triggered flip-flop's logic symbol, as you'll learn later. So, be sure to use it in `always` blocks that are intended to create sequential logic.

5.9.5 begin-end Blocks

begin-end block
begin keyword
end keyword

The first part of Table 5-20 shows the basic syntax of a ***begin-end** block*, simply a list of one or more procedural statements enclosed by the keywords ***begin*** and ***end***. As shown in the second part of the table, a `begin-end` block can have its own local parameters or variables (typically `integer` or `reg`). In this case, the block must be named so that these items can be tracked during simulation and synthesis, using the name. Also, a `begin-end` block can be named even if it has no local parameters or variables.

Note that the procedural statements within a `begin-end` block execute sequentially, not concurrently like the instance, continuous-assignment, and other `always` statements at the top level of a module. Of course, sequential execution is what you'd expect in procedural code.

A behavioral model of an alarm circuit using an `always` block is shown in Program 5-10. The model uses the equations we originally showed on page 112, including the definition of an intermediate signal `secure`. A few aspects of this model are noteworthy:

- The intermediate signal is declared as `reg` variable local to the `begin-end` block.
- Since a local variable is declared, the block must be named.

Table 5-20 Syntax of Verilog `begin-end` blocks.

```
begin
  procedural-statement
  . . .
  procedural-statement
end

begin : block-name
  variable declarations
  parameter declarations

  procedural-statement
  . . .
  procedural-statement
end
```

WHEN TO USE A SEMICOLON

You might think of a `begin-end` block as being a list of procedural statements separated by semicolons, but that's not quite right; the syntax is just as we show it above. A semicolon is already included in an assignment statement as defined in Table 5-19. And the "`end`" in a `begin-end` block has the semicolon "built-in." The same is true of the "`endcase`" in a `case` statement, introduced later.

Still, Verilog defines a semicolon all by itself to be a null statement, so it usually doesn't hurt to put in extras.

Program 5-10 Alarm-circuit module using procedural assignments in an `always` block.

```
module VrAlarmCktb (
  input panic, enable, exiting, window, door, garage,
  output reg alarm
);
  always @ (panic, enable, exiting, window, door, garage)
    begin : Ablk
    reg secure;
      secure = window & door & garage;
      alarm = panic | ( enable & ~exiting & ~(window & door & garage) );
    end
endmodule
```

- The output `alarm` must be declared as a `reg` variable, since a value is assigned to it by procedural code.
- Instead of putting "`*`" in the sensitivity list, we've listed all the inputs, just to emphasize the fact that local `reg` variables may *not* be included. If you did include `secure`, you'd actually get an error message, since it's undefined outside the scope of the `begin-end` block.
- In this simple example, there's no particular reason for `secure` to be defined local to the `always` block rather than at the top level of the module; either way works. In a larger module, you might prefer it to be local so it won't be used inadvertently by other concurrent statements.

5.9.6 if and if-else Statements

Other procedural statements, beyond simple assignment and `begin-end` blocks, give designers more powerful ways to model circuit behavior. An ***`if`*** *statement*, with the syntax shown in Table 5-21, is probably the most familiar of these. In the first and simplest form of the statement, a *condition* (a logical expression) is tested, and a procedural statement is executed if the condition is true (that is, if it evaluates to `1'b1`).

`if` statement
`if` keyword
condition

In the `if-else` form, we've added an "***`else`***" clause with another procedural statement that's executed if the condition evaluates to anything but true (including "x", which may occur in a test bench). Note that although an `if-else` statement may contain two semicolons—terminating its two procedural statements—it is still just *one* statement. So `if-else` can be used anywhere that a single statement can be used, for example, as the procedural statement for an `always` statement or as the "`else`" clause of another `if-else` statement.

`if-else` statement
`else` keyword

```
if ( condition ) procedural-statement

if ( condition ) procedural-statement
else procedural-statement
```

Table 5-21 Syntax of Verilog `if` statements.

As in other languages, if and if-else statements can be nested, since any of the "procedural-statements" in Table 5-21 can be if statements. Also as in other languages, an if statement that is nested immediately after the condition in an if-else should be enclosed in a begin-end block to eliminate any ambiguity about which "if" the "else" goes with. Even if your mind works as perfectly as a Verilog parser, someone else who reads your code might make an incorrect association.

Program 5-11 is a version of the prime-number-detector module that uses a nested if statement. It defines a parameter that you can change depending on whether or not you believe that 1 is prime. The first if clause handles the special case, and the second one separates the remaining cases into even and odd. Notice the use of begin-end around the if statement that is nested immediately after the condition expression in its parent.

Also notice in Program 5-11 that a value is assigned to F in every possible execution path through the always block. Suppose that we inadvertently left out the first "else F = 0" clause. Then, for the reasons discussed on page 207, the synthesizer would infer a latch to hold the previous value of F whenever N is even but not 2. One way to avoid latch inference is to ensure that every if statement has an else clause, and that every variable that is assigned a value in one if or else clause is also assigned a value in every other clause. Another method is discussed in the box on page 216.

Program 5-11 Prime-detector module using an if statement.

```
module Vrprimei (N, F);
input [3:0] N;
output reg F;
parameter OneIsPrime = 1; // Change this to 0 if you
                          // don't consider 1 to be prime.
  always @ (*)
    if (N == 1) F = OneIsPrime;
    else if ( (N % 2) == 0 )
      begin if (N == 2) F = 1; else F = 0; end
    else if (N <= 7) F = 1;
    else if ( (N==11) || (N==13) ) F = 1;
    else F = 0;
endmodule
```

PRIME TIME AGAIN

When introducing the prime-detector example in Chapter 3, I explained that mathematicians don't consider "1" to be a prime number. So, to accommodate them, I included the parameter in Program 5-11. It also makes the example more interesting.

EXPENSIVE COMPARATORS?

When compiled, the Verilog code in Program 5-11 can lead to the creation of up to five RTL 4-bit comparators corresponding to conditional expressions. For example, with Xilinx Vivado tools, the logic diagram for the elaborated RTL has two equality comparators (for "`N==11`" and "`N==13`") and one magnitude comparator (for "`N<=7`"). Should we worry about synthesizing "expensive comparators"?

No. One operand in each comparison is a constant, and the other operand is the same in all of them. Even though the RTL shows three comparators plus other operations, the synthesis tool converts the comparators into equivalent boolean equations and combines them, ultimately creating a 4-input combinational function to map into the target technology. This example is small enough that all versions of the prime-number-detector function in this section should yield the same synthesized result, regardless of the RTL starting point.

5.9.7 case Statements

When two or more variables must be tested to determine different outcomes, a nested series of `if` statements is usually the right coding approach. However, if all of the `if` statements would be testing the same variable, as in Program 5-11, it is often more clear to use a `case` statement, described next.

`case` statement

`case` keyword

Table 5-22 shows the syntax of a Verilog *`case` statement*. It begins with the keyword *`case`* and a parenthesized "selection expression," usually one that evaluates to a bit-vector value of a certain width. Next is a series of case items, each of which has a comma-separated list of "choices" and a procedural statement. (If there is only one choice in a particular case item, then the comma is omitted.) A single "default" case item may be included as discussed shortly. The statement ends with the *`endcase`* keyword.

`endcase` keyword

The operation of `case` statement is simple—it evaluates the selection expression, finds the first one of the choices that matches the expression's value, and executes the corresponding procedural statement. Again, the `case` statement executes just *one* procedural statement, corresponding to the first match.

Choices in a `case` statement are typically just constant values compatible with the selection expression, but they can also be more complex expressions. This leads to the possibility that some of the choices may overlap; that is, some values of the selection expression may match multiple choices, and again, only

```
case ( selection-expression )
  choice , ... , choice : procedural-statement
  ...
  choice , ... , choice : procedural-statement
  default : procedural-statement
endcase
```

Table 5-22 Syntax of a Verilog `case` statement.

Program 5-12 Prime-detector module using a `case` statement.

```
module Vrprimecs (N, F);
input [3:0] N;
output reg F;

  always @ (*)
    case (N)
      4'd1, 4'd2, 4'd3, 4'd5, 4'd7, 4'd11, 4'd13 : F = 1;
      default : F = 0;
    endcase
endmodule
```

parallel `case`

the *first* matching choice is executed. When the choices do not overlap, they are said to be "mutually exclusive." This is called a *parallel* `case`. The best Verilog coding practices avoid nonparallel `case` statements.

`default` keyword

Quite often, the listed choices in a `case` statement are not "all-inclusive." That is, they may not include all possible values of the selection expression. The keyword ***`default`*** can be used in the last case item to denote all selection values that have not yet been covered. (Syntactically, the colon after "`default`" is optional.) Even if you're sure that your listed choices are all-inclusive, it's a good practice to include a `default` choice in your `case` statements.

full `case`

A `case` statement in which the listed choices are all-inclusive is called a *full* `case`. In a nonfull `case`, the synthesizer infers latches to retain the previous values of outputs for any cases that are not covered. This is usually not desired, so the best Verilog coding practices use full `case` statements only.

Program 5-12 is yet another version of the prime-number detector, this time coded with a `case` statement. In this very simple example, the `case` statement has, in effect, written out the truth table for the output function `F`.

A slightly more complex use of `case` is shown in Program 5-13. This module transfers one of three 8-bit inputs to its output depending on the value of a 2-bit select code, `sel`. If `sel` is 3, the 8-bit output is set to 0. A couple of aspects of this code are worth noting:

NONPARALLEL `case` STATEMENTS

When the choices in a `case` statement are not mutually exclusive (nonparallel `case`), only the first matching choice has its corresponding procedural statement executed. To ensure this, a synthesizer must infer expensive "priority-encoder" logic to guarantee proper operation.

However, if the synthesizer can determine that the choices are mutually exclusive, it can use faster and less expensive multiplexer logic. So, you should generally avoid writing nonparallel `case` statements. If you need a priority encoder, you should write one explicitly, for example using nested `if` statements or as shown in Section 7.2.2.

Program 5-13 Bus-selector module using a `case` statement.

```
module Vrbytecase (A, B, C, sel, Z);
input [7:0] A, B, C;
input [1:0] sel;
output reg [7:0] Z;

  always @ (*)
    case (sel)
      2'd0 : Z = A;
      2'd1 : Z = B;
      2'd2 : Z = C;
      2'd3 : Z = 8'b0;
      default : Z = 8'bx;
    endcase
endmodule
```

- A `default` choice is coded, even though the choices that precede it are all-inclusive. This is a good coding practice, especially for simulation. It ensures that if `sel` contains any `x` or `z` bits, then `x`'s will be propagated to the output. You can also use a Verilog `$display` command here (discussed in Section 5.10) to flag this case in simulation if you wish; the `$display` command is ignored in synthesis.
- The choices are coded as 2-bit-wide vectors. With most Verilog compilers, we could have gotten away simply with "0, 1, 2, 3," but see the box below.

Verilog has two other `case` statements, identical in syntax to the first, but introduced by the keywords ***`casex`*** and ***`casez`***. The ***`casez`** statement* allows `z` or `?` to be used in one or more bit positions in a binary choice constant in, for example, `4'b10??`. These characters are interpreted as "don't-cares" when the choice constant is matched with the selection expression. Both characters mean the same thing, but `?` is preferred; it won't be confused with the high-impedance

`casex` keyword
`casez` keyword
`?` in choice

MIXING INTEGERS AND VECTORS IN `case` CHOICES

The values of the selection expression and the choices in a `case` statement should normally be vectors of the same width. If the widths don't match, the narrower ones are extended on the left with 0s.

If the choices are integers, they will be converted to a vector with a compiler-dependent width, typically 32 bits. If the selection expression is a 4-bit vector, as in Program 5-12, you would expect the Verilog compiler to figure out that you're only interested in the low-order four bits of the integers, and most will. However, in more complex situations where the vector widths don't match, some compilers may produce unexpected results. Therefore, their suppliers recommend that integer choices be written with explicit widths as in Program 5-12.

AVOIDING INFERRED LATCHES

By now you know that to avoid inferring unwanted latches, you must assign a value to a variable in every possible execution path through an `always` block. The easiest way to do this is to unconditionally assign default values to variables at the *beginning* of the `always` block. This approach works with the `if` and `case` statements that we've covered so far, as well as with the looping statements that are coming next.

In some situations, the appropriate default assignment will be to a value of "`x`" (unknown). This is good if you intend your subsequent code to cover all cases, but you'd like to catch inadvertent omissions in simulation. In other situations, you'd like the default to assign the most commonly needed result, so an assignment need not be repeated in all the subsequent cases. Program 5-14 shows an example of each.

You may say, "But, the signals `F` and `special` now may have values assigned to them twice. Can't this cause glitches in the realized circuit?" No. These statements execute in zero simulated time, and the last assignment in an `always` block prevails. And the synthesizer uses the last assigned value, too.

Program 5-14 Prime-detector module with default assignments.

```
module Vrprimef (N, F, ignore);  // Special prime detector
input [3:0] N;                   // for mathematicians, tells
output reg F, ignore;            // them when to ignore F.

  always @ (*) begin
    F = 1'bx; ignore = 1'b0; // defaults
    if (N == 1) begin F = 1; ignore = 1; end
    else if ( (N % 2) == 0 )
      begin if (N == 2) F = 1; else F = 0; end
    else if (N <= 7) F = 1;
    else if ( (N==11) || (N==13) ) F = 1;
    else F = 0;
  end
endmodule
```

`casex` statement

state. The *`casex` statement* allows `x` also to be used as a "don't-care," but it is not recommended, since in simulation it can hide the existence of unknown (`x`) values generated upstream. Even `casez` is tricky to use, and should be avoided when possible; but we will give an example at the end of Section 7.2.2.

5.9.8 Looping Statements

looping statement
`for` statement
`for` loop
`for` keyword

Another important class of procedural statements are *looping statements*. The most commonly used of these is the *`for` statement* or *`for` loop*, with the syntax in Table 5-23. Here the *loop-index* is a register variable, typically an integer or a bit vector that's being used like one, and *first-expr* is an expression giving a value that is assigned to *loop-index* when the `for` loop begins execution.

```
for ( loop-index = first-expr ; logical-expression ; loop-index = next-expr )
  procedural-statement

for ( loop-index = first; loop-index <= last; loop-index = loop-index + 1; )
  procedural-statement
```

Table 5-23 Syntax of a Verilog `for` statement.

After initializing the *loop-index*, a `for` loop executes *procedural-statement* for a certain number of iterations. At the beginning of each iteration, it evaluates *logical-expression*. If the value is false, the `for` loop stops execution. If the value is true, it executes *procedural-statement*, and at the end of the iteration it assigns *next-expr* to *loop-index*. Iterations continue until *logical-expression* is false.

For synthesis, the kinds of expressions that can be used for iteration control are limited. Typically, *first-expr* must be a constant expression, it must be possible to determine the value of *logical-expression* at compile time, and *next-expr* may be limited to simple incrementing and decrementing. Thus, the last two lines of Table 5-23 show a typical syntax of a `for` loop as it is used for synthesis. The single procedural statement in a `for` loop is often a `begin-end` block, so that a series of other procedural statements may be executed in each iteration.

A simple example using a `for` loop is shown in Program 5-15. The module compares two 8-bit inputs `X` and `Y` and asserts its `gt` output if `X` is greater than `Y`. Rather than use Verilog's built-in operator, it illustrates a `for` loop by doing the comparison bit by bit, starting with the LSB. Prior to the loop, `gt` is initialized to 0. In the loop, if (`X[i]`,`Y[i]`) is (1,0), then `X>Y` so far. If it's (0,1), then `X<Y` so far. If `X` and `Y` are equal, `gt` keeps whatever value it had at the previous iteration.

The `for` loop looks very "sequential," but keep in mind that it's modeling combinational logic. In simulation, the entire loop executes in zero simulated time. And the intermediate values that `gt` has during the loop's execution don't appear on the synthesized circuit's output. The value that `gt` acquires in response to an input combination is merely being *specified* sequentially.

Program 5-15 An 8-bit comparator module using a `for` loop.

```
module Vrcomp (X, Y, gt);
input [7:0] X, Y;
output reg gt;     // Will be 1 if X > Y
integer i;

  always @ (X, Y) begin
    gt = 0; // starts out as 'not greater'
    for ( i=0 ; i<=7 ; i=i+1 )
      if (X[i] & ~Y[i]) gt = 1;
      else if (~X[i] & Y[i]) gt = 0;
      // otherwise, X[i]==Y[i], and there's no change to gt
  end
endmodule
```

AYE, AYE, SIR! Some designers like to use variable names like `ii`, `jj`, and `kk` for loop control, rather than the more traditional `i`, `j`, and `k`. Among other things, the former are easier find in text searches during code rewrites and debugging.

Program 5-16 An 8-bit comparator module using `for` and `disable`.

```
module Vrcompdis (X, Y, gt);
input [7:0] X, Y;
output reg gt;     // Should be 1 if X > Y
integer i;

  always @ (*) begin : COMP
    gt = 0; // default is 'not greater'
    for ( i=7 ; i>=0 ; i=i-1 )
      if ( X[i] & ~Y[i] )
        begin gt = 1; disable COMP; end
      else if ( ~X[i] & Y[i] )
        begin gt = 0; disable COMP; end
  end
endmodule
```

`disable` statement
`disable` keyword

Another statement is sometimes seen in conjunction with Verilog looping statements, but it is not synthesizable by all tools and should be avoided. The ***`disable`*** *statement* can be used anywhere within a named `begin`-`end` block. It consists of the ***`disable`*** keyword, followed by the name of the block, followed by a semicolon. When executed, it immediately terminates execution of the block; subsequent statements in the block are not executed. For example, Program 5-16 is a version of the 8-bit comparator that checks `X` and `Y` bit-by-bit starting with the MSB. It executes a `disable` statement at the first iteration where `X[i]` and `Y[i]` are different, since the result is then certain.

COMPARING COMPARATORS As we've implied, the purpose of our comparator examples was to illustrate the use of `for` statements, not to build the world's best comparators. Verilog has built-in comparison operations, and the simplest equivalent of the `Vrcomp` model would have a single dataflow statement, "`assign gt = (X>Y);`".

Most tools will do a better job synthesizing circuits to perform commonly used operations, too. For example, when Program 5-15 or 5-16 is targeted to a Xilinx 7-series FPGA using Vivado tools, it uses four LUTs in four levels of logic and has a maximum internal delay of 2.505 ns. When synthesizing a built-in comparison as in the assignment statement above, Vivado knows how to use specialized FPGA resources that optimize comparators and adders. The result still uses four LUTs, but now in only two levels of logic and has a maximum delay of 1.526 ns.

IN ITS PRIME

Our good old prime-number detector is coded one more time in Program 5-17, this time using a `for` loop. This is truly a behavioral model—we have actually modeled combinational hardware that divides `N` by all odd numbers that are less than its square root. We've also increased the width of `N` to 16 bits, just for fun.

A bad thing about this design is that it may not be synthesizable. The `for` loop is fine, but as mentioned previously, the modulo operation (`%`) may be synthesizable only if its divisor is a power of two, corresponding to a right shift. For other divisors, combinational divider circuits are needed and not all synthesizers can create them. To "unroll" the `for` loop, the synthesizer would have to create over 100 such combinational dividers. Xilinx Vivado tools can actually do it, though, using 8,362 LUTs in 23 levels of logic with a maximum delay of about 50 ns. "Don't try this at home."

Program 5-17 Prime-number detector using a `for` statement.

```
module Vrprimebv (N, F);
input [15:0] N;
output reg F;
reg prime;
integer i;
  always @ (*) begin
    prime = 1;   // initial values
    if ( (N==1) || (N==2) ) prime = 1; // Special cases
    else if ((N % 2) ==0) prime = 0;   // Even, not prime
    else for ( i = 3 ; i <= 255 ; i = i+2 )
      if ( ((N % i) == 0) && (N != i) )
        prime = 0;   // Set to 0 if N is divisible by any i
    if (prime==1) F = 1; else F = 0;
  end
endmodule
```

The other Verilog looping statements are `repeat`, `while`, and `forever`, with syntax shown in Table 5-24; each controls a single procedural statement. A ***`repeat`* statement** repeats it a number of times given by *integer-expression*. A ***`while`* statement** repeats it until *logical-expression* is false. And a ***`forever`* statement** repeats it "forever." As always, the procedural statement may be a `begin`-`end` block containing a series of other procedural statements.

`repeat` statement
`while` statement
`forever` statement

`repeat` (*integer-expression*)
  procedural-statement

`while` (*logical-expression*)
  procedural-statement

`forever`
  procedural-statement

Table 5-24 Syntax of Verilog `repeat`, `while`, and `forever` statements.

WHERE THE WHILE THINGS ARE

The `repeat`, `while`, and `forever` statements cannot be used to synthesize combinational logic, only sequential logic, and then only if the procedural statement is a `begin`-`end` block that includes timing control that waits for a signal edge. This is not covered here, since the most predominant Verilog coding practices consistently use other means (discussed in Sections 10.3.2 and 10.8.4) to create sequential-circuit behavior. These statements can also be used in advanced test-bench code, but that's not covered in this text.

5.10 Functions and Tasks

function

Like a function in a high-level programming language, a Verilog *function* accepts a number of inputs and returns a single result. The inputs may be bits or bit vectors, and they may be any variable type, including `integer` and `reg` variables, and a few other types that we don't cover in this book.

function definition
`function` keyword

The syntax of a Verilog *function definition* is shown in Table 5-25. It begins with the keyword ***`function`***, followed by an optional specification of the result type—`integer`, a bit-vector `[`*msb*`:`*lsb*`]`, or blank for a single-bit result, the default. Next are the function name and a semicolon.

The inputs of a function are listed in order in input declarations. These are declared using the `input` keyword, similar to input declarations in a module declaration, and are single bits or bit vectors. A function may not have any `output` or `inout` declarations. However, as shown in the table, a function may declare its own local variables and parameters. But it may not declare any nets or nested functions and tasks.

The "executable" part of a function is a single procedural statement. Usually this is a `begin`-`end` block containing a series of procedural statements. The function name is implicitly defined to be a local `reg` variable of the declared result type, and somewhere in the function, a value must be assigned to this variable. This value is returned to the function's caller. The function definition ends with the `endfunction` keyword.

`endfunction` keyword

As implied by the format of a module definition, Table 5-1 on page 183, a function can be defined only within a module. That is, its definition is local to the module. If you have a commonly needed function to use in multiple modules,

Table 5-25 Syntax of a Verilog function definition.

```
function result-type function-name ;
  input declarations
  variable declarations
  parameter declarations

  procedural-statement
endfunction
```

MULTIPLE-OUTPUT FUNCTIONS A function can have only one output. But a simple trick lets you to create a function with multiple outputs—you just concatenate the needed outputs before assigning to the function name, and then use part-selects in the caller to pull out the individual values. If you use this trick, you must be extremely careful—the size and order of the concatenated signals in the function and in the caller must match perfectly.

you could define it by itself in a file, and then use an `` `include `` compiler directive to include it in each module where it's needed.

function call

A function is invoked or *called* by writing the function name followed by a parenthesized list of expressions. The expressions are evaluated and assigned to the function's inputs in the order that they appear in the function definition. A function name can be used in an expression, and thus the function can be called, anywhere that a signal of the same type could be used—in `always` blocks, in continuous assignments, and in other functions in the same module.

A function executes in zero simulated time, and therefore cannot contain any delay or other timing-related statements. Also, the values of any local variables are lost from one function call to the next. So, functions are primarily a mechanism to reduce typing and thinking, to minimize inconsistency, and to improve the readability, modularity, and maintainability of Verilog code.

As an example, the module in Program 5-18 is a behavioral version of the `SillyXOR` module (see Program 5-4 on page 200). It defines the function `Inhibit` that acts like a 2-input inhibit gate, and it calls `Inhibit` three times within an `always` block to perform the module's specified function (a rather roundabout XOR operation). The names of the local variables and the structure of the function calls match the logic diagram in Figure 5-2 on page 201 exactly.

Program 5-18 Verilog model for an XOR gate using an "inhibit" function.

```
module VrSillierXOR(in1, in2, out);
  input in1, in2;
  output reg out;

  function Inhibit ;
    input In, invIn;
    Inhibit = In & ~invIn;
  endfunction

always @ (*) begin : IB
    reg inh1, inh2;
    inh1 = Inhibit(in1,in2);
    inh2 = Inhibit(in2,in1);
    out = ~Inhibit(~inh2,inh1);
  end
endmodule
```

RECURSIVE FUNCTION CALLS

Theoretically, you could replace the `always` block in Program 5-18 with a single continuous-assignment statement:

```
assign out = ~Inhibit(~Inhibit(in2,in1), Inhibit(in1,in2));
```

However, most Verilog tools do not support recursive function calls; that is, they do not let a function call itself. In practice, the utility of this sort of program construction is quite limited anyway, and should be avoided even if the tools do support it.

task
task definition
`task` *keyword*

A Verilog *task* is similar to a function, except it does not return a result. Table 5-26 shows the syntax of a *task definition*. It begins with the keyword `task`, followed by the task name. Unlike functions, tasks can also have inout and output arguments, which are declared in the same way as input arguments, but using the keywords `inout` and `output`. Like a function, a task contains a single procedural statement, which is typically a `begin-end` block. A task ends with the `endtask` keyword.

`endtask` *keyword*
task call
task enable

While a function call can be used in the place of an expression, a *task call* (sometimes called a *task enable*) can be used in the place of a statement. Like a function, a task is called using its name and a parenthesized list of expressions. These expressions are associated in the order written with the input, inout, and output declarations in the task definition. Note that a task need not have any inputs and outputs declared, so the parenthesized list may be missing or empty. When present, expressions corresponding to inputs are evaluated when the task is called, and their values are assigned to the corresponding input arguments of the task. When task execution completes, its local inout and output variables are copied to the corresponding "expressions" in the calling code, which must be individual signal names or concatenations.

While they are useful in test benches, tasks are not usually recommended in synthesizable Verilog modules. Although delays can be specified within tasks, they are not synthesizable; a task is synthesized as combinational logic. Some Verilog synthesizers can't handle tasks at all. When the synthesizer does support them, user-defined tasks can be useful in structuring larger module designs, but we won't discuss that application further in this book.

Table 5-26 Syntax of a Verilog task definition.

```
task task-name ;
  input declarations
  inout declarations
  output declarations
  variable declarations
  parameter declarations

  procedural-statement
endtask
```

Verilog has many built-in system tasks and functions that are used in test benches and simulation, including the following:

- `$display`. This task is used to print formatted signal values and text to the "standard output," the system console in simple simulation environments. The arguments to this task are a formatting string, similar to what's used in C's `printf` function, and a list of signals to be printed. This task and other tasks can be called anywhere in a module, and it immediately prints the list of signals in the specified format, followed by a newline character.
- `$write`. This task is the same as `$display`, except that it does not automatically append a newline character at the end.
- `$monitor`. This task is similar to `$display`, except that it remains active continuously, and prints the listed signals whenever any one of them changes. Although multiple `$monitor` calls may be made within a simulation, only one can be active at a time; calling `$monitor` cancels the monitoring specified by any previous call.
- `$monitoroff` and `$monitoron`. These tasks turn off and on the monitoring specified by the most recent `$monitor` call.
- `$fflush`. This task flushes any pending file output, including anything sent to the "standard output." This is worth including at the end of a test bench in some environments, to ensure that you always see the last of the results before the simulation process terminates and the OS ruthlessly throws away any still pending output.
- `$time`. This function has no arguments, and simply returns the value of the current simulated time.
- `$random`. This function returns a pseudorandom 32-bit signed integer to the caller, a different one on each subsequent call, useful for generating "random" inputs in test benches. The simulator's pseudo-random number generator algorithm is fully specified in Verilog-2001, so any simulator will return the same sequence of results. To make different starting points possible, the function has one optional argument, a 32-bit signed integer `seed`, which sets the starting value to be used to get the next pseudorandom result. So, you can get a different pseudorandom sequence by including a value for `seed` on the first call of `$random`.
- `$stop`. This task suspends simulation and returns control to the user. If called with the optional argument "`(1)`", it displays the simulated time and the location in the code.

We'll see example uses of some of these tasks and functions in test benches in Section 5.13 and subsequent chapters. For more details on these functions, including formatting-string options for `$display` and `$write`, consult a Verilog reference manual. There, you can also find information on many other built-in tasks and functions. These include file input/output tasks that are very useful in test benches for larger real-world designs, allowing expected inputs to be read from a file, and output results to be written to another file. This allows great flexibility in creating test-bench inputs and analyzing results, since you can use any convenient programming language to do it.

5.11 The Time Dimension

None of the examples we've shown so far model the time dimension of circuit operation—everything happens in zero simulated time. However, Verilog has very good facilities for modeling time, and it is indeed another significant dimension of the language. In this book, we won't go into great detail on this subject, but we'll introduce just a few ideas here.

# (delay specifier)

time scale

`timescale

Verilog allows you to specify a time delay in a continuous assignment by following the keyword `assign` with a pound sign (#) and a real number, which may include a decimal point. This number indicates the delay in units of the *time scale* then in effect. The default may be 1 ns but can vary by tool, so you should specify it using the *`timescale* compiler directive, with the syntax below:

`timescale *time-unit* / *time-precision*

Here the "time-unit" indicates the new default units that will be associated with all delay numbers until the next `timescale, as well as with time values used by $time and other system functions and tasks. Although you could specify "100 ps", typically single units like "1 ps" and "1 ns" are specified to avoid confusion. The "time-precision," on the other hand, is often given in less round numbers. It specifies the time granularity with which the simulator will operate.

The smallest time unit or precision that can be specified is 1 femtosecond (fs)—or 10^{-15} s. Chips aren't fast enough to require that yet. But even with nanosecond time units (10^{-9} s), a 32-bit timer would "roll over" in about four seconds of simulated time (2^{32} ps). Therefore, time is maintained in Verilog as a 64-bit integer, and 64-bit integer variables can be declared using the keyword *time*. Such variables can be useful in simulation. Recall that the width of ordinary `integer` variables is compiler-dependent, and may be as small as 32 bits.

time keyword

Program 5-19 is a Verilog module with continuous-assignment statements that use delays. It specifies a time-unit of 1 ns and a time-precision of 100 ps. The assignment statements correspond to the individual AND and OR operations in the prime-number detector of Figure 3-24(c) on page 120, including a delay of 2 ns for AND operations and 3.5 ns for the OR operation. In synthesis, these delays are ignored, but in simulation, the outputs will be produced only after the specified delays.

Delays may be specified in procedural assignments, too, by writing the # sign and a delay number after the = or <= symbol. Yet another way to invoke the time dimension within a block of procedural code is with a *delay statement*,

delay statement

Program 5-19 Verilog model for a prime-number detector, with delays.

```
`timescale 1 ns / 100 ps
module Vrprimedly (N, F);
input [3:0] N;
output F;
wire N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0;

  assign #2 N3L_N0     = ~N[3]                 & N[0];
  assign #2 N3L_N2L_N1 = ~N[3] & ~N[2] &  N[1]       ;
  assign #2 N2L_N1_N0  =         ~N[2] &  N[1] & N[0];
  assign #2 N2_N1L_N0  =          N[2] & ~N[1] & N[0];
  assign #3.5 F = N3L_N0 | N3L_N2L_N1 | N2L_N1_N0 | N2_N1L_N0;
endmodule
```

which is simply a # sign and a delay number. A following semicolon is optional. This statement can be used to suspend the procedural block for the specified time period. In Section 5.13, we'll see how delay statements are used in Verilog test benches.

5.12 Simulation

Once you have a Verilog model whose syntax and semantics are correct, you can use a simulator to observe its operation. Although we won't go into great detail, it's useful to have a basic understanding of how such a simulator works.

simulation time

Simulator operation begins at a *simulation time* of zero. At this time, the simulator initializes all signals to a default value of "x". It also initializes any signals or variables for which initial values have been declared explicitly (see a Verilog reference manual for how to do this). Next, the simulator begins the execution of all the concurrent statements in the design.

event list

sensitivity matrix

process

Of course, the simulator can't really simulate all of the concurrent statements simultaneously, but it can pretend that it does, using a time-based *event list* and a *sensitivity matrix* based on all of their individual sensitivity lists. Each concurrent statement—continuous assignment, instance, `always`, or `initial`—gives rise to at least one software *process* in the simulator. Module instantiations give rise to one or more additional processes, depending on the module's definition (e.g., a module containing five continuous assignment statements, like in Program 5-19, gives rise to five software processes).

At simulation time zero, all of the software processes are scheduled for execution, and one of them is selected. If it corresponds to an `always` or `initial` block, all of its procedural statements are executed, unless and until a delay specification or delay statement is encountered, in which case the process is suspended. Execution of procedural statements includes any looping behavior that is specified. When the execution of the selected process is either completed or suspended, another one is selected, and so on, until all of the processes have been executed. This completes one *simulation cycle*.

simulation cycle

During its execution, a process may assign new values to nets and variables. In blocking assignments with no delay specification, the new values are assigned immediately. If a blocking or nonblocking assignment has a delay specification, then a new entry is scheduled on the event list to make the assignment effective after the specified delay.

A nonblocking assignment with no delay specification is supposed to take place in zero simulated time, but it is actually scheduled to occur at the current simulation time plus one "delta delay." The *delta delay* is an infinitesimally short time, such that the current simulation time plus any number of delta delays still equals the current simulation time. This concept allows software processes to execute multiple times if necessary, in zero simulated time.

delta delay

After a simulation cycle completes, the event list is scanned for the signal or signals that change at the next earliest time on the list. This may be as little as one delta delay later, or it may be a real circuit delay later, in which case the simulation time is advanced to this time. In any case, the scheduled signal changes are made. Some processes may be sensitive to the changing signals, as indicated by their sensitivity lists. The sensitivity matrix indicates, for each signal, which processes have that signal in their sensitivity list. All of the processes that are sensitive to a signal that just changed are scheduled for execution in the next simulation cycle, which now begins.

The simulator's two-phase operation of a simulation cycle followed by scanning the event list and making the next scheduled assignments goes on indefinitely, until the event list is empty. At this point the simulation is complete.

The event-list mechanism makes it possible to simulate the operation of concurrent processes even though the simulator may run on a single computer with a single thread of execution. The delta-delay mechanism ensures correct operation even though a process or a set of processes may require multiple executions, spanning several delta delays, before changing signals settle down to a stable value. This mechanism is also used to detect runaway processes (such as implied by "`assign X = ~X`"); if a thousand simulation cycles occur over a thousand delta delays without advancing simulation time by any "real" amount, it's most likely that something's amiss.

5.13 Test Benches

A test bench specifies a sequence of inputs to be applied by the simulator to an HDL model, such as a Verilog module. The entity being tested is often called the *unit under test (UUT)*, in accordance with traditional parlance in the hardware testing field, even though the UUT in this case is not a device, but an HDL model that specifies the behavior of a device.

unit under test (UUT)

```
initial
   procedural-statement

initial begin
   procedural-statement
   . . .
   procedural-statement
end
```

Table 5-27 Syntax of Verilog `initial` blocks.

`initial` block

`initial` keyword

A while ago, we promised to introduce one more kind of Verilog concurrent statement, which is typically used in test benches. An *`initial` block* has the syntax shown in Table 5-27. Like an `always` block, it contains one or more procedural statements, but it does not have a sensitivity list. An `initial` block executes just once, beginning at simulated time zero. As in an `always` block, the `begin-end` block can be named and can have its own variable and parameter declarations.

Program 5-20 is a test bench for our prime-number-detector modules. Just in case, it sets a default time scale of 1 ns. Like all test benches, the module has no inputs or outputs. It begins by declaring local signals `Num` and `Prime`, which are used to apply stimuli and observe the outputs of the UUT. Next, it instantiates the UUT (module `Vrprimed` in Program 5-6 on page 203). By changing just the module name in the instance statement, this test bench could instantiate any of this chapter's prime-number detectors, except for Program 5-14 (which has an extra output) and Program 5-17 (whose input vector has more bits).

The test bench uses an `initial` block and delay statements within a `for` loop to apply all 16 possible input combinations to the UUT. This is just about the simplest possible test bench—it merely applies inputs and does not check them in any way. When a simulator runs the test bench, it produces the output waveforms shown in Figure 5-4, which includes both the decimal value and the individual bits of the 4-bit vector `Num`, as well as the output value `Prime`, for the

Program 5-20 Verilog test bench for a prime-number-detector circuit.

```
`timescale 1 ns / 100 ps
module Vrprime_tb1 () ;
reg [3:0] Num;
wire Prime;

Vrprimed UUT ( .N(Num), .F(Prime) );

  initial begin : TB
    integer i;
    for (i = 0; i <=15; i = i+1 ) begin #10 Num = i;
  end
endmodule
```

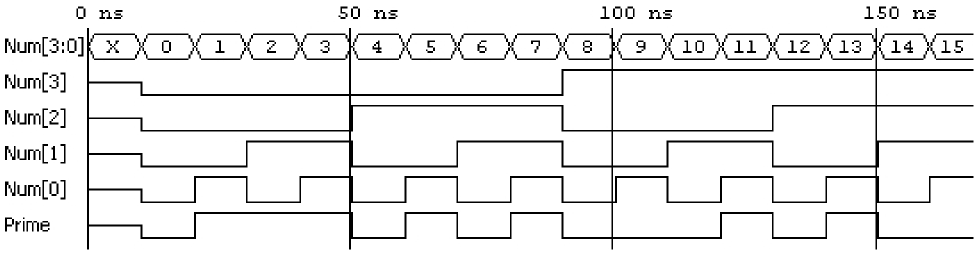


Figure 5-4 Timing waveforms produced by the `Vrprime_tb1` test bench.

16 input combinations that are applied. It's up to the user to look at the waveforms and determine if they make sense—a useful but tedious exercise.

It's normally worth expending a little more effort when writing a test bench to make its output more user friendly. For example, we can rewrite the previous test bench as shown in Program 5-21. It calls the `$write` and `$display` tasks to print the result for each iteration, typically to the "system console," or perhaps redirected to a file, depending on the system environment. In any case, instead of analyzing a timing diagram, we can now see the UUT's outputs displayed in text as in Table 5-28, which is much easier to check than a timing diagram. Also note the test bench's use of the case equality operator === to check the UUT output for 1 or 0 while flagging situations where the output is `x` or `z`.

Another approach is often used for test benches that evaluate a large number of input combinations, where it is not dependable or even feasible for an interactive user to examine a large number of input/output results. Instead, we can write a *self-checking test bench*, which compares the UUT's outputs against what's expected, keeping track of the number of errors if any, and displaying the discrepancies only when they occur.

self-checking test bench

Program 5-21 Improved test bench for a prime-number-detector circuit.

```
`timescale 1 ns / 100 ps
module Vrprime_tb2 () ;
reg [3:0] Num;
wire Prime;

Vrprimed UUT ( .N(Num), .F(Prime) );

initial begin : TB
  integer i;
  for (i = 0; i <=15; i = i+1 ) begin
    Num = i; #10   // Wait 10 ns per iteration
    $write ("Time: %3d  Number: %2d  Prime? ", $time, Num);
    if (Prime===1) $display ("Yes");
    else if (Prime===0) $display("No");
         else $display("Not sure");
  end
end
endmodule
```

Table 5-28 First few lines of output displayed by Program 5-21.

```
Time:  10  Number:  0  Prime? No
Time:  20  Number:  1  Prime? Yes
Time:  30  Number:  2  Prime? Yes
Time:  40  Number:  3  Prime? Yes
Time:  50  Number:  4  Prime? No
Time:  60  Number:  5  Prime? Yes
...
```

Most of the test-bench examples in this book are self-checking. Such a test bench for the prime-number detector is shown in Program 5-22. Here we use a `case` statement within the `for` loop to enumerate the expected value of the UUT's output for each input combination. And we define a "helper" task, `Check`, to print an error message if the UUT's output is different from what we expect. This test bench has the same compile-time option as some of our prime-detector modules—the value of parameter `OneIsPrime` should match the assumption that is used in the UUT.

Program 5-22 Self-checking test bench for prime-number detectors.

```
`timescale 1 ns / 100 ps
module Vrprime_tbc () ;
reg [3:0] Num;
wire Prime;
integer i, errors;
parameter OneIsPrime = 1; // Change to 0 if 1 is not prime.

task Check;
input xpect;
  if (Prime !== xpect) begin
    $display("Error: N = %b, expect %b, got %b",Num,xpect,Prime);
    errors = errors + 1;
  end
endtask

Vrprimedly UUT ( .N(Num), .F(Prime) );

initial begin
  errors = 0;
  for (i = 0; i <= 15; i = i+1) begin
    Num = i; #10 ;
    case (Num)
      4'd1 : Check(OneIsPrime);
      4'd2, 4'd3, 4'd5, 4'd7, 4'd11, 4'd13 : Check(1);
      default Check(0);
    endcase
  end
  $display("Test ended, %2d errors", errors); $stop(1);
end
endmodule
```

ROLLOVER ACCIDENTS

Notice in Program 5-22 that we use an `integer` variable `i` to control the `for` loop, and then assign the lower four bits of this integer to `Num` inside the loop. You may ask, "Why not just use `Num` to control the loop directly?" You'd have a subtle, unexpected problem if you did this. Since `reg` variable `Num` is only four bits wide, when it's 15 and you add 1 to it, it rolls over to 0 and the `for` loop keeps going forever.

In Program 5-22, we embedded the expected output values into the test-bench code, which is no fun if there are a lot of them. It is much more effective to calculate the expected output values *algorithmically* if at all possible. Verilog is used in test benches for programming, not modeling, and you can do many things with it that you could not or would not do in a synthesizable model.

For example, Program 5-23 shows an algorithmic, self-checking test bench for the 16-bit prime-number detector of Program 5-17. The test bench has its own local array of 2^{16} bits, indexed by 16-bit integer values, where a bit is 1 if and only if its index is prime. These bits are precomputed and stored in the array using the "Sieve of Eratosthenes" method when the test bench runs. After that, it is a simple matter for the test bench to apply all 2^{16} possible input combinations to the UUT and compare each resulting output with the correct value.

Partway between these approaches, designers in large projects may write test benches that read their inputs from and write their outputs to files, using Verilog's file I/O capabilities. This allows a designer to use any convenient and familiar programming language to create the test inputs and to check the outputs. After achieving satisfactory functional performance, a designer may even save the test bench's output file as a "golden" reference, which may be used subsequently in "regression testing" to ensure that subsequent modifications of the design (usually for performance, not functionality) do not change its functional output behavior.

Every test-bench module in this section and throughout this book is stored in a single text file with the following structure:

- Declare the module name, UUT's input and output signals, and any local variables that are used in testing.
- Instantiate one or more UUTs, each of which is defined in its own file(s).

DON'T KNOW WHEN TO $STOP?

Some of our test benches use `$stop` at the end; others don't. In interactive testing (as you'd likely be doing with the test benches in this book), it doesn't matter if there's no `$stop`. Upon reaching the end of the test bench, the simulator is still running, but there are no "events" left for it to process. It should return control to the system console after simulating no activity until the end of the simulation interval that was specified when it was invoked.

Program 5-23 Self-checking test bench that creates values for comparison algorithmically.

```
`timescale 1 ns / 100 ps
module Vrprimebv_tb ();
reg [15:0] N;
wire F;
reg prime [0:65535];       // Array to precompute primes; prime[i] = 1 if i is prime
integer i, try, errors;
parameter OneIsPrime = 1;  // Change to 0 if 1 is not prime.

Vrprimebv UUT (.N(N), .F(F));

initial begin
  for (i=0; i<=65535; i=i+1) prime[i] = 1; // All integers are potentially prime
  prime[0] = 0; prime[1] = OneIsPrime;     // except 0 and maybe 1
  for (try=2; try<=255; try=try+1) // Init array using "Sieve of Eratosthenes" method
    if (prime[try])                // Mark off multiples of primes; they're nonprimes
      for (i=try+try; i<=65535; i=i+try) prime[i] = 0;//
  // prime array is now initialized; check UUT operation
  errors = 0;
  for (i=0; i<=65535; i=i+1) begin
    N = i; #10;
    if (F !== prime[i]) begin
      errors = errors + 1;
      $display("Error: i=%5d, prime=%b, F=%b", i, prime[i], F);
    end
  end
  $display("Test complete, %d errors", errors); $stop(1);
end
endmodule
```

- Define helper tasks as needed.
- Create one or more code blocks (`always` and `initial`) for generating clocks and stimulus patterns and for checking results if applicable.

However, some designers or their companies prefer an alternative file structure that splits the above items into two files:

- The "top-level" test-bench file is a module that declares the UUT's input and output signals, instantiates one or more UUTs, and has an `` `include `` statement to fetch a *stimulus file* that contains the bulk of the test-bench code.

stimulus file

- The stimulus file contains Verilog code that defines local variables and helper tasks as needed, generates clocks and stimulus patterns, and checks results as applicable.

This alternative structure makes it easier to manage projects where different modules have the same inputs and outputs and can be checked with the same test

TABS VS. SPACES

Colleagues have told me that the choice of which test bench structure to use has become a religious issue for some designers. It's like the common "tabs vs. spaces" arguments about how best to indent code, which amusingly led to the end of a budding relationship in an episode of the TV series *Silicon Valley*.

In any programming language, there are many ways to express and accomplish the same thing—or sometimes *almost* the same thing, which is what often leads to the arguments. You'll see many examples of such alternatives throughout this book. In the end, you should learn and use whatever style is dictated in your work environment. At that point, the choice is not so much about "religion," but rather about the efficiencies that are facilitated by having everyone be "on the same page."

patterns—which can now be written and modified in just one place. It's also useful if modules have slightly different signal names or definitions, or have signals that are unused or tied connected to constant values in a particular application—a small amount of code can be used in the top-level module to adapt the UUT's inputs and outputs to the existing stimulus code, without having to change the stimulus file (which could break it for another application).

5.14 Verilog Features for Sequential Logic Design

Just one more Verilog language feature is used to describe common sequential-circuit behavior. We introduce it here, and will come back to it in later chapters.

Most Verilog-based digital design is directed to clocked, synchronous systems that use edge-triggered flip-flops. Like combinational behavior, edge-triggered behavior in Verilog is specified using `always` blocks. The difference between the two is governed by the sensitivity list of the `always` block.

`posedge` keyword
`negedge` keyword

Normally, an `always` block is executed upon any change in a signal named in the sensitivity list. When keyword `posedge` or `negedge` is placed in front of a signal name, the block is executed only upon the positive (rising) or negative (falling) edge of the named signal, as indicated. For the compiler to effectively map sequential `always` blocks into available RTL elements for synthesis, they must also match certain "templates." We'll show many examples of sequential behaviors starting in Section 10.3.2 and continuing through Chapter 13.

5.15 Synthesis

As we discussed at the beginning of this chapter, Verilog was originally designed as a logic circuit description and simulation language, and it was only later adapted to synthesis. Thus, the language has several features and constructs that cannot be synthesized. However, the subset of the language and the style of models that we've presented in this chapter are synthesizable by modern tools.

Still, the code that you write can affect the size and performance of circuits that are synthesized from your models. A few examples are listed below:

- "Serial" control structures like `if`, `else if`, `else if`, `...` `else` can result in a corresponding serial chain of logic gates to test conditions. Sometimes it's better to use a `case` statement, especially if the conditions are mutually exclusive, thereby suggesting a potentially more efficient multiplexer to select among alternatives.
- Loops in procedural code are generally "unwound" to create multiple copies of combinational logic, one copy for each iteration of the statements in the loop. If you want to use just one copy, then you have to design a sequential circuit, as described in later chapters.
- When you use conditional statements in procedural code, failing to code an outcome for some input combination will cause the synthesizer to create an inferred latch to hold the old value of a signal that might otherwise change. Such latches are generally not intended and affect performance.

In addition, some language features and constructs may just be unsynthesizable, depending on the tool. Naturally, you have to consult the documentation to find out what's disallowed, allowed, and recommended for a particular tool. A tool's synthesis manual also typically recommends templates for modeling various behaviors and hardware structures.

For the foreseeable future, digital designers will need to pay reasonably close attention to their coding style in order to obtain good synthesis results. And for the moment, the definition of "good coding style" depends somewhat on both the synthesis tool and the target technology. The examples in the rest of this book, while syntactically and semantically correct, barely scratch the surface of coding methods for large HDL-based designs, where art and practice are still evolving.

References

Verilog and VHDL are used every day by thousands of digital designers, and their associated compilers and other tools are well supported by many different suppliers. Both HDLs have active user communities, and there are frequent workshops and conferences devoted to applications and enhancements of the languages and tools. Because of all this activity, you can easily find up-to-date HDL references, examples, and tutorials on the Web. For example, searching for "Verilog tutorial" yields many solid hits.

There are good print references, too. If you're struggling with my concise introduction to Verilog, try *Starter's Guide to Verilog 2001* by Michael D. Ciletti (Pearson, 2003). Also consider *Digital Design and Verilog HDL Fundamentals* by Joseph Cavanaugh (CRC Press, 2008) which, like the book you're reading, covers general topics using Verilog as the design language, but is twice as thick.

WHAT IS "LRM"? The IEEE Std 1364-2001 document is 791 pages long and is commonly known as a language reference manual (LRM). The document uses the acronym "LRM" in a few places but never gives its definition; to get that, you have to read this box or look in the 1315-page IEEE 1800-2012 SystemVerilog standard.

Verilog HDL is defined in the *IEEE Standard Verilog Hardware Description Language*, IEEE Std 1364-2001. If you like to read specs, you can buy the complete standard document from the IEEE, but Stuart Sutherland wrote an excellent *Verilog Quick Reference Guide* for Verilog-2001 based on the standard and published it on his company's website, `sutherland-hdl.com`.

As the Verilog-2001 standard was published, various companies continued to extend Verilog's HDL-based design capabilities. After several years of work, IEEE Std 1364-2005 was published with a few extensions of Verilog-2001. But at the same time, over 100 substantial enhancements were made to Verilog's design and verification capabilities to create a superset of the language, called "SystemVerilog," formally defined in IEEE Std 1800-2005. The "plain" 1394 version of Verilog was eventually merged with SystemVerilog to create a unified IEEE Std 1800-2008, which supersedes IEEE 1364. The latest unified standard is IEEE 1800-2012. All of the Verilog features used in this book exist in Verilog-2001 and later.

Keep in mind that the IEEE standards are specifications, not tutorials. For an introduction to SystemVerilog, you can consult a text like *Logic Design and Verification Using SystemVerilog* by Donald Thomas (CreateSpace, 2016).

As mentioned previously, there's a lot of good HDL reference material on the Web. But the practical and insightful articles by Clifford E. Cummings and his colleagues on Verilog features, usage, and coding styles are especially worth reading (go to `www.sunburst-design.com` or search for 'Cummings Verilog'). For example, our discussion of the rules for blocking versus nonblocking assignments on page 209 is based on one of his articles.

All of the Verilog examples in this chapter and throughout this book were compiled and simulated using the free "WebPack" edition of the Xilinx Vivado tool suite (Xilinx, Inc., San Jose, CA 95124, `www.xilinx.com`). Like most other digital-design tools, it runs on a PC using the Windows operating system.

Drill Problems

5.1 Write a structural Verilog module corresponding to the NAND-gate based logic circuit in Figure 4-18.

5.2 Write a structural Verilog module for the combinational circuit in Figure 6-15.

5.3 Write a dataflow-style Verilog module for the alarm circuit in Figure 3-16.

5.4 Can Verilog's built-in `and` and `or` components be used with just one input?

5.5 Write a Verilog module for the alarm circuit in Figure 3-16 using an `always` block and a behavioral description style.

5.6 Write a structural Verilog module `Vr3inckt_s` for the logic circuit in Figure 3-5.

5.7 Write a dataflow-style Verilog module `Vr3inckt_d` for the logic circuit in Figure 3-5.

5.8 Write a Verilog module `Vr3inckt_b` for the logic circuit in Figure 3-5, using an `always` block and a behavioral description style.

5.9 Write a Verilog test bench that instantiates all three modules from Drills 5.6 through 5.8, and applies all eight possible input combinations to them, at 10 ns per step. Your test bench should display the output value for each input combination, but need not check them. Instead, you should manually compare the outputs with the values shown in Figure 3-6.

5.10 Write a Verilog test bench that instantiates all three modules from drills 5.6 through 5.8, and applies all eight possible input combinations to them, comparing the results and displaying an error message if any of the results differ. Introduce a bug into each module to ensure that your error detection and display code is working properly.

5.11 Rewrite the module of Program 5-2, 5-6, or 5-8 using ANSI-style declarations.

5.12 Which assignment operator should you use in Verilog `always` blocks intended to synthesize combinational logic, = or <=?

5.13 If multiple values are assigned to the same signal in a Verilog combinational `always` block, what is the signal's value when the `always` block completes execution: (a) the AND of all values assigned; (b) the OR of all values; (c) the last value assigned; or (d) it depends?

5.14 Assume that `A`, `B` and `C` are 2-bit `reg` vectors with values `2'b01`, `2'b10`, and `2'b11`, respectively, when an `always` block is entered. The block executes a sequence of three assignment statements, "`C=B; A=C; B=A;`". What are the final values of `A`, `B` and `C`?

5.15 Repeat Drill 5.14 with the statement sequence, "`C<=B; A<=C; B<=A;`".

5.16 Write a test bench to check your answers for Drills 5.14 and 5.15.

5.17 Synthesize the `VrSillyXOR` Verilog module in Program 5-4, targeting your favorite programmable device. Determine whether the synthesizer is smart enough to realize the module using a single XOR gate.

5.18 A possible definition of a BUT gate (Exercise 3.37) is "Y1 is 1 if A1 and B1 are 1 *but* either A2 or B2 is 0; Y2 is defined symmetrically." Write a behavioral-style Verilog model for such a BUT gate.

5.19 Run the test bench in Program 5-22 with and without the `$stop` statement. What difference does it make in your environment?

Exercises

5.20 Verilog-2001 has a syntax option for defining parameters and their default values as part of an ANSI-style module declaration. Look it up online, and rewrite the declarations in the `Maj` module of Program 5-5 using this option.

5.21 Find a situation where adding an extra semicolon (treated as a null statement) in Verilog procedural code creates a syntax error.

5.22 The logical expression "`(N % 2) == 0`" in Program 5-14 can be written more concisely using only five characters and still yield the same result. Show the alternate formulation, and comment on its pros and cons.

5.23 Write a Verilog module `VrM35dec` for a logic function with six inputs N5–N0 representing an integer between 0 and 63, and two outputs M3 and M5 indicating whether the integer is a multiple of 3 or 5, respectively. Target your design to an available programmable device and determine how many resources it uses.

5.24 After completing the preceding exercise, write a Verilog test bench that compares the outputs of your module for all possible input combinations against results computed by the simulator using its own arithmetic. The test bench should stop and display the actual and computed outputs if there is a mismatch. Test your test bench by putting a bug in your original Verilog model and running the test bench. You get extra credit (in your own mind, at least) if you already had an unintentional bug in your initial module and detected it!

5.25 Write a structural Verilog model that instantiates a single 2-input OR gate and the BUT gate component of Drill 5.18 to realize the 4-input logic function F = $\Sigma_{W,X,Y,Z}(5,7,10,11,13,14)$. Write a test bench that checks your circuit's output for all 16 possible input combinations and displays a message if there's an error.

5.26 Modify the `Vrprimebv` module in Program 5-17 to find 8-bit primes. Then use this module in a test bench to print all the primes between 0 and 255.

5.27 Write a dataflow-style Verilog module corresponding to the full-adder circuit in Figure 8-1. Then instantiate multiple copies of it to create a structural Verilog model for a 4-bit ripple adder using the structure of Figure 8-2.

5.28 After doing Exercise 5.27, write a Verilog test bench that tests the adder for all possible pairs of 4-bit addends. The test bench should stop and display the actual and expected outputs if there is any mismatch.

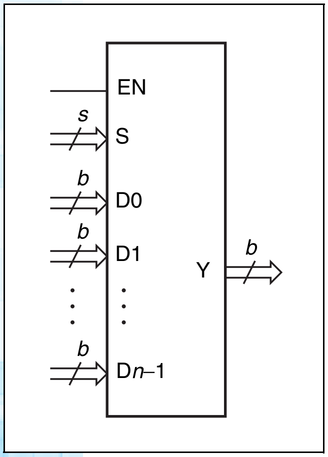
5.29 Using the module that you defined in Exercise 5.27, write a structural Verilog model for a 16-bit ripple adder along the lines of Figure 8-2. Research and then use a `generate` statement to create the 16 full adders and their connections.

5.30 After doing Exercise 5.29, write a Verilog test bench that tests the adder for a subset of the 2^{32} possible pairs of 16-bit addends, using the `$random` function. The test bench should stop and display the actual and expected outputs if there is any mismatch.

5.31 Write a Verilog test bench to prove that `W|X&Y|Z` is the same as `W|(X&Y)|Z`.

5.32 Do some Verilog research, and determine how Verilog handles "ambiguous" logical values and expressions that may be either true or false, such as `4'bxx00`. Write a small test bench program that demonstrates that they are handled as you say they are.

5.33 Do some Verilog research, and learn about Verilog file I/O. Then write a test bench that checks the output of one of the `prime` modules (such as Program 5-6, 5-7, or 5-9) for all possible inputs, reading the expected output values from a file.



chapter 6

Basic Combinational Logic Elements

The theoretical principles used in combinational logic design were described previously in Chapter 3. Now, we'll build on that foundation and describe many of the devices, structures, and methods used by engineers to solve practical digital design problems. We'll give examples using individual gates and drawing logic diagrams as we did in Chapter 4, and we'll also give examples using the hardware description language Verilog from Chapter 5.

A practical combinational circuit may have dozens of inputs and outputs, and could require hundreds, thousands, even millions of terms to describe as a sum of products, and billions of rows to describe in a truth table. Thus, most real combinational logic design problems are too large to solve by the "brute-force" application of theoretical techniques.

But wait, you say, "How could any human being conceive of such a complex logic circuit in the first place?" Large, complex systems such as software applications, communication networks, and transportation networks are usually described hierarchically, and digital systems are no exception. The key is hierarchical thinking. A complex circuit or system is conceived as a collection of smaller subsystems, each of which has a much simpler description.

In combinational logic design, there are several common operations—decoding, selecting, comparing, and the like—that turn up quite regularly, and there are corresponding structures for performing these operations with gate-level circuits, functional building blocks, or Verilog models. These structures may be combined with each other and with sequential-circuit structures to build larger systems, as we'll show in later chapters.

Digital systems are hardware, and when they were simpler and smaller, they would be designed and specified at the high level using block diagrams, and at the low level using schematic diagrams that show physical components and their wired interconnections. Nowadays, it is much more common for lower-level elements in the hierarchy to be specified using HDLs, either by instantiating predefined library components that perform the needed functions, or by defining purpose-specific modules that perform the functions. And as you know, HDLs themselves are hierarchical, allowing much more than just the bottom level of the digital system's hierarchy to be specified in the HDL.

Although modern HDLs like Verilog and VHDL allow a digital system or subsystem to be specified structurally—by defining a collection of physical components and their interconnections—they are used much more often to describe a system or subsystem *behaviorally.* A synthesis tool then translates the behavioral description into a physical structure that has the described behavior.

Either way, when the time comes to create a physical realization of the overall design, an EDA tool "flattens" the hierarchy and specifies its implementation as cells and interconnections in an ASIC, or in a programmable device like an FPGA, perhaps also including off-the-shelf components to be interconnected at the printed-circuit-board level.

With combinational circuits, there are several different, traditional ways to describe or specify a given function or behavior. Each of these happens to correspond to an implementation approach that works well with one or more modern technologies, so it's useful for you to be familiar with them:

- *Truth tables* are the most basic and the most exhaustive way of specifying a combinational logic function, and can be programmed into a read-only memory (ROM) to implement any such function—as long as the ROM has enough inputs and outputs. Once considered a slow and inefficient way of implementing combinational logic, truth tables and ROMs have become very important in the past two decades because of the relentless increases in the size and performance of FPGAs where they are used. Even when a function is too big to fit in a single ROM, as most are, modern tools can decompose the function to fit into multiple, interconnected ROMs.
- *Two-level sum-of-products and product-of-sums* expressions and resulting gate-level circuits (AND-OR/NAND-NAND and OR-AND/NOR-NOR) were the focus of traditional logic design in the days of discrete SSI gates. Automated tools have long been available to minimize the size of such circuits, starting with a functional specification such as a truth table or a non-minimized logic expression. These structures continue to be important for arbitrary logic functions implemented using either discrete gates in an

ASIC or in a programmable logic device which contains a programmable AND-OR array. Even when a logic expression is too large to be implemented in just two levels of logic or in a single FPGA LUT, experience has shown that a minimized two-level expression is still a good starting point for factoring and other transformations that can "fit" the function into the available logic structures.

- *Building blocks* for many commonly used logic operations were offered as individual chips in the days of board-level logic design using MSI devices, and comparable building blocks are still available in many ASIC and other component libraries. Such building blocks are still important because many designs are conveniently modeled in terms of the operations they perform and designers may structure their designs using these operations.

building-block logic

Building-block logic typically performs a function that is easily described in words, often coming directly from the way we think about the problem it's solving in the first place, such as:

- Recognizing an input value and activating a corresponding output.
- Converting a set of input values into a corresponding but different set of output values.
- Selecting one of multiple input buses to send to an output bus.
- Comparing input buses for equality or other relationships (e.g., arithmetic less-than).
- Combining inputs to produce an output (as in addition and subtraction).

It is usually possible to describe these functions behaviorally in an HDL in a way that is well-structured and quite succinct, using language features that have been provided for just such purposes. As you'll see, the hardware circuit for realizing one of these functions typically has a regular and easily recognizable structure.

Still, there are many design problems that don't match the common building blocks. Combinational logic is often used to evaluate a set of conditions or other inputs, and to activate one or more outputs as a function of them. We showed a few functions of that kind in Chapter 3; for example, Figure 3-16 on page 113 showed the circuit for a combinational function that activates an alarm signal based on the values of six different condition inputs. We'll give a much more elaborate example in Section 7.5. Such logic is sometimes called "*random logic*," but there's really nothing "random" about it; it almost always has a definite, non-random purpose! A better name would be "arbitrary logic." Still, such logic circuits often appear, like Figure 3-16, to be a collection of logic gates that have been randomly thrown together (which is surely how "random logic" got its name in the first place!).

random logic

We'll start this chapter by describing two "universal" structures for combinational logic, ROMs and PLAs/PLDs, that can implement arbitrary logic functions including random logic. Then we describe decoders and multiplexers, the most commonly used combinational-logic building blocks. We'll describe where each might be used, show how it can be created at the gate level, and also show how its behavior can be specified using Verilog. Chapter 7 does the same for other combinational building blocks and concludes with a "random logic" example. Chapter 8 focuses on combinational structures for arithmetic operations like addition and multiplication.

6.1 Read-Only Memories (ROMs)

read-only memory (ROM)

You may already be familiar with *read-only memories (ROMs)*, or at least their application in computers and portable devices where very large ROMs store programs and data. You may know them as "flash memories." Although these memories can be written, at least at initialization, they are *mostly* read-only; we'll talk about that in Section 15.1. In any case, here we will focus on the use of ROMs, usually much smaller ones, as combinational logic elements.

address input

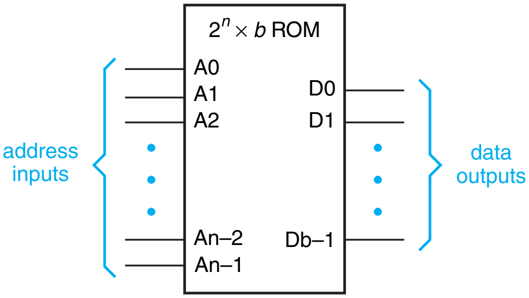
data output

A basic ROM is a combinational circuit with n inputs and b outputs, as shown in Figure 6-1. Like other memories, internally the ROM is a two-dimensional array where each row or "location" stores a b-bit "word" of data. The inputs are called *address inputs* and are traditionally named An–1, An–2, …, A1, A0, and the bit-vector A[n–1:0] is typically interpreted as an unsigned n-bit integer. Each of the 2^n binary combinations of A[n–1:0] selects a corresponding location in the ROM. The outputs are called *data outputs*, and they are typically named Db–1, Db–2, … , D1, D0.

Figure 6-2 shows the basic timing diagram for a ROM's operation. Signal values are applied to the address inputs [An–1:A0]. Once they are stable, the data outputs D[b–1:0] are stable after a propagation delay time, t_{pd}, and equal the data value stored at the applied address. Even though the word "memory" is in its name, a ROM is a combinational circuit, because its output is always (except for propagation delay) a function of its current input.

So, you can treat a ROM like any other combinational logic element. A ROM is called a "memory" mainly because of the organizational paradigm that

Figure 6-1
Basic structure of a $2^n \times b$ ROM.



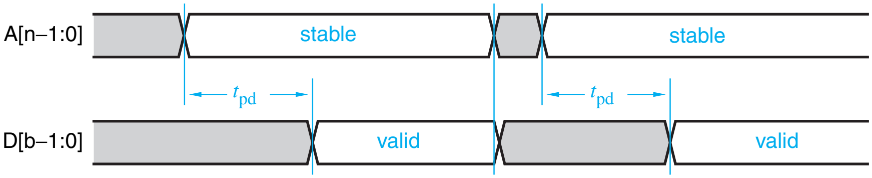


Figure 6-2
Basic ROM timing.

describes its operation. Also, you can think of information as being "stored" in the ROM when it's programmed—we'll discuss how that's done in Section 15.1.

Although we think of ROM as being a type of memory, it has an important difference from many other types of integrated-circuit memory. A true ROM is *nonvolatile memory*; that is, its contents are preserved even when no power is applied.

nonvolatile memory

6.1.1 ROMs and Truth Tables

It's perhaps even more clear that a ROM is a combinational circuit when we realize that it can "store" the truth table of an *n*-input, *b*-output combinational logic function. For example, Table 6-1 is the truth table of a 3-input, 4-output combinational function; it could be stored in a $2^3 \times 4$ (8×4) ROM. Except for signal propagation delay, a ROM's data outputs at all times equal the output bits in the truth-table row selected by the address inputs.

6.1.2 Using ROMs for Arbitrary Combinational Logic Functions

Table 6-1 is actually the truth table of a 2-to-4 decoder with an output-polarity control, a simple variant of a commonly used logic function that we'll describe in Section 6.3. This function can be built using discrete gates, as shown in Figure 6-3. Thus, there are at least two different ways to build the decoder—with

Inputs			*Outputs*			
A2	***A1***	***A0***	***D3***	***D2***	***D1***	***D0***
0	0	0	1	1	1	0
0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	1
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

Table 6-1
Truth table for a 3-input, 4-output combinational logic function.

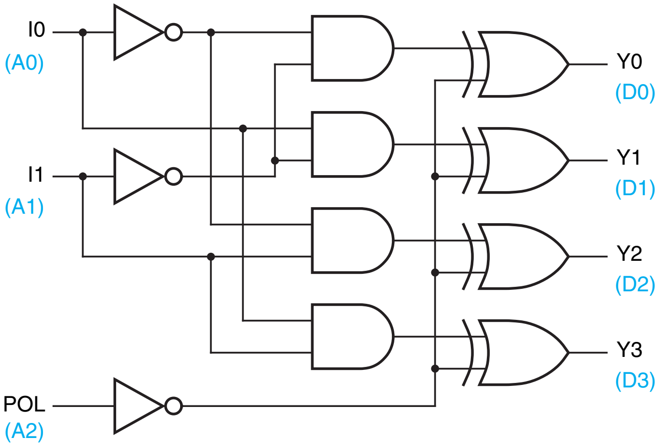
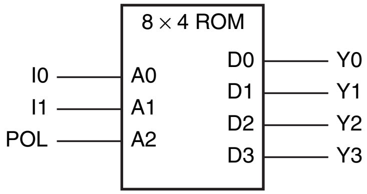


Figure 6-3
A 2-to-4 decoder with output-polarity control.

discrete gates, or with an 8 ×4 ROM that contains its truth table, as shown in Figure 6-4.

When constructing a ROM to store a given truth table, we normally assign input and output signals, reading from right to left in the truth table, to ROM address inputs and data outputs with ascending labels. Each address or data combination may then be read as a corresponding binary integer with the bits numbered in the "natural" way. A data file is typically used to specify the truth table to be stored in the ROM when it is programmed. The data file may give the

Figure 6-4
Connections to build the 2-to-4 decoder using an 8 × 4 ROM that stores Table 6-1.



LET ME COUNT THE WAYS

The assignment pattern of decoder inputs and outputs to ROM inputs and outputs in Figure 6-4 is a consequence of the way that the truth table in Table 6-1 is constructed. Thus, the physical ROM realization of the decoder is not unique. That is, we could write the rows or columns of the truth table in a different order and use a physically different ROM to perform the same logic function, simply by assigning the decoder signals to different ROM inputs and outputs. Another way to look at this is that we can rename the individual address inputs and data outputs of the ROM.

Because there are 3! ways to arrange the inputs, and 4! ways to arrange the outputs, there are 3! × 4! or 144 possible assignments of inputs and outputs to the ROM pins, each with a corresponding arrangement of the truth table in the ROM.

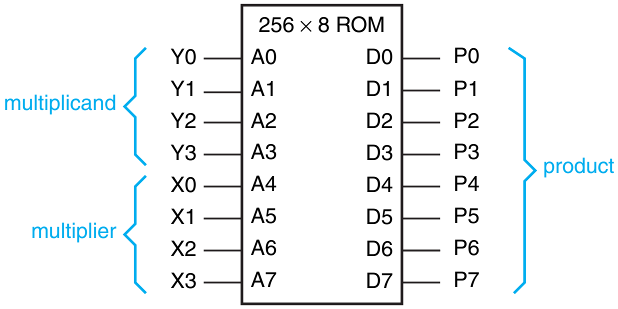


Figure 6-5 Connections to perform a 4 × 4 unsigned binary multiplication using a 256 × 8 ROM.

address and data values as hexadecimal numbers. For example, a data file may specify Table 6-1 by saying that ROM addresses 0–7 should store the values E, D, B, 7, 1, 2, 4, 8, respectively.

Another simple example of a combinational logic function that can be built with ROM is a 4×4 unsigned binary multiplication. As we'll see in Section 8.3, multipliers are fairly complex combinational circuits, and they can be slow, requiring many levels of logic to get the job done. Alternatively, we can realize a 4×4 multiplier using a $2^8 \times 8$ (256 ×8) ROM with the connections shown in Figure 6-5. Table 6-2 is a hexadecimal listing of the 4 ×4 multiplier ROM contents. Each row gives a starting address in the ROM, and specifies the 8-bit data values stored at 16 successive addresses. A nice thing about ROM-based design is that you can usually write a simple program in a high-level language to calculate the values to be stored in the ROM (see Exercise 6.20).

Chapter 15 has several Drills and Exercises on the subject of using ROMs for larger combinational logic functions.

```
00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
20: 00 02 04 06 08 0A 0C 0E 10 12 14 16 18 1A 1C 1E
30: 00 03 06 09 0C 0F 12 15 18 1B 1E 21 24 27 2A 2D
40: 00 04 08 0C 10 14 18 1C 20 24 28 2C 30 34 38 3C
50: 00 05 0A 0F 14 19 1E 23 28 2D 32 37 3C 41 46 4B
60: 00 06 0C 12 18 1E 24 2A 30 36 3C 42 48 4E 54 5A
70: 00 07 0E 15 1C 23 2A 31 38 3F 46 4D 54 5B 62 69
80: 00 08 10 18 20 28 30 38 40 48 50 58 60 68 70 78
90: 00 09 12 1B 24 2D 36 3F 48 51 5A 63 6C 75 7E 87
A0: 00 0A 14 1E 28 32 3C 46 50 5A 64 6E 78 82 8C 96
B0: 00 0B 16 21 2C 37 42 4D 58 63 6E 79 84 8F 9A A5
C0: 00 0C 18 24 30 3C 48 54 60 6C 78 84 90 9C A8 B4
D0: 00 0D 1A 27 34 41 4E 5B 68 75 82 8F 9C A9 B6 C3
E0: 00 0E 1C 2A 38 46 54 62 70 7E 8C 9A A8 B6 C4 D2
F0: 00 0F 1E 2D 3C 4B 5A 69 78 87 96 A5 B4 C3 D2 E1
```

Table 6-2 Hexadecimal text file specifying the contents of a 4 × 4 multiplier ROM.

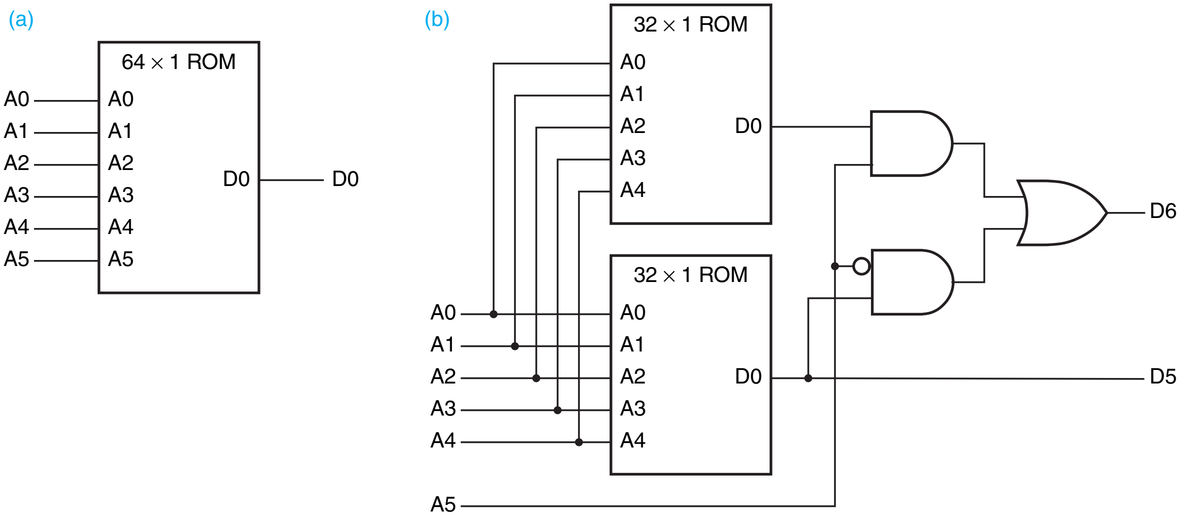


Figure 6-6 Xilinx 7-series 6-input LUT: (a) simple model; (b) actual structure.

6.1.3 FPGA Lookup Tables (LUTs)

lookup table (LUT)

FPGAs use small read-only memories called *lookup tables (LUTs)* to perform logic functions. By storing a truth table as explained in the preceding subsections, a $2^n \times 1$ LUT can perform any 1-output logic function of up to n inputs, where n is typically in the range of 4 to 6. So, a key task of any FPGA synthesis tool is to "decompose" logic functions with more than 4–6 inputs into an interconnected collection of smaller ones, each of which fits within an available LUT.

The LUTs in one of the more advanced FPGA families, the Xilinx 7 series, have six inputs; thus, each LUT can be thought of as being a 64 × 1-bit ROM as shown in Figure 6-6(a). However, each LUT is actually built as two 32 × 1-bit ROMs sharing the same low-order address bits A4-A0 as shown in (b). The most significant input bit, A5, selects whether the output named D6 is taken from one or the other of the two 32 × 1-bit ROMs, thus providing the overall functionality of a 64 × 1-bit ROM.

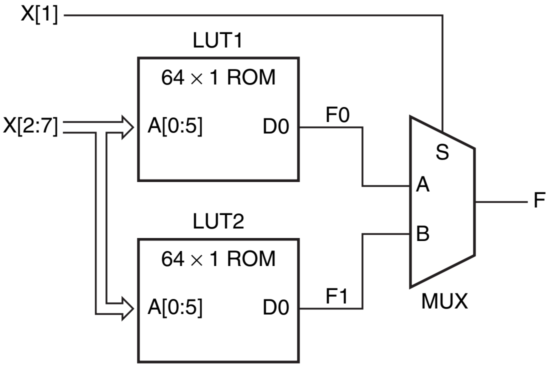


Figure 6-7 Shannon decomposition of a 7-input function using two LUTs.

AT LEAST IT ACTS LIKE A ROM

When we say that a lookup table is stored in ROM, we are talking about how it is used in normal operation—as a read-only table. However, the lookup table must be programmed into this "ROM" at some point, and there are different ways to do this, depending on the particular FPGA device. Often, the read-only configuration data for all of the FPGA's lookup tables is stored in an external ROM chip, and that data is written into small RAM-based lookup tables in the FPGA at system power-up. That's how it's done in the Xilinx 7 series.

Some FPGA devices include an erasable, programmable ROM such as flash memory right on the same chip. This eliminates the external ROM chip, but the now on-chip configuration data is still written into small RAM-based lookup tables at system power-up or initialization.

In yet another variation, some FPGAs actually do contain small on-chip ROMs, one per lookup table, which are programmed just once, when the device is installed in a system. Unlike the other variations, these do not have to be reprogrammed every time the FPGA is powered up.

The structure in Figure 6-6(b) provides an optional way for the LUT to implement any 2-output logic function of five inputs, A4-A0. When the sixth input, A5, is held at a constant 1 value, the top 32 × 1-bit ROM generates any desired function of the five inputs, A4-A0, and places it on the D6 output. At the same time, the bottom 32 × 1-bit ROM independently generates any desired function of the same five inputs, and places it on the output named D5.

OUT OF LUT?

What happens if we need to implement a combinational logic function with say, seven inputs, and the LUTs in our FPGA have only six inputs? Are we out of luck?

To solve this problem, the 7-input function, or any logic function, can be "decomposed" into two or more functions which have six or fewer inputs each, according to a well- known theory of functional decomposition. One straightforward way to do this is using Shannon's decomposition theorem, shown in Table 3-3 on page 97. According to T15, any 7-input function $F(X_1, X_2, \ldots, X_7)$ may be realized as shown in Figure 6-7 using two 6-input LUTs and a 2-input multiplexer ("MUX"), which selects one or the other of its left-hand inputs depending on the value of the top input.

Shannon decomposition can be repeated, for example, to realize an 8-input function as two 7-bit functions, and then further decomposing each of those. Xilinx 6-series and later FPGAs actually have such muxes (called "F7MUX" and "F8MUX") and connections built in. Similarly, Altera Stratix-IV and later FPGAs have LUT-combining logic to build larger functions from smaller LUTs. FPGA synthesis tools can use Shannon decomposition and other approaches to realize even larger combinational logic functions using multiple LUTs.

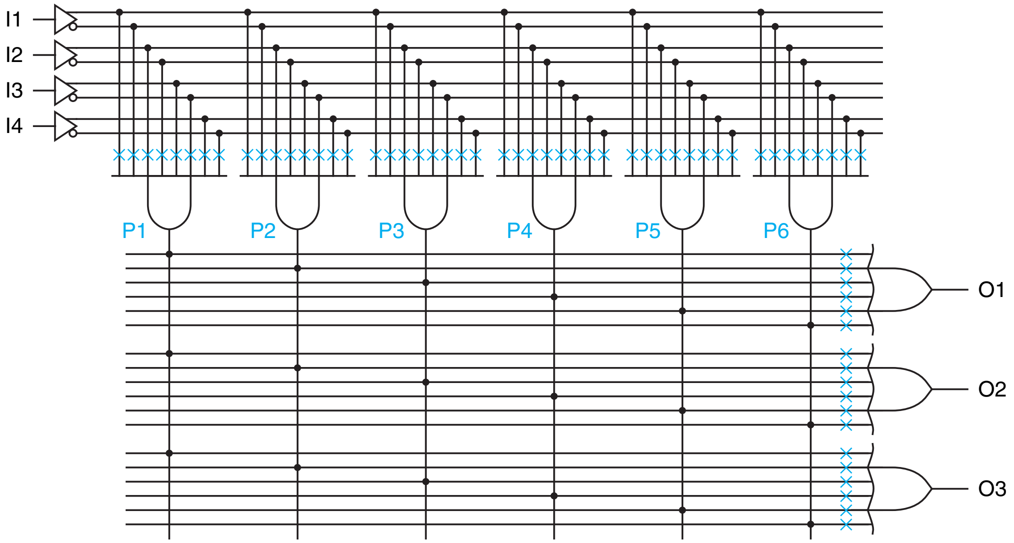


Figure 6-8 A 4 × 3 PLA with six product terms.

*6.2 Combinational PLDs

*6.2.1 Programmable Logic Arrays

programmable logic array (PLA)

Historically, the first programmable logic devices (PLDs) were *programmable logic arrays (PLAs)*, and they form an important basis for understanding later PLDs. A PLA is simply a combinational, two-level AND-OR device that can be programmed to realize any sum-of-products logic expression, subject to the size limitations of the device. Limitations are:

inputs

outputs

product terms

- the number of inputs (n),
- the number of outputs (m), and
- the number of product terms (p).

We might describe such a device as "an $n \times m$ PLA with p product terms." In general, p is far less than the number of n-variable minterms (2^n). Thus, unlike a LUT, a PLA cannot perform arbitrary n-input, m-output logic functions; its usefulness is limited to functions that can be expressed in sum-of-products form using p or fewer product terms.

An $n \times m$ PLA with p product terms contains p $2n$-input AND gates, and m p-input OR gates. Figure 6-8 shows a small PLA with four inputs, six AND gates, and three OR gates and outputs. Each input connects to a buffer/inverter that produces both a true and a complemented version of the signal for use within the array. Potential connections in the array are indicated by X's; the device is programmed by making only the connections that are actually needed.

*Throughout this book, optional sections are marked with an asterisk.

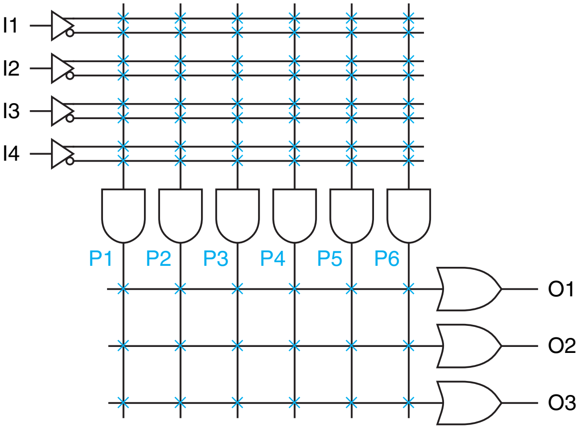


Figure 6-9
Compact representation of a 4 × 3 PLA with six product terms.

The selected connections are made by *fuses*, which in newer devices typically are not actually fuses, but are nonvolatile memory cells that can be programmed to make a connection or not. Thus, each AND gate's inputs can be any subset of the primary input signals and their complements. Similarly, each OR gate's inputs can be any subset of the AND-gate outputs.

PLA fuses

As shown in Figure 6-9, a more compact diagram can be used to represent a PLA. Moreover, the layout of this diagram more closely resembles the actual internal layout of a PLA on-chip.

PLA diagram

The PLA in Figure 6-9 can perform any three 4-input combinational logic functions that can be written as sums of products using a total of six or fewer distinct product terms, for example:

$$\begin{aligned} O1 &= I1 \cdot I2 + I1' \cdot I2' \cdot I3' \cdot I4' \\ O2 &= I1 \cdot I3' + I1' \cdot I3 \cdot I4 + I2 \\ O3 &= I1 \cdot I2 + I1 \cdot I3' + I1' \cdot I2' \cdot I4' \end{aligned}$$

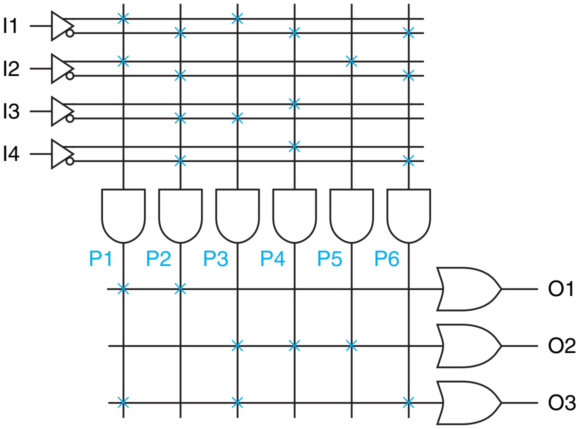


Figure 6-10
A 4 × 3 PLA programmed with a set of three logic equations.

These equations have a total of eight product terms, but the first two terms in the O3 equation are the same as the first terms in the O1 and O2 equations. The programmed connection pattern in Figure 6-10 matches these logic equations.

Our example PLA has too few inputs, outputs, and AND gates (product terms) to be very useful. An n-input PLA could conceivably use as many as 2^n product terms to realize all possible n-variable minterms. The actual number of product terms in typical commercial PLAs is far fewer, on the order of 4 to 16 per output, regardless of the value of n.

*6.2.2 Programmable Array Logic Devices

programmable array logic (PAL) device

A special case of a PLA, and the basis of the most commonly used PLDs, is the *programmable array logic (PAL) device*. Unlike a PLA, in which both the AND and OR arrays are programmable, a PAL device has a *fixed* OR array.

The first PAL devices were introduced in the late 1970s and used bipolar transistors technology, not today's CMOS technology. Key innovations in the first PAL devices, besides the introduction of a catchy acronym, were the use of a fixed OR array and bidirectional input/output pins.

PAL16L8

These ideas are well illustrated by the *PAL16L8* device, which is shown in Figure 6-11. Its programmable AND array has 64 rows and 32 columns, so there are 64 × 32 = 2048 fuses. Each of the 64 AND gates in the array has 32 inputs, accommodating 16 variables and their complements. The device has up to 16 inputs and 8 outputs; hence the "16" and the "8" in "PAL16L8".

Eight AND gates are associated with each output pin. Seven of them are inputs to a fixed 7-input OR gate. The eighth connects to the three-state enable input of an inverting output buffer; the buffer is enabled and drives its output pin only when the eighth AND gate has a 1 output. Thus, a PAL16L8 output can perform only logic functions that can be written as inverted sums of seven or fewer product terms. Each product term can be a function of any or all 16 inputs, but only seven product terms are available per output.

Although the PAL16L8 has up to 16 inputs and up to 8 outputs, it comes in a package that has only 18 input/output pins. This pin-count savings is achieved because of six bidirectional pins that may be used as inputs or outputs or both. Modern programmable devices still use this idea to provide lots of application flexibility without dedicating a package pin to every possible input and output.

COMBINATIONAL, NOT COMBINATORIAL!

A step *backward* in the introduction of PAL devices was the manufacturer's use and popularization of the word "combinatorial" to describe combinational circuits. *Combinational* circuits have no memory—their output at any time depends on the current input *combination*. For well-rounded computer engineers, the word "combinatorial" should conjure up vivid images of binomial coefficients, problem-solving complexity, and computer-science-great Donald Knuth.

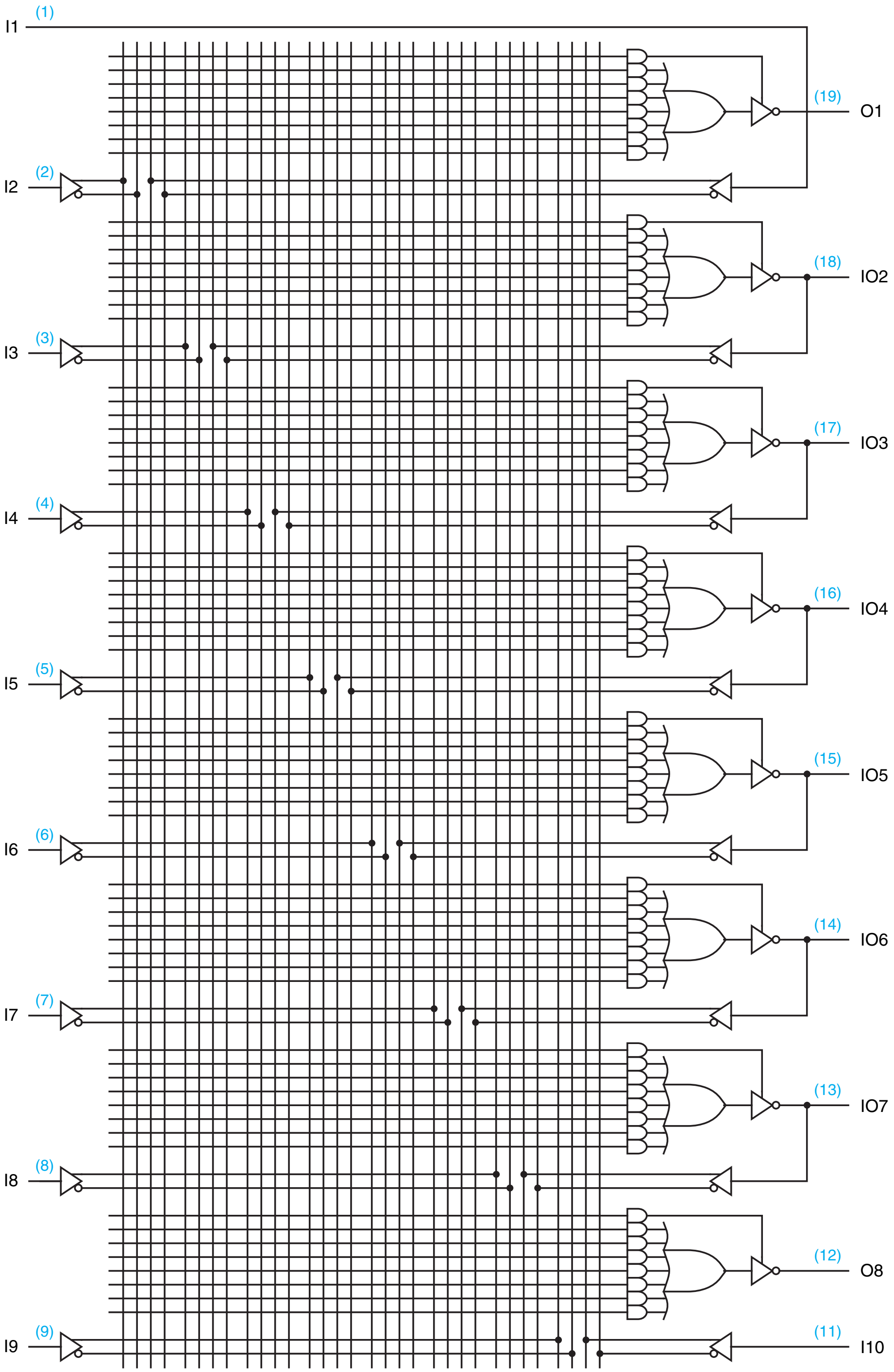


Figure 6-11 Logic diagram of the PAL16L8.

6.3 Decoding and Selecting

Many applications require a combinational circuit that activates one or more other circuits, elements, or operations based on the value of inputs that specify the desired operation. For example, a computer might have four USB ports, each of which is activated by an "enable" input. A program on the computer might provide a 2-bit "select" value to denote which of the four USB ports is to be used at a particular time. As shown in Figure 6-12, a circuit could receive the 2-bit "port-select" value from a program instruction, and provide four outputs connected to the four USB ports' enable inputs. This circuit is said to "decode" the port-select value, and is called a *decoder*. The decoder in this example is called a *2-to-4 binary decoder*,

decoder

2-to-4 binary decoder

The decoder in this example is designed so that no more than one output is asserted at any time, and only if the corresponding port-select value appears on the input. Most decoders also have one or more *enable inputs*, like EN_USB in Figure 6-12, so the selected output is asserted only if the enable input is asserted.

enable input

A decoder's input code-word bits are often called *address bits*. One of the most common applications of decoders is to decode addresses, to selectively enable memories and other components and devices as in our earlier example, and in a computer's memory system, as shown next.

address bits

Consider a 64-bit desktop computer system that is capable of addressing up to 1 terabyte (TB) or 2^{40} bytes of RAM memory. Suppose an "entry-level" version of the system is built with only 4 gigabytes (GB) or 2^{32} bytes of memory, using four inexpensive modules of 1 GB (2^{30} bytes) each. Note that in a typical 64-bit computer, each memory module is actually 8 bytes wide; in this case, each 1-GB module stores 2^{27} 8-byte values sometimes known as longwords.

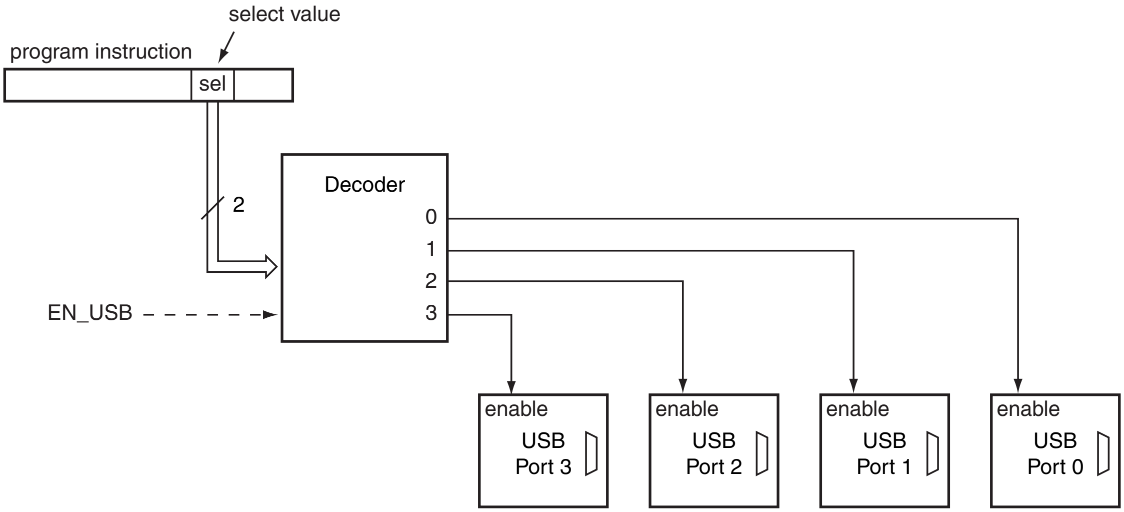


Figure 6-12 Typical decoder application in a computer.

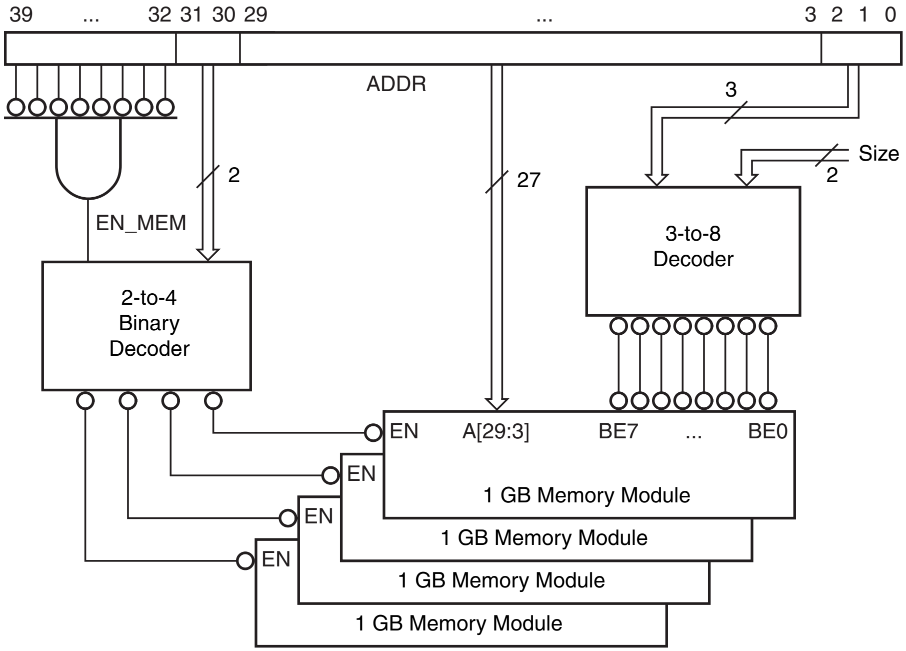


Figure 6-13 Memory-module decoding in a computer system.

Based on these considerations, the memory in our entry-level system may be addressed as shown in Figure 6-13. To select a location in the memory system, the computer processor supplies a 40-bit "physical address" ADDR shown at the top of the figure. The 8 high-order bits must be all 0s to select the lowest 4 GB of the total 1-TB physical-address space; an 8-input NOR gate asserts EN_MEM when this is true. The next two bits select which of the four modules contains the addressed longword. A 2-to-4 decoder is enabled by EN_MEM and decodes address bits 30 and 31 to assert one of its four outputs, each of which enables a corresponding 1-GB memory module. Note that the decoder has active-low outputs to match the modules' EN inputs. The next 27 address bits are connected to all four modules, to select the addressed longword within the enabled module.

Beyond this point, the design is a little more complicated than the previous example. For 1-byte operations, it is necessary to select which of the eight bytes within the selected longword is to be accessed. Each module has eight "Byte Enable" (BE) inputs for this purpose. So another circuit, a 3-to-8 decoder, is used to create the byte-enable inputs BE[7:0] which are connected to all modules to select a byte based on the three low-order bits of the supplied memory address.

But wait, there's more! In addition to longwords and single bytes, some computer instructions may access 16-bit halfwords or 32-bit words. Therefore, the computer processor provides (and the 3-to-8 decoder uses) two additional input bits to indicate the size of the operation: 00–11 for sizes 1, 2, 4, or 8 bytes, respectively. The decoder must assert multiple byte-enable outputs, depending on the size of the operation, and it must assert the appropriate ones depending on the three low-order address bits; for example, BE3 and BE2 for a 2-byte operation at any address ending in 010. So, this decoder is somewhat more complex than the 2-to-4 binary decoders in this and the previous example. Also, it has the special characteristic that multiple outputs may be asserted simultaneously.

*6.3.1 A More Mathy Decoder Definition

decoder

We can define decoders using the idea of "codes" that we introduced in Sections 2.10 through 2.13. In this definition, a *decoder* is any multiple-input, multiple-output combinational logic circuit that converts or "maps" an input code word into an output code word, where the input and output codes are different. With this definition, the general structure of a decoder circuit is shown in Figure 6-14. The enable inputs, if present, must be asserted for the decoder to perform its normal mapping function. Otherwise, the decoder maps all input code words into a single, "disabled," output code word.

The most commonly used input code is an n-bit binary code, where an n-bit word represents one of 2^n different coded values, normally the integers from 0 through 2^n-1, as in our USB and memory examples. Sometimes an n-bit binary code is truncated to represent fewer than 2^n values. For example, a computer with five USB ports might use a 3-bit "select" value, with binary values 001 through 101 for USB ports 1 through 5, and other values unused. In another example, the BCD code uses 4-bit combinations 0000 through 1001 to represent the decimal digits 0–9, and combinations 1010 through 1111 are not used.

The most commonly used output code is a 1-out-of-n code, which has n bits, one of which is asserted at any time, as in the 2-to-4 decoders in our USB and memory examples. Note that n need not be a power of 2, but often is. In a 1-out-of-4 code with active-high outputs, the normal code words are 0001, 0010, 0100, and 1000, with 0000 serving as the "disabled" code word. With active-low outputs, the code words are 1110, 1101, 1011, and 0111, with 1111 as the "disabled" code word.

6.3.2 Binary Decoders

A simple decoder, like the 2-to-4 binary decoder, is pretty easy to design at the gate level, which is what we'll look at here. Later, we'll go on to HDL-based models of both simple and more complex decoders.

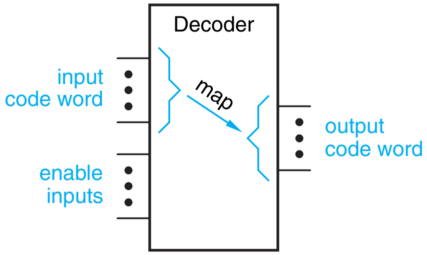


Figure 6-14 Decoder circuit structure.

Inputs			Outputs			
EN	A1	A0	Y3	Y2	Y1	Y0
0	x	x	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

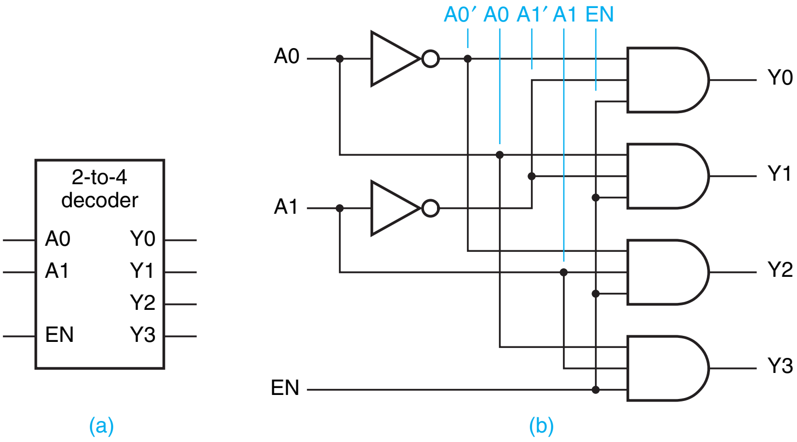
Table 6-3
Truth table for a 2-to-4 binary decoder.

The most common decoder circuit is an n-to-2^n binary decoder; it has an n-bit binary input code and a 1-out-of-2^n output code. Such a decoder is used when you need to activate exactly one of 2^n outputs based on an n-bit input value.

For example, Table 6-3 is the truth table, and Figure 6-15(a) shows the inputs and outputs, of a 2-to-4 decoder that could be used in our USB and memory examples. The input code word A1,A0 represents an integer in the range 0–3. The output code word Y3,Y2,Y1,Y0 has Yi equal to 1 if and only if the input code word is the binary representation of i and the enable input EN is 1. If EN is 0, then all of the outputs are 0. A gate-level circuit for the 2-to-4 decoder is shown in Figure 6-15(b). Expressions for the signals on each vertical line are shown in color at the top of the diagram; each is either an input signal or its complement. Each AND gate is said to *decode* one combination of the input code word A1,A0.

decode

The binary decoder's truth table introduces a "don't-care" notation for input combinations. If one or more input values do not affect the output values for some combination of the remaining inputs, they are marked with an "x" for that input combination, denoting "don't-care." This convention can often greatly



(a)

(b)

Figure 6-15
A 2-to-4 decoder:
(a) inputs and outputs;
(b) logic diagram.

Table 6-4 Truth table for a 74x138 3-to-8 decoder.

Inputs						***Outputs***							
G1	G2A_L	G2B_L	C	B	A	Y7_L	Y6_L	Y5_L	Y4_L	Y3_L	Y2_L	Y1_L	Y0_L
0	x	x	x	x	x	1	1	1	1	1	1	1	1
x	1	x	x	x	x	1	1	1	1	1	1	1	1
x	x	1	x	x	x	1	1	1	1	1	1	1	1
1	0	0	0	0	0	1	1	1	1	1	1	1	0
1	0	0	0	0	1	1	1	1	1	1	1	0	1
1	0	0	0	1	0	1	1	1	1	1	0	1	1
1	0	0	0	1	1	1	1	1	1	0	1	1	1
1	0	0	1	0	0	1	1	1	0	1	1	1	1
1	0	0	1	0	1	1	1	0	1	1	1	1	1
1	0	0	1	1	0	1	0	1	1	1	1	1	1
1	0	0	1	1	1	0	1	1	1	1	1	1	1

reduce the number of rows in the truth table, as well as make the functions of the inputs more clear.

Extrapolating from the 2-to-4 decoder's truth table and logic diagram, you can pretty easily create and understand binary decoders with more inputs and outputs, and in some cases different active levels for their signals. For example, Table 6-4 is the truth table for a 3-to-8 binary decoder with active-low outputs and three enable inputs, all of which must be asserted to enable the selected output. A circuit with this truth table was sold as a very popular MSI decoder part, the 74x138, and has the logic symbol shown in Figure 6-16(a) and the internal logic diagram in Figure 6-17. Note that the decoder's truth table matches the logic diagram and specifies the logic function in terms of the *external pins* of the device, that is, the signals in Figure 6-16(b). A truth table for the function performed *inside* the symbol outline would be different (see Drill 6.6).

FUNCTION VS. TRUTH TABLES

The truth tables in some manufacturers' data books use L and H to denote the input and output signal voltage levels, so there can be no ambiguity about the electrical function of the device; a truth table written this way is sometimes called a *function table*. However, since we use positive logic throughout this book, we can use 0 and 1 without ambiguity.

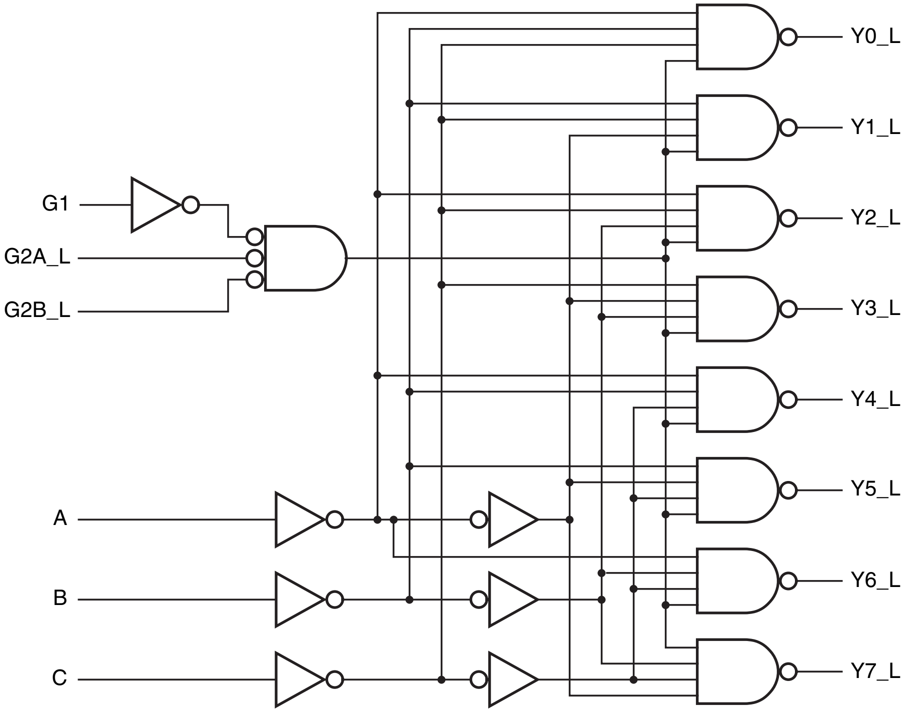


Figure 6-17 Logic diagram for the 74x138 3-to-8 decoder.

It is not necessary to use all of the outputs of a decoder, or even to decode all possible input combinations. For example, a *decimal* or *BCD decoder* decodes only the first ten binary input combinations 0000–1001 to produce outputs Y0–Y9.

decimal decoder

BCD decoder

Also, the input code of an *n*-bit binary decoder need not represent the integers from 0 through 2^n-1. For example, Table 6-5 shows the 3-bit Gray-code

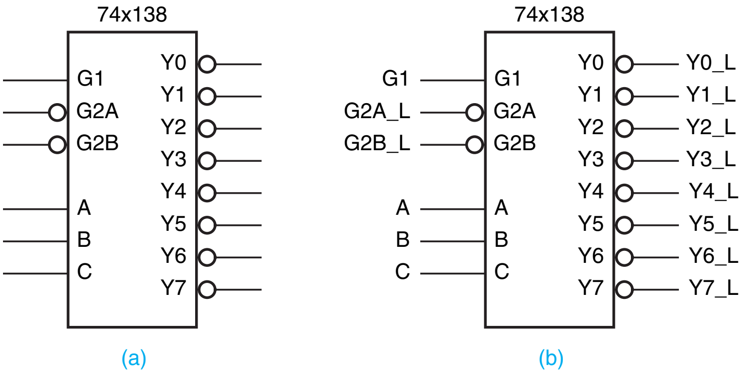
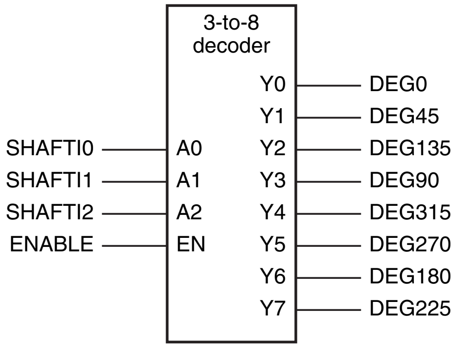


Figure 6-16 Logic symbol for the 74x138 3-to-8 decoder: (a) conventional symbol; (b) default signal names associated with external pins.

Table 6-5
Position encoding for a 3-bit mechanical encoding disk.

Disk Position	A2	A1	A0	***Binary Decoder Output***
0°	0	0	0	Y0
45°	0	0	1	Y1
90°	0	1	1	Y3
135°	0	1	0	Y2
180°	1	1	0	Y6
225°	1	1	1	Y7
270°	1	0	1	Y5
315°	1	0	0	Y4

Figure 6-18
Using a 3-to-8 binary decoder to decode a Gray code.



output of a mechanical encoding disk with eight positions, as in Figure 2-6 on page 61. The eight disk positions can be decoded with a 3-bit binary decoder by using the appropriate assignment of signals to the decoder outputs, as shown in Figure 6-18.

6.3.3 Larger Decoders

It's easy to write the logic equations for a binary decoder with any desired number of binary inputs and enable inputs. Each decoder output is simply the AND (perhaps inverted) of the enable inputs and the minterm for the decoded input combination. However, more inputs require wider AND gates, which typically cannot be realized in one "level" of transistors

cascading

It's also possible to design a decoder for more inputs by *cascading* (connecting in series) multiple small decoders. For example, see Figure 6-19 for a 5-to-32 decoder built from 3-to-8 decoders. The widest gates in each 3-to-8 decoder have only four inputs.

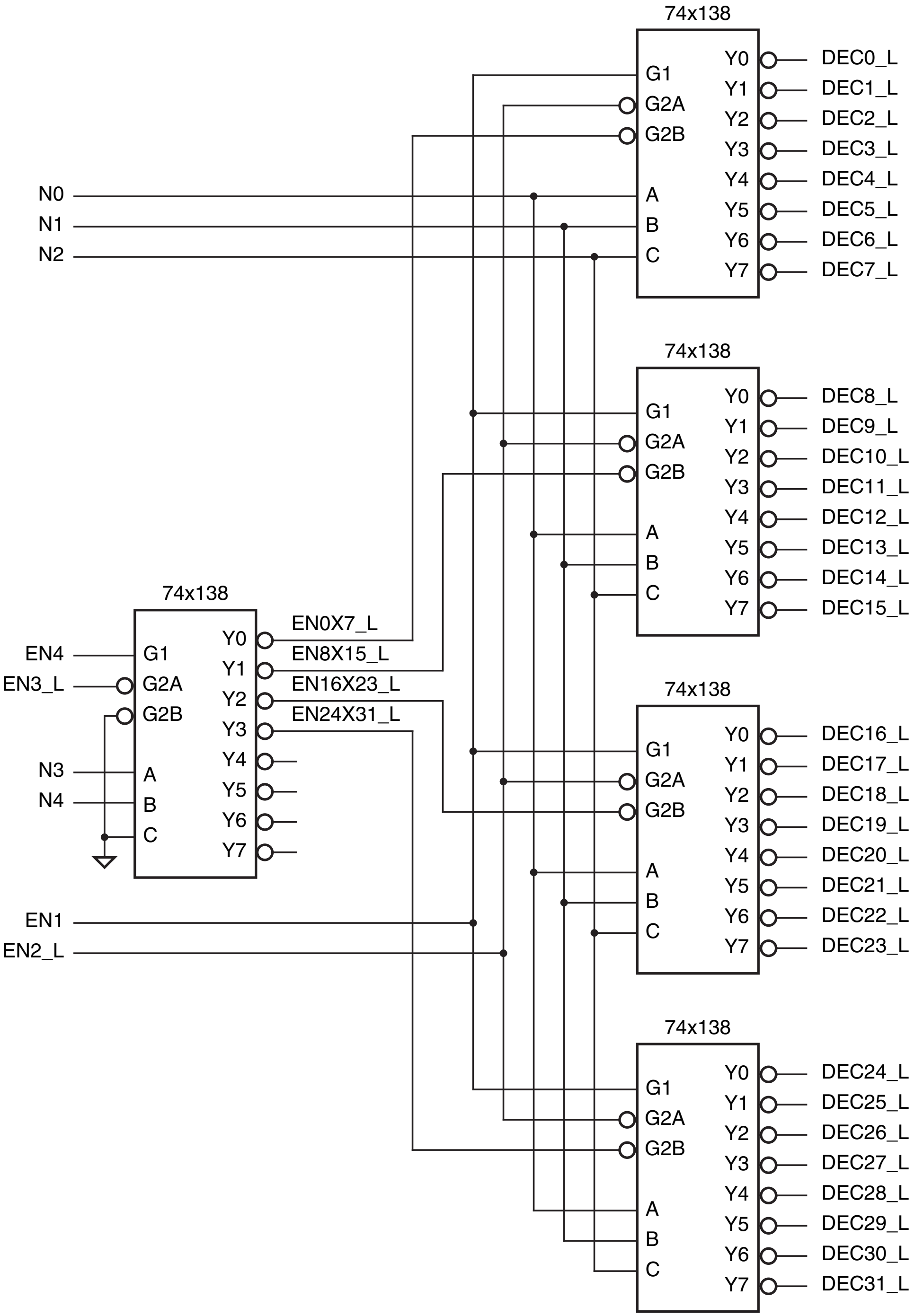


Figure 6-19
Cascading 3-to-8 decoders to make a 5-to-32 binary decoder.

In ASICs and custom VLSI, it's often necessary to build decoders with an even larger number of inputs and outputs. For example, the decoders in memory chips can easily have ten or more address inputs and over a thousand outputs. These applications use a method called *predecoding* to accommodate the target technology's practical limit on the number of gate inputs, while also optimizing the chip area used by the gates' transistors and the wiring that connects the gates to each other. Predecoding structures are also designed to minimize circuit delays, which are affected not only by the number of gates in a signal path, but also by the number of gate inputs that are connected to each gate output.

predecoding

Figure 6-20 shows a possible predecoding structure for a 6-to-64 binary decoder. The idea is to break up the overall address into two or more groups with an equal or near equal number of bits in each group, and decode those groups individually. The example has three 2-bit groups: A5–4, A3–2, and A1–0. Each n-bit group has its own first-level n-to-2^n binary decoder, so the example has three 2-to-4 decoders. The overall decoded outputs can then be obtained by using AND gates at the next level to combine appropriate predecoded signals, one from each predecoder, corresponding to the input combination for each overall decoded output.

For example, in Figure 6-20 you can see that the AND gate for output Y2 combines the predecoded outputs for A5–4=00, A3–2=00, and A1–0=10. The AND gate for output Y61 combines predecoded outputs for A5–4=11, A3–2=11, and A1–0=01.

Notice in the figure that the overall enable input EN is needed in only one of the 2-to-4 predecoders. So, the bottom predecoder uses 3-input AND gates internally, while the others need to use only 2-input AND gates. The maximum AND-gate width in the design is therefore 3.

As you know, inverting gates are generally faster than noninverting ones, so in Figure 6-20 you might consider using 2-to-4 predecoders with active-low outputs, and changing the output AND gates to NOR gates to compensate. However, also note that each predecoder output drives 16 inputs. In a chip design, the predecoder outputs would probably be faster if they were implemented with NAND gates followed by inverters sized appropriately to drive the 16 inputs and the long wires that connect to them.

Other predecoder structures for a 6-to-64 decoder, besides the one shown in Figure 6-20, are possible. For example, the input address could be divided into just two 3-bit groups, so two 3-to-8 predecoders would be used. This increases the width of the AND gates in the predecoders, and it increases the number of vertical interconnect lines from 12 to 16, neither of which is desirable. However, this alternative structure also halves the number of inputs driven by each predecoder output, and it saves an input on the final AND gates, which means in a chip design they are smaller and can be packed more tightly in the

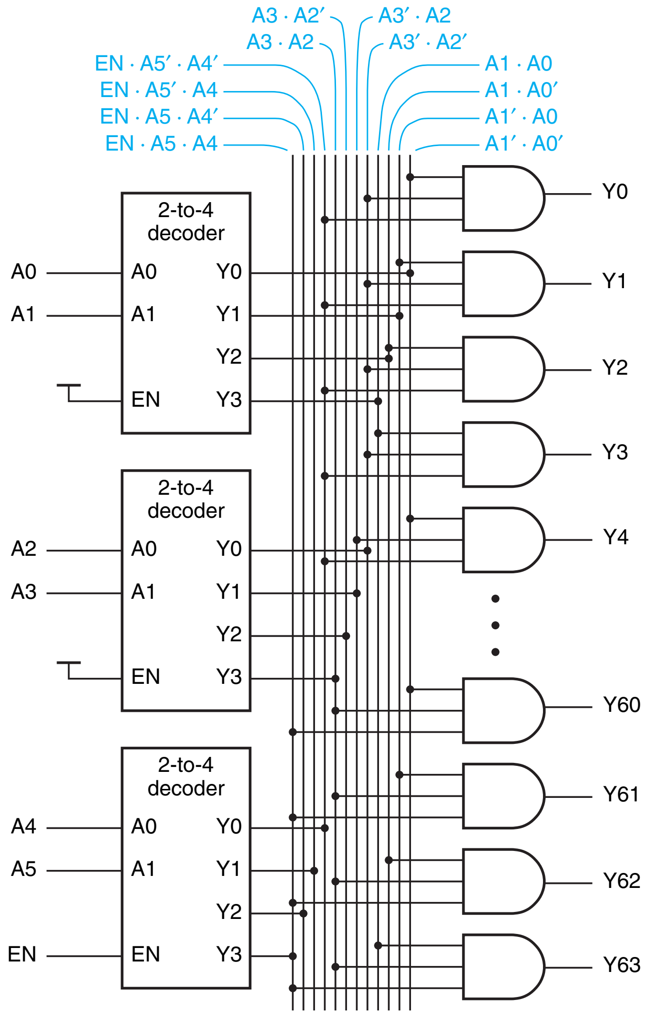


Figure 6-20
Predecoding structure for a 6-to-64 binary decoder.

vertical dimension. This alternative might better match the vertical dimension of the predecoders, yielding a smaller rectangular area for the overall decoder.

In larger decoders, the number of groups may be greater than the number of inputs on an AND gate, in which case the structure can be extended horizontally to additional levels (see Exercise 6.24). Based on the discussion here, you can understand how a chip designer would have a very rich set of alternatives and trade-offs to explore when designing a much larger decoder.

In the next subsection, we'll see how decoders can be modeled in Verilog, and then we'll look at some more complex decoder examples and applications, including the byte-enable decoder from our earlier memory example.

Program 6-1 Structural-style Verilog module for the decoder in Figure 6-15.

```
module Vr2to4dec_s(A0, A1, EN, Y0, Y1, Y2, Y3);
  input A0, A1, EN;
  output Y0, Y1, Y2, Y3;
  wire NOTA0, NOTA1;

  not U1 (NOTA0, A0);
  not U2 (NOTA1, A1);
  and U3 (Y0, NOTA0, NOTA1, EN);
  and U4 (Y1,    A0, NOTA1, EN);
  and U5 (Y2, NOTA0,    A1, EN);
  and U6 (Y3,    A0,    A1, EN);
endmodule
```

6.3.4 Decoders in Verilog

Decoding-type logic is often incorporated in a larger Verilog module, along with other functions. However, instead of dealing with large, complex examples, in this subsection we'll show how to define and test standalone decoder modules with explicit decoded outputs, as in the gate-level building-block examples of the previous subsection.

There are several ways to model standalone decoders in Verilog, the most primitive being to write a structural equivalent of a gate-level decoder circuit. Just for practice in structural Verilog modeling, this is done in Program 6-1 for the 2-to-4 binary decoder of Figure 6-15 on page 253. The design uses Verilog's built-in components not and and. The port list of each component begins with its output, and is followed by its one or more inputs. Figure 6-21 is a logic diagram of the circuit as it is modeled in the Verilog module.

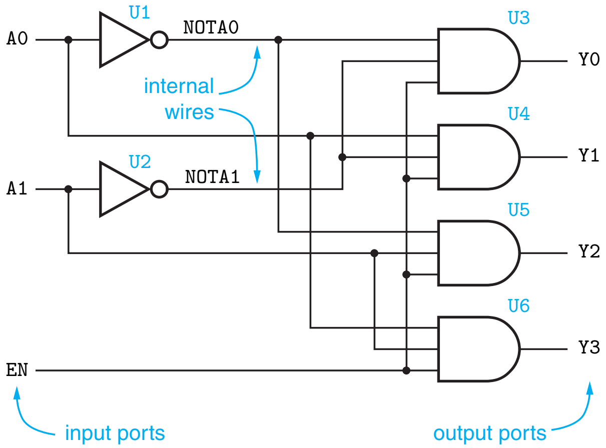


Figure 6-21 Logic diagram for structural model of 2-to-4 decoder.

The approach just shown, designing a gate-level logic diagram and then mechanically converting it into the equivalent of a netlist, would usually defeat the purpose of using Verilog to create a design that's understandable and maintainable. Probably the only reason to use the structural approach for a decoder

JUST-IN-TIME VERILOG FOR PROGRAM 6-1

In case you haven't studied Chapter 5, this box and others with similar titles introduce Verilog concepts the first time they are used in this chapter. You can find more details in Chapter 5.

module

A Verilog model begins with the keyword `module` and a parenthesized list of the module's "port" names, which are just signal names. Note that case is significant in Verilog names, and only letters, digits, and special characters _ and $ are allowed.

input
output

Verilog has a couple of syntax styles for declaring the type of each module port, but in this book we mostly use the syntax where each signal type is defined in an input or output declaration at the beginning of the module. Input ports are defined using the `input` keyword followed by optional size and net-type specifications and the names of the ports being defined; likewise for output ports, using the `output` keyword. The input and output declarations in this example have no optional specifications, so the signals have the default size and type, "1-bit wire."

wire

A Verilog "wire" corresponds to a signal wire in a physical circuit, and provides one bit of connectivity between modules or between modeled elements inside a module. For the latter case, local signals are declared using the `wire` keyword as in the example. All local names declared within a module have a limited scope, local to the module, as in most programming languages.

0, *1*, *z*, *x*

Verilog uses a four-valued logic system; a 1-bit signal like a wire can be only `0`, `1`, `z` (high-impedance), or `x` (unknown, meaningful only in simulation).

Declarations and statements in Verilog must be terminated with a semicolon, unless they end with a terminating keyword that has the semicolon "built in," like `end`, `endcase`, or the like, which we'll see later.

not, *and*, *nand*
or, *nor*, *xor*, *xnor*

Verilog supports a few different styles of modeling logic, the first of which is structural. Using text-based statements, the language specifies the wired interconnections among primitive gates as well as larger elements. Several gates types are built in and have corresponding keywords, including `not`, `and`, `nand`, `or`, `nor`, `xor`, and `xnor`.

The first statement in the body of Program 6-1 "instantiates" a built-in logic gate by listing its corresponding keyword, `not`, followed by a reference designator selected by the designer, followed by a parenthesized list containing the signal names of its output wire and its one or more inputs, in that order.

The second statement instantiates another `not` gate, and the remaining four instantiate 3-input `and` gates, each one listing the output wire name followed by three input-signal names. Together, these six statements model the gate components and their wired interconnections shown in Figure 6-15 on page 253.

endmodule

A module ends with the `endmodule` keyword.

CASCADING AND PREDECODING

We showed an example of cascading decoders in Figure 6-19, and predecoding in Figure 6-20. When such a decoder is designed for and targeted to an ASIC or a custom VLSI chip, a Verilog model can be created to mimic its structure. The outputs of this structure can then be checked in a test bench, either algorithmically or by comparing against the outputs of a simpler behavioral design, to ensure that are no errors in the hookups specified in the structure, as explored in Exercises 6.23, 6.25, and 6.26.

would be to model and test a large structure prior to eventual implementation in an ASIC using cascading or predecoding.

Instead of using the structural style, we would normally create a Verilog module that makes our decoder design more understandable and maintainable. One possibility is to use the dataflow style of Verilog to show what's going on, as in Program 6-2. Here, we have used a continuous assignment statement for each output wire, setting it to 0 if `EN` is 0. If `EN` is 1, the output's value is determined by a conditional expression that is 1 only if the current value of the 2-bit vector `{A1,A0}` selects that output.

Another way to specify the decoder, and arguably the most readable and maintainable, is to use the behavioral style of Verilog. Actually, there are several different ways that Verilog can model the decoder's behavior. One way is shown

JUST-IN-TIME VERILOG FOR PROGRAM 6-2, PART 1

This model has a different module name but the same input and output names as the first, so it should be possible to use it anywhere that the first one is used. Of course, it's up to the designer to ensure that its function is the same, or at least correct.

As in the first model, the inputs and outputs are declared as 1-bit wires, the default. However, no additional, internal wires are defined, since none are used.

assign

This module uses the "dataflow" style of modeling. The value of each combinational output is specified by a "continuous assignment statement," introduced by the keyword `assign`. The statement gets its name because in the model, and in any synthesized circuit derived from the model, the value on the righthand side of the = sign is continuously assigned to the signal named on the left. That is, any time that anything changes on the righthand side, the signal on the lefthand side may respond and change immediately, subject to propagation delays in the real circuit, or in simulation subject to delays specified in the model (none in this example).

Note that continuous assignment statements "execute" concurrently. In a corresponding physical circuit, all of the gates or other components that calculate the righthand side operate in parallel. And a simulator performs all of the assignments at the same simulated time.

Program 6-2 Dataflow-style Verilog module for a 2-to-4 binary decoder.

```
module Vr2to4dec_d(A0, A1, EN, Y0, Y1, Y2, Y3);
  input A0, A1, EN;
  output Y0, Y1, Y2, Y3;

  assign Y0 = EN ? ({A1,A0}==2'b00) : 0;
  assign Y1 = EN ? ({A1,A0}==2'b01) : 0;
  assign Y2 = EN ? ({A1,A0}==2'b10) : 0;
  assign Y3 = EN ? ({A1,A0}==2'b11) : 0;
endmodule
```

JUST-IN-TIME VERILOG FOR PROGRAM 6-2, PART 2

?:

Now we can talk about what's actually being calculated on the righthand side of each continuous assignment statement in Program 6-2, using Verilog’s so-called “conditional operator” which has the syntax *logical-expr* ? *T-expr* : *F-expr*. This operator produces a result equal to either *T-expr* or *F-expr* depending on whether *logical-expr* is true or false. In this example, *logical-expr* is just one signal wire which has a value of 0 or 1. In Verilog, a 1-bit value of 1 is “true,” and 0 is “false.”

The *T-expr* and *F-expr* on either side of the : must produce values that match or are at least compatible with the signal on the lefthand side of the = sign. In this example, the *F-expr* in each statement is simply 0, which is an integer constant, and it’s being assigned to a 1-bit wire. When an integer is assigned to or compared with a wire, Verilog uses the LSB of the integer, which in this case is of course 0.

The *T-expr* in this example is a parenthesized logical expression, which has a value of true or false. In Verilog, when a logical value is assigned to or compared with a wire, Verilog uses a 1-bit value of 1 for true, and 0 for false.

vectors

{}

The parenthesized logical expression introduces a couple more new things. Verilog supports vectors, which are one-dimensional arrays of 1-bit elements, like wires. Braces may be used to concatenate multiple bits into a vector, so `{A1,A0}` is a 2-bit vector with MSB equal to `A1` and LSB equal to `A0`.

vector literals

Verilog also supports vector literals, indicated by `'`. In the example, 2 is the number of bits in the literal, `'` indicates the literal, and `b` is the base of the digits that follow, binary in this example. So the literals in the four statements are 2-bit vectors with binary values of 00 through 11. Other bases can be used for digits, so the literal `2'b11` could also be written `2'd3` (decimal), `2'h3` (hexadecimal), or `2'o3` (octal). Regardless of the base, the leading number (`2`) is always the number of bits, and is always written in decimal.

The parenthesized logical expression in each statement is an equality comparison. As in C and some other languages, the `==` operator performs an equality comparison. In each statement, the comparison is a 2-bit vector `{A1,A0}` corresponding to the decoder's address inputs and the righthand side is a 2-bit literal. The assignment statement sets each output signal `Yi` to 1 if the address inputs match the corresponding output-signal number `i`, else 0.

Program 6-3 Behavioral-style Verilog module for a 2-to-4 binary decoder.

```
module Vr2to4dec_b1(A0, A1, EN, Y0, Y1, Y2, Y3);
  input A0, A1, EN;
  output reg Y0, Y1, Y2, Y3;

  always @ (A0, A1, EN)
    if (EN==1)
      {Y3,Y2,Y1,Y0} = 4'b0000;
    else
      case ({A1,A0})
        2'b00: {Y3,Y2,Y1,Y0} = 4'b0001;
        2'b01: {Y3,Y2,Y1,Y0} = 4'b0010;
        2'b10: {Y3,Y2,Y1,Y0} = 4'b0100;
        2'b11: {Y3,Y2,Y1,Y0} = 4'b1000;
        default: {Y3,Y2,Y1,Y0} = 4'b0000;
      endcase
endmodule
```

in Program 6-3. Here, we use an `always` statement whose sensitivity list has all of the decoder's inputs. Notice that the output variables are now declared to have type `reg` so their values can be assigned in procedural statements. An `if` statement tests the enable input. If `EN` is 0, then all of the outputs are set to 0. When `EN` is asserted, the decoder's functionality translates very nicely into HDL code that activates one output based on the current input combination: a `case` statement checks the address inputs and assigns the output values accordingly.

The `default` choice is included in the `case` statement to handle the possibility of `A0` or `A1` being `x` or `z` in simulation; it may be advisable to set the outputs to `4'bxxxx` in this case to propagate the error.

The procedural statements in Program 6-3 in some ways simply mimic the truth table for the decoder. To model a larger decoder with this approach, we would have to write more cases and longer literals for the assignments, which would be error-prone. We can capture the decoder's behavior more succinctly using the approach shown in Program 6-4. While it's a simple module, several aspects are worth noting:

A reg IS NOT A REGISTER

In Program 6-3 and others, outputs like `Y0` are declared as `reg` variables so their values can be set procedurally within an `always` block. But keep in mind that despite the name, a Verilog `reg` declaration does *not* create a hardware register (a set of flip-flops for storage). It simply creates an internal variable used by the simulator and the synthesizer. Mechanisms for creating actual flip-flops in Verilog modules will be discussed in Section 10.3.

JUST-IN-TIME VERILOG FOR PROGRAM 6-3

The behavioral style of Verilog modeling uses “procedural code” to define logical behaviors that may be later synthesized into hardware. A key requirement of logical modeling for synthesis is to use code “templates” that the compiler knows how to translate into RTL structures for subsequent targeting into real hardware.

procedural code variables

A very important concept for understanding Verilog behavioral modeling is the use of `reg` variables. Only `reg` variables, not wires, can be assigned values in procedural code. Despite the poorly named keyword, a `reg` is not a hardware register! A `reg` is a software *variable* that is used within a module, and it is assigned values by statements in procedural code. A `reg` may or may not have physical significance in a circuit depending on how it is used in the Verilog module.

`reg`

In the model in Program 6-3, the keyword `reg` in the `output` declaration indicates that the named outputs are to be used as `reg` variables within the module. However, the current value of each such `reg` variable in the module is continuously assigned to the named output-port signal, a wire for connection to other modules. Local `reg` variables can also be declared for use inside the module only, but there are none in this example.

`always`
sensitivity list

The `always` keyword introduces procedural code. It is followed by a parenthesized list of signal names, called the “sensitivity list.” If the value of one or more of the listed signals changes, the statement following the list is executed, in zero simulated time. Hardware synthesized from the model also mimics this behavior. If execution of the statement causes a further change in a listed signal, the statement is run again, still in zero simulated time. Execution continues until all of the listed signals have stabilized, but will happen again if any listed signal changes later.

`if`
`else`

In the example, the procedural statement following the sensitivity list is an `if-else` statement, introduced by the `if` keyword. It tests a parenthesized logical expression, which is followed by a statement that is then executed if the expression was true. If the optional `else` keyword is present, then the next statement is executed if the logical expression was false.

In the example, if `EN` is 0, then the output variables `{Y3,Y2,Y1,Y0}` (another concatenation) are set to constant 0s. Otherwise, the “case” statement that follows is executed.

`case`
`default`

The keyword `case` introduces a Verilog case statement. It is followed by a parenthesized “selection expression,” whose value is an integer or a vector in most uses of this statement. Next is a sequence of case items, five in the example. Each case item begins with a “choice” followed by a colon and a single procedural statement. The `case` statement finds the first choice whose value matches that of the selection expression and executes the corresponding procedural statement. If no match is found, then the optional `default` choice is executed, if present.

In the example, the selection expression is a 2-bit vector `{A1,A0}` and the case items enumerate all four of its possible binary values, setting the output variables `{Y3,Y2,Y1,Y0}` to a corresponding constant bit pattern in each case.

`endcase`

The `case` statement ends with the `endcase` keyword, which has a “built-in” statement-terminating semicolon. The module ends with `endmodule` as usual.

Program 6-4 Another behavioral-style module for the 2-to-4 binary decoder.

```
module Vr2to4dec_b2(A0, A1, EN, Y0, Y1, Y2, Y3);
  input A0, A1, EN;
  output reg Y0, Y1, Y2, Y3;
  reg [3:0] IY;
  integer i;

  always @ (A0 or A1 or EN) begin
    IY = 4'b0000;            // Default, outputs all 0
    if (EN==1)               // If enabled...
      for (i=0; i<=3; i=i+1) // set output bit i where i={A1,A0}
        if (i == {A1,A0}) IY[i] = 1;
    {Y3,Y2,Y1,Y0} = IY;      // Copy internal variable to outputs
  end
endmodule
```

- An “internal” version of the outputs, IY, is declared as a 4-bit vector `reg` variable to facilitate setting an individual, numbered bit selected by an integer variable `i` in the code.
- Although the code may look very “sequential,” with `IY` being initialized and possibly then having one of its bits set later in the `for` loop, this all happens in zero simulated time when it’s simulated.
- Likewise, in synthesis, the `for` loop is merely an instruction to the tool to synthesize a combinational logic structure that compares `{A1,A0}` against each of the four possible values of `i` in the loop, and sets bit `i` of `IY` upon a match. Think of it as the combinational logic equations for the `IY[i]` outputs being *specified* sequentially.

JUST FOR VARIETY

There are many different stylistic choices that can be made in Verilog coding, everything from indentation and spacing to syntax for constants. Sometimes a particular style is required by an employer, just to maintain consistency among design teams. Still, we’ll use a few different styles in different examples in this book, just to expose you to and remind you of the different syntactic options that are available.

For example, in this section, you’ll sometimes see a sensitivity list with individual signal names separated by “`or`” or a comma, and sometimes see the wildcard “`*`” instead, which means “all signals that might affect this block.”

You’ll sometimes see a 1-bit logic 1 value written very precisely as “`1'b1`,” and sometimes as simply “`1`,” which is technically an integer constant but which the Verilog compiler will interpret as “`1'b1`” when it matches it up with any 1-bit signal or variable that it is comparing it with or assigning it to.

And you’ll see the “choices” of a `case` statement written as literals in either binary or decimal as in Programs 6-3 and 6-12, respectively, but never as integers, to avoid the problem explained in the box on page 215.

JUST-IN-TIME VERILOG FOR PROGRAM 6-4

This module has declarations similar to those of the first behavioral module, but it also declares an internal `reg` variable `IY`, a vector of four bits that's ultimately copied to the four output-port bits. As specified by "`[3:0]`" in the declaration, the vector's elements are numbered from 3 down to 0 from left to right. Vector elements can also be numbered in ascending order, and any starting and ending indices may be used.

[:], vectors

The module declares `reg` variable `IY` is so that the decoder's behavior can be specified later by certain vector operations which aren't available for the individually named output-port bits. The module also declares an integer `i` that it uses to control a `for` loop, as explained shortly.

The sensitivity list in the `always` statement uses the `or` keyword as the separator, instead of a comma. This has nothing to do with an OR function or Verilog's built-in `or` component. It's just optional syntax from the original Verilog-1995.

Like many other languages, Verilog supports block-structured coding, where a list of statements may be used in the place of a single statement. In Verilog, a block begins with the keyword `begin`, contains a list of procedural statements, and ends with the keyword `end`. The block's procedural statements are executed sequentially, in order. In the example, the `begin-end` block is treated as the single procedural statement that follows the `always` statement, creating an "`always` block".

begin
end
always block

This example introduces Verilog's `for` loop, which begins with the keyword `for`. Next comes a parenthesized list with three elements that manipulate a *loop-index* variable, typically an integer (`i` in this example), to control the looping behavior. The first element assigns an initial value to *loop-index*; the second is a logical expression that is evaluated prior to executing the body of the loop and must be true for execution to proceed; and the third assigns a next value to *loop-index* each time after the body is executed. The body of the loop is a single procedural statement, which in this example is an `if` statement, used here without an `else` clause.

for

Another Verilog feature that appears for the first time in this module is the use of brackets `[]` to select one bit of a vector as specified by a "bit-select." The bit-select is an expression whose value is an integer or can be converted to one. The integer value of course denotes the bit number to be selected. We could also select a range of bits, by instead using a "part-select" which is two integer values separated by a colon; these denote the starting and ending indices of a contiguous group of bits within the vector, as we'll see in later examples.

[], bit-select
[:], part-select

- The last statement in the `always` block assigns `IY` to the module's output variables. We could have avoided this extra work (and the concatenation `{A1,A0}`) if we had declared the module's inputs and outputs in the first place as a vectors `A[1:0]` and `Y[3:0]`, but we'll do that later in another example.

Yet another behavioral model for the decoder is shown in Program 6-5. This is the most succinct version of all. After initializing the output bits to all 0s, it simply sets the bit of `IY` with index `{A1,A0}` to 1.

Program 6-5 Yet another behavioral-style module for the 2-to-4 decoder.

```
module Vr2to4dec_b3(A0, A1, EN, Y0, Y1, Y2, Y3);
  input A0, A1, EN;
  output reg Y0, Y1, Y2, Y3;
  reg [3:0] IY;

  always @ (A0, A1, EN) begin
    IY = 4'b0000;               // Default, outputs all 0
    if (EN==1) IY[{A1,A0}] = 1; // Set selected output if enabled
    {Y3,Y2,Y1,Y0} = IY;         // Copy internal var to output
  end
endmodule
```

JUST-IN-TIME VERILOG FOR PROGRAM 6-5

There's nothing really new in this module, but something is used in a slightly tricky way; can you spot it? Remember, the syntax `IY[`*p*`]` is looking for a "part-select" *p* which specifies part of a vector. In the present example, the concatenation `{A1,A0}` is a 2-bit vector that the compiler treats as an integer for the part-select. So, the expression `IY[{A1,A0}]` denotes bit `{A1,A0}` of the vector `IY`.

Program 6-6 Test bench for a 2-to-4 decoder.

```
`timescale 1 ns / 100 ps
module Vr2to4dec_tb () ;
  reg A0s, A1s, ENs;
  wire Y0s, Y1s, Y2s, Y3s;
  integer i, errors;
  reg [3:0] expectY;

  Vr2to4dec_s UUT ( .A0(A0s),.A1(A1s),.EN(ENs), // Instantiate unit under test (UUT)
                    .Y0(Y0s),.Y1(Y1s),.Y2(Y2s),.Y3(Y3s) );
  initial begin
    errors = 0;
    for (i=0; i<=7; i=i+1) begin
      {ENs, A1s, A0s} = i;                    // Apply test input combination
      #10 ;
      expectY = 4'b0000;                      // Expect no outputs asserted if EN = 0
      if (ENs==1) expectY[{A1s,A0s}] = 1'b1; // Else output {A1,A0} should be asserted
      if ({Y3s,Y2s,Y1s,Y0s} !== expectY) begin
        $display("Error: EN A1A0 = %b %b%b, Y3Y2Y1Y0 = %b%b%b%b",
                ENs, A1s, A0s, Y3s, Y2s, Y1s, Y0s);
        errors = errors + 1;
      end
    end
    $display("Test complete, %d errors",errors);
  end
endmodule
```

Even though the decoder designs are very simple, we should write a test bench to make sure they're right. Program 6-6 is a self-checking test bench that does the job. With only three inputs, the decoder has only eight different input combinations, and the test bench uses a variable `i` to step through all of them. However, rather than check the decoder's `Y` outputs against its truth table in Table 6-3 on page 253, the test bench instead embeds the functionality of the decoder's enable and address inputs in its `if` statements. This serves as a sort of "two-way check" on the designer's logic in creating both the decoder and its test bench. Since all five of our decoder modules have the same inputs, outputs, and functionality, this test bench can be used with any of them by changing just one line of code—the one that instantiates the UUT.

JUST-IN-TIME VERILOG FOR PROGRAM 6-6, PART 1

A Verilog test bench does not model hardware. Rather, it is a program that a simulator executes to apply inputs to and observe outputs of a hardware model, often called the "unit under test" (UUT). A test bench normally has no inputs and outputs per se, hence the null list following the module name.

The test bench instantiates the UUT by listing its name, a designer-selected reference designator (`UUT` in the example), and a list of input/output associations. Each association has a dot, followed by the name of the UUT input or output signal, followed by parentheses enclosing the name of the local signal that should be "connected" to that UUT input or output. UUT outputs must be connected to signals with type `wire`, hence the declaration of `Y1s`–`Y4s` as that type.

The first declaration in the test bench declares three `reg` variables `A0s`, `A1s`, and `ENs` which are used as the UUT inputs. The test bench assigns values to these variables and hence to the UUT inputs using procedural code.

`initial`

The `initial` keyword introduces procedural code that is executed by the simulator once, beginning when the module begins at time zero. It is followed by one procedural statement, usually a `begin-end` block, creating an "`initial` block."

The main body of the initial block sets an error count to 0, and then a `for` loop executes a `begin-end` block on each iteration. The block's first statement assigns values to the UUT's inputs. Its lefthand side is a 3-bit vector, and the righthand side is an integer. In the case of such a "mismatched" assignment, Verilog truncates the value of the integer, using as many bits as needed starting from the LSB to match the vector length. Thus, the assigned values will range from 000 to 111 binary.

`` `timescale ``

The next statement, "`#10;`", instructs the simulator to delay simulated time by 10 units, the unit being the first listed value (`1 ns`) in the `` `timescale `` directive at the beginning of the module. A new input will be applied to the UUT every 10 ns.

The next statement sets `expectY` to all 0s, and if `ENs` is 1, the `if` statement sets the `expectY` bit selected by `{A1s,A0s}` to 1. The second `if` statement compares the output values `{Y3s,Y2s,Y1s,Y0s}` with what's expected. A fine point is that this comparison uses the "case inequality operator" `!==`, which properly handles `x` or `z` values that might occur on the `Yi` outputs in simulation. If there is any mismatch, the `begin-end` block is executed to display the error and increment the error count.

JUST-IN-TIME VERILOG FOR PROGRAM 6-6, PART 2
$display

Errors are displayed by the built-in `$display` system task, which displays a line of text (terminated by a "newline") on the system console. The syntax of its arguments is similar to that of formatted I/O in C. The first argument is a text formatting string (delimited by `"`) that specifies what is to be printed. Within that string, each %*f* is a placeholder for another argument that is to be printed in the format specified by a letter *f*, where `b` means binary. Other options include `d`, `h`, and `o`. After the formatting string, the current values of the additional arguments are substituted, in order, for the %*f* placeholders. The number of placeholders and additional arguments must match.

The last statement of the `initial` block announces the test completion and displays the number of errors found. The module ends with `endmodule` as usual.

In a simple design like this, it is quite possible that the test bench will find no errors, even on its very first run. However, it is always possible and advisable to insert an error or two in the UUT just to make sure that the test bench is really able to detect errors. Drill 6.8 is a bit of a riddle on how to very easily "test the test bench" in this example.

When a module may be reused in different designs, it may make sense to parameterize it so key characteristics can be readily changed without rewriting the whole thing. In the case of the decoder, the number of address bits and the number of output bits are key. Based on our preceding behavioral design, Program 6-7 shows an n-to-s binary decoder, where n is the number of address bits and s is the number of output bits, usually 2^n. In this version, we have declared both `A` and `Y` as vectors to make it easy to parameterize the code, with a default of 3 bits for `A` and 8 bits for `Y`. This also simplifies the code, since we no longer need the temporary variable `IY`. A couple of other things are noteworthy:

- We initialize `Y` to 0, which is an integer constant. However, the compiler, as usual, converts this into a bit vector, extending with 0 bits on the left to match the width of the vector it's assigned to, `Y`. This shortcut wouldn't work if we were trying to initialize to all 1s.

Program 6-7 Parameterized N-to-S binary decoder module.

```
module VrNtoSbindec(A, EN, Y);
parameter N=3, S=8;
  input [N-1:0] A;
  input EN;
  output reg [S-1:0] Y;

  always @ (*) begin
    Y = 0;                  // Default, outputs all 0
    if (EN==1) Y[A] = 1;    // Set selected output bit if enabled
  end
endmodule
```

JUST-IN-TIME VERILOG FOR PROGRAMS 6-7 AND 6-8

parameter
#

A declaration introduced by the keyword parameter assigns constant values to named parameters for later use in the module. Multiple assignments, separated by commas, can be made in one declaration. New constant values may be assigned to parameters in a module when it is instantiated in a test bench or other module, similar to the way that signals themselves are assigned. After the module name, a # sign is followed by a parenthesized list of parameter names and their new values.

*
**

In Program 6-7, the wildcard "*" in the sensitivity list means "all signals that might affect this always block." This puts the burden on the compiler to figure out which signal changes should cause the block to be re-executed. In Program 6-8, exponentiation in the "control" part of the for statement is denoted by "**".

- As before, everything happens in zero simulated time. Even though a value may be assigned to Y in two places, only the final value shows up in the simulated or synthesized circuit output.

We can also write a new test bench, based on the previous one, to test the parameterized binary decoder, as shown in Program 6-8. Notice that it passes its own parameter values, which happen to be the same but could be changed, to the

Program 6-8 Parameterized test bench for the *n*-to-*s*-bit binary decoder module.

```
`timescale 1 ns / 100 ps
module VrNtoSbindec_tb () ;
parameter N=3, S=8;
  reg [N-1:0] A;
  reg EN;
  wire [S-1:0] Y;
  integer i, errors;
  reg [S-1:0] expectY;

  VrNtoSbindec #(.N(N),.S(S)) UUT ( .A(A),.EN(EN),.Y(Y) );    // Instantiate the UUT
  initial begin
    errors = 0;
    for (i=0; i<(2**(N+1)); i=i+1) begin
      {EN, A} = i;                            // Apply test input combination
      #10 ;
      expectY = 0;                            // Expect no outputs asserted if EN = 0
      if (EN==1) expectY[A] = 1'b1;           // Else output A should be asserted
      if (Y !== expectY) begin
        $display("Error: EN A = %b %b, Y = %b", EN, A, Y);
        errors = errors + 1;
      end
    end
    $display("Test complete, %d errors",errors);
  end
endmodule
```

AN UNEXPECTED NON-BUG

The code in Program 6-7 works even if `S` is not the power of 2 corresponding to `N`. For example, suppose you set `S` to 6 to get a 3-to-6 decoder, with outputs `Y[5:0]` and with address-input combinations 110 and 111 selecting nothing. It would appear that when `A` is 110 or 111, the code has an error because it attempts to set a nonexistent bit of `Y[A]` to 1. However, the Verilog language reference manual (LRM) is clear that out-of-range assignments in vectors are simply ignored (see box on page 189). The test bench in Program 6-8 works properly in this case for the same reason.

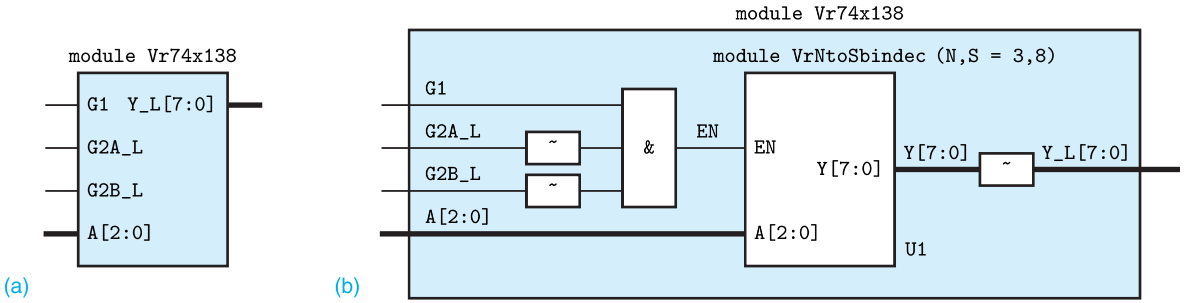


Figure 6-22 Verilog module `Vr74x138`: (a) top level; (b) internal structure with `VrNtoSbindec`.

decoder module when it instantiates it as the UUT. Also notice the use of the parameter `N` in computing the bounds of the `for` loop, whose body is executed 2^{N+1} times. Finally, notice that we used the same names for signals in the test bench as in the UUT—because of the scope rules, the compiler keeps everything straight. We do that in many other test benches, but the choice is up to you.

Once a "generic" module like `VrNtoSbindec` has been defined, it can be used as a building block in other designs. For example, suppose we needed a 3-to-8 decoder module with functionality like that of the 74x138 MSI part—two active-low and one active-high enable inputs, and active-low outputs. Such a module `Vr74x138` can be defined hierarchically based on `VrNtoSbindec`. The hierarchical relationship between the modules is shown in Figure 6-22, and the

Program 6-9 Hierarchically defined 74x138-like 3-to-8 decoder.

```
module Vr74x138(G1, G2A_L, G2B_L, A, Y_L);
  input G1, G2A_L, G2B_L;
  input [2:0] A;
  output [7:0] Y_L;
  wire [7:0] Y;

  assign EN = G1 & ~G2A_L & ~G1A_L;  // Convert, combine enables
  assign Y_L = ~Y;                   // Convert outputs
  VrNtoSbindec #(.N(3),.S(8)) U1 (.EN(EN),.A(A),.Y(Y));
endmodule
```

MATCHING UP NAMES

In Figure 6-22, the port names of a module are drawn inside the corresponding box. The names of the signals that are connected to the ports when the module is used are drawn on the signal lines. Notice that the signal names may match, but they don't have to. The Verilog compiler keeps everything straight, associating a scope with each name. The situation is completely analogous to the way variable and parameter names are handled in hierarchically structured, procedural programming languages like C.

corresponding Verilog code is shown in Program 6-9. The top-level module, `Vr74x138`, instantiates `VrNtoSbindec`, specifying both the signals that are to be connected to its input and output ports and the constant values to be assigned to its parameters. The top-level module also has continuous-assignment statements to combine the enable signals and to perform active-level conversions as needed.

6.3.5 Custom Decoders

Decoders can be customized in many different ways. In HDL-based design, such customization would normally be done in the context of a larger module design where decoding functionality is included among other things. Customizations are usually easy to do, and may include any of the following:

- Having different numbers of inputs and data outputs, in some cases fewer than 2^n data outputs and with different unused address-input combinations.
- Having active-low inputs (especially enables) or outputs.
- Asserting an output for two or more address-input combinations.
- Asserting multiple outputs for an input combination.

An interesting example is the memory-module decoding arrangement that we described in connection with Figure 6-13 on page 251. Before proceeding with the Verilog design, let us make one simplifying assumption: when the operation size is larger than a byte, the operand's address will be "aligned" on a boundary corresponding to that size. That is, operations on halfwords, words, and longwords will have addresses that are multiples of 2, 4, and 8, respectively.

JUST-IN-TIME VERILOG FOR PROGRAM 6-9

~, &, |

For the first time in our examples, Program 6-9 uses Verilog "bitwise boolean" operators to combine signals. The NOT, AND, and OR operations for individual bits or multibit vectors are denoted by symbols `~`, `&`, and `|`, respectively. When applied to vectors, these operations act bit-by-bit in corresponding positions.

Verilog has a different set of operations for combining truth values in *logical* expressions, denoted by `!`, `&&`, and `||`, as we'll show in another example soon. The difference is subtle but very important.

A Verilog module that creates the memory-module enables (EN_L[3:0]) and the byte enables (BE_L[7:0]) is shown in Program 6-10. Its inputs are the ten high-order and the three low-order bits of the memory address, and the 2-bit operation size. It uses reg vectors EN and BE for the enable signals internally, and it creates the required external active-low outputs at the very end of the module

Program 6-10 Verilog model for the memory-module decoder of Figure 6-13.

```
module Vrmemdec (HADDR, LADDR, SIZE, EN_L, BE_L);
  input [39:30] HADDR;
  input [2:0] LADDR;
  input [1:0] SIZE;
  output [3:0] EN_L;         // Active-low outputs
  output [7:0] BE_L;
  reg EN_MEM;                // Internal master enable
  reg [3:0] EN;              // Active-high internal versions of outputs
  reg [7:0] BE;
  integer i;

  parameter BYTE  = 2'b00,   // Encoding for operation size
            HWORD = 2'b01,
            WORD  = 2'b10,
            LWORD = 2'b11;

  always @ (*) begin
    EN = 4'b0000; BE = 8'h00;              // Default, outputs not enabled
    EN_MEM = (HADDR[39:32] == 8'h00);      // Check first whether mem is enabled
    if (EN_MEM) begin
      for (i=0; i<=3; i=i+1)               // Enable module addressed by HADDR
        if (HADDR[31:30] == i) EN[i] = 1'b1;
      if (SIZE == LWORD) BE = 8'hFF;       // Longword, enable all bytes
      else if (SIZE == WORD) begin         // Word (4 bytes)
        if (LADDR == 3'b000) BE = 8'h0F;   // LADDR is aligned, enable bytes
        else if (LADDR == 3'b100) BE = 8'hF0;
      end                                  //   else no enables
      else if (SIZE == HWORD)              // Halfword (2 bytes)
        case (LADDR)
          3'b000: BE = 8'b00000011;        // Four cases of aligned LADDR
          3'b010: BE = 8'b00001100;
          3'b100: BE = 8'b00110000;
          3'b110: BE = 8'b11000000;
          default BE = 8'b00000000;        // No enables if LADDR not aligned
        endcase
      else                                 // SIZE == BYTE
        for (i=0; i<=7; i=i+1)
          if (LADDR == i) BE[i] = 1'b1;
    end
  end

  assign EN_L = ~EN; assign BE_L = ~BE;  // Create the active-low module outputs
endmodule
```

using continuous-assignment statements. A parameter statement defines the encoding for the operation size.

The module uses an always block to model the decoder behaviorally. First it checks the eight high-order address bits to determine if the memory is enabled at all. If so, a for loop asserts the bit of EN corresponding to the memory module selected by address input bits HADDR[31:30]. Next comes the calculation of the byte enables, which is done separately for each of the four possible operation sizes. Just for illustrative purposes, four different methods are used. For longword operations, BE is unconditionally set to all 1s. For word operations, an if statement sets the appropriate bits of BE to enable the low-order or high-order four bytes depending on the value of the low-order address bits LADDR[2:0]. For halfword operations, a case statement is used to enable the appropriate pair of bytes. And for byte operations. a for loop compactly decodes the three low-order address bits to enable the corresponding single byte, in much the same way as a for loop decoded HADDR[31:30] at the beginning of the always block.

A self-checking test bench for the decoder module is shown in two parts in Program 6-11. The first part has the declarations; defines a task displayerrors to count errors and display the UUT's inputs and outputs when one is detected; and instantiates the UUT. The second part contains the main body of the test bench, an initial block.

Program 6-11 Test bench module for the Vrmemdec module of Program 6-10 (part 1).

```
module Vrmemdec_tb();
  reg [39:30] HADDR;
  reg [2:0] LADDR;
  reg [1:0] SIZE;
  wire [3:0] EN_L;
  wire [7:0] BE_L;
  reg [3:0] EN, ENMASK;     // Active-high internal versions of outputs
  reg [7:0] BE, BEMASK;;
  reg [1:0] MADDR;          // Gets set to module addr (HADDR[31:30])
  integer i, ahi, alo, sz, errors;

  parameter BYTE  = 2'b00,  // Encoding for operation size
            HWORD = 2'b01,
            WORD  = 2'b10,
            LWORD = 2'b11;

  task displayerror;
    begin
      errors = errors+1;
      $display("Error: HADDR=%10b, LADDR=%3b, SIZE=%2b, EN=%4b, BE=%8b",
                HADDR, LADDR, SIZE, EN, BE);
    end
  endtask

  Vrmemdec UUT ( .HADDR(HADDR), .LADDR(LADDR),           // Instantiate the UUT
                 .SIZE(SIZE), .EN_L(EN_L), .BE_L(BE_L) );
```

Program 6-11 (part 2)

```
  initial begin
    errors = 0;
    for (ahi=0; ahi<1024; ahi=ahi+1) for (alo=0; alo<8; alo=alo+1)
          for (sz=0; sz<4; sz=sz+1) begin
      HADDR = ahi; LADDR = alo; SIZE = sz; // Set up UUT inputs
      MADDR = HADDR[31:30];                // Set module-select part of HADDR
      #10 ;                                // Wait for valid decoder outputs
      EN = ~EN_L;  BE = ~BE_L;             // Get active-high versions
      ENMASK = ~(2**(MADDR));              // All 1s except selected module enable bit
      if (HADDR[39:32]!=8'b0) begin        // Memory not enabled
        if (EN!==4'b0000) displayerror;
      end else begin                       // Memory enabled
        if ( (EN[MADDR] !== 1'b1) || ((EN & ENMASK)!==4'b0000) ) // Check for EN errors
          displayerror;
        if (SIZE==BYTE)                    // Check for BE errors according to SIZE
          begin
            BEMASK = ~(2**(LADDR));        // All 1s except selected byte's BE bit
            if ( (BE[LADDR] !== 1'b1) || ((BE & BEMASK)!==8'h00) ) displayerror;
          end
        else if (SIZE==HWORD)
          case (LADDR)
            3'b000: if (BE !== 8'b00000011) displayerror;
            3'b010: if (BE !== 8'b00001100) displayerror;
            3'b100: if (BE !== 8'b00110000) displayerror;
            3'b110: if (BE !== 8'b11000000) displayerror;
            default if (BE !== 8'b00000000) displayerror;
          endcase
        else if (SIZE==WORD)
          case (LADDR)
            3'b000: if (BE !== 8'b00001111) displayerror;
            3'b100: if (BE !== 8'b11110000) displayerror;
            default if (BE !== 8'b00000000) displayerror;
          endcase
        else      // SIZE == LWORD
          if ((LADDR==3'b000) && (BE !== 8'b11111111)) displayerror;
          else if ((LADDR!=3'b000) && (BE !== 8'b00000000)) displayerror;
      end
    end
  end
endmodule
```

As shown, the test bench uses a triple nested `for` loop to apply all possible combinations on `HADDR`, `LADDR`, and `SIZE` to the UUT. For each combination, it first checks to see if it's one that enables the memory at all. If not, it ensures that all of the enables are negated. Otherwise, it goes on to check whether the module-enable and byte-enable signals (`EN[3:0]` and `BE[7:0]`) have the correct values for the current input combination. For the module enable, it checks that the `EN` bit corresponding to the current value of `HADDR[31:30]` (`MADDR`) is 1, and

JUST-IN-TIME VERILOG FOR PROGRAM 6-11

task

endtask

Declared at the beginning of a module, Verilog “tasks” are used primarily in test benches, to automate repetitive tasks or otherwise improve the module’s structure and readability. A task begins with the keyword `task`, followed by the task name and a semicolon. A task may have input and output arguments, and they are declared just after the task name with the `input` and `output` keywords as in a module; there are none in the example. A task may also declare local variables (`reg` or `integer` but not `wire`), whose values are not preserved from one invocation of the task to another. The task declarations are followed by a single procedural statement, usually a `begin-end` block, and the keyword `endtask`.

!, &&, ||

Besides boolean operators for combining signals, which we’ve already seen, Verilog has different operators for combining truth values in logical expressions that are used to control `if` and `for` statements and the like. Similar to the C programming language, Verilog uses `!`, `&&`, and `||` for NOT, AND, and OR, respectively. If a truth value is assigned to a signal, it uses a 1-bit value of 1 (`1'b1`) for true and 0 for false. Conversely, if a signal is used as a truth value in a logical expression, a bit or vector that has any 1s is interpreted as true; only a 0 bit or a vector of all 0s is false. This can lead to some frustrating bugs if you’re sloppy in your use of logical operators; see the discussion in Section 5.5 beginning on page 194.

that the others are all 0. Notice how the code constructs the 4-bit variable `ENMASK` to be all 1s except in the bit position that should be 1, and “knocks out” that bit in `EN` using an AND operation, so the remaining `EN` bits can be compared against 0.

After checking the module enables, the test bench checks the byte enables, using different code depending on `SIZE`. For byte operations, it uses a method similar to one in the module-enable code to ensure that the `BE` bit for the selected byte is 1 and all others are 0. For the other operation sizes, it compares the `BE` vector against its expected value as a function of the low-order address bits.

Note that the test bench expects operations of all sizes normally to be “aligned” on corresponding address boundaries, as we indicated in the original problem statement. If they are not aligned, then it expects the `BE` bits to all be 0. Running the test bench against the Verilog module in Program 6-10 shows that the module is not quite correct—we failed to check for proper alignment for longword operations. Correcting this error is left as Exercise 6.29.

The next subsection gives another, classic example of a decoder that asserts multiple outputs at a time.

6.3.6 Seven-Segment Decoders

seven-segment display

Look at your wristwatch and you may see a *seven-segment display*. This type of display, which normally uses light-emitting diodes (LEDs) or liquid-crystal display (LCD) elements, is used in watches, calculators, and instruments to display decimal data. A digit is displayed by illuminating a subset of the seven line segments shown in Figure 6-23(a).

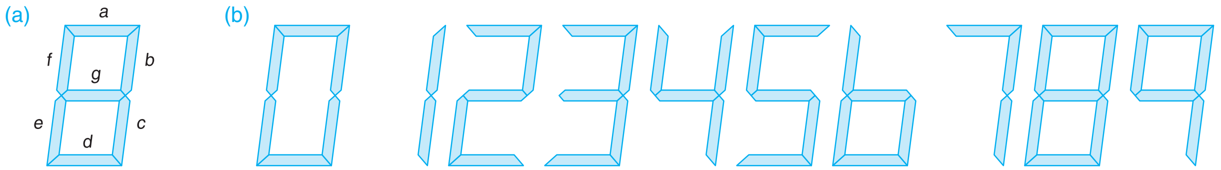


Figure 6-23 Seven-segment display: (a) segment identification; (b) decimal digits.

seven-segment decoder

A *seven-segment decoder* has 4-bit BCD as its input code and the “seven-segment code,” which is graphically depicted in Figure 6-23(b), as its output code. This is perhaps the best example of a decoder that is not a binary decoder.

Program 6-12 is a Verilog model for a seven-segment decoder with 4-bit BCD digit input DIG, active-high enable input EN, and segment outputs SEGA–SEGG. Note the use of concatenation and an auxiliary variable SEGS to make the model more readable. The model can be easily modified for different encodings and features, for example, to add “tails” to digits 6 and 9 (in Exercise 6.37) or to display hexadecimal digits A–F instead of treating these input combinations as “don’t-cares” (in Exercise 6.38).

Program 6-12 Verilog module for a seven-segment decoder.

```
module Vr7segdec(DIG, EN, SEGA, SEGB, SEGC, SEGD,
                          SEGE, SEGF, SEGG);
  input [3:0] DIG;
  input EN;
  output reg SEGA, SEGB, SEGC, SEGD, SEGE, SEGF, SEGG;
  reg [1:7] SEGS;

  always @ (DIG or EN or SEGS) begin
    if (EN)
      case (DIG)
    // Segment patterns  abcdefg
        4'd0:  SEGS = 7'b1111110;  // 0
        4'd1:  SEGS = 7'b0110000;  // 1
        4'd2:  SEGS = 7'b1101101;  // 2
        4'd3:  SEGS = 7'b1111001;  // 3
        4'd4:  SEGS = 7'b0110011;  // 4
        4'd5:  SEGS = 7'b1011011;  // 5
        4'd6:  SEGS = 7'b0011111;  // 6 (no 'tail')
        4'd7:  SEGS = 7'b1110000;  // 7
        4'd8:  SEGS = 7'b1111111;  // 8
        4'd9:  SEGS = 7'b1110011;  // 9 (no 'tail')
        default SEGS = 7'bxxxxxxx;
      endcase
    else SEGS = 7'b0000000;
    {SEGA, SEGB, SEGC, SEGD, SEGE, SEGF, SEGG} = SEGS;
  end
endmodule
```

I DON'T CARE

Note that the SEGS output in default case in the seven-segment decoder module of Program 6-12 has been specified as seven bits of x, which some synthesizers interpret as "don't-care." If nondecimal input values never occur in normal operation, don't-cares may allow the synthesizer to reduce the number of gates needed in a gate-level realization, for example, in an ASIC. On the other hand, if the decoder is realized using a table lookup as in an FPGA, the don't- cares will not provide any savings, and the designer may prefer to specify a value like all 0s or all 1s instead.

A TRICKY TEST BENCH

A test bench for the seven-segment decoder is shown in Program 6-13. The test bench merely steps through the 16 possible input combinations on DIG, and displays the output for each one. However, this test bench is rather unusual in that, rather than printing out a list of output values, it actually reproduces the visual appearance of the seven-segment display by writing out spaces, underscores, vertical bars, and newlines corresponding to each segment output. Study it, or better yet, try it!

Program 6-13 Verilog test bench for a seven-segment decoder.

```
`timescale 1ns / 100ps
module Vr7seg_tb ();
  reg EN;
  reg [3:0] DIG;
  wire SEGA, SEGB, SEGC, SEGD, SEGE, SEGF, SEGG;
  integer i;

  Vr7segdec UUT (.DIG(DIG),.EN(EN),.SEGA(SEGA),.SEGB(SEGB),
    .SEGC(SEGC),.SEGD(SEGD),.SEGE(SEGE),.SEGF(SEGF),.SEGG(SEGG));
  initial begin
    EN = 1; // Enable all
   for (i=0; i<16; i=i+1)
    begin
     DIG = i;
     #5 ;
     $write("Iteration %0d\n", i);
     if (SEGA) $write(" __\n"); else $write("\n");
     if (SEGF) $write("|"); else $write(" ");
     if (SEGG) $write("__"); else $write("  ");
     if (SEGB) $write("|\n"); else $write("\n");
     if (SEGE) $write("|"); else $write(" ");
     if (SEGD) $write("__"); else $write("  ");
     if (SEGC) $write("|\n"); else $write("\n");
     #5 ;
    end
    $write("Done\n");
  end
endmodule
```

JUST-IN-TIME VERILOG FOR PROGRAM 6-13

$write

This test bench uses Verilog's built-in `$write` task, which behaves exactly the same as `$display`, except it does not append a newline at the end of its output. The facilitates the construction of the displayed digits, segment by segment.

The "logical expression" in each `if` statement is just a 1-bit wire, which is not technically a truth value. A "proper" logical expression would be "`SEGA==1'b1`". However, the compiler will treat a 1-bit value of 1 as "true." In simple cases like this, it seems more clear to use the terse formulation of the condition expression.

6.3.7 Binary Encoders

binary encoder

In Section 6.3.1, we defined a decoder to be any multiple-input, multiple-output combinational logic circuit that converts an input code word into an output code word in a different code. With that definition, a circuit that converts in the opposite direction as a binary decoder is also a decoder, but it's usually called a *binary encoder*. As shown in Figure 6-24(a), its input code is the 1-out-of-2^n code and its output code is n-bit binary. The equations for an 8-to-3 encoder with inputs I0–I7 and outputs Y0–Y2 are given below:

$$\begin{aligned} \text{Y0} &= \text{I1} + \text{I3} + \text{I5} + \text{I7} \\ \text{Y1} &= \text{I2} + \text{I3} + \text{I6} + \text{I7} \\ \text{Y2} &= \text{I4} + \text{I5} + \text{I6} + \text{I7} \end{aligned}$$

The corresponding logic circuit is shown in (b). In general, a 2^n-to-n encoder can be built from n 2^{n-1}-input OR gates. Bit i of the input code is connected to OR gate output j if bit j in the binary representation of i is 1.

A standard binary encoder's output is meaningful only if exactly one input is asserted; after all, it expected a 1-out-of-2^n coded input. Its output is pretty much useless if two or more of its inputs are asserted—the output code word is the bit-by-bit logical OR of the code words corresponding to all of the asserted inputs. In situations where multiple inputs may be asserted simultaneously, a designer can use a "priority encoder," where the output code word corresponds to the asserted input with the highest priority, where "priority" is based on the input numbering. We'll show how these are designed later, in Section 7.2.

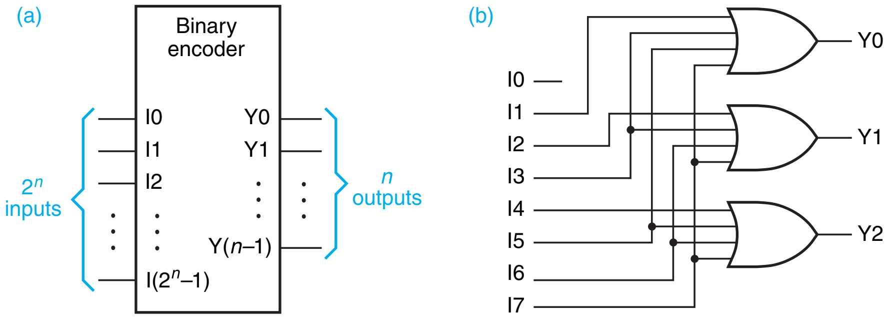


Figure 6-24
Binary encoder:
(a) general structure;
(b) 8-to-3 encoder.

6.4 Multiplexing

In the preceding section, we saw that decoding and selecting are basic requirements in many applications, and there are specific circuits—decoders—to match. A common selecting operations is to pick a source of data which is to be transferred to a destination across a shared medium; this operation is common enough to have a name—*multiplexing*. In digital applications, a typical medium is a wire or bus, though it could be a fiber-optic cable or even a radio channel.

multiplexing

A *multiplexer* is a digital switch—it connects data from a one of *n* sources to its output, as depicted in Figure 6-25. A select input S selects which of the *n* data inputs is to be transferred to the output, and an optional enable input EN may be provided to allow or block the transfer. If the S input has *s* bits, then *n* may be as large as 2^s. The individual data sources and the output may each be one bit wide as in the figure, or they may be *b*-bit-wide buses, where the 1-bit switches are simply replicated and controlled by the same S and EN inputs, as we'll show later. By the way, a multiplexer is often called a *mux* for short.

multiplexer

mux

Multiplexers have an affinity with binary decoders since they perform a selection function, plus they perform a transfer based on the selection. Thus, a multiplexer can be thought of (and actually implemented as) a collection of individual switches controlled by a binary decoder, as shown in Figure 6-26. The multiplexer's enable and select inputs are connected to the decoder's enable and

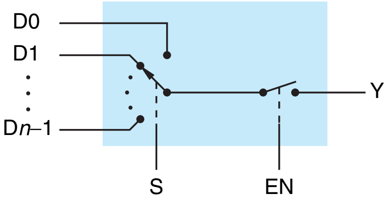


Figure 6-25
A multiplexer as a multi-position switch.

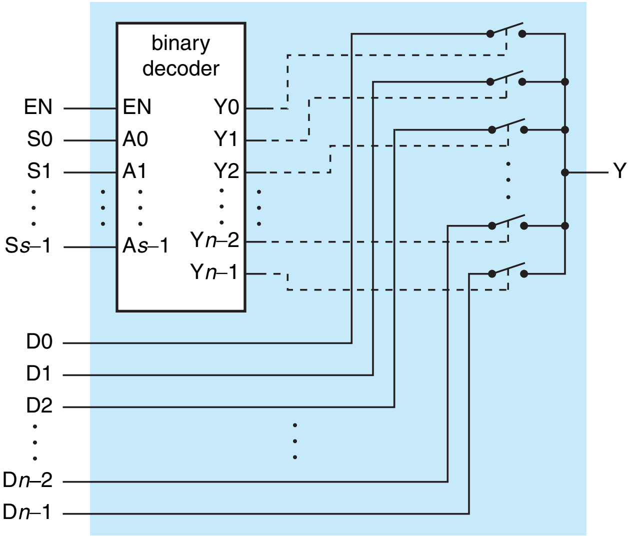


Figure 6-26
Implementing a multiplexer with a decoder and switches.

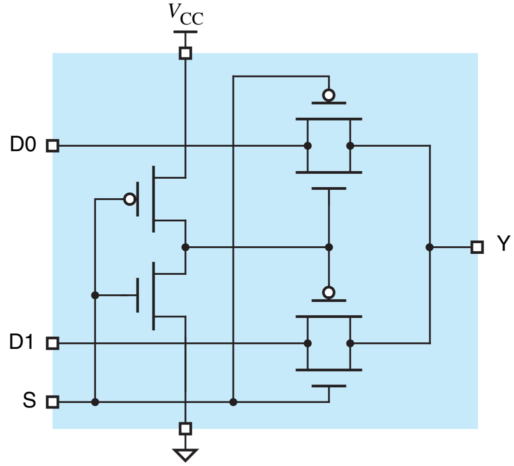


Figure 6-27
Two-input multiplexer using CMOS transmission gates.

address inputs. The decoder's outputs are connected to individual switches corresponding to the like-numbered data sources. With a standard binary decoder, at most one switch will be activated at a time, transferring the connected data source to the Y output.

CMOS circuits often implement multiplexers in exactly the way shown in the figure, because they have a component—the transmission gate—that acts just like a switch and has very little propagation delay (see Section 14.5.1). For example, the CMOS transistor-level circuit for a 2-input, 1-bit wide mux is shown in Figure 6-27. The leftmost pair of transistors is a CMOS inverter, and each of the other two pairs is a transmission gate. When S is 1, the path from D1 to Y is enabled, and when it's 0, the path from D0 to Y is enabled. The inverter and the transmission gates require typical CMOS delays to change when S changes state, but once they have settled, the delay through the enabled transmission gate is extremely fast, almost as fast as a wire in advanced CMOS technologies.

6.4.1 Gate-Level Multiplexer Circuits

A gate-level implementation of a multiplexer is different, since we have no switches to work with. Instead, we can use the decoder outputs to enable AND gates, one per data source, and combine their outputs with an OR gate, as shown in Figure 6-28. If you compare this circuit carefully with a standard binary decoder circuit, you realize that the AND functions performed by the n AND gates can actually be subsumed into the n AND gates that are already present in the s-to-n binary decoder. This adds one input—the corresponding data source—to each of the AND gates, and one more for the EN input, and yields the classic gate-level multiplexer circuit of Figure 6-29 when s=2 and n=4.

8-input, 1-bit multiplexer

Similarly, the logic diagram for an *8-input, 1-output multiplexer* is shown in Figure 6-30(a), with its traditional logic symbol in (b). The multiplexer's logic function is probably obvious to you from the word description of muxes, but we

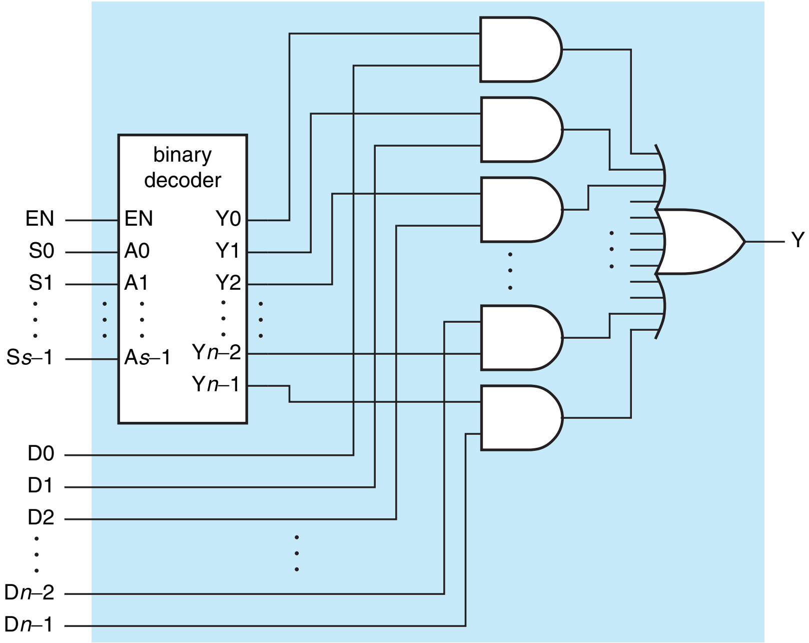


Figure 6-28
Multiplexer circuit using a decoder and gates.

have written its truth table in Table 6-6 to illustrate another extension of our truth-table notation. Up until now, our truth tables have specified an output of 0 or 1 for each input combination. In Table 6-6, only the "control" inputs are listed under the "Inputs" heading. The output is specified as a constant (in this case, 0) or as a simple logic function of the "data" inputs (e.g., D0). This notation saves eight columns and eight rows in the table, and presents the logic function more clearly than a larger table would.

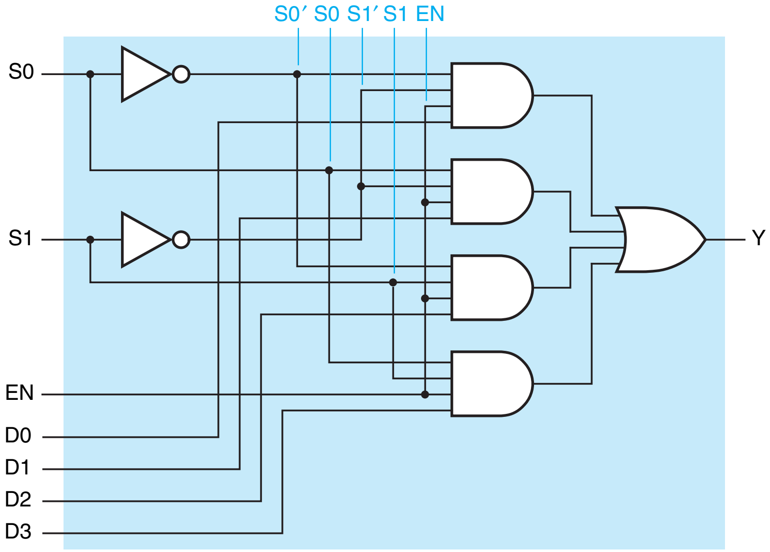


Figure 6-29
Four-input multiplexer circuit using gates.

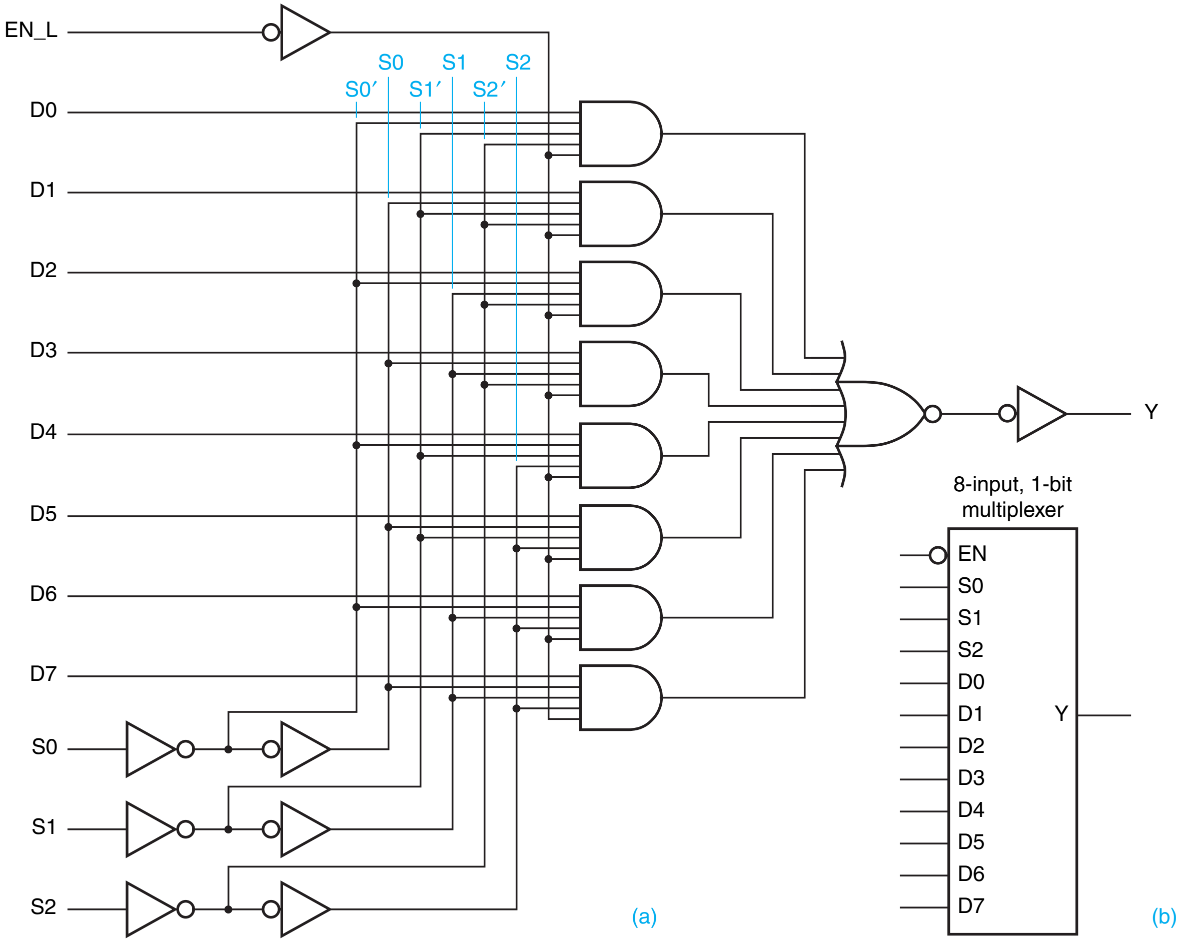


Figure 6-30 An 8-input, 1-bit multiplexer: (a) logic diagram; (b) traditional logic symbol.

EXTRA INVERTERS

Notice that the logic diagram in Figure 6-30 has some extra inverters in it. Depending on the circuit implementation, especially in an ASIC, performance may suffer if more than a few gate inputs are driven by any given input signal, as discussed in Section 14.4. The extra inverters on EN_L (which is now active-low) and the S inputs provide extra electrical buffering which hides the load of the mux's eight internal AND gates from the circuit that drives it and perhaps other logic.

In most modern design environments, the synthesis tools automatically take care of adding extra buffering where it's needed for performance. The logic diagram in Figure 6-30 is taken from an MSI 8-input multiplexer component, which had such buffering built into every chip.

Inputs				*Output*
EN_L	S2	S1	S0	Y
1	x	x	x	0
0	0	0	0	D0
0	0	0	1	D1
0	0	1	0	D2
0	0	1	1	D3
0	1	0	0	D4
0	1	0	1	D5
0	1	1	0	D6
0	1	1	1	D7

Table 6-6
Truth table for an 8-input, 1-bit multiplexer.

Going back to the general case, a multiplexer's data inputs and output may be (and usually are) more than one bit wide. An n-input, b-bit multiplexer has the inputs and outputs shown in Figure 6-31(a). There are n sources of data, each of which is b bits wide, and there are b output bits. In many applications, $n = 2, 4, 8$, or 16, and $b = 1, 2, 4, 8, 16, 32$, or more. There are s inputs that select among the n sources, so $s = \lceil \log_2 n \rceil$ (the ceiling of $\log_2 n$, i.e., the smallest integer greater than or equal to $\log_2 n$). An enable input EN allows the selected source to be transferred to the output; when EN = 0, all of the outputs are 0.

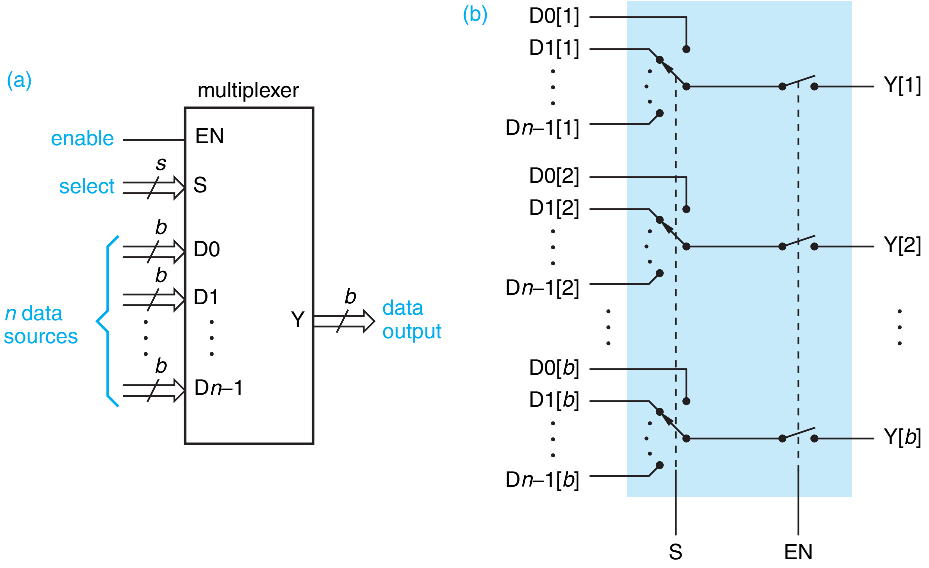


Figure 6-31
General multiplexer structure: (a) inputs and outputs; (b) functional equivalent.

HAVING IT BOTH WAYS

When a multiplexer is realized with gate-level circuits, as in Figures 6-29 and 6-30, information flows unidirectionally from inputs to outputs. However, some technologies, in particular CMOS transmission gates, actually can implement bidirectional, logic-controlled switches to mimic the structure shown in Figure 6-26. Multiplexer-like devices using such technologies are often called “multiplexers/demultiplexers,” since they can select one of multiple data sources on the left to connect to a single destination on the right, like a multiplexer, or connect a single data source on the right to a selected one of multiple destinations on the left, like a demultiplexer. They are also sometimes simply called “switches.”

Figure 6-31(b) shows a switch circuit that is roughly equivalent to the multiplexer. However, unless otherwise stated, a multiplexer is a unidirectional device: information flows only from inputs (on the left) to outputs (on the right). Information flows bidirectionally only in actual switches. Notice that the *b* bits from a particular input source, say D0, are spread out across *b* switches, each of which has *n* inputs to accommodate the *n* different sources.

2-input, 4-bit multiplexer

A multiplexer may have as few as two inputs. Figure 6-32 shows the gate-level circuit for a *2-input, 4-bit multiplexer* which selects between two 4-bit inputs, again with an active-low enable input. Our extended truth-table notation makes the device’s description very compact and understandable, as shown in Table 6-7. (The figure and table have a change in signal naming compared to Figure 6-31(b), matching the original MSI naming for this function.)

Multiplexers are obviously useful devices in any application in which data must be switched from multiple sources to a destination. One common use in microprocessor systems is in input/output (I/O) devices that have several registers for storing data and control information, where any one of those registers may be selected periodically to be read by software. Suppose there are eight 32-bit registers, and a 3-bit field in the I/O address selects which one to read. This 3-bit field is connected to the select inputs of an 8-input, 32-bit multiplexer. The multiplexer’s data inputs are connected to the eight registers, and its data outputs are connected to the microprocessor’s data bus to read the selected register.

Table 6-7 Truth table for a 2-input, 4-bit multiplexer.

Inputs		*Outputs*			
EN_L	S	1Y	2Y	3Y	4Y
1	x	0	0	0	0
0	0	1D0	2D0	3D0	4D0
0	1	1D1	2D1	3D1	4D1

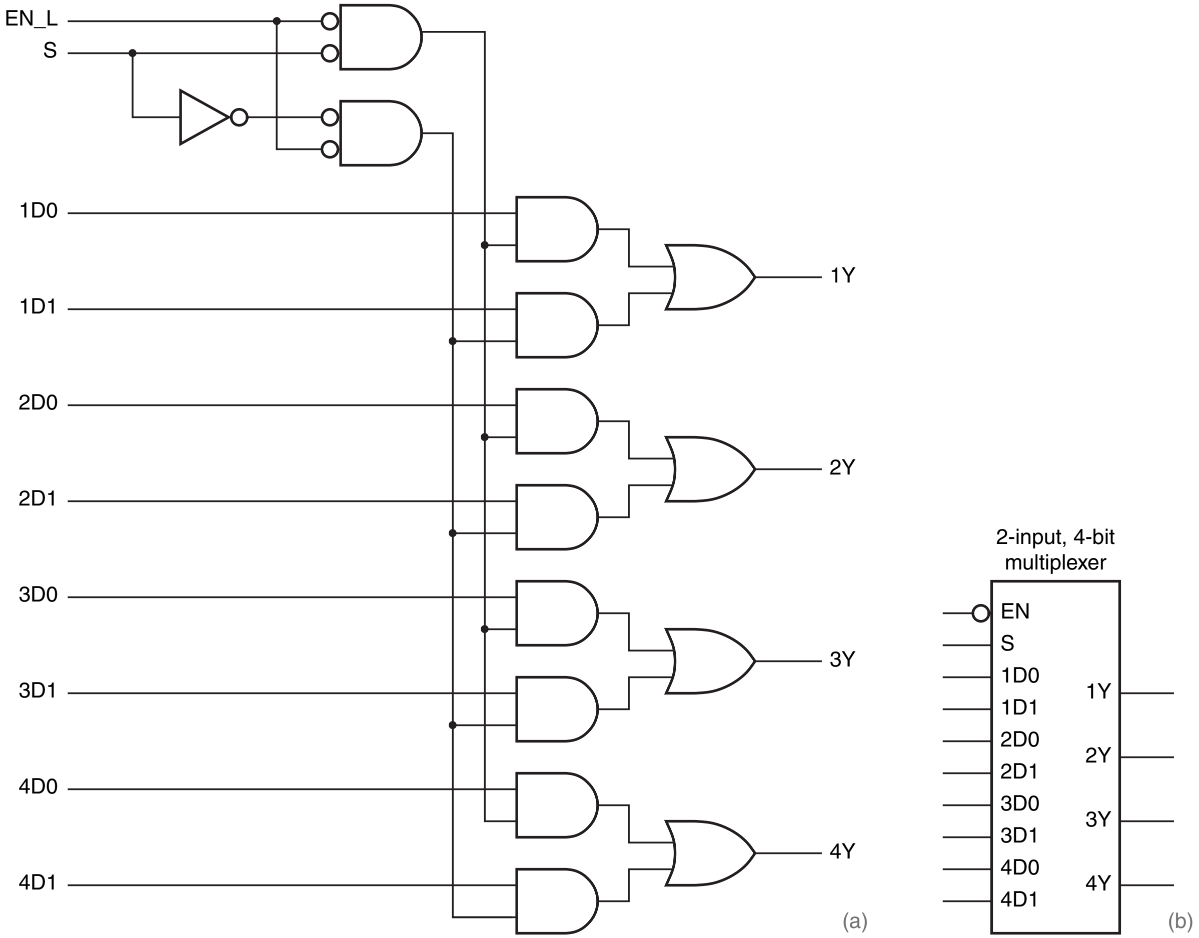


Figure 6-32 A 2-input, 4-bit multiplexer: (a) logic diagram; (b) traditional logic symbol.

6.4.2 Expanding Multiplexers

As we'll see in the next subsection, the size of a multiplexer in an HDL model can be modified at will to match the characteristics of the problem at hand, by simply by changing the appropriate parameters in the multiplexer's definition. However, in ASIC design, optimized multiplexer cells may be provided only in a few fixed sizes, and it may be necessary for the designer to construct a large multiplexer from a collection of smaller ones.

For example, we suggested earlier that an 8-input, 32-bit multiplexer might be used in the design of a computer processor. This function could be performed by 32 8-input, 1-bit multiplexers or equivalent ASIC cells, each one handling one bit of all the inputs and the output. The appropriate 3-bit field of the I/O address would be connected to the S2–S0 inputs of all 32 muxes, so they would all select the same register source at any given time.

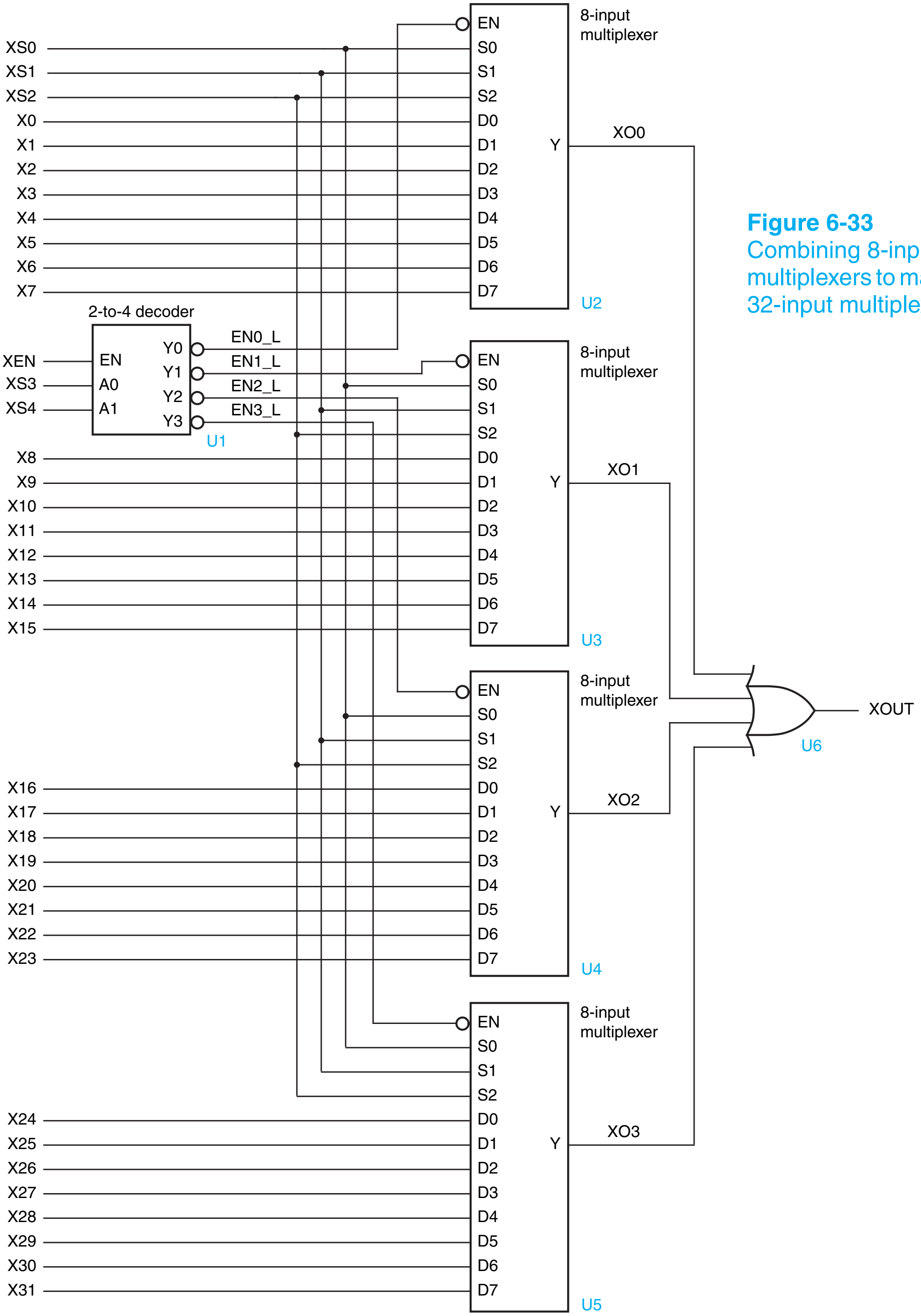


Figure 6-33
Combining 8-input multiplexers to make a 32-input multiplexer.

Another dimension in which multiplexers can be expanded is the number of data sources. For example, suppose we needed a 32-input, 1-bit multiplexer. Figure 6-33 shows one way to build it. Five select bits are required. A 2-to-4 decoder with active-low outputs is connected to the two high-order select bits to enable one of four 8-input, 1-bit multiplexers of the types we showed in Figure 6-30 on page 284. Since only one 8-input multiplexer is enabled at a time, their outputs can simply be ORed to obtain the final output.

6.4.3 Multiplexers, Demultiplexers, and Buses

A multiplexer can be used to select one of n sources of data to transmit on a bus. At the far end of the bus, a *demultiplexer* can be used to route the bus data to one of m destinations. Such an application, using a 1-bit bus, is depicted in terms of our switch analogy in Figure 6-34(a). In fact, block diagrams for logic circuits often depict multiplexers and demultiplexers using the wedge-shaped symbols in (b), to suggest visually how a selected one of multiple data sources is directed onto a bus and is then routed to a selected one of multiple destinations.

demultiplexer

The function of a demultiplexer is just the inverse of a multiplexer's. For example, a 1-bit, n-output demultiplexer has one data input and s inputs to select one of $n = 2^s$ data outputs. In normal operation, all outputs except the selected one are 0; the selected output equals the data input. This definition may be generalized for a b-bit, n-output demultiplexer; such a device has b data inputs, and its s select inputs choose one of $n = 2^s$ sets of b data outputs.

A binary decoder with an enable input can be used as a demultiplexer, as shown in Figure 6-35. The decoder's enable input is connected to the data line, and its select inputs determine which of its output lines is driven with the data bit. The remaining output lines are negated.

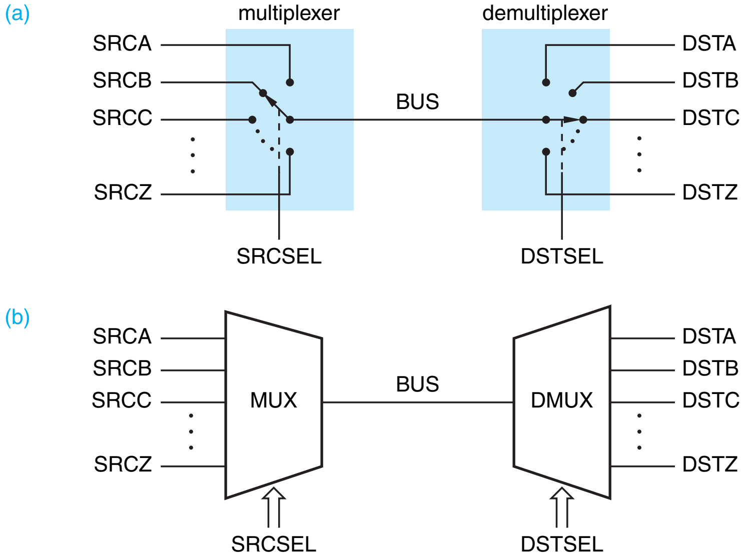


Figure 6-34 A mux driving a bus and a demultiplexer receiving the bus: (a) switch equivalent; (b) block-diagram symbols.

WORTHLESS? One of the reviewers of this book "curses every author who has ever mentioned a demultiplexer—possibly the most worthless element ever created." I get the point. After all, why waste gates to send data selectively to one of *n* different destinations, when you could just hook the destination wires together and send the data to all *n* destinations? Let the ones who don't want or need the data ignore it!

Well, there *are* sometimes reasons to prefer the demultiplexer, both in ASICs and in large multimodule systems. The key difference between *n* demultiplexer outputs and *n* wires hooked together is that the *n*–1 unselected demultiplexer outputs are inactive. So, no power is used driving data that won't get used, which may be significant if the driven wires are many (as in wide backplane buses) or long (and may radiate a lot of electrical noise). Also, there's no chance of the unused data being snooped or triggering unwanted activity. Some of these factors may be particularly relevant in systems that use serial communication, where commands and data are sent between subsystems using just a single wire.

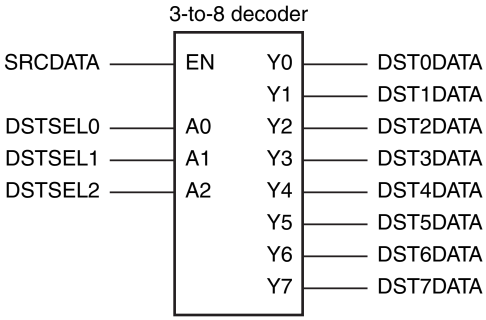


Figure 6-35 Using a 3-to-8 binary decoder as a 1-bit, 8-output demultiplexer.

6.4.4 Multiplexers in Verilog

Multiplexers can be described easily in Verilog in several different ways. In the dataflow style, a series of conditional operators (`?:`) can provide the required functionality, as shown in Program 6-14, a dataflow-style Verilog module for a 2-input, 8-bit multiplexer.

There are a couple options for coding multiplexers in behavioral style. One approach is to use a series of nested `if` statements, one for each value of the

Program 6-14 Dataflow Verilog module for a 2-input, 8-bit multiplexer.

```
module Vrmux2in8b_d(EN_L, S, D0, D1, Y);
  input EN_L, S;
  input [1:8] D0, D1;
  output [1:8] Y;

  assign Y = (~EN_L == 1'b0) ? 8'b0 : (
               (S == 1'd0) ? D0: (
                 (S == 1'd1) ? D1: 8'bx));
endmodule
```

Program 6-15 Behavioral Verilog for the mux using nested `if` statements.

```
module Vrmux2in8b_b(EN_L, S, D0, D1, Y);
  input EN_L, S;
  input [1:8] D0, D1;
  output reg [1:8] Y;

  always @ (*) begin
    if (~EN_L == 1'b0) Y = 8'b0;
    else if (S == 1'b0) Y = D0;
      else if (S == 1'b1) Y = D1;
        else Y = 8'bx;
  end
endmodule
```

Program 6-16 Behavioral Verilog for a 4-input, 8-bit multiplexer using `case`.

```
module Vrmux4in8b(EN_L, S, A, B, C, D, Y);
  input EN_L;
  input [1:0] S;
  input [1:8] A, B, C, D;
  output reg [1:8] Y;

  always @ (*) begin
    if (~EN_L == 1'b0) Y = 8'b0;
    else case (S)
      2'd0: Y = A;
      2'd1: Y = B;
      2'd2: Y = C;
      2'd3: Y = D;
      default: Y = 8'bx;
    endcase
  end
endmodule
```

select input, as shown in Program 6-15 for the 2-input, 8-bit multiplexer. However, this approach quickly becomes awkward because of the deep nesting when there are more than a few select values.

A more natural approach is to use a `case` statement with one case for each value of the select input, as shown in Program 6-16. This approach is much more readable and maintainable, especially when there are a lot of cases (values of the

JUST-IN-TIME VERILOG FOR PROGRAM 6-14

padding

If a Verilog vector has fewer bits than a vector it is assigned to or combined with, it is padded on the left with 0s to match lengths. Thus, the literal `8'b0` is equivalent to `8'b00000000`. However, if its leftmost bit before padding is `x` or `z`, then that bit is used; so `8'bx` is equivalent to `8'bxxxxxxxx`.

Table 6-8 Function table for a specialized 4-input, 18-bit multiplexer.

S2	S1	S0	*Input to Select*
0	0	0	A
0	0	1	B
0	1	0	A
0	1	1	C
1	0	0	A
1	0	1	D
1	1	0	A
1	1	1	B

select input). This code also flows very naturally from the multiplexer's truth table, if it's written in the extended, compact style of Table 6-6 or 6-7.

As you might imagine, it is very straightforward to extend the case-based behavioral multiplexer with additional cases or specialized cases. For example, consider a specialized 4-input, 18-bit multiplexer with the selection criteria in Table 6-8. Program 6-17 is a case-based module for such a multiplexer.

In each example module in this subsection, if the select inputs are not valid (e.g., contain z's or x's), the output bus is set to "unknown" to help catch errors during simulation.

A self-checking test bench for the 2-input, 8-bit multiplexers is shown in Program 6-18; it can be used with either module, since they have the same input/output signals and function. Notice that this test bench has a user-defined task displayerror which saves typing and unclutters the main body of the code, allowing us to focus on the test cases, as we'll discuss next.

Program 6-17 Behavioral Verilog for a specialized 4-input, 18-bit multiplexer.

```
module Vrmux4in18b(S, A, B, C, D, Y);
  input [2:0] S;
  input [1:18] A, B, C, D;
  output reg [1:18] Y;

  always @ (*)
    case (S)
      3'd0, 3'd2, 3'd4, 3'd6: Y = A;
      3'd1, 3'd7: Y = B;
      3'd3: Y = C;
      3'd5: Y = D;
      default: Y = 18'bx;
    endcase
endmodule
```

Program 6-18 Verilog test bench for 2-input, 8-bit multiplexer.

```
`timescale 1 ns / 100 ps
module Vrmux2in8b_tb ();
  reg EN, S;
  reg [1:8] D0, D1;
  wire [1:8] Y;
  integer i, errors;

  task displayerror;
    begin
      errors = errors+1;
      $display("Error: EN=%b, S=%b, D0=%b, D1=%b, Y=%b", EN, S, D0, D1, Y);
    end
  endtask

  Vrmux2in8b_b UUT ( .EN_L(~EN), .S(S), .D0(D0), .D1(D1), .Y(Y) );

  initial begin
    errors = 0;
    for (i=0; i<2500; i=i+1) begin
      EN = 0; S = 0; #10 ;
      if (Y !== 0) displayerror;
      S = 1; #10 ;
      if (Y !== 0) displayerror;
      EN = 1; D0 = $random % 256; D1 = $random % 256;
      S = 0; #10 ;
      if (Y !== D0) displayerror;
      S = 1; #10 ;
      if (Y !== D1) displayerror;
    end
    $display("Test done, %d errors",errors);
  end
endmodule
```

With any *n*-input combinational logic circuit, it is always at least theoretically possible to devise test cases that exercise the circuit and check its outputs for all 2^n possible input combinations. But if *n* is large, this is not practical. For the 2-input, 8-bit multiplexer, *n* is 18 (about 250,000 input combinations), and an exhaustive test can run in a few seconds. With a 16-bit version (34 inputs), we might have to wait all night or longer, and circuits with even more inputs or more complicated functions might be impossible to test in this way. So, as a method that can be applied to any of these examples, the test bench in Program 6-18 uses Verilog's built-in `$random` function (described on page 223) to generate a smaller number of pseudorandom inputs.

In any circuit where there are both "control" and "data" inputs, it is most important to test all combinations of the control inputs, since the corresponding module's use of such inputs is where the most variation and likely errors occur.

SAVING CASE In the multiplexer test bench in Program 6-18, you may ask, why test both values of `S` when the multiplexer is disabled? In that case, `S` is a don't-care. That's right—if the multiplexer has been modeled correctly. There are simple coding errors and typos where that may not be true, and that's exactly what we're trying to detect (see Exercise 6.43).

The data inputs should also be checked with multiple values, but if data buses are handled uniformly, typically using Verilog vectors, then the circuit's functional correctness can normally be assured using a relatively small number of random data-input combinations. If there are any special "corner cases" that are handled differently or are otherwise likely to cause problems in the circuit, those should be coded in the test bench explicitly; there are no such cases in our multiplexer.

So, the `Vrmux2in8_tb` test bench in Program 6-18 tests 2,500 random combinations of the multiplexer's 8-bit data inputs `D0` and `D1` (even 25 might be enough, but why skimp if you don't have to?). For each data-input combination, it tests all four combinations for the control inputs—disabled or enabled with either value of the select input `S`.

We'll continue with more combinational-logic elements in Chapter 7.

References

The first PAL devices were invented at Monolithic Memories, Inc. (MMI) in 1978 by John Birkner and H. T. Chua. The inventors earned a U.S. patent for their invention, and MMI rewarded them by buying them a brand new Porsche and Mercedes, respectively! Seeing value in this technology (PAL devices, not fast cars), Advanced Micro Devices (AMD) acquired MMI in the early 1980s and became a leading developer and supplier of new PLDs and CPLDs. AMD eventually sold its PLD operations to former competitor Lattice Semiconductor.

Meanwhile, FPGA architectures were created and evolved, featuring key innovations from and fierce competition between Xilinx, Inc. and Altera Corporation, which was acquired by Intel in 2015. In recent years, new CPLD development has ended, primarily because FPGA architectures have scaled more effectively. However, many suppliers continue to offer "legacy" PLDs and CPLDs, since they continue to find use in lower-density applications, especially where low cost or low power consumption are important considerations.

Probably the best resources for learning about programmable devices are provided by their manufacturers. Xilinx, Inc. publishes a comprehensive set of FPGA and CPLD data books, user guides, and application notes on their website (`www.xilinx.com`). Other comprehensive websites include those of GAL inventor Lattice Semiconductor (`www.latticesemi.com`), and Intel's "Programmable Solutions Group," still at its original URL (`www.altera.com`).

Drill Problems

6.1 Give three examples of combinational logic circuits that require billions of rows to describe in a truth table. For each circuit, describe its inputs and output(s) and indicate exactly how many rows the truth table contains; you need not write out the truth table. (*Hint:* You can find several such circuits in Chapters 6–8.)

6.2 What logic element is pictured on the first page of this chapter? Describe its inputs, outputs, associated parameters, and function.

6.3 Which CMOS circuit would you expect to be faster: a decoder with active-high outputs, or one with active-low outputs?

6.4 Why do you think that the Xilinx 7-series LUT outputs in Figure 6-6 are named “D5” and “D6” instead of “D0” and “D1”?

6.5 Prove that an active-high output function of the PAL16L8 in Figure 6-11 can be any product of up to seven sum terms involving the available inputs.

6.6 In the style of Table 6-4, write the truth table for the logic function performed *inside* the 74x138 symbol outline.

6.7 Show how to build each of the following logic functions using one or more 74x138 binary decoders and NAND gates. (*Hint:* Each realization should be equivalent to a sum of minterms.)

(a) $F = \Sigma_{X,Y,Z}(2,5,7)$ (b) $F = \prod_{A,B,C}(2,4,5,6,7)$

(c) $F = \Sigma_{A,B,C,D}(0,6,10,14)$ (d) $F = \Sigma_{W,X,Y,Z}(1,4,5,6,11,12,13,15)$

(e) $F = \Sigma_{W,X,Y,Z}(0,2,4,7)$ (f) $F = \Sigma_{A,B,C,D}(8,11,12,15)$

$G = \Sigma_{W,X,Y}(1,2,3,5)$

6.8 Run the test bench of Program 6-6 for the 2-to-4 decoder module `Vr2to4dec_s` in Program 6-1, showing that there are no errors. Then do three more runs, each time inserting or deleting just one character in `Vr2to4dec_s`, resulting in the test bench reporting 2, 4, and 8 errors.

6.9 When the author ran the test bench of Program 6-6 for his initial version of the 2-to-4 decoder module `Vr2to4dec_d` in Program 6-2, it detected a big error in a most interesting way. The author had inadvertently typed `{Y1,Y0}` instead of `{A1,A0}` in four places, and the module compiled OK—no syntax errors. Try it. What does the simulator do when it runs the test bench, and why?

6.10 Write a structural-style Verilog module `Vr2to4decp_s` corresponding to the 2-to-4 binary decoder with polarity control shown in Figure 6-3. Use individual signal names as in the logic diagram, not vectors.

6.11 Write a dataflow-style Verilog module `Vr2to4decp_d` corresponding to the 2-to-4 binary decoder with polarity control shown in Figure 6-3. Use a vector `I[1:0]` for the select inputs, and a vector `Y[0:3]` for the outputs.

6.12 Write a behavioral-style Verilog module `Vr2to4decp_b` for the 2-to-4 binary decoder with polarity control shown in Figure 6-3. Use a vector `I[1:0]` for the select inputs, and a vector `Y[0:3]` for the outputs. Be sure that your code does not create an “inferred latch.”

6.13 Write a test bench Vr2to4dec_tb that instantiates all three of the 2-to-4 binary decoders in Exercises 6.10, 6.11, and 6.12 and verifies that they produce identical outputs for all input combinations, displaying the input combination and outputs if any are different. If you don't detect any mismatch, insert an error of some kind into one of the modules to verify that your error messages are working.

6.14 Write a structural-style Verilog module Vr8to3enc_s that corresponds to the binary encoder of Figure 6-24.

6.15 Write a Verilog module Vr3to8dec_bc for a 3-to-8 binary decoder with active-low outputs Y_L[7:0] and four enable inputs where G1, G2, or both G3_L and G4_L must be asserted to enable the selected output. Your module should instantiate the VrNtoSbindec of Program 6-7 and use other statements to satisfy the above design requirements.

6.16 Write a Verilog module Vrluckyprime for a "lucky/prime encoder" with an 8-bit input representing an unsigned binary integer, and two output bits indicating whether the number is prime or divisible by 7.

6.17 After completing the preceding exercise, write a Verilog test bench that compares the outputs of your module for all possible input combinations against results computed by the simulator using its own arithmetic, and display all mismatches. Test your test bench by putting a bug in your original Verilog model.

6.18 Write a behavioral Verilog module Vrmux2in4b for a 2-input, 4-bit multiplexer with the function table shown in Table 6-7. Name the data input and output vectors D0, D1, and Y, and index them from 1 to 4.

6.19 Write a Verilog test bench Vrmux2in4b_tb that tests for correct functioning of the Vrmux2in4b module in Drill 6.18. For each value of the function inputs, it should check that the output is correct for all combinations of data-input values, and display an informative error message if it is incorrect. Insert one or more errors into Vrmux2in4b to verify that your error messages are working.

Exercises

6.20 Write code in your favorite programming language that generates the contents of a 4 × 4 multiplier ROM in the same format as Table 6-2.

6.21 Suppose that advances in silicon technology allow a 64 × 4-bit "ROM" to be fit into the same chip area that now holds a 64 × 1-bit LUT ROM that can realize any logic function of 6 variables. Design extra circuitry and write the user instructions for using a 64 × 4-bit ROM to perform any logic function of 8 inputs A0-A7 on output D8, any two functions of 7 inputs A0-A6 on outputs D7 and D8, or any four functions of 6 inputs A0-A5 on outputs D5 through D8.

6.22 In the previous exercise, without adding any more inputs or outputs, can you devise user instructions, and add circuitry as necessary, for the ROM also to perform any function of 7 inputs A0-A6 plus any two functions of 6 inputs A0-A5? Show how to do it, or explain why it can't be done.

6.23 Write a Verilog module Vr6to64decpre for a 6-to-64 binary decoder, using a generate statement to create a predecoding structure equivalent to Figure 6-20. Also write a self-checking test bench to test it for proper operation.

6.24 Sketch the design of an 8-to-256 binary decoder, with no enable input, using a multi-level predecoding structure. Assume the maximum number of inputs in an AND gate is two, so your design must divide the address inputs into four 2-bit groups, and the first level of your design will use 2-to-4 decoders. Show the number of elements vertically in each level and the number of vertical wires between levels. Also show a few typical logic equations for the signals at each level.

6.25 Write a Verilog module `Vr9to512decpre` for a 9-to-512 binary decoder with one enable input, using a generate statement to create a predecoding structure similar to Figure 6-20. You should instantiate `Vr3to8decb` decoder modules at the first level, and use only 3-input AND gates beyond that. Also write a simple behavioral module `Vr9to512decb` for the same decoder function, and write a test bench `Vr9to512dec_tb2` that compares the outputs of the two decoders.

6.26 Write a Verilog module `Vr8to256decpre` for a 8-to-256 binary decoder with a multi-level predecoding structure as described in Exercise 6.24, using a generate statement. Also write a simple behavioral module `Vr8to256decb` for the same decoder function, and write a test bench `Vr8to256dec_tb2` that compares the outputs of the two decoders.

6.27 Design a Verilog module `Vrmultidec8` for a customized decoder that has the function table in Table X6.27. Use a coding style that is easy to write and check against the function table.

CS_L	A2	A1	A0	*Output to Assert*
1	x	x	x	none
0	0	0	x	BILL
0	0	x	0	MARY
0	0	1	x	JOAN
0	0	x	1	PAUL
0	1	0	x	ANNA
0	1	x	0	FRED
0	1	1	x	ATIF
0	x	1	1	KATE

Table X6.27

6.28 Write another function table for the customized decoder in Table X6.27, one that includes all eight rows where CS_L is 0 (no don't-care inputs). In the last column, write a list of all the outputs that are asserted for each input combination. As a double check on your answer, write a test bench that instantiates `Vrmultidec8` and displays the names of asserted output signals for each input combination.

6.29 Modify the Verilog memory decoder module of Program 6-10 so that it properly handles alignment errors during longword operations. Test your new module using the test bench in Program 6-11.

6.30 Not all computers require shorter-sized memory operations to be aligned on corresponding address boundaries. Modify the Verilog memory decoder module of Program 6-10 for use in such an environment, where any halfword or word operation is legal as long as the entire addressed halfword or word fits within the same longword. For example, a word operation at address 3 would select bytes 3–6 of the first longword in memory, while word operations at addresses 5–7 would be illegal. Also provide a new output, AERR, which is asserted if an illegal operation is attempted, and ensure that all BE output bits are negated in that situation.

6.31 Modify the test bench in Program 6-11 to work with the unaligned memory decoder of Exercise 6.30.

6.32 Show how to build all four of the following functions using one 3-to-8 decoder with active-low outputs and four 2-input NAND gates:

$F1 = X' \cdot Y' \cdot Z' + X \cdot Y \cdot Z'$ $F2 = X' \cdot Y' \cdot Z + X' \cdot Y \cdot Z'$

$F3 = X' \cdot Y \cdot Z' + X \cdot Y' \cdot Z$ $F4 = X \cdot Y' \cdot Z' + X' \cdot Y \cdot Z$

unary code
thermometer code

6.33 A certain system has a 3-bit output N[2:0] which represents an integer in the range 0–7. The designer has decided to display its value seven in LEDs driven by active-high signals L[1:7] where the number of lit LEDs corresponds to the value of N (the first LED to be lit is driven by L[1]). Draw a logic diagram for an encoder circuit that converts from N[2:0] to L[1:7] (sometimes called a *unary code* or a *thermometer code*), using a 3-to-8 decoder with active-high outputs and no more than eight 2-input OR gates.

6.34 Write a Verilog module `Vrbin3una7` for the 3-to-7 binary-to-unary converter in Exercise 6.33, and write a test bench `Vrbin3una7_tb` to check its operation.

6.35 Write a Verilog module `Vrbin3una7_g` for the 3-to-7 binary-to-unary converter described in Exercise 6.33 using a `generate` statement. If you haven't already, write a test bench `Vrbin3una7_tb` that checks your module's operation.

6.36 Based on the description in Exercise 6.33, write a parameterized Verilog module `VrbinBunaM` for a *B*-bit to *M*-bit binary-to-unary converter using a `generate` statement. If *M* is less than the maximum value of a *B*-bit number, your circuit should light all of the LEDs when the value of N[B-1:0] is greater than *M*. Write a test bench `VrbinBunaM_tb` that algorithmically checks the module's operation.

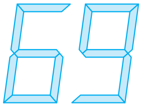


Figure X6.37

6.37 Starting from Program 6-12, write a new Verilog seven-segment-decoder module `Vr7segE` where the digits 6 and 9 have. tails, as shown in Figure X6.37. Also, display the character "E" for nondecimal inputs 1010 through 1111. Check your module with the test bench in Program 6-13.

6.38 Starting from Program 6-12, write a new Verilog module `Vr7segx` for a seven-segment decoder with the following enhancements:

- Two new inputs, ENHEX and ERRDET, control segment-output decoding.
- If ENHEX = 0, the outputs match the behavior of Program 6-12.
- If ENHEX = 1, then the outputs for digits 6 and 9 have tails, and the outputs for digits A–F are controlled by ERRDET.
- If ENHEX = 1 and ERRDET = 0, then the outputs for digits A–F look like the letters A–F, as in shown in Figure X6.38.
- If ENHEX = 1 and ERRDET = 1, then digits A–F look like a question mark without its period, also shown in Figure X6.38.

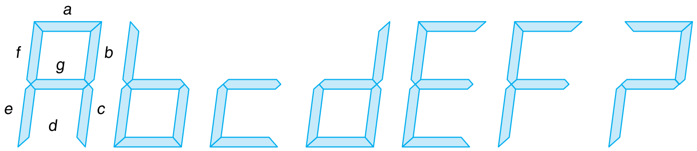


Figure X6.38

6.39 Update and then use the test bench in Program 6-13 to test the enhanced seven-segment decoder of Exercise 6.38.

6.40 Write behavioral Verilog code for a module `Vr1of8check` with eight inputs, I[0:7] and one output, VALID. The output should be 1 if and only if the input is a valid codeword in the 1-out-of-8 code.

6.41 Draw the logic diagram for a 16-to-4 encoder using just four 8-input NAND gates. What are the active levels of the inputs and outputs in your design?

6.42 Design a gate-level circuit 10-to-4 encoder with inputs in the 1-out-of-10 code and outputs in a code like normal BCD except that input lines 8 and 9 are encoded into "E" and "F", respectively.

6.43 Delete the second `if` statement in the test bench of Program 6-18. Then insert a simple typo into the 2-input, 8-bit multiplexer module of Program 6-15 that makes it work incorrectly in some cases, but where the modified test bench doesn't detect the error. *Hint*: Change just one string to a different string.

6.44 Suppose you are working in a technology that implements 4-input multiplexers of any width very efficiently using a native cell, while custom multiplexers are slower and larger. Show how to implement the functionality of Table 6-8 using a 4-input, 18-bit multiplexer in this technology, and a "code converter" with inputs S[2:0] and outputs CC[1:0] such that CC = 00,01,10,11 when S[2:0] selects A,B,C,D, respectively. Write the logic equations for the code converter.

6.45 Write a Verilog module `Vrmux4in18b_cc` with the same inputs as Program 6-17, but uses the code converter of Exercise 6.44. Use a hierarchical coding style that is easy to understand, easy to change if a different conversion pattern is needed, and easy to modify to instantiate the native 4-input multiplexer cells if the synthesis tool fails to infer them automatically.

6.46 Continuing Exercise 6.45, synthesize your `Vrmux4in18b_cc` module as well as the original in Program 6-17, targeting your favorite FPGA. Determine how many LUTs are required in each of the two realizations, and explain the cause of the difference, if any.

6.47 Write a Verilog module `Vrabcdemux` for a customized multiplexer with five 8-bit input buses A, B, C, D, and E, selecting one of the buses to drive a 8-bit output bus T according to Table X6.47. Synthesize the module for your favorite FPGA and determine how many internal resources it uses.

Table X6.47

S2	S1	S0	*Input to Select*
0	0	0	A
0	0	1	B
0	1	0	B
0	1	1	C
1	0	0	C
1	0	1	D
1	1	0	E
1	1	1	A

6.48 Write a Verilog module `Vrabcdemux2` for a customized multiplexer similar to the previous exercise, but selecting which bus to drive T according to Table X6.48. Synthesize the module for your favorite FPGA and determine how many internal resources it uses. Compare with Exercise 6.47 and if the resource requirements are different, explain why.

Table X6.48

S2	S1	S0	*Input to Select*
0	0	0	A
0	0	1	B
0	1	0	A
0	1	1	C
1	0	0	A
1	0	1	D
1	1	0	A
1	1	1	E

6.49 Continuing from Exercise 6.48, rewrite the module hierarchically to create a module `Vrabcdemux3` that uses fewer resources when LUTs have six inputs and up to two outputs as in the configuration of Figure 6-6 for Xilinx 7-series FPGAs. *Hint*: You should be able to get it down to 12 LUTs.

6.50 Write a behavioral Verilog module `Vrpqrtmux` for a customized multiplexer with four 8-bit input buses P, Q, R, T, and three select inputs S[2:0] that choose one of the buses to drive an 8-bit output bus Y according to Table X6.50.

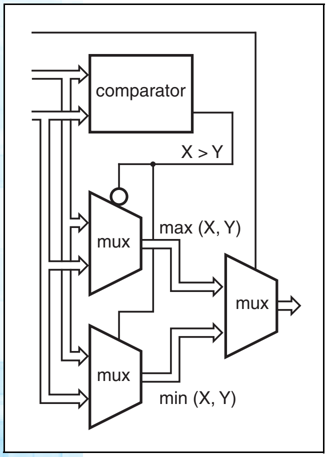
Table X6.50

S2	S1	S0	*Input to Select*
0	0	0	P
0	0	1	Q
0	1	0	Q
0	1	1	P
1	0	0	R
1	0	1	P
1	1	0	R
1	1	1	T

6.51 Repeat Exercise 6.50, creating a new module `Vrpqrtmuxc` with two more control inputs C[1:0] such that the output bus Y is the selected input bus, its complement, all 0s, or all 1s, depending on whether C[1:0] is 00, 01, 10, or 11 respectively.

6.52 Synthesize the module in Exercise 6.51, targeting it to your favorite programmable device, and determine how many internal resources it uses. Then change the selection encoding used in `C[1:0]` and determine whether the resource utilization changes. If it stays the same, try other encodings or coding approaches. Then explain why the resource utilization changes or stays the same.

6.53 The 74H87 is an ancient TTL zero/one/true/complement element having a 4-bit output equal to all 0s, all 1s, its 4-bit input, or the complement of its 4-bit input, depending on the value a 2-bit control input. Write a parameterized Verilog module that does the same thing for an *n*-bit input vector.



chapter 7

More Combinational Building Blocks

This chapter continues our discussion of combinational building blocks. We start with three-state devices that can “disconnect” their outputs from the signal lines that they would otherwise be driving with 0s and 1s. Then we describe priority encoders, just the thing for anyone who wants to “pick a winner.” Next we cover Exclusive OR gates and parity functions, which are essential building blocks for error-detecting and -correcting circuits in digital systems.

We then have a fairly lengthy a discussion of equality and magnitude comparators. You might think of magnitude comparison as being an arithmetic function, which is fair, because two numbers can be compared by subtracting one from the other. However, comparison can also be done without subtraction, and that’s the method we’ll show in this chapter, deferring arithmetic circuits to Chapter 8. Moreover, because comparator circuits can be fairly large, they are nontrivial synthesis examples and they give us an opportunity to study the synthesis results that are obtained when different Verilog models are targeted to FPGAs.

We will close the chapter with a fairly large “random logic” example—large enough that you probably wouldn’t attempt it to do it without the benefit of an HDL like Verilog.

You may prefer to skip ahead to Chapter 9 on state machines, to begin your study of sequential circuits. That’s OK, but the topics here and in Chapter 8 are important, so you should plan to come back at some point.

7.1 Three-State Devices

In Section 14.5.3, we'll describe the electrical design of CMOS devices whose outputs may be in one of three states—0, 1, or high impedance ("Hi-Z"). In the third state, they behave as if they're not even connected to the circuit, except for some small analog effects that we can ignore in digital analysis. This is a good place to introduce three-state devices, because at the printed-circuit-board level, they are a widely used alternative to multiplexers for selecting one of multiple data sources to send to one or more destinations.

7.1.1 Three-State Buffers

three-state buffer
three-state driver

The most basic three-state device is a *three-state buffer*, often called a *three-state driver*. The logic symbols for four physically different three-state buffers are shown in Figure 7-1. The basic symbol is that of a noninverting buffer [(a), (b)] or an inverter [(c), (d)]. The extra signal at the top of the symbol is a *three-state enable* input, which may be active high or active low. When the enable is asserted, a three-state buffer behaves like an ordinary buffer or inverter. When the enable is negated, the device output "floats"; that is, it goes to a high-impedance (Hi-Z) disconnected state and functionally behaves as if it weren't even there.

three-state enable

Three-state devices allow multiple sources to share a single "party line," as long as only one device "talks" on the line at a time. Figure 7-2 gives an example of how this can be done. Three input bits, SSRC2–SSRC0, select one of eight sources of data that may drive a single line, SDATA. A 3-to-8 decoder with active-low outputs ensures that only one of the eight SEL lines is asserted at a time, enabling only one three-state buffer to drive SDATA. However, if the EN line is negated, then none of the three-state buffers is enabled. The logic value on SDATA is undefined in this case.

Typical three-state devices are designed so that they go into the Hi-Z state faster than they come out of the Hi-Z state. In terms of the specifications on a data sheet, t_{pLZ} and t_{pHZ} are both less than t_{pZL} and t_{pZH}. This means that if the outputs of two three-state devices are connected to the same party line, and we simultaneously disable one and enable the other, the first device will get off the party line before the second one gets on. This is important because, if both devices were to drive the party line at the same time, and if both were trying to maintain opposite output values (0 and 1), then excessive current would flow and create noise in the system, as will be discussed in Section 14.5.7. This is often called *fighting*.

fighting

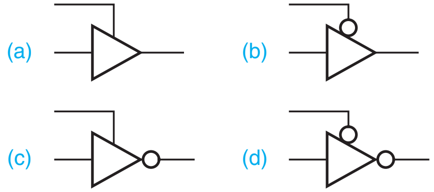


Figure 7-1
Various three-state buffers:
(a,b) noninverting;
(c,d) inverting;
(a,c) active-high enable;
(b,d) active-low enable.

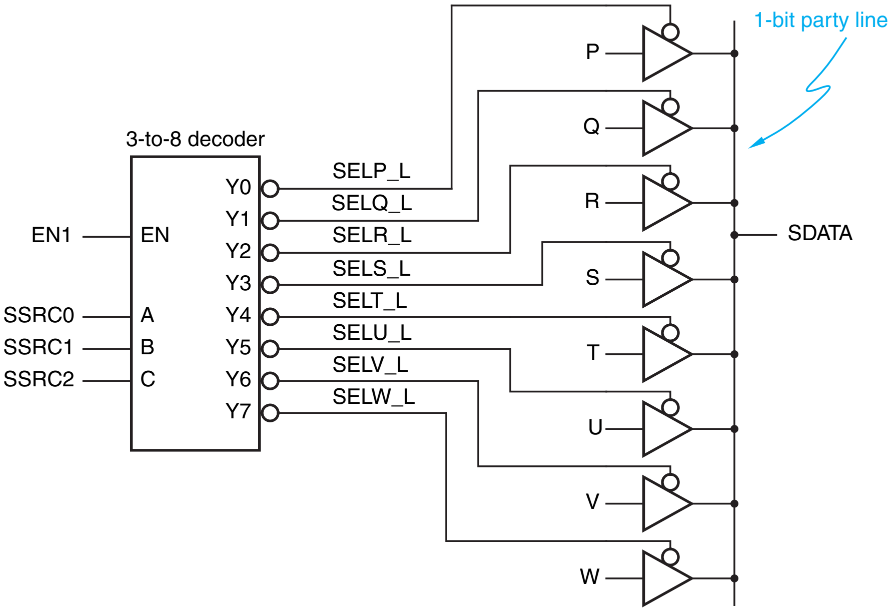


Figure 7-2
Eight sources sharing a three-state party line.

Unfortunately, delays and timing skews in control circuits make it difficult to ensure that the enable inputs of different three-state devices change "simultaneously." Even when this is possible, a problem arises if three-state devices from different-speed logic families (or even different ICs manufactured on different days) are connected to the same party line. The turn-on time (t_{pZL} or t_{pZH}) of a "fast" device may be shorter than the turn-off time (t_{pLZ} or t_{pHZ}) of a "slow" one, and the outputs may still fight.

THREE-STATE DEVICE APPLICATIONS

Three-state outputs are rarely used on-chip, that is, *inside* ASICs and FPGAs. Although multiplexers require more chip area for both gates and wires, using them to select data sources usually provides higher performance than on-chip three-state outputs. Also, they don't suffer from some of the problems of using on-chip three-state devices, like the possibility of excessive power consumption when buses are floating, electrical noise when the driving sources are changing, the inability of many EDA tools to properly model their electrical performance, and difficulties in circuit testing.

Three-state outputs and buses are widely used off-chip, however. On printed-circuit boards, they are almost always used in the interconnections among microprocessors, memories, and all kinds of coprocessors and interface chips, including custom ASICs and FPGAs, which often have one or more three-state output ports. At the system level, three-state is often used in the connections among modules, like the individual network interfaces in a large network router, and the DIMMs that fill the expansion-memory slots in a desktop or laptop computer.

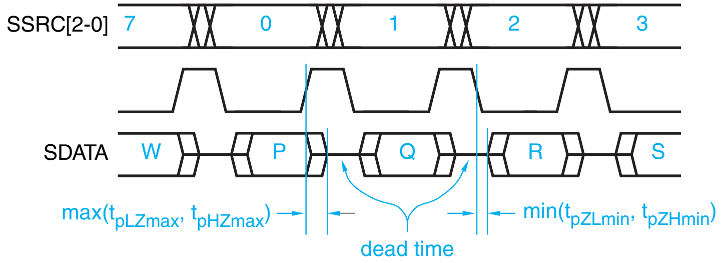


Figure 7-3 Timing diagram for the three-state party line.

dead time

The only really safe way to use three-state devices is to design control logic that guarantees a *dead time* on the party line during which no one is driving it. The dead time must be long enough to account for the worst-case differences between turn-off and turn-on times of the devices, and for skews in the three-state control signals. A timing diagram that illustrates this sort of operation for the party line of Figure 7-2 is shown in Figure 7-3. This timing diagram also illustrates a drawing convention for three-state signals—when in the Hi-Z state, they are shown at an “undefined” level halfway between 0 and 1.

DEFINING “UNDEFINED”

The actual voltage level of a floating signal depends on circuit details, like resistive and capacitive load, and may vary over time. Also, its interpretation by other circuits depends on their characteristics, so it’s best not to count on a floating signal as being anything other than “undefined.”

Sometimes a pull-up resistor is used on three-state party lines to ensure that a floating value is pulled to a HIGH voltage and interpreted as logic 1. Otherwise, CMOS devices whose input voltage is halfway between logic 0 and 1 may consume excessive current. An alternative in CMOS-based systems is to use a “bus-holder,” a sequential circuit that actively holds the last value on the party line when no other device is actively driving it, as described in Section 10.5.2.

*7.1.2 Standard MSI Three-State Buffers

Like logic gates, several independent three-state buffers may be packaged in a single SSI IC, but such chips are rarely used in new designs. Most party-line applications use a bus with more than one bit of data anyway. For example, in an 8-bit microcontroller system, the data bus is eight bits wide, and peripheral devices normally place data on the bus eight bits at a time. Thus, a peripheral device enables eight three-state drivers to drive the bus, all at the same time. Independent enable inputs, as in the application in Figure 7-2, are not necessary.

So, to reduce the package size in wide-bus applications, MSI three-state buffers are typically set up with all of the buffers, or sometimes groups of them, having common enable inputs. For example, Figure 7-4 shows the logic diagram

* Throughout this book, optional sections are marked with an asterisk.

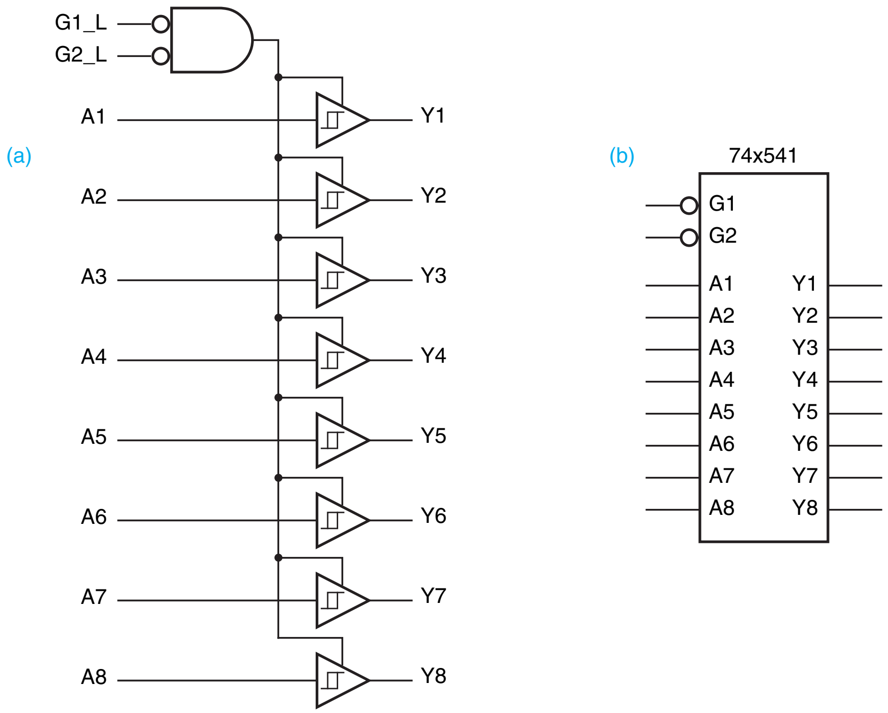


Figure 7-4
The 74x541 octal three-state buffer: (a) logic diagram; (b) traditional logic symbol.

and symbol for a *74x541* octal noninverting three-state buffer. *Octal* means that the part contains eight individual buffers. Both enable inputs, G1_L and G2_L, must be asserted to enable the device's three-state outputs. The little rectangular symbols inside the buffer symbols indicate *hysteresis*, an electrical characteristic of the inputs that improves noise immunity, as we'll explain in Section 14.5.2. The 74x541 inputs typically have up to 0.4 volts of hysteresis.

74x541
octal
hysteresis

Many other varieties of octal three-state buffers are available. For example, the *74x540* is identical to the 74x541 except it contains inverting buffers. There are also 16-bit and even 32-bit three-state buffers, such as the *74x16541* and the *74x32244*, respectively. The first part has the same functionality as two 74x541s in one package, while the second has a separate enable input for each of eight independent groups of four bits. These parts come in larger packages with more pins, of course.

74x540
74x16541
74x32244

Figure 7-5 on the next page shows part of a microcontroller system with an 8-bit data bus, DB[0–7], and a 74x541 used as an input port. The microcontroller selects Input Port 1 by asserting INSEL1, and it requests a read operation by asserting READ. The selected 74x541 responds by driving the microcontroller data bus with user-supplied input data. Other input ports may be selected when a different INSEL line is asserted along with READ.

A *bus transceiver* has pairs of three-state buffers connected in opposite directions between each pair of pins, which are now "I/O" pins where data can transfer in either direction. For example, Figure 7-6 shows the logic diagram and

bus transceiver

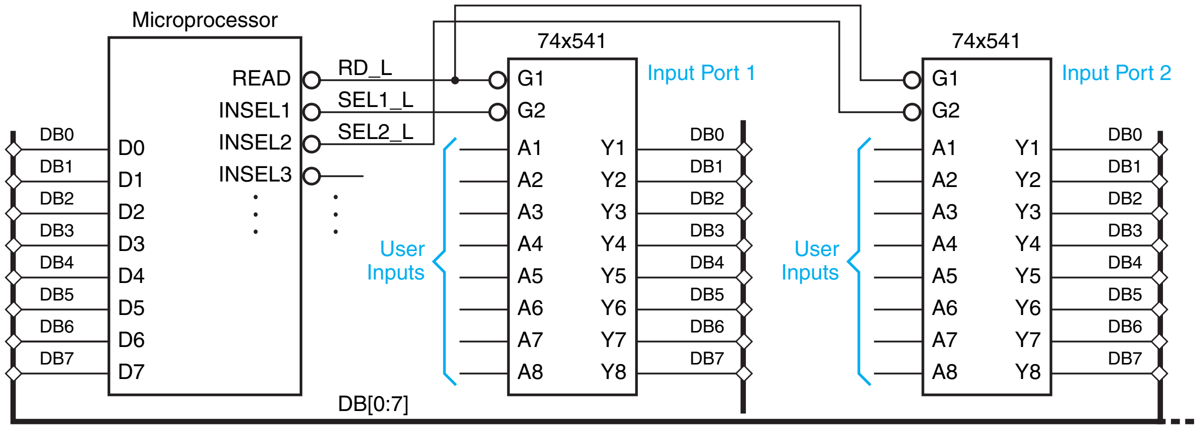


Figure 7-5 Using a 74x541 as a microprocessor input port.

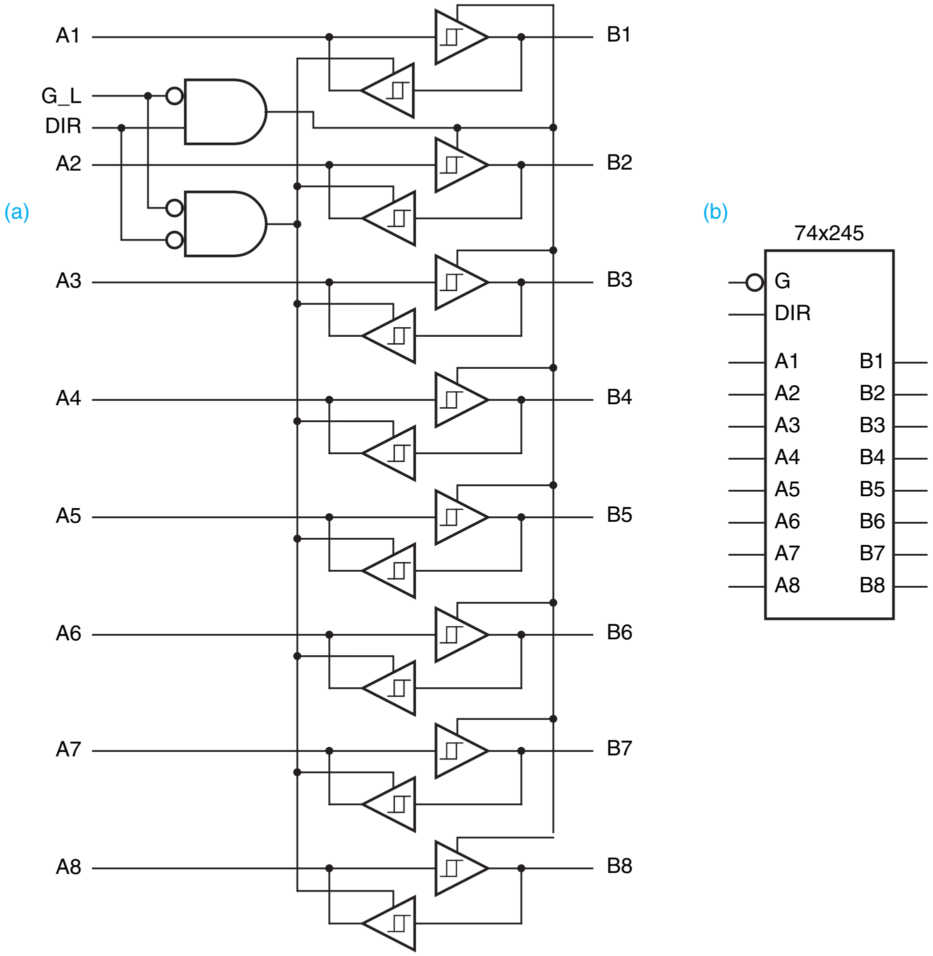


Figure 7-6
The 74x245 octal three-state transceiver:
(a) logic diagram;
(b) traditional logic symbol.

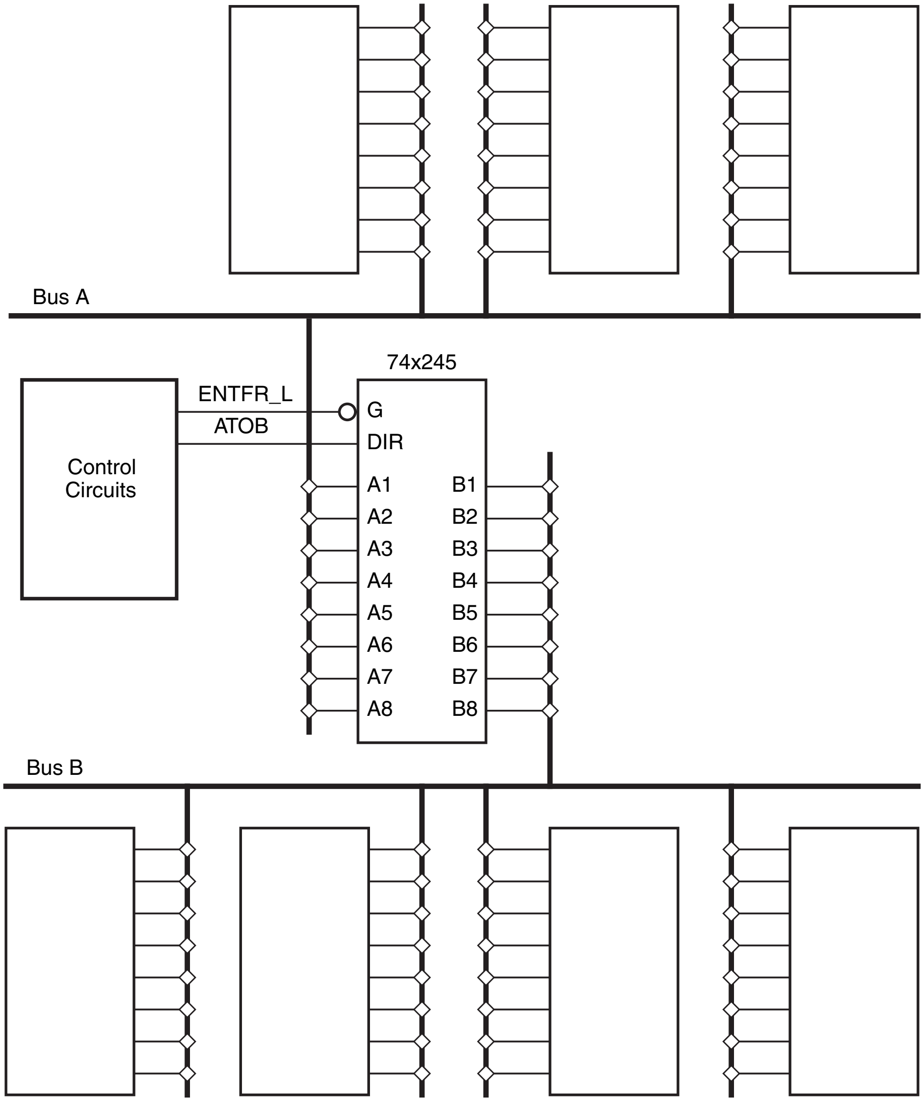


Figure 7-7
Bidirectional buses and transceiver operation.

74x245

symbol for a *74x245* octal three-state transceiver. The DIR input determines the direction of transfer, from A to B (DIR = 1) or from B to A (DIR = 0). The three-state buffer for the selected direction is enabled only if G_L is asserted.

bidirectional bus

A bus transceiver is typically used between two *bidirectional buses*, as shown in Figure 7-7. Three different modes of operation are possible, depending on the values of G_L and DIR, as shown in Table 7-1 on the next page. As usual, it is the designer's responsibility to ensure that neither bus is ever driven simultaneously by two devices. However, independent transfers where both buses are driven at the same time may occur when the transceiver is disabled, as indicated in the last row of the table.

Table 7-1 Modes of operation for a pair of bidirectional buses.

ENTFR_L	ATOB	*Operation*
0	0	Transfer data from a source on bus B to a destination on bus A.
0	1	Transfer data from a source on bus A to a destination on bus B.
1	x	Transfer data on buses A and B independently.

7.1.3 Three-State Outputs in Verilog

Verilog has built-in bit-data value ‘z’ for the high-impedance state, so it is very easy to specify three-state outputs. For example, Program 7-1 shows a Verilog module for an 8-bit noninverting three-state buffer similar to the 74x541. With the conditional operator (?:), it takes just one continuous-assignment statement to specify the output—a copy of the input if the device is enabled, and eight bits of “z” otherwise.

The Vr74x541 module uses its three-state port for output only, but output ports can be used as inputs as well if they are declared as type “inout”. This capability can be used in a transceiver application with functionality similar to the MSI 74x245 transceiver in Figure 7-6 on page 306. The Verilog version is shown in Program 7-2.

Another example Verilog application of inout ports is in a four-way, 8-bit bus transceiver with the following specifications:

- The transceiver handles four 8-bit bidirectional buses, A[1:8], B[1:8], C[1:8], and D[1:8].
- Each bus has its own active-low output enable input, AOE_L–DOE_L, and a “master” enable MOE_L must also be asserted for any bus to be driven.
- The same source of data is driven to all the buses, as selected by three select inputs, S[2:0]. If S2 is 0, the buses are driven with a constant value equal to the low-order select inputs S[1:0], replicated four times. If S2 is 1, they are driven with one of the other buses A–D, depending on the value of S[1:0], 00–11.
- When the selected source is a bus, the selected source bus cannot be driven, even if it is output-enabled.

Program 7-1 Verilog module for a 74x541-like 8-bit three-state driver.

```
module Vr74x541(G1_L, G2_L, A, Y);
  input G1_L, G2_L;
  input [1:8] A;
  output [1:8] Y;

  assign Y = (~G1_L & ~G2_L) ? A : 8'bz;
endmodule
```

ZZZZzzzz . . .” Recall that while Verilog sized literals like 8'b1 are normally padded on the left with 0s, if the leftmost specified bit is x or z, that value is used for the padding.

Program 7-2 Verilog module for a 74x245-like 8-bit transceiver.

```
module Vr74x245(G_L, DIR, A, B);
  input G_L, DIR;
  inout [1:8] A, B;

  assign A = (~G_L & ~DIR) ? B : 8'bz;
  assign B = (~G_L &  DIR) ? A : 8'bz;
endmodule
```

This functionality is provided by the Verilog module in Program 7-3, using a mix of procedural and continuous assignments. The procedural assignments appear in an always block that sets an internal variable, ibus, to the value that should be driven onto any output-enabled port. Notice the use of concatenation to make four copies of the two low-order bits of S[2:0] when a constant source is selected. The continuous assignments at the end of the module drive the output buses when enabled, and include logic to ensure that the selected source bus is not driven, even if otherwise output-enabled.

Program 7-3 Verilog module for a four-way, 8-bit bus transceiver.

```
module VrXcvr4x8(A,B,C,D, S, AOE_L, BOE_L, COE_L, DOE_L, MOE_L);
  input [2:0] S;
  input AOE_L, BOE_L, COE_L, DOE_L, MOE_L;
  inout [1:8] A, B, C, D;
  reg [1:8] ibus;

  always @ (A or B or C or D or S) begin
    if (S[2] == 0) ibus = {4{S[1:0]}};
    else case (S[1:0])
      2’b00: ibus = A;
      2’b01: ibus = B;
      2’b10: ibus = C;
      2’b11: ibus = D;
    endcase
  end

  assign A = ((~AOE_L & ~MOE_L) && (S[2:0]!=3'b100)) ? ibus:8'bz;
  assign B = ((~BOE_L & ~MOE_L) && (S[2:0]!=3'b101)) ? ibus:8'bz;
  assign C = ((~COE_L & ~MOE_L) && (S[2:0]!=3'b110)) ? ibus:8'bz;
  assign D = ((~DOE_L & ~MOE_L) && (S[2:0]!=3'b111)) ? ibus:8'bz;
endmodule
```

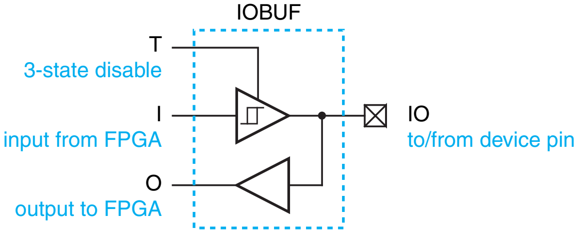


Figure 7-8 IOBUF input/output buffer component.

7.1.4 Three-State Outputs in FPGAs

While some older FPGA devices provided three-state elements to drive internal buses, modern FPGAs do not. Selecting one of multiple sources to drive an internal bus is done with multiplexers, as described in Section 6.4. However, all FPGA, CPLD, and ASIC libraries provide three-state input/output cells for driving external pins. Three-state buses are still used quite commonly in board-level design to minimize the wiring needed for multiple components (such as microprocessors, memories, and input/output interfaces) to communicate with each other, as in the examples of Figures 7-5 and 7-7.

Figure 7-8 shows an input/output buffer cell in a typical FPGA. In Xilinx libraries, this cell is a predefined component named IOBUF. It contains an input buffer whose output is on the lefthand side of the diagram and is named “O”. It also contains a three-state buffer with input “I” and three-state disable input “T”. (The output is Hi-Z when T is 1.) The signal on the righthand side of the component is named “IO” and connects directly to an I/O pin of the FPGA IC package.

Like any other library component, an FPGA IOBUF may be instantiated explicitly using a instance statement. (Note that this will lead to a synthesis error if the signal connected to IO has not been defined to be an external pin.) The synthesis engine is also able to “infer” an IOBUF if an external three-state output is specified in procedural code, as in our previous examples, if each is specified to be the “top-level” module in the design.

Modules targeted to be used inside FPGAs and ASICs do not use three-state outputs, and they instead define separate buses for their inputs and their outputs, as in Program 7-4. However, sometimes we may need to use an existing “internal” module design and hook up its inputs and outputs directly to an exter-

Program 7-4 Declarations in module with 8-bit input and output buses.

```
module VrmyModule(CLK, I1, I2, IBUS, O1, O2, OBUS);
  input CLK, I1, I2;
  input [7:0] IBUS;
  output O1, O2;
  output [7:0] OBUS;
  ...
```

GENERATE BLOCKS

Verilog-2001 supports the creation of *generate blocks* which create a structural or dataflow model using algorithmic statements. A generate block begins with the keyword `generate` and ends with `endgenerate`. Within the block, `if`, `case`, and `for` statements may be used to control whether or not other statements are executed.

The most common examples of generate blocks use an iterative loop (`for`) to create a repetitive hardware structure, which is how we use it in this book. Such a `for` loop must be controlled by a new integer variable type (`genvar`), and its parenthesized control list is typically followed by a named `begin-end` block containing one or more instance and continuous-assignment statements. Using the block's name, the compiler can generate unique component identifiers and, if needed, net names for all instances and nets that are created within the loop, so they can be tracked during simulation and synthesis.

Program 7-5 Top-level Verilog module wrapped around `VrmyModule`.

```
module VrmyDesign_top(CLK, IN1, IN2, OUT1, OUT2, IOBUS, IOBUS_OE);
  input CLK, IN1, IN2, IOBUS_OE;
  inout [7:0] IOBUS;
  output OUT1, OOT2;
  wire [7:0] INBUS, OUTBUS;
  genvar g;

  Vrmymodule U1 (.CLK(CLK), .I1(IN1), .I2(IN2), .O1(OUT1),
                 .O2(OUT2), .IBUS(INBUS), .OBUS(OUTBUS) );
  generate
    for (g=0; g<=7; g=g+1) begin: io
      IOBUF U2 (.I(OUTBUS[g]), .O(INBUS[g]),
                .IO(IOBUS[g]), .T(~IOBUS_OE) );
    end;
  endgenerate
endmodule
```

nal three-state bus. Instead of modifying the module code to use three-state outputs, we can hook it up to an external three-state bus by "wrapping" it in a top-level module like Program 7-5. The top-level module instantiates the original module as-is, and then uses a generate block (see box) to instantiate eight `IOBUF` cells for the external I/O. Internal wires `IBUS` and `OBUS` are declared to make the connections between `VrmyModule` and the `IOBUF` cells.

Figure 7-9 shows a schematic of the resulting synthesized design as it is generated by the tools but omitting the I/O buffers for bits 1–6. We've also expanded the topmost IOBUF component to show what's inside—just what you would expect: an input buffer and a three-state output buffer.

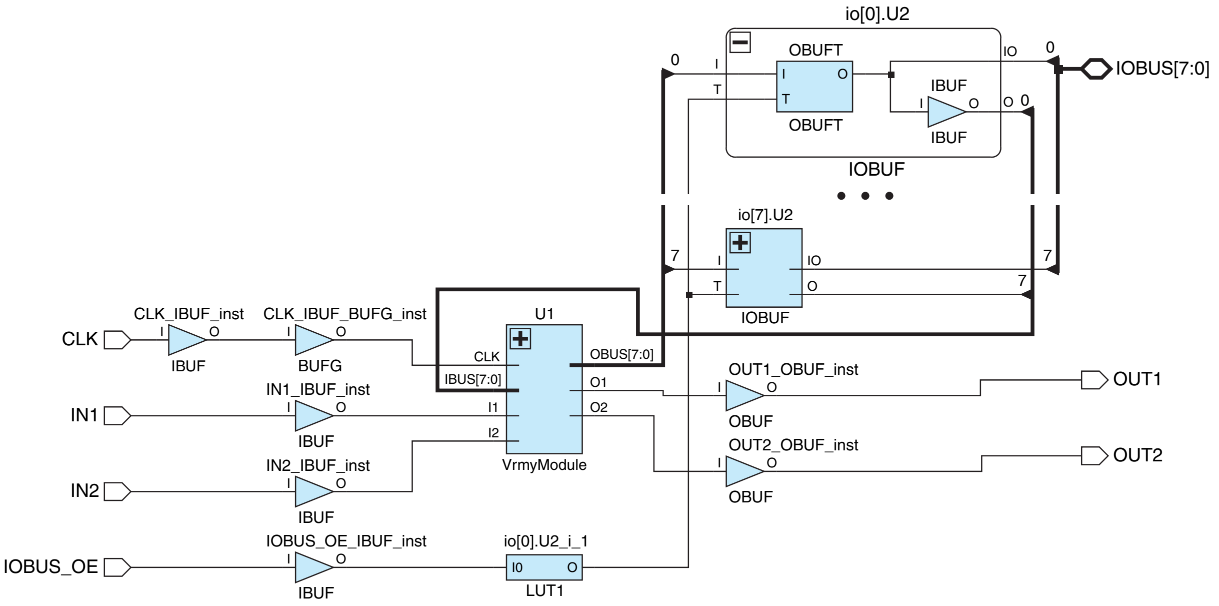


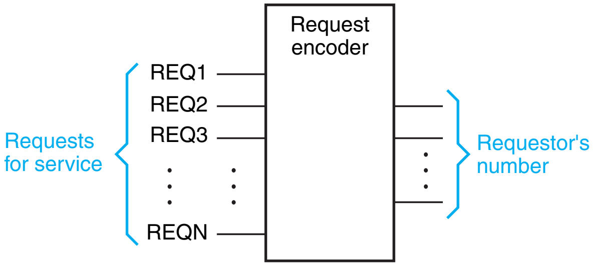
Figure 7-9 Schematic diagram of the `VrmyDesign_top` module as synthesized.

7.2 Priority Encoding

In typical shared-bus systems like the examples in the preceding section, different devices may request to drive the bus at different times, and some mechanism must be provided to ensure that only one at a time gets access. In other applications, multiple entities may request a resource or service that can be granted to only one entity at a time; in a microprocessor input/output subsystem, these might be interrupt requests. In these systems and applications, there are typically up to 2^n inputs, each of which indicates a request for service (as in Figure 7-10) and it is quite possible for multiple requests to be made simultaneously.

We saw in Section 6.3.7 (binary encoders) how an asserted signal on one out of 2^n inputs could be easily converted into the corresponding binary number, but what if multiple inputs are asserted simultaneously? The solution is to assign

Figure 7-10 A system with 2^n requestors, and a "request encoder" that indicates which request signal is asserted at any time.



priority to the input lines, so when multiple requests are asserted, the identifying number of the highest-priority requestor is output. A device that does this is called a *priority encoder.*

priority

priority encoder

The logic symbol for an 8-input priority encoder is shown in Figure 7-11. Input I7 is defined to have the highest priority; others have lower priority in decreasing numerical order. Outputs A2–A0 contain the number of the highest-priority asserted input, if any. The IDLE output is asserted if no inputs are asserted.

Priority encoders can be specified quite easily and naturally using some of the language constructs in HDLs like Verilog, but for better understanding, let's look at them first using logic equations. To develop equations for the priority encoder's outputs, we first define eight intermediate variables H0–H7, such that Hn is 1 if and only if input In is the highest priority 1 input:

$$\begin{aligned}
\mathsf{H7} &= \mathsf{I7} \\
\mathsf{H6} &= \mathsf{I6} \cdot \mathsf{I7}' \\
\mathsf{H5} &= \mathsf{I5} \cdot \mathsf{I6}' \cdot \mathsf{I7}' \\
&\dots \\
\mathsf{H0} &= \mathsf{I0} \cdot \mathsf{I1}' \cdot \mathsf{I2}' \cdot \mathsf{I3}' \cdot \mathsf{I4}' \cdot \mathsf{I5}' \cdot \mathsf{I6}' \cdot \mathsf{I7}'
\end{aligned}$$

Note that because of the way these signals are defined, at most one of them can be asserted at any time. Using them, the equations for the A2–A0 outputs are similar to the ones for a simple binary encoder:

$$\begin{aligned}
\mathsf{A2} &= \mathsf{H4} + \mathsf{H5} + \mathsf{H6} + \mathsf{H7} \\
\mathsf{A1} &= \mathsf{H2} + \mathsf{H3} + \mathsf{H6} + \mathsf{H7} \\
\mathsf{A0} &= \mathsf{H1} + \mathsf{H3} + \mathsf{H5} + \mathsf{H7}
\end{aligned}$$

The IDLE output is 1 if no inputs are 1:

$$\begin{aligned}
\mathsf{IDLE} &= (\mathsf{I0} + \mathsf{I1} + \mathsf{I2} + \mathsf{I3} + \mathsf{I4} + \mathsf{I5} + \mathsf{I6} + \mathsf{I7})' \\
&= \mathsf{I0}' \cdot \mathsf{I1}' \cdot \mathsf{I2}' \cdot \mathsf{I3}' \cdot \mathsf{I4}' \cdot \mathsf{I5}' \cdot \mathsf{I6}' \cdot \mathsf{I7}'
\end{aligned}$$

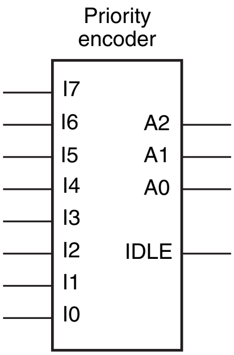
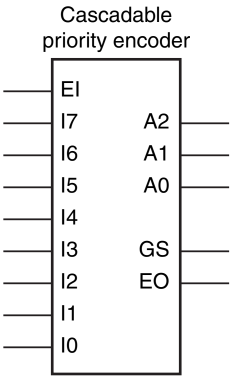


Figure 7-11 Logic symbol for a generic 8-input priority encoder.

Figure 7-12
Logic symbol for a cascadable 8-input priority encoder.



7.2.1 Cascading Priority Encoders

cascade

cascadable priority encoder

It is straightforward to extend the equations and approach on the previous page to create a priority encoder with any desired number of inputs. However, there may be occasions where the inputs are distributed among two or more subsystems, and the subsystems themselves are to be arranged in priority order, so the highest priority active input in the highest priority subsystem is recognized. In this situation, it is possible to combine or *cascade* the information among multiple subsystems, using an individual *cascadable priority encoder* for the inputs in each subsystem, and then combining their outputs.

Figure 7-12 is the logic symbol for an 8-input cascadable priority encoder that could be used within each subsystem. Besides the usual request inputs I7–I0 and outputs A2–A0, the device has an enable input EI, an enable output EO, and a “group select” output GS. The complete truth table of this device is given in Table 7-2.

The EI input must be asserted for any of the outputs to be asserted. The GS output is asserted when the device is enabled and one or more of the request inputs are asserted. The EO output is used for cascading—it is designed to be connected to the EI input of another device that handles lower-priority requests. EO is asserted if EI is asserted but no request input is asserted, thus enabling the lower-priority device.

Figure 7-13 on page 316 shows how four of these cascadable priority encoders can be connected to accept 32 request inputs and produce a 5-bit output, RA4–RA0, indicating the highest-priority requestor. Since the A2–A0 outputs of at most one device may be asserted at any time, the outputs of the individual devices can be ORed to produce RA2–RA0. Likewise, the individual GS outputs can be combined in a 4-to-2 encoder to produce RA4 and RA3. The RGS output is asserted if any GS output is asserted.

Table 7-2 Truth table for a cascadable 8-input priority encoder.

Inputs									*Outputs*				
EI	I0	I1	I2	I3	I4	I5	I6	I7	A2	A1	A0	GS	EO
0	x	x	x	x	x	x	x	x	0	0	0	0	0
1	x	x	x	x	x	x	x	1	1	1	1	1	0
1	x	x	x	x	x	x	1	0	1	1	0	1	0
1	x	x	x	x	x	1	0	0	1	0	1	1	0
1	x	x	x	x	1	0	0	0	1	0	0	1	0
1	x	x	x	1	0	0	0	0	0	1	1	1	0
1	x	x	1	0	0	0	0	0	0	1	0	1	0
1	x	1	0	0	0	0	0	0	0	0	1	1	0
1	1	0	0	0	0	0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	0	0	0	0	0	0	1

7.2.2 Priority Encoders in Verilog

There are many ways to model the behavior of priority encoders in Verilog. One way is to use a nested series of Verilog `if` statements, as shown in Program 7-6. This nicely matches with our understanding of the priority encoder's behavior. However, this approach is unwieldy and error-prone if there are a lot of inputs.

Program 7-6 Verilog module for an 8-input priority-encoder, using nested `ifs`.

```
module Vr8inprior2(I, A, IDLE);
  input [7:0] I;
  output reg [2:0] A;
  output reg IDLE;

  always @ (*) begin
    IDLE = 0;
    if (I[7]) A = 3'd7;
    else if (I[6]) A = 3'd6;
         else if (I[5]) A = 3'd5;
            else if (I[4]) A = 3'd4;
                 else if (I[3]) A = 3'd3;
                      else if (I[2]) A = 3'd2;
                           else if (I[1]) A = 3'd1;
                                else if (I[0]) A = 3'd0;
                                     else begin A = 3'd0; IDLE = 1; end;
  end
endmodule
```

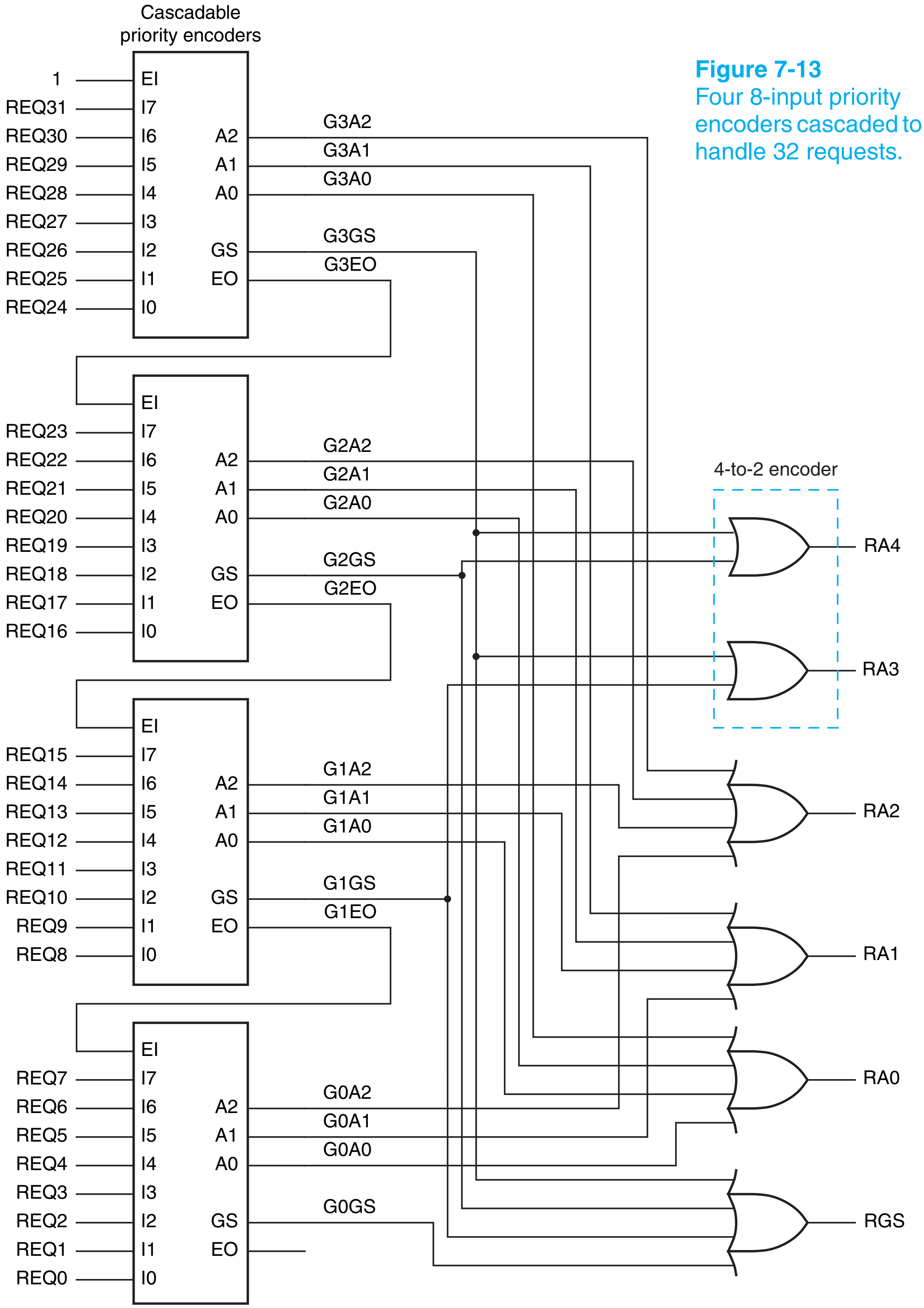


Figure 7-13
Four 8-input priority encoders cascaded to handle 32 requests.

Program 7-7 Verilog module with a `for` loop for an 8-input priority encoder.

```
module Vr8inprior3(I, A, IDLE);
  input [7:0] I;
  output reg [2:0] A;
  output reg IDLE;
  integer j;

  always @ (*) begin
    IDLE = 1; A = 0;       // default output values
    for (j=0; j<=7; j=j+1) // check low priority first
                 if (I[j]==1) begin IDLE = 0; A = j; end
  end
endmodule
```

Another possible behavioral model of a priority encoder uses a `for` loop, as shown in Program 7-7. The first two statements in the `always` block initialize the outputs as if no asserted input will be found. Then the `for` loop looks for an asserted input, working from the lowest priority to the highest. In the end, `A` will be set to the number of the last (and therefore highest priority) asserted input that was found, if any. This module is easily modified to use a different priority order or a different number of inputs, or to add more functionality such as finding a second-highest-priority input, as requested in Exercise 7.27.

This is a good place to mention priority in the Verilog `case` statement: it has a prioritizing behavior built-in because it finds the *first* one of its choices that matches the selection expression's value, and then executes the corresponding procedural statement. Thus, a `case` statement can specify a priority encoder as shown in Program 7-8. Two aspects of the `case` statement are worth noting:

- The selection expression is a literal, `1'b1`. This may seem a little odd, but it's perfectly legal. The first choice that matches it is executed.
- Verilog evaluates the choices in exactly the order in which they're written. To make `I[0]` the highest priority, reverse the order of the statements. You can even scramble the priority order, which may be very useful when the inputs are named functionally, not numerically, providing excellent documentation of the priority of the named functions.

UPS AND DOWNS Another possible strategy for the `for` loop in Program 7-7 would be to start with the highest-priority input (`I[7]`) and search down until an asserted input is found. Once one was found, a `disable` statement would be used to exit the `for` loop, so `A` would be set to the number of the first (and therefore highest-priority) asserted input. However, the Verilog `disable` statement is not supported by all synthesis tools, while the version in Program 7-7 always works.

Program 7-8 Verilog module for an 8-input priority encoder using `case`.

```
module Vr8inprior4(I, A, IDLE);
  input [7:0] I;
  output reg [2:0] A;
  output reg IDLE;

  always @ (*) begin
  IDLE = 1; A = 0;        // default output values
    case (1'b1)
      I[7]: begin IDLE = 0; A = 7; end  // Highest priority
      I[6]: begin IDLE = 0; A = 6; end  //   (as first choice
      I[5]: begin IDLE = 0; A = 5; end  //    in case statement)
      I[4]: begin IDLE = 0; A = 4; end
      I[3]: begin IDLE = 0; A = 3; end
      I[2]: begin IDLE = 0; A = 2; end
      I[1]: begin IDLE = 0; A = 1; end
      I[0]: begin IDLE = 0; A = 0; end
    endcase
  end
endmodule
```

A self-checking test bench for the 8-input priority encoders is shown in Program 7-9. It loops through all 256 possible combinations of the inputs and has a single `if` statement that checks multiple potential error conditions for each combination, displaying the input combination and outputs when there's an error. The first two conditions check the value of `IDLE`, which should be 1 when the input vector `I` is zero. The next two check the value of `A` when `I` is nonzero.

The conditions checked by the test bench make use of the fortuitous relationship between the bit-numbering and numerical value of `I` and the definition of `A`. For a given value of `A`, input bit `A` is 1, so the integer value of `I` must be at least 2**`A` (2^A). However, for `A` to be the highest priority bit, no higher numbered bit may be set, so the integer value of `I` must be less than 2**`(A+1)`. The logic in these conditions is quite different from what is modeled in any of the priority-encoder modules, which is actually good. If the test bench simply parrotted the same condition tests that are used in the module(s), then it could easily miss any errors in the designer's thinking. (But see Exercise 7.17.)

MISSING AN ERROR

The test bench in Program 7-9 doesn't catch errors where `A` contains any `x` or `z` bits. If that happens, the two comparisons involving `I` and `A` each return a value of `x`, which is not considered true, so the error is not counted.

So, the list of error cases needs one more to detect whether `A` contains any `x` or `z` bits. An easy way to check `A` is with the expression "`^A===1'bx`". If any bit of `A` is `x` or `z`, the reduction XOR operator will return a value of `x`.

Program 7-9 Test bench for the 8-input priority-encoder modules.

```
`timescale 1 ns / 100 ps
module Vr8inprior_tb();
  reg [7:0] I;
  wire [2:0] A;
  wire IDLE;
  integer ii, errors;

  Vr8inprior1 UUT ( .I(I), .A(A), .IDLE(IDLE) );

  initial begin
    errors = 0;
    for (ii=0; ii<256; ii=ii+1) begin
      I = ii;
      #10 ;
      if (                                   // Identify all error cases
           ( (I==8'b0) && (IDLE!=1'b1) )     // Should be idle
        || ( (I>8'b0)  && (IDLE==1'b1) )     // Should not be idle
        || ( (I>8'b0)  && (I<2**A)     )     // I should be at least 2**A
        || ( (I>8'b0)  && (I>=2**(A+1) ) ) ) //   but less than 2**(A+1)
        begin
          errors = errors+1;
          $display("Error: I=%b, A=%b, IDLE=%b", I, A, IDLE);
        end
      end
    $display("Test done, %d errors\n",errors);
  end
endmodule
```

GET OFF MY CASE!

There's still another way to use the priority of Verilog's case statements to model a priority encoder, this time using casez. I put the description in this box so you can ignore it if you've already had enough!

casez

The occasionally used casez statement allows "don't-cares" to be used in its choices; a don't-care bit is denoted by "?". The 8-input priority encoder is modeled using casez in Program 7-10. This case statement feels a little more natural when the input I, rather than a constant, is used in the selection expression. But like Program 7-8, it still has the requirement (and the documentation benefit) that the choices are written in the order of their priority. So, writing and understanding either version requires you to remember the prioritizing behavior that is built into the Verilog case statements. Still, my favorite way to model a priority encoder using a case statement does not have this requirement; you are invited to discover it in Exercise 7.19.

Program 7-10 Verilog module an 8-input priority encoder using `casez`.

```
module Vr8inprior5(I, A, IDLE);
input [7:0] I;
  output reg [2:0] A;
  output reg IDLE;

always @ (*) begin
  IDLE = 1; A = 0;       // default output values
  casez(I)
      8'b1???????: begin IDLE = 0; A = 7; end
      8'b?1??????: begin IDLE = 0; A = 6; end
      8'b??1?????: begin IDLE = 0; A = 5; end
      8'b???1????: begin IDLE = 0; A = 4; end
      8'b????1???: begin IDLE = 0; A = 3; end
      8'b?????1??: begin IDLE = 0; A = 2; end
      8'b??????1?: begin IDLE = 0; A = 1; end
      8'b???????1: begin IDLE = 0; A = 0; end
    endcase
  end
endmodule
```

7.3 Exclusive-OR Gates and Parity Functions

This section introduces Exclusive OR (XOR) and related functions, which are important in four primary applications:

- *Comparison*. An XOR gate can compare two bits for equality, and the outputs of multiple gates may be combined to perform multibit equality comparisons.
- *Parity generation and checking*. A multibit XOR function calculates the "sum modulo 2" or parity of its inputs, providing a way to detect and correct errors in data transmission and storage as explained in Section 2.15.
- *Addition*. XOR functions are used to form the sum bits in addition.
- *Counting*. Sequential circuits called binary counters use XOR to form the next value of each bit when counting, either as part of a T flip-flop or as an explicit function in their next-state logic.

We give examples of the first two applications in this section, and the third in Chapter 8. We'll see them again in T flip-flops in Section 10.2.6, and in binary counters in Section 11.1.3.

7.3.1 Exclusive-OR and Exclusive-NOR Gates

Exclusive OR (XOR)

An *Exclusive-OR (XOR)* gate is a 2-input gate whose output is 1 if exactly one of its inputs is 1. It gets its name because it's an OR function that *excludes* the case where both inputs are 1. Stated another way, an XOR gate produces a 1 out-

X	*Y*	*X ⊕ Y* *(XOR)*	*(X ⊕ Y)′* *(XNOR)*
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

Table 7-3
Truth table for XOR and XNOR functions.

put if its inputs are different. An *Exclusive NOR (XNOR)* or *Equivalence* gate is just the opposite—it produces a 1 output if its inputs are the same. A truth table for these functions is shown in Table 7-3. The XOR operation is sometimes denoted by the symbol “⊕”, that is,

Exclusive NOR (XNOR)
Equivalence

⊕

$$X \oplus Y = X' \cdot Y + X \cdot Y'$$

While Exclusive OR is not one of switching algebra’s basic functions, discrete XOR gates are often used as a component in larger functions like parity trees and comparators, as we’ll see later.

Most switching technologies cannot perform the XOR function directly; instead, they use multigate designs like the ones shown in Figure 7-14(a) and (b). The design in (c) or a variant of it often appears in CMOS ASIC device libraries, since the 2-input, 1-bit multiplexer can be implemented with a small number of transistors configured as a pair of CMOS transmission gates to pass

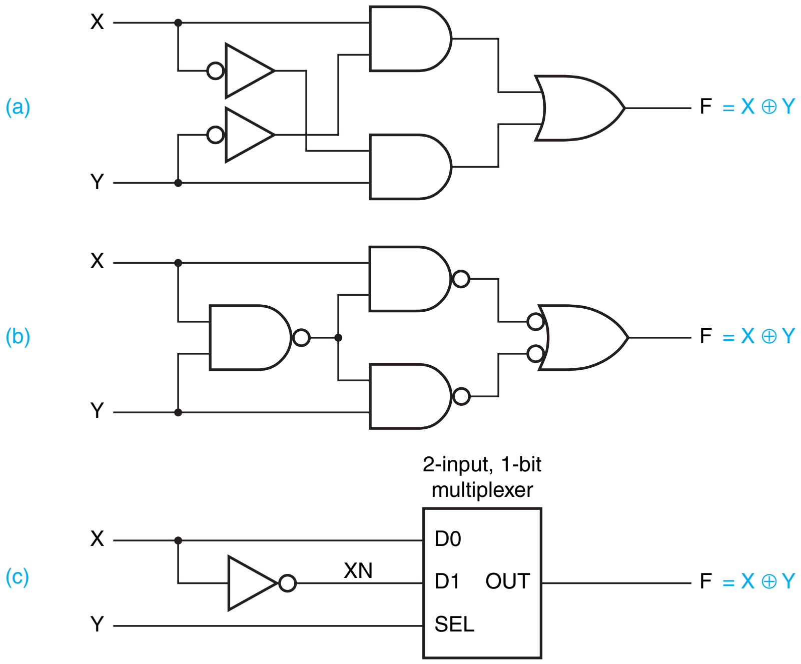


Figure 7-14
Multigate designs for the 2-input XOR function: (a) AND-OR; (b) three-level NAND; (c) multiplexer-based.

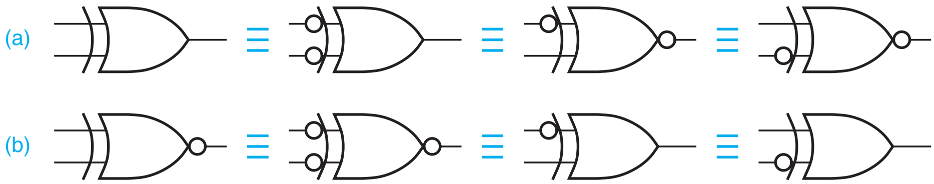


Figure 7-15 Equivalent symbols for (a) XOR gates; (b) XNOR gates.

either the true or the complemented value of X, depending on whether Y is 0 or 1, respectively (see Exercise 7.32).

The logic symbols for XOR and XNOR functions are shown in Figure 7-15. There are four equivalent symbols for each function. All of these alternatives are a consequence of a simple rule:

- Any two signals (inputs or output) of an XOR gate or XNOR gate may be complemented without changing the resulting logic function.

In bubble-to-bubble logic design, we can choose the symbol that is most expressive of the logic function being performed.

PLDs and CPLDs typically connect each output to an XOR gate whose other input is programmable, for output-polarity selection. Also, many FPGAs provide XOR gates in their configurable logic blocks for polarity selection of clock and reset inputs, typically implementing them as shown in Figure 7-14(c) and using the SEL input for programming. The 2-input multiplexer is typically implemented using transmission gates as in Figure 6-27 for high speed. The logic blocks in some of these devices also include XOR gates whose inputs are product terms or LUT outputs to support efficient realizations of adders and counters. XOR and XNOR gates are also readily available in FPGA and ASIC component libraries and as primitives in HDLs.

7.3.2 Parity Circuits

odd-parity circuit

parity tree

even-parity circuit

As shown in Figure 7-16(a), n XOR gates may be cascaded to form a circuit with $n + 1$ inputs and a single output. This is called an *odd-parity circuit*, because its output is 1 if an odd number of its inputs are 1. The circuit in (b) is also an odd-parity circuit, but it's faster because its gates are arranged in a treelike structure, sometimes called a *parity tree*, yielding a shorter worst-case delay path. If the output of either circuit is inverted, we get an *even-parity circuit*, whose output is 1 if an even number of its inputs are 1.

7.3.3 Parity-Checking Applications

In Section 2.15, we described error-detecting codes that use an extra bit, called a parity bit, to detect errors in the transmission and storage of data. In an even-parity code, the parity bit is chosen so that the total number of 1 bits in a code word is even. Parity circuits are used both to generate the correct value of the parity bit when a code word is stored or transmitted, and to check the parity bit when a code word is retrieved or received.

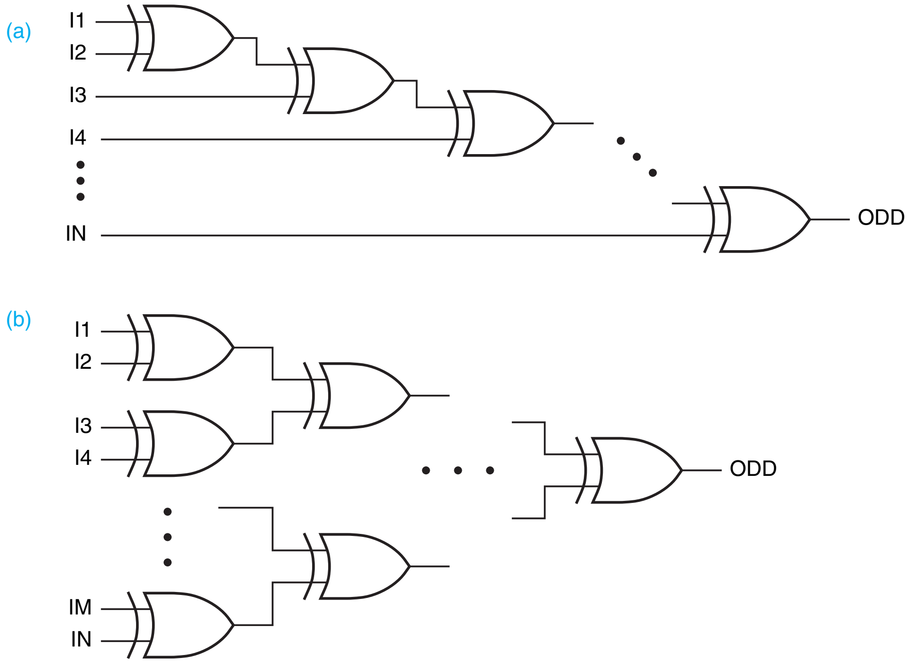


Figure 7-16 Cascading structures for XOR gates: (a) daisy chain; (b) tree.

Figure 7-17 shows how parity circuits might be used to detect errors in the memory of a microprocessor system. The memory stores 8-bit bytes, plus a parity bit for each byte. The memory chips have two separate buses DATAIN[0:7] and DATAOUT[0:7] for data transfers in and out, respectively. Two control lines, RD and WR, are used to indicate whether a read or write operation is desired, and

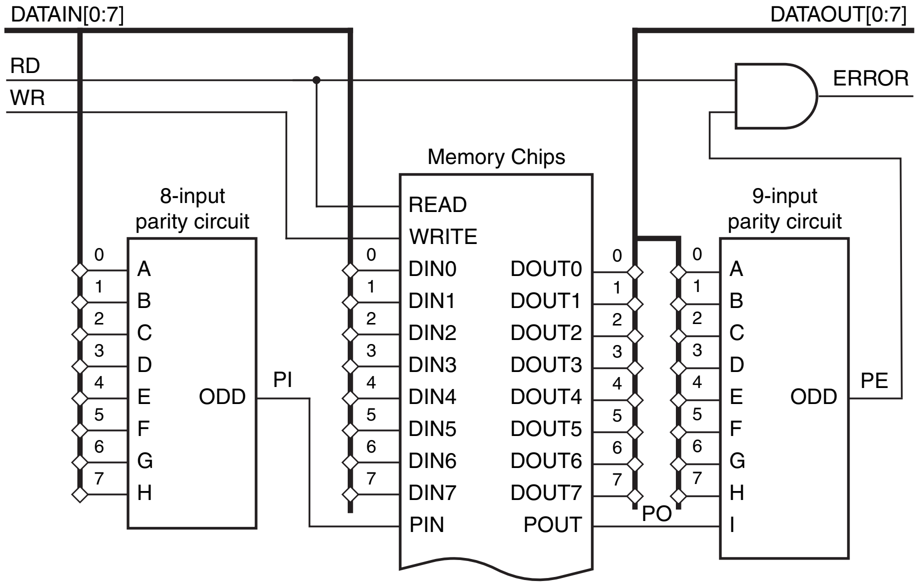


Figure 7-17 Parity generation and checking for an 8-bit-wide memory.

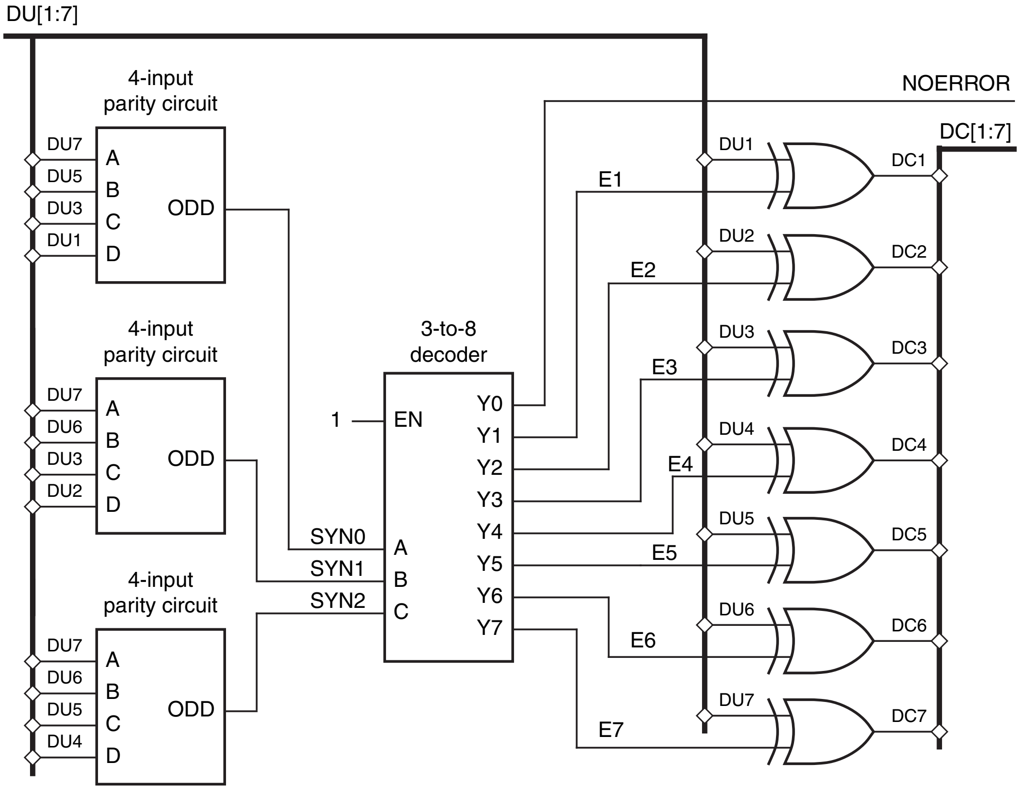


Figure 7-18 Error-correcting circuit for a 7-bit Hamming code.

an ERROR signal is asserted to indicate parity errors during read operations. Complete details of the memory chips, like addressing inputs, are not shown; memory chips will be described in detail in Chapter 15. For parity checking, we are concerned only with the data connections to the memory.

To store a byte into the memory chips, we specify an address (not shown), place the byte on DATAIN[0–7], and assert WR. The 8-input parity circuit asserts its ODD output if the byte has odd parity, placing the value on PI. This value is stored into the memory at the same address as the 8-bit data.

To retrieve a byte, we specify an address and assert RD; the byte value appears on DATAOUT[0–7], and its parity appears on PO. The 9-input parity circuit asserts its ODD output if the 9-bit value has odd parity, indicating that an error has occurred. Thus, at the AND-gate output, ERROR is asserted if RD is asserted and the retrieved 9-bit value has odd parity.

Parity circuits are also used with most error-correcting codes such as the Hamming codes described in Section 2.15.3. We showed the parity-check matrix for a 7-bit Hamming code in Figure 2-13 on page 72. We can correct errors in this code as shown in Figure 7-18. A 7-bit word, possibly containing an error, is presented on DU[1–7]. Three 4-input parity circuits check the parity of

USING IT BOTH WAYS

In an 8-bit memory system where both read and write data use the same bidirectional bus, a single 9-bit parity circuit can be connected to the bus and be used to both generate and check parity, as shown in Figure 7-19. During write operations, control logic forces the ninth input of the parity circuit to 0 to generate the proper parity value for writing. During read operations, it connects the parity output of the memory to the ninth input so the parity circuit checks the overall 9-bit parity.

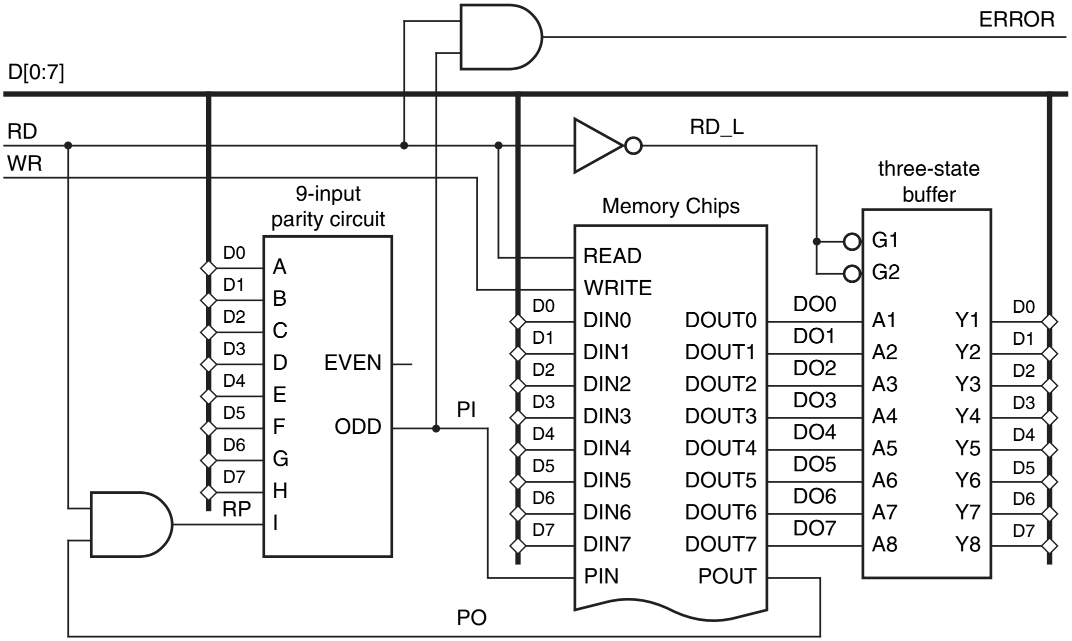


Figure 7-19 Parity generation and checking for an 8-bit-wide memory with shared I/O bus.

the three bit-groups defined by the parity-check matrix, each producing a 1 output if its group has odd parity. These outputs form the syndrome, which is the bit number of the erroneous input bit, if any. A 3-to-8 decoder is used to decode the syndrome. If the syndrome is zero (000), the NOERROR signal is asserted. Otherwise, the erroneous bit is corrected by complementing it. The corrected code word appears on the DC bus.

7.3.4 Exclusive-OR Gates and Parity Circuits in Verilog

In Verilog code, the XOR and XNOR functions are specified by the ^ and ~^ operators, respectively. For example, Program 7-11 is a dataflow-style module for a 3-input XOR device using the XOR operator. It's also possible to specify XOR or parity functions behaviorally, as Program 7-12 does for a 9-input parity function similar to the circuit we used in the preceding subsection.

Program 7-11 Dataflow-style Verilog module for a 3-input XOR device.

```
module Vrxor3(A, B, C, Y);
  input A, B, C;
  output Y;

  assign Y = A ^ B ^ C;
endmodule
```

Program 7-12 Behavioral Verilog module for a 9-input parity circuit.

```
module Vrparity9(I, ODD);
  input [1:9] I;
  output reg ODD;
  integer j;

  always @ (*) begin
    ODD = 1'b0;
    for (j =1; j <= 9; j = j+1)
      if (I[j]) ODD = ~ODD;
  end
endmodule
```

Typical ASIC libraries provide two- and three-input XOR and XNOR functions as primitives. These primitives are usually realized very efficiently in CMOS at the transistor level using transmission gates, as shown for example in Exercise 7.32. Fast and compact XOR trees can be built using these primitives.

When a Verilog module containing large XOR functions is synthesized, the synthesis tool will do the best it can to realize the function in the targeted device technology. However, some synthesis tools may not smart enough to create an efficient tree structure from a behavioral model like Program 7-12. Instead, we can use a structural model to get exactly what we want.

For example, Program 7-13 is a structural Verilog module for a 9-input XOR function implemented as a two-level tree of 3-input XORs. In this example, we've used the previously defined `Vrxor3` module as the basic building block of

Program 7-13 Structural Verilog module for a 9-input parity circuit.

```
module Vrparity9s(I, ODD);
  input [1:9] I;
  output ODD;
  wire Y1, Y2, Y3;

  Vrxor3 U1 (I[1], I[2], I[3], Y1);
  Vrxor3 U2 (I[4], I[5], I[6], Y2);
  Vrxor3 U3 (I[7], I[8], I[9], Y3);
  Vrxor3 U4 (Y1, Y2, Y3, ODD);
endmodule
```

Program 7-14 Behavioral Verilog module for Hamming error correction.

```
module Vrhamcorr(DU, DC, NOERROR);
  input [7:1] DU;
  output reg [7:1] DC;
  output reg NOERROR;
  integer i;

  function [2:0] syndrome;
    input [7:1] D;
    begin
      syndrome[0] = D[1] ^ D[3] ^ D[5] ^ D[7];
      syndrome[1] = D[2] ^ D[3] ^ D[6] ^ D[7];
      syndrome[2] = D[4] ^ D[5] ^ D[6] ^ D[7];
    end
  endfunction

  always @ (*) begin
    DC = DU;
    i = syndrome(DU);
    if (i == 3'b0) NOERROR = 1'b1;
    else begin
      NOERROR = 1'b0; DC[i] = ~DU[i];
    end
  end
endmodule
```

the XOR tree. In an ASIC, we would replace the `Vrxor3` module with a 3-input XOR primitive from the ASIC library.

Our final parity example is a behavioral Verilog module for the Hamming decoder circuit of Figure 7-18, and is shown in Program 7-14. The function `syndrome` is defined to return the 3-bit syndrome of a 7-bit data input vector `D`. In the main `always` block, the corrected data output vector `DC` is initially set equal to the uncorrected data input vector `DU`. The `syndrome` function is then called to get the 3-bit syndrome. If the syndrome is zero, either no error or an undetectable error has occurred and the output `NOERROR` is set to 1. If the syn-

JUST-IN-TIME VERILOG FOR PROGRAM 7-14

`function`
`endfunction`

A Verilog module may declare a local function that returns a result to a caller. Its declaration begins with the keyword `function`, followed by a result-type, function name, and semicolon. It has one or more inputs and local variables. Its declarations are followed by a single procedural statement, usually a `begin`-`end` block, and the keyword `endfunction`. The function name is implicitly defined to be a local `reg` variable of the declared result type, and somewhere in the function, a value must be assigned to this variable. This value is returned to the function's caller. A function is called in a module by writing its name followed by a parenthesized list of expressions which are assigned to its inputs, and its procedural statement is executed.

SOMETIMES, THEY JUST DON'T LISTEN

The basic combinational-logic building block in Xilinx 7-series FPGAs is a 6-input, 1-output lookup table (LUT), that can realize *any* logic function of six inputs, including a 6-input parity function. Therefore, using seven LUTs, a Xilinx 7-series FPGA should be able to realize a 36-input parity tree analogous to the structure in Program 7-13 (`Vrparity9s`), with a maximum delay path that goes through just two levels of logic (LUTs).

So, I wrote code for `Vrparity36s` (a structural module) incorporating `VrXOR6` (a behavioral module), and tried it. Sure enough, the synthesized design was a tree with seven LUTs, six at the first level implementing 6-input XORs, with outputs combined by a single-LUT-based 6-input XOR at the second level. Curiously, though, the inputs of the first-level LUTs were scrambled—the first six inputs were not hooked up to the first first-level LUT as specified in `Vrparity36s`, and so on.

Next, I tried synthesizing a *behaviorally* specified 36-input XOR module, `Vrparity36`, created by changing all instances of "9" to "36" in Program 7-12. As you might expect, before optimization, the tool showed the circuit to be a 36-gate-long daisy chain of XOR gates in the style of Figure 7-16(a). With optimization, however, the synthesizer still came up with a tree with seven LUTs, and still with scrambled input connections as in the structurally specified design. Modern synthesis tools are very good, and we have to give this one credit for finding the most efficient available structure for a behaviorally specified design. But why didn't it follow the specified input hookup pattern in the first, structurally specified case?

As it turns out, by default a good synthesis tool will "flatten" a hierarchically specified design like `Vrparity36s` to give itself more opportunities to optimize the synthesized design—to get what the EDA industry calls "higher QoR" (quality of results). When doing this, it loses some or all of any structural information that is present, and it simply optimizes the logic function at the top-level output (`ODD` in the case of `Vrparity36` and `Vrparity36s`).

Since the behavioral and the structural code both ultimately specify the same output function, in this example the synthesizer was able to come up with the same optimized circuit structure in both cases. But its definition of optimization did not include making the input connection order pretty for the professor.

There are many reasons that a designer may prefer to maintain the hierarchy specified in a large Verilog model, regardless of whether its individual modules are specified structurally or behaviorally. Besides convenience of understanding the synthesized circuit structure, reasons may include ease of timing analysis and debugging. Therefore, typical synthesis tools include many options for constraining the synthesizer's behavior and its use of optimizations.

keep_hierarchy

Using the Xilinx Vivado tools, I was able to insert the `keep_hierarchy` synthesis constraint in the definition of my `Vrxor6` module, which forces the synthesizer to keep all of the module's logic in a dedicated set of one or more LUTs (in this case, one). This made the synthesized circuit be just what I wanted it to be—so pretty that I just had to show it to you, in Figure 7-20.

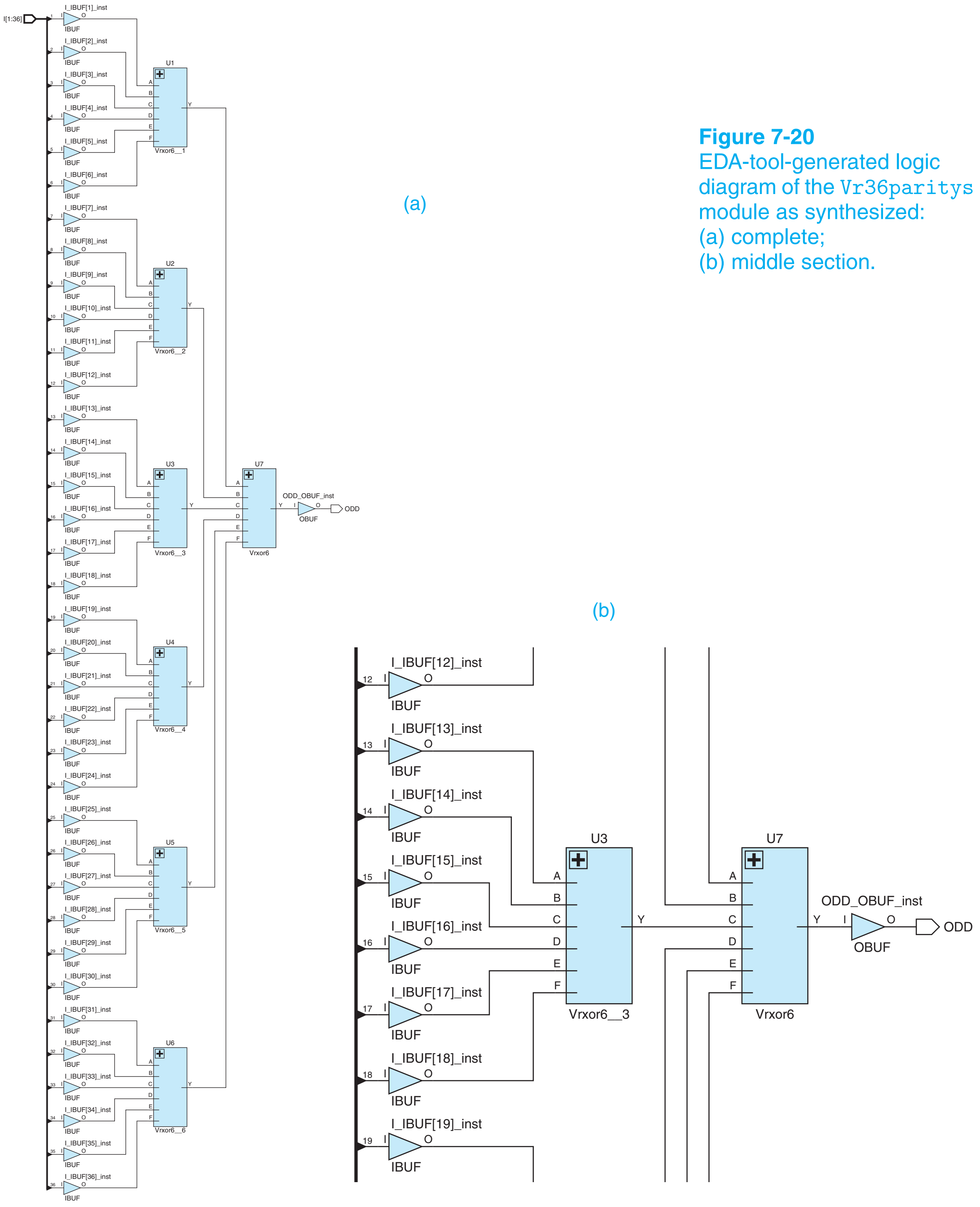


Figure 7-20 EDA-tool-generated logic diagram of the `Vr36paritys` module as synthesized: (a) complete; (b) middle section.

drome is nonzero, the corresponding bit of DC is complemented to correct the assumed 1-bit error, and NOERROR is cleared to 0.

A self-checking test bench for the Hamming correction module is shown in Program 7-15. This test bench takes a high-level functional approach. For each possible combination of data bits (there are only 16), it calculates the three check bits, creating a 7-bit vector DI. Then it applies DI as well as seven variants of it, each with a 1-bit error, to the DU input of the Hamming correction module. For each of these, it checks that a properly corrected result DC is returned and that NOERROR has the correct value. The same testing approach could be applied to Hamming correction modules for much wider data buses, except that the data values would be selected randomly instead of exhaustively to make the testing time feasible.

Program 7-15 Test bench for the Hamming error-correction module.

```
`timescale 1 ns / 100 ps
module Vrhamcorr_tb();
  reg [7:1] DI, DU;
  wire [7:1] DC;
  wire NOERR;
  reg [3:0] DATA;
  integer nib, i, errors;

Vrhamcorr UUT (.DU(DU), .DC(DC), .NOERROR(NOERR));

initial begin
  errors = 0;
  for (nib=0; nib<=15; nib=nib+1) begin
    DATA[3:0] = nib;
    DI[7:5] = DATA[3:1]; DI[3] = DATA[0]; // Merge in data value
    DI[4] = DI[7] ^ DI[6] ^ DI[5];        // Merge in check bits
    DI[2] = DI[7] ^ DI[6] ^ DI[3];
    DI[1] = DI[7] ^ DI[5] ^ DI[3];
    DU = DI; #10 ;             // Check no-error case
    if ((DC!==DI) || (NOERR!==1'b1)) begin
      errors = errors + 1;
      $display("Error, DI=%b, DU=%b, DC=%b, NOERR=%b",DI,DU,DC,NOERR);
    end
    for (i=1; i<=7; i=i+1) begin      // Insert error in each bit position
      DU = DI; DU[i] = ~DI[i]; #10 ;  // and check that it's corrected
        if ((DC!==DI) || (NOERR!==1'b0)) begin
          errors = errors + 1;
          $display("Error, DI=%b, DU=%b, DC=%b, NOERR=%b",DI,DU,DC,NOERR);
        end
    end
  end
  $display("Test completed, %0d errors",errors);
end
endmodule
```

7.4 Comparing

Comparing two binary words for equality is a commonly used operation in computer systems, device interfaces, and many other applications. For example, in Figure 2-7(a) on page 65, we showed a system structure in which devices are enabled by comparing a “device select” word with a predetermined “device ID.” A circuit that compares two binary words and indicates whether they are equal is called a *comparator*. Some comparators interpret their input words as signed or unsigned numbers and also indicate an arithmetic relationship (greater or less than) between the words. These devices are often called *magnitude comparators*

comparator

magnitude comparator

All of the magnitude comparators in this section are for unsigned numbers. When their inputs are signed two’s-complement numbers, they produce correct greater-than and less-than results when the signs of the operands are identical. But when the signs are different, they produce the opposite of the correct result. Any unsigned number whose MSB is 1 is greater than one whose MSB is 0. But in the signed interpretation, any number whose MSB is 1 is negative and therefore less than any positive one (MSB=0). Table 7-4 shows examples using 4-bit vectors, including the corresponding decimal numbers.

Table 7-4 Comparisons of signed and unsigned 4-bit vectors.

Unsigned interpretation		***Signed interpretation***	
0101 > 0001	(5 > 1)	0101 > 0001	(5 > 1)
1110 > 1001	(14 > 9)	1110 > 1001	(–2 > –7)
1111 > 0000	(15 > 0)	1111 < 0000	(–1 < 0)
1011 > 0100	(11 > 4)	1011 < 0100	(–5 < 4)

7.4.1 Comparator Structure

Exclusive-OR and Exclusive-NOR gates may be viewed as 1-bit comparators. Figure 7-21(a) shows an interpretation of a 2-input XOR gate as a 1-bit comparator. The active-high output, DIFF, is asserted if the inputs are different. The outputs of four XOR gates are ORed to create a 4-bit comparator in (b). The DIFF

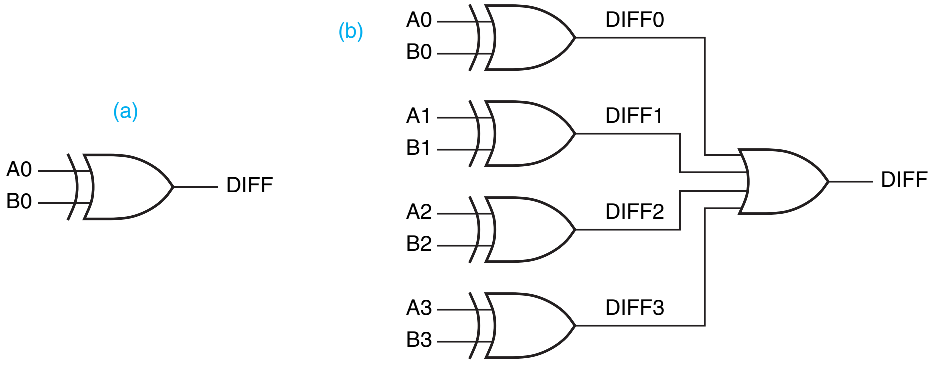


Figure 7-21 Comparators using XOR gates: (a) 1-bit comparator; (b) 4-bit comparator.

WIDE GATES FOR COMPARATORS

There is a practical limit to the width of an individual AND or OR gate in any technology. Wider functions can be obtained by cascading individual gates, as we did for wider XOR functions in Figure 7-16 on page 323. As in that example, arranging the gates in a tree-like structure rather than a linear cascade makes for a faster circuit.

For a wide AND or OR function, there is an opportunity to make the circuit even faster. At the transistor level, inverting gates are typically faster and smaller than noninverting ones. For example, an AND-gate circuit is typically designed as a NAND gate followed by an inverter, in the style of Figure 14-15 on page 743. With this in mind, Figure 7-22 shows two different approaches to building a 16-input OR function. In (a), we use two levels of OR gates, which really yields four levels of gate delay at the transistor level with "typical" transistor-level noninverting gate design. In (b), we use one level of NOR gates followed by a NAND gate, resulting in only two levels of gate delay and a smaller circuit at the transistor level, too.

The above analysis assumes "typical" internal gate circuit design. In a specific application, circuit area and delay may vary depending on arcane details of the technology. So, in HDL-based design of ASICs, FPGA, and PLDs, you're usually best served by ignoring these details and just letting the synthesis tool figure out the best realization. Just be aware that logic functions that require very wide gates will typically be not only larger, but also slower than ones that use only narrow gates.

output is asserted if any of the input-bit pairs are different. We can build an *n*-bit comparator using *n* XOR gates and an *n*-input OR gate.

Comparators can also be built using Exclusive-NOR (XNOR) gates, which are sometimes called Equivalence gates for that reason. A 2-input XNOR gate produces a 1 output if its two inputs are equal. A multibit comparator can be constructed using one XNOR gate per bit, and ANDing all of their outputs together. The output of the AND function is 1 if all of the individual bits are pairwise equal.

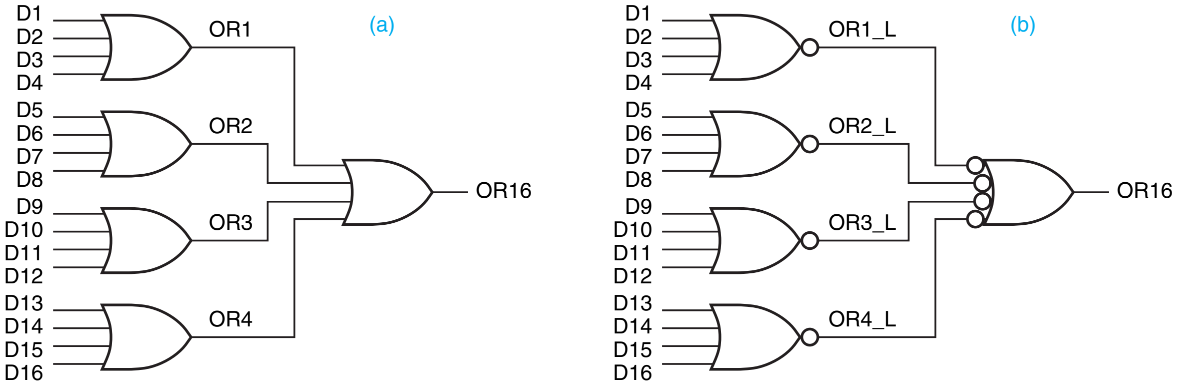


Figure 7-22 16-input OR functions: (a) using OR gates; (b) using NOR and NAND gates.

The n-bit comparators in this subsection are sometimes called *parallel comparators* because they look at each pair of input bits simultaneously and deliver the 1-bit comparison results in parallel to an n-input OR or AND function. It is also possible to design an "iterative comparator" that looks at its bits one at a time using a small, fixed amount of logic per bit. Before looking at an iterative comparator design, we'll describe the general class of "iterative circuits" in the next subsection. This class of circuits also includes adders, which we'll cover in Chapter 8.

parallel comparator

7.4.2 Iterative Circuits

An *iterative circuit* is a special type of combinational circuit, with the structure shown in Figure 7-23. The circuit contains n identical modules, each of which has both *primary inputs and outputs* and *cascading inputs and outputs*. The leftmost cascading inputs are called *boundary inputs* and are connected to fixed logic values in many iterative circuits. The rightmost cascading outputs are called *boundary outputs* and usually provide important information.

iterative circuit

primary inputs and outputs

cascading inputs and outputs

boundary inputs

boundary outputs

Iterative circuits are well-suited to problems that can be solved by a simple iterative algorithm:

1. Set C_0 to its initial value and set i to 0.
2. Use C_i and PI_i to determine the values of PO_i and C_{i+1}.
3. Increment i.
4. If $i < n$, go to step 2.

In an iterative circuit, the loop of steps 2–4 is "unwound" by providing a separate combinational-circuit module that performs step 2 for each value of i.

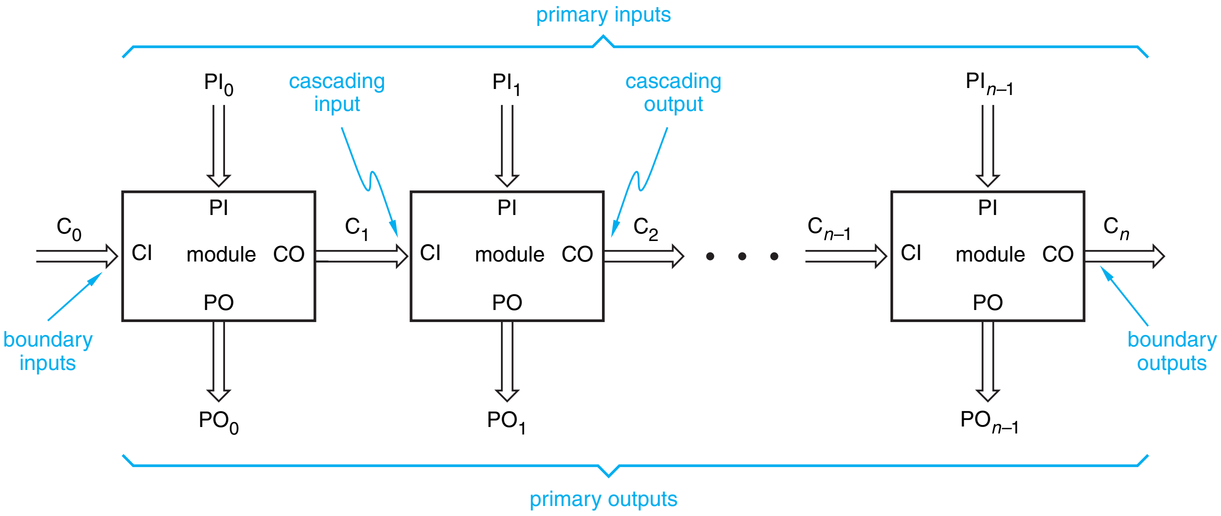


Figure 7-23 General structure of an iterative combinational circuit.

Examples of iterative circuits are the comparator circuit in the next subsection and the ripple adders in Sections 8.1.2 and 8.1.5. In Section 11.3, we'll explore the relationship between iterative circuits and corresponding sequential circuits that execute the previous page's 4-step algorithm in discrete time steps.

7.4.3 An Iterative Comparator Circuit

Two *n*-bit values *X* and *Y* can be compared one bit at a time in *n* steps using a single bit EQ_i at each step to keep track of whether all of the bit-pairs so far have been equal, as follows:

1. Set EQ_0 to 1 and set *i* to 0.
2. If EQ_i is 1 and X_i and Y_i are equal, set EQ_{i+1} to 1. Else set EQ_{i+1} to 0.
3. Increment *i*.
4. If $i < n$, go to step 2.

Figure 7-24 shows a corresponding iterative circuit. Note that this circuit has no primary outputs; the boundary output is all that interests us. Other iterative circuits, like the ripple adder of Section 8.1.2, have primary outputs of interest.

Given a choice between the iterative comparator circuit in this subsection and one of the parallel comparators shown previously, you would probably prefer the parallel comparator. The iterative comparator saves little if any cost, and it's very slow because the cascading signals need time to "ripple" from the leftmost to the rightmost module. Iterative circuits with individual modules that process more than one bit at a time, like the ones we'll show for comparators in the next subsection and for adders in Section 8.1.5, are much more likely to be used in practical designs.

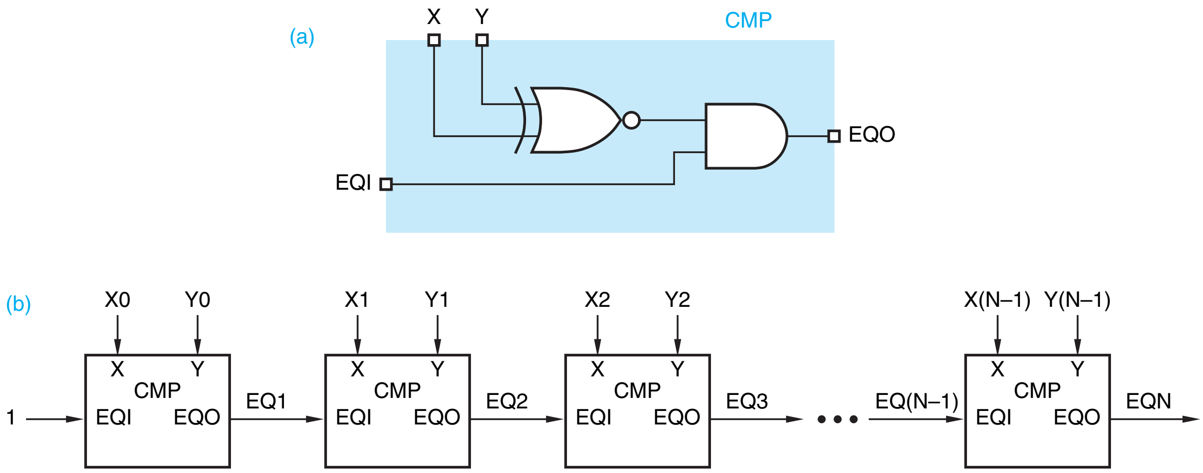


Figure 7-24 An iterative comparator circuit: (a) module for one bit; (b) complete circuit.

7.4.4 Magnitude Comparators

A binary magnitude comparator compares two binary numbers and indicates whether one is less than, equal to, or greater than the other. One way to do this is to subtract one number from the other and look at the results. The numbers are equal if the difference is zero, of course. With unsigned numbers, the less-than/greater-than relationship is indicated by the borrow out of the MSB—1 if the subtrahend is greater than the minuend, 0 otherwise. With two's-complement, signed numbers, the sign bit of the difference is 1 if the subtrahend is greater than the minuend, and 0 otherwise. So, a magnitude comparator can be made easily from a subtractor; but we won't describe adders and subtractors until Section 8.1. In this section, we'll discuss magnitude comparators that operate "directly," without looking at a subtraction result. Depending on their implementation, they may be smaller and faster than subtractor-based comparators, since they don't need any logic for an actual subtraction result.

Figure 7-25 is a logic symbol for a magnitude comparator for two 8-bit unsigned numbers. It has three active-high outputs that indicate the comparison relationship between 8-bit inputs P[7:0] and Q[7:0], with 7 being the most significant bit.

A logic diagram for the magnitude comparator is given in Figure 7-26. The top half of the circuit checks the two 8-bit input words for equality. Each XNOR-gate output is asserted if its inputs are equal, and the PEQQ output is asserted if all eight input-bit pairs are equal. The bottom half of the circuit compares the input words arithmetically and asserts PGTQ if P > Q. Each AND gate connects to a pair of input bits (Pi,Qi) and zero or more of the XNOR outputs, forcing PGTQ to 1 if (Pi,Qi) is (1,0) and all the higher-order bits are pairwise equal.

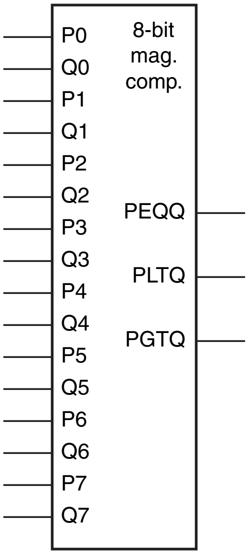


Figure 7-25
Logic symbol for an 8-bit magnitude comparator.

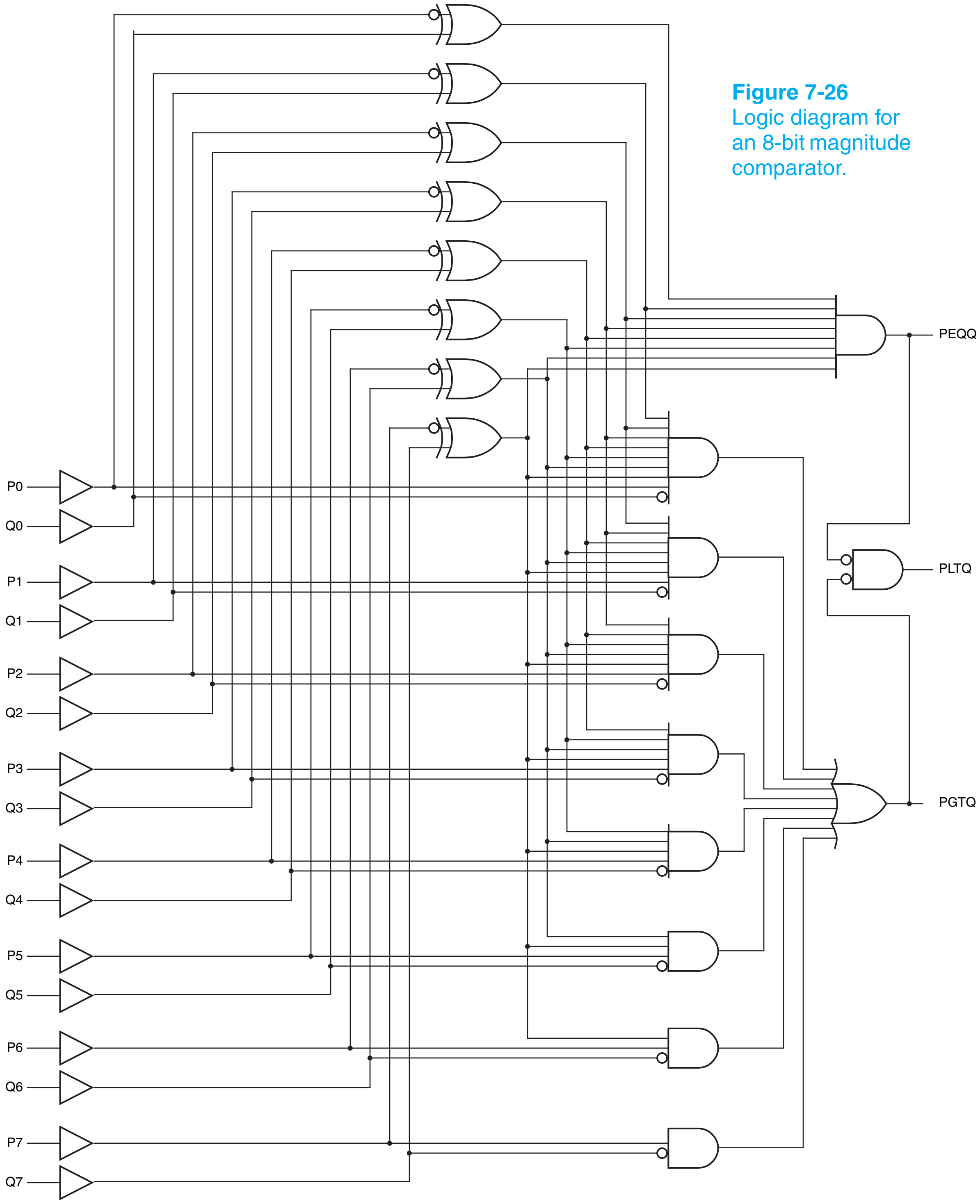


Figure 7-26 Logic diagram for an 8-bit magnitude comparator.

Although a similar idea could be used to create the “less-than” output, in this circuit it’s done with just one 2-input NOR gate at the expense of a little extra delay: PLTQ is asserted if both of the other two outputs are negated. It should be evident that any two of the three outputs fully describe the comparison result. The remaining one can be derived from the other two with a 2-input NOR gate, since at all times exactly one out of the three outputs should be asserted.

The 8-bit magnitude comparator may be used as a building block to create a larger comparator. In a purely iterative circuit with no other components, n 8-bit comparators can be used to compare two $7n+1$-bit numbers. Starting with the LSBs of P and Q, input bits are assigned to the comparators. The PGTQ and PLTQ outputs of each comparator are connected to the P0 and Q0 inputs, respectively, of the next comparator, which handles the seven next more significant bits of P and Q. This works because the 1-bit comparison made between P0 and Q0 by the second and successive comparators is a proxy for comparing all of the less significant bits of P and Q, based on the following three possibilities:

- P = Q so far: P0,Q0 = 0,0.
- P > Q so far: P0,Q0 = 1,0.
- P < Q so far: P0,Q0 = 0,1.

Figure 7-27 illustrates the connections for a 22-bit comparator.

Using the above approach, we could build a 64-bit comparator using nine 8-bit comparators. Because the comparators would be connected in series, the total delay from any of the LSBs to the overall 64-bit-comparison outputs would be equal to nine times the delay of one 8-bit comparator.

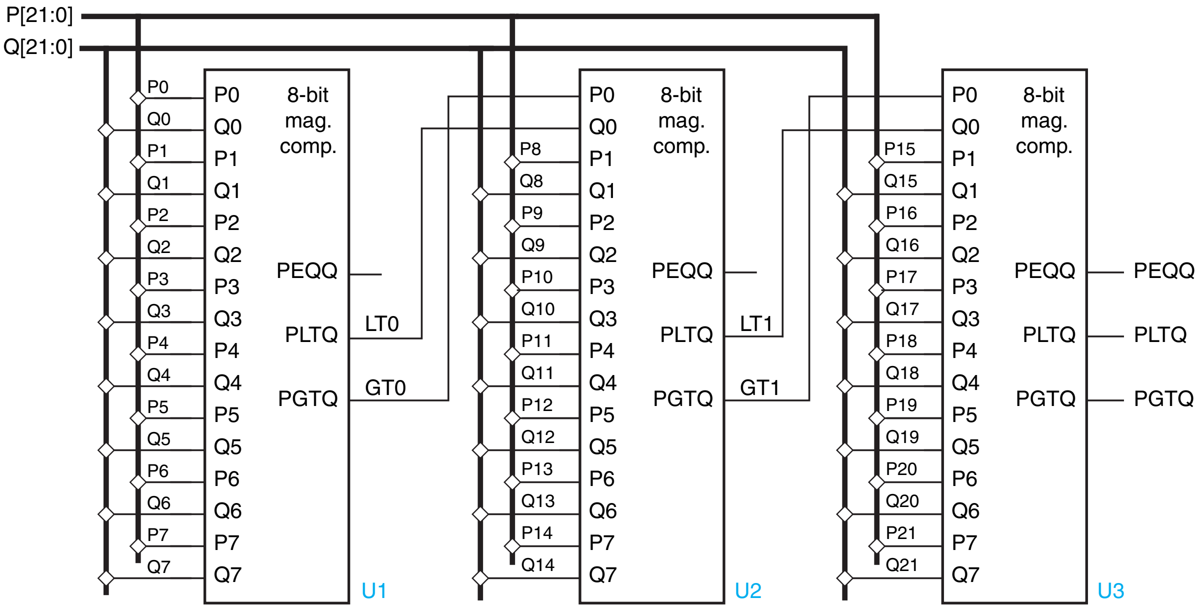
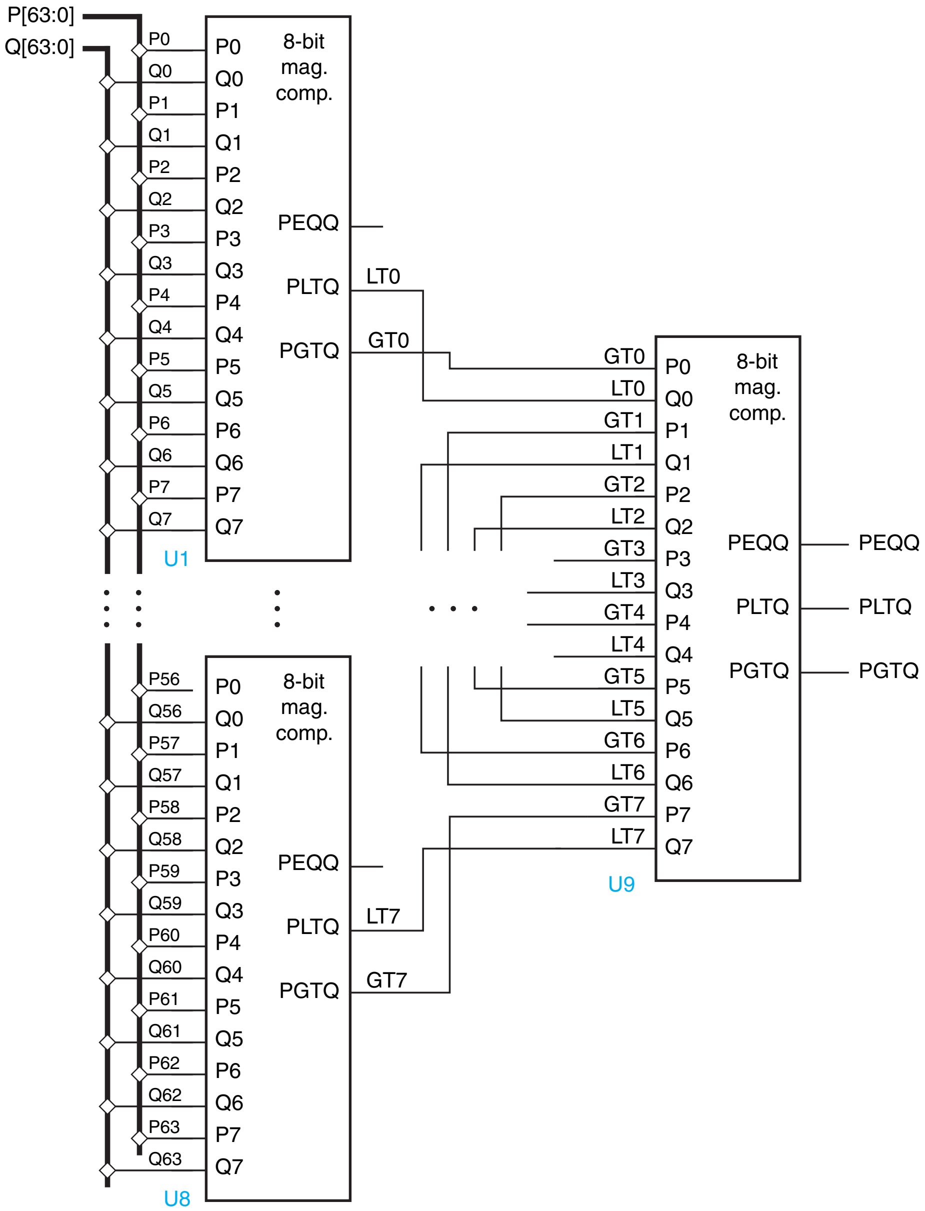


Figure 7-27 Three 8-bit magnitude comparators cascaded to compare 22 bits.

Figure 7-28
Comparator tree for 64 bits using 8-bit comparators.



But there's a better way—the nine comparators could be configured in a tree, with just two levels of 8-bit comparators. The 64 input bits P[63:0] and Q[63:0] connect to eight first-level comparators. The PGTQ and PLTQ outputs of these comparators connect to the P0-P7 and Q0-Q7 inputs of one second-level comparator in order of significance, as shown in Figure 7-28. So the total delay to the overall 64-bit-comparison outputs from any input bit will be just twice the delay of a single 8-bit comparator.

7.4.5 Comparators in HDLs

HDLs like Verilog have built-in operators for equality and magnitude comparison. So, you may think that comparators are easy to design in HDLs, because an EDA tool does the heavy lifting for you. But, it's better to think of comparators as being easy only to *specify* in your design. With just a few simple relational expressions in an HDL model, you can cause a lot of big, potentially slow comparators to be synthesized. Thus, it's important for you to have a feel for what kind of logic will be synthesized when you specify comparison operations in your models.

Comparing two vectors of bits for equality or inequality is very easy to do in an HDL model, in Verilog using the "`==`" and "`!=`" operators in relational expressions. Thus, given the expression "`(P==Q)`", where `P` and `Q` are bit vectors each with *n* elements, the compiler can create the following logic expression:

$$((\mathrm{P1} \oplus \mathrm{Q1}) + (\mathrm{P2} \oplus \mathrm{Q2}) + \ldots + (\mathrm{Pn} \oplus \mathrm{Qn}))'$$

Recall that "⊕" is the Exclusive OR operator. The logic expression for "`P!=Q`" is just the complement of the one above, or

$$(\mathrm{P1} \oplus \mathrm{Q1}) + (\mathrm{P2} \oplus \mathrm{Q2}) + \ldots + (\mathrm{Pn} \oplus \mathrm{Qn})$$

In the preceding logic expressions, it takes one 2-input XOR function to compare each bit. Since a 2-input XOR function can be realized as a sum of two product terms, the complete expression can be realized, for example in an ASIC with discrete gates or in a PLD, as a relatively modest, possibly complemented sum of 2*n* product terms:

$$(\mathrm{P1} \cdot \mathrm{Q1}' + \mathrm{P1}' \cdot \mathrm{Q1}) + (\mathrm{P2} \cdot \mathrm{Q2}' + \mathrm{P2}' \cdot \mathrm{Q2}) + \ldots + (\mathrm{Pn} \cdot \mathrm{Qn}' + \mathrm{Pn}' \cdot \mathrm{Qn})$$

Magnitude comparison is another story—there are at least three different ways that HDL synthesis tools could create the logic for a greater-than or less-than condition, and all of them result in a much larger circuit:

1. Use a subtractor to subtract one *n*-bit vector from the other, determine the greater-than/less-than condition from the borrow out, and if needed, derive the equals condition from the *n*-bit difference output. Prune off any part of the circuit that does not contribute to the condition outputs.
2. Whether the equals condition is needed or not, check each bit pair for equality using an XOR gate as in the equations above. Use and combine their outputs with a set of AND gates, one per bit position and followed by an OR gate, in the style of Figure 7-26, to determine the greater-than/less-than condition.
3. Use an iterative approach to create a nested set of fixed-size equations, one per bit position, and manipulate these equations in the best way possible to obtain a structure suitable for the targeted implementation technology.

All three approaches should ultimately yield the same logic expression, but because the structures of the starting points are different, the final realizations for that logic expression may also be different. Thus, an HDL tool that targets a particular implementation technology will use an approach that is most likely to yield an efficient circuit in that technology. As an example, equations for the third approach are described below.

Consider the relational expression "`(P>Q)`". To construct the corresponding logic expression, the HDL compiler can first build n equations of the form

$$\mathrm{Gi} = (\mathrm{Pi} \cdot (\mathrm{Qi}' + \mathrm{Gi\text{-}1})) + (\mathrm{Pi}' \cdot \mathrm{Qi}' \cdot \mathrm{Gi\text{-}1})$$

for $i = 1$ to n, and G0 = 0 by definition. This is, in effect, an iterative (some would call it recursive) definition of the greater-than function, starting with the least significant bit. Each Gi variable is asserted if, as of bit i, P is greater than Q. This is true if Pi is 1 and either Qi is 0 or P was greater than Q as of the previous bit, or if Pi and Qi are both 0 and P was greater than Q as of the previous bit.

The logic equation for "`(P>Q)`" is simply the expression for Gn. So, after creating the n equations above, the HDL compiler can collapse them into a single equation for Gn involving only elements of P and Q. It does this by substituting the Gn-1 equation into the righthand side of the Gn equation, then substituting the Gn-2 equation into this result, and so on, until substituting 0 for G0. In the case of a synthesis tool that targets a PLD or other sum-of-products realization, the final step is to derive a minimal sum-of-products expression from the Gn expression. In other cases, the tool may simply create a long chain of logic, with length corresponding to the nesting in the Gn expression, and then use its "standard" internal methods for optimizing long chains of logic based on the limitations of the targeted technology (like number of inputs available per ASIC gate or FPGA LUT).

COMPARING COLLAPSED COMPARATOR CIRCUITS

Collapsing an iterative circuit into a two-level sum-of-products realization often creates an exponential expansion of product terms. The greater-than and less-than functions do this, requiring 2^n-1 product terms for an n-bit comparator. Thus, comparators larger than a few bits cannot be realized practically as a two-level AND-OR circuit in an ASIC or PLD; too many product terms are needed.

FPGA-based realizations are limited as well. A typical FPGA uses a LUT to implement combinational logic functions, and a typical LUT has just six inputs, enough for one output of just a 3-bit comparator.

For larger comparators, the compiler may synthesize a set of smaller comparators, and then cascade or combine their outputs to obtain the larger comparison result. Some FPGAs, like the Xilinx 7 series, have special "carry" logic blocks that can be used to optimize the size and performance of adders and subtractors. When these are available, the compiler's best strategy for doing comparisons is usually just to synthesize a subtractor, and to derive the comparison outputs from that.

7.4.6 Comparators in Verilog

Verilog has built-in comparison operators, >, >=, <, <=, ==, and !=, which can be applied to bit vectors. The bit vectors are interpreted as unsigned numbers with the most significant bit on the left, regardless of how they are numbered. Verilog-2001 and later also supports signed arithmetic, using the language extensions described in the box on page 191. When a Verilog module uses a comparison, the compiler synthesizes corresponding comparator logic.

Verilog tries to do "the right thing" to match up operands of different lengths. With unsigned operands, it adds zeroes on the left of the shorter one. With signed operands, it may extend the sign of the shorter one to the left, but it may not. (Again, see the box on page 191.) So, in complicated length mismatch situations, it is better to pad out the shorter operand explicitly.

The size and speed of synthesized comparator logic depends on the target technology and the optimization capabilities of the synthesis tool. Equality and inequality checkers are fairly small and fast. As shown in the preceding subsection, they can be built from n XOR (or XNOR) gates and an n-input AND or OR gate. The XOR or XNOR gates all operate in parallel, and a reasonably fast AND or OR gate of any size can be built using a tree-like structure.

Checking for greater-than or less-than conditions requires a larger circuit. As we discussed, there are several possible approaches that the compiler might use depending on the targeted implementation technology.

Comparison operations usually are not standalone but are embedded into larger Verilog modules. Still, we'll present some standalone examples in the rest of this subsection to explore the results and to give you some examples of different Verilog coding approaches. But unless performance is critical, a designer shouldn't have to worry about picking a particular coding style or structure; and as we'll see, there's no guarantee that a particular one will give the best results.

Program 7-16 is a first attempt at creating a behavioral Verilog module for the 8-bit magnitude comparator in Figure 7-25 on page 335. It outputs indicate whether P is greater than, less than, or equal to Q. But it has two problems.

Program 7-16 Verilog module for an 8-bit magnitude comparator.

```
module Vr8bitcmp_xi(P, Q, PGTQ, PEQQ, PLTQ);
  input [7:0] P, Q;
  output reg PGTQ, PEQQ, PLTQ;

  always @ (*)
    if (P == Q)
      begin PGTQ = 1'b0; PEQQ = 1'b1; PLTQ = 1'b0; end
    else if (P > Q)
      begin PGTQ = 1'b1; PEQQ = 1'b0; PLTQ = 1'b0; end
    else if (P < Q)
      begin PGTQ = 1'b0; PEQQ = 1'b0; PLTQ = 1'b1; end
endmodule
```

Program 7-17 Second attempt at an 8-bit magnitude comparator module.

```
module Vr8bitcmp_xc(P, Q, PGTQ, PEQQ, PLTQ);
  input [7:0] P, Q;
  output reg PGTQ, PEQQ, PLTQ;

  always @ (*)
    if (P == Q)
      begin PGTQ = 1'b0; PEQQ = 1'b1; PLTQ = 1'b0; end
    else if (P > Q)
      begin PGTQ = 1'b1; PEQQ = 1'b0; PLTQ = 1'b0; end
    else if (P < Q)
      begin PGTQ = 1'b0; PEQQ = 1'b0; PLTQ = 1'b1; end
    else
      begin PGTQ = 1'bx; PEQQ = 1'bx; PLTQ = 1'bx; end
endmodule
```

First, although the code has three `if` clauses that cover all of the possible comparison outcomes perfectly, the Verilog compiler doesn't know that. As far as it knows, none of the `if` conditions may match, in which case no new value is specified for the condition outputs. This is a classic situation where the compiler will "infer a latch" to hold the old value of the condition outputs, which is not the designer's intention. You may think that no harm is done since the no-match case will never really occur, but the inferred latches still add size and delay to the final synthesized circuit, even though they are never used to store an old value.

The inferred latches in Program 7-16 can be avoided by making sure that a value is assigned to the outputs in all situations. We do it in this example with a final `else` clause as shown in Program 7-17. Since we actually know that this `else` clause will never be reached (exactly one of the first three comparisons will always be true), we really don't care what output values are specified in it and set them to "x", since some tools interpret an "x" on the righthand side of an assignment as a "don't-care" and use it to optimize the synthesized circuit.

Program 7-18 Corrected 8-bit magnitude comparator module.

```
module Vr8bitcmp(P, Q, PGTQ, PEQQ, PLTQ);
  input [7:0] P, Q;
  output reg PGTQ, PEQQ, PLTQ;

  always @ (*)
    if (P == Q)
      begin PGTQ = 1'b0; PEQQ = 1'b1; PLTQ = 1'b0; end
    else if (P > Q)
      begin PGTQ = 1'b1; PEQQ = 1'b0; PLTQ = 1'b0; end
    else
      begin PGTQ = 1'b0; PEQQ = 1'b0; PLTQ = 1'b1; end
endmodule
```

Program 7-19 Comparator module using continuous assignments.

```
module Vr8bitcmp_dx(P, Q, PGTQ, PEQQ, PLTQ);
  input [7:0] P, Q;
  output PGTQ, PEQQ, PLTQ;

  assign PGTQ = ( (P > Q) ? 1'b1 : 1'b0 ) ;
  assign PEQQ = ( (P == Q) ? 1'b1 : 1'b0 ) ;
  assign PLTQ = ( (P < Q) ? 1'b1 : 1'b0 ) ;
endmodule
```

But the new model still has a problem. A typical Verilog compiler is not smart enough to know that the three comparison outcomes are mutually exclusive, and that the third outcome always occurs if the first two don't. Therefore, it synthesizes *two* magnitude comparators, one for the P>Q case and the other for P<Q, requiring significantly more chip resources. This problem is solved in Program 7-18, where we have used our knowledge of comparator functionality to set the outputs to the less-than condition if neither of the first two tests were true, without doing a redundant less-than test. While this may add a little delay (the less-than output is valid one gate- or LUT-delay after the two others), in most applications the savings in chip resources is preferable.

Another approach to modeling the comparator is shown in Program 7-19, using dataflow-style Verilog code. This module uses a continuous-assignment statement to specify the value of each condition output. As in Program 7-17, there is a good chance that the compiler will synthesize an extra comparator for PLTQ. It doesn't know that this condition can be derived from the greater-than and equals conditions, so we do that explicitly in Program 7-20.

What about larger comparators? Any of the modules that we've shown can be modified easily to make a comparator for bit vectors with any number of bits, just by changing the definitions of P and Q at the beginning of the module. If we're using a lot of comparators with different widths, it could be useful to use a parameter to set the width, so it can be specified when the module is instantiated. A parameterized version of Program 7-20 is shown in Program 7-21, with a default width (N) of 8.

Program 7-20 Comparator module using continuous assignments and eliminating potential extra comparator.

```
module Vr8bitcmp_d(P, Q, PGTQ, PEQQ, PLTQ);
  input [7:0] P, Q;
  output PGTQ, PEQQ, PLTQ;

  assign PGTQ = ( (P > Q) ? 1'b1 : 1'b0 ) ;
  assign PEQQ = ( (P == Q) ? 1'b1 : 1'b0 ) ;
  assign PLTQ = ~PGTQ & ~PEQQ;
endmodule
```

Program 7-21 Comparator module with a parameter for vector width.

```
module VrNbitcmp_d(P, Q, PGTQ, PEQQ, PLTQ);
  parameter N=8;
  input [N-1:0] P, Q;
  output PGTQ, PEQQ, PLTQ;

  assign PGTQ = ( (P > Q) ? 1'b1 : 1'b0 ) ;
  assign PEQQ = ( (P == Q) ? 1'b1 : 1'b0 ) ;
  assign PLTQ = ~PGTQ & ~PEQQ;
endmodule
```

While the module for a very wide comparator might "blow up" if it were targeted to a PLD requiring a 2-level sum-of-products implementation, a high-quality EDA tool for FPGAs and ASICs can synthesize a reasonably good implementation using more levels of logic, even for a very large comparator (say, 64 bits). As we discussed previously, a comparator design can be based on a subtractor, and most EDA tools (and some FPGA and ASIC technologies) have special facilities to optimize adders and subtractors, and therefore comparators.

However, there may be cases where you want to leave nothing to chance, and in fact may be able to get a better result in terms of speed, size, or both, by specifying and structuring a large-comparator design in more detail. The tree structure that we described in the last paragraph of Section 7.4.4 is a good basis for doing this.

Program 7-22 is a top-level structural model for a two-level hierarchy based on our previous description, using eight 8-bit comparators (`Vr8bitcmp`) at

Program 7-22 Hierarchical, structural Verilog module for a 64-bit magnitude comparator using nine 8-bit magnitude comparators.

```
module Vr64bitcmp_sh(P, Q, PGTQ, PEQQ, PLTQ);
  input [63:0] P, Q;
  output PGTQ, PEQQ, PLTQ;
  wire [7:0] GT, EQ, LT;

  Vr8bitcmp U1(P[7:0], Q[7:0], GT[0], EQ[0], LT[0]);
  Vr8bitcmp U2(P[15:8], Q[15:8], GT[1], EQ[1], LT[1]);
  Vr8bitcmp U3(P[23:16], Q[23:16], GT[2], EQ[2], LT[2]);
  Vr8bitcmp U4(P[31:24], Q[31:24], GT[3], EQ[3], LT[3]);
  Vr8bitcmp U5(P[39:32], Q[39:32], GT[4], EQ[4], LT[4]);
  Vr8bitcmp U6(P[47:40], Q[47:40], GT[5], EQ[5], LT[5]);
  Vr8bitcmp U7(P[55:48], Q[55:48], GT[6], EQ[6], LT[6]);
  Vr8bitcmp U8(P[63:56], Q[63:56], GT[7], EQ[7], LT[7]);
  Vr8bitcmp U9(GT, LT, PGTQ, PEQQ, PLTQ);
endmodule
```

IT'S JUST A SUGGESTION

EDA tools may use a hierarchical specification only as a starting point for synthesis. Advanced tools have the ability to combine or share logic near the output of one module with logic near the input of another that it drives, with the goal of optimizing delay or resource utilization or both. But in doing so, they may blur or eliminate the boundaries between the originally specified modules. For example, in the module of Program 7-22, the implemented circuit may not have any signals with the same functionality (meaning the same logic expressions) as the GT and LT signals in the Verilog code.

Despite the benefits of optimization, a designer may have good reasons, like ease of debugging, to preserve the originally specified hierarchy and signals in the implemented circuit, and advanced tools let the designer specify "constraints" to do so. For example, Xilinx tools allow the keep_hierarchy constraint to be embedded in a Verilog module definition to prevent the synthesis tool from combining any part of that module with any others, thus forcing it to preserve input and output signals exactly as originally defined in the code.

In a test run, I used the Xilinx Vivado tool to implement Program 7-22 in a large, high-performance FPGA. With the keep_hierarchy constraint in place, the tool yielded an implementation with 82 LUTs and about 17.4 ns of worst-case delay, and the hierarchy and intermediate signals specified in the code were clearly visible in the final schematic diagram and netlist. Removing the constraint yielded an implementation with the same delay and only 75 LUTs, but the original hierarchy and intermediate signals were completely gone.

the first level and one at the second level. The module declares 8-bit wires GT, EQ, and LT to connect the outputs of the first-level comparators to the second-level comparator. Note that the EQ wires do not contribute to the output of Vr64bitcmp_sh, but must be declared to carry the unused PEQQ outputs in the instantiations of the first-level comparators U1-U8. The EQ wires and any logic that is used only to create the signals on them are automatically pruned away by the synthesis tool during optimization.

7.4.7 Comparator Test Benches

A comparator is so easy to describe behaviorally, you might wonder if there's even a need to write a test bench to make sure you got it right. Well, there's always the possibility of typing errors that yield a syntactically correct but functionally incorrect description, especially when comparisons are embedded in larger modules. And in a structural model, there are many more opportunities to get it wrong. So, here we'll look at a simple comparator test bench and then point out some pitfalls that can occur when testing comparators as well as the arithmetic elements in the next chapter.

Program 7-23 is a self-checking test bench for the comparators of the previous subsection. It is parameterized so it can be used with a comparator of different widths, not just 8 bits. Instead of checking all possible input combinations, it uses Verilog's `$random` task to generate random inputs—an exhaustive test would run far too long if the comparator's width were 16 bits or more. Such a comparator would have double that number of inputs, and billions of unique input combinations. As written, the test bench applies a modest 10,000 pseudorandom input combinations to the UUT.

At each iteration of the `for` loop, the test bench generates two new random numbers with `$random` and assigns them to `P` and `Q`. Recall that `$random` returns a 32-bit signed integer result, regardless of the data width of the computer system that is hosting the Verilog tools. The usual rules for assigning an integer to an unsigned vector are followed, so the *N* low-order bits of the integer result are copied into `P` and `Q`. We'll come back to that point shortly.

The test bench works well enough with all of the comparators of Programs 7-16 through 7-21, providing some additional confidence that we got it right in these simple comparator designs. However, two aspects of the test bench are noteworthy. First, it does not detect the presence of the unnecessary inferred latches in Program 7-16, because these latches never do anything functionally,

Program 7-23 Test bench for an *N*-bit comparator.

```
`timescale 1 ns / 100 ps
module VrNbitcmp_tb();
  parameter N = 8;     // Input width of comparator UUT
  parameter SEED = 1;  // Set a different pseudorandom seed here if desired
  reg [N-1:0] P, Q;
  wire PGTQ, PEQQ, PLTQ;
  integer ii, errors;

  Vr8bitcmp_sh UUT ( .P(P), .Q(Q), .PGTQ(PGTQ), .PEQQ(PEQQ), .PLTQ(PLTQ) );

  initial begin
    errors = 0;
    P = $random(SEED);    // Set pattern based on seed parameter
    for (ii=0; ii<10000; ii=ii+1) begin
      P = $random; Q = $random;
      #10 ;
      if ( (PGTQ !== (P>Q)) || (PLTQ !== (P<Q)) || (PEQQ !== (P==Q)) ) begin
        errors = errors + 1;
        $display("P=%b(%0d), Q=%b(%0d), PGTQ=%b, PEQQ=%b, PLTQ=%b",
                  P, P, Q, Q, PGTQ, PEQQ, PLTQ);
      end
    end
    $display("Test done, %0d errors", errors);
  end
endmodule
```

as discussed previously. They just add size and delay to the synthesized circuit. The only practical way to find such inferred latches is not through simulation and test benches, but by noticing any warning messages that are produced in synthesis. For example, Vivado warns "[Synth 8-327] inferring latch for variable PGTQ_reg [Vr8bitcmp_xi.v:7] (2 more like this)."

The second aspect of the test bench shows up only if you think really hard about it or, as I did, happen to look at the output waveforms produced by the UUT when the test bench runs. One or the other of the UUT's PGTQ and PLTQ outputs gets asserted, quite unpredictably, on almost every test cycle, but PEQQ is almost never asserted—only a few dozen times out of 10,000 test cycles! Once you see this, the "Duh" moment comes quite quickly—since P and Q are pseudorandom 8-bit numbers, they will be equal only one time out of 256 on the average, and that's assuming that $random even has the capability of producing two equal low-order 8-bit values in a row (which is not true for all pseudorandom number generators, depending on how they are constructed).

In this example, we tested 8-bit comparators; if we were testing 16-bit comparators, the coverage of the PEQQ output would be far worse, with only one of about 65,000 pseudorandom inputs asserting PEQQ. The fix for this deficiency is to modify our test bench to generate equals-cases to check along with the more typical cases that were easily generated above. The initial block and a helper task in the new version, VrNbitcmp_tb2, are shown in Program 7-24. Here, we generate just one random number but perform two tests per iteration of the for loop. First, we apply the current random number to both P and Q to test an equality case. Then, we generate a new random number and apply it to Q to test what's likely a greater-than or less-than case.

Program 7-24 Body of an improved test bench for an *N*-bit comparator.

```
  task checkcmp;
      if ( (PGTQ !== (P>Q)) || (PLTQ !== (P<Q)) || (PEQQ !== (P==Q)) ) begin
        errors = errors + 1;
        $display("P=%b(%0d), Q=%b(%0d), PGTQ=%b, PEQQ=%b, PLTQ=%b",
                  P, P, Q, Q, PGTQ, PEQQ, PLTQ);
      end
  endtask

  initial begin
    errors = 0;
    P = $random(SEED);    // Set pattern based on seed parameter
    for (ii=0; ii<10000; ii=ii+1) begin
      Q = P; #10 ; checkcmp;        // lots of = cases
      P = $random; #10 ; checkcmp;  // ... and mostly != cases
    end
    $display("Test done, %0d errors", errors);
  end
endmodule
```

This example goes to show that "random" test inputs, even on "data" inputs, don't necessarily provide uniform coverage of potential errors in datapath logic. Going back to the gate-level comparator design of Figure 7-26 on page 336, you can see that the PEQQ output is generated by logic that is somewhat distinct from PGTQ and PLTQ logic, and it deserves a thorough test in its own right. In a synthesized realization in an FPGA or other technology, and especially in a structural design, there are ample opportunities for errors, like misconnects in cascading, that affect PEQQ and not the other outputs and only for a small subset of input combinations.

So, not just in comparators but in all "datapath" circuits, it is important for the designer to recognize any input combinations that are handled specially or that cause unusual outputs, and to devise test-bench inputs that exercise these cases adequately.

After all this, we still haven't tested the 64-bit comparator module in Program 7-22. Either of the preceding two test benches with work with it, but they won't work very well. Recall once again that $random returns a 32-bit signed integer result. If P and Q are wider than 32 bits, then before the Verilog compiler assigns the random value to P or Q, it first sign-extends it to the required width. So, the high order bits of P and Q (beyond bit 31) will be all 0s or all 1s, not a very effective set of test inputs for those bits.

This problem can be remedied by enhancing the test bench further. The new version, VrNbitcmp_tb3, replaces the "P=$random" statement with a series of statements that calls $random multiple times to fill all of P if it is wider than 32 bits:

```
P[31:0] = $random;
if (N>32) P[63:32] = $random;
if (N>64) P[95:64] = $random;
if (N>96) P[127:96] = $random;
```

The code above works for vector widths up 128 bits. You might think that to eliminate the 128-bit width limitation, it would be better to write a more general for loop that calls $random as many times as necessary for a particular value of N. You would be right, except that the approach above would require a variable to be used in P's index for the assignment with $random, and most Verilog compilers don't support that, even in simulation.

It's important to understand that the comparator designs in Programs 7-16 through 7-21 all work on arbitrarily wide vector operands, even if the vectors are wider than the "native" integer width of the tools, which is always at least 32 bits and more typically 64 bits nowadays. That's true because modern Verilog tools know how to simulate and synthesize comparison operations on the wider vectors; the Verilog language reference manual (LRM) requires vector widths of at least 64K bits to be supported.

Program 7-25 *N*-bit comparator test bench using a reference UUT (U1).

```
`timescale 1 ns / 100 ps
module VrNbitcmp_tb4();
  parameter N = 64;     // Input width of comparator UUT
  parameter SEED = 1;  // Set a different pseudorandom seed here if desired
  reg [N-1:0] P, Q;
  wire PGTQ1, PEQQ1, PLTQ1, PGTQ2, PEQQ2, PLTQ2;
  integer ii, errors;

  task checkcmp;
    begin
      if ( (PGTQ1 !== PGTQ2) ||
           (PLTQ1 !== PLTQ2) ||
           (PEQQ1 !== PEQQ2) ) begin
      errors = errors + 1;
      $display("P=%b(%0d), Q=%b(%0d), PGTQ1=%b, PEQQ1=%b, PLTQ1=%b, PGTQ2=%b, PEQQ2=",
                "%b, PLTQ2=%b", P, P, Q, Q, PGTQ1, PEQQ1, PLTQ1, PGTQ2, PEQQ2, PLTQ2);
      end
    end
  endtask

  VrNbitcmp_d #(.N(N)) U1 ( .P(P), .Q(Q), .PGTQ(PGTQ1), .PEQQ(PEQQ1), .PLTQ(PLTQ1) );
  Vr64bitcmp_sh UUT ( .P(P), .Q(Q), .PGTQ(PGTQ2), .PEQQ(PEQQ2), .PLTQ(PLTQ2) );

  initial begin
    errors = 0;
    P = $random(SEED);       // Set pattern based on seed parameter
    for (ii=0; ii<10000; ii=ii+1) begin
      Q = P; #10 ; checkcmp; // lots of = cases
      P[31:0] = $random;
      if (N>32) P[63:32] = $random;
      if (N>64) P[95:64] = $random;
      if (N>96) P[127:96] = $random;
      #10 ; checkcmp;        // ... and mostly != cases
    end
    $display("Test done, %0d errors", errors);
  end
endmodule
```

Yet another approach to create a comparator test bench, or any test bench, is to compare the outputs of the UUT with those of a reference design using an appropriate set of inputs. In Program 7-25, we've used `VrNbitcmp_d` (`U1`) as the reference design, based on our confidence that Verilog "does the right thing" when comparing vectors, even wide ones, using its built-in operators. Then we compare its outputs with those of the UUT, the 64-bit hierarchical module `Vr64bitcmp_sh`, for a sequence of 10,000 equal and 10,000 mostly unequal random inputs, much the same as in the other test benches in this section.

TAKING A PASS You may very well want to skip the optional subsection below. It's all about the performance trade-offs that occur in different approaches to comparator design when targeting to a particular example technology, a large high-performance FPGA. If you'll soon be rolling up your sleeves on a project that requires many or very high-performance comparators, you may find the details and discussion illuminating. If not, you should be able to get by with just these takeaways:

- Modern synthesis tools are very good at creating reasonably sized, high-performance comparators that have been specified behaviorally, so you're unlikely to get more than a 10–15% improvement by "rolling your own" structural or hierarchical designs.
- The results obtained with any given design are highly dependent on both the targeted technology, which may or may not have elements for optimizing arithmetic functions including comparators, and the capabilities of the synthesis tool.
- Hierarchical designs look good "on paper," and typically yield the minimum number of logic levels and the shortest "logic delay." But again depending on the targeted technology, "your mileage may vary." In the FPGA-targeted examples in the next subsection, internal wiring and input/output buffer delays dominate the total delay: 15.14 ns out of 15.64 ns in our large 81-bit hierarchical comparator.

*7.4.8 Comparing Comparator Performance

Now that we've presented many different comparator designs, we can compare their relative speed and size in a particular technology. This exercise used the Xilinx Vivado tools to target a large, high-performance FPGA. Each configurable logic slice in this FPGA has one "CARRY4" logic element alongside each set of four LUTs. The CARRY4 element can be used to optimize the performance of large adders and subtractors; and it is present and available in the slice "for free" (except for delay) whether it's used or not. Therefore, the Vivado synthesis tool tries to implement magnitude comparators using subtractors—the first comparator design method we listed in Section 7.4.5.

Each row of Table 7-5 shows key results for one of the 8-bit magnitude comparator modules with different coding styles that we presented in Programs 7-16 through 7-20, as noted in the first four columns of the table. The last six columns give the following information; delays are in nanoseconds:

- "# of LUTs" is the total number of LUTs used after optimization, including "free" CARRY4 elements if any.
- "Logic Levels" is the number of logic levels in the worst-case path from input to output, including LUTs, CARRY4 elements if any, and the input and output buffers that drive the inputs and outputs on and off chip.

* Throughout this book, optional sections are marked with an asterisk.

- "Delay (est.)" is the estimated worst-case delay after the tool synthesizes the circuit but before it actually places and routes the circuit on the chip, including connections to the chip's input/output pins.
- "Delay (final)" is the actual worst-case delay as calculated after layout.
- "Logic Delay" is the portion of the delay resulting from LUTs, CARRY4 elements, and input/output buffers; it omits the delay of on-chip wiring.
- "Comp. Delay" is the delay of just the actual comparator logic—no on-chip-wiring and no input/output buffers.

The first module, Vr8bitcmp_xi, has six logic levels in its worst-case signal path, an estimated delay of 8.57 ns, and a final delay of 10.50 ns calculated after layout. It's interesting that in this and all of these examples, about half or more of the final delay is in the on-chip wiring—simply connecting the output of one logic element to the input of another. In Vr8bitcmp_xi, only 5.10 ns of the final delay is in logic elements—LUTs, CARRY4 elements, and input/output buffers. Drilling down even further, it turns out that most of that delay is in the input/output buffers; only 1.53 ns is in the four levels of LUTs and CARRY4 elements that implement the actual comparator function.

Despite the dominance of wiring and input/output in the speed of these five comparator modules, we can still see differences. Clearly the unwanted, inferred latch in Vr8bitcmp_xi is costly—eliminating it in Vr8bitcmp_xc saves both delay and LUTs. In Vr8bitcmp, eliminating the extra comparator saves LUTs, but it doesn't have much effect on delay. In fact, logic delay goes down while final delay goes up, perhaps due to quirks in the layout.

Comparing the Vr8bitcmp_dx dataflow module with Vr8bitcmp_xc, we still have an extra comparator but the implementation has gotten smaller and faster, with one less logic level. Eliminating the extra comparator in Vr8bitcmp_dx further reduces the number of LUTs, but increases delay with one more logic level—why? In this case, the compiler is faithfully implementing PLTQ as a function of PEQQ and PGTQ as specified in Program 7-20, requiring one more level of logic which the synthesis tool could not eliminate.

Table 7-5 Synthesis and implementation results for 8-bit comparators with various coding styles.

Module Name	Bits	Coding Style	Notes	# of LUTs	Logic Levels	Delay (est.)	Delay (final)	Logic Delay	Comp. Delay
Vr8bitcmp_xi	8	behavioral	inferred latch	18	6	8.57	10.50	5.10	1.53
Vr8bitcmp_xc	8	behavioral	extra comp.	14	5	6.88	9.09	4.34	0.78
Vr8bitcmp	8	behavioral		10	5	6.88	9.21	4.21	0.63
Vr8bitcmp_dx	8	dataflow	extra comp.	12	4	6.26	8.09	4.10	0.53
Vr8bitcmp_d	8	dataflow		9	5	6.88	9.23	4.30	0.73

In general, explicitly deriving one signal from others, specifying a more "serial" design, may increase delay while reducing "redundant" logic. Keep in mind that all of these Verilog modules specify the same 8-bit comparator logic function, but even for a relatively small design like this, a synthesis tool cannot explore all possible implementations and opportunities for optimization. So, the synthesis result still depends on the starting point.

Among the five different 8-bit comparator module designs in Table 7-5, Vr8bitcmp_dx has the fastest implementation, while Vr8bitcmp_d has the smallest. As they say, "your mileage may vary," in this case as a function of the target technology, the synthesis tool, the overall coding style, and even small details of the code, which may lead the synthesis tool down one implementation path versus another. Two things we know for certain from these examples is that inferred latches are bad, and there may be a size versus speed trade-off when outputs are specified independently rather than being derived from other outputs .

We can also explore results for comparator modules of different sizes and add hierarchical implementations to the mix. A LUT in a typical FPGA technology (including the Xilinx 7-series used for Table 7-5) has just six inputs, enough

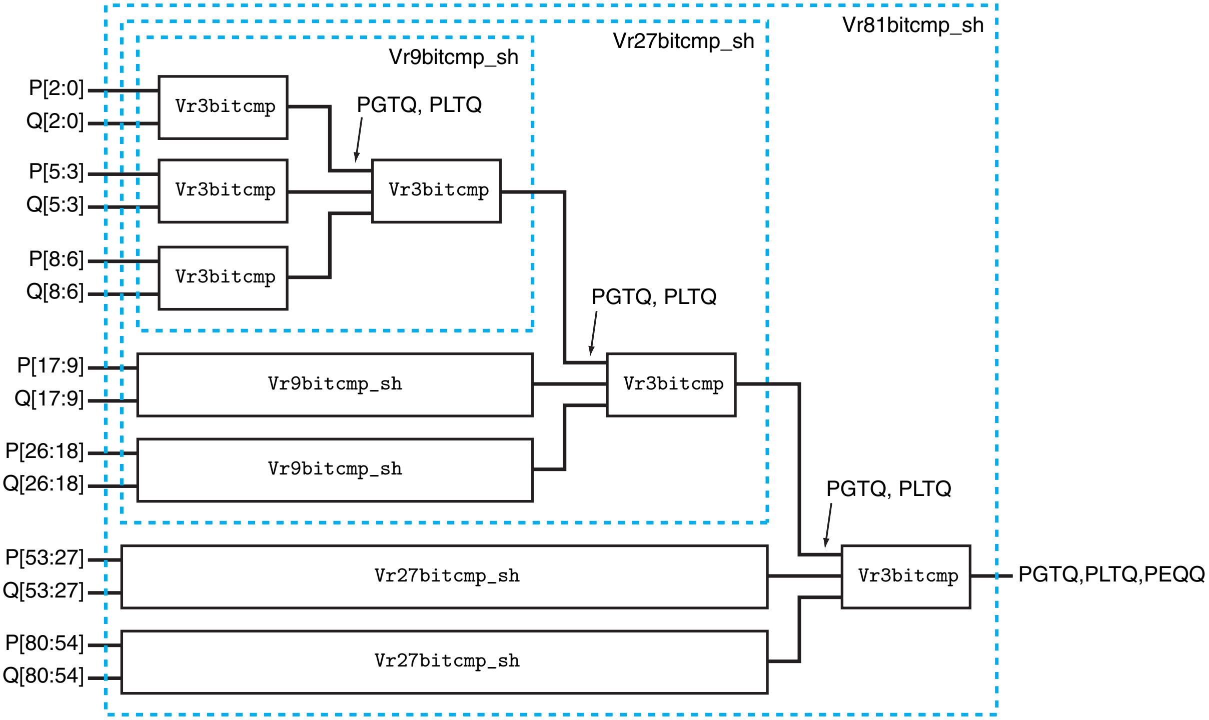


Figure 7-29 Hierarchical structure for an 81-bit comparator using 3-bit comparators (1 LUT/output).

Program 7-26 Structural Verilog for the 81-bit comparator of Figure 7-29.

```
module Vr81bitcmp_sh(P, Q, PGTQ, PEQQ, PLTQ);
  input [80:0] P, Q;
  output PGTQ, PEQQ, PLTQ;
  wire  GT0, EQ0, LT0, GT1, EQ1, LT1, GT2, EQ2, LT2;

  Vr27bitcmp_sh U3(P[80:54], Q[80:54], GT2, EQ2, LT2);
  Vr27bitcmp_sh U2(P[53:27], Q[53:27], GT1, EQ1, LT1);
  Vr27bitcmp_sh U1(P[26:0], Q[26:0], GT0, EQ0, LT0);
  Vr3bitcmp U4({GT2, GT1, GT0}, {LT2, LT1, LT0}, PGTQ, PEQQ, PLTQ);
endmodule

(* keep_hierarchy = "yes" *) module Vr27bitcmp_sh(P, Q, PGTQ, PEQQ, PLTQ);
  input [26:0] P, Q;
  output PGTQ, PEQQ, PLTQ;
  wire  GT0, EQ0, LT0, GT1, EQ1, LT1, GT2, EQ2, LT2;

  Vr9bitcmp_sh U3(P[26:18], Q[26:18], GT2, EQ2, LT2);
  Vr9bitcmp_sh U2(P[17:9], Q[17:9], GT1, EQ1, LT1);
  Vr9bitcmp_sh U1(P[8:0], Q[8:0], GT0, EQ0, LT0);
  Vr3bitcmp U4({GT2, GT1, GT0}, {LT2, LT1, LT0}, PGTQ, PEQQ, PLTQ);
endmodule

(* keep_hierarchy = "yes" *) module Vr9bitcmp_sh(P, Q, PGTQ, PEQQ, PLTQ);
  input [8:0] P, Q;
  output PGTQ, PEQQ, PLTQ;
  wire  GT0, EQ0, LT0, GT1, EQ1, LT1, GT2, EQ2, LT2;

  Vr3bitcmp U3(P[8:6], Q[8:6], GT2, EQ2, LT2);
  Vr3bitcmp U2(P[5:3], Q[5:3], GT1, EQ1, LT1);
  Vr3bitcmp U1(P[2:0], Q[2:0], GT0, EQ0, LT0);
  Vr3bitcmp U4({GT2, GT1, GT0}, {LT2, LT1, LT0}, PGTQ, PEQQ, PLTQ);
endmodule

(* keep_hierarchy = "yes" *) module Vr3bitcmp(P, Q, PGTQ, PEQQ, PLTQ);
  input [2:0] P, Q;
  output reg PGTQ, PEQQ, PLTQ;

  always @ (P or Q)
    if (P == Q)
      begin PGTQ = 1'b0; PEQQ = 1'b1; PLTQ = 1'b0; end
    else if (P > Q)
      begin PGTQ = 1'b1; PEQQ = 1'b0; PLTQ = 1'b0; end
    else
      begin PGTQ = 1'b0; PEQQ = 1'b0; PLTQ = 1'b1; end
endmodule
```

for one output of just a 3-bit comparator. The first row of Table 7-6 shows the size and delay of such a 3-bit comparator. Each of the comparator's three outputs is produced by a single LUT having a delay of just 0.13 ns, as shown in the last column of the table, but as in other examples, on-chip wiring and input/output buffers add considerable delay to the overall circuit.

To build larger comparators, an efficient approach might be to model them using a hierarchical, tree-based structure, as we explained at the end of Section 7.4.4, starting with 3-bit comparators (one LUT per output) as the basic building block. As illustrated in Figure 7-29 on page 352, a 9-bit comparator can be built using three 3-bit comparators at the first level of the tree, and combining their outputs using a single 3-bit comparator at the second level. A 27-bit comparator can be built using three such 9-bit comparators and again combining their outputs with a single 3-bit comparator. And an 81-bit comparator can be built by combining the outputs of three such 27-bit comparators. The Verilog code for the top-level 81-bit comparator module and all the levels below it is shown in Program 7-26 on page 353.

Table 7-6 Synthesis and implementation results for comparators with various sizes and coding styles.

Module Name	Bits	Coding Style	Notes	# of LUTs	Logic Levels	Delay (est.)	Delay (final)	Logic Delay	Comp. Delay
Vr3bitcmp	3	behavioral	1 LUT/output	3	3	5.23	6.88	3.62	0.13
Vr9bitcmp_xc	9	behavioral	extra comp.	14	6	7.04	9.93	5.05	1.30
Vr9bitcmp	9	behavioral		9	6	6.81	9.51	5.05	1.30
Vr9bitcmp_dx	9	dataflow	extra comp.	13	5	6.37	8.43	4.67	0.91
Vr9bitcmp_d	9	dataflow		9	5	6.81	9.47	4.66	0.98
Vr9bitcmp_sh	9	hierarchical		9	4	6.48	8.26	3.82	0.25
Vr27bitcmp_xc	27	behavioral	extra comp.	38	8	7.30	14.12	5.17	1.60
Vr27bitcmp	27	behavioral		24	7	7.05	14.00	5.22	1.65
Vr27bitcmp_dx	27	dataflow	extra comp.	37	7	6.46	12.87	4.83	1.26
Vr27bitcmp_d	27	dataflow		24	7	7.02	13.54	4.95	1.38
Vr27bitcmp_sh	27	hierarchical		27	5	7.74	13.61	3.94	0.38
Vr81bitcmp_xc	81	behavioral	extra comp.	110	15	8.01	16.66	5.96	2.39
Vr81bitcmp	81	behavioral		69	15	7.86	16.42	6.11	2.54
Vr81bitcmp_dx	81	dataflow	extra comp.	109	14	7.42	15.40	5.64	2.07
Vr81bitcmp_d	81	dataflow		69	14	7.86	16.27	5.75	2.18
Vr81bitcmp_sh	81	hierarchical		81	6	8.99	15.64	4.15	0.50

Table 7-6 includes synthesis results for the structured hierarchical modules (named with suffix "`_sh`") and also includes corresponding results for other design approaches, where we have simply changed the width to 9, 27, or 81 bits.

The table shows that when we triple the input width in a "`_sh`" hierarchical design, we triple the number of LUTs but add just one level of logic delay for the actual comparator function, about 0.13 ns per level. Yet the final delay for each design is still dominated by input/output buffer and on-chip wiring delays.

A big jump in delay occurs between all of the 9-bit designs and the 27-bit designs. This is not because there is anything particularly bad about a 27-bit design, except that in the target technology, the number of input and output signals (57) is now large enough to use I/O pins on both sides of the physical FPGA chip. As a result, some signal paths most cross the entire chip, which increases the worst-case wiring delay.

Examining the 9-, 27-, and 81-bit behavioral- and dataflow-style designs, the table shows results similar to what we saw in Table 7-5 for corresponding 8-bit designs. The designs with an extra comparator require more LUTs, though in some but not all cases, the extra comparator results in a shorter final delay. Where it does not, for example in `Vr27bitcmp_xc` versus `Vr27bitcmp`, the extra comparator reduces *logic* delays, but increases wiring delays even more—because there are more LUTs to interconnect, spanning a larger area of the chip.

Comparing each hierarchical design with the same-size behavioral and dataflow designs, we see that the hierarchical designs have the most consistent and predictable size and logic delay. However, the other designs, especially because of their use of "free" CARRY4 elements, are sometimes smaller or faster or both.

So, what does it all mean? In the targeted FPGA technology, wiring delay is a large portion of overall delay and reduces the benefits of reducing levels of logic, and "free" resources like CARRY4 reduce the relative advantages of the efficient, hierarchical approach. If the same designs were targeted to a typical ASIC technology, where every gate consumes chip area and wiring is non-programmable and therefore faster, the results would likely be different.

MODELING CHOICES AND IMPLEMENTATION

As you can see from the examples in this section, it's hard to predict whether a particular Verilog modeling style will yield the best size or speed of a design's implementation in any given technology. As in other types of programming and coding, a primary goal should be to assure understandability and maintainability, focusing on performance (smaller size or higher speed) only when necessary.

Also as in other types of programming, the 80/20 rule (also known as the Pareto principle) usually holds true: 20% of the code is responsible for 80% of the performance. Thus, a designer should not worry unduly about performance tuning until the parts of the design that most affect total performance can be identified.

*7.5 A Random-Logic Example in Verilog

It's quite possible that the requirements for a combinational logic circuit do not fit within the structure of the building-block functions of this or the previous chapter, or the arithmetic functions of the next. And it may be quite challenging to try to write logic equations for the circuit directly. Still, it may be quite straightforward to write behavioral Verilog according to the requirements, and then to synthesize a corresponding circuit.

Such a "random-logic" example is a combinational circuit that picks a player's next move in Tic-Tac-Toe, the traditional kids' game of Xs and Os. The circuit's inputs encode the current state of the game's 3 × 3 grid and the outputs identify the cell for the next move. To avoid confusion between "O" and "0" later in our Verilog code, we'll call the second player "Y".

There are many ways to code the state of one cell in the grid. Because the game is symmetric, we'll use a symmetric encoding that can help later:

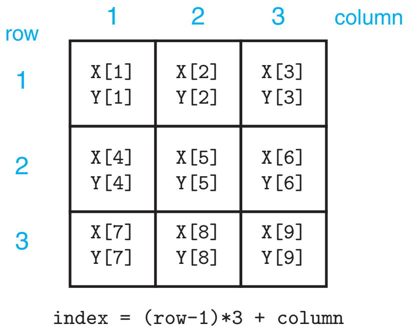
00	Cell is empty.
10	Cell is occupied by X.
01	Cell is occupied by Y.

So, we can encode the 3 × 3 grid's state in 18 bits—nine bits to indicate which cells are occupied by X, and nine more to indicate which ones are occupied by Y. Throughout the Verilog Tic-Tac-Toe modules in this subsection, we'll use a pair of 9-bit vectors `X[1:9]` and `Y[1:9]` to represent the Tic-Tac-Toe grid. A vector bit is 1 if the like-named player has a mark in the corresponding cell. Figure 7-30 shows the correspondence between signal names and cells in the grid. To translate between two-dimensional (row, column) coordinates in the grid and a bit number in `X[1:9]` or `Y[1:9]`, we use the formula in the figure.

We also need an encoding for moves. A player has nine possible moves, so the encoding should define nine values plus one for the case where no move is possible. The parameter definitions in Program 7-27 correspond to one possible 4-bit move encoding. A name like "`MOVE12`" denotes a move to row 1, column 2 of the grid. Different encodings might lead to smaller, larger, faster, or slower circuits. The parameter definitions in the table are stored in a file, `TTTdefs.v`,

TIC-TAC-TOE, IN CASE YOU DIDN'T KNOW

The game of Tic-Tac-Toe is played by two players on a 3 × 3 grid of cells that are initially empty. One player is "X" and the other is "O". The players alternate in placing their mark in an empty cell; "X" always goes first. The first player to get three of his or her own marks in the same row, column, or diagonal wins. Although the first player to move (X) has a slight advantage, it can be shown that a game between two intelligent players will always end in a draw; neither player will get three in a row before the grid fills up.



row	1	2	3
1	X[1] Y[1]	X[2] Y[2]	X[3] Y[3]
2	X[4] Y[4]	X[5] Y[5]	X[6] Y[6]
3	X[7] Y[7]	X[8] Y[8]	X[9] Y[9]

`index = (row-1)*3 + column`

Figure 7-30 Tic-Tac-Toe grid and Verilog signal names.

which is `include`'d in the modules as needed. Thus, we can easily change the move encoding later, in one place, without having to change the modules that use it (for example, see Exercise 7.50).

Now we need a strategy for picking the next move, so we can create a behavioral model that uses it. Let us try to emulate the typical human's strategy by following the decision steps below:

1. Look for a row, column, or diagonal that has two of my marks (X or Y, depending on which player I am) and one empty cell. If one exists, place my mark in the empty cell; I win!
2. Else, look for a row, column, or diagonal that has two of my opponent's marks and one empty cell. If one exists, place my mark in the empty cell to block a potential win by my opponent.

Program 7-27 `TTTdefs.v` definition file for the Tic-Tac-Toe project.

```
parameter MOVE11 = 4'b1000,
          MOVE12 = 4'b0100,
          MOVE13 = 4'b0010,
          MOVE21 = 4'b0001,
          MOVE22 = 4'b1100,
          MOVE23 = 4'b0111,
          MOVE31 = 4'b1011,
          MOVE32 = 4'b1101,
          MOVE33 = 4'b1110,
          NONE   = 4'b0000;
```

A VERILOG-2001 LIMITATION

It would be nice to declare the state of the Tic-Tac-Toe grid as two 2-dimensional arrays, `X[1:3][1:3]` and `Y[1:3][1:3]`. Unfortunately, Verilog-2001 does not allow arrays to be used as module ports, and in our hierarchical design of the Tic-Tac-Toe circuit, we need to do that. Hence, we have declared `X` and `Y` as simple 9-bit vectors and we translate from "`i,j`" to a bit number within a vector as shown in Figure 7-30.

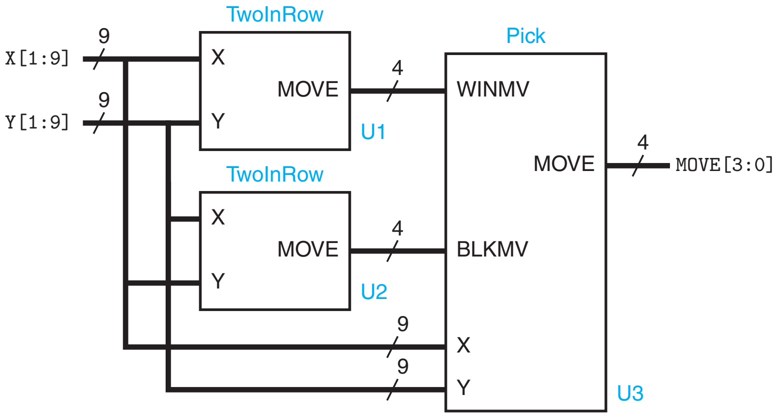


Figure 7-31 Module partitioning for the Tic-Tac-Toe game.

3. Else, pick a cell based on experience. For example, if the middle cell is open, it's usually a good bet to take it. Otherwise, the corner cells are good bets. Intelligent players can also notice and block a developing pattern by the opponent or "look ahead" to pick a good move.

Rather than try to design the Tic-Tac-Toe move-finding circuit as a single monolithic module, it makes sense for us to try to partition it into smaller pieces. In fact, partitioning it along the lines of the three-step strategy that we gave at the beginning of this section seems like a good idea.

We note that steps 1 and 2 of our strategy are very similar; they differ only in reversing the roles of the player and the opponent. A circuit that finds a winning move for me can also find a blocking move for my opponent. Looking at this characteristic from another point of view, a circuit that finds a winning move for me can find a blocking move for me if the encodings for me and my opponent are swapped. Here's where our symmetric encoding pays off—we can swap players merely by swapping signals X[1:9] and Y[1:9].

With this in mind, we can use two copies of the same module, TwoInRow, to perform steps 1 and 2 as shown in Figure 7-31. Notice that signal X[1:9] is connected to the top input of the first TwoInRow module, but to the bottom input of the second; similarly for Y[1:9]. A third module, Pick, picks a winning move

Program 7-28 Top-level structural Verilog module for picking a move.

```
module GETMOVE ( X, Y, MOVE );
  input [1:9] X, Y ;
  output [3:0] MOVE;
  wire [3:0] WIN, BLK;

  TwoInRow U1 ( .X(X), .Y(Y), .MOVE(WIN) );
  TwoInRow U2 ( .X(Y), .Y(X), .MOVE(BLK) );
  Pick U3 ( .X(X), .Y(Y), .WINMV(WIN), .BLKMV(BLK), .MOVE(MOVE) );
endmodule
```

if one is available from U1, else it picks a blocking move if available from U2, else it uses "experience" (step 3) to pick a move.

Program 7-28 is structural Verilog code for the top-level module, GETMOVE. It instantiates two other modules, TwoInRow and Pick, which will be defined shortly. Its only internal signals are WIN and BLK, which pass winning and blocking moves from the two instances of TwoInRow to Pick, as in Figure 7-31. The statement part of the module has just three statements to instantiate the three blocks in the figure.

Now we need to design the individual modules in Figure 7-31. Let's do a "top-down" design and work on Pick next. In a top-down design, it's usually possible to "stub in" simplified versions of the lower-level modules in order to test and refine the higher-level ones, though we won't have to do that here. The Pick module in Table 7-29 uses fairly straightforward, deeply nested if-else statements to select a move. First priority is given to a winning move, followed by a blocking move. Otherwise, function MT is called for each cell, from best (middle) to worst (side), to find an available move.

Program 7-29 Verilog module to pick a winning or blocking Tic-Tac-Toe move or else pick a move using "experience."

```
module Pick ( X, Y, WINMV, BLKMV, MOVE);
  input [1:9] X, Y;
  input [3:0] WINMV, BLKMV;
  output reg [3:0] MOVE;
  `include "TTTdefs.v"

  function MT;  // Determine if cell i,j is empty
    input [1:9] X, Y;
    input [1:0] i, j;
    MT = ~X[(i-1)*3+j] & ~Y[(i-1)*3+j];
  endfunction

  always @ (X or Y or WINMV or BLKMV) begin // If available, pick:
    if       (WINMV != NONE) MOVE = WINMV;   // winning move
    else if (BLKMV != NONE) MOVE = BLKMV;   // else blocking move
    else if (MT(X,Y,2,2))   MOVE = MOVE22;  // else center cell
    else if (MT(X,Y,1,1))   MOVE = MOVE11;  // else corner cells
    else if (MT(X,Y,1,3))   MOVE = MOVE13;
    else if (MT(X,Y,3,1))   MOVE = MOVE31;
    else if (MT(X,Y,3,3))   MOVE = MOVE33;
    else if (MT(X,Y,1,2))   MOVE = MOVE12;  // else side cells
    else if (MT(X,Y,2,1))   MOVE = MOVE21;
    else if (MT(X,Y,2,3))   MOVE = MOVE23;
    else if (MT(X,Y,3,2))   MOVE = MOVE32;
    else                    MOVE = NONE;    // else grid is full
  end
endmodule
```

The TwoInRow module requires more work, as shown in Program 7-30. This module defines four functions, each of which determines whether there is a winning move (from X's point of view) in a particular cell i,j. A winning move exists if cell i,j is empty and the other two cells in the same row, column, or

Program 7-30 Behavioral Verilog TwoInRow module.

```
module TwoInRow ( X, Y, MOVE );
  input [1:9] X, Y;
  output reg [3:0] MOVE;
  reg G11, G12, G13, G21, G22, G23, G31, G32, G33;
  `include "TTTdefs.v"

  function R;  // Find 2-in-row with empty cell i,j
    input [1:9] X, Y;
    input [1:0] i, j;
    integer jj;
    begin
      R = 1'b1;
      for (jj=1; jj<=3; jj=jj+1)
        if (jj==j) R = R & ~X[(i-1)*3+jj] & ~Y[(i-1)*3+jj];
        else R = R & X[(i-1)*3+jj];
    end
  endfunction

  function C;  // Find 2-in-column with empty cell i,j
    input [1:9] X, Y;
    input [1:0] i, j;
    integer ii;
    begin
      C = 1'b1;
      for (ii=1; ii<=3; ii=ii+1)
        if (ii==i) C = C & ~X[(ii-1)*3+j] & ~Y[(ii-1)*3+j];
        else C = C & X[(ii-1)*3+j];
    end
  endfunction

  function D;          // Find 2-in-diagonal with empty cell i,j
    input [1:9] X, Y;  // This is for 11, 22, 33 diagonal
    input [1:0] i, j;
    integer ii;
    begin
      D = 1'b1;
      for (ii=1; ii<=3; ii=ii+1)
        if (ii==i) D = D & ~X[(ii-1)*3+ii] & ~Y[(ii-1)*3+ii];
        else D = D & X[(ii-1)*3+ii];
    end
  endfunction
```

diagonal contain an X. Functions R and C look for winning moves in cell i,j's row and column, respectively. Functions D and E look in the two diagonals.

Within the module's always block, nine 1-bit variables G11–G33 are used to indicate whether each of the cells has a winning move possible. Assignment statements at the beginning of the block set each variable to 1 if there is such a move, calling and combining all of the appropriate functions for cell i,j.

The rest of the module is a series of deeply nested if-else statements that look in all possible cells for a winning move. If none is possible, the value NONE is assigned. As we showed before, two instances of the TwoInRow module are instantiated with Pick in Program 7-28 to complete the Tic-Tac-Toe model.

Program 7-30 (continued)

```
function E;          // Find 2-in-diagonal with empty cell i,j
    input [1:9] X, Y; // This is for 13, 22, 31 diagonal
    input [1:0] i, j;
    integer ii;
    begin
      E = 1'b1;
      for (ii=1; ii<=3; ii=ii+1)
        if (ii==i) E = E & ~X[(ii-1)*3+4-ii] & ~Y[(ii-1)*3+4-ii];
        else E = E & X[(ii-1)*3+4-ii];
    end
  endfunction

  always @ (X or Y) begin
    G11 = R(X,Y,1,1) | C(X,Y,1,1) | D(X,Y,1,1);
    G12 = R(X,Y,1,2) | C(X,Y,1,2);
    G13 = R(X,Y,1,3) | C(X,Y,1,3) | E(X,Y,1,3);
    G21 = R(X,Y,2,1) | C(X,Y,2,1);
    G22 = R(X,Y,2,2) | C(X,Y,2,2) | D(X,Y,2,2) | E(X,Y,2,2);
    G23 = R(X,Y,2,3) | C(X,Y,2,3);
    G31 = R(X,Y,3,1) | C(X,Y,3,1) | E(X,Y,3,1);
    G32 = R(X,Y,3,2) | C(X,Y,3,2);
    G33 = R(X,Y,3,3) | C(X,Y,3,3) | D(X,Y,3,3);
    if      (G11) MOVE = MOVE11;
    else if (G12) MOVE = MOVE12;
    else if (G13) MOVE = MOVE13;
    else if (G21) MOVE = MOVE21;
    else if (G22) MOVE = MOVE22;
    else if (G23) MOVE = MOVE23;
    else if (G31) MOVE = MOVE31;
    else if (G32) MOVE = MOVE32;
    else if (G33) MOVE = MOVE33;
    else          MOVE = NONE;
  end
endmodule
```

ANOTHER CASE

When I targeted the Tic-Tac-Toe model to a Xilinx 7-series FPGA using Vivado tools, the synthesized design used 63 LUTs with a maximum delay path of 5 LUTs.

Since the nested `if-else` statements in `TwoInRow` create a priority encoder of sorts, they can be replaced with a `case` statement in the style of Program 7-8 on page 318. So, to try for a better synthesis result, I did that, using the new code in Program 7-31. To my amazement, the new design used 668 LUTs! All by itself, `TwoInRow` required 10 times as many LUTs (210 vs. 21)!

Since I couldn't believe my eyes, I wrote a test bench to compare the outputs of both versions of `TwoInRow` for all 2^{18} input combinations, expecting find an error or at least a difference in semantics that would have made one version's function more difficult than the other to synthesize. No, the functions performed by the two models were exactly the same.

So, what is the moral of the story? Very large "random-logic" functions may give seemingly random synthesis results, and you may get significantly better (or worse!) results using a different model for the same thing.

Program 7-31 Two-in-a-row detection using a `case` statement.

```
case (1'b1)
  (G11): MOVE = MOVE11;
  (G12): MOVE = MOVE12;
  (G13): MOVE = MOVE13;
  (G21): MOVE = MOVE21;
  (G22): MOVE = MOVE22;
  (G23): MOVE = MOVE23;
  (G31): MOVE = MOVE31;
  (G32): MOVE = MOVE32;
  (G33): MOVE = MOVE33;
  default MOVE = NONE;
endcase
```

TIME FOR A BREAK

More combinational logic functions will be discussed in Chapter 8, including gate-level, building-block, and Verilog descriptions as in this chapter. All of those functions are ones for which Verilog has built-in operators: comparing, adding, shifting, multiplying, and dividing. Therefore, if you're doing an HDL-based design, and implementation size and performance are not critical, it is perfectly reasonable to use Verilog's operators and let the synthesis tool do the heavy lifting. In fact, it's quite common in HDL-based design to initially write, test, and synthesize behavioral code in these and other areas, and come back to them with technology-targeted (perhaps structural) modules only when size and performance are known to be issues.

For those reasons, it's perfectly alright for you to skip Chapter 8 for now, and move on to the "good stuff" on sequential circuits beginning in Chapter 9.

Drill Problems

7.1 What's terribly wrong with the circuit in Figure X7.1? Suggest a change that eliminates the terrible problem.

7.2 What is the function of the block labeled "LUT1" in Figure 7-9?

7.3 Write a behavioral-style Verilog module `Vr32inprior3` for a 32-input priority encoder with inputs, outputs, and functions similar to those of the 8-input priority encoder in Program 7-7. Synthesize each module, targeting to an FPGA, and compare their size and speed—number of LUTs and levels of LUT delay.

7.4 An odd-parity circuit with 2^n inputs can be built with $2^n - 1$ XOR gates. Describe two different structures for this circuit, one of which gives a minimum worst-case input to output propagation delay and the other of which gives a maximum. For each structure, state the worst-case number of XOR-gate delays, and describe a situation where that structure might be preferred over the other.

7.5 A certain parity circuit in the style of Figure 7-16(a) uses an odd number of XNOR gates. Does it generate odd parity, even parity, or neither? If neither, what function does it generate?

7.6 What is the maximum number of inputs of an even-parity function that can be realized in a single Xilinx 7-series LUT with the structure shown in Figure 6-6?

7.7 Construct a table that shows the number of LUTs and speed (maximum number of LUT delays) of an *n*-input even-parity tree built with Xilinx 7-series LUTs. In each row, the first column of the table should give a range of values for *n*, and the second and third columns give the number of LUTs required and the maximum

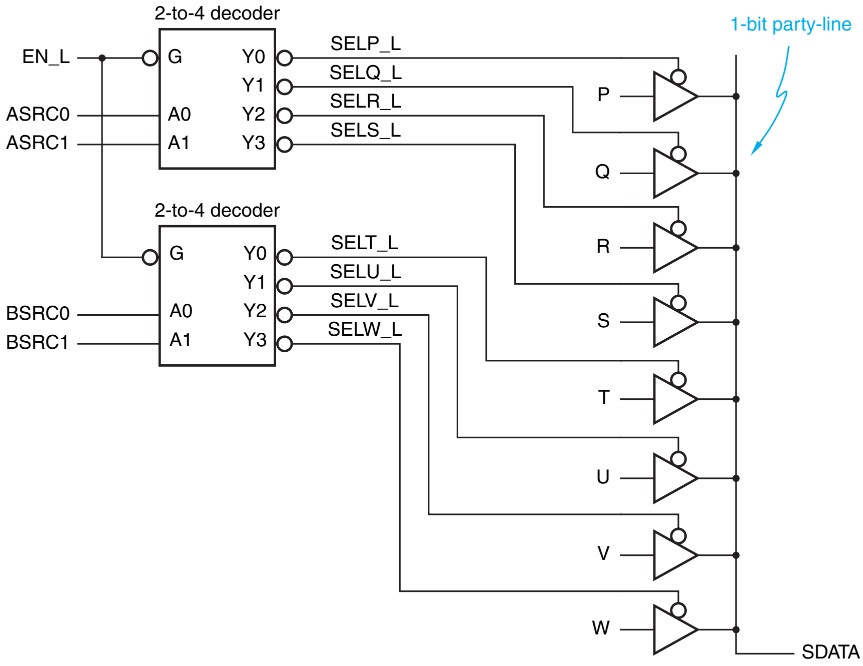


Figure X7.1

number of LUTs in the signal path from any input to output, respectively. Your table should have enough rows to cover all values of *n* from 1 to 99. If you wish, you may write and synthesize a Verilog module `Vrbigxor` to spot check a few of your entries at key break points in the table.

7.8 Suppose you had to realize the Hamming error-correction circuit in Figure 7-18 using a 3-to-8 decoder with active-low outputs like the 74x138's. What changes could you make to the circuit to avoid adding eight inverters to flip the active level of the decoder outputs?

7.9 Draw a logic diagram, in the style of Figure 7-21(b), for a 4-bit comparator using XNOR and AND gates. Be sure to use logic symbols and signal names that make sense in terms of active levels.

7.10 Starting with the magnitude-comparator logic diagram in Figure 7-26, write a logic expression for the PEQQ output in terms of the inputs.

7.11 Write a parameterized behavioral Verilog module `VrNbitcmp` for a comparator with *n*-bit input vectors `P` and `Q` and outputs `PGTQ`, `PLTQ`, and `PEQQ`. Test your module with the test bench in Program 7-24 for values of `N` of 8 and 32.

7.12 Augment the magnitude-comparator logic diagram in Figure 7-26 to provide two additional outputs, SPLTQ and SPGTQ, that give the comparison result for signed, two's-complement numbers. Do not redraw the whole logic diagram; just show how additional logic gates should be connected to existing signals.

7.13 Concisely describe in words the function performed by the block diagram on page 301, assuming the top input is named SEL, the *n*-bit input buses are, in order, X and Y, and the *n*-bit output bus is Z.

7.14 Using the same assumptions as in Drill 7.13, write a dataflow-style Verilog module corresponding to the block diagram, using continuous assignments to all of the signals, including local wires corresponding to the internal signals named in the block diagram. Use a parameter for *n* with a default value of 8.

7.15 Using the same assumptions as in Drill 7.13, write a concise dataflow-style Verilog module that performs the same function as the block diagram using just one continuous assignment statement and no local wires. Use a parameter for *n* with a default value of 8.

7.16 After doing Drills 7.14 and 7.15, write a test bench that compares the outputs of the two modules against the expected values and each other for all input values.

Exercises

7.17 This exercise is meant to show the importance of specifying desired behaviors. If you don't spec it, you may not test for it, and if you don't test for it, you may not always get it.

The specification of the A outputs in the 8-input priority encoder in Section 7.2 is a little ambiguous: "Outputs A2–A0 contain the number of the highest-priority asserted input, if any." What value should they contain if no input is asserted, or are they "don't-cares"? Our logic equations and our subsequent Verilog modules assumed they should be all 0s, so let's add that to the spec now.

Show that if the statement "`A = 3'd0`" is not included in the last line of the `begin-end` block in Program 7-6, the resulting priority encoder sometimes produces an incorrect output even after it's been running for a long time, but the test bench in Program 7-9 fails to detect the error. Explain the nature of the error, and modify the test bench so that it detects the error.

7.18 Twenty years ago, a famous logic designer decided to quit teaching and make a fortune by licensing the circuit design shown in Figure X7.18.

(a) Label the inputs and outputs of the circuit with appropriate signal names, including active-level indications.

(b) What does the circuit do? Be specific and account for all inputs and outputs.

(c) Draw the logic symbol that would go on the data sheet of this circuit.

(d) Write a behavioral Verilog model for the circuit.

(e) With what standard building blocks did the new circuit compete? Do you think that it was successful as an MSI part?

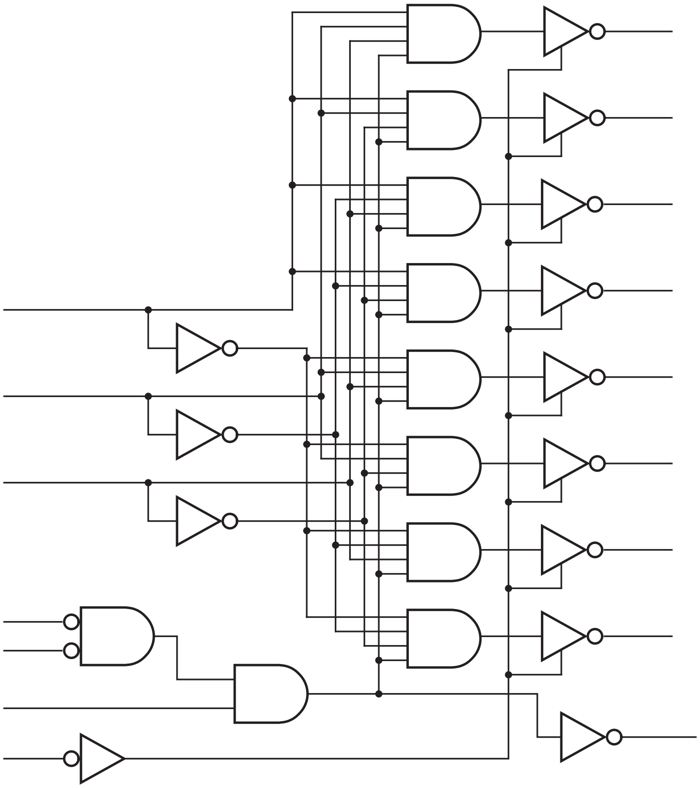


Figure X7.18

7.19 Modify the 8-input priority encoder module of Program 7-10, which uses a `casez` statement, so it does not require its case choices to be written in the order of their priority. Use the test bench of Program 7-9 to check your new module for proper operation, both as you originally write it and after scrambling the choices. *Hint*: Fewer than three dozen characters in the original module must be changed.

7.20 Write a behavioral-style Verilog module `Vr8inpriorcasc` with the same inputs, outputs, and functions as the cascadable 8-input priority encoder shown in Figure 7-12. Synthesize the module, targeting to an FPGA, and determine its size and speed—number of LUTs and levels of LUT delay.

7.21 Write a structural-style Verilog module `Vr32inpriorcasc` that uses four copies of the cascadable 8-input priority encoder `Vr8inpriorcasc` in Exercise 7.20 using the structure of Figure 7-13 to create a 32-input priority encoder. Synthesize the module, targeting to an FPGA, and compare its size and speed with the "simple" 32-input priority encoder of Drill 7.3—number of LUTs and levels of LUT delay. Was the improvement, if any, worth the extra effort?

7.22 Modify the 32-input priority-encoder modules of Drill 7.3 and Exercise 7.21 to have only 24 inputs. Then write a test bench that instantiates them and compares their outputs for all 16 million input combinations. Are they always the same? If not, explain. Naturally, the test bench must accommodate the modules' different signal names and the fact that `IDLE` in one module equals `~RGS` in the other.

7.23 Draw the logic diagram for a circuit that uses the cascadable priority encoder of Figure 7-12 to resolve priority among eight active-low inputs, I0_L–I7_L, where I7_L has the highest priority. The circuit should produce active-low address outputs A2_L–A0_L to indicate the number of the highest-priority asserted input. If no input is asserted, then A2_L–A0_L should be 111, and an active-low IDLE_L output should be asserted. You may use discrete gates in addition to the priority encoder. Be sure to name all signals with the proper active levels.

7.24 Draw the logic diagram for a circuit that uses the cascadable priority encoder of Figure 7-12 to resolve priority among eight active-high inputs, I0–I7, where I0 has the highest priority. The circuit should produce three active-low address outputs A2_L–A0_L to indicate the number of the highest-priority asserted input. If at least one input is asserted, then an AVALID output should be asserted. Be sure to name all signals with the proper active levels. You may use discrete gates in addition to the priority encoder, but minimize the number of them. Be sure to name all signals with the proper active levels.

7.25 A purpose of Exercise 7.24 was to demonstrate that it is not always possible to maintain consistency in active-level notation unless you are willing to define alternate logic symbols for building blocks that can be used in different ways. For reference purposes, add pin numbers to the cascadable priority encoder symbol in Figure 7-12, and then define an alternate symbol for the same device with the same pin numbers that provides this consistency in Exercise 7.24.

7.26 Design a combinational circuit with eight active-high request inputs, R0–R7, and eight outputs, A2–A0, AVALID, B2–B0, and BVALID, where the R7 input has the highest priority, the "A" outputs identify the highest priority asserted input, and the "B" outputs identify the second-highest priority. Your design may use discrete gates, decoders, and the 8-input priority encoder of Figure 7-11.

7.27 Repeat Exercise 7.26 using Verilog, writing a behavioral module `Vr2prior` and synthesizing it for your favorite programmable device. *Hint*: Use a `for` loop that takes care of both the first and the second priorities within the same loop, working from the highest priority to the lowest.

7.28 The approach suggested in Exercise 7.27 is easy to code but maybe better results are possible. Write a new module `Vr2priori` that uses nested `if` statements to determine the highest-priority input in the same fashion as Program 7-6, and then uses a second set of nested `if` statements to find the second-highest-priority input. Synthesize the new module and compare its size and delay with the first version. Even if the synthesis results are better, was it worth the work?

7.29 Write a test bench that instantiates the two priority encoders in Exercises 7.27 and 7.28 and verifies that they produce identical outputs for all input combinations, displaying the input combination and outputs if they are different. Insert an error of some kind into one of the modules to verify that your display code works.

7.30 Starting with Program 7-7, write a priority-encoder module `Vr8inprior_dis` where the `for` loop starts with the highest-priority input and searches down, using the Verilog `disable` statement to exit the loop when an asserted input is found. Synthesize and target both Program 7-7 and your module to your favorite programmable device and compare the synthesized results. (*Note:* `disable` is not supported by all Verilog tools.)

7.31 Write the truth table and draw a logic diagram for the logic function performed by the CMOS circuit in Figure X7.31. (The circuit contains transmission gates, which were introduced in Figure 1-16.))

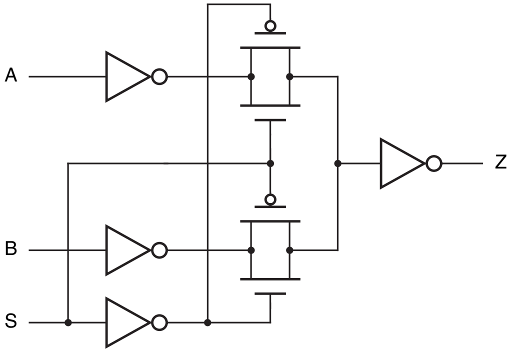


Figure X7.31

7.32 What logic function is performed by the CMOS circuit shown in Figure X7.32?

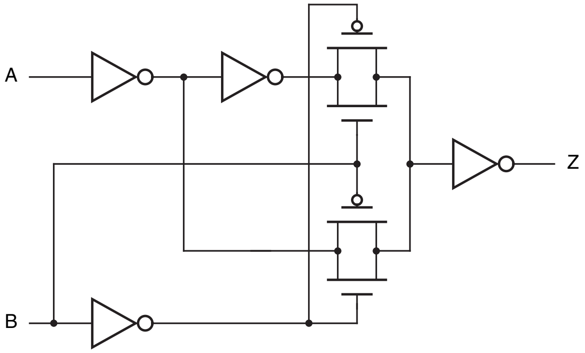


Figure X7.32

7.33 Add a three-state-output control input `OE` to the Verilog multiplexer module in Program 6-16. Your solution should have only one `always` block.

7.34 A digital designer who built the circuit in Figure 7-19 accidentally used NAND gates instead of AND gates in the circuit, and found that the circuit still worked, except for a change in the active level of the ERROR signal. How was this possible?

7.35 Write a Verilog module for a Hamming encoder with 4-bit data inputs `DI[3:0]` and output bits `DO[6:0]`, where `DO[3:0]` equals `DI[3:0]` and `DO[6:4]` corresponds to check bits 421 in the Hamming matrix of Figure 2-13 when `DI[3:0]` corresponds to bits 7653, maintaining all correspondences in left-to-right order.

7.36 Update the Hamming error-correction module of Program 7-14 with one more input bit `DU[8]` and corresponding output bit `DC[8]`, for use with a data bus where eighth bit is even parity for the entire bus, creating a distance-4 code. Also add a new output `UCERR` that indicates an uncorrectable error has occurred.

7.37 A set of parity-check equations for a distance-4 Hamming code with 64 data bits and eight parity-check bits are specified by the eight 72-bit constants below, each representing one row the parity-check matrix:

```
C[1] = 72'h800000000000000007f; C[2] = 72'h400000003fffffff80;
C[3] = 72'h20001fffc0007fff80; C[4] = 72'h100fe03fc07f807f80;
C[5] = 72'h0871e3c3c78787878f; C[6] = 72'h04b66cccd9999999b3;
C[7] = 72'h02dab5556aaaaaaad5; C[8] = 72'hffffffffffffffffff;
```

Assuming that bits are numbered `D[71:0]`, bits `D[71:64]` are the check bits, and `D[63:0]` are the data bits. Based on these parity-check equations, write a Verilog model `Vrhamenc64` for a Hamming encoder with 64-bit data inputs `DI[63:0]` and a 72-bit encoded data output `DO[71:0]`.

7.38 Using the same parity-check equations as in Exercise 7.37, write a Verilog model for a Hamming error-correcting decoder for a 72-bit bus that uses this code, based on Program 7-14 and including a `UCERR` output as in Exercise 7.36.

7.39 Write a test bench that connects the outputs of the module in Exercise 7.37 to the inputs of the one in Exercise 7.38 and ensures that the overall 64-bit output and input match for a random sequence of data inputs. That should be pretty easy. Once that's working, update your test bench to inject random 1-, 2-, and 3-bit errors into the 72-bit connection between modules. Keep track of the number of miscorrected errors of each size. There shouldn't be any miscorrected 1- and 2-bit errors.

7.40 Write a four-step iterative algorithm corresponding to the iterative comparator circuit of Figure 7-24.

7.41 Write a Verilog module `Vr16bitcmpg` for a 16-bit iterative comparator using the structure of Figure 7-24. Use the language's "`generate`" capability. Write a test bench `Vr16bitcmpg_tb` to test your module for random input combinations against Verilog's built-in comparison operation.

7.42 The `VrNbitcmp_tb3` test bench suggested on page 348 works on comparators with up to 128-bit inputs. Rewrite the statements that assign a random value to `P` there to use a `for` loop, so the test bench will work with any width input vectors. Does the modified test bench compile and run successfully in your environment? If not, can you figure out a way to code it so it works with any vector width in your environment, without writing out a long list of assignments as we did just to get the width to 128, as we did on page 348?

7.43 Modify the Verilog module in Program 7-18 to create a new module `Vr8bitscmp` that works with signed input vectors, using the Verilog signed declarations and arithmetic. Write or modify a test bench to ensure that your modifications really work. Then modify and test your modules for 80-bit comparisons, and be sure that your results don't depend on your system's integer width or the result width returned by `$random`.

7.44 Write a parameterized behavioral Verilog module `VrNbitscmp` for a comparator with n-bit input vectors `P` and `Q` and outputs `PGTQ`, `PLTQ`, and `PEQQ`. Your module should work on both signed and unsigned vectors, and have a control input `SGN` that is asserted when the inputs should be interpreted as signed. Adapt the `VrNbitcmp_tb2` test bench to check your module's operation with random input vectors and both values of the `SGN` input. Does it make sense to use Verilog's signed declarations and arithmetic for this exercise?

7.45 A student mistakenly believed that the Verilog “>” and “<“ relational operators only worked on integers, not vectors, and wrote the N-bit comparator module `VrNbitcmp_err` in Program X7.45. The module could be synthesized without errors, and yielded no errors when run in the `VrNbitcmp_tb2` test bench of Program 7-24 with N=8 or 16 or even 31. However, when run with N=32, it displayed errors in about half of the test iterations, over 5,000 errors total. Analyze and explain the reason for this behavior, and point out what specific problems in the module, the test bench, or both causes this behavior. Does the module as written actually produce correct results for N=32?

Program X7.45

```
module VrNbitcmp_err(P, Q, PGTQ, PEQQ, PLTQ);
  parameter N = 8;
  input [N-1:0] P, Q;
  output reg PGTQ, PEQQ, PLTQ;
  integer IP, IQ;

  always @ (P or Q) begin
    IP = P; IQ = Q;
    if (IP == IQ)
      begin PGTQ = 1'b0; PEQQ = 1'b1; PLTQ = 1'b0; end
    else if (IP > IQ)
      begin PGTQ = 1'b1; PEQQ = 1'b0; PLTQ = 1'b0; end
    else
      begin PGTQ = 1'b0; PEQQ = 1'b0; PLTQ = 1'b1; end
  end
endmodule
```

7.46 Design a 24-bit comparator using three 8-bit comparators of the kind shown in Figure 7-25 and a few discrete gates as required. Your circuit should compare two 24-bit unsigned numbers P and Q and produce two output bits that indicate whether P = Q or P > Q.

7.47 A possible definition of a BUT gate (see Exercise 3.37) is “Y1 is 1 if A1 and B1 are 1 *but* either A2 or B2 is 0; Y2 is defined symmetrically.” Write the truth table and find sum-of-products expressions for the BUT-gate outputs. Minimize the expressions using Boolean algebra or Karnaugh maps. Draw the logic diagram for a NAND-NAND circuit for the expressions; assume that only uncomplemented inputs are available. You may use inverters and NAND gates with 2, 3, or 4 inputs.

7.48 If you’ve already studied Chapter 14 or equivalent, find a CMOS gate-level design for the BUT gate defined in Exercise 7.47, using a minimum number of transistors. You may use inverting gates with up to 4 inputs, AOI or OAI gates, transmission gates, or other transistor-level tricks. Write the output expressions (which need not be two-level sums of products), and draw the logic diagram.

butification

7.49 Butify the function $F = \Sigma_{W,X,Y,Z}(5,7,10,11,13,14)$. That is, show how to perform F with a single BUT gate as defined in Exercise 7.47 and a single 2-input OR gate.

7.50 Synthesize the Tic-Tac-Toe design of Section 7.5, targeting your favorite FPGA, and determine how many internal resources it uses. Then try to reduce the resource requirements by specifying a different encoding of the moves in the `TTTdefs.v` file.

7.51 Write a test bench for the Tic-Tac-Toe `TwoInRow` module that compares the outputs of two different versions of the module for all 2^{18} input combinations, as discussed in the box on page 362. Optionally, write code that graphically displays the state of the grid and the `MOVE` outputs from both modules when there is a mismatch. To test your test bench, insert an error; the author made a doozy that was hard to find, swapping `X` and `Y` in one of the UUT instantiations.

7.52 An awkward aspect of the Tic-Tac-Toe modules in Section 7.5 is its explicit, error-prone, and ugly use of a formula to compute an index to reference the appropriate bit of a 9-bit vector corresponding to a grid location `i,j`, every time such a bit value is retrieved. Write a Verilog function `ix(i,j)` that does this cleanly, modify the modules to call the function, and test your modifications. Where is the best place to declare this function?

7.53 The Tic-Tac-Toe module in Section 7.5 playing as Y against an intelligent first player X will get to the grid state shown in Figure X7.53 if X’s first two moves are (3,2) and (2,3). And from there it will lose. Write and run a test bench to prove that this is true. Then modify the `Pick` module to avoid losing in this and similar situations and verify your design using the test bench. Also, synthesize your new top-level module and compare its resource requirements with the original. Are the extra resources justified by the improved play?

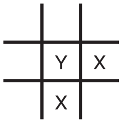
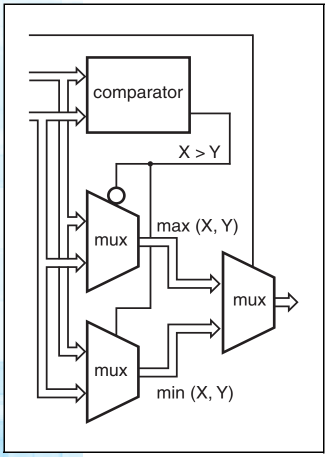


Figure X7.53



chapter 8

Combinational Arithmetic Elements

In this chapter, we introduce combinational logic elements that perform arithmetic functions—adding, shifting, multiplying, and dividing. If you're doing an HDL-based design, it is usually quite reasonable to use the built-in operators when you need one of these functions, and let the synthesis tool do the heavy lifting. This chapter is meant to prepare you for situations where you need to go beyond that.

In both full-custom VLSI and semi-custom ASICs, many structures are available. Here, the designer can specify the exact configuration of gates to perform a function, and even control their physical layout on the chip. In many cases, however, the designer may not have to do this work. The VLSI or ASIC component library may already contain preconfigured, perhaps parameterized modules for common functions like addition and multiplication. In such cases, the designer's main responsibility is to understand the available options and to specify needed arithmetic functions in a way that lets the synthesis tool to recognize that a preconfigured module can be used.

With FPGAs, on the other hand, the basic logic capabilities are set. For combinational logic, there's no way to do custom gate-level structures; there are only programmable interconnections of LUTs. However, the FPGA may also contain specialized internal structures that the synthesizer can use to optimize arithmetic and other common functions. So, the best approach for an FPGA is to let the synthesis tool figure out what to do. You only need to "help it" to do something else if the results are inadequate.

Compared to what a good FPGA design tool can do, the structures in this chapter rarely improve size and performance by more than 10–15%, and may actually worsen them. Still, in the barrel-shifter design of Section 8.2.2, we were actually able to reduce size by 50%, with a slight impact (plus or minus) on speed. In ASIC designs, again "your results may vary," depending on the quality of both the synthesis tools and the available libraries for arithmetic functions.

8.1 Adding and Subtracting

adder

subtractor

Addition is the most commonly performed arithmetic operation in digital systems. An *adder* combines two arithmetic operands using the addition rules described in Chapter 2. As we showed in Section 2.6, the same addition rules (and thus the same adders) are used for both unsigned and two's-complement numbers. An adder can perform subtraction as the addition of the minuend and the complemented (negated) subtrahend, but you can also build *subtractor* circuits that perform subtraction directly. Functional blocks and ASIC modules called ALUs, described in Section 8.1.7, perform addition, subtraction, or any of several other operations according to an operation code supplied to the device.

8.1.1 Half Adders and Full Adders

half adder

The simplest adder, called a *half adder*, adds two 1-bit operands A and B, producing a 2-bit sum. The sum can range from 0 to 2 (base 10), which requires two bits to express. The low-order bit of the sum may be named HS (half sum), and the high-order bit may be named CO (carry-out). We can write the following equations for HS and CO:

$$\begin{aligned} \mathsf{HS} &= \mathsf{A} \oplus \mathsf{B} \\ &= \mathsf{A} \cdot \mathsf{B}' + \mathsf{A}' \cdot \mathsf{B} \\ \mathsf{CO} &= \mathsf{A} \cdot \mathsf{B} \end{aligned}$$

full adder

To add operands with more than one bit, we must provide for carries between bit positions. The building block for this operation is called a *full adder*. Besides the addend-bit inputs A and B, a full adder has a carry-bit input, CIN. The sum of the three inputs can range from 0 to 3, which can still be expressed with just two output bits, S and COUT, having the following equations:

$$\begin{aligned} \mathsf{S} &= \mathsf{A} \oplus \mathsf{B} \oplus \mathsf{CIN} \\ &= \mathsf{A} \cdot \mathsf{B}' \cdot \mathsf{CIN}' + \mathsf{A}' \cdot \mathsf{B} \cdot \mathsf{CIN}' + \mathsf{A}' \cdot \mathsf{B}' \cdot \mathsf{CIN} + \mathsf{A} \cdot \mathsf{B} \cdot \mathsf{CIN} \\ \mathsf{COUT} &= \mathsf{A} \cdot \mathsf{B} + \mathsf{A} \cdot \mathsf{CIN} + \mathsf{B} \cdot \mathsf{CIN} \end{aligned}$$

Here, S is 1 if an odd number of the inputs are 1, and COUT is 1 if two or more of the inputs are 1. These equations represent the same operation that was specified by the binary addition table in Table 2-3 on page 42.

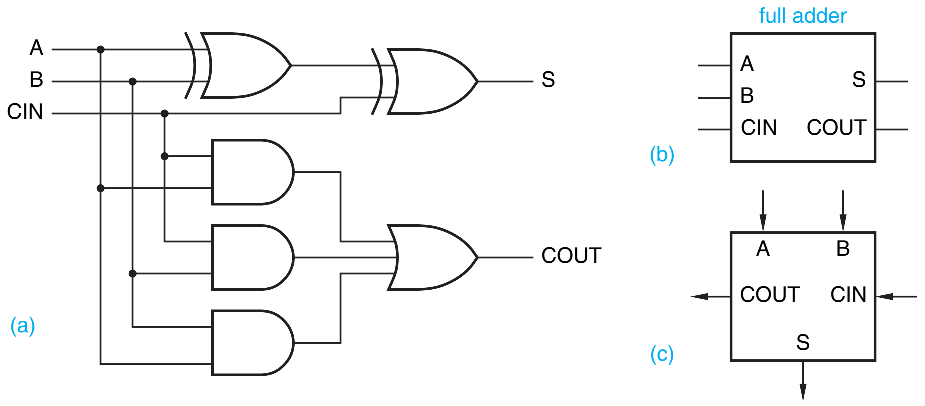


Figure 8-1
Full adder: (a) gate-level logic diagram; (b) logic symbol; (c) alternate logic symbol suitable for cascading.

One possible circuit that realizes the full-adder equations is shown in Figure 8-1(a). The corresponding logic symbol is shown in (b). Sometimes the symbol is drawn as shown in (c), so that cascaded full adders can be easily drawn with carry signals flowing from right to left, as in the next subsection.

8.1.2 Ripple Adders

ripple adder

Two binary words, each with n bits, can be added using a *ripple adder*—a cascade of n full-adder stages, each of which handles one bit. Figure 8-2 shows the circuit for a 4-bit ripple adder. The carry input to the least significant bit (c_0) is normally set to 0, and the carry output of each full adder is connected to the carry input of the next most significant full adder. The ripple adder is a classic example of an iterative circuit as defined in Section 7.4.2.

A ripple adder is slow, since in the worst case, a carry must propagate from the least significant full adder to the most significant one. This occurs if, for example, one addend is 11 … 11 and the other is 00 … 01. Assuming that all of the addend bits are presented simultaneously, the total worst-case delay is

$$t_{\text{ADD}} = t_{\text{ABCout}} + (n-2) \cdot t_{\text{CinCout}} + t_{\text{CinS}}$$

where t_{ABCout} is the delay from A or B to COUT in the least significant stage, t_{CinCout} is the delay from CIN to COUT in each of the n–2 middle stages, and t_{CinS} is the delay from CIN to S in the most significant stage.

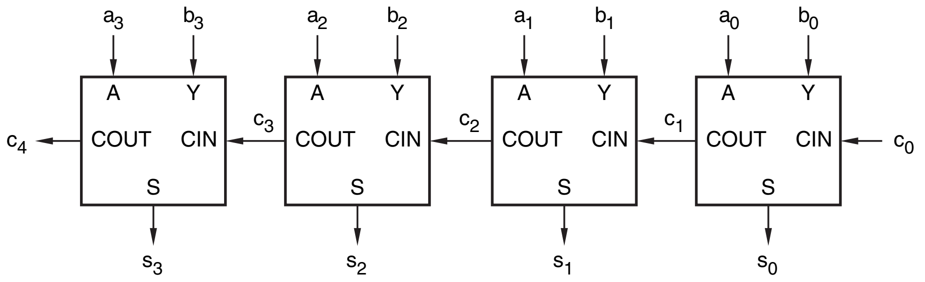


Figure 8-2
A 4-bit ripple adder.

A faster adder can be built by obtaining each sum output s_i with just two levels of logic. This can be accomplished by writing an equation for s_i in terms of x_0–x_i, y_0–y_i, and c_0, a total of $2i + 3$ inputs, "multiplying out" or "adding out" to obtain a sum-of-products or product-of-sums expression, and building the corresponding AND-OR or OR-AND circuit. Unfortunately, beyond s_2 the resulting expressions have too many terms, requiring too many first-level gates and more inputs than typically possible on the second-level gate. For example, even assuming $c_0 = 0$, a two-level AND-OR circuit for s_2 requires 14 4-input ANDs, four 5-input ANDs, and an 18-input OR gate; higher-order sum bits are even worse. Nevertheless, it is possible to build adders with just a few levels of delay using a more reasonable number of gates, as we'll soon see in Section 8.1.4. But first, let's get subtractors out of the way.

8.1.3 Subtractors

full subtractor

A binary subtraction operation analogous to binary addition was also specified in Table 2-3 on page 42. A *full subtractor* handles one bit of the binary subtraction algorithm, having input bits A (minuend), B (subtrahend), and BIN (borrow in), and output bits D (difference) and BOUT (borrow out). We can write logic equations corresponding to the binary subtraction table as follows:

$$\begin{aligned}\mathsf{D} &= \mathsf{A} \oplus \mathsf{B} \oplus \mathsf{BIN}\\ \mathsf{BOUT} &= \mathsf{A'} \cdot \mathsf{B} + \mathsf{A'} \cdot \mathsf{BIN} + \mathsf{B} \cdot \mathsf{BIN}\end{aligned}$$

These equations are very similar to equations for a full adder, which should not be surprising. We showed in Section 2.6 that a two's-complement subtraction operation, $A - B$, can be performed by an addition operation, namely by adding the two's complement of B to A. The two's complement of B is $\overline{B} + 1$, where $\overline{B}$ is the bit-by-bit complement of B. We also asked you to show in Exercise 2.38 that a binary adder can be used to perform an unsigned subtraction operation $A - B$ by performing the operation $A + \overline{B} + 1$. We can now confirm that these statements are true by manipulating the logic equations above:

$$\begin{aligned}\mathsf{D} &= \mathsf{A} \oplus \mathsf{B} \oplus \mathsf{BIN} &&\\ &= \mathsf{A} \oplus \mathsf{B'} \oplus \mathsf{BIN'} && \text{(complementing XOR inputs)}\\ \mathsf{BOUT} &= \mathsf{A'} \cdot \mathsf{B} + \mathsf{A'} \cdot \mathsf{BIN} + \mathsf{B} \cdot \mathsf{BIN} &&\\ \mathsf{BOUT'} &= (\mathsf{A} + \mathsf{B'}) \cdot (\mathsf{A} + \mathsf{BIN'}) \cdot (\mathsf{B'} + \mathsf{BIN'}) && \text{(generalized DeMorgan's theorem)}\\ &= \mathsf{A} \cdot \mathsf{B'} + \mathsf{A} \cdot \mathsf{BIN'} + \mathsf{B'} \cdot \mathsf{BIN'} && \text{(multiplying out and simplifying)}\end{aligned}$$

For the first manipulation, recall that we can complement the two inputs of an XOR gate without changing the function performed.

Comparing these with the equations for a full adder, the above equations tell us that we can build a full subtractor from a full adder by using the complements of the subtrahend and the borrows (B′, BIN′, and BOUT′) or, equivalently,

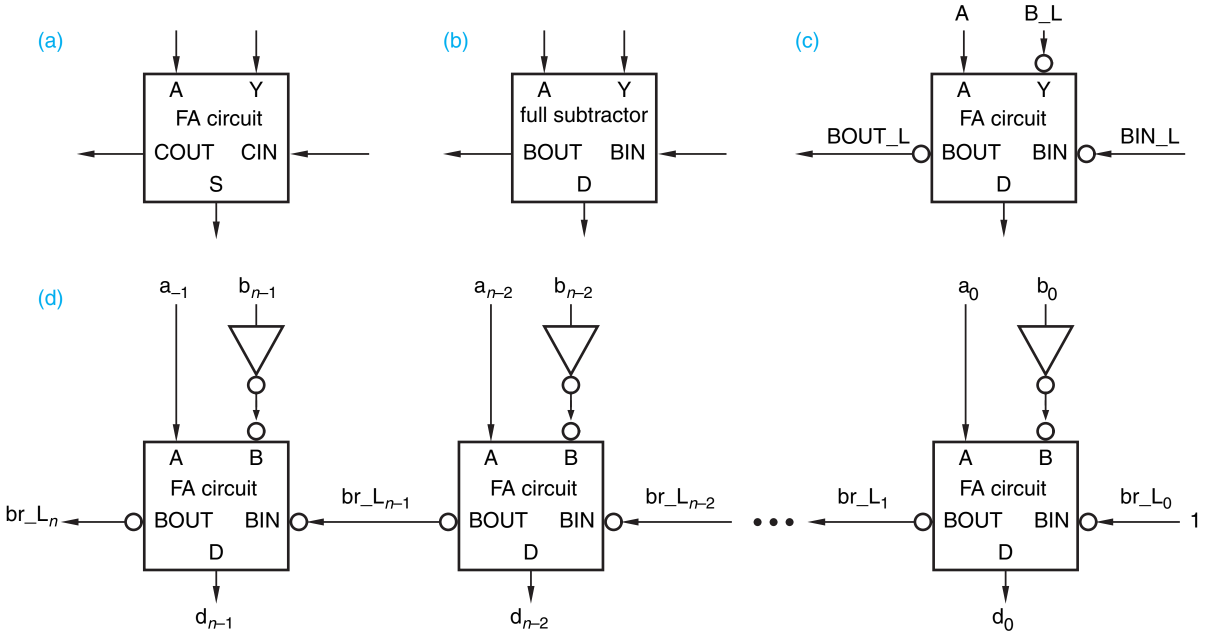


Figure 8-3 Subtractor design using adders: (a) "FA" full-adder circuit; (b) generic full subtractor; (c) interpreting FA circuit as a full subtractor; (d) ripple subtractor.

substituting active-low versions of the corresponding signals as shown in the equations below:

$$\text{BOUT\_L} = \text{A} \cdot \text{B\_L} + \text{A} \cdot \text{BIN\_L} + \text{B\_L} \cdot \text{BIN\_L}$$

$$\text{D} = \text{A} \oplus \text{B\_L} \oplus \text{BIN\_L}$$

That is, the physical circuit that we usually call a full adder, which we've labeled "FA circuit" in Figure 8-3(a), is also a full subtractor if its inputs and outputs are renamed appropriately, with active-low subtrahend, borrow-in, and borrow-out signals as shown in (c).

Thus, to build a ripple subtractor for two *n*-bit active-high operands, we can use *n* FA circuits and inverters, as shown in Figure 8-3(d). Note that for the subtraction operation, the least significant bit's borrow input must be negated (no borrow), which means that active-low physical input pin must be 1 or HIGH for the "no borrow" condition. This behavior is just the opposite of addition's, where the same input pin is an active-high carry-in that is 0 or LOW for the "no carry" condition. The borrow out of the most significant bit is also active-low, again the opposite of the ripple adder's built with the same FA circuits.

Using the math in Chapter 2, we can show that this sort of manipulation works for all adder and subtractor circuits, not just ripple adders and subtractors. That is, any *n*-bit adder circuit can function as a subtractor by complementing the subtrahend and treating the carry-in and carry-out signals as borrows with the opposite active level. The rest of this section discusses addition circuits only, with the understanding that they can easily be made to perform subtraction.

As discussed in Section 7.4.4, a magnitude comparator can be built from a subtractor. Consider the operation $A - B$. If this operation produces a borrow, then $B > A$. One way to build a somewhat smaller magnitude comparator is to start with a subtractor, but eliminate all of the logic that is used only for generating difference bits (typically one XOR gate per bit); only the final borrow is needed for the result. If you wanted to know whether $B \geq A$, you could keep the difference bits and check them for all zeroes for the "equals" case. But depending on the application, a more efficient design might be to swap the subtractor's operands in a second step to determine whether $A > B$; if not, then $B \geq A$.

8.1.4 Carry-Lookahead Adders

carry lookahead

Adders are very important logic elements, as we've said, so a lot of effort has been spent over the years to improve their performance. In this subsection, we'll look at the most well known speedup method, called *carry lookahead*.

The logic equation for sum bit *i* of a binary adder can actually be written quite simply:

$$s_i = a_i \oplus b_i \oplus c_i$$

While all of the addend bits are normally presented to an adder's inputs and are valid more or less simultaneously, the output of the above equation can't be determined until the carry input is valid too. And in a ripple-adder design, it takes a long time for the most significant carry input bit to be valid.

In Figure 8-4, the block labeled "Carry-Lookahead Logic" calculates c_i in a fixed, small number of logic levels if *i* is not too large. Two definitions are the key to carry-lookahead logic:

carry generate

carry propagate

1. For a particular combination of inputs a_i and b_i, adder stage *i* is said to *generate* a carry if it produces a carry-out of 1 ($c_{i+1} = 1$) independent of the inputs on $a_0 - a_{i-1}$,$b_0 - b_{i-1}$, and c_0.
2. For a combination of inputs a_i and b_i, adder stage *i* is said to *propagate* carries if it produces a carry-out of 1 ($c_{i+1} = 1$) in the presence of an input combination on the lower-order bits that causes a carry-in of 1 ($c_i = 1$).

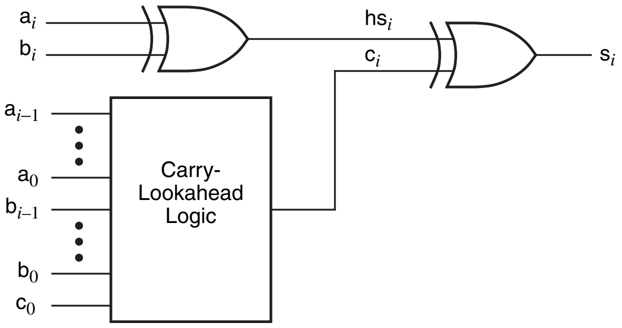


Figure 8-4 Structure of one stage of a carry-lookahead adder.

Corresponding to the first definition, we can write a logic equation for a carry-generate signal g_i for each stage of a carry-lookahead adder:

$$\mathsf{g}_i = \mathsf{a}_i \cdot \mathsf{b}_i$$

That is, a stage unconditionally generates a carry if both of its addend bits are 1.

Corresponding to the second definition, we can write and use either of two different equations for a carry-propagate signal p_i. The first equation propagates a carry if exactly one of the addend bits is 1:

$$\mathsf{p}_i = \mathsf{a}_i \oplus \mathsf{b}_i$$

The second equation recognizes that if both addend bits are 1, it's still OK to "propagate" a carry since we're going to generate one anyway, so we can combine the addend bits with OR instead of XOR:

$$\mathsf{p}_i = \mathsf{a}_i + \mathsf{b}_i$$

This second version of the propagate signal is sometimes called a carry "alive" signal, because one way or another, an incoming carry keeps going.

Depending on the implementation technology, OR and NOR gates may be faster and smaller than XOR and the second version of p_i would be preferred. On the other hand, depending on the overall adder design, we may explicitly need half sum signals (HS_i) elsewhere in the design, and their equation happens to be the same as the first p_i equation above, so we would just use those signals in both places. In either case, the carry output of a stage can now be written in terms of the generate and propagate signals:

$$\mathsf{c}_{i+1} = \mathsf{g}_i + \mathsf{p}_i \cdot \mathsf{c}_i$$

That is, a stage produces a carry if it generates a carry, or if it propagates a carry and the carry input is 1. To eliminate carry ripple, we recursively expand the c_i term for each stage and multiply out to obtain a two-level AND-OR expression. Using this technique, we can obtain the following carry equations for the first four adder stages:

$$\begin{aligned}
\mathsf{c}_1 &= \mathsf{g}_0 + \mathsf{p}_0 \cdot \mathsf{c}_0 \\
\mathsf{c}_2 &= \mathsf{g}_1 + \mathsf{p}_1 \cdot \mathsf{c}_1 \\
&= \mathsf{g}_1 + \mathsf{p}_1 \cdot (\mathsf{g}_0 + \mathsf{p}_0 \cdot \mathsf{c}_0) \\
&= \mathsf{g}_1 + \mathsf{p}_1 \cdot \mathsf{g}_0 + \mathsf{p}_1 \cdot \mathsf{p}_0 \cdot \mathsf{c}_0 \\
\mathsf{c}_3 &= \mathsf{g}_2 + \mathsf{p}_2 \cdot \mathsf{c}_2 \\
&= \mathsf{g}_2 + \mathsf{p}_2 \cdot (\mathsf{g}_1 + \mathsf{p}_1 \cdot \mathsf{g}_0 + \mathsf{p}_1 \cdot \mathsf{p}_0 \cdot \mathsf{c}_0) \\
&= \mathsf{g}_2 + \mathsf{p}_2 \cdot \mathsf{g}_1 + \mathsf{p}_2 \cdot \mathsf{p}_1 \cdot \mathsf{g}_0 + \mathsf{p}_2 \cdot \mathsf{p}_1 \cdot \mathsf{p}_0 \cdot \mathsf{c}_0 \\
\mathsf{c}_4 &= \mathsf{g}_3 + \mathsf{p}_3 \cdot \mathsf{c}_3 \\
&= \mathsf{g}_3 + \mathsf{p}_3 \cdot (\mathsf{g}_2 + \mathsf{p}_2 \cdot \mathsf{g}_1 + \mathsf{p}_2 \cdot \mathsf{p}_1 \cdot \mathsf{g}_0 + \mathsf{p}_2 \cdot \mathsf{p}_1 \cdot \mathsf{p}_0 \cdot \mathsf{c}_0) \\
&= \mathsf{g}_3 + \mathsf{p}_3 \cdot \mathsf{g}_2 + \mathsf{p}_3 \cdot \mathsf{p}_2 \cdot \mathsf{g}_1 + \mathsf{p}_3 \cdot \mathsf{p}_2 \cdot \mathsf{p}_1 \cdot \mathsf{g}_0 + \mathsf{p}_3 \cdot \mathsf{p}_2 \cdot \mathsf{p}_1 \cdot \mathsf{p}_0 \cdot \mathsf{c}_0
\end{aligned}$$

Each equation corresponds to a circuit with as few as three levels of delay—one for the generate or propagate signal, and two for the sum of products shown. A *carry-lookahead adder* uses three-level equations like these in each adder stage for the block labeled “Carry-Lookahead Logic” in Figure 8-4. Each stage’s sum output is produced by combining its carry bit above with two addend bits.

carry-lookahead adder

In any given technology, the carry equations beyond a certain bit position cannot be implemented effectively in just three levels of logic; they require gates with too many inputs. While wider AND and OR functions can be built with two or more levels of logic, a more economical approach is to use carry lookahead only for a small group where the equations can be implemented in three levels, and then use ripple carry between groups. As shown in the next subsection, legacy 4-bit MSI adders used this approach, and they may be used as the basis of efficient gate-level designs in some ASIC libraries.

8.1.5 Group Ripple Adders

74x283

The *74x283* is an MSI 4-bit binary adder that forms its sum and carry outputs with just a few levels of logic, using the carry-lookahead technique. Figure 8-5 is a logic symbol for the 74x283.

The logic diagram for the ’283, shown in Figure 8-6, has a few details worth noting relative to the general carry-lookahead design described in the preceding subsection. First, it uses the “OR” version of the carry-propagate signal, that is, $\mathsf{p}_i = \mathsf{a}_i + \mathsf{b}_i$. Second, it produces active-low versions of the carry-generate (g_i') and carry-propagate (p_i') signals, since inverting gates are generally faster than noninverting ones. Third, it uses an algebraic manipulation of the half-sum equation:

$$\begin{aligned}
\mathsf{hs}_i &= \mathsf{a}_i \oplus \mathsf{b}_i \\
&= \mathsf{a}_i \cdot \mathsf{b}_i' + \mathsf{a}_i' \cdot \mathsf{b}_i \\
&= \mathsf{a}_i \cdot \mathsf{b}_i' + \mathsf{a}_i \cdot \mathsf{a}_i' + \mathsf{a}_i' \cdot \mathsf{b}_i + \mathsf{b}_i \cdot \mathsf{b}_i' \\
&= (\mathsf{a}_i + \mathsf{b}_i) \cdot (\mathsf{a}_i' + \mathsf{b}_i') \\
&= (\mathsf{a}_i + \mathsf{b}_i) \cdot (\mathsf{a}_i \cdot \mathsf{b}_i)' \\
&= \mathsf{p}_i \cdot \mathsf{g}_i'
\end{aligned}$$

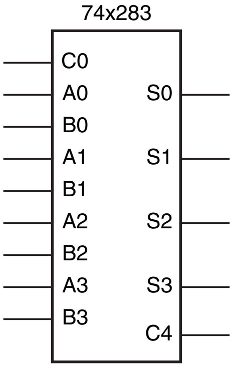
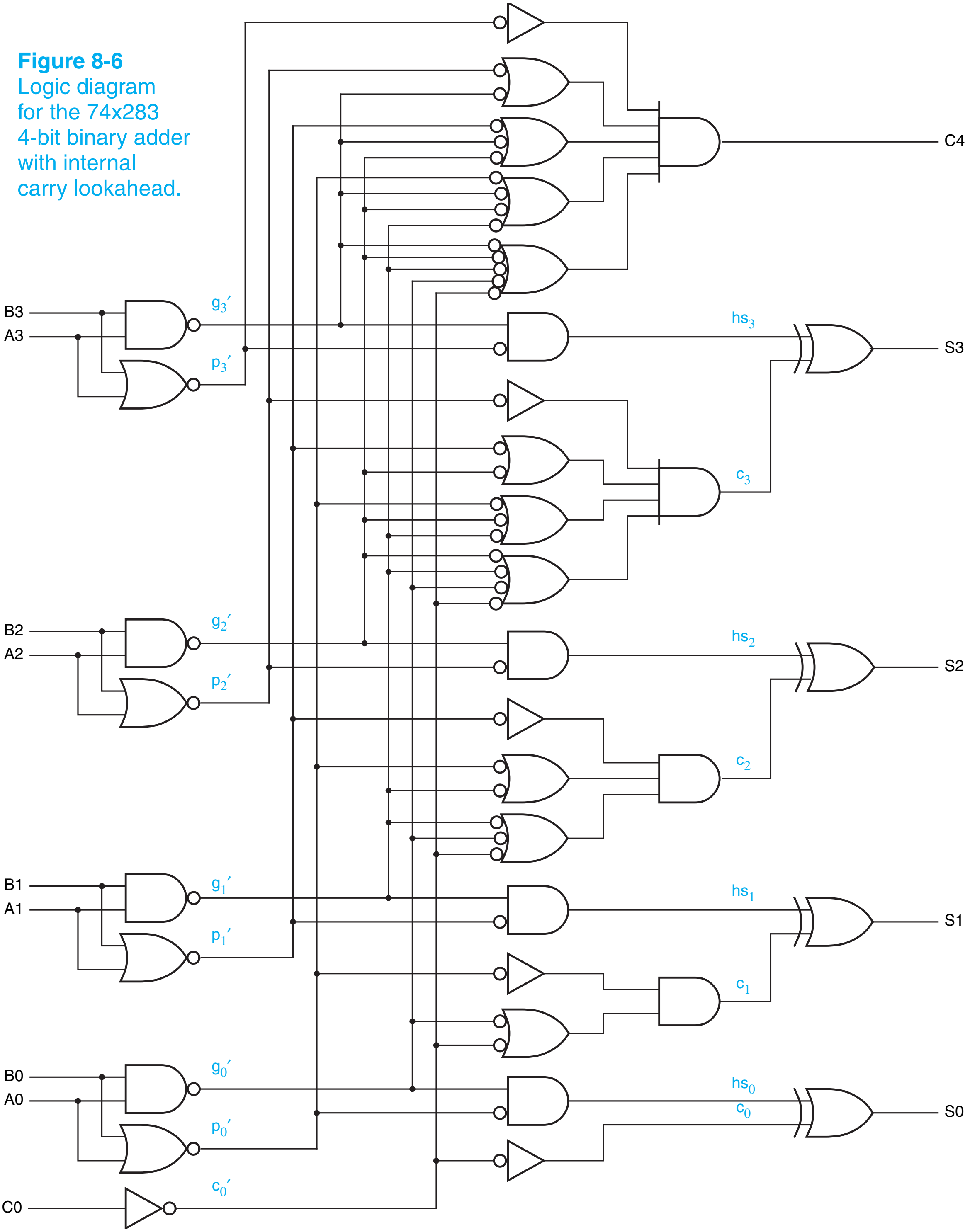


Figure 8-5
Traditional logic symbol for the 74x283 4-bit binary adder with internal carry lookahead.

Figure 8-6 Logic diagram for the 74x283 4-bit binary adder with internal carry lookahead.



CARRY MANIPULATIONS

A little insight and some algebraic manipulation is needed to see how the carry equations in the 74x283 work, compared to the generic carry lookahead equations in the preceding subsection. First, the c_{i+1} equation uses the term $p_i \cdot g_i$ instead of g_i. This has no effect on the output, since p_i is always 1 when g_i is 1. However, it allows the equation to be factored as follows:

$$
\begin{aligned}
c_{i+1} &= p_i \cdot g_i + p_i \cdot \\
c_i \quad &= p_i \cdot (g_i + c_i)
\end{aligned}
$$

This leads to the following carry equations, which are used by the circuit:

$$
\begin{aligned}
c_1 &= p_0 \cdot (g_0 + c_0) \\
c_2 &= p_1 \cdot (g_1 + c_1) \\
&= p_1 \cdot (g_1 + p_0 \cdot (g_0 + c_0)) \\
&= p_1 \cdot (g_1 + p_0) \cdot (g_1 + g_0 + c_0) \\
c_3 &= p_2 \cdot (g_2 + c_2) \\
&= p_2 \cdot (g_2 + p_1 \cdot (g_1 + p_0) \cdot (g_1 + g_0 + c_0)) \\
&= p_2 \cdot (g_2 + p_1) \cdot (g_2 + g_1 + p_0) \cdot (g_2 + g_1 + g_0 + c_0) \\
c_4 &= p_3 \cdot (g_3 + c_3) \\
&= p_3 \cdot (g_3 + p_2 \cdot (g_2 + p_1) \cdot (g_2 + g_1 + p_0) \cdot (g_2 + g_1 + g_0 + c_0)) \\
&= p_3 \cdot (g_3 + p_2) \cdot (g_3 + g_2 + p_1) \cdot (g_3 + g_2 + g_1 + p_0) \cdot (g_3 + g_2 + g_1 + g_0 + c_0)
\end{aligned}
$$

If you've followed the derivation of these equations and can obtain the same ones by reading the '283 logic diagram, then congratulations, you're up to speed on switching algebra! If not, you may want to review Sections 3.1 and 3.2.

Thus, an AND gate with an inverted input can be used instead of an XOR gate to create each half-sum bit. That's generally smaller and faster than an XOR gate.

group-ripple adder

Finally, the '283 creates the carry signals using an INVERT-OR-AND structure (the DeMorgan equivalent of an AND-OR-INVERT), which has about the same delay as a single CMOS inverting gate, as we'll show in Section 14.1.7. Thus, the propagation delay from the C0 input to the C4 output of the '283 is very short, about the same as two inverting gates. As a result, fairly fast *group-ripple adders* with more than four bits can be made simply by cascading the carry outputs and inputs of '283s, as shown in Figure 8-7 for a 16-bit adder. The total propagation delay from C0 to C16 in this circuit is about the same as that of eight inverting gates.

8.1.6 Group-Carry Lookahead

group-carry lookahead

The preceding subsection showed how to ripple carries between individual carry-lookahead adders—that's easy. But we can actually take carry lookahead to the next level, creating *group-carry-lookahead* outputs for each *n*-bit group, and combining these in two levels of logic to provide the carry inputs for all of the groups without rippling carries between them.

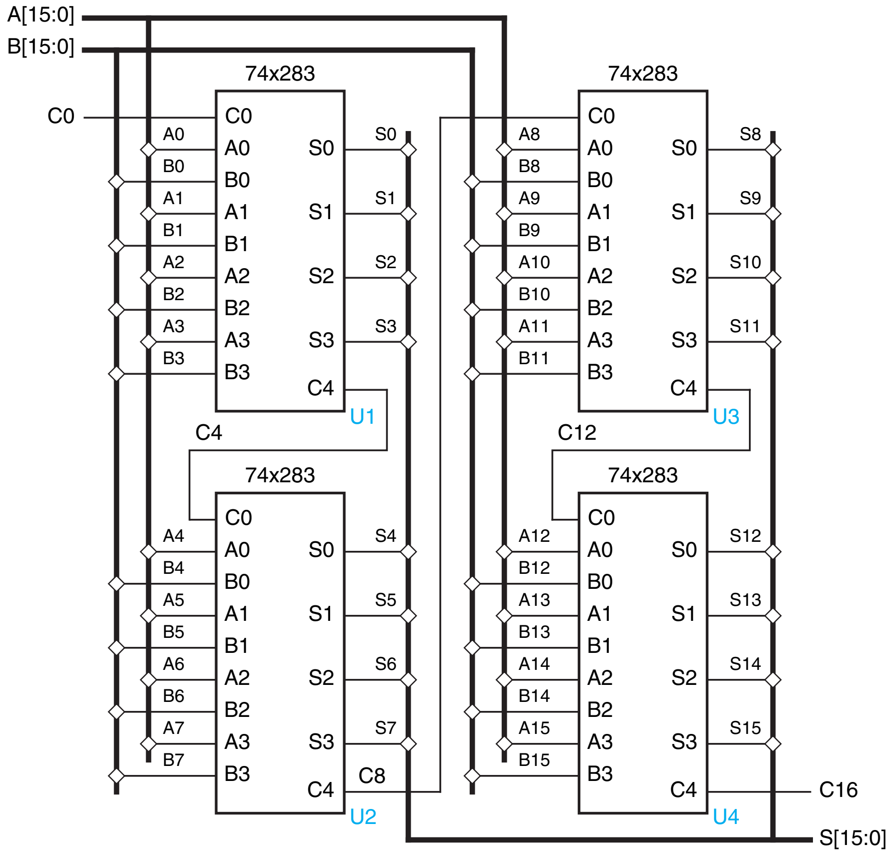


Figure 8-7
A 16-bit group-ripple adder with 4-bit groups.

Figure 8-8 shows the idea for four 4-bit groups. Each group adder has group-carry-lookahead outputs Gg and Pg. The Gg output is asserted if the adder generates a carry—that is, if it will produce a carry-out (C4 = 1) whether or not there is a carry-in (i.e., even if C0 = 0). It can create Gg as a two-level sum of products using the adder's internal generate and propagate signals defined at the bit level in Section 8.1.4:

$$\mathsf{Gg} \;=\; g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0$$

That is, the adder generates a carry if the most significant bit position generates a carry, or if a carry generated by a lower-order bit position is guaranteed to be propagated to and through the most significant bit position. The Pi output is asserted if the adder propagates a carry—that is, if the adder will produce a carry-out if it has a carry-in:

$$\mathsf{Pg} \;=\; p_3 \cdot p_2 \cdot p_1 \cdot p_0$$

lookahead carry circuit

As shown in the figure, these Gg and Pg signals from the groups are combined in a 4-group *lookahead carry circuit* that determines the carry inputs of the

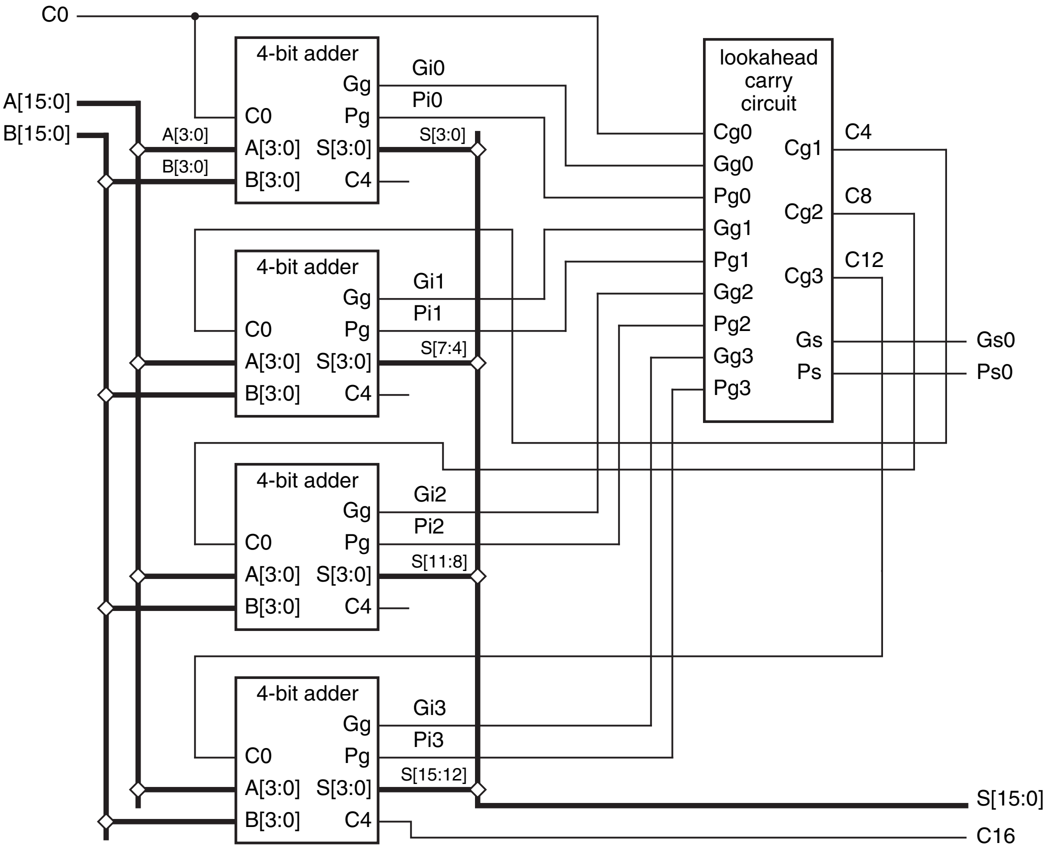


Figure 8-8 A 16-bit group-carry-lookahead adder with 4-bit groups.

three high-order groups, based on the carry into the low-order group and the lookahead outputs. The carry equations used in this circuit can be obtained by "adding out" the basic carry-lookahead equation of Section 8.1.4:

$$c_{i+1} = g_i + p_i \cdot c_i$$

Expanding for the first three values of *i*, we obtain the following equations:

$$\begin{aligned}
\mathrm{Cg1} &= \mathrm{Gg0} + \mathrm{Pg0} \cdot \mathrm{Cg0} \\
\mathrm{Cg2} &= \mathrm{Gg1} + \mathrm{Pg1} \cdot \mathrm{Gg0} + \mathrm{Pg1} \cdot \mathrm{Pg0} \cdot \mathrm{Cg0} \\
\mathrm{Cg3} &= \mathrm{Gg2} + \mathrm{Pg2} \cdot \mathrm{Gg1} + \mathrm{Pg2} \cdot \mathrm{Pg1} \cdot \mathrm{Gg0} + \mathrm{Pg2} \cdot \mathrm{Pg1} \cdot \mathrm{Pg0} \cdot \mathrm{Cg0}
\end{aligned}$$

The carry lookahead scheme can be expanded to make even wider fast adders. Notice that the lookahead carry circuit in Figure 8-8 has its own Gs and Ps outputs; these are asserted if the 16-bit "supergroup" in the figure will respectively generate or propagate a carry. Thus, to obtain a fast 64-bit adder, Figure 8-8 may be replicated for four 16-bit supergroups, with the supergroup lookahead outputs Gs0–3 and Ps0–3 connected to their own second-level lookahead carry circuit to create the carry inputs to the high-order supergroups. Compared to the 16-bit adder, this structure adds only the delay of the second-level lookahead circuit, typically another two gate delays. A Verilog equivalent of this structure will be shown later, in Program 8-9.

Also notice in Figure 8-8 that C16, the carry out of the 16-bit adder, is taken from the high-order 4-bit group. It would also be possible for the lookahead carry circuit to create C16 (call the output Cg4) as a function of its inputs in the same way that it creates Cg1, Cg2, and Cg3. Determining which way is faster is left as Exercise 8.18.

*8.1.7 MSI Arithmetic and Logic Units

arithmetic and logic unit (ALU)

An *arithmetic and logic unit (ALU)* is a combinational circuit that can perform any of a number of different arithmetic and logical operations on a pair of *b*-bit operands. The operation to be performed is specified by a set of function-select inputs. Legacy MSI ALUs have 4-bit operands and three to five function-select inputs, allowing up to 32 different functions to be performed.

74x381
74x382

The logic symbols for the *74x381* and *74x382* legacy MSI ALUs are shown in Figure 8-9(a) and (b), respectively. They each provide the eight different functions as detailed in Table 8-1. Note that the identifiers A, B, and F in the table refer to the 4-bit words A3–A0, B3–B0, and F3–F0; and the symbols ·, +, and ⊕ refer to bit-by-bit logical AND, OR and XOR operations.

74x182

The only difference between the two ALUs is that the ’381 provides active-low group-carry-lookahead outputs, while the ’382 provides ripple carry and overflow outputs. The *74x182*, with the logic symbol shown in Figure 8-9(c), is a lookahead carry circuit designed to be used with the ’381, with active-low lookahead inputs and outputs.

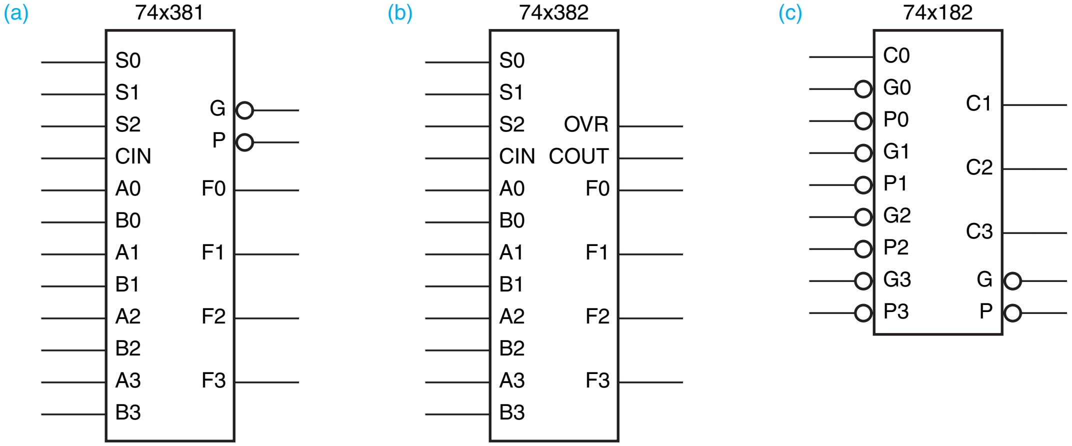


Figure 8-9 Logic symbols for legacy ALU components: (a) 74x381; (b) 74x382; 74x182.

*Throughout this book, optional subsections are marked with an asterisk.

Table 8-1 Functions performed by the 74x381 and 74x382 4-bit ALUs.

Inputs			*Function*
S2	S1	S0	
0	0	0	F = 0000
0	0	1	F = B minus A minus 1 plus CIN
0	1	0	F = A minus B minus 1 plus CIN
0	1	1	F = A plus B plus CIN
1	0	0	F = A ⊕ B
1	0	1	F = A + B
1	1	0	F = A · B
1	1	1	F = 1111

Historically, the two ALU variants were produced instead of a slightly larger part with both sets of outputs because of the pin limitations of the 20-pin IC package. Today, a typical FPGA or ASIC library would provide a component with both sets of outputs, and a synthesis tool would automatically prune away the extra logic for any outputs that weren't being used in a given application.

8.1.8 Adders in Verilog

Verilog has built-in addition (+) and subtraction (−) operators for bit vectors. Just about the simplest possible adder module is shown in Program 8-1, with a parameter N to specify the width of the addends and sum. Since an *n*-bit addition of unsigned numbers can produce an *n*+1-bit sum, the lefthand side of the assignment statement concatenates carry-out bit COUT with the *n*-bit sum output S to receive the *n*+1-bit sum.

In Verilog-2001, the bit vectors can actually be considered to be either unsigned or two's-complement signed numbers, as we've discussed. And as we showed in Section 2.6, the actual addition or subtraction operation is exactly the same for either interpretation of the bit vectors. Since exactly the same logic circuit is synthesized for either interpretation, the Verilog compiler doesn't necessarily need to know how you're interpreting your bit vectors. Only the han-

Program 8-1 A simple Verilog adder module.

```
module VrNbitadder(A, B, CIN, S, COUT);
  parameter N=16;  // Addend and sum width
  input [N-1:0] A, B;
  input CIN;
  output [N-1:0] S;
  output COUT;

  assign {COUT, S} = A + B + CIN;
endmodule
```

Program 8-2 Verilog module with addition of signed and unsigned numbers.

```
module Vradders(A, B, C, D, S, T, OVFL, COUT);
  input [7:0] A, B, C, D;
  output [7:0] S, T;
  output OVFL, COUT;

  // S and OVFL -- signed interpretation
  assign S = A + B;
  assign OVFL = (A[7]==B[7]) && (S[7]!=A[7]);
  // T and COUT -- unsigned interpretation
  assign {COUT, T} = C + D;
endmodule
```

dling of carry, borrow, and overflow conditions differs by interpretation (e.g., in comparisons), and that's done separately from the addition or subtraction itself.

For example, Program 8-2 is a module showing both interpretations. In the first addition, 8-bit addends A and B and sum S are considered to be two's-complement numbers. In two's-complement addition, any carry out of the high-order bit is discarded, so the sum S is declared to have the same bit width as the addends. Since the module does not use the carry out of the high-order bit, the tools will not synthesize any logic circuits for it. (It would have been S[8] if we had declared S as S[8:0], 9 bits wide.) However, we have declared an additional output bit OVFL to indicate an overflow condition, which by definition occurs if the signs of the addends are the same and the sign of the sum is different.

In the second addition, we are considering 8-bit addends C and D to be unsigned numbers. The resulting sum may therefore take up to 9 bits to express, and we could have declared sum T as a 9-bit vector to receive the full sum. Instead, as in our first example we gave T the same width as the addends, and declared a separate 1-bit output COUT to receive the high-order bit of the sum, which is assigned to the 9-bit concatenation of COUT and T.

Addition and subtraction circuits are relatively large, so most compilers will attempt to reuse adder blocks when possible. For example, Program 8-3 is a module with two different additions. Figure 8-10(a) shows a circuit that might be synthesized if the compiler follows the Verilog code literally. However, many

Program 8-3 Verilog module that allows adder sharing.

```
module Vraddersh(SEL, A, B, C, D, S);
  input SEL;
  input [7:0] A, B, C, D;
  output reg [7:0] S;

  always @ (*)
    if (SEL) S = A + B;
    else S = C + D;
endmodule
```

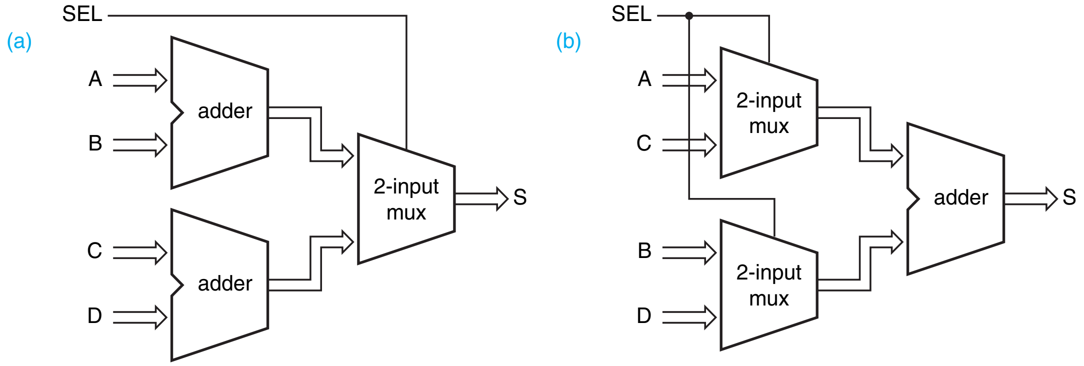


Figure 8-10 Two ways to synthesize a selectable addition: (a) two adders and a selectable sum; (b) one adder with selectable inputs.

compilers are clever enough to use the approach shown in (b). Rather than synthesizing two adders and selecting one's output with a multiplexer, the compiler synthesizes just one adder and selects its inputs using multiplexers. This yields a smaller circuit, since an *n*-bit 2-input multiplexer is smaller than an *n*-bit adder.

The module in Program 8-4 has the same functionality as Program 8-3, this time using a continuous-assignment statement and the conditional operator; a typical compiler should synthesize the same circuit for either module.

A more complicated Verilog module using addition and subtraction is shown in Program 8-5. This module has the same functionality as a 74x381 ALU, including group generate and propagate outputs, except that it has *n*-bit inputs and outputs as specified by a parameter `N` (default 8).

The module's first `for` loop is used to create the internal carry-generate `G[i]` and -propagate `P[i]` signals for each adder stage (`i` ranges from 0 to `N-1`); note here how the bits of `A` and `B` are complemented if they are being subtracted. The second `for` loop combines these signals to create the group carry-generate `G_L` and -propagate `P_L` signals for the *n*-bit group. These signals are specified iteratively (that is, by a `for` loop) in a natural way. At iteration `i`, the variable `GG` indicates whether ALU will generate a carry as of adder stage `i`—that is, if the stage generates a carry (`G[i]`=1) or if it will propagate a previously generated carry (`P[i]`=1 and `GG` was 1 in the `for`-loop's previous iteration). Note that since

Program 8-4 Alternate version of Program 8-3, using a continuous-assignment statement.

```
module Vraddersc(SEL, A, B, C, D, S);
  input SEL;
  input  [7:0] A, B, C, D;
  output  [7:0] S;

  assign S = (SEL) ? A + B : C + D;
endmodule
```

Program 8-5 Verilog module for an *n*-bit 74x381-like ALU.

```
module VrNbitALU(S, A, B, CIN, F, G_L, P_L);
  parameter N = 8;   // Operand widths
  input [2:0] S;
  input [N-1:0] A, B;
  input CIN;
  output reg [N-1:0] F;
  output reg G_L, P_L;
  reg GG, GP;        // Accumulating vars for G and P outputs
  reg [N-1:0] G, P;  // G and P at each bit position
  integer i;

  always @ (*) begin
    for (i = 0; i <= N-1; i = i + 1) begin
      G[i] = (A[i]^(S==3'd1)) & (B[i]^(S==3'd2)); // generate
      P[i] = (A[i]^(S==3'd1)) | (B[i]^(S==3'd2)); // propagate
    end
    GG = G[0]; GP = P[0];  // Accumulate G and P for N-bit group
    for (i = 1; i <= N-1; i = i + 1) begin
      GG = G[i] | (GG & P[i]);
      GP = P[i] & GP;
    end
    G_L = ~GG;  P_L = ~GP; // Set outputs to accumulated values
    case (S)               // Set F outputs as function of select
      3'd0: F = {N{1'b0}};
      3'd1: F = B - A - 1 + CIN;
      3'd2: F = A - B - 1 + CIN;
      3'd3: F = A + B + CIN;
      3'd4: F = A ^ B;
      3'd5: F = A | B;
      3'd6: F = A & B;
      3'd7: F = {N{1'b1}};
      default: F = {N{1'b0}};
    endcase
  end
endmodule
```

GG is a variable in an always block, the assignment to it in each iteration takes effect immediately and is propagated to the next iteration. The output signal G_L is just the complement of GG's value after the last iteration.

In a similar way, at iteration i, the variable GP indicates whether ALU will propagate a carry as of adder stage i, that is, if all the P[i] signals through that stage are 1. The final value of GP is just the AND of the P[i] signals for all values of i, and the output signal P_L is the complement of this value.

A case statement selects one of eight functions for the output function F. Three of these functions involve addition or subtraction, and the code as written relies on Verilog compiler to synthesize the adder and subtractor blocks.

BIG-ADDER PERFORMANCE

In a "tweak" of the Verilog in Program 8-5, we could try to give the compiler some help by writing code to define specify carry bits `C[i]` for each stage `i`, based on the already-available `GG` and `GP` variables and the `CIN` signal. That is, the carry `C[i]` into stage `i` is 1 if `GG` in the previous stage was 1, or if `GP` was 1 and `CIN` is 1. The output function `F` can then be specified for the addition and subtraction cases without Verilog's built-in addition and subtraction operators; for example, `F = A ^ B ^ C` for case 3 (addition), and `F = ~A ^ B ^ C` for case 1 (subtraction). (See Exercise 8.32.)

But does this really help? The answer depends on the target technology and the compiler. For example, when targeted to the same 7-series FPGAs that we used for comparator examples in Section 7.4.6 using Xilinx Vivado tools, the tweaked version used slightly fewer chip resources—21 LUTs vs. 24 for the original. However, it was actually slower (11.12 vs. 10.15 ns total delay), because the compiler did not use the FPGA's CARRY4 elements that optimize adder performance—the poor compiler didn't even know it was synthesizing an adder.

When implementing high-level functions like ALUs in any given target technology, it usually helps the designer to look for a suitable library element that has already been optimized by the technology provider. For example, a 74x381-like ALU that has been hand-designed at the gate level in an ASIC "standard-cell" library will usually be much smaller and faster than an ALU synthesized from behavioral code by a compiler targeting any comparable FPGA or ASIC technology.

And what if a suitable library function is not available? As in this example, it may still be better to let the compiler see the high-level function that you're trying to perform. It may very well know a better way than you to optimize its performance in the target technology.

The test bench in Program 8-6 can be used to check the addition operation of any *n*-bit adder with group carry lookahead outputs. For example, to test the addition function of the *n*-bit ALU of Program 8-5, it instantiates `VrNbitALU` with a constant value applied to the `S` function-select inputs to perform addition. Because *n* should be relatively small (you'd be unlikely to use a carry lookahead group wider than 8 bits), the test bench has nested `for` loops that go through all possible combinations of the addend and carry inputs: 2^{17} of them for the default of 8-bit addends.

The test bench uses a task `checkadd` to check the results at each iteration against the expected sum, generate, and propagate values. The expected sum is calculated using Verilog's built-in addition function; the calculated value should be 1 if the sum of `A` and `B`, without an input carry, requires more than *n* bits; and the propagate value is 1 if at every bit position at least one addend has a 1.

For large adders, for example with 16-bit or wider addends, it's impractical to check all possible input combinations. So, a test bench for wide adders, regardless of their internal design, can use a test bench that generates random

Program 8-6 Test bench for N-bit group-carry-lookahead adder.

```
`timescale 1ns/100ps
module VrNbitgcladd_tb();
  parameter N = 8;   // Operand widths
  reg [N-1:0] A, B;
  reg CIN;
  wire [N-1:0] S;
  wire G_L, P_L;
  integer ai, bi, ci, errors;
  reg xpectG, xpectP;
  reg [N-1:0] xpectS;

  VrNbitALU #(.N(N)) UUT (.S(3'b011),.A(A),.B(B),.CIN(CIN),.F(S),.G_L(G_L),.P_L(P_L));

  task checkadd ();
    begin
      xpectS = A+B+CIN;  xpectG = ((A+B) >= 2**N);
      xpectP = (&(A|B)===1'b1);         // P=1 if there's a 1 in every bit of (A|B)
      if ( (xpectS !== S) || (xpectG !== ~G_L) || (xpectP !== ~P_L) ) begin
        errors = errors + 1;
        $write("ERROR: CIN,A,B = %1b,%8b,%8b, S,G_L,P_L = %8b,%1b,%1b,");
        $display(" should be %8b,%1b,%1b", CIN,A,B, S,G_L,P_L, xpectS,~xpectG,~xpectP);
      end
    end
  endtask

  initial begin
    errors = 0;
    for (ci=0; ci<=1; ci=ci+1)
      for (ai=0; ai<2**N; ai=ai+1)
        for (bi=0; bi<2**N; bi=bi+1) begin
          A = ai; B = bi; CIN = ci;  #10 ;  // Apply test vector and wait
          checkadd;                         // check values
        end
    $display("Errors: %d", errors); $stop(1);
  end
endmodule
```

inputs. However, just as we saw with comparator test benches, the “random” inputs should be carefully selected or adjusted to exercise borderline cases.

Program 8-7 shows such a test bench for an *n*-bit adder with a carry output. Several aspects of this test bench are worth noting:

- Like many of our test benches, it is parameterized for the operand width, and it passes the width parameter to the UUT.
- The UUT is the simple *n*-bit adder module of Program 8-1, but any *n*-bit adder can be tested.

Program 8-7 Adder test bench for wide adders using random test inputs.

```
`timescale 1ns/100ps
module VrNbitadder_tb();
  parameter N = 64;   // Operand widths
  parameter SEED = 1; // Change for a different random sequence
  reg [N-1:0] A, B;
  reg CIN;
  wire [N-1:0] S;
  wire COUT;
  integer i, errors, msb;
  reg xpectCOUT;
  reg [N-1:0] xpectS;

  VrNbitadder #(.N(N)) UUT ( .A(A), .B(B), .CIN(CIN), .S(S), .COUT(COUT) );

  task checkadd;
    begin
      xpectS = A+B+CIN;  xpectCOUT = ( (A+B+CIN) >= {1'b1,{N{1'b0}}} );
      if ( (xpectCOUT!==COUT) || (xpectS!==S) ) begin
        errors = errors + 1;
        $display("ERROR: CIN,A,B = %1b,%8b,%8b, COUT,S = %1b,%8b, should be %1b,%8b",
                 CIN, A, B, COUT, S, xpectCOUT, xpectS );
      end
    end
  endtask

  initial begin
    errors = 0;
    A = $random(SEED);    // Set pattern based on seed parameter
    for (i=0; i<10000; i=i+1) begin
      B = ~A; CIN = 0;  #10 ; checkadd; // Apply test vector and comp., wait, check
      CIN = 1; #10 ; checkadd;          // Check both values of CIN
      msb = 31; A[31:0] = $random;      // Get random number, maybe > 32 bits wide
      while (msb < N-1) begin A = A<<32; A[31:0] = $random; msb = msb+32; end
      CIN = 0; #10 ; checkadd;          // Check again
      CIN = 1; #10 ; checkadd;          // Try both values of CIN
    end
    $display("Errors: %0d", errors); $stop(1);
  end
endmodule
```

- The “`checkadd`” task in our previous group-carry-lookahead adder test bench used the expression “`(A+B) >= 2**N)`” to determine whether the adder would generate a carry. We got away with that for this narrow adder, but it won’t work for wide adders. Notice that the righthand side of this expression is an *integer*, which may be as narrow as 32 bits depending on the Verilog tool environment. The addition on the lefthand side, on the other hand, is performed on vectors, which in Program 8-7 may be as wide as 64K bits according to the Verilog LRM. Certainly adders of at least 64–128 bits are found in many applications. So, for the righthand side, the new test bench constructs an *n*+1-bit vector with a single leading 1 bit, so the simulator carries out the comparison on *n*-bit vectors, not integers.

- Another width-related issue is the random test-input generation. Recall that the result produced by the `$random` Verilog system function is a 32-bit signed integer, which is sign-extended when assigned to a wider vector. That doesn't provide very full test coverage for adders wider than 32 bits. Our test bench uses a `while` loop to build up an arbitrarily wide test vector as needed, 32 bits at a time, using multiple calls to `$random`.
- Depending on the internal implementation of the UUT, some logic may be tested only in a few cases. In particular carry-propagate logic, if present, may be tested only if the `A` and `B` inputs of the propagate group are exactly bit-by-bit complementary. Otherwise, carry propagation may be blocked entirely or carry generation may be the dominating logic. Therefore, at each iteration, the test bench applies a random value and its bit-by-bit complement to the `A` and `B` inputs, with `CIN` values of both 0 and 1, which should yield `COUT` values of 0 and 1. Then it generates an unrelated random number to apply to `A` and checks that addition with both values of `CIN`.

In general, we should trust that the tools do "the right thing" when we specify an adder module using built-in Verilog language constructs. However, a test bench like Program 8-7 is very useful to check our work if we do a custom, structural design of an adder circuit. For example, Programs 8-8 and 8-9 are a set of structured Verilog modules to perform a 16-bit addition using the group lookahead structure of Figure 8-8.

The first module, `VrNbitGCLAadder`, is a parameterized *n*-bit group carry lookahead adder with a design identical to the adder portion of Program 8-5 on page 387. The second module, `Vr4iLACckt`, is a 4-input lookahead carry circuit with functionality similar to the 74x182.

The third module, `Vr16bGCLAadder_s` in Program 8-9, instantiates four copies of `VrNbitGCLAadder` (with `N`=4) and one copy of `Vr4iLACckt` to create a 16-bit group-carry-lookahead adder with its own supergroup lookahead outputs. Four instances of the 16-bit module may be further combined with another instance of `Vr4iLACckt` to make a 64-bit adder, as requested in Exercise 8.25.

Notice that the `Vr16bGCLAadder_s` module creates the four instances of the 4-bit adder and connects their inputs and outputs using a generate block (see the box on page 311). An alternative is to use four separate component instantiations, but there's a trade-off. It might be clearer to write four instantiations, and in fact there would be no avoiding this if the interconnections were individually named signals rather than the bits of vectors indexed by g. With the generate block, figuring out the correct index expressions may be a little more difficult, but once it's done, the model is probably less error-prone.

Program 8-8 Bottom-level modules for hierarchical group-carry-lookahead adder design.

```
module VrNbitGCLAadder(A, B, CIN, S, Gg, Pg);
  parameter N = 4;   // Operand widths
  input [N-1:0] A, B;
  input CIN;
  output reg [N-1:0] S;
  output reg Gg, Pg;
  reg GGa, GPa;         // Accumulating vars for Gg and Pg outputs
  reg [N-1:0] G, P, C;  // G, P, and C at each bit position
  integer i;

  always @ (*) begin
    for (i = 0; i <= N-1; i = i + 1) begin
      G[i] = A[i] & B[i];    // per-bit generate and propagate
      P[i] = A[i] | B[i];
    end
    GGa = G[0]; GPa = P[0];  // Accumulate Gg and Pg for the N-bit group
    C[0] = CIN;              // Carry in to LSB
    for (i = 1; i <= N-1; i = i + 1) begin
      C[i] = GGa | (CIN & GPa);   // Carry in from previous bit
      GGa = G[i] | (GGa & P[i]);
      GPa = P[i] & GPa;
    end
    Gg = GGa;  Pg = GPa; // Set outputs to final accumulated values
    S = A ^ B ^ C;   // Compute sum
  end
endmodule

module Vr4iLACckt(C0, Gg, Pg, C, Gs, Ps);
  input C0;
  input [3:0] Gg, Pg;  // Note, active-high G and P in and out in this version
  output reg [4:1] C;
  output reg Gs, Ps;

  always @ (C0 or Gg or Pg) begin
    C[1] = Gg[0] | (Pg[0] & C0);    // Carry outputs go back to the groups
    C[2] = Gg[1] | (Pg[1] & C[1]);
    C[3] = Gg[2] | (Pg[2] & C[2]);
    C[4] = Gg[3] | (Pg[3] & C[3]);  // and the final carry output
    Gs = Gg[3] | (Pg[3] & Gg[2]) | (Pg[3] & Pg[2] & Gg[1])  // Generate and propagate
               | (Pg[3] & Pg[2] & Pg[1] & Gg[0]);           // for the supergroup
    Ps = &Pg;
  end

endmodule
```

Program 8-9 Top-level module for a 16-bit group-carry-lookahead adder with four 4-bit groups.

```
module Vr16bGCLAadder_s(A, B, CIN, S, Gs, Ps, COUT);
  input [15:0] A, B;
  input CIN;
  output wire [15:0] S;
  output wire Gs, Ps, COUT; // Generate, propagate, carry out for 16-bit adder
  wire [3:0] Gi, Pi;        // Generate, propagate outputs for the 4-bit groups
  wire [4:0] C;             // Carry inputs for the 4-bit groups; 16-bit carry out
  genvar g;

  assign C[0] = CIN;

  generate
    for (g=0; g<=3; g=g+1) begin : a   // Generate the four 4-bit adders
      VrNbitGCLAadder #(.N(4)) U1 ( .A(A[(4*g+3):4*g]),.B(B[(4*g+3):4*g]),.CIN(C[g]),
                        .S(S[(4*g+3):4*g]),.Gg(Gi[g]),.Pg(Pi[g]) );
    end
  endgenerate
  // Now hook up the lookahead carry circuit
  Vr4iLACckt U2 ( .C0(CIN), .Gg(Gi), .Pg(Pi), .C(C[4:1]), .Gs(Gs), .Ps(Ps) );
  // If we need a carry output too, we can get it this way:
  assign COUT = C[4]; // or this way: assign COUT = Gs | (CIN & Ps);
endmodule
```

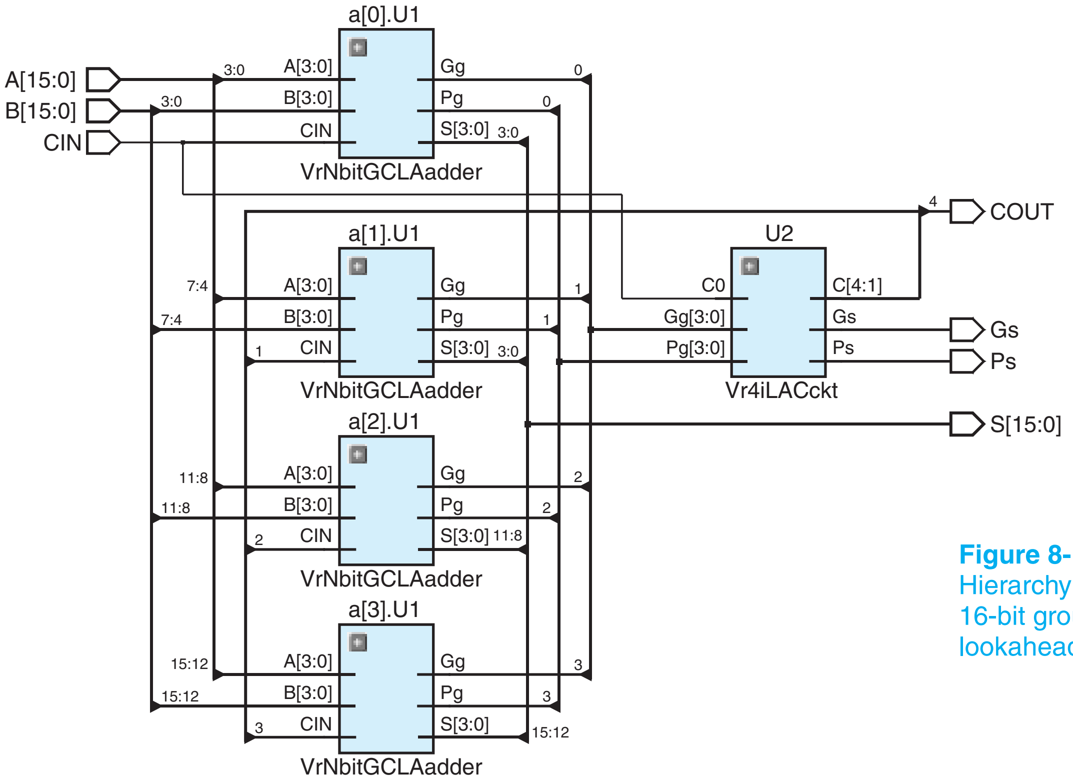


Figure 8-11 Hierarchy of a 16-bit group-carry-lookahead adder.

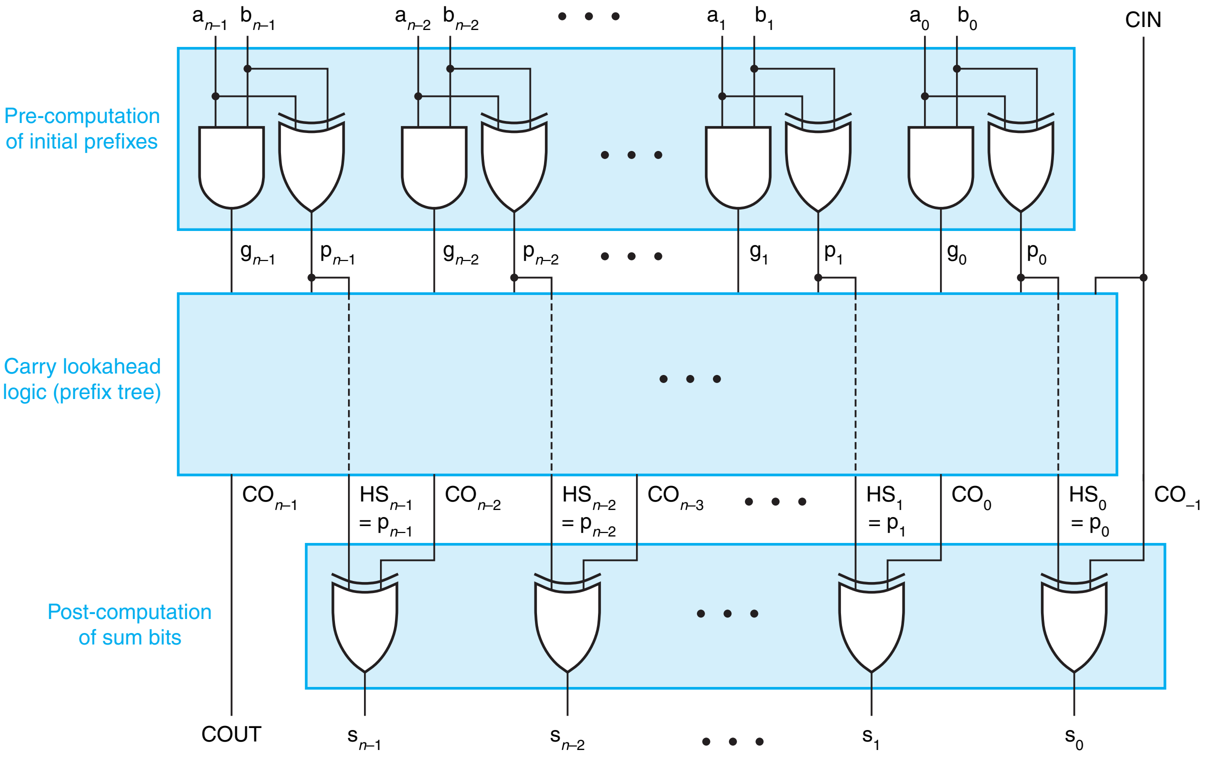
Figure 8-11 on the previous page shows the hierarchical schematic produced by the compiler based on the `Vr16bGCLAadder_s` module definition. Notice how the compiler used the `begin-end` block name and the `genvar` index to name the generated components. Except for using bus notation for the carry and lookahead signals, this schematic pretty much matches the structure in Figure 8-8 on page 382.

*8.1.9 Parallel-Prefix Adders

parallel-prefix adder

The carry-lookahead adder structure in Section 8.1.4 is just one of a class of structures known as *parallel-prefix adders*. The word "prefix" has a formal meaning in mathematical descriptions of how these adders work, but it also just refers to the results of a pre-computation using the adder's inputs, normally the creation of generate and propagate signals for each bit position. As we saw in Section 8.1.4, this can be done in parallel for all bit positions in the addition, hence the name.

The general structure of an n-bit parallel-prefix adder has three blocks as shown in Figure 8-12. The topmost block receives the addends and calculates



a_{n-1} b_{n-1}
a_{n-2} b_{n-2}
• • •
a_1 b_1
a_0 b_0
CIN
Pre-computation of initial prefixes
g_{n-1} p_{n-1} g_{n-2} p_{n-2} • • • g_1 p_1 g_0 p_0
Carry lookahead logic (prefix tree)
CO_{n-1} $HS_{n-1} = p_{n-1}$ CO_{n-2} $HS_{n-2} = p_{n-2}$ CO_{n-3} • • • $HS_1 = p_1$ CO_0 $HS_0 = p_0$ CO_{-1}
Post-computation of sum bits
COUT
s_{n-1} s_{n-2} • • • s_1 s_0

Figure 8-12 General structure of a parallel-prefix adder.

the initial prefixes—generate and propagate signals—in parallel. The next block receives the initial prefixes and the carry CIN into the addition's LSB position, and calculates the individual carry-out signals out from all of the bit positions. This block is where different parallel-prefix adders use different structures and strategies, including the calculation of intermediate prefixes, to optimize delay, circuit area, or some combination of the two. The bottom block receives the carry-out signals and combines these with the half-sum signals from the top block (shown routed "under" the middle block), one XOR gate per sum bit. Note that VLSI and ASIC implementations generally use the "XOR" version of the propagate signals as shown, since these are also the half-sum signals that are used by the bottom block.

Going back to the logic diagram for the 74x283 4-bit carry lookahead Figure 8-6 on page 379, it's easy to draw three rectangles around the logic to see how it corresponds to Figure 8-12. But at the same time, note the drawbacks of the 74x283 structure: the g_i and p_i signals have high fanout, and the widths of the gates that combine them increase as we go to higher-order bits.

Kogge-Stone adder

The first parallel-prefix adder that was targeted to VLSI, the *Kogge-Stone adder* (named after its inventors), avoids the drawbacks of the 74x283's classic lookahead structure. Regardless of the width of the addition, the Kogge-Stone adder does *not* increase the width of the gates or the fanout needed to process lookahead information, which is a great performance benefit for CMOS or almost any other circuit implementation. The Kogge-Stone adder does add one or more "levels" of lookahead logic as the addition width grows, but only slowly. Specifically, each doubling of width adds just two gate delays—an AND-OR or equivalently a NAND-NAND—to the worst-case carry path. We'll see how that's accomplished when we look at the overall design.

The classic lookahead structure in Section 8.1.4 considered lookahead information—generate and propagate—at each bit position. The Kogge-Stone structure starts out the same way, considering each bit position in the first level of lookahead logic. But at each successive level, it doubles the width of a bit group that it considers—2, 4, 8 and more, up to the desired adder width. Before showing how the overall structure does this, we must make a few definitions.

The Kogge-Stone adder represents lookahead information at bit position *i* as a prefix that we'll call GPN_i, a signal pair with two components (GN_i, PN_i), where GN_i is asserted if a carry is generated and PN_i is asserted if a carry is propagated by a group of up to N adjacent bit positions. (We'll elaborate on why "up to" later.) With this definition, $\mathrm{G1}_i$ and $\mathrm{P1}_i$ are the classic generate and propagate signals that we defined in Section 8.1.4 for one (N=1) bit position, that is,

$$\begin{aligned}\mathrm{G1}_i &= a_i \cdot b_i\\ \mathrm{P1}_i &= a_i \oplus b_i\\ \mathrm{GP1}_i &= (\mathrm{G1}_i\,,\ \mathrm{P1}_i)\end{aligned}$$

For larger values of N, the group of bits in the definition of GPN_i starts on the left with bit i and continues to the right, that is, down to bit i–N+1. At each level of the lookahead logic, we double the value of N, so N = 1, 2, 4, 8 and so on.

With this definition, we can design a simple, fixed-size "GPN reduction circuit" (GPR) that combines the GPN_i prefixes from two adjacent bit groups into one GPM_i prefix, where M=2N, and GPM_i gives lookahead information for the double-width group. The GPR circuit implements the following equations:

$$\begin{aligned}\mathsf{GM}_i &= \mathsf{GN}_i + \mathsf{PN}_i \cdot \mathsf{GN}_{i-N}\\ \mathsf{PM}_i &= \mathsf{PN}_i \cdot \mathsf{PN}_{i-N}\\ \mathsf{GPM}_i &= (\mathsf{GM}_i\,, \mathsf{PM}_i)\end{aligned}$$

That is, the double-width group generates a carry if its lefthand half generates a carry, or if its lefthand half propagates a carry and its righthand half generates one. And the double-width group propagates a carry if both halves propagate a carry. The function performed by the GPR circuit is sometimes called the "fundamental carry operation (FCO)."

prefix graph

A logic diagram for the GPR circuit is shown in Figure 8-13(a). We've drawn it with inputs at the top and outputs at the bottom to match the layout of the *prefix graphs* that are used to describe carry generation and propagation using this circuit in prefix adders. This same circuit is used at all levels of the lookahead logic, with a little bit of pruning at the boundaries as we'll see.

So, the basic idea of the Kogge-Stone adder is not all that difficult. At the first level of the carry lookahead logic, we create the traditional generate and propagate signals on all of the 1-bit groups—n $\mathsf{GP1}_i$ generate/propagate pairs for an n-bit adder. At the second level, we combine each $\mathsf{GP1}_i$ pair with the one to its right, creating the $\mathsf{GP2}_i$ pairs. At the third level, we combine each $\mathsf{GP2}_i$ pair

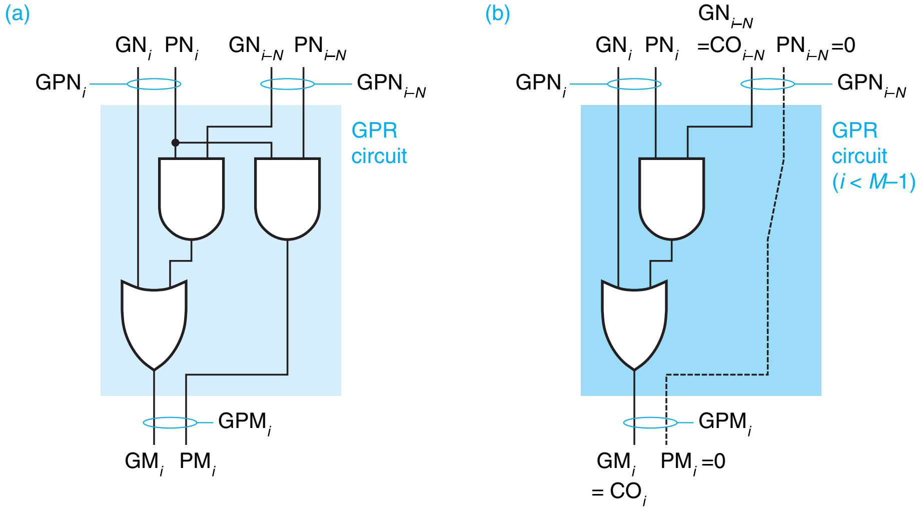


Figure 8-13
GPN reduction circuit: (a) full circuit; (b) as pruned at boundaries.

with the one to its right, creating the GP4$_i$ pairs, then combine these to get GP8$_i$ pairs, and so on. But eventually we can stop.

Consider the case of a 16-bit adder. The GP16$_{15}$ pair tells us whether the 16-bit group consisting of bits 15 down to 0 of the addition will generate or propagate a carry. That's *all* the bits of the addition, so we merely need to combine that with the carry in to the LSB to determine whether there will be a carry out of bit 15 of the addition. And it requires only 5 levels of GPR circuit (for input-group widths 1, 2, 4, 8, 16 bits) to do it. In the general case of an n-bit adder, it takes $\lceil \log_2(n+1) \rceil$ levels of GPR circuit.

This strategy is illustrated in Figure 8-14, the prefix graph (aka *prefix tree*) for a 16-bit Kogge-Stone adder. This is what you would "plug in" to the general parallel-prefix adder structure of Figure 8-12. The colored circles or "nodes" in Figure 8-14 each represent an instance of the GPR circuit, with the GPN prefix for two adjacent N-bit groups coming in at the top and the GPM prefix for the corresponding 2N-bit group coming out at the bottom.

prefix tree

The inputs to the top row of GPR nodes are the conventional 1-bit generate and propagate lookahead signals. This row's outputs give lookahead information for a 2-bit group, the next row for a 4-bit group, and so on. Note that the inputs to the GPR circuits at each level of the tree become available more or less simulta-

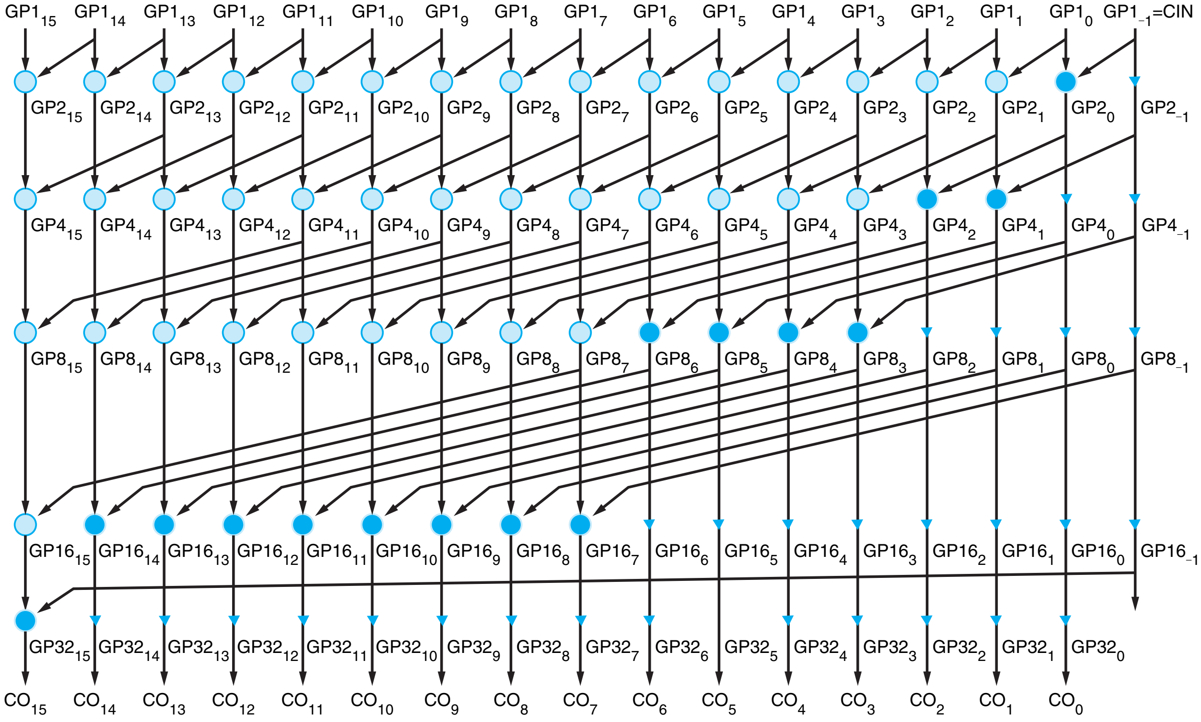


Figure 8-14 Prefix graph of a 16-bit Kogge-Stone adder.

neously (earlier at boundaries), and they are processed in parallel, in keeping with the name “parallel prefix adder.”

To understand the prefix graph, we also need to look at the boundary conditions, starting with the top-right GPR circuit that calculates $GP2_0$. According to the definition of GPM_i, the righthand input of this circuit should be $GP1_{-1}$ or ($G1_{-1}$, $P1_{-1}$). By convention, and also according to common sense, $G1_{-1}$ is the carry that is “generated” in the bit position just to the right of bit-position 0; so, it is in fact the carry in, CIN, to the LSB of the overall addition. On the other hand, $P1_{-1}$ is 0, since there are no bit positions further to the right that can generate a carry to be propagated into bit-position 0. Since $P1_{-1}$ is 0, we can prune the GPR circuit for the boundary case as shown in Figure 8-13(b) on page 396; we represent this pruned circuit in Figure 8-14 using a darker-color circle. Note that the GM_i output of the pruned GPR circuit is in fact the final carry output CO_i for this bit position, while the PM_i output is 0, since successive levels will have no more generated or propagated carries coming in.

The easiest way to understand the remaining boundary conditions is to start with the assumption that a GPR circuit may be needed at every bit position at every level in the prefix graph, and see where that leads. Now consider the second-level GPR circuit that computes $GP4_0$. Besides $GP2_0$ ($G2_0$, $P2_0$), its other inputs should be $GP2_{-2}$ ($G2_{-2}$, $P2_{-2}$). But there are no groups to the right of group –1, so those “signals” if they existed would be always 0. Considering the operation of Figure 8-13(a) with those 0 inputs, the circuit can be pruned to be a buffer or even a wire that simply copies or renames $GP2_0$ to $GP4_0$. We represent this buffer or wire in Figure 8-14 with a small colored triangle. In $GP4_0$ and all of the subsequent GPN_0 prefixes, the propagate component PN_0 is 0 and the generate component GN_i is in fact CO_0, the carry out of bit 0.

In the general case of GPR circuits that calculate GPM_i from GPN_i and GPN_{i-N}, similar reasoning applies:

- If i+1<N, then there is no possibility of a carry being generated N bits to the right, and GPN_i is simply copied to GPM_i; the propagate component is 0 and the generate component is a carry output CO_i.
- Else if i+1<M, then a carry may be generated by or propagated by the group above, but nothing can be propagated from further to the right, so the pruned GPR circuit of Figure 8-13(b) is used.
- Otherwise, the full GPR circuit of Figure 8-13(a) is used (that is, it’s not a boundary case).

Looking at the overall prefix graph in Figure 8-14, as you would expect, the shortest carry delay paths are for the least significant output bits. The longest is for the carry out of the most significant bit, going through five levels of GPR circuits with two gate delays each.

Remember that the lookahead carry logic represented by Figure 8-14 is just the “middle” portion of the overall adder in Figure 8-12 on page 394. The bottom portion is a set of XOR gates that combine the carry out CO_i signals at each bit position at the bottom of the figure with the corresponding half-sums $HS_i = A_i \oplus B_i$. In typical ASIC implementations, the XOR version of the propagate signals $P1_i$ is used in the prefix tree, so they are also used as the HS_i signals as shown in the figure.

In summary and as promised, the Kogge-Stone carry lookahead structure adds just two gate delays—the GPR circuit—for each doubling of the addition width. The number of inputs of each gate in the GPR circuit is fixed and small—just 2. And the fanout of each logic signal in the overall structure is small—for most cases, 2 for GN_i and 3 for PN_i; and for boundary cases, no more than the number of levels in the prefix graph (5 in Figure 8-14) for CIN and less for the other carry signals. (Optionally, also see the box on page 402.)

As we mentioned at the start of this subsection, different prefix graphs can be used to make various trade-offs in parallel-prefix-adder performance. For example, Figure 8-15 shows the prefix graph for a 16-bit Brent-Kung adder. This lookahead structure uses a lot fewer GPR circuits and interconnections than the Kogge-Stone adder, which makes for less circuit area in an ASIC. On the other hand, its worst-case delay path is longer and most of the nodes along that path have more fanout. which makes for slower performance. In between these extremes, there are also hybrid structures that share some of the characteristics of both and achieve an optimal balance between speed and size—depending on the definition of "optimal" for a particular technology and application.

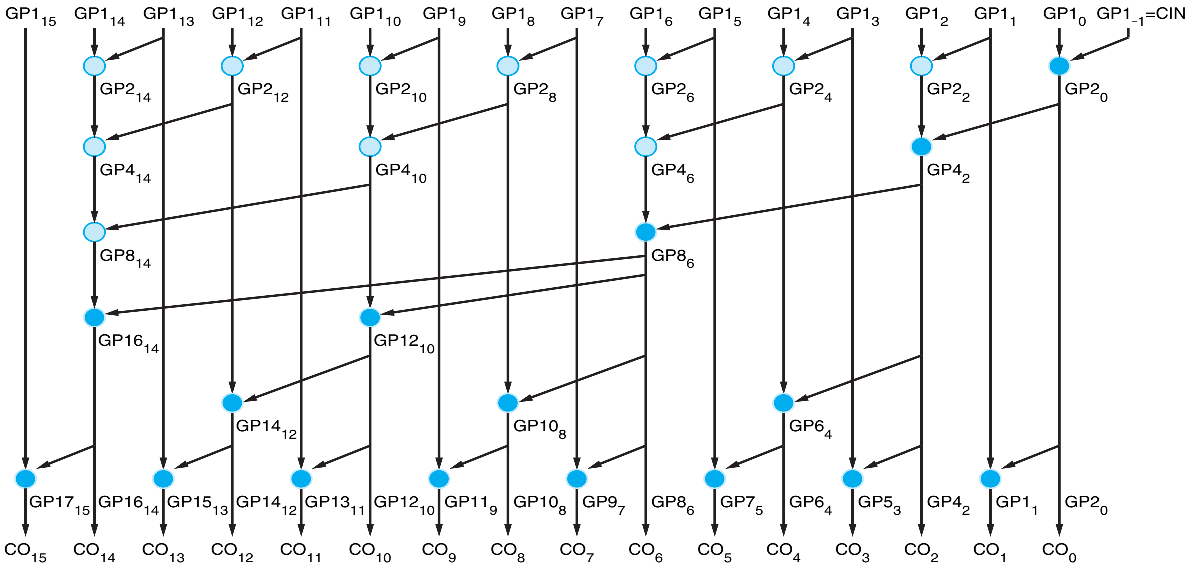


Figure 8-15 Prefix graph of a 16-bit Brent-Kung adder.

*8.1.10 FPGA CARRY4 Element

CARRY4 element

In our discussions of FPGA implementations of both comparators and adders, we've mentioned a *CARRY4 element* which is used to optimize their performance. We now know a few different ways to speed up an adder's carry path, so what is CARRY4? Is it group carry lookahead logic? No. Is it prefix-adder logic? Nope. It's just a 4-bit ripple-carry chain; moreover, in larger adders, carries are also rippled between multiple CARRY4 elements. Despite the long carry chain, CARRY4 enables very fast addition by using a clever signal topology and fast technology both inside and between CARRY4 elements, yielding much lower overall delay than you would get with a normal, programmable interconnection of LUTs in an FPGA.

Figure 8-16 shows the environment and structure of the CARRY4 element in a Xilinx 7-series FPGA. All of the FPGA's 6-input LUTs are arranged in sets

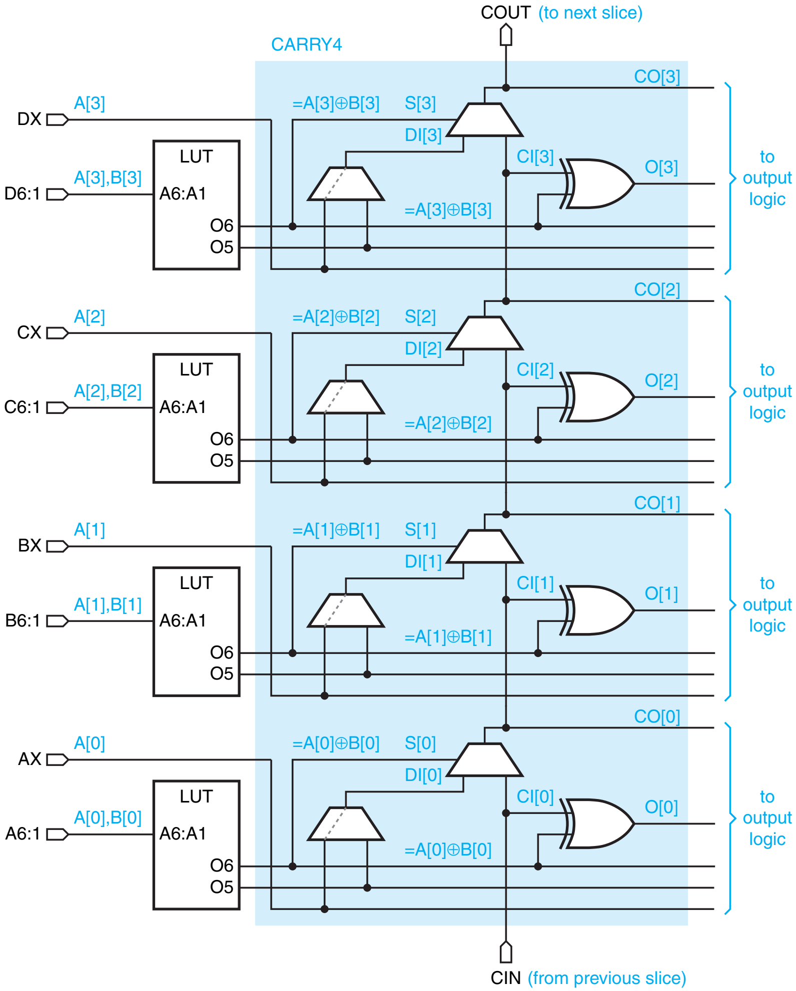


Figure 8-16
CARRY4 logic element.

of four in a *slice* along with flip-flops and other logic (not shown until Section 10.7). The logic in the figure includes:

7-series slice

- Four sections, named "A" through "D" in the Xilinx literature, each having a LUT with six address inputs and two outputs that are functions of five or six inputs, as we explained in Section 6.1.3.
- An auxiliary or "extra" input for each section, AX through DX.
- A programmable multiplexer in each section that selects between the extra input and the LUT O5 output, programmed in this application always to select the extra input.
- A multiplexer in each section that is controlled by the corresponding LUT O6 output. This multiplexer is in the ripple-carry chain, which goes from the bottom to the top of the slice.
- An XOR gate in each section that combines the section's incoming carry with its LUT O6 output.
- The shaded area is the CARRY4 element. Its inputs, shown in color, are carry-select bits S[3:0], data inputs DI[3:0] (programmed in this application to be DX–AX), and the slice carry input CIN. The CARRY4 outputs are XOR outputs O[3:0], carry outputs CO[3:0], and slice carry output COUT.

A 4-bit addition is performed using the slice's CARRY4 element and four LUTs, as shown in color in the figure:

- One addend, say A[3:0], is applied to the slice's extra inputs DX–AX.
- The first addend and a second addend, say B[3:0], is applied to the LUTs' address inputs. Note that each bit of the second addend could be a combinational logic function of up to five independent signals, because six inputs are available on each LUT.
- Each LUT combines its addend bits, A[i] and B[i]. The result, A[i]⊕B[i], is the half sum for bit i and is further XOR'ed with the incoming carry for that position to create a sum bit O[i]. And the result is also used as a propagate signal, as described next.
- The propagate signal A[i]⊕B[i] controls the multiplexer in the carry chain at bit position i. If it's 1, the mux selects the incoming carry from below. Otherwise, it selects A[i]. Why select A[i]? If a carry is to be generated at this bit position, then both A[i] and B[i] must be 1. If none is to be generated, then both must be 0. A[i] always has the correct value to generate a carry at this bit position if one is needed.
- The carry out of each multiplexer is propagated both to the section or slice above and to the output logic of the slice (for possible use elsewhere, but not typically in this application of CARRY4).

So, why is CARRY4 so fast? There are two reasons. First, the connection between the carry output of each bit position to the carry input of the next is fixed using the fastest type of "wire" in the FPGA's chip process (usually metal); this is true for carries both inside and between 4-bit slices, which are arranged on the FPGA chip in a "vertical" stack. Other, general connections between LUTs in the FPGA go through programmable interconnect which is a lot slower.

Second, the two-input multiplexers in the carry chain can be implemented using transmission gates, as described in Section 6.4 on page 282. Once a transmission-gate mux's select input has been set up by the LUT output (which happens in parallel for all the LUTs in this application), the carry delay through the selected mux path is extremely fast. In fact, in a typical 7-series FPGA, the "vertical" delay from CIN through a slice's entire CARRY4 ripple-carry chain and to CIN for the slice above it is about the same as the "horizontal" delay for any other signal from a LUT's address input to and through the output logic, where it needs to go to get to other LUTs; and at that point we haven't even included the substantial delay of programmable interconnect to the next LUT.

So, in a Xilinx 7-series FPGA, even a 64-bit ripple adder using CARRY4 elements is faster than a 64-bit group-carry-lookahead adder built using the structure of Figure 8-8 on page 382 and Exercise 8.25—15.37 vs. 19.58 ns worst-case delay. Moreover, the ripple adder is more compact—64 LUTs plus 17 "free" CARRY4 elements (1 LUT per bit), vs. 141 LUTs (over 2 LUTs per bit). The moral of the story is that it usually behooves a designer to give the tool

PARALLEL-PREFIX ADDERS AGAIN

Sorry, there's still something to say about parallel-prefix adders. In layout, I moved the discussion here so I could keep most of the CARRY4 description together on the two preceding pages.

So, does the adder circuit that results from plugging many instances of Figure 8-13 into Figure 8-14, and then plugging that into Figure 8-12 really work? This would be a great opportunity for you to find out by writing a hierarchical model for the structure and then testing its the results against Verilog's built-in addition function using a test bench like Program 8-7 (see Exercise 8.33).

Actually, this sort of exercise is similar to what a designer would do when designing a custom circuit block for an ASIC or commercial chip that requires high performance in a particular, critical area, such as addition or multiplication. The initial design may be specified behaviorally, but an optimized custom block could be specified structurally and then translated into a gate-level implementation, which may even be laid out by hand to optimize size and speed. Correct functioning of this structural design would be checked using a test bench that compares its outputs against the results obtained using the behavioral specification. In the case of the parallel-prefix adders covered here, a correct behavioral specification is easy to write, because addition is a built-in Verilog function.

a chance to optimize a design for a given technology using its built-in methods, before going off and trying to do better. The tool's approach may be "good enough," or as in the present example, better than the designer's best.

8.2 Shifting and Rotating

Shifting or rotating a word of data bits is a common operation in computer programs. *Shifting* means moving the bits one or more positions to the left or the right, allowing the extra bit(s) to "fall off" at shift-direction end, and providing new bits (usually 0) at the other end. *Rotating* is similar, except the bits that "fall off" the shift-direction end are used to fill the vacated position on the other end. Rotating is sometimes called *circular shifting*.

shifting

rotating

circular shifting

Shifting a word of data bits one position to the left, with a 0 going into the vacated position on the right, is equivalent to multiplying by two if the word represents an unsigned integer. Shifting an unsigned data word to the right with a 0 entering on the left is equivalent to dividing by two, losing any remainder that "falls off" the end (i.e., rounding the result towards 0). This is sometimes called *logical shifting*, whether or not it's intended to perform unsigned arithmetic.

logical shifting

If a data word represents a signed, two's-complement integer, then the operations are a little different and are called *arithmetic shifting*. A left arithmetic shift still has a 0 shifted into the right and multiplies by two, but in a computer the shift may record an "overflow" if the leftmost, sign bit changes during the shift, because the range of representable numbers has been exceeded. A right arithmetic shift sort-of divides by two (see box) and has a copy of the leftmost bit shifted into the left end, preserving the sign of the data word.

arithmetic shifting

8.2.1 Barrel Shifters

A *barrel shifter* is a combinational logic circuit with *n* data inputs, *n* data outputs, and a set of control inputs that specify how to shift the data between input and output. A barrel shifter that is part of a microprocessor CPU can typically specify the direction of shift (left or right), the type of shift (circular,

barrel shifter

ROUNDING OUT THE DISCUSSION

The formal definition of integer division in most computer languages provides that non-integer quotients are rounded to the next integer *towards zero*. However, if you try to use an arithmetic right shift to divide a negative, two's-complement integer by two, then the quotient will be rounded towards minus infinity. The simplest example in which you can see that is with the two's complement representation of –1, which is all 1s. An arithmetic right shift on a word of all 1s yields all 1s, which still represents –1. According to the formal definition above, the correct result is 0 (with a remainder of –1). So, you must be very careful and be aware of perhaps unexpected consequences if you use right shifts to divide by powers of two.

logical, or arithmetic), and the amount of shift (typically 0 to $n-1$ bits, but sometimes 1 to n bits).

In this subsection, we'll look at the design of a simple 16-bit barrel shifter that does left circular shifts only, using a 4-bit control input S[3:0] to specify the amount of shift. For example, if the input word is ABCDEFGHIJKLMNOP (where each letter represents one bit), and the control input is 0101 (5), then the output word is FGHIJKLMNOPABCDE.

From one point of view, this problem is deceptively simple. Each output bit can be obtained from a 16-input multiplexer controlled by the shift-control inputs, where each multiplexer data input is connected to the appropriate data bit. However, when we look at the details of the design, we'll see that there are trade-offs in the speed and size of the multiplexing circuit.

Let us first consider a gate-level design that uses 16 1-bit-wide 16-input multiplexers, each one being a 16-input version of the 8-input multiplexer in Figure 6-30 on page 284. That design can be adapted for the barrel shifter as follows:

- The enable input is not needed.
- Four pairs of inverters may be used to create and buffer the true and complemented versions of the control inputs S[3:0].
- Each AND gate at the first level needs five inputs—one for a data input and four to decode the corresponding control-input value S[3:0].
- The output OR gate needs 16 inputs. Since it is impractical to build gates that wide in one level, it would likely be implemented in the style of Figure 7-22 on page 332.
- The AND-OR circuit can of course be realized as the equivalent NAND-NAND circuit for optimal size and speed in CMOS technology.

The resulting multiplexer has the symbol shown in Figure 8-17(a).

The overall barrel shifter design uses 16 of these 16-input multiplexers connected as shown in Figure 8-17(b). The select inputs of all of the muxes are connected in common to the S[3:0] inputs, which specify the shift amount. The data inputs listed for each mux are connected to D15 to D0 in the left-to-right order listed on the input bus. In the top mux, for example, DIN[0] connects to D15, DIN[15] to D14, and so on, until DIN[1] to D0.

Let us now consider the size and performance of this barrel-shifter design as it might be implemented in an ASIC. Each 16-input multiplexer described above requires eight inverters, 16 5-input NAND gates, and one 16-input NAND gate (probably implemented as four 4-input NAND gates, a 4-input NOR gate, and an inverter). As we'll see in Chapter 14, in CMOS gates with a small number of inputs (up to about four), the number of transistors needed is twice the number of inputs. In the enumeration above, we have 8+80+16+4+1 = 109 inputs or 218 transistors per mux, or 3,488 total for 16 muxes. It would also be a good idea to

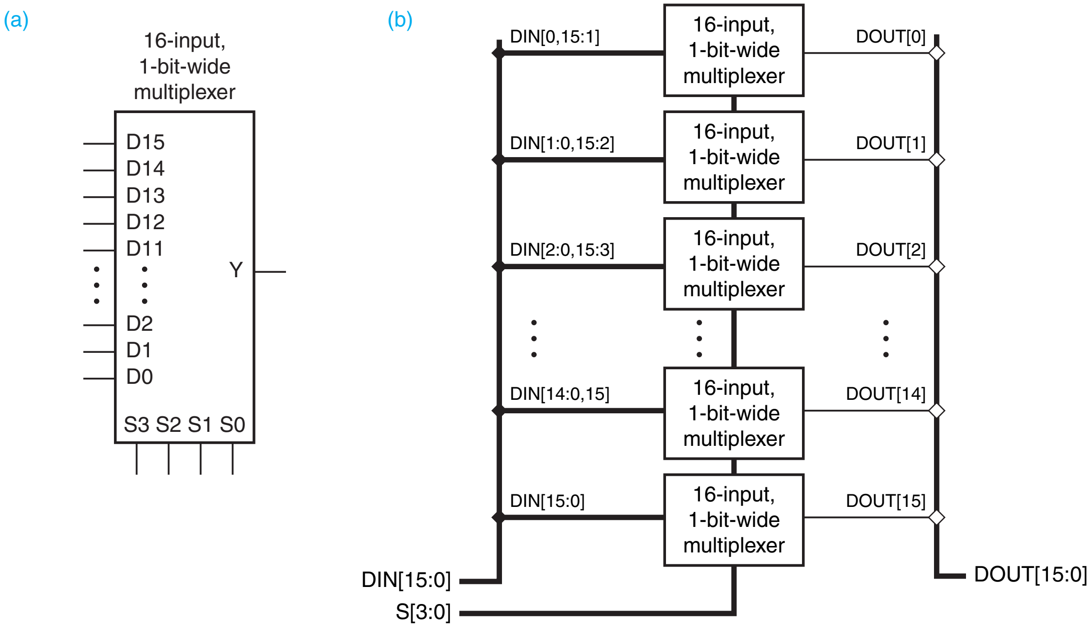


Figure 8-17 A 16-bit barrel-shifter design: (a) 16-input, 1-bit multiplexer component; (b) hookup for left circular shifts.

BUFFER TRANSISTORS

An inverting buffer (i.e., an inverter) in CMOS uses just two transistors, while a non-inverting buffer uses four. We assumed inverting buffers in our transistor count. That's no problem for the S bus, because we can just rename the multiplexer data inputs to match. For the DIN bus, we'll now have complemented data going into the mux, and rather than add an inverter to the output, we can simply delete the final inverter in our implementation of the 16-input NAND gate, which also makes the mux a little faster.

THE SIZE MAY VARY

Counting transistors gives us only a rough approximation of the amount of area that our barrel-shifter circuit will require in an ASIC. Larger transistor sizes may be used in gates with more inputs or heavier loads in order to equalize their delay compared to other gates. And we also have not considered the area needed for wiring, which is relatively large in the present design because each data input connects to all 16 multiplexers.

provide a buffer for each signal in the DIN and S buses, since each drives 16 inputs (one in each mux). That brings the total to about 3,500 transistors.

This design is fairly speedy. Assuming that the DIN and S signals arrive at the same time, the worst-case delay path is from S to DOUT, with a total of five inverting-gate delays (see Drill 8.12).

Now let's consider a design using a cascade of four 16-bit-wide 2-input multiplexers, each of which is designed using gates similar to the 4-bit 2-input multiplexer in Figure 6-32 on page 287, adapted to this problem as follows:

- The enable input is not needed and a simple pair of inverters is used to create and buffer true and complemented versions of the select input S to be used with all 16 bits.
- Each AND gate at the first level still has just two inputs—one for a data input and one for the control input S or its complement.
- The output OR gate also has just two inputs.
- As usual, the AND-OR circuit can be realized as a NAND-NAND circuit.

The resulting multiplexer has the symbol shown in Figure 8-18. Either the A inputs or the B inputs are copied to the corresponding Y outputs, depending on whether S is 0 or 1, respectively.

Figure 8-19 shows the structure of the 16-bit barrel shifter using a cascade of four of these muxes. The first mux rotates DIN by 0 or 1 bits depending on the value of the S[0] input, putting the result on internal bus X. The next mux rotates X by 0 or 2 bits depending on S[1]; the next rotates Y by 0 or 4 bits depending on S[2]; and the last rotates Z by 0 or 8 bits depending on S[3]. The total number of positions rotated will equal the unsigned integer represented by S[3:0].

Let's now compare the size and performance of this barrel-shifter design with the previous one. Each 2-input multiplexer described above requires two inverters and 48 2-input NAND gates, a total of 98 inputs or 196 transistors per mux, or 784 total for four muxes. We probably don't need any buffers for the DIN and S buses, since each bit of DIN drives only two inputs, and S only four. This is a *lot* smaller than the previous design, which used about 3,500 transistors.

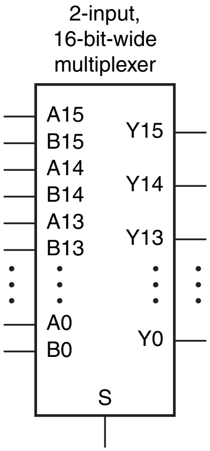


Figure 8-18
Logic symbol for a 2-input, 16-bit-wide multiplexer.

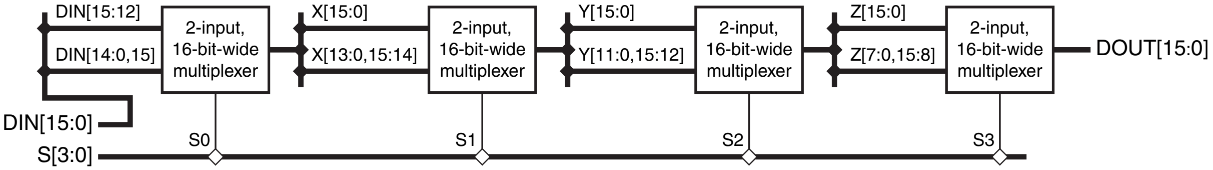

DIN[15:12]
DIN[14:0,15]
DIN[15:0]
S[3:0]
2-input, 16-bit-wide multiplexer
S0
X[15:0]
X[13:0,15:14]
2-input, 16-bit-wide multiplexer
S1
Y[15:0]
Y[11:0,15:12]
2-input, 16-bit-wide multiplexer
S2
Z[15:0]
Z[7:0,15:8]
2-input, 16-bit-wide multiplexer
S3
DOUT[15:0]

Figure 8-19 A 16-bit barrel-shifter design for left circular shifts using 2-input multiplexers.

The trade-off compared of this design compared to the first one is that it appears not to be as speedy. The worst-case path has a total of ten inverting-gate delays—four in the first multiplexer, and two in each of the others. Although each of the gates may be a little faster than in the previous design because they have fewer inputs, that's probably not enough to make up for having twice as many of them.

Another approach to building the 2-input multiplexers is to use CMOS transmission gates, as we discussed in Section 6.4. Each bit of the multiplexer requires just two transmission gates or four transistors, as shown in Figure 8-20. The path to the output through each transmission gate is controlled by S and its complement S_L. As in the gate-based multiplexer, a single pair of inverters can be used to provide S and S_L to all 16 bits. Thus, this approach requires only 68 transistors per multiplexer.

Moreover, once enabled, the delay through a transmission gate is very short, almost as fast as a wire in the most advanced CMOS technologies. When transmission gates are placed in series, as they would be when these multiplexers are used in Figure 8-19, additional buffers may be needed on the data lines to ensure signal speed and integrity. Even allowing for these, an implementation of Figure 8-19 using transmission-gate-based multiplexers is likely to at least as fast as the NAND-gate implementation of Figure 8-17(b) in the same CMOS technology, and as little as one tenth of the size. For that reason, this approach is the one most commonly used in custom VLSI chips and ASICs.

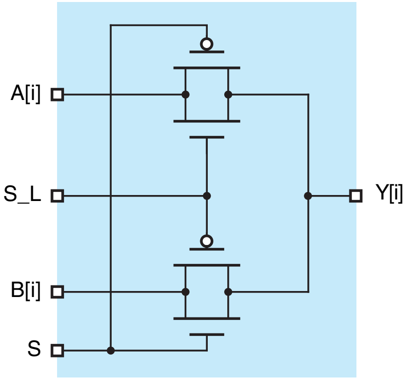


Figure 8-20 One bit of a 2-input multiplexer using transmission gates.

8.2.2 Barrel Shifters in Verilog

In the preceding subsection, we showed how to design a simple barrel shifter that performed only left circular shifts. In this section, we'll show how Verilog can be used to model both the behavior and structure of a more capable barrel shifter for FPGA or ASIC realization.

Our target is a 16-bit barrel shifter that does six different types of shifts, as specified by a 3-bit shift-mode input C[2:0] and detailed in Table 8-2. A 4-bit shift-amount input S[3:0] specifies the amount of shift. For example, if C specifies a right-logical shift and the input word is ABCDEFGHIJKLMNOP and S[3:0] is 0110 (6), then the output word is 000000ABCDEFGHIJ.

As noted at the beginning of Section 8.2, left arithmetic and left logical shifts actually shift their bits identically; in a computer processor or indeed even in Verilog there may be different side effects depending on which version is used. In the present barrel-shifter design, then, both shifts should do the same thing and without two copies of the shifting circuitry, even though two different codes on C[2:0] are used to specify them.

The shift types listed in the table are circular (rotate), logical, and arithmetic, each with directions left and right. Program 8-10 is an excerpt from a behavioral Verilog module for a 16-bit barrel shifter that performs any of the six different combinations of shift type and direction. As shown in the module's declarations, a 4-bit control input `S` gives the shift amount, and a 3-bit control input `C` gives the shift mode (type and direction). A `parameter` statement defines the control coding in accordance with Table 8-2.

The complete `Vrbarrel16` module must define six shifting functions, listed in the "Function" column of Table 8-2, one for each kind of shift on a 16-bit vector. Each function has a 16-bit input `D[15:0]`, a 4-bit input `S[3:0]` to specify the amount of shift, and a 16-bit output.

Program 8-10 shows the details of only the first function (`Vrol`); the rest are similar with only a one-line change (see Exercise 8.37). We define an integer variable `ii` for controlling its loop and a variable `N` to hold the integer equivalent of `S` for the loop-termination comparison. (See the box on page 230 for an explanation of why we don't like to use bit vectors like `S` in a `for` loop's control

Table 8-2
Shift types, codings, and function names for a barrel shifter.

Shift Type	***Name***	***Code***	***Function***	***Note***
Left rotate	`Lrotate`	000	`Vrol`	Wrap-around
Right rotate	`Rrotate`	001	`Vror`	Wrap-around
Left logical	`Llogical`	010	`Vsll`	0 into LSB
Right logical	`Rlogical`	011	`Vslr`	0 into MSB
Left arithmetic	`Larith`	100	`Vsla`	0 into LSB
Right arithmetic	`Rarith`	101	`Vsra`	Replicate MSB

Program 8-10 Verilog behavioral description of a 6-function barrel shifter.

```
module Vrbarrel16 (DIN, S, C, DOUT);
  input [15:0] DIN;            // Data inputs
  input [3:0] S;               // Shift amount, 0-15
  input [2:0] C;               // Mode control
  output [15:0] DOUT;          // Data bus output
  reg [15:0] DOUT;
  parameter Lrotate  = 3'b000, // Define the coding of
            Rrotate  = 3'b001, // the different shift modes
            Llogical = 3'b010,
            Rlogical = 3'b011,
            Larith   = 3'b100,
            Rarith   = 3'b101;

  function [15:0] Vrol;
    input [15:0] D;
    input [3:0] S;
    integer ii, N;
    reg [15:0] TMPD;
    begin
      N = S; TMPD = D;
      for (ii=1; ii<=N; ii=ii+1) TMPD = {TMPD[14:0], TMPD[15]};
      Vrol = TMPD;
    end
  endfunction
  ...

  always @ (DIN or S or C)
    case (C)
      Lrotate :  DOUT = Vrol(DIN,S);
      Rrotate :  DOUT = Vror(DIN,S);
      Llogical : DOUT = Vsll(DIN,S);
      Rlogical : DOUT = Vsrl(DIN,S);
      Larith :   DOUT = Vsla(DIN,S);
      Rarith :   DOUT = Vsra(DIN,S);
      default :  DOUT = DIN;
    endcase
endmodule
```

statement.) The input vector D is assigned to a local variable TMPD, which is shifted N times within the for loop. The body of the for loop is just an assignment statement that concatenates the 15 rightmost bits of the input data (TMPD[14:0]) with the bit that "falls off" the left end in a left shift (TMPD[15]).

Other shift types can be created using similar operations in the five other shift functions. For some of the shift types, it is possible to use Verilog's built-in shift operators (see Exercise 8.38). Note that these six shift functions might not have to be defined in other, nonbehavioral versions of the Vrbarrel16 module, like the structural version that we'll describe later. Also, based on what we've said about left logical and arithmetic shifts, the `Vsll` and `Vsla` functions should be identical.

After the function declarations, the rest of the module is a single `always` block. In it, a `case` statement assigns a result to `DOUT` by calling the appropriate shift function based on the value of the mode-control input `C`.

The Verilog module in Program 8-10 is a nice behavioral description of the barrel shifter, but most synthesis tools cannot synthesize a circuit from it. The problem is that most tools require the range of a `for` loop to be static at the time it is analyzed. The range of the `for` loop in the `Vrol` function is dynamic; it depends on the value of input signal `S` when the circuit is operating.

We can rewrite the offending function and others like it with a one-line change to the `for` loop:

```
for (ii=1; ii<=15; ii=ii+1) if (ii<=N) TMPD = {TMPD[14:0], TMPD[15]};
```

It's not much of a difference, but the modified version (`Vrbarrel16_f`) can be synthesized. When targeted to a Xilinx 7-series FPGA using Vivado tools, the resulting implementation requires 146 LUTs and its worst-case delay paths have three LUTs and a fast, dedicated multiplexer (an F7MUX, see box on page 245). Still, we might be able to do better.

As a first step in optimizing the implementation, let's do left circular shifts using a cascade of four 16-bit, 2-input multiplexers, as shown in Figure 8-19 on page 407. We can express the same kind of behavior and structure using the Verilog module in Program 8-11. Even though this module uses an `always` block and is "behavioral" in style, we can be pretty sure that most synthesis tools will generate a 2-input multiplexer for each "`if`" statement in the module, thereby creating a similar cascade. With further optimization when targeting an FPGA, the tools might do even better. For example, the Vivado tools create an elaborated design that looks very much like Figure 8-19, and after targeting that to LUTs

Program 8-11 Verilog for a 16-bit barrel shifter for left circular shifts only.

```
module Vrrol16 (DIN, S, DOUT);
  input [15:0] DIN;         // Data inputs
  input [3:0] S;            // Shift amount, 0-15
  output [15:0] DOUT;       // Data bus output
  reg [15:0] DOUT, X, Y, Z;

  always @ (DIN or S) begin
    if (S[0] == 1'b1) X = {DIN[14:0], DIN[15]}; else X = DIN;
    if (S[1] == 1'b1) Y = {X[13:0], X[15:14]}; else Y = X;
    if (S[2] == 1'b1) Z = {Y[11:0], Y[15:12]}; else Z = Y;
    if (S[3] == 1'b1) DOUT = {Z[7:0], Z[15:8]}; else DOUT = Z;
  end

endmodule
```

and optimizing, the synthesized result requires 32 LUTs and has just two LUTs in the worst-case delay paths.

Of course, our problem statement requires a barrel shifter that can shift both left and right. Program 8-12 revises the previous module to do circular shifts in either direction. An additional input, DIR, specifies the shift direction: 0 for left, 1 for right. Each rank of shifting is specified by a case statement that picks one of four possibilities based on the values of DIR and the bit of S that controls that rank. The elaborated design looks much like the previous one except with a 3-input mux at each rank, and the synthesized result uses 64 LUTs with a worst-case delay path of two LUTs and an F7MUX.

Program 8-12 Verilog for a 16-bit barrel shifter for left and right circular shifts.

```
module Vrrolr16 (DIN, S, DIR, DOUT);
  input [15:0] DIN;        // Data inputs
  input [3:0] S;           // Shift amount, 0-15
  input DIR;               // Shift direction, 0=>L, 1=>R
  output [15:0] DOUT;      // Data bus output
  reg [15:0] DOUT, X, Y, Z;

  always @ (*) begin
    case ( {S[0], DIR} )
      2'b00, 2'b01 : X = DIN;
      2'b10 :        X = {DIN[14:0], DIN[15]};
      2'b11 :        X = {DIN[0], DIN[15:1]};
      default :      X = 16'bx;
    endcase

    case ( {S[1], DIR} )
      2'b00, 2'b01 : Y = X;
      2'b10 :        Y = {X[13:0], X[15:14]};
      2'b11 :        Y = {X[1:0], X[15:2]};
      default :      Y = 16'bx;
    endcase

    case ( {S[2], DIR} )
      2'b00, 2'b01 : Z = Y;
      2'b10 :        Z = {Y[11:0], Y[15:12]};
      2'b11 :        Z = {Y[3:0], Y[15:4]};
      default :      Z = 16'bx;
    endcase

    case ( {S[3], DIR} )
      2'b00, 2'b01 : DOUT = Z;
      2'b10, 2'b11 : DOUT = {Z[7:0], Z[15:8]};
      default :      DOUT = 16'bx;
    endcase
  end
endmodule
```

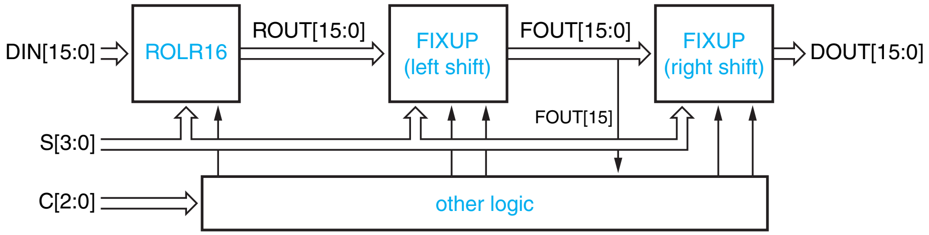


Figure 8-21 Barrel-shifter components.

So, now we have a barrel shifter that will do left or right circular shifts, but we're not done yet—we need to take care of the logical and arithmetic shifts in both directions. Figure 8-21 shows our strategy for completing the design. We start out with the ROLR16 component that we just completed, and we use other logic to control the shift direction as a function of C.

Next we must "fix up" some of the result bits if we are doing a logical or arithmetic shift. For a left logical or arithmetic *n*-bit shift, we must set the rightmost *n*–1 bits to 0. For a right logical or arithmetic *n*-bit shift, we must set the leftmost *n* – 1 bits to 0 or the original leftmost bit value, respectively.

As shown in Figure 8-21, our strategy is to follow the circular shifter (ROLR16) with a fix-up circuit (FIXUP) that plugs in appropriate low-order bits for a left logical or arithmetic shift, and follow that with another fix-up circuit that plugs in high-order bits for a right logical or arithmetic shift.

Program 8-13 is a behavioral Verilog module for the left-shift fix-up circuit. The circuit has 16 bits of data input and output, DIN and DOUT. Its control inputs are the shift amount S, an enable input FEN, and the new value FDAT to be plugged into the fixed-up data bits. For each output bit DOUT[ii], the circuit puts out the fixed-up bit value if ii is less than S and the circuit is enabled; else it puts out the unmodified data input DIN[ii].

For right shifts, fix-ups start from the opposite end of the data word, so it would appear that we need a second version of the fix-up circuit. However, we

Program 8-13 Behavioral Verilog module for left-shift fix-ups.

```
module Vrfixup (DIN, S, FEN, FDAT, DOUT);
  input [15:0] DIN;        // Data inputs
  input [3:0] S;           // Shift amount, 0-15
  input FEN, FDAT;         // Fixup enable and data
  output [15:0] DOUT;      // Data bus output
  reg [15:0] DOUT;
  integer ii;

  always @ (DIN or S or FEN or FDAT)
    for (ii=0; ii<=15; ii=ii+1)
      if ( (ii < S) && (FEN == 1'b1) ) DOUT[ii] = FDAT;
      else DOUT[ii] = DIN[ii];
endmodule
```

can use the original version if we just reverse the order of its input and output bits, as we'll soon see.

Program 8-14 is a structural Verilog module for the full 6-function, 16-bit barrel shifter using the design approach of Figure 8-21. The module inputs, output, and parameters for Vrbarrel16_s are unchanged from the original ones in Program 8-10 on page 409. The module instantiates Vrrolr16 and two instances of Vrfixup, and it has several assignment statements to create needed control signals (the "other logic" in Figure 8-21).

The first assignment asserts DIR_RIGHT if C specifies one of the right shifts. The next four assignments set the proper values for enable inputs

Program 8-14 Verilog structural module for the 6-function barrel shifter.

```
module Vrbarrel16_s (DIN, S, C, DOUT);
  input [15:0] DIN;            // Data inputs
  input [3:0] S;               // Shift amount, 0-15
  input [2:0] C;               // Mode control
  output [15:0] DOUT;          // Data bus output
  wire [15:0] DOUT;
  wire [15:0] ROUT, FOUT, RFIXIN, RFIXOUT;   // Local wires
  wire DIR_RIGHT, FIX_RIGHT, FIX_RIGHT_DAT, FIX_LEFT, FIX_LEFT_DAT;
  genvar ii;
  parameter Lrotate  = 3'b000, // Define the coding of
            Rrotate  = 3'b001, // the different shift modes
            Llogical = 3'b010,
            Rlogical = 3'b011,
            Larith   = 3'b100,
            Rarith   = 3'b101,
            unused1  = 3'b110,
            unused2  = 3'b111;

  assign DIR_RIGHT = ((C==Rrotate) || (C==Rlogical) || (C==Rarith))      ? 1'b1 : 1'b0;
  assign FIX_LEFT  = ((DIR_RIGHT==1'b0) && ((C==Llogical)||(C==Larith))) ? 1'b1 : 1'b0;
  assign FIX_RIGHT = ((DIR_RIGHT==1'b1) && ((C==Rlogical)||(C==Rarith))) ? 1'b1 : 1'b0;
  assign FIX_LEFT_DAT  = (C == Larith) ? DIN[0] : 1'b0;
  assign FIX_RIGHT_DAT = (C == Rarith) ? DIN[15] : 1'b0;
  Vrrolr16 U1 ( .DIN(DIN), .S(S), .DIR(DIR_RIGHT), .DOUT(ROUT) );
  Vrfixup U2 ( .DIN(ROUT), .S(S), .FEN(FIX_LEFT), .FDAT(FIX_LEFT_DAT), .DOUT(FOUT) );
  generate
    for (ii=0; ii<=15; ii=ii+1)
      begin : U3 assign RFIXIN[ii] = FOUT[15-ii]; end
  endgenerate
  Vrfixup U4 (.DIN(RFIXIN),.S(S),.FEN(FIX_RIGHT),.FDAT(FIX_RIGHT_DAT),.DOUT(RFIXOUT));
  generate
    for (ii=0; ii<=15; ii=ii+1)
      begin : U5 assign DOUT[ii] = RFIXOUT[15-ii]; end
  endgenerate
endmodule
```

INFORMATION-HIDING STYLE

Based on the encoding of C, you might like to replace the first assignment statement in Program 8-14 with "DIR_RIGHT <= C[0]", which would be guaranteed to lead to a more efficient realization for that control bit—just a wire! However, this would violate a programming principle of information hiding and lead to possible bugs.

We wrote the shift encodings using parameter definitions in the Vrbarrel16 module declaration. The rest of the module does not depend on the encoding details. Suppose that we nevertheless made the coding change suggested above. If somebody else (or we!) came along later and changed the parameter definitions to a different encoding, the rest of the module would not use the new encodings!

FIX_LEFT and FIX_RIGHT and fix-up data FIX_LEFT_DAT and FIX_RIGHT_DAT for the left and right fix-up circuits, needed for logical and arithmetic shifts.

While all the statements in the module execute concurrently, they are listed in Program 8-14 in the order of the actual dataflow to improve readability. First, Vrrolr16 (U1) is instantiated to perform the basic left or right circular shift as specified. Its outputs are hooked up to the inputs of the first Vrfixup component (U2) to handle fix-ups for left logical and arithmetic shifts. Next is a generate block that reverses the order of the data inputs for the next Vrfixup component (U4), which handles fix-ups for right logical and arithmetic shifts. The final generate block undoes the previous bit-reversing. Note that in synthesis, the two generate blocks don't generate any logic; they merely shuffle wires.

When targeted to a Xilinx 7-series FPGA using Vivado tools, the module in Program 8-14 uses 131 LUTs and its worst-case delay paths go through three LUTs and a fast, dedicated multiplexer. So, it's about 10% smaller than, and about the same speed as, our original behaviorally-specified design. Additional changes can be made to knock at least 40% off that size, while yielding slightly higher or lower worst-case delay paths (see Exercises 8.41 and 8.42).

A test bench for the barrel shifter is shown in Program 8-15. Not shown are the parameter definitions for the 3-bit mode-control values applied to C, and the behaviorally defined functions for performing the six shifts, the first of which (Vrol) we showed in Program 8-10 on page 409. As discussed previously, these functions are typically not synthesizable as written, but they work perfectly well in simulation. And they satisfy our usual test-bench goal of using a different approach than what is done in the unit under test, so conceptual errors as well as "typos" are more likely to be detected.

In the test bench, a Verilog task checksh compares the output of the UUT (DOUT) for each value of C with the shifted value produced by the corresponding function, or the unshifted input (DIN) when one of the two unused values is applied to C. As usual, a case (in)equality operator (!==) is used in checksh rather than simple inequality (!=), so any x and z outputs produced by the UUT will be detected as errors.

Program 8-15 Verilog test bench for the 6-function barrel shifter.

```
`timescale 1 ns / 100 ps
module Vrbarrel16_tb () ;
  reg [15:0] DIN;            // Data inputs
  reg [3:0] S;               // Shift amount, 0-15
  reg [2:0] C;               // Mode control
  wire [15:0] DOUT;          // Data bus output
  integer i, sh, errors;
  parameter SEED = 1;

  task checksh;  // Task to compare UUT output (DOUT) with expected (WANT)
    input [15:0] WANT;
    begin
      if (WANT!==DOUT) begin
        errors = errors + 1;
        $display("Error: C=%3b, S=%4b, DIN=%16b, want %16b, got %16b",
                  C, S, DIN, WANT, DOUT);
      end
    end
  endtask

  Vrbarrel16_s UUT ( .DIN(DIN), .S(S), .C(C), .DOUT(DOUT) );

  initial begin
    errors = 0; DIN = $random(SEED);
    for (i=0; i<2500; i=i+1) begin      // Test 2500 random input data vectors
      DIN = $random;                    // Apply random data input
      for (sh=0; sh<=15; sh=sh+1) begin // Test all possible shift amounts
        S = sh;                         // Apply shift amount
                                        // And test all eight control values
        C = Lrotate; #10  ; checksh(Vrol(DIN,S));
        C = Rrotate; #10  ; checksh(Vror(DIN,S));
        C = Llogical; #10 ; checksh(Vsll(DIN,S));
        C = Rlogical; #10 ; checksh(Vsrl(DIN,S));
        C = Larith; #10   ; checksh(Vsla(DIN,S));
        C = Rarith; #10   ; checksh(Vsra(DIN,S));
        C = unused1; #10  ; checksh(DIN);
        C = unused2; #10  ; checksh(DIN);
      end
    end
    $display("Test done, %0d errors", errors);
    $stop(1);
  end
endmodule
```

Lo and behold, the test bench in Program 8-15 does find errors in the barrel shifter, Program 8-14; can you see the problem? In the test bench, we have assumed that if C has one of the two “unused” values, then the UUT should copy DIN to DOUT unchanged. But in the actual barrel-shifter design we made no such provision, and our original word description of the function is silent about what should happen in these cases. Analyzing Program 8-14 or the test bench output, you can see what actually happens: the input is rotated left by S bits.

So in this example, the test bench has discovered a "bug" in the problem specification itself. Depending on the application, copying the input to the output when an "unused" mode is selected may or may not be needed. If it is, then the design must be modified (see Exercise 8.43). If it isn't, then the test bench should be updated. Either way, the ambiguity should be removed from the spec.

8.3 Multiplying

Multiplication is a common operation. It can be done by a sequential circuit, typically using the shift-and-add algorithm that we described briefly in Section 2.8. But we're not doing sequential circuits yet. It can also be done by combinational circuits as we describe in this section. Verilog's built-in multiplication operator leads to the synthesis of a combinational multiplier.

8.3.1 Combinational Multiplier Structures

Although the shift-and-add algorithm multiplication algorithm emulates the way that oldsters used to do paper-and-pencil multiplication of decimal numbers, there is nothing inherently "sequential" or "time dependent" about multiplication. That is, given two n-bit input words X and Y, it is possible to write a truth table that expresses the $2n$-bit product $P = X \cdot Y$ as a *combinational* function of X and Y. A *combinational multiplier* is a logic circuit with such a truth table.

combinational multiplier

Many approaches to combinational multiplication are based on the paper-and-pencil shift-and-add algorithm. Figure 8-22 illustrates the basic idea for an 8×8 multiplier for two unsigned integers, multiplicand $X = x_7x_6x_5x_4x_3x_2x_1x_0$ and multiplier $Y = y_7y_6y_5y_4y_3y_2y_1y_0$. We call each row a *product component*, a shifted multiplicand that is multiplied by 0 or 1 depending on the corresponding multiplier bit. Each small box represents one product-component bit y_ix_j, the logical AND of multiplier bit y_i and multiplicand bit x_j. The product $P = p_{15}p_{14}\ldots p_2p_1p_0$ has 16 bits and is obtained by adding together all the product components.

product component

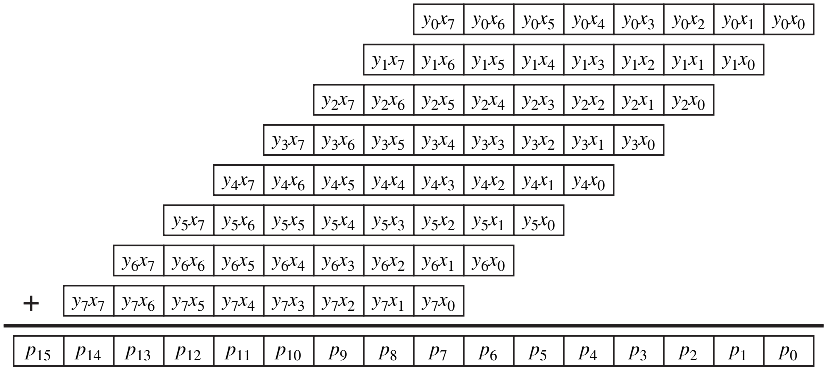

y_0x_7 y_0x_6 y_0x_5 y_0x_4 y_0x_3 y_0x_2 y_0x_1 y_0x_0
y_1x_7 y_1x_6 y_1x_5 y_1x_4 y_1x_3 y_1x_2 y_1x_1 y_1x_0
y_2x_7 y_2x_6 y_2x_5 y_2x_4 y_2x_3 y_2x_2 y_2x_1 y_2x_0
y_3x_7 y_3x_6 y_3x_5 y_3x_4 y_3x_3 y_3x_2 y_3x_1 y_3x_0
y_4x_7 y_4x_6 y_4x_5 y_4x_4 y_4x_3 y_4x_2 y_4x_1 y_4x_0
y_5x_7 y_5x_6 y_5x_5 y_5x_4 y_5x_3 y_5x_2 y_5x_1 y_5x_0
y_6x_7 y_6x_6 y_6x_5 y_6x_4 y_6x_3 y_6x_2 y_6x_1 y_6x_0
+ y_7x_7 y_7x_6 y_7x_5 y_7x_4 y_7x_3 y_7x_2 y_7x_1 y_7x_0
p_{15} p_{14} p_{13} p_{12} p_{11} p_{10} p_9 p_8 p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0

Figure 8-22
Partial products in an 8×8 multiplier.

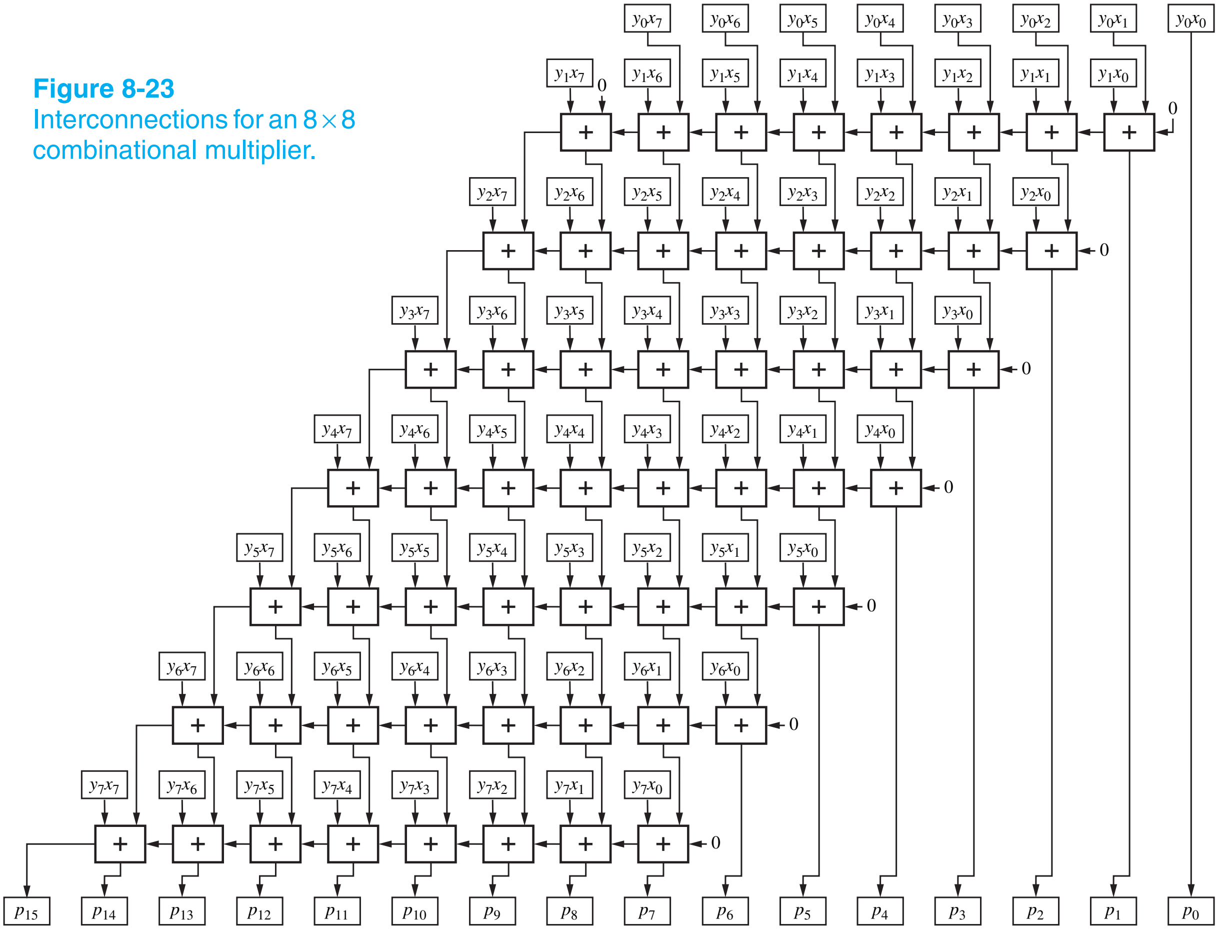
Figure 8-23 shows one way to add up the product components. Here, the product-component bits have been spread out to make space, and each "+" box is a full adder, equivalent to Figure 8-1(c) on page 373. The carries in each row of full adders are connected to make an 8-bit ripple adder. Thus, the first ripple adder combines the first two product components to produce the first partial product, as defined in Section 2.8. Subsequent adders combine each partial product with the next product component.

It is interesting to study the propagation delay of the circuit in Figure 8-23. In the worst case, the inputs to the least significant adder (y_0x_1 and y_1x_0) can affect the MSB of the product (p_{15}). If we assume for simplicity that the delays from any input to any output of a full adder are equal, say t_{pd}, then the worst-case path goes through 20 adders and its delay is $20t_{\text{pd}}$. If the delays are different, then the answer depends on the relative delays (see Exercise 8.44).

Combinational multipliers in general are usually structured as an array of full adders, and are thus often called *array multipliers*. Besides Figure 8-23,

array multiplier

Figure 8-23
Interconnections for an 8 × 8 combinational multiplier.



many other structures can be used, often with better performance and with opportunities to optimize for a particular target technology. Here we'll look at just one other variation, originally inspired by a particular type of sequential-circuit multiplier.

sequential multiplier

Sequential multipliers use a single adder and a register to accumulate the partial products. The partial-product register is initialized to the first product component, and for an $n \times n$-bit multiplication, $n-1$ steps are taken and the adder is used $n-1$ times, once for each of the remaining $n-1$ product components to be added to the partial-product register.

carry-save addition

Some sequential multipliers use a trick called *carry-save addition* to speed up multiplication. The idea is to break the carry chain of the ripple adder to shorten the delay of each addition. This is done by applying the carry output from bit i during step j to the carry input for bit $i+1$ during the *next* step, $j+1$. After the last product component is added, one more step is needed in which the carries are hooked up in the usual way and allowed to ripple from the least to the most significant bit.

The combinational equivalent of an 8×8 multiplier using carry-save addition is shown in Figure 8-24. Notice that the carry out of each full adder in the first seven rows is connected to an input of an adder *below* it. Carries in the eighth row of full adders are connected to create a conventional ripple adder. Although this adder uses exactly the same amount of logic as the previous one (64 2-input AND gates and 56 full adders), its propagation delay is substantially shorter. Its worst-case delay path goes through only 14 full adders. This design is called a *Braun multiplier* and has many variations. For example, its delay can be further improved by using a carry-lookahead or parallel-prefix adder for the last row.

Braun multiplier

The regular structure of array multipliers makes them ideal for VLSI and ASIC realization. The importance of fast multiplication in microprocessors, digital video, and many other applications has led to much study and development of structures and circuits for array multipliers (see the References).

*8.3.2 Multiplication in Verilog

Verilog has a built-in multiplication operator, "*", that operates on two bit-vectors that are interpreted as unsigned numbers. The width of the resulting product is the sum of the widths of the two input vectors. Thus, it is very easy to specify an unsigned multiplier in Verilog, as shown in Program 8-16 for two 8-bit inputs, producing a 16-bit product. However, multipliers should not be specified casually, since this simple code, when targeted for example to a Xilinx 7-series FPGA, creates a combinational circuit that uses over 70 LUTs and has about 10 levels of logic (LUTs and CARRY4 elements) in its worst-case input-to-output path.

If an application requires a multiplier, then of course we should use one. But if we specify the design in more detail, we might get smaller size or better

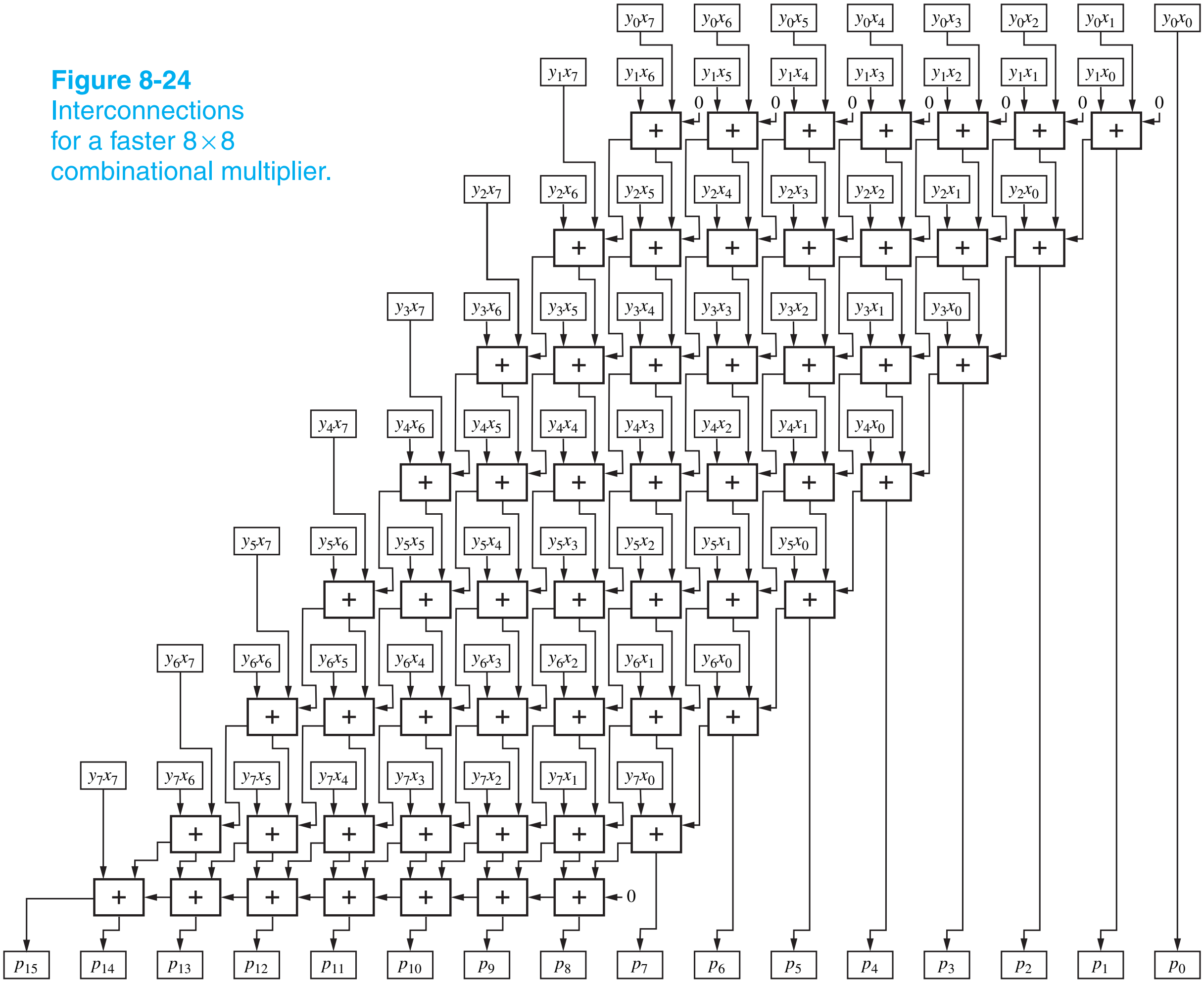


Figure 8-24 Interconnections for a faster 8×8 combinational multiplier.

performance than what the synthesis tool creates. Or we might not. It takes engineering judgment to decide whether optimizing this particular aspect of a larger design could make enough of a difference to make the extra work worthwhile. And it takes experience with the target technology and its tools to know whether a handcrafted design might improve size or performance compared with what the synthesis tool can do.

Program 8-16 Verilog module for an 8×8 combinational multiplier.

```
module Vrmul8x8i(X, Y, P);
  input [7:0] X, Y;
  output [15:0] P;

  assign P = X * Y;
endmodule
```

As an example, suppose the multiplier of Program 8-16 is on the critical timing path for our application and needs to be fast, or perhaps many instances of it are required, so it needs to be as small as possible. It may be worthwhile to try to do better than the synthesis tool, perhaps by creating a handcrafted array-multiplier design. In Figure 8-24 on page 419, we showed the structure of a

Program 8-17 Behavioral Verilog for an 8 x 8 combinational multiplier.

```
module Vrmul8x8p(X, Y, P);
  input [7:0] X, Y;
  output reg [15:0] P;        // output variable for assignment
  reg PC [0:7][7:0];          // product-component bits (2-dim array)
  reg PCS [0:7][7:0];         // full-adder sum bits    (2-dim array)
  reg PCC [0:7][7:0];         // full-adder carry output bits (ditto)
  reg [6:0] RAS;              // ripple adder sum
  reg [7:0] RAC;              //   and carry bits
  integer i, j;

  function MAJ;
    input I1, I2, I3;
      MAJ = (I1 & I2) | (I1 & I3) | (I2 & I3);
  endfunction

  always @ (*) begin
    for (i=0; i<=7; i=i+1)
      for (j=0; j<=7; j=j+1)
        PC[i][j] = Y[i] ? X [j] : 1'b0; // Get product-component bits
    for (j=0; j<=7; j=j+1) begin
      PCS[0][j] = PC[0][j];  // Set up outputs of first-row "virtual"
      PCC[0][j] = 1'b0;      //   full adders (not shown in figure).
    end
    for (i=1; i<=7; i=i+1) begin  // Do all "real" full adders.
      for (j=0; j<=6; j=j+1)
        PCS[i][j] = PC[i][j] ^ PCS[i-1][j+1] ^ PCC[i-1][j];
        PCC[i][j] = MAJ(PC[i][j], PCS[i-1][j+1], PCC[i-1][j]);
      end
      PCS[i][7] = PC[i][7];
    end
    RAC[0] = 1'b0;    // No carry into final ripple adder.
    for (i=0; i<=6; i=i+1) begin                  // Final ripple-adder
      RAS[i] = PCS[7][i+1] ^ PCC[7][i] ^ RAC[i];      // sum
      RAC[i+1] = MAJ(PCS[7][i+1], PCC[7][i], RAC[i]); // and carry bits
    end
    for (i=0; i<=7; i=i+1) begin
      P[i] = PCS[i][0];  // first 8 product bits from full-adder sums
    end
    for (i=8; i<=14; i=i+1)
      P[i] = RAS[i-8];   // next 7 bits from ripple-adder sums
    P[15] = RAC[7];      // last bit from ripple-adder carry-out
  end
endmodule
```

reasonably fast 8×8 Braun multiplier, and Program 8-17 is a behavioral Verilog module `Vrmul8x8p` corresponding to it.

The module begins by declaring its inputs, output, and internal variables. Two-dimensional arrays are used for internal variables `PC`, `PCS`, and `PCC`. Each of these has eight rows indexed from 0 to 7 and eight columns indexed from 7 down to 0. Variable `PC` holds the product-component bits, and variables `PCS` and `PCC` hold the sum and carry outputs, respectively, of the main array of full adders. The bit vectors `RAS` and `RAC` hold the sum and carry outputs of the final ripple adder. Integer variables `i` and `j` are used as loop indexes for rows and columns. Figure 8-25 shows the relationship between the signals in the multiplier circuit on page 419 and the corresponding variable names in the Verilog module.

Next, the function `MAJ` is defined to perform the majority function on three input bits; it is used for full-adder carry outputs later in the module.

In the main body of the module, the first nested `for` statements create the 64 product-component bits in 8 rows of 8 bits each. Each bit `PC[i][j]` equals either the corresponding multiplicand bit `X[j]` or 0, depending on the value of the corresponding multiplier bit, `Y[i]`. The next `for` loop initializes boundary conditions at the top of the multiplier, using the notion of row-0 "virtual" full adders, not shown in the figure, whose sum outputs equal the first row of `PC` bits and whose carry outputs are 0.

The next nested `for` statements correspond to the main array of 49 adders in Figure 8-24, but not the final ripple adder. Note how the index ranges (`i` from 1 to 7 and `j` from 0 to 6) correspond to the "shift" that is evident in the figure. In

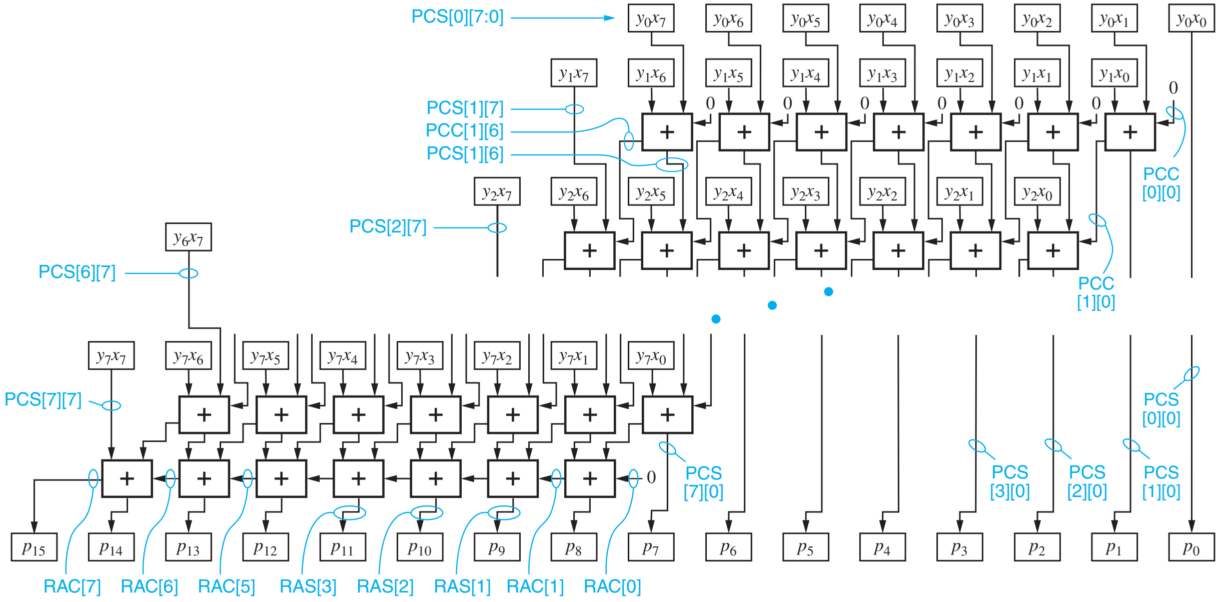


Figure 8-25 Verilog variable names for the 8 × 8 multiplier.

ONE-DIMENSIONAL THINKING Program 8-17 can be rewritten to use one-dimensional arrays of vectors instead of two-dimensional arrays of bits. If you are using ancient tools, this approach works with Verilog-1995, which does not support multidimensional arrays. There are other pros and cons (see Exercise 8.46).

this way, the `PCS[i][j]` and `PCC[i][j]` full-adder outputs are properly derived from the `PCS` and `PCC` row above. The leftmost sum output bit `PCS[i][7]` is handled as a special case, setting it equal to bit `PC[i][7]`, and `PCC[i][7]` is not used and therefore is not computed.

The next `for` loop corresponds to the final ripple adder in Figure 8-24. The last two `for` loops and the final statement assign the appropriate adder outputs to the multiplier output signals.

Note that while the Verilog module in Program 8-17 attempts to illustrate the logic that would be used in a faithful realization of Figure 8-24, a synthesizer could legitimately create quite a different structure from this behavioral code—and does, if it is targeting an FPGA rather than an ASIC. In an ASIC, the synthesizer *might* follow the structure of the behavioral code, but it doesn't have to. If you want to control the structure, then you must use structural Verilog, as we'll soon show. But first, let's check our work so far.

Program 8-18 is a test bench for 8 × 8-multiplier modules. The component instantiation near the beginning of the code determines which version is tested, either of the two shown so far, or the structural model coming up next. This test bench uses the "brute-force" testing method, checking every possible input combination against the expected result. There are a few details to note about it:

- The `initial` block is named so that its local variables `i` and `j` can be declared.
- A pair of nested `for` loops are used to generate all 2^{16} input combinations.
- A task, `checkP`, is defined to compare the circuit's output with the expected product, computed by the simulator as the product of `i` and `j`.
- The comparison in `checkP` uses `!==` rather than `!=`, so that any `x`'s in the circuit's output are detected. This was especially important for testing the structural model; its complex topology provides many opportunities to omit connections, which leads to `x`'s.

It's impressive how quickly a simulator can simulate the synthesized modules for all 65,536 input combinations—about three seconds on a 2-Ghz laptop computer. Still, for larger designs, like 16 × 16 multiplication, we would want to generate random input patterns as in some of our previous examples, rather than use the exhaustive approach.

Program 8-18 Verilog test bench for 8×8 combinational multipliers.

```
module Vrmul8x8_tb();
  reg [7:0] X, Y;
  wire [15:0] P;

  Vrmul8x8i UUT ( .X(X), .Y(Y), .P(P) ); // Instantiate the UUT

  task checkP;
    input i, j, P;
    integer i, j, prod;
    reg [15:0] P;
    begin
      prod = i*j;
      if (P !== prod) begin
        $display($time," Error: i=%d, j=%d, expected %d (%16b), got %d (%16b)",
                 i, j, prod, prod, P, P); end;
    end
  endtask

  initial begin : TB   // Start testing at time 0
    integer i, j;
    for ( i=0; i<=255; i=i+1 )
      for ( j=0; j<=255; j=j+1 ) begin
        X = i; Y = j;
        #10;           // wait 10 ns, then check result
        checkP (i, j, P);
      end
    $display($time," Test ended );     // end test
  end
endmodule
```

Next, we'll show a structural Verilog model of the 8×8 Braun multiplier. It will use a specialized full-adder component `FAblk` defined in Program 8-19, which is like a full adder except that it ANDs two pairs of inputs before applying to the full adder: `A0` and `A1` to get the usual `A`, and `B0` and `B1` to get the usual `B`.

Program 8-19 Full-adder module for optimized structural code.

```
(* keep_hierarchy = "yes" *) module FAblk(A0, A1, B0, B1, CIN, S, COUT);
  input A0, A1, B0, B1, CIN; // full adder with ANDed terms for A and B inputs
  output S, COUT;

  function MAJ;
    input I1, I2, I3;
      MAJ = (I1 & I2) | (I1 & I3) | (I2 & I3);
  endfunction

  assign S = (A0 & A1) ^ (B0 & B1) ^ CIN;
  assign COUT = MAJ ((A0 & A1), (B0 & B1), CIN) ;
endmodule
```

Because of this, the product-component terms PC[i][j] of Program 8-17 can be handled right inside the FAblk. This is efficient when our structural module is targeted to Xilinx 7-series and other FPGAs, where each two-output FAblk, including product-component-bit generation, fits into just one 6-input LUT configured as two 5-input LUTs (see Figure 6-6 on page 244). The module definition also includes the keep_hierarchy constraint to force the tool to keep the FAblk's signals together in synthesis so its inputs and outputs remain present and visible in the synthesized circuit in exactly the way they are defined in the code. This approach gives the designer more control over the circuit structure that is synthesized from the model, as might be desired in an ASIC realization.

Program 8-20 Structural Verilog for an 8 x 8 combinational multiplier.

```
module Vrmul8x8sho(X, Y, P);
  input [7:0] X, Y;
  output [15:0] P;
  wire PCS [0:7][7:0];         // full-adder sum bits
  wire PCC [0:7][7:0];         // full-adder carry output bits
  wire [6:0] PCSv, PCCV;       // temp vectors used for final addition
  genvar i, j;

  generate
    for (j=0; j<=6; j=j+1) begin: FAgenrow1 // FA row 1 has two ANDed terms
                                            // per A and B input but no CIN (1'b0)
      FAblk U1 (.A0(Y[1]), .A1(X[j]), .B0(Y[0]), .B1(X[j+1]), .CIN(1'b0),
                .S(PCS[1][j]), .COUT(PCC[1][j]));
    end
    // remaining FA rows have two ANDed terms on A inputs, plus B and CIN inputs
    for (i=2; i<=7; i=i+1) begin: FAgenrow
     for (j=0; j<=5; j=j+1) begin: col // most FAs have two ANDed terms only on A inputs
        FAblk U2 (.A0(Y[i]), .A1(X[j]), .B0(PCS[i-1][j+1]), .B1(1'b1),
                  .CIN(PCC[i-1][j]), .S(PCS[i][j]), .COUT(PCC[i][j]));
      end
      // leftmost FA of each row is special, uses ANDed terms on both A and B inputs
      FAblk U3 (.A0(Y[i]), .A1(X[6]), .B0(Y[i-1]), .B1(X[7]), .CIN(PCC[i-1][6]),
                .S(PCS[i][6]), .COUT(PCC[i][6]));
    end
  endgenerate

  // take care of boundary cases, do the final addition, and hook up the outputs
  assign PCS[7][7] = Y[7] & X[7];  assign PCC[7][7] = 1'b0;;   // boundary cases
  assign P[0] = X[0] & Y[0];                       // LSB of product
  for (i=1; i<=7; i=i+1) assign P[i] = PCS[i][0]; // next 7 bits come from FA sums
  for (j=0; j<=6; j=j+1) begin
    assign PCSV[j] = PCS[7][j+1]; // make vectors to use built-in add function
    assign PCCV[j] = PCC[7][j];   //    for final 8-bit addition ...
  end
  assign P[15:8] = PCSV + PCCV;   //    ...  to get 8 MSBs of product
endmodule
```

Program 8-20 is the full structural module. It uses generate blocks to create the actual two-dimensional structure of full-adder components (FAblk) and their connections in the pattern shown in Figure 8-25. When used in the first row of full adders, FAblk's CIN input is set to 0; two inputs corresponding to a product-component bit are applied to the A inputs; and another two to B. In most cases when FAblk is used in the second and subsequent rows of Figure 8-25, only one of the B inputs (B0) is used and receives a sum output from the row above; the other B input (B1) is set to 1. However, in the leftmost FAblk in each row, both the A and the B input pairs are used to form product-component bits.

Another interesting aspect of Program 8-20 is that it uses Verilog's built-in addition function, instead of specifying a ripple adder to do the final addition as in Program 8-17. The last for loop extracts the needed bits from the two-dimensional PCS and PCC arrays, creating a pair of vectors to combine using the built-in addition operator. The idea is that when the synthesizer encounters an explicit addition, a common operation, it should know an implementation that is well-suited to the target technology, probably at least as good as a handcrafted design. Compared to a ripple adder using FAblk modules, such an implementation is likely to be smaller or faster or both. In the case of Xilinx 7-series FPGAs, we know that the synthesizer does a good job creating compact and fast adders using the 7-series CARRY4 elements, and in this example, the results are indeed smaller and faster.

PERFORMANCE RESULTS FOR FPGA-OPTIMIZED STRUCTURAL VERILOG CODE

The multiplier structure in Figures 8-24 and 8-25 can be very effectively targeted to Xilinx 7-series FPGAs. Recall that the 7-series LUT can perform any two logic functions of five inputs. The enhanced full adder FAblk in Program 8-19 matches this capability perfectly, so it fits into just one 7-series LUT.

For comparison purposes, I synthesized all of the Vrmul8x8 modules, targeting Xilinx 7-series FPGAs using Vivado tools. The truly behavioral architecture, Vrmul8x8i in Program 8-16, had good QoR (quality of results), using 71 LUTs and having a worst-case delay of 13.38 ns. The explicit behavioral architecture Vrmul8x8p in Program 8-17, despite all my work in creating it, gave worse results, 75 LUTs and 14.83 ns. A structural version, Vrmul8x8s similar to Program 8-20 but using a ripple adder for the final addition (without CARRY4 elements), used only 58 LUTs but had a worst-case delay of 20.49 ns. The optimized structural version in Program 8-20, Vrmul8x8sho, used the fewest LUTs (only 57), and had a worst-case delay at 16.96 ns, still much longer than the tool's implementation of the easy-to-write behavioral version, Vrmul8x8i.

As always, the relative QoR of the four versions could be quite different when targeted to different technologies, like ASICs, and even using different releases of the same software tools, since tools' internal algorithms may get "tweaked." In fact, when I synthesized the same modules with an earlier tool release from a year before, the handcrafted version Vrmul8x8sho won!

*8.4 Dividing

Division is a less common operation than multiplication in computers and digital applications, but still, it happens. Like multiplication, division can be done by a sequential circuit, typically based on a shift-and-subtract algorithm as we discussed briefly in Section 2.9. Many variations of the algorithm have been devised to improve performance.

In this section, we'll describe the most basic bit-at-a-time shift-and-subtract division algorithm, and then show how it can be modeled by a Verilog module and synthesized into a combinational circuit. An easier way to divide is just to use Verilog's built-in division operator, which yields of a combinational divider in synthesis, but depending on the application and the tools, you may get a much more efficient circuit without too much effort by creating your own model of a division circuit based on the basic algorithm.

*8.4.1 Basic Unsigned Binary Division Algorithm

As we discussed in Section 2.9, the division instructions in typical computers divide a 2*n*-bit dividend by an *n*-bit divisor and produce an *n*-bit quotient and an *n*-bit remainder, and setting an "overflow" condition if the divisor is 0 or if the quotient would require more than *n* bits to represent. For simplicity's sake, in this section, we'll divide an *n*-bit dividend by an *n*-bit divisor, so the quotient can always be represented in *n* bits, and we won't worry about the divide-by-0 case.

We'll use four *n*-bit variables in the algorithm:

- DVND — dividend
- DVSR — divisor
- QUOT — quotient
- REM — remainder

The definitions of division is such that DVND = QUOT · DVSR + REM. Even though the basic algorithm uses only *n*-bit inputs and outputs, it also uses a 2*n*-bit register or variable which we'll call RDIV—the "reduced" dividend. To begin the division, the left half of RDIV is initialized to 0 and the right half is loaded with DVND. The algorithm calculates the quotient bits from left to right with *n* repetitions of the following steps, with *i* equal to *n*–1 initially:

1. RDIV is shifted left by one bit.
2. DVSR is compared with the left half of RDIV. If DVSR is less than or equal to RDIV[2n–1:n], then the difference (RDIV[2n–1:n] – DVSR) is loaded into the left half RDIV[2n–1:n] and QUOT bit *i* is set to 1; otherwise, QUOT bit *i* is set to 0. Then *i* is reduced by 1; the last repetition is with *i* = 0.

At the end of *n* steps, all of the bits of QUOT have been set to their proper values, and the left half RDIV[2n–1:n] contains the value of REM.

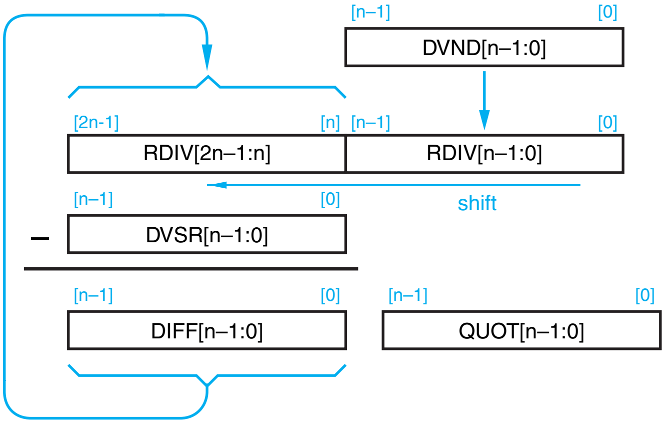


Figure 8-26 Variables used by division algorithm.

The less-than-or-equal comparison in step 2 on the previous page can be performed using subtraction—if (RDIV[2n–1:n] – DVSR) doesn't cause a borrow out of the MSB, then DVSR is less than or equal to RDIV[2n–1:n]. That's convenient, because we can put the subtraction result into a variable DIFF and use that to load the left half of RDIV when there's no borrow. Figure 8-26 shows how the variables are used.

*8.4.2 Division in Verilog

It's easy to specify integer division in Verilog using the language's built-in divide and modulo operators, / and %. Program 8-21 is a module for finding a 32-bit quotient and remainder using the built-in operators and the variables defined in the preceding subsection. It also checks for the divide-by-0 case and sets QUOT and REM to all-1s if that happens.

When Program 8-21 is targeted to a Xilinx 7-series FPGA using Vivado 2016.3 tools, the synthesized combinational circuit uses about 2200 LUTs. Even

Program 8-21 Verilog module for 32-bit division.

```
module Vrdiv32by32 ( DVND, DVSR, QUOT, REM );
  input [31:0] DVND, DVSR;
  output reg [31:0] QUOT;
  output reg [31:0] REM;

  always @ (DVND, DVSR) begin
    if (DVSR==32'b0)
      begin QUOT = 32'hffffffff; REM = 32'hffffffff; end
    else begin
      QUOT = DVND / DVSR;
      REM = DVND % DVSR;
    end
  endmodule
```

Program 8-22 Structural Verilog for a 32 × 32 combinational divider.

```
module Vrdiv32by32_s ( DVND, DVSR, QUOT, REM ); // Integer 32-bit divider
  input [31:0] DVND, DVSR;                      // 32-bit dividend and divisor
  output wire [31:0] QUOT, REM;                 // 32-bit quotient and remainder
  wire [63:0] RDIV[31:-1], SDIV[31:0];          // Reduced and shifted dividends
  wire [32:0] DIFF[31:0];                       // Trial differences
  genvar g;

  assign RDIV[31] = {32'b0,DVND};
  generate
    for (g=31; g>=0; g=g-1) begin: SUB
      assign SDIV[g] = RDIV[g]<<1;
      assign DIFF[g] = {1'b0,SDIV[g][63:32]} - {1'b0,DVSR};
      assign RDIV[g-1] = {(DIFF[g][32]? SDIV[g][63:32]  : DIFF[g][31:0]),SDIV[g][31:0]};
      assign QUOT[g] = DIFF[g][32] ? 0 : 1;
    end
  endgenerate
  assign REM = RDIV[-1][63:32];
endmodule
```

though the algorithm in the preceding subsection creates the remainder as a natural side effect of computing the quotient, the synthesis tool appears not to take advantage of that in Program 8-21—synthesizing a circuit for only QUOT or REM but not both requires only about 1100 LUTs.

We can also write a structural Verilog module that computes the quotient and remainder together, using the algorithm and variables from the preceding subsection, as shown in Program 8-22. This module computes the results combinationally, using an array of 33 64-bit RDIV vectors for the initial value of RDIV and the updated values after each of the 32 iterations of the for loop in the generate block. Another array SDIV holds the shifted value of RDIV as computed at the beginning of each iteration, and an array DIFF of 33-bit vectors holds the results of the subtraction that occurs at each iteration. Note DIFF and the subtraction are set up as 33 bits wide to hold the borrow from bit 31 in the MSB (bit 32).

In synthesis, the first assign statement in the for loop doesn't generate any logic; in effect it's just copying (renaming) signals for the current iteration. The second assign statement does create a real 32-bit subtractor for each iteration, and the third creates a 32-bit multiplexer which selects one of two inputs based on the borrow bit DIFF[g][32]) and also does 32 bits of copying (renaming). The last assign statement sets the value of one quotient bit according to DIFF[g][32]. After the for loop, the remainder is copied (renamed) from the left half of the last value of RDIV.

When Program 8-22 is targeted to a Xilinx 7-series FPGA using Vivado 2016.3 tools, the synthesized combinational circuit uses only about 1500 LUTs. So, the handcrafted module is 25% smaller than the tools' when both QUOT and

Program 8-23 Test bench for a 32 × 32 combinational divider.

```
`timescale 1ns/100ps
module Vrdiv32by32_tb ( );
  reg [31:0] DVND, DVSR;
  wire [31:0] QUOT;
  wire [31:0] REM;
  integer i;

  task DispResults;
    begin
      $display("DVND,DVSR,QUOT,REM: %010d,%010d,%010d,%010d", DVND, DVSR, QUOT, REM);
      $display("DVND/DVSR,DVND%%DVSR:                      %010d,%010d",
                DVND/DVSR, DVND%DVSR);
    end
  endtask

  Vrdiv32by32_s UUT ( .DVND(DVND), .DVSR(DVSR), .QUOT(QUOT), .REM(REM) );

  initial begin
    DVND = 0; DVSR = 0; #50 DispResults; // Check a few divide-by-0 results first
    for (i=1; i<=10; i=i+1) begin
      DVND = $random; DVSR = 0; #50 DispResults;
    end
    for (i=1; i<=100; i=i+1) begin       // Test full 32-bit random DVND and DVSR
      DVND = $random; DVSR = $random ; #50 DispResults;
    end
    for (i=1; i<=1000; i=i+1) begin      // Also test with 8-bit DVSR for bigger QUOTs
      DVND = $random; DVSR = $random & 8'hff; #50 DispResults;
    end
  $stop(1);
  end
endmodule
```

REM are needed. But the tools' built-in method of synthesizing division is more efficient when only one of the results is required.

A test bench for the divider is shown in Program 8-23. It uses Verilog's built-in $random function to generate 32-bit test inputs, and for each input pair the DispResults task displays the dividend, the divisor, and the quotient and remainder produced both by the UUT and by the Verilog simulator's built-in / and % functions, in the output's first and second line, respectively. If desired, this task can be easily modified to compare results and keep track of the number of mismatches (see Drill 8.15).

The test bench's first for loop begins by checking a few divide-by-0 cases. If you study the logic in Program 8-22, you can figure out what the divide-by-0 results should be, but running the test bench confirms it (see Exercise 8.49). The second for loop checks the UUT's operation with random values for both operands. Since most of the randomly generated 32-bit operands will have some 1s in the high-order bits, the operands will usually be large numbers of the about the same magnitude, and the quotients will usually be small—0 about half of the time! The last `for` loop reduces the random divisor to 8 bits, so cases with larger quotients and "interesting" divisors like 0 and 1 are more likely to be generated.

Dividing by a constant can be done more efficiently than dividing by a variable, and there are applications where such division is required. A typical one is in converting a binary number into a string of BCD digits; for example, for use with a seven-segment display, using the algorithm described in Section 2.3. This algorithm repeatedly divides a given binary number by 10, generating the BCD digits from right to left, each one being the remainder from a division by 10.

Suppose we need to convert a 32-bit number to a sequence of BCD digits. The largest value of an unsigned 32-bit number is 2^{32}–1 or 4,294,967,295, which has 10 digits. So we'll need nine instances of a divide-by-10 circuit if we do the conversion with a combinational circuit, which we will later in this subsection. With 10 instances, it behooves us to do a good job on the divide-by-10 circuit.

Program 8-24 is a very simple and straightforward divide-by-10 module that uses Verilog's built-in division and modulus operators. Its outputs are both a 32-bit quotient and a 4-bit remainder, since both will be needed in the binary-to-BCD circuit. When targeted to a Xilinx 7-series FPGA using Vivado 2016.3 tools, the synthesized combinational circuit uses 614 LUTs. Note that in successive steps in the conversion algorithm, fewer and fewer of the `Vrdiv10` module's high-order dividend and quotient bits will be needed, so we can expect the synthesis tool to prune away any unneeded logic when it puts it all together. Still, compared to the full 32×32 divider in Program 8-21 (1100 LUTs), it doesn't seem like we saved a lot by using a constant divisor; maybe we can do better.

We know that our structural 32×32 divider module in Program 8-22 was smaller than one using the built-in Verilog operators when both quotient and remainder were required, so maybe instantiating that with a constant divisor would yield a better synthesized circuit. Program 8-25 shows how to do it. The divisor is set to be a 32-bit constant decimal 10, and the remainder that we know to be just four bits is returned to a 32-bit internal wire `IWIRE` for subsequent

Program 8-24 Verilog module to divide a 32-bit number by 10.

```
module Vrdiv10 ( D, QUOT, REM );  // Integer divide by 10
  input [31:0] D;                 // 32-bit dividend
  output reg [31:0] QUOT;         // 32-bit quotient
  output reg [3:0] REM;           // 4-bit remainder (<10)

  always @ (D) begin
    QUOT = D / 10;
    REM = D % 10;
  end
endmodule
```

Program 8-25 Hierarchical module to divide a 32-bit number by constant 10.

```
module Vrdiv10_sf ( D, QUOT, REM ); // Integer divide by 10
  input [31:0] D;                   // 32-bit dividend
  output wire [31:0] QUOT;          // 32-bit quotient
  output wire [3:0] REM;            // 4-bit remainder (<10)
  wire [31:0] IREM;                 // Internal REM for assignment

  Vrdiv32by32_s U1 (.DVND(D),.DVSR(32'd10),.QUOT(QUOT),.REM(IREM));
  assign REM = IREM[3:0];
endmodule
```

assignment to REM. Unfortunately, the tool could do even less optimization on this version of the circuit, and it synthesized a result with 747 LUTs. But we're not giving up!

Program 8-26 is a structural module for dividing by 10 that uses the same basic division algorithm as our previous structural module, but it has several optimizations:

- In concept, the 4-bit constant divisor is shifted to the right, under the 32-bit dividend in RDIV which does not shift; in the more general algorithm, the 0-padded dividend is shifted to the left in a 64-bit RDIV.
- Since the divisor is known to be four bits wide, it can be "lined up" under RDIV[31:28] for the first trial subtraction, eliminating the first three trial subtractions and all but one bit of the 0-initialized left half of RDIV in the general algorithm (RDIV is now only 33 bits wide).

Program 8-26 Optimized structural module to divide a 32-bit number by constant 10.

```
module Vrdiv10_so ( D, QUOT, REM ); // Integer divide by 10
  input [31:0] D;                   // 32-bit dividend
  output wire [31:0] QUOT;          // 32-bit quotient
  output wire [3:0] REM;            // 4-bit remainder (<10)
  wire [32:0] RDIV[28:-1];          // Reduced dividends (MSB unused except at g=28)
  wire [4:0] DIFF[28:0];            // Trial differences
  genvar g;

  assign RDIV[28] = {1'b0,D};  assign QUOT[31:29] = 3'b000;
  generate
    for (g=28; g>=0; g=g-1) begin: SUB
      assign DIFF[g] = RDIV[g][g+4:g] - 5'b01010;
      assign RDIV[g-1][g+3:g] = DIFF[g][4] ? RDIV[g][g+3:g] : DIFF[g][3:0];
      if (g>=1) assign RDIV[g-1][g-1:0] = RDIV[g][g-1:0]; // No copy on last iteration
      assign QUOT[g] = DIFF[g][4] ? 0 : 1;
    end
  endgenerate
  assign REM = RDIV[-1][3:0];
endmodule
```

Program 8-27 Test bench for divide-by-10 modules.

```
`timescale 1ns/100ps
module Vrdiv10_tb ( );
  reg [31:0] D;
  wire [31:0] QUOT;
  wire [3:0] REM;
  integer i;

  Vrdiv10_so UUT ( .D(D), .QUOT(QUOT), .REM(REM) );

  initial begin
    for (i=1; i<=1000; i=i+1) begin
      D = $random; #10 ;
      $display ("Random number:  %010d",D);
      $display ("DIV by 10, REM: %010d, %1d", QUOT, REM);
      if ((QUOT!==D/10) || (REM!==D%10)) $display("*****ERROR*****");
    end
  $stop(1);
  end
endmodule
```

- Also because of the known 4-bit divisor, the three leftmost quotient bits are known to always be 0.
- The operands and results of the trial subtractions are explicitly formulated to be only five bits wide—four bits for the divisor and the fifth bit on the left to capture the borrow.

This version requires only 84 LUTs, such a big improvement that it's hard to believe that the design is correct! But we've got a test bench to prove it, in Program 8-27. All three of the divide-by-10 modules in this subsection pass with no errors.

Now that we have a pretty good divide-by-10 module, we can go ahead and design the full 32-bit to BCD conversion circuit. It's not too difficult using a structural approach, as shown in Program 8-28. The module instantiates nine copies of the `Vrdiv10_so` module, passing the `QUOT` output of each to the `D` input of the next. The `REM` outputs are the BCD digits, generated from right to left, and packed into a 40-bit vector to hold the ten 4-bit digits. The ninth, final divide-by-10 is a special case, where the four low-order bits of its `QUOT` output are in fact the most significant BCD digit.

When Program 8-28 is targeted to a Xilinx 7-series FPGA using Vivado 2016.3 tools, the synthesized combinational circuit uses 332 LUTs in 18 levels and has a maximum delay of about 16 ns. If we resynthesize it using our original `Vrdiv10` module, the results are 5766 LUTs in 210 levels, and 88 ns of delay. So, even when targeting an FPGA with tens of thousands of LUTs available, the extra effort to optimize this design was well worth it.

Program 8-28 Verilog module for 32-bit binary to 10-digit BCD conversion.

```
module Vrbintodec32 ( BIN, DEC );
  input [31:0] BIN;
  output wire [39:0] DEC;
  wire [31:0] quot [9:0];
  genvar g;

  assign quot[0] = BIN;
  generate
    for (g=0; g<=8; g=g+1) begin: DIV
      Vrdiv10_so U1 (.D(quot[g]), .QUOT(quot[g+1]), .REM(DEC[4*g+3:4*g]));
    end
  endgenerate
  assign DEC[39:36] = quot[9][3:0];
endmodule
```

UNNATURAL SELECTION

The module in Program 8-28 uses a 40-bit vector to output ten BCD digits. It might seem more natural to declare the output as an array of 4-bit vectors, for example, "`output wire [3:0] DIGITS [9:0]`", which would make it easier to select the individual digits. But standard Verilog does not allow an array to be used as an input or output port; you have to move to SystemVerilog for that capability. So the only option is to pack the array into a vector as we have here, and unpack it inside the module as needed.

References

Descriptions of algorithms for arithmetic operations appear in *Digital Arithmetic* by Miloš Ercegovac and Tomas Láng (Morgan Kaufmann, 2003). A thorough discussion of arithmetic techniques and floating-point number systems can be found in *Introduction to Arithmetic for Digital Systems Designers* by Shlomo Waser and Michael J. Flynn (Oxford University Press, 1995).

A detailed, comprehensive treatment of arithmetic algorithms and implementation is given by Behrooz Parhami in *Computer Arithmetic* (Oxford University Press, 2009, second edition). For a book that focuses in particular on Verilog implementations, see *Computer Arithmetic and Verilog HDL Fundamentals* by Joseph Cavanaugh (CRC Press, 2009).

Drill Problems

8.1 Write an algebraic expression for s_3, the fourth sum bit of a binary adder, as a function of inputs a_0, a_1, a_2, a_3, b_0, b_1, b_2, and b_3. Assume that $c_0 = 0$, and do not attempt to "multiply out" or minimize the expression.

8.2 Assume that an inverting gate has a delay of 1 unit, an AND-OR or OR-AND circuit with no complemented inputs has a delay of 2 units, and an XOR or XNOR gate has a delay of 3 units. What is the worst-case delay from any input to any sum output for the 4-bit ripple adder in Figure 8-2? What is the worst-case delay to the carry output?

8.3 Using the same assumptions as in Drill 8.2, determine the worst-case delay from any input to any sum output, as well as the worst-case delay to the carry output, for the 4-bit carry lookahead adder in Figure 8-6.

8.4 Using the information in Table 4-3 for 74HC components operating at 4.5 V, determine the maximum propagation delay from any input to any output of the 16-bit group ripple adder of Figure 8-7.

8.5 Suppose that the 4-bit carry lookahead adder in Figure 8-6 is augmented to provide group carry lookahead outputs using the equations in Section 8.1.6. Using the same assumptions as in Drill 8.2, determine the worst-case delay from any input to any group carry lookahead output.

8.6 Write a dataflow-style Verilog module `Vradder8` for an adder with two 8-bit inputs `A` and `B`, carry input `CIN`, 8-bit sum output `S`, and carry output `COUT`.

8.7 Write a dataflow-style Verilog module `Vr74x182` that performs the same function as the 74x182 lookahead carry circuit, but with active-high generate and propagate signals.

8.8 Write a simple behavioral Verilog module `Vraddbytes64` for a circuit that adds the bytes in a 64-bit input longword `D`, considering each byte as an unsigned integer, and returning a 12-bit result `S`.

8.9 Write a test bench `Vraddbytes64_tb` that checks the module in Drill 8.8 for correct operation for 10,000 random input values on `D`.

8.10 Repeat Drills 8.8 and 8.9 for a module `Vraddbytes64_g` that uses `generate`.

8.11 Write a simple behavioral Verilog module `Vrcnt1s` for a 1s-counting circuit with a 32-bit input `D` and a 5-bit output `SUM` which gives the number of 1 bits in `D`.

8.12 For the barrel-shifter design of Figure 8-17, and using the design assumptions in the text, sketch the delay paths from DIN and S to DOUT and determine how many inverting gates are on the worst-case path(s). Be sure to read the box on page 405.

8.13 Repeat Exercise 8.12 assuming that noninverting buffers are used on DIN and S.

8.14 Which 16-bit barrel-shifter design is likely to require more chip area for wiring, Figure 8-17 or Figure 8-19?

8.15 Modify the test bench in Program 8-23 to compare the results produced by the UUT and Verilog's built-in functions for each test case, and to run a much larger number of cases. Be sure your code handles divide-by-0 cases sensibly.

Exercises

8.16 Suppose that the 4-bit adders in Figure 8-8 do not have a C4 output. (This was the case with some MSI adders with group-carry-lookahead outputs.) Write the logic equation for the carry out of the overall addition ("C16") as a function of the existing signals in the logic diagram.

8.17 Assume that an inverting gate has a delay of 1 unit, an AND-OR or OR-AND circuit with no complemented inputs has a delay of 2 units, and an XOR or XNOR gate has a delay of 3 units. Determine the maximum delay from any input to any sum output in the 16-bit group-carry-lookahead adder of Figure 8-8, further assuming that the carry lookahead logic is implemented using the equations in Section 8.1.6. Similarly, determine the delay to the carry output. You may build on the results of Drills 8.3 and 8.5.

8.18 Suppose that the C16 output in Figure 8-8 is implemented inside the lookahead carry circuit, using logic similar to its other carry outputs as suggested in the last paragraph of Section 8.1.6. Using the same assumptions as in Exercise 8.17, determine whether such a C16 output would have a shorter delay than the one calculated in Exercise 8.17.

8.19 Repeat Drills 8.8 and 8.9, but considering each byte of D to be a *signed* integer, and producing a 16-bit signed output.

8.20 Write a dataflow-style Verilog module Vr2bgcladder for a 2-bit group carry lookahead adder with inputs A, B, and CIN, and outputs S, G, and P (note that the generate and propagate signals are active high).

8.21 Write a structural Verilog module Vr8bgcladder_s for an 8-bit group carry lookahead adder by instantiating the Vr74x182 module of Drill 8.7 and four copies of Vr2bgcladder from Exercise 8.20. The 8-bit module should have the same kinds of inputs and outputs as the 2-bit module.

8.22 Write a test bench module Vr8bgcladder_tb that instantiates the 8-bit adder in Exercise 8.21 and checks for correct outputs for all 2^{17} input combinations. If the test passes the first time, insert one or more errors into your adder module(s) to ensure that your error-detection and display code works properly.

8.23 Synthesize the 8-bit adder module Vr8bgcladder_s of Exercise 8.21, targeting a Xilinx 7-series FPGA. Also synthesize the Vradder8 module of Drill 8.6 and target it to the same FPGA. Compare the resources (number of LUTs) required for the two designs and their speeds (worst-case delays). Comment on and explain any significant differences. Based on your observations, should one design approach be favored over the other for FPGA-based adders?

8.24 Find a way to delete a single character in Program 8-5 such that the Verilog compiler detects no errors and the synthesized module always produces the correct sum output, but the test bench in Program 8-6 now detects tens of thousands of errors. (The purpose of this exercise is to strengthen your belief in the usefulness of test benches!)

8.25 Write a structural, hierarchical Verilog module Vr64bGCLAadder_s for a 64-bit group-carry-lookahead adder by instantiating modules Vr16bGCLAadder_s and Vr4iLACckt in Programs 8-8 and 8-9. Adapt the test bench in Program 8-7 to test your module, including code to test the 64-bit adder's super-supergroup lookahead outputs.

8.26 Starting with the logic diagram for the 74x283 in Figure 8-6, write a logic expression for the S2 output in terms of the inputs, and prove algebraically that it does indeed equal the third sum bit in a binary addition as advertised. You may assume that $c_0 = 0$ (i.e., ignore c_0).

8.27 Estimate the number of product terms in a minimal sum-of-products expression for the c_{32} output of a 32-bit binary adder. Be more specific than "billions and billions," and justify your answer.

8.28 Draw the logic diagram for a 64-bit fast adder using sixteen 4-bit group-carry-lookahead adders and five 4-group lookahead carry circuits. For the 4-bit adders, you need show only the Gg and Pg outputs and the carry inputs and outputs.

8.29 Write a structural Verilog module `Vr74x283_8s` for an 8-bit carry-lookahead adder, similar in structure to the 74x283 4-bit adder, using a generate statement. Check your design with the test bench in Program 8-7.

8.30 Augment the Verilog module in Program 8-5 for a 74x381-like ALU by adding COUT (carry-out) and OVFL (overflow) outputs. Write or adapt a test bench to verify your design.

8.31 Which if any signals are produced differently in Figure 8-11 as compared to Figure 8-8? Explain the reason for any differences.

8.32 Modify the Verilog module in Table 8-5 for a 74x381-like ALU by including an *n*-bit variable `C` and computing the sum and differences using `C`, as discussed in the box on page 388. Write or adapt a test bench to verify your design for addition and both subtraction operations, for all input combinations.

8.33 Write a hierarchical Verilog module for a 16-bit Kogge-Stone adder based on Figures 8-12, 8-13, and 8-14. Use `generate` statements and `for` loops to instantiate all of the GPR circuits; do not hook up all their inputs and outputs "by hand." Check your module for correct operation using the test bench in Program 8-7.

8.34 The values of N in the GPN prefixes of the Brent-Kung prefix-adder graph in Figure 8-15 are not all powers of 2. Why?

8.35 Write a hierarchical Verilog module for a 16-bit Brent-Kung adder based on Figures 8-12, 8-13, and 8-15. Use `generate` statements and `for` loops to instantiate the GPR circuits as best as you are able; do not hook up all their inputs and outputs "by hand." Check your module for correct operation using the test bench in Program 8-7.

8.36 Write a structural Verilog module `Vrcnt1s_s` for a 1s-counting circuit with a 32-bit input `D` and a 5-bit output `SUM` which gives the number of 1 bits in `D`. Your structural module should break up `D` into *b*-bit chunks, where *b* is the number of inputs in your favorite FPGA's LUTs, for example 6 in the Xilinx 7 series. Define a module `CNT`*b*, for example `CNT6`, that counts the number of 1s in a *b*-bit chunk. Then instantiate `CNT`*b* multiple times in `Vrcnt1s_s`, and add those results to get the final `SUM` value. Synthesize your design for the selected FPGA family, and compare its size and speed of with that of the simple behavioral approach in Exercise 8.11. (*Hint*: Depending on the version of the tools, the author was able to achieve improvements of 10% and 5% in size and speed. You may also wish to explore other hierarchical structures.)

8.37 Write the Verilog functions for `Vror`, `Vsll`, `Vsrl`, `Vsla`, and `Vsra` that are needed in Program 8-10 using the corresponding shift operations defined in Table 8-2.

8.38 Determine which of the Verilog functions for Vror, Vsll, Vsrl, Vsla, and Vsra in Table 8-10 can be easily coded using one of Verilog's built-in shift operators instead of a for loop, and write and test the new code.

8.39 Write a test bench Vrrolr16_tb for the Verilog left/right barrel shifter module in Program 8-12 for random data inputs and all possible combinations of control inputs, and use it to test the module.

8.40 Redesign the Verilog left/right barrel shifter module in Program 8-12, creating a new module Vrrolr16_h that it simply instantiates the Vrrol16 module of Program 8-11 using a value of S that is modified appropriately if DIR is 1. Use the test bench of Exercise 8.39 to test your design. Assuming that the synthesizer faithfully follows the structure implied by each module version, discuss the pros and cons of each version. Then, target each module to your favorite programmable device and determine whether the choice of design approach makes any difference to the size and speed of the implementation.

8.41 Rewrite the Vrbarrel16_s module in Program 8-14 to make a new module Vrbarrel16_sr using the structure shown in Figure X8.41. Use the existing ROL16 and FIXUP modules; it's up to you to come up with MAGIC and the other logic. Compare the size and speed of the new synthesized module with the original.

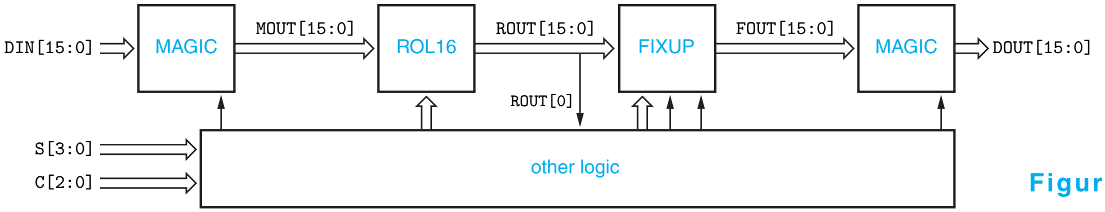


Figure X8.41

8.42 Rewrite the Vrbarrel16_s module of Program 8-14 using Vrrolr16_h from Exercise 8.40 and target the new module to your favorite programmable device. Compare the size and speed of the new synthesized module with the original and optionally with Vrbarrel16_sr from Exercise 8.41.

8.43 Modify the Vrbarrel16_s module of Program 8-14 so its DOUT output is a copy of the DIN input when either of the unused mode values is applied to C. Try to minimize the impact on size and speed and compare with the original module.

8.44 Determine the worst-case propagation delay of the multiplier in Figure 8-23, assuming that the propagation delay from any full-adder input to its sum output is twice as long as the delay to the carry output. Repeat, assuming the opposite relationship. If you were designing the adder cell from scratch, which path would you favor with the shortest delay? Is there an optimal balance?

8.45 Repeat the preceding exercise for the multiplier in Figure 8-24.

8.46 Modify the Vrmul8x8p multiplier module of Program 8-17 to use one-dimensional arrays of 1-byte-wide vectors for PC, PCS, and PCC. What are the pros and cons of this approach? Test your module using the test bench in Program 8-18. *Optional*: Are the synthesis results for the two versions identical?

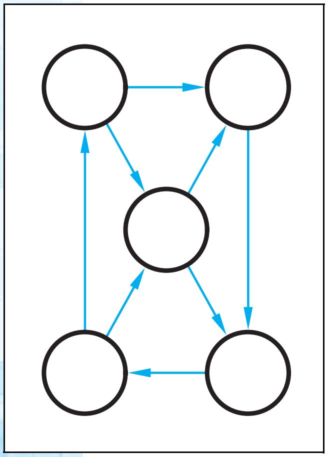
8.47 When I synthesized the Vrmul8x8sho module of Program 8-20 using Xilinx Vivado tools version 2016.3 and targeting a 7-series FPGA, the tools *insisted* on using two separate 6-input LUTs for each FAblk regardless of the tools' option settings, even though it can clearly be implemented in one LUT configured as two 5-input LUTs as in Figure 6-6 on page 244. This forced me to do a workaround—defining a new "FAblkLUT" module structurally, as a single instantiation of the Xilinx 7-series LUT6_2 library component. To do this, I had to manually create truth tables corresponding to the two output functions and then convert them into a 64-bit string to initialize the LUT6_2 component's lookup table using its INIT parameter in the instantiation. Figure out how to do all this yourself and write your own FAblkLUT module. Check your work by substituting your FAblkLUT into the Vrmul8x8sho module and testing it with Vrmul8x8_tb.

8.48 Using the latest version of Xilinx Vivado tools, synthesize the Vrmul8x8sho module of Program 8-20, targeting a 7-series FPGA. Determine whether Xilinx has fixed the "limitations" (some would say bugs) that led to the creation of Exercise 8.47. How many LUTs are used by the synthesized module?

8.49 Study the logic of the Verilog 32-bit structural divider module in Program 8-22, and determine what results it produces when DVSR is 0, including the case where DVND is also 0. Run the test bench in Program 8-23 to confirm your analysis, and why these results occur. Modify the module so it produces the same results as Program 8-21 in the divide-by-0 cases, and confirm by again running the test bench against both modules. Compare the resource requirements of the modified module with the original when targeted to your favorite FPGA.

8.50 Design a Verilog module Vrbcd10div3 whose input is a 10-digit BCD integer packed into a 40-bit vector DIGS. The module's output should be a single signal DIV3 that is 1 if the input number is evenly divisible by 3. Use Verilog's built-in multiplication and addition operations to compute the binary equivalent of the 10-digit number and to divide it by 3; do not attempt to design any custom circuits for multiplication by 10 or division by 3. Synthesize the module for your favorite FPGA and determine how many resources (LUTs) it uses and how many LUTS are in its worst case delay path.

8.51 A well-known math trick is that a decimal number is evenly divisible by 3 if and only if the sum of its digits is evenly divisible by 3. Use this trick in a new module Vrbcd10div3t for the problem statement in Exercise 8.50. Also write a test bench Vrbcd10div3_tb that compares the outputs of the two modules for 10,000 random 10-digit integers and ensures that they are equal. Synthesize your new module and compare its resource requirements and delay with the original.



chapter 9

State Machines

We've previously said that logic circuits are classified into two types: combinational and sequential. A combinational logic circuit's outputs depend only on its current inputs, while a sequential circuit's output may also depend its past inputs, possibly arbitrarily far back in time. In practice, almost all logic circuits are sequential circuits, because almost all applications need the kind of functionality that sequential circuits provide.

To describe the functionality of a combinational logic circuit, we can use an input/output table—the truth table—that simply specifies the circuit's outputs for all possible input combinations. This approach is practical as long as the number of input combinations is not too large.

With sequential circuits, you might think to extend this approach to use an input/output table that lists output values as a function of the *sequence* of input combinations that has been received up until the current time. But how long of a sequence is needed? As we've said, a sequential circuit's output may depend on inputs received arbitrarily far back in time, and the circuit may have been operating for a *long* time.

For example, in the introduction of Chapter 3 we described a fan-speed control circuit operated by up/down pushbuttons. With this circuit, it's not possible to determine the current fan speed by looking only at a predetermined number of previous up/down pushes, whether that number is 1, 10, or 1000; the circuit may have received even more up/down pushes.

9.1 State-Machine Basics

We can determine what the output of the fan-speed circuit should be in response to an input (up/down push) if we know its current "state," and this can be done very concisely. After decades of digital-design experience, the best definition of "state" that I've seen is still the one in Herbert Hellerman's book on *Digital Computer System Principles* (McGraw-Hill, 1967):

state
state variable

> The *state* of a sequential circuit is a collection of *state variables* whose values at any one time contain all the information about the past necessary to account for the circuit's future behavior.

In the fan-speed example, the fan's current speed is the current state. Inside a three-speed fan, this state might be stored as two binary state variables representing a decimal number between 0 and 3, with 0 corresponding to "off" and 3 to the highest speed. Given the current state (speed 0–3), we can always predict the next state as a function of the inputs (presses of the up/down pushbuttons).

Of course, we need more than just the state variables to describe a sequential circuit's operation—we need to know how the circuit will act in response to a given input combination when it is in any given state. This information may be formally specified in a set of tables that we'll describe in detail in Section 9.2.

Another example of a simple sequential circuit would be a traffic-light controller. (I say "would be" because nowadays they all use microprocessors to run a program with the control algorithm, instead of using a hardwired circuit.) Let's consider a sequential circuit that controls North-South and East-West directions of traffic flow, provides a double-red interval between direction changes for safety, and also has a "flashing-red" mode of operation.

In the traffic-light example, the controller circuit's current state cannot necessarily be deduced just by looking at the outputs. For example, if the N-S lights are green and E-W are red, we know the state. But if both directions are red, what is the state? All the lights may be about to blink off, for flashing-red operation. Or, the circuit may be in a double-red interval between direction changes and if so, which direction will be green next, N-S or E-W? Drivers may guess and hit the accelerator at their own peril. So, while the controller circuit's current state determines its current output, the current output does not always imply the current state.

finite-state machine (FSM)
state machine

Instead, we must go back to the definition of state, and the idea of state variables. In a digital circuit, state variables are binary values, corresponding to certain logic signals in the circuit, as we'll see in later sections. A circuit with n binary state variables has 2^n possible states. As large as it might be, 2^n is always finite, never infinite, so sequential circuits are sometimes called *finite-state machines (FSMs)*, or more often simply *state machines*.

State variables need not have direct physical significance, and there are an unlimited number of ways to choose them to describe a particular sequential circuit, many of which make sense for one reason or another. For example, in the

NON-FINITE-STATE MACHINES

A group of mathematicians recently proposed a non-finite-state machine, but they're still busy listing its states. . . . Sorry, that's just a joke. There *are* mathematical models for infinite-state machines, such as Turing machines. They typically contain a small finite-state-machine control unit, and an infinite amount of auxiliary memory, such as an endless tape.

traffic controller, and for simplicity assuming no yellow lights, we need six states: two for N-S green and the double-red afterwards, a similar two for E-W, and two for flashing-red (to loop between double-red and double-off). We could encode these states in three bits and build a combinational circuit that turns on lights as needed as a function of these three bits. Or, we could use six bits with *some* physical significance: the N-S and E-W green and red outputs (four bits that control lights directly), and two more bits to distinguish among the three states where both green lights are off and both reds are on. While these six state bits could encode up to 64 different states, in the traffic controller they would take on only six different combinations for six states.

When do state changes happen? In most sequential circuits, they can occur only at times specified by a free-running *clock* signal. Figure 9-1 gives timing diagrams and nomenclature for typical clock signals. By convention, a clock signal is active high if state changes occur at the clock's rising edge (transitioning from LOW to HIGH), and active low if they occur at the falling edge. The edge at which state changes occur may be called the *active* or *triggering edge*. The *clock period* is the time between successive transitions in the same direction, and the *clock frequency* is the reciprocal of the period. The triggering edge is often called a *clock tick*. The *duty cycle* is the percentage of time that the clock signal is at its asserted level (i.e., HIGH for an active-high clock). As indicated in the figure, state changes occur only at the triggering clock edge. Between triggering edges, the state remains stable.

clock

active edge
triggering edge
clock period
clock frequency
clock tick
duty cycle

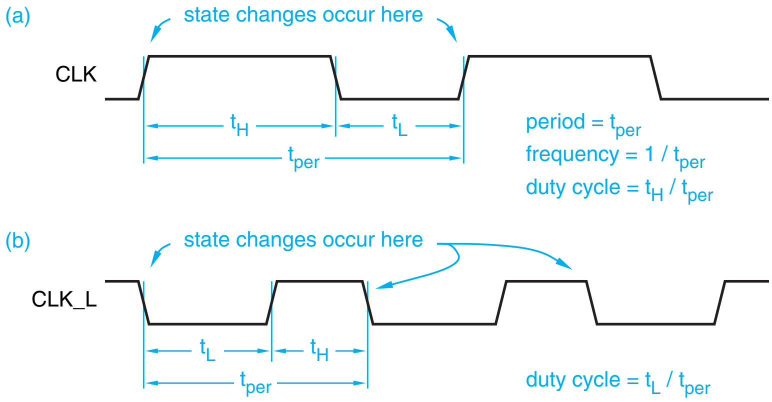


Figure 9-1
Clock signals:
(a) active high;
(b) active low.

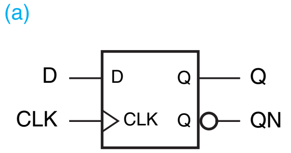
FAST CLOCKS A clock as fast as 4 GHz typically is not distributed at the PCB level. Rather, a slower clock, say 200 MHz, is distributed to ICs such as microprocessors that run faster internally. Each of these has an on-chip *digital phase-locked loop (DPLL)* that can generate an internal clock at an integer multiple of the 200 MHz reference frequency. The multiple can even be changed dynamically, for example to slow down a microprocessor clock to save power when it doesn't have a lot to do.

Typical digital systems, from digital watches to supercomputers, use a quartz-crystal oscillator to generate a free-running clock signal. Clock frequencies might range from 32.768 kHz (for a watch) to 4 GHz (for a CMOS microprocessor with a cycle time of 250 ps). At the PCB level, typical systems using CMOS parts have clock frequencies in the 5–500 MHz range. The very highest clock frequencies are normally achieved only on-chip by an internally generated clock, as in the 4-GHz microprocessor example.

positive-edge-triggered D flip-flop

Most sequential circuits and almost all state machines use a particular type of element to store their state variables, namely an edge-triggered D flip-flop. The logic symbol for a *positive-edge-triggered D flip-flop* is shown in Figure 9-2(a), and its "function table" is shown in (b). The circuit's inputs are D and CLK; its outputs are Q and optionally QN, which is the complement of Q. The outputs may change only at the rising ("positive") edge of the controlling CLK signal. When CLK transitions from LOW to HIGH, the circuit samples its D input and places the current value of D on the Q output, also placing the complement of that value on QN if present. Between LOW-to-HIGH clock transitions, the flip-flop maintains the value previously stored on Q (and QN). Figure 9-3 shows this functional behavior for an example input sequence.

Figure 9-2 Positive-edge-triggered D flip-flop: (a) logic symbol; (b) function table.



(b)

D	CLK	Q	QN
0	↑	0	1
1	↑	1	0
x	0	last Q	last QN
x	1	last Q	last QN

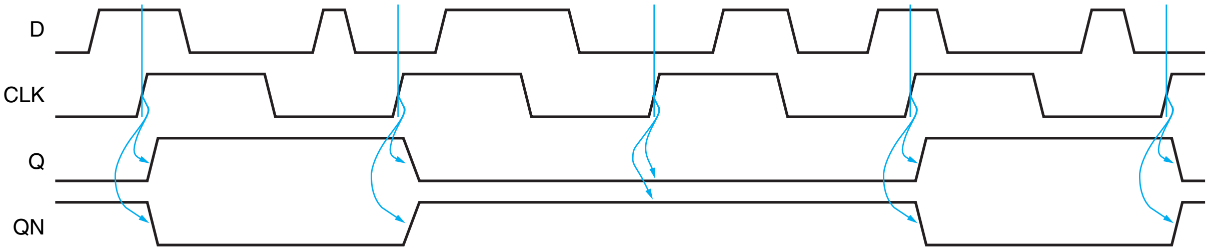


Figure 9-3 Functional behavior of a positive-edge-triggered D flip-flop.

LET'S NOT BE NEGATIVE

There are also negative-edge-triggered D flip-flops which sample their inputs and change their outputs on HIGH-to-LOW clock transitions. As a mathematician would say, "without loss of generality" we'll stick with positive-edge triggered D flip-flops in our state-machine discussions.

However, you may eventually encounter a situation—almost certainly *not* in a state machine—where both positive- and negative-edge triggered flip-flops are used in the same circuit. This is done to achieve so-called "double data rate" (DDR) operation, where data is sampled and stored on both edges of the clock. The same result could be achieved with positive-edge-triggered flip-flops and a double-frequency clock, but there are certain electrical advantages using DDR. As you might expect, there are drawbacks as well, but the trade-offs are such that DDR has been popular in a number of very common applications, including the memory interfaces of PCs.

This chapter focuses on state machines with D flip-flops as they are used in the majority of practical designs, but there are other types of sequential circuits. A *feedback sequential circuit* uses ordinary gates and feedback loops to obtain memory in a logic circuit, thereby creating sequential-circuit building blocks such as the D flip-flops themselves. Most digital designers never need to design such circuits from scratch, because they already exist within a larger component or device library. However, a basic understanding of feedback sequential circuits is useful and we'll give a short introduction to them in Section 10.8. Other sequential circuit types, such as general fundamental mode, multiple-pulse mode, and multiphase circuits, are sometimes useful in high-performance systems and VLSI and are discussed in advanced papers and texts.

feedback sequential circuit

9.2 State-Machine Structure and Analysis

Historically, several different approaches and storage elements have been used to create state machines, but the vast majority today are *clocked synchronous state machines* that use edge-triggered D flip-flops. They are *clocked* because their storage elements employ a clock input; and they are *synchronous* because all of their flip-flops use the *same* clock signal. Such a state machine changes state only when a triggering edge or "tick" occurs on the clock signal. Henceforth, we'll just call them "state machines."

clocked synchronous state machine

clocked

synchronous

9.2.1 State-Machine Structure

Figure 9-4 on the next page shows the general structure of a state machine. The *state memory* is a set of n flip-flops that store the current state of the machine, and it has 2^n distinct states. The flip-flops are all connected to a common clock signal that causes them to change state at each *tick* of the clock. What constitutes a tick depends on the flip-flop type; the majority of state machines use positive-edge-triggered D flip-flops, so a tick is the rising edge of the clock signal.

state memory

tick

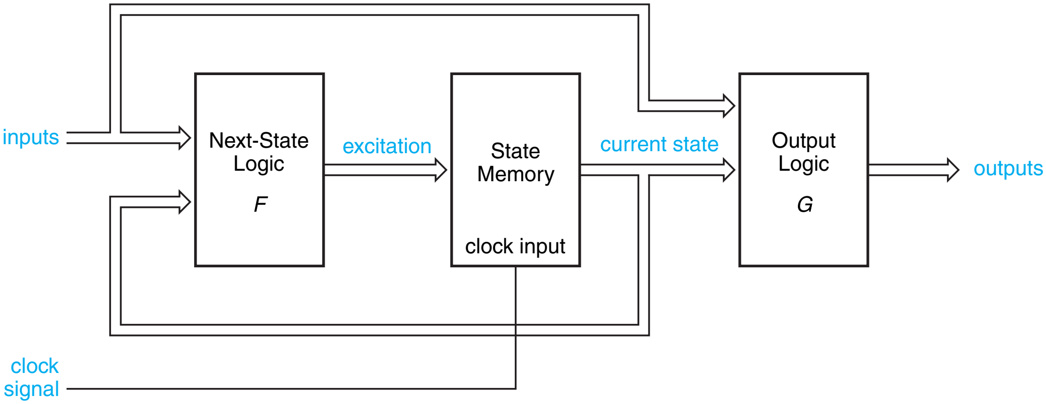


Figure 9-4 Mealy state-machine structure.

next-state logic
output logic

The next state of the state machine in Figure 9-4 is determined by the *next-state logic F* as a function of the current state and input. The *output logic G* determines the output as a function of the current state and input. Both *F* and *G* are strictly combinational logic circuits. We can write

$$\text{Next state} = F(\text{current state, input})$$
$$\text{Output} = G(\text{current state, input})$$

9.2.2 Output Logic

Mealy machine

A sequential circuit whose output depends on both state and input as shown in Figure 9-4 is called a *Mealy machine*. In some sequential circuits the output depends on the state alone:

$$\text{Output} = G(\text{current state})$$

Moore machine

Such a circuit is called a *Moore machine*, and its general structure is shown in Figure 9-5.

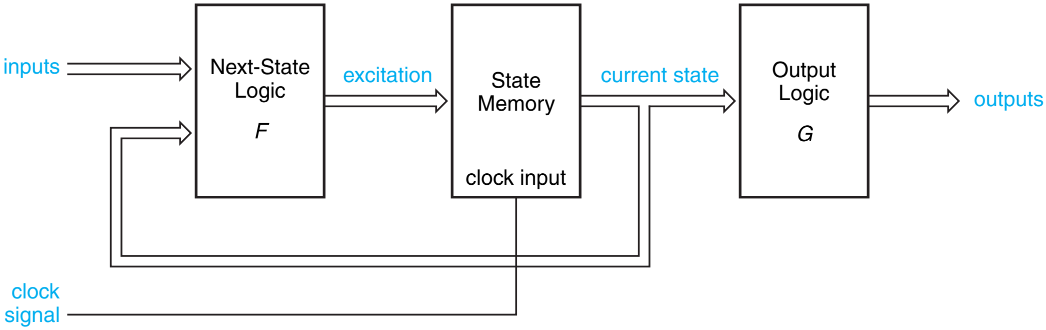


Figure 9-5 Moore state-machine structure.

Obviously, the only difference between the two state-machine models is in how outputs are generated. In practice, most state machines are categorized as Mealy machines, because they have one or more *Mealy-type outputs* that depend on input as well as state. However, many of these same machines also have one or more *Moore-type outputs* that depend only on state.

Mealy-type outputs

Moore-type outputs

In the design of high-speed circuits, it is often necessary to ensure that state-machine outputs are available as early as possible and do not change during each clock period. One way to get this behavior is to encode the state so that the state variables themselves serve as outputs. We call this an *output-coded state assignment*; it yields a Moore machine in which the output logic of Figure 9-5 is nothing more than wires.

output-coded state assignment

Another approach is to design the state machine so that the outputs during one clock period depend on the state and inputs during the *previous* clock period. We call these *pipelined outputs*, and they are obtained by attaching another stage of memory (D flip-flops) to a machine's outputs, as shown for a Mealy machine in Figure 9-6.

pipelined outputs

With appropriate circuit or drawing manipulations, you can map one state-machine model into another. For example, you could declare the flip-flops that produce pipelined outputs from a Mealy machine to be part of its state memory, and thereby obtain a Moore machine with an output-coded state assignment.

The exact classification of a state machine into one style or another is not a big deal. What's important is how you think about output structure and how it satisfies your overall design objectives, including timing and flexibility. For example, pipelined outputs are great for fast timing, but you can use them only in situations where you can figure out the desired next output value one clock period in advance. In any given application you may use different styles for different output signals. We'll see in Section 12.1.5 that different statement structures can be used to specify different output styles in Verilog.

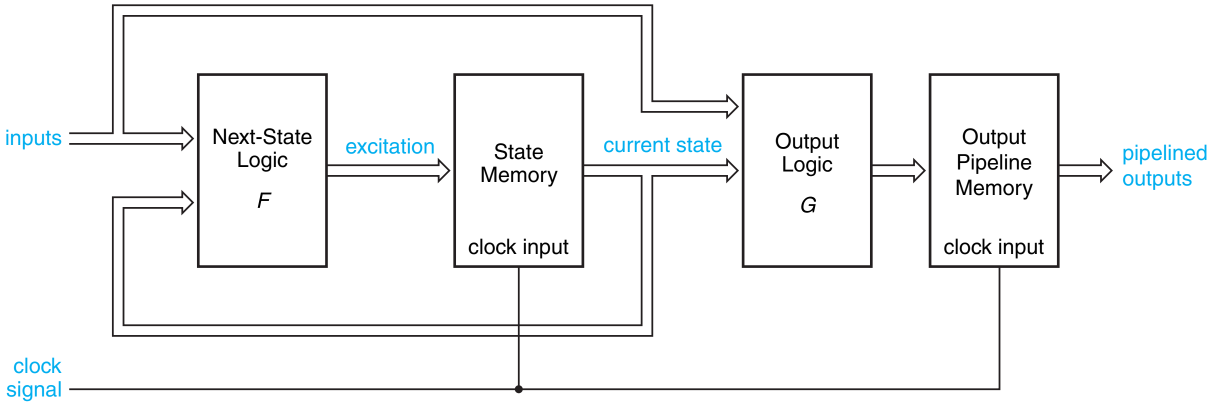


Figure 9-6 Mealy machine with pipelined outputs.

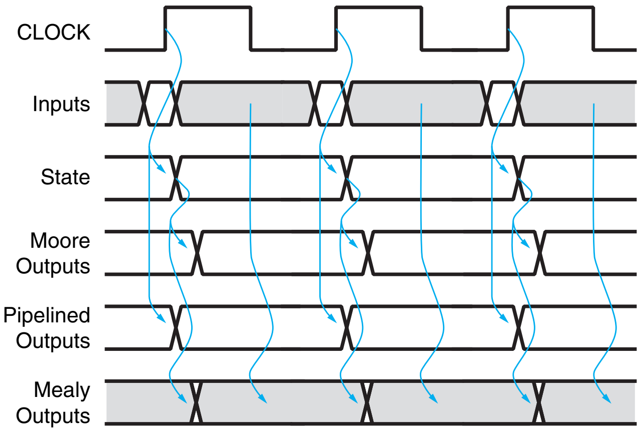


Figure 9-7
State-machine timing.

9.2.3 State-Machine Timing

Figure 9-7 shows the timing relationships among the clock, inputs, and outputs of a state machine that uses positive-edge-triggered D flip-flops. The shaded areas show where signal values may be changing, and colored arrows indicate causality, that is, which input transitions cause which output transitions. The state-machine inputs must not change during a short interval before and after the triggering clock edge; we'll have more to say about that when we look at flip-flop characteristics in detail in Section 10.2. Input changes during the rest of the clock period have no effect on the machine's state.

The state variables change just after the clock edge. Moore-type outputs, which are functions of the state only, change after that. Pipelined outputs, on the other hand, change at about the same time as state outputs, because they come directly from flip-flop outputs, usually having the same speed as the state flip-flops and of course clocked by the same clock.

Like Moore-type outputs, Mealy-type outputs change in response to changes in the state variables. Since they are also functions of the state-machine inputs, they also may change any time that the inputs change, depending on details of the output equations.

9.2.4 Analysis of State Machines with D Flip-Flops

At some point you may need to predict the behavior of a state machine based on its circuit, without the benefit of any other description. To do that, consider the formal definition of a Mealy state machine that we gave previously:

$$\text{Next state} = F(\text{current state, input})$$

$$\text{Output} = G(\text{current state, input})$$

Recalling our notion that "state" embodies all we need to know about the past history of the circuit, the first equation tells us that what we next need to know can be determined from what we currently know and the current input. The second equation tells us that the current output can be determined from the same information. The goal of sequential circuit analysis is to determine the next-state and output functions *F* and *G* so that the circuit's behavior can be predicted.

The analysis of a state machine has three basic steps:

1. Based on the logic diagram, determine the next-state and output functions *F* and *G*.
2. Use *F* and *G* to construct a *state/output table* that completely specifies the next state and output of the circuit for every possible combination of current state and input.
3. (Optional) Draw a *state diagram* that presents the information from the previous step in graphical form.

state/output table

state diagram

Figure 9-8 shows a simple state machine with two positive-edge-triggered D flip-flops. To determine the next-state function *F*, we must first consider the behavior of the state memory. At the rising edge of the clock signal, each D flip-flop samples its D input and transfers this value to its Q output. Therefore, to determine the next value of Q, which we denote as Q*, we must first determine the current value of D.

** suffix*

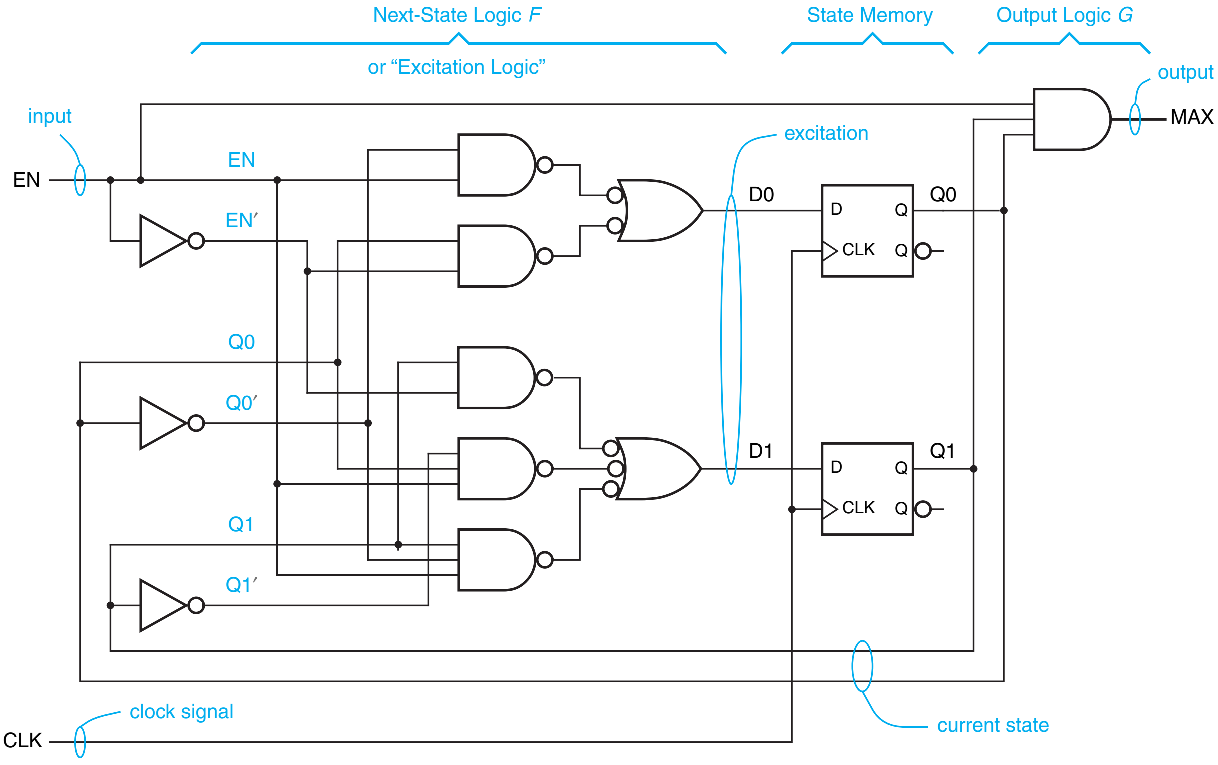


Figure 9-8 A state machine using positive-edge-triggered D flip-flops.

excitation

excitation logic

excitation equation

In Figure 9-8, there are two D flip-flops, and we have named the signals on their outputs Q0 and Q1. These two outputs are the state variables; their value is the current state of the machine. The corresponding D-input signals, D0 and D1, provide the *excitation* for the D flip-flops at each clock tick. The circuits that create these signals as functions of the current state and input are usually called *excitation logic*. They have *excitation equations* which can be derived from the logic diagram:

$$\begin{aligned} \mathsf{D0} &= \mathsf{Q0} \cdot \mathsf{EN}' + \mathsf{Q0}' \cdot \mathsf{EN} \\ \mathsf{D1} &= \mathsf{Q1} \cdot \mathsf{EN}' + \mathsf{Q1}' \cdot \mathsf{Q0} \cdot \mathsf{EN} + \mathsf{Q1} \cdot \mathsf{Q0}' \cdot \mathsf{EN} \end{aligned}$$

As noted previously, the next value of a state variable after a clock tick is denoted by appending a star to the state-variable name, for example, Q0* or Q1*. Since the value of a D flip-flop output after the clock tick is just the D input value before the tick, we can describe the next-state function of the example machine with equations for its state variables' next values:

$$\begin{aligned} \mathsf{Q0}* &= \mathsf{D0} \\ \mathsf{Q1}* &= \mathsf{D1} \end{aligned}$$

Substituting the excitation equations for D0 and D1, we can write

$$\begin{aligned} \mathsf{Q0}* &= \mathsf{Q0} \cdot \mathsf{EN}' + \mathsf{Q0}' \cdot \mathsf{EN} \\ \mathsf{Q1}* &= \mathsf{Q1} \cdot \mathsf{EN}' + \mathsf{Q1}' \cdot \mathsf{Q0} \cdot \mathsf{EN} + \mathsf{Q1} \cdot \mathsf{Q0}' \cdot \mathsf{EN} \end{aligned}$$

transition equation

These equations, which express the next value of the state variables as a function of current state and input, are called *transition equations*.

For each combination of current state and input value, the transition equations predict the next state. Each state is described by two bits, the current values of Q0 and Q1: (Q1 Q0) = 00, 01, 10, or 11. The reason for "arbitrarily" picking the order (Q1 Q0) instead of (Q0 Q1) in this example will become apparent shortly. For each state, our example machine has just two possible input values, EN = 0 or EN = 1, so there are a total of 8 state/input combinations. In general, a machine with s state bits and i inputs has 2^{s+i} state/input combinations.

transition table

Table 9-1(a) shows a *transition table* that is created by evaluating the transition equations for every possible state/input combination. Traditionally, a transition table lists the states along the left and the input combinations along the top of the table, as shown in the example.

The function of our example machine is evident from its transition table—it is a 2-bit "counter" with an enable input EN. When EN = 0, the machine maintains its current count, but when EN = 1, the binary count advances by 1 at each clock tick, rolling over to 00 when it reaches a maximum value of 11.

state names

If we wish, we may assign alphanumeric *state names* to each state. The simplest naming is 00 = A, 01 = B, 10 = C, and 11 = D. Substituting the state

(a)

	EN	
Q1 Q0	***0***	***1***
00	00	01
01	01	10
10	10	11
11	11	00
	Q1* Q0*	

(b)

	EN	
S	***0***	***1***
A	A	B
B	B	C
C	C	D
D	D	A
	S*	

(c)

	EN	
S	***0***	***1***
A	A, 0	B, 0
B	B, 0	C, 0
C	C, 0	D, 0
D	D, 0	A, 1
	S*, MAX	

Table 9-1
Transition, state, and state/output tables for the state machine in Figure 9-8.

names for combinations of Q1 and Q0 (and Q1* and Q0*) in Table 9-1(a) produces the *state table* in (b). Here "S" denotes the current state and "S*" denotes the next state of the machine. A state table is often easier to understand than a transition table, because in complex machines we can use state names that have meaning. However, a state table contains less information than a transition table because it does not include the binary values of the state variables in each named state.

state table

After creating the state table, we have only the output logic of the machine left to analyze. In the example machine there is only a single output signal, and it is a function of both current state and input (this is a Mealy machine). So we can write a single *output equation*:

output equation

$$\mathrm{MAX} = \mathrm{Q1} \cdot \mathrm{Q0} \cdot \mathrm{EN}$$

The output behavior predicted by this equation can be combined with the next-state information to produce a *state/output table* as shown in Table 9-1(c).

state/output table

State/output tables for Moore machines are slightly simpler. For example, in the circuit of Figure 9-8, suppose we removed the EN signal from the AND gate that produces the MAX output, producing a Moore-type output MAXS. Since MAXS is a function of the state only, the state/output table can list MAXS in one column, independent of the input values. This is shown in Table 9-2.

	EN		
S	***0***	***1***	***MAXS***
A	A	B	0
B	B	C	0
C	C	D	0
D	D	A	1
	S*		

Table 9-2
State/output table for a Moore machine.

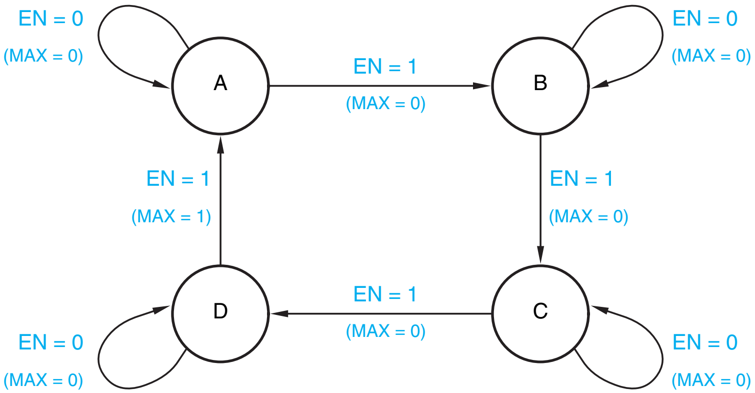


Figure 9-9 State diagram corresponding to the Mealy state machine of Table 9-1.

state diagram
node
directed arc

A *state diagram* presents the information from the state/output table in a graphical format. It has one circle (or *node*) for each state and an arrow (or *directed arc*) for each transition. Figure 9-9 shows the state diagram for our example state machine. The letter inside each circle is a state name. Each arrow leaving a given state points to the next state for a given input combination; it also shows the output value produced in the given state for that input combination.

The state diagram for a Moore machine can be somewhat simpler. In this case, the output values can be shown inside each state circle, since they are functions of state only. A state diagram using this convention for the Moore machine of Table 9-2 is shown in Figure 9-10.

The original logic diagram of our example state machine, Figure 9-8, was laid out to match our conceptual model of a Mealy machine. However, nothing requires us to group the next-state logic, state memory, and output logic in this way. Figure 9-11 shows another logic diagram for the same state machine. To analyze this circuit, the designer (or analyzer, in this case) can still extract the required information from the diagram as drawn. The only circuit difference in

LITTLE ARROWS, LITTLE ARROWS EVERYWHERE

Since there is only one input in our example Mealy machine, there are only two possible input combinations, and two arrows leaving each state. In a machine with n inputs, we would have 2^n arrows leaving each state. This is messy if n is large. Later, in Figure 9-14, we'll describe a convention whereby a state needn't have one arrow leaving it for each input combination, only one arrow for each different next state.

A CLARIFICATION

The state-diagram notation for output values in Mealy machines is a little misleading. You should remember that the listed output value is produced continuously when the machine is in the indicated state and has the input on an arrow leaving that state, not just during the transition to the next state along that arrow.

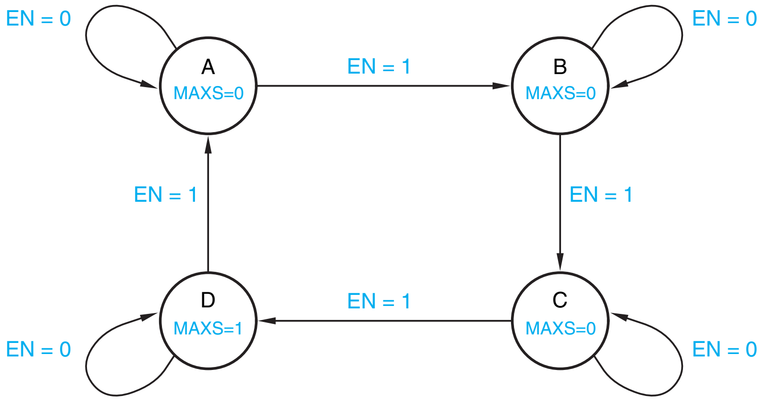


Figure 9-10 State diagram corresponding to the Moore state machine of Table 9-2.

the new diagram is that we have used the flip-flops' QN outputs (which are normally the complement of Q) to save a couple of inverters.

In summary, to analyze a state machine based on D flip-flops, the detailed steps are as follows:

1. Based on the logic diagram or other description of the excitation logic, determine the excitation equations for the flip-flop D inputs (D0, D1, etc.). *excitation equations*
2. Substitute the symbol for each state variable's next value (Q0∗, Q1∗, etc.) into the lefthand side of the corresponding excitation equation to obtain transition equations. *transition equations*
3. Use the transition equations to construct a transition table. *transition table*
4. Determine the output equations. *output equations*
5. Add output values to the transition table for each state (Moore) or state/input combination (Mealy) to create a transition/output table. *transition/output table*
6. (Optional) Name the states and substitute state names for state-variable combinations in the transition/output table to obtain a state/output table. *state names* *state/output table*
7. (Optional) Draw a state diagram corresponding to the state/output table. *state diagram*

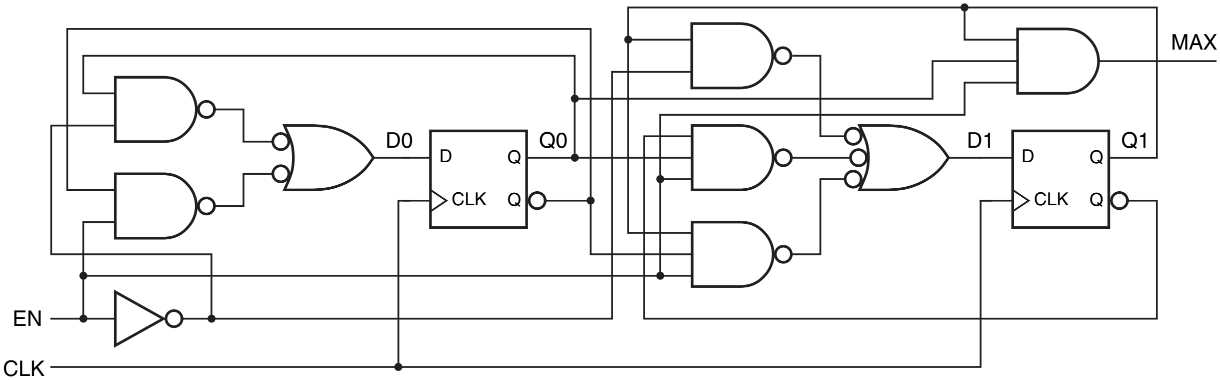


Figure 9-11 Redrawn logic diagram for a state machine.

SUGGESTIVE DRAWINGS

Using the transition, state, and output tables, we can construct a timing diagram that shows the behavior of a state machine for any desired starting state and input sequence. For example, Figure 9-12 shows the behavior of our example machine with a starting state of 00 (A) and a particular pattern on the EN input.

Notice that the value of the EN input affects the next state only at the rising edge of the CLOCK input; that is, the counter counts only if EN = 1 at the rising edge of CLOCK. On the other hand, since MAX is a Mealy-type output, its value is affected by EN at all times. If we also provide a Moore-type output MAXS as suggested in the text, its value depends only on state, as shown in the figure.

The timing diagram is drawn in a way that shows changes in the MAX and MAXS outputs occurring slightly later than the state and input changes that cause them, reflecting the combinational-logic delay of the output circuits. Naturally, the drawings are merely suggestive; precise timing is normally indicated by a timing table of the type described in Section 4.2.1.

We'll go through this complete sequence of steps to analyze another state machine, shown in Figure 9-13. Reading the logic diagram, we find that the excitation equations are as follows:

$$\begin{aligned}
\mathrm{D0} &= \mathrm{Q1}' \cdot \mathrm{X} + \mathrm{Q0} \cdot \mathrm{X}' + \mathrm{Q2} \\
\mathrm{D1} &= \mathrm{Q2}' \cdot \mathrm{Q0} \cdot \mathrm{X} + \mathrm{Q1} \cdot \mathrm{X}' + \mathrm{Q2} \cdot \mathrm{Q1} \\
\mathrm{D2} &= \mathrm{Q2} \cdot \mathrm{Q0}' + \mathrm{Q0}' \cdot \mathrm{X}' \cdot \mathrm{Y}
\end{aligned}$$

Substituting the symbol for each state variable's next value, we get the following transition equations:

$$\begin{aligned}
\mathrm{Q0}* &= \mathrm{Q1}' \cdot \mathrm{X} + \mathrm{Q0} \cdot \mathrm{X}' + \mathrm{Q2} \\
\mathrm{Q1}* &= \mathrm{Q2}' \cdot \mathrm{Q0} \cdot \mathrm{X} + \mathrm{Q1} \cdot \mathrm{X}' + \mathrm{Q2} \cdot \mathrm{Q1} \\
\mathrm{Q2}* &= \mathrm{Q2} \cdot \mathrm{Q0}' + \mathrm{Q0}' \cdot \mathrm{X}' \cdot \mathrm{Y}
\end{aligned}$$

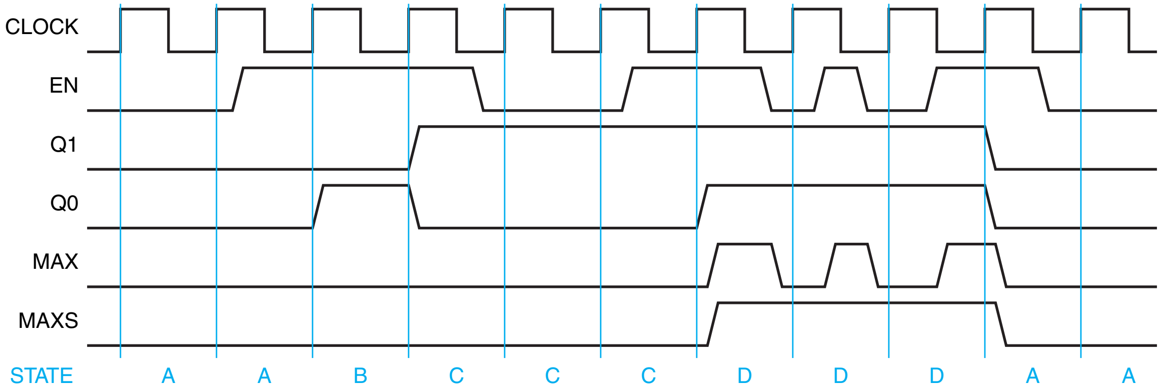


Figure 9-12 Timing diagram for example state machine.

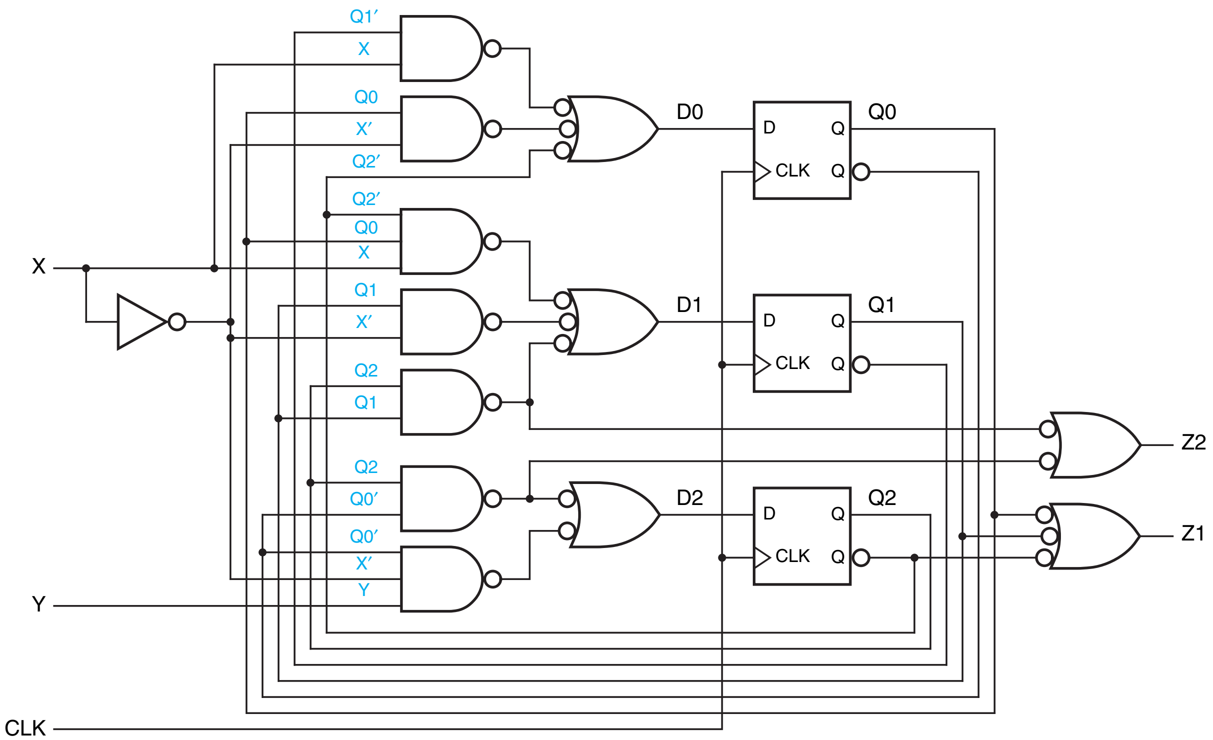


Figure 9-13 A state machine with three flip-flops and eight states.

A transition table based on these equations is shown in Table 9-3(a). Reading the logic diagram, we can write two output equations:

$$Z1 = Q2 + Q1' + Q0'$$

$$Z2 = Q2 \cdot Q1 + Q2 \cdot Q0'$$

(a)

	X Y				
Q2 Q1 Q0	***00***	***01***	***10***	***11***	***Z1 Z2***
000	000	100	001	001	10
001	001	001	011	011	10
010	010	110	000	000	10
011	011	011	010	010	00
100	101	101	101	101	11
101	001	001	001	001	10
110	111	111	111	111	11
111	011	011	011	011	11
	Q2* Q1* Q0*				

(b)

	X Y				
S	***00***	***01***	***10***	***11***	***Z1 Z2***
A	A	E	B	B	10
B	B	B	D	D	10
C	C	G	A	A	10
D	D	D	C	C	00
E	F	F	F	F	11
F	B	B	B	B	10
G	H	H	H	H	11
H	D	D	D	D	11
	S*				

Table 9-3 Transition/output and state/output tables for the state machine in Figure 9-13.

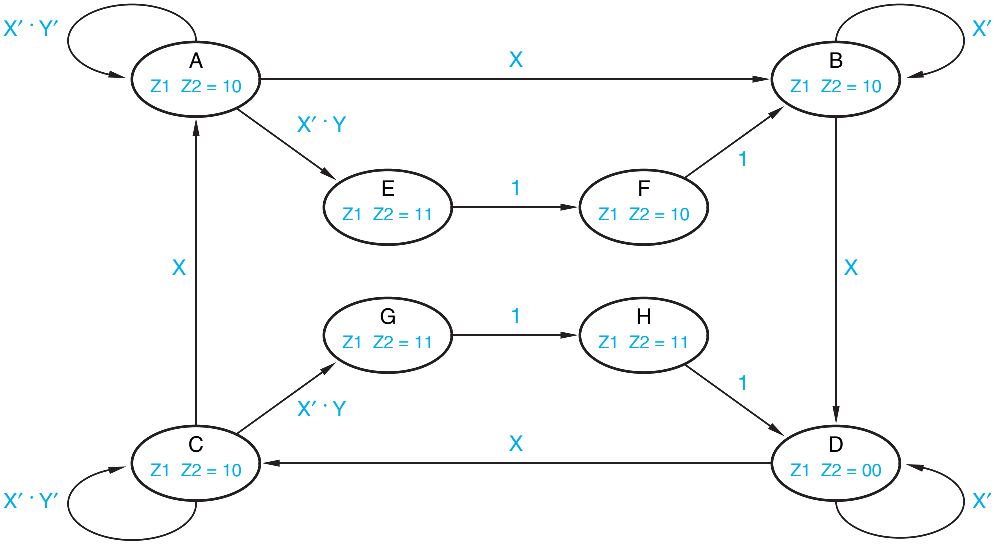


Figure 9-14 State diagram corresponding to Table 9-3.

The resulting output values are shown in the last column of Table 9-3(a). After assigning state names A–H, we obtain the state/output table shown in (b).

A state diagram for the example machine is shown in Figure 9-14. Since our example is a Moore machine, the output values are written with each state. This example introduces another, more efficient way of labeling transitions in a state machine that has multiple inputs. Instead of drawing an arc for each transition in the state table (there are 32 of them in Table 9-3), we draw an arc for each unique pair of starting and ending states. Each arc is labeled with a *transition expression*; a transition is taken for input combinations for which the transition expression is 1.

transition expression

So, how did we come up with the transition expressions in Figure 9-14? Starting with the state table, we write the transition expression for a particular current state and next state as a sum of minterms corresponding to the input combinations that cause that transition. If desired, the expression can then be minimized to give the information in a more compact form. For example, there are three transitions out of state A:

$$\begin{array}{lll} \mathrm{A} \longrightarrow \mathrm{A}: & \mathrm{X\ Y} = 00 & \mathrm{X}' \cdot \mathrm{Y}' \\ \mathrm{A} \longrightarrow \mathrm{E}: & \mathrm{X\ Y} = 01 & \mathrm{X}' \cdot \mathrm{Y} \\ \mathrm{A} \longrightarrow \mathrm{B}: & \mathrm{X\ Y} = 10{,}11 & \mathrm{X} \cdot \mathrm{Y}' + \mathrm{X} \cdot \mathrm{Y} = \mathrm{X} \end{array}$$

Note that if *all* of the transitions leaving a particular state go to the same next state, then the sum of minterms after minimization will be logic 1. Transitions labeled “1” are *always* taken, of course.

9.3 State-Machine Design with State Tables

Aside from planning the overall architecture of a digital system, designing state machines is probably the most creative task of a digital designer. There are a few different ways to design a state machine, including writing its description from the outset using an HDL like Verilog. However, the traditional way is to start from an informal word description or specification, and then proceed to a state table or a state diagram, performing just about the reverse of the analysis steps that we used in the preceding section:

1. Construct a state/output table corresponding to the word description or specification, using mnemonic names for the states. (It's also possible to start with a state diagram or an ASM chart; these methods will be discussed in Sections 9.4 and 9.5.) *state/output table*
2. (Optional) Minimize the number of states in the state/output table. *state minimization*
3. Choose a set of state variables and assign state-variable combinations to the named states. *state assignment*
4. Substitute the state-variable combinations into the state/output table to create a transition/output table that shows the desired next state-variable combination and output for each state/input combination. *transition/output table*
5. Choose a flip-flop type for the state memory. In today's implementation technologies, there is almost never a choice—it is almost always an edge-triggered D flip-flop, and that's a good choice.
6. Construct an excitation table that shows the excitation values required to obtain the desired next state for each state/input combination. *excitation table*
7. Derive excitation equations from the excitation table. *excitation equations*
8. Derive output equations from the transition/output table. *output equations*
9. Draw a logic diagram that shows the state-variable storage elements and realizes the required excitation and output equations. *logic diagram*

In this section, we'll describe each of these basic steps in traditional state-machine design. Step 1 is the most important, since it is here that the designer really *designs*, going through the creative process of translating a (perhaps ambiguous) English-language description of the state machine into a formal tabular description. Step 2 is hardly ever performed by experienced digital designers, but designers bring much of their experience to bear in step 3. *design*

Once the first three steps are completed, all of the remaining steps can be completed by "turning the crank," that is, by following a well-defined synthesis procedure. Steps 4 and 6–9 are the most tedious, but they are automated when you design state machines using an HDL. Still, it's useful for you to understand the traditional synthesis procedure, both to give you an appreciation of the HDL compiler's function and to give you a chance of figuring out what's really going on when the compiler produces unexpected results. Therefore, all nine steps of the traditional state-machine design procedure will be discussed in the rest of this section.

9.3.1 State-Table Design Example

There are several different ways to describe a state machine's state table. Later, we'll see how Verilog can be used to specify a state table indirectly. In this section, however, we deal only with state tables that are specified directly, in the same tabular format that we used in the previous section for analysis.

We'll present the state-table design process here as well as the synthesis procedure in later subsections, using the simple design problem below:

> Design a state machine with two inputs, A and B, and a single output Z that is 1 if:
>
> - A had the same value at each of the two previous clock ticks, *or*
> - B has been 1 since the last time that the first condition was true.
>
> Otherwise, the output should be 0.

If the meaning of this specification isn't crystal clear to you at this point, don't worry. Part of your job as a designer is to convert an informal specification into a state table (or HDL equivalent) that is absolutely unambiguous. Even if the state table doesn't match what was originally intended, it at least forms a basis for further discussion and refinement of the design. Or, during the development of the state table, you may discover that the original problem statement is ambiguous or just plain wrong and must be adjusted.

As an additional "hint" or requirement, state-table design problems often include timing diagrams that show the state machine's expected behavior for one or more sequences of inputs. Such a timing diagram is unlikely to specify unambiguously the machine's behavior for all possible sequences of inputs, but it's a good starting point for discussion and a benchmark against which proposed designs can be checked. Figure 9-15 is such a timing diagram for our example state-table design problem.

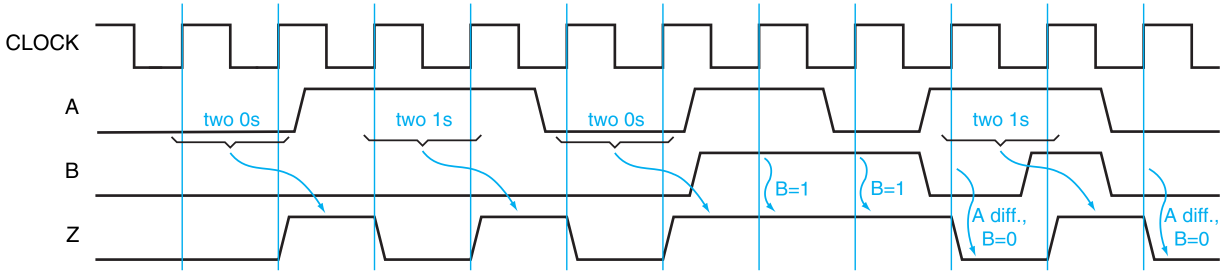


Figure 9-15 Timing diagram for example state machine.

STATE-MACHINE DESIGN AS A KIND OF PROGRAMMING

Designing a state machine (using a state table, a state diagram, an ASM chart, or an HDL) is a creative process that is like writing a computer program in many ways:

- You start with a fairly precise description of inputs and outputs, but a possibly ambiguous description of the desired relationship between them, and usually no clue about how to actually obtain the desired outputs from the inputs.
- During the design you may have to identify and choose among different ways of doing things, sometimes using common sense, and sometimes arbitrarily.
- You may have to identify and handle special cases that weren't included in the original description.
- You will probably have to keep track of several ideas in your head during the design process.
- Since the design process is not an algorithm, there's no guarantee that you can complete the state table or program using a finite number of states or lines of code. However, unless you work for the government, you must try to do so.
- When you finally run the state machine or program, it will do exactly what you told it to do—no more, no less.
- There's no guarantee that the thing will work the first time; you may have to debug it and iterate on the whole process.
- HDL models that specify state machines look a lot like computer programs, but like other HDL models, they're not!

Although state-machine design is a challenge, there's no need to be intimidated. If you've made it this far in your education, then you've written a few computer programs that worked, and you can become just as good at designing state machines.

The first step in the state-table design is to construct a template. From the word description, we know that our example is a Moore machine—its output depends only on the current state, that is, what happened in previous clock periods. Thus, as shown in Figure 9-16(a), we provide one next-state column for each possible input combination, and a single column for the output values. The order in which the input combinations are written doesn't affect this part of the process, but we've written them in Gray-code order, where only one input value changes between successive columns. That's out of habit from previous editions of this book, where that order (the same one used in Karnaugh maps) simplified the tedious process of deriving excitation equations by hand.

In a Mealy machine, we would omit the output column and write the output values along with the next-state values under each input combination. The leftmost column is simply an English-language reminder of the meaning of each state or the "history" associated with it.

(a)

		A B				
Meaning	S	00	01	11	10	Z
Initial state	INIT					0
	. . .					
	. . .					
	. . .					
		S*				

(b)

		A B				
Meaning	S	00	01	11	10	Z
Initial state	INIT	A0	A0	A1	A1	0
Got a 0 on A	A0					0
Got a 1 on A	A1					0
		S*				

(c)

		A B				
Meaning	S	00	01	11	10	Z
Initial state	INIT	A0	A0	A1	A1	0
Got a 0 on A	A0	OK	OK	A1	A1	0
Got a 1 on A	A1					0
Got two equal A inputs	OK					1
		S*				

(d)

		A B				
Meaning	S	00	01	11	10	Z
Initial state	INIT	A0	A0	A1	A1	0
Got a 0 on A	A0	OK	OK	A1	A1	0
Got a 1 on A	A1	A0	A0	OK	OK	0
Got two equal A inputs	OK					1
		S*				

Figure 9-16 Evolution of a state table.

initial state

The word description isn't specific about what happens when this machine is first started, so we'll just have to improvise. We'll assume that when power is first applied to the system, the machine enters an *initial state*, called INIT in this example. We write the name of the initial state (INIT) in the first row and leave room for enough rows (states) to complete the design. We can also fill in the value of Z for the INIT state; common sense says it should be 0 because there were *no* inputs beforehand.

Next, we must fill in the next-state entries for the INIT row. The Z output can't be 1 until we've had at least two clock ticks and seen input values on A at least twice, so we'll provide two states, A0 and A1, that "remember" the value of A on the previous clock tick, as shown in Figure 9-16(b). In both of these states Z is 0, since we haven't satisfied the conditions for a 1 output yet. The precise meaning of state A0 is "Got A = 0 on the previous tick, A ≠ 0 on the tick before that, and B ≠ 1 at some time since the previous pair of equal A inputs." State A1 is defined similarly. At this point we know that our state machine has at least three states, and we have created two more blank rows to fill in. Hmmmm, this isn't such a good trend! In order to fill in the next-state entries for *one* state (INIT), we had to create *two* new states A0 and A1. If we kept going this way, we could end up with 4,097 states by bedtime! Instead, we should be on the lookout for existing states that have the same meaning as new ones that we might otherwise create. Let's see how it goes.

In state A0, we know that input A was 0 at the previous clock tick. Therefore, if A is 0 again, we go to a new state OK with Z = 1, as shown in Figure 9-16(c). If A is 1, then we don't have two equal inputs in a row, so we go

(a)

		A B				
Meaning	S	00	01	11	10	Z
Initial state	INIT	A0	A0	A1	A1	0
Got a 0 on A	A0	OK	OK	A1	A1	0
Got a 1 on A	A1	A0	A0	OK	OK	0
Got two equal A inputs	OK	?	OK	OK	?	1
		S*				

(b)

		A B				
Meaning	S	00	01	11	10	Z
Initial state	INIT	A0	A0	A1	A1	0
Got a 0 on A	A0	OK0	OK0	A1	A1	0
Got a 1 on A	A1	A0	A0	OK1	OK1	0
Two equal, A=0 last	OK0					1
Two equal, A=1 last	OK1					1
		S*				

(c)

		A B				
Meaning	S	00	01	11	10	Z
Initial state	INIT	A0	A0	A1	A1	0
Got a 0 on A	A0	OK0	OK0	A1	A1	0
Got a 1 on A	A1	A0	A0	OK1	OK1	0
Two equal, A=0 last	OK0	OK0	OK0	OK1	A1	1
Two equal, A=1 last	OK1					1
		S*				

(d)

		A B				
Meaning	S	00	01	11	10	Z
Initial state	INIT	A0	A0	A1	A1	0
Got a 0 on A	A0	OK0	OK0	A1	A1	0
Got a 1 on A	A1	A0	A0	OK1	OK1	0
Two equal, A=0 last	OK0	OK0	OK0	OK1	A1	1
Two equal, A=1 last	OK1	A0	OK0	OK1	OK1	1
		S*				

Figure 9-17 Continued evolution of a state table.

to state A1 to remember that we just got a 1. Likewise in state A1, shown in (d), we go to OK if we get a second 1 input in a row, or to A0 if we get a 0.

Once we get into the OK state, the machine description tells us we can stay there as long as B = 1, irrespective of the A input, as shown in Figure 9-17(a). If B = 0, we have to look for two 1s or two 0s in a row on A again. However, we've got a little problem in this case. The current A input may or may not be the second equal input in a row, so we may still be "OK" or we may have to go back to A0 or A1. We defined the OK state too broadly—it doesn't "remember" enough to tell us which way to go.

The problem is solved in Figure 9-17(b) by splitting OK into two states, OK0 and OK1, that "remember" the previous A input. Now all of the next states for OK0 and OK1 can be selected from existing states, as shown in (c) and (d). For example, if we get A = 0 in OK0, we can just stay in OK0; we don't have to create a new state that "remembers" three 0s in a row, because the machine's description doesn't require us to distinguish that case. Thus, we have achieved "closure" of the state table, which now describes a *finite*-state machine. As a sanity check, Figure 9-18 repeats the timing diagram of Figure 9-15, listing the states that should be visited according to our final state table.

INITIAL VERSUS IDLE STATES

The example state machine that we've been designing visits its initial state only during reset. Many machines are designed instead with an "idle" state that is entered both at reset and whenever the machine has nothing in particular to do.

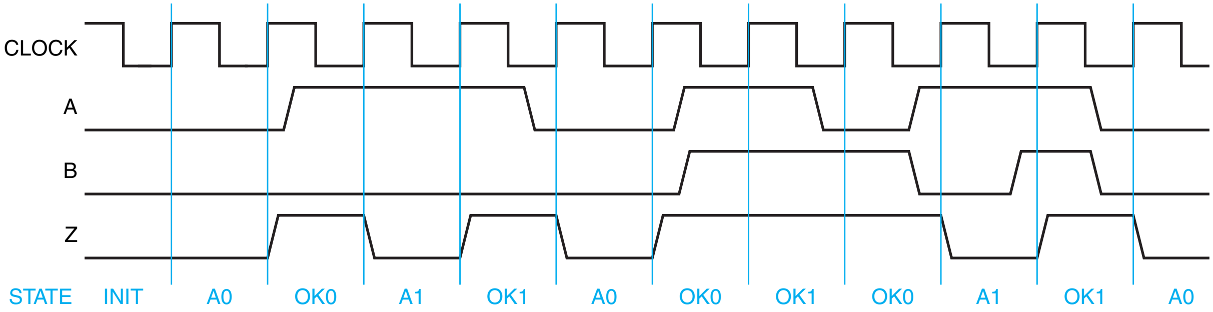


Figure 9-18 Timing diagram and state sequence for example state machine.

REALIZING RELIABLE RESET

For proper system operation, the hardware design of a state machine should ensure that it enters a known initial state on power-up, such as the INIT state in our design example. Most systems have a RESET signal that is asserted during power-up.

With increasing levels of integration, reset circuits have gotten more sophisticated over the years, and are often called “voltage supervisors.” During power-up, such a circuit detects the power supply reaching a threshold close to its full voltage (say, 3.0 V in a 3.3-V system), and follows that with a delay (say, 200 ms) to ensure that all components have had time to stabilize before it “unresets” the system. The circuit also detects the voltage falling below the threshold voltage and resets the system immediately if that happens.

Besides power-supply voltage detection, a typical voltage supervisor also has an input for a manual reset button, and a logic input for a “watchdog timer.” Used in more complex systems, the watchdog timer resets the system if software or other logic does not periodically change the signal value on the watchdog input.

*9.3.2 State Minimization

Figure 9-17(d) is a “minimal” state table for our original word description, in the sense that it contains the fewest possible states. However, Figure 9-19 shows other state tables, with more states, that also do the job. Formal procedures can be used to minimize the number of states in such tables. If enough states are eliminated, fewer state variables may be needed (e.g., going from nine states to eight or less reduces the number of state flip-flops from four to three).

equivalent states

The basic idea of formal minimization procedures is to identify *equivalent states*, where two states are equivalent if it is impossible to distinguish them by observing only the current and future *outputs* of the machine (and *not* the internal state variables). A pair of equivalent states can be replaced by a single state.

Two states S1 and S2 are equivalent if and only if two conditions are true. First, S1 and S2 must produce the same values at the state-machine output(s); in

*Throughout this book, optional sections are marked with an asterisk.

(a)

		A B				
Meaning	S	00	01	11	10	Z
Initial state	INIT	A0	A0	A1	A1	0
Got a 0 on A	A0	OK00	OK00	A1	A1	0
Got a 1 on A	A1	A0	A0	OK11	OK11	0
Got 00 on A	OK00	OK00	OK00	OKA1	A1	1
Got 11 on A	OK11	A0	OKA0	OK11	OK11	1
OK, got a 0 on A	OKA0	OK00	OK00	OKA1	A1	1
OK, got a 1 on A	OKA1	A0	OKA0	OK11	OK11	1
		S*				

(b)

		A B				
Meaning	S	00	01	11	10	Z
Initial state	INIT	A0	A0	A1	A1	0
Got a 0 on A	A0	OK00	OK00	A1	A1	0
Got a 1 on A	A1	A0	A0	OK11	OK11	0
Got 00 on A	OK00	OK00	OK00	A001	A1	1
Got 11 on A	OK11	A0	A110	OK11	OK11	1
Got 001 on A, B=1	A001	A0	AE10	OK11	OK11	1
Got 110 on A, B=1	A110	OK00	OK00	AE01	A1	1
Got bb...10 on A, B=1	AE10	OK00	OK00	AE01	A1	1
Got bb...01 on A, B=1	AE01	A0	AE10	OK11	OK11	1
		S*				

Figure 9-19 Nonminimal state tables equivalent to Figure 9-17(d).

a Mealy machine, this must be true for all input combinations. Second, for each input combination, S1 and S2 must have either the same next state or equivalent next states.

Thus, a formal state-minimization procedure can shows that state OK00 and OKA0 in Figure 9-19(a) are equivalent because they produce the same output and their next-state entries are identical. Since the states are equivalent, state OK00 may be eliminated and its occurrences in the table replaced by OKA0, or vice versa. Likewise, states OK11 and OKA1 are equivalent.

To minimize the state table in Figure 9-19(b), a formal procedure must use a bit of circular reasoning. States OK00, A110, and AE10 all produce the same output and have almost identical next-state entries, so they might be equivalent. They are equivalent only if A001 and AE01 are equivalent. Similarly, OK11, A001, and AE01 are equivalent only if A110 and AE10 are equivalent. In other words, the states in the first set are equivalent if the states in the second set are, and vice versa. So, let's just go ahead and say they're equivalent.

But is state minimization really necessary? Almost always, no. Unless minimization lowers the number of states enough to reduce the number of bits needed to encode them, it doesn't even save a flip-flop. The excitation equations may or may not be simpler, and that is irrelevant anyway for some implementation technologies. For example, there is no savings in an FPGA implementation if the number of state *variables* is not reduced, since a LUT's ability to realize a logic equation depends only on the number of logic variables (primary inputs plus state variables in an excitation equation), not the number of product terms.

By carefully matching state meanings to the requirements of the problem, experienced digital designers produce state tables for small problems with a minimal or near-minimal number of states, without ever using a formal minimization procedure. Also, there are situations where *increasing* the number of states or state variables may simplify the design, reduce its cost, or increase its performance, so even an automated state-minimization procedure doesn't necessarily help. A designer can do more to improve a state machine during the state-assignment phase of the design, discussed in the next subsection.

Table 9-4
State and output table for example problem.

	A B				
S	***00***	***01***	***11***	***10***	***Z***
INIT	A0	A0	A1	A1	0
A0	OK0	OK0	A1	A1	0
A1	A0	A0	OK1	OK1	0
OK0	OK0	OK0	OK1	A1	1
OK1	A0	OK0	OK1	OK1	1
	S*				

9.3.3 State Assignment

coded state
total number of states

The next step in the design process is to determine how many binary variables are required to represent the states in the state table, and to assign a specific combination to each named state. We'll call the binary combination assigned to a particular state a *coded state*. The *total number of states* in a machine with n flip-flops is 2^n, so the number of flip-flops needed to code s states is $\lceil \log_2 s \rceil$, the smallest integer greater than or equal to $\log_2 s$.

unused states

For reference, the state/output table of our example machine is repeated in Table 9-4. It has five states, so it requires three flip-flops. Of course, three flip-flops provide a total of eight states, so there will be $8-5=3$ *unused states*. We'll discuss alternatives for handling the unused states at the end of this subsection. Right now, we have to deal with lots of choices for the five coded states, a few of which are shown in Table 9-5.

The simplest assignment of s coded states to 2^n possible states is to use the first s binary integers in binary counting order, as shown in the first assignment column of the table. This is often a good choice, and certainly in the following circumstances:

- You are using an HDL to design the state machine, and you want to get *something* specified so you can test the machine's functional behavior in simulation.

Table 9-5
Some possible state assignments for the state machine in Table 9-4.

	Assignment			
State Name	***Simplest Q1–Q3***	***Decomposed Q1–Q3***	***One-Hot Q1–Q5***	***Almost One-Hot Q1–Q4***
INIT	000	000	00001	0000
A0	001	100	00010	0001
A1	010	101	00100	0010
OK0	011	110	01000	0100
OK1	100	111	10000	1000

COMBINATORIAL MATH

The number of different ways to choose m coded states out of a set of n possible states is given by a *binomial coefficient*, denoted $\binom{n}{m}$, whose value is $\frac{n!}{m! \cdot (n-m)!}$. (We used binomial coefficients previously in Section 2.10 in the context of decimal coding.) In our example, there are $\binom{8}{5}$ different ways to choose five coded states out of eight possible states, and 5! ways to assign the five named states to each different choice. So there are $\frac{8!}{5! \cdot 3!} \cdot 5!$ or 6720 different ways to assign the five states of our example machine to combinations of three binary state variables. We don't have time to look at all of them.

- Only a single instance of this state machine will be used in your design, so it is not critical to minimize its implementation cost.
- The timing performance of the state-machine (e.g, clock-to-output time, maximum clock frequency, etc.) is not critical to system performance.
- Glitch-free decoding of the current state is not required, so it's OK for multiple state variables to change on state transitions.
- In debugging, either in simulation or in actual hardware, it is not necessary to be able to determine the current state by looking at just one signal.

However, the simplest state assignment does not always lead to the simplest excitation equations, output equations, and resulting logic circuit, which also may not be the most convenient to debug. In fact, the state assignment may have a substantial effect on the state-machine circuit's cost and performance, and it may also affect the cost and convenience of using the state machine to interact with other system elements, because of the encoding and timing of its outputs.

So, how do we choose the best state assignment for a given problem? In general, the only formal way to find the *best* assignment is to try *all* the assignments. That's too much work, even for students. Instead, most digital designers rely on experience and several practical guidelines for making reasonable state assignments:

- Choose a coded state into which the machine can easily be forced at initialization (often 00. . . 00 or 11. . . 11), typically by asserting a dedicated "reset" input for one or more clock ticks.
- Minimize the number of state variables that change on each transition.
- Maximize the number of state variables that don't change in a group of related states (i.e., a group in which most transitions stay in the group).

- Exploit symmetries in the problem specification and the corresponding symmetries in the state table. That is, suppose that one state or group of states means almost the same thing as another. Once an assignment has been established for the first, a similar assignment, differing only in one bit, can be used for the second.
- If there are unused states (i.e., if $s < 2^n$ where $n = \lceil \log_2 s \rceil$), then choose the "best" of the available state-variable combinations to achieve the foregoing goals. That is, don't limit the choice of coded states to the first s n-bit integers.

decomposed state assignment

- *Decompose* the set of state variables into individual bits or fields, where each bit or field has a well-defined meaning with respect to the input effects or output behavior of the machine.
- Consider using more than the minimum number of state variables to make a decomposed assignment possible.

Some of these ideas are incorporated in the decomposed state assignment in Table 9-5. As before, the INIT state is 000, which is easy to force either asynchronously (applying the RESET signal to the flip-flop CLR inputs) or synchronously (by ANDing RESET′ with all of the D flip-flop inputs). In a typical FPGA- or PLD-based implementation, one or both of these options may be available more or less "for free." The assignment uses one bit, Q1, to indicate whether or not the machine has left the INIT state. When Q1 is 1, Q2 and Q3 distinguish among the four non-INIT states.

The non-INIT states in the "decomposed" column of Table 9-5 appear to have been assigned in binary counting order, but that's just a coincidence. State bits Q2 and Q3 actually have individual meanings in the context of the state machine's inputs and output. Q3 gives the previous value of A, and Q2 indicates that the conditions for a 1 output are satisfied in the current state. By decomposing the state-bit meanings in this way, we can expect the next-state and output logic to be simpler than in a "random" assignment of Q2,Q3 combinations to the non-INIT states. We'll continue the state-machine design based on this assignment in the next subsection.

Sometimes the current state of a machine needs to be decoded for use in a larger circuit, and in some cases the decoded output needs to be "glitch free"—for example, if it is applied to the asynchronous input of a flip-flop,or if it is used with a different clock. If multiple state variables change on a state transition, then glitch-free decoding may not be possible. For example, in the "simplest" state assignment in Table 9-5, a transition between states A0 (001) and A1 (010) may briefly look like state OK0 (011) or INIT (000), depending on the Q2 and Q3 flip-flops' output timing (e.g., if 0-to-1 transitions have timing different from 1-to-0). Thus, a 3-input AND gate that decodes the OK0 or the INIT state may produce a short glitch during an A0–A1 transition.

Glitch-free decoding is possible if one state variable changes on each state transition. State assignments that provide this property are sometimes called *Gray assignments*, after the Gray codes which have a similar property. A potential Gray assignment for a given state table can be analyzed by means of a *state adjacency diagram*, a simplified state diagram that omits self-loops and does not show the direction of other transitions (A→B is drawn the same as B→A) or the input combinations that cause them. The adjacency diagram for our example state machine (see Table 9-4 on page 462) is shown in Figure 9-20(a). For glitch-free decoding we would like the state assignments for each adjacent pair of states to differ in only one bit.

Gray assignment

state adjacency diagram

As it turns out, quite fortuitously in this example, the "decomposed" state assignment in Table 9-5 has the desired property for all of the "main" states—all except INIT—as shown in Figure 9-20(b). And that's really the best we can do for this particular adjacency diagram. It's a bit of a brain teaser, but you should be able to convince yourself at least by trial and error that there's no way to assign coded states to INIT, A0, and A1 loop, or any loop with an odd number of states for that matter, so that all state pairs differ in only one bit. In this example, state A0 may be briefly decoded on the transition from INIT to A1. For success in general, we must fit the nodes and arcs of the adjacency diagram onto corresponding nodes and arcs of an *n*-cube (Figure 2-8 on page 67), without any gaps.

Fortunately, there's another, simpler solution for glitch-free state decoding that works for any state machine, namely to use a *one-hot assignment* as shown in Table 9-5. This assignment may use a lot more than the minimum number of state variables, since it uses one bit per state, but state decoding is trivial, of course. In addition to being simple, a one-hot assignment has the advantage of usually leading to small excitation equations, since each flip-flop must be set to 1 for transitions into only one state. It's also a convenient state assignment for debugging, because you can determine when the machine has entered a particular state by looking at just one signal.

one-hot assignment

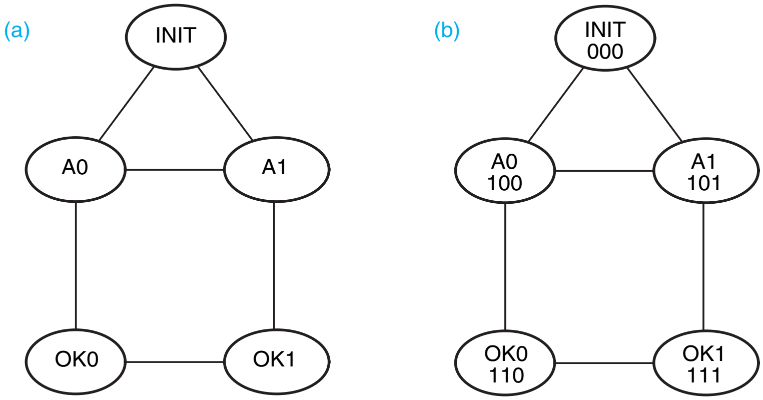


Figure 9-20 Adjacency diagram for the state table of Table 9-4.

GETTING EVEN

We said that it's not possible to encode the states with only a single-bit change on every transition in a loop with an odd number of states if the length of the loop is odd. However, if the length is even, it's always possible to achieve this goal by starting with a pure Gray code of length 2^n that's at least as great as the required length, and then repeatedly removing pairs of code words until achieving the required length.

Since Gray code is a "reflected" code, we can remove the pair of code words immediately above and below the "reflection line" halfway through the list of code words, and the newly adjacent pair of code words will still differ in only one bit (the MSB). You can see this in Table 2-8 on page 62. Alternatively, you can remove the first and last states in the list of code words. In either case, you can repeat the process until you reach the desired even number of code words.

An obvious disadvantage of a one-hot assignment, especially for machines with many states, is that it requires (a lot) more than the minimum number of flip-flops. However, if timing performance is critical, and some other part of your system needs to know as soon as possible that a particular state has been entered, this encoding is ideal. No additional combinational logic is needed to decode that particular state; the needed signal is available immediately after the triggering clock edge on the state variable's flip-flop output.

The last column of Table 9-5 is an "almost one-hot assignment" that uses the "none-hot" combination for the initial state. This is useful for two reasons: it's easy to initialize most storage devices to the all-0s state, and the initial state in this machine is never revisited once the machine gets going. Completing the state-machine design using this state assignment is considered in Exercise 9.22.

unused states

Now let's consider the disposition of *unused states* when the number of states available with n flip-flops, 2^n, exceeds the number of states required, s. There are two reasonable approaches, depending on the design requirements:

- *Minimal risk.* This approach assumes that it is possible for the state machine somehow to get into one of the unused (or "illegal") states, perhaps because of a hardware failure, an unexpected input, or a design error. Therefore, all of the unused state-variable combinations are identified and explicit next-state entries are made so that, for any input combination, the unused states go to the "initial" state, the "idle" state, an explicitly named "error" state, or some other "safe" state. This is an automatic consequence of some design methodologies if the "safe" state is coded 00. . . 00.
- *Minimal cost.* This approach assumes that the machine will never enter an unused state. Therefore, in the transition and excitation tables, the next-state entries of the unused states can be marked as "don't-cares." In most cases this simplifies the excitation logic. However, the machine's behavior if it ever does enter an unused state may be pretty weird.

	A B				
Q1 Q2 Q3	**00**	**01**	**11**	**10**	**Z**
000	100	100	101	101	0
100	110	110	101	101	0
101	100	100	111	111	0
110	110	110	111	101	1
111	100	110	111	111	1
	Q1* Q2* Q3* or D1 D2 D3				

Table 9-6 Transition/excitation and output table for example problem.

Given the low cost of excitation logic in modern implementation technologies, it makes sense to prefer the minimal-risk approach, which may also reduce engineering design time (because, for example, Verilog does not provide a convenient way to specify "don't-cares" in excitation logic). The only really compelling reason to go for minimum cost would be if we were designing a chip that contains a great many physical instances of the state machine.

*9.3.4 Synthesis Using D Flip-Flops

Once we've assigned coded states to the named states of a machine, the rest of the design process is pretty much "turning the crank." In this section, we'll show you a table-based method for doing this for small state machines, using the state table as a starting point. But it's far more convenient to design state machines directly in an HDL, avoiding the error-prone process of working with tables filled with 0s and 1s. We'll show you a Verilog method in Section 9.6.

Once we have a state table and have selected a state encoding, the next step is to substitute coded states for named states in the state table, thereby obtaining a *transition table*. The transition table shows the next coded state for each combination of current coded state and input. Table 9-6 shows the transition and output table that is obtained from the example state machine of Table 9-4 on page 462 using the "decomposed" assignment of Table 9-5.

transition table

Since our state memory uses D flip-flops, the values in the transition table—the next values of the coded-state variables—are also the "excitation" values that must be applied to their D inputs to go from each starting state to each next state. We've reflected that fact in the alternate naming for the table's entries, shown at the bottom of the table, and we can call it a *transition/excitation table*.

transition/excitation table

The transition/excitation table is like a multiple-output truth table, in this example for three combinational logic functions (D1, D2, D3) of five variables (A, B, Q1, Q2, Q3). We now have a tabular description of the excitation logic that we can hook up with the state memory (D flip-flops) to implement the state machine. However, unless all 2^n possible combinations of the state variables appear as rows in the transition/excitation table, these logic functions are incompletely specified. At this point, we must decide whether to do a minimal-risk or a minimal-cost disposition of unused states:

- For a minimal-risk disposition, we pick a sensible default destination state, like the reset or idle state, for all transitions out of unused states, and use that as appropriate when synthesizing the excitation logic.
- For a minimal-cost disposition, we use "don't-cares" for all transitions out of unused states when synthesizing the excitation logic.

Depending on the implementation technology and the design environment, there are at least two ways we could implement these logic functions:

1. For a gate-level design, whether in an ASIC or in a PLA or PLD, we can derive a minimal two-level sum-of-products or product-of-sums expression for each function and implement the corresponding circuit (AND-OR, NAND-NAND, etc.).
2. For an FPGA-based design, we can transfer the transition/excitation table into a corresponding set of LUTs.

That brings us to the question of how we can convert a multiple-output truth table like Table 9-6 to equations in the first method above, or to transfer it into LUTs in the second. For small problems in the "bad old days," this was sometimes done by hand using Karnaugh maps (as you can see for this very example in Section 7.4.4 of the fourth edition of this book if you're really interested). However, a designer should really strive to use automated tools for such tasks whenever possible, not only to reduce work, but also to eliminate errors.

A Verilog module that does the job for either disposition of the unused states is shown in Program 9-1. It basically embodies the next-state entries of the transition/excitation table into a `case` statement whose choices are selected by the current state/input combination, the 5-bit value of `{Q1,Q2,Q3,A,B}`, with a

DON'T GET SO EXCITED

In state-machine design with D flip-flops, the excitation table is a trivial renaming of the transition table. That's because a D flip-flop has an almost trivial characteristic equation: Q∗ = D. With a D flip-flop, to go to a particular coded next state, you simply apply that state's encoded value to state flip-flops' D inputs.

That's not true with other flip-flop types. For example, if you somehow got stuck building your state memory out of T flip-flops with enable, your excitation table would have EN=1 for each entry where the next value of a state variable is different from the current, and EN=0 where it is the same. If you were using J-K flip-flops, you would have *two* entries for each transition, one for the value of J, and another for K. Yes, at one time state-machine designers really did have to deal with this sort of thing, so consider yourself lucky!

Program 9-1 Verilog module for the transition logic specified by Table 9-6.

```verilog
module VrExTrantbl(Q1, Q2, Q3, A, B, D1, D2, D3);
  input Q1, Q2, Q3, A, B;
  output reg D1, D2, D3;
  reg [4:0] incomb;
  reg [2:0] d;

  always @ (*) begin
    incomb = {Q1, Q2, Q3, A, B};
    case (incomb)
       5'b00000:d=3'b100; 5'b00001:d=3'b100; 5'b00011:d=3'b101; 5'b00010:d=3'b101;
       5'b10000:d=3'b110; 5'b10001:d=3'b110; 5'b10011:d=3'b101; 5'b10010:d=3'b101;
       5'b10100:d=3'b100; 5'b10101:d=3'b100; 5'b10111:d=3'b111; 5'b10110:d=3'b111;
       5'b11000:d=3'b110; 5'b11001:d=3'b110; 5'b11011:d=3'b111; 5'b11010:d=3'b101;
       5'b11100:d=3'b100; 5'b11101:d=3'b110; 5'b11111:d=3'b111; 5'b11110:d=3'b111;
       default: d=3'b000;
    endcase
    {D1, D2, D3} = d;
  end
endmodule
```

single statement per case that assigns the listed next-state value to {D1,D2,D3}. We have set up for minimal risk by coding the `default` choice, which is taken for the unspecified state/input combinations, to transition to the all-0s, or INIT state.

For the second method, when Program 9-1 is targeted to an FPGA using Xilinx Vivado tools, the compiler and synthesizer generate bit patterns corresponding to the transition/excitation table to be downloaded into three LUTs that implement D1, D2, and D3. As a bonus, if you know where to look, it also derives and displays minimal sum-of-products equations for these three signals, which can be used in the first method:

$$
\begin{aligned}
\mathrm{D1} &= \mathrm{Q1} + \mathrm{Q2}' \cdot \mathrm{Q3}' \\
\mathrm{D2} &= \mathrm{Q1} \cdot \mathrm{Q3}' \cdot \mathrm{A}' + \mathrm{Q1} \cdot \mathrm{Q3} \cdot \mathrm{A} + \mathrm{Q1} \cdot \mathrm{Q2} \cdot \mathrm{B} \\
\mathrm{D3} &= \mathrm{Q1} \cdot \mathrm{A} + \mathrm{Q2}' \cdot \mathrm{Q3}' \cdot \mathrm{A}
\end{aligned}
$$

Thankfully, these equations precisely match the ones derived by the old-fashioned Karnaugh-map-based minimization method mentioned earlier. (See the box on page 471 for the minimal-cost approach using Verilog.)

Figure 9-21 shows the logic diagram with 3 LUTs for the FPGA-based implementation in the first method, as derived by Vivado. Even though the Xilinx 7-series LUT can implement any combinational logic function of up to six variables, by minimizing the excitation equations the synthesis tool has determined that the D1 and D3 outputs can be implemented using just 3-input and 4-input LUTs, respectively.

FROM TABLE TO EQUATIONS

"Turning the crank" to derive excitation equations from a transition/excitation table is not a lot of fun. In the old days, we would painstakingly copy the specified excitation values from the table onto Karnaugh maps, a tedious and error-prone process. For unused states, in a minimal-risk disposition we would enter the coded value needed to get to the default state from each unused state; at least that was easy if we the default state was coded as all 0s or all 1s—just enter all 0s or all 1s. In a minimal-cost disposition, we would enter "don't-cares" as the excitation value in the unused states. Then, in either case, we would use the maps to manually derive a minimized logic expression which could finally be converted to a circuit.

Nowadays, we have logic-minimization software that can do minimization and derive excitation equations somewhat painlessly; we coerced Verilog to do this for us in Program 9-1 for the example. However, that does not eliminate the still error-prone tedium of copying a transition/excitation table into a Verilog module, which we had to do to create Program 9-1. We could have used a `parameter` statement there to define the next-state encodings and eliminate some of the tedium (so we could write in the `case` choices, for example, "`5'b00000:d=A0`"). But the best way to eliminate that tedium is never to create a transition/excitation table in the first place!

When we design a state machine using an HDL, the compiler internally derives excitation equations based on our higher-level specifications of next-state behavior and state encoding, minimizing them as appropriate, and the synthesis tool creates an implementation of these equations in the target technology, whether it is discrete ASIC gates, a PLD, or LUTs in an FPGA. We'll see examples of this going forward.

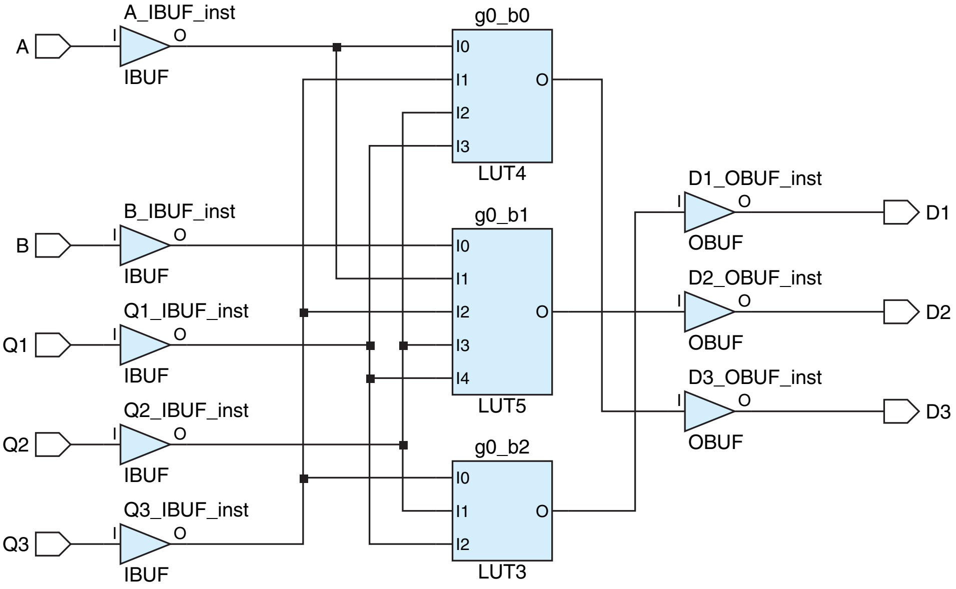


Figure 9-21 Xilinx FPGA logic diagram for the excitation equations derived from Program 9-1.

MINIMAL-COST SOLUTION

If we choose in our example to derive minimal-cost excitation equations, we write "don't-cares" in the next-state entries for the unused states. If we derive the resulting excitation equations the old-fashioned way using Karnaugh maps (as shown in previous editions of this book), two of them are somewhat simpler than before:

$$\begin{aligned} \mathsf{D1} &= 1 \\ \mathsf{D2} &= \mathsf{Q1} \cdot \mathsf{Q3}' \cdot \mathsf{A}' + \mathsf{Q3} \cdot \mathsf{A} + \mathsf{Q2} \cdot \mathsf{B} \\ \mathsf{D3} &= \mathsf{A} \end{aligned}$$

The corresponding change in Program 9-1 would be to change the next-state for the `default` case to "`d = 3'bxxx`". Unfortunately, while the righthand side looks like three don't-care bits, it's not. In Verilog, "`x`" means "unknown," not "don't-care." These `x`'s may or may not be treated as "don't-cares" in synthesis, depending on the tools. For example, Vivado treats them as 0s, and synthesizes exactly the same excitation logic as in the original module. So, when designing and implementing state machines using Verilog, we should just be content with the minimal-risk disposition of unused states, which is usually the best choice anyway.

An output equation can easily be developed directly from the information in Table 9-6. The output equation is simpler than the excitation equations in this example, because the output is a function of state only. It's easy to find the output function algebraically, by writing it as the sum of the minterms for the two coded states (110 and 111) in which Z is 1:

$$\begin{aligned} \mathsf{Z} &= \mathsf{Q1} \cdot \mathsf{Q2} \cdot \mathsf{Q3}' + \mathsf{Q1} \cdot \mathsf{Q2} \cdot \mathsf{Q3} \\ &= \mathsf{Q1} \cdot \mathsf{Q2} \end{aligned}$$

At this point, we're just about done with the state-machine design. The final step is to combine the excitation logic and the output logic with the state memory as in the structure of Figure 9-5 on page 444, in a logic diagram or in an HDL or other representation that can be used to synthesize or build the circuit.

This example has shown, in principle, how we could design and synthesize a state machine, going from a word description to a state table to logic equations for next-state and output logic. In this approach, the state table is both beneficial and troublesome:

- The state table specifies, by definition, the next state for every possible combination of current state and input. Constructing it forces the designer to explicitly consider every possibility—this a benefit.
- The size of the state table, and the amount of work needed to construct it, grows proportionally with the number of states. This is unavoidable.
- The size of the state table grows exponentially with the number of inputs, doubling for each additional input. This makes it difficult and tedious to design state machines that have more than a few inputs.

9.3.5 Beyond State Tables

Since state-table size can grow exponentially with the number of inputs, we need a method of state-machine design where the work is more in line with the complexity of the next-state decisions that are made in each state, not the number of inputs that are being examined. There are two other traditional descriptive structures for state machines that have the desired characteristic. The first is the state diagram; we constructed examples of them in state-machine *analysis* at the end of Section 9.2, and we'll show how to *design* with them in Section 9.4.

As we'll show in Section 9.4, the potential for creating ambiguities in state diagrams leads to the second structure, the "Algorithmic State Machine" (ASM) chart, which will be described in Section 9.5 and does not have this problem. ASM charts are closely related to both early state-machine description languages and modern HDLs that can use familiar programming constructs like `if-then-else` and `case` along with boolean conditional expressions to describe next-state behavior unambiguously. In fact, if you plan to design state machines only using HDLs, you can skip the next two sections of this chapter, even though they have some technical and historical interest. We'll give a sneak preview of state-machine design with Verilog in Section 9.6, and go on to a complete treatment and lots of examples of Verilog state machines in Chapter 12.

*9.4 State-Machine Design with State Diagrams

Many people like to design visually, so state diagrams are often used to design small- to medium-sized state machines; we'll give an example in this section. Once you have a state diagram, you can code the state diagram into a Verilog model as we'll show in Program 9-2 on page 485.

state diagram
node
directed arc
transition expression

Recapping the definition, a *state diagram* has one circle (or *node*) for each state and an arrow (or *directed arc*) for each transition. Each arc is labeled with a *transition expression*; the labeled transition is taken for input combinations for which the transition expression is 1.

Designing a state diagram is much like designing a state table, which, as we showed in Section 9.3.1, is much like writing a program. However, state tables and state diagrams are fundamentally different in a way that makes state diagrams less tedious to construct but also more error-prone:

- A state table is an exhaustive listing of the next states for each state/input combination. No ambiguity is possible.
- A state diagram contains a set of arcs labeled with transition expressions. Even when there are many inputs, only one transition expression is required per arc. However, when a state diagram is constructed, there is no guarantee that the transition expressions written on the arcs leaving a particular state cover all of the input combinations exactly once.

* Throughout this book, optional sections are marked with an asterisk.

In an improperly constructed (*ambiguous*) state diagram, some state/input combinations may have no next state specified, which is generally undesirable, while others may have multiple next states, which is just wrong. Thus, considerable care must be taken in the design of state diagrams.

ambiguous state diagram

We didn't have any worries with state diagrams when we *analyzed* state machines in Section 9.2. We derived transition equations from the logic diagram of the circuit, and using the resulting state table we were able to derive transition expressions to use in the corresponding state diagram. To *design* a state machine using state diagrams, we work in the opposite direction, and there is one very important rule that we must observe from the beginning to avoid creating an ambiguous state diagram. The transition expressions on arcs leaving a particular state must be mutually exclusive and all inclusive:

- No two transition expressions can equal 1 for the same input combination, since a machine can't have two next states for one input combination.

mutual exclusion

- For every possible input combination, some transition expression must equal 1, so that all next states are defined.

all inclusion

We'll keep this in mind in our state-diagram design example.

*9.4.1 T-Bird Tail Lights Example

Our example is a state machine that controls the tail lights of a 1965 Ford Thunderbird, shown in Figure 9-22. There are three lights on each side, and for turns they operate in sequence to animate the turning direction, as illustrated in Figure 9-23. The state machine has two input signals, LEFT and RIGHT, that carry the driver's request for a left turn or a right turn. It also has an emergency-flasher input, HAZ, that requests the tail lights to be operated in hazard mode—all six lights flashing on and off in unison. The state machine uses a free-running clock signal whose frequency equals the desired flashing rate for the lights.

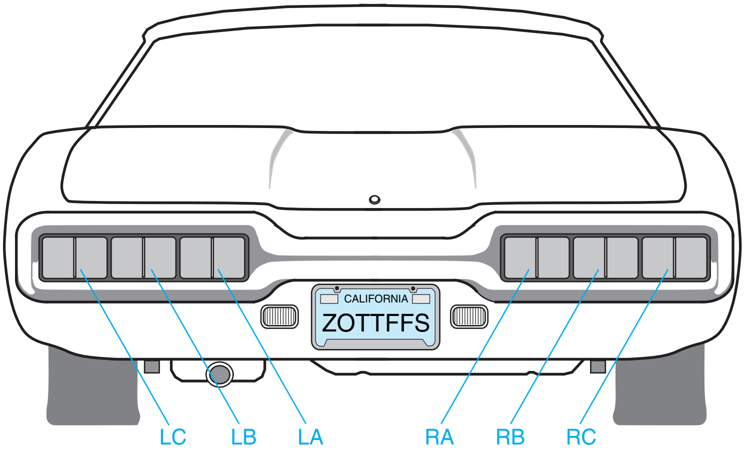


Figure 9-22
T-bird tail lights.

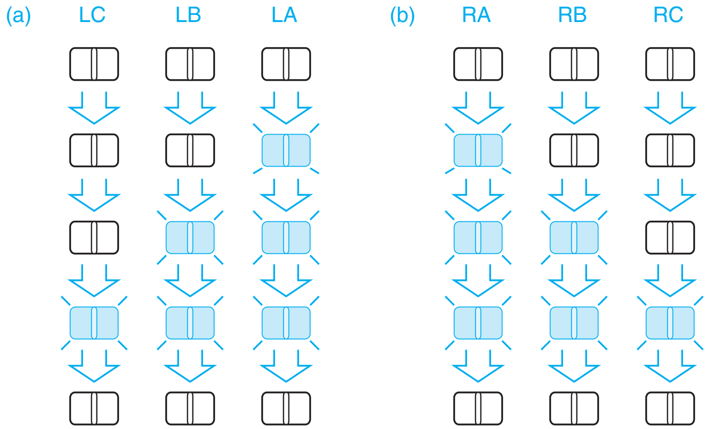


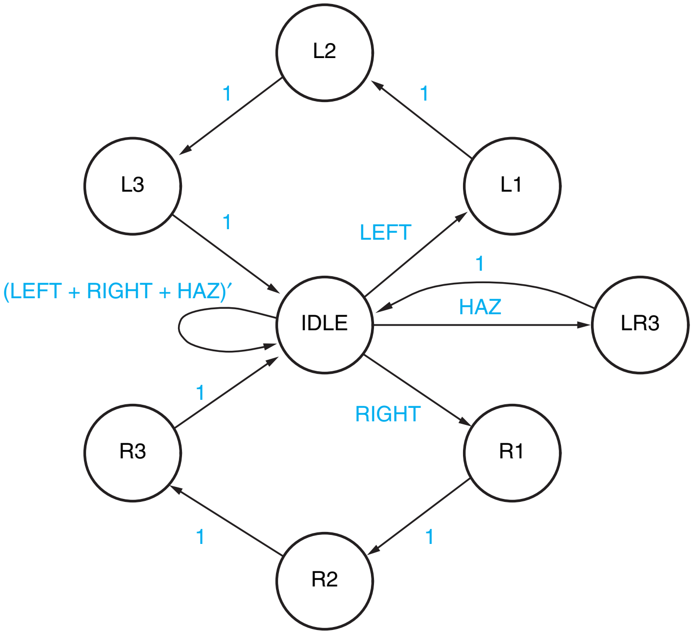
Figure 9-23 Flashing sequence for T-bird tail lights: (a) left turn; (b) right turn.

Given the foregoing specifications, we can design a state machine to control the T-bird tail lights. We will design a Moore machine, so that the state alone determines which lights are on and which are off. For a left turn, the machine should cycle through four states in which the righthand lights are off and 0, 1, 2, or 3 of the lefthand lights are on. Likewise, for a right turn, it should cycle through four states in which the lefthand lights are off and 0, 1, 2, or 3 of the righthand lights are on. In hazard mode, only two states are required—all lights on and all lights off.

Figure 9-24 shows our first cut at a state diagram for the machine. A common IDLE state is defined in which all of the lights are off. When a left turn is requested, the machine goes through three states in which 1, 2, and 3 of the lefthand lights are on, and then back to IDLE; right turns work similarly. In the hazard mode, the machine cycles back and forth between the IDLE state and a state in which all six lights are on. Since there are so many outputs, we've included a separate output table rather than writing output values on the state diagram. Even without assigning coded states to the named states, we can write output equations from the output table, if we let each state name represent a logic expression that is 1 only in that state:

$$
\begin{aligned}
\mathsf{LA} &= \mathsf{L1} + \mathsf{L2} + \mathsf{L3} + \mathsf{LR3} & \qquad \mathsf{RA} &= \mathsf{R1} + \mathsf{R2} + \mathsf{R3} + \mathsf{LR3} \\
\mathsf{LB} &= \mathsf{L2} + \mathsf{L3} + \mathsf{LR3} & \qquad \mathsf{RB} &= \mathsf{R2} + \mathsf{R3} + \mathsf{LR3} \\
\mathsf{LC} &= \mathsf{L3} + \mathsf{LR3} & \qquad \mathsf{RC} &= \mathsf{R3} + \mathsf{LR3}
\end{aligned}
$$

There's one big problem with the state diagram of Figure 9-24—it doesn't properly handle multiple inputs asserted simultaneously. For example, what happens in the IDLE state if both LEFT and HAZ are asserted? According to the state diagram, the machine goes to two states, L1 and LR3, which is impossible. In reality, the machine would have only one next state, which could be L1, LR3, or a totally unrelated (and possibly unused) third state, depending on details of the state machine's realization (see Exercises 9.37 and 9.40).



Output Table

State	LC	LB	LA	RA	RB	RC
IDLE	0	0	0	0	0	0
L1	0	0	1	0	0	0
L2	0	1	1	0	0	0
L3	1	1	1	0	0	0
R1	0	0	0	1	0	0
R2	0	0	0	1	1	0
R3	0	0	0	1	1	1
LR3	1	1	1	1	1	1

Figure 9-24
Initial state diagram and output table for T-bird tail lights.

Figure 9-24 is an ambiguous state diagram, and it’s fixed in Figure 9-25, where we have given the HAZ input priority. We’ve also enhanced the state machine’s functionality to treat LEFT and RIGHT asserted simultaneously as a hazard request, since the driver is clearly confused and needs help.

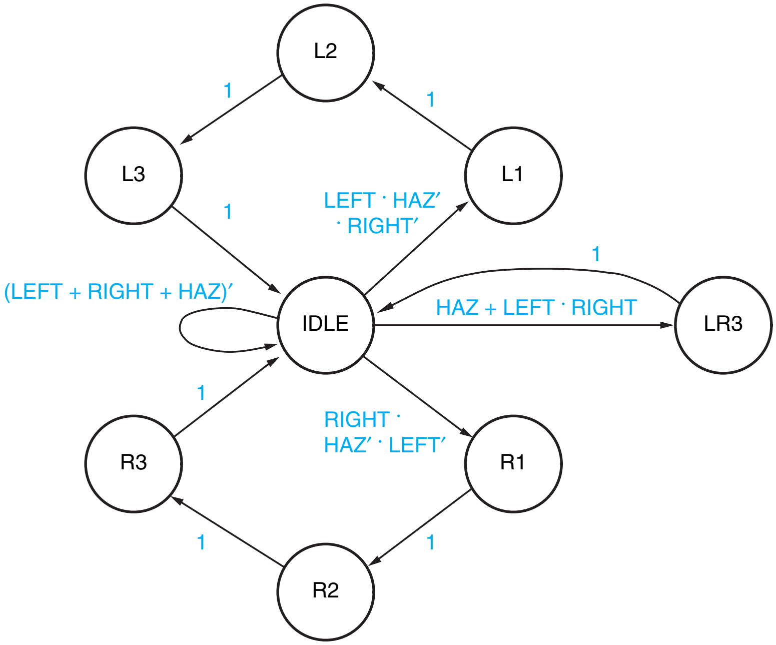


Figure 9-25
Corrected state diagram for T-bird tail lights.

How do we know that the new state diagram is unambiguous, that is, that the transition expressions on the arcs leaving each state are mutually exclusive and all-inclusive? This can be confirmed algebraically for this or any other state diagram by performing two steps:

mutual exclusion

1. *Mutual exclusion.* For each state, show that the logical product of each possible pair of transition expressions on arcs leaving that state is 0. If there are n arcs, then there are $n(n-1)/2$ logical products to evaluate.

all inclusion

2. *All inclusion.* For each state, show that the logical sum of the transition expressions on all arcs leaving that state is 1.

That may sound like a lot of work, and in fact it is, for all but the simplest state diagrams. In Figure 9-25, most of the states have a single arc with a transition expression of 1, so verification for those is trivial. Real work is needed only to verify the IDLE state, which has four transitions leaving it. This can be done on a sheet of scratch paper by listing the eight combinations of the three inputs and checking off the combinations covered by each transition expression. Each combination should have exactly one check. As another example, consider the state diagram in Figure 9-14 on page 454; it can be verified mentally using fairly basic switching algebra.

At this point, we can synthesize the T-bird tail lights state machine from the state diagram. As in synthesis from a state table, the next step is to choose a state encoding. There are eight states, and we'll use the minimum of three flip-flops to encode them. Obviously, many state assignments are possible (8! to be exact); we'll use the one in Table 9-7 for the following reasons:

1. An initial (idle) state of 000 is compatible with typical D flip-flops, which are easily initialized to the 0 state.
2. Two state variables, Q1 and Q0, are used to "count" in Gray-code sequence for the left-turn cycle (IDLE→L1→L2→L3→IDLE). This minimizes the number of state-variable changes per state transition, which can often simplify the excitation logic.

Table 9-7
State assignment for the T-bird tail lights state machine.

State	*Q2*	*Q1*	*Q0*
IDLE	0	0	0
L1	0	0	1
L2	0	1	1
L3	0	1	0
R1	1	0	1
R2	1	1	1
R3	1	1	0
LR3	1	0	0

3. Because of the symmetry in the state diagram, the same sequence on Q1 and Q0 is used to "count" during a right-turn cycle, while Q2 is used to distinguish between left and right.
4. The remaining state-variable combination is used for the LR3 state.

The next step is to write a sort of transition table. However, we must use a format different from the transition tables of Section 9.3.4, because transitions in a state diagram are specified by expressions rather than by an exhaustive tabulation of next states. We'll call the new format a *transition list*, because it has one row for each transition or arc in the state diagram.

transition list

Table 9-8 is the transition list for the state diagram of Figure 9-25 and the state assignment of Table 9-7. Each row contains the current state, next state, and transition expression for one arc in the state diagram. Both named and coded versions of the current state and next state are shown in each row. Named states are useful for reference purposes, while coded states are used when developing transition equations.

Once we have a transition list, the rest of the synthesis steps are pretty much just "turning the crank." For each next-state variable V∗ we need a transition equation that defines its value in terms of the current state and input. The transition list can be viewed as a sort of hybrid truth table in which the state-variable combinations for the current state are listed explicitly, and input combinations are listed algebraically. Reading down a V∗ column in a transition list, we find a sequence of 0s and 1s, indicating the value of V∗ for various (if we've done it right, all) state/input combinations.

A row's *transition p-term* is defined to be the product of the current state's minterm and the transition expression in the row. The transition equation for V∗ has one transition p-term for each row of the transition list that has a 1 in the V∗

transition p-term

S	*Q2*	*Q1*	*Q0*	*Transition Expression*	*S∗*	*Q2∗*	*Q1∗*	*Q0∗*
IDLE	0	0	0	(LEFT + RIGHT + HAZ)′	IDLE	0	0	0
IDLE	0	0	0	LEFT · HAZ′ · RIGHT′	L1	0	0	1
IDLE	0	0	0	HAZ + LEFT · RIGHT	LR3	1	0	0
IDLE	0	0	0	RIGHT · HAZ′ · LEFT′	R1	1	0	1
L1	0	0	1	1	L2	0	1	1
L2	0	1	1	1	L3	0	1	0
L3	0	1	0	1	IDLE	0	0	0
R1	1	0	1	1	R2	1	1	1
R2	1	1	1	1	R3	1	1	0
R3	1	1	0	1	IDLE	0	0	0
LR3	1	0	0	1	IDLE	0	0	0

Table 9-8 Transition list for the T-bird tail lights state machine.

column. Thus, the transition equation for Q2* can be written as the sum of the p-terms for the four rows where Q2* is 1:

$$\begin{aligned} \mathsf{Q2*} \;=\; & \mathsf{Q2'} \cdot \mathsf{Q1'} \cdot \mathsf{Q0'} \cdot (\mathsf{HAZ} + \mathsf{LEFT} \cdot \mathsf{RIGHT}) \\ & + \mathsf{Q2'} \cdot \mathsf{Q1'} \cdot \mathsf{Q0'} \cdot (\mathsf{RIGHT} \cdot \mathsf{HAZ'} \cdot \mathsf{LEFT'}) \\ & + \mathsf{Q2} \cdot \mathsf{Q1'} \cdot \mathsf{Q0} \\ & + \mathsf{Q2} \cdot \mathsf{Q1} \cdot \mathsf{Q0} \end{aligned}$$

The equations for Q1* and Q0* have been left as an exercise (9.26).

Using D flip-flops for the state memory, the transition equations are also the excitation equations for the D inputs, and we can synthesize them in the target technology. Now we are almost done, with only the output logic left to do. In this particular example, we already wrote equations on page 474 for its Moore-type outputs in terms of its symbolic state names, so we need only to substitute the minterms corresponding to the named states, and we really are done.

More examples of designing state machines with state diagrams can be found in Sections 7.6–7.7 of this book's fourth edition. However, the extra worry and work that is needed to create unambiguous state diagrams makes it worth considering a more trouble-free method to define state machines graphically, as described in the next section.

*9.5 State-Machine Design with ASM Charts

ASM chart

An *algorithmic state machine (ASM) chart* is a graphical specification of state-machine behavior that looks more like a programmer's flowchart than a state diagram. Figure 9-26 shows the basic elements that appear in ASM charts:

state box

- *State box*. An ASM chart has one state box per state, showing the state name and optionally the state's coding, and containing a list of Moore-type outputs that are asserted in that state (unlisted outputs are negated in that state). The most important difference between a state box and a node in a state diagram is that the state box has just a single exit point representing the next-state transition, shown by a single transition arrow leaving the box. This arrow leads to another state box or to a decision box.

decision box
condition expression

- *Decision box*. A single transition arrow is split into two alternative transitions by a decision box, which contains a *condition expression*—a logic expression involving the machine's inputs. For input combinations where the expression is 1, the exit path labeled 1 is taken; otherwise, the exit path labeled 0 is taken. Each exit path leads to a state box or to another decision box. Multiple decision boxes with different condition expressions may be placed in series when a state has multiple next states.

conditional output box

- *Conditional output box*. This element is placed on top of the exit path of a decision box to specify Mealy-type outputs. It lists outputs that are asserted in the current state (which is determined by looking back along the path to

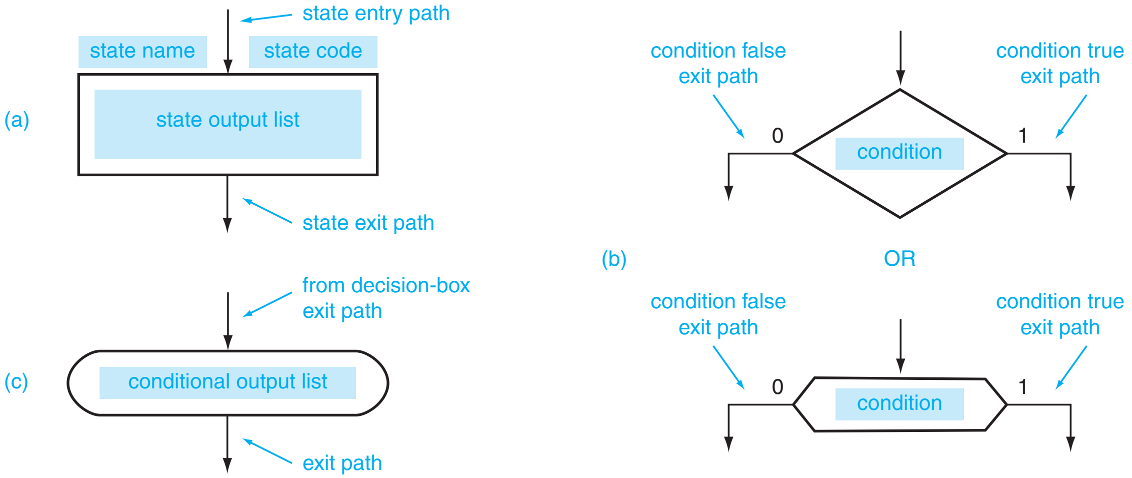


Figure 9-26 Elements of ASM charts: (a) state box; (b) decision box; (c) conditional output box.

a state box), given an input combination that would cause the path to be taken at the next clock tick. Like the state-diagram notation for Mealy-type outputs, this is a little misleading, since the outputs are normally asserted for the entire portion of the clock period in which the conditions are satisfied, not just at the clock tick when the transition is taken.

A few simple ASM charts are shown in Figure 9-27, including the names of the state variables to be used in synthesis. The first chart, in (a), is for a free-running "divide-by-4 counter." Its state is encoded in two bits (Q1 Q0) and it has

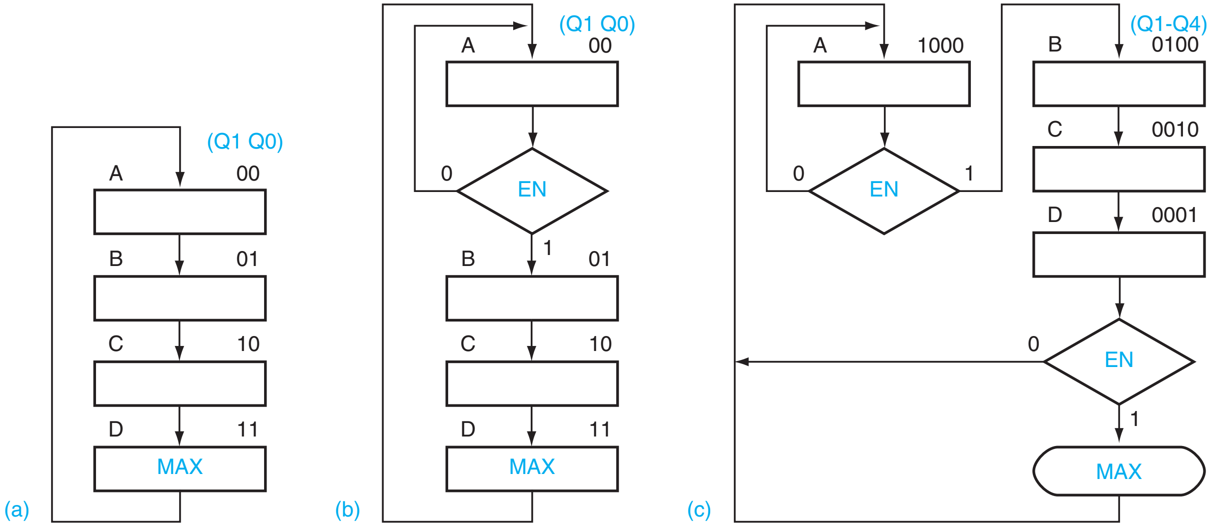


Figure 9-27 ASM charts: (a) free-running modulo-4 counter; (b) modulo-4 counter with enable; (c) modulo-4 counter with a Mealy-type output.

a single Moore-type output MAX which is asserted in state D. In (b), we have provided an enable input, EN, which is tested only in state A. Finally, in (c) we use a 1-hot state encoding, and MAX is now a Mealy-type output. Note that the transition from state D to A is still unconditional; the bottom decision box affects only whether the conditional output box is traversed.

*9.5.1 T-Bird Tail Lights with ASM Charts

Figure 9-28 shows an ASM chart for the T-bird tail lights example from the previous section. The IDLE state in the original state-diagram-based design has several transitions leaving it, the ASM chart requires a few decision boxes in series to define the transitions. Each state box contains a list of the outputs that are asserted in that state. The drawing conventions are somewhat relaxed; for example, exit paths leave the most convenient side of a decision box. You are encouraged to study the ASM chart well enough to convince yourself that it does the job.

The big advantage of designing with ASM charts is that a properly constructed chart is guaranteed to provide an unambiguous description of next-state behavior. That is, in each state, every input combination leads to exactly one next state in the chart. This is true because every input combination yields a definite outcome in each decision box and, in a properly constructed chart, all exit paths go to a single next state or to another decision box. Thus, the mutual-exclusion and all-inclusion properties that we required for state diagrams are provided automatically.

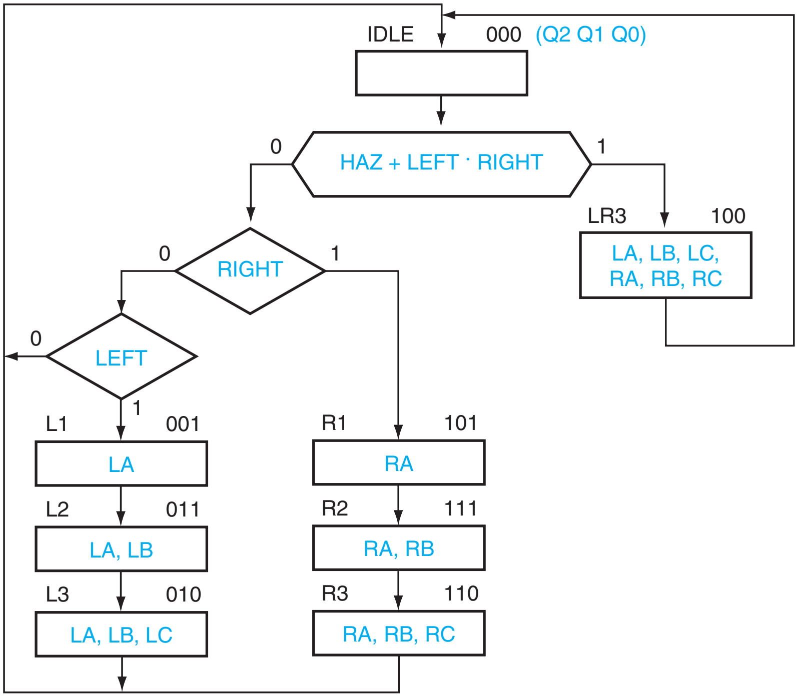


Figure 9-28
ASM chart for
T-bird tail lights.

For example, notice in Figure 9-28 that the LEFT and RIGHT condition expressions leading out of the IDLE state are not mutually exclusive, but they don't have to be. The structure of the ASM chart guarantees that RIGHT would have priority over LEFT, even if we hadn't decided to interpret LEFT=RIGHT=1 as a hazard condition in the first decision box.

You can construct a transition list from an ASM chart by hand, if you really want to. In a state diagram, each possible transition from a state has its own arc leaving the state node. In an ASM chart, each state box itself has only one exit path; the possible transitions correspond to all possible paths to next states going through zero or more decision boxes. Each path yields one transition p-term and row in the transition list. For example, in the T-bird tail lights machine, there are four possible paths leaving the IDLE state, as shown in Figure 9-29.

The transition p-term corresponding to an ASM-chart path is the logical product of the current state's minterm and the condition expressions that appear in the decision boxes along the chosen path. Each condition expression is complemented if the chosen path goes through the "0" (false) exit path of its decision box; otherwise it appears as-is. Thus, the IDLE state in Figure 9-28 has the transition p-terms shown in Table 9-9. The remaining states have no condition boxes; each has just one transition p-term, its current state's minterm.

The transition p-terms in Table 9-9 are algebraically equal to the ones in Table 9-8 on page 477 that we derived in the state-diagram based design for the same state machine (you have to do a little Boolean algebra to show that; see Exercise 9.29). Once you have all the transition p-terms, you can derive transition equations for all of the state variables in the same way that we described

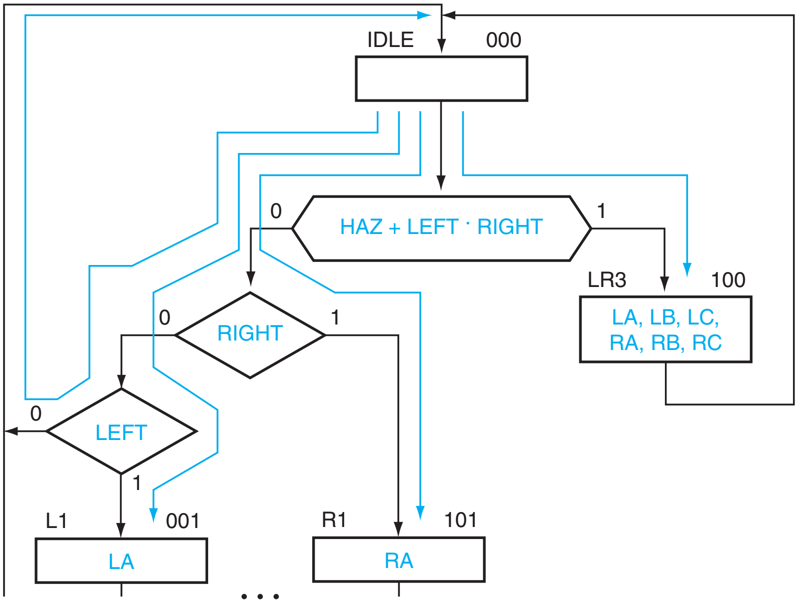


Figure 9-29
Paths leaving the IDLE state in the T-bird machine.

Table 9-9 Transition p-terms for the IDLE state in Figure 9-28.

Condition Expression Values				
HAZ + LEFT·RIGHT	RIGHT	LEFT	*Transition p-term*	*S**
1	–	–	Q2′·Q1′·Q0′ · (HAZ + LEFT·RIGHT)	LR3
0	1	–	Q2′·Q1′·Q0′ · (HAZ + LEFT·RIGHT)′ · (RIGHT)	R1
0	0	1	Q2′·Q1′·Q0′ · (HAZ + LEFT·RIGHT)′ · (RIGHT)′ · (LEFT)	L1
0	0	0	Q2′·Q1′·Q0′ · (HAZ + LEFT·RIGHT)′ · (RIGHT)′ · (LEFT)′	IDLE

there: the transition equation for a variable has one transition p-term for each row of the transition list where that variable is 1 in the next state.

The procedure for constructing for deriving p-terms and constructing a transition list is similar to what an HDL compiler has to do to derive excitation equations when nested `if-else` statements are used to define a state machine's behavior. So, even if you never need to construct an ASM chart, at least they should have given you some insight into how an HDL compiler and synthesizer can construct a working state machine from your behavioral description.

IMPROPERLY CONSTRUCTED ASM CHARTS (AND VERILOG STATE MACHINES)

In this section, we've made a big deal about how a properly constructed ASM chart yields an unambiguous description of a state machine's behavior. So what is an improperly constructed ASM chart?

Some ASM-chart authors allowed two or more exit paths from a state box, and that is exactly the characteristic of a state diagram—multiple arcs leaving a node—that can lead to ambiguity. With two or more exit paths from a state box, there can be two or more parallel decision boxes. And if their condition expressions are not mutually exclusive and all inclusive, we have an ambiguous ASM chart.

There is no such thing as an ambiguous Verilog behavioral state-machine description, in the sense that the language's behavior for any legal model is well defined. For example, if the case for one state had multiple individual `if` statements, rather than a single nested `if-else`, then two or more state-register assignments may be executed for some input combination. However, only the last one takes effect, both in simulation and in the synthesized circuit. Still, the model's reader may not understand what's really going to happen; for example, they may be focused on the first assignment.

Therefore, even in Verilog you should "properly construct" next-state behaviors so it is obvious to the reader (including yourself!) what is intended. Besides a single nested `if-else`, analogous to a series of ASM-chart decision boxes, we'll give examples of some recommended styles that work, and caution against some that may not work, in Chapter 12.

9.6 State-Machine Design with Verilog

This section introduces state-machine design with Verilog; we'll cover this topic in the depth it deserves with a lot more variations and examples in Chapter 12.

In Section 9.3 we illustrated the state-table design process using the simple design problem below:

> Design a clocked synchronous state machine with two inputs, A and B, and a single output Z that is 1 if:
>
> - A had the same value at each of the two previous clock ticks, *or*
> - B has been 1 since the last time that the first condition was true.
>
> Otherwise, the output should be 0.

In a Verilog environment, there are many ways to create a module that meets the stated requirements. We'll look at just one here, and others in Chapter 12.

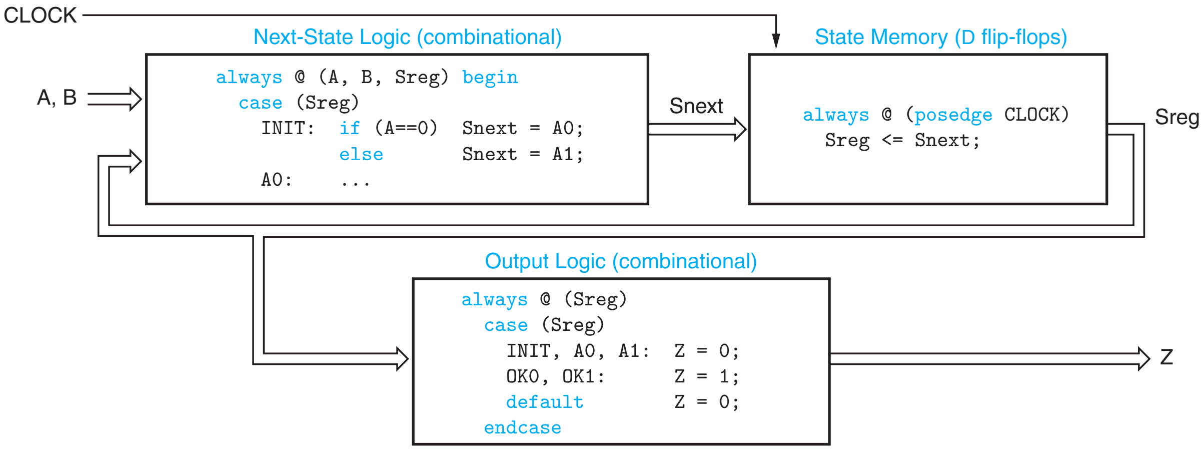
We already developed a state and output table for the above design problem in Section 9.3.1, and doing that was more than just "turning the crank." We had to think about the problem requirements and evaluate different situations in the state machine's operation. So we'll go ahead and use that state table as the basis of a Verilog module that realizes the machine. In many situations we can design a state machine without writing out a state table, but we'll save that discussion for Chapter 12. Here, we'll show how to convert an existing state table into a Verilog module without all the fuss of Sections 9.3.2 through 9.3.4.

We've written the state table again in Table 9-10. Even though we're using a manually constructed state table, the big difference here is that we don't have to construct a transition/excitation table by hand. Instead, we can convert the state table directly into a corresponding Verilog module with five sections:

1. Declarations of inputs, outputs, and local variables. The output and local variables will have "reg" types since we will use behavioral code to specify the machine's operation.
2. A parameter statement to assign a state-variable combination to each named state.

	A B				
S	***00***	***01***	***11***	***10***	***Z***
INIT	A0	A0	A1	A1	0
A0	OK0	OK0	A1	A1	0
A1	A0	A0	OK1	OK1	0
OK0	OK0	OK0	OK1	A1	1
OK1	A0	OK0	OK1	OK1	1
		S*			

Table 9-10 State and output table for the example state machine.


CLOCK
Next-State Logic (combinational)
State Memory (D flip-flops)
A, B
Snext
Sreg
Output Logic (combinational)
Z

```
always @ (A, B, Sreg) begin
  case (Sreg)
    INIT:  if (A==0)  Snext = A0;
           else       Snext = A1;
    A0:    ...
```

```
always @ (posedge CLOCK)
  Sreg <= Snext;
```

```
always @ (Sreg)
  case (Sreg)
    INIT, A0, A1:  Z = 0;
    OK0, OK1:      Z = 1;
    default        Z = 0;
  endcase
```

Figure 9-30 Moore state-machine structure implied by Verilog coding style.

3. A first `always` block to create the state memory, corresponding to the state memory of a generic Moore-machine structure in Figure 9-5 on page 444.
4. A second `always` block to define the next-state behavior, corresponding to the next-state (excitation) logic *F* in the generic Moore-machine structure in Figure 9-5.
5. A third `always` block to define the output logic, corresponding to the output logic *G* in Figure 9-5.

Figure 9-30 shows how the three Verilog `always` blocks correspond to the generic Moore-machine structure. The complete Verilog module corresponding to the example state table is shown in Program 9-2.

As usual, the Verilog module declaration specifies its inputs and outputs—`CLOCK`, `A`, `B`, and `Z` in this example. Next, the module declares `reg` variables `Sreg` and `Snext` for the machine's current and next states, respectively.

Significantly, the module uses a `parameter` statement to specify the state assignment, defining a constant to associate each of the machines's five states with a unique multibit value. Here, we use the "simplest" state assignment of Table 9-5 on page 462, using the first five of eight available 3-bit combinations. However, we could have used any of the state assignments in the table, simply by changing the definitions in the `parameter` statement and changing the width of `Sreg` and `Snext` if more than three bits are used to encode the state. By using the `parameter` statement in this way, we're making the compiler do some of the tedious and error-prone work for us, substituting state values for state names.

The first `always` block in the module is a "sequential `always` block" that creates the state memory. Notably, this block's sensitivity list uses Verilog's `posedge` keyword (see Section 5.14), so it executes only on the rising edge of the named signal, `CLOCK`. At that time, it loads the next state `Snext` into the state

Program 9-2 Verilog module for state-machine example.

```
module VrSMex( CLOCK, A, B, Z );
  input CLOCK, A, B;
  output reg Z;
  reg [2:0] Sreg, Snext;         // State register and next state

  parameter [2:0] INIT = 3'b000, // Define the states
                  A0   = 3'b001,
                  A1   = 3'b010,
                  OK0  = 3'b011,
                  OK1  = 3'b100;

  always @ (posedge CLOCK)       // Create the state memory
    Sreg <= Snext;

  always @ (A, B, Sreg) begin    // Next-state logic
    case (Sreg)
      INIT:   if (A==0)  Snext = A0;
              else       Snext = A1;
      A0:     if (A==0)  Snext = OK0;
              else       Snext = A1;
      A1:     if (A==0)  Snext = A0;
              else       Snext = OK1;
      OK0:    if (A==0)  Snext = OK0;
              else if ((A==1) && (B==0)) Snext = A1;
              else                       Snext = OK1;
      OK1:    if ((A==0) && (B==0))      Snext = A0;
              else if ((A==0) && (B==1)) Snext = OK0;
              else                       Snext = OK1;
      default Snext = INIT;
    endcase
  end

  always @ (Sreg)        // Output logic
    case (Sreg)
      INIT, A0, A1: Z = 0;
      OK0, OK1:     Z = 1;
      default       Z = 0;
    endcase
endmodule
```

flip-flops Sreg[2:0]. During synthesis, positive-edge-triggered D flip-flops will be inferred for Sreg; in the next four chapters, we'll see many examples of sequential always blocks that create flip-flops in this way.

The second always block specifies the combinational next-state logic using a case statement. It assigns a value to Snext in six cases, corresponding to the five explicitly defined states and a default for other, undefined states. For robustness (minimum risk), the default case sends the machine back to the INIT state.

In each case-statement choice, we've used an "if" statement and a final "else" to ensure that a value is always assigned to Snext. If there were any state/input combinations in which no value was assigned Snext, the Verilog compiler would infer a latch to hold the value of Snext for those combinations, which is something we don't want.

In formulating the if statements in Program 9-2 and the boolean conditions that they test, we have not written separate clauses for all four possible combinations of inputs A and B to mimic the four input-combination columns in the state table. Instead, we have mentally simplified the conditions while going along, in part by recalling the reasoning we used when we developed the state table in the first place. For example, we know that the transition out of the INIT state depends only on the value of A, and we don't need separate tests depending on whether B is 0 or 1.

The third and final always block in Program 9-2 handles the machine's single Moore output, Z, which is set to a value as a combinational function of the current state. It would be easy to define Mealy outputs here as well, by making Z be a function of the inputs as well as the state in each enumerated case. If this is done, then the inputs would also be added to the sensitivity list of the always block, either explicitly or by just using the "*" shorthand.

State machines can be specified in Verilog in many different ways, and we'll look at a few in Chapter 12, including direct coding without a state table or diagram. Before we get there, we'll spend some time in Chapter 10 looking at the basic sequential elements like D flip-flops that are used in state machines and other sequential circuits. All clocked sequential circuits are technically state machines, but some are so common and so easily described that they have their own names—counters and shift registers—and we'll look at them and some of their applications in Chapter 11. In Chapter 12, we'll return to state machines in Verilog, including both design and test benches.

References

pulse-mode circuit
pulse input

The clocked synchronous state machines we discussed in this chapter are a special case of a more general class of *pulse-mode circuits*. Such circuits have one or more *pulse inputs* such that (a) only one pulse occurs at a time; (b) nonpulse inputs are stable when a pulse occurs; (c) only pulses can cause state changes; and (d) a pulse causes at most one state change. In clocked synchronous state machines, the clock is the single pulse input, and a "pulse" is the triggering edge of the clock. However, it is also possible to build circuits with multiple pulse inputs, and it is possible to use storage elements other than the familiar edge-triggered flip-flops. These possibilities are discussed thoroughly in *Logic Design Principles* by Edward J. McCluskey (Prentice Hall, 1986).

A particularly important type of pulse-mode circuit that is discussed by McCluskey and others is the *two-phase latch machine*. The rationale for a two-phase clocking approach in VLSI circuits is discussed in the seminal VLSI book by Carver Mead and Lynn Conway, *Introduction to VLSI Systems* (Addison-Wesley, 1980). These machines eliminate the possibility of going to the wrong state because of internal timing dependencies, called “essential hazards,” that are present in all edge-triggered flip-flops, by using pairs of latches that are enabled by nonoverlapping clocks.

two-phase latch machine

Formal methods for minimizing state tables are described in advanced logic design texts, including McCluskey’s 1986 book. A more mathematical discussion of these methods and other “theoretical” topics in sequential machine design appears in *Switching and Finite Automata Theory*, by Zvi Kohavi and Niraj K. Jha (Cambridge University Press, 2010, third edition).

ASM charts were pioneered at Hewlett-Packard by Thomas E. Osborne and developed by his colleague Christopher R. Clare in a book, *Designing Logic Systems Using State Machines* (McGraw-Hill, 1973). Design and synthesis methods using ASM charts subsequently found a home in many digital design texts, including the first two editions of the book you’re reading.

Drill Problems

9.1 A clock signal CLK is HIGH for 10 ns and LOW for 30 ns. What are its frequency and duty cycle?

9.2 A clock signal CLK_L is HIGH for 8 ns and LOW for 12 ns. What are its frequency and duty cycle?

9.3 How many states are there in a state machine with seven D flip-flops in its state memory?

9.4 Analyze the state machine in Figure X9.4. Write excitation equations, excitation/transition table, and state/output table (use state names A–D for Q1 Q2 = 00–11).

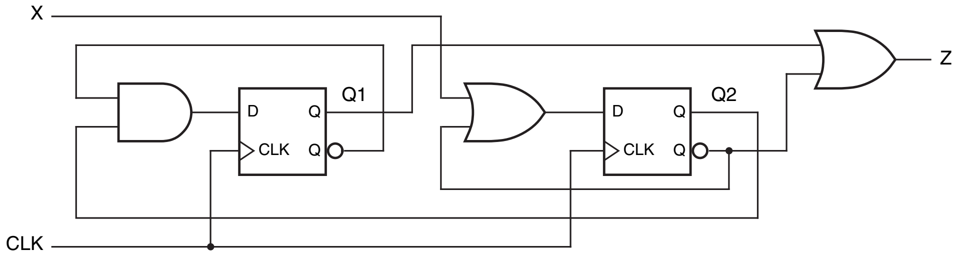


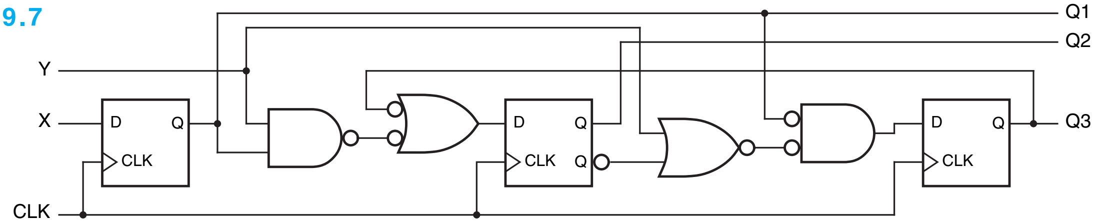
Figure X9.4

9.5 Repeat Drill 9.4, changing the AND gate to NAND and the OR gate to NOR in the excitation logic. Is there any discernible relationship between the new state table and the original?

9.6 Repeat Drill 9.4, swapping AND and OR gates in the logic diagram. Is the new state/output table the “dual” of the original one? Explain.

9.7 Analyze the state machine in Figure X9.7. Write excitation equations and then construct an excitation/transition table and a state/output table using state names A–H for Q1 Q2 Q3 = 000–111).

Figure X9.7



9.8 Analyze a state machine with three flip-flops, two inputs A and B, output Z, and excitation and output equations given below. Construct an excitation/transition table and a state/output table, using state names A–H for Q1 Q2 Q3 = 000–111.

$$Q1* = A$$

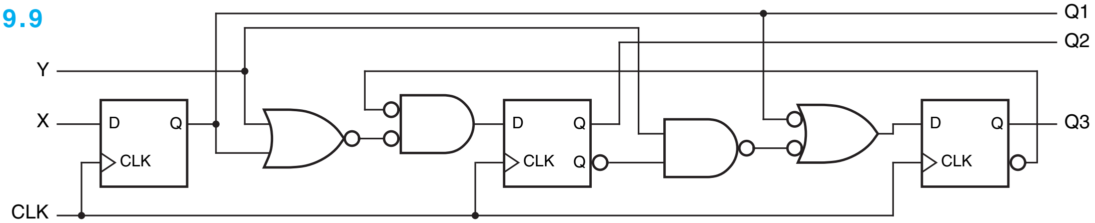
$$Q2* = Q1$$

$$Q3* = B \cdot (Q3 + (Q2' \oplus Q1))$$

$$Z = Q3 + (Q2' \oplus Q1)$$

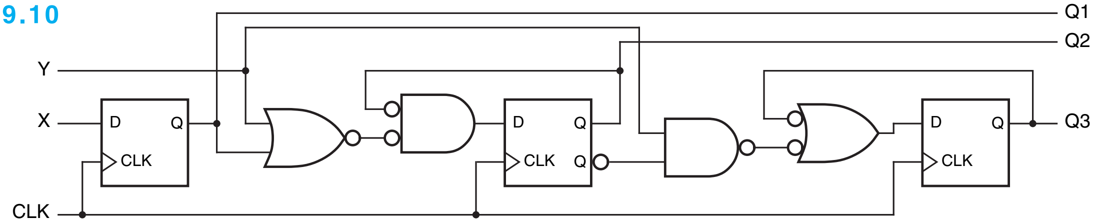
9.9 Analyze the state machine in Figure X9.9. Write excitation equations, excitation/transition table, and state/output table (use state names A–H for Q1 Q2 Q3 = 000–111).

Figure X9.9



9.10 Analyze the state machine in Figure X9.10. Write excitation equations and then construct the excitation/transition table and a state/output table using state names A–H for Q1 Q2 Q3 = 000–111).

Figure X9.10



9.11 The outputs of the state machine in Figure X9.10 are its state variables. Suppose that instead, the machine has a single output with the equation Z = Q2 · Q3′. Find the equivalent states in the machine and construct an equivalent state/output table having fewer states.

9.12 Analyze the state machine in Figure X9.12. Write excitation equations and then construct the excitation/transition table and a state table using state names A–H for Q2 Q1 Q0 = 000–111).

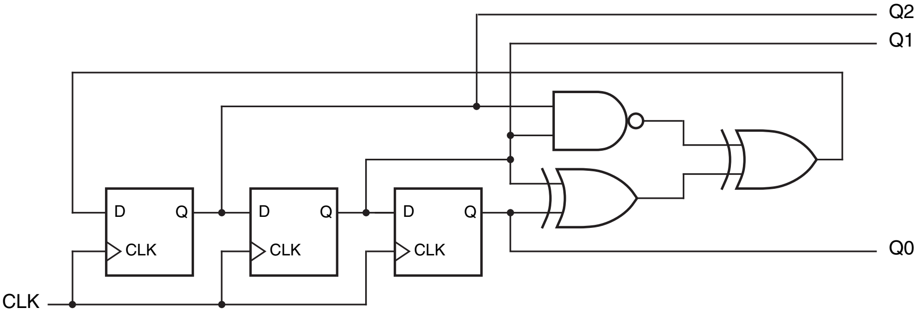


Figure X9.12

9.13 Draw a state diagram for the state machine described by Table 9-4.

9.14 Construct a state table equivalent to the state diagram in Figure X9.14. Note that the diagram is drawn with the convention that the state does not change except for input conditions that are explicitly shown.

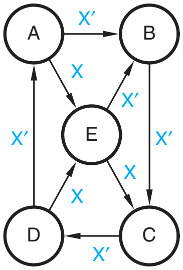


Figure X9.14

9.15 Construct a state and output table equivalent to the state diagram in Figure X9.15. Note that the diagram is drawn with the convention that the state does not change except for input conditions that are explicitly shown.

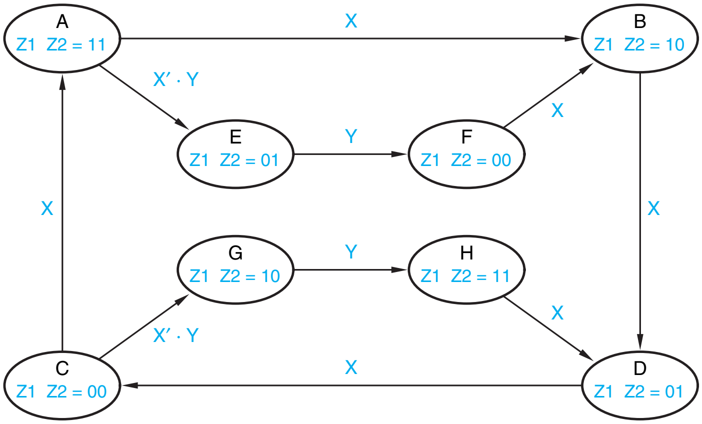


Figure X9.15

9.16 Design a state machine that checks data words received on a serial data line for even parity. The circuit should have two inputs, SYNC and DATA, in addition to CLOCK. The number of bits per input data word is variable, but SYNC is asserted during the clock period preceding the first data bit and during the last data bit. SYNC is negated during the first data bit of the next word; if SYNC remains asserted after the last bit, there is a gap between successive data words. The circuit should have one Moore-type output, ERROR, which is asserted only for one clock period after the last bit of a received word if that word had odd parity. You should be able to do the job using fewer than eight states. Create a state table for your design and provide a short description of the meaning or purpose of each state.

Exercises

9.17 You've studied Section 9.2 on state-machine analysis, and your professor has handed you a logic diagram and asked you to derive the state/output table for a machine with four edge-triggered D flip-flops with outputs Q3–Q0, four inputs I3–I0, and one Moore output Z. You realize that the state table will have 16 rows and 16 columns, and it is going to be a real pain to derive all 256 next-state entries. However, you have a Verilog simulator available to you, and it wouldn't take you very long to create a structural or dataflow-style module `VrSMtblckt` that does most of the work for you.

Write a test bench `Vr4x4x1SMtbl_tb` that instantiates `VrSMtblckt` and prints out the resulting state and output table for you. Your test bench should work for any state machine with the inputs and outputs specified above, where the details of the specific state machine are embodied in `VrSMtblckt`. To make formatting easy, use state names Q0–Q9,Qa–Qf for state values 0000–1111. To test your test bench, write and instantiate a simple `VrSMtblckt1` module where the next state is the 4-bit sum of the current state Q[3:0] and I[3:0], and the Z output is asserted only if the current state is 0000.

9.18 Admittedly, the state-table pattern created by the `VrSMtblckt1` module that was suggested in Exercise 9.17 is easy to write out by hand without the automated test bench. But now it's time for the real assignment. Derive the state and output table for a state machine with the following next-state equations and output equations:

$$\mathrm{Q0}{*} = \mathrm{Q3}' \cdot \mathrm{Q1}' \cdot \mathrm{I0}' + \mathrm{Q3} \cdot \mathrm{Q2} \cdot \mathrm{Q1} \cdot \mathrm{I1} + \mathrm{Q0} \cdot \mathrm{I3}' \cdot \mathrm{I2}'$$

$$\mathrm{Q1}{*} = (\mathrm{Q1}' + \mathrm{I0}' + \mathrm{I3} \cdot \mathrm{I2}) \cdot (\mathrm{Q3} + \mathrm{Q2}' + \mathrm{Q0}' \cdot \mathrm{I1}')$$

$$\mathrm{Q2}{*} = \mathrm{Q2} \cdot (\mathrm{I3}' \cdot \mathrm{I1} \cdot \mathrm{I0} + \mathrm{I3} \cdot \mathrm{I2}' \cdot \mathrm{I0} + \mathrm{Q3}' \cdot \mathrm{I3} \cdot \mathrm{I2}' + \mathrm{Q0} \cdot \mathrm{I3} \cdot \mathrm{I2})$$

$$\mathrm{Q3}{*} = \mathrm{Q3}' \cdot \mathrm{I3} + \mathrm{Q2}' \cdot \mathrm{I2} + \mathrm{I3} \cdot \mathrm{I2} \cdot \mathrm{I1} \cdot \mathrm{I0} + \mathrm{Q1} \cdot \mathrm{Q0}' \cdot \mathrm{I2} \cdot \mathrm{I1}'$$

$$\mathrm{Z} = (\mathrm{Q3} + \mathrm{Q0}') \cdot \mathrm{Q1} + \mathrm{Q2}' \cdot \mathrm{Q1} \cdot \mathrm{Q0} + \mathrm{Q1}' \cdot (\mathrm{Q2} + \mathrm{Q0}') \cdot (\mathrm{Q2}' + \mathrm{Q0})$$

9.19 Draw a state diagram for a state machine with two inputs, INIT and X, and one Moore-type output Z. As long as INIT is asserted, Z is continuously 0. Once INIT is negated, Z should remain 0 until X has been 0 for two successive ticks and 1 for two successive ticks, regardless of the order of occurrence. Then Z should go to 1 and remain 1 until INIT is asserted again. Your state diagram should be neatly drawn and planar (no crossed lines). (*Hint:* No more than ten states are required.)

9.20 Synthesize a state machine with the state/output table shown in Table X9.20. Use two state variables, Q1 Q2, with state assignment A = 00, B = 01, C = 11, D = 10. Write out the excitation equations and draw the logic diagram using NAND gates and D flip-flops with true and complemented outputs.

	X		
S	0	1	Z
A	B	D	0
B	C	B	0
C	B	A	1
D	B	C	0
	S*		

Table X9.20

9.21 Write a new transition table and derive minimal-risk excitation and output equations for the state table in Table 9-4 using the "simplest" state assignment in Table 9-5 and D flip-flops. Compare the cost of your excitation and output logic (when realized with a two-level AND-OR circuit) with the equations in the box on page 471.

9.22 Repeat Exercise 9.21 using the "almost one-hot" state assignment in Table 9-5.

9.23 Determine the full 8-state table for the state machine with the excitation equations in the box on page 471. Use the names U1, U2, and U3 for the states that are unused in the original state table (001, 010, and 011). Draw a state diagram and explain the behavior of the unused states.

9.24 List all of the ambiguities in the state diagrams in Figure X9.24.

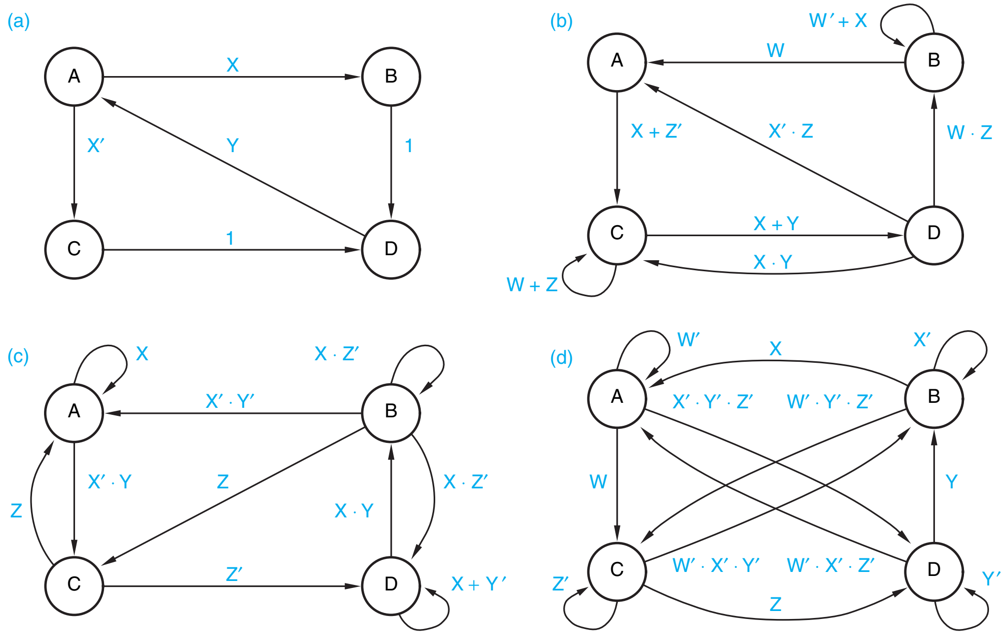


Figure X9.24

9.25 Enhance the state diagram of Figure 9-25 or the ASM chart of Figure 9-28 so the machine immediately goes to the hazard-flash state if a hazard condition is detected during a turning sequence, and immediately goes to the idle state if the turn signal request is negated during a turning sequence.

9.26 Derive the transition equations for Q1* and Q0* for the T-bird tail lights machine based on the transition list in Table 9-8. Comment on whether and how the state-assignment strategy for the machine did or did not pay off in the equations.

9.27 Synthesize a circuit for the state diagram of Figure 9-25 using six variables to encode the state, where the LA–LC and RA–RC outputs equal the state variables themselves. Write a transition list, a transition equation for each state variable as a sum of p-terms, and simplified transition/excitation equations for a realization using D flip-flops.

9.28 Repeat Exercise 9.27 for the enhanced diagram or chart of Exercise 9.25.

9.29 Using switching algebra, show that the transition p-terms for the IDLE state that we derived in Table 9-9 from the T-bird tail lights ASM chart are equal to the corresponding ones in Table 9-8 that we derived from the state diagram.

9.30 Create an unambiguous state diagram or an ASM chart for a "sticky-counter" state machine with eight states, S0–S7. Besides CLOCK, your machine should have two inputs, RESET and ENABLE, and one output, DONE. The machine should go to state S0 when RESET is asserted. When RESET is negated, it should move to next-numbered state only if ENABLE is asserted. However, once it reaches state S7, it should stay there unless RESET is again asserted. The DONE output should be 1 if and only if the machine is in state S7 and ENABLE is asserted.

9.31 Create an unambiguous state diagram or an ASM chart for a state machine with one input X and one Moore-type output EDGE, which detects transitions on X. The machine tests its X input at each tick of the clock and asserts EDGE if the value of X at that tick is different from the value at the previous tick. Use state names A, B, C, and so on as needed. Also create a state and output table corresponding to the state diagram.

9.32 Create an unambiguous state diagram or an ASM chart for a state machine with two inputs X and INIT and two Moore-type outputs EDGE and MISS, which dependably detects transitions on X. The machine tests its X input at each tick of the clock and asserts EDGE if the value of X at that tick is different from the value at the previous tick. Once it is asserted, EDGE remains asserted until INIT is asserted for at least one tick. The MISS output is asserted if one or more edges were missed prior to INIT being asserted after EDGE was asserted, and it also remains asserted until INIT is asserted.

9.33 In many applications, the outputs produced by a state machine during or shortly after reset are irrelevant, as long as the machine begins to behave correctly a short time after the reset signal is removed. If this idea is applied to Table 9-4, the INIT state can be removed and only two state variables are needed to code the remaining four states. Redesign the state machine using this idea. Write a new state table and a transition/excitation table for D flip-flops. Derive the excitation and output equations; you may do this algebraically or embed the new transition/excitation table in a Verilog module like Program 9-1 and find the equations in the synthesis results. Draw a logic diagram for the state machine using D flip-flops and discrete gates, assuming that flip-flops have both true and complemented outputs. Compare the cost of the new design (gates and flip-flops) with the minimal-risk design that was completed in Section 9.3.4.

9.34 The output of a *finite-memory machine* is completely determined by its current input and its inputs and outputs during the previous *n* clock ticks, where *n* is a finite, bounded integer. For example, Figure X9.34 shows the realization of a finite-memory machine with one input and one output. Note that a finite-*state* machine need not be a finite-*memory* machine; for example, a modulo-*n* counter with an enable input and a “MAX” output has only *n* states, but its output may depend on the value of the enable input at every clock tick since initialization.

finite-memory machine

Can the state-machine example of Section 9.3.1 be realized as a finite-memory machine? If so, state how many flip-flops are required and show how they are arranged, and if not, describe changes to the machine's description that would allow a finite-memory realization.

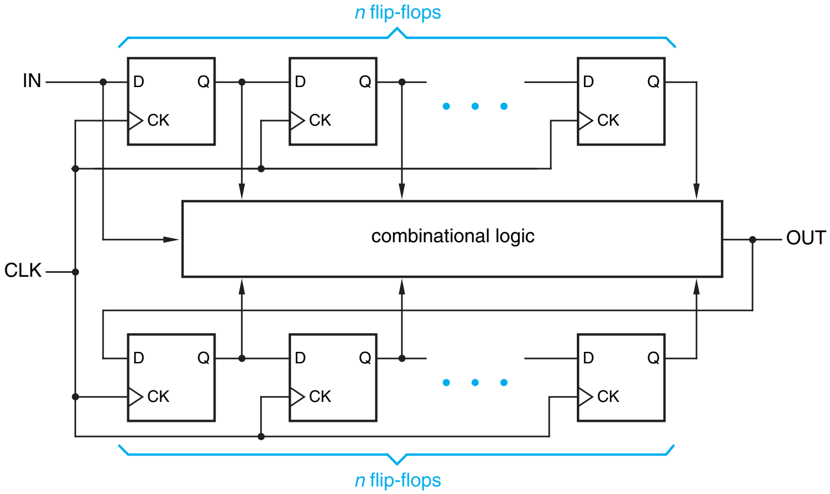


Figure X9.34

9.35 Synthesize a circuit for the ambiguous state diagram in Figure 9-24. Use the state assignment in Table 9-7. Write a transition list, a transition equation for each state variable as a sum of p-terms, and simplified transition/excitation equations for a realization using D flip-flops. Determine the actual next state of the circuit, starting from the IDLE state, for each of the following input combinations on (LEFT, RIGHT, HAZ): (1,0,1), (0,1,1), (1,1,0), (1,1,1). Comment on the machine's behavior in these cases.

9.36 What does the personalized license plate in Figure 9-22 stand for? (*Hint:* It's the author's old plate, a computer engineer's version of OTTFFSS.)

9.37 Suppose that for a state SA and an input combination I, an ambiguous state diagram indicates that there are two next states, SB and SC. The actual next state SD for this transition depends on the state machine's realization. If the state machine is synthesized using the method of Section 9.4 (V∗ = sum of p-terms where V∗ = 1) to obtain transition/excitation equations for D flip-flops, what is the relationship between the coded states for SB, SC, and SD? Explain.

9.38 Repeat Exercise 9.37, where the ambiguous state diagram specifies *no* next state for state SA and input combination I. What is the coding of the actual next state SD for this transition?

9.39 In state-machine synthesis method of Section 9.4, if there are fewer 0s than 1s in the transition-list column for a particular variable $V*$, it may be easier to derive the *complement* of that variable's transition equation, that is, $V*' =$ sum of p-terms where $V* = 0$. Explain why this method works.

9.40 Repeat Exercises 9.37 and 9.38, assuming that the machine is synthesized using the method ($V*' =$ sum of p-terms where $V* = 0$) for all state variables.

9.41 Suppose that for a state SA and an input combination I, an ambiguous state diagram does not define a next state. The actual next state SD for this transition depends on the state machine's realization. Suppose that the state machine is synthesized using the method ($V* = \Sigma$ p-terms where $V* = 1$) to obtain transition/excitation equations for D flip-flops. What coded state is SD? Explain.

9.42 Repeat Exercise 9.41, assuming that the machine is synthesized using the method ($V*' = \Sigma$p-terms where $V* = 0$).

9.43 Using the style of Program 9-2, write a Verilog module corresponding to the state diagram of Figure 9-25. Write a test bench that graphically displays the state machine's output sequence for a typical input sequence. *Suggestion*: a sequence of lamp states may be displayed using an "O" or "." for each lamp depending on whether it's on or off; for example, in a left turn,

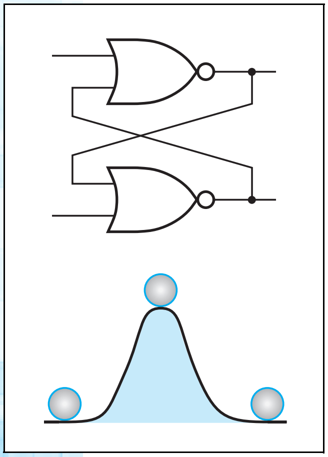
```
... ...
..O ...
.OO ...
OOO ...
```

9.44 Using the style of Program 9-2, write a Verilog module based on the state diagram in Drill 9.14. Use the "simplest" state assignment.

9.45 Update the Verilog test bench from Exercise 9.17 to include a function `writeS` to print state names as `A`–`P` instead of `Q0`–`Qa`. Test the test bench using the state machine in the same exercise.

9.46 Write a Verilog module corresponding to the excitation logic in the state machine of Drill 9.12. Use the test bench of Exercise 9.45 to print the machine's state table. Compare with your results from Drill 9.12; if they're different, find and correct your error(s).

9.47 Using the style of Program 9-2, write a Verilog module based on the state diagram of the parity-checking state machine you designed in Drill 9.16. Write a test bench that generates a typical input sequence for the machine, including different word lengths, both even and odd parity, and sometimes gaps between successive words. *Suggestion*: You may find it useful to write a task "`Genser(N,W,P,G)`" that generates an N+1-bit serial pattern of an N-bit data word (`W`) plus odd or even parity (`P`), followed by a gap of 0 or more clock periods (`G`), without having to write out all the details for each different test pattern.



chapter 10

Sequential Logic Elements

In the preceding chapter, we introduced edge-triggered D flip-flops, which are the most commonly used elements for storing the state in state machines. But there are other types of storage elements as well, ones that are more convenient or efficient for non-state-machine applications. These include latches, which can be used to capture a condition or information based on the level, not an edge, of a control input, with about half of the cost of an edge-triggered flip-flop in terms of circuit area. There are also edge-triggered devices that have multiple control inputs that are useful in some applications. Finally, most elements also have variants with separate initialization (reset) inputs to force their state to a desired value when the circuit starts up, without regard to activity on the other inputs.

In this chapter, we'll start with the very simplest sequential element and work our way up to the more complex ones. In addition to functional behavior, timing is very important for sequential elements, so we'll take a close look at both the timing requirements on their inputs and the timing behaviors of their outputs. In preparation for actually using sequential elements in larger circuits, we'll see how they may be grouped together and combined with other elements in building-block components, FPGAs, and PLDs, and how they can be invoked explicitly from component libraries or "inferred" by behavioral HDL code. We'll close the chapter with an optional section on "feedback sequential circuits," which helps to explain how these sequential elements operate internally.

10.1 Bistable Elements

The simplest sequential circuit consists of a pair of inverters forming a feedback loop, as shown in Figure 10-1. It has *no* inputs and two outputs, Q and Q_L.

10.1.1 Digital Analysis

bistable

The circuit of Figure 10-1 is often called a *bistable*, since a strictly digital analysis shows that it has two stable states. If Q is HIGH, then the bottom inverter has a HIGH input and a LOW output, which forces the top inverter's output HIGH as we assumed in the first place. But if Q is LOW, then the bottom inverter has a LOW input and a HIGH output, which forces Q LOW, another stable situation. We could use a single state variable, the state of signal Q, to describe the state of the circuit; there are two possible states, Q = 0 and Q = 1.

The bistable element is so simple that it has no inputs and therefore no way of controlling or changing its state. When power is first applied to the circuit, it randomly comes up in one state or the other and stays there forever. Still, it serves our illustrative purposes very well, and we *will* actually show a common application for a variant of it in Section 10.5.2.

10.1.2 Analog Analysis

transfer function

voltage transfer diagram

gain

The analysis of the bistable has more to reveal if we consider its operation from an analog point of view. For the non-EE reader, we first introduce the notion of the *transfer function* of a 1-input, 1-output analog circuit: it is a mathematical function that gives the "steady-state" output voltage produced in response to a given input voltage, that is, the final output voltage that is produced after any dynamic effects have settled out. The transfer function can be calculated by meticulous analog circuit analysis, but for our purposes it will be sufficient to simply plot it on a graph, in what is sometimes called a *voltage transfer diagram*.

Figure 10-2(a) shows the transfer function for a typical CMOS inverter using a 3.0-V power supply. The output voltage plotted on the vertical axis is a function of input voltage, $V_{out} = T(V_{in})$. When the V_{in} is less than about 1 volt, V_{out} is close to 3 volts; and when V_{in} is greater than about 2 volts, V_{out} is close to 0 volts. But when V_{in} is in the "no-man's land" between 1 and 2 volts, the V_{out} goes sharply lower as V_{in} goes higher and vice versa. From the analog point of view, the inverter has very high *gain* in this region of the graph, because a small change in its input voltage creates a much larger change in its output voltage.

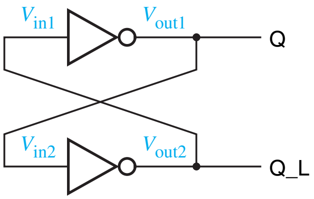


Figure 10-1
A pair of inverters forming a bistable element.

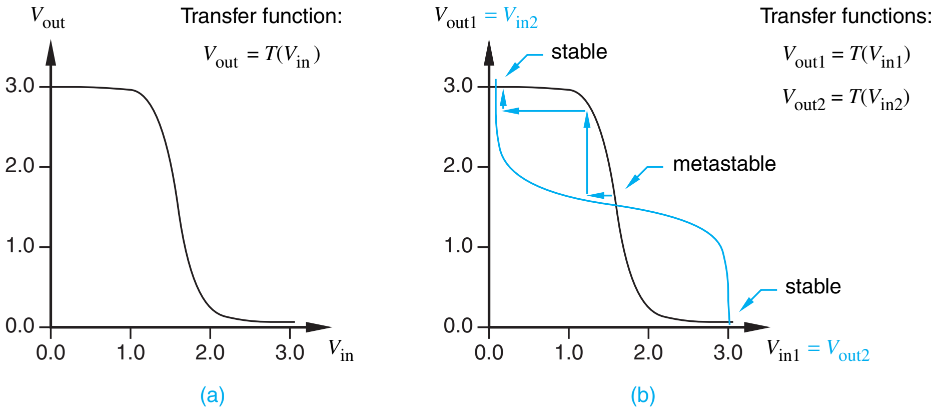


Figure 10-2 Transfer functions: (a) for a single CMOS inverter; (b) for a pair of inverters in a bistable feedback loop.

With two inverters connected in a feedback loop as in Figure 10-1, we know that $V_{in1} = V_{out2}$ and $V_{in2} = V_{out1}$; therefore, we can plot the transfer functions for both inverters on the same graph labeling the axes appropriately. Thus, in Figure 10-2(b), the black line is the transfer function is as before and applies to the top inverter in Figure 10-1. The colored line is the transfer function for the bottom inverter, plotted with its input on the *vertical* axis and its output on the *horizontal*.

Considering only the steady-state behavior of the bistable's feedback loop, and not dynamic effects, the loop is in equilibrium if the input and output voltages of both inverters are constant voltages consistent with the loop connection and the inverters' transfer functions. That is, we must have

$$\begin{aligned} V_{in1} &= V_{out2} \\ &= T(V_{in2}) \\ &= T(V_{out1}) \\ &= T(T(V_{in1})) \end{aligned}$$

Likewise, we must have

$$V_{in2} = T(T(V_{in2}))$$

We can find these equilibrium points graphically from Figure 10-2(b); they are the points at which the two transfer curves meet. Surprisingly, we find that there are not two but *three* equilibrium points. Two of them, labeled *stable*, correspond to the two states that our "strictly digital" analysis identified earlier, with Q either 0 (LOW) or 1 (HIGH).

stable

The third equilibrium point, called a *metastable state*, occurs with V_{out1} and V_{out2} about halfway between a valid logic 1 voltage and a valid logic 0 voltage; so Q and Q_L are not valid logic signals at this point. Yet the loop equations are satisfied; if we can get the circuit to operate at the metastable point, it could theoretically stay there indefinitely. This behavior is called *metastability*.

metastable state

metastability

10.1.3 Metastable Behavior

Closer analysis of the situation at the metastable point shows that it is aptly named. It is not truly stable, because random noise will tend to drive a circuit that is operating at the metastable point toward one of the stable operating points, as we'll now demonstrate.

Suppose the bistable is operating precisely at the metastable point in Figure 10-2(b). Now let us assume that a small amount of circuit noise reduces $V_{\text{in}1}$ by a tiny amount. This change causes $V_{\text{out}1}$ to *increase* by an even greater amount because of the inverter's high gain in this region. Since $V_{\text{out}1}$ produces $V_{\text{in}2}$, we can follow the first horizontal arrow from near the metastable point to the second transfer characteristic, which now demands a lower voltage for $V_{\text{out}2}$, which is $V_{\text{in}1}$. Now we're back where we started, except we have a much larger change in voltage at $V_{\text{in}1}$ than the original noise produced, exacerbated by the high gain of both inverters, and the operating point is still changing. This "regenerative" process continues until we reach the stable operating point at the upper lefthand corner of Figure 10-2(b). However, if we perform such a "noise" analysis for either of the stable operating points, we find that feedback brings the circuit back towards the stable operating point, rather than away from it.

Metastable behavior of a bistable can be compared to the behavior of a ball dropped onto a hill, as shown in Figure 10-3. If we drop a ball from overhead, it will probably roll down immediately to one side of the hill or the other. But if it lands right at the top, it may sit there precariously for a while before random forces (wind, rodents, earthquakes) start it rolling down the hill. Like the ball at the top of the hill, the bistable may stay in the metastable state for an unpredictable length of time before randomly settling into one stable state or the other.

If the *simplest* sequential circuit is susceptible to metastable behavior, you can be sure that *all* sequential circuits are susceptible. And this behavior is not something that only occurs at power-up.

Returning to the ball-and-hill analogy, consider what happens if we try to kick the ball from one side of the hill to the other. Apply a strong force (Superman), and the ball goes right over the top and lands in a stable resting place on the other side. Apply a weak force (Clark Kent), and the ball falls back to its original starting place. But apply a wishy-washy force (Charlie Brown), and the ball goes to the top of the hill, teeters, and eventually falls back to one side or the other.

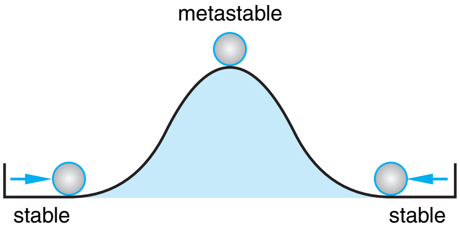


Figure 10-3
Ball and hill analogy for metastable behavior.

This behavior is completely analogous to what happens to latches and flip-flops under marginal triggering conditions. For example, we'll soon study S-R latches, where a pulse on the S input forces the latch from the 0 state to the 1 state. A minimum pulse width is specified for the S input. Apply a pulse of this width or longer, and the latch immediately goes to the 1 state. Apply a very short pulse, and the latch stays in the 0 state. Apply a pulse just under the minimum width, and the latch may go into the metastable state. Once the latch is in the metastable state, its operation depends on "the shape of its hill." Latches and flip-flops built from high-gain, fast transistors tend to come out of metastability faster than ones built from low-performance technologies.

We'll say more about metastability in the next section in connection with specific latch and flip-flop types, and in Section 13.4 with respect to synchronous design methodology and synchronizer failure.

10.2 Latches and Flip-Flops

Latches and flip-flops are the basic building blocks of most sequential circuits. The latches and flip-flops in typical digital systems are functionally specified devices already prepackaged in a standard integrated circuit. In ASIC design environments, they are typically predefined cells specified by the ASIC vendor. However, within a standard IC or an ASIC, each predefined latch or flip-flop cell typically has been designed as a feedback sequential circuit using individual logic gates and feedback loops. We'll look at such discrete designs here to better understand the behavior of the prepackaged elements.

flip-flop

latch

All digital designers use the name *flip-flop* for a sequential device that normally samples its inputs and changes its outputs only when a clocking signal is changing. On the other hand, most digital designers use the name *latch* for a sequential device that watches its inputs continuously and can change its outputs at any time (although in some cases requiring an enable signal to be asserted). We follow this standard convention in this text. However, some textbooks and digital designers may incorrectly use the name "flip-flop" for a device that we call a "latch."

In any case, because the functional behaviors of latches and flip-flops are quite different, it is important for the logic designer to know which type is being used in a design, from the device's part number (e.g., 74x373 vs. 74x374), from the FPGA or ASIC library-element name (e.g., TLAT vs. DFF), or from other contextual information. We will discuss the most commonly used latch and flip-flop types in the following subsections.

10.2.1 S-R Latch

S-R latch

The simplest sequential circuit that has control inputs can be built from just two 2-input NOR gates as shown in Figure 10-4(a). This is called an *S-R (set-reset) latch*. The circuit has two inputs, S and R, and two outputs, labeled Q and QN,

$\overline{\text{Q}}$ VERSUS QN In most applications of an S-R latch, the QN (a.k.a. $\overline{\text{Q}}$) output is always the complement of the Q output. However, the $\overline{\text{Q}}$ name is not quite correct, because there is one case where this output is not the complement of Q. If both S and R are 1, as they are in several places in Figure 10-5(b), then both outputs are forced to 0. Once we negate either input, the outputs return to complementary operation as usual. However, if we negate both inputs simultaneously, the latch goes to an unpredictable next state, and it may in fact oscillate or enter the metastable state. Metastability may also occur if a 1 pulse that is too short is applied to S or R.

where QN is normally the complement of Q. Signal QN is sometimes labeled $\overline{\text{Q}}$ or Q_L. An S-R latch is typically used to detect an event, using the S input to "set" the latch when the event occurs, and using R to "reset" it later.

If S and R are both 0, the circuit behaves like the bistable element—we have a feedback loop that retains one of two logic states, Q = 0 or Q = 1. As shown in Figure 10-4(b), either S or R may be asserted to force the feedback loop to a desired state. S *sets* or *presets* the Q output to 1; R *resets* or *clears* the Q output to 0. After the S or R input is negated, the latch remains in the same state. Figure 10-5(a) shows the functional behavior of an S-R latch for a typical sequence of inputs. Colored arrows indicate causality, that is, which input transitions cause which output transitions.

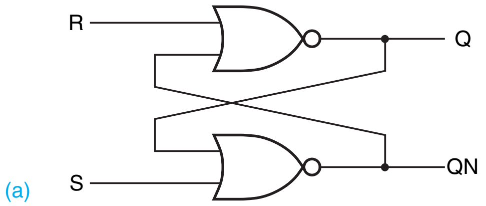
set

preset *reset*

clear

Four different logic symbols for the same S-R latch circuit are shown in Figure 10-6. The symbols differ in the treatment of the complemented output. Historically, the first symbol (a) was used, showing the active-low or comple-

Figure 10-4 S-R latch: (a) circuit design using NOR gates; (b) function table.



(b)

S	R	Q	QN
0	0	last Q	last QN
0	1	0	1
1	0	1	0
1	1	0	0

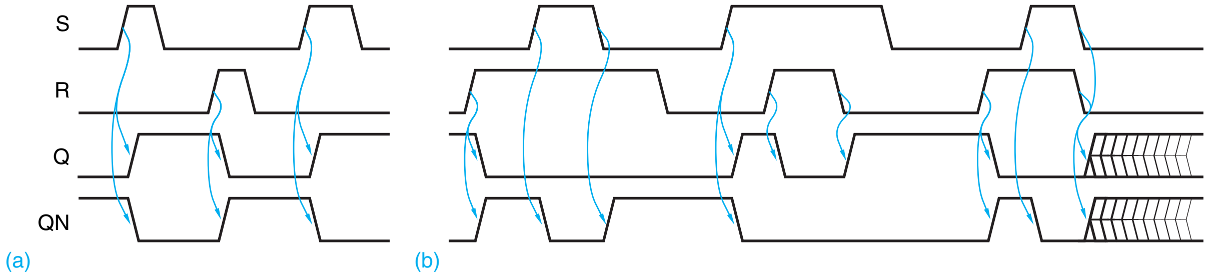


Figure 10-5 Typical operation of an S-R latch: (a) "normal" inputs; (b) S and R asserted simultaneously.

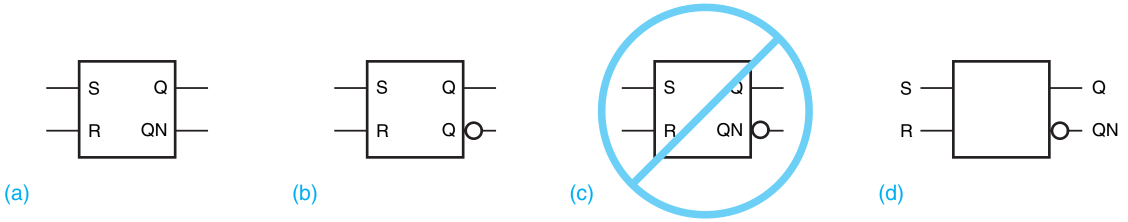


Figure 10-6 Symbols for S-R latch: (a) without bubble; (b) preferred for bubble-to-bubble design; (c) incorrect because of double negation; (d) standalone symbol for an ASIC cell.

mented signal name inside the function rectangle. However, in bubble-to-bubble logic design the second form (b) of the symbol is preferred, showing an inversion bubble outside the function rectangle. The third form (c) of the symbol is wrong because the bubble negates the already negated QN output. The fourth form (d) appears in some ASIC libraries' standalone descriptions of the same circuit. It does not follow the usual convention of placing the input and output names *inside* the rectangle, but it is technically correct; think of it as equivalent to (a) but with the signal names written on the outside of a somewhat more evocative symbol shape.

propagation delay

Figure 10-7 defines timing parameters for an S-R latch. The *propagation delay* is the time it takes for a transition on an input signal to produce a transition on an output signal. A given latch or flip-flop may have several different propagation-delay specifications, one for each pair of input and output signals. Also, the propagation delay may be different depending on whether the output makes a LOW-to-HIGH or HIGH-to-LOW transition. With an S-R latch, a LOW-to-HIGH transition on S can cause a LOW-to-HIGH transition on Q, so a propagation delay $t_{\mathrm{pLH(SQ)}}$ occurs, as shown in transition 1 in the figure. Similarly, a LOW-to-HIGH transition on R can cause a HIGH-to-LOW transition on Q, with propagation delay $t_{\mathrm{pHL(RQ)}}$ as shown in transition 2. Not shown in the figure are the corresponding transitions on QN, which would have propagation delays $t_{\mathrm{pHL(SQN)}}$ and $t_{\mathrm{pLH(RQN)}}$.

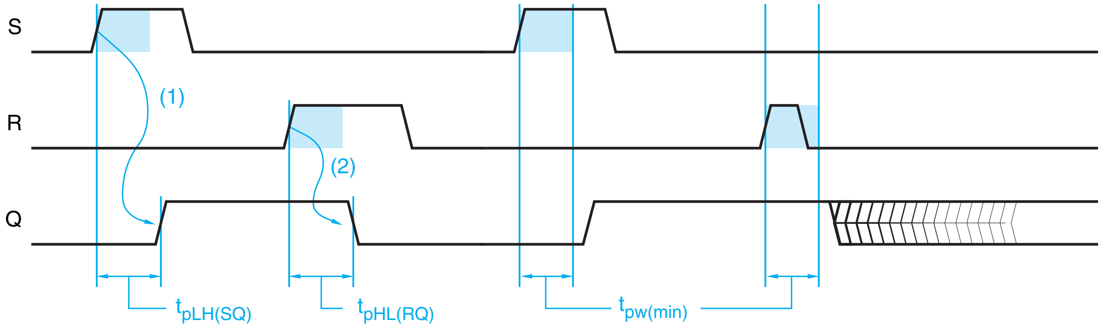


Figure 10-7 Timing parameters for an S-R latch.

HOW CLOSE IS CLOSE?

As mentioned in the previous boxed comment, an S-R latch may go into the metastable state if S and R are negated simultaneously. Often, but not always, a commercial latch's specifications define "simultaneously" (e.g., S and R negated within 5 ns of each other). This parameter is sometimes called the *recovery time*, t_{rec}. It is the minimum delay between negating S and R for them to be considered nonsimultaneous and it is closely related to the minimum-pulse-width specification. Both specifications are measures of how long it takes for the latch's feedback loop to stabilize during a change of state.

minimum pulse width

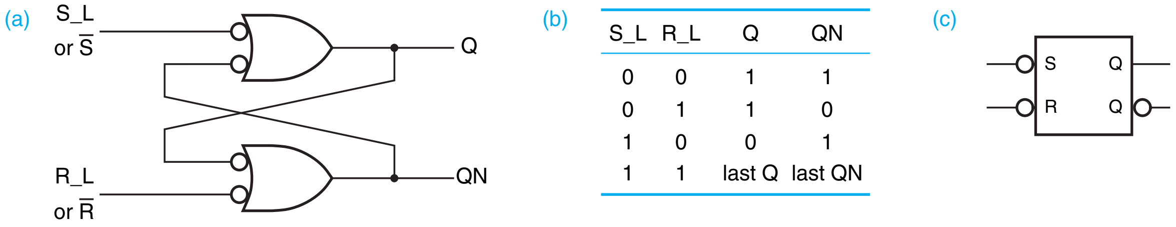
Minimum-pulse-width specifications are usually given for the S and R inputs. As shown in Figure 10-7, the latch may go into the metastable state and it may remain there for a random length of time if a pulse shorter than the minimum width $t_{\text{pw(min)}}$ is applied to S or R. Think of it as a wish-washy kick in the ball-and-hill analogy. The latch can be deterministically brought out of the metastable state only by applying a pulse to S or R that meets or exceeds the minimum-pulse-width requirement.

10.2.2 $\overline{\text{S}}$-$\overline{\text{R}}$ Latch

$\overline{S}$-$\overline{R}$ latch

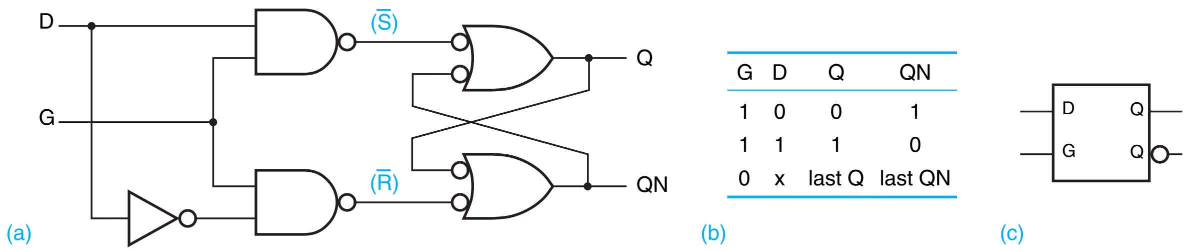
An *$\overline{S}$-$\overline{R}$ latch* (read "S-bar-R-bar latch"), which has active-low set and reset inputs, may be built from NAND gates as shown in Figure 10-8(a). In many CMOS logic families and ASIC libraries, $\overline{\text{S}}$-$\overline{\text{R}}$ latches are used more often than S-R latches because NAND gates are preferred over NOR gates for reasons of speed, size, or both.

As shown by the function table, Figure 10-8(b), operation of the $\overline{\text{S}}$-$\overline{\text{R}}$ latch is similar to that of the S-R, with two major differences. First, $\overline{\text{S}}$ and $\overline{\text{R}}$ are active low, so the latch remembers its previous state when $\overline{\text{S}} = \overline{\text{R}} = 1$; the active-low inputs are clearly indicated in the symbol in (c). Second, when both $\overline{\text{S}}$ and $\overline{\text{R}}$ are asserted simultaneously, both latch outputs go to 1, not 0 as in the S-R latch. Except for these differences, operation of the $\overline{\text{S}}$-$\overline{\text{R}}$ is the same as the S-R, including timing and metastability considerations.



S_L	R_L	Q	QN
0	0	1	1
0	1	1	0
1	0	0	1
1	1	last Q	last QN

Figure 10-8 $\overline{\text{S}}$-$\overline{\text{R}}$ latch: (a) circuit design using NAND gates; (b) function table; (c) logic symbol.



G	D	Q	QN
1	0	0	1
1	1	1	0
0	x	last Q	last QN

Figure 10-9 D latch: (a) circuit design using NAND gates; (b) function table; (c) logic symbol.

10.2.3 D Latch

S-R latches are useful in control applications, where we often think in terms of setting a flag in response to some condition and resetting it when the condition changes. So, we control their set and reset inputs somewhat independently. However, we often need latches simply to store bits of information—each bit is arrives on a signal line, and we'd like to store it somewhere. A *D latch* may be used in such a "data" application.

D latch

Figure 10-9 shows a D latch. The righthand side of its logic diagram is just an $\overline{S}$-$\overline{R}$ latch. Two additional NAND gates have been provided on the lefthand side so that either $\overline{S}$ or $\overline{R}$ is asserted when the control input G is asserted, depending on the value of the single data input D. This eliminates the troublesome situation in S-R latches where S and R may be asserted simultaneously. The control input G is sometimes named ENABLE, CLK, or C. It is active low in some D-latch designs, and it always has a minimum pulse-width requirement.

An example of a D latch's functional behavior is given in Figure 10-10. When the G input is asserted, the Q output follows the D input. In this situation, the latch is said to be "open" and the view from the Q output back to the D input is "transparent"; the circuit is often called a *transparent latch* for this reason. When the G input is negated, the latch "closes"; the Q output retains its last value and no longer changes in response to D, as long as G remains negated.

transparent latch

More detailed timing behavior of the D latch is shown in Figure 10-11. Four different delay parameters are shown for signals that propagate from the G

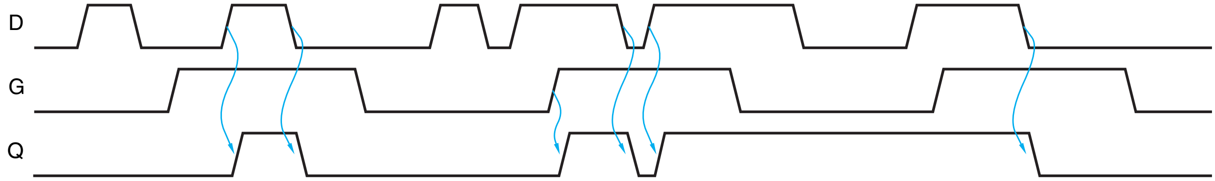


Figure 10-10 Functional behavior of a D latch for various inputs.

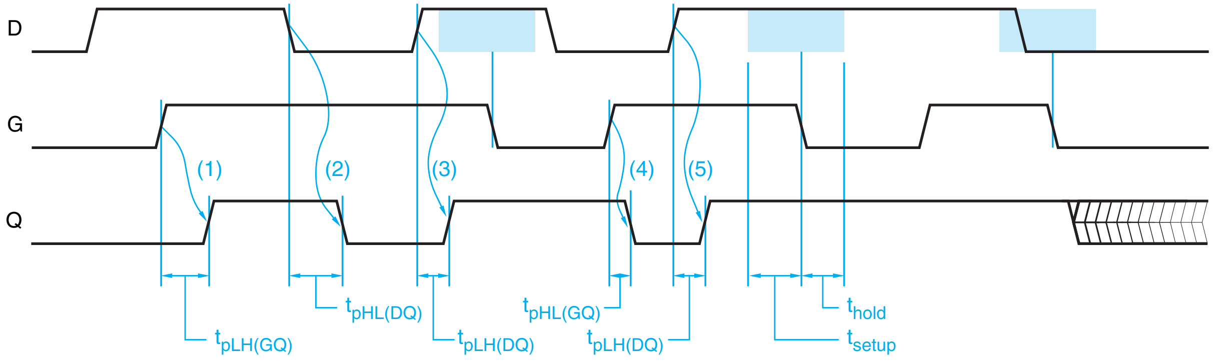


Figure 10-11 Timing parameters for a D latch.

or D input to the Q output. For example, at transitions 1 and 4, the latch is initially "closed" and the D input is the opposite of Q output, so that when G goes to 1 the latch "opens up" and the Q output changes after delay $t_{\mathrm{pLH(GQ)}}$ or $t_{\mathrm{pHL(GQ)}}$. At transitions 2 and 3, the G input is already 1 and the latch is already open, so that Q transparently follows the transitions on D with delay $t_{\mathrm{pHL(DQ)}}$ and $t_{\mathrm{pLH(DQ)}}$. Four more parameters specify the delay to the QN output, not shown.

setup time

hold time

Although the D latch eliminates the S = R = 1 problem of the S-R latch, it does not eliminate the metastability problem. As shown in Figure 10-11, there is a (shaded) window of time around the falling edge of G when the D input must not change. This window begins at time t_{setup} before the falling (latching) edge of G; t_{setup} is called the *setup time*. The window ends at time t_{hold} afterward; t_{hold} is called the *hold time*. If D changes at any time during the setup- and hold-time window, the output of the latch is unpredictable and may become metastable, as shown for the last latching edge in the figure.

10.2.4 Edge-Triggered D Flip-Flop

positive-edge-triggered D flip-flop

We introduced the *positive-edge-triggered D flip-flop* in Chapter 9 as the most commonly used sequential element for storing the state variables of a state machine. A D flip-flop need not be part of a formal state machine; it may also be used simply to store a bit of "data." It is distinguished from a D latch by its edge-triggered behavior: it samples its D input and changes its Q and QN outputs only at the rising (positive) edge of a controlling CLK signal.

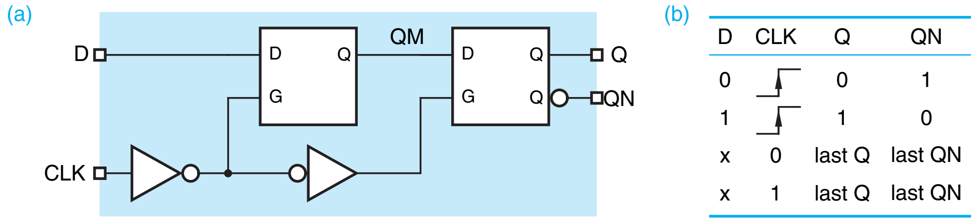
master

slave

Figure 10-12 shows how a positive-edge-triggered D flip-flop can be built from a pair of D latches. The first latch is called the *master*; it is open and follows the input when CLK is 0. When CLK goes to 1, the master latch is closed and its output is transferred to the second latch, called the *slave*. The slave latch is open all the while that CLK is 1, but changes only at the beginning of this interval, because the master is closed and unchanging during the rest of the interval.

dynamic-input indicator

The triangle on the D flip-flop's CLK input indicates edge-triggered behavior and is called a *dynamic-input indicator*. Examples of the flip-flop's



D	CLK	Q	QN
0	↑	0	1
1	↑	1	0
x	0	last Q	last QN
x	1	last Q	last QN

Figure 10-12 Positive-edge-triggered D flip-flop: (a) circuit design using D latches; (b) function table; (c) logic symbol.

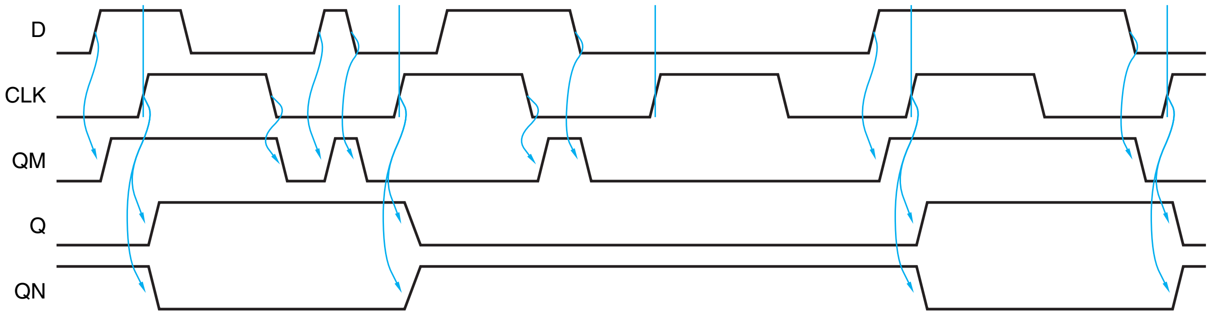


Figure 10-13 Functional behavior of a positive-edge-triggered D flip-flop.

functional behavior for several input transitions are shown in Figure 10-13. The QM signal shown is the output of the master latch. Notice that QM changes only when CLK is 0. When CLK goes to 1, the current value of QM is transferred to Q, and QM is prevented from changing until CLK goes to 0 again.

Figure 10-14 shows more detailed timing behavior for the D flip-flop. All propagation delays are measured from the rising edge of CLK, since that's the only event that causes an output change. Different delays may be specified for LOW-to-HIGH and HIGH-to-LOW output changes.

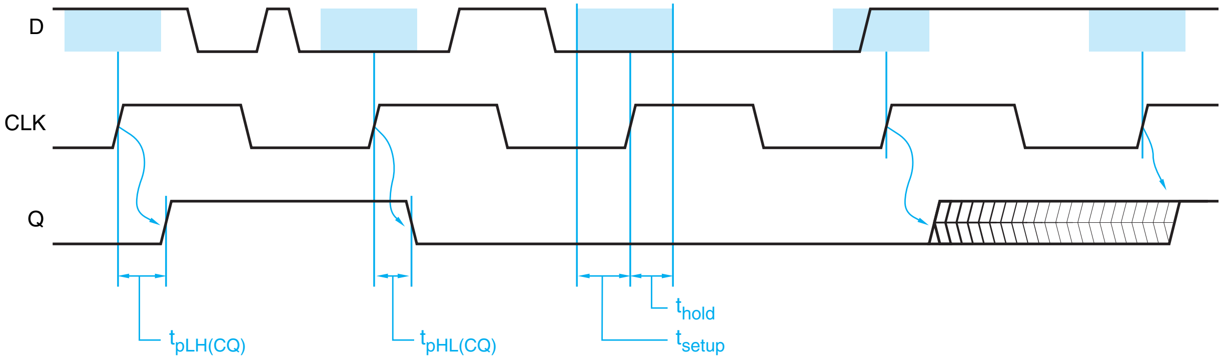
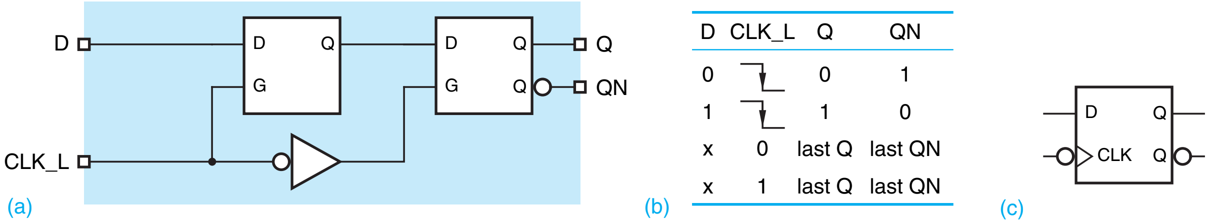


Figure 10-14 Timing behavior of a positive-edge-triggered D flip-flop.



D	CLK_L	Q	QN
0	↓	0	1
1	↓	1	0
x	0	last Q	last QN
x	1	last Q	last QN

Figure 10-15 Negative-edge triggered D flip-flop: (a) circuit design using D latches; (b) function table; (c) logic symbol.

Like a D latch, the edge-triggered D flip-flop has a setup- and hold-time window during which the D inputs must not change. This window occurs around the triggering edge of CLK, and is indicated by shaded color in Figure 10-14. If the setup and hold times are not met, the flip-flop output will usually go to a stable, though unpredictable, 0 or 1 state. In some cases, however, the output will oscillate or go to a metastable state halfway between 0 and 1, as shown at the second-to-last clock tick in the figure. If the flip-flop goes into the metastable state, it will return to a stable state on its own only after a probabilistic delay, as explained in Section 13.4. It can also be forced into a stable state by applying another triggering clock edge with a D input that meets the setup- and hold-time requirements, as shown at the last clock tick in the figure.

negative-edge-triggered D flip-flop

A *negative-edge-triggered D flip-flop* simply inverts the clock input, so that all the action takes place on the falling edge of CLK_L; by convention, a falling-edge trigger is considered to be active low. This flip-flop's function table and logic symbol are shown in Figure 10-15.

asynchronous inputs
preset
clear

Some D flip-flops have *asynchronous inputs* that may be used to force the flip-flop to a particular state independent of the CLK and D inputs. These inputs, typically labeled PR (*preset*) and CLR (*clear*), behave like the set and reset

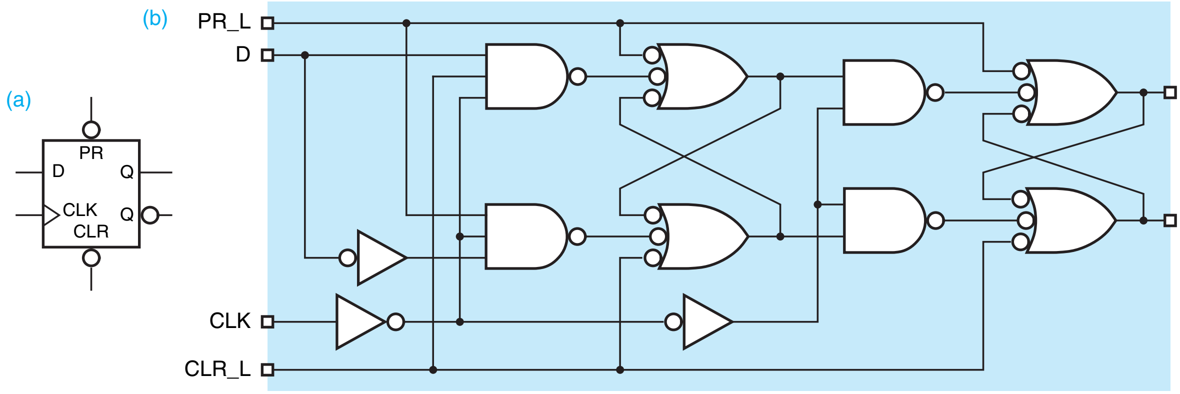


Figure 10-16 Positive-edge-triggered D flip-flop with preset and clear: (a) logic symbol; (b) circuit design using NAND gates.

HOLD-TIME VIOLATIONS

If a flip-flop or latch input does not satisfy its timing requirement relative to another input such CLK or G, that's called a *setup-* or *hold-time violation*. Microprocessor architect John Chu has written a nice science-fiction short story with the same title as this box; you can easily find the concept woven into it (see Exercise 10.59).

inputs on an S-R latch. The logic symbol and a NAND-based circuit for an edge-triggered D flip-flop with these inputs is shown in Figure 10-16. Although some logic designers use asynchronous inputs to perform tricky sequential functions, they are best reserved for initialization and testing purposes only, to force a clocked circuit into a known starting state; we'll have more to say about this when we discuss synchronous design methodology in Section 13.2.

10.2.5 Edge-Triggered D Flip-Flop with Enable

A commonly desired function in D flip-flops is the ability to hold the last value stored, rather than load a new value, at the clock edge. This is accomplished by adding an *enable input*, called EN or CE (*clock enable*). While the name "clock enable" may be descriptive, the extra input's function usually is not obtained by controlling the clock signal in any way whatsoever. Rather, as shown in Figure 10-17(a), a 2-input multiplexer controls the value applied to the internal flip-flop's D input. If EN is asserted, the external D input is selected; if EN is negated, the flip-flop's current output is used. The resulting function table is shown in (b). The flip-flop symbol is shown in (c); in some flip-flops, the enable input is active low, denoted by an inversion bubble on the input.

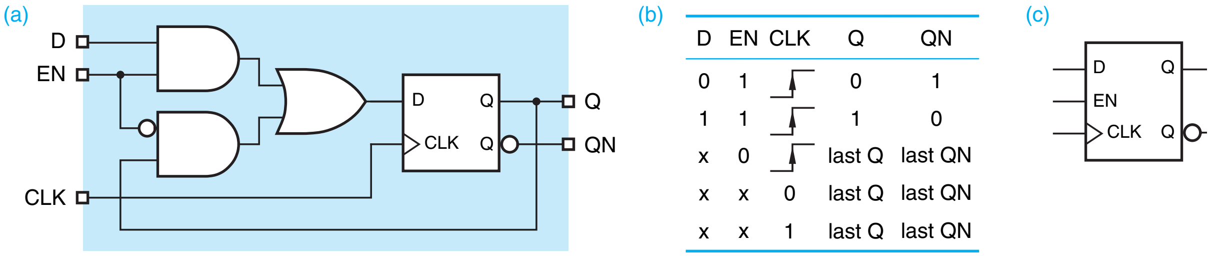
enable input

clock-enable input

10.2.6 T Flip-Flops

A *T (toggle) flip-flop* changes state on every tick of the clock. Figure 10-18 shows the symbol and illustrates the behavior of a positive-edge-triggered T flip-flop. Notice that the signal on the flip-flop's Q output has precisely half the frequency of the T input. Figure 10-19(a) shows how to obtain a T flip-flop from a D flip-flop. T flip-flops are most often used in counters and frequency dividers, which we'll describe in Section 11.1.

T flip-flop



D	EN	CLK	Q	QN
0	1	↑	0	1
1	1	↑	1	0
x	0	↑	last Q	last QN
x	x	0	last Q	last QN
x	x	1	last Q	last QN

Figure 10-17 Positive-edge-triggered D flip-flop with enable: (a) circuit design; (b) function table; (c) logic symbol.

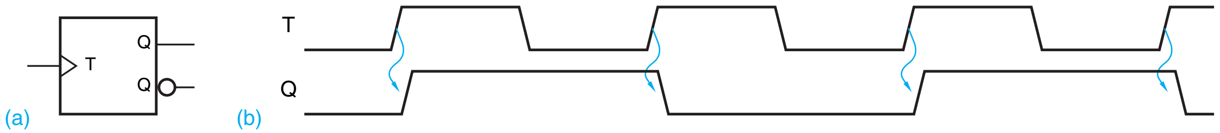
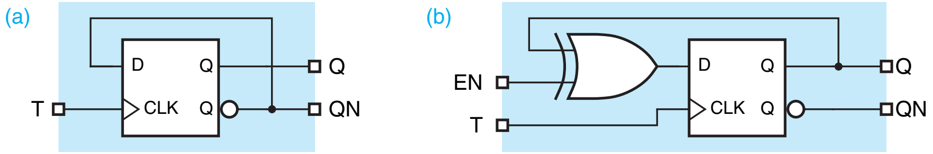


Figure 10-18 Positive-edge-triggered T flip-flop: (a) logic symbol; (b) functional behavior.

Figure 10-19 Circuits for T flip-flops using D flip-flops: (a) basic circuit; (b) with enable.



T flip-flop with enable

In many applications of T flip-flops, the flip-flop need not be toggled on every clock tick. Such applications can use a *T flip-flop with enable*. As shown in Figure 10-20, the flip-flop changes state at the triggering edge of the clock only if the enable signal EN is asserted. Like the D and CE inputs on other edge-triggered flip-flops, the EN input must meet specified setup and hold times with respect to the triggering clock edge. The circuit in Figure 10-19(a) is easily augmented to provide the EN input as shown in (b).

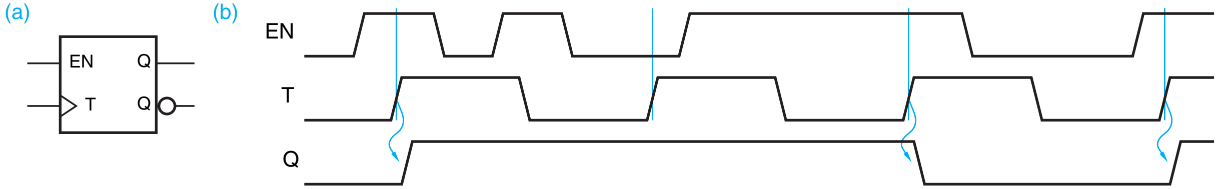


Figure 10-20 Positive-edge-triggered T flip-flop with enable: (a) logic symbol; (b) functional behavior.

10.3 Latches and Flip-Flops in Verilog

There are basically two ways to specify latches and flip-flops in Verilog, and the choice of method depends on the design goals and methodology. If a design is being targeted to an ASIC or other specific implementation technology, the designer may want to ensure that the implementation uses specific flip-flops that are provided in that technology's component library. If the design is being targeted to an FPGA or to a nonspecific technology, the designer will normally specify the flip-flops, as well as most of the rest of the design, behaviorally, and let the compiler "infer" the appropriate components. These two approaches are described below.

A third approach would be to specify a latch or flip-flop by writing code in the structural or dataflow style of Verilog equivalent to the various gate-level implementations that we showed in Section 10.2. This is done so rarely, if ever, that we won't mention it further.

10.3.1 Instance Statements and Library Components

In Section 5.7, we introduced instance statements that are used in the structural style of coding to instantiate Verilog's built-in gates in a design. We also showed how these statements could be used to instantiate a module or component that we designed ourselves. We can also instantiate a module from a library of components provided by a supplier or designed by a colleague.

Depending on the design environment and the component, we may or may not have to do anything special in our Verilog code for the compiler to "find" a component that's named in an instance statement. For example, the Xilinx Vivado tool automatically searches its built-in "UNISIM" library for any component name that is not already defined in one of the user's modules. In other environments, or for other libraries, a module that uses a library component must specify the path and filename of the component's definition, typically using an `` `include `` compiler directive, for example,

```
`include "C:/Xilinx/Vivado/2016.2/ids/ISE/verilog/src/unisims/LDC.v"
```

In the Xilinx environment, the base filename for a component definition is always the same as the component name (`LDC` in this example), followed by the `.v` extension. Once a definition has been "included," the component can be named in an instance statement using one of the formats in Table 5-16 on page 199, for example,

```
  LDC U1 (.G(myG), .D(myD), .CLR(myCLR), .Q(myQ) );
```

A typical library provided by an ASIC supplier has many different latch and flip-flop components. Most of types described in the Section 10.2 are usually offered, including variations that may have fewer or additional inputs—such as a D flip-flop with a PR input but no CLR, and vice versa. When your structural Verilog code is targeted to an ASIC, an instantiation like the example above may yield a gate- or transistor-level realization that has exactly the specified functionality, and no more.

Table 10-1 lists some of the latches and flip-flops in three different component libraries. Different components may have different inputs and functions, such as asynchronous clear and preset, synchronous set and reset, and synchronous clock enabling. "Asynchronous" means that asserting the input has the stated effect at all times; "synchronous" means that it has an effect only if it's asserted at the triggering clock edge. Note that the signal names and nomenclature for similar functions, like "set" vs. "preset," may vary among suppliers.

The first two columns of the table describe components in the "unified" library that Xilinx provides with their ISE 8.1 tools, which are used with their large catalog of FPGAs and CPLDs including many older "legacy" parts. The first column is the component name, as it would appear in a Verilog instance statement, and the second column gives the names of the components' non-clock

Table 10-1 Some latches and positive-edge-triggered flip-flops in Xilinx and LSI Logic libraries.

Xilinx ISE 8.1 Name & Inputs		***Xilinx 7-Series Name & Inputs***		***LSI Logic ASIC Name & Inputs***		***Function***
				LSR0	S,R	S-R latch
LDCE	D,G,GE,CLR	LDCE	D,G,GE,CLR			D latch w/ gate-enable, async clear
LDPE	D,G,GE,PRE	LDPE	D,G,GE,PRE			D latch w/ gate-enable, async preset
				LD1	D,G	D latch
				LD3	D,G,CD	D latch w/ async clear
FD	D			FD1	D	D f-f
FDC	D,CLR			FD2	D,CD	D f-f w/ async clear
FDCP	D,CLR,PRE			FD3	D,CD,SD	D f-f w/ async clear, preset
FDE	D,CE					D f-f w/ clock enable
FDCE	D,CLR,CE	FDCE	D,CLR,CE	FDCE	D,CLR,CE	D f-f w/ async clear, clk enable
FDPE	D,PRE,CE	FDPE	D,PRE,CE			D f-f w/ async preset, clk enable
FDRE	D,R,CE	FDRE	D,R,CE			D f-f w/ sync reset, clk enable
FDSE	D,S,CE	FDSE	D,S,CE			D f-f w/ sync set, clk enable
FDR	D,R			FDS2	D,CR	D f-f w/ sync reset
FDS	D,S					D f-f w/ sync set
FDSRE	D,S,R,CE					D f-f w/ sync set, reset, clk enable
FTC	T,CLR					T f-f w/ enable, async clear
				FT2	CD	T f-f w/ async clear

inputs. In the Xilinx libraries, a flip-flop's clock input is always named "C" and a latch or flip-flop output is always named "Q".

Very few of the component types in the unified library are available "natively" in every Xilinx device—you must consult the Xilinx documentation to determine if a given type is available in a particular device. If it is not, then the synthesis tool will construct a circuit with equivalent functionality, but using more internal device resources to achieve that functionality. For example, to achieve T flip-flop functionality in an FPGA that has only D flip-flops, the synthesis tool would combine a D flip-flop with some combinational logic in the style of Figure 10-19 on page 508.

The next two columns of Table 10-1 show the latch and flip-flop components provided by Xilinx Vivado tools in the UNISIM library for their 7-series FPGAs. In this case, the library provides only flip-flop types that can be synthesized within their configurable logic blocks (CLBs) using "free" resources—without consuming other resources like LUTs, conserving those resources to implement user-specified combinational logic within the same CLB.

WORN-OUT SHOES The LSI Logic library components in the last two columns of Table 10-1 were used to design some pretty old ASICs. I couldn't easily give you a newer library example because nowadays ASIC manufacturers treat their libraries as proprietary IP, and they typically release details, even high-level descriptions like the ones in our table, only to customers.

The last two columns of the table show some latch and flip-flop components from an LSI Logic ASIC library. Each of the listed devices in that library provides a QN as well as a Q output.

Program 10-1 is a Verilog module that instantiates each of the 7-series latch and flip-flop types in Table 10-1, with varying hookups. The two D latches are connected to different G control inputs G1 and G2. The first latch's "gate enable" (GE) input, which is ANDed with the G input to open the latch, is connected to an input signal GE, but the second one's is set to a constant 1. Similarly, two of the four D flip-flops use the GE input signal for their clock-enable (CE) input, while the other two have their clocks always enabled.

All of the components' asynchronous and synchronous clear and preset inputs are connected to a common CLR signal; likewise the preset and set inputs are connected to a common PR signal. At the module level, there are only four unique D inputs, and the two D latches use the same ones as the first two D flip-flops. But each component has a unique Q output—it has to!

Program 10-2 is a test bench for the VrFFandLatches module. It instantiates the module, creates a free-running clock signal Tclk with a 20 ns period, and also generates the other input signals used by the module. The PR and CLR signals are negated early in the test so the effects of other inputs can be observed afterwards. The other inputs—latch enables and data—are changed at regular

Program 10-1 Structural module that instantiates latches and flip-flops.

```
module VrFFandLatches(CLK, D[1:4], G1, G2, GE, CLR, PR, Q[1:6]);
  input CLK, G1, G2, GE, CLR, PR;
  input [1:4] D;
  output [1:6] Q;

  LDCE U1 ( .G(G1), .GE(GE), .D(D[1]), .CLR(CLR), .Q(Q[1]) );
  LDPE U2 ( .G(G2), .GE(1'b1), .D(D[2]), .PRE(PR), .Q(Q[2]) );
  FDCE U3 ( .C(CLK), .CE(1'b1), .D(D[1]), .CLR(CLR), .Q(Q[3]) );
  FDPE U4 ( .C(CLK), .CE(GE), .D(D[2]), .PRE(PR), .Q(Q[4]) );
  FDRE U5 ( .C(CLK), .CE(1'b1), .D(D[3]), .R(CLR), .Q(Q[5]) );
  FDSE U6 ( .C(CLK), .CE(GE), .D(D[4]), .S(PR), .Q(Q[6]) );

endmodule
```

Program 10-2 Test bench for latches and flip-flops module.

```
`timescale 1 ns / 100 ps
module VrFFandLatchTB ();
reg Tclk, G1, G2, GE, CLR, PR, D1, D2;
wire Q1, Q2, Q3, Q4, Q5, Q6;

VrFFandLatches UUT ( .CLK(Tclk), .D1(D1), .D2(D2),
     .G1(G1), .G2(G2), .GE(GE), .CLR(CLR), .PR(PR),
     .Q({Q1,Q2,Q3,Q4,Q5,Q6}) );        // instantiate UUT

always begin // create free-running test clock with 20 ns period
  #0.2 Tclk = 1; #10; // 10 ns high  (tiny offset for
  Tclk = 0; #9.8;     // 10 ns low    waveform readability)
end

always begin                  // Change D1 and D2 on a 15 ns cycle
  #2 D1 = ~D1; D2 = ~D2;  // offset 2 ns from Tclk edges
  #5 D2 = ~D2;
  #5 D1 = ~D1; #3 ;
end

always begin             // Change G1 every 20 ns and G2 every 30 ns,
  #4  G1 = ~G1; G2 = ~G2;  // offset 4 ns
  #20 G1 = ~G1; #10 G2 = ~G2; #10 G1 = ~G1; #16 ;
end

always begin    // Change GE every 60 ns, offset 2 ns
  #2 GE = ~GE; #58 ;
end

initial begin          // Here's what to do starting at time 0
  CLR = 1; PR = 1;     // Apply clear and preset
  D1 = 0; D2 = 1;      // Initialize D inputs for desired waveform
  GE = 0; G1 = 0; G2 = 0;   // Latches are initially closed
  #100                 // Wait 100 ns for FPGA global reset
  #15                  // Nothing should happen yet
  CLR = 0; PR = 0;     // Now negate clear and preset
  #300                 // Run another 300 ns
  $stop(1);            // End test
end
endmodule
```

intervals that are different from each other and different from the clock period, so that a variety of timing situations can be observed. Figure 10-21 shows the waveforms produced by the test bench in a behavioral simulation.

The module was targeted to a Xilinx FPGA which has an internal "global reset" signal that holds all latches and flip-flops in an initial state as the device is powered up and programmed. Although this reset signal does not appear in the

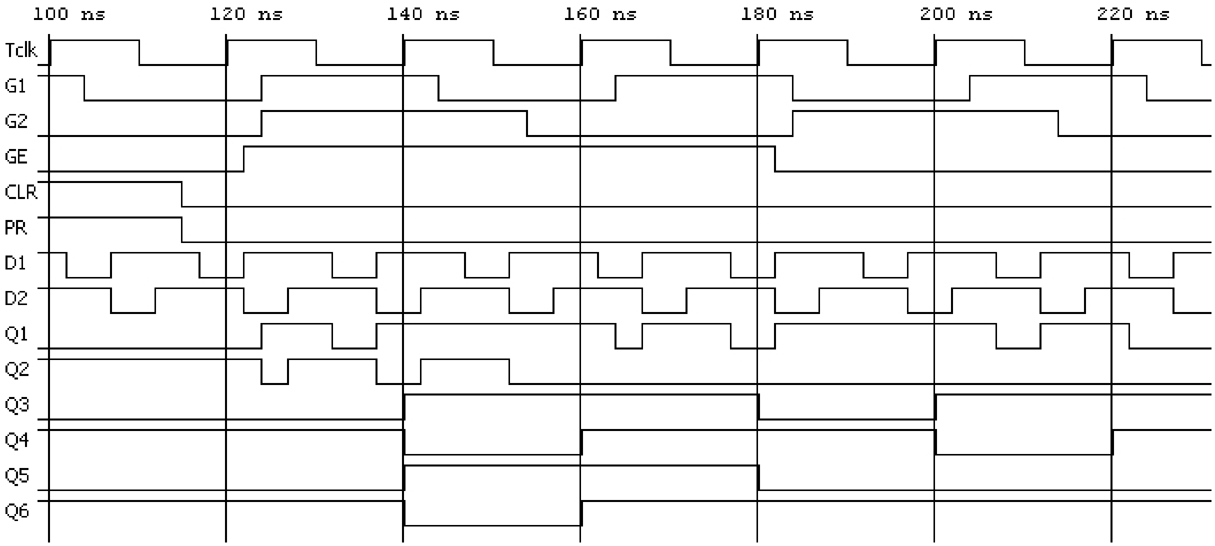


Figure 10-21 Timing waveforms created by test bench in behavioral simulation.

Verilog module, the Vivado tools mimic it by holding all simulated latches and flip-flops in an initial state for the first 100 ns of simulated time. Therefore, the "interesting" part of the waveforms produced by the test bench, as shown in Figure 10-21, begins around 100 ns.

As can be seen in the waveforms, the component outputs Q1–Q6 remain at their initial values until after the PR and CLR signals are negated at 115 ns. The Q1 output, coming from a D latch whose GE input is a constant 1, follows its input D1 as long as G1 is asserted (at 124, 164, and 204 ns), and latches its output when G1 is negated (at 144, 184, and 224 ns). The second latch's output Q2 follows D2 as long as G2 and GE are asserted (at 124 ns), and latches when either is negated (at 154 ns). The Q3–Q6 outputs are from the edge-triggered flip-flops in the module, and therefore change only on a triggering (positive) edge of the Tclk signal, and then only if the corresponding clock enable is asserted (Q3–Q4 have constant 1 clock-enables and Q5–Q6 use GE).

10.3.2 Behavioral Latch and Flip-Flop Models

Latches and flip-flops can be modeled behaviorally in Verilog, and in fact that is the most common method of specifying them. Verilog compilers are designed to recognize very specific coding patterns for these behaviors (see the box on page 518), and the synthesis tools will "infer" either an appropriate component or programmable-device resources to implement each behavior, depending on the targeted technology. In this subsection, we'll look at behavioral modules and coding patterns corresponding to some of the common latch and flip-flop types that we introduced in Section 10.2.

Program 10-3 Behavioral model for a basic D latch.

```
module VrDlatch(D, G, Q);
  input D, G;
  output reg Q;

  always @ (D or G) begin
    if (G==1) Q <= D;
  end
endmodule
```

Program 10-3 gives behavioral Verilog code to model a basic D latch. The output may be affected whenever D or G changes, so those inputs are in the sensitivity list of the always block; we could also have just used "*" as the sensitivity list. Notice that the if statement does not have a corresponding else clause. You may find this disturbing, after writing behavioral Verilog for combinational circuits, and having it beaten into you that conditional statements like if and case should cover all alternatives to avoid generating inferred latches. Well, in the present situation we *want* to infer a latch, so the alternative of G being 0 is left out of the code. The simulator thereby recognizes that Q should not change if G is 0, and the synthesis engine recognizes that a latch is needed here. However, the code still works the same if we include a redundant "else Q<=Q;" clause.

Another way to specify the D latch uses dataflow code, with Q declared as a wire: "assign Q = G ? D : Q;". Most synthesis tools can recognize this pattern as well, but we'll stick with the behavioral version for the rest of our D latches.

The basic code can be augmented as shown in Program 10-4 to create a D latch with asynchronous clear and gate-enable inputs, with the same functionality as the Xilinx LDCE library component. It's obvious from the structure of the if statement that when asserted, the CLR input overrides the other inputs. It's also obvious from the code that the G and GE inputs have equivalent functions—they are ANDed to open the latch. It's just semantics calling one of them the "gate" and the other "gate enable"; they could have been named G1 and G2.

Program 10-4 Model for a D latch with gate enable and asynchronous clear.

```
module VrDlatchCE(D, G, GE, CLR, Q);
  input D, G, GE, CLR;
  output reg Q;

  always @ (D or G or GE or CLR) begin
    if (CLR==1) Q <= 0;
    else if ((G==1)&&(GE==1)) Q <= D;
  end
endmodule
```

Program 10-5 Behavioral code for an *n*-to-*s*-bit decoder with latched outputs.

```
module VrNtoSdec_latch(G, CLR, A, Y);
parameter N=3, S=8;
  input [N-1:0] A;
  input G, CLR;
  output reg [S-1] Y;
  integer i;

  always @ (*) begin
    if (CLR) Y <= 0;
    else if (G) begin
      Y <= 0;
      for (i=0; i<=S-1; i=i+1)
          if (i == A) Y[i] <= 1;
      end
  end
endmodule
```

A nice aspect of the behavioral models is that you can easily specify other logic within the model and combine it with the storage element. For example, suppose you needed an *n*-to-*s* binary decoder like Program 6-7 on page 270, but with a "latching" enable input. The new decoder's outputs `Y[S-1:0]` should decode the inputs when the enable input `G` is asserted. It should maintain the last decoded output values on `Y[S-1:0]` when `G` is negated, and it negates all outputs when a new, `CLR` input is asserted. Program 10-5 provides the required behavior. Here, asserting `CLR` overrides the other inputs including `G`. When `CLR` is negated and `G` is asserted, the outputs `Y[S-1:0]` decode the current input combination on `A[N-1:0]`, and the output values are held latched if `G` is negated.

Figure 10-22 shows the circuit that is synthesized by Xilinx Vivado tools when Program 10-5 is targeted to a 7-series FPGA with n=3 and s=8. The lefthand side of the logic diagram contains 8 LUTs to implement the 3-input AND functions that decode `A[2:0]`, and the righthand side has 8 D latches of the `LDCE` variety, controlled by `CLR` and `G` as you would expect.

The next devices that we'll cover are edge-triggered, and to model them in Verilog, we need the `posedge` and `negedge` keywords that we introduced briefly in Section 5.14. Recall that in the sensitivity list of an `always` block, one of these keywords is placed in front of a signal name to indicate that the block should be executed at the indicated edge of the named signal.

Thus, we can model a basic positive-edge-triggered D flip-flop very simply as shown in Program 10-6. The `always` block is executed at the positive edge of `CLK`, and `Q` gets set to `D`. Nothing happens at any other time, so `Q` is held at the same value at least until the next positive edge.

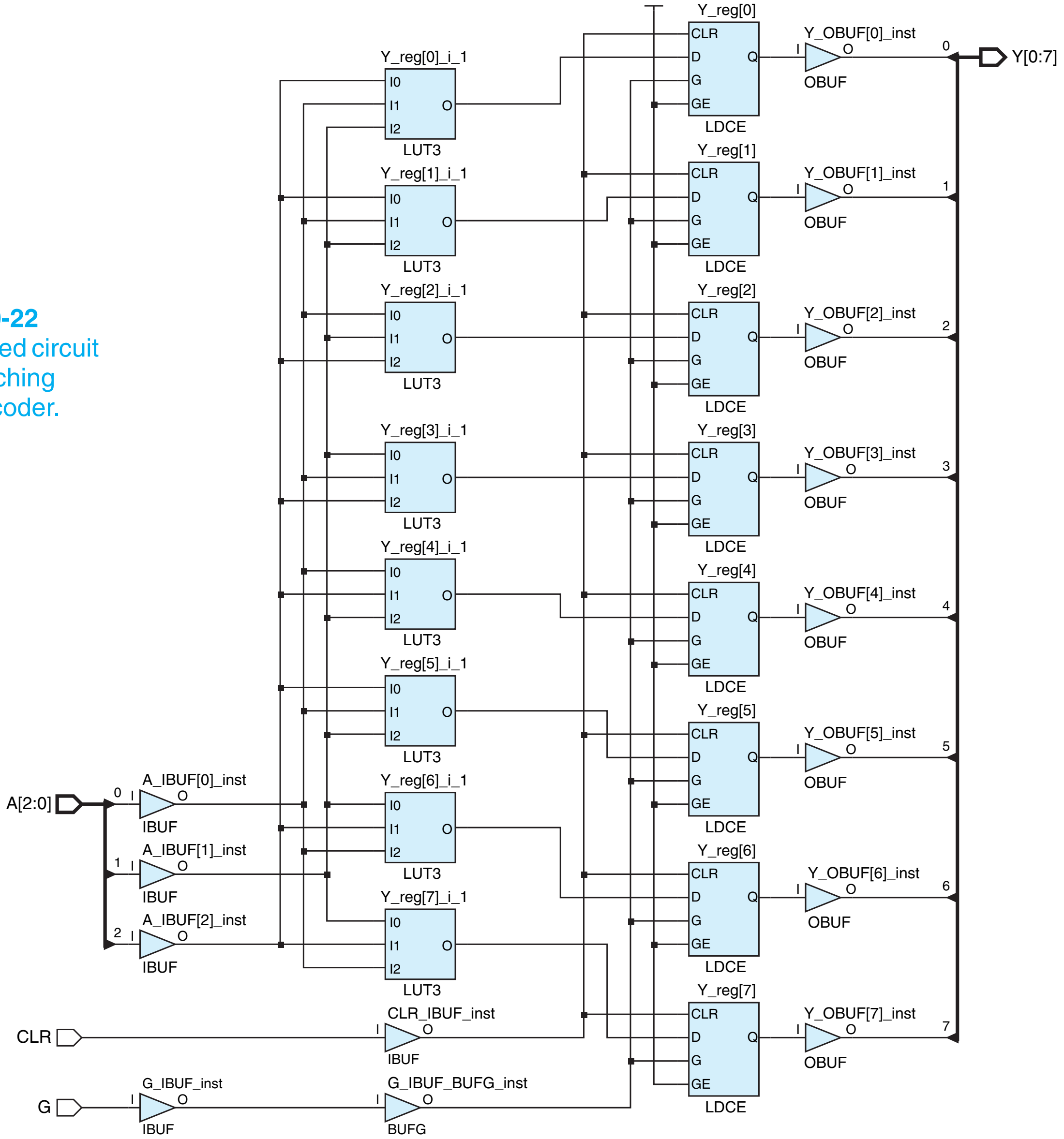


Figure 10-22 Synthesized circuit for the latching 3-to-8 decoder.

Program 10-6 Behavioral model for a basic D flip-flop.

```
module VrDff(CLK, D, Q);
  input CLK, D;
  output reg Q;

  always @ (posedge CLK)
    Q <= D;
endmodule
```

Program 10-7 Behavioral model for a D flip-flop with asynchronous clear.

```
module VrDffC(CLK, CLR, D, Q);
  input CLK, CLR, D;
  output reg Q;

  always @ (posedge CLK or posedge CLR)
    if (CLR==1) Q <= 0;
    else Q <= D;
endmodule
```

It takes a little code and some explaining to add an asynchronous clear, as shown in Program 10-7. The sensitivity list now includes the `CLR` input, which of course can cause the output to change. But why is the `posedge` keyword used with `CLR`, an asynchronous input? One answer is that this is a convenient way to code the needed behavior. Whenever `CLR` begins to be asserted (positive edge), the `always` block executes, and the `if` statement clears the `Q` output and exits. On the other hand, if the `always` block is executing and `CLR` is *not* asserted, then it must be executing because a positive edge occurred on `CLK`, and therefore `Q` is set equal to `D`. By the way, it would be a mistake to omit “`posedge`” and execute the `always` block on *any* change in `CLR`; on a 1-to-0 transition, the `else` clause would execute and set `Q` equal to `D` even though no `CLK` edge had occurred.

Another answer to the question “why write it this way?” for the D flip-flop with asynchronous clear in Program 10-7 is “because I say so.” (See the box on page 518 for details.)

In a typical FPGA- or CPLD-based design environment, a flip-flop need not have a `QN` output, since normally the next stage of combinational logic can invert any input signal at no cost. However, suppose we wanted to model a `QN` output anyway; Program 10-8 is a first attempt at providing one. Can you see the error? Remember that the non-blocking assignment operator `<=` makes its assignment an infinitesimal time *after* the `always` block completes. Therefore, the value assigned to `QN` by “`QN <= ~Q`“ is the old value of `~Q`; so in this model, `QN` equals `~Q` but delayed by one clock tick.

Program 10-8 Incorrect model for a D flip-flop with a QN output.

```
module VrDffCNoops(CLK, CLR, D, Q, QN);
  input CLK, CLR, D;
  output reg Q, QN;

  always @ (posedge CLK or posedge CLR) begin
    if (CLR==1) Q <= 0;
    else Q <= D;
    QN <= ~Q;
  end
endmodule
```

BAD BEHAVIOR

This is a good time to remind you that Verilog was originally designed as a *simulation* language, and that it was adapted for logic synthesis only later. While the language and its simulation semantics are well-defined by IEEE standards 1364 and 1800, there is no fully implemented standard on how a Verilog module should be synthesized into real hardware; there's only general industry agreement on the most widely used Verilog constructs. In synthesis, a Verilog tool examines modules for predefined, commonly used patterns of code—templates—and matches those to physical components like flip-flops. This is often called "inference," but it can be infernal!

It is very easy to write a Verilog model that can be simulated, but that cannot be synthesized into real hardware by any tool. Similarly, it's pretty easy to specify a behavior that works as expected in simulation, but that works inefficiently or even incorrectly in synthesized hardware, because the synthesis tool did not have a matching template, and inferred inefficient or incorrect hardware.

Consider, for example, the D flip-flop with asynchronous clear as specified behaviorally in Program 10-7. This is the syntax that Xilinx recommends for an FDCE flip-flop to be inferred in its FPGAs, and it works. But suppose you flipped the condition and the order of the if-else statement as follows:

```
if (CLR==0) Q <= D;
    else Q <= 0;
```

There's no difference in meaning, right? However, when Xilinx Vivado tools synthesize the modified module, the circuit in Figure 10-23 results. Not only is this much bigger, requiring an FDPE, an LDCE, and three LUTs, but worse, it doesn't even work correctly! What's going on? Some research led me to the following statement about sequential always blocks in the Xilinx UG901 Synthesis Guide:

> If optional asynchronous control signals are modeled, the always block is structured as follows:
>
> ```
> always @ (posedge CLK or posedge ACTRL)
> begin
>   if (ACTRL)
>     <asynchronous part>
>   else
>     <synchronous part>
>  end
> ```

They should say "*must be*," not "is" above. If the order of the asynchronous and synchronous action parts is reversed, as in the present example, all bets are off!

The bottom line is that when you write behavioral Verilog code for synthesis, you must write it in a style that matches the templates that are expected by the tools. And to learn what's expected, you'll have to read the documentation. For the most common elements, like the D flip-flop with asynchronous clear, there are "standard" (but not formally standardized) templates. Thus, a post-synthesis simulation is just about the only way to find out almost for sure whether your circuit will probably work as you might have expected (the caveats are deliberate, sadly).

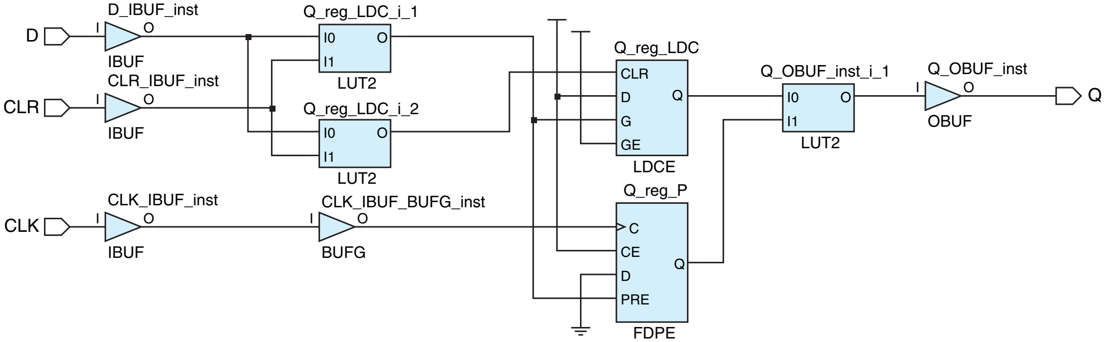


Figure 10-23 Incorrectly synthesized D flip-flop with asynchronous clear.

Program 10-9 Corrected behavioral model for a D flip-flop with a QN output.

```
module VrDffCN(CLK, CLR, D, Q, QN);
  input CLK, CLR, D;
  output reg Q, QN;

  always @ (posedge CLK or posedge CLR)
    if (CLR==1) begin Q <= 0; QN <= 1; end
    else begin Q <= D; QN <= ~D; end
endmodule
```

There are a couple of different ways to correct the error. In Program 10-9, `begin-end` blocks are used to set `QN` properly in each place that `QN` is set. This is still a purely behavioral model. Another approach is to combine a dataflow-style continuous assignment with the original behavioral code of Program 10-7, as shown in Program 10-10, to make `QN` always be the complement of `Q`. That works too; it's also more convenient and would be preferred by most designers.

Program 10-10 Another correct model for a D flip-flop with a QN output.

```
module VrDffCN2(CLK, CLR, D, Q, QN);
  input CLK, CLR, D;
  output reg Q;
  output QN;

  always @ (posedge CLK or posedge CLR)
    if (CLR==1) Q <= 0;
    else Q <= D;
  assign QN = ~Q;
endmodule
```

ALWAYS USE NONBLOCKING ASSIGNMENTS IN SEQUENTIAL always BLOCKS

In all of our flip-flop examples, we used the nonblocking assignment operator "<=" to assign a value to `Q`. These modules would have compiled and synthesized correctly even if we had used the blocking assignment operator "=", but there are subtle reasons why nonblocking assignments should *always* be used in sequential `always` blocks.

In models with multiple sequential `always` blocks using *blocking* assignments, the simulation results can vary depending on the order in which the simulator chooses to execute those blocks. Using *nonblocking* assignments ensures that the righthand sides of all assignments are evaluated before new values are assigned to any of the lefthand sides. This makes the results independent of the order in which the righthand sides are evaluated. More details are given in an excellent 1998 paper by Clifford Cummings, titled "State-Machine Coding Styles for Synthesis."

Old timers have a memory trick to remind them which assignment operator to use. The logic symbol for an edge-triggered flip-flop has a little wedge, the dynamic indicator, on the clock input. And the non-blocking assignment operator, used in clocked `always` blocks, has a similar wedge in "<=".

WHERE'S THE S-R LATCH?

In this section, we did not show any behavioral Verilog code to model an S-R latch. As simple as the basic latch can be—a pair of cross-coupled NAND or NOR gates—modeling it behaviorally in Verilog can be very tricky. If you really need an S-R latch, you're better off specifying it with structural or dataflow code that synthesizes directly into cross-coupled NAND or NOR gates in an ASIC (but be careful if you're targeting a programmable device; see Exercise 10.36.)

It's easy enough to model an S-R latch behaviorally if it has only a Q output. For example, consider the following module fragment:

```
always @ (S or R)
  if (S==1) Q <= 1;
   else if (R==1) Q <= 0;
```

set-dominant latch
reset-dominant latch

An S-R latch with this behavior, where set overrides reset, is called *set-dominant*. Similarly, you can specify a *reset-dominant* S-R latch with this code:

```
always @ (S or R)
  if (R==1) Q <= 0;
   else if (S==1) Q <= 1;
```

The problem occurs if you want to have a QN output too, and correctly model the situation where Q and QN are both 0 or both 1 when S and R are asserted simultaneously. I challenge you to come up with synthesizable behavioral code yielding a circuit that correctly produces this behavior as efficiently as a pair of cross-coupled NAND or NOR gates does, especially when targeting the code to a programmable device. (See Exercise 10.35.)

Program 10-11 Behavioral model for a D flip-flop with clock enable and synchronous set.

```
module VrDffSE(CLK, S, CE, D, Q);
  input CLK, S, CE, D;
  output reg Q;

  always @ (posedge CLK)
    if (S==1) Q <= 1;
    else if (CE==1) Q <= D;
endmodule
```

Our final example of a D flip-flop has a synchronous set input S and a clock-enable input CE, the same as the Xilinx FDSE library component. A Verilog model for that component is shown in Program 10-11. Notice that there is no final else clause in the if statement. If neither S nor CE is asserted, then the previous value of Q is retained. Like the D latch in Program 10-3, this code may seem to contradict our habit of using final else clauses in combinational logic to avoid creating inferred latches, but it's not needed or done in sequential logic like this.

Creating behavioral modules for other edge-triggered flip-flop types is straightforward and is left as a series of exercises for the reader (see Exercises 10.21–10.23, 10.37–10.38).

10.3.3 More about clocking in Verilog

In the test bench for a clocked circuit, one of things you need to do is to generate a system clock signal. This can be done easily with an always block, as shown in Program 10-12 for a 100-MHz clock with a 60% duty cycle. At time 0, MCLK is set to 1 by the initial block. Then, the always block waits 6 ns, sets MCLK to 0, waits 4 ns, sets MCLK to 1, and repeats forever. This gives a rising edge every 10 ns. Note that the `` `timescale `` directive has been used to set up the simulator with a default time unit of 1 ns and a precision of 100 ps.

Program 10-12 Clock generation within a test bench.

```
`timescale 1 ns / 100 ps
module Vrmclkgen(MCLK);
  output reg MCLK;

initial begin
  MCLK = 1;            // Start clock at 1 at time 0
end

always begin           // Free-running clock with 10 ns period
  #6 MCLK = 0;         // 6 ns HIGH
  #4 MCLK = 1;  end    // 4 ns LOW
endmodule
```

TINY OFFSETS The free-running clocks and other generated inputs in of the many test benches in this book are typically defined with tiny offsets, like 0.1 ns, so their edges don't fall exactly on a 5- or 10-ns boundary. That's just the author's nitpicking. The vertical reference lines in Xilinx Vivado timing diagrams are drawn *on top of* instead of *under* signal transitions. So, if a signal transition occurs exactly at a multiple of the reference interval, it is obscured by the reference line, even if it's a different color. "My way looks nicer," says the author, whose code often looks like this:

```
always begin         // 10 ns clock generation
  #5.9 MCLK = 0;     // 6 ns HIGH
  #4 MCLK = 1;       // 4 ns LOW
  #0.1 ;             // Includes 0.1 ns offset for readability
end
```

10.4 Multibit Registers and Latches

register

A collection of two or more D flip-flops with a common clock input is called a *register*. Registers are often used to store a collection of related bits, like a byte of data in a computer. In Section 9.6, we used Verilog to create registers to store state variables in state machines. A single register can also be used to store unrelated bits of data or control information; the only real constraint is that all of the bits are stored using the same clock signal.

10.4.1 MSI Registers and Latches

74x374

Many digital systems, including computers, telecommunications devices, and audio/visual equipment, process information 8, 16, 32, or 64 bits at a time; as a result, MSI ICs that handle these larger chunks of data are still used occasionally in input/output systems and the like. One such device is the *74x374* octal edge-triggered D flip-flop, also known simply as an 8-bit register. ("Octal" means that the device has eight sections.)

As shown in Figure 10-24(a), the 74x374 contains eight edge-triggered D flip-flops that all sample their inputs and change their outputs on the rising edge of a common CLK input. Each flip-flop output drives a three-state buffer that in turn drives an active-high output. All of the three-state buffers are enabled by a common, active-low output-enable input, OE_L. Like other three-state outputs, when OE_L is negated, the '374's outputs behave as if they were disconnected from the signals lines they would otherwise be driving.

74x373

One variation of the 74x374 is the *74x373,* whose symbol is shown in Figure 10-25. The '373 uses D latches instead of edge-triggered D flip-flops. Therefore, its outputs follow the corresponding inputs whenever G is asserted, and they latch the last input values when G is negated.

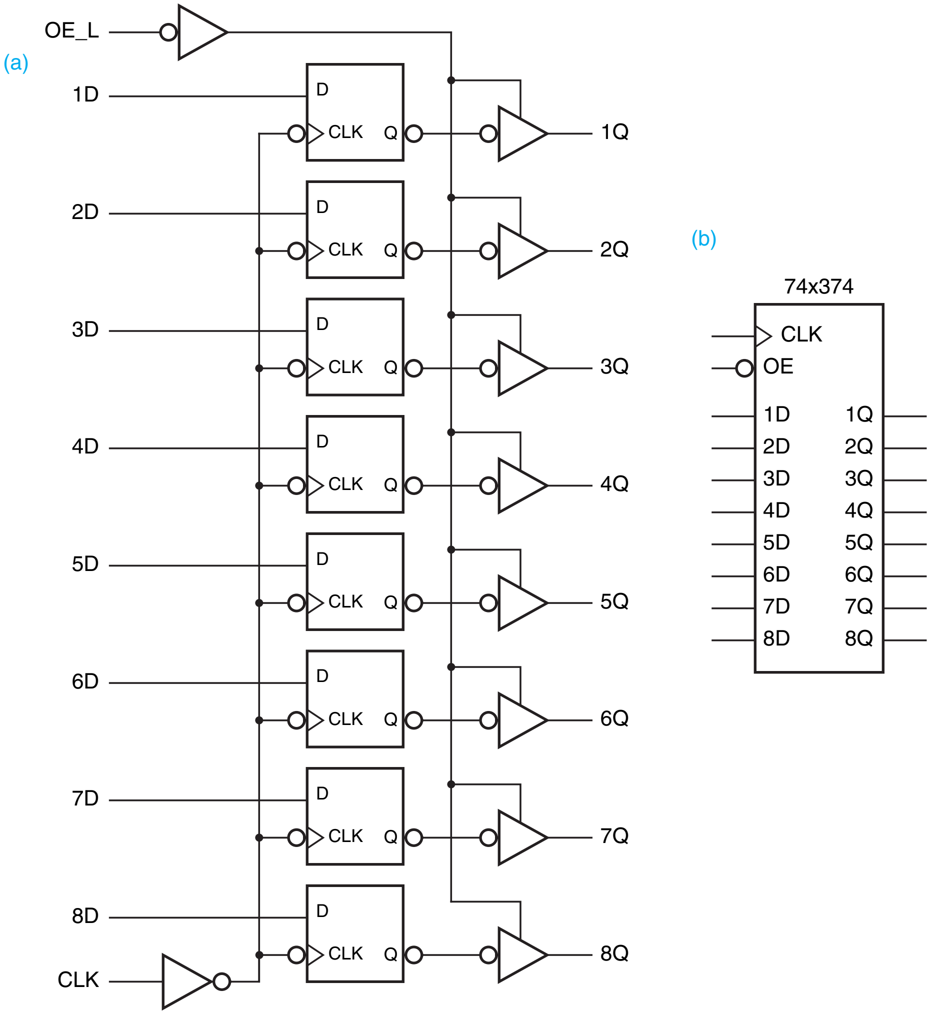


Figure 10-24
The 74x374 8-bit register:
(a) logic diagram;
(b) traditional logic symbol.

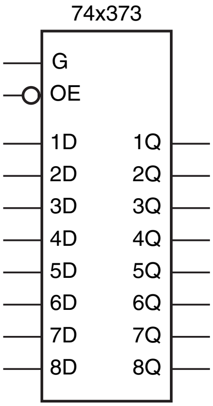


Figure 10-25
Logic symbol for the 74x373 8-bit latch.

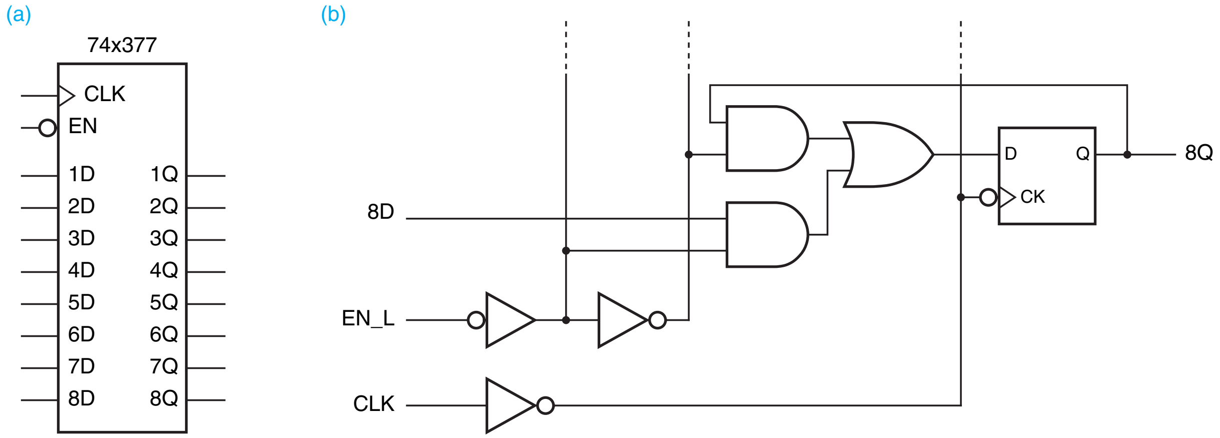


Figure 10-26 The 74x377 8-bit register with clock enable: (a) logic symbol; (b) logical behavior of one bit.

74x377

The *74x377*, whose symbol is shown in Figure 10-26(a), is an edge-triggered register like the ’374, but it does not have three-state outputs. It does however have an active-low clock-enable input EN_L. If EN_L is asserted (LOW) at the rising edge of the clock, then the flip-flops are loaded from the data inputs; otherwise, they retain their present values, as shown logically in (b).

High pin-count surface-mount packaging supports even wider registers, drivers, and transceivers. Most common are 16-bit devices, but there are also devices with 18 bits (for byte parity) and 32 bits. Also, the larger packages can offer more control functions, such as clear, clock enable, multiple output enables, and even a choice of latching vs. registered behavior all in one device. The data inputs of some devices in some CMOS logic families (such as LVC, ALVC, and AVC, described in Section 14.7) also feature bus-holder circuits, which we’ll describe in Section 10.5.2.

10.4.2 Multibit Registers and Latches in Verilog

Multibit registers and latches can be easily modeled using behavioral Verilog. We can use the same kind of code as we used in Section 10.3.2 for individual devices, except that we declare the signals as multibit vectors instead of single

Program 10-13 Verilog for a 74x377-like 8-bit register with clock enable.

```
module Vr74x377(CLK, EN_L, D, Q);
  input CLK, EN_L;
  input [1:8] D;
  output reg [1:8] Q;

  always @ (posedge CLK)
    if (EN_L==0) Q <= D;
endmodule
```

Program 10-14 Verilog module for a 16-bit register with many features.

```
module Vrreg16( CLK, CLKEN, OE, CLR, D, Q );
  input CLK, CLKEN, OE, CLR;
  input [1:16] D;
  output [1:16] Q;
  reg [1:16] IQ;

  always @ (posedge CLK or posedge CLR)
    if (CLR==1) IQ <= 16'b0;
    else if (CLKEN==1) IQ <= D;

  assign Q = (OE==1) ? IQ : 16'bz;
endmodule
```

bits. For example, Program 10-13 shows a Verilog module for a 74x377-like 8-bit register with clock enable.

In Verilog it's easy to define registers with more inputs and with additional features. For example, Table 10-14 is a module for a 16-bit register with three-state outputs and clock-enable, output-enable, and synchronous clear inputs. An internal signal vector IQ holds the flip-flop outputs, and the three-state outputs are defined and enabled as in Section 7.1.3.

As always, it's possible to parameterize the Verilog modules that define commonly needed components so they may be easily instantiated with different options—data widths in particular. For example, the 16-bit register in Table 10-14 can be parameterized by including "parameter WID = 16" along with its other declarations, and replacing all occurrences of "16" with "WID". When the component is instantiated, the register width will default to 16 bits, but any width can be obtained by specifying the desired value in the instance statement, for example:

```
Vrreg #(.WID(24)) U1 ( .CLK(myCLK), .CLKEN(myCLKEN), .OE(myOE),
                          .CLR(myCLR), .D(myD), .Q(myQ) );
```

*10.5 Miscellaneous Latch and Bistable Applications

Two simple but commonly used applications of S-R latches and bistables are described here.

*10.5.1 Switch Debouncing

A common application of simple bistables and latches is switch debouncing. We're all familiar with electrical switches from experience with lights, garbage disposals, and other appliances. Switches connected to sources of constant logic 0 and 1 are often used in digital systems to supply "user inputs." However, in

* Throughout this book, optional sections are marked with an asterisk.

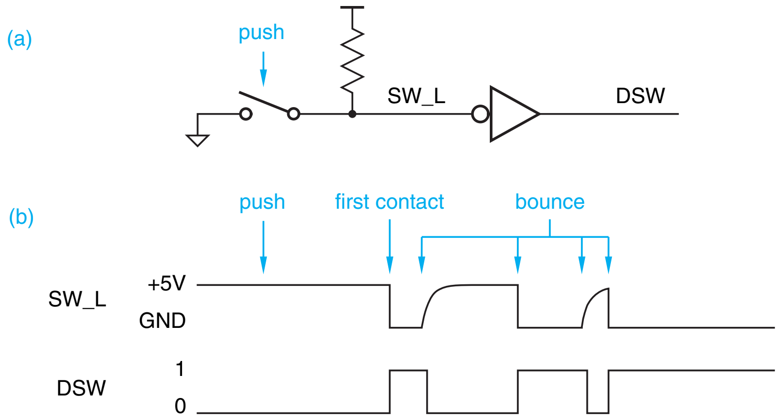


Figure 10-27 Switch input without debouncing: (a) logic circuit; (b) timing diagram

digital logic applications we must consider another aspect of switch operation, the time dimension. A simple make or break operation, which occurs instantly as far as we slow-moving humans are concerned, actually has several phases that are discernible by high-speed digital logic.

Figure 10-27(a) shows how a single-pole, single-throw (SPST) switch can be used to generate a single logic input. A pull-up resistor provides a logic-1 (HIGH) value when the switch is open, and the switch contact is tied to ground to provide a logic-0 (LOW) value when the switch is pushed.

As shown in (b), it takes a while after a push for the wiper to hit the bottom contact. Once it hits, it doesn't stay there for long; it bounces a few times before finally settling. The result is that several transitions are seen on the SW_L and DSW logic signals for each single switch push. This behavior is called *contact bounce*. Typical switches bounce for 10–20 ms, a very long time compared to the switching speeds of logic gates.

contact bounce

Contact bounce may or may not be a problem, depending on the switch application. For example, some computers and other devices have configuration information specified by small switches, called *DIP switches* because they have the same form factor as a dual in-line pin (DIP) package. Since DIP switches are normally changed only when the device is inactive, there's no problem. Contact bounce *is* a problem if a switch like a pushbutton is being used to count or signal some event (e.g., laps in a race). Then we must provide a circuit (or, in microprocessor-based systems, software) to *debounce* the switch—to provide just one signal change or pulse for each external event.

DIP switch

debounce

An $\overline{\text{S}}$-$\overline{\text{R}}$ latch and pull-up resistors can be used to debounce a single-pole, double-throw (SPDT) switch as shown in Figure 10-28. The switch contacts and wiper have a "break before make" behavior, so the wiper terminal is "floating" at some time halfway through the switch depression. Before the button is pushed, the top contact holds SWR_L at 0 V, a valid logic 0, which holds DSW_L HIGH and DSW LOW—the latch is reset. When the button is first pushed and the wiper terminal is floating, the latch is still in the "reset" state and holds DSW LOW.

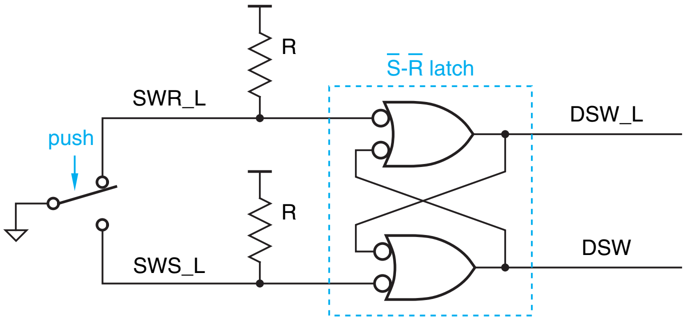


Figure 10-28 Switch input using an $\overline{\text{S}}$-$\overline{\text{R}}$ latch for debouncing.

Eventually, when the wiper hits the bottom contact, SWS_L is pulled to 0 V, setting the latch. Thus, DSW goes HIGH and it stays HIGH even when the wiper bounces and loses its connection with the bottom contact one or more times. (It does not bounce far enough to touch the top contact again.).

Depending on the application, system designers may prefer to use software to debounce a switch, simply by using time delays to ignore the bounces. Some drawbacks of the hardware $\overline{\text{S}}$-$\overline{\text{R}}$-latch solution are the SPDT switch's higher cost compared to SPST and its consumption of a second, possibly scarce input pin when the $\overline{\text{S}}$-$\overline{\text{R}}$-latch is in an FPGA or ASIC.

*10.5.2 Bus-Holder Circuits

In Section 7.1, we described three-state outputs and how they are tied together to create three-state buses. At any time, at most one output can drive the bus; sometimes, no output is driving the bus, and the bus is "floating." When high-speed CMOS inputs are connected to a bus that is left floating for a long time (in the fastest circuits, more than a clock tick or two), bad things can happen. For example, noise, crosstalk, and other analog effects can drive the high-impedance floating bus signals to a nonlogic voltage level near the CMOS devices' input switching threshold, which in turn causes excessive current flow in the devices' outputs. For this reason, it is desirable and customary to provide pull-up resistors that quickly pull a floating bus to a valid HIGH logic level.

Pull-up resistors aren't all goodness—they cost money and they occupy valuable printed-circuit-board real estate. A big problem that they have in very high-speed circuits is that there's no good resistance value. If it's too high, when a bus goes from LOW to floating, the transition from LOW to pulled-up (HIGH) will be slow due to the high *RC* time constant, and input levels may spend too much time near the switching threshold. But if the pull-up resistance is lowered, devices trying to pull the bus LOW will have to sink more current, even to the point of consuming far more power than the CMOS inputs on the bus.

The solution to this problem is to eliminate pull-up resistors in favor of the active *bus-holder circuit* shown in Figure 10-29. This is nothing but a bistable with a resistor *R* in one leg of the feedback loop. The bus holder's INOUT signal is connected to the three-state bus line which is to be held. When the three-state

bus-holder circuit

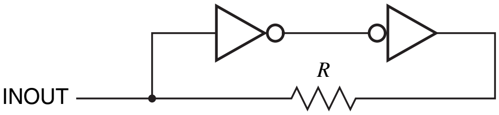


Figure 10-29
Bus-holder circuit.

output currently driving the line LOW or HIGH changes to floating, the bus holder's righthand inverter holds the line in its current state. When an enabled three-state output actively tries to change the line from LOW to HIGH or vice versa, it must source or sink a small amount of additional current through R to overcome the bus holder. This additional current flow persists only for the short time that it takes for the bistable to flip into its other stable state.

The choice of the value of R in the bus holder is a compromise between low override current (high R) and good noise immunity on the held bus line (low R). In a typical example, bus-holder circuits in the 3.3-V CMOS LVC family specify a maximum override current of 500 μA, implying $R \approx 3.3 / 0.0005 = 6.6$ KΩ.

Bus-holder circuits are often built into the inputs of another MSI device, like an octal CMOS bus driver or transceiver. They require no extra pins and very little chip area, so they are essentially free. And there's no real problem in having multiple (n) bus holders on the same signal line, as long as the bus drivers can provide n times the override current for a few nanoseconds during switching.

Note that bus holders normally are not effective on buses that have any TTL inputs attached to them, since TTL inputs require significant input current, especially in the LOW state. When the bus holder is trying to hold the bus LOW, this current creates a voltage drop across its resistor, raising the supposedly LOW bus voltage, possibly high enough to be a nonlogic voltage, which is bad.

*10.6 Sequential PLDs

generic array logic (GAL) device

The earliest, bipolar PLD families featured some devices with only combinational outputs, some with only registered outputs, and still others with a certain number of each output type. We described the combinational PAL16L8 in Section 6.2.2, and the registered types were described in previous editions of this book. All of these were supplanted by more versatile CMOS *generic array logic* (*GAL*) *devices* where the type of output, combinational or registered, can be selected when the device is programmed, as we describe here. These devices are still used when systems need a small amount of inexpensive, programmable "glue" logic.

GAL16V8

output logic macrocell (OLM)

The GAL16V8 (aka "16V8") PLD has eight outputs, and it was one of the first programmable logic devices to allow the user to select among two or more configurations of an *output logic macrocell* (*OLM*) for each output. The OLM's combinational configuration is shown in Figure 10-30(a). This looks like the original PAL16L8 output configuration (see Figure 6-11 on page 249)—seven ORed product terms and an eighth term to control the three-state output enable—with the useful addition of a configurable inversion on the signal path.

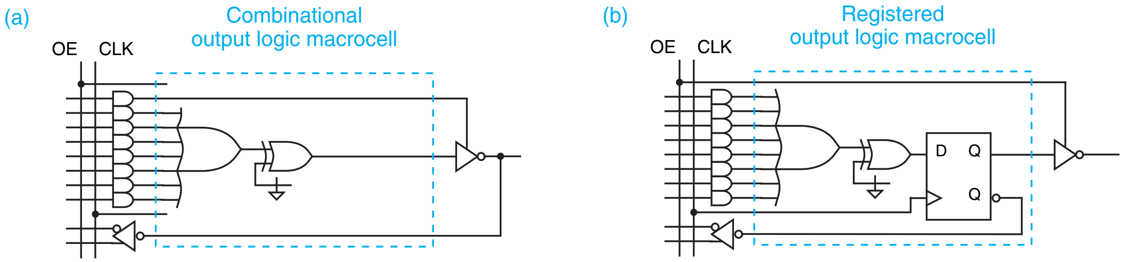

(a)
Combinational output logic macrocell
OE
CLK
(b)
Registered output logic macrocell
OE
CLK
D
Q
Q

Figure 10-30 Output logic macrocells for the 16V8R: (a) combinational; (b) registered.

The registered configuration of the OLM, shown in Figure 10-30(b), has all eight product terms connected to the OR gate and connects the logical sum, inverted or not, to the input of a D flip-flop. All of the D flip-flops in the device use the same common clock and drive an output pin through a three-state buffer controlled by a common output-enable signal. Figure 10-31 shows the structure of the 16V8 when all of the outputs are programmed to be registered, but any number of outputs can be configured this way.

GAL20V8

Two other PLDs are popular for applications that require slightly more capability than the 16V8. The *20V8* is similar to the 16V8 but has four extra input-only pins. Each product term in the 20V8 has 20 signals and their complements (12 input-only pins and 8 input/output pins), hence the "20" in "20V8."

GAL22V10

The *22V10*, has the same number of signal pins as the 20V8 (22), but has more internal architectural "goodies" than the 20V8, including the following:

- There are 10 outputs and OLMs instead of 8.
- Each output has its own product-term-controlled three-state enable.
- Each OR gate has up to 16 product terms, with a minimum of 8.
- There is a global synchronous preset signal that sets all internal flip-flops to 1 on the rising edge of the clock, controlled by a single product term.
- There is a global asynchronous reset signal that resets all internal flip-flops to 0 when asserted, controlled by a single product term.
- The common clock signal for the internal flip-flops is also available as a combinational input to any product term.

PLD manufacturers evolved their macrocell architectures significantly after introducing the devices above, learning from both successes and failures of designers who targeted practical circuits into each successive architecture generation. For example, to take advantage of increasing chip densities, they created more complex architectures to interconnect multiple PLDs within a single complex PLD (CPLD) chip. As densities increased, though, experience showed that FPGA architectures could evolve more effectively than PLD and CPLD architectures, so the newest, highest-density, and highest-performance programmable devices today are FPGAs.

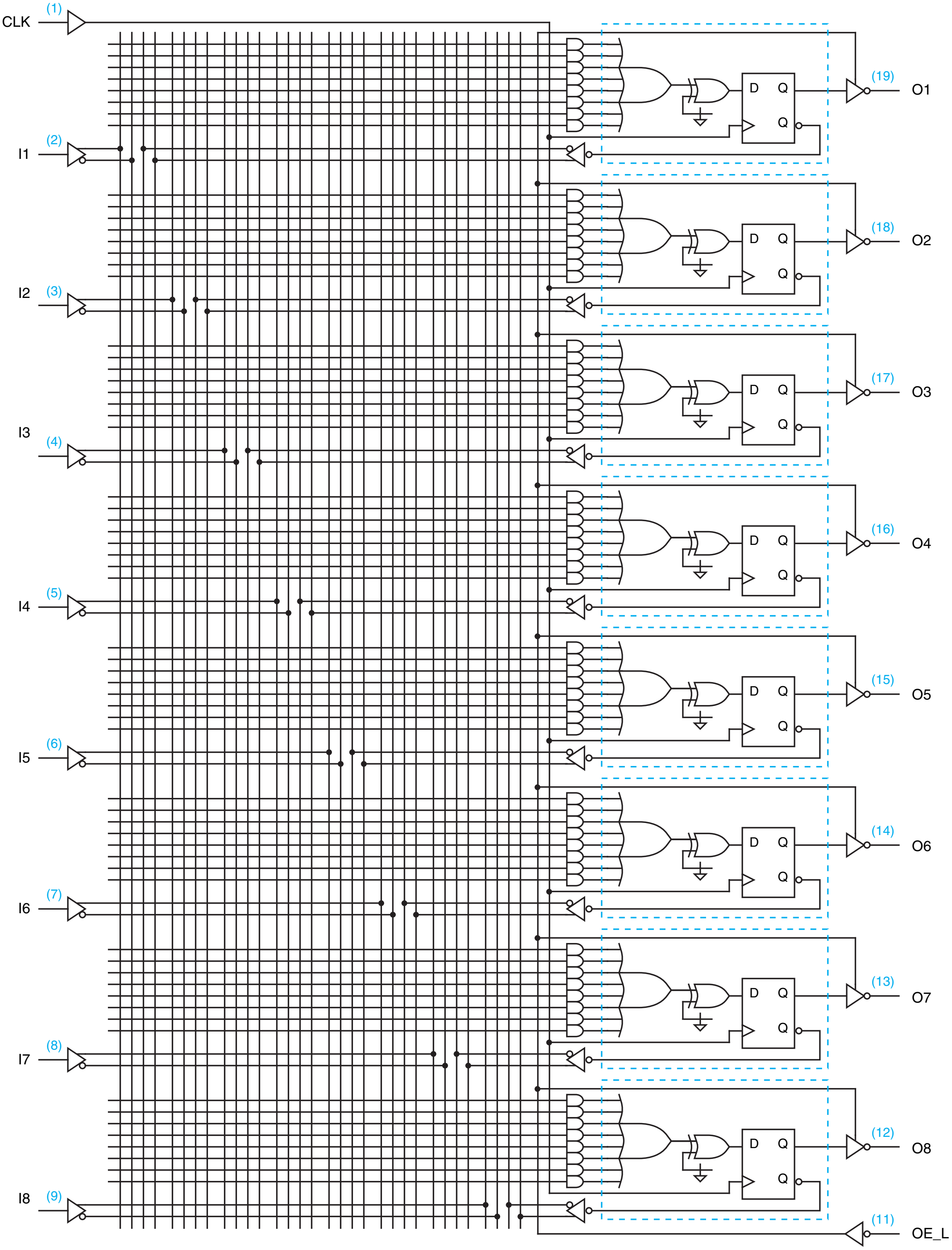


Figure 10-31 Logic diagram for the GAL16V8 in the “registered” configuration.

10.7 FPGA Sequential Logic Elements

Over the years, FPGAs have evolved tremendously, not just in size, but also in their capabilities and sophistication. We'll cover many aspects of one of the latest architectures, the Xilinx 7 Series, in Section 15.5, but for now we'll focus only on its sequential elements which are coupled to the combinational LUTs that you've already seen.

As we showed in Section 1.10, FPGA logic in general is broken into a large number of *configurable logic blocks (CLBs)* that are individually much smaller than a PLD. They are distributed across the entire chip in a sea of programmable interconnections, and the entire array is surrounded by programmable I/O blocks. A typical FPGA's configurable logic block is much less capable than a typical PLD. However, an FPGA chip contains a lot more of these blocks than a PLD, which has only one; and even more than a CPLD with the same die size. Modern FPGAs have at least hundreds of CLBs, and the largest have many tens of thousands. We'll discuss FPGA programmable interconnections and I/O in Section 15.5, but here we'll focus on the logic blocks.

configurable logic block (CLB)

Figure 10-32 shows some internals of a Xilinx 7-series FPGA chip. Each CLB contains a pair of *slices*, and each slice has four logic blocks that we'll call *programmable logic element (PLEs)*. Each PLE has one 6-input LUT and two flip-flops, as we'll show in detail. The PLEs have some control signals in common and are connected by an internal carry chain (the CARRY4 element that we described in Section 8.1.10). So, the 7-series CLB contains two slices or eight PLEs, for a total of eight LUTs, 16 flip-flops, and two 4-bit carry chains.

slice

programmable logic element (PLE)

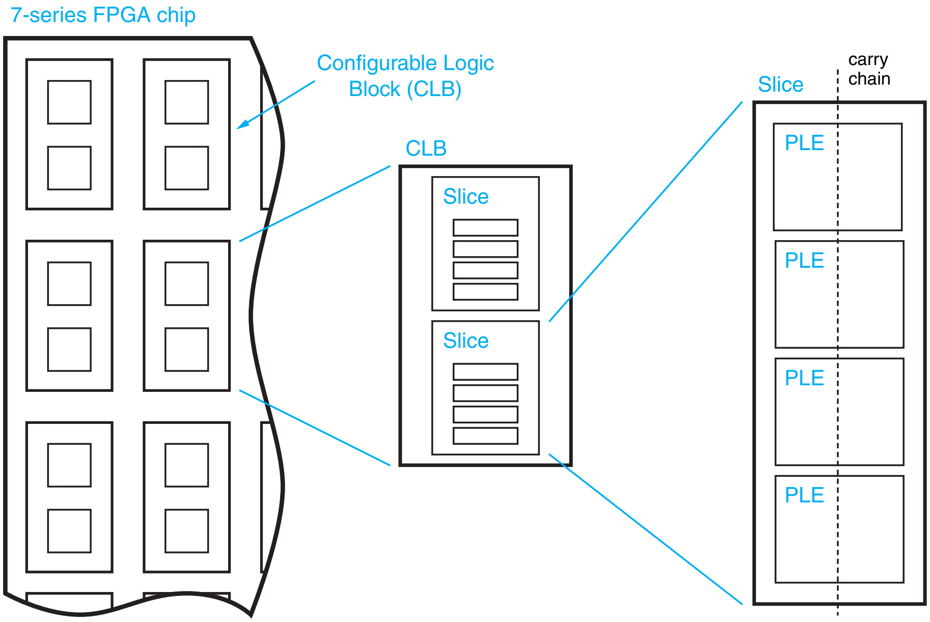


Figure 10-32 CLBs in a Xilinx 7-series FPGA.

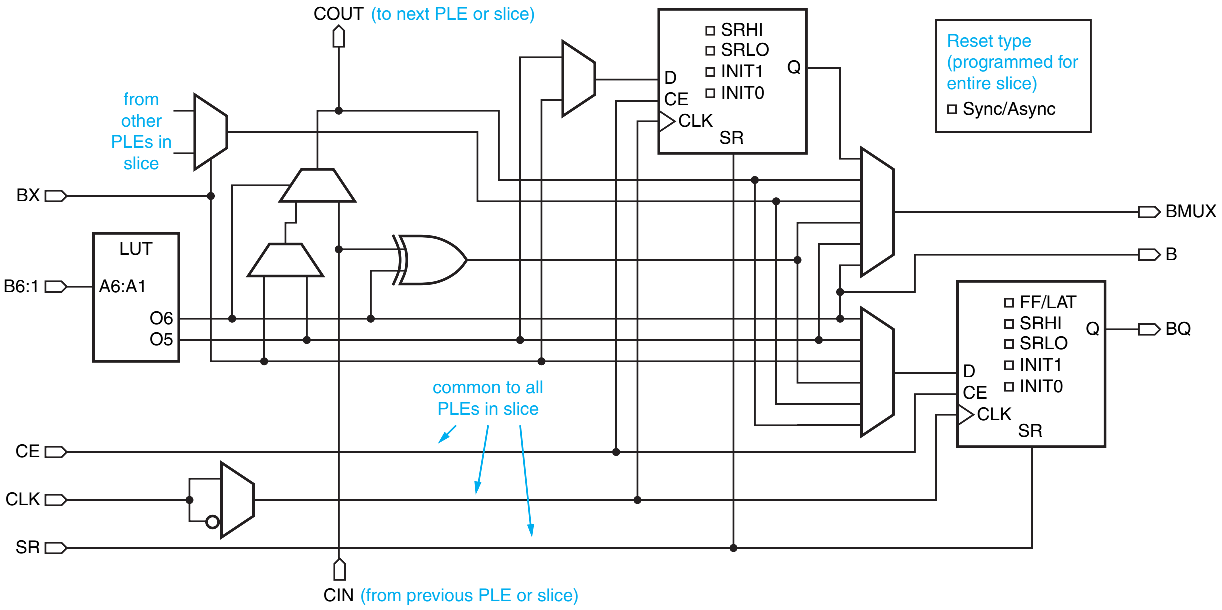


Figure 10-33 Structure of Xilinx 7-series programmable logic element (PLE).

The 7-series PLE has the structure shown in Figure 10-33, which we'll describe starting from the lefthand side. The 7-series LUT has six inputs and two outputs. Based on the programming of LUT's 64-bit "ROM," the O6 output can perform any logic function of the six inputs B[6:1]. Alternatively, as explained in Section 6.1.3, when B[6] is set to a constant 1, O6 and O5 independently perform any two logic functions of B[5:1].

Next are two 2-input multiplexers that control the configuration of the CARRY4 carry chain which runs vertically through the four PLEs in a slice. We showed the slice's carry chain in Figure 8-16 on page 400. The PLE's carry-chain input CIN comes from the PLE or slice below it. Its output COUT can come from, depending on the value of the O6 LUT output, either CIN or one of two other sources selected by programming—either the PLE's "extra input" BX or the LUT's O5 output. That's a lot of choices, and the synthesis tool has algorithms that choose an optimal configuration when it recognizes a known logic function to realize at the slice level, usually the carry chain for an adder.

THE NAMES HAVE BEEN CHANGED TO PROTECT THE INNOCENT

In this case, the "innocent" are the students who may have noticed that the signal names and numbers in the LUT in Figure 6-6 on page 244 are different from the ones used here in Figure 10-33. That's because the first figure uses traditional numbering for ROM address and data bits as introduced Section 6.1, and here we conform to the Xilinx naming, in case curiosity leads them to the Xilinx documentation.

The carry logic also has an XOR gate to combine CIN with the O6 LUT output, which is used to create a sum bit by XORing a half-adder sum bit from the LUT with CIN.

After those preliminaries, we can finally describe the PLE's two storage elements. The first, a D flip-flop, can programmably get its D input from BX or the LUT's O5 output. The second can be programmed to be either a D flip-flop or a D latch and can programmably get its D input from BX, the LUT's O5 or O6 output, the XOR output, or COUT.

Both of the PLE's storage elements (and in fact all storage elements in a slice) use the same clock signal CLK, which can be programmed at the slice level to be active high or active low. When the second storage element is programmed to be a latch, CLK serves as the G (enable) input. The slice also has an active-high clock-enable signal CE used by all storage elements. That may sound like a lot of choices, but wait, there's more!

The storage elements can be set or reset by a common SR (set-reset) signal for the slice. As suggested by the "check boxes" within each storage-element symbol, each element can be programmed to use SR as a set, a reset, or not at all. Moreover, the slice itself can be programmed to have all of its sets and resets be synchronous or asynchronous. Also, the initial state of each storage element can be programmed to be a 0 or a 1 at system initialization—that is, at power up or upon the assertion of the device's "global reset" signal.

Finally, we come to the PLE's outputs. There is a dedicated combinational output B which is the LUT's O6 output. A second combinational output BMUX is driven by a multiplexer which can programmably select the output of the left-hand flip-flop, COUT, the XOR output, O5, or O6. The third output on the righthand side of the figure is dedicated as the output of the second flip-flop, BQ. And don't forget the carry-chain output COUT at the top of the PLE, which we discussed above.

A logic designer would need to get a lot of experience with PLE and slice structure to be able to effectively map any given sequential logic design into the most effective configuration. Some things may be simple, like choosing a latch or a flip-flop and the type and polarity of the reset signal, depending on the requirements of the design. But others are not at all obvious, like which of the

MYSTERY MUX On top of all of the primary PLE features that we've described, the 7-series slice has a few more goodies that can be seen when considering a set of four PLEs as they are combined in a slice. For example, the mux in the upper-left corner of the PLE, when configured properly with similar muxes in other PLEs in the slice, allows the outputs of all four LUTs to be combined to implement an arbitrary 7- or 8-bit logic function. This is called an F7MUX or F8MUX, depending on its connections and which PLE it's in, and is discussed in the box on page 245.

two flip-flops and which registered outputs to use for which signals, and how to use the bypass input and the carry chain.

Fortunately, the PLE and slice were never intended to be used that way by a designer. Instead, the manufacturer's synthesis tools have "fitting" algorithms that can map commonly used HDL structures into very effective PLE and slice configurations. The algorithms can map most HDL code into corresponding configurations that are "good enough." Indeed, the manufacturer's architects of the PLE and slice had to work hand-in-hand with the tool designers to ensure that their architecture supported the right features and programmable choices to make effective algorithms possible, while still optimizing the size and performance of the PLE and slice structure—there can be hundreds of thousands of PLEs on the chip, so every little bit counts!

*10.8 Feedback Sequential Circuits

The simple bistable and the various latches and flip-flops that we studied earlier in this chapter are all feedback sequential circuits. Each contains one or more feedback loops that, ignoring their behavior during state transitions, store a 0 or a 1 at all times. The feedback loops are memory elements, and each circuit's behavior depends on both the current inputs and the values stored in the loops. In this section, we'll show how to analyze such circuits to give you some insight on how they work.

*10.8.1 Basic Analysis

fundamental-mode circuit

Feedback sequential circuits are the most common example of *fundamental-mode circuits*. In the normal operation of such circuits, inputs usually are not expected or allowed to change simultaneously. The analysis procedure assumes that inputs change one at a time, allowing enough time between successive changes for the circuit to settle into a stable internal state. This differs from

KEEP YOUR FEEDBACK TO YOURSELF

Only rarely does a logic designer encounter a situation where a feedback sequential circuit must be analyzed or designed. The most commonly used feedback sequential circuits are the flip-flops and latches that are used as the building blocks in larger sequential circuits. Their internal design and operating specifications are supplied by an IC manufacturer.

Even an ASIC designer typically does not design gate-level flip-flop or latch circuits, since these elements are supplied in a library of commonly used functions in any given ASIC technology. Still, you may be curious about how flip-flops and latches "do their thing"; this section shows you how to analyze such circuits.

*This section and all of its subsections are optional.

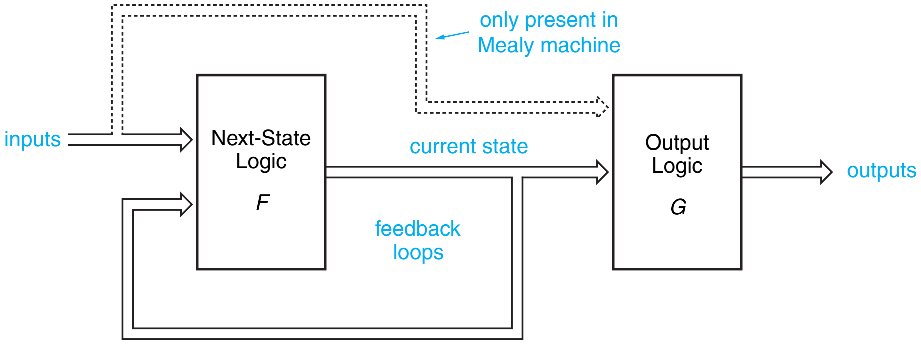


Figure 10-34 Feedback sequential circuit structure for Mealy and Moore machines.

clocked circuits, in which multiple inputs can change at almost arbitrary times without affecting the state, and all input values are sampled and state changes occur at the edge of a clock signal.

Like clocked synchronous state machines, feedback sequential circuits may be structured as Mealy or Moore machines, as shown in Figure 10-34. A circuit with *n* feedback loops has *n* binary state variables and 2^n states.

To analyze a feedback sequential circuit, we must break the feedback loops in Figure 10-34 so that the next value stored in each loop can be predicted as a function of the circuit inputs and the current value stored in all loops. Here, we'll look at the simplest possible situation—just one loop. Figure 10-35 shows how to break the feedback loop in the NAND circuit for a D latch. We conceptually break the loop by inserting a fictional buffer in the loop as shown. The output of the buffer, named Y, is the single state variable for this example.

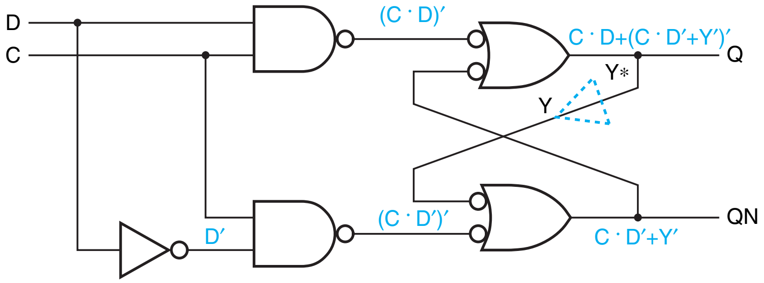


Figure 10-35 Feedback analysis of a D latch.

JUST ONE LOOP The way the circuit in Figure 10-35 is drawn, it may look like there are two feedback loops. However, once we make one break as shown, there are no more loops. That is, each signal can be written as a combinational function of the other signals, not including itself.

Figure 10-36 Transition table for the D latch in Figure 10-35.

	CD			
Y	00	01	11	10
0	0	0	1	0
1	1	1	1	0
	Y*			

Let us assume that the propagation delay of the fictional buffer is 10 ns (but any nonzero number will do) and that all of the other circuit components have zero delay. If we know the circuit's current state (Y) and inputs (D and C), then we can predict the value Y will have in 10 ns. The next value of Y, denoted Y*, is a combinational function of the current state and inputs. Thus, reading the logic diagram, we can write an *excitation equation* for Y*:

excitation equation

$$
\begin{aligned}
Y* &= (C \cdot D) + (C \cdot D' + Y')' \\
&= C \cdot D + C' \cdot Y + D \cdot Y
\end{aligned}
$$

Now the state of the feedback loop (and the circuit) can be written as a function of the current state and input, and enumerated by a *transition table* as shown in Figure 10-36. Each cell in the transition table shows the fictional-buffer output value that will occur 10 ns (or whatever delay you've assumed) after the corresponding state and input combination occurs.

transition table

A transition table has one row for each possible combination of the state variables, so a circuit with n feedback loops has 2^n rows in its transition table. The table has one column for each possible input combination, so a circuit with m inputs has 2^m columns in its transition table.

By definition, a fundamental-mode circuit such as a feedback sequential circuit does not have a clock to tell it when to sample its inputs. Instead, we can imagine that the circuit is evaluating its current state and input *continuously* (or every 10 ns, if you prefer). As the result of each evaluation, it goes to a next state predicted by the transition table. Most of the time, the next state is the same as the current state; this is the essence of fundamental-mode operation. We make some definitions next that will help us study this behavior in more detail.

total state
internal state
input state
stable total state
unstable total state
state table

In a fundamental-mode circuit, a *total state* is a particular combination of *internal state* (the values stored in the feedback loops) and *input state* (the current value of the circuit inputs). A *stable total state* is a combination of internal state and input state such that the next internal state predicted by the transition table is the same as the current internal state. If the next internal state is different, then the combination is an *unstable total state*. We have rewritten the transition table for the D latch in Figure 10-37 as a *state table*, giving the names S0 and S1 to the states and drawing a circle around the stable total states.

S	CD 00	CD 01	CD 11	CD 10
S0	S0	S0	S1	S0
S1	S1	S1	S1	S0
	S*			

Figure 10-37
State table for the D latch in Figure 10-35, showing stable total states.

To complete the analysis of the circuit, we must also determine how the outputs behave as functions of the internal state and inputs. There are two outputs and hence two *output equations*:

output equation

$$\begin{aligned} Q &= C \cdot D + C' \cdot Y + D \cdot Y \\ QN &= C \cdot D' + Y' \end{aligned}$$

Note that Q and QN are *outputs*, not state variables. Even though the circuit has two outputs, which can theoretically take on four combinations, it has only one state variable Y, and hence only two states.

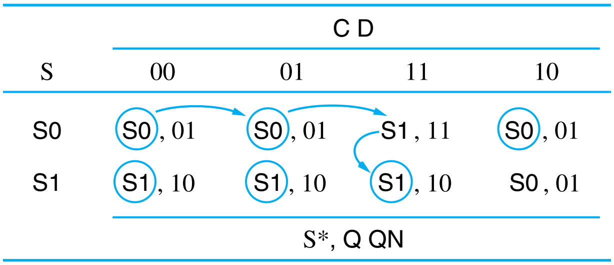
The output values predicted by the Q and QN equations can be incorporated in a combined state and output table that completely describes the operation of the circuit, as shown in Figure 10-38. Although Q and QN are normally complementary, it is possible for them to have the same value of 1 momentarily, during the transition from S0 to S1 under the C D = 11 column of the table.

We can now predict the behavior of the circuit from the transition and output table. First of all, notice that we have written the column labels in our state tables in "Karnaugh map" or "Gray code" order, so that only a single input bit changes between adjacent columns of the table. This layout helps our analysis because we assume that only one input changes at a time, and that the circuit always reaches a stable total state before another input changes.

At any time, the circuit is in a particular internal state and a particular input is applied to it; we called this combination the total state of the circuit. Let us start at the stable total state "S0/00" (S = S0, C D = 00), as shown in

S	CD 00	CD 01	CD 11	CD 10
S0	S0, 01	S0, 01	S1, 11	S0, 01
S1	S1, 10	S1, 10	S1, 10	S0, 01
	S*, Q QN			

Figure 10-38
State and output table for the D latch.



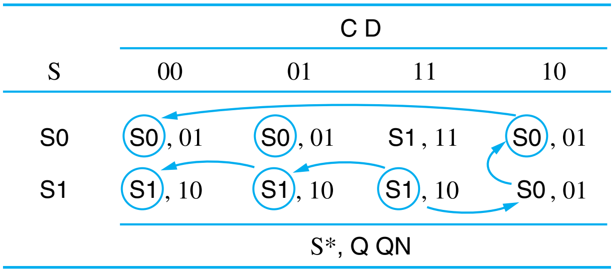
	C D			
S	00	01	11	10
S0	S0, 01	S0, 01	S1, 11	S0, 01
S1	S1, 10	S1, 10	S1, 10	S0, 01
	S*, Q QN			

Figure 10-39
Analysis of the D latch for a few transitions.

Figure 10-39. Now suppose that we change D to 1. The total state moves to one cell to the right; we have a new stable total state, S0/01. The D input is different, but the internal state and output are the same as before. Next, let us change C to 1. The total state moves one cell to the right to S0/11, which is unstable. The next-state entry in this cell sends the circuit to internal state S1, so the total state moves down one cell, to S1/11. Examining the next-state entry in the new cell, we find that we have reached a stable total state. We can trace the behavior of the circuit for any desired sequence of single input changes in this way.

Now we can revisit the question of simultaneous input changes. Even though "almost simultaneous" input changes may occur in practice, we must assume that nothing happens simultaneously in order to analyze the behavior of sequential circuits. The impossibility of simultaneous events is supported by the varying delays of circuit components themselves, which depend on voltage, temperature, and fabrication parameters. What this tells us is that a set of *n* inputs that appear to us to change "simultaneously" may actually change in any of *n*! different orders from the point of view of the circuit operation.

For example, consider the operation of the D latch as shown in Figure 10-40. Let us assume that it starts in stable total state S1/11. Now suppose that C and D are both "simultaneously" set to 0. In reality, the circuit behaves as if one or the other input went to 0 first. Suppose that C changes first. Then the sequence of two left-pointing arrows in the table tells us that the circuit goes to stable total state S1/00. However, if D changes first, then the other sequence of arrows tells us that the circuit goes to stable total state S0/00. So the final state of the circuit is unpredictable, a clue that the feedback loop may actually become metastable if we set C and D to 0 simultaneously. The time span



	C D			
S	00	01	11	10
S0	S0, 01	S0, 01	S1, 11	S0, 01
S1	S1, 10	S1, 10	S1, 10	S0, 01
	S*, Q QN			

Figure 10-40
Multiple input changes with the D latch.

over which this view of simultaneity is relevant is the setup- and hold-time window of the D latch.

Simultaneous input changes don't always cause unpredictable behavior and thus may be allowable in normal operation. However, we must analyze the effects of all possible orderings of signal changes to determine this; if all orderings give the same result, then the circuit output is predictable. For example, consider the behavior of the D latch starting in total state S0/00 with C and D simultaneously changing from 0 to 1; it always ends up in total state S1/11. This corresponds to the fact that in practice, a D latch has no setup hold requirement on the D input with respect to the 0-to-1 transition on C.

*10.8.2 Analyzing Circuits with Multiple Feedback Loops

In circuits with multiple feedback loops, we must break all of the loops, creating one fictional buffer and state variable for each loop that we break. There are many possible ways, which mathematicians call *cut sets*, to break the loops in a given circuit, so how do we know which one is best? The answer is that any *minimal cut set*—a cut set with a minimum number of cuts—is fine. Mathematicians can give you an algorithm for finding a minimal cut set, but as a digital designer working on small circuits, you can just eyeball the circuit to find one.

cut set

minimal cut set

Different cut sets for a circuit lead to different excitation equations, transition tables, and state/output tables. However, the stable total states derived from one minimal cut set correspond one-to-one to the stable total states derived from any other minimal cut set for the same circuit. That is, state/output tables derived from different minimal cut sets display the same input/output behavior, with only the names and coding of the states changed.

If you use more than the minimal number of cuts to analyze a feedback sequential circuit, the resulting state/output table will still describe the circuit correctly. However, it will use 2^m times as many states as necessary, where m is the number of extra cuts. Formal state-minimization procedures can be used to reduce this larger table to the proper size, but it's a much better idea to select a minimal cut set in the first place.

A good example of a sequential circuit with multiple feedback loops is an edge-triggered D flip-flop. CMOS flip-flops typically use transmission gates in their feedback loops. For example, Figure 10-41 shows the circuit design of the "FD1Q" positive-edge-triggered D flip-flop in LSI Logic's old LCA500K series of CMOS gate arrays. Such a flip-flop can be analyzed in the same way as a purely logic-gate-based design, once you recognize the feedback loops. Figure 10-41 has two feedback loops, each of which has a pair of transmission gates in a mux-like configuration controlled by CLK and CLK′, yielding the following loop equations:

$$\mathrm{Y1*} = \mathrm{CLK}' \cdot \mathrm{D}' + \mathrm{CLK} \cdot \mathrm{Y1}$$

$$\mathrm{Y2*} = \mathrm{CLK} \cdot \mathrm{Y1}' + \mathrm{CLK}' \cdot \mathrm{Y2}$$

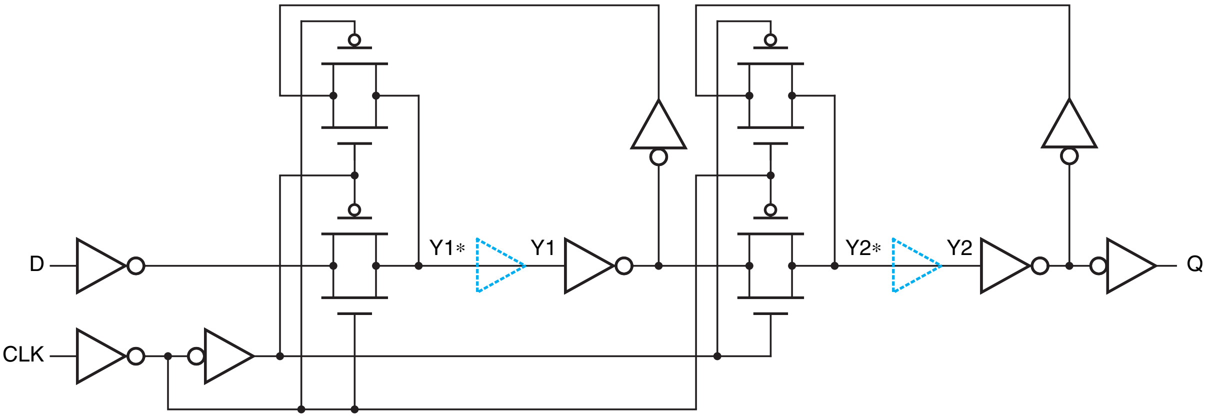


Figure 10-41 Positive edge-triggered CMOS D flip-flop for analysis.

Except for the double inversion of the data as it goes from D to Y2* (once in the Y1* equation and again in the Y2* equation), these equations are very reminiscent of the master/slave-latch structure of the D flip-flop in Figure 10-12 on page 505. The corresponding transition table is shown in Figure 10-42, with the stable total states circled.

race

critical race

A transition table with multiple state variables may have *races*, where two or more state variables change during the transition from one stable total state to the next. In a *critical race*, the final state depends on the order in which the variables change. Luckily for us, examination of all the possible transitions in the flow table in Figure 10-42 shows that it has no critical races; in fact, no races at all. Since we're analyzing a mature a commercial design, we should have expected that; otherwise, it's operation could be unreliable, depending on factors like voltage, temperature, and the phase of the moon.

At this point, we no longer need to refer to state variables. Instead, we can name the state-variable combinations and determine the output values for each state/input combination to obtain a state/output table, like Figure 10-43. Some

Figure 10-42 Transition table for the D flip-flop in Figure 10-41.

	CLK D			
Y1 Y2	00	01	11	10
00	10	00	01	01
01	11	01	01	01
10	10	00	10	10
11	11	01	10	10
	Y1* Y2*			

S	CLK D			
	00	01	11	10
S0	S2 , 0	S0 , 0	S1 , 0	S1 , 0
S1	S3 , 1	S1 , 1	S1 , 1	S1 , 1
S2	S2 , 0	S0 , 0	S2 , 0	S2 , 0
S3	S3 , 1	S1 , 1	S2 , 1	S2 , 1
	S∗ , Q			

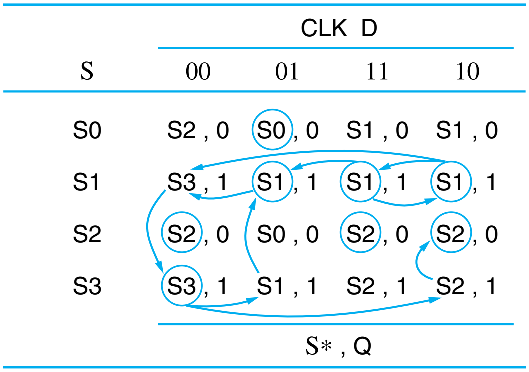
Figure 10-43 State/output table for the D flip-flop in Figure 10-41.

circuits (though not this one) may require multiple "hops" to get from one stable total state to the next one. The state table for such a circuit may be further simplified to create a *flow table* that eliminates multiple hops and shows only the ultimate destination for each transition.

flow table

The flip-flop's edge-triggered behavior can be observed in the series of state transitions shown in Figure 10-44. Let us assume that the flip-flop starts in internal state S1/10. That is, the flip-flop is storing a 0 (since Q = 0), CLK is 1, and D is 0. Now suppose that we change D to 1; the flow table shows that we move one cell to the left, still a stable total state with the same output value. We can change D between 0 and 1 as much as we want, and just bounce back and forth between these two cells. Similarly, if D is 1, we can change CLK between 0 and 1 as much as we want, and just bounce back and forth between two cells in this row. However, if both CLK and D change to 0, we move to internal state S3; but still the output Q is unchanged at 1. Now, if we change D back to 0, we go back up to the S1 row and can repeat the behavior there.

The moment of truth finally comes when CLK changes to 1 while we are in internal state S3. This moves us to internal state S3, where the output Q changes to 0, capturing the value that was present on D at the rising edge of CLK. Similar behavior involving S2 and S0 can be observed on a rising clock CLK edge that causes Q to change from 1 to 0.



S	CLK D			
	00	01	11	10
S0	S2 , 0	S0 , 0	S1 , 0	S1 , 0
S1	S3 , 1	S1 , 1	S1 , 1	S1 , 1
S2	S2 , 0	S0 , 0	S2 , 0	S2 , 0
S3	S3 , 1	S1 , 1	S2 , 1	S2 , 1
	S∗ , Q			

Figure 10-44 State and output table showing the D flip-flop's edge-triggered behavior.

*10.8.3 Feedback Sequential-Circuit Design

The feedback sequential circuits that we've analyzed in the previous subsections exhibit quite reasonable behavior, since, after all, they are latch and flip-flop circuits that have been used for years. However, if we throw together a "random" collection of gates and feedback loops, we won't necessarily get "reasonable" sequential circuit behavior. In a few rare cases, we may not get a sequential circuit at all (see Exercise 10.48), and in many cases, the circuit may be unstable for some or all input combinations (see Exercise 10.56).

Thus, the design of feedback sequential circuits continues to be something of a black art and is practiced only by a tiny fraction of digital designers. Some simple examples were shown in previous editions of this book. The basic design steps are as follows:

primitive flow table

1. Construct a primitive flow table from the circuit's word description. This table has only one stable total state per row, to keep things simple.

state minimization

2. Minimize the number of states in the flow table, using a formal minimization procedure.

state assignment

3. Find a race-free assignment of coded states to named states, adding auxiliary states or splitting states as required. Eliminating critical races can be tricky, and may also significantly increase the number of states required.

transition table

4. Construct the transition table.

excitation equations

5. Determine excitation equations corresponding to the transition table.
6. Find a realization of the excitation equations that is free of static hazards. If an excitation equation has a hazard, its output can have a "glitch" during an input transition, and that can cause the feedback loop to lose its current state even when the equation says it should be in the same state before and after the transition.

essential hazards

7. Check for essential hazards, which are the possibility of the circuit going to an incorrect next state as a result of a single input change. Such hazards are inherent in the circuit's flow table, regardless of its logic realization, and can be eliminated only by guaranteeing that delays on certain feedback paths within the circuit are greater than maximum delays on certain input-logic paths.

logic diagram

8. Draw the logic diagram.

For all but the simplest circuits, the seventh step is the toughest. It turns out that a fundamental-mode circuit must have at least three states to have an essential hazard, so latches don't have them. On the other hand, all flip-flops (circuits that sample inputs on a clock edge) do. And this is why digital designers always use predesigned flip-flops that have been modeled and tested over a range of operating conditions, rather than "rolling their own."

Program 10-15 Dataflow-style Verilog code for an S-R latch with cross-coupled NOR gates.

```
`timescale 1 ns / 100 ps
module VrSRlatchNOR_d ( S, R, Q, QN );
input S, R;
  output Q, QN;

  assign #1 QN = ~(S | Q);
  assign #1 Q  = ~(R | QN);
endmodule
```

*10.8.4 Feedback Sequential Circuits in Verilog

The `always` block and the simulator's event list are Verilog's fundamental mechanisms for handling feedback sequential circuits. Feedback sequential circuits may change state in response to input changes, and these state changes are manifested by changes propagating in a feedback loop until the feedback loop stabilizes. In simulation, this is manifested by the simulator putting signal changes on the event list and scheduling processes to rerun in "delta time" and propagate these signal changes until no more signal changes are scheduled.

Program 10-15 is dataflow-style Verilog code for an S-R latch, equivalent to a pair of cross-coupled NOR gates. In simulation, each of its two continuous-assignment statements implies a software process as discussed in Section 5.12. These processes interact to emulate the simple latching behavior of an S-R latch. The module includes a `timescale` directive and a 1-ns delay in each `assign` statement to produce a more interesting simulation result.

When we introduced the S-R latch in Section 10.2.1, we showed a timing diagram for a particular input sequence in Figure 10-5 on page 500. To test the Verilog module, we created a test bench with similar input timing. The resulting timing diagram produced by simulator is shown in Figure 10-45. The Verilog simulation is of course faithful enough to handle the cases where both S and R are asserted simultaneously.

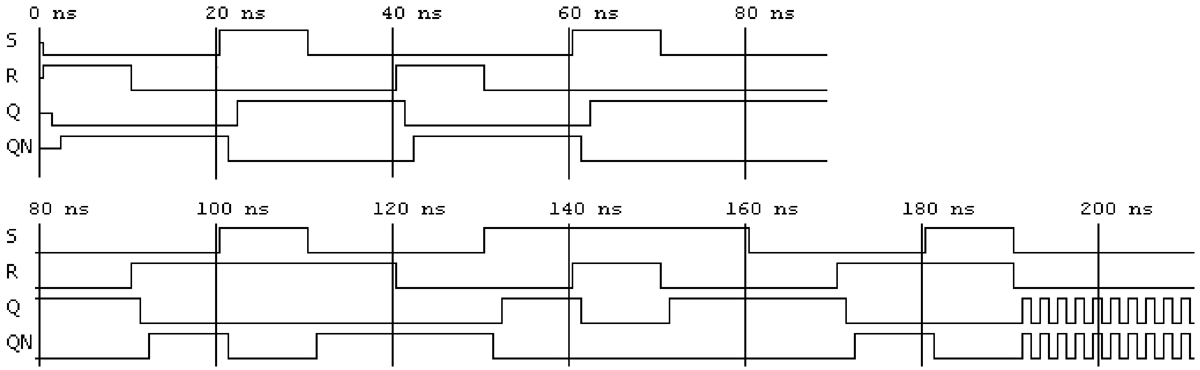


Figure 10-45 Simulation timing diagram for the S-R latch.

STRUCTURAL LATCH CODE

An S-R-latch module with the same functionality as Program 10-15 could be written in structural style by instantiating a pair of Verilog's built-in NOR gates, or in behavioral style with an `always` block. These models should be equivalent to the original in both simulation and synthesis, except possibly in simulation when S and R are negated simultaneously, depending on operation of the simulator when it encounters this unusual case.

The most interesting result in simulation occurs at the end, when S and R are negated simultaneously. Recall from the box on page 500 that a real S-R latch circuit may oscillate or go into a metastable state in this situation. If Program 10-15 were written without the 1-ns delays, the simulation could actually loop forever as each execution of one assignment statement triggers another execution of the other. After some number of repetitions, a well-designed simulator will discover the problem—for example by noticing that delta time keeps advancing while simulated time does not—and stop looping, which was the case with the Xilinx Vivado simulator.

However, with the 1-ns delays in place, we can actually see the oscillation that would occur if real circuit behavior was identical to what is simulated—zero signal rise and fall times, precise signal delays, and no noise or other parasitic electronic effects. In a real circuit, an oscillation may occur but with more sine-wave-like transitions, or both outputs may coast into a metastable state halfway between HIGH and LOW, until the outputs finally settle into one stable state or the other after a nondeterministic time.

References

The problem of metastability has been around for a long time. Greek philosophers wrote about the problem of indecision thousands of years ago. A later group of philosophers named Devo sang about metastability in the title song of their *Freedom of Choice* album. The U.S. Congress still can't figure out how to "save" Social Security. And I could have said that in every previous edition, too!

The latches and flip-flops described in this chapter are by far the most commonly used types today. Their Verilog models as shown in this chapter should work with any vendor's synthesis tool, which should "infer" the element that we've shown, if it exists in the target technology. But to be absolutely sure, your best reference is the documentation for the particular synthesis tool you're using and the technology that you're targeting. Consider, for example, the surprising, incorrect synthesized behavior described in the box on page 518.

Also, if a particular latch or flip-flop does not exist in the target technology, instead of rejecting it a synthesis tool may quietly create a decidedly ugly and inefficient emulation of it, as in Exercise 10.50. You may be better off restructuring your logic to use a more universally available flip-flop or latch. The only way to avoid these situations is to read and understand the vendor's documentation.

A more complete high-level discussion of the analysis and design of feedback sequential circuits can be found in earlier editions of the book you're reading. That may be enough to satisfy your curiosity. But if you're planning to actually design any such circuits from scratch, you should consult one or more of the really authoritative and comprehensive classic works on the subject, such as Edward J. McCluskey's *Logic Design Principles* (Prentice Hall, 1986) and Zvi Kohavi and Niraj K. Jha's *Switching and Finite Automata Theory* (Cambridge University Press, 2010, third edition).

Drill Problems

10.1 Give two examples of metastability that occur in sports, other than ones already mentioned in this chapter.

10.2 (1960s) In what song do the Lovin' Spoonful sing about metastability?

10.3 (1980s) Find the lyrics for the title song in Devo's *Freedom of Choice* album and write out the lines that refer to metastability.

10.4 (21st century) Identify a song that was popular in the current century and whose lyrics concern metastability, and write out those lines.

10.5 Would you expect the propagation delay from the set input to the Q output to be faster in a set-reset latch built from a pair of NAND gates or one built from a pair of NOR gates? Explain.

10.6 True or false: While set and reset in an S-R latch are asserted simultaneously, the Q output goes to a non-logic voltage halfway between logic 0 and 1. If true, explain what causes this, and if false describe another situation, if any, that can cause this condition.

10.7 What do the lyrics of a 1960s hit by the Rolling Stones have in common with Figure 10-6(c)?

10.8 Sketch the outputs of an S-R latch of the type shown in Figure 10-4 for the input waveforms shown in Figure X10.8. Assume that input and output rise and fall times are zero, that the propagation delay of a NOR gate is 10 ns, and that each time division below is 10 ns.

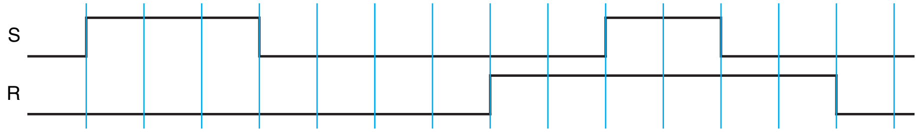


Figure X10.8

10.9 Repeat Drill 10.8 using the input waveforms shown in Figure X10.9. Although you may find the result unbelievable, this behavior can actually occur in real devices whose transition times are short compared to their propagation delay.

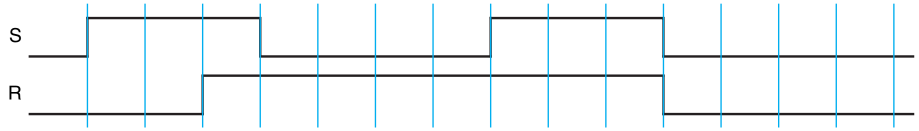


Figure X10.9

10.10 An S-R latch with a Q output is built from a pair of cross-coupled NOR gates. Is the latch set dominant or reset dominant?

10.11 An $\overline{\text{S}}$-$\overline{\text{R}}$ latch with a Q output is built from a pair of cross-coupled NAND gates. Is the latch set dominant or reset dominant?

10.12 Write a structural Verilog module `VrSRlatchNOR_s` corresponding to the S-R latch in Figure 10-4. Use Verilog's built-in `nor` component, and specify a simulated delay of 1 ns through each gate using a `` `timescale `` compiler directive and using the delay specifier "`#1`" after each `nor` keyword.

10.13 Create a Verilog test bench for the S-R latch of Drill 10.12. In the test bench, create input waveforms on S and R with timing as shown in Figure X10.8 followed by Figure X10.9. Run the test bench and print or draw the simulator's input and output waveforms (S, R, Q, and QN). What does the simulator do on the last input transition?

10.14 Write a structural Verilog module `VrSRlatchNAND_s` corresponding to the S-R latch in Figure 10-8. Use Verilog's built-in `nand` component, and specify a simulated delay of 1 ns through each gate. Then use the test bench of Drill 10.13 on this module as specified there, modifying as needed for the active-low inputs.

10.15 In what situations, if any, do the latches in Drills 10.12 and 10.14 produce different outputs for the same input sequence? To help find the answer, you may write a test bench similar to the one in Drill 10.13 that instantiates both modules.

10.16 A *positive-edge-triggered S-R flip-flop* has two control inputs S and R with the same meanings as in an S-R latch, except that the control inputs are sampled and the output changes state only at the rising edge of a CLK input. Show how to build a set-dominant S-R flip-flop using a D flip-flop and combinational logic.

10.17 A *positive-edge-triggered J-K flip-flop* has two control inputs J and K that control the device's behavior at the rising edge of CLK. If only J is asserted, the Q output is set to 1; if only K is asserted, Q is cleared to 0; if both are asserted, Q is toggled; and if neither is asserted, Q is not changed. Show how to build a J-K flip-flop using a D flip-flop and combinational logic.

10.18 Show how to build a T flip-flop with enable using a J-K flip-flop.

10.19 Figure 10-19(b) showed how to build a T flip-flop with enable using a D flip-flop and combinational logic. Show how to build a D flip-flop using a T flip-flop with enable and combinational logic.

10.20 Show how to build an S-R latch using a single positive-edge-triggered D flip-flop of the kind shown in Figure 10-16 and *no* other components.

10.21 Write a behavioral Verilog module `VrDnegEC` for a negative-edge-triggered D flip-flop with enable and asynchronous active-low clear. Also write a test bench that instantiates your flip-flop and exercises its operation for a comprehensive input sequence.

10.22 Write a behavioral Verilog module `VrTposE` for a positive-edge-triggered T flip-flop with enable. Also write a test bench that instantiates your flip-flop and exercises its operation for a comprehensive input sequence.

10.23 Write a behavioral Verilog module VrJKposP for a positive-edge-triggered J-K flip-flop with asynchronous active-low preset. Also write a test bench that instantiates your flip-flop and exercises its operation for a comprehensive input sequence.

10.24 What is the maximum number of edge-triggered D flip-flops that can be utilized within a single Xilinx 7-series slice? What control inputs, if any, must be identical for all of these flip-flops? (You may have to consult Xilinx documentation.)

10.25 What is the maximum number of D latches that can be utilized within a single Xilinx 7-series slice? What control inputs, if any, must be identical for all of these latches? (You may have to consult Xilinx documentation.)

10.26 Write a parameterized, behavioral Verilog module Vrreg_WID for a multibit register with width WID bits (default 16), clock-enable CLKEN, three-state output-enable OE, and synchronous clear CLR.

10.27 Write a structural Verilog module Vr74x377_s whose behavior is identical to that of Program 10-13. Your module should instantiate appropriate flip-flops from the component library of your favorite programmable device. Write a test bench that instantiates both Vr74x377_s and Vr74x377, and compares their outputs for a comprehensive input sequence.

10.28 Write a behavioral Verilog module Vr74x373 whose behavior is identical to that of the MSI 74x373 component, including its three-state outputs. Write a test bench that exercises your module for a comprehensive input sequence.

10.29 Write a structural Verilog module Vr74x373_s whose behavior is identical to that of the Vr74x373 module in Drill 10.28. Your module should instantiate appropriate latches from the component library of your favorite programmable device. Write a test bench that instantiates both Vr74x373_s and Vr74x373, and compares their outputs for a comprehensive input sequence.

Exercises

10.30 Explain how metastability occurs in the D latch of Figure 10-9 when its setup and hold times are not met, analyzing the behavior of its internal feedback loop.

10.31 Describe a situation, other than the metastable state, in which the Q and QN outputs of the edge-triggered D flip-flop in Figure 10-16 may be noncomplementary for an arbitrarily long time.

10.32 Determine and discuss one other situation, besides the one described in the last paragraph of Section 10.2.3, where the output of a D latch may become metastable. To what timing specification of a D latch does this situation relate?

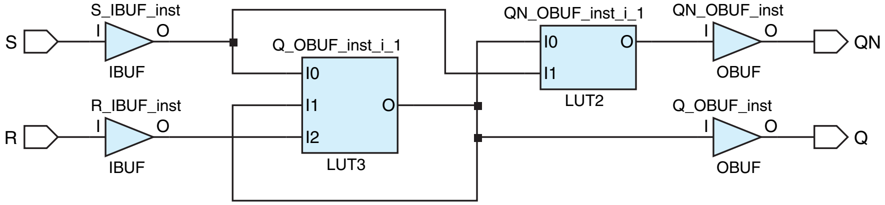
10.33 Write a parameterized Verilog test bench VrNtoSdec_latch_tb that checks the latching decoder in Program 10-5 for a comprehensive set of inputs. You may use the test bench in Program 6-8 as a starting point.

10.34 Write a behavioral Verilog module that has the same inputs and outputs as the S-R latch in Figure 10-4 and faithfully mimics its behavior in all respects except possibly metastability. Synthesize your module, targeting your favorite programmable device, and compare its resource requirements with the pair of cross-coupled NOR gates in the discrete gate-level implementation.

10.35 Write a behavioral Verilog module that has the same inputs and outputs as the $\overline{S}$-$\overline{R}$ latch in Figure 10-8 and faithfully mimics its behavior in all respects except possibly metastability. Synthesize your module, targeting your favorite programmable device, and compare its resource requirements with the pair of cross-coupled NAND gates in the discrete gate-level implementation.

10.36 Using Xilinx Vivado tools, the author wrote the solution for Drill 10.12 and then targeted it to a Xilinx 7-series FPGA, resulting in an implementation with the schematic shown in Figure X10.36. Here, the LUT2 output is QN = S′ · Q′, and the LUT3 output is Q = R′ · (S+Q). Explain whether this implementation is guaranteed to faithfully mimic the behavior of the pair of cross-coupled NOR gates that it's based upon.

Figure X10.36



JFW flip-flop

10.37 A famous logic designer decided to quit teaching and make a fortune by patenting and licensing a new kind of positive-edge-triggered device, the *JFW flip-flop*. Besides a clock input CLK, this device has a Q output and three inputs that control the device's behavior at the rising edge of CLK:

J Sets the Q output to 1 if no other control input is asserted.

F Flips the meaning of the J input; that is, clears the Q output to 0 if J is asserted and toggles the Q output if J is negated.

W Whatever—if J is negated, sets the Q output to whatever it was one cycle before, flipping that value if F is asserted. If J is asserted along with W, the device sets both Q and its memory of the current Q to whatever the next Q would be if W were negated.

Write a behavioral Verilog module `VrJFWff` for the device. Without adding a reset input, is there a way to deterministically initialize (clear or preset) the device from an unknown state in one clock tick? Do you think the device was successful in the marketplace, or was it a flop? Explain your reasoning.

10.38 Write a test bench that exercises the JFW flip-flop of Drill 10.37. Check for correct operation by applying a 16-tick input sequence 111–000 followed by 000–111 (in binary counting order) on the inputs J, F, W. The output sequence on Q should be 0011 0100 0011 1100.

10.39 Show how a bus-hold circuit of the kind shown in Section 10.5.2 can be used to create a debounced switch input.

10.40 Suppose you are asked to design a circuit that produces a debounced logic input from an SPST (single-pole, *single*-throw) switch. What inherent problem are you faced with?

10.41 Figure X10.41 shows another way to wire up an SPDT pushbutton switch to provide a logic input in a CMOS system. The top contact is connected to logic 1, and the bottom contact to logic 0. Like most switches, this one has "break before make" behavior, so the SW signal floats for several milliseconds as the button is pushed or released. Add analog and digital components to the switch circuit to obtain a debounced SW signal that stays at a valid logic level as the button is pushed or released.

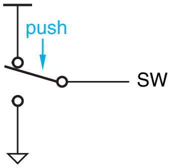


Figure X10.41

10.42 Find a way to debounce and clean up the SW signal in Exercise 10.41, consuming just one pin on a standard CMOS MSI device. Search the Web and identify one or more standard devices that can do this for you.

10.43 A clever digital designer, but one with no analog knowledge or experience, tried to solve the "floating SW" problem in Exercise 10.41 by simply replacing the switch with one that had "make before break" behavior. What happened the first time someone pushed the switch with the power on?

10.44 A particular Xilinx 7-series slice has been configured with three D latches. How many edge-triggered D flip-flops can be utilized in the same slice? (*Hint*: You'll have to search the Web for the correct answer; it's not in this text.)

10.45 This exercise is meant to challenge your understanding of latches and timing, even though state machines are never built this way anymore. Suppose that a clocked synchronous state machine with the structure of Figure 9-4 is designed using D latches with active-high G inputs, instead of edge-triggered D flip-flops, as storage elements. For proper next-state operation, what relationships must be satisfied among the following timing parameters?

t_{Fmin}, t_{Fmax}	Minimum and maximum propagation delay of the next-state logic.
t_{GQmin}, t_{GQmax}	Minimum and maximum enable-to-output delay of a D latch.
t_{DQmin}, t_{DQmax}	Minimum and maximum data-to-output delay of a D latch.
t_{setup}, t_{hold}	Setup and hold times of a D latch.
t_{H}, t_{L}	Clock HIGH and LOW times.

10.46 Analyze the feedback sequential circuit in Figure 10-16, assuming that the PR_L and CLR_L inputs are always 1. Derive excitation equations, construct a transition table, and analyze the transition table for critical and noncritical races. Name the states, and write a state/output table and, if different, a flow/output table. Show that the circuit performs the same function as Figure 10-42.

10.47 Show that a ones'-complement adder built as a binary adder with its carry output connected to its carry input ("end-around carry") is a feedback sequential circuit.

10.48 Draw the logic diagram for a circuit that has one feedback loop but is *not* a sequential circuit. That is, the circuit's output should be a function of its current input only. In order to prove your case, break the loop and analyze the circuit as if it were a feedback sequential circuit, and demonstrate that the outputs for each input combination do not depend on the "state."

10.49 Any practical single-loop feedback sequential circuit is just a variation of an S-R or D latch and has an excitation equation of the form

$$Q* = (\text{forcing term}) + (\text{holding term}) \cdot Q$$

Why aren't there any practical circuits whose excitation equation substitutes Q′ for Q above?

10.50 As shown in Table 10-1, the Xilinx ISE library has an edge-triggered D flip-flop component FDCP that has two asynchronous inputs, clear and preset, with functionality similar to Figure 10-16. However, such a flip-flop component does not exist in a Xilinx 7-series FPGA. If an FDCP is instantiated in a user's Verilog module, Vivado emulates it using two natively available edge-triggered flip-flops, a D latch, and a 3-input LUT, as shown in Figure X10.50. The LUT's function is $O = I0 \cdot I2' + I1 \cdot I2$. Explain in words how this works, and write a test bench that exercises the FDCP's operation for a comprehensive input sequence. Are the asynchronous inputs set-dominant or reset-dominant?

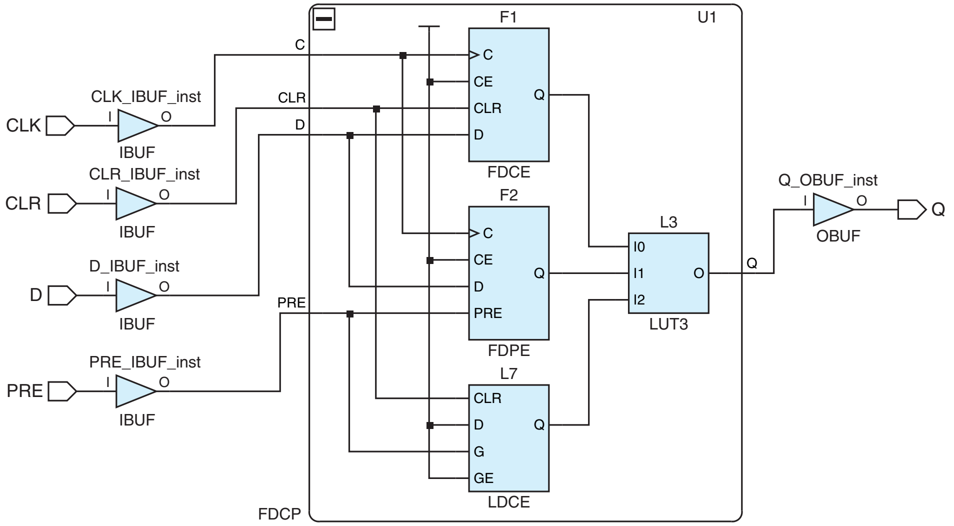


Figure X10.50

10.51 In general, the excitation logic in a feedback sequential circuit must be free of static and dynamic hazards, defined in Section 3.4. For example, consider a D latch whose excitation logic is a two-level AND-OR circuit having the form in Exercise 10.49 with a forcing term of C · D and a holding term of C′. Find the static-1 hazard in the excitation logic and explain how the latch may operate when a hazardous input transition occurs. Determine how to modify the excitation logic to eliminate the hazard.

10.52 Simulate the latch circuit that is initially described in Exercise 10.51 under the conditions described there. Use a Verilog structural model in which each gate has a delay of 1 ns, or draw the waveforms by hand, again assuming a delay of 1 ns per gate. How does the circuit behave at the hazardous input transition(s)? Next, increase the delay of just the inverter in the circuit to 3 ns, repeat the simulation, and explain the results. What would you expect to happen in the real circuit?

10.53 Compare the circuit in Figure X10.53 with the D latch in Figure 10-9. Prove that the circuits function identically. In what way is Figure X10.53, which is used in some commercial D latches, better?

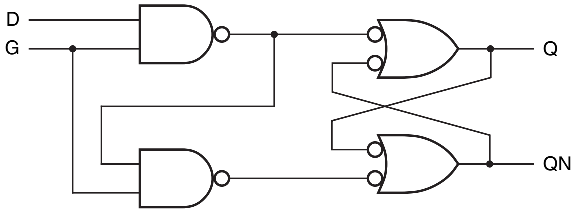


Figure X10.53

10.54 Suppose you are designing a circuit that requires an S-R flip-flop, and targeting it to an FPGA that has only LUTs, edge-triggered D flip-flops, and D latches. The FPGA has no native S-R flip-flops, and even simple gates like NAND and NOR are realized using LUTs. You've read ahead and studied the riddle on page 720, so you know that you cannot safely implement the S-R flip-flop as a pair of cross-coupled LUTs. And while the FPGA's edge-triggered D flip-flops and D latches have asynchronous CLR inputs, you don't want to use them because they are needed for other signals at reset. Show how to safely realize an S-R flip-flop in this environment using only the available elements. In what situations, if any, is its behavior different from that of the cross-coupled NAND- or NOR-gate design?

BUT flop

10.55 A *BUT flop* may be constructed from an NBUT gate as shown in Figure X10.55. (An *NBUT gate* is simply a BUT gate with inverted outputs; see Exercise 3.37 for the definition of a BUT gate.) Analyze the BUT flop as a feedback sequential circuit and obtain excitation equations, transition table, and flow table. Is this circuit good for anything, or is it a flop?

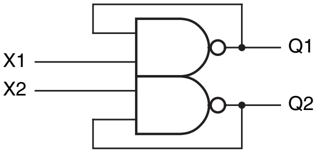


Figure X10.55

10.56 Repeat Exercise 10.55 for the asymmetric BUT flop in Figure X10.56.

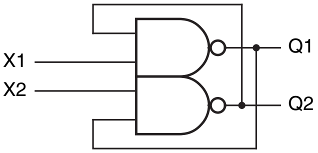
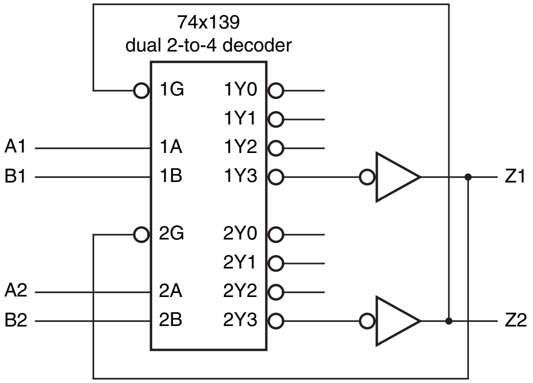


Figure X10.56

10.57 A “clever” student, Sam, designed the circuit in Figure X10.57 to create a BUT gate based on the definition in Exercise 3.37 using an available 2-to-4 decoder. The circuit appears to have feedback, but Sam analyzed the circuit for all 16 input combinations to make sure it was combinational. That is, Sam applied each input combination to A1, A2, B1, and B2, and assuming that Z1 and Z2 had the correct values, checked that the decoder and inverter outputs were consistent with that assumption. The circuit seemed to work correctly for all 16 possible input combinations.

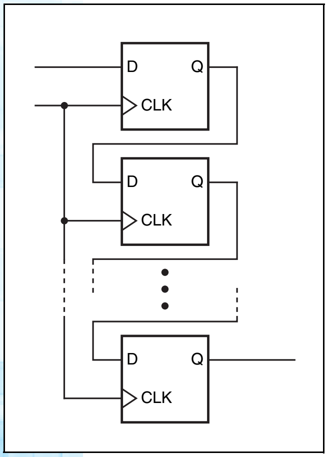
But in simulation, when the inputs were simultaneously changed from all 0s to all 1s, the simulator stopped after 5000 simulation cycles, complaining that the outputs hadn’t stabilized. And when the circuit was built and the same input transition was tried, the circuit’s outputs sometimes oscillated before settling down. Analyze the circuit as a feedback sequential circuit and explain why this happens.

Figure X10.57



10.58 Build a verbal flip-flop—a logical word puzzle that can be answered correctly in either of two ways depending on state. How might such a device be adapted to the political arena?

10.59 Read John Chu’s science-fiction short story “Hold-Time Violations,” and quote the passage that first introduces and explains them. How are they eliminated? Can a similar approach be used in digital logic?



chapter 11

Counters and Shift Registers

Any sequential circuit is technically a state machine, having storage elements, excitation logic, output logic, and a well-defined next-state behavior. However, there are some commonly used sequential circuits whose behaviors are so familiar and easy to describe that they have their own names—counters and shift registers.

In the days of MSI-based design, many different predesigned single-chip counter and shift registers circuits were commercially available, each in its own IC package. Because of these parts' ubiquity, designers developed several different ways to perform more elaborate sequential functions using the MSI components as a starting point and adding a handful of gates to obtain a more specialized function, such as a customized counting sequence, a timing generator, or even a random-number generator.

Naturally, counters and shift registers today are embedded in larger ICs like ASICs, PLDs, and FPGAs, either as library components or as HDL modules. Besides being used as needed in larger designs to perform their basic functions, they are also used as the starting point for specialized sequential functions like the ones mentioned above.

So, in this chapter, we'll describe the basic design of counters and shift registers at the gate and flip-flop level and in Verilog; and we'll also show how they can perform various specialized functions.

11.1 Counters

counter
modulus

The name *counter* is generally used for any clocked sequential circuit whose state diagram contains a single cycle, as in Figure 11-1. The *modulus* of a counter is the number of states in the cycle. A counter with m states is called a *modulo-m counter* or, sometimes, a *divide-by-m counter.* A counter with a non-power-of-2 modulus has extra states that are not used in normal operation.

modulo-m counter
divide-by-m counter
n-bit binary counter

Probably the most commonly used counter type is an *n-bit binary counter.* Such a counter has n flip-flops and has 2^n states, which are visited in the sequence 0, 1, 2, … , $2^n - 1$, 0, 1, … . Each of these states is encoded as the corresponding n-bit binary integer.

11.1.1 Ripple Counters

An n-bit binary counter can be constructed with just n flip-flops and no other components, for any value of n. Figure 11-2 shows such a counter for $n = 4$. Recall that a T flip-flop changes state (toggles) on every rising edge of its clock input. Thus, each bit of the counter toggles if and only if the immediately preceding bit changes from 1 to 0. This corresponds to a normal binary counting sequence—when a particular bit changes from 1 to 0, it generates a carry to the next most significant bit. The counter is called a *ripple counter* because the carry information ripples from the less significant bits to the more significant bits, one bit at a time.

ripple counter

Although a ripple counter requires fewer components than any other type of binary counter, it does so at a price—it is slower than any other type of binary counter. In the worst case, when the most significant bit must change, the output is not valid until time $n \cdot t_{TQ}$ after the rising edge of CLK, where t_{TQ} is the propagation delay from input to output of a T flip-flop. Also, ripple counters don’t fit well or at all into FPGA- and PLD-based designs, where all flip-flops or groups of flip-flops share a common clock signal.

Thus, ripple counters are rarely used in practice, but they can be useful in very low-power applications, such as digital watches. A flip-flop consumes additional, “dynamic” power every time that it is clocked, even if its state isn’t

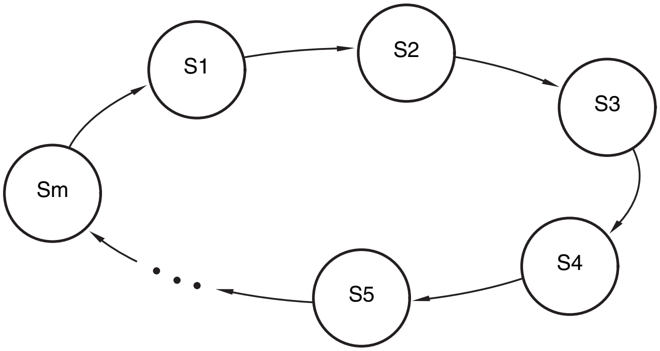


Figure 11-1
General structure of a counter’s state diagram, a single cycle.

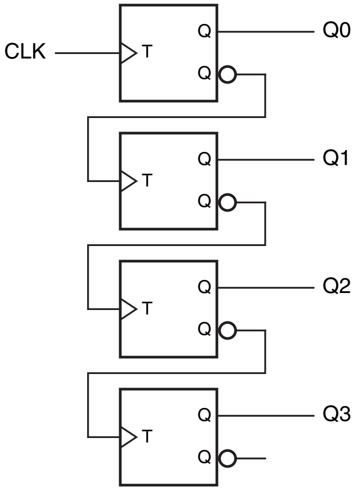


Figure 11-2
A 4-bit binary ripple counter.

changing. With a ripple counter, only the low-order bit is clocked at the full clock frequency, and each higher-order bit operates at half of the frequency and consumes only half of the dynamic power of the one before it.

11.1.2 Synchronous Counters

synchronous counter

A *synchronous counter* connects all of its flip-flop clock inputs to the same common CLK signal, so that all of the flip-flop outputs change at the same time, after only t_{TQ} ns of delay. As shown in Figure 11-3, this can be done using T flip-flops with enable inputs; the output toggles on the rising edge of T if and only if EN is asserted. Combinational logic on the EN inputs determines which, if any, flip-flops toggle on each rising edge of T.

As shown in Figure 11-3, it is also possible to provide a master count-enable signal CNTEN. Each T flip-flop toggles if and only if CNTEN is asserted

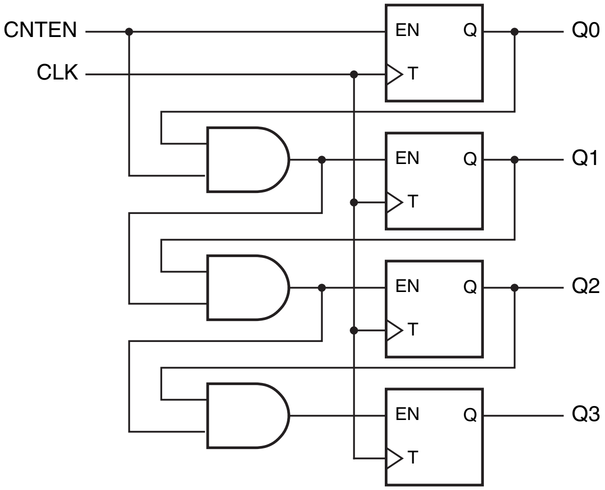


Figure 11-3
A synchronous 4-bit binary counter with serial enable logic.

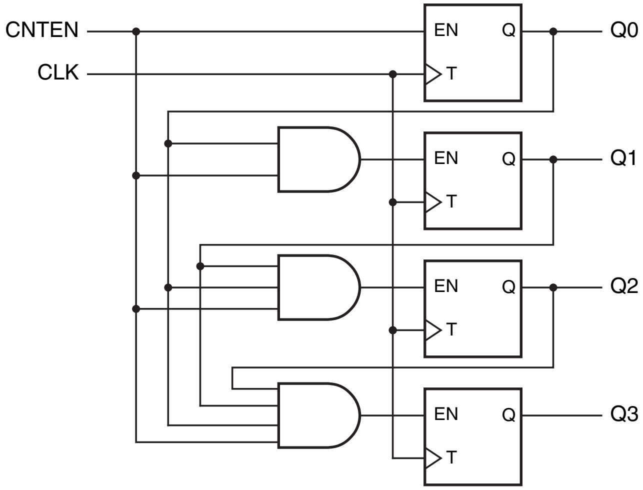


Figure 11-4
A synchronous 4-bit binary counter with parallel enable logic.

and all of the lower-order counter bits are 1. Like the binary ripple counter, a synchronous *n*-bit binary counter can be built with a fixed amount of logic per bit—in this case, a T flip-flop with enable and a 2-input AND gate.

synchronous serial counter

The counter structure in Figure 11-3 is sometimes called a *synchronous serial counter* because the combinational enable signals propagate serially from the least significant to the most significant bits. If the clock period is too short, there may not be enough time for a change in the counter's LSB to propagate to the MSB. Moreover, in an FPGA- or PLD-based implementation, the serial enable chain is not at all efficient to implement. These problems are eliminated in Figure 11-4 by driving each EN input with a dedicated AND gate, just a single level of logic. Called a *synchronous parallel counter,* this is the fastest binary counter structure.

synchronous parallel counter

11.1.3 A Universal 4-Bit Counter Circuit

This subsection shows the gate-level design of a synchronous 4-bit binary counter with synchronous load and clear inputs, based on the most popular MSI counter of its day, the 74x163. We'll call it simply a CNTR4U. Most designers today would use an HDL model to create such a counter, but it's worthwhile to study its internals because it has such a classic and efficient design.

The CNTR4U's internal logic diagram is shown in Figure 11-5, and its function is summarized by the state table in Table 11-1 (with "middle" current states 0010–1100 omitted).

The CNTR4U uses D flip-flops rather than T flip-flops, which facilitates the load and clear functions. That also makes the CNTR4U good to study since almost all FPGAs and PLDs use only D flip-flops internally. Each D input is driven by a 2-input multiplexer consisting of two AND gates and an OR gate. The multiplexer output is 0 if the CLR input is asserted. Otherwise, the top AND gate passes the data input (D3, D2, D1, or D0) to the output if LD is asserted. If neither

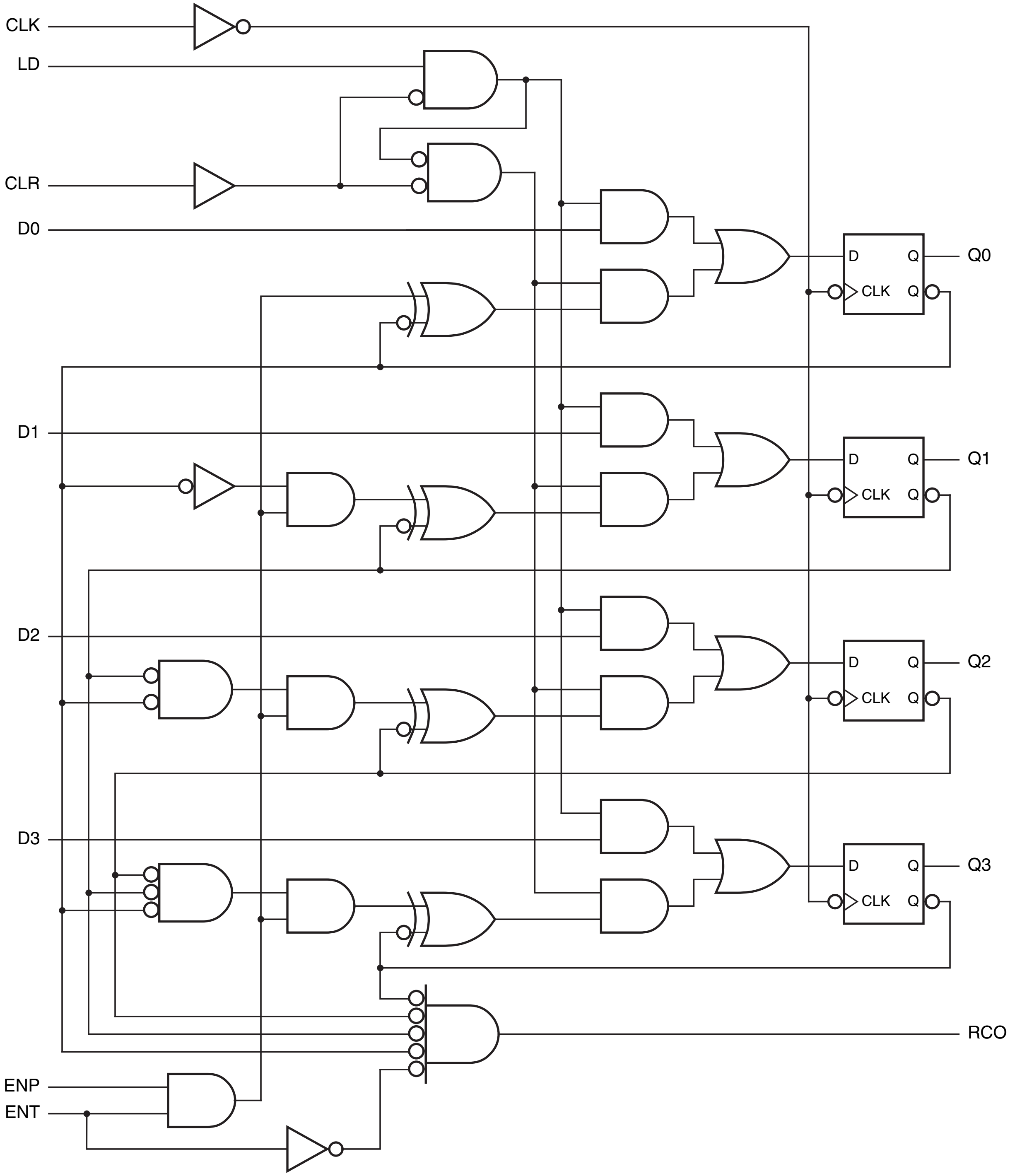


Figure 11-5 Logic diagram for the CNTR4U synchronous 4-bit binary counter.

Table 11-1 State table for the CNTR4U 4-bit binary counter.

Inputs				*Current State*				*Next State*			
CLR	***LD***	***ENT***	***ENP***	***Q3***	***Q2***	***Q1***	***Q0***	***Q3****	***Q2****	***Q1****	***Q0****
1	x	x	x	x	x	x	x	0	0	0	0
0	1	x	x	x	x	x	x	D3	D2	D1	D0
0	0	0	x	x	x	x	x	Q3	Q2	Q1	Q0
0	0	x	0	x	x	x	x	Q3	Q2	Q1	Q0
0	0	1	1	0	0	0	0	0	0	0	1
0	0	1	1	0	0	0	1	0	0	1	0
						. . .					
0	0	1	1	1	1	0	1	1	1	1	0
0	0	1	1	1	1	1	0	1	1	1	1
0	0	1	1	1	1	1	1	0	0	0	0

CLR nor LD is asserted, then the bottom AND gate passes the output of an XNOR gate to the multiplexer output.

The XNOR gates perform the counting function in the CNTR4U, allowing D instead of T flip-flops to be used. One input of each XNOR is the corresponding count bit (Q3, Q2, Q1, or Q0); the other input is asserted, complementing the count bit, if both enables ENP and ENT are asserted and all of the lower-order count bits are 1. Notice that the lower-order count bits are ANDed together by gates with ever-increasing widths in a "parallel enable" structure. The RCO ("ripple carry out") signal indicates a carry from the most significant bit position and is 1 when all of the count bits are 1 and ENT is asserted. This signal can be used for cascading, that is, building wider counters with multiple CNTR4Us.

free-running counter

Even though most counters are designed with enable inputs, counters are often used in a *free-running* mode in which they are enabled continuously. Figure 11-6 shows a traditional logic symbol for the CNTR4U and connections

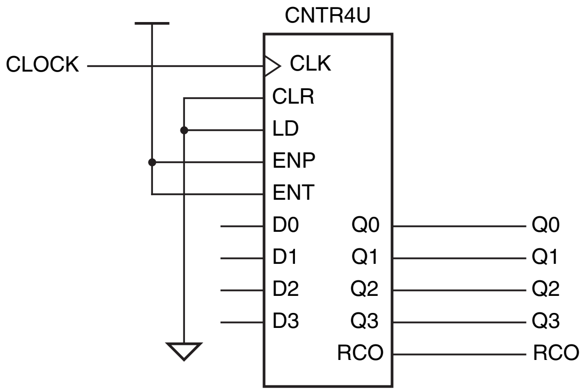


Figure 11-6 Connections for the CNTR4U to operate in a free-running mode.

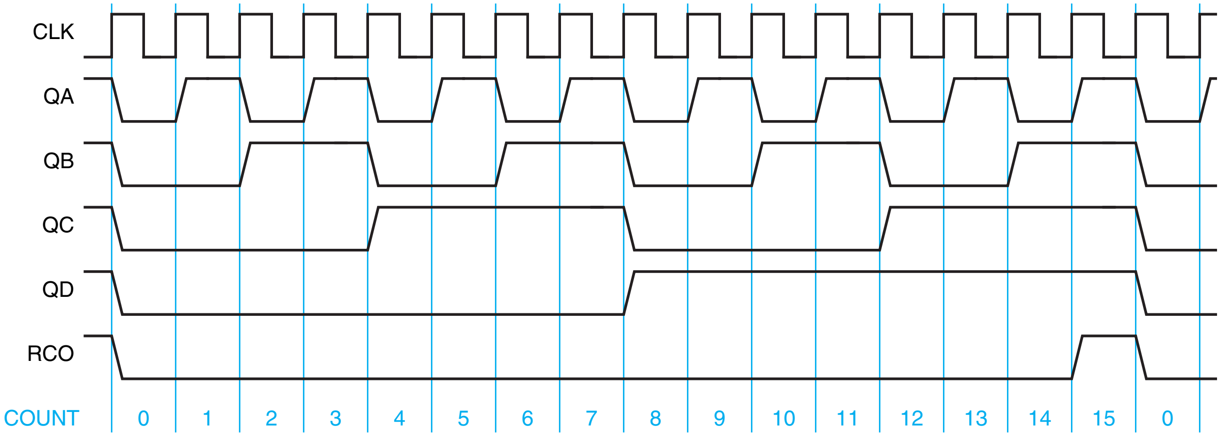


Figure 11-7 Clock and output waveforms for a free-running divide-by-16 counter.

to make it operate in free-running mode. Figure 11-7 shows the resulting output waveforms. Notice that, starting with Q0, each signal has half the frequency of the preceding one. Thus, a free-running CNTR4U can be used as a divide-by-2, -4, -8, and -16 counter.

Although the CNTR4U is a modulo-16 counter, it can be made to count in a modulus less than 16 by using the CLR or LD input to shorten the normal counting sequence. For example, to make a modulo-N counter that counts from 0 to $N - 1$, we create an active-high signal that is asserted in state $N - 1$ and apply that to the CLR input, sending the counter back to state 0 on the next clock tick. This can typically be done with a single AND gate whose inputs are the state bits that are 1 in the binary encoding of $N - 1$. Remember that the CLR input is a synchronous clear; this method wouldn't work properly with an asynchronous clear input. You can also make a modulo-N counter that counts from $16 - N$ to 15 using *no* additional gates (see Drill 11.4).

11.1.4 Decoding Binary-Counter States

A binary counter may be combined with a decoder to obtain a set of 1-out-of-*m*-coded signals, where one signal is asserted in each counter state. This is useful when counters are used to control a set of devices, where a different device is enabled in each counter state. In this approach, each output of the decoder enables a different device.

Figure 11-8 shows how a CNTR4U wired as a modulo-8 counter can be combined with a 3-to-8 decoder to provide eight signals, each one representing a counter state. Figure 11-9 shows typical timing for this circuit. Each decoder output is asserted during a corresponding clock period.

decoding glitches

Notice that the decoder outputs may contain "glitches" on state transitions where two or more counter bits change, even though the CNTR4U outputs are glitch free and the 3-to-8-decoder outputs do not have any static hazards. In a

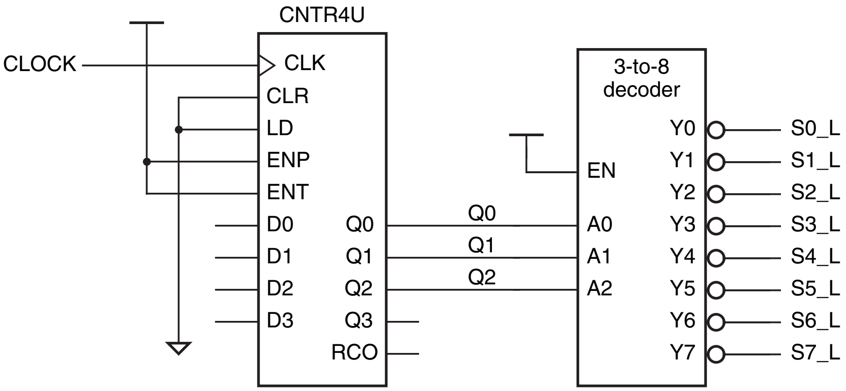


Figure 11-8 A modulo-8 binary counter and decoder.

synchronous counter like the CNTR4U, the outputs don't change at exactly the same time. More important, different signal paths in a decoder can have different delays; for example, the path from A1 to Y1_L may be faster than the path from A0 to Y1_L. Thus, even if the decoder input changes "simultaneously" from 011 to 100, the decoder may behave as if its input were temporarily 001, and the Y1_L output may have a glitch. In the present example, it can be shown that the glitches can occur in *any* realization of the binary decoder function; this problem is an example of a *function hazard.*

function hazard

In most applications, the decoder output signals portrayed in Figure 11-9 would be used as function inputs to registers, counters, and other devices that

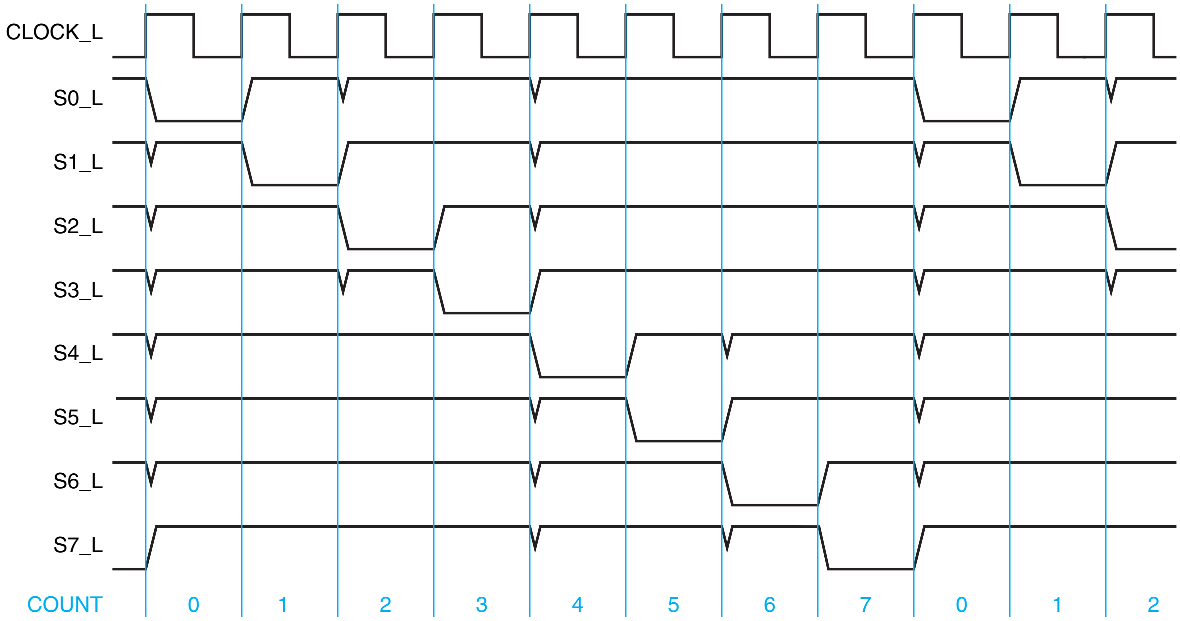


Figure 11-9 Timing diagram for a modulo-8 binary counter and decoder, showing decoding glitches.

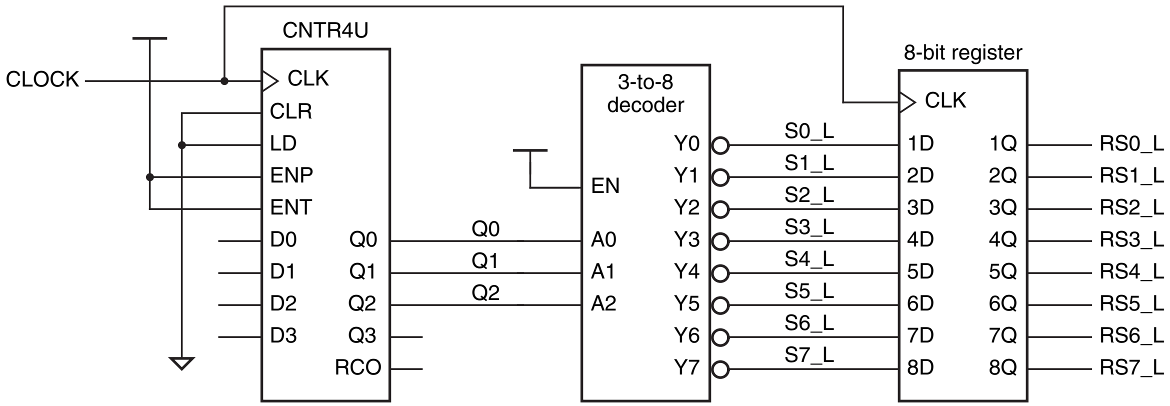


Figure 11-10 A modulo-8 binary counter and decoder with glitch-free outputs.

sample those inputs on a clock edge (e.g., CE in a multibit register with clock enable, or LD or ENP in another CNTR4U). In such a case, the decoding glitches in the figure are not a problem if all devices use the same clock signal, since the glitches occur *after* the clock edge. They are long gone before the next tick comes along, when the decoder outputs are sampled by the other edge-triggered devices. However, the glitches *would* be a problem if they were applied to asynchronous control inputs like the S_L or R_L inputs of an $\overline{\text{S}}$-$\overline{\text{R}}$ latch. Likewise, using such potentially glitchy signals as clocks for edge-triggered devices is a definite no-no.

If necessary, one way to “clean up” the glitches in Figure 11-9 is to connect the decoder outputs to another register that samples the stable decoded outputs on the next clock tick, as shown in Figure 11-10. However, once you decide to add an 8-bit register, a less costly solution is to use an 8-bit “ring counter,” which provides glitch-free decoded outputs directly, without the cost of the 3-bit counter and decoding logic, as we’ll show in Section 11.2.3.

11.1.5 Counters in Verilog

Verilog allows counters to be specified very easily. Program 11-1 is a Verilog model with the behavior of the CNTR4U counter of the preceding subsection. Edge-triggered behavior is obtained using the `posedge` keyword in the first `always` block, and a series of `if-else` statements determine the counter’s next-state behavior based on the control inputs. A second `always` block is used to specify the `RCO` combinational output behavior.

Notice that the counter module uses addition to specify counting, but a typical tool will not synthesize an entire adder to perform this operation. As a minimum, the tool will reduce the amount of logic required because a constant is being added. And if the target technology has features that streamline counter realization (like XOR gates or T flip-flops), it will “infer” them if it can figure out that the designer is specifying a counter.

Program 11-1 Verilog module for CNTR4U 4-bit universal binary counter.

```
module Vrcntr4u( CLK, CLR, LD, ENP, ENT, D, Q, RCO );
  input CLK, CLR, LD, ENP, ENT;
  input [3:0] D;
  output reg [3:0] Q;
  output reg RCO;

  always @ (posedge CLK)  // Create the counter f-f behavior
    if (CLR == 1)                      Q <= 4'd0;
    else if (LD == 1)                  Q <= D;
    else if ((ENT == 1) && (ENP == 1)) Q <= Q + 1;
    else                               Q <= Q;

  always @ (Q or ENT)     // Create RCO combinational output
    if ((ENT == 1) && (Q == 4'd15))    RCO = 1;
    else                               RCO = 0;
endmodule
```

It is very easy to modify the counter module to have other behaviors. For example, Program 11-2 shows how to modify the two `always` blocks for decimal (divide-by-10) counting behavior. The resulting counter counts from 0 to 9 and repeats.

Next, Program 11-3 shows modifications for the excess-3 decimal counting sequence (counting from 3 to 12 and repeating). For some applications, the excess-3 counting sequence is advantageous because its high-order bit is a square wave (50% duty cycle).

Finally, Program 11-4 is a Verilog module for a 4-bit up/down counter which has an extra input UPDN to control the counting direction. Note the use of subtraction for counting down, and the logic for RCO that depends on direction, activating for a count of 15 when counting up, and 0 when counting down.

The clear input of any of these counters can be made asynchronous simply by adding "`posedge CLR`" to the first `always` block's sensitivity list.

Program 11-2 Verilog for a 4-bit decimal counter module `Vrcntr4udec`.

```
always @ (posedge CLK)   // Create the counter f-f behavior
  if (CLR)                              Q <= 4'd0;
  else if (LD)                          Q <= D;
  else if (ENT && ENP && (Q == 4'd9))  Q <= 4'd0;
  else if (ENT && ENP)                  Q <= Q + 1;
  else                                  Q <= Q;

always @ (Q or ENT)     // Create RCO combinational output
  if (ENT && (Q == 4'd9))               RCO = 1;
  else                                  RCO = 0;
```

Program 11-3 Verilog for an excess-3 decimal counter module `Vrexcess3`.

```
  always @ (posedge CLK)   // Create the counter f-f behavior
    if (CLR)                              Q <= 4'd3;
    else if (LD)                          Q <= D;
    else if (ENT && ENP && (Q == 4'd12))  Q <= 4'd3;
    else if (ENT && ENP)                  Q <= Q + 1;
    else                                  Q <= Q;

  always @ (Q or ENT)       // Create RCO combinational output
    if (ENT && (Q == 4'd12))              RCO = 1;
    else                                  RCO = 0;
```

Program 11-4 Verilog module for a 4-bit up/down counter.

```
module Vrupdn4 (CLK, CLR, LD, ENP, ENT, UPDN, D, Q, RCO);
  input CLK, CLR, LD, ENP, ENT, UPDN;
  input [3:0] D;
  output reg [3:0] Q;
  output reg RCO;

  always @ (posedge CLK)       // Create the counter f-f behavior
    if (CLR)                                Q <= 4'd0;
    else if (LD)                            Q <= D;
    else if (ENT && ENP &&  UPDN)           Q <= Q + 1;
    else if (ENT && ENP && !UPDN)           Q <= Q - 1;
    else                                    Q <= Q;

  always @ (Q or ENT or UPDN)  // Create RCO combinational output
    if       (ENT &&  UPDN && (Q == 4'd15))  RCO = 1;
    else if (ENT && !UPDN && (Q == 4'd0 ))  RCO = 1;
    else                                    RCO = 0;
endmodule
```

WHAT ELSE? The "`else Q <= Q`" clauses in Programs 11-1 through 11-4 are not needed; the Verilog compiler, simulator, and synthesizer know that the output of a behaviorally specified flip-flop should be preserved if no new value is assigned at the clock edge. Thus, we didn't include such an `else` clause in the behavioral model of a clock-enabled D flip-flop in Program 10-11 on page 521. However, from the point of view of program readability and maintainability, it seems like a good thing to do when there's a longer list of assignment cases, both here and later in Program 11-11.

Including the "`else Q <= Q`" clause may lead to a more or a less efficient implementation, depending on the synthesis tool. The Xilinx Vivado tool disables the clock for this case, using the clock-enable input that's already be available "for free" in 7-series FPGAs to enable the clock only for cases where something new is assigned to `Q`. A less sophisticated tool may synthesize a multiplexer to feed back the Q outputs into the flip-flop D input in the style of Figure 10-17(a) on page 507.

Program 11-5 Verilog test bench to exercise four 4-bit counters.

```
`timescale 1 ns / 100 ps
module VrcntrTB1 ();
  reg CLK, CLR, LD, ENP, ENT, UPDN;
  reg [3:0] D;
  wire [3:0] cntr4uQ, cntr4decQ, excess3Q, updn4Q;
  wire cntr4uRCO, cntr4decRCO, excess3RCO, updn4RCO;

  always begin           // 10 ns period for clock generation
    #5.5 CLK = 0;        // 5.5 ns HIGH
    #4.0 CLK = 1;        // 4.0 ns LOW
    #0.5 ;               // Plus 0.5 ns HIGH for readability
  end

  Vrcntr4u   U1 ( .CLK(CLK), .CLR(CLR), .LD(LD), .ENP(ENP), .ENT(ENT), .D(D),
                  .Q(cntr4uQ), .RCO(cntr4uRCO) );
  Vrcntr4dec U2 ( .CLK(CLK), .CLR(CLR), .LD(LD), .ENP(ENP), .ENT(ENT), .D(D),
                  .Q(cntr4decQ), .RCO(cntr4decRCO) );
  Vrexcess3  U3 ( .CLK(CLK), .CLR(CLR), .LD(LD), .ENP(ENP), .ENT(ENT), .D(D),
                  .Q(excess3Q), .RCO(excess3RCO) );
  Vrupdn4    U4 ( .CLK(CLK), .CLR(CLR), .LD(LD), .ENP(ENP), .ENT(ENT), .D(D),
                  .UPDN(UPDN), .Q(updn4Q), .RCO(updn4RCO) );

  initial begin
    CLR = 0; LD = 0; ENP = 0; ENT =0; D = 0; UPDN = 0;   // All inputs 0
    #105 ;                     // Wait for FPGA global reset to end
    CLR = 1; D = 4'b1111; #10  // Make sure counter clears
    #10 ;
    CLR = 0; LD = 1; #10       // Now load 1111
    LD = 0; ENP = 1; #10       // No counting yet (ENT not 1)
    ENT = 1; UPDN = 1; #40     // Now count (up) 4 ticks
    UPDN = 0; #40              // Then count (down) 6 ticks
    UPDN = 1; #200             // Finally count (up) 20 ticks
    ENP = 0; #30               // And stop counting
    $stop(1);
  end
endmodule
```

All four variants of the counter can be instantiated and exercised in the same test bench, as shown in Program 11-5. Unlike many of our other test benches, this one is not "self checking;" it merely provides inputs that cause the counters to count. The designer can then check the resulting waveforms for the correct counting sequence, which is straightforward for a typical counter. The waveforms produced by the simulator are shown in Figure 11-11.

In the preceding subsection, we showed how binary-counter outputs can be decoded to create a set of mutually exclusive "enable" inputs for a set of devices. A Verilog module with similar functionality is shown in Program 11-6. It still

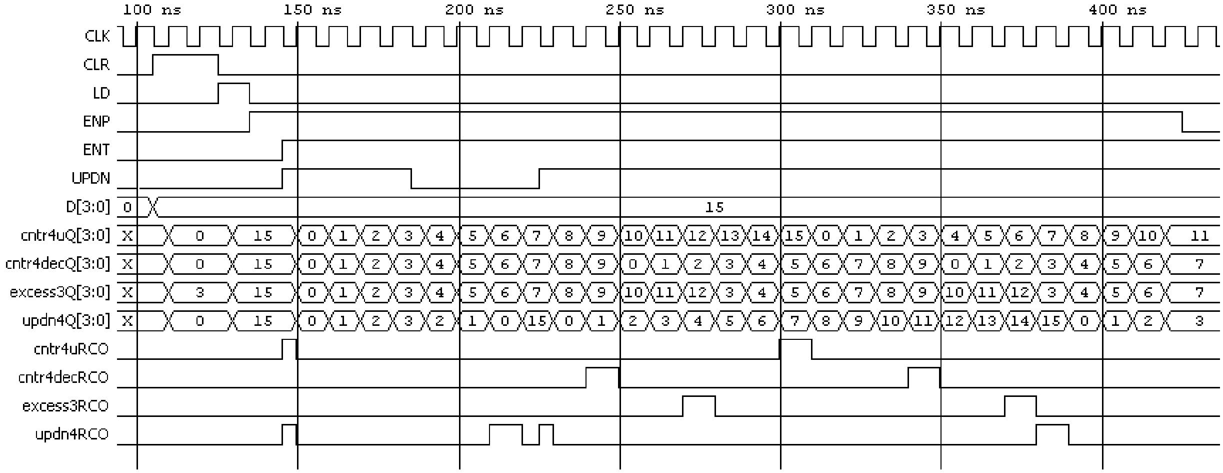


Figure 11-11 Simulation timing diagram for the counter test bench.

contains a 3-bit counter, created by the first `always` block, but unlike our other Verilog counters, its flip-flop bits are not visible outside the module. The second `always` block, which is combinational, decodes the counter outputs to create the external outputs.

We also showed in the preceding subsection how to obtain glitch-free decoded counter outputs by following the counter with a register, and we can create a Verilog module with the same functionality. The declarations for such a module can be exactly the same as in Program 11-6, but we replace its sequential and combinational `always` blocks with the single sequential `always` block shown in Program 11-7.

Program 11-6 Verilog code for a 3-bit counter and decoded outputs.

```
module Vr3bitctrdec ( CLK, CLR, S_L );
  input CLK, CLR;
  output reg [0:7] S_L;
  reg [2:0] Q;
  integer i;

  always @ (posedge CLK)  // Create the counter f-f behavior
    if (CLR)      Q <= 3'd0;
    else          Q <= Q + 1;

  always @ (Q) begin   // Decode counter states to create outputs
      S_L = 8'b11111111;
      for (i=0; i<=7; i=i+1)
        if (i == Q) S_L[i] = 0;
  end
endmodule
```

Program 11-7 Verilog changes for a 3-bit counter module `Vr3bitctrdecreg` with registered, decoded outputs.

```
always @ (posedge CLK) begin
  if (CLR)      Q <= 3'd0;   // Create the counter f-f behavior
  else          Q <= Q + 1;
  S_L <= 8'b11111111;        // Default for outputs is negated
  for (i=0; i<=7; i=i+1)     // Decode counter states to assert
    if (i == Q) S_L[i] <= 0; //   one active-low output
end
```

11.2 Shift Registers

11.2.1 Shift-Register Structure

shift register

serial input

serial output

A *shift register* is an *n*-bit register with a provision for shifting its stored data by one bit position at each tick of the clock. Figure 11-12 shows the structure of a serial-in, serial-out shift register. The *serial input,* SERIN, specifies a new bit to be shifted into one end at each clock tick. This bit appears at the *serial output,* SEROUT, after *n* clock ticks, and is lost one tick later. Thus, an *n*-bit serial-in, serial-out shift register can be used to delay a signal by *n* clock ticks.

serial-in, parallel-out shift register

serial-to-parallel conversion

A *serial-in, parallel-out shift register,* shown in Figure 11-13, has outputs for all of its stored bits, making them available to other circuits. Such a shift register can be used to perform *serial-to-parallel conversion.*

parallel-in, serial-out shift register

parallel-to-serial conversion

Conversely, it is possible to build a *parallel-in, serial-out shift register.* Figure 11-14 shows the general structure of such a device. At each clock tick, the register either loads new data from inputs D1–Dn or it shifts its current contents, depending on the value of the LOAD/SHIFT control input (which could be named LOAD or SHIFT_L). Internally, the device uses a 2-input multiplexer on each flip-flop's D input to select between the two cases. A parallel-in, serial-out shift register can be used to perform *parallel-to-serial conversion.*

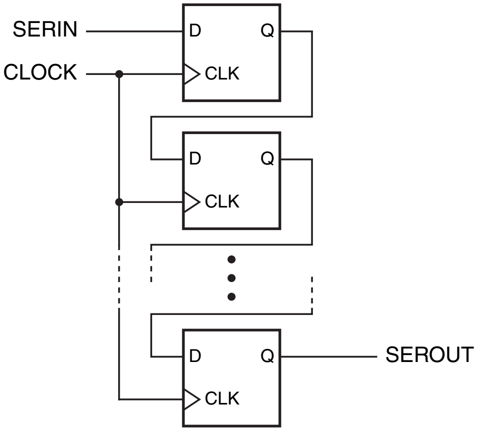


Figure 11-12 Structure of a serial-in, serial-out shift register.

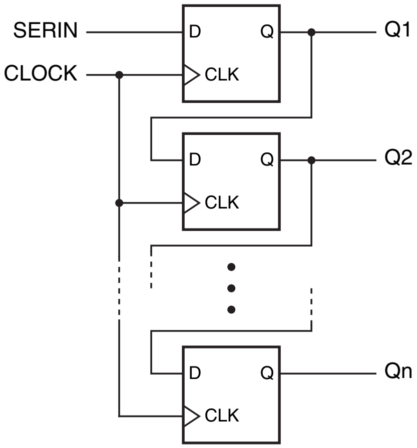


Figure 11-13 Structure of a serial-in, parallel-out shift register.

By providing outputs for all of the stored bits in a parallel-in shift register, we obtain the *parallel-in, parallel-out shift register* shown in Figure 11-15. Such a device is general enough to be used in any of the applications of the previous shift registers.

parallel-in, parallel-out shift register

unidirectional shift register

bidirectional shift register

All of the shift registers that we've shown so far are called *unidirectional shift registers* because they shift in only one direction. A *bidirectional shift register* has the ability to shift in either direction, "left" or "right," depending on the value of a control input. We can combine this enhancement with the ability to load or hold on each clock tick to create a "universal" shift register. Thus, Figure 11-16(a) is the logic diagram for a 4-bit-wide universal shift register with

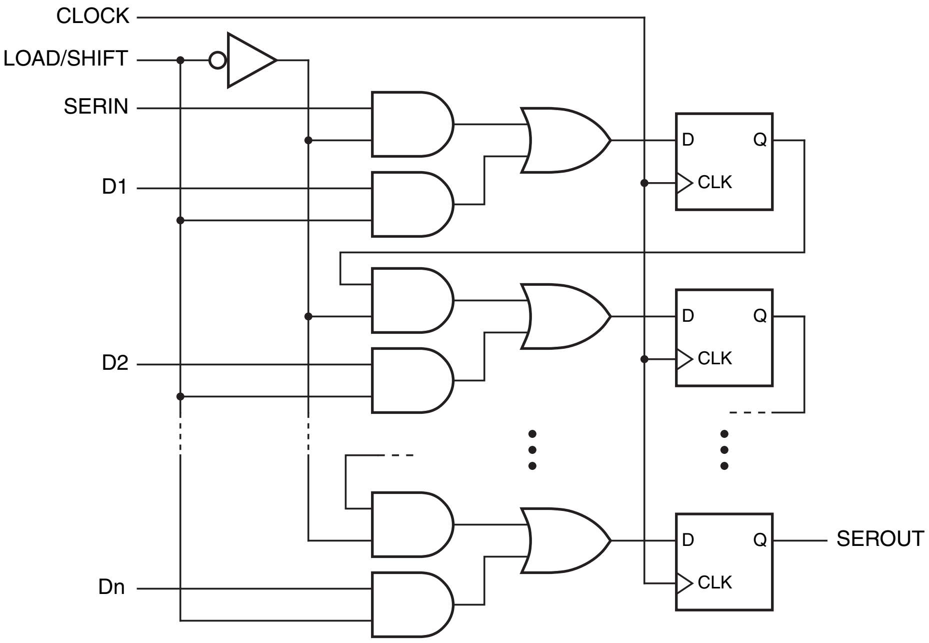


Figure 11-14 Structure of a parallel-in, serial-out shift register.

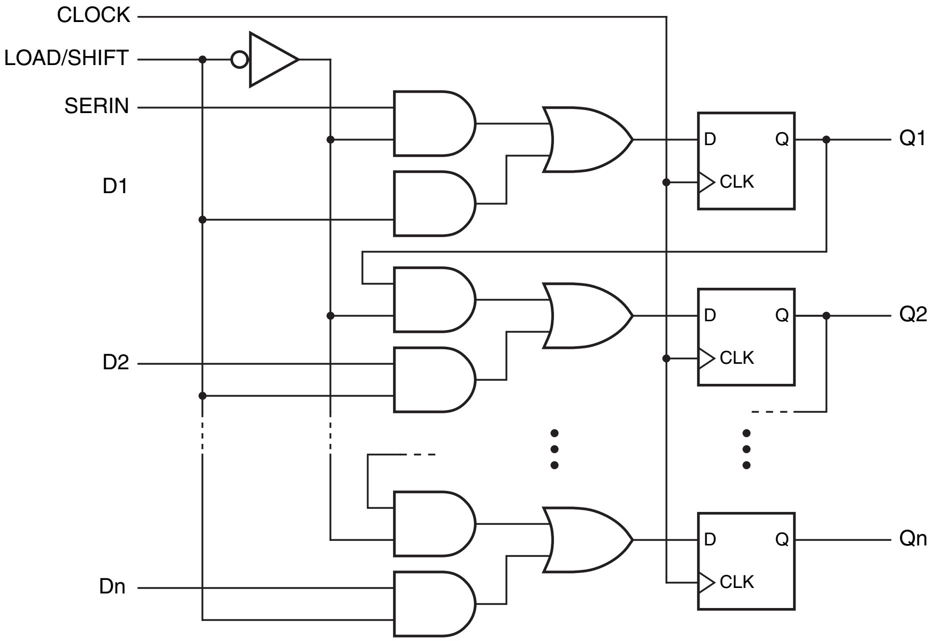


Figure 11-15 Structure of a parallel-in, parallel-out shift register.

synchronous clear that we'll call a SHRG4U, with the logic symbol in (b). The two directions are called "left" and "right," even though the logic diagram and the symbol aren't necessarily drawn that way. In the SHRG4U, *left* means "in the direction from QD to QA," and *right* means "in the direction from QA to QD." Figure 11-16 is consistent with these names if you rotate it 90° clockwise.

left
right

Table 11-2 is a function table for the SHRG4U. The function table is highly compressed, since it does not contain columns for most of the inputs (A–D, RIN, LIN) or the current state QA–QD. Still, by expressing each next-state value as a function of these implicit variables, it almost completely defines the operation of the SHRG4U for all 2^{13} possible combinations of current state and input, and it sure beats a 8192-row table!

Note that the SHRG4U's LIN (left-in) input is conceptually located on the "righthand" side of the circuit, but it is the serial input for *left* shifts. Similarly, RIN is on the "lefthand" side but is the serial input for right shifts.

Table 11-2 Function table for the SHRG4U 4-bit universal shift register.

	Inputs			*Next state*			
Function	***CLR***	***S1***	***S0***	***QA****	***QB****	***QC****	***QD****
Clear	1	x	x	0	0	0	0
Hold	0	0	0	QA	QB	QC	QD
Shift right	0	0	1	RIN	QA	QB	QC
Shift left	0	1	0	QB	QC	QD	LIN
Load	0	1	1	A	B	C	D

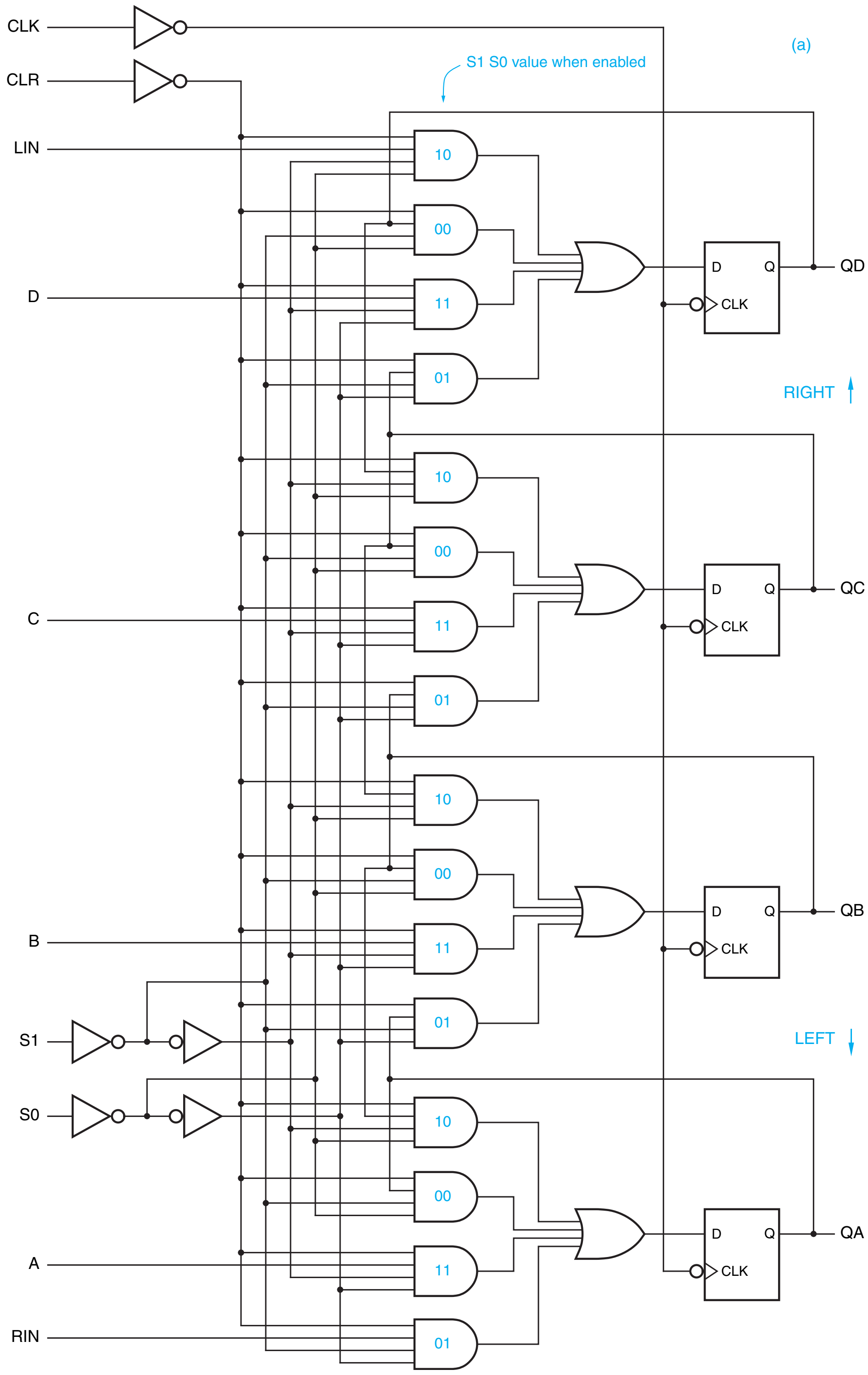
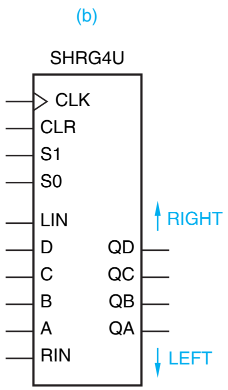


Figure 11-16
SHRG4U 4-bit universal shift register:
(a) logic diagram;
(b) logic symbol.



11.2.2 Shift-Register Counters

shift-register counter

Serial/parallel conversion is a “data” application, but shift registers have “non-data” applications as well. A shift register can be combined with combinational logic to form a state machine whose state diagram is cyclic. Such a circuit is called a *shift-register counter.* Unlike a binary counter, a shift-register counter does not count in an ascending or descending binary sequence, but it is useful in many “control” applications nonetheless. The next three subsections show three different ways to build shift-register counters. Each approach yields a different kind of counting sequence with its own particular advantages.

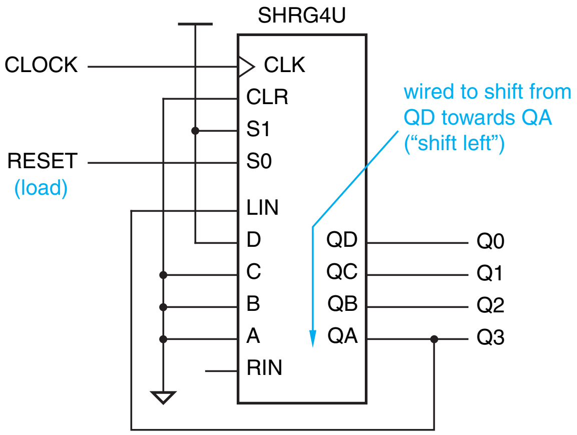
11.2.3 Ring Counters

ring counter

The simplest shift-register counter uses an n-bit shift register to obtain a counter with n states, and is called a *ring counter*. Figure 11-17 is the logic diagram for a 4-bit ring counter. The SHRG4U universal shift register is wired so that S1 S0 is normally 10 and it performs a left shift. However, when RESET is asserted, S1 S0 is 11 and it loads A-D, which is 0001 (refer to the SHRG4U’s function table, Table 11-2 on page 568). Once RESET is negated, the SHRG4U shifts left on each clock tick. The LIN serial input is connected to the “leftmost” output, so the next states are 0010, 0100, 1000, 0001, 0010, …. Thus, the counter visits four unique states before repeating. A timing diagram is shown in Figure 11-18. In general, an n-bit ring counter visits n states in a cycle.

The ring counter in Figure 11-17 has one major problem—it is not robust. If its single 1 output is lost due to a temporary hardware problem, the counter goes to state 0000 and stays there “forever.” Likewise, if an extra 1 output is set (e.g., state 0101 is created), the counter will go through an incorrect cycle of states and stay in that cycle forever. These problems are quite evident if we draw the *complete* state diagram for the counter circuit, which has 16 states. As shown in Figure 11-19, there are 12 states that are not part of the normal counting cycle. If a glitch sends the counter off its normal cycle, it stays off it unless another glitch puts it back.

Figure 11-17
Simplest design for a 4-bit, 4-state ring counter with a single circulating 1.



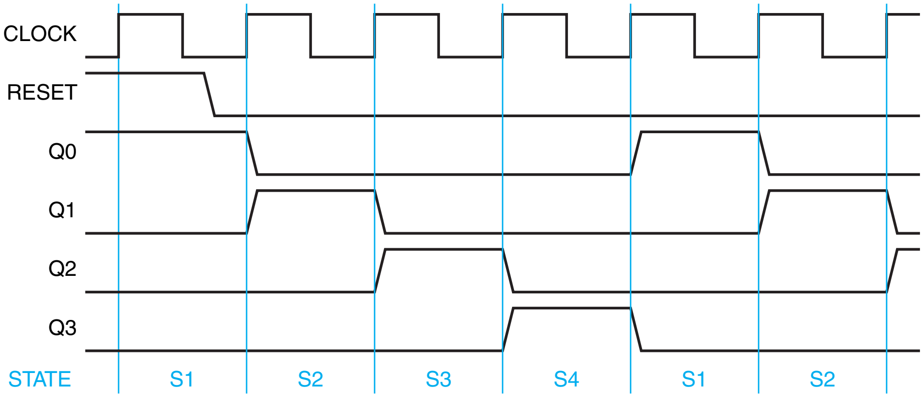


Figure 11-18 Timing diagram for a 4-bit ring counter.

A *self-correcting counter* is designed so that all abnormal states have transitions leading to normal states. Self-correcting counters are desirable for the same reason that we recommended a minimal-risk approach to state assignment at the end of Section 9.3.3: if something unexpected happens, a counter or state machine should go to a "safe" state.

self-correcting counter

A *self-correcting ring counter* circuit is shown in Figure 11-20. The circuit uses a NOR gate to shift a 1 into LIN only when the three least significant bits are 0. This results in the state diagram in Figure 11-21; all abnormal states lead back into the normal cycle. Notice that, in this circuit, an explicit RESET signal is not necessarily required. Regardless of the initial state of the shift register on power-up, it reaches state 0001 within four clock ticks. However, an explicit reset signal should still normally be provided as shown. This ensures that the counter starts up at the same clock tick with other devices in the system and also provides a known starting point in simulation (see Exercise 11.45).

self-correcting ring counter

In the general case, an n-bit self-correcting ring counter uses an $(n-1)$-input NOR gate and corrects an abnormal state within $n-1$ clock ticks.

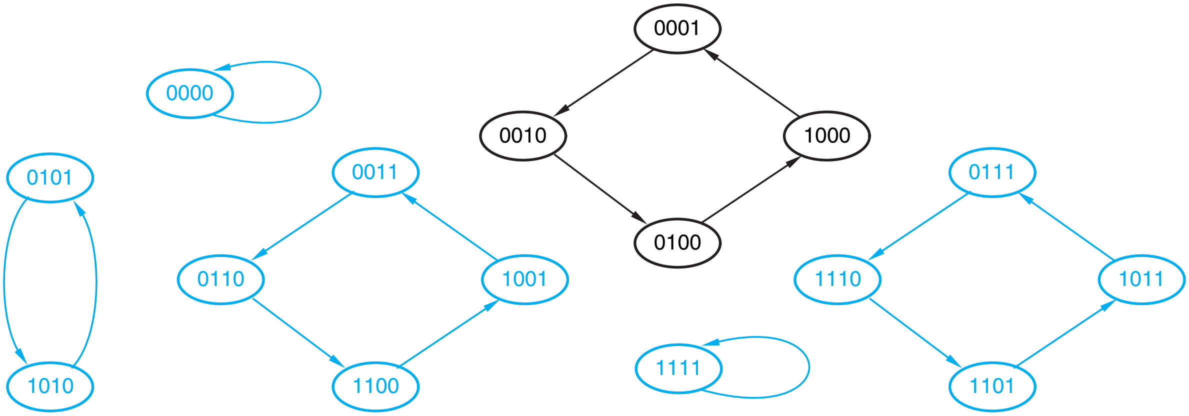


Figure 11-19 State diagram for a simple ring counter.

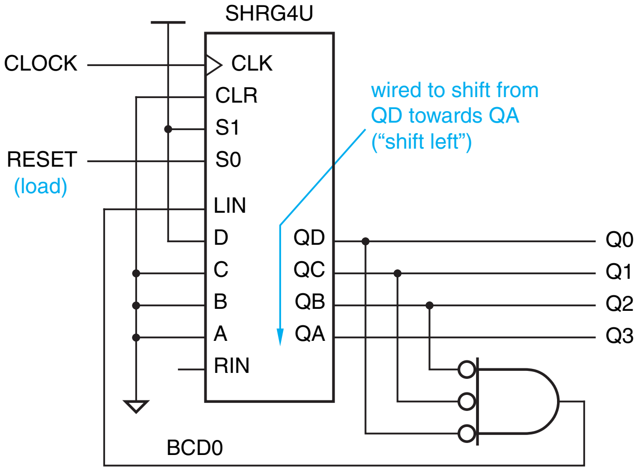


Figure 11-20 Self-correcting 4-bit, 4-state ring counter with a single circulating 1.

The major appeal of a ring counter for control applications is that its states appear in 1-out-of-n decoded form directly on the flip-flop outputs. That is, exactly one flip-flop output is asserted in each state. Furthermore, these outputs are "glitch free"; compare with the binary counter and decoder approach of Figure 11-8 on page 560.

*11.2.4 Johnson Counters

twisted-ring counter
Moebius counter
Johnson counter

An n-bit shift register with the complement of the serial output fed back into the serial input is a counter with $2n$ states and is called a *twisted-ring*, *Moebius*, or *Johnson counter*. Figure 11-22 is the basic circuit for a Johnson counter and Figure 11-23 is its timing diagram. The normal states of this counter are listed in Table 11-3. If both the true and complemented outputs of each flip-flop are available, each normal state of the counter can be decoded with a 2-input AND or NAND gate, as shown in the table. The decoded outputs are glitch free.

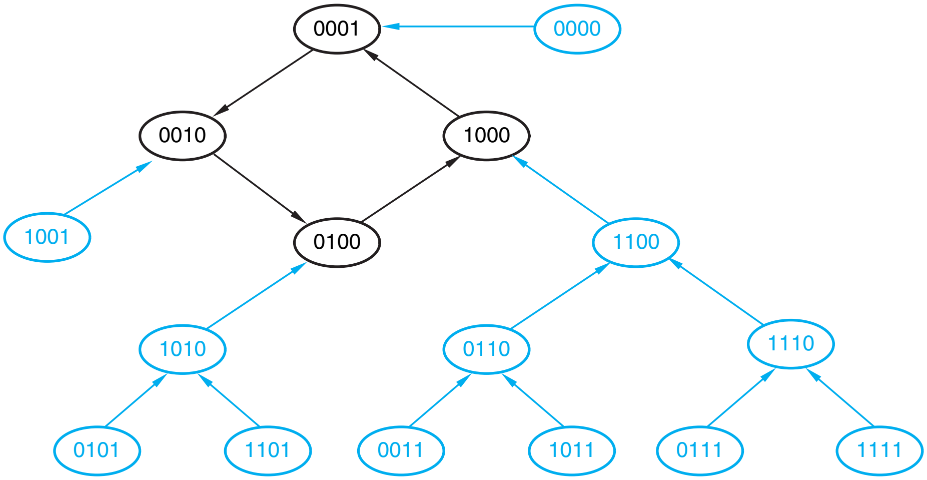


Figure 11-21 State diagram for a self-correcting ring counter.

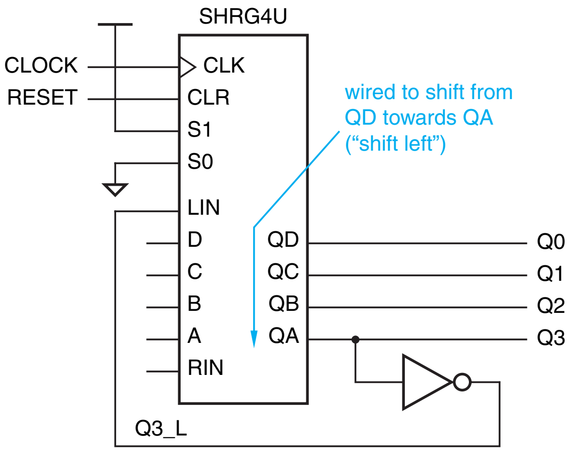


Figure 11-22 Basic 4-bit, 8-state Johnson counter.

Table 11-3 States of a 4-bit Johnson counter.

State Name	***Q3***	***Q2***	***Q1***	***Q0***	***Decoding***
S1	0	0	0	0	Q3′ · Q0′
S2	0	0	0	1	Q1′ · Q0
S3	0	0	1	1	Q2′ · Q1
S4	0	1	1	1	Q3′ · Q2
S5	1	1	1	1	Q3 · Q0
S6	1	1	1	0	Q1 · Q0′
S7	1	1	0	0	Q2 · Q1′
S8	1	0	0	0	Q3 · Q2′

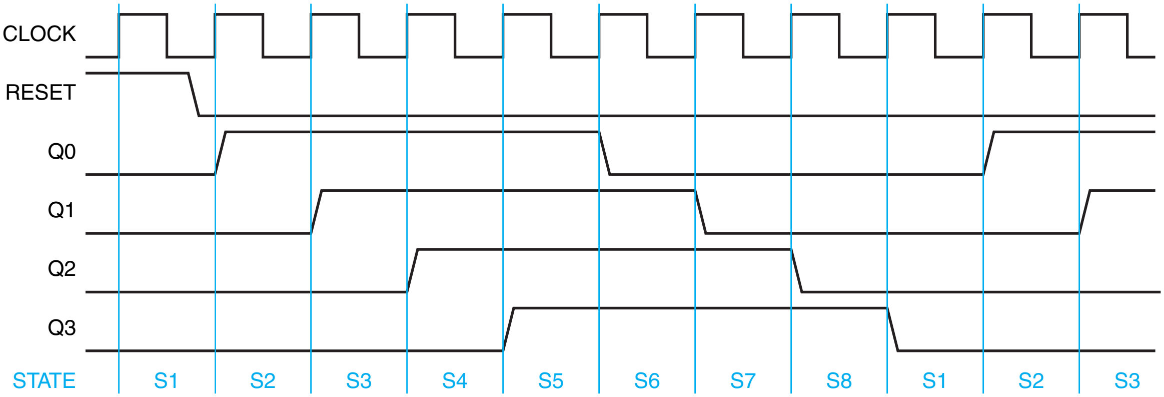


Figure 11-23 Timing diagram for a 4-bit Johnson counter.

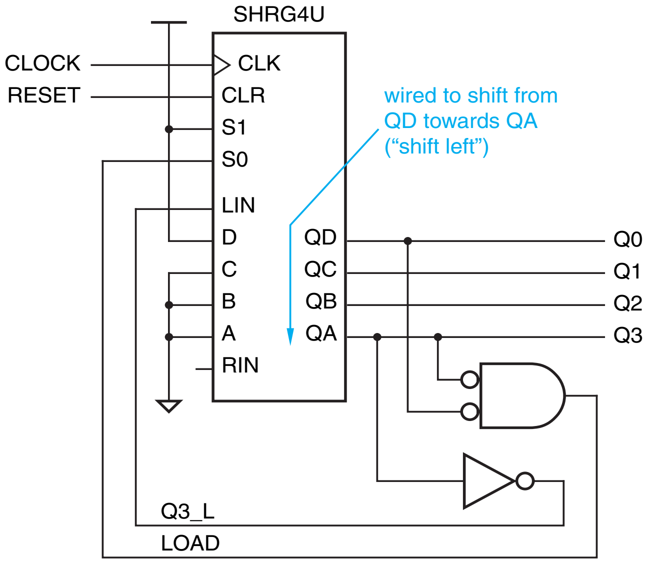


Figure 11-24 Self-correcting 4-bit, 8-state Johnson counter

self-correcting Johnson counter

An n-bit Johnson counter has $2^n - 2n$ abnormal states and is therefore subject to the same robustness problems as a ring counter. A 4-bit *self-correcting Johnson counter* can be designed as shown in Figure 11-24. This circuit loads 0001 as the next state whenever the current state is 0xx0. A similar circuit using a single 2-input NOR gate can perform correction for a Johnson counter with any number of bits. The correction circuit must load 00…01 as the next state whenever the current state is 0x…x0.

THE SELF-CORRECTION CIRCUIT IS ITSELF CORRECT!

We can prove that the Johnson-counter self-correction circuit corrects any abnormal state as follows. An abnormal state can always be written in the form x…x10x…x, since the only states that can't be written in this form are normal states (00…00, 11…11, 01…1, 0…01…1, and 0…01). Therefore, within $n-2$ clock ticks, the shift register will contain 10x…x. One tick later it will contain 0x…x0, and one tick after that the normal state 00…01 will be loaded.

11.2.5 Linear Feedback Shift-Register Counters

maximum-length sequence generator

The n-bit shift register counters that we've shown so far have far less than the maximum of 2^n normal states. An n-bit *linear feedback shift-register (LFSR) counter* can have $2^n - 1$ states, almost the maximum. Such a counter is sometimes called a *maximum-length sequence generator.*

finite fields

The design of LFSR counters is based on the theory of *finite fields,* which was developed by French mathematician Évariste Galois (1811–1832) shortly before he was killed in a duel with a political opponent. The operation of an LFSR counter corresponds to operations in a finite field with 2^n elements.

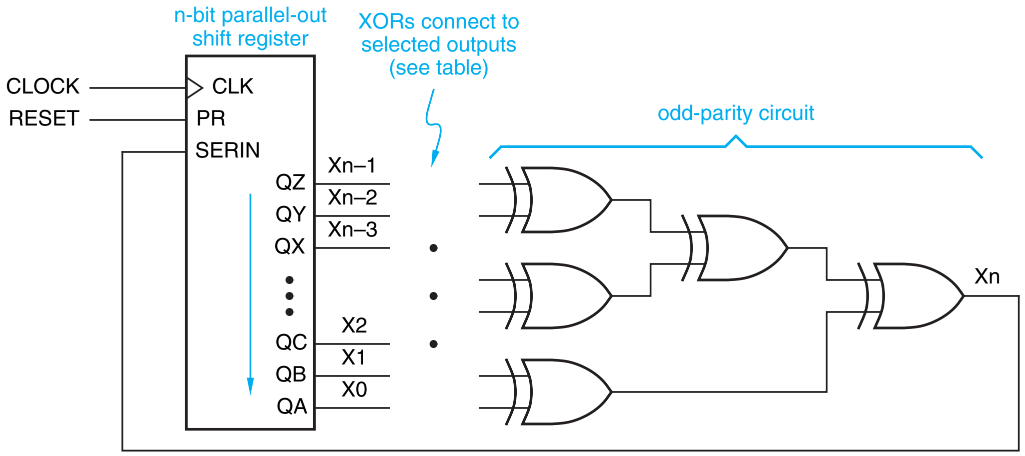

n-bit parallel-out shift register
XORs connect to selected outputs (see table)
odd-parity circuit
CLOCK
RESET
CLK
PR
SERIN
QZ
QY
QX
QC
QB
QA
Xn–1
Xn–2
Xn–3
X2
X1
X0
Xn

Figure 11-25 General structure of a linear feedback shift-register counter.

Figure 11-25 shows the structure of an n-bit LFSR counter. The shift register's serial input is connected to the sum modulo 2 of a certain subset of its output bits. These feedback connections determine the counting sequence. By convention, the outputs are always numbered and shifted in the direction shown.

maximum-length sequence

Using finite-field theory, it can be shown that for any value of n, there exists at least one feedback equation that makes the counter go through all $2^n - 1$ nonzero states before repeating. This is called a *maximum-length sequence.*

Table 11-4 lists feedback equations that yield maximum-length sequences for selected values of n. For each value of n greater than 3, there are many other feedback equations that result in maximum-length sequences, all different.

Table 11-4 Feedback equations for linear feedback shift-register counters.

n	***Feedback Equation***
2	X2 = X1 ⊕ X0
3	X3 = X1 ⊕ X0
4	X4 = X1 ⊕ X0
5	X5 = X2 ⊕ X0
6	X6 = X1 ⊕ X0
7	X7 = X3 ⊕ X0
8	X8 = X4 ⊕ X3 ⊕ X2 ⊕ X0
12	X12 = X6 ⊕ X4 ⊕ X1 ⊕ X0
16	X16 = X5 ⊕ X4 ⊕ X3 ⊕ X0
20	X20 = X3 ⊕ X0
24	X24 = X7 ⊕ X2 ⊕ X1 ⊕ X0
28	X28 = X3 ⊕ X0
32	X32 = X22 ⊕ X2 ⊕ X1 ⊕ X0

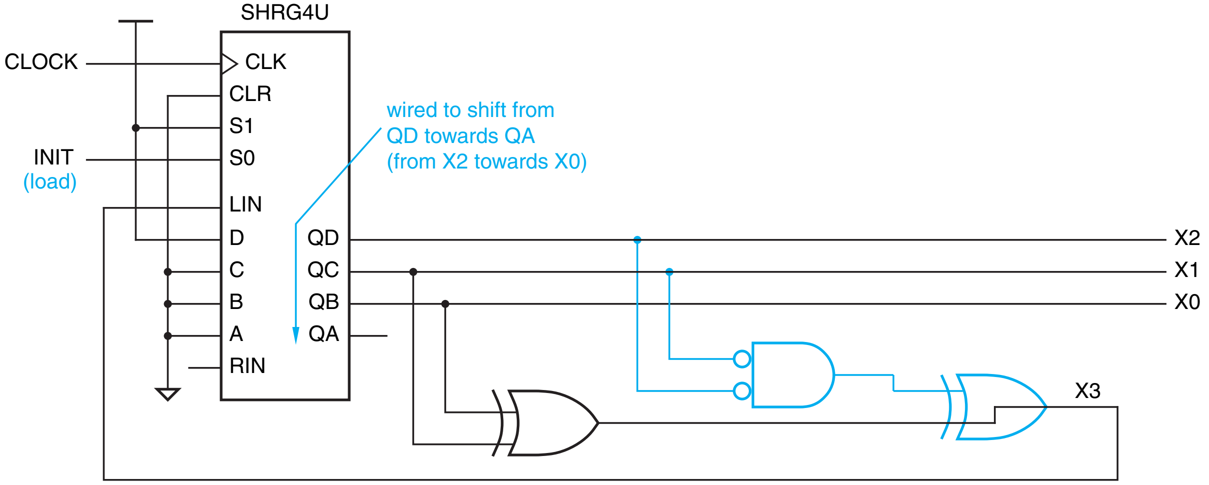


Figure 11-26 A 3-bit LFSR counter; modifications to include the all-0s state are shown in color.

An LFSR counter designed according to Figure 11-25 can never cycle through all 2^n possible states. Regardless of the connection pattern, the next state for the all-0s state is the same—all 0s.

The logic diagram for a 3-bit LFSR counter is shown in Figure 11-26. The state sequence for this counter is shown in the first three columns of Table 11-5. Starting in any nonzero state, 100 at reset and in the table, the counter visits six other states before returning to the starting state, for a total of seven states.

An LFSR counter can be modified to have 2^n states, including the all-0s state, as shown in color for the 3-bit counter in Figure 11-26. The resulting state sequence is given in the last three columns of Table 11-5. In an n-bit LFSR counter, an extra XOR gate and an $n - 1$ input NOR gate connected to all shift-register outputs except X0 accomplishes the same thing.

Table 11-5 State sequences for the 3-bit LFSR counter in Figure 11-26.

Original Sequence			***Modified Sequence***		
X2	***X1***	***X0***	***X2***	***X1***	***X0***
1	0	0	1	0	0
0	1	0	0	1	0
1	0	1	1	0	1
1	1	0	1	1	0
1	1	1	1	1	1
0	1	1	0	1	1
0	0	1	0	0	1
1	0	0	0	0	0
.	.	.	1	0	0
.	.	.	.	.	.

The states of an LFSR counter are not visited in binary counting order. However, LFSR counters are typically used in applications where this characteristic is an advantage. A major application of LFSR counters is in generating test inputs for logic circuits. In many cases, the “pseudorandom” counting sequence of an LFSR counter may be more likely than a binary counting sequence to detect errors, especially if only a subset of the 2^n possible n-bit values are used. LFSRs are also used in the encoding and decoding circuits for certain error-detecting and error-correcting codes, including CRC codes, which we introduced in Section 2.15.4.

In data communications, LFSR counters are often used to “scramble” and “descramble” the serial data patterns transmitted by high-speed modems and network interfaces, including 100-Mbps and 1-Gbps Ethernet. This is done by clocking the LFSR at the same frequency and XORing one bit of its output with successive bits of the user’s serial data stream. Even when the user data stream contains a long run of 0s or 1s (not uncommon), combining it with the LFSR’s pseudorandom output improves the DC balance of the transmitted signal and creates a rich set of transitions that allows clocking information to be recovered more easily at the receiver. For descrambling, the receiver uses an LFSR with the same counting sequence, initialized to start at the same point in the incoming data stream, and XORs the same LFSR output bit with the data stream.

WORKING IN THE FIELD

A finite field has a finite number of elements and two operators, addition and multiplication, that satisfy certain properties. An example of a finite field with P elements, where P is any prime, is the set of integers modulo P. The operators in this field are addition and multiplication modulo P.

According to finite-field theory, if you start with a nonzero element E and repeatedly multiply by a “primitive” element α, after $P-2$ steps you will generate the rest of the field’s nonzero elements in the field before getting back to E. It turns out that in a field with P elements, any integer in the range $2, \ldots, P-1$ is primitive. You can try this yourself using $P = 7$ and $\alpha = 2$, for example. The elements of the field are $0, 1, \ldots, 6$, and the operations are addition and subtraction modulo 7.

The paragraph above gives the basic idea behind maximum-length sequence generators. However, to apply them to a digital circuit, you need a field with 2^n elements, where n is the number of bits required by the application. On one hand, we’re in luck, because Galois proved that there exist finite fields with P^n elements for any integer n, as long as P is prime, including $P = 2$. On the other hand, we’re out of luck, because when $n > 1$, the operators in fields with P^n (including 2^n) elements are quite different from ordinary integer addition and multiplication. Also, primitive elements are harder to find.

If you enjoy math, you’d be fascinated by the finite-field theory used by LFSR circuits for maximum-length sequence generators and other applications; see the References. Otherwise, you can confidently follow the “cookbook” approach here.

Program 11-8 Verilog module for a serial-in, parallel-out 8-bit shift register.

```
module Vr8bitSRparout ( CLK, CLR, SERIN, Q );
  input CLK, CLR, SERIN;
  output reg [WID-1:0] Q;
  parameter WID = 8;

  always @ (posedge CLK)
    if (CLR == 1) Q <= 0;            //Synchronous clear
    else Q <= {Q[WID-2:0], SERIN};   // Shift
endmodule
```

11.2.6 Shift Registers in Verilog

It's easy to describe shift registers behaviorally in Verilog, including all of the types and applications that we encountered in previous subsections; here we'll look at several of them.

A serial-in, parallel-out shift register is coded behaviorally in the Verilog module in Program 11-8, which has a few aspects worth mentioning. The shift-register width is parameterized, with a default of 8 bits. Concatenation "`{}`" and part-select "`[]`" are used to construct the shifted 8-bit vector from the rightmost bits of `Q` and the serial input—this is a "left" shift. You could also shift using a Verilog shift operator, as in "`Q <= (Q<<1) | SERIN`," but only if you understand exactly how Verilog works in this construction: the rightmost bit of the shifted `Q` is 0, and `SERIN` is padded with 0s on the left before the OR operation (also see Drill 11.16). The module in Program 11-8 may be modified to obtain a serial-in, *serial*-out shift register by declaring `Q` as a `reg` only (not `output`), and assigning the value of `Q[7]` to a separately declared output wire `SEROUT`.

A behavioral Verilog module corresponding to the SHRG4U universal 4-bit shift register is shown in Program 11-9. Concatenation is used to group the

Program 11-9 Verilog module for a universal 4-bit shift register.

```
module Vrshrg4u( CLK, CLR, RIN,LIN, S0,S1, A,B,C,D, QA,QB,QC,QD );
  input CLK, CLR, S0, S1, RIN, LIN, A, B, C, D;
  output reg QA, QB, QC, QD;

  always @ (posedge CLK) begin
    if (CLR == 1'b1) {QA,QB,QC,QD} <= 4'b0;
    else case ({S1,S0})
      2'b00: ;                                // Hold
      2'b01: {QA,QB,QC,QD} <= {RIN,QA,QB,QC}; // Shift right
      2'b10: {QA,QB,QC,QD} <= {QB,QC,QD,LIN}; // Shift left
      2'b11: {QA,QB,QC,QD} <= {A,B,C,D};      // Load
      default: {QA,QB,QC,QD} <= 4'bx;         // shouldn't occur
    endcase
  end
endmodule
```

inputs and the outputs into vectors to more easily describe and code the shifting operations. A `case` statement is used to select the appropriate operation as a function of `S1` and `S0`, including no operation in the 00 case, which results in the register value being held. This is a “full case” (the choice list contains all combinations of the selection expression), so the `default` case should never occur, except in simulation if an `x` or `z` value is present on `S1` or `S0`.

A test bench for the shift-register module is shown in Program 11-10. Its testing strategy has three parts that check the module’s operation for a comprehensive set of inputs. The first part checks the load, hold and synchronous clear functions for all 16 combinations of the `A`–`D` inputs. Continuing on the next page, the second part checks the right-shift function for all 64 combinations of the `A`–`D` *and* the `RIN` and `LIN` inputs. The last two are especially important since there can easily be coding errors where these inputs are used incorrectly (e.g., swapped). Note that the test bench compares the UUT’s right-shift results against ones that use Verilog’s built-in shift operations, a double-check on the alternative formulation in the UUT, which uses part-selects and concatenation. The third part of the test bench does similar checks for the left-shift function.

Program 11-10 Verilog test bench for the 4-bit universal shift register (part 1).

```
module Vrshrg4u_tb() ;
reg Tclk, CLR, S0, S1, RIN, LIN;
reg [3:0] I;    // A-D = I[3:0]
wire [3:0] Q;   // QA-QD = Q[3:0]

Vrshrg4u UUT ( .CLK(Tclk), .CLR(CLR), .RIN(RIN), .LIN(LIN), .S0(S0), .S1(S1),
               .A(I[3]), .B(I[2]), .C(I[1]), .D(I[0]),
               .QA(Q[3]), .QB(Q[2]), .QC(Q[1]), .QD(Q[0]) );

always begin
  #0.5 ; Tclk = 1'b1; #5 ;             // Rising edges will occur at 10.5, 20.5, etc.
  Tclk = 1'b0; #4.5 ;
end

initial begin : TB
integer ii, j;
  #116 ;                             // Wait for FPGA global reset to end
  RIN = 1'b0; LIN = 1'b0;            // Don't check RIN and LIN yet
  $display("Starting load, hold, and clear test");
  for (ii=0; ii<=15; ii=ii+1) begin  // Do loads and holds for all data-input combs.
    CLR = 1'b0; {S1,S0} = 2'b11; I[3:0] = ii; #10 ;  // Load next value, wait for tick
    if (Q != I[3:0]) $display("S1S0=11, ABCD=%4b, QA-QD=%4b, load failed", I, Q);
    {S1,S0} = 2'b00; #10 ;           // hold value, wait for tick
    if (Q != I[3:0]) $display("S1S0=00, ABCD=%4b, now QA-QD=%4b, hold fails",I,Q);
    CLR = 1'b1; #10 ;                // Do clear and give it a cycle to take effect
    if (Q != 4'b0) $display("CLR=1, QA-QD=%4b, clear failed", Q);
  end
  $display("Clear, load, and hold test completed");
  CLR = 1'b0;                        // No clear for the rest of the test bench
```

Program 11-10 (parts 2 and 3)

```
  $display("Starting shift-right test for all states");
  for (ii=0; ii<=63; ii=ii+1) begin  // Now test right shifts from all starting states
    {S1,S0} = 2'b11; {LIN, RIN, I[3:0]} = ii[5:0]; #10 ; // Load next, wait for tick
    {S1,S0} = 2'b01;  #10 ;          // Shift right, wait for tick
    if (Q != ((I>>1) | (RIN<<3)) )
      $display("S1S0=01, old QA-QD=%4b, LIN,RIN=%2b, QA-QD=%4b, shift-right failed",
          I, {LIN,RIN}, Q);
  end
  $display("All states shift-right test completed");
  $display("Starting shift-left test for all states");
  for (ii=0; ii<=63; ii=ii+1) begin  // Now test left shifts from all starting states
    {S1,S0} = 2'b11; {LIN, RIN, I[3:0]} = ii[5:0]; #10 ; // Load next, wait for tick
    {S1,S0} = 2'b10;  #10 ;          // Shift left, wait for tick
    if (Q != ((I<<1) | {3'b000,LIN}) )
      $display("S1S0=10, old QA-QD=%4b, LIN,RIN=%2b, QA-QD=%4b, shift-left failed",
          I, {LIN,RIN}, Q);
  end
  $display("All states shift-left test completed");
end
endmodule
```

Next, we consider a universal 8-bit parallel-in, parallel-out shift register with an extended set of functions, controlled by three function-select inputs as shown in Table 11-6. Besides the hold, load, and shift functions of the SHRG4U, it performs circular and arithmetic shift operations as defined in the table. Corresponding behavioral Verilog code is shown in Program 11-11. A case statement is used to specify the shift register's operation for the possible values of the

Table 11-6 Function table for an extended-function 8-bit shift register.

	Inputs				*Next state*							
Function	***CLR***	***S2***	***S1***	***S0***	***Q7****	***Q6****	***Q5****	***Q4****	***Q3****	***Q2****	***Q1****	***Q0****
Clear	1	0	0	0	0	0	0	0	0	0	0	0
Hold	0	x	x	x	Q7	Q6	Q5	Q4	Q3	Q2	Q1	Q0
Load	0	0	0	1	D7	D6	D5	D4	D3	D2	D1	D0
Shift right	0	0	1	0	RIN	Q7	Q6	Q5	Q4	Q3	Q2	Q1
Shift left	0	0	1	1	Q6	Q5	Q4	Q3	Q2	Q1	Q0	LIN
Shift circular right	0	1	0	0	Q0	Q7	Q6	Q5	Q4	Q3	Q2	Q1
Shift circular left	0	1	0	1	Q6	Q5	Q4	Q3	Q2	Q1	Q0	Q7
Shift arithmetic right	0	1	1	0	Q7	Q7	Q6	Q5	Q4	Q3	Q2	Q1
Shift arithmetic left	0	1	1	1	Q6	Q5	Q4	Q3	Q2	Q1	Q0	0

Program 11-11 Verilog module for an extended-function 8-bit shift register.

```
module Vrshrg8ext ( CLK, CLR, RIN, LIN, S, D, Q );
  input CLK, CLR, RIN, LIN;
  input [2:0] S;
  input [7:0] D;
  output reg [7:0] Q;

  always @ (posedge CLK)
    if (CLR == 1) Q <= 0;
    else case (S)
      3'd0: Q <= Q;               // Hold
      3'd1: Q <= D;               // Load
      3'd2: Q <= {RIN, Q[7:1]};   // Shift right
      3'd3: Q <= {Q[6:0], LIN};   // Shift left
      3'd4: Q <= {Q[0], Q[7:1]};  // Shift circular right
      3'd5: Q <= {Q[6:0], Q[7]};  // Shift circular left
      3'd6: Q <= {Q[7], Q[7:1]};  // Shift arithmetic right
      3'd7: Q <= {Q[6:0], 1'b0};  // Shift arithmetic left
      default Q <= 8'bx;          // should not occur
    endcase
endmodule
```

select inputs S[2:0]. As before, concatenation and part-selects are used to construct the shifted 8-bit vector from seven bits of Q and the appropriate eighth bit.

Moving on to shift-register counters, Program 11-12 is a Verilog module for a self-synchronizing 8-bit ring counter. We've add two functions that weren't present in our previous design in Figure 11-20 on page 572: counting occurs only if CNTEN is asserted, and an INIT input is provided to force the counter into an initial state S[7:0]=00000001. Both INIT and CNTEN are synchronous inputs, sampled on the rising edge of the clock.

Perhaps the most interesting feature in Program 11-12 is its use of "&" as the boolean reduction AND operator for a vector, something you don't see too often. The expression ”&(~S[6:0])” complements the seven high-order bits of S, and then ANDs them together. Its value is 1 if and only if the low-order bits of S are 0000000, exactly what is needed as the serial input for self synchronization using the method described in Section 11.2.3.

Program 11-12 Verilog module for an 8-bit self-synchronizing ring counter.

```
module Vr8bitringctr ( CLK, INIT, CNTEN, S );
  input  CLK, INIT, CNTEN;
  output reg [7:0] S;

  always @ (posedge CLK)
    if (INIT == 1)       S <= 8'b00000001;          //Synchronous initialization
    else if (CNTEN == 1) S <= {S[6:0], &(~S[6:0])}; // Shift, w/ self-sync logic
    else                 S <= S;      // not needed, S won't change if omitted
endmodule
```

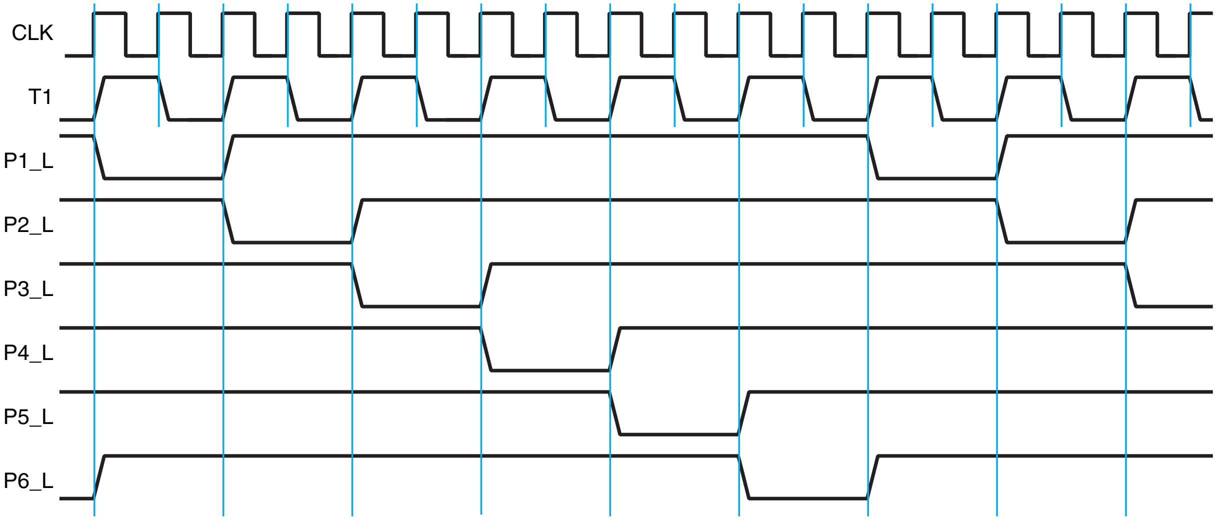


Figure 11-27 Six-phase timing waveforms required in a certain digital system.

11.2.7 Timing-Generator Examples

Ring counters are often used to generate multiphase clocks or enable signals in digital systems, and the requirements in different systems are many and varied. The ability to easily model and modify such a counter's behavior easily is a distinct advantage of an HDL-based design.

Figure 11-27 shows a set of clock or enable signals that might be required in a digital system with six distinct phases of operation. Each phase lasts for two ticks of a master clock signal, CLK, during which the corresponding active-low phase-enable signal Pi_L is asserted. We can obtain this sort of timing from a ring counter if we provide an extra flip-flop T1 to distinguish the two ticks of each phase, so that a shift occurs on the *second* tick of each phase. We'll also define a few control inputs for additional functionality:

RESET When this input is asserted, no outputs are asserted. The counter always goes to the first tick of phase 1 after RESET is negated.

RUN When asserted, this input allows the counter to advance to the second tick of the current phase, or to the first tick of the next phase; otherwise, the current tick of the current phase is extended.

RESTART Asserting this input causes the counter to go back to the first tick of phase 1, even if RUN is not asserted.

A Verilog module that provides the corresponding behavior is shown in Program 11-13. A 6-bit active-high variable, IP, is used for what eventually becomes the circuit's output; this internal signal is inverted by a continuous-assignment statement to produce the required 6-bit active-low output P_L. During reset, IP is held at all-0s, so no output is asserted. The shift register is

Program 11-13 Verilog module for a six-phase timing generator.

```
module Vrtimegen6 ( CLK, RESET, RUN, RESTART, P_L );
  input CLK, RESET, RUN, RESTART;
  output [1:6] P_L;
  reg [1:6] IP;  // internal active-high phase signals
  reg T1;        // first tick within phase

  always @ (posedge CLK)
    if (RESET == 1) begin T1 <= 1; IP <= 6'b0; end
    else if ( (IP == 6'b0) || (RESTART == 1) )
      begin T1 <= 1; IP <= 6'b100000; end
    else if (RUN == 1)
      begin T1 <= ~T1; if (T1==0) IP <= {(IP[1:5]==0),IP[1:5]}; end

  assign P_L = ~IP;  // active-low phase outputs
endmodule
```

loaded with a single 1 just after reset is negated (determined by recognizing the all-0s state) or if RESTART is asserted. If RUN is asserted, T1 is complemented and the register is shifted if T1 is 0. The shift uses self-correcting logic as explained previously in connection with Figure 11-20 on page 572.

A test bench for the timing generator is shown in Program 11-14. This module does not check the UUT's outputs against its spec; it only applies a clock

Program 11-14 Test bench for the timing generator.

```
module Vrtimegen_tb ();
  reg CLK, RESET, RUN, RESTART;
  wire [1:6] P_L;

  Vrtimegen6 UUT ( .CLK(CLK), .RESET(RESET), .RUN(RUN), .RESTART(RESTART), .P_L(P_L) );

  always begin // create free-running test clock with 10 ns period
    #0.5 CLK = 1; #5; // 5 ns high  (small offset for
    CLK = 0; #4.5;    // 5 ns low    waveform readability)
  end

  initial begin
    RESET = 1; RESTART = 0; RUN = 0;
    #115
    RESET = 0;   #20 RUN = 1;     #30
    RESTART = 1; #10 RESTART = 0; #50
    RESTART = 1; #10 RESTART = 0; #30  RUN = 0; #20 RUN = 1; #40
    RESTART = 1; #20 RESTART = 0; #100 RUN = 0; #10 RUN = 1; #40
    RESTART = 1; #10 RESTART = 0; #150
    RESTART = 1; #10 RESTART = 0; #180
    $stop(1);
  end
endmodule
```

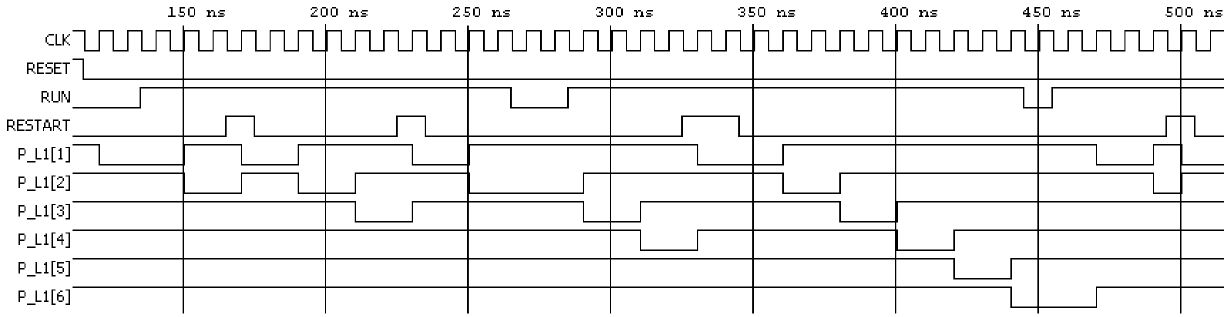


Figure 11-28 Timing waveforms produced by test bench for `Vrtimegen6` module.

and a sequence of control values, requiring the user to check the resulting output waveforms visually. Part of the resulting waveforms are shown in Figure 11-28. Several aspects of the UUT's operation can be checked in the waveforms:

- When `RESET` is released, operation begins in phase 1 as required. Since `RUN` is still negated, the outputs are "stuck" in the first half of phase 1. After `RUN` is asserted, they advance to the second half of phase 1 and then phase 2.
- Asserting `RESTART` returns the output sequence back to phase 1.
- `RESTART` may last longer than one tick, which lengthens phase 1.
- `RESTART` may occur at the end or in the middle of a 2-tick phase. In the latter case, the last tick of that phase is aborted.
- Negating `RUN` lengthens the current phase accordingly.
- After phase 6, the outputs cycle back to phase 1 as expected.

The set of test cases in a test bench like this one are only as comprehensive as the designer's thinking about various situations that might occur, and they check the UUT only as well as the designer can visually recognize incorrect or unwanted outputs. This example does not show every possible situation, nor does it check the results automatically like a self-checking test bench would. But it's good enough to show that the basic operation of the module is correct and to exhibit a couple of behaviors that the designer might want to change or specify more fully in the spec and the resulting module (see Exercise 11.55).

Before showing another timing generator, this is a good place to introduce an alternative test-bench file structure that yields essentially the same test bench but is preferred by some designers and in some environments. We'll see the utility of this different structure when we get to the next timing generator.

The test bench in Program 11-14, and each of our test benches before it, declares the UUT's input and output signals, instantiates the UUT, and then follows that with Verilog code for stimulating the UUT. Some of our previous test benches have also included code here to check the UUT's outputs, as well as

Program 11-15 Alternate-style top-level test bench for the timing generator.

```
module Vrtimegen_tba ();
  reg CLK, RESET, RUN, RESTART;
  wire [1:6] P_L;

  Vrtimegen6 UUT ( .CLK(CLK), .RESET(RESET), .RUN(RUN), .RESTART(RESTART), .P_L(P_L) );

`include "Vrtimegen_stim.v"

endmodule
```

"helper" tasks to aid the checking and displaying of results. The alternative file structure has a much smaller top-level test-bench file with just the first two parts followed by an `` `include `` statement:

- Declare the UUT's input and output signals.
- Instantiate one or more UUTs.
- Use an `` `include `` statement to fetch a "stimulus file" that contains Verilog test code for generating clocks and stimulus patterns, checking results, and defining local variables and "helper" tasks as needed.

Note that the stimulus file does not define a module, but it exists within a module which is the top-level test bench itself. The top-level test bench and stimulus file in the alternate style are shown in Programs 11-15 and 11-16, respectively; each was simply extracted from our original test bench in Program 11-14. We'll reuse Program 11-16 in the other top-level test benches in this subsection.

Program 11-16 Stimulus file for alternate-style timing-generator test bench.

```
always begin // create free-running test clock with 10 ns period
   #0.5 CLK = 1; #5; // 5 ns high  (small offset for
   CLK = 0; #4.5;    // 5 ns low    waveform readability)
 end

 initial begin
   RESET = 1; RESTART = 0; RUN = 0;
   #115
   RESET = 0;   #20 RUN = 1;     #30
   RESTART = 1; #10 RESTART = 0; #50
   RESTART = 1; #10 RESTART = 0; #30  RUN = 0; #20 RUN = 1; #40
   RESTART = 1; #20 RESTART = 0; #100 RUN = 0; #10 RUN = 1; #40
   RESTART = 1; #10 RESTART = 0; #150
   RESTART = 1; #10 RESTART = 0; #180
   $stop(1);
 end
```

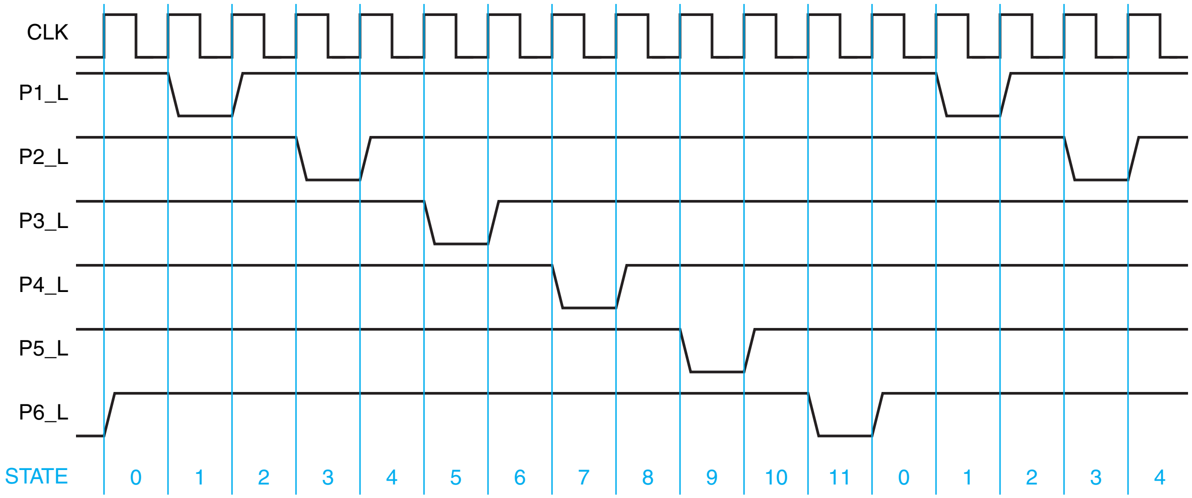


Figure 11-29 Modified timing waveforms for a digital system.

Now we can show a modification of previous timing generator that is useful for some applications. The new design must produce output waveforms that are asserted only during the second tick of each two-tick phase, producing the waveforms shown in Figure 11-29. The active parts of the Pi_L waveforms are guaranteed be completely non-overlapping, even during the brief transitions between phases. This is important in some applications, for example, providing the output-enable inputs for multiple devices that may drive the same three-state bus. This change has a subtle but important effect on the design approach.

In the original design, we used a 6-bit ring counter and one auxiliary state bit T1 to keep track of the two states within each phase. With the new waveforms this is not possible. In the states between active-low pulses (STATE = 0, 2, 4, etc. in Figure 11-29) the phase outputs are all negated, so they can no longer be used to figure out which state should be visited next. Something else is needed to keep track of the state.

There are many different ways to solve this problem. One obvious way to get the waveforms of Figure 11-29 from Program 11-13 would be to keep `IP` defined as is, and combinationally "AND" each 2-tick-wide phase bit with `T1` so `P_L` is asserted only during the second half of the 2-tick cycle, so that for each phase `i`, `P_L[i] = ~(IP[i] & ~T1)`. However, that's a bad idea if these signals are themselves going to be used as clocks, because they may have glitches, as we'll explain next.

Regardless of the module's target realization—ASIC, FPGA, or PLD—the Pi and T1 signals are all outputs from flip-flops clocked by the same master clock CLK. Although these signals change at approximately the same time, their timing is never quite exact. One output may change sooner than another; this is

Program 11-17 Verilog for a modified six-phase timing generator.

```
module Vrtimegen12r ( MCLK, RESET, RUN, RESTART, P_L );
  input MCLK, RESET, RUN, RESTART;
  output reg [1:6] P_L;
  reg [1:12] IP;         // internal active-high phase signals

  always @ (posedge MCLK)
    if (RESET == 1) IP <= 12'b0;
    else if ( (IP == 12'b0) || (RESTART == 1) ) IP <= 12'h800;
    else if (RUN == 1) IP <= {IP[12],IP[1:11]};
    else IP <= IP;

  always @ (*) // Combinational output logic -- just inverters
      for (ii=1; ii<=6; ii=ii+1) P_L[ii] = ~IP[2*ii];
endmodule
```

output timing skew

called *output timing skew*. For example, suppose that on the transition from state 1 to 2 in Figure 11-29, `IP[2]` flip-flop output changes to 1 before `~T1` changes to 0. In this case, a short glitch could appear on the output of any combinational circuit that realizes `P_L[2] = ~(IP[2] & ~T1)`.

To get glitch-free outputs, we should design the circuit so that each phase output is a registered output. One way to do this is to build a 12-bit ring counter and use only alternate outputs to yield the desired waveforms. Program 11-17 is a Verilog module that does the job. The first `always` block is the sequential part, creating the 12 flip-flops with the needed initialization and shifting behaviors. The second `always` block is the combinational part, just "picking off" alternate shift-register outputs and inverting them.

Another approach is to recognize that, since the waveforms cycle through 12 states, we can build a modulo-12 binary counter and decode the states of that counter. A Verilog module using this approach is shown in Program 11-18. The states of the counter correspond to the "STATE" values shown in Figure 11-29. Since the phase outputs must be glitch free, they are decoded one cycle early and then stored in a register, to appear on the output at the correct time. Also note that while `P_L` is set by the `case` statement at the beginning of the `always` block, this

ALTERNATIVE BENEFITS

Notice that compared to previous modules, the new time generators in Programs 11-17 and 11-18 happened to use a different name for the master clock (`MCLK` instead of `CLK`), but the alternative test-bench file structure lets us handle that easily in the instantiation without having to copy over the rest of the test code. The alternative test-bench file structure is also useful if we might later enhance the checking capabilities in the stimulus file (`Vrtimegen_stim.v`), since we no longer have to update the same code in two or more different test benches.

Program 11-18 Counter-based Verilog module for a modified six-phase timing generator.

```
module Vrtimegen12ct1 ( MCLK, RESET, RUN, RESTART, P_L );
  input MCLK, RESET, RUN, RESTART;
  output reg [1:6] P_L;
  reg [3:0] S;     // internal state variables for mod-12 counter

  always @ (posedge MCLK) begin
    if (RUN == 1) begin
      P_L <= 6'b111111;           // default all outputs negated
      case (S)                    // assert the appropriate output
        4'd1:  P_L[1] <= 0;       //  in counts 1,3,5,7,9,11
        4'd3:  P_L[2] <= 0;
        4'd5:  P_L[3] <= 0;       // Note these may be overridden below if
        4'd7:  P_L[4] <= 0;       //   RESET or RESTART is asserted
        4'd9:  P_L[5] <= 0;
        4'd11: P_L[6] <= 0;
      endcase
    end
    if (RESET == 1) begin S <= 0; P_L <= 6'b111111; end  // reset case
    else if (RUN == 1) begin                             // run cases
      if ((RESTART==1)                S <= 0;            // restart from any state
      else if (S == 11)               S <= 0;            // wraparound
      else if (RUN == 1)              S <=  S+1;         // Normal count
    end
    // maintain state if not reset, restarting, or running
  end
endmodule
```

value may be changed in the `if-else` statement that follows if RESET is asserted. Because of this, no output is asserted during reset.

A top-level test bench for the two new modules for modified waveforms is shown in Program 11-19. This test bench instantiates both modules so we can compare their output waveforms with each other, as in Figure 11-30. The first module's waveforms, `P_L1[1:6]`, look alright, with characteristics very similar to what we observed for the original time-generator design:

Program 11-19 Top-level test bench for the modified six-phase timing generators.

```
module Vrtimegen_tba12 ();
  reg CLK, RESET, RUN, RESTART;
  wire [1:6] P_L1, P_L2;

  Vrtimegen12r   U1 (.MCLK(CLK),.RESET(RESET),.RUN(RUN),.RESTART(RESTART),.P_L(P_L1));
  Vrtimegen12ct1 U2 (.MCLK(CLK),.RESET(RESET),.RUN(RUN),.RESTART(RESTART),.P_L(P_L2));

`include "Vrtimegen_stim.v"

endmodule
```

- When RESET is released, operation might begin in phase 1 as required, but we can't tell for sure. Since RUN is still negated, P_L1[1] is "stuck" in the first half of phase 1. After RUN is asserted, though, operation continues to the second half of phase 1 where P_L1[1] is asserted, and then to phase 2.
- Asserting RESTART returns the output sequence back to phase 1.
- RESTART may last longer than one tick, which lengthens the first "half" of phase 1, with P_L1[1] still negated.
- RESTART may occur while P_L1[i] is asserted or negated. In the latter case, the second half of phase i is aborted, and two ticks elapse with no phase signal asserted (more if RESTART lasts longer than one tick).
- Negating RUN lengthens the current phase accordingly. Either the negated or the asserted portion of the phase signal may be lengthened, depending on when RUN is negated.
- After phase 6, the outputs cycle back to phase 1 as expected.

In the second module's waveforms, P_L2[1:6], a problem immediately jumps out: the phase signals are one tick behind the corresponding signals in the first module! But a little thought immediately reveals the reason for this. To eliminate output glitches, we stored the decoded phase outputs in a register, but that delays the output signals by one clock tick. We can eliminate that delay by decoding the register state corresponding to the first half of each phase, so that the registered output signal is asserted one tick later, in the second half of the phase. The required changes are shown in Program 11-20.

Rerunning the test bench to compare the outputs of Vrtimegen12ct2 with those of Vrtimegen12r, we find that they completely match except for one "corner case." Finding and correcting that case is left as Exercise 11.52.

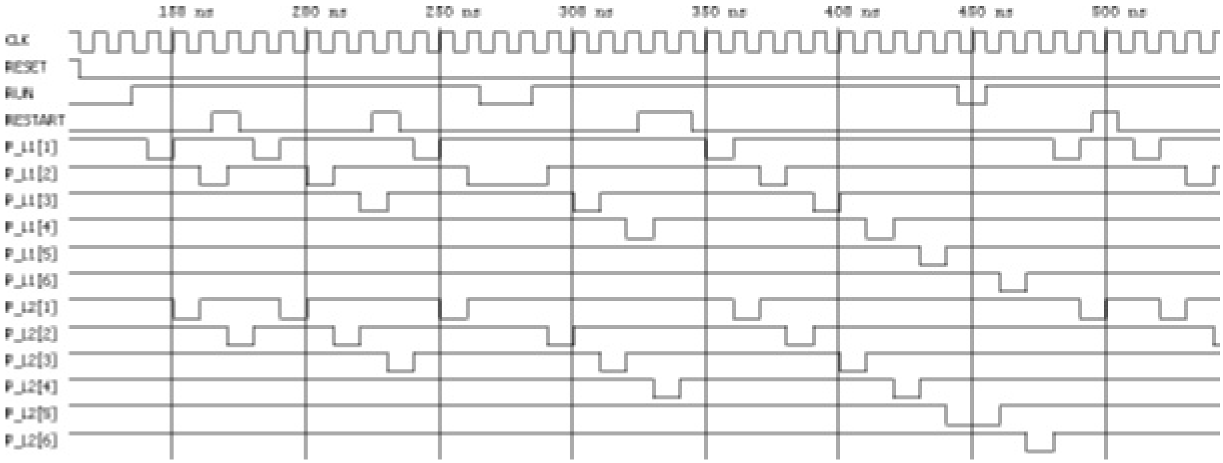


Figure 11-30 Timing waveforms produced by the Vrtimegen_tba12 test bench.

Program 11-20 Corrected Verilog for counter-based timing generator.

```
module Vrtimegen12ct2 ( MCLK, RESET, RUN, RESTART, P_L );
  ...
      case (S)             // Assert the appropriate output
        4'd0: P_L[1] <= 0; //  in tick after counts 0,2,4,6,8,10
        4'd2: P_L[2] <= 0;
        4'd4: P_L[3] <= 0; // These may be overridden below if
        4'd6: P_L[4] <= 0; //   RESET or RESTART is asserted
        4'd8: P_L[5] <= 0;
        4'd10:P_L[6] <= 0;
      endcase
  ...
```

SYNTHESIS RESULTS

The modified six-phase timing generator in Program 11-17 is essentially a 12-bit ring counter with some extra features, so it uses 12 flip-flops and some additional logic for the extra features—8 LUTs when targeted to a Xilinx 7-series FPGA. The corresponding counter-based timing generator in Program 11-20 needs only four flip-flops for its modulo-12 counter, but also six more to create glitch-free phase outputs for a total of 10 flip-flops, plus 6 LUTs for additional logic.

So, the counter-based version is a little smaller, and its worst-case delay path in the targeted FPGA turns out to be a little shorter, so it can run about 10% faster. But the ring-counter version was easier to design, in my opinion. Both approaches are good to have in your bag of tricks.

11.2.8 LFSR Examples

Our last examples involve LFSRs using the design approach in Section 11.2.5. Program 11-21 is a parameterized Verilog module for an LSFR with `N` bits `QX[N-1:0]`. The following parameters are defined:

- `N` is the width of the LFSR in bits, with default of 8.
- `FE[N-1:0]` defines the feedback equation, with a 1 for each output bit that it includes, from left to right for `QX[N-1]` down to `QX[0]`. The default value is based on the n=8 row in Table 11-4 on page 575.
- `SEED` is the initial value for `QX[N-1:0]`. Any nonzero value will generate a maximum-length sequence; an all-0s value would leave `QX` stuck in the all-0s state.

Of course, an LFSR will generate a maximum-length sequence only if the feedback equation is chosen properly, for example from Table 11-4. A test bench that checks the `Vrlfsr` module for this aspect of proper operation is shown in Program 11-22. After resetting the LFSR, it stores the LFSR output, which is the seed value, in a variable `seedQX`. If the LFSR is operating properly, this value should not be repeated until exactly $2^n - 1$ clock ticks later.

Program 11-21 Parameterized Verilog module for an LSFR with 2 to N bits.

```
module Vrlfsr ( CLK, RESET, RUN, QX );
  parameter N = 8;                 // width of the LFSR
  parameter FE = 8'b00011101;      // Define the 1 bits of the feedback equation
  parameter SEED = 8'b00000001;    // Define initial state value (seed)
  input CLK, RESET, RUN;
  output reg [N-1:0] QX;           // state bits for the LFSR
  reg XN;                          // feedback value into QX[N-1]
  integer i;

  always @ (posedge CLK) begin
    if (RESET == 1) QX <= SEED;
    else if (RUN == 1) begin
      XN = 0;
      for (i=0; i<N; i=i+1)        // XOR the bits of QX corresponding to
        XN = XN ^ (FE[i] & QX[i]); //   nonzero terms in the feedback eqn
      QX <= {XN, QX[N-1:1]};       // Shift right, feeding in XN on left
    end
  end
endmodule
```

Program 11-22 Verilog test bench for an N-bit LFSR.

```
module Vrlfsr_tb ();
  parameter N = 8;
  reg CLK, RESET, RUN;
  wire [N-1:0] QX;
  reg [N-1:0] seedQX;   // Capture the starting value of QX
  integer steps;        // Count the number of steps

  Vrlfsr UUT (.CLK(CLK),.RESET(RESET),.RUN(RUN),.QX(QX));

  always begin // create free-running test clock with 10 ns period
    #0.2 CLK = 1; #5; // 5 ns high  (tiny offset for
    CLK = 0; #4.8;    // 5 ns low    waveform readability)
  end

  initial begin
    RESET = 1; RUN = 0; #115
    RESET = 0;   #20
    seedQX = QX; steps = 1;
    RUN = 1;     #10            // Do one step
    while ((QX != seedQX) && (steps < 2**N)) begin // Keep going until
      steps = steps + 1; #10 ;      //  the seed repeats or too many steps
    end
    $display("Executed %0d steps, QX = %b\n",steps,QX);
    if ((QX != seedQX)) $display("Seed %b never repeated!\n",seedQX);
  end
endmodule
```

TEST-BENCH RESULTS

I tested all of the feedback equations in Table 11-4 with the LFSR test bench running on the Xilinx Vivado simulator, and they all passed. The simulation of the 32-bit LFSR ran for about 10 hours on my Windows laptop—over four billion steps! And to successfully complete it, I had to turn off waveform generation to avoid using up all of the laptop's disk storage. They were pretty boring waveforms anyway.

The test bench uses a `while` statement to execute a loop whose duration matches the clock period, counting how many clock periods (steps) have elapsed and comparing `QX` with `seedQX` at each step. When they match, the simulation displays the number of steps taken and stops. The test bench terminates the `while` loop if no match has occurred after 2^n steps. Note that it works properly only if n is less than the integer width used by the simulator.

As mentioned in Section 11.2.5, LFSRs are sometimes used to generate pseudorandom numbers. Many years ago, before "interactive art" was popular, the author's digital-designer friend JC Heater constructed an interactive-art piece incorporating a white-noise-based random number generator. The idea was to generate a sequence of random multi-digit decimal numbers, a new one every few seconds, and to display the sequence on a big, bright, multi-digit seven-segment display. Amazingly, passersby would stop and stare at the display for a long time, waiting for an "interesting" number like their address or birthday to appear somewhere in the dozen or so digits!

We can write a Verilog module that instantiates two existing modules to make a 10-digit pseudorandom version of Heater's artistic creation. As shown in Program 11-23, the `Vrrandomart` module instantiates the `Vrlfsr` module with the parameter values needed to create a 32-bit LFSR, and connects the LFSR's output to a 32-bit to 10-digit binary-to-decimal decoder module `Vrbintodec32`, Program 8-28 on page 433. The module's output, `DIGITS[39:0]`, contains ten 4-bit BCD digits that can be hooked up to seven-segment displays with built-in decoders. Alternatively, the module could be enhanced to incorporate ten seven-segment decoders, allowing its outputs to drive the segments directly (see Exercise 11.72).

Program 11-23 Verilog module for an LFSR-based interactive-art project.

```verilog
module Vrrandomart ( CLK, RESET, RUN, DIGITS );
  input CLK, RESET, RUN;
  output wire [39:0] DIGITS; // Digits for the 7-seg displays
  wire [31:0] RAND;

  Vrlfsr #(.N(32), .FE(32'h00400007), .SEED(32'h12345679)) U2
    (.CLK(CLK), .RESET(RESET), .RUN(RUN), .QX(RAND));
  Vrbintodec32 U3 (.BIN(RAND), .DEC(DIGITS));
endmodule
```

Program 11-24 Verilog test bench for the LFSR-based interactive-art project.

```
`timescale 1ns/100ps
module Vrrandomart_tb ();
  reg CLK, RESET, RUN;
  wire [39:0] DIGITS;     // Digits for the 7-seg displays
  integer i,d;
  reg [39:0] dg;

  Vrrandomart UUT ( .CLK(CLK), .RESET(RESET), .RUN(RUN), .DIGITS(DIGITS) );

  always begin         // create free-running test clock with 50 ns period
    CLK = 0; #25;      // 25 ns high
    CLK = 1; #25;      // 25 ns low
  end

  initial begin
    RESET = 1; RUN = 0;
    #150 RESET = 0; RUN = 1;
    for (i=1; i<=1000; i=i+1) begin
      #50 $write ("Random number: ");
      dg = DIGITS;
      for (d=9; d>=0; d=d-1)
        begin $write ("%1d", dg[39:36]); dg = dg << 4; end
      $write ("\n");
    end
    $stop(1);
  end
endmodule
```

A test bench for the `Vrrandomart` module is shown in Program 11-24. If you use this test bench to perform a post-synthesis functional or timing simulation, it may run slowly enough that you can enjoy staring at its outputs scrolling by on your computer, and wait for your own “significant number” to appear!

*11.3 Iterative versus Sequential Circuits

We introduced iterative circuits in Section 7.4.2. The function of an *n*-module iterative circuit can be performed by a sequential circuit that uses just one copy of the module but requires *n* steps (clock ticks) to obtain the result. This is an excellent example of a space/time trade-off in digital design.

As shown in Figure 11-31, flip-flops are used in the sequential-circuit version to store the cascading outputs at the end of each step; the flip-flop outputs are used as the cascading inputs at the beginning of the next step. The flip-flops must be initialized to the boundary-input values before the first clock tick, and they contain the boundary-output values after the *n*th tick.

*Through out this book, optional sections are marked with an asterisk.

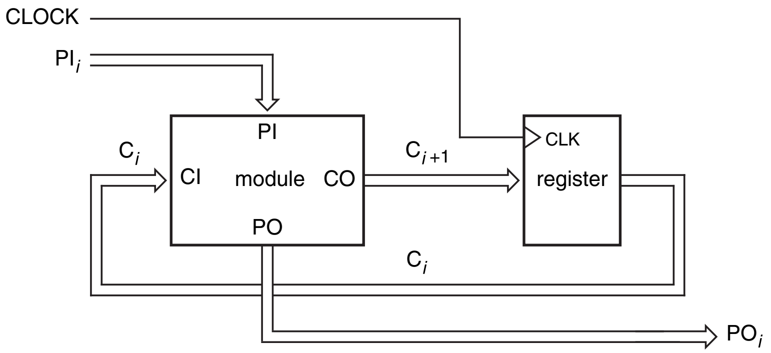


Figure 11-31 General structure of the sequential-circuit version of an iterative circuit.

Since an iterative circuit is a combinational circuit, all of its primary and boundary inputs may be applied simultaneously, and its primary and boundary outputs are all available after a combinational delay. In the sequential-circuit version, the primary inputs must be delivered sequentially, one per clock tick, and the primary outputs are produced with similar timing. Therefore, serial-out shift registers are often used to provide the inputs, and serial-in shift registers are used to collect the outputs. For this reason, the sequential-circuit version of an “iterative widget” is often called a “serial widget.”

serial comparator

For example, Figure 11-32 shows the basic design for a *serial comparator* circuit. The shaded block is identical to the module used in the iterative comparator of Figure 7-24 on page 334. The circuit is drawn in more detail in Figure 11-33, which includes a synchronous reset input. When RESET_L is asserted, the initial value of the cascading flip-flop is forced to 1 at the next clock tick. The initial value of the cascading flip-flop corresponds to the boundary input in the iterative comparator.

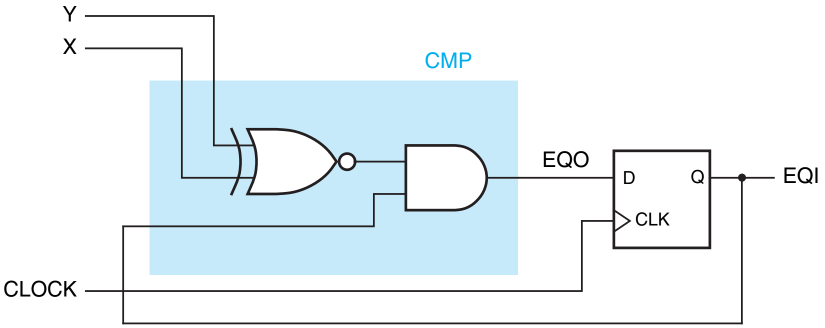


Figure 11-32 Simplified serial comparator circuit.

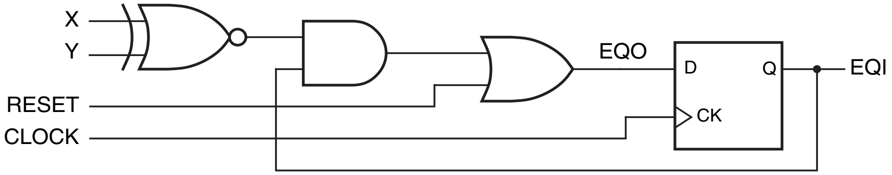


Figure 11-33 Detailed serial comparator circuit.

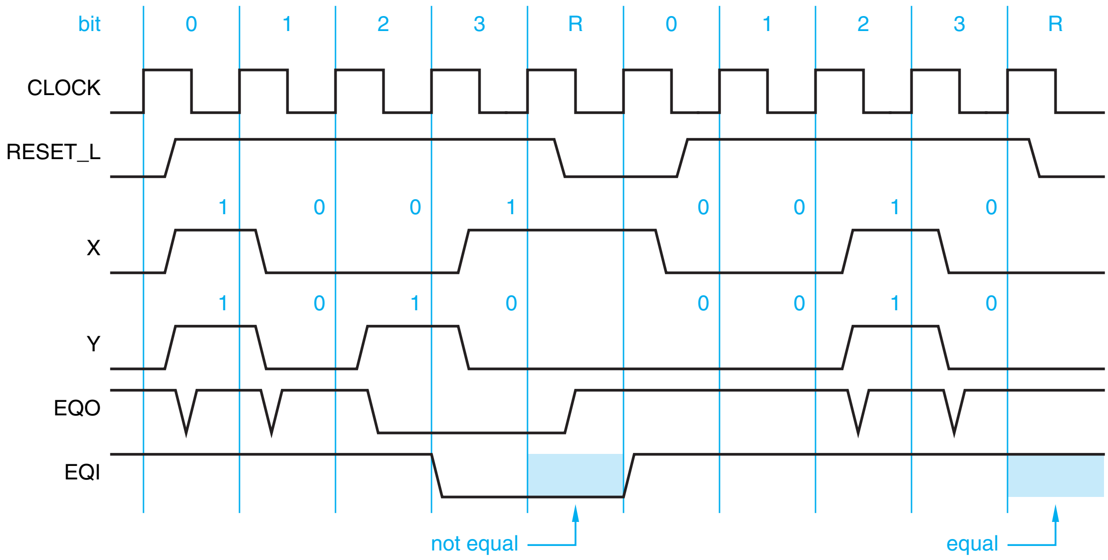


Figure 11-34 Timing diagram for serial comparator circuit.

An n-bit serial comparison requires $n + 1$ clock ticks. RESET_L is asserted at the first clock tick. RESET_L is negated and data bits are applied at the next n ticks. The EQI output gives the comparison result during the clock period after the last tick. A timing diagram for two successive 4-bit comparisons is shown in Figure 11-34. The spikes in the EQO waveform indicate the time when the combinational outputs are settling in response to new X and Y input values.

serial binary adder

A *serial binary adder* circuit for addends of any length can be constructed from a full adder and a D flip-flop, as shown in Figure 11-35. The flip-flop, which stores the carry between successive bits of the addition, is cleared to 0 at reset. Addend bits are presented serially on the A and B inputs, starting with the LSB, and sum bits appear on S in the same order.

Because of the large size and high cost of digital logic circuits in the early days, many computers and calculators used serial adders and other serial versions of iterative circuits to perform arithmetic operations. Even though these arithmetic circuits aren't used much today, they are an instructive reminder of

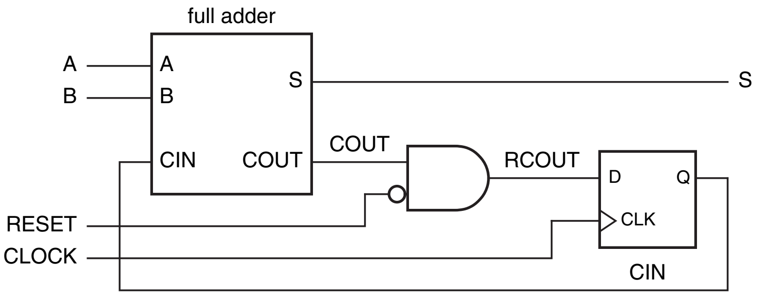


Figure 11-35 Serial binary adder circuit.

the space/time trade-offs that are possible in digital design. Iterative circuits and corresponding sequential circuits can also be considered for applications where each basic computation is performed on a unit larger than a bit, like a nibble or byte (for example, see Exercises 11.75–11.76).

References

Logic hazards have been known since at least the 1950s, and function hazards were discussed by Edward J. McCluskey in *Logic Design Principles* (Prentice Hall, 1986). Galois fields were invented centuries ago, and their applications to error-correcting codes, as well as to the LFSR counters of this chapter, are described in introductory books on coding theory, including *Error-Control Techniques for Digital Communication* by A. M. Michelson and A. H. Levesque (Wiley-Interscience, 1985).

Some PLDs and CPLDs contain XOR structures that allow large counters to be designed without a large number of product terms. This requires a somewhat deeper understanding of counter excitation equations, as described in Section 10.5 of the second edition of this book. Fortunately, the synthesis tools for these devices should know how to figure this out for you.

The logic elements in some FPGAs, including the Xilinx 7 series, can be configured to create large serial-in, serial-out shift registers using far fewer resources than would be consumed using the device's fully programmable flip-flops and interconnect. Recall that a 7-series LUT has 64 bits of memory that are normally initialized to specify a combinational logic function; each memory bit is essentially a 1-bit latch. Using a special "SRL" configuration, a single LUT's 64 latches may be hooked up in series as 32 edge-triggered D flip-flops (each one consuming two latches), forming a serial-in, serial-out shift register up to 32 bits in length, independent of the handful of fully programmable flip-flops in the same slice. See Xilinx publication UG474, *7 Series FPGA Configurable Logic Block*, for more information and options.

Drill Problems

11.1 Design a 4-bit ripple *down* counter using four T flip-flops and no other components.

11.2 Design a 4-bit ripple *up* counter using four D flip-flops of the type shown in Figure 10-12 and no other components.

11.3 Assume that the propagation delay from clock to output of a D flip-flop is 5 ns. What is the maximum propagation delay from clock to output for the 4-bit ripple counter of Drill 11.2?

11.4 Describe the connections needed for a CNTR4U to be set up as a modulo-N counter that counts from $16 - N$ to 15, using *no* additional gates.

11.5 Consider the divide-by-16 counter timing diagram in Figure 11-7. Are there any transitions in which the RCO output could have a glitch?

11.6 In the discussion of Figure 11-8, the text states that in the decoder, "the path from A1 to Y1_L may be faster than the path from A0 to Y1_L." Explain how this can be true. *Hint*: Consider Figure 6-17.

11.7 What is the counting sequence of the counter circuit in Figure X11.7? Ignoring initial behavior right after reset, is there a way to eliminate the AND gate and still get a counter with the same modulus?

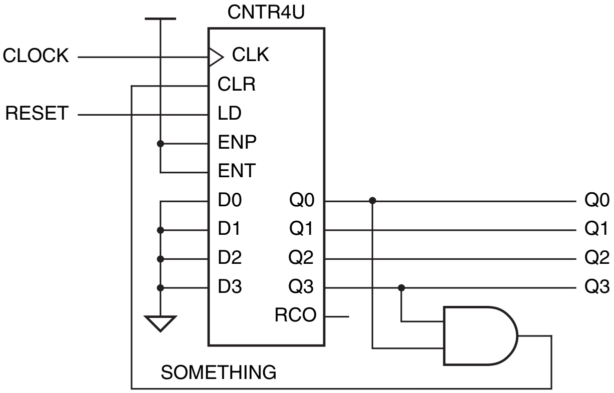


Figure X11.7

11.8 What is the counting sequence of the counter circuit in Figure X11.7 if the bottom AND-gate input is connected to Q2 instead of Q0?

11.9 A CNTR4U counter is hooked up with inputs ENP, ENT, and D3 always HIGH, inputs D0–D2 always LOW, input LD = Q0 · Q2, and input CLR = Q1 · Q3. The CLK input is hooked up to a free-running clock signal. Draw a logic diagram for this circuit. Assuming that the counter starts in state 0000, write the output sequence on Q3–Q0 for the next 15 clock ticks.

11.10 Multiple instances of the CNTR4U counter may be cascaded to create a binary counter with an arbitrarily large number of bits, 4*n*, using no additional logic. Determine and describe the cascading arrangement: how should the RCO outputs be hooked up, and which control inputs should be hooked up to the same inputs to control the overall 4*n*-bit counter?

11.11 Determine the widths of the glitches shown in Figure 11-9 on the Y2_L output of the 3-to-8 decoder, assuming that the decoder is internally structured as shown in Figure 6-17 on page 255, and that each internal gate has a delay of 10 ns.

11.12 A certain design with a 100-MHz clock signal CLK also requires a 10-MHz signal CLK10 with a 50% duty cycle. Identify one or more outputs of existing Verilog modules in Section 11.1.5 that have the required characteristics when the module is clocked by CLK.

11.13 Write a Verilog test bench that instantiates the two versions of a 3-bit counter with decoding in Programs 11-6 and 11-7, and runs them for a few dozen clock ticks. Examine the resulting output waveforms to confirm that the outputs of the second version are the same as the first's except delayed by one clock tick.

11.14 Repeat Drill 11.13 with an error in Program 11-7: changing the last non-blocking assignment to S_L to a blocking assignment. How does this change the module's output and why? Then change *all four* assignments to blocking, run the test bench again, and explain the output behavior. What rule have you violated?

CNTR4UD

11.15 A CNTR4UD is a 4-bit up/down counter with the same inputs and outputs as the CNTR4U binary counter, plus an UP/DN input that controls whether it counts up (UP/DN=1) or down. The function of the RCO output also depends on UP/DN; it is asserted in state 1111 when counting up, and 0000 when counting down. What is the counting sequence of the circuit shown in Figure X11.15?

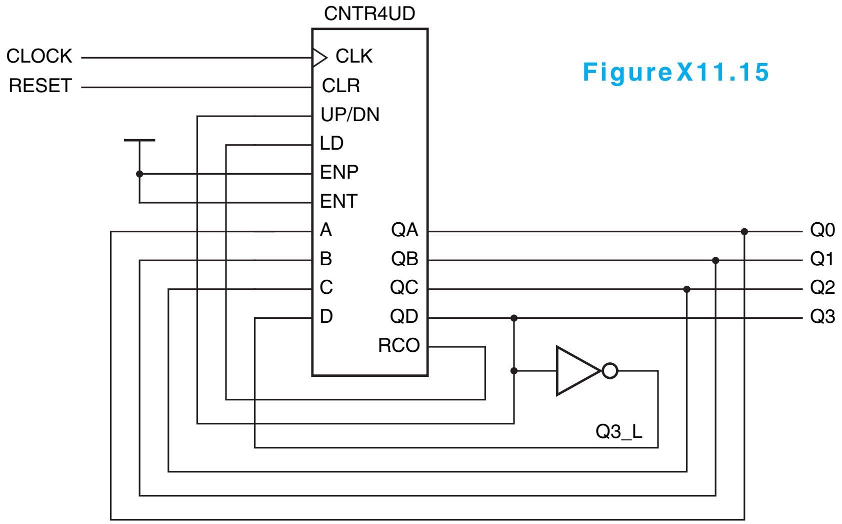


Figure X11.15

11.16 The shift-register module in Program 11-8 uses part-select and concatenation to specify the left-shifted value, while the text offered another way to do it using a Verilog built-in shift operator. Corresponding to these two approaches, write two different Verilog expressions for specifying a right-shifted value.

11.17 Starting with state 00001, write the sequence of the first ten states for a 5-bit LFSR counter designed according to Figure 11-25 and Table 11-4.

11.18 As we explained in the box on page 195, in some places you can get away with using logical negation (`!`) when technically, bitwise negation (`~`) should be used. But the continuous-assignment statement in Program 11-13 is not one of them. Determine what happens if you mistakenly used logical negation.

11.19 A digital designer built the self-synchronizing ring counter of Figure 11-20 after substituting a 3-input NAND gate for the NOR gate. What was the counting sequence of the resulting circuit? Was the counter still self-synchronizing?

11.20 Write a parameterized Verilog module `Vrringn` for an *n*-bit self-synchronizing ring counter with synchronous `INIT` and `CNTEN` inputs, and outputs `Q_L[n-1:0]`, with default *n*=6. It should have a single circulating 0, starting at `Q_L[0]` at when `INIT` is asserted and shifting left at each clock tick when `CNTEN` is 1.

11.21 Write a Verilog module for a 4-bit ripple counter similar to Figure 11-2 plus a reset input. Not that you'd ever want to build such a nasty thing, but target your module to your favorite FPGA and comment on the synthesis and implementation (place and route) results.

Exercises

11.22 What limits the maximum counting speed of a ripple counter, if you don't insist on being able to read the counter value at all times? At what times can you read it?

11.23 Write a formula for the maximum clock frequency of the synchronous serial binary counter circuit in Figure 11-3. In your formula, let t_{TQ} denote the propagation delay from T to Q in a T flip-flop, t_{setup} the setup time of the EN input to the rising edge of T, and t_{AND} the delay of an AND gate.

11.24 Repeat Exercise 11.23 for the synchronous parallel binary counter circuit shown in Figure 11-4, and compare results.

11.25 Repeat Exercise 11.23 for an n-bit synchronous serial binary counter.

11.26 Repeat Exercise 11.23 for an n-bit synchronous parallel binary counter. Beyond what value of n is your formula no longer valid?

11.27 Using a CNTR4U 4-bit binary counter, design a modulo-11 counter circuit with the counting sequence 4, 5, 6, …, 13, 14, 4, 5, 6, ….

11.28 Find the datasheet on the Web and look up the internal logic diagram for a 74x162 synchronous decade counter, and write its state table in the style of Table 11-1, including its counting behavior in the normally unused states 10–15.

11.29 Devise a cascading scheme for the CNTR4U, analogous to the synchronous parallel counter structure of Figure 11-4, such that the maximum counting speed is the same for any counter with up to $4(n+1)$ bits (using $n+1$ CNTR4Us), where n is the maximum number of inputs of a high-speed AND gate. Identify and name the relevant timing parameters of the CNTR4U and write a formula for the maximum counting frequency based on those parameters and the propagation delay of the AND gates.

11.30 Design a modulo-129 counter using two CNTR4Us and no additional gates.

11.31 Based on the `Vrcntr4u` module in Program 11-1, write a new module `Vrcntr32` that uses a parameter `WID` to set the counter width, with a default of 32 bits. Using your favorite synthesis tool, target the new module to your favorite FPGA and determine how many LUTs and flip-flops it requires. Repeat with `WID=64`. *Hint*: With Xilinx Vivado 2016.3 tools and a 7-series FPGA, the number of LUTs was at least 25% greater than the counter width.

11.32 Write a Verilog test bench `Vrcntr32_tb` based on Program 11-5 to test the module in Exercise 11.31. Based on the width of the instantiated counter, be sure that the new test bench exercises it for a several ticks before and after it rolls over from all 1s to all 0s, without having to run (or examine!) a zillion cycles.

11.33 Write a Verilog module for an 8-bit modulo-N counter with clear and load inputs, where the value of N is specified by a constant `N` in the module.

11.34 Repeat the preceding exercise, but let N be determined by a value that is loaded from the data inputs into a second 8-bit register when a control signal "`MLOAD`" is asserted. Use comments to document what happens when more than one control input is asserted; your design should exhibit reasonable behavior in these cases.

11.35 Design a clocked synchronous circuit with four inputs, N3, N2, N1, and N0, that represent an integer N in the range 0–15. The circuit has a single output Z that is asserted for exactly N clock ticks during any 16-tick interval (assuming that N is held constant during the interval of observation). (*Hints:* Use combinational logic with a CNTR4U set up as a free-running divide-by-16 counter. The ticks in which Z is asserted should be spaced as evenly as possible, that is, every second tick when $N = 8$, every fourth when $N = 4$, and so on.)

binary rate multiplier

11.36 Modify the circuit of Exercise 11.35 so that Z produces *N transitions* in each 16-tick interval. The resulting circuit is called a *binary rate multiplier* and was once sold as a TTL MSI part, the 7497. (*Hint:* Gate the clock with the level output of the previous circuit.)

11.37 Repeat Exercises 11.35 and 11.36 using an 8-bit input N7–N0, and model the design using a behavioral Verilog module for an available programmable device.

11.38 Design a modulo-16 counter, using one CNTR4UD (see Drill 11.15) and at most one discrete logic gate, with the following counting sequence: 7, 6, 5, 4, 3, 2, 1, 0, 8, 9, 10, 11, 12, 13, 14, 15, 7, ….

11.39 Write a Verilog module for an n-bit counter that realizes a counting sequence similar to the one in Exercise 11.38. Write your code so that the size of the counter can be changed by changing the value of a single constant `N`.

11.40 Write a Verilog module for a binary up/down counter intended to be used in the elevator controller in a 20-story building. The counter should have enable and up/down control inputs. It should stick at state 1 when counting down, stick at state 20 when counting up, and skip state 13 in either mode. Write a test bench that exercises your module for a comprehensive set of inputs.

11.41 As defined in Section 11.1, a counter is any sequential circuit whose state diagram is a single cycle. Write a Verilog module `Vr4bitanyctr` with two inputs, CLK and RESET, and a 4-bit output Q[3:0]. The module should specify the desired counting sequence in a list that can be easily modified with a one-line change to obtain *any* desired 16-state, 4-bit counting sequence. At reset, your module should go to the first state in the list. Write a test bench `Vr4bitanyctr_tb` that runs your counter for a few dozen clock ticks so you can observe its counting sequence. Test your module first with a sequential counting sequence and then with a jumbled one. *Hint*: Investigate and use Verilog's capability to initialize the values of a `reg` array.

11.42 Write a parameterized Verilog module `Vrshrgnu` based on Program 11-9 for an n-bit universal shift register, with a default of $n = 8$.

11.43 Write a parameterized Verilog test bench `Vrshrgnu_tb` based on Program 11-10 to test the `Vrshrgnu` module in Exercise 11.42. Use the Verilog `$random` function to load random data into the shift register, and test 1000 random data values per function regardless of the value of n.

11.44 Suppose you are asked to design a serial computer, one that moves and processes data one bit at a time. The first decision you must make is which bit to transmit and process first, the LSB or the MSB. Which would you choose, and why?

11.45 Write a Verilog module for a 4-bit self-synchronizing ring counter that does *not* have a reset input. Write a test bench that initializes the counter to an unknown state (all `x`'s) and then exercises it for a dozen clock ticks. Does the counter ever self-synchronize in simulation? Is there a way in Verilog to simulate what happens in the real circuit? Explain, and comment on the value, if any, of eliminating the additional circuitry required for an actual reset input.

11.46 Write a Verilog test bench that checks whether the self-synchronizing ring counter of Program 11-12 always returns to a valid state from any of its possible 256 states within *n* clock ticks. Also, use the test bench to determine (or confirm if you think you already know it) the value of *n*. Do not modify the original module in any way. However, you should test your test bench's ability to find errors by changing "`[6:0]`" to "`[7:1]`" in the original module, an error that actually occurred in a draft of this chapter. *Hint*: "Use the _____, Luke."

11.47 Write a Verilog module for an 8-bit self-correcting ring counter whose states are 11111110, 11111101, …, 01111111. Include reset and enable inputs, where the counter goes to the initial state when reset is asserted, and counts only if the enable input is asserted.

11.48 Write a Verilog module `Vr8bitringsed` for an 8-bit self-error-detecting ring counter that has a single recirculating 1. The counter should be designed so that if it ever detects an error state—with no 1s or more than one 1 in its output—it goes to the all-0s state and stays there and asserts an ERROR output until it is reset. When your module is targeted to your favorite FPGA, how do its resource requirements compare with those of a "plain" 8-bit self-correcting ring counter?

11.49 Write a test bench that checks the operation of the `Vr8bitringsed` module in Exercise 11.48, ensuring that it goes to the "error" state if it somehow reaches any of the 248 possible invalid states. Can you find a way to write the test bench or otherwise use the tools so you do not have to modify the UUT design in any way (like providing new inputs in the UUT to load an invalid starting state)? If not, what's the best you can do?

11.50 Write a Verilog module for a 12-state self-correcting Johnson counter. Write a test bench that stimulates the counter, and check the resulting waveforms for correctness. Synthesize the module, targeting to your favorite FPGA, and determine how many resources (flip-flops and LUTs) it requires.

11.51 Modify the module in Exercise 11.50 to provide a new input `TSTLD` which, when asserted at the clock edge, loads the counter's flip-flops with an arbitrary value taken from a new set of data inputs. Write a test bench which, using `TSTLD` and the new data inputs, determines whether the counter eventually returns to the normal Johnson counting sequence from all possible starting states. Your test bench should display each possible starting state and indicate whether or not the counter returns to the Johnson counting sequence from that state, and total the number of starting states for which it fails. Test the test bench by running it with a non-self-correcting version of the Johnson counter.

11.52 Update the test bench in Program 11-19 to instantiate timing generator modules `Vrtimegen12r` and `Vrtimegen12ct2`. Run it and show that their outputs mostly match, but find at least one case where they don't. Make a corrected module `Vrtimegen12ct3` so the outputs do match completely, re-running the test bench to prove it.

11.53 Write a new stimulus file Vrtimegen_stim2.v for use with the test bench in Program 11-19, where the new version compares the outputs of the two UUTs at each time step and displays an informative message when they are different. Test the new version by using Vrtimegen12r and Vrtimegen12ct2 as the UUTs.

11.54 Starting with your solution to Exercise 11.53, update the stimulus file to generate a comprehensive set of stimulus inputs algorithmically, varying the lengths of time that RUN and RESTART are asserted and negated and overlapped. Use the new stimulus file to compare the output of Vrtimegen12r against the outputs of both Vrtimegen12ct2 and Vrtimegen12ct3 (your solution to Exercise 11.52), and determine whether there are any more corner cases to be discovered.

11.55 Modify the Vrtimegen6 timing-generator module in Program 11-13 so that if RUN is negated, it does not stop until the currently asserted phase signal has been negated (i.e., phase signals are never shortened or lengthened). Check your modified module with the Vrtimegen_tb test bench, adding additional tests if necessary to prove that you've got it right.

11.56 Modify the Vrtimegen6 timing-generator module in Program 11-13 so that the phases are always at least two clock ticks long, even if RESTART is asserted at the beginning of a phase. However, RESET should still take effect immediately. Check your modified module with the Vrtimegen_tb test bench, adding additional tests if necessary to prove that you've got it right.

11.57 Suppose that the Vrtimegen12r or the Vrtimegen12ct2 timing-generator module is used to control a dynamic memory system, such that all six phases must be completed to read or write the memory. If the timing generator is reset or restarted during a write operation without completing all six phases, the memory contents may be corrupted. Modify the module to avoid this problem.

11.58 Design two different 2-bit, 4-state counters, where each design uses two edge-triggered D flip-flops of the kind shown in Figure 10-12(c) and no other gates.

11.59 Design a 4-bit Johnson counter and decoding for all eight states using four D flip-flops (Figure 10-12(c)) and eight gates. Your counter need not be self-correcting.

11.60 Write a Verilog module for an 8-bit Johnson counter that starts in the all-0s state. Include reset and enable inputs, where the counter goes to the initial state when reset is asserted, and counts only if the enable input is asserted.

11.61 Prove that an even number of shift-register outputs must be connected to the odd-parity circuit in an n-bit LFSR counter if it generates a maximum-length sequence. (Note that this is a necessary but not a sufficient requirement. Also, although Table 11-4 is consistent with what you're supposed to prove, simply quoting the table is not a proof!)

11.62 Prove that X0 must appear on the righthand side of any LFSR feedback equation that generates a maximum-length sequence. (*Note:* Assume the LFSR bit ordering and shift direction are as given in the text; that is, the LFSR counter shifts right, toward the X0 stage.)

11.63 Suppose that an n-bit LFSR counter is designed according to Figure 11-25 and Table 11-4. Prove that if the odd-parity circuit is changed to an even-parity circuit, the resulting circuit is a counter that visits $2^n - 1$ states, including all of the states except 11…11.

11.64 Find a feedback equation for a 4-bit LFSR counter, other than the one given in Table 11-4, that produces a maximum-length sequence.

11.65 Given an n-bit LFSR counter that generates a maximum-length sequence ($2^n - 1$ states), prove that connecting an extra XOR gate and an $n - 1$ input NOR gate as shown in Figure 11-26 produces a counter with 2^n states.

11.66 Prove that a sequence of 2^n states is still obtained if a NAND gate is substituted for a NOR above, but that the state sequence is different.

11.67 In the Verilog LFSR module of Program 11-21, find a way to eliminate the `for` loop by using a Verilog reduction operator. Check the correctness of your new module using the test bench in Program 11-22. Synthesize both modules and target to your favorite programmable device. Do they have identical synthesis results, or if you can't tell, identical resource requirements?

11.68 Correct the test bench in Program 11-22 to make a new test bench `Vrlfsr_tbc` that works correctly even if the LFSR width is the same as or wider than integers. Find a way to test it that doesn't take all night or longer.

11.69 Try to guess a polynomial that yields a maximum-length sequence for a 10-bit LFSR counter. Use the test bench in Exercise 11-22 to determine whether each of your guesses works.

11.70 After completing Exercise 11.69, modify your test bench to discover and display *all* 10-bit polynomials that yield maximum-length sequences. How many are there?

11.71 Design an iterative circuit for checking the parity of a 16-bit data word with a single even-parity bit. Does the order of bit transmission matter?

11.72 Instantiate the `Vrrandomart` module in Program 11-23 in a hierarchical design with outputs that are suitable for directly driving the segment inputs of ten seven-segment displays; use the `Vr7segdec` module of Program 6-12.

11.73 Designer JC Heater told the author that his original random-number interactive art had one other feature that captivated some viewers. Random numbers with one or more leading zeroes didn't occur very frequently—ten times less likely for each additional zero. To highlight these events, the seven-segment decoding logic included "leading-zero blanking" (leading zeroes are not displayed). Design an enhanced module `Vrrandomart_bl` that incorporates this feature and whose outputs can directly drive the segment inputs of ten seven-segment displays. You can incorporate the `Vr7segdec` module of Program 6-12.

11.74 Design an iterative circuit with one input B_i per stage and two boundary outputs X and Y such that X = 1 if at least two B_i inputs are 1 and Y = 1 if at least two *consecutive* B_i inputs are 1.

11.75 Sketch the structure of an iterative circuit that converts a 32-bit binary number into ten BCD digits, based on the `Vrbintodec32` combinational Verilog module in Program 8-28. Indicate the circuit's boundary inputs and outputs, primary inputs and outputs, and cascading inputs and outputs.

11.76 Write a Verilog module `Vrbintodec32_seq` for a clocked sequential circuit that performs the same conversion as the iterative circuit in Exercise 11.75, in ten clock ticks using just one instance of the `Vrdiv10_so` module. Besides a clock input CLK, your module should have a 32-bit data input DIN, a LOAD control input that is asserted for one tick to start a conversion, and a 4-bit DIG output. The BCD digits corresponding to DIN should appear on DIG, least significant digit first, during the next 10 clock periods after LOAD is asserted. Your circuit should use no more than 40 flip-flops.

11.77 Write a self-checking test bench for the module in Program 11.76.

11.78 Write a structural Verilog module `Vrrev8ser` for a sequential circuit with inputs CLK, RESET, and SERIN, and output SEROUT, with the following behavior that repeats every eight clock ticks after reset. The circuit receives eight bits of input, one bit per clock tick, on SERIN. After the eighth tick, it outputs the bits one at a time, on SEROUT in *reverse order*. During this time, it also gathers the next eight bits which will also be put out in reverse order immediately after being received. During the first eight clock periods after reset, SEROUT should be 0.

Now, here's the challenge. Your design should instantiate just one `Vrcntr4u` and two `Vrshrg4u` components and it should not contain any other registers, though it may contain a small amount of combinational logic, limited to one continuous assignment statement. Note that the `Vrcntr4u` and `Vrshrg4u` components have more functionality than is needed in this application, but their unused logic will be pruned away in synthesis. Write a test bench that checks for proper operation of your module for a sequence of 1000 random 8-bit inputs (8000 clock ticks).

```
module ButSM (
  input CLK, A, B, C, D,
  output reg Q1, Q2
 );
always @ (posedge CLK)
 begin
  if ((A==B)&&(C!=D))
   {Q1,Q2}<={1'b1,~Q1};
  else Q1 <= 0;
  if ((C==D)&&(A!=B))
   {Q1,Q2}<={~Q2,1'b1};
  else Q2 <= 0;
 end
endmodule
```

chapter 12

State Machines in Verilog

There are many possible coding styles for creating state machines in Verilog, including using no consistent style at all. Some styles, and especially an inconsistent style, are guaranteed to get you in trouble. Without the discipline of a consistent coding style, it is fairly easy to write syntactically correct Verilog code where the simulator's operation, the synthesized hardware's operation, and what you think the machine should be doing are all different!

The basic coding style that we'll introduce in the first section of this chapter has been used for years by digital-design professionals to give a minimum of errors. This style also has the advantage of separating the major sections of a state machine's operation and structure, making the design easy to understand and maintain. We'll also show some variations and simplifications of the style. What you eventually end up using in your own designs may well depend on the standards that are dictated by your own design organization or team.

Test benches are important companions to state-machine designs, and there are a few different ways to construct them, as discussed in the second section of this chapter. Following that, the rest of the chapter will be devoted to numerous examples of state machines and test benches, interspersed with a few new concepts including "don't-care" state encodings and decomposition of state machines.

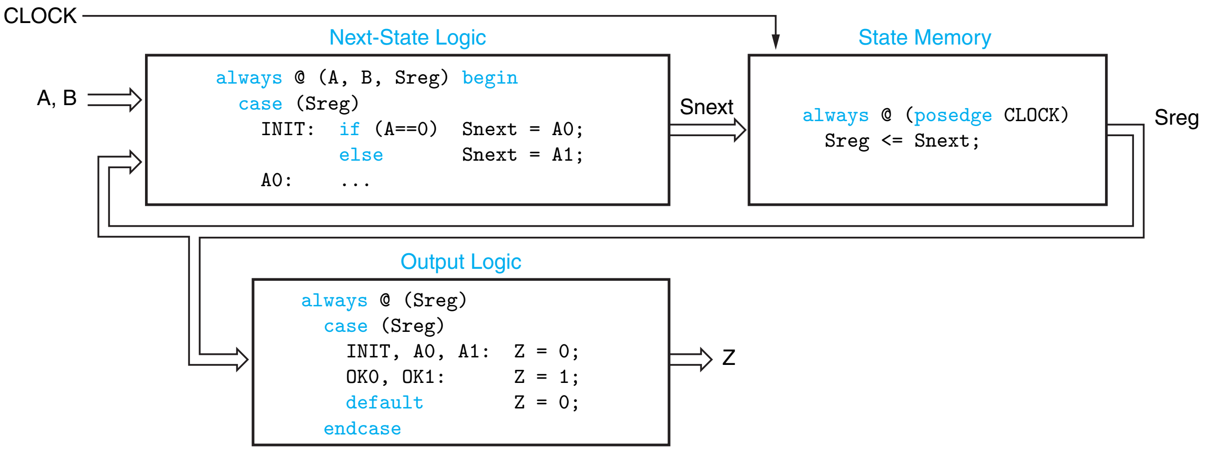
12.1 Verilog State-Machine Coding Styles

12.1.1 Basic Coding Style

Our basic Verilog state-machine coding style matches the general structure of Mealy and Moore state machines that we presented in Figures 9-4 and 9-5 on page 444. The code can be divided into three parts:

- *State memory*. This may be specified in behavioral form using an `always` block that is sensitive to a clock-signal edge, as we showed in Section 10.3.2 and Program 10-6 on page 516 for edge-triggered D flip-flops; or it can use a structural style with explicit flip-flop instantiations, as we showed in Section 10.3.1.
- *Next-state (excitation) logic*. This is written as a combinational `always` block whose sensitivity list includes the machine's current state and inputs. This block usually contains a `case` statement that enumerates all possible values of the current state.
- *Output logic*. This is another combinational `always` block that is sensitive to the current state and inputs. It may or may not include a `case` statement, depending on the complexity of the output function.

The detailed coding within each section may vary. When there is a tight coupling of next-state and output-logic specifications, it may be desirable to combine the next-state and output logic into a single combinational `always` block, and indeed, into a single `case` statement. When pipelined outputs are used, the output memory could be specified along with the state memory, or a separate `always` block or structural code could be used. You've already seen all of the Verilog features needed to support state-machine design, but we'll refresh your memory as we go along.



```
always @ (A, B, Sreg) begin
  case (Sreg)
    INIT:  if (A==0)  Snext = A0;
           else       Snext = A1;
    A0:    ...
```

```
always @ (posedge CLOCK)
  Sreg <= Snext;
```

```
always @ (Sreg)
  case (Sreg)
    INIT, A0, A1:  Z = 0;
    OK0, OK1:      Z = 1;
    default        Z = 0;
  endcase
```

Figure 12-1 Moore state-machine structure implied by Verilog coding style.

Figure 12-1 shows the relationship between our coding style and state-machine structure for the example state-machine in the next subsection. Two other very important aspects of any coding style are `parameter` statements for defining state encodings, and reset capability for the state memory, as we'll see.

12.1.2 A Verilog State-Machine Example

In Section 9.3, we illustrated the state-table design process using the simple design problem below:

> Design a clocked synchronous state machine with two inputs, A and B, and a single output Z that is 1 if:
>
> - A had the same value at each of the two previous clock ticks, *or*
> - B has been 1 since the last time that the first condition was true.
>
> Otherwise, the output should be 0.

We constructed a state/output table for this machine in Section 9.3, and we've repeated it here in Table 12-1. Also, we showed a corresponding Verilog module which is repeated in Program 12-1.

As usual, the Verilog module declaration specifies its inputs and outputs—`CLOCK`, `A`, `B`, and `Z` in this example. Next, it declares variables `Sreg` and `Snext` for the machine's current and next states, respectively. A `parameter` statement specifies the state assignment, defining a unique constant for each of the machines's five states. Here we just assigned five 3-bit values in sequence, but other assignments could be made by changing the definitions in the `parameter` statement. State encodings with more than three bits would also require a corresponding change in the width of `Sreg` and `Snext`.

The first `always` block in the module creates the state memory. This block executes on the rising edge of `CLOCK` and loads the next state `Snext` into the state register `Sreg`. A synthesizer infers positive-edge-triggered D flip-flops for `Sreg`.

The second `always` block specifies the next-state logic using a `case` statement. It assigns a value to `Snext` in six cases, corresponding to the five explicitly defined states and a default for other, undefined states. For robustness, the default case sends the machine back to the `INIT` state. Each "`if`" statement has a

	A B				
S	***00***	***01***	***11***	***10***	***Z***
INIT	A0	A0	A1	A1	0
A0	OK0	OK0	A1	A1	0
A1	A0	A0	OK1	OK1	0
OK0	OK0	OK0	OK1	A1	1
OK1	A0	OK0	OK1	OK1	1
	S*				

Table 12-1
State and output table for the example state machine.

Program 12-1 Verilog module for state-machine example.

```
module VrSMex( CLOCK, A, B, Z );
  input CLOCK, A, B;
  output reg Z;
  reg [2:0] Sreg, Snext;         // State register and next state
  parameter [2:0] INIT = 3'b000, // Define the states
                  A0   = 3'b001,
                  A1   = 3'b010,
                  OK0  = 3'b011,
                  OK1  = 3'b100;

  always @ (posedge CLOCK) // Create the state memory
    Sreg <= Snext;

  always @ (A, B, Sreg) begin  // Next-state logic
    case (Sreg)
      INIT:   if (A==0)                  Snext = A0;
              else                       Snext = A1;
      A0:     if (A==0)                  Snext = OK0;
              else                       Snext = A1;
      A1:     if (A==0)                  Snext = A0;
              else                       Snext = OK1;
      OK0:    if (A==0)                  Snext = OK0;
              else if ((A==1) && (B==0)) Snext = A1;
              else                       Snext = OK1;
      OK1:    if ((A==0) && (B==0))      Snext = A0;
              else if ((A==0) && (B==1)) Snext = OK0;
              else                       Snext = OK1;
      default Snext = INIT;
    endcase
  end

  always @ (Sreg)        // Output logic
    case (Sreg)
      INIT, A0, A1: Z = 0;
      OK0, OK1:     Z = 1;
      default       Z = 0;
    endcase
endmodule
```

final “else” to ensure that a value is always assigned to Snext. If no value were assigned to Snext for any state/input combinations, the Verilog compiler would unnecessarily infer a latch to hold the value of Snext for those combinations.

The third and final always block in Program 12-1 handles the machine’s single Moore output, Z, which is set to a value as a function of the state only. It would be easy to define Mealy outputs here as well, by making Z be a function of the inputs as well as the state in each enumerated case. If this is done, then the inputs would also be added to the sensitivity list of the always block, either explicitly or by just using the “*” shorthand.

NEXT-STATE CODING-STYLE VARIATIONS

As we recommended when we introduced case statements back in Section 5.9.7, the best Verilog coding practices use *full case* statements only—ones where all possible choices are covered, either explicitly or by the use of a default case. The latter is used in next-state case statement in Program 12-1.

Nevertheless, some designers would precede this case statement with one more line of code, “Snext = INIT”. This establishes a “default” next state for the machine if the case statement fails to cover all state/input combinations. While the default case at the end handles unused states, it does nothing for any uncovered input combinations that may exist in the other cases. There are no such uncovered input combinations for any state in Program 12-1, because each case has a single if statement with a final else clause that picks up any input combinations that weren’t explicitly tested. But nothing except the designer’s care in coding makes that true.

Another useful variation occurs in machines where most transitions stay in the current state. Then, the case statement can be preceded by a line of code that maintains the current state as default, “Snext = Sreg”.

Yet another variation is the default “Snext = 3'bx” (or whatever width the state register has). In simulation, this ensures that if the next-state logic ever sees an unspecified state/input combination, the state will be undefined (x’s), which is easy to detect in simulation.

Finally, you may have noticed that the begin-end block surrounding the case statement in the next-state always block is not really necessary syntactically, since the case is the single procedural statement needed to follow always. However, the begin-end would still be needed if we wanted to add any other statements to the always block, like a default assignment to Snext.

If the output logic is uncomplicated, especially in Moore machines like the present example, it may be more convenient to specify it using a continuous assignment statement. In the present example, the final always block could be replaced by just one continuous assignment:

```
  assign Z = (Sreg==OK0) || (Sreg==OK1);
```

Of course, this would also necessitate declaring Z as type wire instead of reg.

12.1.3 Combined State Memory and Next-State Logic

The structure of the three always blocks in Program 12-1 makes it very clear, to the designer at least, that we are creating a state machine with the three blocks shown in Figure 12-1—state memory, next-state (excitation) logic, and output logic. This structure also makes it very easy to instantiate explicit flip-flop or register components for the state memory, instead of having the compiler infer them from a behavioral description.

On the other hand, depending on the design environment, when the Verilog compiler processes the next-state logic, it doesn’t necessarily “know” that this is next-state logic for a state machine. As far as it’s concerned, this is just general

Program 12-2 Combined state memory and next-state logic in `VrSMexc`.

```
always @ (posedge CLOCK) // State memory and next-state logic
  case (Sreg)
    INIT:   if (A==0)  Sreg <= A0;
            else       Sreg <= A1;
    A0:     if (A==0)  Sreg <= OK0;
            else       Sreg <= A1;
    A1:     if (A==0)  Sreg <= A0;
            else       Sreg <= OK1;
    OK0:    if (A==0)  Sreg <= OK0;
            else if ((A==1) && (B==0)) Sreg <= A1;
            else                       Sreg <= OK1;
    OK1:    if ((A==0) && (B==0))      Sreg <= A0;
            else if ((A==0) && (B==1)) Sreg <= OK0;
            else                       Sreg <= OK1;
    default Sreg <= INIT;
  endcase
```

combinational logic and the compiler may do some things that are silly in the state-machine context, like inferring latches for unspecified state/input combinations. For that reason, some designers prefer to combine the state memory and next-state logic into a single `always` block. We used this style for all of the counters and shift registers in Chapter 11, which are technically state machines.

To combine the state memory and the next-state logic, the first two `always` blocks in previous state machine are rewritten as shown in Program 12-2, in a new module `VrSMexc`. The next-state values for `Sreg` are set directly using non-blocking assignment statements within the sequential `always` block. If the next value of the state register is not specified, the synthesis tool understands that by convention, the register value is not to change at the clock edge. For example, if we deleted the last `else` clause in Program 12-2, we would still get the same synthesized circuit.

12.1.4 Reset Inputs

State machines and other clocked sequential circuits should normally have a reset or other initialization input to force them into a known starting state. Even if the starting state is unimportant in physical operation—for example, in a counter whose sole purpose is to provide a lower frequency output for other circuits—an initialization input is usually needed in simulation and in device testing to provide a known starting point.

Reset and initialization inputs may be synchronous or asynchronous. The latter use the flip-flops' asynchronous preset or clear inputs and can be asserted at any time, overriding the devices' CLOCK and other inputs. Synchronous reset and initialization inputs may use dedicated flip-flop control inputs that have

TOO MANY CODING STYLES

ABEL, the early HDL for programming PLDs, had basically just one coding style for designing state machines; Verilog allows many. Why? There is really no significant upside in all the different possibilities offered by Verilog, and there are a lot of ways they can get you into trouble.

The answer is in the history of the two languages. ABEL was designed first and foremost as a hardware description language, and it was used immediately to target real devices and real digital-design projects. Verilog, on the other hand, was designed as a simulation language, and a fairly general one at that. Its features were directed primarily at the needs of simulation, including the need to get fast simulation performance in an era when computers were not so fast and cheap. Verilog's use for synthesis came later.

So, when using Verilog for state-machine design, it is up to the designer to adopt a consistent coding style, one that will avoid errors and will be easily recognizable and maintainable by fellow team members. Normally, style guidelines will be published and maintained by the designer's company, but lacking that, feel free to use the guidelines recommended here!

been provided for that purpose, like the synchronous R and S inputs of the FDRE and FDSE flip-flop library components that we introduced in Table 10-1 on page 510. Or they may be obtained by gating the combinational logic signal that would otherwise be applied to a flip-flop's normal synchronous input, for example, ANDing R′ or ORing S with the normal D input signal.

Program 12-3 shows two readily synthesizable examples of state memory that have an active-high `RESET` input, the first synchronous and the second asynchronous. As we noted in the box on page 518, some synthesis tools are picky about the behaviorally specified asynchronous reset, requiring the asynchronous assignment to be written in the `if` clause and the synchronous assignment to be in the `else`.

The initial state upon reset need not be all-0s or all-1s, but the circuit cost of providing an arbitrary starting-state value varies with the target technology. For example, in the case of Xilinx 7-series FPGAs, we noted in Figure 10-33 on

Program 12-3 Synchronous and asynchronous reset for state machines in Verilog for new modules `VrSMexrs` and `VrSMexra`.

```
// State memory with active-high synchronous reset
always @ (posedge CLOCK) // Create state memory
    if (RESET==1) Sreg <= INIT; else Sreg <= Snext;

// State memory with active-high asynchronous reset
always @ (posedge CLOCK or posedge RESET) // Create state memory
    if (RESET==1) Sreg <= INIT; else Sreg <= Snext;
```

page 532 that each flip-flop's S/R input may be programmed at initialization to be S or R. On the other hand, in the 22V10 PLD (page 529), all of the flip-flops must be set or reset together.

An asynchronous initialization input can be asserted at any time, but in general, it cannot be negated at any time. A problem can occur if it is negated too close to a triggering clock edge. Some flip-flops may respond to the edge, and others may not, and the system may start operation in an invalid state. In larger circuits and systems, this problem is more likely simply because there are more opportunities for delays to vary across the physical circuit or system. To prevent this problem, reset signals that originate from outside sources are typically synchronized with the internal clock or clocks before being applied to internal flip-flops, regardless of whether their initialization inputs are synchronous or asynchronous. The details of such reset synchronization circuits of course depend on timing and start-up-sequencing requirements of the system.

12.1.5 Pipelined Moore Outputs in Verilog

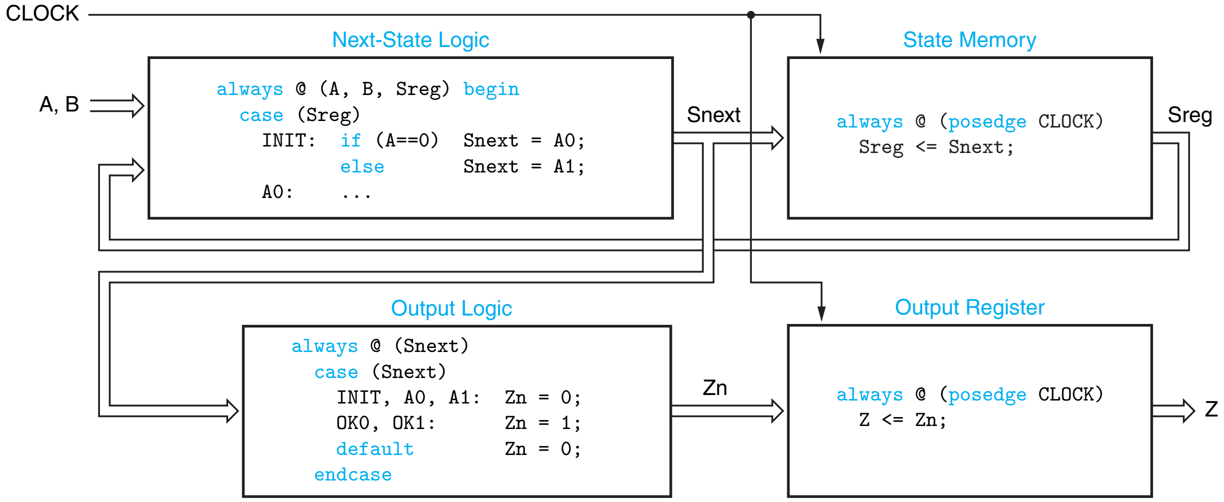
Another interesting variation is possible in our example Verilog state machine. As written, the module defines a conventional Moore-type state machine with the structure shown in Figure 12-1 on page 606. But we can convert the machine to have pipelined outputs with the structure shown in Figure 12-2. To do this, we need only to declare a "next-output" variable `Zn` and replace the original Verilog state-memory and output code of Program 12-1 with the code shown in Program 12-4, corresponding to the structure shown in Figure 12-2. Here, instead of computing `Z` from the inputs and `Sreg`, the output logic computes `Zn` from the inputs and `Snext`. The value of `Zn` is loaded into the pipelined output register to produce `Z` at the next clock tick.

The new machine's behavior is indistinguishable from that of the original machine, except for timing. We've reduced the propagation delay from `CLOCK` to `Z` by producing `Z` directly on a register output, but we may have also increased the setup-time requirements of `A` and `B` to `CLOCK`. In addition to their propagation delay through the next-state logic, changes in `A` and `B` must also propagate through the output logic in time to meet the setup time requirement of the output

Program 12-4 Verilog pipelined output code.

```
always @ (posedge CLOCK)   // Create output register -- combine
  Z <= Zn;                 // with state-memory code if desired

always @ (Snext)           // Output logic
  case (Snext)
    INIT, A0, A1: Zn = 0;
    OK0, OK1:     Zn = 1;
    default       Zn = 0;
  endcase
```



```
always @ (A, B, Sreg) begin
  case (Sreg)
    INIT:  if (A==0)  Snext = A0;
           else       Snext = A1;
    A0:    ...
```

```
always @ (posedge CLOCK)
  Sreg <= Snext;
```

```
always @ (Snext)
  case (Snext)
    INIT, A0, A1:  Zn = 0;
    OK0, OK1:      Zn = 1;
    default        Zn = 0;
  endcase
```

```
always @ (posedge CLOCK)
  Z <= Zn;
```

Figure 12-2 Structure of Verilog state machine with pipelined outputs.

flip-flop's D input. Depending on the implementation technology and the synthesis tools, this extra delay may or may not actually occur. When the new design was targeted to a Xilinx FPGA using their Vivado tools, the synthesizer was able to "flatten" the logic for Zn to fit in a single LUT driving the D input of the output (Z) flip-flop. So, the timing path for Zn to the D input was about the same as the path for any of the Snext bits to the state-memory D input.

12.1.6 Direct Verilog Coding Without a State Table

All of the variations in the example state-machine design that we've shown so far rely on the state table that we originally constructed by hand in Section 9.3.1. However, it is possible to write a Verilog model directly, without creating a state table or state diagram at all.

Based on the original problem statement on page 607, the key simplifying idea is to remove the last value of A from the state definitions, and instead to have a separate register that keeps track of it (LASTA). Then only two non-INIT states must be defined: LOOKING ("still looking for a match") and OK ("got a match or B has been 1 since last match").

A Verilog module based on this approach is shown in Program 12-5. The first always block creates the state memory and the LASTA register. The second one creates the next-state logic using the idea in the preceding paragraph. The Z output is a simple combinational decode of the OK state, so we create it using a continuous-assignment statement instead of the more verbose always block and case statement. Note this requires Z to be declared as a wire rather than a reg.

The next-state logic in Program 12-5 is easier to understand and relate to the word description than the original, and we avoided the tedium of developing

Program 12-5 Simplified Verilog state-machine design.

```
  module VrSMexa( CLOCK, RESET, A, B, Z );
  input CLOCK, RESET, A, B;
  output wire Z;      // declared as wire for continuous assignment
  reg LASTA;                       // LASTA holds last value of A
  reg [1:0] Sreg, Snext;           // State register and next state
  parameter [1:0] INIT    = 2'b00, // Define the states
                  LOOKING = 2'b10,
                  OK      = 2'b11;

  always @ (posedge CLOCK) begin // State memory (with sync. reset)
    if (RESET==1) Sreg <= INIT; else Sreg <= Snext;
    LASTA <= A;
  end

  always @ (A, B, LASTA, Sreg) begin  // Next-state logic
    case (Sreg)
      INIT:    Snext = LOOKING;
      LOOKING: if (A==LASTA)         Snext = OK;
               else                  Snext = LOOKING;
      OK:      if (B==1 || A==LASTA) Snext = OK;
               else                  Snext = LOOKING;
      default                        Snext = INIT;
    endcase
  end

  assign Z = (Sreg==OK) ? 1 : 0;      // Output logic
endmodule
```

a state table. Clearly, this is what HDL-based state-machine design is all about. The rest of the examples in this chapter are developed without a state table or state diagram. We'll also look at state-machine test benches, starting right after the following optional subsection.

*12.1.7 State-Machine Extraction

Some Verilog synthesis tools have a feature for recognizing state machines in HDL code, as long as they match a prescribed template. Once they recognize a state machine, the tools can do interesting things with it, such as trying different state encodings or using "special" optimization methods that are tuned to the target technology. In the Xilinx Vivado tool suite, this feature is called *FSM extraction*. The prescribed templates for state-machine extraction typically include requirements such as the following:

FSM extraction

- Next-state behavior is specified by a `case` statement where the selection expression is the state variable, which itself is a multibit vector of type `reg`.

* Throughout this book, optional sections are marked with an asterisk.

- The state variable is assigned only constant values, if not directly, then indirectly in a way that the tool can figure out. For example, in our recommended state-machine coding style, a variable, `Snext`, is assigned to `Sreg`, but `Snext` is assigned only constant values and Vivado can determine this.
- The number of different constant values (states) is at least some minimum; in Vivado the default minimum is five.
- Ideally, the constant values are defined in a `parameter` statement, though that may not be a strict requirement.
- There are no assignments to individual bits or multibit parts of the state variable.
- The state variable is not declared as a module output.

Once the synthesis tool recognizes a state machine, it converts it to an internal symbolic representation. At that point, the most common optimization is to try different state encodings and to select the one that gives the best results according to some metric, like overall circuit size or timing performance. In Vivado, there are several encodings that can either be tried automatically by the tool or forced by a user option:

- *Sequential*. The states are encoded with a minimum number of bits, with constant values assigned in binary counting order as they are encountered in the internal symbolic representation, which is not necessarily the same order as in the original `case` statement.
- *Gray*. The states are assigned in a Gray-code counting sequence matching the loop structure of the machine's next-state behavior. This only works well if the machine has a main loop structure or other long chains of states with no branching. One goal of Gray-code order is to have only one state variable change on each transition, but this goal cannot be completely achieved even in a pure loop if the number of states is not even (see the box on page 466).
- *Johnson*. The states in the machine's loop structure, if it has one, are assigned in the same sequence as those of a Johnson counter. Note that an n-bit Johnson counter provides a maximum of $2n$ states. The main benefit of Johnson encoding, even without a looping next-state behavior, is that each state can be decoded from just two state bits regardless of the total number of state bits. This may reduce the total resource requirements, especially in an FPGA if decoding all of the state bits plus inputs would require more than one LUT.
- *One-hot*. Each state has a corresponding state bit that is set to 1 in that state and to 0 in all others, so there are n bits for n states. States in this encoding require the most flip-flops but have the simplest state decoding. This may result in a faster circuit, with fewer levels of logic needed for decoding.

State-machine optimizations and transformations (like using different state encodings) are local to the module in which they are declared. That's why they cannot be done if the state variable is declared as a module output—any re-encoding of the state assignment would appear outside the module, and other modules using that output would be unaware of it. If the variable is local, and outputs are subsequently derived from it, that's OK; the synthesis tool can create logic to derive the same output values from the new state encoding.

The registers in all of the sequential modules in Chapter 11 were declared as module outputs, which is one reason that the synthesis tool does not attempt state-machine extraction on any of them. Another reason is that all of their registers have variables, not constants, assigned to them in one or more places. Most of the state-machine examples in the present chapter, however, can use state-machine extraction. We'll look at one example in particular in Section 12.5.

12.2 Verilog State-Machine Test Benches

We explained the general concept of Verilog test benches in Section 5.13. The test bench for a state machine has four basic parts:

1. Declaration of the test-bench module itself. Note that this module has no inputs and outputs of its own.
2. Component instantiation of the state-machine entity to be tested, often called the unit under test (UUT).
3. An `always` block to create a free-running clock.
4. Statements to initialize the UUT, to apply a sequence of test-vector inputs at each clock tick, and to check for expected output values.

Part 4 requires the most work, typically more than is required for combinational circuits. For combinational circuits with a relatively small number of inputs, we can often use the "brute force" approach of applying all possible inputs and checking the outputs against a "functionally" determined result (e.g., selected or decoded value, arithmetic result, etc.). This is even true for sequential circuits that perform easily described functions, like the counters and shift registers in Chapter 11. The function of a general state machine may not be so easily described; its only authoritative description may be its state table or diagram, if one exists, or the HDL code itself. But correctness of the HDL code is what we're trying to check—we may have a chicken and egg problem! How can we devise a test sequence that checks the machine for all situations?

The short answer to the question is that, much the same as with combinational circuits and the functionally specified sequential circuits of Chapter 11, we should try to come at the design of state-machine test benches from a different perspective than we used when creating the state table or diagram or the HDL code itself. There are at least two methods of test-bench construction that do this, discussed next.

12.2.1 State-Machine Test-Bench Construction Methods

The first method contrasts with our early statement on state-machine design, that you can't easily design a state machine just by looking at an example input sequence and the resulting output waveforms. However, there's nothing to stop us from seeing if a state-machine-as-designed is operating as we expect it to, by examining the waveforms that it actually does produce from an example input sequence. For this method to be effective, we need the input sequence to fully exercise the machine, and it can do that in one of two ways:

1. The input sequence should cause the machine to visit every normal state, and eventually to take every possible transition out of every normal state.
2. The input sequence should exercise every "feature" of the machine under every possible variation of circumstances or exceptions.

The first way can be fairly precise, since some tools can track which of a module's HDL statements are executed in simulation. Even if they can't, we can manually modify the machine's next-state logic (`case` statement) to track this, as we'll show later.

The second way is a little "softer," but it has one benefit—we may think about the features and their variations a little differently at this stage than when we first wrote the next-state logic, so there is an opportunity (and perhaps even a tendency) to consider the features and variations more comprehensively.

Regardless of which of the above ways a designer uses to generate the input sequence, it remains the designer's responsibility to manually examine the resulting output waveforms and to determine if they are correct according to the machine's high-level description. This may require a great amount of attention to detail.

The second construction method is to create a self-checking test bench, with much the same goals and benefits that we had for creating self-checking test benches for combinational circuits, and for the more easily described sequential functions in Chapter 11. Once again, we must construct an input sequence that fully exercises the machine. In this case, however, we check the machine's output at each step along the way to be sure that it matches what's expected according to the machine's high-level description.

In applying this second method, one possible approach would be to check the machine's output in every state (for a Moore machine) or for every state/input combination (in a Mealy machine), but this would be a mere parroting of the machine's state diagram, state/output table, or HDL description. It makes more sense in the second method to construct the input sequence at a higher level to exercise its every feature and variation, and then to check the resulting outputs at the same level.

It's quite possible in the construction of a self-checking test bench that we can make errors in our prediction of what the machine's output should be for some parts of the input sequence. Thus, test-bench construction and use may be an iterative process in which we find and correct errors in the test bench as well as in the UUT.

Finally, there is a third type of test bench, similar to what we had with some combinational circuits in previous chapters. If we already have a known-good, "golden" module that implements the state machine exactly as desired, we can write a test bench that compares the outputs of any new implementation with those of the golden module for a comprehensive input sequence. In this case, no functional understanding of the modules' operation is strictly required. Rather, we need only to use the first way of creating the test bench's input sequence, ensuring that it visits every normal state and eventually takes every possible transition out of every normal state of the golden machine.

12.2.2 Example Test Benches

We'll illustrate the test-bench-construction methods using our familiar example state machine of Program 12-1 on page 608. But first, you should notice that this machine doesn't have a reset input. Therefore, we can't simulate it properly—since its state memory can't be initialized, its starting state is unknown. So, we'll only test versions of the machine with a `RESET` input as in Programs 12-3, 12-4 and 12-5 on pages 611, 612 and 614. With this in mind, we can write a test-bench module according to the first construction method as shown in Program 12-6.

The first section of the module declares local variables to apply to the inputs and to capture the output of the state machine. It also declares two "test vectors" `Avec` and `Bvec` which will be initialized with 30-bit constants that will be sequenced into `A` and `B`, one bit per clock tick, in the main body of the test. The second section instantiates the state machine, giving it the component name "`UUT`"; as usual, the local variable names can be the same as or different from the port names.

Next, the test bench has an `always` block to create a free-running clock `Tclk` with a 10-ns period. Since we are running a functional simulation with no embedded timing information, it doesn't matter what clock period we use. However, one subtlety is that, regardless of its period, the clock's transition from undefined to 1 at time 0 may be considered to be a rising edge, and the state flip-flops may respond accordingly. But we won't count on this; we'll keep the reset input asserted at the beginning of the test.

The `initial` block is devoted to applying the actual test inputs. Since we're new at this, we start with a `$monitor` task that prints all signals values on the system console any time one of them changes. The reset input is asserted for quite a while. The first 100 ns of that is to get beyond the "global reset" in the Xilinx FPGA environment, which was explained previously in conjunction with Program 10-2 on page 512. The extra 15 ns is to get to the middle of the second post-reset clock period, by which time the machine should have found its way into the `INIT` state.

Program 12-6 Test bench for applying an input sequence to a VrSMex state machine.

```
`timescale 1ns/100ps
module VrSMex_tbv ();
  reg Tclk, RST, A, B;
  wire Z;
  reg [1:30] Avec, Bvec;
  parameter Aseq = 30'b10110010010000111010110010110111,
            Bseq = 30'b00000011111000001110001110100000;
  integer i;

  VrSMex UUT ( .CLOCK(Tclk), .RESET(RST), .A(A), .B(B), .Z(Z) ); // instantiate UUT

  always begin         // create free-running test clock with 10 ns period
    #0.5 Tclk = 1; #5; // 5 ns high  (small offset for
    Tclk = 0; #4.5;    // 5 ns low    waveform readability)
  end

  initial begin    // What to do starting at time 0
    $monitor("Time:%d  RST=%b Tclk=%b A=%b B=%b Z=%b",
             $time, RST, Tclk, A, B, Z); // Monitor all signals
    RST = 1;       // Apply reset
    A = 1; B = 1;  // A and B are 1 too
    Tclk = 1;      // Start clock at 1 at time 0
    Avec = Aseq; Bvec = Bseq;         // Init A and B input-sequence vectors
    #115;          // Wait 115 ns
    RST = 0;       // unreset
    for (i=1; i<=30; i=i+1) begin // Apply input sequence for 30 ticks
      A = Avec[1]; Avec = Avec<<1;
      B = Bvec[1]; Bvec = Bvec<<1;
      #10 ;
    end
    $stop(1);                         // end test
  end
endmodule
```

After negating the reset input, the test bench executes a `for` loop that applies test inputs to `A` and `B`, as defined by `Aseq` and `Bseq` and shifted out of `Avec` and `Bvec`, every 10 ns for 30 clock ticks. The resulting `Z` outputs can be observed both on the system console because of the `$monitor` task, and in the simulator's waveform display.

The test vectors `Aseq` and `Bseq` were chosen purposefully but without a lot of deep thought, to provide successive values on `A` that sometimes matched and sometimes did not, and to combine those with values on `B` that would sometimes hold the `Z` output at 1 regardless and sometimes would not. Do the chosen test vectors visit all of the states and exercise all of the transitions out of those states in the UUT? There's no easy way to know without using a tool that tracks that, or instrumenting the UUT's next-state code as we'll show later. However, as shown

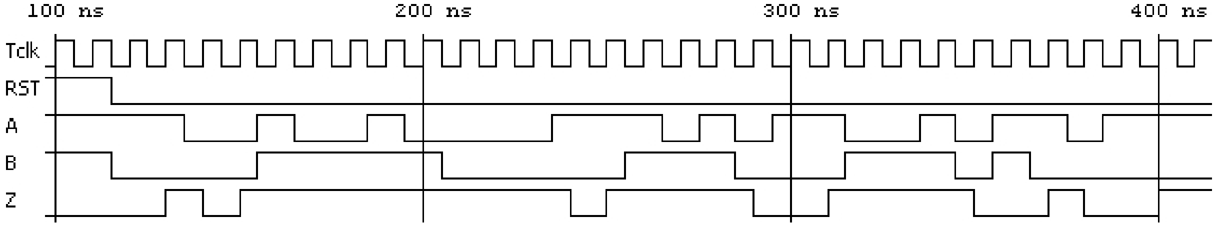


Figure 12-3 Timing waveforms created by `VrSMex_tbv` test bench.

in the waveforms in Figure 12-3, they do create a good variety of responses on the `Z` output, and careful inspection shows that `Z` does match what's expected based on machine's functional specification.

The second construction approach, a self-checking test bench, is shown in Program 12-7. It begins similarly to the first test bench, but does not define test vectors; the test sequence is embedded later. It does, however, define a task `checkZ` that will be used to apply test inputs and compare the simulated output `Z` with an expected value, and print an error message and stop if they are different. We defined it to save typing and clutter in the test code, which begins as before at 115 ns by negating the reset input.

Since the state definitions are "hidden" in the UUT's module definitions, the test bench cannot check the state directly. But we know that the output should be 0 in the `INIT` state, so our first call of the `checkZ` task prints a message and stop the simulation if `Z` is not 0 at this point. Then we negate `RST` and call `checkZ` again to apply the next values to `A` and `B`, wait 10 ns, and compare the new value of `Z` with what's expected. If any of the actual outputs fail to match, the simulation stops and prints an error message so we can investigate the problem.

Note that the test bench is checking the machine's *functional* behavior, pretty much at the level of our original word description. Except for a little special knowledge of what happens at initialization, the test bench makes no reference to the actual internal states of the machine. So, the same test bench could be used with different versions of the same state machine with different states or state assignments, or other subtle differences.

Suppose that we are satisfied with the performance of the `VrSMexa` state machine as tested by the preceding test bench, or perhaps even in actual usage,

LOOKING UNDER THE HOOD

Beyond the port definitions for a particular UUT instantiated by a test bench, Verilog purposely hides the inner workings of the module's implementation. But for debugging purposes, you'd really like to see what's going on inside. So, when you run your test bench on an interactive simulator, you can normally drill down and see signal values within the UUT module's implementation.

Program 12-7 Self-checking Verilog test bench for the `VrSMex` state machines.

```
`timescale 1ns/100ps
module VrSMex_tb ();
reg Tclk, RST, A, B;
wire Z;

VrSMexa UUT ( .CLOCK(Tclk), .RESET(RST), .A(A), .B(B), .Z(Z) ); // instantiate UUT

task checkZ;  // Task to apply inputs, wait, check output, and print if wrong
  input stepnum, ai, bi, expectZ;
  integer stepnum; reg ai, bi, expectZ;
  begin
    A = ai; B = bi; #10 ;
    if (Z != expectZ) begin
      $display($time," Error, step %d, expected %b, got %b",
               stepnum, expectZ, Z); $stop(1); end;
  end
endtask

always begin    // create free-running test clock with 10 ns period
  #6 Tclk = 0;  // 6 ns high
  #4 Tclk = 1;  // 4 ns low
end

initial begin       // What to do starting at time 0
  $monitor("Time:%d RST=%b Tclk=%b A=%b B=%b Z=%b", $time, RST, Tclk, A, B, Z);
  RST = 1;          // Apply reset (synchronous for this UUT)
  A = 1; B = 1;     // A and B are 1 too
  Tclk = 1;         // Start clock at 1 at time 0
  #115;             // Wait 15 ns, past at least on rising clock edge
  checkZ(1,1,1,0);  // Expect Z=0 initially
  RST = 0;          // unreset
  checkZ(2,1,1,0);  // still Z=0 after INIT
  checkZ(3,1,0,1);  // Two 1s in a row, want Z=1
  checkZ(4,0,1,1);  // B=1 should hold Z=1
  checkZ(5,1,0,0);  // B=0 releases it
  checkZ(6,0,1,0);  // B=1 but nothing to hold
  checkZ(7,0,0,1);  // But now two 0s in a row
  checkZ(8,1,1,1);  // B=1 should hold Z=1
  checkZ(9,0,0,0);  // B=0 releases it
  $stop(1);         // end test
  end
endmodule
```

so we would like to use it as a “golden” reference design. In the third method of test-bench construction, we can check other designs of the same state machine against the reference design. Program 12-8 shows the approach. The new test bench instantiates two UUTs, the first being the “golden” `VrSMexa` module, and

Program 12-8 Test bench to compare `VrSMex` state machines for a long random input sequence.

```
`timescale 1ns/100ps
module VrSMex_tbr ();
reg Tclk, RST, A, B;
wire Z1, Z2;
integer i;

VrSMexa U1 ( .CLOCK(Tclk), .RESET(RST), .A(A), .B(B), .Z(Z1) ); // instantiate UUT
VrSMexp U2 ( .CLOCK(Tclk), .RESET(RST), .A(A), .B(B), .Z(Z2) ); // instantiate UUT

always begin    // create free-running test clock with 10 ns period
  #6 Tclk = 0;  // 6 ns high
  #4 Tclk = 1;  // 4 ns low
end

initial begin
  RST = 1;        // Apply reset
  A = 1; B = 1;   // A and B are 1 too
  Tclk = 1;       // Start clock at 1 at time 0
  #115;           // Wait 115 ns
  RST = 0;        // unreset
  for (i=1; i<=3000; i=i+1) begin // Apply random inputs sequence for 3000 ticks
    A = $random; // This will get the LSB of new random number
    B = $random; // This will get the LSB of next random number
    #10 ;
    if (Z1 !== Z2) $display("Iteration %d error, Z1,Z2 = %b,%b", i, Z1, Z2);
  end
  $display("Test completed");
  $stop(1);       // end test
  end
endmodule
```

the second being the pipelined-output version of Program 12-4 on page 612. After the usual clock creation and signal initialization, the test bench executes a `for` loop that applies a randomly generated 3000-tick input sequence to both UUTs, using Verilog's built-in `$random` function (only the LSB is used). Their `Z` outputs are compared at each tick, and any mismatch is flagged.

Generating an effective random input sequence for this machine was pretty easy, but that's not always true. Depending on the machine, "random" inputs may not be too likely to produce common operational scenarios. We saw this for some combinational circuits, like comparators in Section 7.4.7. In sequential circuits, a very specific and long input sequence may be needed to set up the machine to be ready to exercise one or more behaviors. Thus, even for testing against a "golden" module, it may be necessary to provide an input sequence that has been customized to exercise all of the machine's features, rather than to rely on a random one.

PUNTING ON RESET

If you tried, you might actually get away with simulating a state machine with no reset. Your simulator may initialize flip-flops to a known state, usually 0, instead of unknown. In the example machine with the simplest state assignment, all-0s corresponds to `INIT`, which is what we wanted anyway.

This behavior may be accurate, but it's dangerous. It may accurately model the physical design, because the flip-flops in many PLDs and FPGAs are guaranteed to come up in the 0 state, as long as power is applied smoothly. It's dangerous for many reasons, some of which are explained below.

At some point during normal circuit operation, the power supply voltage may experience a glitch, enough to change some flip-flop states but not enough to activate the device's automatic power-up reset circuit. This could leave your state machine in an unknown state with no way to bring it back. You might not notice this potential pitfall while debugging in the lab.

During the design process, you might decide to change your machine's state encoding, such that the device's power-on reset state is no longer a good one in all cases. But you might not notice this in simulation.

Late in the game, you (or your production department, or by this point, your successor) might change the PLD or FPGA device containing your state machine to one that has a different or no guaranteed power-on reset state. In the rush to get the revision into production, no one notices.

All of these cases are ones that were never caught in simulation, because the machine always "just worked" with no reset. So, please always provide a reset for your state machines, and use it in simulation.

12.2.3 Instrumenting Next-State Logic for Testing

We mentioned previously that it is possible to "instrument" a UUT's next-state logic to determine whether all of its transitions are exercised by a test pattern. Note that this must be done in the UUT, not in the test bench, since a given functional behavior may be achieved by different but equivalent state machines having different states and transitions. Program 12-9 shows the changes needed to instrument the `VrSMex` state machine of Program 12-1. At the beginning of the module, we define a "global" integer variable `savetr` and a task `Tchk` with two inputs: an integer `tr` for a transition number and a vector `next` for a next-state value. When invoked, the task simply sets `Snext` equal to `next` and `savetr` to `tr`. The `always` block that creates the state memory is modified to display the saved transition number when it is finally taken at the clock edge. The last step is to modify the next-state logic: each place where we used to assign a next state to `Snext`, we instead invoke `Tchk` with a unique integer and that next state.

When we run a test bench on an instantiation of the instrumented module, the identifying number of each transition will be displayed. Using the 30-input test sequence in Program 12-6, the display shows that transitions 1, 5, and 13 are never taken. Transition 1 not being taken makes sense, since the test bench only

Program 12-9 Changes to the `VrSMex` module to display taken state transitions.

```
integer savetr;   // Int to save taken-transition number

task Tchk;        // Task to display and take transition
  input tr, next;
  integer tr; reg [2:0] next;
  begin
    Snext = next;
    savetr = tr;
  end
endtask

always @ (posedge CLOCK)        // Create state memory
  if (RESET==1) Sreg <= INIT;   // Sync reset
  else begin
    Sreg <= Snext; // Save new state and display which transition did it
    $display("Time: %4d, took transition %2d",$time,savetr);
  end

always @ (A, B, Sreg) begin            // Next-state logic
  case (Sreg)
    INIT:   if (A==0)                  Tchk(1,A0);
            else                       Tchk(2,A1);
    A0:     if (A==0)                  Tchk(3,OK0);
            else                       Tchk(4,A1);
    A1:     if (A==0)                  Tchk(5,A0);
            else                       Tchk(6,OK1);
    OK0:    if (A==0)                  Tchk(7,OK0);
            else if ((A==1) && (B==0)) Tchk(8,A1);
            else                       Tchk(9,OK1);
    OK1:    if ((A==0) && (B==0))      Tchk(10,A0);
            else if ((A==0) && (B==1)) Tchk(11,OK0);
            else                       Tchk(12,OK1);
    default                            Tchk(13,INIT);
  endcase
end
```

HOLD YOUR HORSES!

It would be premature to display the value of `tr` immediately, when `Tchk` is invoked in the next-state logic. Since that's combinational logic, `Snext` may undergo further changes before the clock edge. Not all inputs that affect `Snext` necessarily change at the same time, so `Snext` may undergo several changes before the clock edge when it is finally stored in `Sreg`. Only the final, taken transition number is of interest.

NOT SO EASY Checking that “all possible” next-state transitions are taken is not as easy as it first may seem in this example. A single transition as it appears in an `if` or `else` clause in the next-state logic could actually be taken for multiple, different input combinations, depending on the logic of the `if` condition. For example, transitions 1, 3, 5, and 7 in Program 12-9 could be taken with either `B=0` or `B=1`. But in the test bench’s operation, each of their transition numbers will be displayed for either value of `B`, even if the other value is never seen.

Still, checking that each listed transition is taken at least once is a very good sanity check for next-state logic. If a listed transition is never taken, and moreover if it is then found to be difficult or impossible to provoke, it could indicate an error in either the next-state logic or in the designer’s understanding of how the machine is supposed to work.

leaves the `INIT` state once, with `A=1`. And we never expected transition 13, the `default` case, to be taken in normal (non-error) operation. But the absence of transitions 5 would have much been harder to spot without the instrumentation.

We can also run the self-checking test bench of Program 12-7 using the instrumented module. This “hand crafted” test bench actually does much worse than the previous one, missing transitions 1, 4, 7, 12, and 13. Updating both test benches for complete transition coverage is left as Exercises 12.6 and 12.7.

12.2.4 In Summary

As you’ve seen, creating a comprehensive set of functional-test patterns for a large state machine by hand can be a painstaking process. The test patterns should visit all of the states and use all of the transitions out of each state; our examples didn’t! Using these patterns, you must make sure that the machine’s behavior at each step “makes sense.” This is more than determining that the machine does what your code says it should—it will in most cases, because you used automated tools to go from the code to the implementation. More important is to determine that *what your code says* makes sense in all cases. This is especially important in infrequently used and perhaps unconsidered “corner cases.” Success in this step comes only with practice and (often bad) experience.

Creating test patterns for use in manufacturing to detect hardware failures is another matter. Here, the assumption is that your machine’s functional definition is correct, and you want to verify that the actual hardware as built matches the specified behavior. Test patterns for this purpose are typically and best left to an automatic test-pattern-generation program. Instead of starting with a Verilog description, such a program typically uses a gate-level or other component-level description of the implemented circuit to generate tests. In this way, it can create tests that are likely to catch errors based on the expected physical failure modes of the target device.

12.3 Ones Counter

Our first new Verilog example in this chapter is a “1s-counting machine” with the following specification:

> Design a clocked synchronous state machine with two inputs, X and Y, and one output, Z. The output should be 1 if the number of 1 inputs on X and Y since reset is a multiple of 4, and 0 otherwise.

At first glance, you might think the machine needs an infinite number of states, since it counts 1 inputs over an arbitrarily long time. However, since the output indicates the number of inputs received *modulo 4*, four states are sufficient to keep track of the count, and we can just use two bits, Q[1:0], to encode them.

In this example, we can make good use of Verilog’s arithmetic capabilities, in particular addition, to simplify our coding task. Program 12-10 is a Verilog module that does the job. Instead of naming the states and defining the next-state logic using a `case` statement with four cases, we have simply encoded the state as a 2-bit number, and used addition to advance the state by 0, 1, or 2 depending on the number of 1s in the `X` and `Y` inputs. The output logic simply asserts `Z` if `Q[1:0]` is 0.

On the surface, this is a very simple example to understand—the code seems very natural—so rather than dwell on the logic, this is a good place to mention some of the finer points and potential pitfalls in Verilog coding, and what’s going on “under the hood” when the synthesis tool processes our code.

So, let us dissect the first `if` statement in the next-state logic, which tests whether `X` and `Y` are both 1 and if so, adds 2 to `Q`. How is the condition tested and what does the compiler see? The expression “`X&Y`” performs a *bitwise boolean* AND operation on two 1-bit signals and its result is a 1-bit signal. The `if` state-

Program 12-10 Verilog module for 1s-counting machine.

```
 module VronescntSM( CLOCK, RESET, X, Y, Z );
  input CLOCK, RESET, X, Y;
  output reg Z;
  reg [1:0] Q, Qnext;

  always @ (posedge CLOCK)                 // Create state memory
    if (RESET==1) Q <= 0; else Q <= Qnext; // Sync reset

  always @ (X, Y, Q)        // Next-state logic
    if (X & Y) Qnext = Q + 2;
    else if (X | Y) Qnext = Q + 1;
    else Qnext = Q;

  always @ (Q) begin        // Output logic
    if (Q==0) Z = 1; else Z = 0;
  end
endmodule
```

ment is looking for a true/false value, and that a 1-bit signal is considered to be true if its value is `1'b1`, and false if it's `1'b0` or anything else, including `1'bx` (an "unknown" value in simulation). A technically more correct but also much more verbose way to write the condition would be "`(X==1'b1)&&(Y==1'b1)`," which compares the `X` and `Y` inputs against the proper signal value, producing true/false results and combining those results with the *logical* AND operator `&&`. See the discussion of logical operators and expressions in Section 5.5.

The next fine point in the `if` statement is how we add 2 to `Q`. In this module, "`Q`" has been defined as a 2-bit vector (default unsigned), while "2" is a integer constant, a "signed" number that happens to be positive. Because at least one of the operands is unsigned, the Verilog addition operation defaults to unsigned, and we get the desired result in both simulation and synthesis. With different operand types we could get different results. See the discussion of vectors and arithmetic in Section 5.3.

Why are these fine points important? Depending on the application, the coding details, and the environment, you could have situations where the simulation doesn't match the actual circuit's operation, unknown values (`1'bx`) go undetected in simulation, or both the circuit and the simulation operate incorrectly (for example, if you naively expect an expression like "`(2'b10 & 1'b1)`" to evaluate to "true").

There are of course many other ways to code the addition operation in the next-state logic in Program 12-10 that do *not* change its meaning. An elegant way is to avoid the `if` statement entirely, replacing it with a direct addition of `X` and `Y` to `Q[1:0]` as follows:

```
Qnext = Q + {1'b0,X} + {1'b0,Y};
```

Another coding-style option that's easily applied to this machine is to combine the state register and the next-state logic (i.e., the first two `always` statements) into a single sequential `always` statement as shown in Program 12-11. In this simple machine, when all is said and done, such stylistic changes typically don't change the resulting synthesized circuit, which requires just two flip-flops and two LUTs in a Xilinx 7-series FPGA.

Program 12-12 is a self-checking test bench for the 1s-counting machine. It follows our recommended approach of using a different method to determine the state machine's outputs than what is likely used internally. In this example, it generates random inputs for the machine's `X` and `Y` inputs, and calculates their

Program 12-11 Combined state memory and next-state logic.

```
  always @ (posedge CLOCK)             // Create state memory ...
     if (RESET==1) (RESET==1) Q <= 0; // Sync reset
     else if (X & Y) Q <= Q + 2;      // ... and next-state logic
     else if (X | Y) Q <= Q + 1;
//   else Q <= Q;                      // optional
```

Program 12-12 Verilog self-checking test bench for the 1s-counting machine.

```
module VronescntSM_tb ();
  reg Tclk, RST, X, Y;
  wire Z;
  integer i, sum;

  VronescntSM UUT (.CLOCK(Tclk), .RESET(RST), .X(X), .Y(Y), .Z(Z)); // Instantiate UUT

  always begin    // create free-running test clock with 10 ns period
    #6 Tclk = 0;  // 6 ns high
    #4 Tclk = 1;  // 4 ns low
  end

  initial begin
    RST = 1;          // Apply reset
    X = 0; Y = 0;     // Inputs 0 to begin
    Tclk = 1;         // Start clock at 1 at time 0
    #115;             // Wait 115 ns
    RST = 0;          // unreset
    sum = 0;          // Track # of 1s in test inputs
    for (i=1; i<=3000; i=i+1) begin     // Apply random inputs for 3000 ticks
      X = $random;    // This will get the LSB of new random number
      Y = $random;    // This will get the LSB of next random number
      sum = sum + X + Y;
      #10 ;
      if (Z!==((sum % 4)===0)) $display("Iteration %4d error, sum=%0d, Z=%b",i,sum,Z);
    end
    $stop(1);          // end test
  end
endmodule
```

running sum. To calculate the expected Z output value at each clock tick, it uses Verilog's modulo operator to divide the sum by 4, and then uses that result in a logical expression that is 1 if the remainder is 0.

12.4 Combination Lock

The next example is a "combination lock" state machine that activates an "unlock" output when a certain binary input sequence is received, and also provides a "hint" output to guide the user who knows how to use it:

> Design a clocked synchronous state machine with one input, X, and two outputs, UNLK and HINT. The UNLK output should be 1 if and only if X is 0 and the sequence of inputs received on X at the preceding seven clock ticks was 0110111 (the rightmost bit being the most recently received). The HINT output should be 1 if and only if the current value of X is the correct one to move the machine closer to being in the "unlocked" state (i.e., to get UNLK = 1).

It is apparent from the word description that this is a Mealy machine. The UNLK output depends on both the past history of inputs and X's current value, and HINT depends on both the state and the current X (indeed, if the current X produces HINT = 0, then the clued-in user will want to change X before the clock tick).

This machine is a little different in its requirements from previous examples, but let's give it a try. For the UNLK output at least, it seems obvious that we need to know the input sequence we have seen so far, so that we can decide whether the next input gets us one step closer to our goal or forces us to backtrack. Thus, we can construct the boilerplate, state definitions, next-state logic, and output logic at least for `UNLK` as shown in Program 12-13. Here, we have used state names corresponding to the initial part of the combination sequence that we have received so far.

Program 12-13 Verilog module for the combination-lock machine.

```
module VrcomblockSM( CLOCK, RESET, X, UNLK, HINT );
  input CLOCK, RESET, X;
  output wire UNLK, HINT;   // declare as wires for continuous assignment
  reg [2:0] Sreg, Snext;    // State register and next state
  parameter [7:1] COMBINATION = 7'b0110111; // not used, but put here for reference
  parameter [2:0] GOTZIP     = 3'b000, // Define the state encoding
                  GOT0       = 3'b001, // Each state indicates we've received
                  GOT01      = 3'b011, //   progressively more of the unlock sequence.
                  GOT011     = 3'b010, //
                  GOT0110    = 3'b110, // States are Gray coded to potentially
                  GOT01101   = 3'b111, //   simplify the excitation logic.
                  GOT011011  = 3'b101,
                  GOT0110111 = 3'b100;

  always @ (posedge CLOCK)   // State memory (with sync. reset)
    if (RESET==1) Sreg <= GOTZIP;
    else Sreg <= Snext;

  always @ (Sreg or X)        // next-state logic
    case (Sreg)
      GOTZIP:     if (X) Snext = GOTZIP;     else Snext = GOT0;
      GOT0:       if (X) Snext = GOT01;      else Snext = GOT0;
      GOT01:      if (X) Snext = GOT011;     else Snext = GOT0;
      GOT011:     if (X) Snext = GOTZIP;     else Snext = GOT0110;
      GOT0110:    if (X) Snext = GOT01101;   else Snext = GOT0;
      GOT01101:   if (X) Snext = GOT011011;  else Snext = GOT0;
      GOT011011:  if (X) Snext = GOT0110111; else Snext = GOT0110;
      GOT0110111: if (X) Snext = GOTZIP;     else Snext = GOT0;
      default:                                    Snext = GOTZIP;
    endcase
  // Output logic -- Detect combination
  assign UNLK = ( (Sreg==GOT0110111) && (X==0) ) ? 1 : 0;
  assign HINT = 1'b0;  // Haven't figured out how to do this yet
endmodule
```

In the next-state case statement, each case advances us to the next state if we get the correct input, and otherwise it sends us back. However, we don't always have to go all the way back to the beginning (GOTZIP or GOT0); sometimes a wrong input sends us back only partway (e.g., after receiving a 0 in state GOT011011).

Notice that the HINT output is set to 0 in Program 12-13, because we haven't yet figured out an easy way to generate it correctly; we'll come back to that. As it is, it was not particularly easy to write the next-state logic either. We had to define all the state names, based on the particular unlocking combination, and then tailor the if statement in each next-state case according to the next value needed to match the combination. On top of that, in case of a mismatch, we had to look for opportunities to backtrack only partway instead of all the way to the beginning. And if we need to change the machine to recognize a different combination, we have to do this all over again. If the combination is a variable stored in its own register, rather than a design-time constant, this approach doesn't work at all. There's got to be a better way!

finite-memory machine

That brings us to the topic of finite-memory machines. The output required of the combination-lock machine can always be determine from the current and previous seven inputs. Formally, a *finite-memory machine* is one whose output is completely determined by its current input and its inputs and outputs during the previous *n* clock ticks, where *n* is a finite, bounded integer. The combination lock clearly falls into this category. Figure 12-4 shows a generalized realization of a finite-memory machine with one input and one output.

To realize the combination lock as a finite-memory machine, we need to provide a 7-bit memory for the last seven input values on X, and then we can determine the UNLK output by comparing the stored values with the combina-

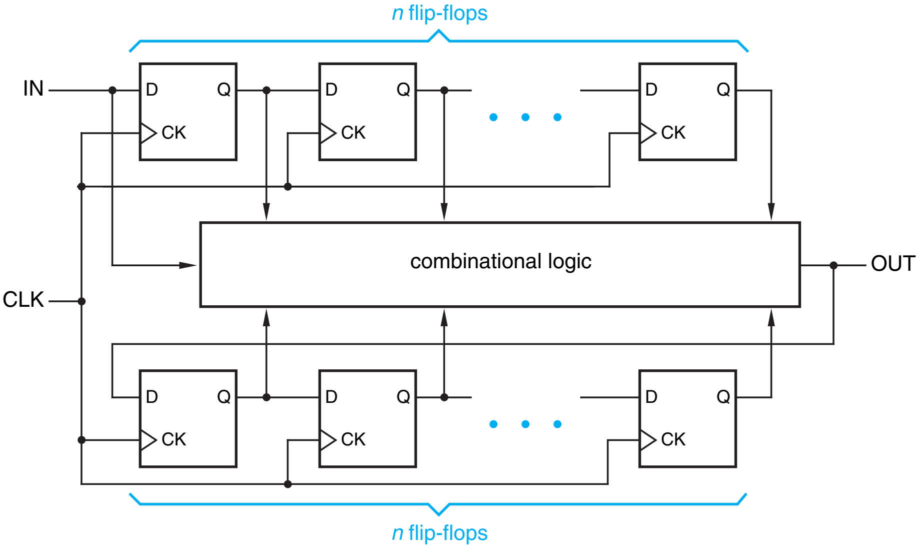


Figure 12-4
A finite-memory machine with one input and one output.

tion and checking that X is 0; the previous *output* values aren't needed in this example. Program 12-14 shows the Verilog module for this approach.

With the finite-memory approach, the state-machine design got a lot easier. There is no next-state logic at all; it is effectively subsumed in the definition of the "XHISTORY" register. And the output logic merely compares XHISTORY with the unlock combination and tests that the current value of X is 0. Note that this works fine even if the combination is a variable stored in a register or arriving on its own input signals.

That finally brings us back to the question of how to calculate the HINT output. After being able to use such a nice, universal approach for UNLK, it wouldn't be very satisfying to design HINT logic that works only for the one particular combination value that we defined in the module. Moreover, we would like to avoid any error-prone code constructs that require us to analyze the combination bit-by-bit, as in the next-state logic in our original Program 12-13.

The solution is shown in the last always block in Program 12-14, which contains a long chain of nested if-else statements. The chain begins by checking whether the 7-bit history of X matches the unlock combination; if so, HINT is asserted if X equals 0 as required. Otherwise, we check whether the last 6 bits received (the rightmost bits of XHISTORY) match the initial 6 bits of the

Program 12-14 Finite-memory realization of the combination-lock machine.

```
module VrcomblockFM( CLOCK, RESET, X, UNLK, HINT );
  input CLOCK, RESET, X;
  output wire UNLK;          // declared as wire for continuous assignment
  output reg HINT;           // declared as reg for always block
  reg [7:1] XHISTORY; // 7-tick history of X
  parameter [7:1] COMBINATION = 7'b0110111;

  always @ (posedge CLOCK)   // State memory (with sync. reset)
    if (RESET==1) XHISTORY <= 7'b1111111; // all-1s so no phantom initial 0 at reset
    else XHISTORY <= {XHISTORY[6:1], X};  // save the last 6 and new X

  // Output logic -- Detect combination pattern and 0 input
  assign UNLK = ( (XHISTORY==COMBINATION) && (X==0) ) ? 1 : 0;

  // Output logic -- Determine hint
  always @ (XHISTORY or X)
    if (XHISTORY[7:1]==COMBINATION) HINT = (X==0);
    else if (XHISTORY[6:1]==COMBINATION[7:2]) HINT = (X==COMBINATION[1]);
    else if (XHISTORY[5:1]==COMBINATION[7:3]) HINT = (X==COMBINATION[2]);
    else if (XHISTORY[4:1]==COMBINATION[7:4]) HINT = (X==COMBINATION[3]);
    else if (XHISTORY[3:1]==COMBINATION[7:5]) HINT = (X==COMBINATION[4]);
    else if (XHISTORY[2:1]==COMBINATION[7:6]) HINT = (X==COMBINATION[5]);
    else if (XHISTORY[1]==COMBINATION[7])     HINT = (X==COMBINATION[6]);
    else HINT = (X==COMBINATION[7]);
endmodule
```

IS IT WORTH IT? A disadvantage of finite-memory state-machine design is that it will almost always require more state memory than a design with customized, optimized state meanings. When I targeted Program 12-13 to a Xilinx 7-series FPGA, the Vivado tools synthesized a circuit using 2 LUTs and 3 registers for the state memory. When I synthesized Program 12-14, the much easier and less error-prone finite-memory design, the result used 3 LUTs and 7 registers.

Are the extra resources worth it? In my book, yes—saving engineering time always trumps saving transistors in one-of-a-kind circuits (as opposed to large, repetitive structures like memory arrays). This is even more true when you consider that the 7-series FPGA that I was targeting contains 53,200 LUTs and106,400 registers!

combination; if so, the user moves closer to unlocking only if `X` equals the next combination bit to the right. We continue in this fashion, checking fewer and fewer rightmost bits of `XHISTORY` and initial bits of the combination, in each case setting `HINT` equal to the combination bit just to the right of the matched part. If there's no match at all, the last `else` statement asserts `HINT` only if `X` equals the very first bit of the combination.

The priority order of the nested `if-else` statements is important—it ensures that the machine gives the best possible hint. For example, if `XHISTORY` is `7'b1011011`, then based on the rightmost three bits it could suggest a 0 input to drive the machine to the final state in three more ticks after that (with further inputs 111). But it's better to suggest a 1 input based on the rightmost six bits, and arrive in the final state on the very next tick. Related to this, the troublesome design problem for Program 12-13, determining when to backtrack only partway, is nonexistent in this approach because of the priority order. Another advantage of this approach is that its output logic for `HINT`, like its output logic for `UNLK`, works even if the unlock combination is a variable.

Because we now have two quite different designs for the combination-lock machine, they are good candidates for testing with a test bench that compares their outputs for a long, random input sequence. See Exercises 12.38 and 12.39.

12.5 T-Bird Tail Lights

We described the function of the T-bird tail lights state machine in Section 9.4.1. Just for fun, Figure 12-5 repeats our drawing of 1965 Ford Thunderbird's tail lights. The tail lights state machine has three inputs requesting left, right, and hazard signals, and two sets of three outputs to illuminate the three signal lamps on each side of the car in sequence shown in Figure 12-6.

Several steps are needed to design this or any state machine in Verilog:

1. Determine the inputs and outputs of the machine. We've already done that above, working from its informal specification.

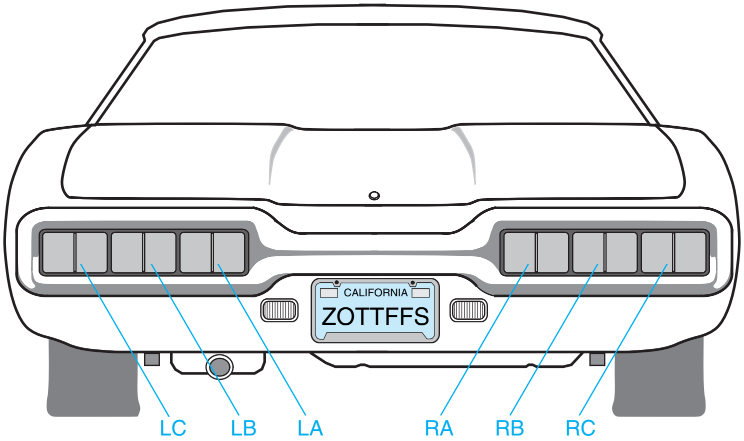


Figure 12-5
T-bird tail lights.

2. Define a set of states needed to implement the machine, naming each state as we go along. We may not be able immediately to think of all the states that are needed, but as we work through the next-state conditions for the states we may recognize the need to define additional states.

   For T-bird tail lights, we need a state where all the lights are off (call it IDLE), and then we'll need three states where one to three lights are on for left and right turns, respectively (call them L1–L3 and R1–R3). For most state machines and environments, the best way to document the names and meanings of the states is in the code itself.

3. Choose or define an additional state to serve as the initial state after a reset. For T-bird tail lights, IDLE works well as the initial state.

4. In our preferred style of behavioral Verilog state-machine coding, create the outline of a `case` statement, where the "selection expression" is simply the state register, and each "choice" is a named state. For each named state,

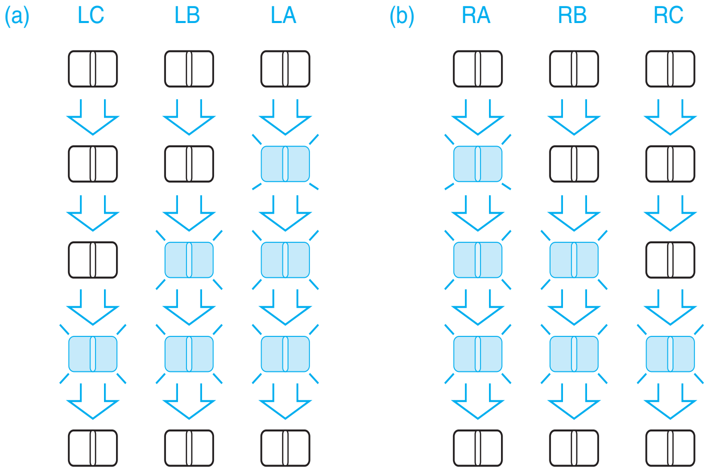


Figure 12-6
Flashing sequence for T-bird tail lights:
(a) left turn;
(b) right turn.

write a statement that specifies the next state as a function of the state machine's inputs. In our preferred coding style, this would be a possibly nested `if-else` statement, always with a final `else` clause, so a value is assigned to the next-state register for all possible input combinations.

5. Interactively in step 4, define and name additional states as needed to handle unanticipated situations. In step 2 above, we forgot about the hazard lights case for T-bird tail lights, where the machine loops through two states instead of four, and we can create a new state LR3 to deal with this.
6. Once we have achieved closure on the next-state functions (as listed in the case-statement choices), select a next state for any "unused" states to transition to unconditionally, and add a "default" choice to the `case` statement to handle these. "Unused" states will exist in the physical machine if the number of states is not a power of 2, or if we use a sparse state encoding, like 1-hot.
7. Now that we know the number of states, we can select a state assignment. As described in Section 9.3.3, there are always many possibilities. So, whatever assignment we choose, it is best to embody it in a `parameter` declaration. That way, we can deal with states symbolically in the rest of the Verilog module, without having to edit anything but the `parameter` declaration if we decide to change the state assignment later.
8. If we haven't already, create the basic "boilerplate" for the Verilog module, including the module input/output and variable declarations.
9. Write the statement that creates the state memory, either behaviorally (an `always` block) or structurally (a component instantiation), as described previously in Section 12.1.1.
10. Write the statement that creates the output logic.

This may seem like a lot of steps, but most of them are not very difficult. The most critical is step 4, defining the next-state behavior of the machine. Putting it all together for T-bird tail lights, we can write the Verilog module in Program 12-15. The declarations are straightforward, including internal variables `Sreg` and `Snext` to hold the current and next state, respectively. A `parameter` declaration defines the 3-bit encodings of the eight states, the same ones that we used for this machine in the state-diagram and ASM-chart versions in Chapter 9.

The first `always` block creates the 3-bit state register `Sreg`, including an asynchronous reset input. The second `always` block is the heart of the machine, with a `case` statement that defines the next-state behavior for the eight states. The last `always` block also has a `case` statement, with one assignment statement per state to define the values of the six Moore-type outputs as functions of the current state.

Program 12-15 Verilog module for T-bird tail lights state machine.

```
module VrTbirdSM( CLOCK, RESET, LEFT, RIGHT, HAZ, LA, LB, LC, RA, RB, RC );
  input CLOCK, RESET, LEFT, RIGHT, HAZ;
  output reg LA, LB, LC, RA, RB, RC;
  reg [2:0] Sreg, Snext;          // State register and next state
  parameter [2:0] IDLE = 3'b000,  // Define the states and their codes
                  L1   = 3'b001,  // left turn, one light on
                  L2   = 3'b011,  // left turn, two lights on
                  L3   = 3'b010,  // left turn, three lights on
                  R1   = 3'b101,  // right turn, one light on
                  R2   = 3'b111,  // right turn, two lights on
                  R3   = 3'b110,  // right turn, three lights on
                  LR3  = 3'b100;  // hazard, all lights on

  always @ (posedge CLOCK or posedge RESET)          // Create state memory
    if (RESET==1) Sreg <= IDLE; else Sreg <= Snext;  // Async reset

  always @ (LEFT, RIGHT, HAZ, Sreg) begin            // Next-state logic
    case (Sreg)
      IDLE:   if (HAZ | (LEFT & RIGHT) ) Snext = LR3;
              else if (RIGHT)            Snext = R1;
              else if (LEFT)             Snext = L1;
              else                       Snext = IDLE;
      R1:     Snext = R2;
      R2:     Snext = R3;
      R3:     Snext = IDLE;
      L1:     Snext = L2;
      L2:     Snext = L3;
      L3:     Snext = IDLE;
      LR3:    Snext = IDLE;
      default Snext = IDLE;
    endcase
  end

  always @ (Sreg) begin                          // Output logic
    case (Sreg)
      IDLE:   {LC,LB,LA,RA,RB,RC} = 6'b000000;   // Leave them all off
      R1:     {LC,LB,LA,RA,RB,RC} = 6'b000100;   // Set up the rotating patterns
      R2:     {LC,LB,LA,RA,RB,RC} = 6'b000110;   //   for right turn
      R3:     {LC,LB,LA,RA,RB,RC} = 6'b000111;
      L1:     {LC,LB,LA,RA,RB,RC} = 6'b001000;   //   and left turn
      L2:     {LC,LB,LA,RA,RB,RC} = 6'b011000;
      L3:     {LC,LB,LA,RA,RB,RC} = 6'b111000;
      LR3:    {LC,LB,LA,RA,RB,RC} = 6'b111111;   // All-on flashing for hazard
      default {LC,LB,LA,RA,RB,RC} = 6'b000000;   // All-off in any unused states
    endcase
  end
endmodule
```

Program 12-16 Verilog next-state logic for an enhanced T-bird tail lights state machine.

```
always @ (LEFT, RIGHT, HAZ, Sreg) begin            // Next-state logic
  case (Sreg)
    IDLE:   if (HAZ | (LEFT & RIGHT) ) Snext = LR3;
            else if (RIGHT)            Snext = R1;
            else if (LEFT)             Snext = L1;
            else                       Snext = IDLE;
    R1:     if (HAZ) Snext = LR3; else Snext = R2;
    R2:     if (HAZ) Snext = LR3; else Snext = R3;
    R3:     if (HAZ) Snext = LR3; else Snext = IDLE;
    L1:     if (HAZ) Snext = LR3; else Snext = L2;
    L2:     if (HAZ) Snext = LR3; else Snext = L3;
    L3:     if (HAZ) Snext = LR3; else Snext = IDLE;
    LR3:    Snext = IDLE;
    default Snext = IDLE;
  endcase
end
```

It is fairly easy to make changes in the Verilog behavioral description of a state machine, and then re-synthesize the machine using the available tools. For example, the T-bird tail lights machine as originally designed checks the HAZ input only in the IDLE state. It would be more desirable functionally for the machine to start flashing for a hazard as soon as possible after the HAZ input is asserted. Only the next-state logic in the original machine needs to be modified to make this happen, as shown in Program 12-16. In each of the "turning" states, we now check the HAZ input; if it is asserted, we go to the hazard-flashing state LR3 next.

Program 12-17 Verilog changes for an output-coded state assignment for T-bird tail lights.

```
 module VrTbirdSMeoc( CLOCK, RESET, LEFT, RIGHT, HAZ, LA, LB, LC, RA, RB, RC );
  input CLOCK, RESET, LEFT, RIGHT, HAZ;
  output reg LA, LB, LC, RA, RB, RC;
  reg [5:0] Sreg, Snext;             // State register and next state
  parameter [5:0] IDLE = 6'b000000,  // Define the states and their codes
                  L1   = 6'b001000,  // left turn, one light on
                  L2   = 6'b011000,  // left turn, two lights on
                  L3   = 6'b111000,  // left turn, three lights on
                  R1   = 6'b000100,  // right turn, one light on
                  R2   = 6'b000110,  // right turn, two lights on
                  R3   = 6'b000111,  // right turn, three lights on
                  LR3  = 6'b111111;  // hazard, all lights on
...
  always @ (Sreg)             // Output logic
    {LC,LB,LA,RA,RB,RC} = Sreg;
endmodule
```

VIVADO FSM EXTRACTION

The first time that I ran the three versions of the T-bird tail lights machine through the Xilinx Vivado synthesis tools, I was very surprised to find that all three versions gave very similar synthesis results. It turns out that Vivado's "FSM extraction" option had been turned on by default. As explained in Section 12.1.7, this feature analyzes the Verilog module, looking for a structure that appears to be a finite-state machine (FSM). If it finds one, it throws out the designer's state encoding, and selects one according to its own ideas of what's good.

For all three T-bird tail lights examples, Vivado selected its "sequential" state assignment, using the minimum number of state bits with values assigned in binary counting order. So, Vivado's state assignments used for Programs 12-15 and 12-16, which have slightly different next-state logic, were similar but not identical 3-bit encodings. Modifying the state assignment in Program 12-16 to the one in Program 12-17 yielded an identical synthesized circuit, even though Program 12-17 explicitly calls for a 6-bit output-coded state assignment! However, once I disabled the "FSM extraction" option, the tools dutifully used the exact state assignments specified in the modules.

So, how good were Vivado's state assignments? When the state machines were targeted to Xilinx 7-series FPGAs, Programs 12-15 required 9 LUTs, and Programs 12-16 and 12-17 required 8 LUTs. However, when the designer's (my) original state assignments were used, Program 12-15 used only 4 LUTs, and Program 12-16 used 5. So for now, experienced designers can still do a better job than the tools—yeah!

Another possible change would be to change the state assignment to an output-coded one. This requires no change to the next-state logic, but instead to the state-register and `parameter` declarations, and to the output logic, as shown in Program 12-17.

Because T-bird tail lights are so "visual," the state machine is an excellent one to check with a test bench that displays the LED patterns that it produces for a typical input sequence. (See Exercise 12.23.)

12.6 Reinventing Traffic-Light Controllers

Our next example is also from the world of driving. Traffic-light controllers in California, especially in the city of Sunnyvale, are designed to *maximize* the waiting time of cars at intersections. An infrequently used intersection (one that might have only a "yield" sign if it were in Chicago) has the sensors and signals shown in Figure 12-7. The lights are controlled by a state machine operating with a 1-Hz clock and whose inputs are the sensors and two signals from a timer:

NSCAR This sensor output is asserted when a car on the north-south road is over either sensor on either side of the intersection.

EWCAR This sensor output is asserted when a car on the east-west road is over either sensor on either side of the intersection.

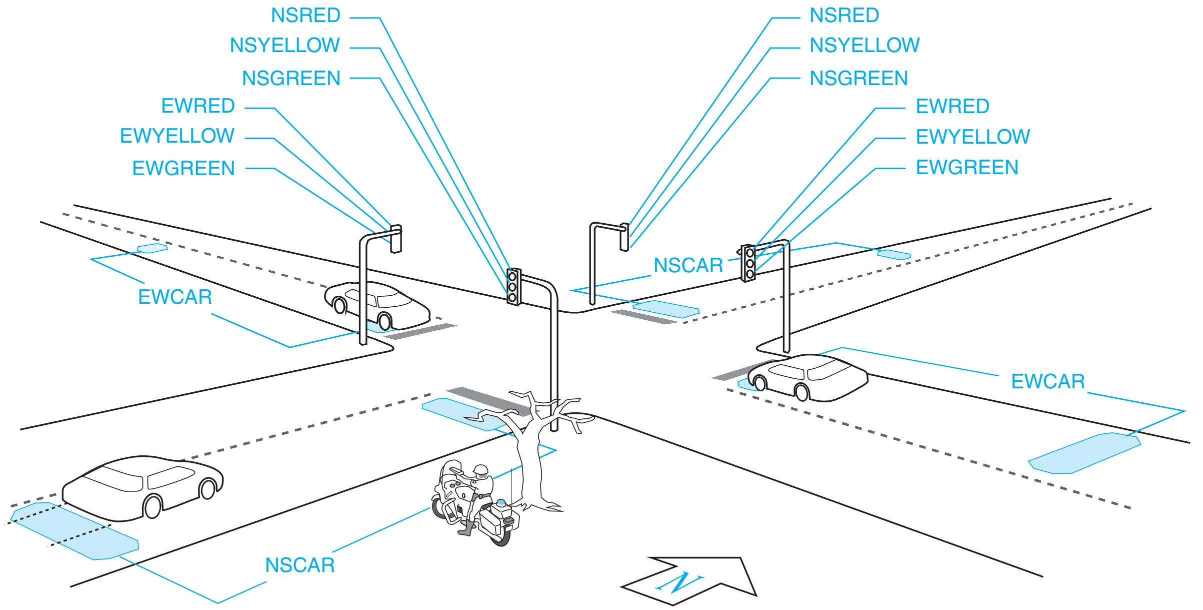


Figure 12-7 Traffic sensors and signals at an intersection in Sunnyvale, California.

TMLONG This timer output is asserted if more than five minutes has elapsed since the timer started; it remains asserted until the timer is reset.

TMSHORT This timer output is asserted if more than five seconds has elapsed since the timer started; it remains asserted until the timer is reset.

The state machine has seven outputs:

NSRED, NSYELLOW, NSGREEN Control the north-south lights.

EWRED, EWYELLOW, EWGREEN Control the east-west lights.

TMRESET When asserted, resets the timer and negates TMSHORT and TMLONG. The timer starts timing when TMRESET is negated.

A typical, municipally approved algorithm for controlling the traffic lights is embedded in the Verilog module of Program 12-18. This algorithm produces two frequently seen behaviors of "smart" traffic lights. At night, when traffic is light, it holds a car stopped at the light for up to five minutes, unless a car approaches on the cross street, in which case it stops the cross traffic and finally lets the waiting car go. (The "early warning" sensor is far enough back to change the lights before the approaching car reaches the intersection.) During the day, when traffic is heavy and there are always cars waiting in both directions, it cycles the lights every five seconds, thus minimizing the utilization of the intersection and maximizing everyone's waiting time, thereby creating a public demand for more taxes to fix the problem.

Program 12-18 Verilog module for Sunnyvale traffic-light controller.

```
module Vrsvale ( CLOCK, RESET, NSCAR, EWCAR, TMSHORT, TMLONG,
                 OVERRIDE, FLASHCLK, NSRED, NSYELLOW, NSGREEN,
                 EWRED, EWYELLOW, EWGREEN, TMRESET );
  input CLOCK, RESET, NSCAR, EWCAR, TMSHORT, TMLONG, OVERRIDE, FLASHCLK;
  output NSRED, NSYELLOW, NSGREEN, EWRED, EWYELLOW, EWGREEN, TMRESET;
  reg [2:0] Sreg, Snext;       // State register and next state
                               // State encodings
  parameter NSGO    = 3'b000, NSWAIT  = 3'b001, NSWAIT2 = 3'b010, NSDELAY = 3'b011,
            EWGO    = 3'b100, EWWAIT  = 3'b101, EWWAIT2 = 3'b110, EWDELAY = 3'b111;

  always @ (posedge CLOCK)           // Create state memory with sync reset
    if (RESET) Sreg <= NSDELAY; else Sreg <= Snext;

  always @ (*)                                    // Next-state logic.
    case (Sreg)
      NSGO :                                      // North-south green.
        if      (~TMSHORT)        Snext = NSGO;   // Minimum 5 seconds.
        else if (TMLONG)          Snext = NSWAIT; // Maximum 5 minutes.
        else if ( EWCAR & ~NSCAR) Snext = NSGO;   // Make EW car wait.
        else if ( EWCAR &  NSCAR) Snext = NSWAIT; // Thrash if cars both ways.
        else if (~EWCAR &  NSCAR) Snext = NSWAIT; // New NS car? Make it stop!
        else                      Snext = NSGO;   // No one coming, stay as is.
      NSWAIT  : Snext = NSWAIT2;                  // Yellow light,
      NSWAIT2 : Snext = NSDELAY;                  //   two ticks for safety.
      NSDELAY : Snext = EWGO;                     // Red both ways for safety.
      EWGO    :                                   // East-west green.
        if      (~TMSHORT)        Snext = EWGO;   // Same behavior as above.
        else if (TMLONG)          Snext = EWWAIT;
        else if ( NSCAR & ~EWCAR) Snext = EWGO;
        else if ( NSCAR &  EWCAR) Snext = EWWAIT;
        else if (~NSCAR &  EWCAR) Snext = EWWAIT;
        else                      Snext = EWGO;
      EWWAIT  : Snext = EWWAIT2;
      EWWAIT2 : Snext = EWDELAY;
      EWDELAY : Snext = NSGO;
      default : Snext = NSDELAY;                  // "Reset" state.
    endcase

  assign TMRESET  = (Sreg==NSWAIT2 || Sreg==EWWAIT2);
  assign NSRED    = (OVERRIDE) ?  FLASHCLK  :
                         (Sreg!=NSGO && Sreg!=NSWAIT && Sreg!=NSWAIT2);
  assign NSYELLOW = (OVERRIDE) ?  0  : (Sreg==NSWAIT || Sreg==NSWAIT2);
  assign NSGREEN  = (OVERRIDE) ?  0  : (Sreg==NSGO);
  assign EWRED    = (OVERRIDE) ?  FLASHCLK  :
                         (Sreg!=EWGO && Sreg!=EWWAIT && Sreg!=EWWAIT2);
  assign EWYELLOW = (OVERRIDE) ?  0  : (Sreg==EWWAIT || Sreg==EWWAIT2);
  assign EWGREEN  = (OVERRIDE) ?  0  : (Sreg==EWGO);
endmodule
```

The next-state logic in Program 12-18 has our typical style of using a `case` statement to specify the behavior for each state, including `if-else` statements as needed to check input dependencies. Regarding the output logic, this is a Moore machine; each output signal is a function of state only. However, instead of a `case` statement, the output logic in this example uses continuous assignments. It was easier to code the lights' operation by thinking about each light separately and writing an expression that is asserted for the states in which that light should be illuminated. Still, rewriting the output logic with a `case` statement is possible and is offered as Exercise 12.30.

Program 12-18 uses a simple binary encoding of states. An output-coded state assignment can be made with the changes shown in Program 12-19. Many of the states can be identified by a unique combination of light-output values. But there are three pairs of states that are not distinguishable by looking at the lights alone: (`NSWAIT`, `NSWAIT2`), (`EWWAIT`, `EWWAIT2`), and (`NSDELAY`, `EWDELAY`). We can handle these by adding one more state variable, `Sreg[7]` or "`EXTRA`", that has different values for the two states in each pair. Thus, `NSWAIT` and `NSWAIT2`, for example, have the same state encoding and light-output values for bits 1–6 but differ in bit 7.

One possible variation on the state-machine specification is to provide an OVERRIDE input that the police can use to disable normal controller operation and put the lights into an all-flashing-red mode as long as this input is asserted. This allows them to manually clear up the traffic snarls created by this wonderful invention. The enhancement is pretty easy to provide with the output-coded state assignment, since the all-0s (all lights off) state is still available for use. As shown in Program 12-20, we make three changes:

- Add the OVERRIDE input to the module.
- Define a new state, `ALLOFF`, with the all-0s state coding.

Program 12-19 Changes to Sunnyvale traffic-lights machine for output-coded state assignment.

```
  reg [1:7] Sreg, Snext;       // State register and next state
  // bit positions of output-coded assignment: [1] NSRED, [2] NSYELLOW, [3] NSGREEN,
  //                                [4] EWRED, [5] EWYELLOW, [6] EWGREEN, [7] (EXTRA)
  parameter NSGO    = 7'b0011000,  // State encodings
            NSWAIT  = 7'b0101000,
            NSWAIT2 = 7'b0101001,
            NSDELAY = 7'b1001000,
            EWGO    = 7'b1000010,
            EWWAIT  = 7'b1000100,
            EWWAIT2 = 7'b1000101,
            EWDELAY = 7'b1001001;
...
                                   // Output logic.
  assign TMRESET  = (Sreg==NSWAIT2 || Sreg==EWWAIT2);
  assign {NSRED, NSYELLOW, NSGREEN, EWRED, EWYELLOW, EWGREEN} = Sreg[1:6];
```

Program 12-20 Changes to add OVERRIDE to the traffic-lights machine with output-coded states.

```
module Vrsvaleocov ( CLOCK, RESET, NSCAR, EWCAR, TMSHORT, TMLONG, OVERRIDE,
                   NSRED, NSYELLOW, NSGREEN, EWRED, EWYELLOW, EWGREEN, TMRESET );
  input CLOCK, RESET, NSCAR, EWCAR, TMSHORT, TMLONG, OVERRIDE;
  ...
  parameter NSGO    = 7'b0011000,  // State encodings
            ...
            EWDELAY = 7'b1001001,
            ALLOFF  = 7'b0000000;
  ...
  always @ (*)                                    // Next-state logic.
  if (OVERRIDE)                                   // All-red flashing if OVERRIDE
    case (Sreg)
      ALLOFF  : Snext = NSDELAY;                  // Double red.
      NSDELAY : Snext = ALLOFF;
      default : Snext = NSDELAY;                  // Come here at start of override.
    endcase
  else                                            // Normal operation
    case (Sreg)
      NSGO :    ...                               // North-south green....
```

- Place the original next-state case statement in the else clause of an if-else statement that executes it if OVERRIDE is negated, and executes a new case statement for a flashing cycle as long as OVERRIDE is asserted.

No changes are needed to the output logic since the new ALLOFF state has just the right output values needed for all-lights-off part of the flashing cycle, and we use an existing state NSDELAY for the all-red part.

As a "visual" application, Sunnyvale traffic lights are another candidate for writing a test bench that simply applies a comprehensive set of inputs and shows the resulting light patterns (see Exercise 12.32). A more ambitious and perhaps depressing test bench would create a random pattern of traffic arrivals and calculate performance metrics like average throughput and waiting time at the intersection (see Exercise 12.33).

MOUNTAIN VIEW TRAFFIC LIGHTS

In March of 2016, it was reported that Google donated $250,000 to Sunnyvale to update the software for their traffic lights. I guess that being called out for 25 years in every previous edition of this book wasn't enough of an incentive for the city to upgrade on their own nickel.

Google is headquartered, of course, in the neighboring city of Mountain View, whose traffic lights had good behavior even before Google came along. Sunnyvale could have achieved the same, and probably cheaply too, simply by replacing the NSGO and EWGO next-state logic in their equivalent of Program 12-18 with something closer to a traditional algorithm, as requested in Exercise 12.31.

12.7 The Guessing Game

Another state-machine example is a “guessing game” that can be built as an amusing lab project:

> Design a state machine with four inputs, G1–G4, that are connected to pushbuttons. The machine has four outputs, L1–L4, connected to LEDs located near the like-numbered pushbuttons. There is also an ERR output connected to a red LED. In normal operation the L1–L4 outputs display a 1-out-of-4 pattern. At each clock tick, the pattern is rotated by one position; the clock frequency is about 4 Hz.
>
> Guesses are made by pressing and holding a debounced pushbutton, which asserts an input Gi. When any Gi input is asserted, the ERR output is asserted if the “wrong” pushbutton was pressed, that is, if the Gi input detected at the clock tick does not have the same number as the LED output that was asserted before the clock tick. Once a guess has been made, play stops and the ERR output maintains the same value for one or more clock ticks until the Gi input is negated, then play resumes.

Clearly, we will have to provide at least four states, one for each position of the rotating pattern; let’s call them S1–S4. And we’ll need at least one more state to indicate that play has stopped. A first cut at possible next-state logic is shown in Program 12-21. The machine cycles through states S1–S4 as long as no Gi input is asserted, and it goes to the STOP state when a guess is made. Each Li output would be asserted in the like-numbered state.

The only problem with the next-state behavior in Program 12-21 is that it doesn’t “remember” in the STOP state whether the guess was correct, so it has no way to control the ERR output. This problem is fixed in the complete Verilog module in Program 12-22, which has two “stopped” states, SOK and SERR. On an incorrect guess, the machine goes to SERR, where ERR is asserted; otherwise, it goes to SOK. Three state variables encode the six states. Although the machine’s word description doesn’t require it, the new next-state logic is designed to go to SERR even if the user tries to cheat the machine by pressing two or more pushbuttons at a time, or by changing guesses while stopped.

Program 12-21 First cut at Verilog next-state logic for the guessing game.

```
always @ (*)    // Next-state logic
   case (Sreg)
     S1  : if (G1 | G2 | G3 | G4) Snext = STOP; else Snext = S2;
     S2  : if (G1 | G2 | G3 | G4) Snext = STOP; else Snext = S3;
     S3  : if (G1 | G2 | G3 | G4) Snext = STOP; else Snext = S4;
     S4  : if (G1 | G2 | G3 | G4) Snext = STOP; else Snext = S1;
     STOP: if (~G1 & ~G2 & ~G3 & ~G4) Snext = S1; else Snext = STOP;
   endcase
```

The output logic in Program 12-22 simply decodes the current state and asserts the appropriate Moore-type output. Since a different output combination is produced in each named state, we can use the outputs as state variables in an output-coded state assignment. In an output-coded version of the state machine, we make `Sreg` and `Snext` 5 bits wide, change the `parameter` declaration, and change the output logic as shown in Program 12-23.

Program 12-22 Corrected and completed Verilog module for the guessing game.

```
module Vrggame ( CLOCK, RESET, G1, G2, G3, G4, L1, L2, L3, L4, ERR );
  input CLOCK, RESET, G1, G2, G3, G4;
  output reg L1, L2, L3, L4, ERR;
  reg [2:0] Sreg, Snext;    // State register and next state

  parameter S1   = 3'b001,  // State encodings for 4 running states,
            S2   = 3'b010,
            S3   = 3'b011,
            S4   = 3'b100,
            SOK  = 3'b101,  // OK state, and
            SERR = 3'b110;  // error state

  always @ (posedge CLOCK)  // Create state memory with sync reset
    if (RESET) Sreg <= SOK; else Sreg <= Snext;

  always @ (G1 or G2 or G3 or G4 or Sreg)   // Next-state logic
    case (Sreg)
      S1   : if (G2 | G3 | G4) Snext = SERR;
             else if (G1)      Snext = SOK;
             else              Snext = S2;
      S2   : if (G1 | G3 | G4) Snext = SERR;
             else if (G2)      Snext = SOK;
             else              Snext = S3;
      S3   : if (G1 | G2 | G4) Snext = SERR;
             else if (G3)      Snext = SOK;
             else              Snext = S4;
      S4   : if (G1 | G2 | G3) Snext = SERR;
             else if (G4)      Snext = SOK;
             else              Snext = S1;
      SOK : if (~G1 & ~G2 & ~G3 & ~G4) Snext = S1; else Snext = SOK;
      SERR: if (~G1 & ~G2 & ~G3 & ~G4) Snext = S1; else Snext = SERR;
      default : Snext = SOK;
    endcase

  always @ (Sreg) begin      // Output logic
    L1  = (Sreg == S1); L2  = (Sreg == S2); L3  = (Sreg == S3);
    L4  = (Sreg == S4); ERR = (Sreg == SERR);
  end

endmodule
```

Program 12-23 Verilog changes for output-coded guessing-game state machine.

```
module Vrggameoc ( CLOCK, RESET, G1, G2, G3, G4, L1, L2, L3, L4, ERR );
  input CLOCK, RESET, G1, G2, G3, G4;
  output wire L1, L2, L3, L4, ERR;
  reg [4:0] Sreg, Snext;      // State register and next state
  parameter S1   = 5'b10000,  // Output coded state assignment
            S2   = 5'b01000,  //  for 4 running states,
            S3   = 5'b00100,
            S4   = 5'b00010,
            SOK  = 5'b00000,  // OK state, and
            SERR = 5'b00001;  // error state
...
  assign {L1,L2,L3,L4,ERR} = Sreg;   // Output logic
endmodule
```

GETTING AN EDGE

The main advantage of an output-coded state assignment is that the Moore-type outputs become valid almost immediately after the triggering clock edge, without the combinational logic delay that would otherwise be required to derive the them from the new contents of the state memory. Depending on the state machine, output coding may require more logic resources or fewer.

For example, when I targeted Program 12-22 to a Xilinx 7-series FPGA, the Vivado tools synthesized a circuit using 12 LUTs and three registers for the state memory. When I synthesized Program 12-23, the output-coded version, two more registers were used but the number of LUTs was reduced to 9—a slight net change in resource requirements. Moreover, the output-coded version is really great for players who need the benefit of a few saved picoseconds of LED delay when playing the game :) .

A self-checking test bench for the guessing-game state machine is shown in Program 12-24. It steps through a sequence of inputs that was constructed by hand to check the machine's operation for various scenarios. At each step, it invokes a task `checkLEDs` whose inputs are a step number and the expected values of the LED outputs at that step. If an unexpected LED value occurs, it is displayed and the test bench stops so the user can investigate. This test bench can of course be used with either version of the machine, Program 12-22 or 12-23.

The input sequence in Program 12-24 is far from comprehensive. It doesn't even begin to check the machine's operation with the G3 and G4 inputs. As you can imagine, it is fairly painstaking to create an input sequence that will exercise all possibilities, including specifying the expected outputs each time `checkLEDs` is invoked. Therefore, this state machine is a good candidate for a test bench that generates random inputs and checks the resulting outputs algorithmically. We'll explore that approach for another version of the machine in Section 12.9.1.

Program 12-24 Test bench for the guessing game.

```
`timescale 1ns/1ns
module Vrggame_tb ();
reg Tclk, RST, G1, G2, G3, G4;
wire L1, L2, L3, L4, ERR;

Vrggame UUT ( .CLOCK(Tclk), .RESET(RST), .G1(G1), .G2(G2), .G3(G3), .G4(G4),
              .L1(L1), .L2(L2), .L3(L3), .L4(L4), .ERR(ERR) );

task checkLEDs;
  input stepnum, expL1, expL2, expL3, expL4, expERR;
  integer stepnum; reg expL1, expL2, expL3, expL4, expERR;
  begin
    if ( {L1, L2, L3, L4, ERR} != { expL1, expL2, expL3, expL4, expERR} ) begin
      $display($time," Error, step %d, expected %5b, got %5b", stepnum,
         { expL1, expL2, expL3, expL4, expERR}, {L1, L2, L3, L4, ERR} );
      $stop(1); end
  end
endtask

always begin    // create free-running test clock with 10 ns period
  #5 Tclk = 0;  // 5 ns high
  #5 Tclk = 1;  // 5 ns low
end

initial begin
  RST = 1;       // Apply reset
  {G1,G2,G3,G4} = 4'b0000;  // All guess inputs are 0
  Tclk = 1;      // Start clock at 1 at time 0
  #115;          // Wait 115 ns
  checkLEDs(1,0,0,0,0,0);   // Expect all LEDs off at reset
  RST = 0;       // unreset
  #10;          checkLEDs( 2,1,0,0,0,0);  // LEDs rotate after un-reset
  #10;          checkLEDs( 3,0,1,0,0,0);  // No guesses yet, just check rotation
  #10;          checkLEDs( 4,0,0,1,0,0);
  #10;          checkLEDs( 5,0,0,0,1,0);
  #10;          checkLEDs( 6,1,0,0,0,0);  // OK, made it back to L1
  G1 = 1; #10;  checkLEDs( 7,0,0,0,0,0);  // Should be correct guess
  #10;          checkLEDs( 8,0,0,0,0,0);  // Stay here as long as G1 still on
  G1 = 0; #10;  checkLEDs( 9,1,0,0,0,0);  // Release G1 and go; start with L1 again
  G2 = 1; #10;  checkLEDs(10,0,0,0,0,1);  // Make a wrong guess, should get ERR on
  #10;          checkLEDs(11,0,0,0,0,1);  // Stay here as long as G2 still on
  #10;          checkLEDs(12,0,0,0,0,1);  // Stay here as long as G2 still on
  G2 = 0; #10;  checkLEDs(13,1,0,0,0,0);  // Release G2 and go; start with L1 again
  G1 = 1;                                 // Try to fool it pressing multiple buttons
  G2 = 1; #10;  checkLEDs(14,0,0,0,0,1);  // Should get ERR on
  $stop(1);                               // end test
end
endmodule
```

*12.8 "Don't-Care" State Encodings

This is a good place to introduce the idea of "don't-care" state codings. Out of the 32 possible coded states using five variables, only six are explicitly used in Program 12-23 in Section 12.7. The rest of the states are unused and because of the `default` case in the next-state logic, they have a next state of `SOK`, or 00000. Another possible disposition for unused states, one that we haven't explored before, is obtained by careful use of "don't-cares" in the coding of current state (`Sreg` value) in the next-state logic's `case` statement.

Table 12-2 shows such a state encoding for the guessing-game machine, derived from the output-coded state assignment used in Program 12-23. In this encoding, every one of the 32 possible values of the current state in `Sreg` is matched by exactly one of the "don't-care" coded states (e.g., 10111 = xS1, 00101 = xS3). However, *next states* are coded using the same unique values as in Program 12-23.

The "don't-care" coded current states can be used in the Verilog module as shown in Program 12-25. The next states are defined by a `parameter` declaration as before, but the current states are defined by a second `parameter` declaration that uses `?`'s in some of bit positions, as in Table 12-2. Recall that in a Verilog literal, "?" means the same as "z" ("high impedance"), but depending on the tool suite, it may also be interpreted as "don't-care." In such tool suites, the compiler and synthesizer can use "don't-cares" to simplify combinational logic.

The other important change in Program 12-25 is in the next-state logic, replacing replace the `case` keyword with `casez`. A `casez` statement allows `z`'s (or equivalent `?`'s) to appear in case choices—as they do in the "z" current states defined in the second `parameter` declaration—and alerts the compiler and synthesizer to treat the `z`'s as "don't-cares" when creating the case logic.

In this approach, each unused current state behaves like a nearby "normal" state; Figure 12-8 illustrates the concept. The machine is well-behaved and goes to a "normal" state if it inadvertently enters an unused state. Yet the approach still allows some simplification of the next-state logic. When Program 12-25 was targeted to a Xilinx 7-series FPGA using Vivado tools, the synthesized

Table 12-2
Current-state encoding for the guessing-game machine using "don't-cares."

State	*L1*	*L2*	*L3*	*L4*	*ERR*
xS1	1	x	x	x	x
xS2	0	1	x	x	x
xS3	0	0	1	x	x
xS4	0	0	0	1	x
xSOK	0	0	0	0	0
xSERR	0	0	0	0	1

Program 12-25 Verilog module for the guessing game using “don’t-care” current-state encodings.

```
module Vrggamedc ( CLOCK, RESET, G1, G2, G3, G4, L1, L2, L3, L4, ERR );
  input CLOCK, RESET, G1, G2, G3, G4;
  output wire L1, L2, L3, L4, ERR;
  reg [1:5] Sreg, Snext;       // State register and next state
  parameter S1    = 5'b10000,  // Output coded state assignment
            S2    = 5'b01000,  //   for 4 running states,
            S3    = 5'b00100,
            S4    = 5'b00010,
            SOK   = 5'b00000,  //   OK state, and
            SERR  = 5'b00001;  //   error state
  parameter zS1   = 5'b1????,  // State coding for cases, with don't-cares
            zS2   = 5'b01???,  //   for 4 running states,
            zS3   = 5'b001??,
            zS4   = 5'b0001?,
            zSOK  = 5'b00000,  //   OK state, and
            zSERR = 5'b00001;  //   error state
  always @ (posedge CLOCK)     // Create state memory with sync reset
    if (RESET) Sreg <= SOK; else Sreg <= Snext;
  always @ (G1 or G2 or G3 or G4 or Sreg)        // Next-state logic
    casez (Sreg)
      zS1  : if (G2 | G3 | G4) Snext = SERR;
             else if (G1)      Snext = SOK;
             else              Snext = S2;
      zS2  : if (G1 | G3 | G4) Snext = SERR;
             else if (G2)      Snext = SOK;
             else              Snext = S3;
      zS3  : if (G1 | G2 | G4) Snext = SERR;
             else if (G3)      Snext = SOK;
             else              Snext = S4;
      zS4  : if (G1 | G2 | G3) Snext = SERR;
             else if (G4)      Snext = SOK;
             else              Snext = S1;
      zSOK : if (~G1 & ~G2 & ~G3 & ~G4) Snext = S1; else Snext = SOK;
      zSERR: if (~G1 & ~G2 & ~G3 & ~G4) Snext = S1; else Snext = SERR;
      default : Snext = SOK;
    endcase
  assign {L1,L2,L3,L4,ERR} = Sreg;   // Output logic
endmodule
```

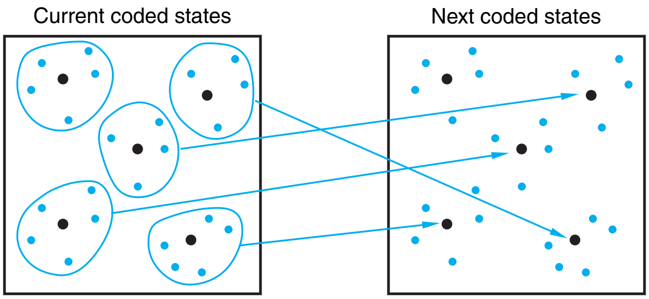


Figure 12-8
State assignment using “don’t-cares” for current states.

DON'T MAKE A CASE OUT OF IT

It's important to use `casez` and not `case` in Program 12-25. Using "`case`" is not an error as far as the Verilog compiler is concerned; it does allow `z`'s to be used in choices in a `case` statement. The simulator correctly considers them to be high-impedance values, and it creates unknown outputs when the choices are executed. Unable to generate unknown ("`x`") outputs in a real circuit, the synthesizer simply creates a circuit with *no* logic corresponding to these choices. It may give you a warning, but it does not stop synthesis or flag this as a critical error.

machine required only 7 LUTs, comparing favorably with 9 LUTs needed for the original Program 12-23.

Still, this approach should be used with great care. First, you must be sure that the "don't-care" state encodings are defined properly, in particular that they are mutually exclusive and all inclusive. If they are not all inclusive and you don't specify a `default` case, then the tools will create "inferred latches" to hold the previous choice for cases where there's no current match, even if those cases should never occur in practice. And if they are not mutually exclusive, you have a "nonparallel case statement" where the tools will create priority logic to ensure that only the first matched case is executed, again even if multiple cases can never be matched in practice because they are unused. Second, in a larger design, if something goes wrong elsewhere in your design and real errors (unexpected `x`'s or `z`'s) are propagated to your "don't-care" case logic, the simulator may mask them, and the simulated behavior may not match the synthesized behavior.

12.9 Decomposing State Machines

Just like large procedures and functions in a programming language, large state machines are difficult to conceptualize, design, and debug. Therefore, when faced with a large state-machine problem, digital designers often look for opportunities to solve it with a collection of smaller state machines.

state-machine decomposition

There's a well-developed theory of *state-machine decomposition* that you can use to analyze any given, monolithic state machine to determine whether it can be realized as a collection of smaller ones. However, decomposition theory is not too useful for designers who want to avoid designing large state machines in the first place. Rather, a practical designer tries to cast the original design problem into a natural, hierarchical structure, so the uses and functions of sub-

REST IN PEACE

Because of this subsection's title, I got some complaints in previous editions for joking about the "buried flip-flops" found in some PLDs. So I'm leaving that out.

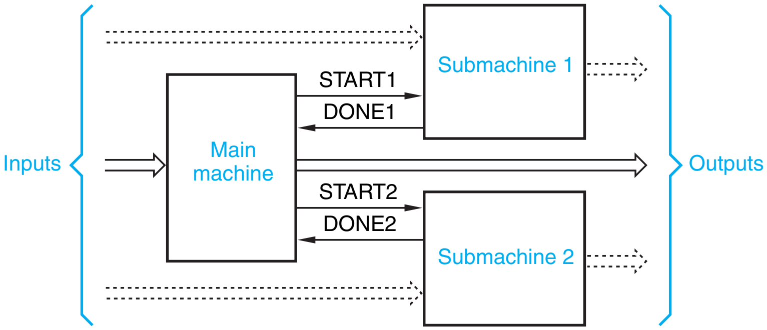


Figure 12-9
A typical, hierarchical state-machine structure.

machines are obvious, making it unnecessary ever to write a state table, state diagram, or HDL model for the equivalent monolithic machine.

The simplest and most commonly used type of decomposition is illustrated in Figure 12-9. A *main machine* typically handles the primary inputs and outputs and executes a top-level control algorithm. *Submachines* perform low-level steps under the control of the main machine and may optionally handle some of the primary inputs and outputs.

main machine

submachines

The hierarchical structure provided by Verilog is ideal for decomposing large state machines. It's easy to define the inputs and outputs used by the main machine and submachines to communicate. Moreover, it is possible to "stub out" the submachines, designing the main machine without coding the details of the submachine until later. This is particularly useful since the designer's initial ideas about the functions of and communication with the submachines may change as more details of the main machine are worked out. Finally, the main machine and submachines may use different coding styles; for example, behavioral for the main machine and structural for a submachine where an existing library component will do the job.

Perhaps the most commonly used submachine is a counter. The main machine starts the counter when it needs to stay in a particular main state for n clock ticks, and the counter asserts a DONE signal when n ticks have occurred. The main machine is typically designed to wait in the same state until DONE is asserted. This adds an extra output and input to the main machine (START and DONE), but it saves $n - 1$ states.

12.9.1 The Guessing Game Again

An example decomposed state machine designed along the lines discussed above is based on the guessing game of Section 12.7. The original guessing game is easy to win after a minute of practice because the LEDs cycle at a very consistent rate of 4 Hz. To make the game more challenging, we can greatly increase the clock speed to, say, 50 Hz, but program the LEDs to stay in each state for a random length of time. Then the user truly must guess whether a given LED will stay on long enough for the corresponding pushbutton to be pressed.

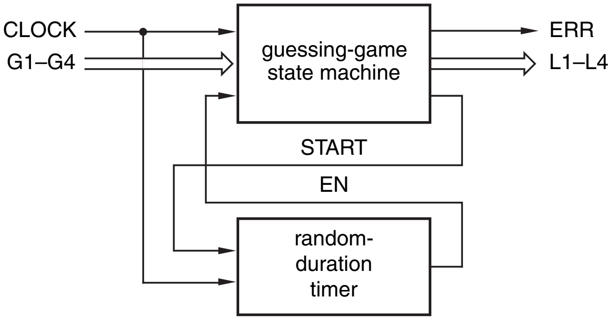


Figure 12-10 Block diagram of guessing game with random delay.

A block diagram for the enhanced guessing game is shown in Figure 12-10. The main machine is basically the same as before, except that it advances from one LED state to the next only if an enable input EN is asserted. The EN input is driven by a random-duration timer which starts running when it sees its START input asserted.

Just drawing the EN and START signals in Figure 12-10 does not define the communication protocol between the main machine and the timer, which requires both thought and documentation, for example as presented below:

- The EN and RUN inputs are checked by the respective machines only on the CLOCK edge.
- The main machine asserts START only when it is about to transition into one of its “running” states S1-S4, coming from a different state.
- The timer starts timing on the clock edge when it sees START asserted, and asserts its EN output immediately after that edge if it has selected a random duration of one clock tick, or during any other clock period thereafter depending on the selected random duration.
- The timer stops running during the period in which it asserts EN, unless and until it sees its START input asserted again, at which point it runs again for a newly selected random number of clock ticks.
- The main machine ignores the EN input if it is not in a running state.

Based on this description and having the benefit of the original guessing-game code in Program 12-22 as a starting point, we can design a top-level main machine for the enhanced guessing game as shown in Program 12-26. No additional states are required, so the state definitions are the same as in the original module, and the next-state logic is similar. However, in each “running” state, we advance to the next one only if EN is 1; otherwise we stay in the current state.

The code for the START output is also important. Here, START is asserted only if the next state will be different from the current state and is one of the four “running” states. Note that START is a Mealy-type output, since it depends on

Program 12-26 Top-level Verilog module for enhanced guessing game.

```
module Vrggamemain ( CLOCK, RESET, G1, G2, G3, G4, EN, L1, L2, L3, L4, ERR, START );
  input CLOCK, RESET, G1, G2, G3, G4, EN;
  output reg L1, L2, L3, L4, ERR, START;
  reg [2:0] Sreg, Snext;    // State register and next state

  parameter S1   = 3'b001,  // State encodings for 4 running states,
            S2   = 3'b010,
            S3   = 3'b011,
            S4   = 3'b100,
            SOK  = 3'b101,  // OK state, and
            SERR = 3'b110;  // error state

  always @ (posedge CLOCK)// Create state memory with sync reset
    if (RESET) Sreg <= SOK; else Sreg <= Snext;

  always @ (*)   // Next-state logic
    case (Sreg)
      S1  : if (G2 | G3 | G4) Snext = SERR;
            else if (G1)      Snext = SOK;
            else if (EN)      Snext = S2;
            else              Snext = S1;
      S2  : if (G1 | G3 | G4) Snext = SERR;
            else if (G2)      Snext = SOK;
            else if (EN)      Snext = S3;
            else              Snext = S2;
      S3  : if (G1 | G2 | G4) Snext = SERR;
            else if (G3)      Snext = SOK;
            else if (EN)      Snext = S4;
            else              Snext = S3;
      S4  : if (G1 | G2 | G3) Snext = SERR;
            else if (G4)      Snext = SOK;
            else if (EN)      Snext = S1;
            else              Snext = S4;
      SOK : if (~G1 & ~G2 & ~G3 & ~G4) Snext = S1; else Snext = SOK;
      SERR: if (~G1 & ~G2 & ~G3 & ~G4) Snext = S1; else Snext = SERR;
      default : Snext = SOK;
    endcase

  always @ (Sreg or Snext) begin      // Output logic
    L1  = (Sreg == S1);  L2  = (Sreg == S2);  L3  = (Sreg == S3);
    L4  = (Sreg == S4);  ERR = (Sreg == SERR);
    START = (Snext != Sreg) & ( (Snext==S1)|(Snext==S2)|(Snext==S3)|(Snext==S4) );
  end

endmodule
```

Program 12-27 Simple fixed timer for the guessing game.

```
module Vrggameftimer ( CLOCK, RESET, START, EN );
  input CLOCK, RESET, START;
  output wire EN;
  reg [1:0] CNT;         // Enough bits to count to 3
  parameter MAXCNT = 2;  // Assert EN after MAXCNT ticks (1, 2, or 3)

  always @ (posedge CLOCK)
    if (RESET) CNT <= 0;    // Counter is stopped when CNT is zero
    else if (START) CNT <= 1;
    else if ((CNT!=0) && (CNT!=MAXCNT)) CNT=CNT+1; // Keep counting after starting
    else CNT <= 0;                                 // but stop after hitting MAXCNT

  assign EN = (CNT==MAXCNT);   // Assert EN for one tick when MAXCNT is reached
endmodule
```

SNEXT which in turn depends on both the current state and the primary inputs. However, according to the inter-machine communication protocol presented above, the timer looks at the value of START only at a triggering CLOCK edge.

To complete the guessing game, we also need a random-timer submachine. We'll show how to build a pseudo-random timer shortly, but for simplicity we can first "stub in" a simple timer with a fixed delay of up to three clock ticks, as shown in Program 12-27. This module has just two bits for the count CNT, which supports a maximum timer count of 3, specified by a parameter MAXCNT. The short fixed delay makes it easier to debug the submachine's communication with the main machine, including corner cases like a delay of 1.

The next step is to create a top-level module that instantiates the main machine and the timer and hooks them up as we showed in Figure 12-10. Such a module Vrggametop appears in Program 12-28. Notice that it instantiates the timer Vrggameftimer with the parameter MAXCNT equal to 1, so the LED pattern shifts on every clock cycle. This should make the new machine operate the same as the original Vrggame machine in Program 12-22, and we can check it using

Program 12-28 Top-level Verilog module for the guessing game.

```
module Vrggametop ( CLOCK, RESET, G1, G2, G3, G4, L1, L2, L3, L4, ERR );
  input CLOCK, RESET, G1, G2, G3, G4;
  output wire L1, L2, L3, L4, ERR;
  wire START, EN;    // Communication between the main and timer machines
// Instantiate the main machine
  Vrggamemain U1 (.CLOCK(CLOCK), .RESET(RESET), .G1(G1), .G2(G2), .G3(G3), .G4(G4),
                  .L1(L1),.L2(L2),.L3(L3), L4(L4), ERR(ERR),.START(START),.EN(EN));
// Instantiate the timer
  Vrggameftimer #(.MAXCNT(1)) U2 (.CLOCK(CLOCK),.RESET(RESET),.START(START),.EN(EN));
endmodule
```

Program 12-29 Pseudorandom timer for guessing-game using LSFR output bits.

```
module Vrggameptimer ( CLOCK, RESET, START, EN );
  input CLOCK, RESET, START;
  output wire EN;
  reg RUNL;
  wire [23:0] QX;

Vrlfsr #(.N(24),.FE(24'b10000111)) U1 (.CLK(CLOCK), .RESET(RESET), .RUN(RUN), .QX(QX));

  always @ (posedge CLOCK)
    if (RESET) RUNL <= 1'b0;                         // All stopped during reset
    else if (START) RUNL <= 1'b1;                    // Start the LFSR running
    else if (QX[2:0]==3'b111) RUNL <= 1'b0;          // Stop after 8 ticks, on average

  assign EN = ( (QX[2:0]==3'b111) && (RUNL==1'b1) ); // Assert EN when LFSR is running
endmodule
```

the existing test bench in Program 12-24. Instantiating Vrggametop in existing test bench does work and the tests pass, as far as they go. We'll work on a more comprehensive test bench later. Now that we know that the basic communication between submachines works properly for at least this one case, we can start working on a real random-timer submachine.

An LFSR can be used to generate a pseudorandom sequence, so that an LED's "on" time depends on the counting sequence of the LFSR. We wrote an LFSR module Vrlfsr in Program 11-21 on page 591, which runs when its RUN input is asserted. The new guessing-game timer module Vrggameptimer in Program 12-29 instantiates Vrlfsr specifying a width of 24 bits. It also defines one bit of state, RUNL, which is applied to the RUN input of the LFSR. The new timer module waits for a random time to assert EN using the following strategy:

- When START is asserted, set RUNL to 1, which enables the LFSR to run.
- Decode the three low-order bits of the LFSR, so when they equal a certain predefined value (3'b111) and the LFSR is running, assert EN, which allows the guessing game LEDs to move to their next state. Note, this is a Mealy output.
- At the end of the clock period in which the predefined value is decoded, set RUNL to 0, which stops the LFSR until RUNL is asserted again.

There are some subtleties in the interactions among START, RUNL, EN, and the LFSR operation, especially in the definition of EN as a Mealy output and in the third bullet above. At the same that RUNL is set to 0, the LFSR will move on to its next random state, so the low-order bits could be different in the next clock period. And if START is already asserted when the predefined value is decoded in the third bullet, then RUNL instead remains at 1 and the LFSR will keep running in the next clock period. These subtleties ensure that the LEDs will shift on every clock tick, not every second clock tick, if the LFSR should contain a long string of 1s in its low-order bits.

With an *n*-bit LFSR, the "on" time for EN can range from one to *n* clock ticks. A larger value of *n* of course gives a larger range for the LED "on" time and also decreases its predictability, yielding the most fun.

Another LFSR-based approach could be to use a counter as in the original fixed timer but specify its maximum count dynamically according to the LFSR's QX output bits, or a subset of them. However, this can result in a high degree of predictability. After all, the LFSR's underlying component is just a shift register, and considering the parallel output QX[N-1:0], each successive value is just a right shift of the previous one with a new high-order bit. Considering QX as a numeric time duration, each successive value is no shorter than about half the previous one, and perhaps a lot longer, which leads to a pretty easy strategy for winning the game. This approach and ways of eliminating its predictability are explored in Exercises 12.42 and 12.46.

An enhanced self-checking test bench for the guess-game machine is shown in Program 12-30. This test bench makes guess inputs randomly. Since unlike a human it can react within one clock tick and has "perfect" knowledge of the LED states, it can then check the game's performance automatically, determining whether the game reacted correctly to its correct or incorrect guess. The body of the test bench has a for loop that applies 10,000 random inputs and examines the results in this way, so it checks the state machine very thoroughly.

The test bench performs a random delay in three places, and a parameter MAXwait specifies the maximum number of clock ticks for these delays. The initial block performs the needed signal initializations and FPGA start-up delay, and then executes the main for loop. The delays are aligned so the for loop will change inputs and test outputs in the middle of a 10 ns clock period.

To make the game more challenging, it's possible that some versions of the state machine may not turn on an LED immediately at the beginning of a turn, so the test bench allows up to 100 clock ticks for an LED to turn on. Once an LED has turned on, the test bench waits for a first random delay before looking at the LED states. Then it "looks at" the LED pattern and saves it in a variable GL for its eventual guess. But it waits for a second random delay before actually making the guess. After the delay, it saves the current, possibly changed, LED pattern in a variable CL. One tick later, the state machine should react to the guess by turning off all LEDs. The test bench compares GL and CL to determine whether its guess was correct and verifies that the state machine's ERR output corresponds. Finally, it waits for a third random delay before releasing the pushbutton (guess input) and goes back to the beginning of the main loop.

Although the test bench in Program 12-30 does a very good job of checking the state machine's operation for "normal" guesses, it is still far from complete. For example, it does not check that the LEDs always display a legal

Program 12-30 Self-checking guessing-game test bench with an automatic approach.

```
`timescale 1ns/100ps
module Vrggame_tba ();
reg Tclk, RST, G1, G2, G3, G4;
wire L1, L2, L3, L4, ERR;
integer ii, j, rand;
reg [1:4] CL, GL;      // Current and guessed LED pattern
parameter MAXwait = 4;

Vrggametop UUT ( .CLOCK(Tclk), .RESET(RST), .G1(G1), .G2(G2), .G3(G3), .G4(G4),
                 .L1(L1), .L2(L2), .L3(L3), .L4(L4), .ERR(ERR) );

always begin    // create free-running test clock with 10 ns period
  #5 Tclk = 0;  // 5 ns high
  #5 Tclk = 1;  // 5 ns low
end

initial begin    // What to do starting at time 0
  RST = 1;       // Apply reset
  {G1,G2,G3,G4} = 4'b0000;  // All guess inputs are 0
  Tclk = 1;      // Start clock at 1 at time 0
  #115;          // Wait 115 ns
  RST = 0; #20 ; // unreset and let it take effect
  for (ii=1; ii<=10000; ii=ii+1) begin
    for (j=1; j<=100; j=j+1) if ({L1,L2,L3,L4} == 4'b0) #10 ; // Wait for LED on
    if ({L1,L2,L3,L4} == 4'b0) begin  // Display error and stop if waited too long
      $display("Time: %d  No LED on after 100 ticks",$time);
      $stop(1); end;
    rand = $random % (MAXwait+1); if (rand<0) rand = -rand;
      #(10*rand);                         // Delay 0-MAXwait ticks before guessing
    GL = {L1,L2,L3,L4};                   // Save LED pattern for guess
    rand = $random % (MAXwait+1); if (rand<0) rand = -rand;
      #(10*rand);                         // Delay 0-MAXwait ticks to make guess
    CL = {L1,L2,L3,L4};                   // LED pattern at time guess is made
    {G1,G2,G3,G4} = GL;                   // Make guess using saved pattern
    #10 ;                                 // Wait for guess to be recognized
    if ({L1,L2,L3,L4} != 4'b0)            // Expect all LEDs off
       $display("Time: %d  LEDs not all off, L1-4=%4b",$time,{L1,L2,L3,L4});
    if (GL==CL) begin                     // Guess was correct
        if (ERR==1) $display("Time: %d  Incorrect ERR assertion",$time);
      end                                 // Else guess was incorrect
      else if (ERR==0) $display("Time: %d  Missed ERR assertion",$time);
    rand = $random % (MAXwait+1); if (rand<0) rand = -rand;
    #(10*rand);                           // Delay 0-MAXwait ticks before releasing PB
    {G1,G2,G3,G4} = 4'b0;                 // Negate guess inputs and continue loop
  end;
  $stop(1);                               // end test
end
endmodule
```

A RANDOM, COSTLY BUG

When I wrote the test bench in Program 12-30, I had a bug that was so bad that the simulator would produce absurd results and sometimes crash, and it took me several hours to find the bug. I wasted so much time on it that this book probably has one less example in it as a result. So, I'm sharing it here in the hope that you can avoid ever making the same error.

Notice the code in three places that gets a random number using Verilog's built-in `$random` function, and adjusts it to be in the range 0–`MAXwait`. My original code left out the `if` statement. Recall that the value returned by `$random` is a 32-bit *signed* integer. And the Verilog's definition of the modulo operation, `a % b`, says that the result has the same sign as the first operand. So, my original code could return a *negative* random delay. The bug is so obvious once you find it, it's laughable.

Unfortunately, different simulators handle negative delays differently, including forcing them to zero, or giving error messages or warnings. In Vivado's case, it silently accommodates them. In my case, the only discernible and maddening hints were that strange black bars sometimes appeared on portions of timing waveforms, and my own displayed messages sometimes appeared out of sequence. The latter shook my confidence in the entire software system, but eventually led me to my bug.

pattern, with only one LED on at a time, nor does it check the machine's ability to detect "cheating," like asserting multiple guess inputs simultaneously. These enhancements are requested in Exercise 12.43.

12.10 The Trilogy Game

The "Trilogy Game" is a two-player game that starts with three heaps of coins, sticks, or other objects, with 3, 5, and 7 objects in the heaps. The players alternate play, and at each turn a player must remove one or more objects from one heap—any nonzero number of objects, but all from the same heap. The loser of the game is the person who removes the last remaining object.

We can design a state machine to keep track of the number of objects in the heaps as the game is played. This is best done as a decomposed machine with three counters, H1, H2, and H3, whose outputs indicate the number of objects in the heaps, and a main machine with additional inputs and outputs as follows:

RST This input initializes the H1, H2, and H3 counters to 3, 5, and 7.

T1, T2, T3 These three inputs are associated with the like-numbered heaps.

NEXT This output indicates that the next player's turn may begin.

OVER This output indicates that the heaps are now all empty and the game is over.

The inputs are all sampled at the rising edge of a free-running CLK signal. As indicated, the counters are initialized when RST is asserted. Subsequently,

THE GAME OF NIM

The name of the Trilogy Game has nothing to do with the fact that it starts with three piles of objects. It's just what my family has always called it after learning the game many years ago by playing it with a crew member during a day trip on a catamaran named *Trilogy* in Hawaii.

The game is actually just one version of the very old and very well-known mathematical game of NIM, which has similar rules but can have many different starting configurations—any number of piles and any number of objects per pile. Also, in traditional NIM, you win the game by taking the last object, while in the Trilogy Game you lose by doing that. Playing to lose according to the normal rules is called *misère* play.

With a simple Web search, you can find many articles on winning strategies for NIM. The decomposed state machine design in this section merely keeps track of the heaps, assuming the Trilogy Game starting configuration. You can combine it with other logic to create a machine that plays the Trilogy Game with a human player (see Exercise 12.59). Since an intelligent first player who goes first can always win the Trilogy Game, you would definitely want to design any such state machine to allow you to go first, but you can test it by letting *it* go first.

the "user interface" for the game is based on the inputs T1–T3, which would be driven by pushbuttons, the status outputs NEXT and OVER, displayed on LEDs, and the counter outputs H1–H3, which could drive 7-segment displays.

The state machine asserts NEXT when it is ready for a player's turn to begin, and it continues to assert NEXT until the player begins. The player asserts Ti at one or more edges of CLK (not necessarily contiguous), and the state machine decrements the counter Hi each time. For proper operation, the machine must not decrement a counter below zero. Also, once the user decrements a particular counter, only that counter can be decremented further. The user indicates that the turn is over either by decrementing all the way to zero, or by asserting a Tj input other than the current input Ti. At that point, the machine asserts NEXT and is ready to accept a new move, unless all the counters are zero, in which case it asserts DONE and waits for RST.

In the specification above, we are expecting the user to synchronize their Ti inputs with CLK, which is not really reasonable for a human user, unless the clock frequency is very slow, a fraction of 1 Hz. Let us instead assume that the free-running CLK frequency is arbitrarily fast. Then we need a circuit that can receive a pushbutton input signal Pi of arbitrarily long duration, but which asserts an output Ti for just one CLK period after its leading edge. Program 12-31 is an edge-detector module that does this by storing `Pi` at two consecutive clock ticks, and looking for a 0 followed by a 1. Thus, we can apply a pushbutton input `Pi` to an instance of this module to obtain a corresponding edge-detect signal `Ti` that is a suitable input for the main machine. (We assume that the pushbutton inputs are already "debounced;" else, see Exercises 12.52 and 12.53.)

Program 12-31 Leading-edge detector module.

```
module Vredgedet ( CLK, P, T ); // Edge detector module
  input CLK, P;                 // Detects rising edge of P
  output reg  T;                // Assert for one tick on edge
  reg SP1, SP2;                 // Synchronize P with CLK

  always @ (posedge CLK) begin
    SP1 <= P; SP2 <= SP1;
    T <= SP1 & ~SP2;
  end
endmodule
```

We also need a Verilog module for the heap counters. We could instantiate the Vrupdn4 counter of Program 11-4 on page 563 with appropriate values to do the job, and rely on the tools to prune away any unneeded logic from this circuit. However, the required function is so simple that we can easily just define a new module to do the job as shown in Program 12-32. While we're at it, we've designed this counter to be "sticky," so it doesn't count below zero if it gets enabled after reaching that minimum count. That might come in handy later.

Now we are ready to tackle the overall state machine. The relationships among the top-level machine and its submachines are shown in Figure 12-11. There are three instances each of the Vredgedet module (pushbutton edge detectors) and the Vrtrilctr module (heap counters). They connect with a main module Vrtrilogymain which we describe next, and they are connected to each other in a top-level module Vrtrilogytop shown in Program 12-33.

The declarations for the main module are shown in Program 12-34. The inputs include CLK, RST, and the edge-detected pushbutton signals T1–T3. Remaining inputs are the heap-counter outputs H1–H3, needed to detect when a player's turn or the game is over. Notice that the H1 counter needs only two bits (initial count of 3) while the others need three. Module outputs are the count enables CNTEN[1:3] for the three heap counters, and the game status NEXT and

Program 12-32 Customized down counter for the Trilogy state machine.

```
module Vrtrilctr ( CLK, LD, EN, Q );
  parameter N = 3;                      // N-bit down counter
  parameter ICNT = 7;                   // with initial value ICNT
  input CLK, LD, EN;
  output reg [N-1:0] Q;                 // Count value

  always @ (posedge CLK) begin
    if (LD) Q <= ICNT;
    else if (EN && (Q!=0)) Q <= Q-1; // Stick at 0 just in case
    else Q <= Q;
  end
endmodule
```

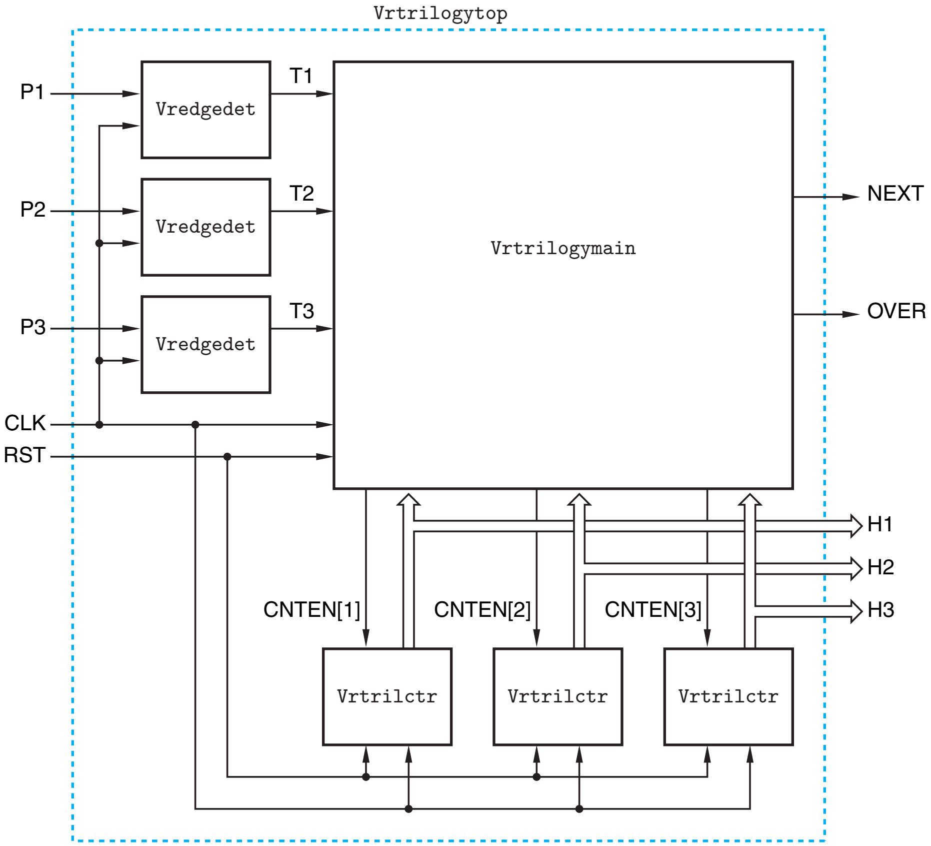


Figure 12-11 Top-level module and submodules for the Trilogy game.

Program 12-33 Top-level structural module for the Trilogy game.

```
module Vrtrilogytop ( CLK, RST, P1, P2, P3, H1, H2, H3, NEXT, OVER );
  input CLK, RST;                 // Clock and reset
  input P1, P2, P3;               // Input pushbuttons
  output [1:0] H1;                // Heap counters
  output [2:0] H2, H3;
  output NEXT, OVER;              // Next-move and game-over status
  wire T1, T2, T3;                // Edge signals detect from pushbuttons
  wire [1:3] CNTEN;               // Count enables for heap counters

  Vredgedet E1 (.CLK(CLK), .P(P1), .T(T1)); // Edge detectors
  Vredgedet E2 (.CLK(CLK), .P(P2), .T(T2));
  Vredgedet E3 (.CLK(CLK), .P(P3), .T(T3));                      // Heap counters

  Vrtrilctr #(.N(2),.ICNT(3)) C1 (.CLK(CLK), .LD(RST), .EN(CNTEN[1]), .Q(H1));
  Vrtrilctr #(.N(3),.ICNT(5)) C2 (.CLK(CLK), .LD(RST), .EN(CNTEN[2]), .Q(H2));
  Vrtrilctr #(.N(3),.ICNT(7)) C3 (.CLK(CLK), .LD(RST), .EN(CNTEN[3]), .Q(H3));

  Vrtrilogymain M1 (.CLK(CLK), .RST(RST), .T1(T1), .T2(T2), .T3(T3), .CNTEN(CNTEN),
                    .H1(H1), .H2(H2), .H3(H3), .NEXT(NEXT), .OVER(OVER));
endmodule
```

Program 12-34 Main-machine declarations for the Trilogy game.

```
module Vrtrilogymain ( CLK, RESET, T1, T2, T3, H1, H2, H3, CNTEN, NEXT, OVER );
  input CLK, RESET;               // Clock and reset
  input T1, T2, T3;               // Input-transition detect
  input [1:0] H1;                 // Heap counters
  input [2:0] H2, H3;
  output [1:3] CNTEN;             // Count enables for heap counters
  output NEXT, OVER;              // Next-move and game-over status
  reg [3:0] Snext, Sreg;

  parameter IDLE = 4'b0000,      // State encodings
            GT1  = 4'b0001,
            WT1  = 4'b0101,
            GT2  = 4'b0010,
            WT2  = 4'b0110,
            GT3  = 4'b0011,
            WT3  = 4'b0111,
            CHK  = 4'b0100,
            DONE = 4'b1000;
```

OVER. The module also declares reg variables for the next state and for the state register. A parameter statement defines the next-state encoding, which we'll discuss later, after we've actually worked out the states.

The main machine's next-state and output logic appear in Program 12-35. When RESET is asserted, a good place to begin is at an IDLE state, and that's what we'll do. In the output logic we assert NEXT in that state to signal the beginning of a turn. The key idea in the next-state logic is that once a player takes an object from a particular heap, the player must continue in only that heap until the end of the turn. Therefore, the IDLE state goes to a different next state depending on which Ti input is asserted, for example GT1 if T1 is first asserted. In that state, the output logic asserts the count enable for the corresponding heap counter, for example CNTEN[1].

Once the machine is in a GTi state, it may accept additional requests to decrement the corresponding heap counter. The edge detector machine that creates Ti can never generate two 1 inputs in a row, but we have designed the main machine to accommodate that situation just in case it is ever used in a situation where that's possible (for example, if another state machine is playing against a human player as in Exercise 12.59). Therefore, if Ti is asserted in the GTi state, the machine stays there to decrement the corresponding heap counter again.

A subtlety is that heap counter Hi might be getting decremented to zero during the current GTi state; if so, we shouldn't decrement it again. One way to handle this is for the output logic to prevent CNTEN[i] from being asserted if the counter is already zero, as shown in the output-logic comments. Another would be for the first line of the GTi next-state logic to check for Hi>=1 instead of H1!=0; either approach could lead to additional resources in the state machine's

Program 12-35 Trilogy game main-machine next-state and output logic.

```
  always @ (posedge CLK) begin
    if (RESET) Sreg <= IDLE; else Sreg <= Snext;
  end

  always @ (*)   // Next-state logic
    case (Sreg)
      IDLE: if       (T1 && (H1!=0))    Snext = GT1;  // Ignore input if
            else if (T2 && (H2!=0))    Snext = GT2;  //   corresponding
            else if (T3 && (H3!=0))    Snext = GT3;  //   heap is empty
            else                       Snext = IDLE;
      GT1:  if       (T1 && (H1!=0))    Snext = GT1;  // Decrement heap if two T1s
            else if (H1==0)            Snext = CHK;  //   in a row (unlikely)
            else if (T2 || T3)         Snext = CHK;  // Go CHK if heap empty or
            else                       Snext = WT1;  //   other Ti; else wait
      WT1:  if       (T1 && (H1!=0))    Snext = GT1;  // Decrement on another T1
            else if (H1==0)            Snext = CHK;  // Go CHK if heap empty or
            else if (T2 || T3)         Snext = CHK;  //   other Ti
            else                       Snext = WT1;  // Else wait
      CHK:  if ((H1==0) && (H2==0) && (H3==0))
                                       Snext = DONE; // Done if all empty
            else                       Snext = IDLE; // Else new turn
      DONE:                            Snext = DONE; // Game over, wait for RESET
      GT2:  if       (T2 && (H2!=0))    Snext = GT2;  // Same logic as GT1, WT1
            else if (H2==0)            Snext = CHK;
            else if (T1 || T3)         Snext = CHK;
            else                       Snext = WT2;
      WT2:  if       (T2 && (H2!=0))    Snext = GT2;
            else if (H2==0)            Snext = CHK;
            else if (T1 || T3)         Snext = CHK;
            else                       Snext = WT2;
      GT3:  if       (T3 && (H3!=0))    Snext = GT3;  // Same logic as GT1, WT1
            else if (H3==0)            Snext = CHK;
            else if (T1 || T2)         Snext = CHK;
            else                       Snext = WT3;
      WT3:  if       (T3 && (H3!=0))    Snext = GT3;
            else if (H3==0)            Snext = CHK;
            else if (T1 || T2)         Snext = CHK;
            else                       Snext = WT3;
      default                          Snext = IDLE;
    endcase
                                  // Output logic
  assign CNTEN[1] = (Sreg==GT1);  // && (H1!=0) optional because of counter design
  assign CNTEN[2] = (Sreg==GT2);  // && (H2!=0) ditto
  assign CNTEN[3] = (Sreg==GT3);  // && (H2!=0) ditto
  assign NEXT = (Sreg==IDLE);
  assign OVER = (Sreg==DONE);

endmodule
```

FSM DETRACTION The state assignments selected by the Xilinx Vivado tool when FSM extraction is enabled are surprisingly poor compared to the state assignment that we described in this section, which we just kind of pulled out of the air based on symmetries in the `Vrtrilogymain` module's next-state logic. Using that state assignment as written in Program 12-34, the synthesized module uses 4 flip-flops and 16 LUTs. With FSM extraction enabled in the "auto" mode, Vivado makes a "sequential" assignment that uses the same number of flip-flops but 35 LUTs! Even if the "Gray" assignment is forced, it uses 24 LUTs. The "Johnson" and "one-hot" options use more flip-flops and 30 and 24 LUTs, respectively. Hooray for experienced designers—for now!

realization (see Exercise 12.55). In this version of the module, we have done neither, since we happened to design the heap counter so it will not decrement below zero even if enabled.

As we've said, in the normal operating environment, a `Ti` input will never be asserted for two clock ticks in a row. However, it may be asserted at one or more later times. Therefore, the next-state logic includes a "wait state" `WTi` for each heap, where the machine goes from `GTi` and waits for additional inputs. If the corresponding `Ti` input is asserted again, it goes back to `GTi` and the cycle repeats.

The next-state logic in both the `GTi` and the `WTi` states check for whether the turn is over, which can be signaled either by the corresponding heap counter reaching zero or by one of the other `Ti` inputs being asserted. In either case, the machine goes to a state `CHK` which checks all of the heaps to determine if the game is over. If all are empty, then it goes to the `DONE` state where it stays until the next `RESET` occurs, and it asserts `OVER` while it waits. Otherwise, it goes back to the `IDLE` state to begin another turn, asserting `NEXT`.

With the next-state and output logic completed, we can circle back to the state assignments. There are a total of nine states, so we'll need to use four bits. It makes sense as usual to encode the reset state (`IDLE`) as all 0s. We'll try to exploit the machine's symmetries in the remaining assignments. There are three pairs of `GTi`/`WTi` states, so we'll encode them with the three different nonzero combinations in the two low-order bits, and 00 and 01 in the two high-order bits to distinguish `GTi` and `WTi`. Since there are only 9 states, not 10 or more, we need to use just one 4-bit combination where the MSB is 1, say 1000, for the `DONE` state; thus the `OVER` output equals the MSB and could be generated without any additional logic. That leaves the 0100 combination for `NEXT`.

A simple test bench for the Trilogy game is shown in Program 12-36. Like our first test bench for the guessing game, this one applies a sequence of user-constructed moves that are applied to the game through a task. The task, `Move`, is called with a heap number, a number of button presses to make, and the number of objects expected to be in each heap after the presses have been made. After

Program 12-36 Test bench for the Trilogy game.

```
`timescale 1ns/100ps
module Vrtrilogy_tb ();
  reg CLK, RST, P1, P2, P3;         // Individual inputs
  wire [1:0] H1; wire [2:0] H2, H3; // Heap counters
  wire NEXT, OVER;                  // Next-move and game-over status

  Vrtrilogytop UUT ( .CLK(CLK), .RST(RST), .P1(P1), .P2(P2), .P3(P3),
                     .H1(H1), .H2(H2), .H3(H3), .NEXT(NEXT), .OVER(OVER) );
  task Move;
    input integer heap, n, exp1, exp2, exp3;
    integer ii;
    begin
      for (ii=1; ii<=n; ii=ii+1) begin // Push button (heap) n times
        #(70 + ($random % 30)); // Random (70+-29) wait til PB press
        case (heap)
          1: P1 = 1; 2: P2 = 1; 3: P3 = 1; default ;
        endcase
        #(70 + ($random % 30)); // Random (70+-29) PB press duration
        P1 = 0; P2 = 0; P3 = 0;
      end
      if ((H1!==exp1) || (H2!==exp2) || (H3!==exp3)) $write("Error: ");
      else $write("       ");
      $display("HEAP:N  H1 H2 H3  NEXT OVER %1d:%1d  %1d %1d %1d  %1b %1b",
               heap, n, H1, H2, H3, NEXT, OVER);
    end
  endtask

  always begin    // create free-running test clock with 10 ns period
    #6 CLK = 0;  // 6 ns high
    #4 CLK = 1;  // 4 ns low
  end

  initial begin
    CLK = 1; RST = 1; P1 = 0; P2 = 0; P3 = 0;
    #115 RST = 0;
    Move(0,0,3,5,7); // Check heap initialization
    Move(1,1,2,5,7); // Do a turn
    Move(3,1,2,5,7); // End the turn
    Move(2,2,2,3,7); // Do a turn
    Move(1,1,2,3,7); // End the turn
    Move(3,7,2,3,0); // Do and end a turn
    Move(3,1,2,3,0); // Try an illegal turn
    Move(1,2,0,3,0); // Do and end a turn
    Move(2,2,0,1,0); // Do a turn
    Move(1,1,0,1,0); // End a turn
    Move(2,1,0,0,0); // Do the last turn
    $stop(1) ;
  end
endmodule
```

each move, it displays the move, the number of remaining objects per heap, and the `NEXT` and `OVER` outputs, and it also flags the error if any object count is different from what's expected.

Of course, the user-constructed sequence of moves in Program 12-36 is not comprehensive, and it doesn't even check the `NEXT` and `OVER` outputs. It's also possible to design a test bench that creates random moves, playing the game multiple times with different move sequences, and checking the results automatically, as we did in Program 12-30 for the guessing game (see Exercise 12.61).

References

Verilog supports many different state-machine coding styles, in fact, too many. Our recommended state-machine coding style is based on a 1998 paper by Clifford E. Cummings titled "State-Machine Coding Styles for Synthesis." This and many other interesting Cummings papers can be found on his website at `www.sunburst-design.com/papers`. For example, two of them (coauthored with Don Mills and Steve Golson) describe in great detail the pros and cons of synchronous versus asynchronous reset signals for state machines and clocked systems in general.

Reset inputs are important for both simulation and hardware testing, but it's also possible to force some state machines from any unknown state into a known state using a sequence of inputs called a "synchronizing sequence." For example, an n-bit serial-in shift register without a load or clear input can still be forced into the all-0s state in n clock ticks simply by shifting in n 0s. Likewise, a "sticky" up/down counter can be forced to a known state by counting up or down long enough. There's actually a very well developed but almost forgotten theory and practice of synchronizing sequences and somewhat less powerful "homing experiments," described by Frederick C. Hennie in *Finite-State Models for Logical Machines* (Wiley, 1968). But unless you've got this old classic on your bookshelf and know how to apply its teachings, please just remember to provide a reset input in every state machine that you design!

A mathematical theory of state-machine decomposition has been studied for years; Zvi Kohavi and Niraj K. Jha discuss the topic in the classic book *Switching and Finite Automata Theory*, (Cambridge University Press, 2010, third edition). They also discuss synchronizing sequences and homing experiments and relate them to issues of testing sequential circuits.

Drill Problems

12.1 Write a Verilog module for the state machine described by the state diagram in Figure X9.15. Note that the diagram is drawn with the convention that the state does not change except for input conditions that are explicitly shown. At reset, the machine should start in state A.

12.2 Write a Verilog module for the state machine described by the state diagram in Figure X9.14. At reset, the machine should start in state A.

12.3 Write a test bench that exercises the Verilog state machine that you wrote in Drill 12.2, ensuring that every transition in the state diagram of Figure X9.14 is taken at least once. You may use the method of Section 12.2.3.

12.4 Write a Verilog module for a state machine with the state/output table shown in Table X9.20. Use two state variables, Q1 Q2, with the state assignment A = 00, B = 01, C = 11, D = 10, and provide a RESET input that initializes the machine to state A. Also, draw a state diagram equivalent to the state table

12.5 Write a Verilog test bench that exercises the state machine of Drill 12.4 for an input sequence that takes each possible transition at least once. Draw a path on your state diagram to show the order in which transitions are taken and states are visited by your test bench.

12.6 Update the test bench in Program 12-6 so it also exercises transitions 1, 5, and 13 in the `VrSMexra_chk` module of Program 12-9.

12.7 Update the self-checking test bench in Program 12-7 so it also checks transitions 1, 4, 7, 12, and 13 in the `VrSMexra_chk` module of Program 12-9.

12.8 Write a Verilog module `Vredge` for a state machine with one input X and one Moore-type output EDGE, which detects transitions on X. The machine tests its X input at each tick of the clock and asserts EDGE if the value of X at that tick is different from the value at the previous tick. Use state names A, B, C, and so on as needed, and use the “direct coding” approach of Section 12.1.6.

12.9 Write a Verilog module `Vrgettwo` for a state machine with two inputs, INIT and X, and one Moore-type output Z. As long as INIT is asserted, Z is continuously 0. Once INIT is negated, Z should remain 0 until X has been 0 for two successive ticks and 1 for two successive ticks, regardless of the order of occurrence. Then Z should go to 1 and remain 1 until INIT is asserted again. Use the “direct coding” approach of Section 12.1.6.

12.10 Write a self-checking test bench for the `Vrgettwo` machine of Drill 12.9 that asserts `INIT` followed by a random sequence of 20 inputs on `X`. It should then assert `INIT` again and follow with a different random sequence, and repeat the process for a total of 20 different random sequences.

12.11 Draw a timing diagram showing the inputs, outputs, and state variables (including `lastA`) of the Verilog state machine in Program 12-5 when the test bench of Program 12-7 is run. You can try to work this out by hand, or you can just run the test bench!

12.12 Run the Verilog test bench in Program 12-7, and verify that it still works for one or more other `VrSMex` state-machine versions. Then, introduce a bug in the `OK` state in the `VrSMexa` version of the UUT, checking `B` against 0 instead of 1, and confirm that the test bench catches the bug. Finally, can you insert a bug in the UUT that the test bench does *not* catch?

12.13 Write a Verilog module for a “sticky-counter” state machine with eight states, S0–S7, assigned in binary counting order. Besides CLOCK, your machine should have two inputs, RESET and ENABLE, and one output, DONE. The machine should go to state S0 whenever RESET is asserted. When RESET is negated, it should move to next-numbered state only if ENABLE is asserted. However, once it reaches state S7, it should stay there unless RESET is again asserted. The DONE output should be 1 if and only if the machine is in state S7 and ENABLE is asserted.

12.14 Write a Verilog test bench to check for proper operation of the sticky counter that you designed in Drill 12.13.

12.15 Write a Verilog module for a state machine that is similar to the one specified in Drill 12.13 except that, when enabled, it counts "two steps forward, one step back." The machine should have one additional output, BACK, that is asserted if ENABLE is asserted and the machine is going to count back on the next clock edge. Once the machine gets to state S7, it never counts back. Comment your module to describe your strategy for creating this behavior and how many additional state bits are needed.

12.16 Write a Verilog test bench to check for proper operation of the state machine you designed in Drill 12.15.

12.17 Update the guessing-game state machine of Program 12-22 to provide an output "OK" which is asserted when the game is stopped after a correct guess. Then update the test bench of Program 12-24 to accommodate and test the additional output.

12.18 As a matter of coding style, it is possible to eliminate the final `else` clause in each of the cases in Program 12-26 by assigning the current state to `Snext` just prior to the `case` statement. Make this change and show that the synthesized module is exactly the same (with most tools). You can run the test bench in Program 12-30 to ensure that everything still works.

12.19 Augment the guessing-game test bench of Program 12-24 to test for both correct and incorrect guesses made with the G3 and G4 pushbuttons.

Exercises

12.20 Draw a state diagram for the state machine in the Verilog module on page 605, naming the states S00–S11 for the four combinations of Q1 Q2, and showing arcs and expressions only for input conditions that cause the state to change (no self loops). Can you find words that easily describe what this state machine does?

12.21 Write a Verilog module `Vredgemiss` for a state machine with two inputs X and INIT and two Moore-type outputs EDGE and MISS, which dependably detects transitions on X. The machine tests its X input at each tick of the clock and asserts EDGE if the value of X at that tick is different from the value at the previous tick. Once it is asserted, EDGE remains asserted until INIT is asserted for at least one tick. The MISS output is asserted if one or more edges were missed prior to INIT being asserted after EDGE was asserted, and it also remains asserted until INIT is asserted. Be careful with "boundary" cases. In particular, if an edge occurs while INIT is asserted, then EDGE should still be asserted, while MISS should be negated.

12.22 Write a Verilog test bench that checks for correct behavior of the `Vredgemiss` state machine of Exercise 12.21. Pay particular attention to boundary cases.

12.23 Write a test bench that graphically displays the outputs of any of the T-bird tail lights machines of Section 12.5 for a comprehensive input sequence. *Suggestion*: a sequence of lamp states may be displayed using an "`O`" or "`.`" for each lamp depending on whether it's on or off; for example, in a left turn:

```
... ...
..O ...
.OO ...
OOO ...
```

12.24 What does the personalized license plate in Figure 12-5 stand for? (*Hint:* It's the author's old plate, a computer engineer's version of OTTFFSS.)

12.25 Convert the T-bird tail lights machine in Program 12-15 to an equivalent module `VrTbirdSMp` that has pipelined outputs. Write a test bench that compares the new module's outputs with those of `VrTbirdSM` for a comprehensive input sequence.

12.26 Synthesize the `VrTbirdSMe` module of Program 12-16 six times using Xilinx tools with FSM extraction, targeting your favorite FPGA and specifying each of the following six state-assignment styles: off, sequential, Gray, Johnson, one-hot, and auto. Also, synthesize the `VrTbirdSMeoc` module of Program 12-17 with FSM extraction disabled ("off" option). Construct a table that gives the following results for each synthesis run: number of LUTs used, number of flip-flops used, maximum clock frequency, and maximum delay from clock input to any module output (not that the application needs fast timing). Comment on the results, referring to the characteristics discussed in Section 12.1.7 as appropriate.

12.27 Repeat Exercise 12.26 for the `Vrsvale` module in Program 12-18. Besides the six results for `Vrsvale`, include results for the `Vrsvaleoc` module, which has the changes in Program 12-19, synthesized without FSM extraction.

12.28 Xilinx Vivado 2016.3 tools cannot perform FSM extraction on the `Vrsvaleocov` module in Program 12-20. Figure out why not, and update the module to create an equivalent module `Vrsvaleocov_fsme` that supports FSM extraction.

12.29 Repeat Exercise 12.26 for the `Vrsvaleocov_fsme` module that you created in Exercise 12.28.

12.30 Modify the `Vrsvale` state machine of Program 12-18 to make a new module `Vrsvale_cs` whose output logic uses a `case` statement. Write a test bench that compares the modules' outputs and confirms that they are identical for a comprehensive input sequence. Also compare the synthesis results for the two modules when they are targeted to your favorite FPGA.

12.31 Modify the `Vrsvale` state machine of Program 12-18 to make a new module `Vrmtnview` whose next-state behavior is more rationale and attempts to minimize the waiting times of cars. You don't need to get fancy, but you can assume that the long timer is reduced to 2 minutes.

12.32 Write a test bench that graphically displays the outputs of the Sunnyvale traffic-lights machine of Program 12-18 for a comprehensive input sequence.

12.33 Write a test bench that stimulates the Sunnyvale traffic-lights machine of Program 12-18 with a random sequence of car arrivals at the sensors, measures the waiting time of each car, and calculates the intersection's average waiting time, capacity (cars per hour), and maximum queue length (number of waiting cars) over a long interval (say, one hour). Make the following assumptions:

- Cars arrive from the north and east directions only, at random intervals between 3 and 18 seconds selected independently for each direction.
- No more than 30 cars wait in the queue in any direction; beyond that, they turn around and go home.
- When a light changes from red to green, the first waiting car if any goes through the intersection immediately, and as long as the light remains green, the next one goes after 6 seconds, and the rest every 3 seconds.
- If there are more than 10 cars in the queue for a given direction, queued drivers treat a yellow light as if it were green.
- If the light is already green and the queue is empty when a car hits the sensor, the car goes through the intersection immediately.

12.34 Re-run the test bench of Exercise 12.33 with the arrival interval set to range between 3 and 66 seconds for each direction. What happens to the intersection's performance metrics?

12.35 Re-run the test bench of Exercise 12.33 against the `Vrmtnview` state machine of Exercise 12.31. Compare the intersection's performance metrics for the two different machines.

Fibonacci sequence

Fibonacci number

12.36 A *Fibonacci sequence* is a sequence of integers in which each integer is the sum of the previous two integers. When the first two integers in the sequence are defined to be 1, the Fibonacci sequence is 1, 1, 2, 3, 5, 8, 13, … . A *Fibonacci number* is an integer that appears in this sequence.

Write a decomposed Verilog state machine whose single output bit is asserted in a Fibonacci sequence. Use $n = 8$ for the purposes of this exercise, but write your code so that n can be easily changed if desired. Besides CLOCK, the machine should have two inputs, RESET and an n-bit input bus D for loading the first two defined Fibonacci numbers (usually 1 and 1). It should have two outputs: FIB and DONE.

On the first clock tick after RESET is negated, the machine should load the first defined Fibonacci number (usually 1) from the D bus into an internal n-bit counter A. On the second clock tick, it should load the second defined Fibonacci number (usually 1 also) from the D bus into a second n-bit counter B and assert FIB.

On successive clock ticks, the machine asserts FIB only if it has been j clock ticks since FIB was last asserted, where j is the next Fibonacci number in sequence (starting with the first). You may define another internal n-bit counter C if needed, as well as a top-level state machine to control the overall operation. Some time after the FIB is asserted for the last n-bit Fibonacci number, the machine asserts DONE and waits for RESET to be asserted again.

12.37 Write a Verilog test bench that checks for proper operation of the state machine you designed in Exercise 12.36 when the first two Fibonacci numbers are 1 and 1, assuring that it asserts FIB at the proper clock ticks (2, 3, 4, 6, 9, 14, …) and no others, and then asserts DONE within a reasonable period of time.

12.38 Write a test bench that compares the UNLK outputs of the combination-lock machines of Programs 12-13 and 12-14 against each other for a 5000-tick random input sequence. Explain or correct any discrepancies in their outputs. How often would you expect UNLK to be asserted, and how often does it actually happen?

12.39 Enhance the combination-lock machine of Program 12-13 to provide a HINT output that actually works. Use an ad hoc approach in which you just write an appropriate equation for HINT based on the current state and input. Test your enhanced machine using the test bench of Exercise 12.38 after first updating it to compare HINT outputs as well.

12.40 Synthesize the two different combination-lock machines of Program 12-14 and Exercise 12.39, targeting your favorite programmable device. Compare the resource requirements of the two design approaches.

12.41 Redesign the T-bird tail lights machine of Section 12.5 to include parking light and brake light functions. When the BRAKE input is asserted, all of the lights should go on immediately, and stay on until BRAKE is negated, independent of any other function. When the PARK input is asserted, each lamp is turned on at 50% brightness at all times when it would otherwise be off. This is achieved by driving the lamp with a 100-Hz signal DIMCLK with a 30% duty cycle. Partition the Verilog design into as many modules as you feel are appropriate, and target the top-level design to a single programmable device. Also, write a short description of how your system works.

12.42 Write a new random timer Vrggamertimer module for use with the decomposed guessing-game state machine of Program 12-28. The new module should use a counter as in the Vrggameftimer module, except that its 8-bit maximum value should be set each time that START is asserted according to the parallel outputs QX[7:0] of an 8-bit LFSR. The LFSR itself should advance by only one state each time that START is asserted.

12.43 Enhance the guessing-game test bench of Program 12-30 so it checks the following additional game behaviors: (1) the game detects an error if the user presses two or more guess buttons at the same time to make a guess; (2) the game detects an error if the user presses a second button while the game is stopped; (3) when the game is running, exactly one LED is on at each tick. Note that the test bench need not check that the LEDs are lit in order; in another version of the game, out-of-order sequencing may be a feature for added difficulty.

12.44 When the guessing-game test bench is run with exactly the configuration shown in Program 12-30 and instantiates the pseudo-random timer in Vrggametop, fewer than 200 of the 10,000 guesses use the G4 guess input. With the fixed timer and MAXCNT=3, *none* of the guesses use the G3 or G4 inputs. Explain why this happens and make improvements to the test bench to get more uniform test coverage on the guess inputs.

12.45 Write a test bench Vrggame_tbc for the guessing game that simply runs the game continuously for 10,000 clock ticks, without ever asserting any guess inputs. Use it to exercise the Vrggametop module using the original Vrggameptimer module of Program 12-29 and observe the output waveforms on L1-L4. Then repeat using the Vrggamertimer module of Exercise 12.42. Which output waveforms seem more "random" in the sense of making the game more difficult to win? Can you describe an effective strategy for winning the game in one or both cases?

12.46 Repeat Exercise 12.42 using a 24-bit LFSR. The counter should still have only 8 bits, loaded from the low-order bits of the LFSR. Observe the new module's output behavior using the `Vrggame_tbc` test bench of Exercise 12.45. With the larger LFSR, `MAXCNT` will sometimes be all-0s; does everything still work? For how many ticks are the LEDs unchanged in this case? In what ways, if any, has the larger LFSR made the game more difficult to beat?

12.47 Suggest and make a simple change in the `Vrggamertimer` random-timer module of Exercise 12.42 that makes the LED timing less predictable without requiring significantly more chip resources, if any. Test your updated module with the test bench of Exercise 12.45 and comment on the results.

12.48 Change Program 12-26 to make a new module `Vrggamemain_seq` so that when the game restarts after a guess, the LED pattern picks up where it left off, instead of starting with L1 again.

12.49 Change Program 12-26 to make a new module `Vrggamemain_rand` so that the LED pattern is random rather than a rotating sequence—random next LED and random duration.

12.50 Write Verilog modules for a decomposed state machine `VrgetN` for a state machine with two 1-bit inputs, INIT and X, a 4-bit input N[3:0] representing an unsigned integer *n*, and one Moore-type output Z. As long as INIT is asserted, Z is continuously 0. Once INIT is negated, Z should remain 0 until X has been 0 for *n*+2 successive ticks and 1 for *n*+2 successive ticks, regardless of the order of occurrence. Then Z should go to 1 and remain 1 until INIT is asserted again.

12.51 Write a self-checking test bench for the `VrgetN` machine of Exercise 12.50 that sets `N[3:0]` to 0000 and asserts `INIT`, followed by a random sequence of 2000 inputs on `X`. It should then assert `INIT` again, increment `N[3:0]`, and follow with a different random sequence, repeating the process for all 16 values of `N[3:0]`. Suggest ways of making the tests more effective when `N[3:0]` is large.

12.52 Design a new pushbutton-input module `VrPBdebedge` for use in the Trilogy game that debounces a pushbutton input before detecting its leading edge and then producing a 1-clock-tick edge-detect signal as in the existing `Vredgedet` module. Assume that the pushbutton is a single-pole, double-throw switch configured as shown in Figure 10-28. However, assuming you are targeting your design to an FPGA, there may be no native S-R latch available, and one built from LUTs and feedback may not work reliably (see the box on page 720). Therefore, design the debounce circuit using a natively available component like a D latch (e.g., Xilinx LDCE). Write a test bench that checks your circuit for proper operation with a "bouncy" switch input.

12.53 Design a new pushbutton-input module `VrPBdebedgecnt` for use in the Trilogy game that debounces a pushbutton input before detecting its leading edge and producing a 1-clock-tick edge-detect signal as in the existing `Vredgedet` module. Assume you have available only a single-pole, *single*-throw pushbutton in the configuration of Figure 10-27. Debounce the switch input by recognizing an edge only when the input has remained in a new state for a certain number of clock ticks, determined by a parameter DBCNT. You may also define a parameter NBCNT which is the number of bits needed in a counter for DBCNT. Write a test bench that checks your circuit for proper operation with a “bouncy” switch input.

12.54 Based on its nature, the Trilogy game is a good candidate for using the unary code to show the heap counts—one LED per object. Instantiate the Vrtrilogytop module in a higher-level module to do this, also making use one of the binary-to-unary modules of Exercises 6.34–6.36.

12.55 Show how to modify the Vrtrilogymain next-state logic in Program 12-35 so it will never assert the CNTEN input for a heap counter when the count is already zero. How does this change affect the total number of LUTs required compared to the original design? How does this compare with the savings would now be possible by omitting the zero-check in the Vrtrilctr module?

12.56 In Section 12.10, the discussion about state assignment for the Vrtrilogymain module says that the OVER output may be created without any additional logic, but in synthesis it actually uses a 4-input LUT. Edit the output-logic Verilog code so this LUT is not required. What are the pros and cons of doing this?

12.57 Using or modifying the modules in Section 12.10, write Verilog modules to track play in a generalized game of NIM with three heaps. Each heap may be initialized with up to 15 objects at the start. To support this capability, the new game requires a 4-bit data input, DI[3:0]. After reset is negated, the user places the desired initial count for the first heap on DI[3:0]. and then presses pushbutton P1. The user then does the same for the second and third heaps, pressing P2 and P3. At that point, the game machine asserts NEXT and play proceeds as in the Trilogy game.

12.58 By searching the Web, you can easily find various strategies for winning the game of NIM for any configuration, including the configuration and rules used in the Trilogy game. Write a Verilog module that uses such a strategy to try to win the game against a human player. Given a current heap configuration in the game, the module should determine what move to make to guarantee a win, if such a move exists. If there is no such move, it should take just one object from the largest heap, to give itself more time for the human to make a mistake. The module may be combinational or sequential. Its inputs are the current heap counts (H1, H2, H3), and its outputs are the heap number to take from (HN) and the number of objects to take (NT). In the case of a sequential module, there is also a clock input and you must define an input and output to initiate the search for a move and to indicate the search’s completion.

12.59 Using Verilog modules, design a machine that plays the Trilogy game against a human player. The machine should use the modules in Section 12.10 to keep track of the game heaps, and the module you designed in Exercise 12.58 to select its own move. Define and document additional inputs and outputs as needed to create a convenient user interface that allows either the human or your machine to move first, and for the human or your machine to enter moves using the same interface that was used in Section 12.10. If you have not done Exercise 12.58, you may use a “dummy” module for your machine to make fixed, random, or semi-intelligent moves on its turns (e.g., take all but one object if it ever encounters a state with objects in only one heap).

12.60 Enhance the Trilogy-game test bench of Program 12-36 so it checks the values of NEXT and OVER during and after each move. Figure out a way to do this without adding any more inputs to the Move task. Augment the test sequence to exercise this feature, and insert one or more bugs into the Vrtrilogymain module to ensure that the feature is working.

12.61 Write a test bench for the Trilogy game that plays the game repeatedly, each time automatically constructing a random sequence of moves and checking for correct values in the heap counts and NEXT and OVER during and after each move. Your test bench should check both legal and illegal moves, like asserting the guess input for a heap that is already empty.

12.62 Design a sequential version of the Verilog Tic-Tac-Toe move-picking circuit by adapting the TwoInRow module in Section 7.5, but instantiating it only once and using multiple clock cycles to determine a move. Besides a new input CLK, your version will need a START input to tell it to start looking for a move and a DONE output to assert when the determined move is on its output. Your circuit should use as few clock cycles as possible to find the move in any given situation.

12.63 Target your solution to Exercise 12.62 to your favorite programmable device. Repeat with the original combinational Tic-Tac-Toe circuit of Program 7-28. Compare the resource requirements of the two synthesized versions.

12.64 Write a Verilog module for a state machine with two outputs, Z and DONE, where the sequence and duration of Z=1 outputs represents a symbol (usually a letter or number) in Morse code, and DONE is asserted in the clock period following the symbol's last Z=1 output. The machine's inputs are CLK, SSTART, and SYM[0:9].

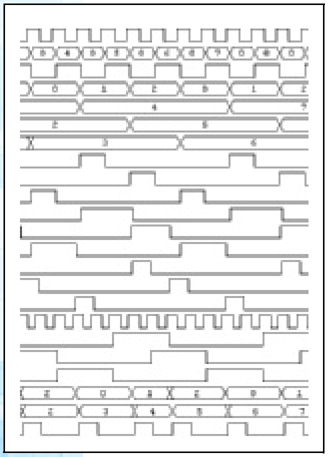
Morse-code symbols are encoded as "dots" and "dashes," where a dot is represented by Z=1 for one clock period, and a dash is represented by Z=1 for three clock periods. A symbol may have one to five dots and dashes that are separated by Z=0 for one clock tick.

A symbol to be transmitted is encoded in SYM[0:9] as five pairs of bits (SYM[0:1], SYM[2:3], and so on); in each pair, 10 encodes a dash, 01 encodes a dot, and in symbols with fewer than five dots and dashes, 00 indicates the end. Dots and dashes are transmitted starting with SYM[0:1].

After reset and when the machine is idle, Z is 0 and DONE is 1. A symbol to be transmitted is presented on SYM[0:9] , and SSTART is asserted, for one clock tick. In the next clock period, DONE should be negated, and in the period after that, the symbol's first dot or dash should begin on Z. The machine should assert DONE in the clock period following the end of the symbol's last dot or dash.

12.65 Using your solution to Exercise 12.64 as a submachine, write a Verilog module to transmit a message in Morse code, as read from a memory MSG[0:9][1:128]. A space symbol between words in the message is indicated by SYM[0:1]=00. On output, multiple symbols in a word are separated by Z=0 for three clock ticks, and words are separated by Z=0 for seven clock ticks. Provide an input MSTART and an output MDONE to start message transmission and to indicate completion.

12.66 Write a test bench to display the output of your module from Exercise 12.65. Determine whether your module meets the problem's specification for the gaps between symbols and words, and modify it if necessary to comply.



chapter 13

Sequential-Circuit Design Practices

Since we haven’t yet done it elsewhere, we begin this chapter with a summary of sequential-circuit documentation standards, focusing on two primary areas. The first area describes the circuits’ high-level behaviors step-by-step in terms of their inputs and outputs, including state-machine specifications. The second describes the low-level or “timing” behaviors of their important input, output, and internal signals, mainly by using timing diagrams and timing specifications.

The past four chapters have described various state machines and sequential-circuit building blocks, almost all of which are clocked. While it is possible to build sequential circuits and systems without a clock, the most commonly used and dependable digital-system design methods use clocked circuits primarily. Therefore, we will continue to emphasize *synchronous systems*—that is, systems and subsystems in which all flip-flops are clocked by the same common clock signal. We’ll give an example that shows how clocked state machines and data-path elements are typically used together to build a synchronous system.

We’ll also highlight some common problems in synchronous system design. For example, when we interconnect digital systems or subsystems that use different clocks, or when a system has interfaces to “the outside world,” we must identify asynchronous signals that need special treatment, and use special methods to move information between clock domains, as we’ll show in the last two sections of this chapter.

13.1 Sequential-Circuit Documentation Practices

13.1.1 General Requirements

Basic documentation practices in areas such as signal naming, logic symbols, schematic layout, and HDL coding style, which we introduced in previous chapters, apply to digital systems as a whole and therefore to sequential circuits in particular. We highlight the following ideas, however, for system elements that are specifically "sequential":

- *Flip-flops.* The symbols for individual sequential-circuit elements, especially flip-flops, should follow the appropriate drawing standards, so that the type, function, and clocking behavior of the elements are clear.
- *State-machine descriptions.* State machines should be described by state tables, state diagrams, or text files in an HDL. Most often, an HDL-based description is treated as definitive, since it contains the source code for the machine's realization. State tables and state diagrams are then secondary resources which explain the machine's operation, and they may be separate documents or may be embedded as comments in the HDL file.
- *State-machine HDL code or drawing structure.* Within an HDL-based design, the flip-flops, next-state logic, and output logic that form each state machine should be specified together in a single module, with no extraneous logic. If you should ever have occasion to draw a state machine's logic diagram, its flip-flops and combinational logic should be drawn together in a logical format on the same page, so it looks like a state machine.
- *Cascaded elements.* In a schematic-based design, any registers, counters, and shift registers that use multiple ICs should have these components grouped together so that the cascading structure is obvious.
- *Timing diagrams.* The documentation package for sequential circuits should include timing diagrams that show the general timing assumptions and timing behavior of the circuit.
- *Timing specifications.* A sequential circuit should be accompanied by a specification of the timing requirements for proper internal operation (e.g., maximum clock frequency), as well as the requirements for any externally supplied inputs (e.g., setup- and hold-time requirements with respect to the system clock, minimum pulse widths, etc.). When EDA tools are used to create a design, "timing closure" is the process by which the designer uses the tools to ensure that the design meets the timing requirements.

13.1.2 Logic Symbols

We introduced traditional symbols for flip-flops in Section 10.2. Flip-flops are always drawn as rectangular-shaped symbols and follow the same general guidelines as other rectangular-shaped symbols—inputs on the left, outputs on the right, bubbles for active levels, and so on. In addition, some specific guidelines apply to the most-used flip-flop symbols:

- A dynamic-input indicator is placed on edge-triggered clock inputs.
- Asynchronous preset and clear inputs may be shown at the top and bottom of a flip-flop symbol—preset at the top and clear at the bottom.

The logic symbols for larger-scale sequential elements, like the counters and shift register in Chapter 11, are drawn with all inputs, including presets and clears, on the left, and all outputs on the right. Bidirectional signals may be drawn on the left or the right, whichever is convenient.

Like individual flip-flops, larger-scale sequential elements use a dynamic indicator to indicate edge-triggered clock inputs. In "traditional" symbols, the names of the inputs and outputs give a clue of their function, but they are sometimes ambiguous, so you must always consult the component specification to be sure how to use an input or interpret an output.

13.1.3 State-Machine Descriptions

In Chapters 9 and 12, we used six different representations of state machines:

- Word descriptions
- State tables
- State diagrams
- ASM charts
- Transition lists
- Verilog models

You might think that having all these different ways to represent state machines is a problem—too much for you to learn! Well, they're not all that difficult to learn, but there *is* a subtle problem here.

Consider a similar problem in programming, where high-level "pseudocode" or perhaps a flowchart can be used to document how a program works. The pseudocode may express the programmer's intentions very well, but errors, misinterpretations, and typos can occur when it is translated into real code. In any creative process, inconsistencies can occur when there are multiple representations of how things work.

inconsistent state-machine representations

The same kind of inconsistencies can occur in state-machine design. As a logic designer you may document a machine's desired behavior with a 100%-correct hand-drawn state diagram, but you can make mistakes translating the diagram into an HDL model, and there used to be *lots* of opportunities for error when you had to "turn the crank" manually to translate the state diagram into a state table, transition table, excitation equations, and logic diagram.

The solution to this problem is similar to the one adopted by programmers who write self-documenting code using a high-level language. The key is to select a representation that is both expressive of the designer's intentions *and* translatable into a physical realization using an error-free, automated process. (You don't hear many programmers screaming "Compiler bug!" when their programs don't work the first time.)

So, the best solution is to write state-machine "programs" directly in an HDL, and to avoid alternate representations, other than general, summary word descriptions. When consistent coding styles and practices are used, HDL state-machine modules are easily readable and allow automated synthesis of the description into an ASIC, FPGA-, or PLD-based realization. That's one reason why we devoted so much space and energy to providing lots of Verilog-based state-machine examples in Chapter 12.

13.1.4 Timing Diagrams and Specifications

We showed many examples of timing diagrams in previous chapters. In the design of sequential circuits, most timing diagrams show the relationship between the clock and various input, output, and internal signals.

Figure 13-1 shows a fairly typical timing diagram that specifies the requirements and characteristics of input and output signals in a synchronous sequential circuit. The first line shows the system clock and its nominal timing parameters. The remaining lines show a range of delays for other signals.

For example, the second line shows that flip-flops change their outputs at some time between the rising edge of CLOCK and time t_{ffpd} afterward. External circuits that sample these signals should not do so while they are changing. The timing diagram is drawn as if the minimum value of t_{ffpd} were zero; a complete documentation package would include a timing table indicating the actual minimum, typical, and maximum values of t_{ffpd} and all other timing parameters.

The third line of the timing diagram shows the additional time, t_{comb}, required for the flip-flop output changes to propagate through combinational

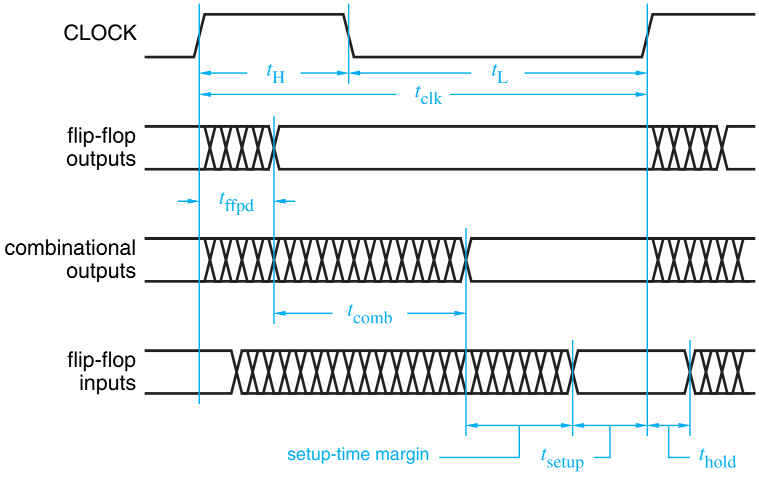


Figure 13-1 A detailed timing diagram showing propagation delays and setup and hold times with respect to the clock.

logic elements, like Moore-type outputs and excitation logic for flip-flops that use the same CLOCK signal. The excitation inputs of flip-flops and other clocked devices require a setup time of t_{setup}, as shown in the fourth line.

For proper operation of a synchronous circuit whose operation is described in Figure 13-1, we must have $t_{\text{clk}} - t_{\text{ffpd}} - t_{\text{comb}} > t_{\text{setup}}$. That is, flip-flop output changes that occur on a clock edge must propagate though the combinational logic and arrive at other flip-flop inputs at least t_{setup} before the next clock edge.

timing margin

Timing margins indicate how much "worse than worst-case" the individual components of a circuit can be without causing the circuit to fail. Well-designed systems have positive, nonzero timing margins to allow for unexpected circumstances (marginal components, power-supply noise, engineering errors, etc.) and clock skew (Section 13.3.1). Timing margins are sometimes called *timing slack*.

timing slack

setup-time margin

The value $t_{\text{clk}} - t_{\text{ffpd(max)}} - t_{\text{comb(max)}} - t_{\text{setup}}$ is called the *setup-time margin;* if this is negative, the circuit won't work. Note that *maximum* propagation delays are used to calculate setup-time margin. Another timing margin involves the hold-time requirement t_{hold}; the sum of the *minimum* values of t_{ffpd} and t_{comb} must be greater than t_{hold}, and the *hold-time margin* is $t_{\text{ffpd(min)}} + t_{\text{comb(min)}} - t_{\text{hold}}$. That is, flip-flop output changes that occur on a clock edge must propagate though the combinational logic slowly enough that they do not arrive at other flip-flop inputs sooner than t_{hold} after that same clock edge.

hold-time margin

The timing diagram in Figure 13-1 does not show the timing differences between different flip-flop inputs or combinational-logic signals, even though such differences may be significant in some circuits. For example, one flip-flop's Q output may be connected directly to another flip-flop's D input, so that t_{comb} for that path is zero, while another's may go the ripple-carry path of a 32-bit adder before reaching a flip-flop input.

When proper synchronous design methodology is used, relative timings of different flip-flop inputs are not critical, since none of these signals affect the state of the circuit until a clock edge occurs. For setup time analysis, you merely have to find the longest delay path from any flip-flop output at one clock edge to any other flip-flop input at the next clock edge to determine whether the circuit will work. However, you may have to analyze many different paths in order to find the worst-case one. Similarly, for hold-time analysis you must find the shortest delay path(s). In modern design environments, an EDA tool does all this for you, but it's important to understand how this information is determined and of course where to look for it with the tool.

Another, perhaps more common, type of timing diagram shows only functional behavior and is not concerned with actual delay amounts; an example is shown in Figure 13-2. Here, the clock is "perfect." Whether to show signal changes as vertical or slanted lines is strictly a matter of personal taste in this and all other timing diagrams, unless rise and fall times must be explicitly indicated. Clock transitions are shown as vertical lines in this and other figures in keeping with the working model that the clock is a "perfect" reference signal.

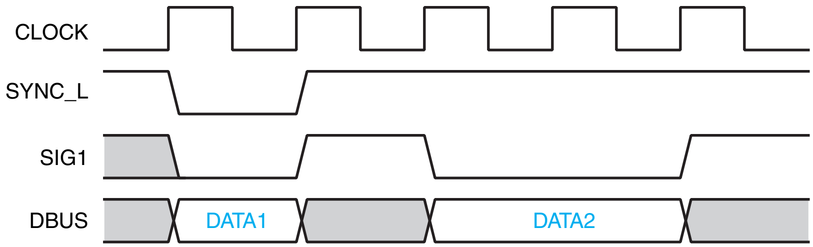


Figure 13-2
Functional timing of a synchronous circuit.

The other signals in Figure 13-2 may be flip-flop outputs, combinational outputs, or flip-flop inputs. Shading is used to indicate "don't-care" signal values; crosshatching as in Figure 13-1 could be used instead. All of the signals are shown to change immediately after the clock edge. In reality, the outputs change sometime later, and inputs may change just barely before the next clock edge. However, "lining up" everything on the clock edge allows the timing diagram to display more clearly which functions are performed during each clock period. Signals that are lined up with the clock are simply understood to change sometime *after* the clock edge, with timing that meets the setup- and hold-time requirements of the circuit. Many timing diagrams of this type appear in this book.

Table 13-1 is an example timing table for various sequential-circuit timing parameters, based on the datasheets of a few 74-series SSI and MSI flip-flops, latches, and registers. These devices are from the same CMOS logic families that we used in examples in Section 4.2.3. Even if you never do a board-level design that uses these parts, the table is representative and instructive of timing parameters used in all kinds of sequential-circuit elements in boards, ASICs, FPGAs, and PLDs, and it's provided so you can work some examples by hand.

The table contains many "t_{pd}" parameters, all of which specify delay to a Q output from an input-signal edge. For a CLK or G input of a flip-flop or latch, respectively, this is the delay from the rising edge of the active-high input signal, and is so indicated in the parameter's description. For the asynchronous preset and clear inputs of the '74 flip-flop, this is the delay from the assertion of the active-low input signal, as is likewise indicated.

NOTHING'S PERFECT

In reality, there's no such thing as a perfect clock signal. One imperfection that most designers of high-speed digital circuits have to deal with is "clock skew." As we show in Section 13.3.1, a given clock edge arrives at different circuit inputs at different times because of differences in wiring delays, loading, and other effects.

Another imperfection, a bit beyond the scope of this text, is "clock jitter." A 10-MHz clock does not have a period of exactly 100 ns on every cycle—it may be 100.05 ns in one cycle and 99.95 ns in the next. This is not a big deal in such a slow circuit, but in a 1-GHz circuit the same 0.1 ns of jitter eats up 10% of the 1-ns timing budget. And the jitter in some clock sources is even higher!

Table 13-1 Timing specifications (in ns) of selected CMOS flip-flops, latches, and registers.

			74AC @ 5.0V		***74HC @ 2.0V***			***74HC @ 4.5V***		
			Min.	***Max.***	***Typ.***	***Maximum***		***Typ.***	***Maximum***	
Part	***Function***	***Parameter***			25°C	25°C	85°C	25°C	25°C	85°C
’74	dual D flip-flop w/ preset and clear	t_{pd}, CLK↑ to Q or $\overline{\text{Q}}$	2.5	10.5	70	175	220	20	35	44
		t_{pd}, $\overline{\text{PR}}$↓ or $\overline{\text{CLR}}$↓ to Q or $\overline{\text{Q}}$	2.0	10.5	70	230	290	20	46	58
		t_{s}, D to CLK↑		3.0		100	125		20	25
		t_{h}, D from CLK↑		0.5		0	0		0	0
		t_{rec}, CLK↑ from $\overline{\text{PR}}$↑ or $\overline{\text{CLR}}$↑		0		25	30		5	6
		t_{w}, CLK low or high		5.0		80	100		16	20
		t_{w}, $\overline{\text{PR}}$ or $\overline{\text{CLR}}$ low		5.0		100	125		20	25
’373	8-bit D latch w/ 3-state outputs	t_{pd}, G↑ to Q	1.5	10.5	63	175	220	25	35	44
		t_{pd}, D to Q	1.5	10.5	50	150	190	22	30	38
		t_{s}, D to G↓		4.5		50	65		10	13
		t_{h}, D from G↓		1.0		5	5		5	5
		t_{pHZ}, $\overline{\text{OE}}$ to Q	1.0	12.5		150	190	30		38
		t_{pLZ}, $\overline{\text{OE}}$ to Q	1.0	10.0		150	190	30		38
		t_{pZH}, $\overline{\text{OE}}$ to Q	1.0	9.5		150	190	30		38
		t_{pZL}, $\overline{\text{OE}}$ to Q	1.0	9.5		150	190	30		38
		t_{w}, G high		4.5	30	80	100	10	16	20
’374	8-bit D flip-flop w/ 3-state outputs	t_{pd}, CLK↑ to Q	1.5	10.5	63	180	225	17	36	45
		t_{s}, D to CLK↑		4.5		100	125		20	25
		t_{h}, D from CLK↑		1.5		10	12		5	5
		t_{pHZ}, $\overline{\text{OE}}$ to Q	2.0	12.5	36	150	190	17	30	38
		t_{pLZ}, $\overline{\text{OE}}$ to Q	1.0	10.0	36	150	190	17	30	38
		t_{pZH}, $\overline{\text{OE}}$ to Q	1.0	9.5	60	150	190	16	30	38
		t_{pZL}, $\overline{\text{OE}}$ to Q	1.0	9.5	60	150	190	16	30	38
		t_{w}, CLK low or high		4.5		80	100		16	20
’377	8-bit D flip-flop w/ clock enable	t_{pd}, CLK↑ to Q	1.5	11.0	56	160	200	15	32	40
		t_{s}, D to CLK↑		4.5		100	125		20	25
		t_{h}, D from CLK↑		1.0		5	5		5	5
		t_{s}, $\overline{\text{EN}}$ to CLK↑		4.5		100	125		20	25
		t_{h}, $\overline{\text{EN}}$ from CLK↑		1.0		5	5		5	5
		t_{w}, CLK low or high		4.5		100	125		20	25

MINIMUM, MAXIMUM, AND TYPICAL

Note that the values listed in Table 13-1 for pulse widths and setup, hold, and recovery times (t_w, t_s, t_h, t_{red}) are the *minimum* values required for proper operation. So, the parameter value listed in a "Typical" column in Table 13-1 is the minimum value required in a typical part under typical conditions, and the value in a "Maximum" column is the highest minimum value that will be encountered in any part under the specified conditions.

You have to be careful when interpreting manufacturers' specifications, since they vary in both nomenclature and in the test specifications that they publish. For example, for most logic families Texas Instruments places the values of t_w, t_s, t_h, and t_{red} under a "Min" column instead of a "Maximum" column as we do. They don't publish a "typical" value of these parameters, while some other manufacturers of the same part do. And different manufacturers may have different definitions for "typical," and slightly different specifications for the same part. That's another good reason for engineers to provide generous timing margins when designing with off-the-shelf components.

Setup and hold times are also important parameters for sequential elements. The D flip-flops in the table all have t_s and t_h specs for the D input from the triggering edge of the clock (i.e., rising edge for these devices). The '377 also has setup and hold specs for the clock-enable input, which is also sampled at the clock edge. Note that you may occasionally see a *negative* hold time spec. That means that the D input is allowed to change *before* the triggering clock edge by the specified amount.

The D latch in the table also has setup and hold specifications, but they are for the falling edge of the active-high enable input G. As a consequence of the device's latching behavior, there is a propagation delay to the Q output from any edge of D or the rising edge of G, but there are also setup and hold requirements to reliably latch the data when G is negated.

All of the devices have minimum pulse widths specified for their "control" inputs—clock, latch-enable, preset, and clear. The Q outputs of two devices, the '373 and '374, have three-state functionality. As explained in Section 7.1, such outputs can be put into a high-impedance state where they are effectively disconnected from the signal lines that they would otherwise drive. Therefore, they have timing parameters that specify the delay from the output-enable input (OE) to moving the output between the high-impedance ("hi-Z") state and one of the "active" states, HIGH or LOW. These parameters are not particular to sequential circuits; they exist in any device with three-state outputs.

For board-level design, keep in mind that all of the specifications in Table 13-1 are merely representative; for a component's exact numbers *and* their definitions, you must consult the data sheet for the particular part, published by its manufacturer.

CROSS YOUR FINGERS

Older, slower logic families, including the 74HC family used in Table 13-1, may not specify minimum propagation delays for flip-flops and combinational logic. This makes it impossible to accurately calculate hold-time margins using the formula given earlier in this subsection. However, the hold-time margin will still be nonnegative even assuming worst-case minimum delays of zero if the flip-flops have a hold time requirement of zero.

Alternatively, you can use the rule of thumb that minimum propagation delays are no more than 20–25% of typical, calculate the hold-time margin that way, and cross your fingers. In an EDA design environment for FPGAs and ASICs, the tools will provide much better estimates, but then you still have other complications to deal with, like on-chip clock skew, to be discussed in Section 13.3.1.

13.2 Synchronous Design Methodology

synchronous system

In a *synchronous system*, all flip-flops are clocked by the same common clock signal, and asynchronous preset and clear inputs are not used, except for system initialization. Although all the world does not march to the tick of a common clock, within the confines of a digital system or subsystem we can try to make it so. When we must interconnect digital systems or subsystems that use different clocks, we can usually identify a limited number of asynchronous signals that need special treatment, as we'll show in Section 13.3.3.

Races and hazards are not problems in synchronous systems, for two reasons. First, the only circuits that might be subject to races or essential hazards are predesigned elements like discrete flip-flops or ASIC cells, guaranteed by their manufacturers to work properly. Second, even though the combinational circuits that drive flip-flop control inputs may contain static, dynamic, or function hazards, these hazards have no effect, since the control inputs are sampled only *after* the hazard-induced glitches have had a chance to settle out.

Aside from designing the functional behavior of each state machine, the designer of a practical synchronous system or subsystem must perform three other well-defined tasks to ensure reliable system operation:

1. Minimize and determine the amount of clock skew in the system, as discussed in Section 13.3.1.
2. Ensure that flip-flops have positive setup- and hold-time margins, including an allowance for clock skew, as described in Section 13.1.4.
3. Identify asynchronous input signals, synchronize them with the clock, and ensure that the synchronizers have an adequately low probability of failure, as described in Sections 13.3.3 and 13.4.

Before we get into these issues, this section introduces a general model for synchronous system structure and gives an example.

13.2.1 Synchronous System Structure

The design examples that we gave in Chapter 12 were either individual or decomposed state machines where each machine had a small number of states. If a sequential circuit has more than a few flip-flops, then it's not desirable (and often not possible) to treat the circuit as a single, monolithic state machine, because the number of states would be too large to handle.

data unit

control unit

Fortunately, most digital systems or subsystems can be partitioned into two or more parts. Whether the system processes numbers, digitized voice signals, or a stream of spark-plug pulses, a certain part of the system—which we'll call the *data unit*—can be viewed as storing, routing, combining, and otherwise just generally processing "data." Another part, which we'll call the *control unit,* can be viewed as starting and stopping actions in the data unit, testing conditions, and deciding what to do next according to circumstances. In general, only the control unit must be designed as a state machine. The data unit and its components are typically handled at a higher level of abstraction, such as:

- *Combinational functions*. These include arithmetic and logic units, comparators, and other operations that combine or modify data.
- *Registers*. A collection of flip-flops is loaded in parallel with many bits of "data," which can then be used or retrieved together.
- *Specialized sequential functions*. These include multibit counters and shift registers, which increment or shift their contents on command. They may also include very complex sequential functions, like data encryption.
- *Read/write memory.* Individual latches or flip-flops in a collection of the same can be written or read out.

The first topic above was covered in Chapters 7 and 8. The next two were in Chapters 10 and 11, and the last is in Chapter 15.

command input

condition input

Figure 13-3 is a general block diagram of a system with a control unit and a data unit. We have also included explicit blocks for input and output, but we could have just as easily absorbed these functions into the data unit itself. The control unit is a state machine whose inputs include *command inputs* that indicate how the machine is to function, and *condition inputs* provided by the data unit. The command inputs may be supplied by another subsystem or by a user to set the general operating mode of the control state machine (RUN/HALT, NORMAL/TURBO, etc.), while the condition inputs allow the control state-machine unit to change its behavior as required by circumstances in the data unit (ZERO_DETECT, MEMORY_FULL, etc.).

A key characteristic of the structure in Figure 13-3 is that the control, data, input, and output units all use the same common clock. Figure 13-4 illustrates the operations of the control and data units during a typical clock cycle:

1. Shortly after the beginning of the clock period, the control-unit state and the data-unit register outputs are valid.

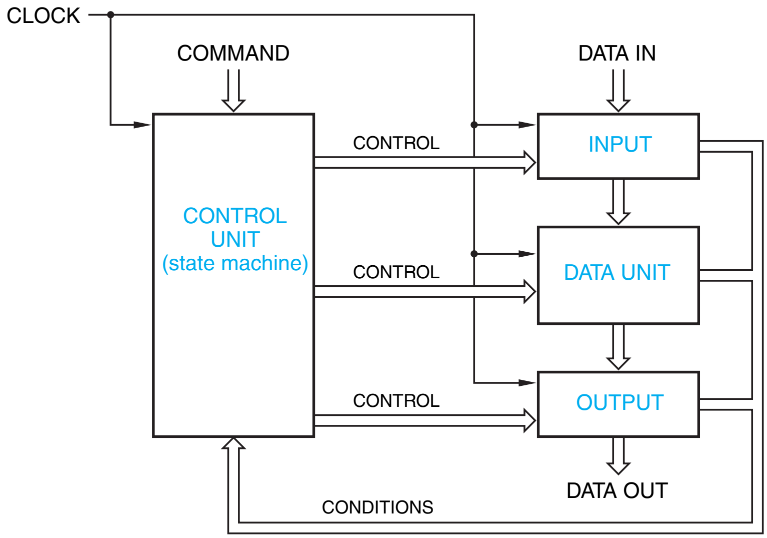


Figure 13-3 Synchronous system structure.

2. Next, after a combinational logic delay, Moore-type outputs of the control-unit state machine become valid. These signals are control *inputs* to the data unit. They determine what data-unit functions are performed in the rest of the clock period—for example, selecting memory addresses, multiplexer paths, and arithmetic operations.
3. Near the end of the clock period, data-unit condition outputs like zero- or overflow-detect are valid and are made available to the control unit.
4. At the end of the clock period, just before the setup-time window begins, the next-state logic of the control-unit state machine has determined the next state based on the current state and the command and condition inputs. At about the same time, computational results in the data unit are available to be loaded into data-unit registers.
5. After the clock edge, the whole cycle may repeat.

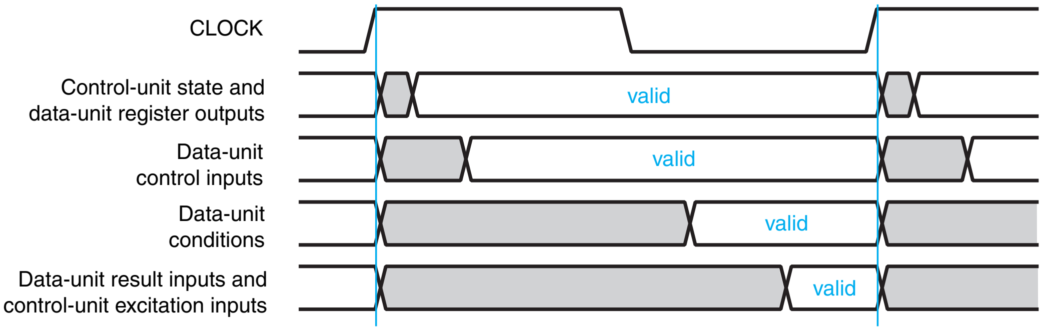

CLOCK
Control-unit state and data-unit register outputs
valid
Data-unit control inputs
valid
Data-unit conditions
valid
Data-unit result inputs and control-unit excitation inputs
valid

Figure 13-4 Operations during one clock cycle in a synchronous system.

PIPELINED MEALY OUTPUTS

Some state machines have pipelined Mealy outputs, discussed in Section 9.2.2. In Figure 13-4, pipelined Mealy outputs would typically be valid early in the cycle, at the same time as control-unit state outputs. Early validity of these outputs, compared to Moore outputs that must go through a combinational logic delay, may allow the entire system to operate at a faster clock rate.

Data-unit control inputs, which are control-unit state-machine outputs, may be of the Moore, Mealy, or pipelined Mealy type; timing for the Moore type was shown in Figure 13-4. Moore-type and pipelined-Mealy-type outputs control the data unit's actions strictly according to the current state and past inputs, which do not depend on *current* conditions in the data unit. In contrast, Mealy-type outputs may select different actions in the data unit according to *current* conditions in the data unit. This increases flexibility, but typically also increases the minimum clock period for correct system operation, since the delay path may be much longer. Also, Mealy-type outputs must not create feedback loops. For example, a signal that adds 1 to an adder's input if the adder output is nonzero causes an oscillation if the adder output is –1.

13.2.2 A Synchronous System Design Example

shift-and-add multiplier

This subsection shows the Verilog design of a *shift-and-add multiplier* for unsigned 8-bit integers which produces a 16-bit product using the algorithm of Section 2.8. The design is synchronous and hierarchical.

As illustrated in Figure 13-5, the multiplier has five modules nested three levels deep. The top-level module `VrMPY8x8` contains both a datapath module `VrMPYdata` and a control-unit module `VrMPYctrl`. The control unit contains both a state machine `VrMPYsm` and a counter `VrMPYcntr`. An "`include`" file `VrMPYdefs` contains parameter definitions used by these modules. Before you look at any details, it's important to understand the basic data-unit registers and functions that are used to perform an 8-bit multiplication, as shown in Figure 13-6:

MPY/LPROD A shift register that initially stores the multiplier and then accumulates the low-order bits of the product as the algorithm is executed.

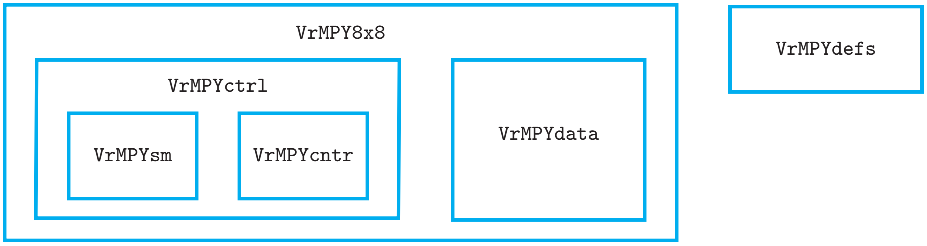


Figure 13-5 Verilog modules and "`include`" file used in the shift-and-add multiplier.

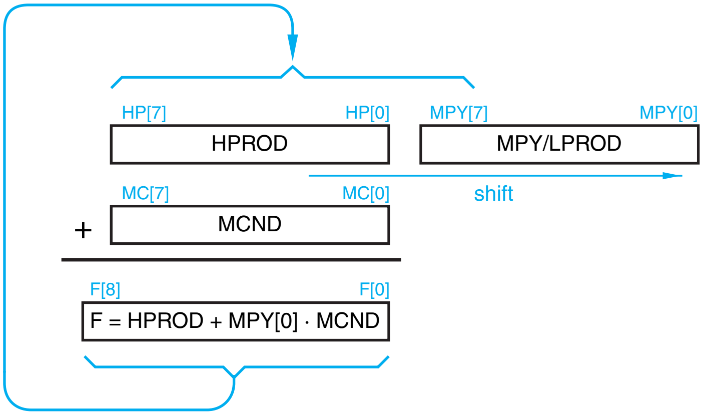


Figure 13-6 Registers and functions used by the shift-and-add multiplication

HPROD A register that is initially cleared, and accumulates the high-order bits of the product as the algorithm is executed.

MCND A register that stores the multiplicand throughout the algorithm.

F A combinational function equal to the 9-bit sum of HPROD and MCND if the low-order bit of MPY/LPROD is 1, and equal to HPROD (extended to 9 bits) otherwise.

The MPY/LPROD shift register serves a dual purpose, holding both yet-to-be-tested multiplier bits (on the right) and unchanging product bits (on the left) as the algorithm is executed. At each step it shifts right one bit, discarding the multiplier bit that was just tested, moving the next multiplier bit to be tested to the rightmost position, and loading into the leftmost position one more product bit that will not change for the rest of the algorithm.

The `VrMPY8x8` top-level module for the multiplier system has the following inputs and outputs:

CLOCK A single clock signal for the state machine and registers.

RESET A reset signal to clear the registers and put the state machine into its starting state before the system begins operation.

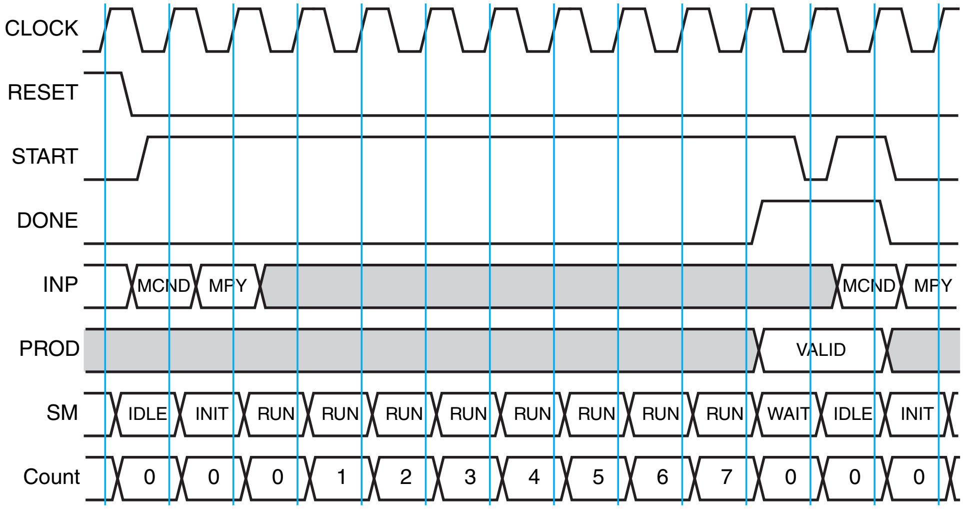
INP[7:0] An 8-bit input bus for the multiplicand and multiplier to be loaded into registers in two clock ticks at the beginning of a multiplication.

PROD[15:0] A 16-bit output bus that will contain the product at the end of a multiplication.

START An input that is asserted prior to a rising clock edge to begin a multiplication. START must be seen negated after the multiplication is complete, before is asserted again to start a new one.

DONE An output that is asserted when the multiplication is done and PROD[15:0] is valid.

Figure 13-7
Timing diagram for multiplier system.



A timing diagram for the multiplier system is shown in Figure 13-7. The first six waveforms show the input/output behavior and how a multiplication takes place in 10 clock periods as described below:

1. START is asserted. The multiplicand is placed on the INP bus and is loaded into the MCND register at the end of this clock period.
2. The multiplier is placed on the INP bus and is loaded into the MPY register at the end of the clock period.

3–10. One shift-and-add step is performed at each of the next eight clock ticks. Immediately following the eighth clock tick, DONE is asserted and the 16-bit product is available on PROD[15:0]. A new multiplication can also be started during this clock tick, but it *may* be started later.

To begin the Verilog design we define parameters in the file `VrMPYdefs.v` as shown in Program 13-1; this is "`include`'d" by all of the modules. The first group of parameters define the width of the multiplication, which is set to 8 bits but can be changed. The second group sets the encoding of the state machine's state, which is used by modules that take actions in certain states as we'll see.

The control unit `VrMPYctrl` is a decomposed state machine, as introduced in Section 12.9. Its state machine `VrMPYsm` controls the overall operation, while a counter `VrMPYcntr` counts the eight shift-and-add steps. These three Verilog modules appear on the next two pages.

The `VrMPYsm` state machine has four states for multiplier control. Multiplication begins when `START` is asserted. The machine goes to the `INIT` state and then the `RUN` state, and stays in the `RUN` state until the `MAX` input, produced by the `VrMPYcntr` module, is asserted after eight clock ticks. Then it goes to the `IDLE` or the `WAIT` state, depending on whether or not `START` has been negated yet.

Program 13-1 Definitions “include” file for shift-and-add multiplier.

```
parameter MPYwidth = 8,            // Operand width
          MPYmsb   = MPYwidth-1,   // Index of operand MSB
          PRODmsb  = 2*MPYwidth-1, // Index of product MSB
          MaxCnt   = MPYmsb,       // Number of shift-and-add steps
          CNTRmsb  = 2;            // Step-counter size [CNTRmsb:0]

parameter IDLE  = 2'b00,           // State-machine states
          INIT  = 2'b01,
          RUN   = 2'b10,
          WAIT  = 2'b11,
          SMmsb = 1,               // SM state-register size [SMmsb:SMlsb]
          SMlsb = 0;
```

Program 13-2 Verilog state-machine module `VrMPYsm`.

```
module VrMPYsm ( RESET, CLK, START, MAX, SM );
`include "VrMPYdefs.v"
  input RESET, CLK, START, MAX;
  output [1:0] SM;
  reg [1:0] Sreg, Snext;

  always @ (posedge CLK) // state memory (w/ sync. reset)
    if (RESET) Sreg <= IDLE;
    else Sreg <= Snext;

  always @ (*)                     // next-state logic
    case (Sreg)
      IDLE : if (START)            Snext <= INIT;
             else                  Snext <= IDLE;
      INIT :                       Snext <= RUN;
      RUN  : if (MAX && ~START)    Snext <= IDLE;
             else if (MAX && START) Snext <= WAIT;
             else                  Snext <= RUN;
      WAIT : if (~START)           Snext <= IDLE;
             else                  Snext <= WAIT;
      default :                    Snext <= IDLE;
    endcase

  assign SM = Sreg;  // Copy state to module output
endmodule
```

The `VrMPYcntr` module, shown in Program 13-3, counts from 0 to `MaxCnt` (`MPYwidth-1`) when the state machine is in the `RUN` state. The state-machine states and counter values during an 8-bit multiplication sequence are shown in the last two waveforms in Figure 13-7.

As shown in Program 13-4, the top-level control unit `VrMPYctrl` instantiates the state machine and the counter, and it also has a small `always` block to

Program 13-3 Verilog counter module `VrMPYcntr`.

```
module VrMPYcntr ( RESET, CLK, SM, MAX );
`include "VrMPYdefs.v"
  input RESET, CLK;
  input [SMmsb:SMlsb] SM;
  output MAX;
  reg [CNTRmsb:0] Count;

  always @ (posedge CLK)
    if (RESET) Count <= 0;
    else if (SM==RUN) Count <= (Count + 1);
    else Count <= 0;

  assign MAX = (Count == MaxCnt);
endmodule
```

implement the `DONE` output function, which requires a 1-bit register. Notice how input signals `RESET`, `CLK`, and `START` simply “flow through” `VrMPYctrl` and become inputs of `VrMPYsm` and `VrMPYcntr`. Also notice how a local signal, `SMi`, is declared to receive the state from `VrMPYsm` and deliver it both to `VrMPYcntr` and to the output of `VrMPYctrl`.

The multiplier data path logic is defined in the `VrMPYdata` module, shown in Program 13-5. This module declares local registers `MPY`, `MCND`, and `HPROD`. Besides the `RESET`, `CLK`, and `INP` inputs and the `PROD` output, which you would naturally need for the data path, this module also has `START` and the state-machine state `SM` as inputs. These are needed to determine when to load the `MPY` and `MCND` registers, and when to update the partial product (in the `RUN` state).

Program 13-4 Verilog control-unit module `VrMPYctrl`.

```
module VrMPYctrl ( RESET, CLK, START, DONE, SM );
`include "VrMPYdefs.v"
  input RESET, CLK, START;
  output reg DONE;
  output [SMmsb:SMlsb] SM;

  wire MAX;
  wire [SMmsb:SMlsb] SMi;

  VrMPYsm   U1 ( .RESET(RESET), .CLK(CLK), .START(START), .MAX(MAX), .SM(SMi) );
  VrMPYcntr U2 ( .RESET(RESET), .CLK(CLK), .SM(SMi), .MAX(MAX) );

  always @ (posedge CLK) // implement DONE output function
    if (RESET) DONE <= 1'b0;
    else if ( ((SMi==RUN) && MAX) || (SMi==WAIT) ) DONE <= 1'b1;
    else DONE <= 1'b0;

  assign SM = SMi;       // Output copy of SM state, visible to other modules
endmodule
```

Program 13-5 Verilog data-path module `VrMPYdata`.

```
module VrMPYdata (RESET, CLK, START, INP, SM, PROD );
`include "VrMPYdefs.v"
  input RESET, CLK, START;
  input [MPYmsb:0] INP;
  input [SMmsb:SMlsb] SM;
  output [PRODmsb:0] PROD;
  reg [MPYmsb:0] MPY, MCND, HPROD;
  wire [MPYmsb+1:0] F;

  always @ (posedge CLK) // implement registers
    if (RESET)         // clear registers on reset
      begin MPY  <= 0; MCND <= 0; HPROD <= 0; end
    else if ((SM==IDLE) && START)  // load MCND, clear HPROD
      begin MCND <= INP; HPROD <= 0; end
    else if (SM==INIT) MPY <= INP; // load MPY
    else if (SM==RUN) begin        // shift registers
      MPY <= {F[0], MPY[MPYmsb:1]};
      HPROD <= F[(MPYmsb+1):1];  end

  assign F = (MPY[0]) ? ({1'b0,HPROD}+{1'b0,MCND}) : {1'b0, HPROD};
  assign PROD = {HPROD, MPY};
endmodule
```

The last statement in the `VrMPYdata` module produces the `PROD` output as a combinational concatenation of the `HPROD` and `MPY` registers. Note the use of concatenation to pad the addends to nine bits in the addition operation that assigns a value to `F`.

Finally, the `VrMPY8x8` module in Program 13-6 instantiates the data-path and control-unit modules to create the multiplier system. Besides the top-level system inputs and outputs, it declares one local signal `SM` to convey the state-machine state from the control unit to the data path.

Program 13-6 Verilog top-level multiplier module `VrMPY8x8`.

```
module VrMPY8x8 (RESET, CLK, START, INP, DONE, PROD );
`include "MPYdefs.v"
  input RESET, CLK, START;
  input [MPYmsb:0] INP;
  output DONE;
  output [PRODmsb:0] PROD;
  wire [SMmsb:SMlsb] SM;

  VrMPYdata U1 ( .RESET(RESET), .CLK(CLK), .START(START), .INP(INP),
                      .SM(SM), .PROD(PROD) );
  VrMPYctrl U2 ( .RESET(RESET), .CLK(CLK), .START(START), .DONE(DONE), .SM(SM) );
endmodule
```

A test bench can be written for the multiplier as shown in Program 13-7. Its always block creates a free-running clock with a 10-ns period. Its initial block does the rest of the work, using a nested for loop to perform multiplication of all possible pairs of 8-bit numbers, taking ten clock ticks for each pair.

After multiplying each pair of numbers, the test bench compares the circuit's result (PROD) with a result calculated using Verilog's built-in multiplication operator, and prints an error message and stops the simulation if there is a mismatch. The error message includes the current simulated time and the current values of ii, jj, and PROD, as well as the expected product.

Program 13-7 Verilog multiplier test bench.

```
`timescale 1ns/100ps
module VrMPY8x8_tb ();
`include "VrMPYdefs.v"
  reg Tclk, RST, START;
  wire DONE;
  reg [MPYmsb:0] INP;
  wire [PRODmsb:0] PROD;
  integer ii, jj, cnt;

  VrMPY8x8 UUT( .CLK(Tclk), .RESET(RST), .START(START), .INP(INP),
              .DONE(DONE), .PROD(PROD) );               // instantiate UUT

  always begin    // create free-running test clock with 10 ns period
    #5 Tclk = 0;  // 5 ns high
    #5 Tclk = 1;  // 5 ns low
  end

  initial begin    // What to do starting at time 0
    RST = 1; START = 0; INP = 0; // Initial inputs
    #115;                        // Wait 15 ns,
    RST = 0;                     // then apply inputs and check outputs.
    for (ii=0; ii<=2**MPYwidth-1; ii=ii+1)  // Try all 256x256 combinations
      for (jj=0; jj<=2**MPYwidth-1; jj=jj+1) begin
        START = 1; INP = ii;
        #10;                     // Wait for 10 ns
        START = 0; INP = jj;
        #10;                     // Wait for 10 ns
        for (cnt=0; cnt<=MPYwidth-1; cnt=cnt+1)
          #10;                   // Shift-and-add MPYwidth times
        if (PROD != ii*jj) begin // Display and stop on error
          $display($time," Error, ii(%d) * jj(%d), expected %d(%b), got %d(%b)",
                   ii, jj, ii*jj, ii*jj, PROD, PROD); $stop(1); end;
      end
    $stop(1);                    // end test
  end
endmodule
```

13.3 Difficulties in Synchronous Design

Although the synchronous approach is the most straightforward and reliable method of digital system design, a few nasty realities can get in the way. We'll discuss them in this section.

13.3.1 Clock Skew

Synchronous systems using edge-triggered flip-flops can be guaranteed to work properly only if all flip-flops see the triggering clock edge at the same time. Figure 13-8 shows what can happen otherwise. Here, two flip-flops are clocked by the same signal, but the clock signal seen by FF2 is delayed by a significant amount relative to FF1's clock. This difference between arrival times of the clock at different devices is called *clock skew.*

clock skew

We've named the delayed clock in Figure 13-8(a) "CLOCKD." If FF1's propagation delay from CLOCK to Q1 is short, and if the physical connection of Q1 to FF2 is short, then the change in Q1 caused by a CLOCK edge may actually reach FF2 *before* the corresponding CLOCKD edge. In this case, FF2 may go to an incorrect next state determined by the *next* state of FF1 instead of the current state, as shown in (b). If the change in Q1 arrives at FF2 only slightly early relative to CLOCKD, then FF2's hold-time specification may be violated, in which case FF2 may become metastable and produce an unpredictable output.

We can determine quantitatively whether clock skew is a problem in a given system by defining t_{skew} to be the amount of clock skew and using the other timing parameters defined in Figure 13-1 on page 676. For proper system operation, we need

$$t_{\text{ffpd(min)}} + t_{\text{comb(min)}} - t_{\text{hold}} - t_{\text{skew(max)}} > 0$$

In other words, clock skew *subtracts* from the hold-time margin that we defined in Section 13.1.4.

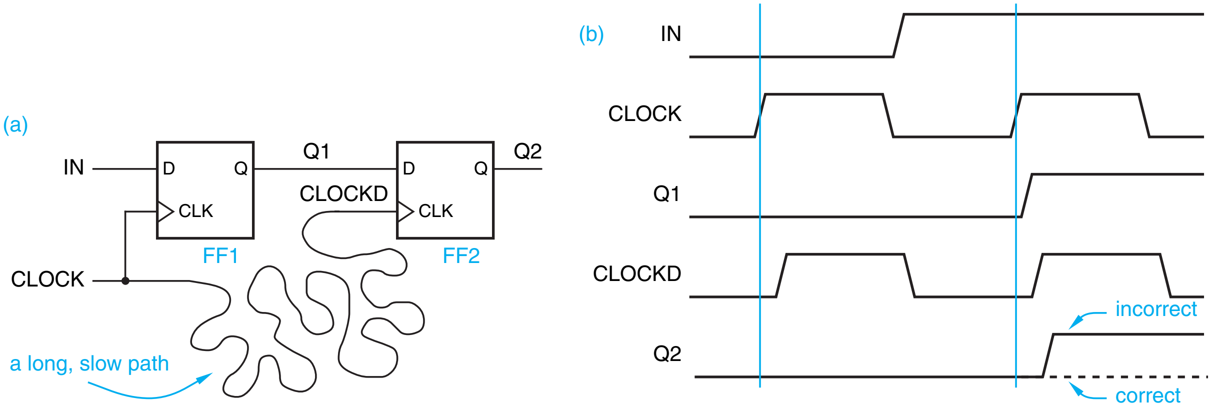


Figure 13-8 Clock-skew example.

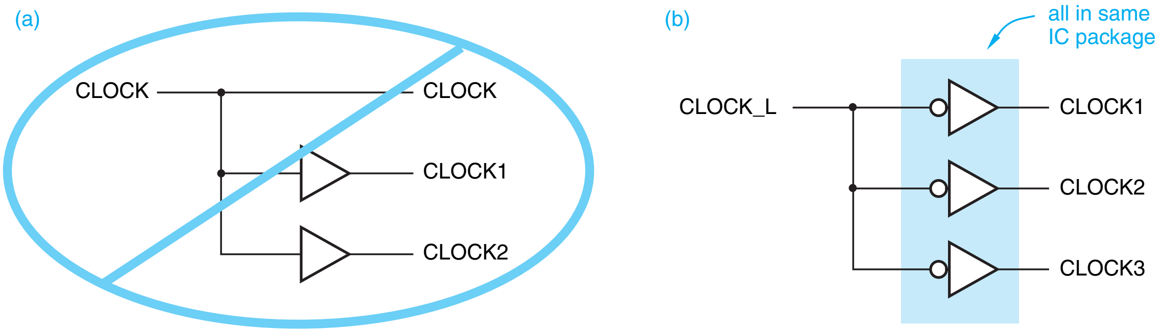


Figure 13-9 Buffering the clock: (a) excessive clock skew; (b) controllable clock skew.

Viewed in isolation, the example in Figure 13-8 may seem a bit extreme. After all, why would a designer provide a short connection path for data and a long one for the clock, when they could just run side by side? There are several ways this can happen; some are mistakes, while others are unavoidable.

In a large system, a single clock signal may not have adequate fanout to drive all of the devices with clock inputs, so it may be necessary to provide two or more copies of the clock signal. The buffering method in Figure 13-9(a) obviously produces unwanted clock skew, since CLOCK1 and CLOCK2 are delayed through an extra buffer compared to CLOCK.

The recommended buffering method is shown in Figure 13-9(b). All of the clock signals go through identical buffers and thus have roughly equal delays. Ideally, all the buffers should be part of the same IC package, as they would be in an ASIC or FPGA, so that they all have similar delay characteristics and are operating at identical temperature and power-supply voltage. To support board-level clock distribution, some manufacturers build special buffers for just this sort of application and specify the worst-case delay variation between buffers in the same package, which can be as low as a few tenths of a nanosecond.

clock tree

If even more copies of the clock signal are needed, the clock distribution scheme of Figure 13-9(b) may be extended using a second rank of buffers driven by the outputs of the first rank, with the second-rank outputs then driving the clocked devices. This scheme can be extended to additional ranks and is often called a *clock tree*.

Even the method in Figure 13-9(b) may produce excessive clock skew if one clock signal is loaded much more heavily than the other; transitions on the more heavily loaded clock appear to occur later because of increases in output-transistor switching delay and signal rise and fall times. Therefore, a careful designer tries to balance the loads on multiple clocks, looking at both DC load (fanout) and AC load (wiring and input capacitance).

Another bad situation can occur when signals on a PCB or in an ASIC are routed automatically by an EDA tool that does *not* have special facilities for

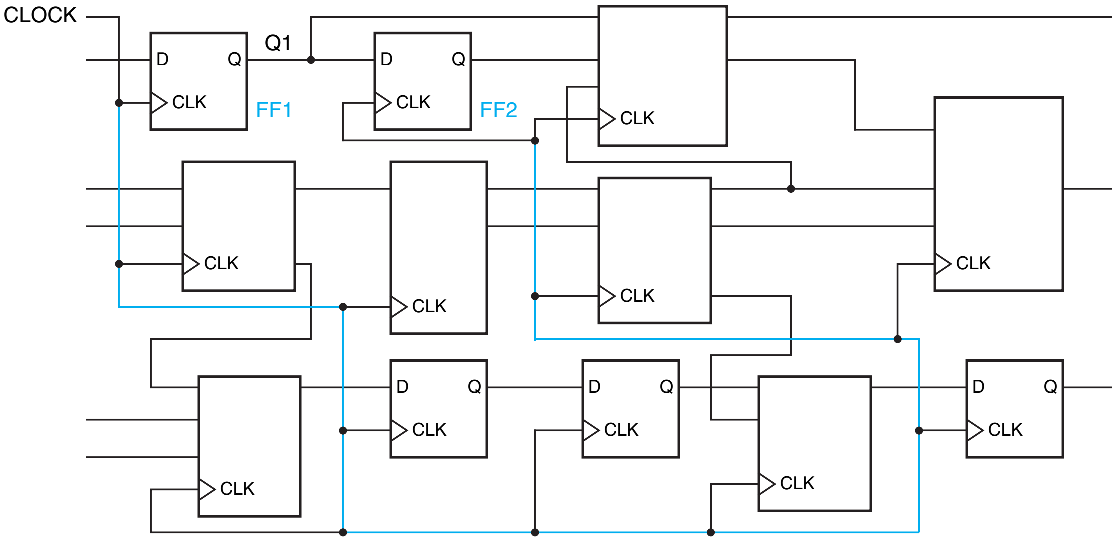


Figure 13-10 A clock-signal path leading to excessive skew.

clock routing (or has them turned off). Figure 13-10 shows a PCB or ASIC with many flip-flops and larger-scale elements, all clocked with a common CLOCK signal. The EDA tool has laid out CLOCK in a serpentine path that winds its way past all the clocked devices. Other signals are routed point-to-point between an output and a few inputs, so their paths are shorter. To make matters worse, in an ASIC or FPGA some types of "wire" may be slower than others (e.g., metal vs. polysilicon in a CMOS process technology). As a result, a CLOCK edge may indeed arrive at FF2 quite a bit later than the data change that it produces on Q1.

One way to minimize this sort of problem is to arrange for CLOCK to be distributed in a balanced, tree-like structure using the fastest type of wire, as illustrated in Figure 13-11. The idea is that the physical distances from the clock source to all of the receivers should be as close to equal as possible. Also, if one or more ranks of clock buffering are needed (as in Figure 13-9(b)), they also should be inserted in a balanced way. Usually, such a clock distribution network must be must be laid out by hand or using a specialized EDA tool. Even then, in a complex design it may not be possible to guarantee that clock edges arrive everywhere before the earliest data change. An EDA timing analysis program is typically used to detect these problems, which generally can be remedied only by inserting extra delay (e.g., buffers) in the too-fast data paths. But this may make the problems worse, because each additional component on the clock distribution path adds to delay variations and uncertainty.

In ASIC design, the EDA tools typically provide options for analyzing and optimizing on-chip clock distribution networks. A "clock-tree synthesis (CTS)" algorithm may be provided as part of the "back end," to be run after the logic

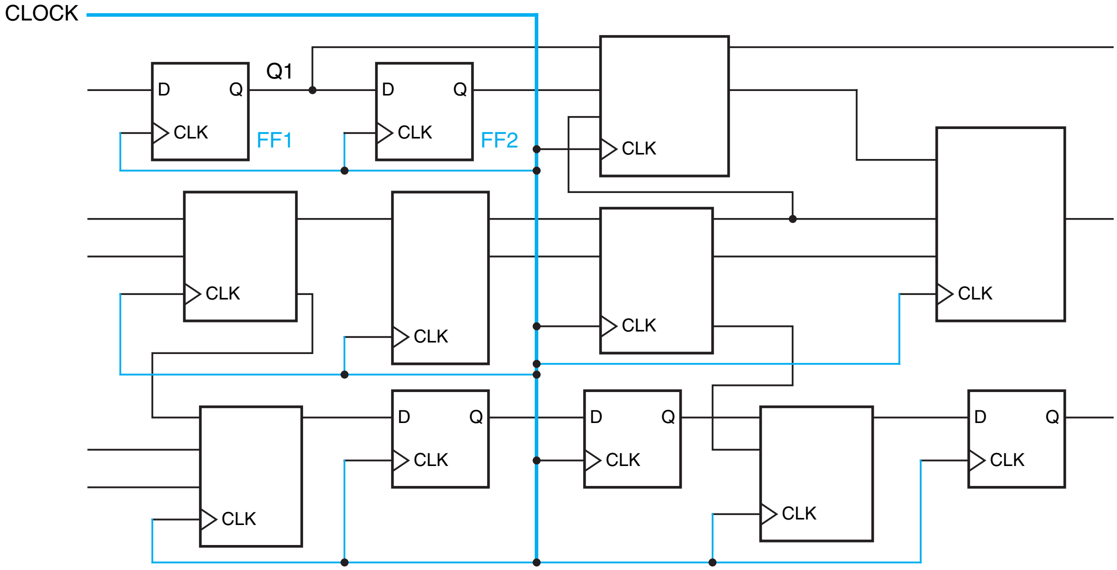


Figure 13-11 Clock-signal routing to minimize skew.

components have been placed on the chip. It attempts to create a clock distribution network with minimal skew. Total worst-case clock skew depends on many factors, not just path length. Individual buffer sizes may be adjusted in each path, changing their speeds to achieve skew goals, and all buffers' delays vary with temperature and voltage, which themselves may vary across a large chip.

Large FPGAs, like the Xilinx 7 series, typically provide pre-routed clock distribution networks on-chip. These networks can distribute one or more clocks with very low skew in multiple regions of the chip. Some FPGAs contain analog circuits (phase-locked loops) that allow the timing of different clocks with the same or related frequencies (multiples of each other) to be delayed relative to each other or to an external reference. Clock distribution in a large FPGA with

HOW NOT TO GET SKEWERED

Unbalanced wire lengths and loads are the most obvious sources of clock skew, but there are many other subtle sources. For example, *crosstalk*, the coupling of energy from one signal line into another, can cause clock skew. Crosstalk is inevitable when parallel wires are packed together tightly on a printed circuit board or in a chip, and energy is radiated during signal transitions. Depending on whether an adjacent signal is changing in the same or opposite direction as a clock, the clock's transition can be accelerated or retarded, making its transition appear to occur earlier or later.

In a large PCB or ASIC design, it's usually not feasible to track down all the possible sources of clock skew. As a result, ASIC suppliers may advise designers to provide extra setup- and hold-time margin, equivalent to many gate delays, over and above the known simulation timing results to accommodate such unknown factors.

hundreds of thousands of flip-flops and LUTs, capable of realizing multiple subsystems in a single FPGA, is a complex art—in fact, the Xilinx guide on 7-series clocking resources is over 100 pages long!

13.3.2 Gating the Clock

There are at least two situations where a designer would like to be able to effectively “turn off” the clock signal going to one or more flip-flops. The first is to prevent the flip-flop(s) from loading a new value, and the second is to save power and possibly circuit area. This is called *gating the clock* and we’ll discuss both situations.

gating the clock

Most of the sequential functions that we’ve introduced have synchronous function-enable inputs. That is, their enable inputs are sampled on the clock edge, along with the data. Our very first example of this was the edge-triggered D flip-flop with clock enable in Section 10.2.5, followed by multibit registers with synchronous load-enable inputs in Section 10.4. Other functions included counters and shift registers with synchronous load-enable, count-enable, and shift-enable inputs in Chapter 11.

Keep in mind that in all of these examples, the clock itself is not really “gated.” Rather, a multiplexer is used to select the current value of Q to apply each flip-flop’s D input in the “load disabled” case.

Nevertheless, many MSI parts, FPGA library components, and ASIC cells do not have synchronous function-enable inputs. In board-level design, for example, the 74x374 8-bit register has three-state outputs but no load-enable input. So, what can a designer do if an application requires an 8-bit register with both a load-enable input *and* three-state outputs? One solution is to use two parts, the first one having the required load-enable, followed by a three-state buffer. Another is to use a single larger part that provides both required functions. But both of these solutions have higher board area and cost compared to gating the clock.

In FPGA, CPLD, or ASIC design, the best solution is to specify exactly what is needed in the HDL-based design, and let the synthesis tools figure out the best way to implement it in the target technology. But “best” may require some definition by the designer, especially considering resource limitations in the target technology and various design goals. That brings us to the second reason for gating the clock—saving power.

In CMOS ASICs and FPGAs, clocks account for a major fraction of the total dynamic (CV^2f) power consumption, which we introduced in Section 1.8, for several reasons:

- In synchronous design, clocks go “everywhere,” so clock signal lines have a lot of capacitance because of their length. And they connect to the inputs of a lot of other circuits, so they also drive a lot of input capacitance. So, the “C” factor in their dynamic power consumption is large.

- Clocks by their very nature may run continuously, in which case the "*f*" factor is the clock's full operating frequency.
- Circuits like flip-flops which are driven by clocks may consume power internally, even if their output state is not changing.

Gating the clock is an opportunity to cut down on dynamic power consumption resulting from the second and third bullets above.

There are trade-offs to consider when deciding whether to gate the clock. Generally, it does not make sense to gate a clock that drives just one or a few clock inputs. The clock-gating circuit requires resources that consume area and power, and they may exist only in limited numbers depending on the technology, for example in FPGAs. For small cases it is typically better to use flip-flops with multiplexer-based clock enables (as in Figure 10-17 on page 507) or otherwise incorporate the required functionality in flip-flops' excitation logic. And having a large number of clocks, gated or not, also complicates the construction of clock-distribution networks and subsequent timing analysis in any design.

That said, Figure 13-12 illustrates an obvious but *wrong* approach to gating the clock. A signal CLKEN is asserted to enable the clock and is simply ANDed with the clock to produce the gated clock GCLK. This has two problems:

1. If CLKEN is a state-machine output or other signal produced by a register clocked by CLK, then CLKEN changes some time *after* CLK has already gone HIGH. As shown in (b), this produces glitches on GCLK and false clocking of the registers controlled by GCLK.
2. Even if CLKEN is somehow produced well in advance of CLK's rising edge (e.g., using a register clocked with the *falling* edge of CLK, an especially nasty kludge), the AND-gate delay gives GCLK excessive clock skew relative to the original ungated CLK, which causes more problems all around.

A method of gating the clock that generates only minimal clock skew is shown in Figure 13-13. Here, both an ungated clock and several gated clocks are generated from the same active-low master clock signal. Gates in the same IC package are used to minimize the possible differences in their delays. The CLKEN signal may change arbitrarily whenever CLK_L is LOW, which is when CLK is HIGH. That's just fine if the CLKEN signal is produced by a state machine whose outputs change right after CLK goes HIGH.

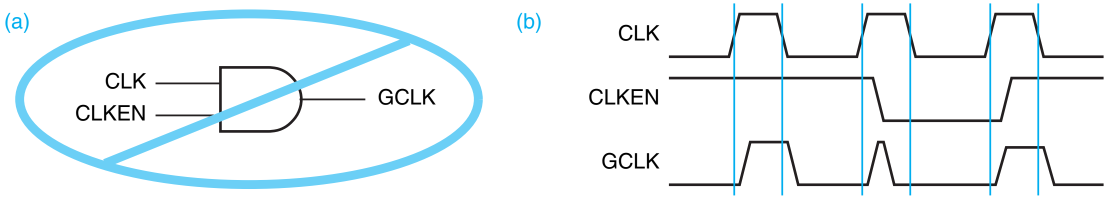


Figure 13-12 Bad clock gating: (a) simple-minded circuit; (b) timing diagram.

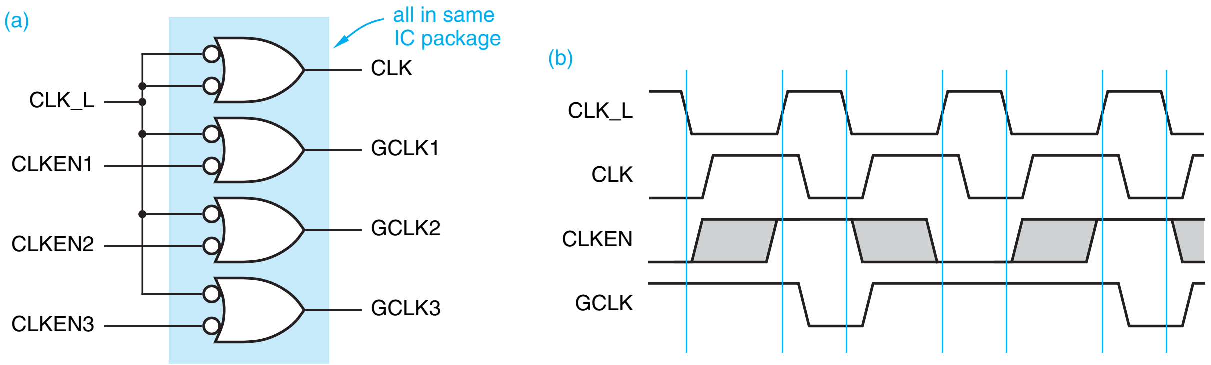


Figure 13-13 Acceptable clock gating: (a) circuit; (b) timing diagram.

The approach of Figure 13-13 is acceptable in a particular application only if the clock skew that it creates is acceptable. Furthermore, note that CLKEN must be stable during the entire time that CLK_L is HIGH (CLK is LOW). Thus, the timing margins in this approach are sensitive to the clock's duty cycle, especially if CLKEN suffers significant combinational-logic delay (t_{comb}) from the triggering clock edge. A truly synchronous function-enable input, like the 74x377's load-enable input, can be changed at almost any time during the entire clock period, up until a setup time before the triggering edge.

Another method of gating the clock combines an AND gate with a D latch as shown in Figure 13-14; we'll call this circuit a *positive-edge clock gate*. Why? As shown by the timing diagram in Figure 13-15, it latches the value of CE just before the positive (rising) edge of CLK, and allows the edge and subsequent CLK HIGH value to pass through to GCLK only if CE was asserted. When CLK is LOW, the AND gate keeps GCLK LOW also. And as long as CE is stable at an appropriate setup time before the rising edge of CLK, CEQ is stable when CLK goes HIGH and the AND gate passes the entire CLK HIGH pulse, or none of it, to the GCLK output.

positive-edge clock gate

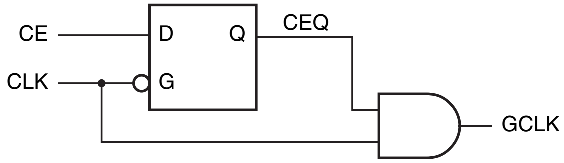


Figure 13-14 Clock gating using a latch.

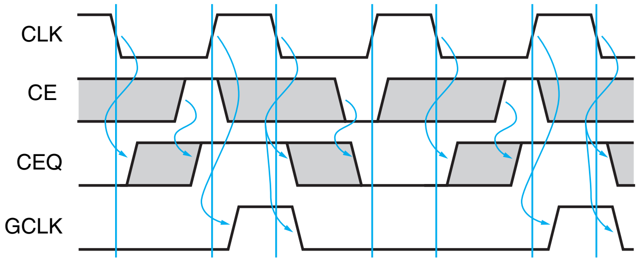


Figure 13-15 Timing of positive-edge clock-gate circuit.

Like the simple clock-gating circuit in Figure 13-12(a), the positive-edge clock gate creates excessive skew between CLK and GCLK. However, if an ungated clock is needed, one can be created with manageable skew using an instance of the positive-edge clock gate of Figure 13-14, preferably in the same IC package as the gated clocks, with CE always asserted.

The positive-edge clock gate is useful and popular enough that it appears as a predefined cell in some ASIC libraries. Also, many copies are provided in the on-chip clock distribution networks of recent Xilinx FPGAs, where it is called a BUFHCE; there are 12 instances per “clocking region” in a 7-series FPGA, which may contain as many as 24 such regions in the largest parts.

You may recall that the Xilinx 7-series slice, described in Section 10.7, has a common clock-enable signal for the eight flip-flops in the slice. This is a good opportunity for the slice to incorporate just one positive-edge clock gate to save power in designs that use the clock enable. It also saves chip area in general by eliminating the need for explicit input multiplexers on the eight clock-enabled flip-flops—the clock-gating circuit is much smaller.

Xilinx also takes the power-saving idea one step further. Not all functions in a design call out explicit clock enables, of course. But the Xilinx synthesis tool can examine a design to determine if there are any groups of flip-flops, like explicit multibit registers, whose outputs are potentially changing on every clock cycle but are not always being used in the next one. If so, it can use one or more LUTs to create clock-enable signals to load those registers only when their outputs might be used in the next cycle, and thereby save power. This uses resources of course—LUTs—but they may be available “for free” anyway. And the tool analyzes each opportunity to do this in the circuit to determine if it’s worthwhile, that is, whether the clock-enable LUTs use less power than the extra flip-flop clocking would.

13.3.3 Asynchronous Inputs

Even though it is theoretically possible to build a computer system that is fully synchronous, you couldn’t do much with it, unless you could synchronize your keystrokes and taps with a 2-GHz clock. Digital systems of all types inevitably must deal with *asynchronous input signals* that are not synchronized with the system clock.

asynchronous input signal

Asynchronous inputs are often requests for service (e.g., interrupts in a computer) or status flags (e.g., a resource has become available). Such inputs normally change slowly compared to the system clock frequency, and they need not be recognized at a particular clock tick. If a transition is missed at one clock tick, it can always be detected at the next one. The transition rates of asynchronous signals may range from less than one per second (the keystrokes of a slow typist) to 200 MHz or more (access requests for a 2-GHz multiprocessor system’s shared memory).

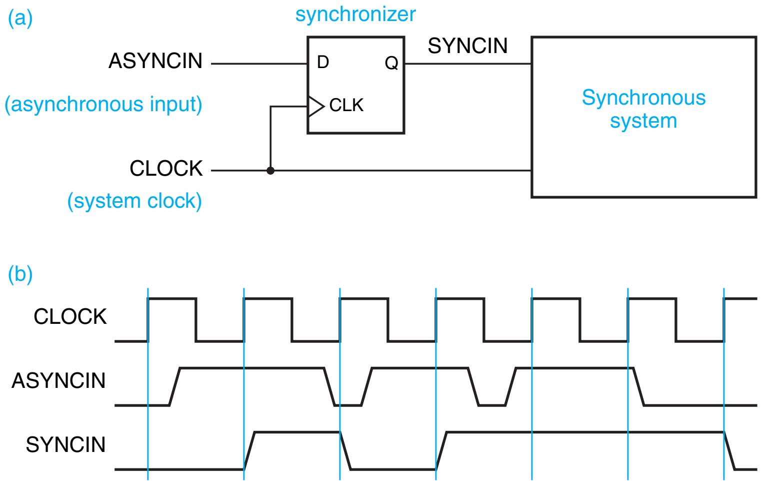


Figure 13-16
A single, simple synchronizer: (a) logic diagram; (b) timing.

synchronizer

Ignoring the problem of metastability for the moment, it is easy to build a *synchronizer,* a circuit that samples an asynchronous input and produces an output that meets the setup and hold times required in a synchronous system. As shown in Figure 13-16, a D flip-flop samples the asynchronous input at each tick of the system clock and produces a synchronous output that is valid during the next clock period.

It is essential for asynchronous inputs to be synchronized at only *one place* in a system; Figure 13-17 shows what can happen otherwise. Because of physi-

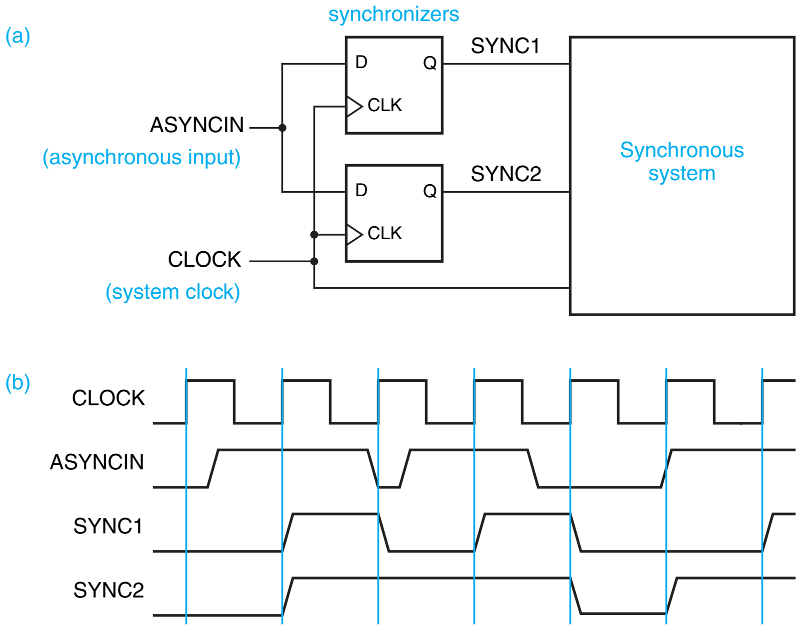


Figure 13-17
Two synchronizers for the same asynchronous input: (a) logic diagram; (b) possible timing.

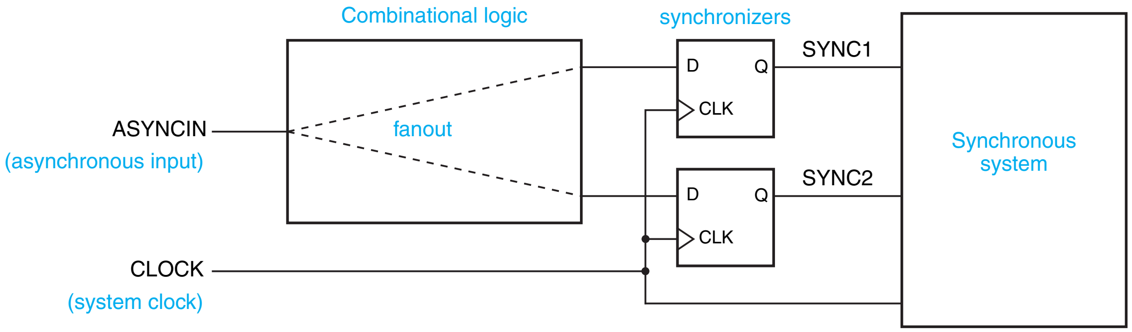


Figure 13-18 An asynchronous input driving two synchronizers through combinational logic.

cal delays in the circuit, the two flip-flops will not see the clock and input signals at precisely the same time and they don't have identical performance either. Therefore, when asynchronous input transitions occur near the clock edge, there is a small window of time during which one flip-flop may sample the input as 1 and the other may sample it as 0. This inconsistent result may cause improper system operation, as one part of the system responds as if the input were 1, and another part responds as if it were 0.

Combinational logic may hide the fact that there are two synchronizers, as shown in Figure 13-18. Since different paths through the combinational logic will inevitably have different delays, the likelihood of an inconsistent result is even greater. This situation is especially common when asynchronous signals are used as inputs to state machines, since the excitation logic for two or more state variables may depend on the asynchronous input. The proper way to use an asynchronous signal as a state-machine input is shown in Figure 13-19. All of the excitation logic sees the same synchronized input signal, SYNCIN.

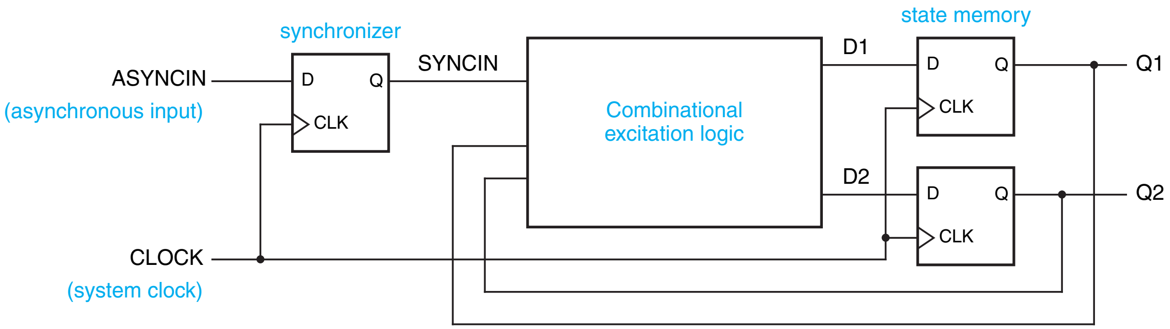


Figure 13-19 An asynchronous state-machine input coupled through a single synchronizer.

WHO CARES? As you should know, even the synchronizers in Figures 13-16 and 13-19 sometimes fail. The reason is that the setup and hold times of the synchronizing flip-flop are sometimes violated, because the asynchronous input can change at any time. "Well, who cares?" you may say. "If the D input changes near the clock edge, then the flip-flop will either see the change this time or miss it and pick it up next time; either way is good enough for me!" The problem is, there is a third possibility, discussed in the next section.

13.4 Synchronizer Failure and Metastability

We showed in Section 10.1 that when the setup and hold times of a flip-flop are not met, the flip-flop may go into a third, *metastable* state halfway between 0 and 1. Worse, the length of time it may stay in this state before falling back into a legitimate 0 or 1 state is theoretically unbounded. When other gates and flip-flops are presented with a metastable input signal, some may interpret it as a 0 and others as a 1, creating the sort of inconsistent behavior that we showed in Figure 13-17. Or the other gates and flip-flops may produce metastable outputs themselves—after all, they are now operating in the *linear* (non-digital) part of their operating range. Luckily, the probability of a flip-flop output remaining in the metastable state decreases exponentially with time, though never all the way to zero.

13.4.1 Synchronizer Failure

synchronizer failure

Synchronizer failure is said to occur if a system uses a synchronizer output while the output is still in the metastable state. You can avoid synchronizer failure only by ensuring that the system waits "long enough" before using a synchronizer's output—"long enough" that the mean time between synchronizer failures is a few orders of magnitude longer than your expected length of employment.

Metastability is more than an academic problem. Many experienced designers of high-speed digital systems have built and shipped circuits that suffer from intermittent synchronizer failures. In fact, the initial versions of many commercial ICs have had metastability problems. In previous editions of this book, we cited metastability problems in microprocessor and peripheral chips from Zilog, Intel, AMD, and Texas Instruments. It's possible to find more stories online of problem parts and products, though most of the storytellers are loathe to admit names. The nature of the problem is such that it will be noticed only if a large quantity of devices having it are in operation, which is precisely when it will cost the most to fix—by redesigning and replacing chips, boards, or systems.

There are two ways to get a flip-flop out of the metastable state:

1. Force the flip-flop into a valid logic state using input signals that meet the published specifications for minimum pulse width, setup time, and so on.
2. Wait "long enough," so the flip-flop comes out of metastability on its own.

Inexperienced designers often attempt to get around metastability in other ways, and they are usually unsuccessful.

Figure 13-20 shows an attempt by a designer who thinks that since metastability is an "analog" problem, it must have an "analog" solution. After all, Schmitt-trigger inputs (see Section 14.5.2) and capacitors can often be used to clean up noisy signals. However, rather than eliminate metastability, this circuit enhances it—with the "right" components, the circuit will oscillate forever, once it is excited by negating S_L and R_L simultaneously. (*Confession:* It was the author who tried this over 30 years ago!)

Exercise 13.26 gives an example of a valiant but also failed attempt to eliminate metastability. These examples should give you the sense that synchronizer problems can be very subtle, so you must be careful. The only way to make synchronizers reliable is to wait long enough for metastable outputs to resolve. We will answer the question "How long is 'long enough'?" later in this section.

13.4.2 Metastability Resolution Time

If the setup and hold times of a D flip-flop are met, the flip-flop output settles to a new value within time t_{pd} after the clock edge. If they are violated, the flip-flop output may be metastable for an arbitrary length of time. In a particular system design, we use the parameter t_{r}, called the *metastability resolution time,* to denote the maximum time that the output can remain metastable without causing synchronizer (and system) failure.

t_{r}
metastability resolution time

Let's be clear about t_{r}. It is *not* the time at which the flip flop is guaranteed to come out of metastability. Rather, it is the time at which the synchronizer *fails* if its output is still metastable then. And it *will* fail, some fraction of the time. When it fails, all of the other circuits that depend on it may themselves produce incorrect or inconsistent outputs.

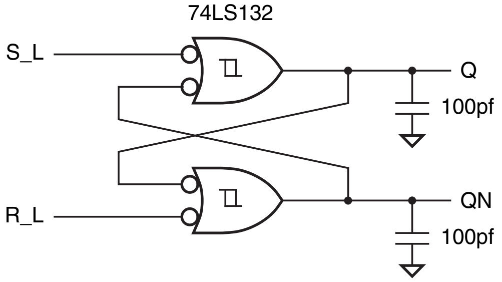


Figure 13-20
A failed attempt to build a metastable-proof $\overline{\text{S}}$-$\overline{\text{R}}$ flip-flop.

For an example of t_r, consider the state-machine structure in Figure 13-19 on page 700. The available metastability resolution time is

$$t_r = t_{clk} - t_{comb} - t_{setup}$$

where t_{clk} is the clock period, t_{comb} is the propagation delay of the combinational excitation logic, and t_{setup} is the setup time of the flip-flops used in the state memory.

t_{clk}
t_{comb}
t_{setup}

13.4.3 Reliable Synchronizer Design

The most reliable synchronizer is one that allows the maximum amount of time for metastability resolution. However, in the design of a digital system, we seldom have the luxury of *slowing down* the clock to make the system work more reliably. Instead, we are usually asked to *speed up* the clock to get higher performance from the system. As a result, we often need synchronizers that work reliably with very short clock periods. We'll present several such designs, and show how to predict their reliability.

We showed previously that a state machine with an asynchronous input, built as illustrated in Figure 13-19 on page 700, has $t_r = t_{clk} - t_{comb} - t_{setup}$. To maximize t_r for a given clock period, we should minimize t_{comb} and t_{setup}. The value of t_{setup} depends on the type of flip-flops used in the state memory; in general, faster flip-flops have shorter setup times. The minimum value of t_{comb} is zero and is achieved by the synchronizer design of Figure 13-21, whose operation we explain next.

Inputs to flip-flop FF1 are asynchronous with the clock and may violate the flip-flop's setup and hold times. When this happens, the META output may become metastable and remain in that state for an arbitrary time. However, we assume that the maximum duration of metastability after the clock edge is t_r. (We show how to calculate the probability that our assumption is correct in the next subsection.) As long as the clock period is greater than t_r plus the FF2's setup time, SYNCIN becomes a synchronized copy of the asynchronous input on the next clock tick without ever becoming metastable itself. The SYNCIN signal is distributed as required to the rest of the system.

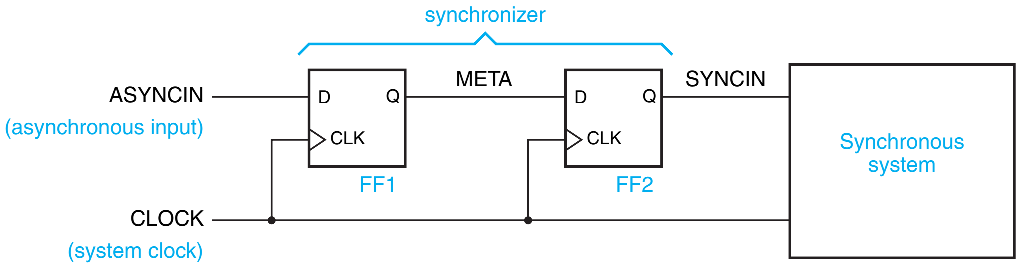


Figure 13-21 Recommended synchronizer design.

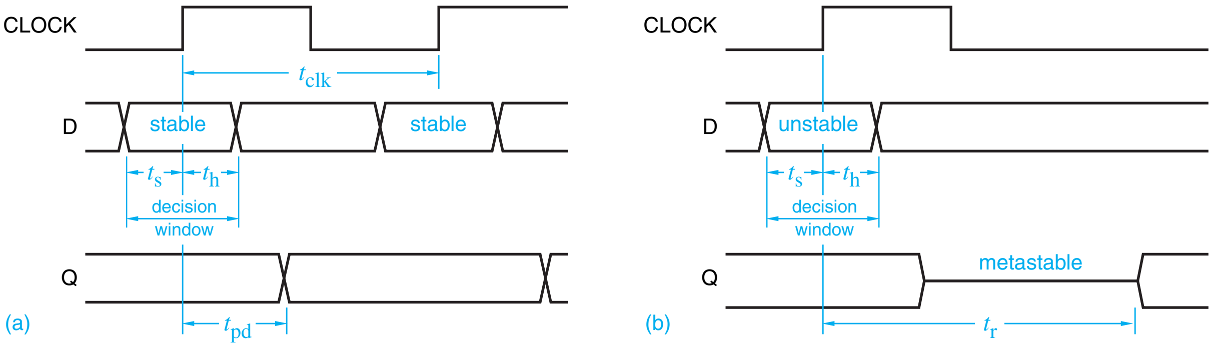


Figure 13-22 Timing parameters for metastability analysis: (a) normal flip-flop operation; (b) metastable behavior.

13.4.4 Analysis of Metastable Timing

Figure 13-22 shows the flip-flop timing parameters that are relevant to our analysis of metastability timing. The published setup and hold times of a flip-flop with respect to its clock edge are denoted by t_s and t_h, and they bracket an interval called the *decision window*, when the flip-flop samples its input and decides to change its output if necessary. As long as the D input changes outside the decision window, as in (a), the manufacturer guarantees that the output will change and settle to a valid logic state before time t_{pd}. If D changes inside the decision window, as in (b), metastability may occur and persist beyond time t_r.

decision window

Theoretical research suggests, and experimental research has confirmed, that when asynchronous inputs change during the decision window, the duration of metastable outputs is governed by an exponential formula:

$$\mathrm{MTBF}(t_r) = \frac{\exp(t_r/\tau)}{T_o \cdot f \cdot a}$$

Here MTBF(t_r) is the mean time between synchronizer failures, where a failure is said to occur if metastability persists beyond time t_r after a clock edge, where $t_r \geq t_{pd}$. This MTBF depends on *f*, the frequency of the flip-flop clock; *a*, the number of asynchronous input changes per second applied to the flip-flop; and T_o and τ, constants that depend on the electrical characteristics of the flip-flop.

f
a
T_o
τ

The 74LS74 is an SSI (discrete) positive-edge-triggered D flip-flop in bipolar TTL technology, and was one of the first devices analyzed by pioneering

DETAILS, DETAILS

In our analysis of the synchronizer in Figure 13-21 we do not allow metastability, even briefly, on the output of FF2, because we assume that the system has been designed with zero timing margins. If the system can in fact tolerate some increase in FF2's propagation delay, the MTBF will be somewhat better than predicted.

metastability researcher Thomas Chaney. He found that for a typical 74LS74, $T_o \approx 0.4$ s and $\tau \approx 1.5$ ns.

Now suppose we built a microprocessor system with a very slow (by today's standards) 10-MHz clock, and synchronize an input using the circuit of Figure 13-21 on page 703 with a pair of 74LS74 D flip-flops. If ASYNCIN changes during the decision window of FF1, the output META may become metastable until time t_r. If META is still metastable at the beginning of the decision window for FF2, then the synchronizer fails, because FF2 may have a metastable output; system operation will be unpredictable in that case.

The setup time t_s of a 74LS74 is 20 ns, and the clock period in our example microprocessor system is 100 ns, so t_r for synchronizer failure is 80 ns. If the asynchronous input changes 100,000 times per second, then the synchronizer MTBF is

$$\text{MTBF(80 ns)} = \frac{\exp(80/1.5)}{0.4 \cdot 10^7 \cdot 10^5} = 3.6 \cdot 10^{11}\,\text{s}$$

That's not bad, about 115 centuries between failures! Of course, if we're lucky enough to sell 11,500 copies of our system, one of them will fail in this way every year. But, no matter, let us consider a more serious problem.

Suppose we upgrade our system to use a faster microprocessor chip with a clock speed of 16 MHz. That's less than twice the clock period, so it doesn't seem like much of a speed-up, right? We may have to replace some components in our system to operate at the higher speed, but 74LS74s are still perfectly good at 16 MHz. Or are they? With a clock period of 62.5 ns, the new synchronizer MTBF is

$$\text{MTBF(42.5 ns)} = \frac{\exp(42.5/1.5)}{0.4 \cdot 1.6 \cdot 10^7 \cdot 10^5} = 3.1\,\text{s}$$

UNDERSTANDING *A* AND *F*

Although a flip-flop output can go metastable *only* if D changes during the decision window, the MTBF formula does not explicitly specify how many such input changes occur. Instead, it specifies the total number of asynchronous input changes per second, a, and assumes that asynchronous input changes are uniformly distributed over the clock period. Therefore, the fraction of input changes that actually occur during the decision window is "built in" to the clock-frequency parameter f—as f increases, the fraction goes up.

If the system design is such that input changes might be clustered in the decision window rather than being uniformly distributed (as when synchronizing a slow input with a fixed but unknown phase difference from the system clock), then a useful rule of thumb is to use a frequency equal to the reciprocal of the decision window (based on published setup and hold times), times a safety margin of, say, 10. But it could be much worse!

The only saving grace of this synchronizer at 16 MHz is that it's so bad, we're likely to discover the problem in the engineering lab before the product ships! Thank goodness the MTBF wasn't one year.

13.4.5 Better Synchronizers

Given the poor performance of the 74LS74 as a synchronizer at moderate clock speeds, digital designers back in the day had to consider alternatives for building more reliable synchronizers. The simplest solution, which worked for many design requirements, was and still is simply to use a flip-flop from a faster technology. Nowadays much faster technologies are available for flip-flops, whether discrete or embedded in ASICs, FPGAs, or PLDs. But at the same time, system clock frequencies have increased with technology improvements, so this isn't always a solution.

The metastability parameters T_o and τ for a few discrete flip-flops and PLDs in older technologies have been derived empirically by researchers and in a few cases they've been published by device manufacturers. Some of these are shown in Table 13-2. The metastability parameters for newer devices are harder to come by, so we'll give examples in the rest of this section using the published parameters for the older device technologies. We'll point to some of the latest work on metastability in newer technologies in the References.

Table 13-2 Metastability parameters for various devices.

Reference	***Device***	***τ (ns)***	***T_o (s)***	***t_r (ns)***
Chaney (1983)	74LS74	1.50	$4.0 \cdot 10^{-1}$	77.7
Chaney (1983)	74S74	1.70	$1.0 \cdot 10^{-6}$	66.1
Chaney (1983)	74F74	0.40	$2.0 \cdot 10^{-4}$	17.7
TI (1997)	74LSxx	1.35	$4.8 \cdot 10^{-3}$	64.0
TI (1997)	74Sxx	2.80	$1.3 \cdot 10^{-9}$	90.3
TI (1997)	74ALSxx	1.00	$8.7 \cdot 10^{-6}$	41.1
TI (1997)	74ASxx	0.25	$1.4 \cdot 10^{3}$	15.0
TI (1997)	74Fxx	0.11	$1.9 \cdot 10^{8}$	7.9
TI (1997)	74HCxx	1.82	$1.5 \cdot 10^{-6}$	71.6
TI (1997)	74ACxx	0.36	$1.1 \cdot 10^{-4}$	15.7
Cypress (1997)	PALC22V10B-20	0.26	$5.6 \cdot 10^{-11}$	7.6*
Cypress (1997)	PALCE22V10-7	0.19	$1.3 \cdot 10^{-13}$	4.4*
Xilinx (1997)	7300-series CPLD	0.29	$1.0 \cdot 10^{-15}$	5.3*
Xilinx (1997)	9500-series CPLD	0.17	$9.6 \cdot 10^{-18}$	2.3*

*t_r is *added* to the normal clock-to-out delay t_{pd}

The numbers in Table 13-2 were all derived experimentally and vary with a chip's internal circuit design and IC fabrication process, and with the measurement test setup that was used. Thus, even if published, metastability numbers can vary dramatically must be used conservatively. For example, Drill 13.6 compares the results of the previous subsection's example, which used Chaney's numbers, with the results obtained using TI's estimates of 74LSxx parameters.

Also note that different authors and manufacturers may specify some metastability parameters and measurement result differently. For example, author Chaney and manufacturer Texas Instruments (TI) measure the metastability resolution time t_r from the triggering clock edge, as in our previous subsection. On the other hand, manufacturers Cypress and Xilinx define t_r as the *additional* delay beyond the normal clock-to-output delay time t_{pd}.

The last column in the table gives a somewhat arbitrarily chosen figure of merit for each device. It is the metastability resolution time t_r required to obtain an MTBF of 1000 years when operating a synchronizer with a clock frequency of 25 MHz and with 100,000 asynchronous input changes per second. For the Cypress and Xilinx devices, their parameter values yield a value of t_r, marked with an asterisk, consistent with their own definition mentioned above.

As you can see, the 74LS74 is one of the worst devices in the table. If we replace FF1 in the 16-MHz microprocessor system of the preceding subsection with a 74ALS74, from a faster TTL family, we get

$$\text{MTBF}(42.5\text{ ns}) = \frac{\exp(42.5/1.00)}{8.7 \cdot 10^{-6} \cdot 1.6 \cdot 10^{7} \cdot 10^{5}} = 2.06 \cdot 10^{11}\text{ s}$$

If you're satisfied with a synchronizer MTBF of about 65 centuries per system shipped, you can stop here. However, if FF2 is also replaced with a 74ALS74, the MTBF gets better, since the 'ALS74 has a shorter setup time than the 'LS74, only 10 ns. With the 'ALS74, the MTBF is over 20,000 times better:

$$\text{MTBF}(52.5\text{ ns}) = \frac{\exp(52.5/1.00)}{8.7 \cdot 10^{-6} \cdot 1.6 \cdot 10^{7} \cdot 10^{5}} = 4.54 \cdot 10^{15}\text{ s}$$

Even if we ship a million systems containing this circuit, we (or our heirs) will see a synchronizer failure only once in 144 years. Now that's job security!

Actually, the margins above aren't as large as they might seem. (How large does 144 years seem *to you*?) Most of the numbers given in Table 13-2 are *averages* and are seldom specified, let alone guaranteed, by the device manufacturer. And as you've seen, calculated MTBFs are extremely sensitive to the value of τ, which in turn may depend on temperature, voltage, variations in the IC fabrication process, and the phase of the moon. So the operation of a given flip-flop, whether discrete or in an ASIC or FPGA, may be much worse (or much better) in a real system than predicted by the manufacturer.

For example, consider what happens if we increase the clock in our 16-MHz system by just 25%, to 20 MHz. Your natural inclination might be to think that metastability will get 25% worse, or maybe 250% worse, just to be conservative. But, if you run the numbers, you'll find that the MTBF using 'ALS74s for both FF1 and FF2 goes down from $4.54 \cdot 10^{15}$ s to just $3.7 \cdot 10^{9}$ s, over a million times worse! The new MTBF of about 429 years is fine for one system, but if you ship a million of them, one will fail every four hours. You've just gone from generations of job security to corporate goat!

13.4.6 Other Synchronizer Designs

We promised to describe other ways to build more reliable synchronizers. The first way we showed was to use faster flip-flops, that is, to reduce the value of τ in the MTBF equation. Having said that, the second way is obvious—to *increase* the value of t_r in the MTBF equation.

multiple-cycle synchronizer

For a given system clock, the best value we can obtain for t_r using the circuit of Figure 13-21 on page 703 is t_{clk}, if FF2 has a setup time of 0. However, we can get values of t_r on the order of $n \cdot t_{clk}$ by using the *multiple-cycle synchronizer* circuit of Figure 13-23. Here we divide the system clock by n to obtain a slower synchronizer clock and longer $t_r = (n \cdot t_{clk}) - t_{setup}$. Typically a value of $n = 2$ or $n = 3$ gives adequate synchronizer reliability.

In the figure, note that the edges of CLOCKN will lag the edges of CLOCK because CLOCKN comes from the Q output of a counter flip-flop that is clocked by CLOCK. This means that SYNCIN, in turn, will be delayed or skewed relative to other signals in the synchronous system that come directly from flip-flops clocked by CLOCK. If SYNCIN goes through additional combinational logic in the synchronous system before reaching its flip-flop inputs, their setup time may be inadequate. If this is the case, the solution in Figure 13-24 can be used. Here, SYNCIN is reclocked by CLOCK using FF3 to produce DSYNCIN, which will have the same timing as other flip-flop outputs in the synchronous system. Of

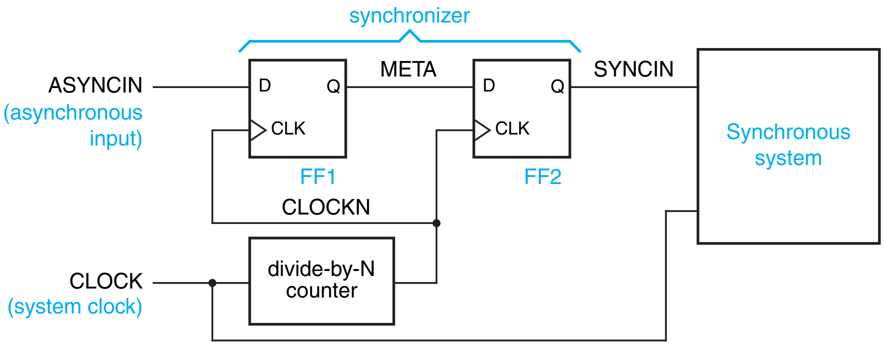


Figure 13-23 Multiple-cycle synchronizer.

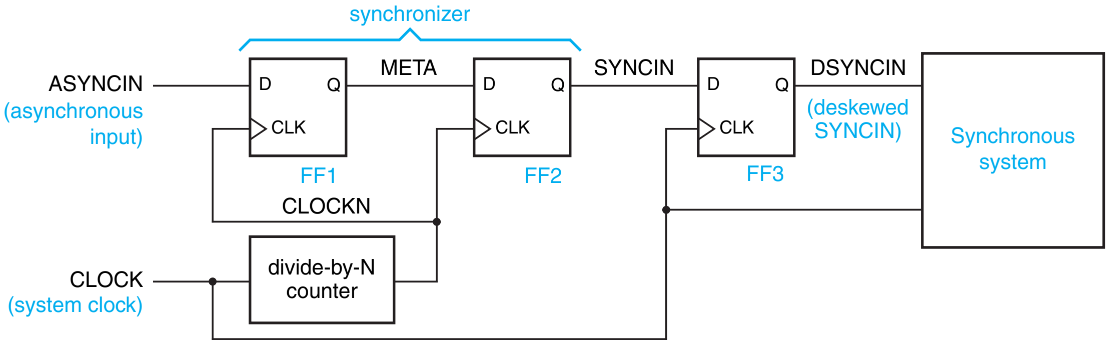


Figure 13-24 Multiple-cycle synchronizer with deskewing.

course, the delay from CLOCK to CLOCKN must still be short enough that SYNCIN meets the setup time requirement of FF3.

In an *n*-cycle synchronizer, the larger the value of *n*, the longer it takes for an asynchronous input change to be seen by the synchronous system. This is simply a price that must be paid for reliable system operation. In typical microprocessor systems, most asynchronous inputs are for external events—interrupts, DMA requests, and so on—that need not be recognized very quickly, relative to synchronizer delays. In the time-critical area of main memory access, experienced designers use the processor clock to run the memory subsystem too, if possible. This eliminates the need for synchronizers and provides the fastest possible system operation.

cascaded synchronizers

At higher frequencies, the feasibility of the multiple-cycle synchronizer design shown in Figure 13-23 tends to be limited by clock skew. For this reason, rather than use a divide-by-*n* synchronizer clock, designers often use *cascaded synchronizers*. This design approach simply uses a cascade (shift register) of *n* flip-flops, all clocked with the high-speed system clock. This approach is shown in Figure 13-25.

With cascaded synchronizers, the idea is that metastability will be resolved with some probability by the first flip-flop, and failing that, with an equal proba-

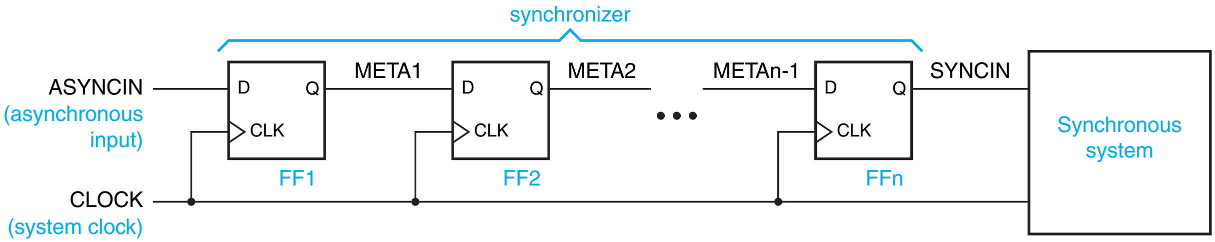


Figure 13-25 Cascaded synchronizer.

bility by each successive flip-flop in the cascade. So the overall probability of failure is on the order of the *n*th power of the failure probability of a single-flip-flop synchronizer at the system clock frequency. While this is partially true, the MTBF of the cascade is poorer than that of a multiple-cycle synchronizer with the same delay ($n \cdot t_{\mathrm{clk}}$). With the cascade, the flip-flop setup time t_{setup} must be subtracted from t_{r}, the available metastability resolution time, *n* times, but in a multiple-cycle design, it is subtracted only once.

The internal flip-flops in FPGAs, PLDs, and ASICs can of course be used in synchronizer designs, where the two or more flip-flops in Figure 13-25 are simply included in that chip. This is very convenient in most applications, because it eliminates the need for external, discrete flip-flops. However, the designer typically must provide commands or constraints to the design tools to handle the synchronizer flip-flops correctly. Otherwise, many things can go wrong, for example:

- Seeing two or more flip-flops in a row on a signal path, advanced tools may move combinational logic forward or back, into the middle of the path, to better balance delays at a higher level, which of course subtracts from t_{r}.
- The tools may place a synchronizer's two or more flip-flops far apart on the chip, providing long, slow wires between them and again subtracting from t_{r}.
- While the chip may have a special flip-flop cell that is optimized for high gain to yield fast metastability resolution (low τ), a command or constraint must be used to force its use. Somewhere in the design process, that information could be accidentally deleted or lost.
- If the design is reused in another project, the tools or the new designer may not understand the synchronizer logic or the need for it, and just remove it!

These and other potential pitfalls are discussed in an excellent paper by Steve Golson, cited in the References.

13.5 Two-Clock Synchronization Example

A very common problem in computer systems is synchronizing external data transfers with the computer system clock. A simple example is the interface between a personal computer's network interface and a 100-Mbps Ethernet link. The interface may be part of a general-purpose I/O interface subsystem, which may be connected to the processor, either on the same chip or a different chip, through a parallel bus, which we will assume for this example has a 33.33-MHz clock. Even though the Ethernet speed is approximately a multiple of the bus speed, the signal received on the Ethernet link is generated by another computer whose transmit clock is not synchronized with the receive clock in any way. Yet the interface must still deliver data reliably to the internal bus.

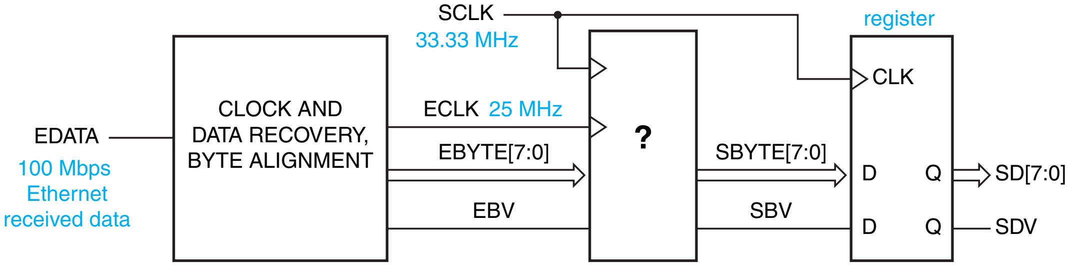


Figure 13-26 100-Mbps Ethernet synchronization.

Figure 13-26 shows the setup. NRZ serial data RDATA is received from the Ethernet at 100 Mbps. The clock-and-data-recovery block on the left uses a digital phase-locked loop (DPLL) internally to recover the original Ethernet transmit clock, and uses that to recover the 100 Mbps data. Instead of using the high-speed 100-Mhz recovered clock to communicate with the rest of the system, it divides the clock by 4 and outputs a 25 MHz ECLK, which is easier to use because of its lower frequency.

At the same time, the clock-and-data-recovery block uses an internal byte-alignment circuit to search for special patterns in the received data stream that indicate byte boundaries. When it detects one of these, it asserts the EBV output

ONE NIBBLE AT A TIME

The explanation of 100-Mbps Ethernet reception above is oversimplified, but it's sufficient for discussing the synchronization problem. In reality, the received data rate is 125 Mbps, where each 4 bits of user data is encoded as a 5-bit symbol using a so-called 4B5B code. By using only 16 out of 32 possible 5-bit codewords, the 4B5B code guarantees that regardless of the user data pattern, the bit stream on the wire will have a sufficient number of transitions to allow clock recovery. Also, the 4B5B code includes a special code that is transmitted periodically to allow nibble (4-bit) and byte synchronization to be accomplished very easily.

Since the Ethernet data is decoded 4 bits at a time, the original 100 Mbps Ethernet "MII" (Media-Independent Interface) was not too different from the one shown in Figure 13-26, except that it delivered the data 4 bits at a time; hence our use of a 25 MHz ECLK. A later interface, the "RMII," bumped the clock speed to 50 MHz and reduced the interface's pin count by delivering only 2 bits at a time.

If the entire Ethernet interface including clock and data recovery is integrated on a single chip, as it usually is nowadays, the designers can structure it any way they want to, as long as it works. In the end, the interface has to be able to deliver received data to the system that uses it, synchronized with the local system clock, one byte wide per tick, or even wider to accommodate 1 Gbps Ethernet data rates. So the details of a real 100-Mbps Ethernet synchronizer may be different from what we present here, but the general principles still apply.

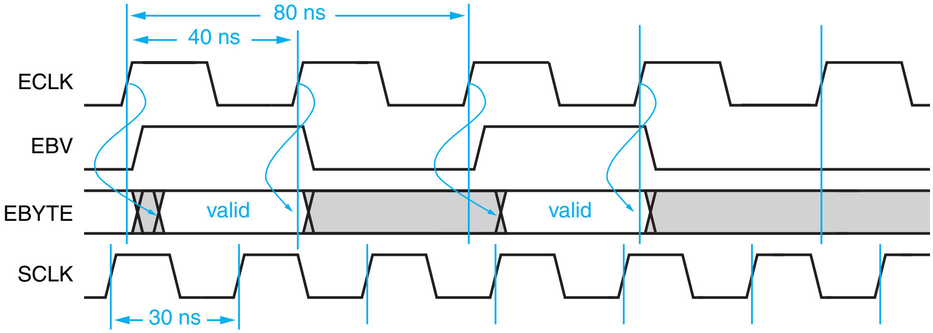


Figure 13-27 Ethernet interface and system clock timing.

and places the received byte on the EBYTE[7:0] output. Based on the now known byte alignment, it asserts EBV and places each subsequent byte on EBYTE as it is received, typically one byte per two clock ticks.

As shown in the timing diagram in Figure 13-27, transitions on both EBV and EBYTE occur on the rising edge of ECLK. The byte on EBYTE is valid only during the clock period in which EBV is asserted. Since the Ethernet data rate of 100 Mbps is equivalent to 12.5 MBps, EBV should be asserted and a new byte should be seen on every second clock tick when Ethernet data is being received.

The rest of the system is clocked by a 33.33 MHz clock SCLK. We need to transfer each received byte EBYTE[7:0] into an interface register in SCLK's domain for further processing. How can we do it?

Since the 33.33 MHz SCLK is faster than the 25-MHz ECLK, it might seem like we could just sample EBV with SCLK and grab EBYTE if we see that EBV is asserted. This is wrong in several ways:

- During any given 40-ns EBV HIGH tick, we might sample EBV once or twice, depending on the relative clock alignment at that time. How do we know if we have one new byte or two?
- From the timing diagram in Figure 13-27, it looks like EBV is never asserted for two ECLK periods in a row. So if SCLK sees it asserted two ticks in a row, couldn't we ignore the second one? No. Even though with 100 Mbps Ethernet, the upstream interface should never have to deliver two bytes in a row at 25 MHz, we assume that its designers won't guarantee that; they might want to use it differently in some situation still to be determined.
- Even if we overcame the previous two issues, if the one and only SCLK edge with EBV HIGH arrives late in the cycle (30–35 ns into it), we don't have a whole lot of time for metastability resolution before we have to do something with EBYTE. It's always better to allow more time for metastability resolution if we can afford it, which we can in this application.
- Even if 5–10 ns is enough time for metastability resolution in the target technology, it's a bad idea to rely on relative clock timings. Theoretically, we could operate SCLK a lot slower, just a little faster than 12.5 MHz, and

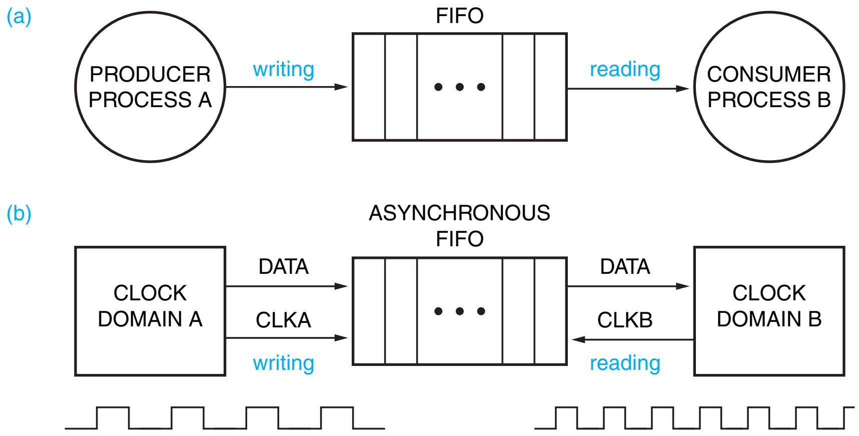


Figure 13-28 FIFOs: (a) typical software FIFO; (b) asynchronous hardware FIFO.

still be able to absorb all of the data bytes from a 100-Mbps Ethernet, with the right design. What if we *did* need to slow down SCLK sometimes, whether for power saving or for debugging?

The typical and most well understood solution to this problem is to use a *first-in, first-out buffer*, or *FIFO*. You may already be familiar with FIFOs in software programming, where they may be used as shown in Figure 13-28(a). One process, A, produces data, and another, B, consumes it. The FIFO is a data structure that receives data written by A and stores it until B is able to read it. "FIFO" means that data is read in the same order as it was written. On the average, A and B write and read at the same rate, but the FIFO is large enough to absorb short-term variations where A is temporarily producing and writing data faster than B can read and consume it. For that reason, a FIFO may also be called an *elastic buffer*.

first-in, first-out buffer (FIFO)

elastic buffer

The application of a hardware FIFO for transferring data between two clock domains, A and B, is shown in Figure 13-28(b). Here, writes to the FIFO are synchronized with CLKA, and reads are synchronized with CLKB. Either clock may be faster or slower than the other, or they may have the same approximate frequency. They may even have exactly the same frequency or be otherwise "locked" to each other, if both are derived from a common upstream clock; this relationship is called *mesosynchronous*. For example, ECLK and SCLK in Figure 13-27 could be derived from a common 100-MHz clock. However, their relative phase relationship may be unknown and may even vary with temperature and other conditions, because of large or unpredictable delays in the overall system that contains the writer and the reader. The FIFO hardware must be able to handle all these possibilities without error, and as such is usually called an *asynchronous FIFO*.

mesosynchronous clocks

asynchronous FIFO

It also goes, almost without saying, that the FIFO must be large enough to absorb the short-term variations in writing and reading. Just as in the software FIFO, some kind of "flow control" is usually provided at a higher level to stop the writer from producing data when the reader can't take much more. When and how to do this is well beyond the scope of our discussion. Here, we will look only at a system where the receiving system is clearly capable of accepting all of the data that is written over a relatively short period, and the FIFO's job is only to absorb the delays and short-term timing variations that occur because of clock synchronization.

There are many ways to design an asynchronous FIFO, and none of them are particularly easy, except for the most obviously wrong ones. For the problem at hand (Ethernet-to-system data transfer), we should be able to use a fairly small FIFO, just a few bytes deep. So we'll take an approach that is not the world's most efficient, especially for deep FIFOs, but is relatively easy to illustrate and explain (though it still has subtleties!).

circular buffer

Figure 13-29 shows the structure of our FIFO, assuming that it's four bytes deep. But we'll design it in Verilog with parameters that are easily changed for different depths. Like a software FIFO, our design uses the idea of a *circular buffer*, a block of memory—four registers in our case—and two pointers that specify where to write it and where to read it. The write and read pointers WRPTR and RDPTR are each incremented after a corresponding operation.

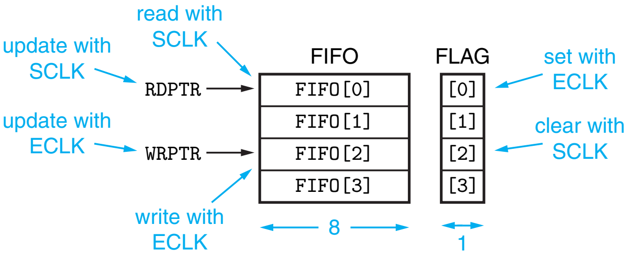
In the software FIFO, empty and full conditions are detected by comparing the pointers, but we can't easily do that in the asynchronous FIFO. Since the pointers are incremented in different clock domains, one or the other might be changing while we're doing a comparison. Here, WRPTR is incremented by ECLK when a new byte is stored in a FIFO register, and RDPTR is incremented by SCLK when a byte is read.

Instead, as shown in Figure 13-29, we'll use a single bit "FLAG" alongside each FIFO register. We'll set FLAG in the ECLK domain when the corresponding register is written, and we'll read and eventually clear it in the SCLK domain when the register is read. The FIFO must be deep enough that the reading and clearing operations of any FLAG bit are completed before it and the corresponding register are reused, but we won't look at "how deep" that needs to be until we've worked out more details of our design.

Before continuing, let us summarize the FIFO operations at a high level:

- At initialization, WRPTR and RDPTR are set to 0 to point to the first FIFO location, and all FLAG bits are cleared.

Figure 13-29 Ethernet FIFO structure for Verilog module.



- When an Ethernet byte is received, it is loaded into FIFO[WRPTR], FLAG[WRPTR] is set to 1, and WRPTR is incremented to point to the next FIFO location. All of this happens in the ECLK domain.
- The system's read-related operations all occur in the SCLK domain. The state of FLAG[RDPTR] is determined. If it's 1, FIFO[RDPTR] is read and transferred to the SBYTE output (see Figure 13-26 on page 711) for one clock period, SBV is asserted for the same period, FLAG[RDPTR] is cleared, and RDPTR is incremented to point to the next FIFO location.

A final consideration for the design is that it must be possible to support back-to-back operations within either clock domain. That is, it must be possible to write two or more bytes into the FIFO on successive ECLK cycles, and to read two or more bytes, if present in the FIFO, on successive SCLK cycles.

Reading FLAG is the one place in this design where we must consider metastability. (Note, there are other approaches to asynchronous FIFO designs that have multiple such places.) As you can understand from the last bullet above, a lot of things happen or not depending on the value of FLAG[RDPTR]; it *must* be read reliably.

There are multiple FLAG bits, one for each byte in the FIFO. Our design uses an S-R latch for each one, as shown in Figure 13-30 for one bit, FLAG[i]. According to the Verilog logic expression on S, the latch is set when the write pointer is pointing to the corresponding FIFO byte (`WRPTR==i`) and a new Ethernet byte is present and about to be written into the FIFO (`EBV==1'b1`). Since the FLAG_set[i] signal goes to an asynchronous input (S), it must be free of glitches. The easiest way to generate it is with a flip-flop that captures the expression's value on the rising edge of ECLK, since any changes in WRPTR and EBV from the preceding ECLK tick will have settled by then.

Next we bring FLAG[i] into the SCLK domain, using a pair of flip-flops FF1 and FF2 in the recommended synchronizer design of Figure 13-21 on page 703. Other logic in the SCLK domain will use the SFLAG signal, not the unsynchronized FLAG signal.

In this design, the available metastability resolution time is the timing slack in the FLAGD signal, that is, one SCLK period minus the setup time of FF2 and the propagation delay of FLAGD from SCLK to the D input of FF2. If more time is needed, one of the methods in Section 13.4.6 may be used to get it. But just

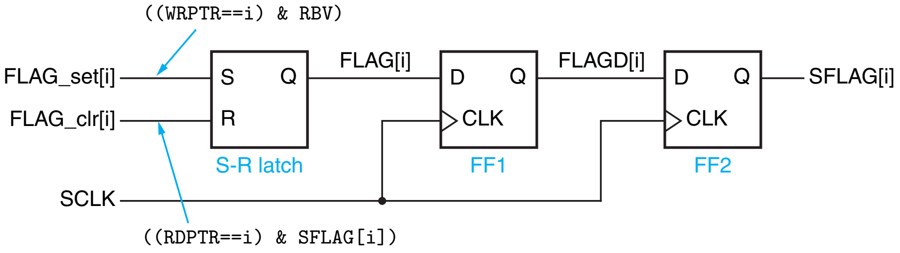


Figure 13-30 FLAG[i] latch and synchronizer.

keep in mind that any additional delay in delivering SFLAG to the logic that uses it may require more FIFO depth to accommodate the additional delay in reading a received Ethernet byte—a price that should be well worth paying.

Now that we can determine the value of FLAG[i] in the SCLK domain, we need to figure out how to reset it when it's found to be 1 and the corresponding FIFO byte has been read. Actually, we can reset the FLAG latch in a way quite analogous to setting it. As shown by the Verilog logic expression on the R input, FLAG[i] is reset when the read pointer is pointing to the corresponding FIFO location (`RDPTR==i`) and a new FIFO byte is present (`SFLAG[i]==1'b1`). Like set, the reset signal must be glitch free. So we'll generate it with a flip-flop that captures the expression's value on the rising edge of SCLK, when any changes in RDPTR and SFLAG[i] from the preceding SCLK tick have settled.

All of these ideas are put together in the Verilog module of Program 13-8, which has the inputs and outputs of the "?" block in Figure 13-26 on page 711, plus `ERST` and `SRST` reset inputs in the ECLK and SCLK domains, respectively.

In the declarations, the module defines two parameters `DEP` and `WID` for the depth of the FIFO and the width needed for the pointers into it. The module instantiates another module, `VrFIFOflagsync`, that creates the FIFO `FLAG` bits and synchronizing flip-flops of Figure 13-30, as we'll describe shortly. Note at this point, however, that parameter `DEP` is passed to `FIFOflagsync` so it can create one `FLAG` bit and a pair of synchronizing flip-flops per FIFO location.

The first `always` block in Program 13-8 performs all of the operations in the ECLK domain, and all of its `reg` variables are set on the positive edge of `ECLK`. Note that even the `FLAG_set` bits which are passed to `FIFOflagsync`, one per `FLAG` bit, are registered outputs. As discussed previously, these signals are applied to the *asynchronous* set inputs of the S-R latches in `FIFOflagsync`, and must therefore be free of decoding glitches which could occur when `WRPTR` or `EBV` is changing.

At reset, the first `always` block clears all of the `FLAG_set` bits, all of the FIFO locations, and `WRPTR`. After reset, it monitors `EBV` at each `ECLK` tick. If a new byte has arrived, it stores it in the current FIFO "write" location, asserts the `FLAG_set` signal for the corresponding `FLAG` bit, and updates `WRPTR` to point to the next FIFO location.

The second `always` block in Program 13-8 performs operations needed in the SCLK domain, and all of its `reg` variables are set on the positive edge of `SCLK`. At reset, it clears `SBYTE`, `SBV`, and `RDPTR`, but sets all of the `FLAG_clr` bits to mark all of the corresponding FIFO locations as empty. After reset, at each tick it checks the synchronized `SFLAG` corresponding to the current FIFO read location. If a new byte is present, it transfers it to the `SBYTE` output, sets the `FLAG_clr` bit for the current read location, and updates `RDPTR` to point to the next FIFO location.

Program 13-8 Verilog module for Ethernet data transfer across clock domains.

```
module VrEthSync ( ECLK, SCLK, ERST, SRST, EBYTE, EBV, SBYTE, SBV );
  input ECLK, SCLK, ERST, SRST, EBV;
  input [7:0] EBYTE;
  output reg [7:0] SBYTE;
  output reg SBV;
  parameter DEP=4, PTRWID=2;
  reg [7:0] FIFO [0:DEP-1];
  wire [0:DEP-1] SFLAG;
  reg [0:DEP-1] FLAG_set, FLAG_clr;
  reg [PTRWID-1:0] WRPTR, RDPTR;
  integer i;

  VrFIFOflagsync #(.DEP(DEP)) U1 ( .SCLK(SCLK), .FLAG_set(FLAG_set),
                                   .FLAG_clr(FLAG_clr), .SFLAG(SFLAG) );
  always @ (posedge ECLK) begin          // Edge-triggered operations in ECLK domain
    if (ERST == 1'b1) begin              // Synchronous reset
      for (i=0; i<=DEP-1; i=i+1) begin   // Clear FIFO registers and FLAG_set bits
        FLAG_set[i] <= 0;
        FIFO[i] <= 8'hff;     // Not strictly needed, but prevents initial x's in sim
      end
      WRPTR <= 0;
    end
    else begin                           // normal operation
      FLAG_set[0:DEP-1] <= 0;            // Don't set any FLAG bits yet
      if (EBV==1'b1) begin               // New byte has arrived
        FIFO[WRPTR] <= EBYTE;            // Put byte into FIFO write location
        FLAG_set[WRPTR] <= 1'b1;         // Assert corresponding FLAG_set bit
        if (WRPTR == DEP-1) WRPTR <= 0;  // Advance WRPTR, with wraparound
        else WRPTR <= WRPTR + 1;
      end
    end
  end
  always @ (posedge SCLK) begin          // Edge-triggered operations in SCLK domain
    if (SRST == 1'b1) begin              // Synchronous reset
      SBYTE <= 0;                        // Not needed, but prevent initial x's in sim
      SBV <= 0;
      RDPTR <= 0;
      FLAG_clr <= {DEP{1'b1}};           // Clear the FLAG latches during reset
    end
    else begin                           // normal operation
      FLAG_clr[0:DEP-1] <= 0;            // Don't clear any FLAG bits yet
      if (SFLAG[RDPTR]==1'b1) begin      // New byte has arrived
        SBYTE <= FIFO[RDPTR];            // Get byte from FIFO read location
        SBV <= 1'b1;                     // Mark output reg SBYTE as valid
        FLAG_clr[RDPTR] <= 1'b1;         // Free up the FIFO location
        if (RDPTR == DEP-1) RDPTR <= 0;  // Advance RDPTR, with wraparound
        else RDPTR <= RDPTR + 1;
      end
      else SBV <= 1'b0;          // No new byte, so mark output reg SBYTE as invalid
    end
  end
endmodule
```

OPTIONAL BUT PROBABLY FREE

At reset, the first always block in Program 13-8 clears all of the FIFO locations. This operation is not strictly needed, because each FIFO location will not be read until the corresponding FLAG bit has been set. However, in our example, the FIFO memory is small and in most target technologies would be realized with discrete registers, where a flip-flop reset input would likely be available "for free."

In the design of a larger FIFO, you would probably implement the FIFO in a read-write memory block, and initializing each location before it is used is likely to be expensive and should be omitted. It's true that some FPGA read-write memory blocks can be cleared or even set to arbitrary initial values at power-up, when the device is configured. But there's still no "reset" input that can clear the entire memory in one step. Once the FPGA is up and running, memory can be reinitialized only by extra logic that steps through the memory and loads locations one by one.

The VrFIFOflagsync module is coded in Program 13-9. This part of the design was done in a separate module because it may be different depending on the target technology, for several reasons:

- Although the S-R latches for FLAG may be specified behaviorally, not all target technologies have an S-R-latch cell, so each latch might have to be synthesized with a feedback loop. That's fine if the target technology is an ASIC, and the synthesizer creates a pair of cross-coupled gates for each latch. But in an FPGA, there are no actual NAND or NOR gates to be cross coupled, only programmable LUTs. Implementing an S-R latch with a LUT and feedback could actually create timing-dependent errors, as explained in the box on page 720. It's important to implement all sequential elements using cells or components that are provided "natively" in the target technology, since the supplier will have thoroughly analyzed and guaranteed their correct operation including avoidance of timing hazards.
- To minimize the probability of synchronizer failure in this or any other design, the available metastability resolution time should be maximized. In Figure 13-30 on page 715, this means that the propagation delay of FLAGD from FF1 to FF2 should be minimized. Some tools have directives to tell the synthesizer to place the two flip-flops as close to each other as possible to minimize wire delays, for example in the same CLB in an FPGA. In the Xilinx Vivado tools, the ASYNC_REG property does this.
- Some target technologies, including both ASICs and FPGAs, have special cells designed especially for synchronizers, which like any other component in the technology library can be specified by an instance statement in a Verilog module. These cells may feature two or more flip-flops already wired in series with fast interconnect, and higher-gain transistors leading to a faster time constant τ in the metastability resolution formula.

Program 13-9 Verilog VrFIFOflagsync module targeted to Xilinx 7-series FPGAs.

```
module VrFIFOflagsync ( SCLK, FLAG_set, FLAG_clr, SFLAG );
parameter DEP = 4;
  input SCLK;                           // Sync on SCLK
  input [0:DEP-1] FLAG_set, FLAG_clr; // Latch set and clear
  wire [0:DEP-1] FLAG;                  // Output of FLAG latch
  output reg [0:DEP-1] SFLAG;           // Output of module
  reg [0:DEP-1] FLAGD;                  // Synchronizing flip-flop
  genvar g;

  generate
    for (g=0; g<=DEP-1; g=g+1) begin: flags   // Latch a 1 when FLAG_set is asserted
      LDCE U1 ( .G(FLAG_set[g]), .GE(1'b1), .D(1'b1),
                .CLR(FLAG_clr[g]),            // Async clear when FLAG_clr is asserted
                .Q(FLAG[g]) );
    end
  endgenerate

  always @ (posedge SCLK) begin  // Capture FLAG in SCLK domain, resolve metastability
    FLAGD <= FLAG; SFLAG <= FLAGD;
  end
endmodule
```

In the case of the VrFIFOflagsync module in Program 13-9, we have targeted Xilinx 7-series FPGAs. Here, as in every other programmable device, there is no native S-R latch component to use for the design's FLAG latch. However, the library does offer a D latch, the LDCE which we introduced in Table 10-1 on page 510. A D latch can operate just like an S-R latch if we apply S to G and latch a constant 1 when G is asserted, and apply R to the asynchronous clear, which is provided in the LDCE component. The synthesizer realizes each instantiation of the LDCE component using one of the FPGA CLB's native, programmable latch/flip-flops shown in Figure 10-33 on page 532.

So, the FIFOflagsync module uses a generate statement to instantiate one LDCE for each FLAG bit, with the corresponding FLAG_set and FLAG_clr inputs connected to each as described above. The module also uses a simple behavioral always block to create the two synchronizing flip-flops per FLAG bit.

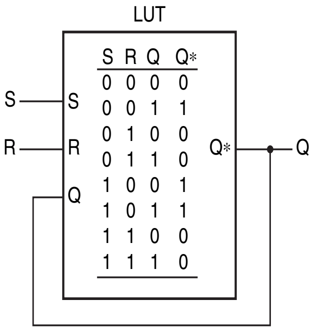
We can check the operation of the Ethernet synchronizer now by using a test bench and simulator to apply inputs and observe outputs. The idea is to initialize the circuit and then apply Ethernet bytes to the EBYTE,EBV inputs. Using a sequence of byte values that are in numerically increasing order, it is easy to spot problems where any output bytes on SBYTE,SBV are missing or out of order. The simulator also gives us the ability to look at signals that are inside the VrEthSync and VrFIFOflagsync modules for debugging purposes.

RIDDLE: WHEN IS A NAND NOT A NAND AND A NOR NOT A NOR?

As you know, an S-R latch can be built very simply in discrete logic as a pair of cross-coupled NAND or NOR gates. If you specify a reset-dominant S-R latch using a behavioral, dataflow, or even structural model in Verilog, and target it to an FPGA using Xilinx Vivado or just about any other tool, you are going to get a LUT with combinational feedback, like the one in Figure 13-31. This circuit realizes the latch's characteristic equation Q* = ~R & (S | Q), and is equivalent to the cross-coupled NOR realization of Figure 10-4 on page 500. Or is it?

Remember, the LUT is a memory, a lookup table that in this example stores the value of the characteristic equation for each of the eight possible combinations of its address inputs S, R, and Q. There is *no* guarantee that the output will remain stable while even just one input is changing between two input combinations that both produce the same output. For example, if S is 1, R is 0, and Q is 1, producing a Q* output of 1, the output may briefly go to 0 as S changes from 1 to 0. This glitch could actually set up an oscillation in the Q*-to-Q feedback loop. That can't happen when a real 2-input gate generates (S | Q).

So the answer to the riddle is, "When it's a LUT!"



S	R	Q	Q*
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Figure 13-31
S-R latch realized with a LUT.

A test-bench module that does the job is shown in Program 13-10. After the needed declarations, the module defines the HIGH and LOW times for the clock waveforms, instantiates the VrEthSync module, and creates the free-running clocks ECLK and SCLK. Notice that the instance statement specifies a FIFO depth parameter (DEP) of 3. For a depth larger than the default of 4, it would also have to specify a new value for the PTRWID parameter.

Next, the test bench asserts the reset inputs for the VrEthSync module and initializes its Ethernet inputs EBYTE, EBV. Like our other test benches targeted to Xilinx FPGAs, it waits at least 100 ns for the FPGA's internal global reset signal to be negated before starting any operations involving sequential elements. It waits an additional time to ensure that synchronous reset SRST has its effect on FLAG_clr, and then negates both module resets ERST and SRST. Then it waits a little more for any post-reset internals to settle (such as the propagation of FLAG

Program 13-10 Test bench for VrEthSync Ethernet data-transfer module.

```
`timescale 1ns/100ps
module VrEthSync_tb ();
  reg ECLK, SCLK, ERST, SRST, EBV;
  reg [7:0] EBYTE;
  wire [7:0] SBYTE;
  wire SBV;
  integer i;

parameter Ehigh = 20, Elow = 20;   // Define ECLK waveform (25 MHz)
parameter Shigh = 18, Slow = 12;   // Define SCLK waveform (33.3 MHz)

VrEthSync #(.DEP(3)) UUT ( .ECLK(ECLK), .SCLK(SCLK), .ERST(ERST),  .SRST(SRST),
            .EBYTE(EBYTE), .EBV(EBV), .SBYTE(SBYTE), .SBV(SBV) ); // instantiate UUT

always begin    // create ECLK, which starts LOW
  ECLK = 0; #Elow  ECLK = 1; #Ehigh ;
end

always begin    // create SCLK, which starts LOW
  SCLK = 0; #Slow  SCLK = 1; #Shigh ;
end

initial begin
  ERST = 1; SRST = 1;     // assert resets
  EBYTE = 8'h00; EBV = 0; // initialize received byte
  # 105 ;                 // Wait for global reset to end
  #(2*(Shigh+Slow)) ;     // Wait two more SCLKs for FLAG_clr (clocked) to take effect
  ERST = 0; SRST = 0;     //   and negate resets
  #(2*(Shigh+Slow)) ;     // Wait two more SCLKs for any more internals to settle
  for (i=1; i<=500; i=i+1) begin  // Then run for 500 received bytes
    EBYTE = i; EBV = 1'b1;        // received byte = i
    #(Ehigh+Elow)  ;              // wait one ECLK tick
    EBYTE = 0; EBV = 1'b0;        // invalidate for one tick
    #(Ehigh+Elow)  ;              // wait one ECLK tick
  end
  $stop(1);
  end
endmodule
```

NEGATING SYNCHRONOUS RESETS

The resets in this example, ERST and SRST, are applied to synchronous reset inputs of flip-flops. They may be asserted asynchronously at system initialization, but in the real system each must be negated synchronously, allowing adequate setup time to the corresponding clock. If synchronous reset inputs are negated just before the clock edge, and setup-time requirements are violated, unreliable operation may result. For example, some flip-flops may "take off" into their normal operations while others remain reset, putting the overall circuit into an inconsistent state.

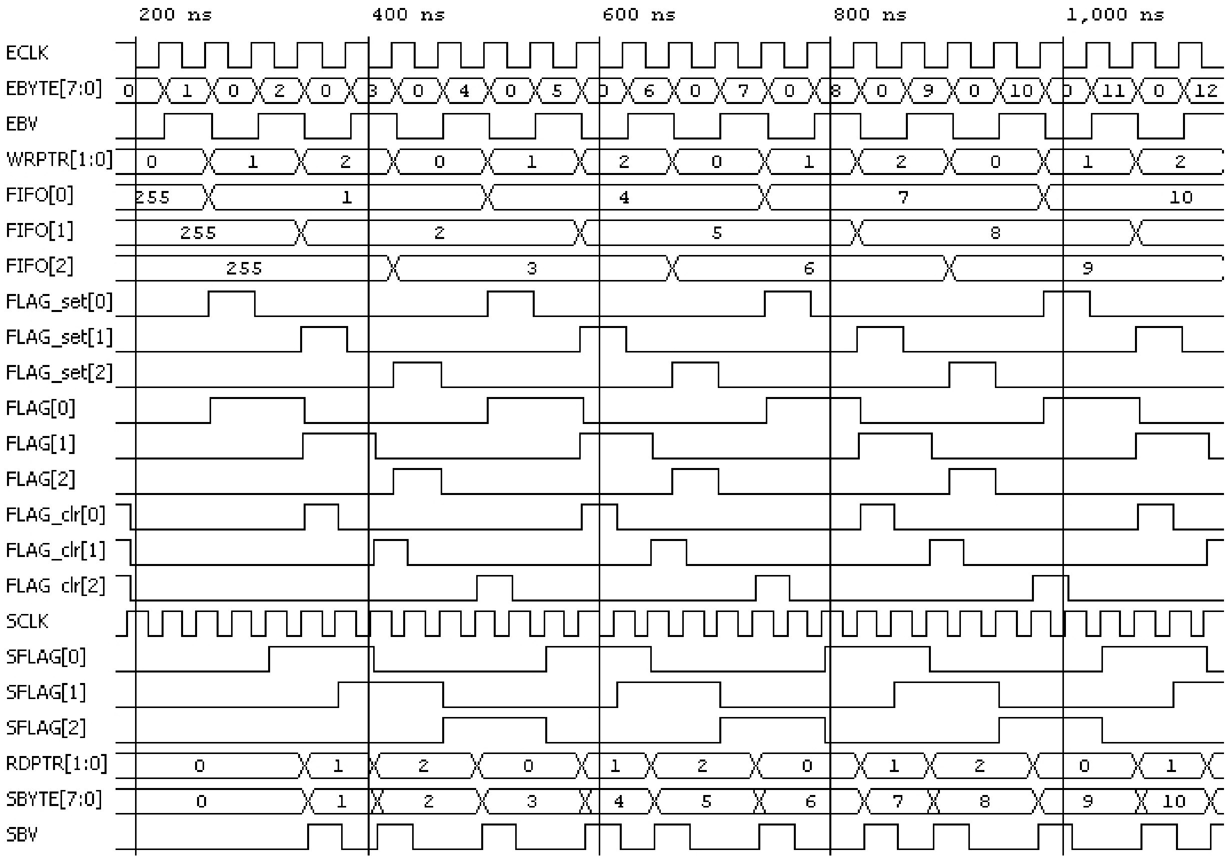


Figure 13-32 `VrEthSync` timing waveforms created by the test bench.

through the synchronizing flip-flops). Finally, it is ready to run the test, which is simply to apply a steadily increasing sequence of values to the `EBYTE`,`EBV` module inputs on alternating `ECLK` ticks, a data rate of 12.5 MBps.

Figure 13-32 shows part of the timing waveforms created by the test bench, beginning after the initial reset operations have finished. Several aspects of the circuit's operation may be noted:

- On alternating `ECLK` cycles, Ethernet data values are applied to `EBYTE` and `EBV` is asserted. Although there would be no harm in normal operation to allow `EBYTE` to be valid for two clock periods, we have purposely set it to zero on unused cycles, so an error would be easy to spot if `EBYTE` were being read at the wrong time. It's also possible to apply `x`'s on these cycles.
- The simulation clearly shows `WRPTR` advancing with wraparound, and the increasing data-byte values being written into successive FIFO locations.

X-RATED The waveforms in Figure 13-32 are from a post-implementation timing simulation, based on the actual placement and routing of the design. You can see a few places, for example at about 400 ns and 600 ns on SBYTE, where the transition between successive multibit values is not a clean "X". This occurs because different bits change at slightly different times, and the post-implementation simulation sees this. If you were to zoom in on the timing diagram at these places, you would see the timing difference between SBYTE bits is measured in only tens of picoseconds!

- For each FIFO location, the operations of FLAG_set[i] and FLAG_clr[i], changed at positive edges of ECLK and SCLK respectively, can be seen. The corresponding FLAG[i] is set as a new byte is loaded, and cleared as the byte is transferred into SBYTE and SBV is asserted.
- As each byte is read from the FIFO, RDPTR is advanced.
- All of the Ethernet bytes appear, in order, on SBYTE. If you were able to look further along the waveforms, you would see this behavior continue.

So, to answer a question that we posed at the beginning of the design, based on the simulation it appears that a 3-byte-deep FIFO is "deep enough" for reliable data transfer. Looking at both the module's Verilog code and the simulation waveforms, we can analyze the timing more closely:

- For a particular FIFO location, FLAG_set is set on the same ECLK rising edge that a byte is written into it. The corresponding FLAG bit is set shortly after that.
- There may be a delay of up to one SCLK period until the next rising SCLK edge occurs and transfers FLAG into FLAGD.
- One more SCLK period after that, SFLAG is asserted as a result.
- Assuming that RDPTR is or is about to be pointing this FIFO location, then one SCLK period later, the FIFO byte is loaded into SBYTE and FLAG_clr is asserted.
- Finally, one SCLK period later, FLAG_clr is negated, completing the activity for this FIFO location.

So, from beginning to end, a FIFO location and FLAG are used for a total of three to four SCLK periods, or 90 to 120 ns for a 33 MHz SCLK, plus certain setup times and propagation delays around the first and last edges, no more than another 5-20 ns depending on the target technology. The maximum total of about 140 ns is somewhat less than the 160 ns required to receive two bytes with 100 Mbps Ethernet, so we might even be able to get by with a FIFO depth of 2.

Looking at the waveforms in Figure 13-32, it's apparent no more than two FLAG bits, or two SFLAG bits, are ever asserted at the same time. In fact, if we

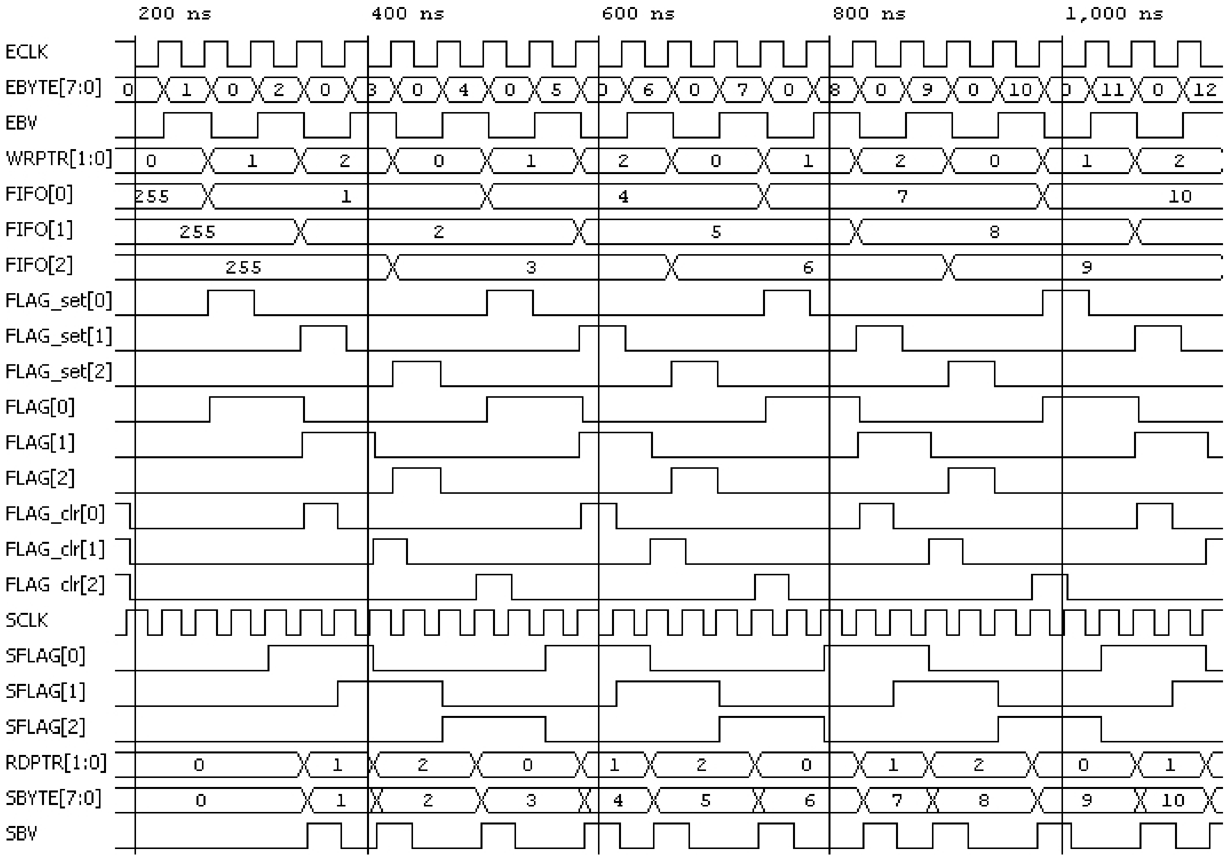


Figure 13-33 `VrEthSync` timing waveforms with `DEP=3` and slow SCLK.

resynthesize the design with parameter `DEP=2` and run the simulation, it still works. But we can't be complacent. We'd like our design to work even if `SCLK` is barely fast enough to keep up with the incoming Ethernet data rate, one byte every 80 ns. So, we can change the `SCLK` period in the test bench from 30 to 79 ns and see what happens. Not only does a FIFO depth of 2 no longer work, neither does 3, as shown in Figure 13-33. Here, `FLAG_set` for a particular FIFO location is being asserted before the `FLAG_clr` from its previous use has been negated. We need a depth of at least 4 for correct behavior, shown in Figure 13-34.

As the timing diagram in Figure 13-34 shows, once the first Ethernet output byte appears on `SBYTE`, they keep coming continuously. Further out, there is a 1-`SCLK`-tick gap at byte 35 and about every 80 `SCLK` ticks thereafter. This makes sense, since the `SCLK` period is just one part in 80 faster than the rate at which Ethernet input bytes arrive. Looking even further out in the simulation waveforms, the circuit continues to deliver Ethernet bytes on `SBYTE`, in order and with no omissions.

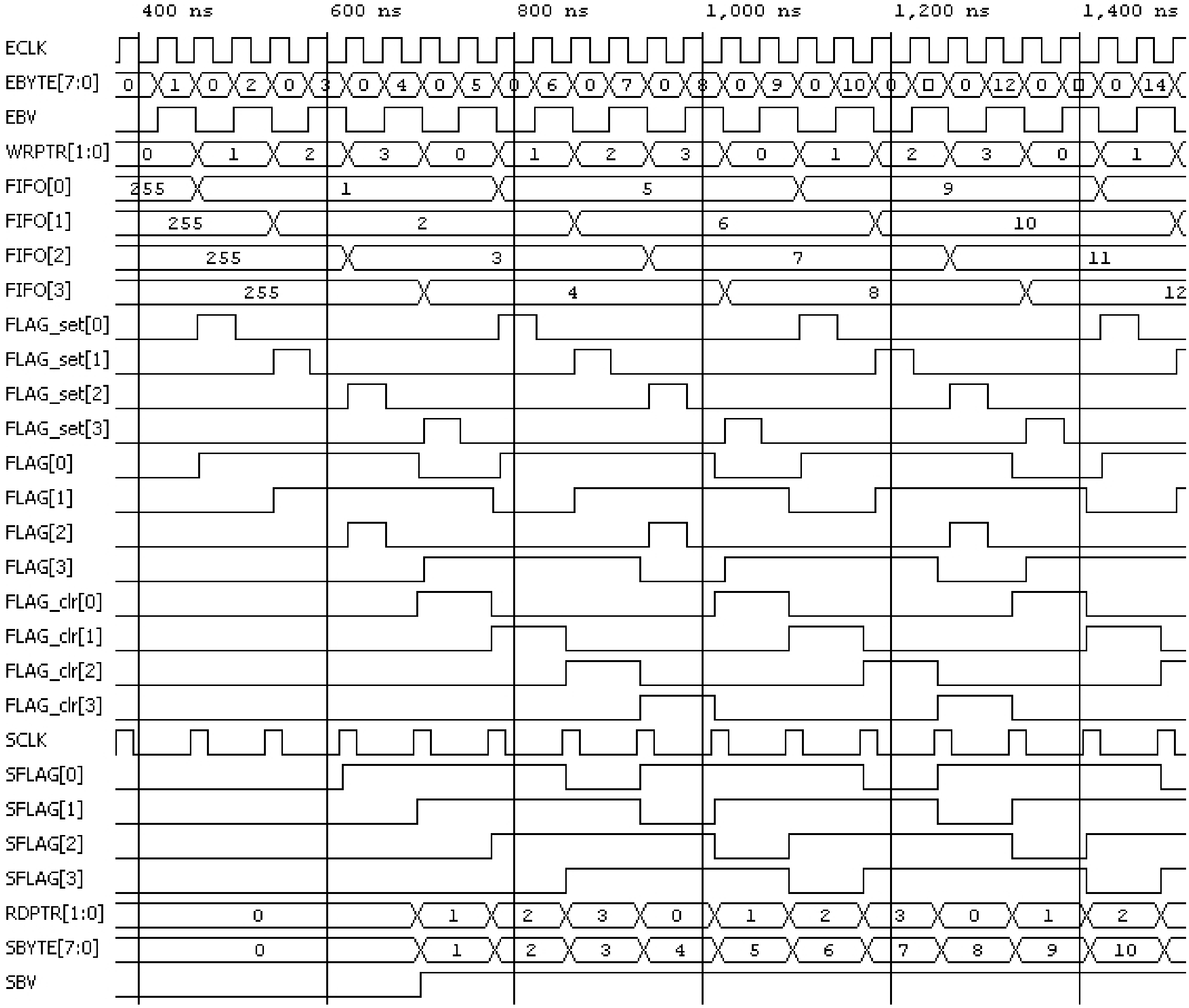


Figure 13-34 VrEthSync timing waveforms with DEP=4 and slow SCLK.

But does this really work? In multi-clock, asynchronous timing situations, it's difficult to simulate all of the possible timing alignments and situations that can occur. So, it's a good idea to do as much timing analysis as we can and see if it corroborates the simulation results. In our original analysis, we concluded that from beginning to end, a given FIFO location and FLAG are used for a total of three to four SCLK periods, plus certain setup times and propagation delays around the first and last edges. In the present design with a FIFO depth of 4, we have four SCLK periods worth of FIFO storage, but what about the "extra" delays mentioned above; why aren't they sometimes causing errors?

According to the timing diagram in Figure 13-34, at about 700 ns, there appears to be about 50 ns of timing margin between FLAG_clr[0] being negated and FLAG_set[0] being asserted to reuse and set FLAG[0] again. That may seem like plenty, but remember that ECLK and SCLK are unsynchronized, and the delays between them and everything that depends on them will vary over time. In fact, if you were to look carefully further along in the simulation, you would notice that just before 1-SCLK-tick gap at byte 35, around 3,400 ns, the margin between FLAG_clr being negated and the corresponding FLAG_set being asserted again has grown to about 80 ns (great!), but right after the gap, the margin shrinks to 3.3 ns—too close for comfort!

Seeing this, a good designer could take one or more precautionary steps:

- Increase the FIFO depth to 5, adding one full SCLK period to the available FLAG timing margins.
- Analyze the circuit further to determine what goes wrong, if anything, when FLAG_clr and the corresponding FLAG_set overlap a little. And how long is “a little?” Perhaps some overlap can be safely tolerated and nothing needs to be changed.
- Determine if the flag operations can be modified so that each flag is used for a little less time. For example, could FLAG_clr be asserted and negated one SCLK period sooner? Perhaps more margin can be obtained without increasing FIFO depth.

The first bullet above is the easiest and safest approach; the last two are left as Exercises 13.22 and 13.23.

After all that, you’re probably hoping that we’re done, but not yet—not if we want the design to work properly over a wide but plausible range of clock frequencies. In particular, what happens if SCLK is very fast relative to ECLK? Since a faster SCLK will empty the FIFO faster, we should have lots of margin compared to the previous case, but we should look at a simulation anyway.

Figure 13-35 shows the timing with a 100 MHz SCLK. With the very fast SCLK, we’ve also temporarily reduced the FIFO depth down to 2 just to simplify the timing diagram. But what’s going on? Starting near 300 ns, SBV gets asserted for four extra back-to-back cycles, and four extra bytes appear on SBYTE. This weird behavior repeats further along in the timing diagram, next at 460 ns. Why?

A close look at the timing diagram shows what’s happening. SCLK is so fast that FLAG_clr[0] is being asserted and negated before the end of the ECLK period during which FLAG_set[0] is asserted. Since FLAG[0] is a latch, albeit a reset-dominant one, it goes right back to being set as soon as FLAG_clr[0] is negated. How can we fix this?

The problem occurs within a single ECLK period, and we are specifically trying to make the circuit work properly even if SCLK is significantly faster than ECLK. So, the solution cannot rely on speeding up ECLK, or looking at half of its clock period or some similar kludge. However, we can change the method for setting the flag—instead of latching a 1 while FLAG_set is asserted, we can use an *edge-triggered* flip-flop to store a 1 on the rising edge of FLAG_set.

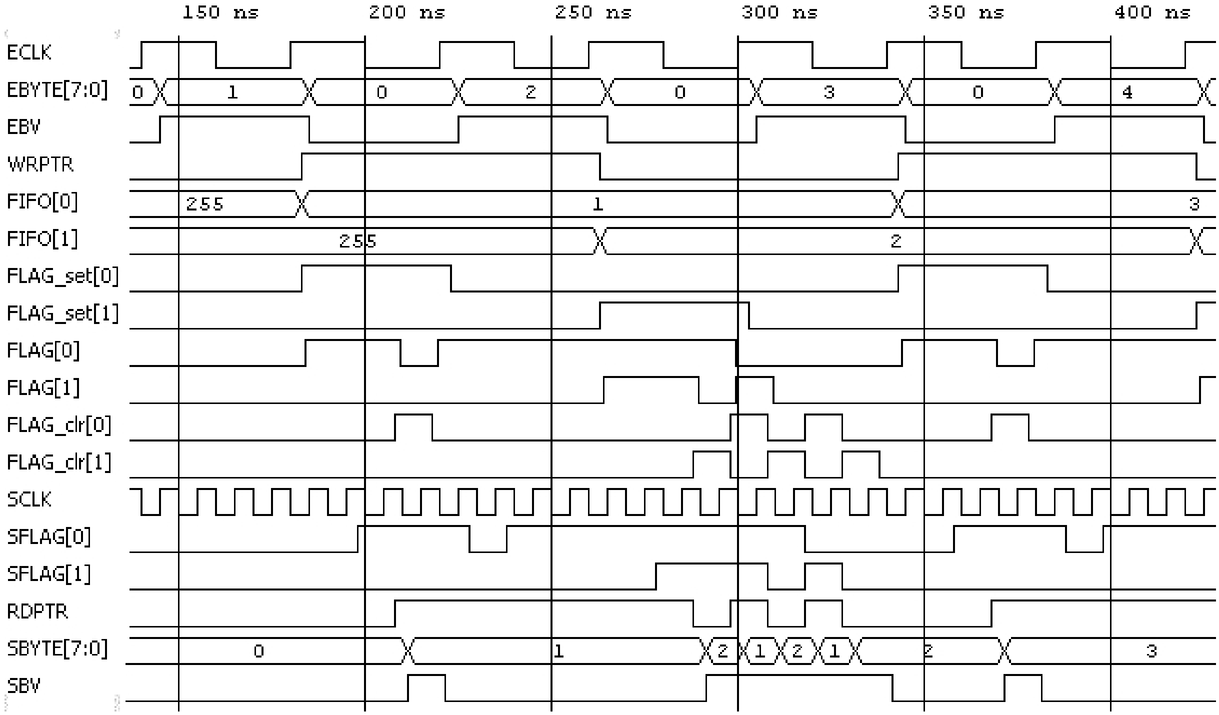


Figure 13-35 `VrEthSync` timing waveforms with `DEP=2` and fast SCLK.

Program 13-11 shows `generate` code for a `VrFIFOflagsync_ET` module based on the new strategy. Instead of a D latch, we instantiate a native, positive-edge-triggered FDCE D flip-flop for each flag bit, with `FLAG_set` connected to the clock input to load in a 1, and we still use an asynchronous clear input to clear the flag.

In our final timing diagram of this section, Figure 13-36 shows the results using the `VrFIFOflagsync_ET` module. All is well. Notice that the `FLAG_clr` pulse may still overlap with `FLAG_set` but that's not a problem. The important timing constraint here for the `FLAG` D flip-flop is that its asynchronous clear input (`FLAG_clr`) must be negated for a certain recovery time before the next rising edge of its clock (`FLAG_set`), and that is easily satisfied.

Program 13-11 Verilog code for `VrFIFOflagsync_ET` module targeted to Xilinx 7-series FPGAs.

```
generate
  for (g=0; g<=DEP-1; g=g+1) begin: flags // Clock in a 1 on rising edge of FLAG_set
    FDCE U1 ( .C(FLAG_set[g]), .CE(1'b1), .D(1'b1),
              .CLR(FLAG_clr[g]),          // Async clear when FLAG_clr is asserted
              .Q(FLAG[g]) );
  end
endgenerate
```

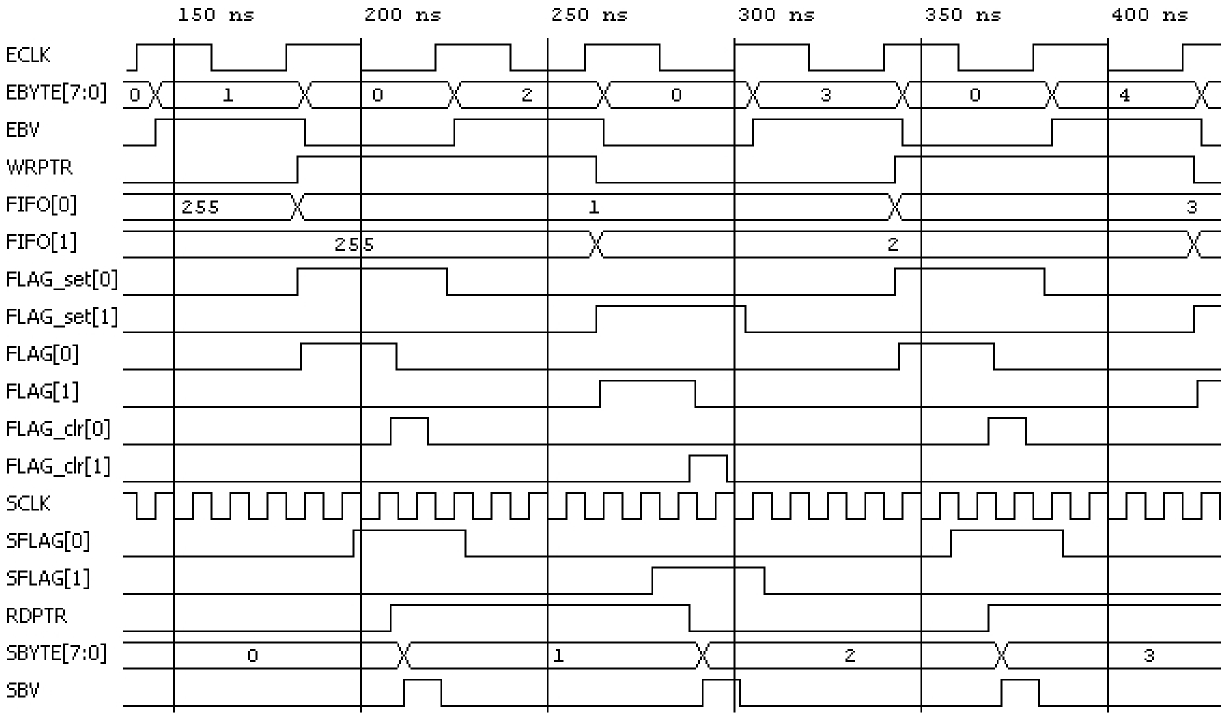


Figure 13-36 `VrEthSync` timing waveforms with `DEP=2`, fast SCLK, and `VrFIFOflagsync_ET`.

In our high-level summary of FIFO operations near the beginning of this section, we mentioned that we wanted the design to support back-to-back operations on both the input and the output buses. We showed that for `SBYTE` in Figure 13-34, but we haven't explored that capability at all for `EBYTE`. Since we already promised that the previous timing diagram was the last, we'll leave that timing study as an exercise for the reader (see Exercises 13.24 and 13.25).

After studying eighteen pages to design and analyze just one "simple" example of data-transfer synchronization across clock domains, and even then being left with exercises on some of the finer points, you should have a strong appreciation of the difficulty of correct synchronization-circuit design. Going forward, several guidelines used by experienced designers can help you:

- Minimize the number of different clock domains in a system.
- Clearly identify all clock boundaries and provide clearly identified synchronizers at those boundaries.
- Provide enough metastability resolution time for each synchronizer so synchronizer failure is rare, much more unlikely than other hardware failures.
- Analyze synchronizer behavior over a range of timing scenarios, including faster and slower clocks that might be applied as a result of system testing or upgrades.

- Simulate system behavior over a wide range of timing scenarios as well.
- Study the simulator results and make sure they make sense in light of the timing analysis, and vice versa.
- Establish and maintain conservative timing margins.

Reliance on simulation results is a catch-all for modern digital designers, who usually depend on sophisticated, high-speed logic simulators to find their bugs. But it's not a substitute for following the other guidelines. Ignoring them can lead to problems that cannot be detected by a typical, small number of simulation scenarios. Of all digital circuits, synchronizers are the ones for which it's most important to be "correct by design"!

References

Manufacturers' websites are an excellent source of information on digital design practices. Texas Instruments has an especially comprehensive site, including application notes in dozens of areas, reference designs, and of course details on all of their ICs and other products.

A good starting point for further reading on metastability is R. Ginosaur's "Metastability and Synchronizers: A Tutorial" (*IEEE Design & Test of Computers*, Sept./Oct. 2011), also published on the author's website at Technion. Another great source of pragmatic information and advice is "Synchronization and Metastability," by Steve Golson (SNUG Silicon Valley 2014); besides an excellent introduction to the topic, highlights of his paper include anecdotes, technology, accessible math, a list of fallacies, and comprehensive references.

Thomas J. Chaney spent decades studying and reporting on the metastability problem. One of his more important works, "Measured Flip-Flop Responses to Marginal Triggering" (*IEEE Trans. Comput.*, Vol. C-32, No. 12, December 1983, pp. 1207–1209), reports some of the results that we showed in Table 13-2. He's still at it, and along with several coauthors he reports recent anecdotes and describes a new tool for estimating synchronizer reliability in "Metastability and Fatal System Errors" (2013, available via online search).

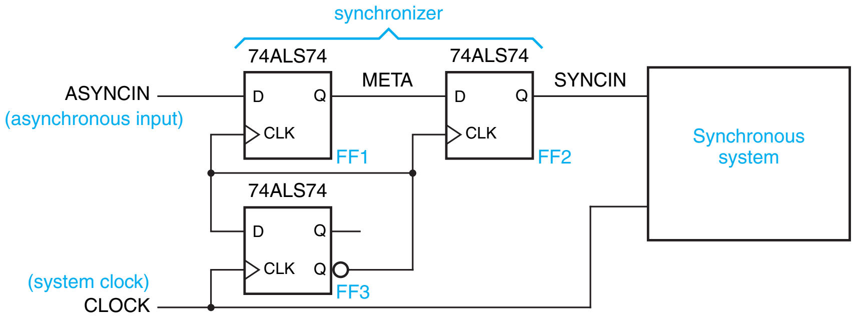
Drill Problems

13.1 The outputs of a 74AC374 drive the N0–4 and EN inputs of the circuit in Figure 6-19, which is built using 74AC138 components. The outputs of the circuit drive another 74AC374, and both '374s are clocked by the same signal with negligible clock skew. Determine the setup- and hold-time margins of the second '374 assuming a clock frequency of 25 MHz. Repeat for 30 MHz. Use the timing specifications in Tables 4-3 and 13-1.

13.2 Repeat Drill 13.1 for 74HC components operating at 2.0 V and 25°C with a clock frequency of 1.5 MHz only.

13.3 Repeat Drill 13.1 for 74HC operating at 4.5 V and 85°C with a 6.0-MHz clock.

Figure X13.9



13.4 The delay of a combinational circuit can be near 0 (say, if it's actually just a wire) but never less than 0. So, in synchronous system design, what good is a flip-flop with a negative hold time?

13.5 Considering the sequential-circuit building blocks discussed in Chapter 11, which ones are most likely to suffer from clock skew and why?

13.6 Redo the two synchronizer MTBF calculations on page 705, but instead of using Chaney's estimates for T_o and τ for the 74LS74, use TI's for the 74LSxx family. What do your results tell you about these sorts of estimates and calculations?

13.7 In some synchronizer applications, the clock frequency f is substituted for the parameter a in metastability MTBF calculations, assuming that an asynchronous input change can occur on *every* clock tick. Redo the two synchronizer MTBF calculations on page 705 under this assumption.

13.8 Calculate the MTBF of a synchronizer built according to Figure 13-21 using 74F74s, assuming a clock frequency of 25 MHz and an asynchronous transition rate of 1 MHz. Assume the setup time of an 'F74 is 5 ns and the hold time is zero.

13.9 Calculate the MTBF of the synchronizer shown in Figure X13.9, assuming a clock frequency of 30 MHz and an asynchronous transition rate of 2 MHz. Assume that the setup time t_{setup} and the propagation delay t_{pd} from clock to Q or QN in a 74ALS74 are both 10 ns.

Exercises

13.10 Repeat Drill 13.1 after changing the second 74AC374 to a 74HC374 operating at 4.5 V and 85°C. At what frequency is the setup-time margin zero?

13.11 The circuit in Figure 7-18 is built to be as fast as possible using only components from Tables 4-2 and 4-3. Assume that the outputs of a 74AC374 drive the DU bus, the DC bus is loaded into another 74AC374, and both '374s are clocked by the same signal with negligible clock skew. Assuming that the outputs of the first '374 are always enabled, determine the setup- and hold-time margins of the second '374 assuming a clock frequency of 15 MHz. Repeat for 20 MHz. Use the timing specifications in Tables 4-2, 4-3, and 13-1.

13.12 Repeat Exercise 13.11 assuming clock skew of up to 2.0 ns.

13.13 Repeat Exercise 13.11 assuming that the output-enable input of the first '374 is asserted by the output of a 74AC377 that uses the same clock-input signal with negligible clock skew.

13.14 The caption of Figure 10-26 says that it shows the "logical" behavior of one bit of the 74x377. Show a way to eliminate the 2-input multiplexer on each flip-flop's D input and still obtain the same logical behavior, reducing the size of the overall circuit. What effect would this have on the circuit's performance?

13.15 Modify the multiplication state machine in Program 13-2 so the START input can be negated for one tick at any time after a new multiplication has begun, and can then be reasserted as soon as one tick later to begin the next multiplication as soon as possible after the current one ends, at least one tick sooner than shown in Figure 13-7. Update the test bench in Program 13-7 to check your modifications. Can the multiplication begin two ticks earlier? Do any other multiplier modules or control signals need to be modified in either case?

13.16 Modify the multiplication modules of Section 13.2.2 to handle signed, two's-complement multiplication, as described in Section 2.8. Try to avoid creating a second adder (subtractor) for the last step. Update the test bench of Program 13-7 to check your design.

13.17 Using Verilog, design a datapath and state machine for dividing a 16-bit dividend by a 16-bit divisor using the algorithm in Section 8.4.1. Your datapath will need 16-bit registers for DVSR, QUOT, and the low- and high-order halves of RDIV. The state machine and datapath should use a control setup similar the multiplication system's in Section 13.2.2, where DVSR and DVND are loaded from an input bus INP during the first two clock ticks and the division proceeds during the next 16. Write a test bench that checks your system's operation for pseudorandom inputs in three categories, similar to Program 8-23. Check your results using Verilog's built-in division operation. Your system need not check for or do anything about divide-by-0, but you can use the test bench to find out what it does in those cases.

13.18 Target your design in Exercise 13.17 to your favorite FPGA and examine the synthesis results. Ensure that it contains only a single subtractor, and determine how many flip-flops it has in addition to the ones used by the state machine.

13.19 Enhance the division system in Exercise 13.17 to check for the divide-by-zero case and ensure that the quotient is all 1s in this case.

13.20 Modify the your design in Exercise 13.17 to use only three 16-bit registers, by loading the quotient bits into the low-order bit of RDIV as they are generated and shifting them at each step along with the rest of RDIV. After the last step, the low-order half of RDIV will contain the quotient. Synthesize your modified design to the same FPGA as in Exercise 13.18, and compare the FPGA resource requirements (LUTs and flip-flops) for the two versions.

13.21 Suppose that the synchronizer in Drill 13.9 is built with 74AC74 flip-flops and a 25-MHz clock, and the SYNCIN signal is connected to a combinational circuit in the synchronous system, which in turn drives the D inputs of 74AC374 flip-flops that are clocked by CLOCK. What is the maximum allowable propagation delay of the combinational logic?

13.22 Analyze the Ethernet data-transfer module of Program 13-8 and determine what goes wrong, if anything, when the assertions of `FLAG_clr` and the corresponding `FLAG_set` overlap by any amount of time. If nothing goes wrong if they overlap by "just a little," determine how long of an overlap can be safely tolerated. State any assumptions or constraints on the design of the VrFIFOflagsync module necessitated by your answer.

13.23 In the Ethernet data-transfer module of Program 13-8, determine if the flag operations can be modified so that each flag is used for a little less time. For example, could FLAG_clr be asserted and negated one SCLK period sooner? State any assumptions or constraints necessitated by your answer.

13.24 Explore the timing behavior of the Program 13-8 Ethernet data-transfer module with back-to-back operations on the EBYTE input bus. Modify the test bench to provide Ethernet input data at the same rate, 100 Mbps, but with successive pairs of bytes appearing on EBYTE for two ECLK ticks in a row followed by two unused ticks. Use the original clock rates of 25 MHz and 33 MHz for ECLK and SCLK, and the original FIFO depth of 3. Determine whether the circuit still operates properly and with sufficient margin, and if not, change it as needed.

13.25 Repeat Exercise 13.24 except now provide input data on EBYTE continuously, for an input data rate of 200 Mbps.

13.26 A famous digital designer devised the circuit shown in Figure X13.26(a), which is supposed to eliminate metastability within one period of a system clock. Circuit M is a memoryless analog voltage detector whose output is 1 if Q is in the metastable state, 0 otherwise. The circuit designer's idea was that if line Q is detected to be in the metastable state when CLOCK goes low, the NAND gate will clear the D flip-flop, which in turn eliminates the metastable output, causing a 0 output from circuit M and thus negating the CLR input of the flip-flop. The circuits are all fast enough that this all happens well before CLOCK goes high again; the expected waveforms are shown in Figure X13.26(b).

Unfortunately, the synchronizer still failed occasionally, and the famous digital designer is now designing pockets for blue jeans. Explain, in detail, how it failed, including a timing diagram.

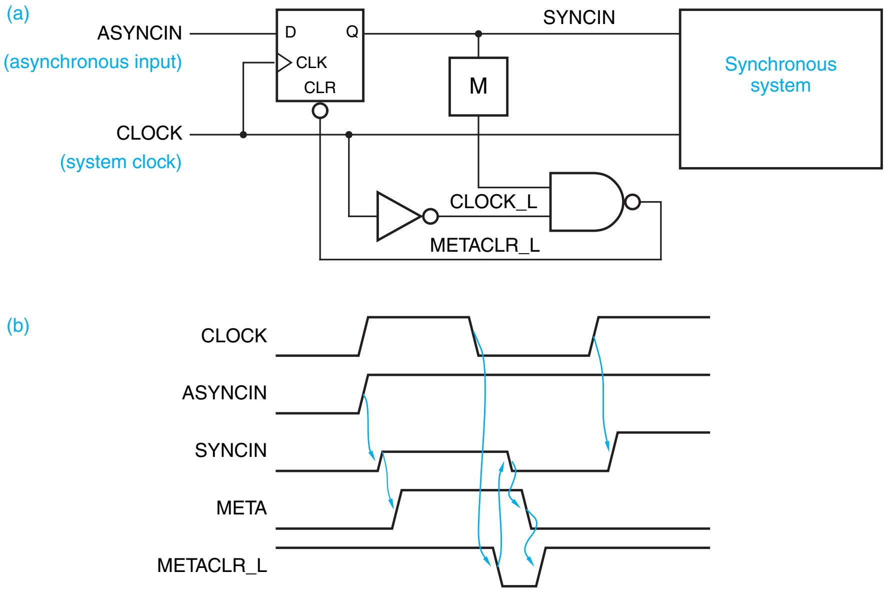
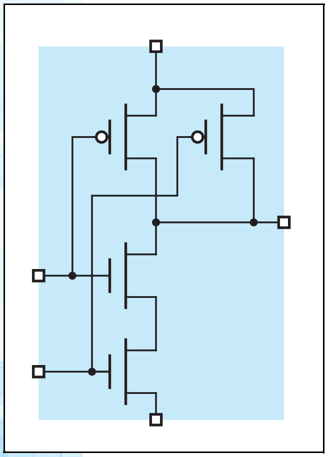


Figure X13.26



chapter 14

Digital Circuits

The purpose of this chapter is to give you a working knowledge of the electrical aspects of digital circuits, enough to give you a basic understanding of real circuits and systems and even begin to build them. You know from previous chapters that with modern software tools, it's possible to "build" circuits in the abstract, using hardware design languages to specify their structure or behavior, and using simulators to test their operation. But to build real, production-quality circuits, either at the board level or the chip level, you'll need to understand the material in this chapter, and more.

Even if you think you'll just be "slinging code" in your career, and you expect to work in an environment that has specialists who deal with all of the "electrical engineering stuff," you need to know enough about it at least to communicate with them intelligently. The logical and the electronic aspects of digital design interact and require trade-offs more often than you might think.

It's probably been a while since you read Chapter 1, but you should still recall the notion of "the digital abstraction," which allows digital designers to work with logic values of 0 and 1 instead of analog quantities. A key aspect of this abstraction is to associate a *range* of analog values with each logic value. As shown in Figure 14-1, a typical gate is not guaranteed to

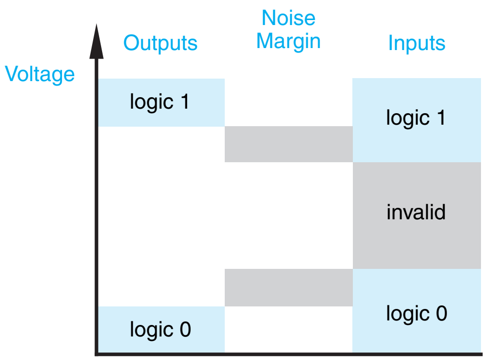


Figure 14-1
Logic values and noise margins.

have a precise voltage level for a logic 0 output. Rather, it may produce a voltage somewhere in a range that is a *subset* of the range guaranteed to be recognized as a 0 by other gate inputs. The difference between the range boundaries is called *noise margin*—in a real circuit, a gate's output can be corrupted by this much noise and still be correctly interpreted at the inputs of other gates.

noise margin

Behavior for logic 1 outputs is similar. Note in the figure that there is an "invalid" region between the input ranges for logic 0 and logic 1. Although any given digital device operating at a particular voltage and temperature will have a fairly well-defined boundary (or threshold) between the two ranges, different devices may have different boundaries. Still, all properly operating devices have their boundary *somewhere* in the "invalid" range. Therefore, any signal that is within the defined ranges for 0 and 1 will be interpreted identically by different devices. This characteristic is essential for reproducibility of results.

It is the job of an *electronic* circuit designer to design logic gates that produce and recognize logic signals that are within the appropriate ranges. This is an analog circuit-design problem, and is typically performed by a specialist who works at the transistor and physical layout level to create individual gates and other elements that become part of an ASIC library or a standard component, anything from an SSI/MSI function to an FPGA or VLSI microprocessor chip. It's impossible to design a circuit that has the desired behavior under every possible condition of power-supply voltage, temperature, loading, and other factors. Instead, the electronic circuit designer or device manufacturer provides *device specifications* (also known as *specs*) that define the conditions under which correct behavior is guaranteed.

device specifications (specs)

As a *digital* designer, then, you need not delve into the detailed analog behavior of a digital device to ensure its correct operation. Rather, you need only study enough about the device's operating environment to determine that it is operating within its published specifications. Granted, some analog knowledge is needed to perform this study, but not nearly what you'd need to design a digital device starting from scratch. This chapter aims to give you just what you need.

A LITTLE BIT OF TTL

Transistor-transistor logic (TTL), which uses bipolar transistors, was introduced in the 1960s and was the most commonly used digital logic family for decades afterwards. As a result of technology improvements, newer but compatible TTL families were introduced periodically. And other technologies, most notably CMOS, offered TTL-compatible versions and interfaces, building on TTL's popularity.

Standard TTL components use a 5-volt power supply, while almost all CMOS today operates at lower voltages. Still, because CMOS and TTL coexisted for a long time, many CMOS families were designed to operate at voltages as high as 5 volts for TTL compatibility.

Although TTL was largely replaced by CMOS in the 1990s, you still may encounter TTL or at least TTL-compatible components in your academic labs. Even in industry there is occasionally a need to design new subsystems with TTL compatibility; for example, to connect new equipment to a legacy bus. For those reasons, this chapter will occasionally mention TTL, especially in reference to compatibility and interfacing with CMOS devices.

Sections 14.6 and 14.7 will cover additional topics pertinent to CMOS/TTL interfacing. If you need more information about TTL external characteristics and internal operation, you can find it in previous editions of this book.

14.1 CMOS Logic Circuits

Besides being ubiquitous, CMOS logic is both the most capable and the easiest to understand commercial digital logic technology. Beginning in this section, we describe the basic structure of CMOS logic circuits and introduce the most commonly used commercial CMOS logic families.

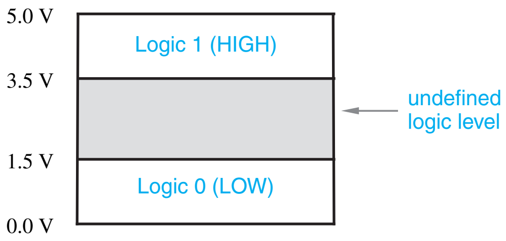
The functional behavior of a CMOS logic circuit is fairly easy to understand, even if your knowledge of analog electronics is not particularly deep. The basic building blocks in CMOS logic circuits are MOS transistors, described shortly. But before getting into that, we need to talk about logic levels.

14.1.1 CMOS Logic Levels

Abstract logic elements process binary digits, 0 and 1. However, real logic circuits process electrical signals such as voltage levels. As we've suggested in Figure 14-1, in any logic circuit there is a range of voltages (or other circuit conditions) that is interpreted as a logic 0, and another, nonoverlapping range that is interpreted as a logic 1.

A typical CMOS logic circuit operates from power supply of 5 volts (V) or less. Many circuits, especially in portable devices, use lower voltages, as low as 1 V, to save power, but for simplicity and consistency, in this chapter we'll assume 5 V until we look at lower voltages in Sections 14.6 and 14.7. Most aspects of CMOS operation scale with voltage, though not always linearly.

Figure 14-2
Logic levels for typical CMOS logic circuits



A 5-V CMOS circuit may interpret any voltage in the range 0–1.5 V as a logic 0, and in the range 3.5–5.0 V as a logic 1. Thus, the definitions of LOW and HIGH for 5-volt CMOS logic are as shown in Figure 14-2. Voltages that are in the intermediate range (1.5–3.5 V) are not expected to occur except during signal transitions, and yield undefined logic values (i.e., a circuit may interpret them as either 0 or 1). CMOS circuits using other power-supply voltages, such as 3.3 or 2.7 volts, partition the voltage range with similar proportions.

14.1.2 MOS Transistors

metal-oxide semiconductor field-effect transistor (MOSFET)
MOS transistor
"off" transistor

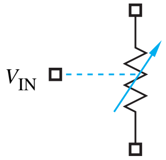
A *metal-oxide semiconductor field-effect transistor (MOSFET)*, or simply *MOS transistor* is modeled as a 3-terminal device that acts like a voltage-controlled resistance. As suggested by Figure 14-3, an input voltage applied to one terminal controls the resistance between the remaining two terminals. In digital logic applications, a MOS transistor is operated so its resistance is always either very high (and the transistor is "off") or very low (and the transistor is "on").

n-channel MOS (NMOS) transistor
gate
source
drain

There are two types of MOS transistors, *n*-channel and *p*-channel; the names refer to the type of semiconductor material used in the controlled resistance. The circuit symbol for an *n-channel MOS (NMOS) transistor* is shown in Figure 14-4. The terminals are called *gate, source,* and *drain*. Note that the "gate" of a MOS transistor is not a "logic gate," though it does "gate" the flow of current between the other two terminals. As you might guess from the orientation of the circuit symbol, the drain is normally at a higher voltage than the source.

The voltage from gate to source (V_{gs}) in an NMOS transistor is normally zero or positive. If $V_{gs} = 0$, then the resistance from drain to source (R_{ds}) is very high, at least a megohm (10^6 ohms) or more. As we increase V_{gs} (i.e., increase the voltage on the gate), R_{ds} decreases to a very low value, 10 ohms or less in some devices.

Figure 14-3
The MOS transistor as a voltage-controlled resistance.



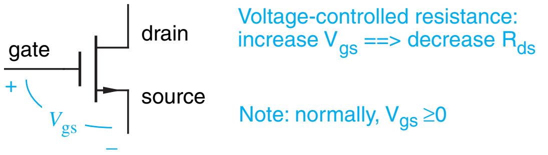

gate
drain
source
+
V_{gs}
−
Voltage-controlled resistance:
increase V_{gs} ==> decrease R_{ds}
Note: normally, V_{gs} ≥0

Figure 14-4
Circuit symbol for an n-channel MOS (NMOS) transistor.

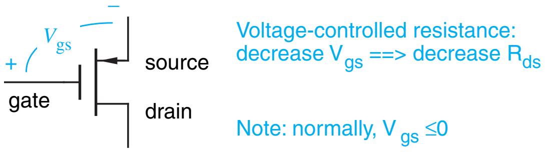

−
V_{gs}
+
gate
source
drain
Voltage-controlled resistance:
decrease V_{gs} ==> decrease R_{ds}
Note: normally, V_{gs} ≤0

Figure 14-5
Circuit symbol for a *p*-channel MOS (PMOS) transistor.

p-channel MOS (PMOS) transistor

The circuit symbol for a *p-channel MOS (PMOS) transistor* is shown in Figure 14-5. Operation is analogous to that of an NMOS transistor, except that the source is normally at a higher voltage than the drain, so V_{gs} is normally zero or negative. If V_{gs} is zero, then the resistance from source to drain (R_{ds}) is very high. As we algebraically decrease V_{gs} (i.e., *decrease* the voltage on the gate), R_{ds} decreases to a very low value.

The gate of a MOS transistor has a very high impedance. That is, the gate is separated from the source and the drain by an insulating material with a very high resistance. However, the gate voltage creates an electric field that enhances or retards (“gates”) the flow of current between source and drain. This is the “field effect” in the “MOSFET” name.

leakage current

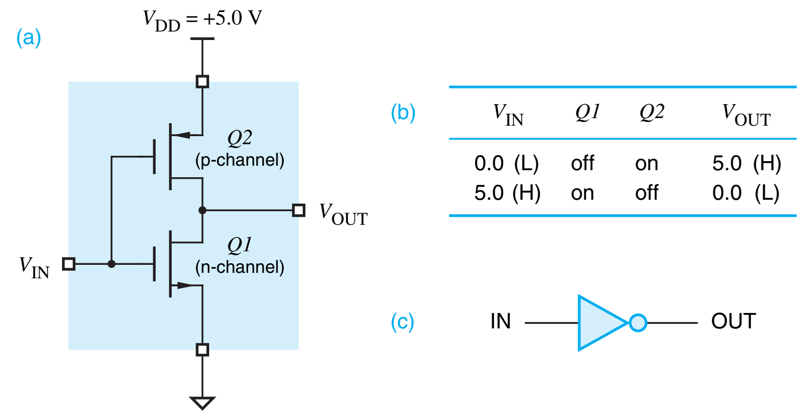
microampere, µA

Regardless of gate voltage, almost no current flows from the gate to source, or from the gate to drain for that matter. The resistance between the gate and the other terminals of the device is extremely high, well over a megohm in CMOS logic families. The small amount of current that flows across this resistance is very small, well under one *microampere* (μA, 10^{-6} A), and is called a *leakage current.*

The MOS transistor symbol itself reminds us that there is no connection between the gate and the other two terminals of the device. However, the gate of a MOS transistor is capacitively coupled to the source and drain, as the symbol might suggest. In high-speed circuits, the power needed to charge and discharge this capacitance on each input-signal transition accounts for a nontrivial portion of a circuit’s power consumption.

IMPEDANCE VS. RESISTANCE

Technically, there’s a difference between the words “impedance” and “resistance,” but electrical engineers often use the terms interchangeably. So do we in this text.



V_{IN}	$Q1$	$Q2$	V_{OUT}
0.0 (L)	off	on	5.0 (H)
5.0 (H)	on	off	0.0 (L)

Figure 14-6 CMOS inverter: (a) circuit diagram; (b) functional behavior; (c) logic symbol.

14.1.3 Basic CMOS Inverter Circuit

CMOS logic

NMOS and PMOS transistors are used together in a complementary way to form *CMOS logic*. The simplest CMOS circuit, a logic inverter, requires only one of each type of transistor, connected as shown in Figure 14-6(a). The power-supply voltage, V_{DD}, typically may be in the range 1–6 V; in some CMOS logic families it may be set to 5.0 V for compatibility with the legacy TTL family.

Ideally, the functional behavior of the CMOS inverter circuit can be characterized by just two cases tabulated in Figure 14-6(b):

1. V_{IN} is 0.0 V. In this case, the bottom, *n*-channel transistor *Q1* is off, since its V_{gs} is 0, but the top, *p*-channel transistor *Q2* is on, since its V_{gs} is a large negative value (–5.0 V). Therefore, *Q2* presents only a small resistance between the power-supply terminal (V_{DD}, +5.0 V) and the output terminal (V_{OUT}), and the output voltage is 5.0 V.
2. V_{IN} is 5.0 V. Here, *Q1* is on, since its V_{gs} is a large positive value (+5.0 V), but *Q2* is off, since its V_{gs} is 0. Thus, *Q1* presents a small resistance between the output terminal and ground, and the output voltage is 0 V.

WHAT'S IN A NAME?

The "DD" in the name "V_{DD}" refers to the *drain* terminals of MOS transistors. This may seem strange, since in the CMOS inverter V_{DD} is actually connected to the *source* terminal of a PMOS transistor. However, CMOS logic circuits evolved from NMOS logic circuits, where the supply *was* connected to the drain of an NMOS transistor through a load resistor, and the name "V_{DD}" stuck.

Also note that ground is sometimes referred to as "V_{SS}" in CMOS and NMOS circuits. Some authors and most circuit manufacturers use "V_{CC}" as the symbol for the CMOS supply voltage, since this name is used in TTL circuits which preceded CMOS. To get you used to both, we'll start using "V_{CC}" in Section 14.2.

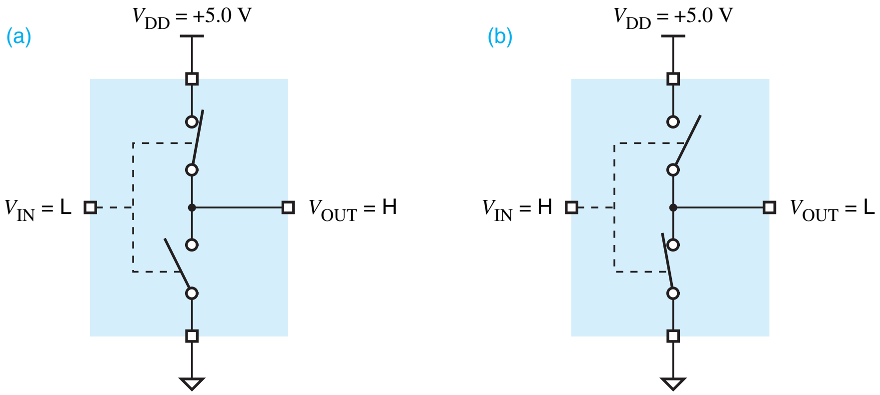


Figure 14-7
Switch model for CMOS inverter:
(a) LOW input;
(b) HIGH input.

With the foregoing functional behavior, the circuit clearly behaves as a logical inverter, since a 0-volt input produces a 5-volt output, and vice versa.

Another way to visualize CMOS operation uses switches. As shown in Figure 14-7(a), the *n*-channel (bottom) transistor is modeled by a normally-open switch, and the *p*-channel (top) transistor by a normally-closed switch. Applying a HIGH voltage "pushes" each switch to the opposite of its normal state, as shown in (b).

The switch model gives rise to a way of drawing CMOS circuits that makes their logical behavior more readily apparent. As shown in Figure 14-8, different symbols are used for the *p*- and *n*-channel transistors to reflect their logical behavior. The *n*-channel transistor (*Q1*) is switched "on," and current flows between source and drain, when a HIGH voltage is applied to its gate; this seems natural enough. The *p*-channel transistor (*Q2*) has the opposite behavior. It is "on" when a LOW voltage is applied; the inversion bubble on its gate indicates this inverting behavior.

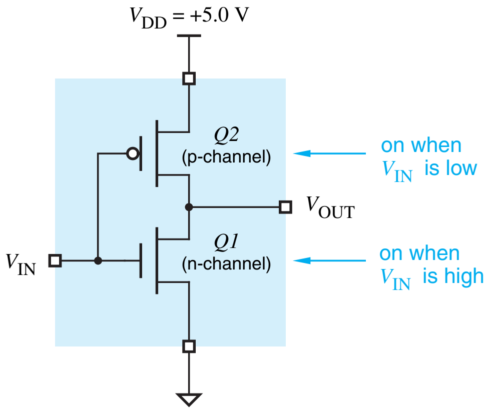
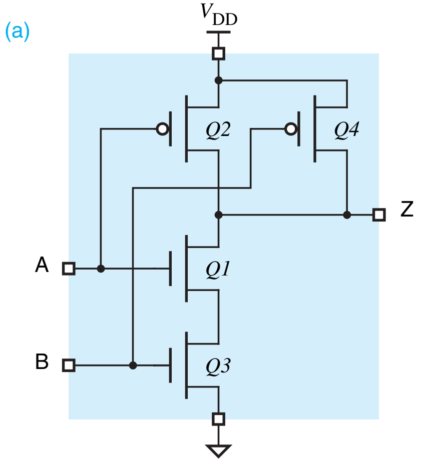


Figure 14-8
CMOS inverter logical operation.



(b)

A	B	*Q1*	*Q2*	*Q3*	*Q4*	Z
L	L	off	on	off	on	H
L	H	off	on	on	off	H
H	L	on	off	off	on	H
H	H	on	off	on	off	L

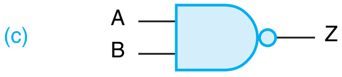


Figure 14-9 CMOS 2-input NAND gate: (a) circuit diagram; (b) function table; (c) logic symbol.

14.1.4 CMOS NAND and NOR Gates

Both NAND and NOR gates can be constructed using CMOS. A *k*-input gate uses *k* *p*-channel and *k* *n*-channel transistors.

Figure 14-9 shows a 2-input CMOS NAND gate. If either input is LOW, the output Z has a low-impedance connection to V_{DD} through the corresponding “on” *p*-channel transistor, and the path to ground is blocked by the corresponding “off” *n*-channel transistor. If both inputs are HIGH, the path to V_{DD} is blocked, and Z has a low-impedance connection to ground. Figure 14-10 shows the switch model for the NAND gate’s operation.

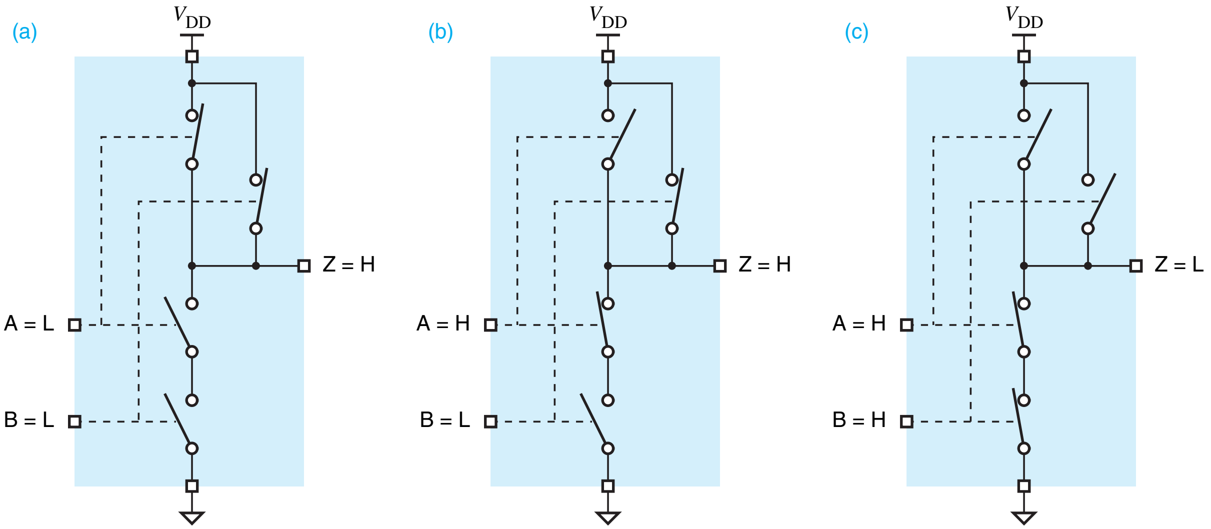
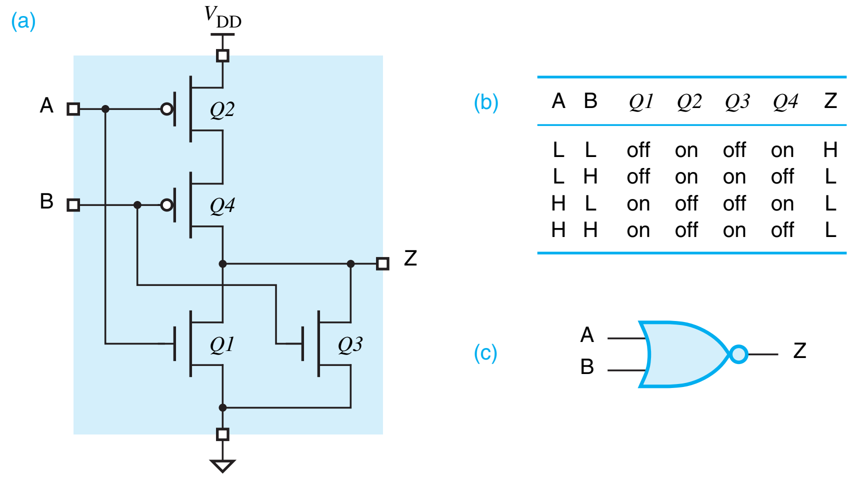


Figure 14-10 Switch model for CMOS 2-input NAND gate: (a) both inputs LOW; (b) one input HIGH; (c) both inputs HIGH.



A	B	*Q1*	*Q2*	*Q3*	*Q4*	Z
L	L	off	on	off	on	H
L	H	off	on	on	off	L
H	L	on	off	off	on	L
H	H	on	off	on	off	L

Figure 14-11
CMOS 2-input NOR gate:
(a) circuit diagram;
(b) function table;
(c) logic symbol.

Figure 14-11 shows a CMOS NOR gate. If both inputs are LOW, then the output Z has a low-impedance connection to V_{DD} through the “on” *p*-channel transistors, and the path to ground is blocked by the “off” *n*-channel transistors. If either input is HIGH, the path to V_{DD} is blocked, and Z has a low-impedance connection to ground.

14.1.5 Fan-In

The number of inputs that a gate can have in a particular logic family is called the logic family’s *fan-in*. CMOS gates with more than two inputs can be obtained by extending series-parallel designs of Figures 14-9 and 14-11 in a straightforward manner. A *k*-input gate has *k* series and *k* parallel transistors. For example, Figure 14-12 on the next page shows a 3-input CMOS NAND gate.

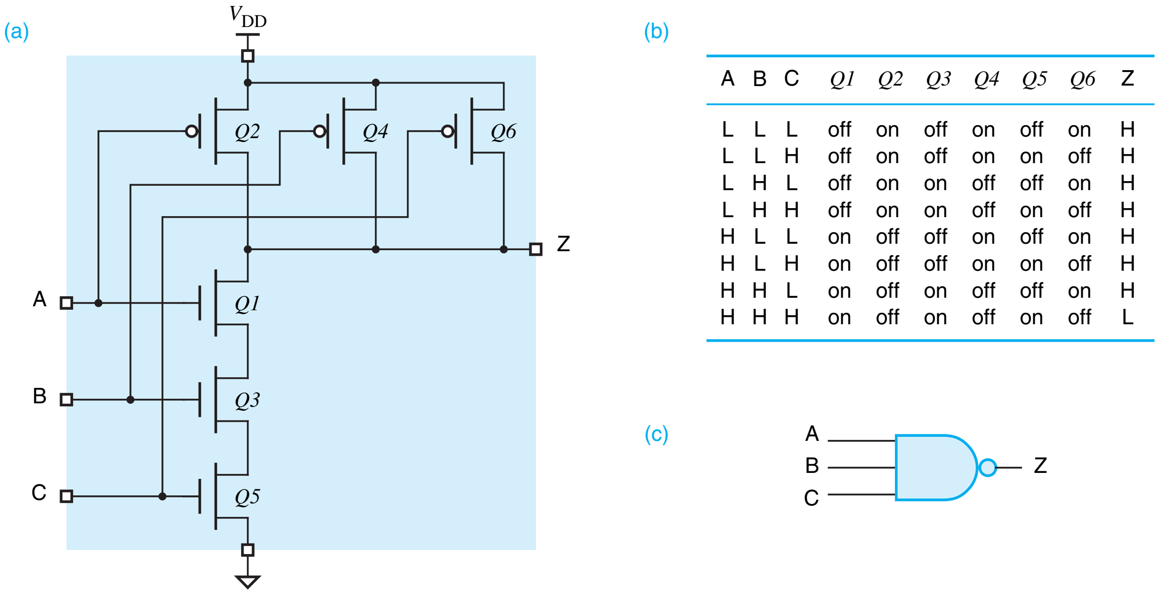
fan-in

In principle, you could design a CMOS NAND or NOR gate with a very large number of inputs. In practice, however, the additive “on” resistance of series transistors limits the fan-in of CMOS gates, typically to 4 for NOR gates and 6 for NAND gates.

As the number of inputs is increased, designers of CMOS gate circuits may compensate by increasing the size of the series transistors to reduce their resistance and the corresponding switching delay. However, at some point this becomes inefficient or impractical. Gates with a large number of inputs can be

NAND VS. NOR

CMOS NAND and NOR gates do not have identical electrical performance. For a given silicon area, an *n*-channel transistor has about half the “on” resistance of a *p*-channel transistor. Therefore, when transistors are put in series, *k* *n*-channel transistors have lower “on” resistance than do *k* *p*-channel ones. For a given silicon area, a *k*-input NAND gate is generally faster than and preferred over a *k*-input NOR gate.



A	B	C	*Q1*	*Q2*	*Q3*	*Q4*	*Q5*	*Q6*	Z
L	L	L	off	on	off	on	off	on	H
L	L	H	off	on	off	on	on	off	H
L	H	L	off	on	on	off	off	on	H
L	H	H	off	on	on	off	on	off	H
H	L	L	on	off	off	on	off	on	H
H	L	H	on	off	off	on	on	off	H
H	H	L	on	off	on	off	off	on	H
H	H	H	on	off	on	off	on	off	L

Figure 14-12 CMOS 3-input NAND gate: (a) circuit diagram; (b) function table; (c) logic symbol.

made faster and smaller by cascading gates with fewer inputs. For example, Figure 14-13 shows the logical structure of an 8-input CMOS NAND gate. The total delay through a 4-input NAND, a 2-input NOR, and an inverter is typically less than the delay of a "one-level" 8-input NAND circuit.

14.1.6 Noninverting Gates

In CMOS, and in most other logic families, the simplest gates are inverters, and the next simplest are NAND gates and NOR gates. A logical inversion comes "for free," and it typically is not possible to design a noninverting gate with a smaller number of transistors than an inverting one.

CMOS noninverting buffers and AND and OR gates can be obtained by connecting an inverter to the output of the corresponding inverting gate. For example, Figure 14-14 shows a noninverting buffer and Figure 14-15 shows an AND gate. Combining Figure 14-11(a) with an inverter yields an OR gate.

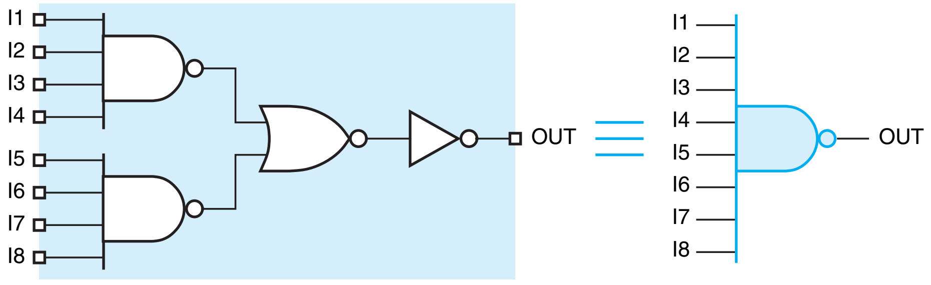
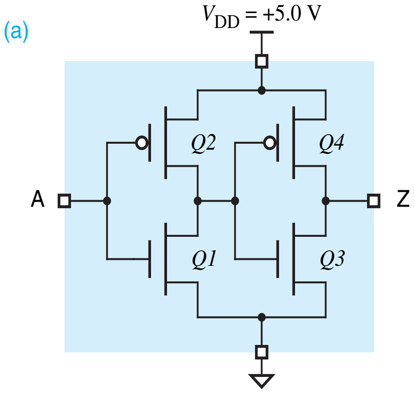


Figure 14-13 Logic diagram equivalent to the internal structure of an 8-input CMOS NAND gate.



(b)

A	*Q1*	*Q2*	*Q3*	*Q4*	Z
L	off	on	on	off	L
H	on	off	off	on	H

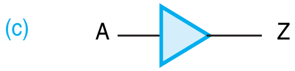
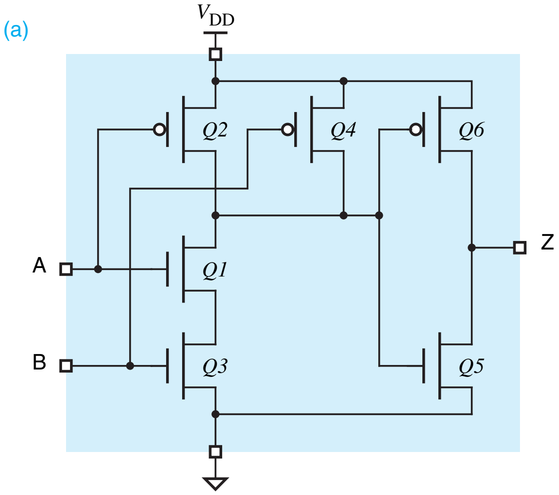


Figure 14-14 CMOS noninverting buffer: (a) circuit diagram; (b) function table; (c) logic symbol.



(b)

A	B	*Q1*	*Q2*	*Q3*	*Q4*	*Q5*	*Q6*	Z
L	L	off	on	off	on	on	off	L
L	H	off	on	on	off	on	off	L
H	L	on	off	off	on	on	off	L
H	H	on	off	on	off	off	on	H

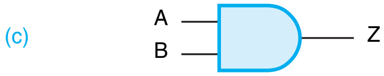


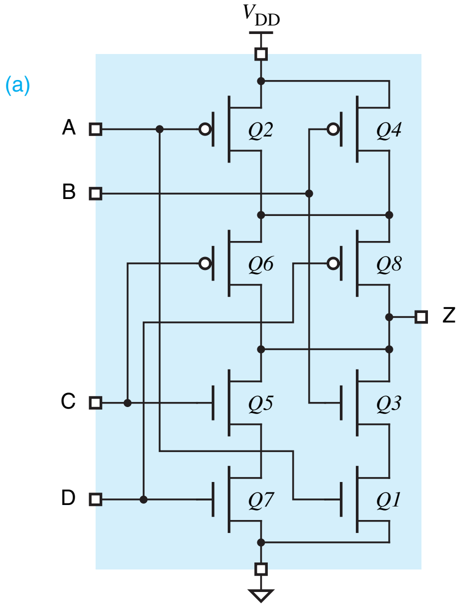
Figure 14-15 CMOS 2-input AND gate: (a) circuit diagram; (b) function table; (c) logic symbol.

14.1.7 CMOS AND-OR-INVERT and OR-AND-INVERT Gates

CMOS circuits can perform two levels of logic with just a single "level" of transistors in a clever series-parallel configuration. For example, the circuit in Figure 14-16(a) is a 2-wide, 2-input CMOS *AND-OR-INVERT (AOI) gate*. The function table for this circuit is shown in (b) and a logic diagram for this function using AND and NOR gates is shown in Figure 14-17. Transistors can be added to or removed from this circuit to obtain an AOI function with a different number of ANDs or a different number of inputs per AND.

AND-OR-INVERT (AOI) gate

The contents of each of the *Q1–Q8* columns in Figure 14-16(b) depends only on the input signal connected to the corresponding transistor's gate. The last column is constructed by examining each input combination and determin-



(b)

A	B	C	D	*Q1*	*Q2*	*Q3*	*Q4*	*Q5*	*Q6*	*Q7*	*Q8*	Z
L	L	L	L	off	on	off	on	off	on	off	on	H
L	L	L	H	off	on	off	on	off	on	on	off	H
L	L	H	L	off	on	off	on	on	off	off	on	H
L	L	H	H	off	on	off	on	on	off	on	off	L
L	H	L	L	off	on	on	off	off	on	off	on	H
L	H	L	H	off	on	on	off	off	on	on	off	H
L	H	H	L	off	on	on	off	on	off	off	on	H
L	H	H	H	off	on	on	off	on	off	on	off	L
H	L	L	L	on	off	off	on	off	on	off	on	H
H	L	L	H	on	off	off	on	off	on	on	off	H
H	L	H	L	on	off	off	on	on	off	off	on	H
H	L	H	H	on	off	off	on	on	off	on	off	L
H	H	L	L	on	off	on	off	off	on	off	on	L
H	H	L	H	on	off	on	off	off	on	on	off	L
H	H	H	L	on	off	on	off	on	off	off	on	L
H	H	H	H	on	off	on	off	on	off	on	off	L

Figure 14-16 CMOS AND-OR-INVERT gate: (a) circuit diagram; (b) function table.

ing whether Z is connected to V_{DD} or to ground by "on" transistors for that input combination. Note that Z is never connected to *both* V_{DD} and ground for any input combination; in such a case the output would be a nonlogic value somewhere between LOW and HIGH, and the output structure would consume excessive power due to the low-impedance connection between V_{DD} and ground.

OR-AND-INVERT (OAI) gate

A circuit can also be designed to perform an OR-AND-INVERT function. For example, Figure 14-19(a) shows a 2-wide, 2-input CMOS *OR-AND-INVERT (OAI) gate*. The function table for this circuit is shown in (b); the values in each column are determined just as we did for the CMOS AOI gate. A logic diagram for the OAI function using OR and NAND gates is shown in Figure 14-19.

The speed and other electrical characteristics of a CMOS AOI or OAI gate are quite comparable to those of a single CMOS NAND or NOR gate. As a result, these gates are very appealing because they can perform two levels of logic

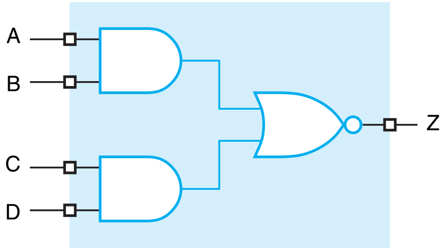
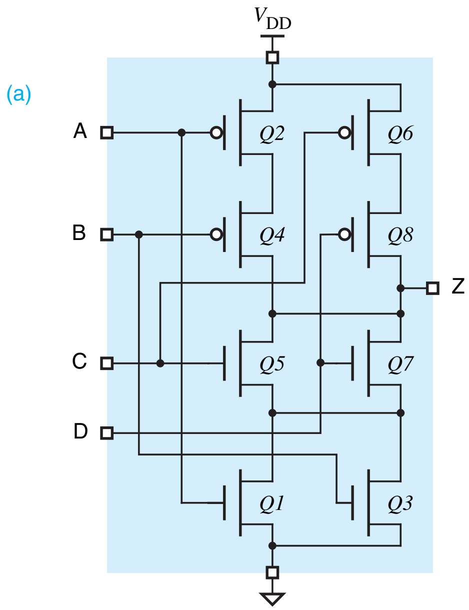


Figure 14-17 Logic diagram for CMOS AND-OR-INVERT gate.



(b)

A	B	C	D	*Q1*	*Q2*	*Q3*	*Q4*	*Q5*	*Q6*	*Q7*	*Q8*	Z
L	L	L	L	off	on	off	on	off	on	off	on	H
L	L	L	H	off	on	off	on	off	on	on	off	H
L	L	H	L	off	on	off	on	on	off	off	on	H
L	L	H	H	off	on	off	on	on	off	on	off	H
L	H	L	L	off	on	on	off	off	on	off	on	H
L	H	L	H	off	on	on	off	off	on	on	off	L
L	H	H	L	off	on	on	off	on	off	off	on	L
L	H	H	H	off	on	on	off	on	off	on	off	L
H	L	L	L	on	off	off	on	off	on	off	on	H
H	L	L	H	on	off	off	on	off	on	on	off	L
H	L	H	L	on	off	off	on	on	off	off	on	L
H	L	H	H	on	off	off	on	on	off	on	off	L
H	H	L	L	on	off	on	off	off	on	off	on	H
H	H	L	H	on	off	on	off	off	on	on	off	L
H	H	H	L	on	off	on	off	on	off	off	on	L
H	H	H	H	on	off	on	off	on	off	on	off	L

Figure 14-18 CMOS OR-AND-INVERT gate: (a) circuit diagram; (b) function table.

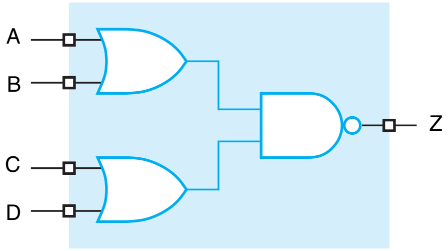


Figure 14-19
Logic diagram for CMOS OR-AND-INVERT gate.

(AND-OR or OR-AND) with just one level of delay. CMOS VLSI devices often use these gates internally, since many HDL synthesis tools can automatically convert AND/OR logic into AOI gates when appropriate.

14.2 Electrical Behavior of CMOS Circuits

This section and the next three discuss electrical, not logical, aspects of CMOS circuit operation. It's important to understand this material when you design real circuits using CMOS or other logic families. Most of this material is aimed at providing a framework for ensuring that the "digital abstraction" is really valid for a given circuit. In particular, a circuit or system designer must provide adequate engineering design margins—insurance that the circuit will work properly even under the worst of conditions.

engineering design margins

IS ALL THIS REALLY NECESSARY?

The behaviors described in the next few sections are a consequence of the electrical design of the CMOS logic gates, including both their transistor-level structure and the analog properties of the transistors themselves. Since you may never design a logic gate yourself, you might think that these topics are unimportant.

However, these behaviors are also a consequence of the way that gates are selected and interconnected to form digital logic circuits, and creating such interconnections is exactly what a digital designer does.

Some technologies, like FPGAs, may hide the electrical consequences of on-chip interconnections from the designer, who can specify the design using an HDL and use a software tool to generate an internal connection pattern that satisfies all electrical requirements. But it is almost *always* necessary for the designer to understand electrical characteristics when two or more chips are interconnected. So, please read on.

14.2.1 Overview

The topics that we'll examine in Sections 14.3–14.5 pertain to both the static and the dynamic behavior of CMOS devices and circuits:

- *Static behaviors*. These topics cover situations where a circuit's input and output signals are not changing. They include things like power consumption, the match-up and tolerances between input and output logic levels, and noise immunity.
- *Dynamic behaviors*. These topics cover situations where a circuit's input and output signals are changing. They include things like the extra power that is consumed as signals change, and the timing from an input-signal change to the resulting output-signal change.

When analyzing or designing a digital circuit, the designer must consider both static and dynamic behaviors. Most of the topics that we will discuss in Sections 14.3–14.5 have both static and dynamic aspects. The topics include the following:

- *Logic voltage levels*. CMOS devices operating under normal conditions are guaranteed to produce output voltage levels within well-defined LOW and HIGH ranges. And they recognize LOW and HIGH input voltage levels over somewhat wider ranges. CMOS manufacturers specify these ranges and operating conditions very carefully to ensure compatibility among different devices in the same family, and to provide a degree of interoperability (if you're careful) among devices in different families.
- *DC noise margins*. Positive DC noise margins ensure that the highest LOW voltage produced by an output is always lower than the highest voltage that an input can reliably interpret as LOW, and that the lowest HIGH voltage

produced by an output is always higher than the lowest voltage that an input can reliably interpret as HIGH. A good understanding of noise margins is especially important in circuits that use devices from a number of different families.

- *Fanout*. This refers to the number and type of device inputs and other loads that are connected to a given output. If too many loads are connected to an output, the DC noise margins of the circuit may be inadequate. Fanout also affects the speed at which the output changes from one state to another.
- *Speed*. The time that it takes a CMOS output to change from the LOW state to the HIGH state, or vice versa, depends on both the internal structure of the device and the characteristics of the other devices that it drives, even to the extent of being affected by the wire or printed-circuit-board traces connected to the output. We'll look at two separate components of "speed"—transition time and propagation delay.
- *Power consumption*. The power consumed by a CMOS device depends on a number of factors, including not only its internal structure, but also the input signals that it receives, the other devices that it drives, and how often its output changes between LOW and HIGH.
- *Noise.* The main reason for providing engineering design margins is to ensure proper circuit operation in the presence of noise. Noise can be generated by a number of sources; several of them are listed below, from the least likely to the (perhaps surprisingly) most likely:
  - Cosmic rays.
  - Power-supply disturbances.
  - Magnetic fields from nearby equipment.
  - The switching action of the logic circuits themselves.
- *Electrostatic discharge.* Would you believe that you can destroy a CMOS device just by touching it? Ordinary "static electricity" can have a voltage potential of a thousand volts or more, enough to puncture and damage the thin insulating material between a MOS transistor's gate and its source and drain.
- *Open-drain outputs.* Some CMOS outputs omit the usual *p*-channel pull-up transistors. In the HIGH state, such an output behaves essentially like a "no-connection," which is useful in some applications.
- *Three-state outputs.* Some CMOS devices have an extra "output enable" control input that can be used to disable both the *p*-channel pull-up transistors and the *n*-channel pull-down transistors, creating a "high-impedance (hi-Z)" output. Many such device outputs can be tied together to create a multisource bus, as long as the control logic is arranged so that at most one output is enabled at a time.

Among these topics, timing is probably the most important, since it's an area where designers typically must spend the most time, even if they're otherwise working at a strictly "logical" level. Even when you use automated tools in a design, for example using an HDL and targeting an FPGA, getting the timing right is often a difficult step. Designers often use the phrase "timing closure" for this step, because they'd like to get it behind them!

14.2.2 Data Sheets and Specifications

data sheet

The manufacturers of real-world devices provide *data sheets* that specify the devices' logical and electrical characteristics. The electrical specifications portion of a minimal data sheet for a simple CMOS device, the 54/74HC00 quadruple NAND gate, is shown in Table 14-1. ("Quadruple" means there are four gates in the same chip and package.) You may never design anything that uses such a simple component, but the information in this datasheet is a subset of what you will find for more complex components, including the many inputs and outputs you will find on an FPGA or other VLSI chip. Different manufacturers typically specify additional parameters, and they may vary in how they specify even the "standard" parameters shown in the table. To define exactly what their parameters mean, they usually also show the test circuits and waveforms for them, as in Figure 14-20. Note that this figure contains information for some additional parameters beyond those used with the 54/74HC00.

Most of the terms in the data sheet and the waveforms in the figure are probably meaningless to you at this point. However, after reading the next three sections, you should know enough about the electrical characteristics of CMOS circuits that you'll be able to understand the salient points of this or any other data sheet. As a digital designer, you'll need this knowledge to create reliable and robust real-world circuits and systems.

DON'T BE AFRAID Computer science students and other non-EE readers should not have undue fear of the material in the next three sections. Only a basic understanding of electronics, at about the level of Ohm's law, is required.

14.3 CMOS Static Electrical Behavior

This section discusses the "DC" or static behavior of CMOS circuits, that is, the circuits' behavior when inputs and outputs are not changing. Electrical engineers also call this "steady-state" behavior, because the electrical state of the inputs is not changing.

14.3.1 Logic Levels and Noise Margins

The table in Figure 14-6(b) on page 738 defined the CMOS inverter's behavior only at two discrete input voltages; other input voltages may yield different

Table 14-1 Manufacturer's data sheet for a typical CMOS device, a 54/74HC00 quad NAND gate.

DC ELECTRICAL CHARACTERISTICS OVER OPERATING RANGE
The following conditions apply unless otherwise specified:
Commercial: $T_A = -40°C$ to $+85°C$, $V_{CC} = 5.0\ V \pm 5\%$; Military: $T_A = -55°C$ to $+125°C$, $V_{CC} = 5.0\ V \pm 10\%$

Sym.	**Parameter**	**Test Conditions**[1]		**Min.**	**Typ.**[2]	**Max.**	**Unit**
V_{IH}	Input HIGH level	Guaranteed logic HIGH level		3.15	—	—	V
V_{IL}	Input LOW level	Guaranteed logic LOW level		—	—	1.35	V
I_{IH}	Input HIGH current	V_{CC} = Max., $V_I = V_{CC}$		—	—	1	μA
I_{IL}	Input LOW current	V_{CC} = Max., V_I = 0 V		—	—	–1	μA
V_{IK}	Clamp diode voltage	V_{CC} = Min., I_N = –18 mA		—	–0.7	–1.2	V
I_{IOS}	Short-circuit current	V_{CC} = Max.,[3] V_O = GND		—	—	–35	mA
V_{OH}	Output HIGH voltage	V_{CC} = Min., $V_{IN} = V_{IL}$	I_{OH} = –20 μA	4.4	4.499	—	V
			I_{OH} = –4 mA	3.84	4.3	—	V
V_{OL}	Output LOW voltage	V_{CC} = Min., $V_{IN} = V_{IH}$	I_{OL} = 20 μA	—	.001	0.1	V
			I_{OL} = 4 mA		0.17	0.33	V
I_{CC}	Quiescent power supply current	V_{CC} = Max. V_{IN} = GND or V_{CC}, I_O = 0		—	2	10	μA

SWITCHING CHARACTERISTICS OVER OPERATING RANGE, C_L = 50 pF

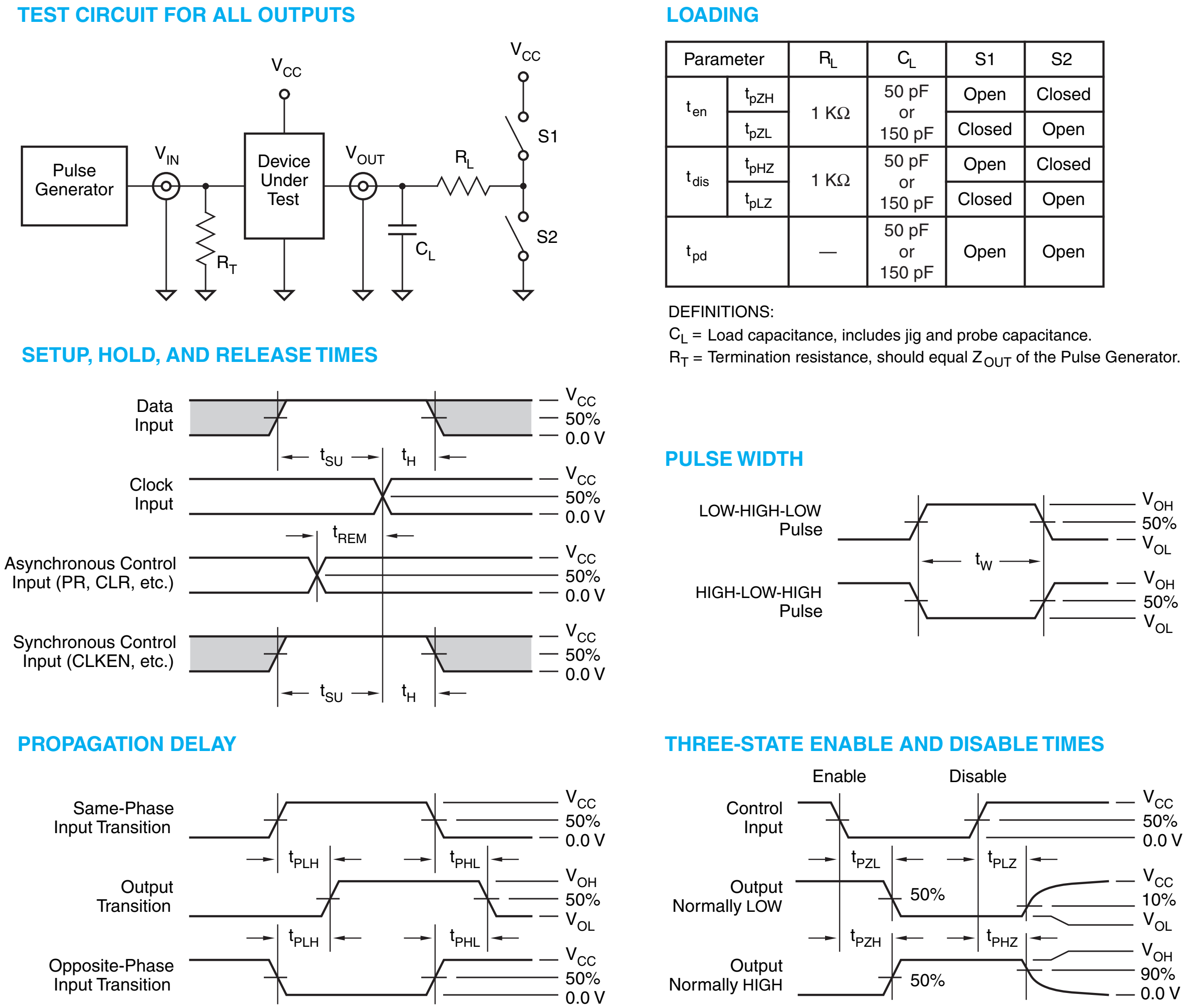
Sym.	**Parameter**[4]	**Test Conditions**		**Min.**	**Typ.**	**Max.**	**Unit**
t_{PD}	Propagation delay	A or B to Y		—	9	19	ns
C_I	Input capacitance	V_{IN} = 0 V		—	3	10	pF
C_{pd}	Power dissipation capacitance per gate		No load	—	22	—	pF

NOTES:

1. *For conditions shown as Max. or Min., use appropriate value specified under Electrical Characteristics.*
2. *Typical values are at V_{CC} = 5.0 V, +25 °C ambient.*
3. *Not more than one output should be shorted at a time. Duration of short-circuit test should not exceed one second.*
4. *This parameter is guaranteed but not tested.*

WHAT'S IN A NUMBER?

Two different prefixes, "74" and "54," are used in the part numbers of legacy SSI and MSI devices. These prefixes simply distinguish between commercial and military versions. A 74HC00 is the commercial part and the 54HC00 is the military version.



TEST CIRCUIT FOR ALL OUTPUTS

LOADING

Parameter		R_L	C_L	S1	S2
t_{en}	t_{pZH}	1 KΩ	50 pF or 150 pF	Open	Closed
	t_{pZL}			Closed	Open
t_{dis}	t_{pHZ}	1 KΩ	50 pF or 150 pF	Open	Closed
	t_{pLZ}			Closed	Open
t_{pd}		—	50 pF or 150 pF	Open	Open

DEFINITIONS:
C_L = Load capacitance, includes jig and probe capacitance.
R_T = Termination resistance, should equal Z_{OUT} of the Pulse Generator.

SETUP, HOLD, AND RELEASE TIMES

PULSE WIDTH

PROPAGATION DELAY

THREE-STATE ENABLE AND DISABLE TIMES

Figure 14-20 Test circuits and waveforms for HC-series logic.

voltage transfer diagram

output voltages. The complete input-output transfer characteristic of a particular inverter can be described by a graph like Figure 14-21, also called a *voltage-transfer diagram*. In this graph, the input voltage is varied from 0 to 5 V, as shown on the *X* axis, and the *Y* axis plots the output voltage.

If we believed the curve in Figure 14-21, we could define a CMOS LOW input level as any voltage under 2.4 V, and a HIGH input level as anything over 2.6 V. Only when the input is between 2.4 and 2.6 V does the inverter produce a nonlogic output voltage under this definition.

Unfortunately, the typical transfer characteristic shown in Figure 14-21 is just that—typical, but not guaranteed. It varies greatly under different conditions such as power-supply voltage, temperature, and output loading. For example, the

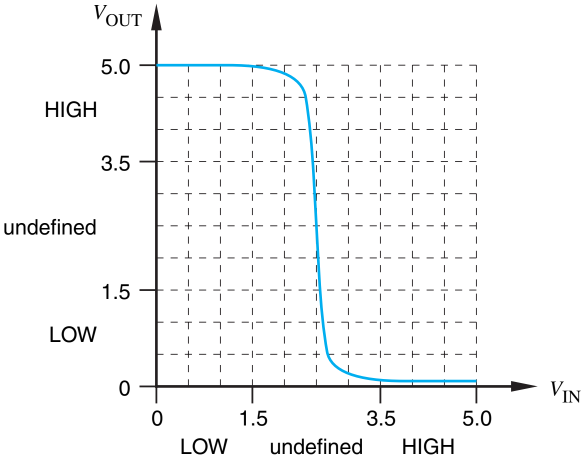


Figure 14-21
Typical input-output transfer characteristic of a CMOS inverter.

transition in the middle of the curve may become more or less steep, and it may shift to the left or the right. The transfer characteristic may even vary depending on when the device was fabricated. For example, after months of trying to figure out why gates made on some days were good and on other days were bad, legend has it that one manufacturer discovered that the bad gates were victims of air-borne contamination by a particularly noxious perfume worn by one of its production-line workers!

Sound engineering practice dictates that we use specifications for LOW and HIGH that are more conservative. Conservative logic-level specs for a typical CMOS logic family (HC-series) are depicted in Figure 14-22. These parameters are specified by CMOS device manufacturers in data sheets like Table 14-1 on page 749, and are defined as follows:

V_{OHmin} The minimum output voltage produced in the HIGH state.

V_{IHmin} The minimum input voltage guaranteed to be recognized as a HIGH.

V_{ILmax} The maximum input voltage guaranteed to be recognized as a LOW.

V_{OLmax} The maximum output voltage produced in the LOW state.

The input voltages are determined mainly by switching thresholds of the device's transistors, while the output voltages are determined mainly by their "on" resistances.

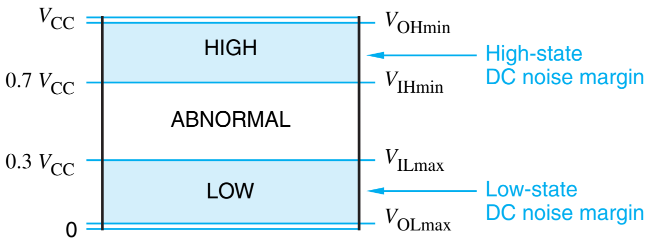


Figure 14-22
Logic levels and noise margins for the HC-series CMOS logic family.

All of the parameters in Figure 14-22 are guaranteed by CMOS manufacturers over a range of temperature and output loading. Parameters are also guaranteed over a range of power-supply voltage V_{CC}, such as 5.0 V±10%.

The data sheet in Table 14-1 specifies values for each of these parameters for HC-series CMOS. Notice that there are two values specified for V_{OHmin} and V_{OLmax}, depending on whether the output current (I_{OH} or I_{OL}) is large or small. When the device outputs are connected only to other CMOS inputs, the output current is low (e.g., $I_{OL} \leq 20\ \mu A$), so there's very little voltage drop across the output transistors. In the next few subsections, we'll discuss both these "pure" CMOS applications, and applications where CMOS outputs are connected to other circuits that require more current.

power-supply rails

The power-supply voltage V_{CC} and ground are often called the *power-supply rails*. CMOS levels are typically a function of the power-supply rails, for example:

V_{OHmin}	$V_{CC} - 0.1$ V
V_{IHmin}	70% of V_{CC}
V_{ILmax}	30% of V_{CC}
V_{OLmax}	ground (0 V) + 0.1 V

Notice in Table 14-1 that V_{OHmin} is specified as 4.4 V. This is only a 0.1-V drop from V_{CC}, since the worst-case number is specified with V_{CC} at its minimum value of 5.0 – 10% = 4.5 V.

DC noise margin

DC noise margin is a measure of how much noise it takes to corrupt a worst-case output voltage into a value that may not be recognized properly by an input. For example, with HC-series CMOS in the LOW state, V_{ILmax} (1.35 V) exceeds V_{OLmax} (0.1 V) by 1.25 V, so the LOW-state DC noise margin is 1.25 V. In the HIGH state, V_{IHmin} (3.15 V) is 1.25 V lower than V_{OHmin} (4.4 V), so there is 1.25 V of HIGH-state DC noise margin as well. In general, CMOS outputs have excellent DC noise margins when driving other CMOS inputs.

Regardless of the voltage applied to the input of a CMOS gate, the input consumes very little current, only the sum of leakage currents of its transistors' gates. The maximum amount of leakage current that can flow is specified by the device manufacturer:

I_{IH} The maximum current that flows into the input in the HIGH state.

I_{IL} The maximum current that flows into the input in the LOW state.

The input current shown in Table 14-1 for the 'HC00 is only ±1 μA. Thus, it takes very little power to maintain a CMOS input in one state or the other. This is in sharp contrast to older bipolar logic circuits like TTL, whose inputs consume significant current (and power) in one or both states.

14.3.2 Circuit Behavior with Resistive Loads

As mentioned previously, CMOS gate inputs have very high impedance and consume very little current from the circuits that drive them. There are other devices, however, which require nontrivial amounts of current to operate. When such a device is connected to a CMOS output, we call it a *resistive load* or a *DC load.* Here are some examples of resistive loads:

resistive load
DC load

- Discrete resistors may not really be present in the circuit, but the load presented by one or more TTL or other non-CMOS inputs may be modeled by a simple resistor network.
- The resistors may be part of or may model a current-consuming device like a light-emitting diode (LED) or a relay coil.
- Discrete resistors may be included to improve signal quality by providing transmission-line termination, discussed in electrical-engineering texts and the first three editions of this book.

When the output of a CMOS circuit is connected to a resistive load, the output behavior is not nearly as ideal as we described previously. In either logic state, the CMOS output transistor that is “on” has a nonzero resistance, and a load connected to the output terminal will cause a voltage drop across this resistance. Thus, in the LOW state, the output voltage may be somewhat higher than 0.1 V, and in the HIGH state, it may be lower than 4.4 V. The easiest way to see how this happens is look at a resistive model of the CMOS circuit and its load.

Figure 14-23(a) shows the resistive model. The *p*-channel and *n*-channel transistors have resistances R_p and R_n, respectively. In normal operation, one resistance is high (> 1 MΩ) and the other is low (perhaps 100 Ω), depending on whether the input voltage is HIGH or LOW. The load in this circuit consists of

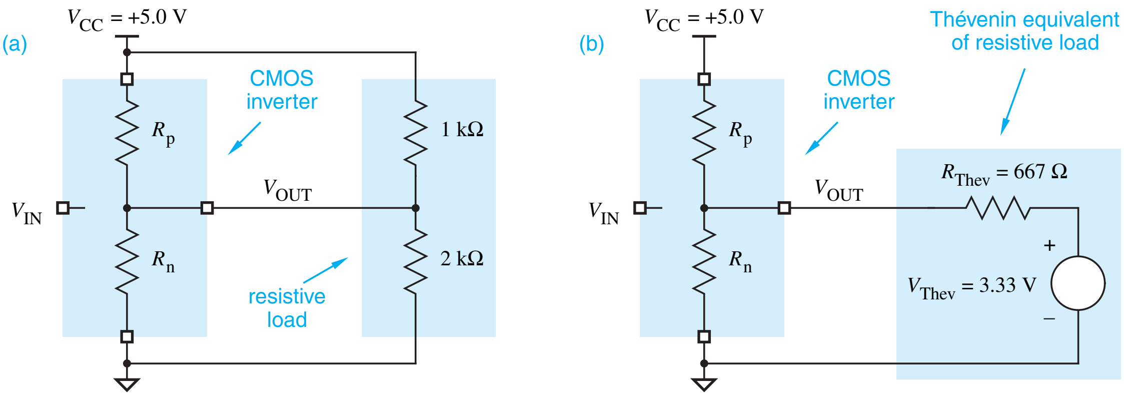


Figure 14-23 Resistive model of a CMOS inverter with a resistive load: (a) showing actual load circuit; (b) using Thévenin equivalent of load.

REMEMBERING THÉVENIN

Any two-terminal circuit consisting of only voltage sources and resistors can be modeled by a *Thévenin equivalent* consisting of a single voltage source in series with a single resistor. The *Thévenin voltage* is the open-circuit voltage of the original circuit, and the *Thévenin resistance* is the Thévenin voltage divided by the original circuit's short-circuit current (defined as the current that flows when the circuit's two terminals are shorted to each other).

In the example of Figure 14-23, the Thévenin voltage of the resistive load, including its connection to V_{CC}, is established by the 1-kΩ and 2-kΩ resistors, which form a voltage divider:

$$V_{\text{Thev}} = \frac{2\ \text{k}\Omega}{2\ \text{k}\Omega + 1\,\text{k}\Omega} \cdot 5.0\ \text{V} = 3.33\ \text{V}$$

The short-circuit current is (5.0 V)/(1 kΩ) = 5 mA, so the Thévenin resistance is (3.33 V) / (5 mA) = 667 Ω. Readers who are electrical engineers may recognize this as the parallel resistance of the 1-kΩ and 2-kΩ resistors.

two resistors attached to the supply rails; a real circuit may have any resistor values, or an even more complex resistive network. In any case, a resistive load, consisting only of resistors and voltage sources, can always be modeled by a Thévenin equivalent network, like the one shown in Figure 14-23(b).

When the CMOS inverter has a HIGH input, the output should be LOW; the actual output voltage can be predicted using the resistive model shown in Figure 14-24. The *p*-channel transistor is "off" and has a very high resistance, high enough to be negligible in the calculations that follow. The *n*-channel transistor is "on" and has a low resistance, which we assume to be 100 Ω. (The actual "on" resistance depends on the CMOS family and other characteristics like operating temperature and whether or not the device was manufactured on a good day.) The "on" transistor and the Thévenin-equivalent resistor R_{Thev} in

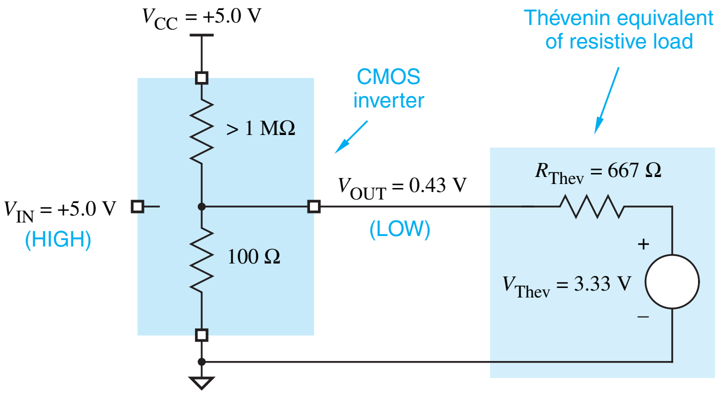


Figure 14-24 Resistive model for CMOS LOW output with resistive load.

Figure 14-24 form a simple voltage divider. The resulting output voltage can be calculated as follows:

$$\begin{aligned} V_{\mathrm{OUT}} &= 3.33\ \mathrm{V} \cdot [100/(100 + 667)] \\ &= 0.43\ \mathrm{V} \end{aligned}$$

Similarly, when the inverter has a LOW input, the output should be HIGH, and the actual output voltage can be predicted with the model in Figure 14-25. We'll assume that the *p*-channel transistor's "on" resistance is 200 Ω. Once again, the "on" transistor and the Thévenin-equivalent resistor R_{Thev} in the figure form a simple voltage divider, and the resulting output voltage can be calculated as follows:

$$\begin{aligned} V_{\mathrm{OUT}} &= 3.33\ \mathrm{V} + (5\ \mathrm{V} - 3.33\ \mathrm{V}) \cdot [667/(200 + 667)] \\ &= 4.61\ \mathrm{V} \end{aligned}$$

In practice, it's seldom necessary to calculate output voltages as in the preceding examples. In fact, IC manufacturers usually don't specify the equivalent resistances of the "on" transistors, so you wouldn't have the necessary information to make the calculation anyway. Instead, IC manufacturers specify a maximum load for the output in each state (HIGH or LOW), and guarantee a worst-case output voltage for that load. The load is specified in terms of current:

I_{OLmax} The maximum current that the output can sink in the LOW state while still maintaining an output voltage no greater than V_{OLmax}.

I_{OHmax} The maximum current that the output can source in the HIGH state while still maintaining an output voltage no less than V_{OHmin}.

These definitions are illustrated in Figure 14-26. A device output is said to *sink current* when current flows from the power supply, through the load, and through the device output to ground as in (a). The output is said to *source current*

sinking current

sourcing current

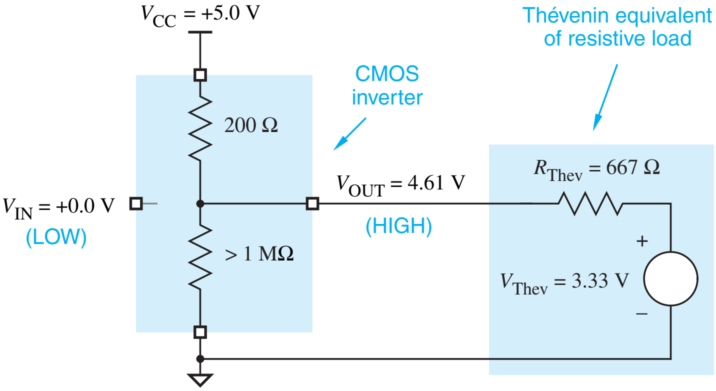


Figure 14-25 Resistive model for CMOS HIGH output with resistive load.

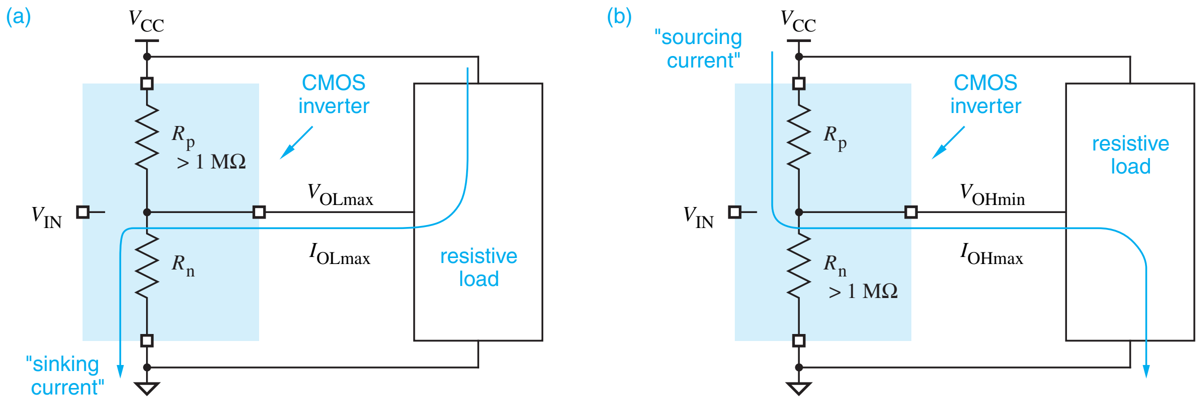


Figure 14-26 Circuit definitions of (a) I_{OLmax}; (b) I_{OHmax}.

when current flows from the power supply, out of the device output, and through the load to ground as in (b).

Most CMOS devices have two sets of loading specifications. One set is for “CMOS loads,” where the device output is connected to other CMOS inputs, which consume very little current. The other set is for “TTL loads,” where the output is connected to resistive loads like TTL inputs or other devices that consume significant current. For example, the specifications for HC-series CMOS outputs were included in Table 14-1 and are repeated in Table 14-2.

current flow

Notice in Table 14-2 that the output current in the HIGH state is shown as a negative number. By convention, the *current flow* measured at a device terminal is positive if positive current flows *into* the device; in the HIGH state, current flows *out* of the output terminal.

As the table shows, with CMOS loads, the CMOS gate’s output voltage is maintained within 0.1 V of the power-supply rail. With TTL loads, the output voltage may degrade quite a bit. Also notice that for the same output current (±4 mA) the maximum voltage drop with respect to the power-supply rail is twice as much in the HIGH state (0.66 V) as in the LOW state (0.33 V). This

Table 14-2 Output loading specifications for HC-series CMOS with a 5V ±10% supply.

	CMOS Load		***TTL Load***	
Parameter	***Name***	***Value***	***Name***	***Value***
Maximum LOW-state output current (mA)	I_{OLmaxC}	0.02	I_{OLmaxT}	4.0
Maximum LOW-state output voltage (V)	V_{OLmaxC}	0.1	V_{OLmaxT}	0.33
Maximum HIGH-state output current (mA)	I_{OHmaxC}	–0.02	I_{OHmaxT}	–4.0
Minimum HIGH-state output voltage (V)	V_{OHminC}	4.4	V_{OHminT}	3.84

suggests that the *p*-channel transistors in HC-series CMOS have a higher "on" resistance than the *n*-channel transistors do. This is natural, since in any CMOS circuit, a *p*-channel transistor has over twice the "on" resistance of an *n*-channel transistor with the same area. Equal voltage drops in both states could be obtained by making the *p*-channel transistors much larger than the *n*-channel transistors, but for various reasons this was not done in the HC family.

Ohm's law can be used to determine how much current an output sources or sinks in a given situation. In Figure 14-24 on page 754, the "on" *n*-channel transistor modeled by a 100-Ω resistor has a 0.43-V drop across it; therefore it sinks (0.43 V)/(100 Ω) = 4.3 mA of current. Similarly, the "on" *p*-channel transistor in Figure 14-25 sources $(0.39\text{V})/(200\Omega) = 1.95$ mA.

The actual "on" resistances of CMOS output transistors usually aren't published, so it's generally not possible to use the exact models of the previous paragraphs. However, you can estimate "on" resistances using the following equations, which rely on specifications that are always published:

$$R_{\mathrm{p(on)}} \approx \frac{V_{\mathrm{CC}} - V_{\mathrm{OHminT}}}{|I_{\mathrm{OHmaxT}}|}$$

$$R_{\mathrm{n(on)}} \approx \frac{V_{\mathrm{OLmaxT}}}{I_{\mathrm{OLmaxT}}}$$

These equations use Ohm's law to compute the "on" resistance as the voltage drop across the "on" transistor (or series of transistors) divided by the current through it (or them) with a worst-case resistive load. Using the numbers given for HC-series CMOS in Table 14-2, we can thus calculate $R_{\mathrm{p(on)}} \approx 165\ \Omega$ and $R_{\mathrm{n(on)}} \approx 82.5\ \Omega$. Note that $V_{\mathrm{CC}} = 4.5$ V (the minimum value) for this calculation.

Very good *worst-case* estimates of output current can be made by assuming that there is *no* voltage drop across the "on" transistor. This assumption simplifies the analysis, and yields a conservative result that is almost always good enough for practical purposes. For example, Figure 14-27 shows a CMOS

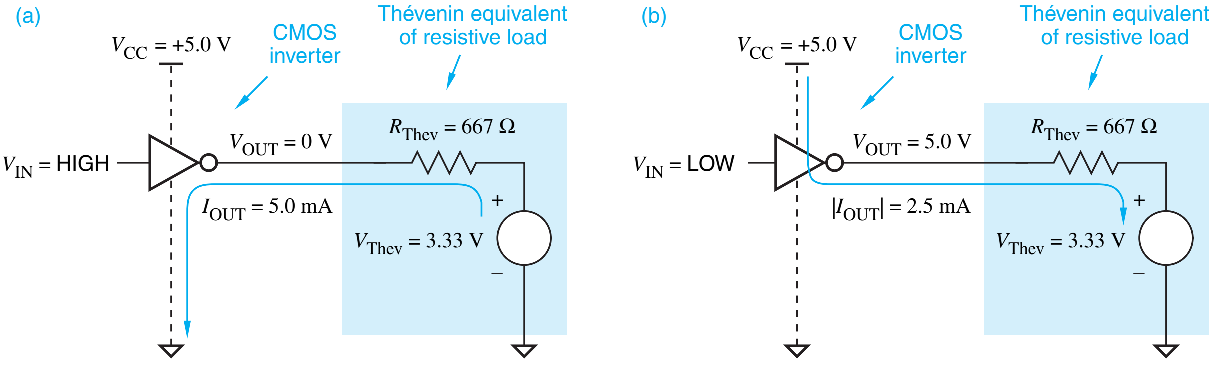


Figure 14-27 Estimating sink and source current: (a) output LOW; (b) output HIGH.

PIDDLING CURRENT?

As we’ve stated elsewhere, an “off” transistor’s resistance is over one megohm, but it’s not infinite. Therefore, a very tiny leakage current actually does flow in “off” transistors, and the CMOS output structure does have a correspondingly tiny but nonzero power consumption. In most applications, this power consumption is tiny enough to ignore.

However, leakage current and the corresponding power consumption can be significant in “standby mode” in battery-powered devices, such as laptop computers. phones, and wearables. Leakage current can also be a significant concern in the densest, highest-performance IC technologies, with tens of millions of transistors per chip. As the transistors get smaller, their individual leakage currents may increase, just as the total number of transistors per chip is increasing. The net effect is that as much as half of the chip’s total power consumption may result from leakage.

inverter driving the same Thévenin-equivalent load that we’ve used in previous examples. The resistive model of the output structure is not shown, because it is no longer needed; we assume that there is no voltage drop across the “on” CMOS transistor. In (a), with the output LOW, the entire 3.33-V Thévenin-equivalent voltage source appears across R_{Thev}, and the estimated sink current is $(3.33\,\mathrm{V})/(667\,\Omega) = 5.0$ mA. In (b), with the output HIGH and assuming a 5.0-V supply, the voltage drop across R_{Thev} is 1.67 V, and the estimated source current is $(1.67\,\mathrm{V})/(667\,\Omega) = 2.5$ mA.

An important feature of the CMOS inverter (or any CMOS circuit) is that the output structure by itself consumes very little current in either state, HIGH or LOW. In either state, one of the transistors is in the high-impedance “off” state. All of the current flow that we’ve been talking about occurs when a resistive load is connected to the CMOS output. If there’s no load, then no significant current flows, and power consumption is near zero. With a load, however, current flows through both the load and the “on” transistor, and power is consumed in both.

14.3.3 Circuit Behavior with Nonideal Inputs

So far, we have assumed that the HIGH and LOW inputs to a CMOS circuit are ideal voltages, very close to the power-supply rails. However, the behavior of a real CMOS inverter circuit depends on the input voltage as well as on the characteristics of the load. If the input voltage is not close to the power-supply rail, then the “on” transistor may not be fully “on” and its resistance may increase. Likewise, the “off” transistor may not be fully “off” and its resistance may be quite a bit less than one megohm. These two effects combine to move the output voltage away from the power-supply rail.

For example, Figure 14-28(a) shows a CMOS inverter’s possible behavior with a 1.5-V input. The *p*-channel transistor’s resistance has doubled at this point, and the *n*-channel transistor is beginning to turn on. (These values are just

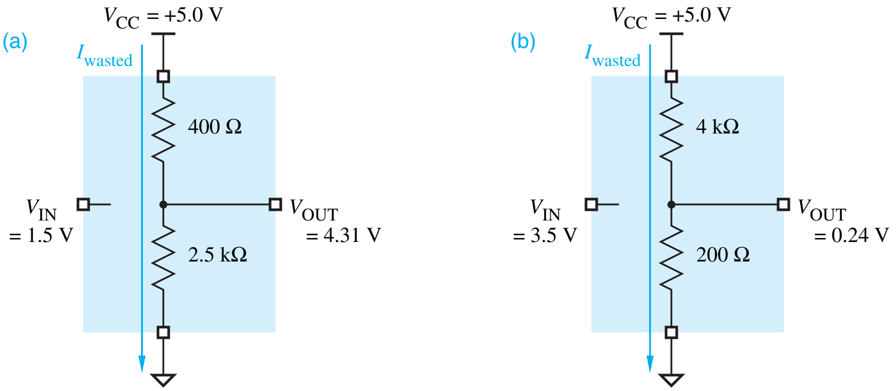


Figure 14-28 CMOS inverter with nonideal input voltages: (a) equivalent circuit with 1.5-V input; (b) equivalent circuit with 3.5-V input.

assumed for the purposes of illustration; the actual values depend on the detailed characteristics of the transistors.)

In the figure, the output at 4.31 V is still well within the valid range for a HIGH signal, but not quite the ideal of 5.0 V. Similarly, with a 3.5-V input in (b), the LOW output is 0.24 V, not 0 V. The slight degradation of output voltage is generally tolerable; what's worse is that the output structure is now consuming a nontrivial amount of power. The current flow with the 1.5-V input is

$$I_{\text{wasted}} = 5.0\ \text{V} / (400\ \Omega + 2.5\ \text{k}\Omega) = 1.72\ \text{mA}$$

and the power consumption is

$$P_{\text{wasted}} = 5.0\ \text{V} \cdot I_{\text{wasted}} = 8.62\ \text{mW}$$

The output voltage of a CMOS inverter deteriorates further with a resistive load. Such a load may exist for any of a variety of reasons discussed previously. Figure 14-29 shows the CMOS inverter's possible behavior with a resistive load. With a 1.5-V input, the output at 3.98 V is still within the valid range for a HIGH

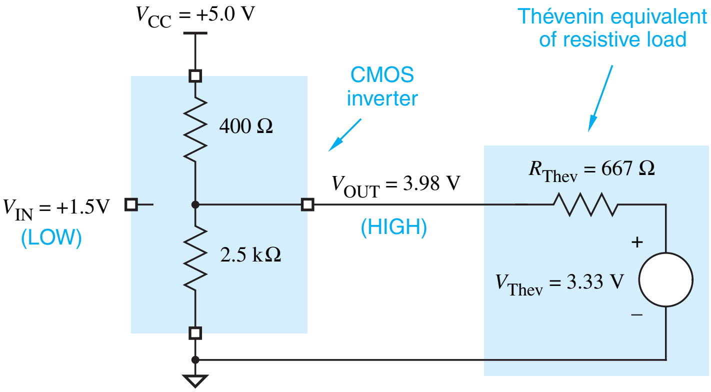
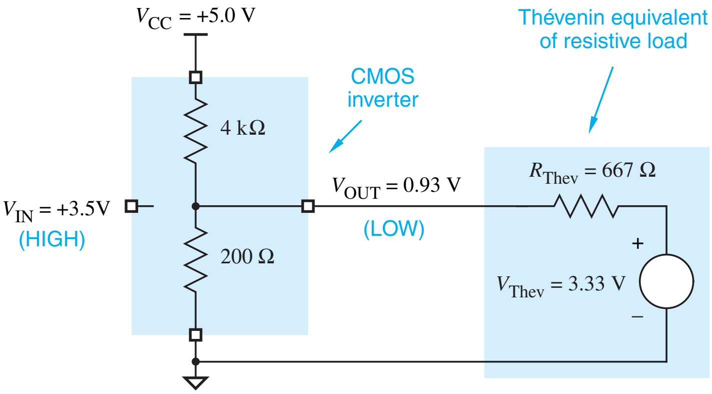


Figure 14-29 CMOS inverter with load and nonideal 1.5-V input.

Figure 14-30 CMOS inverter with load and nonideal 3.5-V input.



output signal, but it is still farther from the ideal of 5.0 V. Similarly, with a 3.5-V input as shown in Figure 14-30, the LOW output is 0.93 V, not 0 V.

In "pure" CMOS systems, all of the logic devices in a circuit are CMOS. Since CMOS inputs have a very high impedance, they present very little resistive load to the CMOS outputs that drive them. Therefore, the CMOS output levels all remain very close to the power-supply rails (0 V and 5 V), and none of the devices waste power in their output structures. In "non-pure" CMOS systems, additional power can be consumed in two ways:

- If nonideal logic signals are connected to CMOS inputs, then the CMOS outputs use power as described in this subsection.
- If resistive loads are connected to CMOS outputs, then the CMOS outputs use power in the way depicted in the preceding subsection.

14.3.4 Fanout

fanout

The *fanout* of a logic gate is the number of inputs that the gate can drive without exceeding its worst-case loading specifications. The fanout depends not only on the characteristics of the output, but also on the inputs that it is driving. Fanout must be examined for both possible output states, HIGH and LOW.

LOW-state fanout

HIGH-state fanout

For example, we showed in Table 14-2 on page 756 that the maximum LOW-state output current I_{OLmaxC} for an HC-series CMOS gate driving CMOS inputs is 0.02 mA (20 μA). We also stated previously that the maximum input current I_{Imax} for an HC-series CMOS input in any state is ±1 μA. Therefore, the *LOW-state fanout* for an HC-series output driving HC-series inputs is 20. Table 14-2 also showed that the maximum HIGH-state output current I_{OHmaxC} is –0.02 mA (–20 μA) Therefore, the *HIGH-state fanout* for an HC-series output driving HC-series inputs is also 20.

overall fanout

Note that the HIGH-state and LOW-state fanouts of a gate aren't necessarily equal. In general, the *overall fanout* of a gate is the minimum of its HIGH-state and LOW-state fanouts, 20 in the foregoing example.

In the fanout example that we just completed, we assumed that we needed to maintain the gate's output at CMOS levels, that is, within 0.1 V of the power-supply rails. If we could live with somewhat degraded output levels specified for "TTL loads," then we could use I_{OLmaxT} and I_{OHmaxT} in the fanout calculation. Table 14-2 gives these specifications as 4.0 mA and –4.0 mA, respectively. Therefore, the fanout of an HC-series output driving HC-series inputs at "TTL levels" is 4000—for practical purposes, virtually unlimited, apparently.

DC fanout

Well, not quite. The calculations that we've just carried out give the *DC fanout,* defined as the number of inputs that an output can drive *with the output in a constant state* (HIGH or LOW). Even if the DC fanout specification is met, a CMOS output driving a large number of inputs may not behave satisfactorily on transitions, LOW-to-HIGH or vice versa.

During transitions, the CMOS output must charge or discharge the stray capacitance associated with the inputs that it drives, including that of the wiring that connects it to them. If this capacitance is too large, the transition from LOW to HIGH (or vice versa) may be too slow, causing improper system operation.

AC fanout

The ability of an output to charge and discharge stray capacitance is sometimes called *AC fanout*, though in board-level design it is seldom calculated as precisely as DC fanout. As you'll see in Section 14.4.1, it's more a matter of deciding how much speed degradation you're willing to tolerate.

14.3.5 Effects of Loading

Loading an output beyond its rated fanout has several effects:

- In the LOW state, the output voltage (V_{OL}) may increase beyond V_{OLmax}.
- In the HIGH state, the output voltage (V_{OH}) may fall below V_{OHmin}.
- Propagation delay to the output may increase beyond specifications.
- Output rise and fall times may increase beyond their specifications.
- The operating temperature of the device may increase, thereby reducing the reliability of the device and hastening device failure.

The first four effects reduce the DC noise margins and the timing margins of the circuit. Thus, a slightly overloaded circuit may work properly in ideal conditions, but experience says that it will fail once it's out of the friendly environment of the engineering lab.

14.3.6 Unused Inputs

In board-level design, some of the inputs of a logic gate, an MSI function, or even an LSI chip might not be used. At the lowest level, in a real design problem, you may need an n-input gate but have only an $(n+1)$-input gate available. Tying together two inputs of the $(n+1)$-input gate gives it the functionality of an n-input gate. You can convince yourself of this fact intuitively now, or use

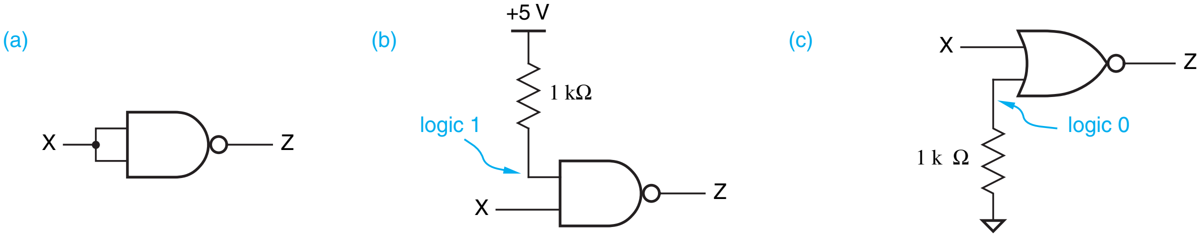


Figure 14-31 Unused inputs: (a) tied to another input; (b) NAND pulled up; (c) NOR pulled down.

switching algebra as you studied in Section 3.1. Figure 14-31(a) shows a NAND gate with its inputs tied together.

You can also tie unused inputs to a constant logic value. An unused AND or NAND input should be tied to logic 1, as in (b), and an unused OR or NOR input should be tied to logic 0, as in (c). With MSI functions and LSI chips, the unused input should be tied to a value appropriate for the unused function, which in some cases may be either value (e.g., for an unused D input of a register). In high-speed circuit design, it's usually better to use method (b) or (c) rather than (a), which increases the capacitive load on the driving signal and may slow things down. In (b) and (c), a resistor value in the range 1–10 kΩ is typically used, and a single pull-up or pull-down resistor can serve multiple unused inputs. It is also possible to tie unused inputs directly to the power-supply rail, though this may not be advisable for one or more reasons:

- In some logic families, a direct connection to V_{CC} is not recommended because of the need to limit input current in certain transient situations which are beyond the scope of our discussion.
- Tying an input directly to a rail makes it more difficult to rework the board or to apply a real signal to it for testing or debugging purposes. The same is true if multiple unused inputs are tied together. (On the other hand, lots of discrete pull-up and pull-down resistors take up lots of space.)

floating input

In any case, unused CMOS inputs should never be left unconnected (or *floating*). On one hand, such an input will behave as if it had a LOW signal applied to it, and it will normally show a value of 0 V when probed with an oscilloscope or voltmeter. So you might think that an unused OR or NOR input can be

SUBTLE BUGS Floating CMOS inputs are often the cause of mysterious circuit behavior, as an unused input erratically changes its effective state based on noise and conditions elsewhere in the circuit. When you're trying to debug such a problem, the extra capacitance of an oscilloscope probe touched to the floating input is often enough to damp out the noise and make the problem go away. This can be especially baffling if you don't realize that the input is floating!

left floating, because it will act as if a logic 0 is applied and will not affect the gate's output. However, since CMOS inputs have such high impedance, it takes only a small amount of circuit noise to temporarily make a floating input look HIGH, creating some very nasty intermittent circuit failures.

Luckily, LSI chips with many inputs (such as FPGAs and microprocessors) normally have internal pull-up or pull-down resistors, usually configurable by programming, so that an external signal does not need to be applied to an otherwise unused input.

14.3.7 How to Destroy a CMOS Device

Hit it with a sledgehammer. Or simply walk across a carpet and then touch an input pin with your finger. Because CMOS device inputs have such high impedance, they are subject to damage from *electrostatic discharge (ESD)*.

electrostatic discharge (ESD)

ESD occurs when a buildup of charge ("static electricity") on one surface arcs through a dielectric to another surface with the opposite charge. In the case of a CMOS input, the dielectric is the insulation between an input transistor's gate and its source and drain. ESD may damage this insulation, causing a short-circuit between the device's input and its output.

Ordinary activities of people, like walking on a carpet, can create static electricity with surprisingly high voltage potentials—1000 V or more. The input structures of modern CMOS devices use various means to reduce their susceptibility to ESD damage, but no device is completely immune. Therefore, to protect CMOS devices from ESD damage during shipment and handling, manufacturers normally package their devices in conductive bags, tubes, or foam. To prevent ESD damage when handling loose CMOS devices, circuit assemblers and technicians usually wear conductive wrist straps that are connected by a coil cord to earth ground; this prevents a static charge from building up on their bodies as they move around the factory or lab.

Ordinary operation of some equipment, such as repeated or continuous movement of mechanical components like doors or fans, can also create static

ELIMINATE RUDE, SHOCKING BEHAVIOR!

Some design engineers consider themselves above such inconveniences, but to be safe you should follow several ESD precautions in the lab:

- Before handling a CMOS device, touch the grounded metal case of a plugged-in instrument or another source of earth ground.
- Before transporting a CMOS device, insert it in conductive foam.
- When carrying a circuit board containing CMOS devices, handle the board by the edges, and touch a ground terminal on the board to earth ground before poking around with it.
- When handing over a CMOS device to a partner, especially on a dry winter day, touch the partner first. Your partner will thank you for it.

electricity. For that reason, printed-circuit boards containing CMOS circuits are carefully designed with ESD protection in mind. Typically, this means grounding connector housings, the edges of the board, and any other points where static might be encountered because of proximity to people or equipment. This "encourages" ESD to take a safe, metallic path to ground, rather than through the pins of CMOS chips mounted on the board.

14.4 CMOS Dynamic Electrical Behavior

Both the speed and the power consumption of a CMOS device depend to a large extent on "AC" or dynamic characteristics of the device and its load, that is, what happens when the output changes between states. As part of the internal design of CMOS ASICs, digital designers must carefully examine the effects of output loading, and resize or redesign circuits where the load is too high. Even in board-level design, the effects of loading must be considered for clocks, buses, and other signals that have high fanout or long interconnections.

Speed depends on two characteristics: transition time and propagation delay, discussed in the next two subsections. Power dissipation will be discussed in the third subsection, and a few nasty real-world effects will be discussed in the last three subsections.

14.4.1 Transition Time

transition time

rise time (t_r)

fall time (t_f)

The amount of time that the output of a logic circuit takes to change from one state to another is called the *transition time*. Figure 14-32(a) shows how we might like outputs to change state—in zero time. However, real outputs cannot change instantaneously, because they need time to charge the stray capacitance of the wires and other components that they drive. A more realistic view of a circuit's output is shown in (b). An output takes a certain time, called the *rise time* (t_r), to change from LOW to HIGH, and a possibly different time, called the *fall time* (t_f), to change from HIGH to LOW.

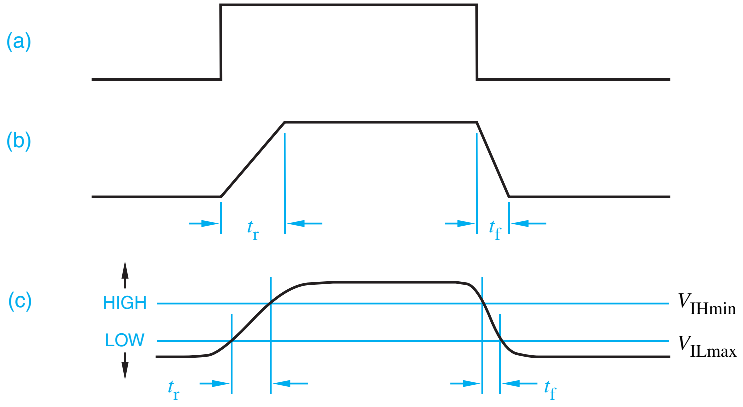


Figure 14-32 Transition times: (a) ideal case of zero-time switching; (b) a more realistic approximation; (c) actual timing, with rise and fall times.

Even Figure 14-32(b) is not quite accurate, because the rate of change of the output voltage does not change instantaneously, either. Instead, the beginning and the end of a transition are smooth, as shown in (c). To avoid difficulties in defining the endpoints, rise and fall times are often measured at the boundaries of the valid logic levels as indicated in the figure, or sometimes at the 10% and 90% points in the signal's voltage range.

With the convention in Figure 14-32(c), the rise and fall times indicate how long an output voltage takes to pass through the "undefined" region between LOW and HIGH. The initial part of a transition is not included in the rise- or fall-time number. Instead, the initial part of a transition contributes to the "propagation delay" number discussed in the next subsection.

The rise and fall times of a CMOS output depend mainly on two factors, the "on" transistor resistance and the load capacitance. A large capacitance increases transition times; since this is undesirable, it is very rare for a digital designer to purposely connect a capacitor to a logic circuit's output. However, *stray capacitance* is present in every circuit; in board-level design, it comes from at least three sources:

stray capacitance

1. Output circuits, including a gate's output transistors, internal wiring, and packaging, have some capacitance associated with them, in the range of 2–10 picofarads (pF) in typical logic families, including CMOS.
2. The wiring that connects an output to other inputs has capacitance, about 1 pF per inch or more, depending on the wiring technology.
3. Input circuits, including transistors, internal wiring, and packaging, have capacitance, from 2 to 15 pF per input in typical logic families.

Stray capacitance is sometimes called a *capacitive load* or an *AC load*.

capacitive load

AC load

A CMOS output's rise and fall times can be analyzed using the equivalent circuit shown in Figure 14-33. As in the preceding section, the p-channel and n-channel transistors are modeled by resistances R_p and R_n, respectively. In normal operation, one resistance is high and the other is low, depending on the

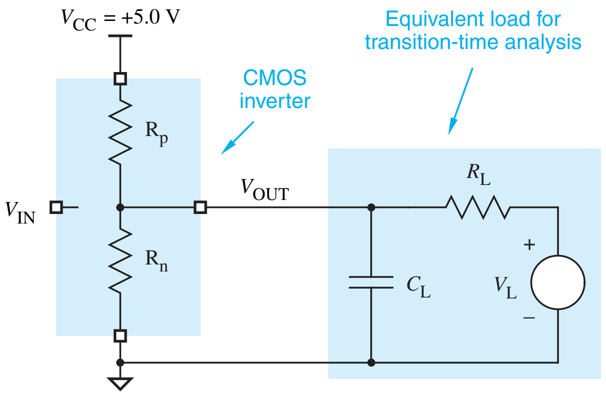


Figure 14-33 Equivalent circuit for analyzing transition times of a CMOS output.

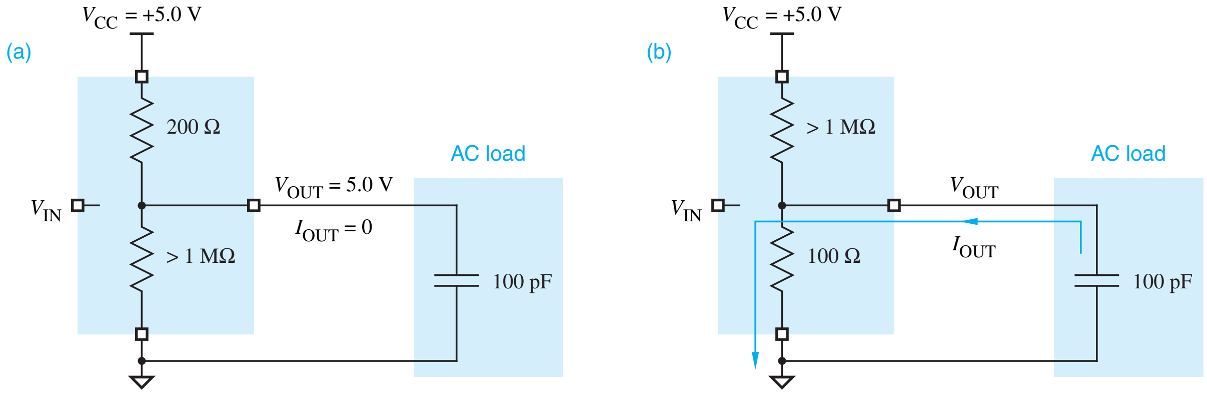


Figure 14-34 Model of a CMOS HIGH-to-LOW transition: (a) in the HIGH state; (b) after *p*-channel transistor turns off and *n*-channel transistor turns on.

equivalent load circuit

output's state. The output's load is modeled by an *equivalent load circuit* with three components:

R_L, V_L These two components represent the DC load. They determine the voltages and currents that are present when the output has settled into a stable HIGH or LOW state. The DC load doesn't have too much effect on transition times when the output changes state.

C_L This capacitance represents the AC load. It determines the voltages and currents that are present while the output is changing, and how long it takes to change from one state to the other.

When a CMOS output drives only CMOS inputs, the DC load is negligible. To simplify matters, we'll analyze only this case, with $R_L = \infty$ and $V_L = 0$, in the remainder of this subsection. The presence of a nonnegligible DC load would affect the results, but not dramatically (see Exercise 14.66).

We can now analyze the transition times of a CMOS output. For the purpose of this analysis, we'll assume C_L= 100 pF, a moderate capacitive load in board-level design. Also, we'll assume that the "on" resistances of the *p*-channel and *n*-channel transistors are 200 Ω and 100 Ω, respectively, as in the preceding subsection. The rise and fall times depend on how long it takes to charge or discharge the capacitive load C_L.

First, we'll look at fall time. Figure 14-34(a) shows the electrical conditions in the circuit when the output is in a steady HIGH state. (R_L and V_L are not drawn; they have no effect, since we assume $R_L = \infty$.) For the purposes of our analysis, we'll assume that when CMOS transistors change between "on" and "off," they do so instantaneously. We'll assume that at time t = 0 the CMOS output changes to the LOW state, resulting in the situation depicted in (b).

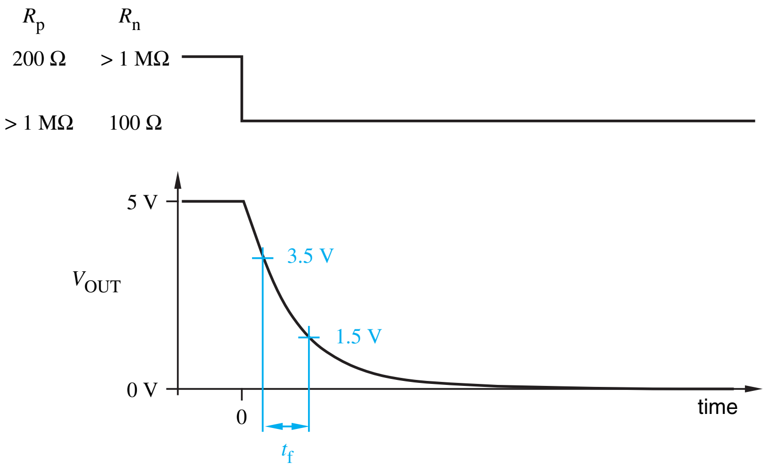


Figure 14-35 Fall time for a HIGH-to-LOW transition of a CMOS output.

At time $t = 0$, V_{OUT} is still 5.0 V. (A useful electrical-engineering maxim is that the voltage across a capacitor cannot change instantaneously.) At time $t = \infty$, the capacitor must be fully discharged and V_{OUT} will be 0 V. In between, the value of V_{OUT} is governed by an exponential law:

$$\begin{aligned} V_{OUT} &= V_{DD} \cdot e^{-t/(R_n C_L)} \\ &= 5.0 \cdot e^{-t/(100 \cdot 100 \cdot 10^{-12})} \text{ V} \\ &= 5.0 \cdot e^{-t/(10 \cdot 10^{-9})} \text{ V} \end{aligned}$$

RC time constant

The factor $R_n C_L$ has units of seconds and is called an *RC time constant*. The preceding calculation shows that the *RC* time constant for HIGH-to-LOW transitions is 10 nanoseconds (ns).

Figure 14-35 plots V_{OUT} as a function of time. To calculate fall time, recall that 1.5 V and 3.5 V are the defined boundaries for LOW and HIGH levels for CMOS inputs being driven by the CMOS output. To obtain the fall time, we must solve the preceding equation for $V_{OUT} = 3.5$ and $V_{OUT} = 1.5$, yielding:

$$\begin{aligned} t &= -R_n C_L \cdot \ln \frac{V_{OUT}}{V_{DD}} \\ &= -10 \cdot 10^{-9} \cdot \ln \frac{V_{OUT}}{5.0} \\ t_{3.5} &= 3.57 \text{ ns} \\ t_{1.5} &= 12.04 \text{ ns} \end{aligned}$$

The fall time t_f is the difference between these two numbers, or about 8.5 ns.

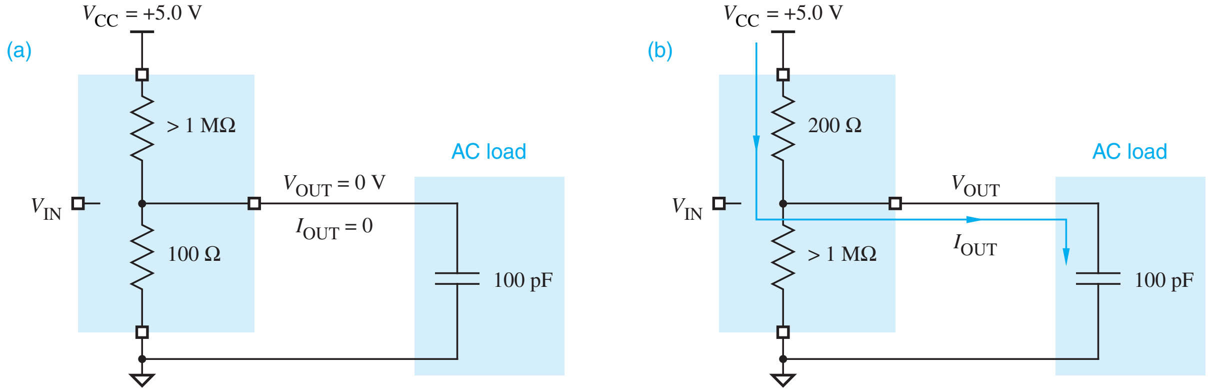


Figure 14-36 Model of a CMOS LOW-to-HIGH transition: (a) in the LOW state; (b) after *n*-channel transistor turns off and *p*-channel transistor turns on.

Rise time can be calculated in a similar manner. Figure 14-36(a) shows the conditions in the circuit when the output is in a steady LOW state. If at time $t = 0$ the CMOS output changes to the HIGH state, the situation depicted in (b) results. Once again, V_{OUT} cannot change instantly, but at time $t = \infty$, the capacitor will be fully charged and V_{OUT} will be 5.0 V. Once again, the value of V_{OUT} in between is governed by an exponential law:

$$\begin{aligned} V_{OUT} &= V_{DD} \cdot (1 - e^{-t/(R_p C_L)}) \\ &= 5.0 \cdot (1 - e^{-t/(200 \cdot 100 \cdot 10^{-12})}) \text{ V} \\ &= 5.0 \cdot (1 - e^{-t/(20 \cdot 10^{-9})}) \text{ V} \end{aligned}$$

The *RC* time constant in this case is 20 ns. Figure 14-37 plots V_{OUT} as a function of time. To obtain the rise time, we must solve the preceding equation for $V_{OUT} = 1.5$ and $V_{OUT} = 3.5$, yielding

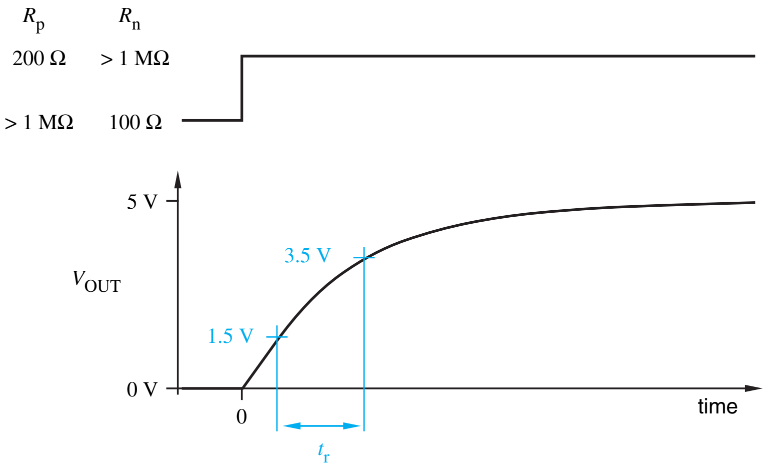


Figure 14-37 Rise time for a LOW-to-HIGH transition of a CMOS output.

$$
\begin{aligned}
t &= -RC \cdot \ln \frac{V_{\mathrm{DD}} - V_{\mathrm{OUT}}}{V_{\mathrm{DD}}} \\
&= -20 \cdot 10^{-9} \cdot \ln \frac{5.0 - V_{\mathrm{OUT}}}{5.0} \\
t_{1.5} &= 7.13 \text{ ns} \\
t_{3.5} &= 24.08 \text{ ns}
\end{aligned}
$$

The rise time t_{r} is the difference between these two numbers, or about 17 ns.

The foregoing example assumes that the *p*-channel transistor has twice the resistance of the *n*-channel transistor, and as a result the rise time is twice as long as the fall time. It takes longer for the "weak" *p*-channel transistor to pull the output up than it does for the "strong" *n*-channel transistor to pull it down; the output's drive capability is "asymmetric." High-speed CMOS devices are sometimes fabricated with larger *p*-channel transistors to make the transition times more nearly equal and output drive more symmetric.

Regardless of the transistors' characteristics, an increase in load capacitance causes an increase in the *RC* time constant and a corresponding increase in the transition times of the output. Thus, it is a goal of high-speed circuit designers to minimize load capacitance, especially on the most timing-critical signals. This can be done by minimizing the number of inputs driven by the signal, by creating multiple copies of the signal, and by careful physical layout of the circuit.

When working with real digital circuits, it's often useful to estimate transition times, without going through a detailed analysis. A useful rule of thumb is that the transition time approximately equals the *RC* time constant of the charging or discharging circuit. For example, estimates of 10 and 20 ns for fall and rise time respectively in the preceding example would have been pretty much on target, especially considering that most assumptions about load capacitance and transistor "on" resistances are approximate to begin with.

Manufacturers of commercial CMOS circuits typically do not specify transistor "on" resistances on their data sheets. If you search carefully, you might find this information published in the manufacturers' application notes. In any case, you can estimate an "on" resistance as the voltage drop across the "on" transistor divided by the current through it with a worst-case resistive load, as we showed in Section 14.3.2:

$$
R_{\mathrm{p(on)}} \approx \frac{V_{\mathrm{DD}} - V_{\mathrm{OHminT}}}{|I_{\mathrm{OHmaxT}}|}
$$

$$
R_{\mathrm{n(on)}} \approx \frac{V_{\mathrm{OLmaxT}}}{I_{\mathrm{OLmaxT}}}
$$

THERE'S A CATCH!

Calculated transition times are actually quite sensitive to the choice of logic levels. In the examples in this subsection, if we used 2.0 V and 3.0 V instead of 1.5 V and 3.5 V as the thresholds for LOW and HIGH, respectively, we would calculate shorter transition times. On the other hand, if we used 0.0 and 5.0 V, the calculated transition times would be infinity! You should also be aware that in some logic families (most notably TTL), the thresholds are not symmetric around the voltage midpoint. Still, it is the author's experience that the "time-constant-equals-transition-time" rule of thumb usually works well enough for practical board-level circuits.

14.4.2 Propagation Delay

signal path

propagation delay t_p

Rise and fall times only partially describe the dynamic behavior of a logic element; we need additional parameters to relate output timing to input timing. A *signal path* is the electrical path from a particular input signal to a particular output signal of a logic element. The *propagation delay* t_p of a signal path is the amount of time that it takes for a change in the input signal to produce a change in the output signal.

A complex logic element with multiple inputs and outputs may specify a different value of t_p for each different signal path. Also, different values may be specified for a particular signal path, depending on the direction of the output change. Assuming zero rise and fall times for simplicity, Figure 14-38(a) shows two different propagation delays for the input-to-output signal path of a CMOS inverter, depending on the direction of the output change:

t_{pHL}

t_{pLH}

t_{pHL} The time between an input change and the corresponding output change when the output is changing from HIGH to LOW.

t_{pLH} The time between an input change and the corresponding output change when the output is changing from LOW to HIGH.

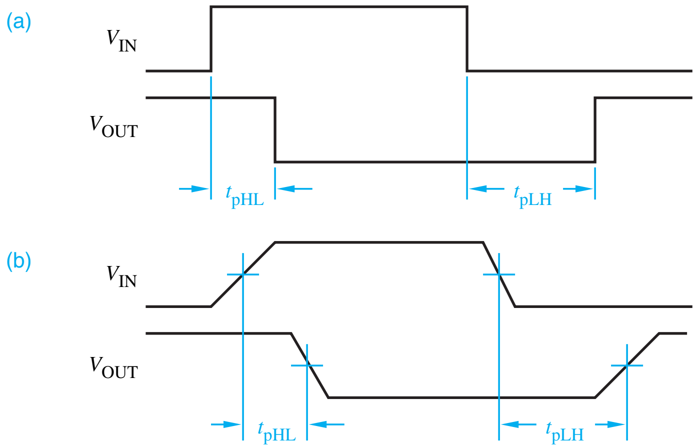


Figure 14-38 Propagation delays for a CMOS inverter: (a) ignoring rise and fall times; (b) measured at midpoints of transitions.

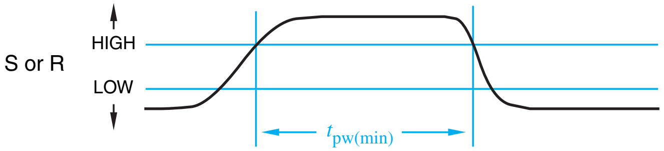


Figure 14-39 Worst-case timing specified using logic-level boundary points.

Several factors lead to nonzero propagation delays. In a CMOS device, the rate at which transistors change state is influenced both by the semiconductor physics of the device and by the circuit environment, including input-signal transition rate, input capacitance, and output loading. Multistage devices such as noninverting gates or more complex logic functions may require several internal transistors to change state before the output can change state. And even when the output begins to change state, with nonzero rise and fall times it takes some time to reach the threshold where additional delay is attributed instead to rise or fall time, as we discussed in the preceding subsection. All of these factors are included in propagation delay.

To factor out the effect of rise and fall times, manufacturers usually specify propagation delays at the midpoints of input and output transitions, as shown in Figure 14-38(b). However, sometimes the delays are specified at the logic-level boundary points, especially if the device's operation may be adversely affected by slow rise and fall times. For example, Figure 14-39 shows how the minimum input pulse width for an S-R latch (discussed in Section 10.2.1) might be specified.

In addition, a manufacturer may specify absolute maximum input rise and fall times that must be satisfied to guarantee proper operation. High-speed CMOS circuits may consume excessive current or even oscillate if their input transitions are too slow.

14.4.3 Power Consumption

static power dissipation

quiescent power dissipation

dynamic power dissipation

The power consumption of a CMOS circuit whose output is not changing is called *static power dissipation* or *quiescent power dissipation.* Most CMOS circuits have very low static power dissipation. This is what makes them so attractive for smartphones, watches, and other low-power applications—when computation pauses, very little power is consumed. A CMOS circuit consumes significant power only during signal transitions; this is called *dynamic power dissipation.*

One source of dynamic power dissipation is the partial short-circuiting of the CMOS output structure during transitions. When the input voltage is not close to one of the power supply rails (0 V or V_{CC}), both the p-channel and n-channel output transistors may be partially "on," creating a series resistance of 600 Ω or less. In this case, current flows through the transistors from V_{CC} to ground. The amount of power consumed in this way depends on both the value of V_{CC} and the rate at which output transitions occur, according to the formula:

$$P_T = C_{PD} \cdot V_{CC}^2 \cdot f$$

The following variables are used in the formula:

P_T The circuit's internal power dissipation due to output transitions.

power-dissipation capacitance

C_{PD} The *power-dissipation capacitance*. This constant is normally specified by the device manufacturer. C_{PD} turns out to have units of capacitance, but does not represent an actual output capacitance. Rather, it embodies the dynamics of current flow through the changing output-transistor resistances during a single pair of output transitions, HIGH-to-LOW and LOW-to-HIGH. For example, C_{PD} for HC-series CMOS gates might be 20–24 pF, even though the actual output capacitance is much less.

V_{CC} The power-supply voltage. As all electrical engineers know, power dissipation across a resistive load (the partially-on transistors) is proportional to the *square* of the voltage.

transition frequency

f The *transition frequency* of the output signal. This implies the number of power-consuming output transitions per second. (But note that the number of transitions per second is the transition frequency times 2.)

The P_T formula is valid only if input transitions are fast enough, leading to fast output transitions. If the input transitions are too slow, then the output transistors stay partially on for a longer time, and power consumption increases. Device manufacturers usually recommend a maximum input rise and fall time, below which the value specified for C_{PD} is valid.

C_L

P_L

A second (and often more significant) source of CMOS power consumption is the capacitive load (C_L) on the output. During a LOW-to-HIGH transition, current flows through a *p*-channel transistor to charge C_L. Likewise, during a HIGH-to-LOW transition, current flows through an *n*-channel transistor to discharge C_L. In each case, power is dissipated in the "on" resistance of the transistor. We'll use P_L to denote the total amount of power dissipated by charging and discharging C_L.

The units of P_L are power, or energy usage per unit time. The energy for one transition could be determined by calculating the current through the charging transistor as a function of time (using the *RC* time constant as in

CONSUMPTION VS. DISSIPATION

The words *consumption* and *dissipation* are used pretty much interchangeably when discussing how much power a device uses. To be precise, however, dissipation includes only the power that is used in the device itself, generating heat in the device. *Consumption* includes additional power that the device consumes from the power supply and delivers to other devices connected to it (like resistive loads).

Section 14.4.1), squaring this function, multiplying by the "on" resistance of the charging transistor, and integrating over time. An easier way is described below.

During a transition, the voltage across the C_{L} changes by $\pm V_{\mathrm{CC}}$. According to the definition of capacitance, the total amount of charge that must flow to make a voltage change of V_{CC} across C_{L} is $C_{\mathrm{L}} \cdot V_{\mathrm{CC}}$. The total amount of energy used in one transition is charge times the average voltage change. The first little bit of charge makes a voltage change of V_{CC}, while the last bit of charge makes a vanishingly small voltage change; hence the average change is $V_{\mathrm{CC}}/2$. The total energy per transition is therefore $C_{\mathrm{L}} \cdot V_{\mathrm{CC}}^2/2$. If there are $2f$ transitions per second, the total power dissipated due to the capacitive load is

$$\begin{aligned} P_{\mathrm{L}} &= C_{\mathrm{L}} \cdot (V_{\mathrm{CC}}^2/2) \cdot 2f \\ &= C_{\mathrm{L}} \cdot V_{\mathrm{CC}}^2 \cdot f \end{aligned}$$

The total dynamic power dissipation of a CMOS circuit is the sum of P_{T} and P_{L}:

$$\begin{aligned} P_{\mathrm{D}} &= P_{\mathrm{T}} + P_{\mathrm{L}} \\ &= C_{\mathrm{PD}} \cdot V_{\mathrm{CC}}^2 \cdot f + C_{\mathrm{L}} \cdot V_{\mathrm{CC}}^2 \cdot f \\ &= (C_{\mathrm{PD}} + C_{\mathrm{L}}) \cdot V_{\mathrm{CC}}^2 \cdot f \end{aligned}$$

Based on this formula, dynamic power dissipation is often called *CV^2f power*. In most applications of CMOS circuits, CV^2f power is by far the major contributor to total power dissipation. Note that CV^2f power is also consumed by bipolar logic circuits like TTL, but at low to moderate frequencies, it is insignificant compared to the static (DC or quiescent) power dissipation of bipolar circuits.

CV^2f power

*14.4.4 Current Spikes and Decoupling Capacitors

The topics in the next three subsections are particularly important for board-level system design. When a CMOS output switches between LOW and HIGH, current flows from V_{CC} to ground through the partially-on p- and n-channel transistors. These currents, often called *current spikes* because of their brief duration, may show up as noise on the power-supply and ground connections in a CMOS circuit, especially when multiple outputs are switched simultaneously.

current spikes

For this reason, systems that use CMOS circuits require *decoupling capacitors* between V_{CC} and ground. These capacitors must be distributed throughout the printed-circuit board, at least one within an inch or so of each chip, to supply current during transitions. The large *filtering capacitors* typically found in the power supply itself don't satisfy this requirement, because stray wiring inductance prevents them from supplying the current fast enough, hence the need for a *physically distributed* system of decoupling capacitors.

decoupling capacitors

filtering capacitors

*Throughout this book, optional sections are marked with an asterisk.

*14.4.5 Inductive Effects

stray inductance

Digital logic circuits rarely contain any discrete inductors but, just like stray capacitance, *stray inductance* arises in circuit wiring, even in straight wires. (Electrical engineers know that discrete inductors are usually formed by a *coil* of wire.)

When the amount of current flowing through an inductor changes, a voltage is developed across that inductor according to the formula

$$V = L \cdot \frac{dI}{dt}$$

henries
nanohenries

where L is the inductance in henries and dI/dt is the current's rate of change in amperes per second. Stray inductance can be on the order of 10 nanohenries (nH, 10^{-9} H) per inch of wire on a printed circuit board.

With such tiny stray inductances, it may not seem possible that significant voltages could be developed across them, and that inductive effects could be safely ignored. This was the case with most digital circuits until the late 1990s.

However, two factors have combined to make inductance a significant factor and sometimes an obstacle in high-speed CMOS design, especially at the printed-circuit-board level. First, the output transistors in modern CMOS circuits are able to switch on or off in extremely short times—on the order of tens of picoseconds or less in the fastest circuits. Changing so quickly from a no-current condition to one in which even just a few milliamperes of current is flowing results in a *rate of change* (dI/dt) that is very high. Second, CMOS circuits' power-supply voltage (V_{CC}) has been steadily declining from 5 V to 1.2 V or less in the densest ASICs. This has resulted in smaller noise margins between logic levels, exacerbating the error-inducing effects of any voltage disturbances.

Under reasonable assumptions (see References), the maximum value of dI/dt when driving a resistive load can be approximated by the formula

$$\left[\frac{dI}{dt}\right]_{\text{Max-resistor}} = \frac{\Delta V}{T_t} \cdot \frac{1}{R}$$

where R is the load resistance, and ΔV is the voltage change and T_t is the rise or fall time for the transition. So, let's consider the voltage that could be developed across a 1-inch PCB trace driving a 2-KΩ load, for a couple of different CMOS logic families. A 5-V 74HC output can have transition times as low as 5 ns. Based on the preceding formula,

$$\left[\frac{dI}{dt}\right]_{\text{Max-resistor}} = \frac{5\ \text{V}}{5\ \text{ns}} \cdot \frac{1}{2000\ \Omega} = 5 \cdot 10^{5}\ \text{A/s}$$

Wow, 500,000 amps per second! Of course, the current doesn't continue to ramp up or down for anywhere near a second, but the rate of change during the 5-ns output transition really is that high. Now, we can plug that number into the voltage formula to see how much voltage is developed across our 1-inch, 10-nH PCB trace:

$$V = 10 \cdot 10^{-9} \cdot 5 \cdot 10^{5} = 5 \text{ mV}$$

When all's said and done, the voltage change across the PCB trace is only 5 mV (plus or minus, depending on the direction of current change). This is nothing to worry about in a logic family that has 1.35 V of DC noise margin in either state.

Now let's consider the case for a heftier CMOS family, 74AC, which can source or sink six times as much current as 74HC, and it can do so with transition times as short as 1 ns. The maximum current-change rate for 74AC driving a 1-KΩ load is

$$\left[\frac{dI}{dt}\right]_{\text{Max-resistor}} = \frac{5 \text{ V}}{1 \text{ ns}} \cdot \frac{1}{1000 \ \Omega} = 5 \cdot 10^{6} \text{ A/s}$$

or 10 times higher than the previous case. So the voltage change across a 1-inch, 10-nH PCB trace is also 10 times higher, or 50 mV. That's still not quite enough to worry about, but wait, there's more!

So far we've considered only resistive loads. As discussed in Section 14.4.1, gate inputs and wiring have stray capacitance, and current must flow to charge or discharge this capacitance. Under reasonable assumptions (once again, see References), the maximum value of dI/dt when driving a capacitive load C can be approximated by the formula

$$\left[\frac{dI}{dt}\right]_{\text{Max-capacstor}} = \frac{1.52\Delta V}{T_t^2} \cdot C$$

HAND WAVING In this subsection, we assumed some transition times for 74HC and 74AC outputs driving certain resistive and capacitive loads. Where did these numbers come from? Minimum transition times, especially as a function of loading, are seldom if ever specified in manufacturers' data sheets. Actually, these numbers came from the author's experience in the lab.

You might also be wondering, what if we had a 10-inch instead of a 1-inch PCB trace, and a 500-pF load instead of a 50-pF load? Could there be 15.2 V across that trace during a transition? No, of course not. Experience shows that the transition time would be much longer, and *dI/dt* would be much less. How can that be? The answer is that the actual electrical model of the circuit output, the PCB trace, and the load is much more complicated than we've shown, with each element having resistive, capacitive, and inductive components.

The approximations in this subsection are just to give you a rough feel for inductive effects. A more detailed study, typically using a circuit analysis tool like SPICE, is needed to predict the dynamic effects of output transitions accurately. Most IC manufacturers provide SPICE models (or equivalent) for their high-speed output circuits to aid electrical engineers who must analyze such dynamic effects.

On a good day, our 74AC output can drive a 50-pF load at the end of a 1-inch PCB trace and deliver a transition time of about 5 ns. Based on the preceding formula,

$$\left[\frac{dI}{dt}\right]_{\text{Max-capacstor}} = \frac{1.52 \cdot 5\text{ V}}{(25 \cdot 10^{-18})\text{ s}^2} \cdot 50 \cdot 10^{-12}\text{ F} = 1.52 \cdot 10^{7}\text{ A/s}$$

Plugging that into the voltage formula, the voltage developed across our 1-inch, 10-nH PCB trace is

$$V = 10 \cdot 10^{-9} \cdot 1.52 \cdot 10^{7} = 152\text{ mV}$$

Although this case eats into the noise margin even more than the last example, it's probably not enough to cause an incorrect logic value to be produced. The real problem occurs when the inductive effects of several changing outputs are concentrated on a single wire, as discussed in the next subsection.

*14.4.6 Simultaneous Switching and Ground Bounce

The current that flows through a gate's output pin has to come from or go to somewhere—from the device's V_{CC} pin when an output is sourcing current, and to the ground pin when it's sinking current. Now let's consider what happens when multiple gates use the same ground pin.

Figure 14-40 shows the situation when eight inverters are fabricated on a single chip with one ground and one V_{CC} pin. The connection from the chip's internal ground has stray inductance due to the chip and package substrates, the bonding wire between the chip and its package, and the wiring between the package and the PCB's ground plane. In the figure, this is shown as a lumped inductance L between the chip's ground pin and the actual ground on the PCB. The amount of stray inductance varies greatly with different packaging technologies, but in a 20-pin plastic DIP package with the ground pin in the corner, L is on the order of 10 nH.

simultaneous switching

Now consider the situation when all eight inputs are LOW, so all eight outputs are HIGH, and all eight inputs are simultaneously changed to HIGH. This kind of event is often called *simultaneous switching*. At that moment, all of the outputs change to LOW, and the single ground pin must sink the current from all eight loads. Assuming these are 74AC outputs each driving a 50 pF load as in the previous subsection, maximum value of dI/dt for each output is $1.52 \cdot 10^{7}$ A/s. With simultaneously switching outputs, the current change across the stray inductance L will be eight times this amount, and the voltage drop across L will be

$$V_{\text{GND}} = L \cdot 8 \cdot \frac{dI}{dt} = 10 \cdot 10^{-9} \cdot 8 \cdot 1.52 \cdot 10^{7}\text{ A/s} = 1.216\text{ V}$$

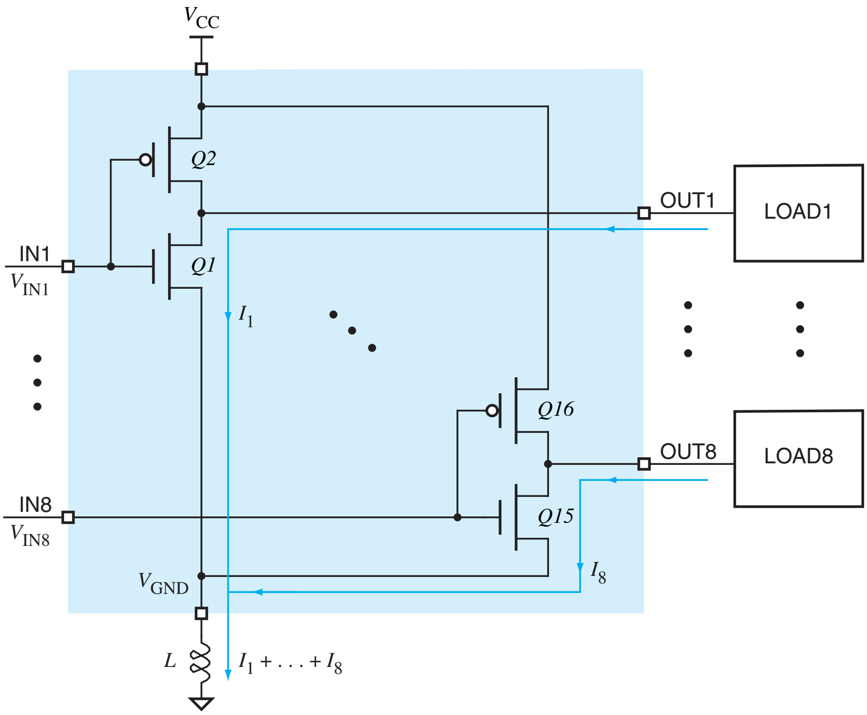


Figure 14-40
Ground bounce in an IC with eight inverters and one ground pin.

ground bounce

This change in the chip's internal ground voltage compared to the PCB and system ground is called *ground bounce*, and its effects can be significant. A chip has many inputs and outputs, and at any given time some of them may be changing while others are supposed to remain static. But consider the effects of a ground-bounce event on outputs that are supposed to remain static. Since LOW output voltages are referenced to a chip's internal ground (through an ON *n*-channel transistor), any increase in V_{GND} will also increase the LOW output voltages, possibly raising them above the valid LOW range and causing misbehavior elsewhere.

Ground bounce on a chip can also affect *inputs* on the same chip. A valid CMOS HIGH input voltage could be as low as 3.15 V. Keep in mind that this voltage is referenced to the chip's internal ground. Suppose a chip input receives a static, valid HIGH signal of 3.2 V from another chip. But then a ground-bounce event temporarily raises the chip's internal ground V_{GND} to 1.2 V. As far as the chip input is concerned, it now sees only 2.0 V with respect to its internal ground, and this is well into the "undefined" region for logic inputs. In fact, a slightly larger event could cause the apparent input voltage to drop well into the valid range for LOW signals. Thus, the ground bounce created by simultaneously switching outputs can change the logic value seen on totally unrelated inputs, as long as they are all referenced to the same ground pin.

A certain amount of ground bounce is inevitable in high-speed CMOS circuit design, but there are several ways that chip and system designers can reduce it enough to mask it safely within the noise margins of the circuit:

- Create or use a logic family whose output circuits are explicitly designed to have slower transition times, like 74FCT versus 74AC/ACT.
- Place the ground pins on the IC package so that the lead lengths to the chip will be shorter and hence inductance will be lower. For example, newer high-speed circuits packaged in DIPs now have V_{CC} and ground pins in the middle of each row of pins instead of on the corners.
- Use an IC package with lower inductance, like the square PLCC form factor versus a long rectangular DIP.
- Use multiple ground pins to split the current demand across multiple paths and thereby reduce the voltage drop across any one path. This is one reason that high-pin-count ICs are designed with lots and lots of ground pins.

At this point, you might be wondering, what about "V_{CC} bounce"? After all, V_{CC} wiring paths have stray inductance similar to ground paths, and they suffer voltage drops when multiple outputs switch from LOW to HIGH. However, logic levels are referenced to ground, not V_{CC}, and CMOS inputs are more sensitive to an input's voltage relative to ground than to V_{CC}. Thus, "V_{CC} bounce" is seldom a problem. Still, most high-pin-count ICs are designed with lots of V_{CC} pins to handle dynamic as well as static current demands with little voltage drop. A typical VLSI chip has at least half as many V_{CC} pins as ground pins and, quite often, just as many.

14.5 Other CMOS Input and Output Structures

Circuit designers have modified the basic CMOS structure in many ways to produce gates that are tailored for specific applications. This section describes some of the more common variations in CMOS input and output structures.

14.5.1 Transmission Gates

transmission gate

A *p*-channel and *n*-channel transistor pair can be connected together to form a logic-controlled switch. Shown in Figure 14-41, this circuit is called a CMOS *transmission gate*.

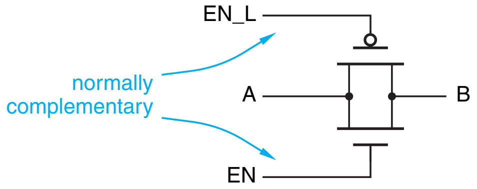


Figure 14-41
CMOS transmission gate.

A transmission gate is operated so that its input signals EN and EN_L are always at opposite levels. When EN is HIGH and EN_L is LOW, there is a low-impedance connection (as low as 1–5 Ω) between points A and B. When EN is LOW and EN_L is HIGH, points A and B are disconnected.

Once a transmission gate is enabled, the propagation delay from A to B (or vice versa) is very short. Because of their short delays and conceptual simplicity, transmission gates are often used internally in larger-scale CMOS devices like multiplexers and flip-flops. For example, Figure 14-42 shows how transmission gates can be used to create a 2-input multiplexer. When S is LOW, the X "input" is connected to the Z "output"; when S is HIGH, Y is connected to Z.

Note that unlike a gate-based multiplexer (e.g., Figure 6-32 on page 287), a multiplexer that uses transmission gates is "two-way." The transmission gate is a switch, and a signal on Z can drive an input on X or Y or vice versa.

At least one manufacturer (Integrated Device Technology) makes a variety of logic functions based on transmission gates. In their multiplexer devices, it takes several nanoseconds for a change in the "select" inputs (such as in Figure 14-42) to affect the input-output path (X or Y to Z). Once a path is set up, however, the maximum propagation delay from input to output may be as little as 0.15 ns; this is the fastest discrete CMOS multiplexer you can buy.

The *p*-channel (top) transistor in Figure 14-41 has a low impedance when its gate (EN_L) is LOW. The *n*-channel transistor has a low impedance when EN is HIGH. Two transistors are used because a typical "on" *p*-channel transistor can't conduct a LOW voltage between points A and B very well, and a typical "on" *n*-channel transistor can't conduct a HIGH voltage very well, but the parallel transistors cover the entire voltage range just fine. Some manufacturers, such as IDT, have improved their *n*-channel transistors enough to omit the *p*-channel transistor. Besides saving a transistor, this approach also eliminates a parasitic diode to V_{CC} that would otherwise result from the chip's physical structure.

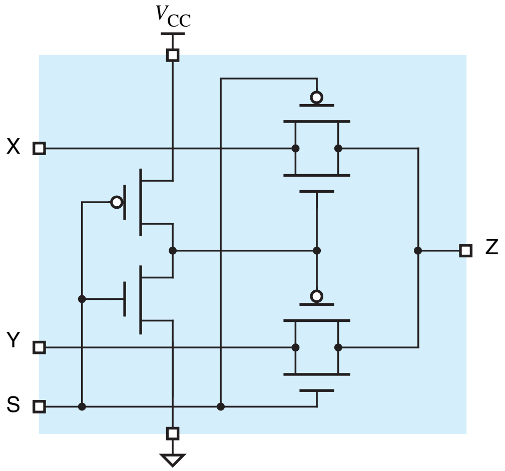


Figure 14-42
Two-input multiplexer using CMOS transmission gates.

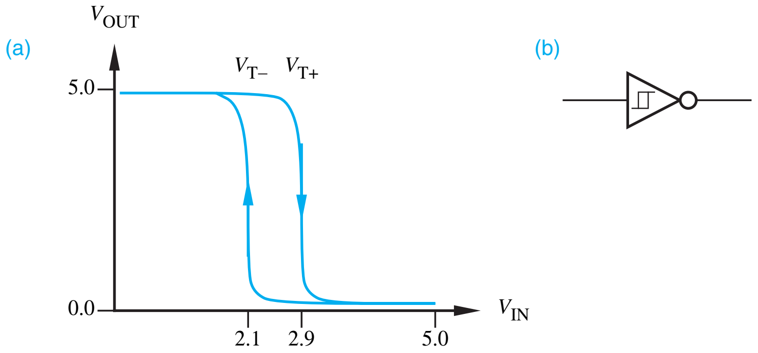


Figure 14-43
A Schmitt-trigger inverter: (a) input-output transfer characteristic; (b) logic symbol.

14.5.2 Schmitt-Trigger Inputs

Schmitt-trigger input

The input-output transfer characteristic for a typical CMOS gate was shown in Figure 14-21 on page 751. The corresponding transfer characteristic for a gate with *Schmitt-trigger inputs* is shown in Figure 14-43(a). A Schmitt trigger is a special circuit that uses feedback internally to shift the switching threshold depending on whether the input is changing from LOW to HIGH or from HIGH to LOW.

hysteresis

For example, suppose the input of a Schmitt-trigger inverter is initially at 0 V, a solid LOW. Then the output is HIGH, close to 5.0 V. If the input voltage is increased, the output will not go LOW until the input voltage reaches about 2.9 V. However, once the output is LOW, it won't go HIGH again until the input decreases to about 2.1 V. Thus, the switching threshold for positive-going input changes, denoted by V_{T+}, is about 2.9 V, and for negative-going input changes, denoted by V_{T-}, is about 2.1 V. The difference between the two thresholds is called *hysteresis*; the Schmitt-trigger inverter provides about 0.8 V of hysteresis.

To demonstrate the usefulness of hysteresis, Figure 14-44(a) shows an input signal with long rise and fall times and about 0.5 V of noise on it. An ordinary inverter, without hysteresis, has the same switching threshold for both positive-going and negative-going transitions, $V_T \approx 2.5$ V. Thus, the ordinary inverter responds to the noise as shown in (b), producing multiple output changes as the noisy input voltage crosses the switching threshold multiple times. However, as shown in (c), a Schmitt-trigger inverter does not respond to the noise, because its hysteresis is greater than the noise amplitude.

FIXING YOUR TRANSMISSION

Schmitt-trigger inputs have better noise immunity than ordinary gate inputs for signals with transmission-line reflections or long rise and fall times. Such signals typically occur in physically long connections, such as input-output buses and computer interface cables. Noise immunity is important in these applications, since long signal lines are more likely to have reflections or to pick up noise from adjacent signal lines, circuits, and appliances.

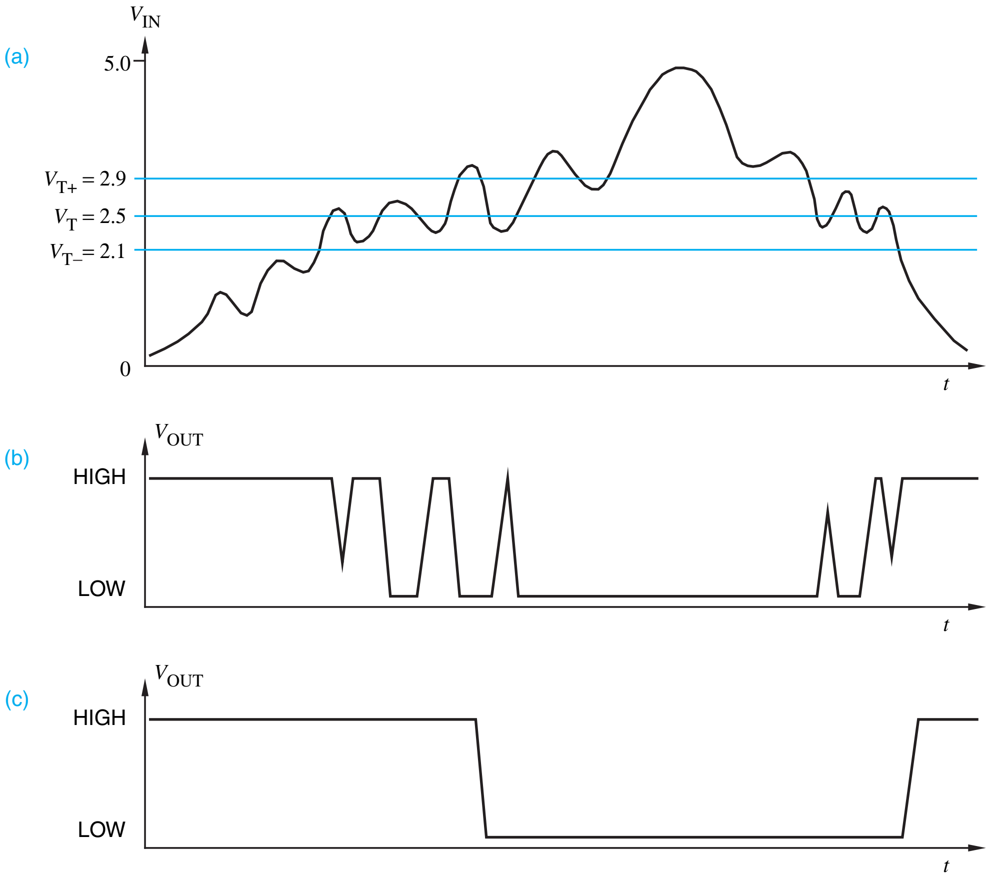


Figure 14-44 Device operation with slowly changing inputs: (a) a noisy, slowly changing input; (b) output produced by an ordinary inverter; (c) output produced by an inverter with 0.8 V of hysteresis.

14.5.3 Three-State Outputs

Logic outputs have two normal states, LOW and HIGH, corresponding to logic values 0 and 1. However, some outputs have a third electrical state that is not a logic state at all, called the *high-impedance, Hi-Z,* or *floating state.* In this state, the output behaves as if it isn't even connected to the circuit, except for a small leakage current that may flow into or out of the output pin. Thus, an output can have one of three states—logic 0, logic 1, and Hi-Z.

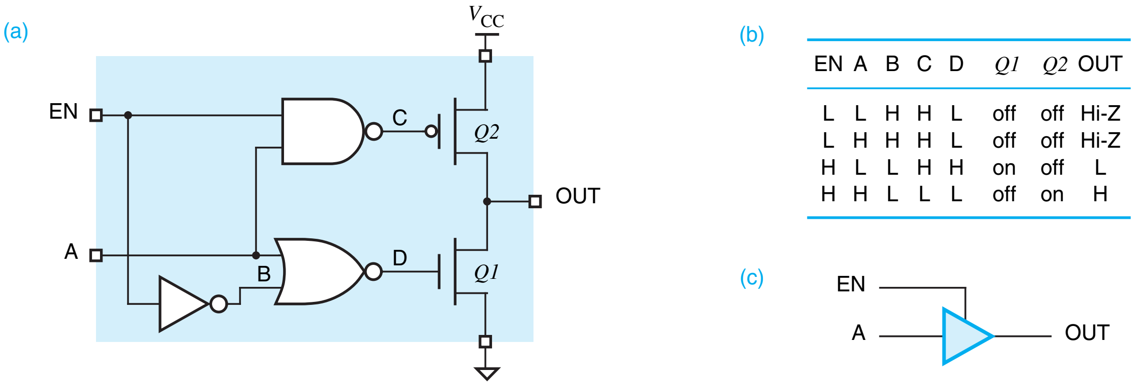
high-impedance state
Hi-Z state
floating state

An output with three possible states is called (surprise!) a *three-state output* or, sometimes, a *tri-state output*. Three-state devices have an extra input, usually called "output enable" or "output disable," for placing the device's output(s) in the high-impedance state.

three-state output
tri-state output

A *three-state bus* is created by wiring several three-state outputs together. Control circuitry for the "output enables" must ensure that at most one output is

three-state bus



EN	A	B	C	D	$Q1$	$Q2$	OUT
L	L	H	H	L	off	off	Hi-Z
L	H	H	H	L	off	off	Hi-Z
H	L	L	H	H	on	off	L
H	H	L	L	L	off	on	H

Figure 14-45 CMOS three-state buffer: (a) circuit diagram; (b) function table; (c) logic symbol.

enabled (not in its Hi-Z state) at any time. The single enabled device can transmit logic levels (HIGH and LOW) on the bus. We discussed this and other examples of three-state applications in Section 7.1.

three-state buffer

A simplified circuit diagram for a CMOS *three-state buffer* is shown in Figure 14-45(a). The internal NAND, NOR, and inverter functions are shown in functional rather than transistor form; they actually use a total of 10 more transistors (see Exercise 14.82). As shown in the function table (b), when the enable (EN) input is LOW, both output transistors are off, and the output is in the Hi-Z state. Otherwise, the output is HIGH or LOW as controlled by the "data" input A. Logic symbols for three-state buffers and gates are normally drawn with the enable input coming into the top, as shown in (c).

In practice, the three-state control circuit may be different from what we have shown, in order to provide proper dynamic behavior of the output transistors during transitions to and from the Hi-Z state. In particular, devices with three-state outputs are normally designed so that the output-enable delay (Hi-Z to LOW or HIGH) is somewhat longer than the output-disable delay (LOW or HIGH to Hi-Z). Thus, if a control circuit activates one device's output-enable input and simultaneously deactivates a second's, the second device should enter the Hi-Z state before the first places a HIGH or LOW level on the bus (though a prudent designer would provide enable signals with a much larger window of nonoverlap to guarantee this under all conditions).

If two three-state outputs on the same bus are enabled at the same time and try to maintain opposite states, the situation is similar to tying standard active-

LEGAL NOTICE The name "TRI-STATE" is a trademark of the National Semiconductor Corporation, which was acquired by Texas Instruments in 2011. Their lawyers thought you'd like to know.

pull-up outputs together as in Figure 14-53 on page 788—a nonlogic voltage is produced on the bus. If fighting is only momentary, the devices probably will not be damaged, but the large current drain through the tied outputs can produce noise pulses that affect circuit behavior elsewhere in the system.

There is a leakage current of up to 10 μA associated with a CMOS three-state output in its Hi-Z state. This current, as well as the input currents of receiving gates, must be taken into account when calculating the maximum number of devices that can be placed on a three-state bus. That is, in the LOW or HIGH state, an enabled three-state output must be capable of sinking or sourcing up to 10 μA of leakage current for every other three-state output on the bus, as well as handling the current required by every input on the bus. As with standard CMOS logic, separate LOW-state and HIGH-state calculations must be made to ensure that the fanout requirements of a particular circuit configuration are met.

*14.5.4 Open-Drain Outputs

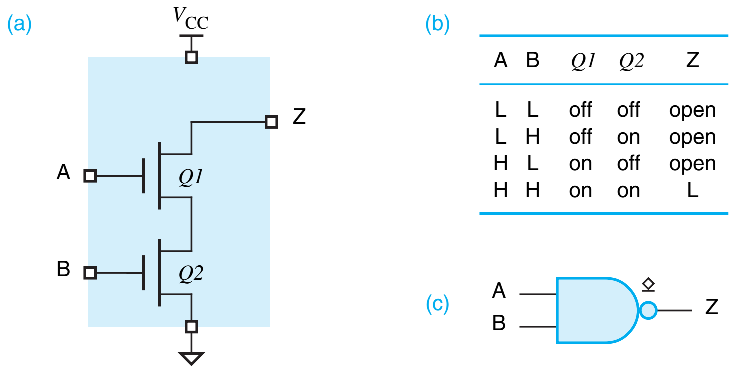
The *p*-channel transistors in CMOS output structures are said to provide *active pull-up*, since they actively pull up the output voltage on a LOW-to-HIGH transition. These transistors are omitted in gates with *open-drain outputs*, like the NAND gate in Figure 14-46(a). The drain of the topmost *n*-channel transistor is left unconnected internally, so if the output is not LOW it is “open,” as indicated in (b). The underscored diamond in the symbol in (c) is sometimes used to indicate an open-drain output. A similar structure, called an “open-collector output,” is provided in the legacy TTL logic families.

active pull-up
open-drain output

An open-drain output requires an external *pull-up resistor* to provide *passive pull-up* to the HIGH level. For example, Figure 14-47 shows an open-drain CMOS NAND gate, with its pull-up resistor, driving a load.

pull-up resistor
passive pull-up

For the highest possible speed, an open-drain output’s pull-up resistor should be as small as possible; this minimizes the *RC* time constant for LOW-to-HIGH transitions (rise time). However, the pull-up resistance cannot be arbitrarily small; the minimum resistance is determined by the open-drain output’s maximum sink current, I_{OLmax}. For example, in HC- and HCT-series



A	B	*Q1*	*Q2*	Z
L	L	off	off	open
L	H	off	on	open
H	L	on	off	open
H	H	on	on	L

Figure 14-46
Open-drain CMOS NAND gate:
(a) circuit diagram;
(b) function table;
(c) logic symbol.

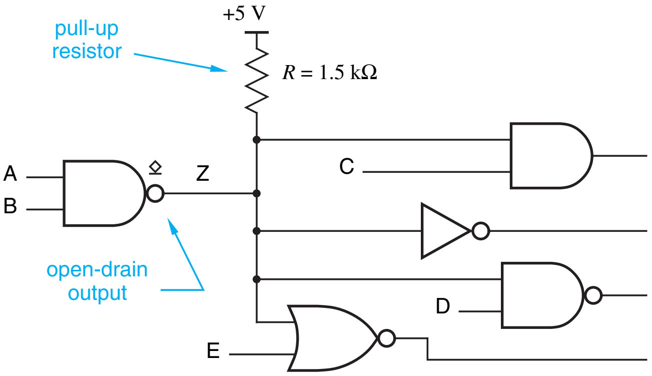


Figure 14-47 Open-drain CMOS NAND gate driving a load.

CMOS, I_{OLmax} is 4 mA, and the pull-up resistor can be no less than 5.0 V/4 mA, or 1.25 kΩ. Since this is an order of magnitude greater than the "on" resistance of the *p*-channel transistors in a standard CMOS gate, the LOW-to-HIGH output transitions are much slower for an open-drain gate than for standard gate with active pull-up.

As an example, let us assume that the open-drain gate in Figure 14-47 is HC-series CMOS, the pull-up resistance is 1.5 kΩ, and the load capacitance is 100 pF. We showed in Section 14.3.2 that the "on" resistance of an HC-series CMOS output in the LOW state is about 80 Ω. Thus, the *RC* time constant for a HIGH-to-LOW transition is about 80 Ω · 100 pF = 8 ns, and the output's fall time is about 8 ns. However, the *RC* time constant for a LOW-to-HIGH transition is about 1.5 kΩ · 100 pF = 150 ns, and the rise time is about 150 ns. This relatively slow rise time is contrasted with the much faster fall time in Figure 14-48. A friend of the author calls such slow rising transitions *ooze*.

ooze

So why use open-drain outputs? Despite slow rise times, they can be useful in three applications discussed next: driving light-emitting diodes (LEDs) and other devices; driving multisource buses; and, in a pinch, performing wired logic.

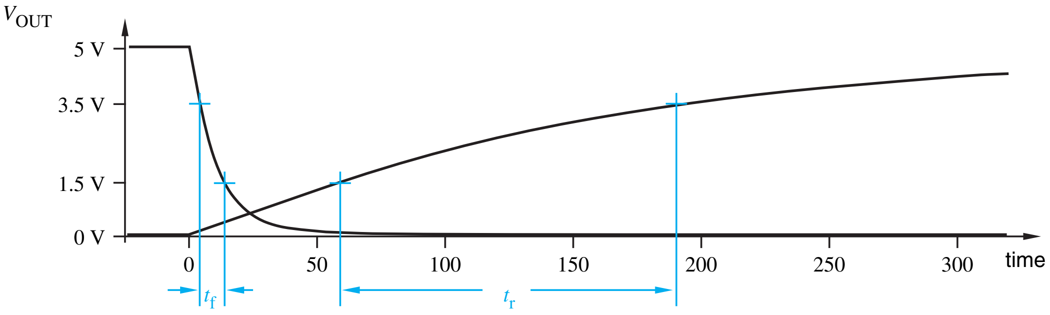


Figure 14-48 Rising and falling transitions of an open-drain CMOS output.

*14.5.5 Driving LEDs and Relays

An open-drain output can drive an LED or other device as shown in Figure 14-49. If either input A or B is LOW, the corresponding *n*-channel transistor is off and the LED is off. When A and B are both HIGH, both transistors are on, the output Z is LOW, and the LED is on. The value of the pull-up resistor *R* is chosen so that the proper amount of current flows through the LED in the “on” state. The same arrangement can be used to drive a relay coil, where the pull-up value *R* is chosen to yield enough current flow to activate the relay.

Typical LEDs require 10 mA for normal brightness. HC- and HCT-series CMOS outputs are only specified to sink or source 4 mA and are not normally used to drive LEDs. However, the outputs in advanced CMOS families such as 74ACT and 74ALVC can sink 24 mA or more and can be used quite effectively to drive LEDs and even relays requiring more current.

The following information is used to calculate the pull-up resistor’s value:

1. The LED current I_{LED} needed for the desired brightness, up to 10 mA for some discrete LEDs but often less.
2. The voltage drop V_{LED} across the LED in the “on” condition, about 1.6 V for typical LEDs.
3. The power-supply voltage for the LED, V_{CCL}, often the same as V_{CC}.
4. The output voltage V_{OL} of the open-drain output that sinks the LED current. In the 74AC and 74ACT CMOS families, V_{OLmax} is 0.37 V. If an output can sink I_{LED} and maintain a lower voltage, say 0.2 V, then the calculation below yields a resistor value that is a little too low, but normally with no harm done. A little more current than I_{LED} will flow and the LED will be just a little brighter than expected.

Using this information, we can write the following equation:

$$V_{\text{OL}} + V_{\text{LED}} + (I_{\text{LED}} \cdot R) = V_{\text{CCL}}$$

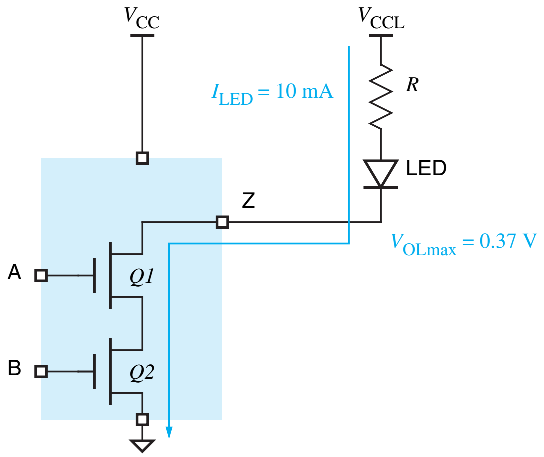


Figure 14-49
Driving an LED with an open-drain output.

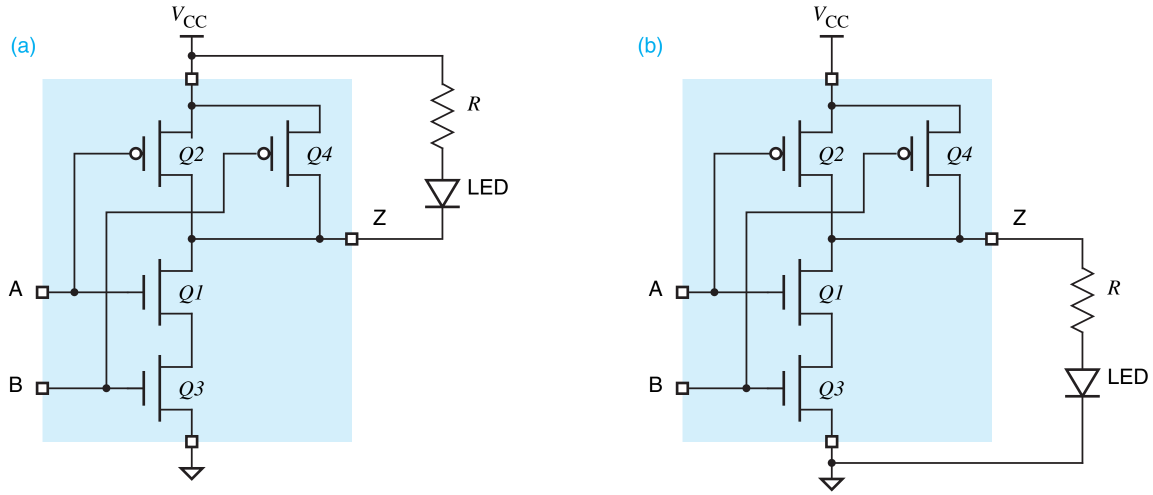


Figure 14-50 Driving an LED with an ordinary CMOS output: (a) sinking current, "on" in the LOW state; (b) sourcing current, "on" in the HIGH state.

Assuming $V_{CCL} = 5.0$ V and the other typical values above, we can solve for the required value of R:

$$
\begin{aligned}
R &= \frac{V_{CCL} - V_{OL} - V_{LED}}{I_{LED}} \\
&= (5.0 - 0.37 - 1.6)\text{V} / 10\text{ mA} = 303\ \Omega
\end{aligned}
$$

An advantage of the open-drain output is that the LED and CMOS driver needn't use the same V_{CC}. If they are the same, an LED can driven by an ordinary CMOS gate output with active pull-up as shown in Figure 14-50(a). If both inputs are HIGH, the bottom (*n*-channel) transistors pull the output LOW as in the open-drain version. If either input is LOW, the output is HIGH; although one or both of the top (*p*-channel) transistors is on, no current flows through the LED.

With some CMOS families, you can turn an LED "on" when the output is in the HIGH state, as shown in Figure 14-50(b). This is possible if the output can source enough current to satisfy the LED's requirements. However, method (b) isn't used as often as (a), because many CMOS and most TTL outputs cannot source as much current in the HIGH state as they can sink in the LOW state.

RESISTOR VALUES

In most applications, the precise value of LED series resistors is unimportant, as long as groups of nearby LEDs have similar drivers and resistors to give equal apparent brightness. In the example in this subsection, one might use an off-the-shelf resistor value of 270, 300, or 330 ohms, whatever is readily available.

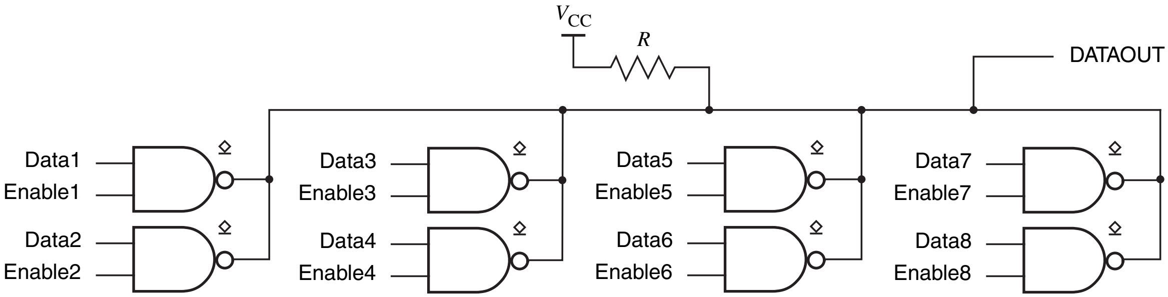


Figure 14-51 Eight open-drain outputs driving a bus.

*14.5.6 Multisource Buses

open-drain bus

Open-drain outputs can be tied together to allow several devices, one at a time, to put information on a common bus. At any time, all but one of the outputs are in their HIGH (open) state. The remaining output either stays in the HIGH state or pulls the bus LOW, depending on whether it wants to transmit a logical 1 or a logical 0 on the bus. Control circuitry selects the particular device that is allowed to drive the bus at any time. This method is used in the popular I^2C bus.

For example, in Figure 14-51, eight 2-input open-drain NAND-gate outputs drive a common bus. The top input of each NAND gate is a data bit, and the bottom input of each is a control bit. At most one control bit is HIGH at any time, enabling the corresponding data bit to be passed through to the bus. (Actually, the complement of the data bit is placed on the bus.) The other gate outputs are HIGH, that is, "open," so the data input of the enabled gate determines the value on the bus.

*14.5.7 Wired Logic

wired logic

wired AND

If the outputs of several open-drain gates are tied together with a single pull-up resistor, then *wired logic* is performed. (That's *wired*, not *weird*!) An AND function is obtained, since the wired output is HIGH if and only if all of the individual gate outputs are HIGH (actually, open); any output going LOW is sufficient to pull the wired output LOW. For example, a 3-input *wired AND* function is shown in Figure 14-52. If any of the individual 2-input NAND gates has both inputs HIGH, it pulls the wired output LOW; otherwise, the pull-up resistor *R* pulls the wired output HIGH.

fighting

Note that wired logic cannot be performed using gates with active pull-up. Two such outputs wired together and trying to maintain opposite logic values result in a very high current flow and an abnormal output voltage. Figure 14-53 shows this situation, which is sometimes called *fighting*. The exact output voltage depends on the relative "strengths" of the fighting transistors, but with 5-V CMOS devices it is typically about 1–2 V, almost always a nonlogic voltage. Worse, if outputs fight continuously for more than a few seconds, the chips can get hot enough to sustain internal damage *and* to burn your fingers!

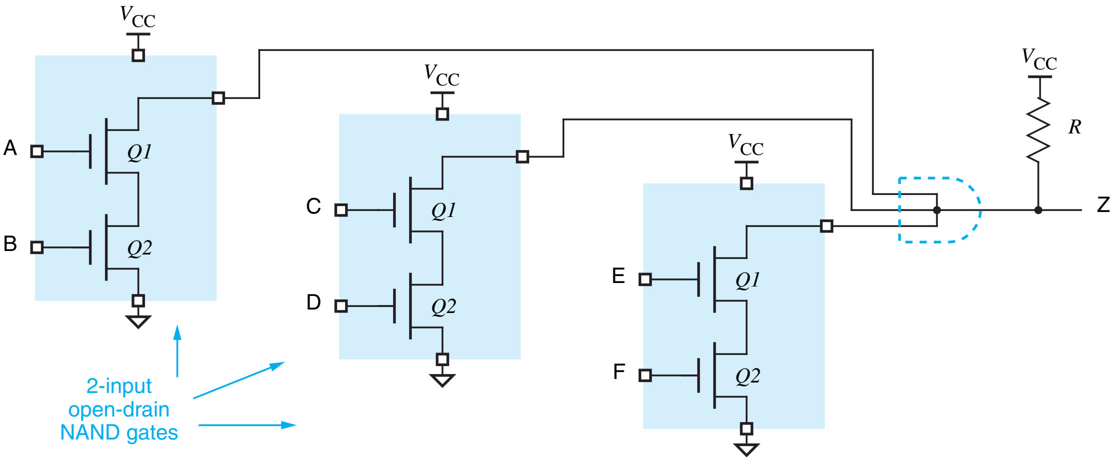


Figure 14-52 Wired-AND function on three open-drain NAND-gate outputs.

*14.5.8 Pull-Up Resistors

pull-up-resistor calculation

A proper choice of value for the pull-up resistor *R* must be made in open-drain applications. Two calculations are made to bracket the allowable values of *R*:

Maximum The voltage drop across *R* in the HIGH state must not reduce the output voltage below V_{IHmin} for driven gates plus any additional noise margin that's desired. This drop is produced by the HIGH-state output leakage current of the wired outputs and the HIGH-state input currents of the driven gates.

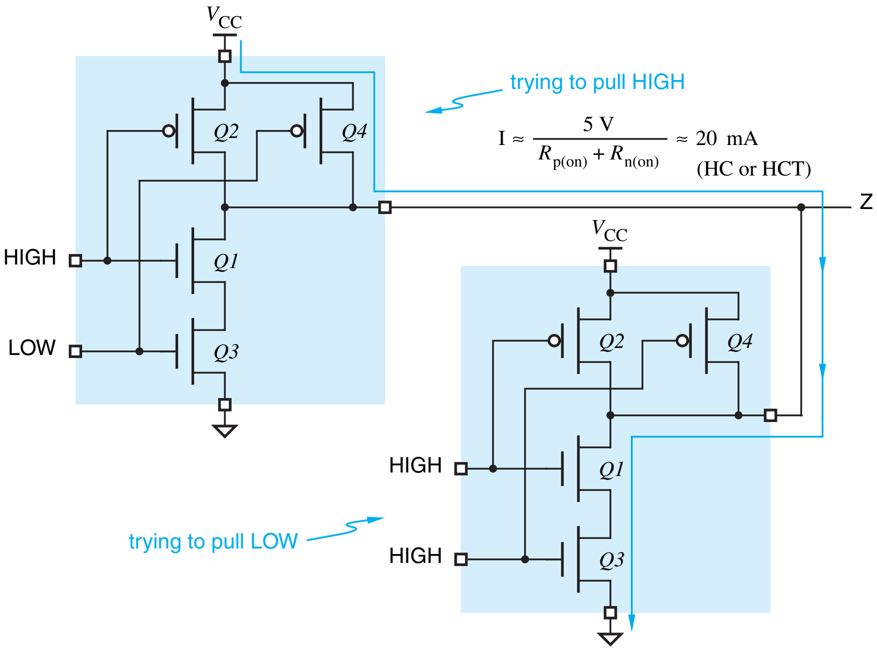


Figure 14-53 Two CMOS outputs trying to maintain opposite logic values on the same line.

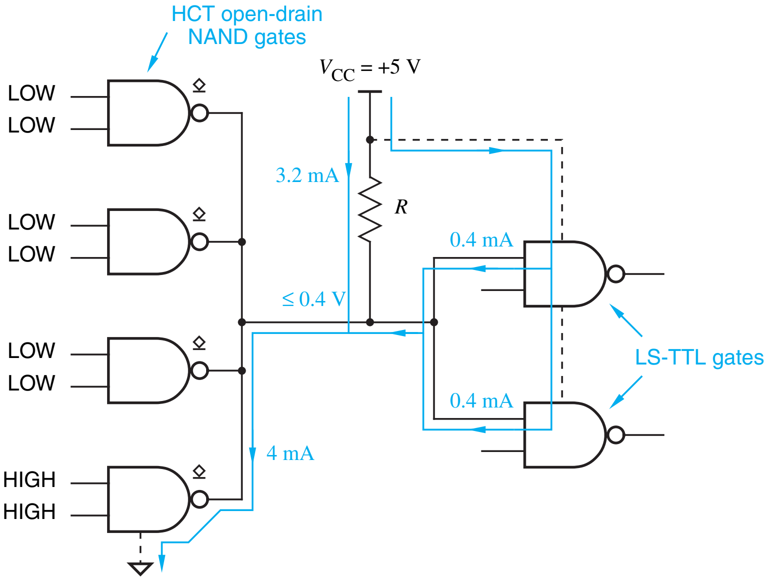


Figure 14-54
Four open-drain outputs driving two inputs in the LOW state.

Minimum The sum of the current through R in the LOW state and the LOW-state input currents of the gates driven by the wired outputs must not exceed the LOW-state driving capability of the active output; for example, 4 mA for HC/HCT and 8 mA for AHC/AHCT devices.

For example, suppose that four HCT open-drain outputs are wired together and drive two legacy LS-TTL inputs as shown in Figure 14-54. A LOW output must sink 0.4 mA from each LS-TTL input as well as sink the current through the pull-up resistor R. For the total current to stay within the HCT I_{OLmax} spec of 4 mA, the current through R may be no more than

$$I_{\mathrm{R(max)}} = 4 - (2 \cdot 0.4) = 3.2 \text{ mA}$$

Assuming that V_{OL} of the open-drain output is 0.0 V, the minimum value of R is

$$R_{\mathrm{min}} = (5.0 - 0.0)/I_{\mathrm{R(max)}} = 1562.5\ \Omega$$

OPEN-DRAIN ASSUMPTION

In our open-drain resistor calculations, we assume that the output voltage can be as low as 0.0 V rather than 0.4 V (V_{OLmax}) in order to obtain a worst-case result. That is, even if the open-drain output is so strong that it can pull the output voltage all the way down to 0.0 V (it's only required to pull down to 0.4 V), we'll never allow it to sink more than 4 mA, so it doesn't get overstressed. Some designers prefer to use 0.4 V in this calculation, figuring that if the output is so good that it can pull lower than 0.4 V, a little bit of excess sink current beyond 4 mA won't hurt it.

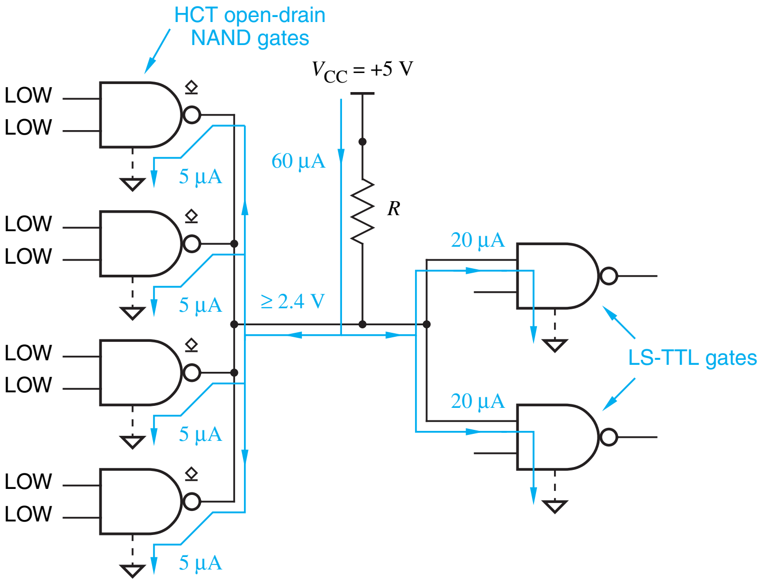


Figure 14-55
Four open-drain outputs driving two inputs in the HIGH state.

In the HIGH state, typical open-drain outputs have a maximum leakage current of 5 μA, and typical LS-TTL inputs require 20 μA of source current. Hence, the HIGH-state current requirement as shown in Figure 14-55 is

$$I_{\mathrm{R(leak)}} = (4 \cdot 5) + (2 \cdot 20) = 60\ \mu\mathrm{A}$$

This current produces a voltage drop across R, and must not lower the output voltage below LS-TTL's $V_{\mathrm{IHmin}} = 2.0$ V plus an additional, say, 400 mV of noise margin; thus the maximum value of R is

$$R_{\max} = (5.0 - 2.4) / I_{\mathrm{R(leak)}} = 43.3\ \mathrm{k}\Omega$$

Hence, any value of R between 1562.5 Ω and 43.3 kΩ may be used. Higher values reduce power consumption and improve the LOW-state noise margin, while lower values increase power consumption but improve both the HIGH-state noise margin and the speed of LOW-to-HIGH output transitions.

14.6 CMOS Logic Families

4000-series CMOS

The first commercially successful CMOS family was *4000-series CMOS*. Although 4000-series circuits offered the benefit of low power dissipation, they were fairly slow and were not easy to interface with the most popular logic family of the time, bipolar TTL. Thus, the 4000 series was supplanted in most applications by the more capable CMOS families discussed in this section.

All of the CMOS devices that we discuss have part numbers of the form "74FAM*nn*," where "FAM" is an alphabetic family mnemonic and *nn* is a

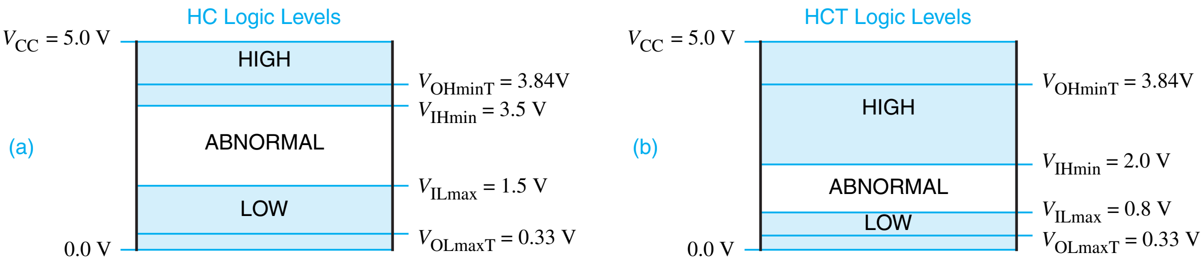


Figure 14-56 Input and output levels for CMOS devices using a 5-V supply: (a) HC; (b) HCT.

numeric function designator. Devices in different families with the same value of *nn* perform the same function. For example, the 74HC30, 74HCT30, 74AC30, 74ACT30, 74AHC30, and 74AHCT30 are all 8-input NAND gates.

The prefix “74” is simply a number that was used by an early, popular supplier of TTL devices, Texas Instruments, probably because it didn’t conflict with any other part-number prefixes. The prefix “54” is used for identical parts that are specified for operation over a wider range of temperature and power-supply voltage, for use in military applications. Such parts are usually fabricated in the same way as their 74-series counterparts, except that they are tested, screened, and marked differently, a lot of extra paperwork is generated, and a higher price is charged, of course.

14.6.1 HC and HCT

HC (High-speed CMOS)

HCT (High-speed CMOS, TTL compatible)

The first two 74-series CMOS families are *HC (High-speed CMOS)* and *HCT (High-speed CMOS, TTL compatible)*. Compared with the original 4000 family, HC and HCT both have higher speed and better current sinking and sourcing capability. The HCT family uses a power-supply voltage V_{CC} of 5 V and can be intermixed with TTL devices, which also use a 5-V supply.

The HC family is optimized for use in systems that use CMOS logic exclusively, which is typical of today’s systems, and can use any power-supply voltage between 2 and 6 V. A higher voltage is used for higher speed, and a lower voltage for lower power dissipation. Lowering the supply voltage is especially effective, since most CMOS power dissipation is proportional to the square of the voltage (CV^2f power).

Even when used with a 5-V supply, HC devices are not quite compatible with TTL, since HC circuits are designed to recognize CMOS input levels. Assuming a supply voltage of 5.0 V, Figure 14-56(a) shows the input and output levels of HC devices. The output levels produced by TTL devices do not quite match this range, so HCT devices use the different input levels shown in (b). These levels are obtained using transistors with different switching thresholds, yielding the different transfer characteristics shown in Figure 14-57. However, the output characteristics of HC and HCT devices are essentially identical.

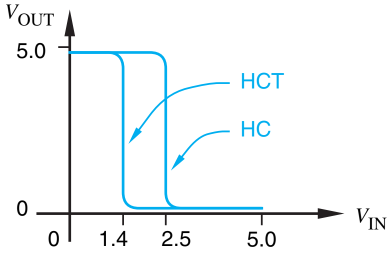


Figure 14-57
Transfer characteristics of HC and HCT circuits under typical conditions.

14.6.2 AHC and AHCT

AHC (Advanced High-speed CMOS)

AHCT (Advanced High-speed CMOS, TTL compatible)

Several new CMOS families were introduced in the 1980s and the 1990s. Two of the most recent and probably the most versatile are *AHC (Advanced High-speed CMOS)* and *AHCT (Advanced High-speed CMOS, TTL compatible)*. These families are two to three times as fast as HC/HCT while maintaining backward compatibility with their predecessors. Like HC and HCT, the AHC and AHCT families differ from each other only in the input levels that they recognize; their output characteristics are the same.

symmetric output drive

asymmetric output drive

Both AHC/AHCT and HC/HCT outputs have *symmetric output drive*. That is, an output can sink or source equal amounts of current; the output is just as "strong" in both states. Some other logic families, especially ones originally designed for TTL compatibility, have *asymmetric output drive;* they can sink much more current in the LOW state than they can source in the HIGH state.

*14.6.3 HC, HCT, AHC, and AHCT Electrical Characteristics

Electrical characteristics of the HC, HCT, AHC, and AHCT families are summarized in this subsection. The specifications assume that the devices are used with a nominal 5-V power supply, although (derated) operation is possible with any supply voltage in the range 2–5.5 V (up to 6 V for HC/HCT). We'll take a closer look at low-voltage and mixed-voltage operation in Section 14.7.

Commercial (74-series) parts are intended to be operated at temperatures between 0°C and 70°C, while military (54-series) parts are characterized for operation between –55°C and 125°C. The specs in Table 14-3 assume an operating temperature of 25°C. A full manufacturer's data sheet provides additional specifications for device operation over the entire temperature range.

Most devices within a given logic family have the same electrical specifications for inputs and outputs, typically differing only in power consumption

VERY = ADVANCED, SORT OF

The AHC and AHCT logic families are manufactured by several companies, including Texas Instruments and NXP (formerly Philips) Semiconductors. Compatible families with similar but not identical specifications are manufactured by STMicro, ON Semiconductor (formerly Fairchild), and Toshiba; they are called VHC and VHCT, where the "V" stands for "Very."

Table 14-3 Speed and power characteristics of selected CMOS families operating at 5 V.

				Family			
Description	***Part***	***Symbol***	***Condition***	***HC***	***HCT***	***AHC***	***AHCT***
Typical propagation delay (ns)	’00	t_{PD}		9	10	3.7	5
	’138			18	20	5.7	7.6
Quiescent power-supply current (μA)	’00	I_{CC}	$V_{in} = 0$ or V_{CC}	2.5	2.5	5.0	5.0
	’138		$V_{in} = 0$ or V_{CC}	40	40	40	40
Quiescent power dissipation (mW)	’00		$V_{in} = 0$ or V_{CC}	0.0125	0.0125	0.025	0.025
	’138		$V_{in} = 0$ or V_{CC}	0.2	0.2	0.2	0.2
Power-dissipation capacitance (pF)	’00	C_{PD}		22	15	2.4	2.6
	’138	C_{PD}		55	51	13	14
Dynamic power dissipation (mW/MHz)	’00			0.55	0.38	0.06	0.065
	’138			1.38	1.28	0.33	0.35
Total power dissipation (mW)	’00		$f = 100$ kHz	0.068	0.050	0.031	0.032
	’00		$f = 1$ MHz	0.56	0.39	0.085	0.09
	’00		$f = 10$ MHz	5.5	3.8	0.63	0.68
	’138		$f = 100$ kHz	0.338	0.328	0.23	0.24
	’138		$f = 1$ MHz	1.58	1.48	0.53	0.55
	’138		$f = 10$ MHz	14.0	13.0	3.45	3.7
Speed-power product (pJ)	’00		$f = 100$ kHz	0.61	0.50	0.11	0.16
	’00		$f = 1$ MHz	5.1	3.9	0.31	0.45
	’00		$f = 10$ MHz	50	38	2.3	3.38
	’138		$f = 100$ kHz	6.08	6.55	1.33	1.79
	’138		$f = 1$ MHz	28.4	29.5	2.99	4.2
	’138		$f = 10$ MHz	251	259	19.7	28.1

and propagation delay. Table 14-3 includes specifications for a 74x00 2-input NAND gate and a 74x138 3-to-8 decoder in the HC, HCT, AHC, and AHCT families. The ’00 NAND gate is included as the smallest logic-design building block in each family, while the ’138 is an MSI part containing the equivalent of about 15 NAND gates.

The first row of Table 14-3 specifies propagation delay. As discussed in Section 14.4.2, two numbers, t_{pHL} and t_{pLH}, may be used to specify delay; the number in the table is the worst case of the two. The propagation delay for the ’138 is somewhat longer than for the ’00, since signals must travel through three or four levels of gates internally.

The second and third rows of the table show that the quiescent power dissipation of these CMOS devices is practically nil, well under a milliwatt (mW) if the inputs have CMOS levels—0 V for LOW and V_{CC} for HIGH. (Note that in the table, the quiescent power dissipation numbers given for the ’00 are per gate, while for the ’138 they apply to the entire MSI device.)

As we discussed in Section 14.4.3, the dynamic power dissipation of a CMOS gate depends on the voltage swing of the output (usually V_{CC}), the output transition frequency (f), and the capacitance that is being charged and discharged on transitions, according to the formula

$$P_{\mathrm{D}} = (C_{\mathrm{L}} + C_{\mathrm{PD}}) \cdot V_{\mathrm{DD}}^2 \cdot f$$

Here, C_{PD} is the power-dissipation capacitance of the device and C_{L} is the capacitance of the load attached to the CMOS output in a given application. The table lists both C_{PD} and an equivalent dynamic power-dissipation factor in units of milliwatts per megahertz, assuming that $C_{\mathrm{L}} = 0$. Using this factor, the total power dissipation is computed at various frequencies as the sum of the dynamic power dissipation at that frequency and the quiescent power dissipation.

speed-power product

Shown next in the table, the *speed-power product* is simply the product of the propagation delay and power consumption of a typical gate; the result is measured in picojoules (pJ). Recall from physics that the joule is a unit of energy, so the speed-power product measures a sort of efficiency—how much energy a logic gate uses to switch its output. In this day and age, it’s obvious that the lower the energy usage, the better.

Table 14-4 gives the input specs of typical CMOS devices in each of the families. Some of the specs assume that the 5-V supply has a ±10% margin; that is, V_{CC} can be anywhere between 4.5 and 5.5 V. These parameters were discussed in previous sections, but for reference purposes, their meanings are summarized here:

I_{Imax} The maximum input current for any value of input voltage. This spec states that the current flowing into or out of a CMOS input is 1 μA or less for any value of input voltage. In other words, CMOS inputs create almost no DC load on the circuits that drive them.

Table 14-4 Input specifications for CMOS families with V_{CC} between 4.5 and 5.5 V.

			Family			
Description	***Symbol***	***Condition***	***HC***	***HCT***	***AHC***	***AHCT***
Input leakage current (μA)	I_{Imax}	V_{in} = any	±1	±1	±1	±1
Maximum input capacitance (pF)	C_{INmax}		10	10	10	10
LOW-level input voltage (V)	V_{ILmax}		1.35	0.8	1.35	0.8
HIGH-level input voltage (V)	V_{IHmin}		3.85	2.0	3.85	2.0

C_{INmax} The maximum capacitance of an input. This number can be used when figuring the AC load on an output that drives this and other inputs. Most manufacturers also specify a lower, typical input capacitance of 2 to 5 pF, which gives a good estimate of AC load if you're not unlucky.

V_{ILmax} The maximum voltage that an input is guaranteed to recognize as LOW. Note that the values are different for HC/AHC versus HCT/AHCT. The "CMOS" value, 1.35 V, is 30% of the minimum power-supply voltage, while the "TTL" value is 0.8 V for compatibility with TTL families.

V_{IHmin} The minimum voltage that an input is guaranteed to recognize as HIGH. The "CMOS" value, 3.85 V, is 70% of the maximum power-supply voltage, while the "TTL" value is 2.0 V for compatibility with TTL families. (Unlike CMOS levels, TTL input levels are not symmetric with respect to the power-supply rails.)

The specifications for TTL-compatible CMOS outputs usually have two sets of output parameters; one set or the other is used depending on how an output is loaded. A *CMOS load* is one that requires the output to sink and source very little DC current, 20 μA for HC/HCT and 50 μA for AHC/AHCT. This is, of course, the case when the CMOS outputs drive only CMOS inputs. With CMOS loads, CMOS outputs maintain an output voltage within 0.1 V of the supply rails, 0 and V_{CC}. (A worst-case $V_{\text{CC}} = 4.5$ V is used for the table entries; hence, $V_{\text{OHminC}} = 4.4$ V.)

CMOS load

A *TTL load* is one that consumes much more sink and source current, up to 4 mA from an HC/HCT output and 8 mA from an AHC/AHCT output. In this case, a higher voltage drop occurs across the "on" transistors in the output circuit, but the output voltage is still guaranteed to be within the normal range for TTL. While it's very unlikely for your designs to use TTL nowadays, these specs are useful if you need to drive any other load that consumes significant current.

TTL load

Table 14-5 lists CMOS output specifications for both CMOS and TTL loads. These parameters have the following meanings:

I_{OLmaxC} The maximum current that an output can supply in the LOW state while driving a CMOS load. Since this is a positive value, current flows *into* the output pin.

I_{OLmaxT} The maximum current that an output can supply in the LOW state while driving a TTL load.

SAVING ENERGY There are practical as well as geopolitical reasons for saving energy in digital systems. Lower energy consumption means lower cost of power supplies and cooling systems. Also, a digital system's reliability is improved more by running it cooler than by any other single reliability improvement strategy.

Table 14-5 Output specifications for CMOS families operating with V_{CC} between 4.5 and 5.5 V.

			Family			
Description	***Symbol***	***Condition***	***HC***	***HCT***	***AHC***	***AHCT***
LOW-level output current (mA)	I_{OLmaxC}	CMOS load	0.02	0.02	0.05	0.05
	I_{OLmaxT}	TTL load	4.0	4.0	8.0	8.0
LOW-level output voltage (V)	V_{OLmaxC}	$I_{\mathrm{out}} \le \mathrm{I}_{\mathrm{OLmaxC}}$	0.1	0.1	0.1	0.1
	V_{OLmaxT}	$I_{\mathrm{out}} \le \mathrm{I}_{\mathrm{OLmaxT}}$	0.33	0.33	0.44	0.44
HIGH-level output current (mA)	I_{OHmaxC}	CMOS load	–0.02	–0.02	–0.05	–0.05
	I_{OHmaxT}	TTL load	–4.0	–4.0	–8.0	–8.0
HIGH-level output voltage (V)	V_{OHminC}	$\lvert I_{\mathrm{out}}\rvert \le \lvert I_{\mathrm{OHmaxC}}\rvert$	4.4	4.4	4.4	4.4
	V_{OHminT}	$\lvert I_{\mathrm{out}}\rvert \le \lvert I_{\mathrm{OHmaxT}}\rvert$	3.84	3.84	3.80	3.80

V_{OLmaxC} The maximum voltage that a LOW output is guaranteed to produce while driving a CMOS load, that is, as long as I_{OLmaxC} is not exceeded.

V_{OLmaxT} The maximum voltage that a LOW output is guaranteed to produce while driving a TTL load, that is, as long as I_{OLmaxT} is not exceeded.

I_{OHmaxC} The maximum current that an output can supply in the HIGH state while driving a CMOS load. Since this is a negative value, positive current flows out of the output pin.

I_{OHmaxT} The maximum current that an output can supply in the HIGH state while driving a TTL load.

V_{OHminC} The minimum voltage that a HIGH output is guaranteed to produce while driving a CMOS load, that is, as long as I_{OHmaxC} is not exceeded.

V_{OHminT} The minimum voltage that a HIGH output is guaranteed to produce while driving a TTL load, that is, as long as I_{OHmaxT} is not exceeded.

The voltage parameters above determine DC noise margins. The LOW-state DC noise margin is the difference between V_{OLmax} and V_{ILmax}. This depends on the characteristics of both the driving output and the driven inputs. For example, the LOW-state DC noise margin of HCT driving a few HCT inputs (a CMOS load) is 0.8 – 0.1 = 0.7 V. With a TTL load, the noise margin for the HCT inputs drops to 0.8 – 0.33 = 0.47 V. Similarly, the HIGH-state DC noise margin is the difference between V_{OHmin} and V_{IHmin}. In general, when different families are interconnected, you have to compare the appropriate V_{OLmax} and V_{OHmin} of the driving gate with V_{ILmax} and V_{IHmin} of all the driven gates to determine the worst-case noise margins.

The I_{OLmax} and I_{OHmax} parameters in the table determine fanout capability and are especially important when an output drives inputs in one or more different families. Two calculations must be performed to determine whether an output is operating within its rated fanout capability:

HIGH-state fanout The I_{IHmax} values for all of the driven inputs are added. The sum must not exceed I_{OHmax} of the driving output.

LOW-state fanout The I_{ILmax} values for all of the driven inputs are added. The sum must not exceed I_{OLmax} of the driving output

Note that the input and output characteristics of specific components may vary from the representative values given in Table 14-5, so you must always consult the manufacturers' data sheets when analyzing a real design.

*14.6.4 AC and ACT

AC (Advanced CMOS)

ACT (Advanced CMOS, TTL compatible)

Introduced in the mid-1980s, a pair of more advanced CMOS families are aptly named— *AC (Advanced CMOS)* and *ACT (Advanced CMOS, TTL compatible)*. These families are very fast, and they can source or sink a lot of current, up to 24 mA in either state. Like HC and HCT, and AHC and AHCT, the AC and ACT families differ only in the input levels that they recognize; their output characteristics are the same. Also like the other CMOS families, AC/ACT outputs have symmetric output drive.

Devices in the AC and especially ACT families were popular because of their ability to drive heavy DC loads, including TTL devices. Their outputs also have very fast rise and fall times, which contributes to faster overall system operation, but at a price. The rise and fall times are so fast that they are often a major source of "analog" problems, including switching noise and ground bounce. As a result, the families in the next subsection were developed, and they gradually supplanted the ACT family in most applications requiring TTL compatibility.

*14.6.5 FCT and FCT-T

FCT (Fast CMOS, TTL compatible)

In the early 1990s, yet another CMOS family was launched. The key benefit of the *FCT (Fast CMOS, TTL compatible)* family was its ability to meet or exceed the speed and the output drive capability of the best TTL families while reducing power consumption and maintaining full compatibility with TTL. FCT output circuits are specifically designed with rise and fall times that are more controlled as compared to those of AC/ACT outputs, so FCT outputs do not create quite the same magnitude of "analog" problems.

FCT-T (Fast CMOS, TTL compatible with TTL V_{OH})

Still, the original FCT family had the drawback of producing a full 5-V CMOS V_{OH}, creating enormous CV^2f power dissipation and circuit noise as its outputs swung from 0 V to almost 5 V in high-speed (25 MHz+) applications. A variation of the family, *FCT-T (Fast CMOS, TTL compatible with TTL V_{OH})*, was quickly introduced with circuit innovations to reduce the HIGH-level output voltage, thereby reducing both power consumption and switching noise while maintaining the same high operating speed as the original FCT. A suffix of "T" is used on part numbers to denote the FCT-T output structure; for example, 74FCT138T versus 74FCT138.

14.7 Low-Voltage CMOS Logic and Interfacing

All of the CMOS logic families that we described in the previous section can and often do operate with a 5-V power supply. However, two factors have led the IC industry to move toward lower power-supply voltages in CMOS devices:

- In most applications, CMOS output voltages swing from rail to rail, so the V in the CV^2f equation is the power-supply voltage. Cutting power-supply voltage reduces dynamic power dissipation more than proportionally.
- As the industry moves toward ever-smaller transistor geometries, the oxide insulation between a CMOS transistor’s gate and its source and drain is getting ever thinner, and thus incapable of insulating voltage potentials as “high” as 5 V.

As a result, JEDEC, an IC industry standards group, selected 3.3 V ± 0.3V, 2.5 V ± 0.2V, 1.8 V ± 0.15V, 1.5 V ± 0.1V, 1.2 V ± 0.1V, and 1.0 V ± 0.1V as the next “standard” logic power-supply voltages. JEDEC standards specify the input and output logic voltage levels for devices operating with these power-supply voltages. In addition to complying with these JEDEC standards, some devices are specified to operate with a supply as low as 0.7 V.

A few of the new low-voltage CMOS logic families and key characteristics are listed below:

- *LV (Low-Voltage)* CMOS devices are specified to operate at 5.0 V, 3.3 V, or 2.5 V, and have CMOS-compatible input thresholds (0.3 and 0.7 times V_{CC}).
- *LVC (Low-Voltage CMOS)* devices are specified to operate at 3.3 V, 2.5 V. and 1.8 V and have high-current outputs for driving buses. They have TTL-compatible input levels when operating at 3.3 V and they tolerate input levels as high as 5.5 V.
- *ALVC (Advanced Low-Voltage CMOS)* devices are similar to LVC, but are intended for use in low-voltage CMOS-only systems and subsystems (only tolerating inputs up to 3.6 V) and have somewhat better performance.
- *AVC (Advanced Very-low-voltage CMOS)* devices are specified to operate at 3.3 V, 2.5 V and 1.8 V.
- *AUC (Advanced Ultra-low-voltage CMOS)* devices are specified to operate at 2.5 V, 1.8 V, and 1.2 V, and are optimized for 1.8-V operation.

MORE CMOS LOGIC FAMILIES

Since the 1990s, still more CMOS logic families have been introduced, mainly to support lower-voltage operation and to take advantage of spec improvements that have been made possible by overall advances in CMOS technology. We’ll discuss low-voltage CMOS logic and interfacing in general in the next section.

The migration to lower voltages has occurred in stages and will continue to do so. For discrete logic families, the trend has been to produce parts that operate and produce outputs at the lower voltage but that can also tolerate inputs at the higher voltage. This approach allowed 3.3-V CMOS families to operate with 5-V CMOS and TTL families when the voltage migration first began, and has continued to ease interoperation of families at adjacent standard voltages.

*14.7.1 3.3-V LVTTL and LVCMOS Logic Levels

The relationships among signal levels for standard TTL and low-voltage CMOS devices operating at their nominal power-supply voltages are illustrated nicely in Figure 14-58, adapted from a Texas Instruments application note. The original, symmetric signal levels for pure 5-V CMOS families such as HC and AHC are shown in (a). TTL-compatible CMOS families such as HCT, AHCT, and FCT shift the voltages downward for compatibility with TTL as shown in (b).

The first step in the progression of lower CMOS power-supply voltages was 3.3 V. The JEDEC standards for 3.3-V logic actually define two sets of levels. *LVCMOS (low-voltage CMOS) levels* are used in pure CMOS applications where outputs have light DC loads (less than 100 µA), so V_{OL} and V_{OH} are maintained within 0.2 V of the power-supply rails. *LVTTL (low-voltage TTL) levels*, shown in Figure 14-58(c), are used when outputs may have significant DC loads, so V_{OL} can be as high as 0.4 V and V_{OH} can be as low as 2.4 V.

LVCMOS (low-voltage CMOS) levels

LVTTL (low-voltage TTL) levels

The positioning of TTL's logic levels at the low end of the 5-V range was really quite fortuitous. As shown in Figure 14-58(b) and (c), it was possible to define the LVTTL levels to match up with TTL levels exactly. Thus, an LVTTL

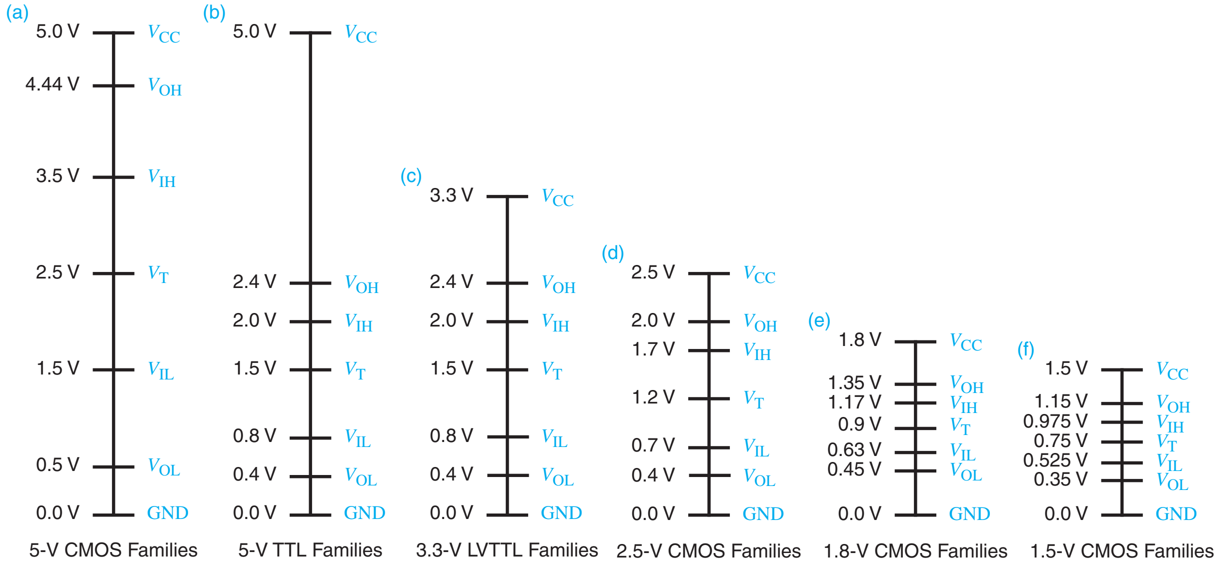


Figure 14-58 Comparison of logic levels: (a) 5-V CMOS; (b) 5-V TTL, including 5-V TTL-compatible CMOS; (c) 3.3-V LVTTL; (d) 2.5-V CMOS; (e) 1.8-V CMOS; (f) 1.5-V CMOS.

MORE POWER (SUPPLIES) TO YOU

Many microprocessors, FPGAs, and ASICs use a simple approach to accommodate different internal and external logic levels—they have two or more power-supply voltages. A low voltage, such as 1.2 V, is supplied to operate the chip's internal gates, or *core logic*. One or more higher voltages, such as 2.5 V or 3.3 V, are used to operate the external input and output circuits, or *pad ring*, for compatibility with older-generation devices in the system. Special buffer circuits are used internally to translate safely and quickly between the core-logic and the pad-ring logic voltages. In modern microprocessors, the internal voltage is often varied dynamically depending on the application's needs—a lower voltage for lower power, and a higher voltage for higher speed.

output can drive a TTL input with no problem, as long as its output current specifications (I_{OLmax}, I_{OHmax}) are respected. Similarly, a TTL output can drive an LVTTL input, except for the problem of driving it beyond LVTTL's 3.3-V V_{CC}, as discussed in the next subsection.

Notice the narrowing of the ranges of valid logic levels and the DC noise margins in the even lower-voltage standards in Figure 14-58(d) through (f). This narrowing further increases the importance of minimizing analog effects like switching noise and ground bounce in modern high-speed designs.

*14.7.2 5-V Tolerant Inputs

The inputs of a gate won't necessarily tolerate voltages greater than V_{CC}. This is a problem when two different logic-voltage ranges are used in a system. For example, 5-V CMOS devices easily produce 4.9-V outputs when lightly loaded, and both CMOS and TTL devices routinely produce 4.0-V outputs even when moderately loaded. The inputs of 3.3-V devices may not like these high voltages.

The maximum voltage V_{Imax} that an input can tolerate is listed in the "absolute maximum ratings" section of the manufacturer's data sheet. For HC devices, V_{Imax} equals V_{CC}. Thus, if an HC device is powered by a 3.3-V supply, its inputs cannot be driven by any 5-V CMOS or TTL outputs without damage. For AHC devices, on the other hand, V_{Imax} is 5.5 V; thus, AHC devices with a 3.3-V power supply may be used to convert 5-V outputs to 3.3-V levels for use with 3.3-V microprocessors, memories, and other devices in a pure 3.3-V subsystem.

clamp diode

Figure 14-59 helps to explain why some inputs are 5-V tolerant and others are not. As shown in (a), the HC and HCT input structure actually contains two reverse-biased *clamp diodes*, which we haven't shown before, between each input signal and V_{CC} and ground. The purpose of these diodes is specifically to shunt any transient input signal voltage less than 0 through *D1* or greater than V_{CC} through *D2* to the corresponding power-supply rail. Such transients often result from transmission-line reflections that can occur on "long" signal lines—ones whose propagation delay is longer than a signal's transition time. Shunting

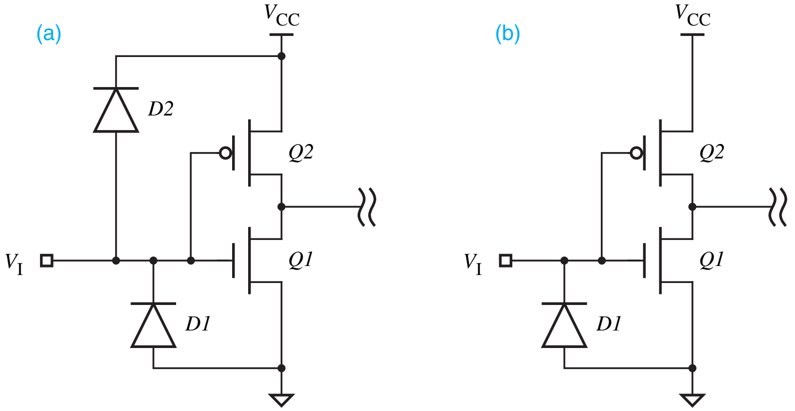


Figure 14-59 CMOS input structures: (a) 5-V intolerant HC; (b) 5-V tolerant AHC.

the transients, called “undershoot” and “overshoot,” to ground or V_{CC} reduces the magnitude and duration of reflections.

Of course, diode *D2* can’t distinguish between transient overshoot and a persistent input voltage greater than V_{CC}. Hence, if a 5-V output is connected to one of these inputs, it will not see the very high impedance normally associated with a CMOS input. Instead, it will see a relatively low impedance path to V_{CC} through the now forward-biased diode *D2*, and excessive current will flow.

Figure 14-59(b) shows a 5-V tolerant CMOS input. This input structure simply omits *D2*; diode *D1* is still provided to clamp undershoot. The AHC family uses this input structure.

The kind of input structure shown in Figure 14-59(b) is necessary but not sufficient to create 5-V tolerant inputs. The transistors in a device’s particular fabrication process must also be able to withstand voltage potentials higher than V_{CC}. On this basis, V_{Imax} in the AHC family is limited to 5.5 V. In some 3.3-V ASIC processes, it’s not possible to get 5-V tolerant inputs, even if you’re willing to give up the transmission-line benefits of diode *D2*.

*14.7.3 5-V Tolerant Outputs

Five-volt tolerance must also be considered for outputs, in particular, when both 3.3-V and 5-V three-state outputs are connected to a bus. When the 3.3-V output is in the Hi-Z, disabled state, a 5-V device may be driving the bus, and a 5-V signal may appear on the 3.3-V device’s *output*.

In this situation, Figure 14-60 explains why some outputs are 5-V tolerant and others are not. As shown in (a), the standard CMOS three-state output has an *n*-channel transistor *Q1* to ground and a *p*-channel transistor *Q2* to V_{CC}. When the output is disabled, circuitry (not shown) holds the gate of *Q1* near 0 V, and the gate of *Q2* near V_{CC}, so both transistors are off and Y is Hi-Z.

Now consider what happens if V_{CC} is 3.3 V and a different device applies a 5-V signal to the output pin Y in Figure 14-60(a). Then the drain of *Q2* (Y) is at 5 V while the gate (V_2) is still at only 3.3 V. With the gate at a lower potential

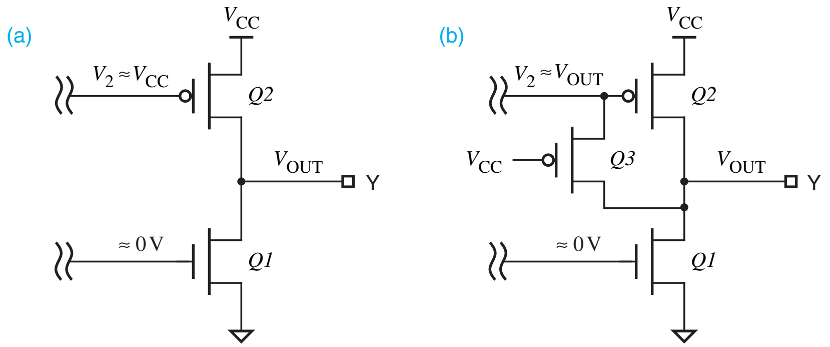


Figure 14-60 CMOS three-state output structures: (a) 5-V intolerant HC and AHC; (b) 5-V tolerant LVC.

than the drain, *Q2* will begin to conduct and provide a relatively low-impedance path from Y to V_{CC}, and excessive current will flow. Both HC and AHC three-state outputs have this structure and therefore are not 5-V tolerant.

In newer, low-voltage CMOS families, manufacturers use various circuit structures to protect three-state outputs in this situation. Figure 14-60(b) shows one that was once used in Texas Instruments' LVC family. An extra *p*-channel transistor *Q3* is used to prevent *Q2* from turning on when it shouldn't. When V_{OUT} is greater than V_{CC}, *Q3* turns on. This forms a relatively low impedance path from Y to the gate of *Q2*, which now stays off because its gate voltage V_2 can no longer be below the drain voltage.

*14.7.4 TTL/LVTTL Interfacing Summary

Based on the information in the preceding subsections, TTL (5-V) and LVTTL (3.3-V) devices can be mixed in the same system subject to three rules:

1. LVTTL outputs can drive TTL inputs directly, subject to the usual constraints on output current (I_{OLmax}, I_{OHmax}) of the driving devices.
2. TTL outputs can drive LVTTL inputs if the inputs are 5-V tolerant.
3. TTL and LVTTL three-state outputs can drive the same bus if the LVTTL outputs are 5-V tolerant.

*14.7.5 Logic Levels Less Than 3.3 V

Issues of input and output voltage tolerance must also be considered when mixing lower-voltage CMOS devices. For example, can the input of an AUC device operating at 1.8 V be connected to a bus driven by a 3.3-V or even a 2.5-V output? The answer is yes, AUC inputs tolerate a maximum of 3.6 V, but you must study the datasheets to learn that.

If the voltage levels in a mixed-level situation are tolerated, there is still the question of whether the logic levels are properly recognized. This usually must be examined in both directions—higher voltage driving lower and vice versa—and both logic states, HIGH and LOW.

For example, a quick look at Figure 14-58(c) and (d) on page 799 shows that V_{OH} of a 2.5-V output equals V_{IH} of a 3.3-V input. In other words, there is zero HIGH-state DC noise margin when a 2.5-V output drives a 3.3-V input—not a good situation, but it could be worse.

Comparing the logic levels for 2.5-V and 1.8-V logic, you can see that the minimum HIGH output voltage for 1.8-V logic is quite a bit lower than what can be recognized as HIGH by a 2.5-V input. A smaller mismatch occurs between 1.8-V and 1.5-V logic, but it still cannot be ignored.

level translator
level shifter

The solution to these problems is to use a *level shifter* (or *level translator)*, a device which is powered by both supply voltages and which internally boosts the lower logic levels to the higher ones. For example, the 74ALVC164245 level shifter can connect two 16-bit buses with different logic levels on its two sides. One side could use 5.0-V or 3.3-V power and logic levels, while the other side uses 2.5-V or 1.8-V power and logic levels.

Many of today's ASICs, FPGAs, and microprocessors contain level translators internally. This allows them to operate, for example, with a 1.8-V or lower core and a 3.3-V pad ring, as we discussed in the box on page 800.

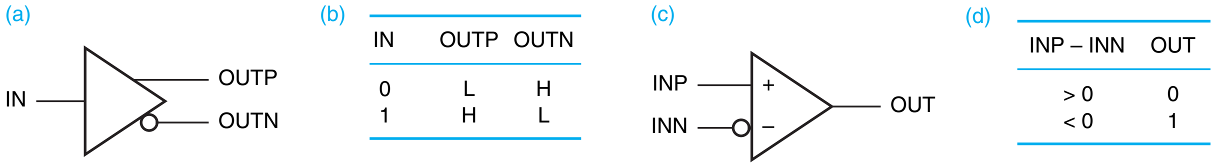
14.8 Differential Signaling

differential signaling

For greater noise immunity, a logic signal can be transmitted on two wires using *differential signaling*. Instead of referencing an absolute voltage level, the logic value on a differential pair depends on the voltage *difference* between the two wires, 1 for a positive difference and 0 for negative. Assuming that the two wires are routed next to each other for their entire signal path, the idea is that any noise will affect both signals equally, leaving their difference more-or-less unchanged. This scheme allows a much lower the absolute voltage swing for each signal polarity while still providing a great deal of noise immunity. The lower voltage swings also allow higher frequency operation, since for any fixed transition speed (V/ns), a smaller voltage difference means shorter transition time.

double-ended signaling
single-ended signaling

Differential signals are sometimes called *doubled-ended*, and ordinary signals on one wire are called *single-ended*. Logic symbols and function tables for differential drivers and receivers are shown in Figure 14-61. The receiver's function table indicates that any positive voltage difference is a 1, and negative is a 0. A device's datasheet will specify the minimum absolute voltage difference required for reliable detection of the input logic level.



(b)

IN	OUTP	OUTN
0	L	H
1	H	L

(d)

INP – INN	OUT
> 0	0
< 0	1

Figure 14-61 Differential signaling: (a) driver and function table; (b,c) receiver and function table.

References

After seeing the results of last few decades' amazing pace of development in digital electronics, it's easy to forget that logic circuits had an important place in technologies that came before the transistor. In Chapter 5 of *Introduction to the Methodology of Switching Circuits* (Van Nostrand, 1972), George J. Klir shows how logic can be (and has been) performed by a variety of physical devices, including relays, vacuum tubes, and pneumatic systems.

For another perspective on the electronics material in this chapter, you can consult almost any modern electronics text. Many contain a much more analytical discussion of digital circuit operation; for example, see *Introduction to Electronic Circuit Design* by R. Spencer and M. Ghausi (Pearson, 2003). A good introduction to ICs and logic families can be found in *Digital Integrated Circuits* by J. M. Rabaey, A. Chandrakasan, and B. Nikolic (Pearson, 2003, second edition).

A light-hearted and very readable introduction to digital circuits can be found in Clive Maxfield's *Bebop to the Boolean Boogie* (Newnes, 2008, third edition). Some people think that the seafood gumbo recipe in Appendix H is alone worth the price! Even without the recipe, the book is a well-illustrated classic that guides you through the basics of digital electronics fundamentals, components, and processes.

A sound understanding of the electrical aspects of digital circuit operation, including capacitive effects, inductive effects, and transmission-line effects, is mandatory for successful high-speed circuit design. Unquestionably, the best book on this subject is *High-Speed Digital Design: A Handbook of Black Magic*, by Howard Johnson and Martin Graham (Prentice Hall, 1993). It combines solid electronics principles with tremendous insight and experience in the design of practical digital systems. Also see Johnson's follow-on book, *High-Speed Signal Propagation: Advanced Black Magic* (Prentice Hall, 2003).

Characteristics of today's logic families can be found in the data sheets published by the device manufacturers. Old-time digital designers are proud of their collections of thick databooks published by the device manufacturers, but nowadays all of the latest specs can be found on the Web. Among the better sites for logic-family data sheets and design application notes are `www.ti.com` (Texas Instruments) and `www.onsemi.com` (formerly Fairchild Semiconductor).

Over the years, the JEDEC (Joint Electron Device Engineering Council) has published and updated standards for digital logic levels from 3.3 V (first published in 1994) all the way down to 1.0 V (2007). Their standards can be found at `www.jedec.org`; registration is required but free.

Drill Problems

14.1 The Stub Series Terminated low Voltage (SSTV) logic family, used for SDRAM modules, defines a LOW signal to be in the range 0.0–0.7 V and a HIGH signal to be in the range 1.7–2.5 V. Under a positive-logic convention, indicate the logic value associated with each of the following signal levels:

(a)	0.1 V	(b)	0.7 V	(c)	1.7 V	(d)	–0.6 V
(e)	1.6 V	(f)	–2.0 V	(g)	2.4 V	(h)	3.3 V

14.2 Repeat Drill 14.1 using a negative-logic convention.

14.3 Discuss how a logic buffer amplifier is different from an audio amplifier.

14.4 Is a buffer amplifier equivalent to a 1-input AND gate or a 1-input OR gate?

14.5 Write three completely different definitions of “gate” used in this chapter.

14.6 How many transistors are used in a 2-input CMOS NOR gate? How many of each type are used?

14.7 (Hobbyists only.) Draw an equivalent circuit for a CMOS NOR gate using two single-pole, double-throw 110-V relays.

14.8 For a given silicon area, which is likely to be faster, a CMOS NAND gate or a CMOS NOR?

14.9 Define “fan-in” and “fanout.” Which one are you or an EDA tool likely to have to calculate?

14.10 Draw the circuit diagram, function table, and logic symbol for a 3-input CMOS NOR gate in the style of Figure 14-12.

14.11 Draw switch models in the style of Figure 14-10 for a 2-input CMOS NOR gate for all four input combinations.

14.12 Draw a circuit diagram, function table, and logic symbol for a CMOS OR gate in the style of Figure 14-15.

14.13 Which has fewer transistors, a 3-input CMOS inverting gate or a noninverting gate?

14.14 Name and draw the logic symbols of two different 3-input CMOS gates that each use six transistors.

14.15 Name and draw the logic symbols for two more 3-input CMOS gates using six transistors each, that you didn’t give in Drill 14.14’s answer.

14.16 Name and draw the logic symbol of a 3-input CMOS gate that uses only three transistors.

14.17 Which 8-input CMOS gate would you expect to be faster, NAND or AND? Why?

14.18 How is it that perfume can be bad for digital designers?

14.19 Using the data sheet in Table 14-1, determine the worst-case LOW-state and HIGH-state DC noise margins of the 74HC00. State any assumptions required by your answer.

14.20 Using the specs in Tables 14-4 and 14-5, determine the HIGH-state DC noise margins of the 74HCT devices driving 74HCT for both CMOS and TTL loads.

14.21 The circuit in Figure X14.21(a) is a type of CMOS AND-OR-INVERT gate. Write a function table for this circuit in the style of Figure 14-11(b), and a corresponding logic diagram using AND and OR gates and inverters.

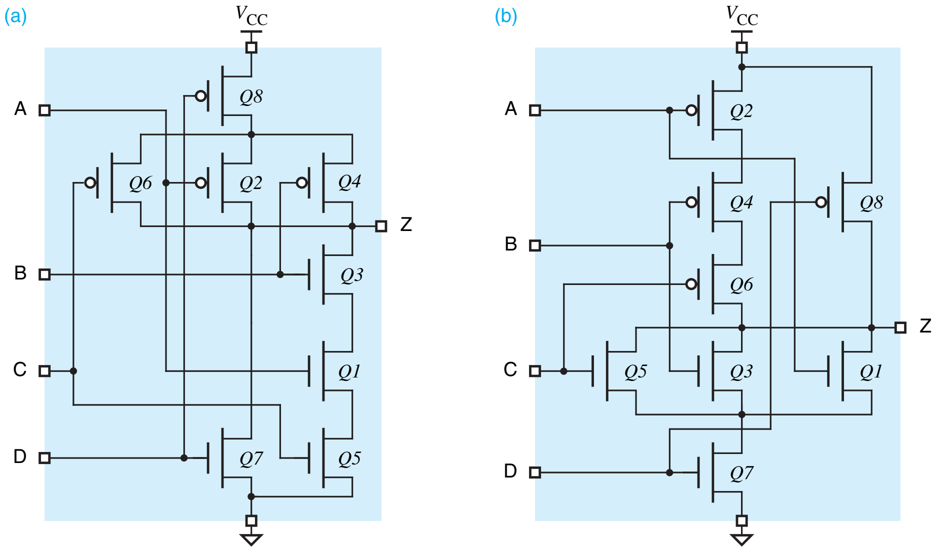


Figure X14.21

14.22 The circuit in Figure X14.21(b) is a type of CMOS OR-AND-INVERT gate. Write a function table for this circuit in the style of Figure 14-11(b), and a corresponding logic diagram using AND and OR gates and inverters.

14.23 Search online for the Texas Instruments data sheet for a 74ALVC00, and determine its worst-case LOW-state and HIGH-state DC noise margins when operated with a 3.3-V (typical) supply and maximum DC loading on its outputs. State any assumptions you make.

14.24 Repeat Drill 14.23 assuming "CMOS loads."

14.25 Section 14.3 defines twelve different electrical parameters for CMOS circuits. Using the data sheet in Table 14-1, determine the worst-case value of each of these for the 74HC00. State any assumptions required by your answer.

14.26 Search online for the Texas Instruments data sheet for a 74AHC00 and repeat Drill 14.25 for that component.

14.27 Based on the conventions and definitions in Section 14.2, if the current at a device output is specified as a negative number, is the output sourcing current or sinking current?

14.28 Across the range of valid HIGH input levels, 3.15–5.0 V, at what input level would you expect the 74HC00 (see Table 14-1) operating at 5.0 V to consume the most power?

14.29 Determine the LOW-state and HIGH-state DC fanout of a 74HC00 when it drives 74ALS00-like inputs. (Refer to Table 14-1 and an online datasheet for the Texas Instruments 74ALS00.)

14.30 Estimate the "on" resistances of the *p*-channel and *n*-channel output transistors of the 74HC00 using information in Table 14-1.

14.31 Recalculate and relabel Figures 14-23 and 14-24, assuming V_{cc} = 3.3 V, "on" resistances R_p = 100 Ω and R_n = 50 Ω, and the same load resistances.

14.32 Repeat Drill 14.31 for Figure 14-25.

14.33 Repeat Drill 14.31 for Figure 14-27.

14.34 How much high-state DC noise margin is available in an inverter whose transfer characteristic under worst-case conditions is shown in Figure X14.34? How much low-state DC noise margin is available? (Assume 1.5-V and 3.5-V thresholds for LOW and HIGH.)

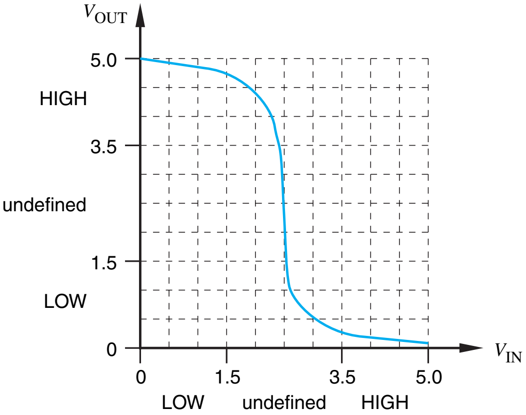


Figure X14.34

14.35 For each of the following resistive loads, determine whether the output drive specifications of the 74HC00 over the commercial operating range are exceeded. Refer to Table 14-1, and use V_{OLmax} = 0.33 V, V_{OHmin} = 3.84 V, and V_{CC} = 5.0 V. You may not exceed I_{OLmax} or I_{OHmax} in any state.

(a)	810 Ω to V_{CC}	(b)	330 Ω to V_{CC} and 470 Ω to GND
(c)	1 kΩ to GND	(d)	680 Ω to V_{CC} and 810 Ω to GND
(e)	1.2 kΩ to V_{CC}	(f)	1 kΩ to V_{CC} and 680 Ω to GND
(g)	2.2 kΩ to V_{CC}	(h)	1.2 kΩ to V_{CC} and 1 kΩ to GND

14.36 Under what circumstances is it safe to allow an unused CMOS input to float?

14.37 Explain why replacing small decoupling capacitors to larger ones with larger capacitance may not be a good idea.

14.38 When is it important to hold hands with a friend?

14.39 Name the two components of CMOS logic gate's delay. How are either or both affected by the direction of the output transition?

14.40 Determine the *RC* time constant for each of the following resistor-capacitor combinations:

(a)	R = 120 Ω, C = 47 pF	(b)	R = 3.3 kΩ, C = 100 pF
(c)	R = 47 Ω, C = 68 pF	(d)	R = 1.5 kΩ, C = 150 pF

14.41 Comparing 2-input CMOS NAND and NOR gates where all the *n*-channel and *p*-channel transistors have the same size, explain why the NOR gate's HIGH-to-LOW output transitions are about four times slower than the NAND's.

14.42 Which would you expect to have a bigger effect on the power consumption of a CMOS circuit, a 10% increase in power-supply voltage, or a 15% increase in load and internal capacitance?

14.43 Explain why the number of CMOS inputs connected to the output of a CMOS gate generally is not limited by DC fanout considerations.

14.44 A particular Schmitt-trigger inverter has V_{ILmax} = 0.7 V, V_{IHmin} = 2.0 V, $V_{\text{T+}}$ = 1.8 V, and $V_{\text{T-}}$ = 1.2 V. How much hysteresis does it have?

14.45 What would happen if three-state outputs turned on faster than they turned off?

14.46 A particular LED has a voltage drop of about 1.6 V in the "on" state and requires about 6 mA of current for normal brightness. Determine an appropriate value for the pull-up resistor when the LED is connected to a 74AC00 NAND gate as shown in Figure 14-50(a), with both power rails at 5.0 V.

14.47 How does the answer for Drill 14.46 change if the LED only requires 3 mA and is connected to a 74HC00 as shown in Figure 14-50(b)?

14.48 Which would you expect to be faster, a CMOS AND gate or a CMOS AND-OR-INVERT gate, assuming all transistors switch at the same speed? Why?

14.49 For a given load capacitance and transition rate, what conditions in a logic family with specs in this chapter leads to the highest dynamic power dissipation? What conditions give the lowest dynamic power dissipation and how do they compare?

14.50 Using Figure 14-58, determine the DC noise margins for 1.5-V CMOS.

14.51 Find a commercially available 74-series device with a very long part number, based on the logic family and the device number, but excluding the package type, temperature range, and so on. You should be able to beat 74ALVCH16244.

Exercises

14.52 Design a CMOS circuit that has the functional behavior shown in Figure X14.52. (*Hint:* Only eight transistors are required.)

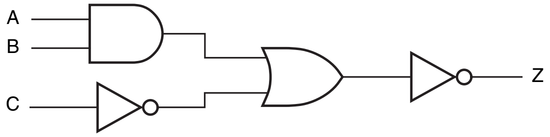


Figure X14.52

14.53 Design a CMOS circuit that has the functional behavior shown in Figure X14.53. (*Hint:* Only eight transistors are required.)

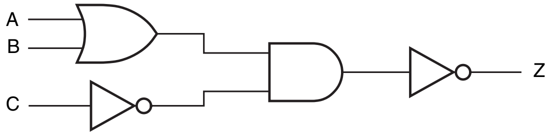


Figure X14.53

14.54 Draw a circuit diagram, function table, and logic symbol in the style of Figure 14-15 for a CMOS gate with two inputs A and B and an output Z, where Z=1 if A=0 or B=1, and Z=0 otherwise. (*Hint:* Only six transistors are needed.)

14.55 Draw a circuit diagram, function table, and logic symbol in the style of Figure 14-15 for a CMOS gate with two inputs A and B and an output Z, where Z=0 if A=1 or B=0, and Z=1 otherwise. (*Hint:* Only six transistors are needed.)

14.56 Draw a figure showing the logical structure of an 8-input CMOS NAND gate, assuming that at most 4-input NAND and 2-input NOR gate circuits are practical. Using your general knowledge of CMOS characteristics, select a circuit structure that minimizes the NAND gate's propagation delay for a given silicon area, and explain why this is so.

14.57 Repeat Exercise 14.56 for a 7-input NAND gate using at most 3-input NAND and 2-input NOR gates.

14.58 Find a gate-level design for the BUT gate defined in Exercise 7.47 that uses a minimum number of transistors when realized in CMOS. You may use inverting gates with up to 4 inputs, AOI or OAI gates, transmission gates, or other transistor-level tricks. Write the output expressions (which need not be two-level sums of products), and draw the logic diagram.

14.59 What logic function is performed by the CMOS circuit shown in Figure X14.59?

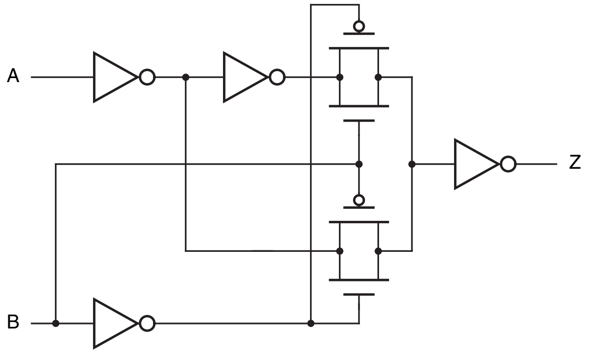


Figure X14.59

14.60 How much current and power are "wasted" in Figure 14-28(b)?

14.61 Using information as needed from Table 14-1, determine the minimum total "on" resistance of two series *p*-channel transistors in the 74HC00 when driving a permissible DC load at a worst-case output voltage.

14.62 Repeat Exercise 14.61 for the two parallel *n*-channel transistors.

14.63 What do the answers to Exercises 14.61 and 14.62 tell you about the relative "on" resistances of the 'HC00's *n*-channel and *p*-channel transistors?

14.64 Show detailed calculations for V_{OUT} in Figures 14-29 and 14-30. (*Hint:* Create a Thévenin equivalent for the CMOS inverter in each figure.)

14.65 Consider the dynamic behavior of a CMOS output driving a given capacitive load. If the resistance of the charging path is double the resistance of the discharging path, is the rise time exactly twice the fall time? If not, what other factors affect the transition times?

14.66 Analyze the fall time of the CMOS inverter output of Figure 14-33 using R_{L}=1 KΩ and V_{L}=2.0 V. Compare your result with the result derived using Figure 14-34 and explain.

14.67 Repeat Exercise 14.66 for rise time.

14.68 Rework the timing calculations corresponding to Figure 14-34 for 74AHC CMOS operating at $V_{CC} = 3.3$ V±0.3 V. You may assume that $R_{p(on)} = 140\ \Omega$ and $R_{n(on)} = 50\ \Omega$.

14.69 Rework the timing calculations corresponding to Figure 14-36 for 74AHC CMOS operating at $V_{CC} = 3.3$ V±0.3 V. You may assume that $R_{p(on)} = 140\ \Omega$ and $R_{n(on)} = 50\ \Omega$.

14.70 Write the truth table and a logic diagram for the logic function performed by the CMOS circuit in Figure X14.70.

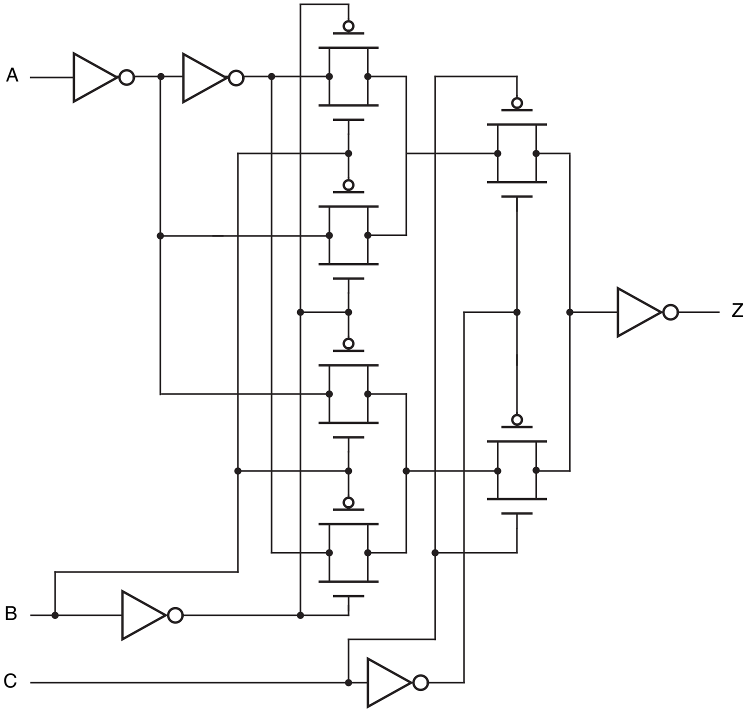


Figure X14.70

14.71 Using the specifications in Table 14-5, estimate the minimum "on" resistances of the *p*-channel and *n*-channel transistors in 74AHC-series CMOS logic when driving a permissible load at the specified worst-case output voltage.

14.72 Repeat Exercise 14.71 using information from an online data sheet for the Texas Instruments 74ALVC00.

14.73 Create a 4×4×2×2 matrix of worst-case DC noise margins for the following CMOS interfacing situations: an (HC, HCT, AHC, or AHCT) output driving an (HC, HCT, AHC, or AHCT) input with a (CMOS, TTL) load in the (LOW, HIGH) state; Figure X14.73 illustrates. (*Hints:* There are 64 different combinations, but many give identical results. Some combinations yield negative margins.)

14.74 Using Figure 14-58, determine the DC noise margins for 5-V-tolerant, 3.3-V CMOS driving 5-V CMOS logic with TTL input levels, and vice versa.

14.75 Using Figure 14-58, determine the DC noise margins for 3.3-V-tolerant, 2.5-V CMOS driving 3.3-V CMOS, and vice versa.

14.76 Using Figure 14-58, determine the DC noise margins for (a) 2.5-V CMOS driving itself, and (b) 1.8-V CMOS driving itself.

14.77 Calculate the approximate output voltage at Z in Figure 14-53, assuming that the gates are HCT-series CMOS.

14.78 In the LED example in Section 14.5.5, a designer chose a resistor value of 680 Ω for a lower-current LED and found that the open-drain gate was able to maintain its output at 0.2 V while driving the LED. How much current flows through the LED, and how much power is dissipated by the pull-up resistor in this case?

14.79 The dynamic power dissipation specification and calculation for an integrated function like a counter is more complicated than the one for a simple gate. Search online for the data sheet for a Texas Instruments CD74HC163 4-bit counter, which is similar to the CNTR4U of Section 11.1.3. Determine its dynamic power dissipation with a 3.3 V power supply assuming that it is continuously enabled, has an input frequency of 10 MHz, and has a load of 25 pF on each output.

14.80 Using only AND and NOR gates, draw a logic diagram for the logic function performed by the circuit in Figure 14-52.

14.81 How many transistors are needed to perform the logic function in Figure 14-52 using an AND-OR-INVERT structure? Sketch the transistor-level circuit.

14.82 Redraw the circuit diagram of a CMOS 3-state buffer in Figure 14-45 using actual transistors instead of NAND, NOR, and inverter symbols. Can you design a circuit for the same function that requires a smaller total number of transistors? If so, draw it.

14.83 Modify the CMOS 3-state buffer circuit in Figure 14-45 so that the output is in the Hi-Z state when the enable input is HIGH. The modified circuit should require no more transistors than the original.

14.84 Using information in Table 14-1, estimate how much current can flow through each output pin if the outputs of two different 74HC00s are fighting.

14.85 A *Thévenin termination* for an open-collector or three-state bus has the structure shown in Figure X14.85(a). The idea is that, with appropriate values of *R1* and *R2*, this circuit is equivalent to the termination in (b) for any desired values of *R* and *V* (between 0 and V_{CC}). The value of *V* determines the voltage on the bus

Input

Output	HC		HCT		VHC		VHCT	
HC	CL	TL	CL	TL	CL	TL	CL	TL
	CH	TH	CH	TH	CH	TH	CH	TH
HCT	CL	TL	CL	TL	CL	TL	CL	TL
	CH	TH	CH	TH	CH	TH	CH	TH
VHC	CL	TL	CL	TL	CL	TL	CL	TL
	CH	TH	CH	TH	CH	TH	CH	TH
VHCT	CL	TL	CL	TL	CL	TL	CL	TL
	CH	TH	CH	TH	CH	TH	CH	TH

Key:
CL = CMOS load, LOW
CH = CMOS load, HIGH
TL = TTL load, LOW
TH = TTL load, HIGH

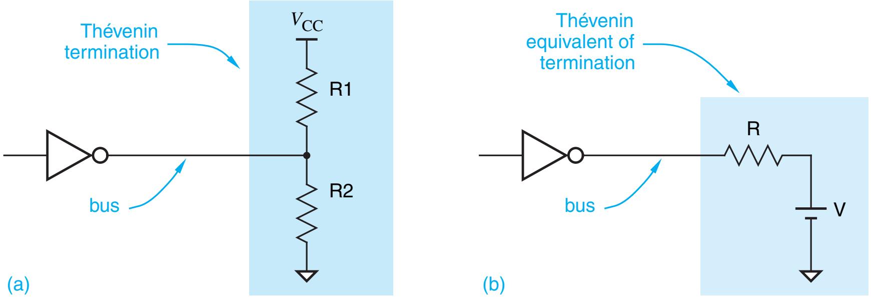
Figure X14.73

when no device is driving it, and the value of R is selected to match the characteristic impedance of the bus for transmission-line purposes. For each of the following desired pairs of V and R, determine the required values of $R1$ and $R2$, assuming $V_{CC} = 5.0$ V:

(a) $V = 2.5, R = 220$ (b) $V = 2.7, R = 180$

(c) $V = 3.0, R = 120$ (d) $V = 2.0, R = 75$

Figure X14.85



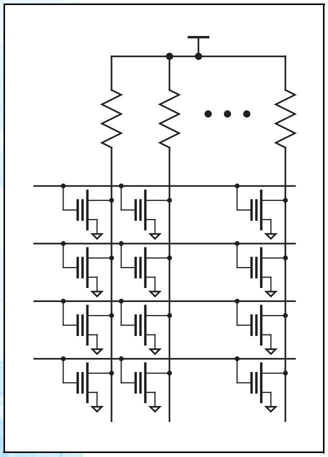
14.86 For each of the $R1$ and $R2$ pairs in Exercise 14.85, determine whether the termination can be properly driven by a three-state output in the 74AHC family having the output specs in Table 14-5. For proper operation, the family's I_{OL} and I_{OH} specs must not be exceeded when $V_{OL} = V_{OLmax}$ and $V_{OH} = V_{OHmin}$, respectively, assuming "TTL" output levels.

14.87 Repeat Exercise 14.85 for the following desired pairs of V and R, assuming that $V_{CC} = 3.3$ V:

(a) $V = 1.5, R = 220$ (b) $V = 2.0, R = 180$

(c) $V = 1.2, R = 120$ (d) $V = 1.67, R = 75$

14.88 Search online for the Texas Instruments 74ALVC125 three-state-buffer data sheet. Then, for each Thévenin termination in Exercise 14.87, determine whether it can be properly driven by a 74ALVC125 output. For proper operation, the device's maximum I_{OL} and I_{OH} specs must not be exceeded when $V_{OL} = V_{OLmax}$ and $V_{OH} = V_{OHmin}$, respectively.



chapter 15

ROMs, RAMs, and FPGAs

Any sequential circuit has memory of a sort, since each flip-flop or latch stores one bit of information. However, we usually reserve the word “memory” to refer to bits that are stored in a structured way, usually as a two-dimensional array in which one row of bits is accessed at a time.

This chapter describes several different memory organizations and a few commercially available memory chips. The same kinds of memory may be embedded into larger VLSI chips, where they are combined with other logic circuits to perform a useful function.

The applications of memory are many and varied. We’ve already seen that memories are a key element in FPGAs, which use tens of thousands of “lookup-table” (LUT) memories to perform logic functions. In a microprocessor’s central processing unit (CPU), a read-only memory may be used to define multiple small steps used to execute each complex instruction in the CPU’s instruction set, or to store “seed” constants used in a division algorithm. Alongside the CPU, a fast, read/write “static memory” may serve as a cache to hold recently used instructions and data. A microprocessor’s main read/write memory subsystem may hold billions of bits in “dynamic memory” that stores the operating system, running applications, and data.

Applications of memory are not limited to microprocessors or even to purely digital systems. For example, network devices like Ethernet switches and internet routers use fast “static memories” as a “switching fabric” to transfer packets between network ports. There are many examples of modern audio/visual equipment that use memories to temporarily store digitized signals for enhancement through digital signal processing. And all kinds of data acquisition equipment converts physical information (such as temperature, moisture, and movement) into digital data, and stores it in a memory for later analysis.

We begin this chapter with a discussion of read-only memory, including both "traditional" ROM and the newer "flash" ROM used in smartphones, tablets, and other portable devices. We then describe the two commonly used types of read/write memory—static and dynamic. We also discuss the internal structures and bus interfaces of the different memory types.

The last section in this chapter looks at a few aspects of FPGA structure, beyond what we've done in previous chapters. By enabling very quick development of customized systems and subsystems, FPGAs have become essential building blocks in modern digital design.

15.1 Read-Only Memory

read-only memory (ROM)

address input

data output

A *read-only memory (ROM)* is a combinational circuit with *n* inputs and *b* outputs, as shown in Figure 15-1. The inputs are called *address inputs* and are traditionally named A0, A1, … , An–1. The outputs are called *data outputs* and are typically named D0, D1, … , Db–1.

We think of ROM as being a type of memory because of the organizational paradigm that describes its operation. Information is "stored" in a ROM when it is programmed (we'll say more about how this is done in Section 15.1.3). ROM has an important difference from many other types of integrated-circuit memory. It is *nonvolatile memory*; that is, its contents are preserved even if no power is applied.

nonvolatile memory

In Section 6.1, we showed how a ROM "stores" the truth table of an *n*-input, *b*-output combinational logic function. Since a ROM is a combinational circuit, you would be correct to say that it's not really a memory at all. In terms of digital circuit operation, you can treat a ROM like any other combinational logic element.

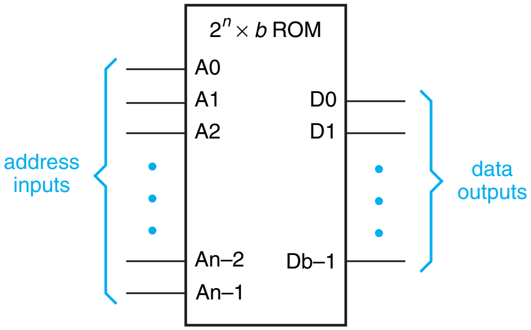


Figure 15-1
Basic structure of a $2^n \times b$ ROM.

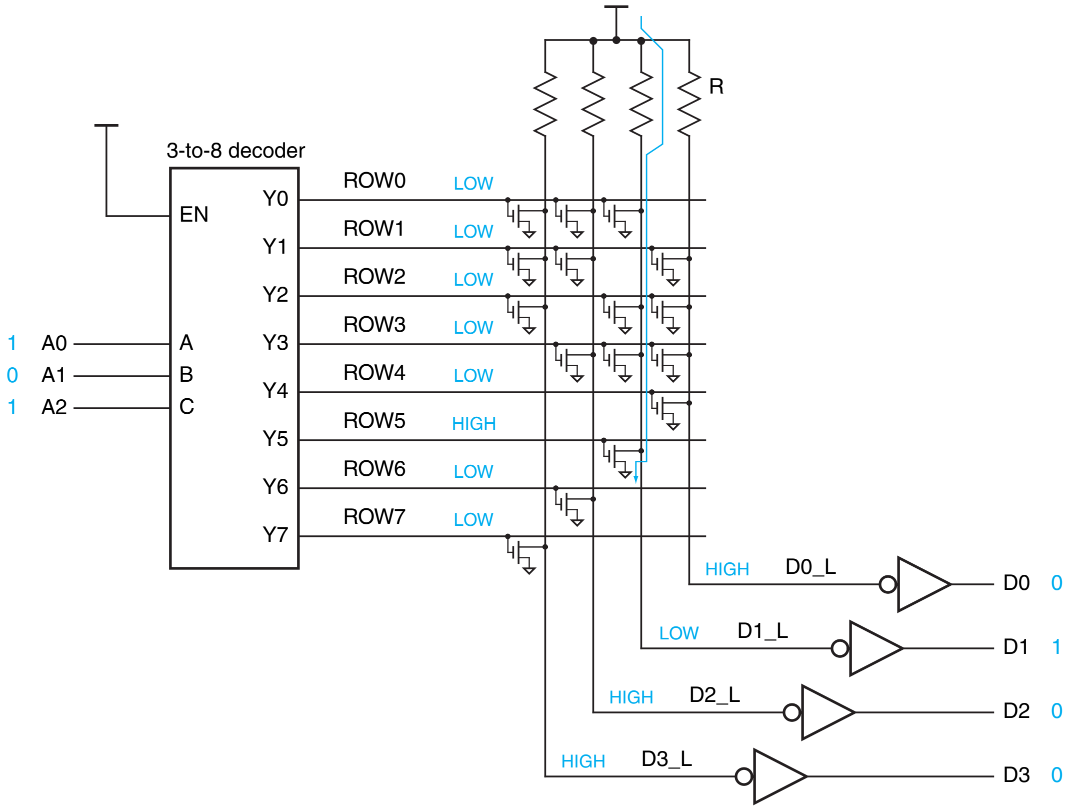


Figure 15-2 Logic diagram of a simple 8×4 ROM.

15.1.1 Internal ROM Structure

The mechanism used by ROMs to “store” information varies with different ROM technologies. Modern ROMs use an MOS transistor, one per stored bit, to distinguish between a 0 and a 1.

Figure 15-2 is the schematic of a primitive 8×4 ROM that you could build yourself using a 3-to-8 decoder and a handful of discrete NMOS transistors. The address inputs select one of the decoder outputs to be asserted. Each decoder output is called a *word line* because it selects one row or word of the table stored in the ROM. The figure shows the situation with A2–A0 = 101 and the ROW5 decoder output asserted.

word line

Each vertical line in Figure 15-2 is called a *bit line* because it corresponds to one output bit of the ROM. An asserted word line turns on a transistor, if one is present, at the intersection of the word line and a bit line. A transistor pulls the bit line LOW when turned on. There is only one transistor in row 5, and when ROW5 is asserted, the corresponding bit line (D1_L) is pulled LOW. All of the other bit lines remain HIGH, since none of the other decoder outputs are asserted

bit line

and all the other transistors in the array are off. The bit lines are buffered through inverters to produce the D3–D0 ROM outputs, 0010 for the case shown.

In the ROM circuit of Figure 15-2, each intersection between a word line and a bit line corresponds to one bit of "memory." If a transistor is present at the intersection, a 1 is stored; otherwise, a 0 is stored. If you were to build this circuit in the lab, you would "program" the memory by inserting and removing transistors at each intersection.

The transistor pattern shown in Figure 15-2 corresponds to the 2-to-4-decoder truth table of Table 6-1 on page 241. This doesn't seem very efficient—we used a 3-to-8 decoder and a bunch of transistors to build the ROM version of a 2-to-4 decoder. We could have profitably used some of the gates in the 3-to-8 decoder directly! However, we'll show a more efficient ROM structure and a more useful example in the next subsection.

15.1.2 Two-Dimensional Decoding

Suppose you wanted to build a 128 × 1 ROM using the kind of structure described in the preceding subsection. Have you ever thought about what it would take to build a 7-to-128 decoder in two levels of logic? Try 128 7-input NAND gates to begin with, and add 14 buffers and inverters with a fanout of 64 each! ROMs with millions of bits and more are available commercially; trust me, they do not contain 20-to-1,048,576 decoders or worse. Instead, they use a different structure, called *two-dimensional decoding*, to reduce the decoder size to something proportional to the square root of the number of addresses.

two-dimensional decoding

The basic idea in two-dimensional decoding is to arrange the ROM cells in an array that is as close as possible to square. For example, Figure 15-3 shows a possible internal structure for a 128 × 1 ROM. The three high-order address bits, A6–A4, are used to select a row. Each row stores 16 bits starting at address (A6, A5, A4, 0, 0, 0, 0). When an address is applied to the ROM, all 16 bits in the selected row are "read out" in parallel on the bit lines. A 16-input multiplexer selects the desired data bit based on the low-order address bits.

By the way, the transistor programming pattern in Figure 15-3 was not chosen at random. It performs a very useful 7-input combinational logic function that would require 35 4-input AND gates to build as a minimal two-level

BOOTSTRAP ROM Primitive though it may seem, owners of the DEC PDP-11 minicomputer (circa 1970) made use of discrete ROM technology similar to Figures 15-2 and 15-3 in the M792 32×16 "bootstrap ROM module." Instead of an NMOS transistor, it used a bipolar diode between the word line and the bit line at each intersection that stores a 1. The module was shipped with 512 diodes soldered in place, and owners programmed it by clipping out the diode at each location where a 0 was to be stored!

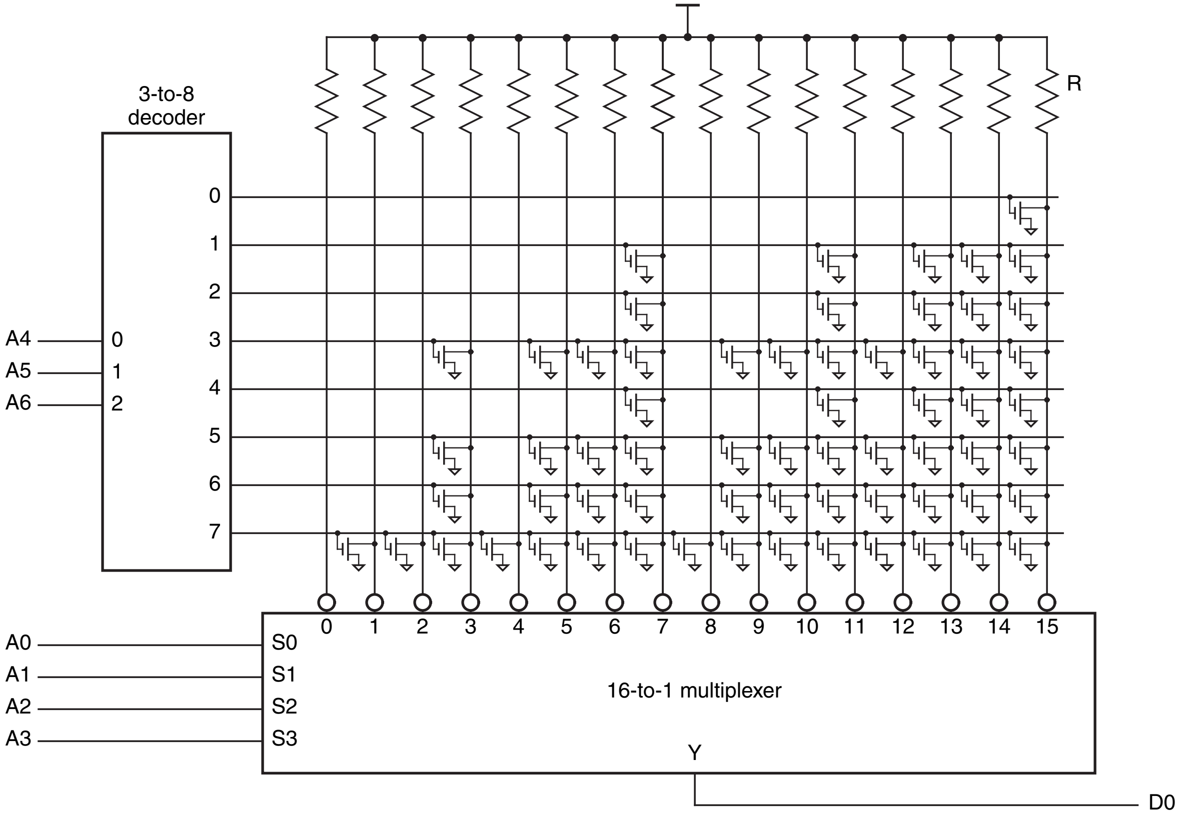


Figure 15-3 Internal structure of a 128 × 1 ROM using two-dimensional decoding.

AND-OR circuit (see Exercise 15.6). The ROM version of this function and ones like it could actually save a fair amount of engineering effort and space compared to a gate-level design.

Two-dimensional decoding allows a 128 × 1 ROM to be built with a 3-to-8 decoder and a 16-input multiplexer (whose complexity is comparable to that of a 4-to-16 decoder). A 1M × 1 ROM could be built with a 10-to-1024 decoder and a 1024-input multiplexer—not easy, but a lot simpler than the one-dimensional alternative.

Besides reducing decoding complexity, two-dimensional decoding has one other benefit—it leads to a chip whose physical dimensions are close to square, important for chip fabrication and packaging. A chip with a 1M × 1 physical array would be *very* long and skinny, and could not be built economically.

In ROMs with multiple data outputs, the storage arrays corresponding to each data output may be made narrower in order to achieve an overall chip layout that is closer to square. For example, Figure 15-4 shows the possible layout of a 32K × 8 ROM chip.

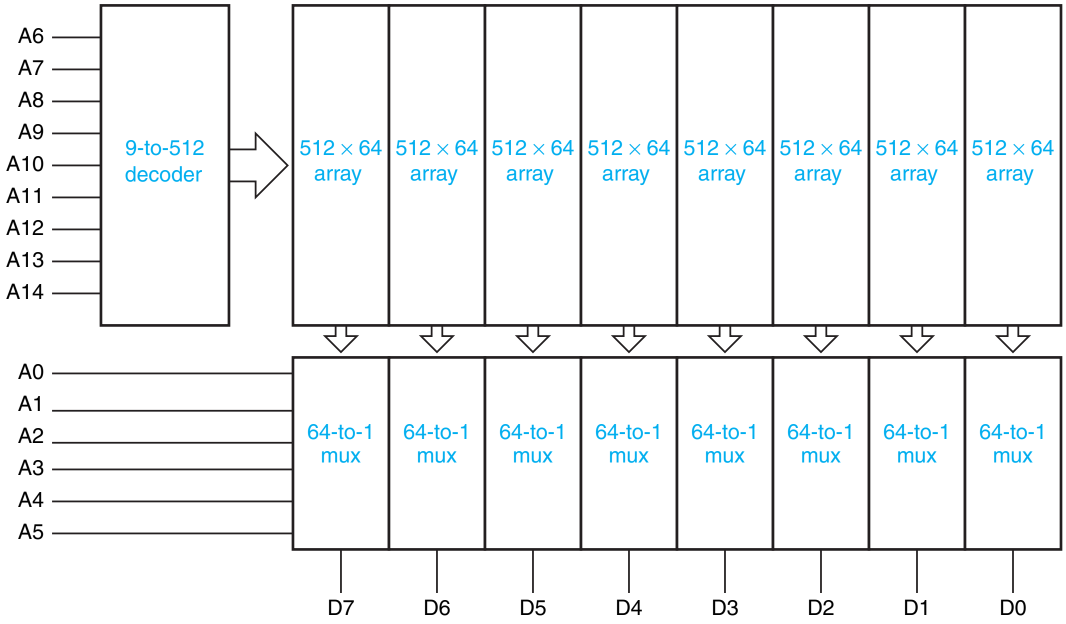


Figure 15-4 Possible layout of a 32K × 8 ROM.

15.1.3 Commercial ROM Types

Unless you visit the Computer History Museum in Mountain View, CA, you won't find any ROM modules built with discrete transistors or diodes. A modern ROM is fabricated as a single IC chip; one that stores 4 gigabits (2^{32} bits) can be purchased for under $5. Various methods have been used to "program" the information stored in a ROM, as discussed below and summarized in Table 15-1.

mask-programmable ROM

mask ROM

mask

Most of the early integrated-circuit ROMs were *mask-programmable ROMs* (or, simply, *mask ROMs*). A mask ROM is programmed by the pattern of connections and no-connections in one of the *masks* used in the IC manufacturing process. To program or write information into the ROM, the customer would give the manufacturer a listing of the desired ROM contents on a disk or other medium. The manufacturer would use this information to create one or more customized masks to manufacture ROMs with the required pattern. Because of mask costs and the four-week delay typically required to obtain programmed chips, mask ROMs were used only in very high-volume applications. For low-volume applications there were more cost-effective choices, discussed next.

programmable read-only memory (PROM)

PROM programmer

A *programmable read-only memory (PROM)* is similar to a mask ROM, except that the customer could store data values (i.e., "program the PROM") in just a few minutes using a *PROM programmer*. A PROM chip is manufactured

Table 15-1 Commercial ROM types.

Type	***Technology***	***Read cycle***	***Write cycle***	***Comments***
Mask ROM	NMOS, CMOS	10–200 ns	4 weeks	Write once; low power
Mask ROM	Bipolar	< 100 ns	4 weeks	Write once; high power; low density
PROM	Bipolar	< 100 ns	10–50 µs/byte	Write once; high power; no mask charge
EPROM	NMOS, CMOS	25–200 ns	10–50 µs/byte	Reusable; low power; no mask charge
EEPROM	NMOS+CMOS	50–200 ns	10–50 µs/byte	10,000–100,000 writes/location limit
NAND flash	NMOS+CMOS	50–200 µs/page	10–50 µs/page	50,000–1,000,000 writes/page limit

with all of its diodes or transistors "connected." This corresponds to having all bits at a particular value, typically 1. The PROM programmer was used to set desired bits to the opposite value. In bipolar technology, this was done by vaporizing tiny *fusible links* inside the PROM corresponding to each bit.

fusible link

Introduced later, an *erasable programmable read-only memory (EPROM)* could be programmed like a PROM, but could also be "erased" to the all-1s state by exposing it to ultraviolet light. No, the light does not cause fuses to grow back! Rather, EPROMs use a different technology, called "floating-gate MOS."

erasable programmable read-only memory (EPROM)

As shown in Figure 15-5, an EPROM has a *floating-gate MOS transistor* at every bit location. Each transistor has two gates. The "floating" gate is not connected to anything and is surrounded by extremely high-impedance insulating material. To program an EPROM, the programmer applies a high voltage to the nonfloating gate at each bit location where a 0 is to be stored. This causes a tem-

floating-gate MOS transistor

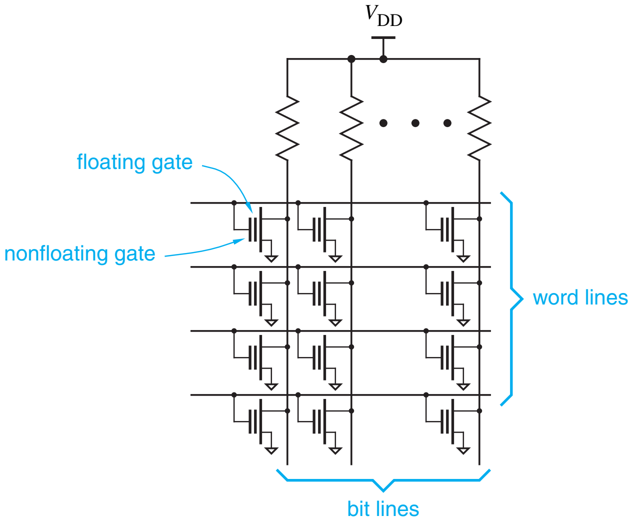


Figure 15-5 Storage matrix in an EPROM using floating-gate MOS transistors (NOR architecture).

porary breakdown in the insulating material and allows a negative charge to accumulate on the floating gate. When the high voltage is removed, the negative charge remains. During subsequent read operations, the negative charge prevents the MOS transistor from turning on when it is selected.

EPROM erasing

one-time programmable (OTP) ROM

Early EPROM manufacturers guaranteed that a properly programmed bit would retain 70% of its charge for at least 10 years, even if the part was stored at 125°C, so EPROMs definitely fell into the category of "nonvolatile memory." However, they could also be erased. The insulating material surrounding the floating gate becomes slightly conductive if it is exposed to ultraviolet light with a certain wavelength. Thus, EPROMs could be erased by exposing the chips to ultraviolet light, typically for 5–20 minutes, when the chip was housed in a package with a transparent quartz lid. Less expensive *one-time programmable (OTP)* versions of these devices were also offered without the quartz lid.

electrically erasable programmable read-only memory (EEPROM)

flash EPROM

flash memory

An *electrically erasable programmable read-only memory (EEPROM)* is like an EPROM, except that individual stored bits may be erased electrically. The floating gates in an EEPROM are surrounded by a much thinner insulating layer and can be erased by applying a voltage of the opposite polarity as the charging voltage to the nonfloating gate. Large EEPROMs (1 Mbit or larger) can be erased only in fixed-size blocks of 128 Kbits to 8 Mbits (16 Kbytes to 1 Mbyte). These memories are called *flash EPROMs* or *flash memories*, because an entire block can be erased "in a flash," like the flash of a camera. The last flash EPROMs in Table 15-1 use a "NAND architecture" internally that brings benefits and limitations, as we'll soon discuss.

As noted in the table, writing an EEPROM location takes much longer than reading it, so an EEPROM is no substitute for the volatile read/write memories discussed later in this chapter. Also, the insulating layer can be worn out by repeated programming operations. As a result, EEPROMs can be reprogrammed only a limited number of times, typically 10,000 to 100,000 times per location; that's a second reason they're no substitute for read/write memory.

EEPROMs are quite suitable for storing information that doesn't change very often, such as the default configuration data and bootstrap programs for computers large and small, or the application software for the embedded processors in all sorts of equipment. On the other hand, when flash memory is used in a computer file system, where some files may be rewritten very frequently, special methods must be used to avoid "wearing out" some locations; we'll say more about that later.

NOR architecture

NAND architecture

The arrangement of transistors in Figure 15-5 is called a *NOR architecture,* because any of the transistors in a column can pull their bit line low, reminiscent of the parallel arrangement of NMOS transistors in a NOR gate. In the mid-1990s, the industry sought ways to build higher density EEPROMs for new applications like digital-camera memories and ultimately high-capacity "solid-state disks" (SSDs) to replace mechanical, magnetic disc drives, and they turned to another transistor arrangement called *NAND architecture*.

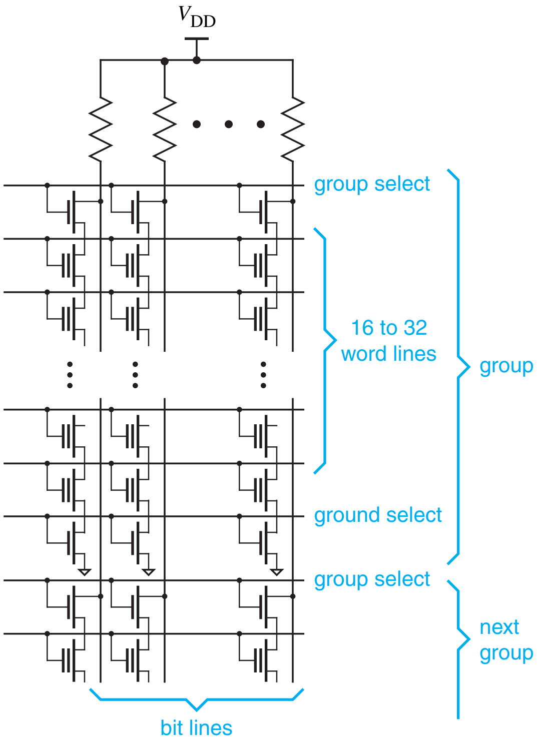


Figure 15-6
NAND architecture for flash memory.

As shown in Figure 15-6, the NAND architecture does not have a ground connection for every bit of storage. Instead, a group of transistors are connected in series, as in a NAND gate, with only the last connected to ground; all must be "on" to pull their bit line low. Typically, there are 16 to 32 transistors in series, and omitting most of the ground connections allows them to be packed more closely together. Compared to an array of NOR cells, a NAND array may be 40% smaller. Note that a complete memory chip will have more groups of 16 to 32 words below the topmost group shown in Figure 15-6, connected to the same bit lines and using the same circuitry to read the values placed on the bit lines.

In an NAND memory, the transistor thresholds and programming levels are set up so that a transistor whose word line is HIGH will be on regardless of whether it stores a 1 or 0. A transistor whose word line is LOW will be off or on depending on whether or not a charge has been stored on its floating gate. Thus, a word (row) is read by setting the group and ground select lines HIGH, and setting all of the word lines HIGH except for the desired word, whose word line is set LOW. Each long column of 16 to 32 transistors in series will pass current or not, depending on the value of its stored bit for the selected word.

The higher density of NAND memory comes at a price in performance, in particular, access time. A row in a NOR memory array can be read out fairly quickly, typically within tens of nanoseconds. In a NAND array, the current that flows in a column is an order of magnitude lower than with NOR, and it can only be sensed reliably by integrating the current (charge transfer) over a relatively long time, on the order of microseconds. Thus, NAND memories are unsuitable for providing random access to instructions or data in a microprocessor system, which requires access times in the tens of nanoseconds.

However, on-chip memory arrays are big, and NAND arrays are still bigger and they can access a lot of data in parallel. So, manufacturers of NAND memory have targeted applications that benefit from relatively fast access to large chunks of data, rather than fast word-by-word random access. It's no surprise that this characterizes NAND memory's most popular applications, including photo storage in digital cameras, program and data storage in notebooks and smartphones, and SSDs in larger computers. When programs or data must be accessed randomly in these applications (for example, when a program is actually invoked and is running), it is first copied into volatile, read/write random-access memory.

The difference between NAND and NOR memory is often described in terms of their external interfaces, which *are* quite different. But the difference between those interfaces has been driven by their different applications, rather than their internal array architectures, as we'll see in the next subsections.

15.1.4 Parallel-ROM Interfaces

As it executes, a microprocessor program specifies a new address on each instruction cycle, potentially a "random" one as it jumps around in the code. So, for a ROM to support direct execution of a program, it needs a simple "parallel" interface based on the structure we showed in Figure 15-1 on page 814. This was in fact the case for most ROMs as they evolved over the years as described in the preceding subsection, prior to the emergence of NAND memory and its applications.

Parallel ROMs are still used today. They use NOR arrays internally and many have the type of external interface described in this subsection, typically an 8-bit data bus and an address bus that's wide enough to receive all of the address bits in parallel. Thus, "legacy" parallel EEPROMs from 32K × 8 to 512K × 8 have the logic symbols shown in Figure 15-7.

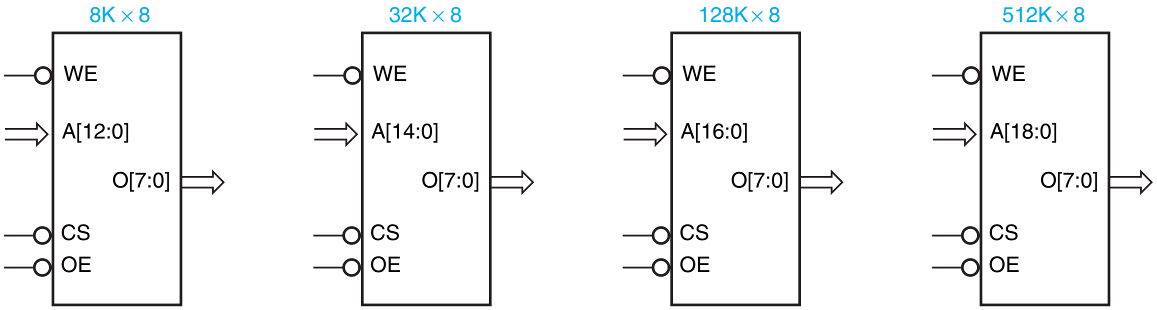


Figure 15-7 Logic symbols for legacy EEPROMs.

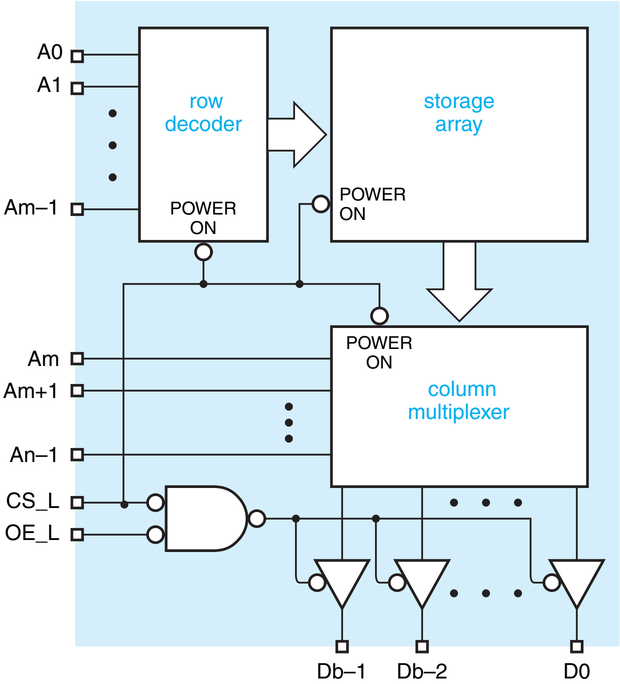


Figure 15-8 Internal ROM structure, showing use of control inputs for reading.

Typical applications can have multiple devices, including ROM, read/write memories, and input/output ports, connected to a three-state bus, where only one device drives the bus at a time. Each device typically has a *chip-select* (*CS*) *input* like the ones in Figure 15-7 which must be asserted to allow it access the bus. An *output-enable* (*OE*) *input* must be asserted to allow it to drive the bus, and a *write-enable* (*WE*) *input* must be asserted to load data from the bus; WE is used in EEPROMs only during programming operations. Figure 15-8 shows the internal structure and logical model of CS and OE use for a typical ROM.

chip-select (CS) input

output-enable (OE) input

write-enable (WE) input

*15.1.5 Parallel-ROM Timing

Figure 15-9 shows typical ROM timing for read operations, including the following parameters:

t_{AA} *Access time from address.* The access time from address of a ROM is the propagation delay from stable address inputs to valid data outputs. A phrase like “100-ns ROM” is usually referring to this parameter.

t_{AA}

access time from address

t_{ACS} *Access time from chip select.* The access time from chip select of a ROM is the propagation delay from the time CS is asserted until the data outputs are valid. In some chips, this is longer than the access time from address, because the chip takes a little while to “power up.” In others, this time is shorter because CS controls only output enabling.

t_{ACS}

access time from chip select

*Throughout this book, optional sections are marked with an asterisk.

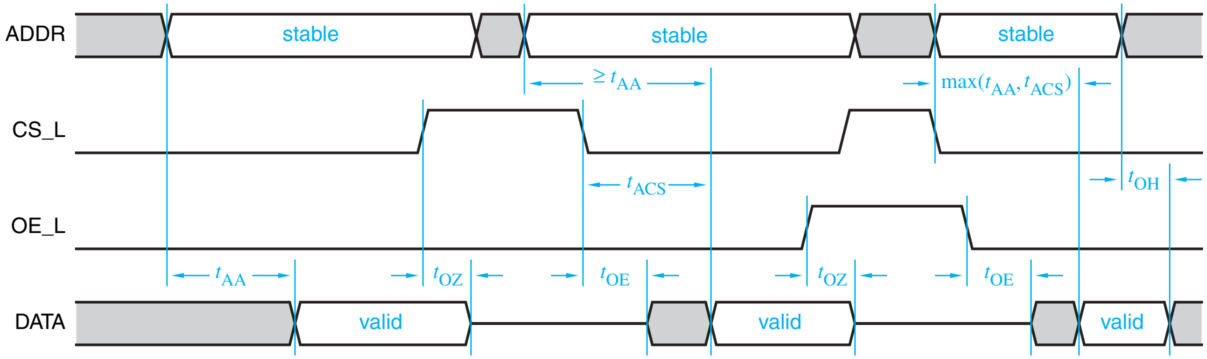


Figure 15-9 ROM timing.

t_{OE}
output-enable time

t_{OE} *Output-enable time.* This parameter is usually much shorter than access time. The output-enable time of a ROM is the propagation delay from OE and CS both asserted until the three-state output drivers have left the Hi-Z state. Depending on whether the address inputs have been stable long enough, the output data may or may not be valid at that point.

t_{OZ}
output-disable time

t_{OZ} *Output-disable time.* The output-disable time of a ROM is the propagation delay from the time OE or CS is negated until the three-state output drivers have entered the Hi-Z state.

t_{OH}
output-hold time

t_{OH} *Output-hold time.* The output-hold time of a ROM is the length of time that the outputs remain valid after a change in the address inputs, or after OE or CS is negated.

As with other components, the manufacturer specifies maximum and, sometimes, typical values for all timing parameters. Usually, minimum values are also specified for t_{OE} and t_{OH}. The minimum value of t_{OH} is usually specified to be 0; that is, the minimum combinational-logic delay through the ROM is 0.

power-down input

standby mode
active mode

As we've described it so far, a CS input is no more than a second output-enable input that is ANDed with OE to enable the three-state outputs. However, in many ROMs, CS also serves as a *power-down input*. When CS is negated, power is removed from the ROM's internal decoders, drivers, and multiplexers. In this *standby mode* of operation, a typical ROM consumes less than 10% of the power it uses in *active mode* with CS asserted.

The largest parallel-interface ROMs store only 1 Mbyte, tiny by today's standards. Although a parallel interface is needed, for example, to execute programs on a microprocessor, it's more economical to use a larger NAND flash memory (described next) for nonvolatile program storage, and transfer programs as needed into read/write memory for execution.

ROMS WITH SERIAL INTERFACES

Besides “legacy” NOR ROMs with parallel interfaces, there are also similarly sized NOR ROMs with 2- or 3-wire serial interfaces for specialized applications, such as downloading programming information into FPGAs. Because the IC die for such a small amount of memory is so tiny, and few pads are needed for the serial interface and power, the chip is small and cheap and can be housed in a package no larger than your little fingernail, making it quite convenient for these applications

15.1.6 Byte-Serial Interfaces for NAND Flash Memories

Because of their slow access times, NAND memories are designed to read, write, and erase large quantities of data during one access interval, with the help of internal registers for temporary storage. Although the internal access is slow, the data can be transferred between the internal registers and the external interface at a very high clock speeds, one byte at a time.

Before we giving details of the external interface, we need to describe the internal organization of a typical flash memory. The smallest unit of storage is called a page and is typically about 512 bytes to 16 Kbytes—the larger page sizes appear in newer, larger memories. Pages are grouped into blocks, typically with 64 to 128 pages per block. And an entire chip may have 2K to 32K or more blocks. These definitions and concepts are illustrated in Figure 15-10.

Read operations are slow, in the range of 10 to 50 µsec per page, depending on the particular chip. Write operations are even slower, in the range of 300 to 700 µsec per page. A page must be erased at some time before it can be written, and erase operations are the slowest. Erasing is done an entire block at a time and its duration is in the range of 1 to 3 ms.

The internal structure and bus interface for a typical flash device are shown in Figure 15-11. The interface is quite simple, as commands, addresses, and data are transferred across the interface using the DQ bus just 8 bits at a time. But it’s

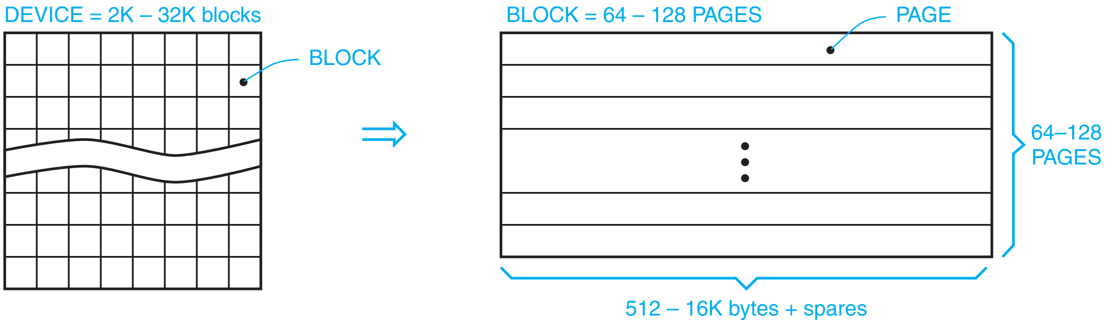


Figure 15-10 NAND-memory block and page structure.

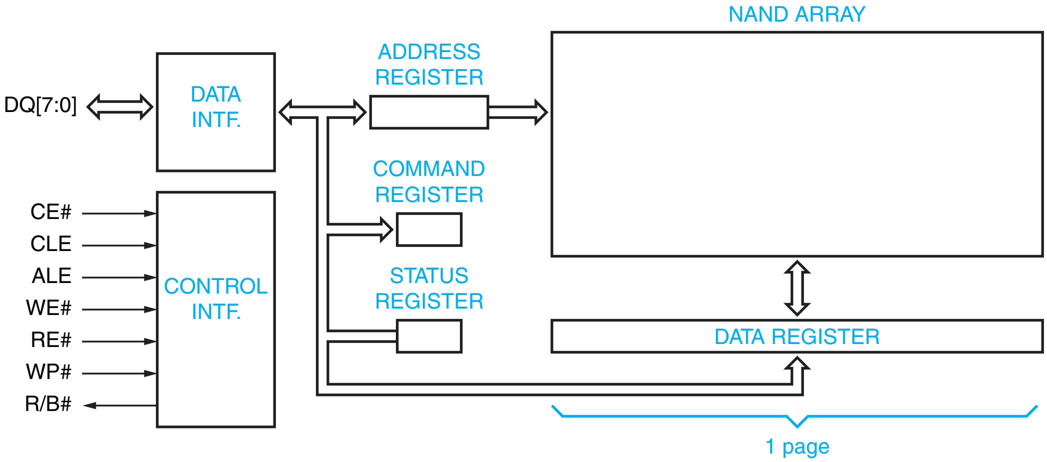


Figure 15-11 NAND byte-serial device bus interface.

speedy; in read operations, for example, once a page has been read into the on-chip internal register, the data can be read as quickly as one byte every 20 ns.

The interface signals are as follows; they use the flash industry's standard designation "#" for an active-low signal:

DQ[7:0] *Data input and output bus.*

CE# *Chip Enable*, must be asserted for other inputs to be used.

CLE *Command Latch Enable*, asserted to write the command register.

ALE *Address Latch Enable*, asserted to write the address register.

WE# *Write Enable*, asserted to write registers or data.

RE# *Read Enable*, asserted to read data or the status register.

WP# *Write Protect*, disables program and erase operations while asserted.

R/B# *Ready/Busy*, asserted by the device when ready.

All of the signals are device inputs, except R/B# which is an output, and DQ[7:0] which is bidirectional (three-state). To be recognized, all of the control inputs require the chip enable CE# to be asserted, which we won't mention further.

So, where are the address signals? There are none. Addresses as well as commands are transferred across the DQ bus at the beginning of an operation. Figure 15-12 shows timing for a typical read operation. It begins with the system placing a command byte value of 00h (hexadecimal) on the DQ bus and asserting CLE and WE#. This tells the device that the address will follow. Five address bytes are then written in sequence, with ALE and WE# asserted for each one.

The first two address bytes specify a "column" address within a page, often but not always 0 (we'll come back to this). So, the page size can be up to 64K bytes, depending on the device; for smaller page sizes, the high order column-

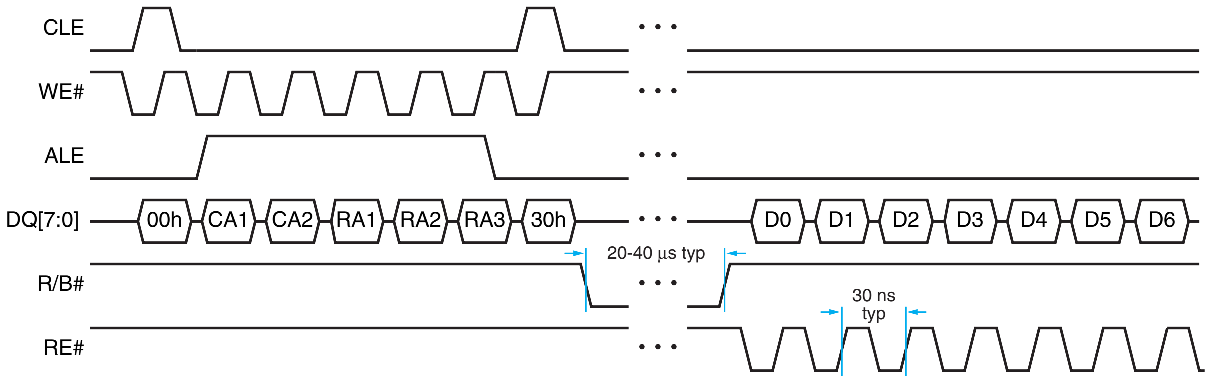


Figure 15-12 NAND page-read timing on the byte-serial bus.

address bits are set to 0. The next three bytes specify the page number within the overall memory array, also known as the "row." So there can be up to 2^{24} pages (rows), again depending on the device, and the unused high-order page-number bits are set to 0.

After writing the address, the system issues the "page-read" command by placing 30h on the DQ bus and asserting CLE and WE#. This triggers an internal state machine that starts the internal read operation for the selected page and eventually transfers the entire contents of the page into the device's internal data register, which has the same width as the page. As the internal operation begins, the device negates R/B#. It reasserts R/B# when the read operation has finished and the entire contents of the page has been transferred to the on-chip internal data register. This is typically tens of microseconds later.

So, there is no "fixed" access time for read operations from a NAND memory. Instead, it's up to the system to monitor the memory's R/B# status output. Once R/B# is asserted, the system may read bytes serially from the internal data register, in order, one byte at a time, starting at the column address that was specified at the beginning of the command. That's why the column address is often set to 0, so the entire page can be read. The individual bytes are read as shown in the timing diagram, using the RE# control signal.

Write operations use the same interface, but two distinct operations are required: erasing and programming. In *programming*, new data is written into an entire page. Moreover, programming can only change a 1 bit to a 0, not the reverse. Therefore, a "program-page" operation must be preceded at some point

programming

HAVE ANOTHER BYTE

Some NAND flash memories have a 16-bit version of the DQ bus. The wider data bus allows higher-bandwidth data transfers (twice as many bits per cycle) or relaxed timing or a combination of both. The high-order byte of the DQ bus is used only for data; commands and addresses use the low-order byte only.

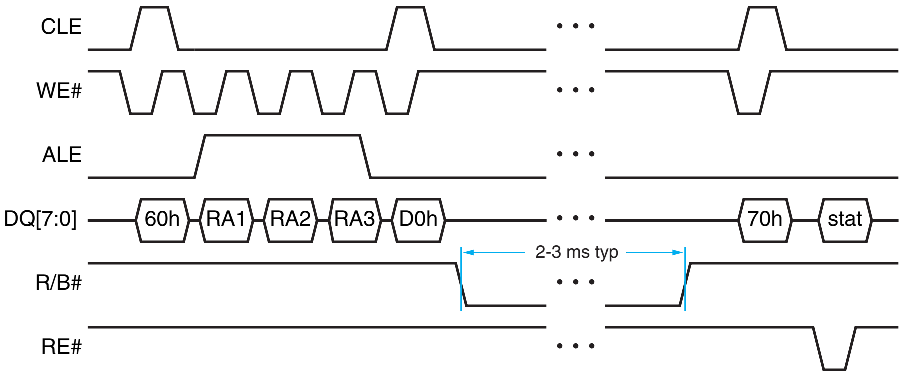


Figure 15-13 NAND block-erase timing on the byte-serial bus.

erasing

by a "block-erase" operation, which sets all of the bits in a block to 1. That's right, *erasing* must be performed on an entire block (multiple pages), which complicates the management of storage in the device, especially for file-system applications and the like.

Figure 15-13 shows the timing for a block-erase operation using the byte-serial interface. The operation begins with the system placing a command byte value of 60h on the DQ bus and asserting CLE and WE#. This tells the device that a 3-byte address will follow. Only a row address is given, and its low-order bits, which specify a page within the larger block, are ignored. Then the system issues the "block-erase" command D0h (not "D'oh!").

The block-erase command triggers an internal state machine in the device that starts the erase operation for the selected block. The state machine performs internal reads to ensure that every bit in the block has been successfully "erased" to 1. As with page reads, the device asserts R/B# when the erase operation has completed, typically a few *milli*seconds after initiation. At that point, the system

RANDOM READS AND WRITES

It's also possible to read one or more bytes starting at a "random" address in a page, once the page has been loaded into the internal data register. The system must issue the command 05h followed by just two (column) address bytes, followed by command E0h. Subsequent reads occur in sequence starting at the specified address. The 05h-E0h command may be issued repeatedly, as long as the page is still in the internal register.

In preparation for programming operations, it's also possible for the system to write bytes into the internal data register starting at a "random" column address, by issuing the 85h command followed by two address bytes and one or more data bytes. However, when the "program" command 10h is ultimately issued, the entire data register is still programmed into the selected page.

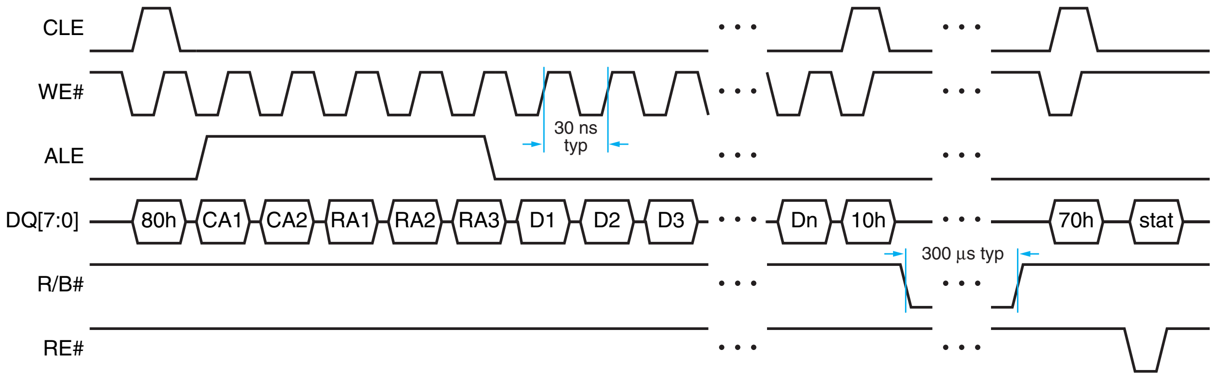


Figure 15-14 NAND program-page timing on the byte-serial bus.

may read the device's internal status register by issuing the "read-status" command 70h; the low-order bit of the returned status byte indicates whether the erase operation was successful.

Once a page has been erased, it may be programmed using the timing shown in Figure 15-14. The beginning of the operation is similar to a page read: the system issues a command, 80h for "program-page," followed by five bytes. The first two are a column address (often 0) and the next three are a page number (row) in the overall memory array. But then it continues with additional writes; each stores a byte from the DQ bus into the internal data register, starting at the column address given in the command and continuing in order.

When all the bytes have been stored in the data register, the system issues the "program" command 10h, which triggers an internal state machine to transfer the contents of the data register into the selected page. Note that regardless of how many bytes were stored into the data register, the entire register is transferred and the entire page is programmed. The device uses its R/B# output to signal completion of the program operation, typically tens of microseconds later. As in an erase operation, the system then reads the status register to determine whether the program-page operation was successful.

SYNCHRONOUS 6X SPEEDUP

Newer NAND flash memories support a "synchronous" timing mode in which the WE# input is replaced with a free-running clock with a period as short as 10 ns. One other signal is added to the interface to control the data transfers, which can now occur on both edges of the clock (a "DDR" interface as discussed in Section 15.4.3). This raises the available data-transfer bandwidth to six times that of the standard "asynchronous" interface with the typical 30-ns minimum cycle times that we discuss next.

*15.1.7 NAND Memory Timing and Access Bandwidth

Figure 15-15 gives more timing details for the NAND-flash byte-serial bus interface operating in its standard, "asynchronous" mode. The following key parameters are used:

t_{CLS}, t_{CLH}

t_{CLS}, t_{CLH} *Command latch setup and hold times.* The command "latch" is really an edge-triggered register that is clocked by the rising edge of WE#, and CLE is a "clock enable" for it. To select or deselect the command register, CLE need not be valid during the entire WE# active pulse (which would be the case with a true latch), only for the specified setup and hold times relative to the rising edge of WE#.

t_{ALS}, t_{ALH}

t_{ALS}, t_{ALH} *Address latch setup and hold times.* Similarly, the address "latch" is really a register, and these are corresponding times for ALE relative to the rising edge of WE#.

t_{DS}, t_{DH}

t_{DS}, t_{DH} *Data setup and hold times.* These times are for data being loaded into the device, whether the data is destined for the command or address register or the internal data register, again relative to WE#.

t_{WP}
write-pulse width

t_{WP} *Write-pulse width.* WE# must be asserted at least this long to reliably select the destination (command, address, or internal data register) and store data into it, and for data-register writes to properly advance the column address to the next byte to be written.

t_{WC}
write-cycle time

t_{WC} *Write-cycle time.* This is the minimum time between successive writes.

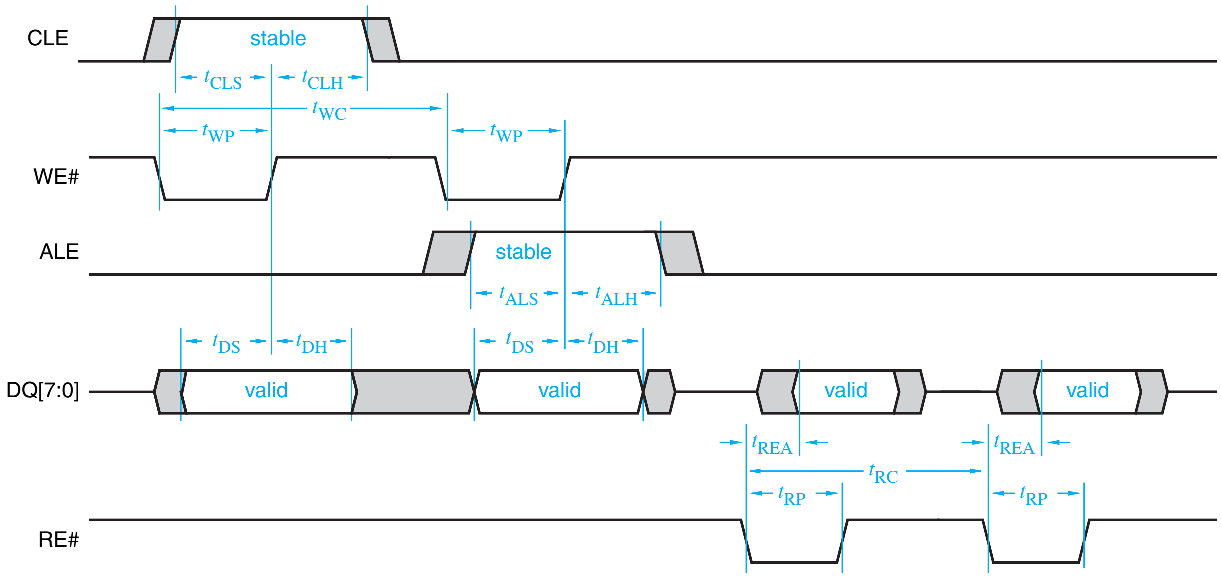


Figure 15-15 NAND byte-serial bus timing parameters.

t_{RP} *Read-pulse width.* RE# must be asserted at least this long to reliably read data from the internal status register or data register and to advance the column address to the next byte to be read.

t_{RP}
read-pulse width

t_{RC} *Read-cycle time.* This is the minimum time between successive read operations.

t_{RC}
read-cycle time

t_{REA} *Read-access time.* This specifies the delay from RE# being asserted until the internal data register or status register becomes valid on the DQ bus.

t_{REA}
read-access time

The minimum read and write cycle times t_{RC} and t_{WC} determine, in part, how quickly data can be loaded into or retrieved from the memory. In a typical device, both specs are the same, for example, 30 ns in an older 1 GB flash device. This implies a peak data transfer rate on the bus of 33 MB/s. Newer devices can increase this rate by as much as a factor of six (see the box on page 829).

The other key determinant of device speed is at a higher level, namely, how quickly pages can be read from and written into the internal NAND array. For example, the device manufacturer specifies a maximum value for t_{R}, the delay from the initiation of a page-read command to the time when the page is available in the internal register and the device has negated R/B#. In newer as well as older devices, this is still on the order of 20–40 µs per page. However, newer devices have larger page sizes (for example 8 Kbytes vs. 2 Kbytes), so their relative speed per byte read can be higher.

Finally, overall bandwidth depends on the access pattern of reads and writes to the device. Programming operations are typically about ten times slower than reads per page, and that's only if there is a page available to be programmed that has already been erased. Storage-management driver software for NAND devices tries to keep a pool of such pages available at all times.

*15.1.8 Storage Management for NAND Memories

To be used effectively in any application, or in some cases even at all, NAND memories require a lot of management compared to NOR-based EEPROM devices. We'll touch upon the most important management areas here.

First of all, the way some people look at it, NAND memories are "broken" from the very start. A NOR memory "just works"—you can program any part of the memory you wish to, perhaps erase and reprogram some areas, and read back everything that you put in, subject to a typical, very small hardware failure rate.

A NAND memory is inherently less reliable, however, because of its higher density and the sensitivity of its read operations. It's impractical to manufacture a high-density NAND memory in which 100% of the bits on 100% of its pages work perfectly at the time it is shipped. Moreover, as the device is used, even more bits will go bad—they'll lose their ability to be read or reprogrammed or both. These problems must be handled as discussed next.

To deal with bad bits on a page, NAND memories are manufactured with spare bytes for each page. For example, what you may think of as a 2-Kbyte page actually contains 2,048 *plus* 64 bytes. All 2,112 bytes are used on every internal operation, and the on-chip data register is 2,112 bytes wide. Low-level driver software uses the extra bytes to manage the page as follows:

- Some of the spare bytes are set up to contain parity bits in an error-correcting code so that when a page is read, errors in one or a few bits can be corrected on the fly.
- If too many bits are bad, the page or the entire block that contains it can be marked as bad, again using the spare bytes to set its status.

Of course, the spare bytes themselves can go bad, so the algorithms for the above operations must be sufficiently robust to handle that possibility, too. And running the management software (particularly error detection and correction) requires software overhead on every access.

Since NAND access occurs one page at a time, error detection and correction are done, of necessity, at the page level. However, permanent failures are infrequent enough that regions of the device are normally marked bad at a higher level, the block level. That is, failures of a page will cause its entire block to be marked bad.

Before a new device is shipped from the factory, it is tested and its bad blocks are marked. The manufacturer guarantees that during the normal life of the device, no more than a certain number of the blocks will go bad, including both the factory-marked ones and ongoing failures. The guaranteed maximum number of bad blocks is typically on the order of 0.2% to 1% of the device total.

The fact that an entire page can go bad means that higher-level applications cannot use NAND storage as a monolithic array of linearly addressable storage locations, like a standard NOR EEPROM or a RAM. Rather, they must acknowledge the device's paged organization and deal with it in much the same way they deal with other block-oriented storage devices that may have bad blocks, like disk drives. Luckily, when NAND storage is used for a file system, the overhead for this kind of mechanism is not particularly burdensome, since file systems have dealt for decades with block-oriented storage that can have bad blocks, by mapping a linear file into a list of arbitrarily addressed blocks.

Aside from error management, there are several other subtleties associated with NAND memory usage. Recall, for example, that erasing occurs at the block level, not the page level. And pages have to be erased before they can be programmed, and erasing takes even longer than programming. So, the management software must try to keep some already-erased pages and blocks available at all times. When one or a few pages (for example in a small file) need to be rewritten, the management software may try to gather and reallocate pages from different files into the same pre-erased block, to most efficiently utilize storage space and erased-page availability. The software must also observe other arcane restrictions, such as the requirement that pages within an erased block be programmed in numerically increasing order of their page numbers.

Another important management activity is "write leveling." Recall that the the individual bits in an EEPROM "wear out" after a certain number of erase and reprogramming cycles, on the order of 100,000 for typical NAND devices. In a file system application, some files are rewritten a lot more often that others. The management software for NAND devices keeps track of how many times each block has been erased and reprogrammed, and during write operations it will favor available pages in blocks that have been reprogrammed the least often. Thus, a "hot" file may bounce around all over the device each time it is updated, while a "cold" one may not move at all until the management software figures out that it can move it to a block that's been overused, to give that block a rest!

15.2 Read/Write Memory

The name *read/write memory (RWM)* is given to memory arrays in which we can store and retrieve information at any time. All of the RWMs used in digital systems nowadays are *random-access memories (RAMs)*, which means that the time it takes to read or write a bit is independent of the bit's location in the RAM. From this point of view, ROMs are also random-access memories, but the name "RAM" is generally used only for read/write random-access memories.

read/write memory (RWM)

random-access memory (RAM)

SERIAL-ACCESS MEMORY

Random-access memory can be contrasted with *serial-access memory*, where some location is always immediately accessible, but accessing others take more time.

Some early computers used electromechanical serial-access memory devices, such as delay lines and rotating drums. Instructions and data were stored in a rotating medium with only one location under the "read/write head" at any time. To access a random location, the computer would have to wait until the constant rotation brought the desired location under the head.

In the 1970s, electronic equivalents of serial-access rotating memories were developed, including memories based on charge-coupled devices (CCDs) and others that used magnetic bubbles. Both types of devices were roughly equivalent to very large serial-in, serial-out shift registers with their serial output connected back into the serial input. This connection point was the logical equivalent of a hard disk's "read/write head." To read a particular location, you would clock the shift register until the desired bit appeared at the serial output, and to write the location, you would substitute the desired new value at the serial input.

Although they offered higher density (more bits) than DRAMs when they were introduced, CCD and magnetic-bubble memories never gained much commercial acceptance. One reason for this was the enormous inconvenience of serial access. Another was that they were never more than a couple years ahead of DRAMs in the densities that they could achieve.

static RAM (SRAM)

dynamic RAM (DRAM)

In a *static RAM (SRAM)* ("S-ram"), once a word is written at a location, it remains stored as long as power is applied to the chip, unless the same location is written again. In a *dynamic RAM (DRAM)* ("D-ram"), the data stored at each location must be refreshed periodically by reading it and then writing it back again, or else it disappears. We'll discuss both types in this chapter.

volatile memory

nonvolatile memory

Most RAMs lose their memory when power is removed; they are a form of *volatile memory*. Some RAMs retain their memory even when power is removed; they are called *nonvolatile memory*. Examples of nonvolatile RAMs are old-style magnetic core memories and modern CMOS static memories in an extra-large package that includes a lithium battery with a 10-year lifetime.

15.3 Static RAM

asynchronous SRAM

The SRAMs in the next four subsections are often called *asynchronous SRAMs* to distinguish them from a newer style discussed in Section 15.3.5. The newer, "synchronous SRAMs" reference their control and data signals to a free-running clock, while the ones discussed next do not.

15.3.1 Static-RAM Inputs and Outputs

write-enable (WE) input

Like a parallel ROM, a RAM has address and control inputs and data outputs, but it also has data inputs. The inputs and outputs of a simple $2^n \times b$-bit static RAM are shown in Figure 15-16. The control inputs are comparable to those of a ROM, with the addition of a *write-enable (WE) input*. When WE is asserted, the data inputs are written into the selected memory location.

The memory locations in a static RAM behave like D latches, rather than edge-triggered D flip-flops. This means that whenever the WE input is asserted, the latches for the selected memory location are "open" (or "transparent"), and input data flows into and through the latch. The actual value stored is whatever is present when the latch closes.

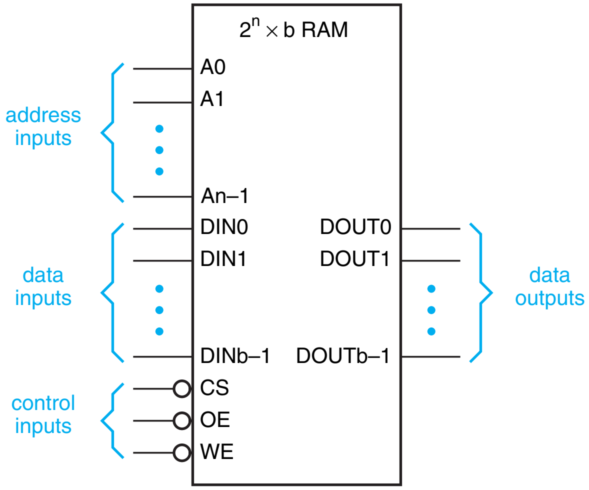


Figure 15-16
Basic structure of a $2^n \times b$ RAM.

Static RAM normally has just two defined access operations:

Read An address is placed on the address inputs while CS and OE are asserted. The latch outputs for the selected memory location are delivered to DOUT.

Write An address is placed on the address inputs and a data word is placed on DIN; then CS and WE are asserted. The latches in the selected memory location open, and the input word is stored.

A certain amount of care is needed when accessing SRAM, because it is possible to inadvertently "clobber" one or more other locations while writing to a selected one, if the SRAM's timing requirements are not met. The following subsection gives details on SRAM internal structure to show why this is so, and the next one explains the actual timing behavior and requirements.

15.3.2 Static-RAM Internal Structure

SRAM cell

Each bit of memory (or *SRAM cell*) in a static RAM has the same functional behavior as the circuit in Figure 15-17. The storage device in each cell is a D latch. When a cell's SEL_L input is asserted, the stored data is placed on the cell's output, which is connected to a bit line. When both SEL_L and WR_L are asserted, the latch is open and a new data bit is stored.

SRAM cells are combined in an array with additional control logic to form a complete static RAM, as shown for an 8 × 4 SRAM in Figure 15-18 on the next page. As in a simple ROM, a decoder on the address lines selects a particular row of the SRAM to be accessed at any time.

Although Figure 15-18 is a somewhat simplified model of internal SRAM structure, it accurately portrays several important aspects of SRAM behavior:

- During read operations, the output data is a combinational function of the address inputs, as in a ROM. No harm is done by changing the address lines while the output data bus is enabled. The access time for read operations is specified from the time that the last address input becomes stable.
- During write operations, the input data is stored in *latches*. This means that the data must meet certain setup and hold times with respect to the *trailing* edge of the latch enable signal. That is, the input data at a latch's D input need not be stable at the moment WR_L is asserted internally; it must only be stable a certain time before WR_L is negated.

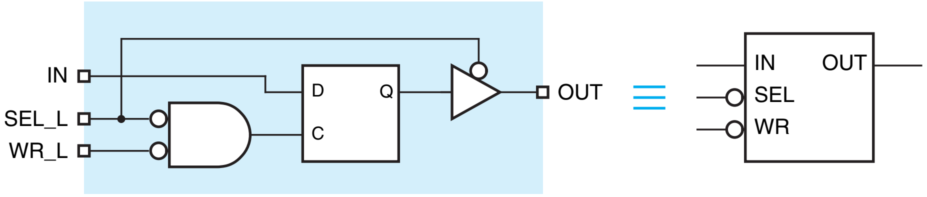


Figure 15-17 Functional behavior of a static-RAM cell.

- During write operations, the address inputs must be stable for a certain setup time *before* WR_L is asserted internally and for a hold time after WR_L is negated. Otherwise, data may be "sprayed" all over the array because of the glitches that may appear on the SEL_L lines when the address inputs of the decoder are changing.

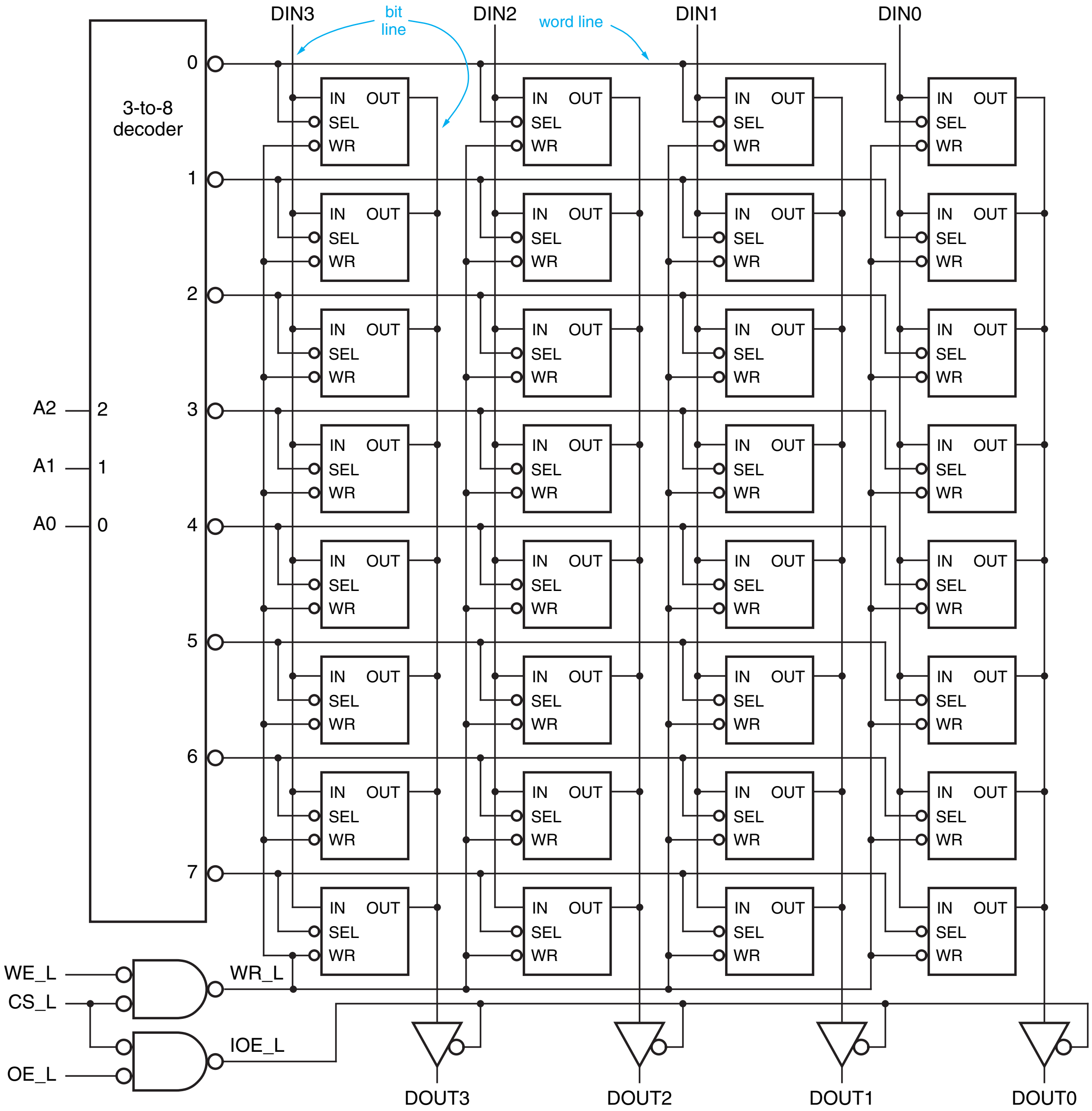


Figure 15-18 Internal structure of an 8 × 4 static RAM.

- Internally, WR_L is asserted only when both CS_L and WE_L are asserted. Therefore, a *write cycle* begins when both CS_L and WE_L are asserted and ends when either is negated. Setup and hold times for address and data are specified with respect to these events.

write cycle

*15.3.3 Static-RAM Timing

Figure 15-19 shows the timing parameters that are typically specified for read operations in a static RAM; they are described below:

t_{AA} *Access time from address.* Assuming that the OE and CS inputs are already asserted, or will be soon enough not to make a difference, this is how long it takes to get stable output data after a change in address. When designers talk about a “70-ns SRAM,” they’re usually referring to this number.

t_{AA} *access time from address*

t_{ACS} *Access time from chip select.* Assuming that the address and OE are already stable, or will be soon enough not to make a difference, this is how long it takes to get stable output data after CS is asserted. Often this parameter is identical to t_{AA}, but sometimes it’s longer in SRAMs with a “power-down” mode and shorter in SRAMs without one.

t_{ACS} *access time from chip select*

t_{OE} *Output-enable time.* This is how long it takes for the three-state output buffers to leave the high-impedance state when OE and CS are both asserted. This parameter is normally less than t_{ACS}, so it is possible for the RAM to start accessing data internally before OE is asserted; this feature is used to achieve fast access times while avoiding “bus fighting” in many applications.

t_{OE} *output-enable time*

t_{OZ} *Output-disable time.* This is how long it takes for the three-state output buffers to enter the high-impedance state after OE or CS is negated.

t_{OZ} *output-disable time*

t_{OH} *Output-hold time.* This parameter specifies how long the output data remains valid after a change in the address inputs.

t_{OH} *output-hold time*

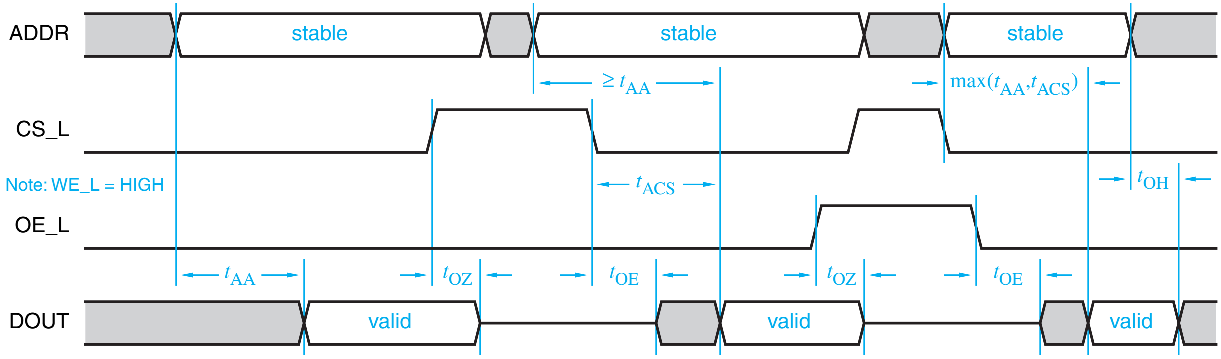


Figure 15-19 Timing parameters for read operations in a static RAM.

If you've been paying attention, you may have noticed that the timing diagram and timing parameters for SRAM read operations are identical to what we discussed for ROM read operations in Section 15.1.4. That's the way it is; when they're not being written, SRAMs can be used just like ROMs. The same is not generally true for DRAMs, as we'll see later.

Timing parameters for write operations are shown in Figure 15-20 and are described below:

t_{AS}
address setup time

t_{AS} *Address setup time before write.* All of the address inputs must be stable at this time before both CS and WE are asserted. Otherwise, the data stored at unpredictable locations may be corrupted.

t_{AH}
address hold time

t_{AH} *Address hold time after write.* Analogous to t_{AS}, all address inputs must be held stable until this time after CS or WE is negated.

t_{CSW}
chip-select setup time

t_{CSW} *Chip-select setup before end of write.* CS must be asserted at least this long before the end of the write cycle in order to select a cell.

t_{WP}
write-pulse width

t_{WP} *Write-pulse width.* WE must be asserted at least this long to reliably latch data into the selected cell.

t_{DS}
data setup time

t_{DS} *Data setup time before end of write.* All of the data inputs must be stable at this time before the write cycle ends. Otherwise, the incorrect data may be latched.

t_{DH}
data hold time

t_{DH} *Data hold time after end of write.* Analogous to t_{DS}, all data inputs must be held stable until this time after the write cycle ends.

WE-controlled write
CS-controlled write

Manufacturers of SRAMs specify two write-cycle types, *WE-controlled* and *CS-controlled*, as shown in the figure. The only difference between these cycles is whether WE or CS is the last to be asserted and the first to be negated when enabling the SRAM's internal write operation.

The write-timing requirements of SRAMs could be relaxed somewhat if, instead of using latches, the cells contained edge-triggered D flip-flops with a common clock input and enable inputs controlled by SEL and WR. However, this just isn't done, because it would at least double the chip area of each cell,

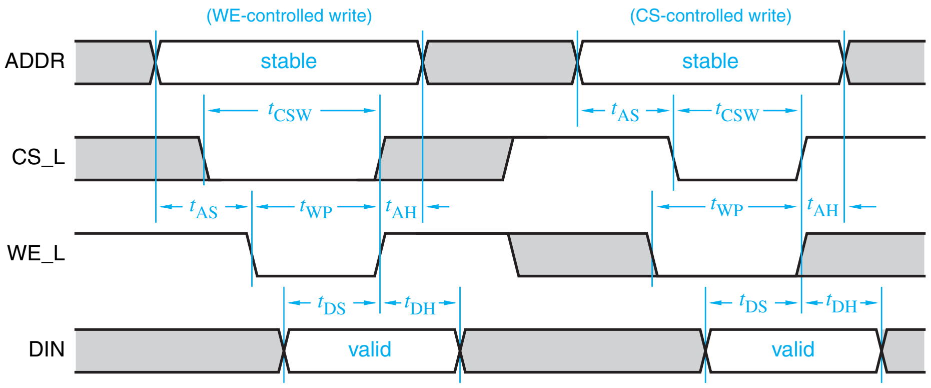


Figure 15-20 Timing parameters for write operations in a static RAM.

since a D flip-flop has roughly the same chip area as two latches. Thus, a logic designer who uses asynchronous SRAM is left to reconcile its latch-type timing behavior with the edge-triggered register and state-machine timing used elsewhere in a system. An alternative is to use a “synchronous SRAM,” as we’ll discuss later, in Section 15.3.5.

A large SRAM does not contain a physical array whose dimensions equal the logical dimensions of the memory. As in a ROM, the SRAM cells are laid out in an almost square array, and an entire row is read internally during read operations. For example, the on-chip layout of a 32K × 8 SRAM might be very similar to that of a 32K × 8 ROM shown in Figure 15-4 on page 818. During read operations, column multiplexers pass the required data bits to the output data bus, as specified by a subset of the address bits (A5–A0 in the ROM example). For write operations, the write-enable circuitry is designed so that only one column in each subarray is enabled, as determined by the same subset of the address bits.

*15.3.4 Standard Asynchronous SRAMs

Asychronous SRAMs are produced in many sizes and speeds, up to 64 Mbits (4M × 16 bits) with access times as fast as 55 ns. Faster access times of 8 ns can be obtained from smaller 256K × 16-bit SRAMs.

Figure 15-21 shows generic logic symbols for standalone SRAM devices ranging in size from 8K × 8 to 512K × 8. All of these devices have bidirectional data buses—that is, they use the same data pins for both reading and writing. This necessitates a slight change in their internal control logic to automatically disable the output buffer whenever WE_L is asserted, even if OE_L is asserted. However, the timing parameters and requirements for read and write operations are almost identical to what we described in the preceding subsection.

Standalone SRAMs like the ones shown were often used to store data in small microprocessor systems, often in “embedded” applications. However, as chip densities increased, the use of standalone SRAMs decreased, because a modest amount of SRAM could be integrated on the same chip with the microprocessor and its input/output interfaces, in a so-called “system on a chip.” And

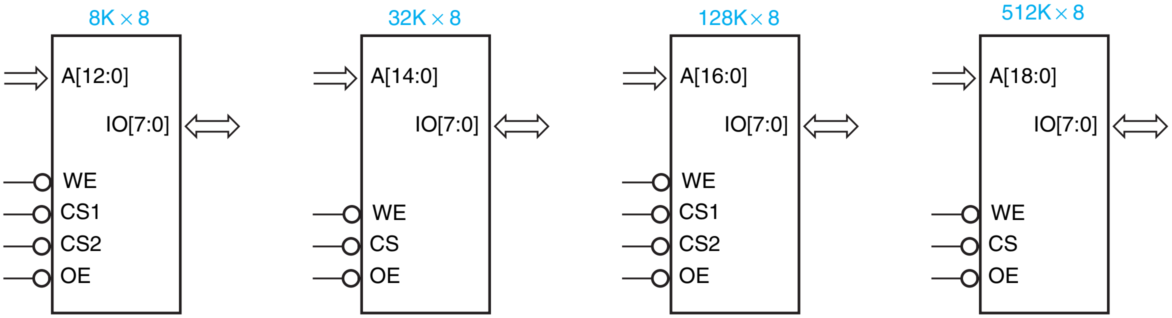


Figure 15-21 Logic symbols for legacy asynchronous SRAMs.

if a lot more memory was needed, as in a general-purpose computer, standalone DRAMs (discussed in Section 15.4) were and are used because their density is greater and their cost per bit lower.

*15.3.5 Synchronous SRAM

synchronous SRAM (SSRAM)

A newer variety of standalone SRAM chips, called *synchronous SRAM (SSRAM)* ("S-S-ram"), was developed to meet the performance demands of the highest speed SRAM applications, typically in high speed communications and networking. An SSRAM still uses latches in its internal storage array but has a clocked interface for control, address, and data. Because critical timing paths are handled in the SSRAM chip itself, it's much easier to interface with the rest of a system that uses the same clock.

As shown in Figure 15-22, an SSRAM places edge-triggered registers AREG and CREG on its internal signal paths for address and control. As a result, an operation that is set up before the rising edge of the clock is performed internally during a subsequent clock period. Register INREG captures the input data for write operations and, depending on whether the device has "pipelined" or "flow-through" outputs, register OUTREG is or is not provided to hold read data.

late-write SSRAM with flow-through outputs

The first variety of SSRAM to be introduced was the *late-write SSRAM with flow-through outputs*. For a read operation, shown in Figure 15-23(a), the control and address inputs are sampled at the rising edge of the clock, and the internal address register AREG is loaded only if ADS_L is asserted. During the next clock period, the internal SRAM array is accessed and read data is delivered to the device's DIO data-bus pins. The device also supports a burst mode, in which data at a sequence of addresses is read. In this mode, AREG behaves as a

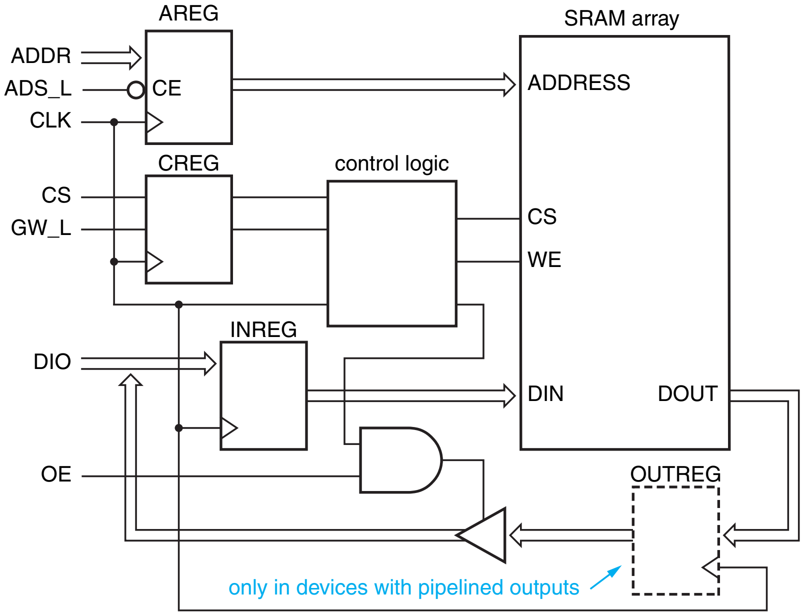


Figure 15-22 Internal structure of a synchronous SRAM.

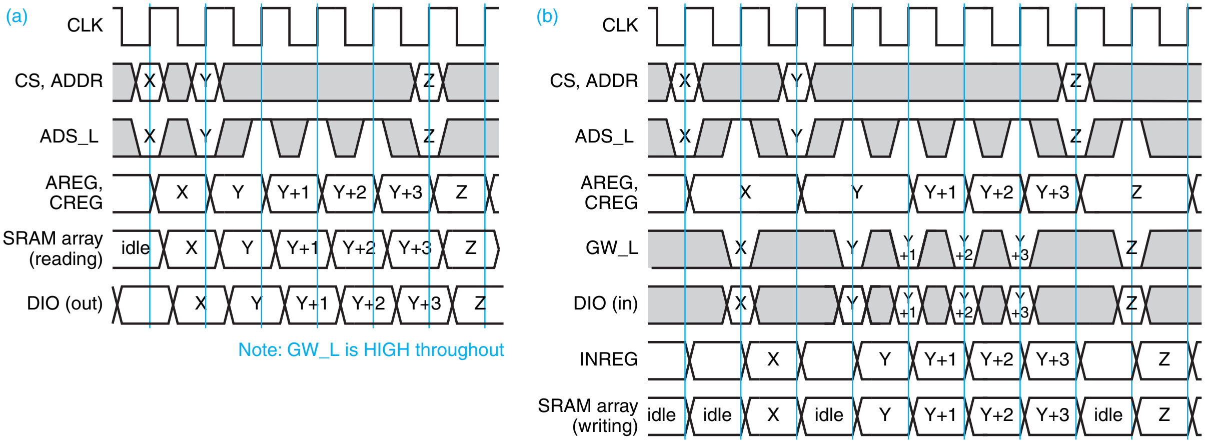


Figure 15-23 Timing behavior for late-write SSRAM with flow-through outputs: (a) read operations; (b) write operations.

counter, eliminating the need to apply a new address at each cycle. (The control signals that support burst mode are not shown in Figures 15-22 or 15-23.)

For a write operation, shown in Figure 15-23(b), the write data is stored temporarily in an on-chip register INREG, which is sampled one clock tick *after* the address register is loaded. Therefore, ADS_L must be inhibited for at least one tick after loading the address, so that the address in AREG is still valid when the write takes place. The write takes place during the clock period following the edge on which the "global write" control signal GW_L is asserted. As with reading, the device has a burst mode where a sequence of addresses can be written without supplying a new address.

Note that the "late-write" protocol makes it impossible to write to two different, nonsequential addresses in successive clock periods; the SRAM array is idle for one clock period between writes (except in burst mode). From the point of view of internal chip capabilities, this behavior is not necessary. However, the late-write protocol was designed this way to match the bus protocols of older microprocessors that used these SSRAMs in their external cache subsystems.

late-write SSRAM with pipelined outputs

A *late-write SSRAM with pipelined outputs* is like the previous version, except that a register OUTREG is placed between the SRAM array output and the device output for read operations. As shown in Figure 15-24 on the next page, this delays the read output data at the device pins until the beginning of the next clock period, but it also provides the benefit that the data is now valid for almost the entire clock period. The write cycle behaves the same as with flow-through outputs. Compared to flow-through outputs, pipelined outputs provide much better setup time for the device receiving the read data, and therefore may allow operation at higher clock frequencies.

As we showed in Figure 15-22, conventional SSRAMs share the same pins for both input data and output data. During a given clock period, the data I/O

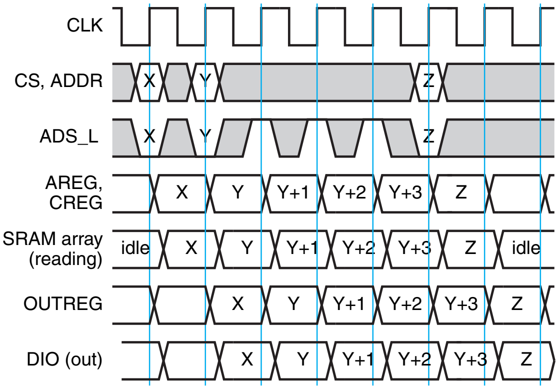


Figure 15-24
Read-timing behavior for late-write SSRAM with pipelined outputs.

pins can be used for reading or writing but not both. If you study the pattern of data-bus and SRAM-array use in both styles of late-write SSRAM, you'll find cases where it's not possible to initiate a read one clock cycle after initiating a write or vice versa, due to resource conflict (see Exercise 15.20). Thus, late-write SSRAMs suffer a *turn-around penalty*, a clock period in which the internal SRAM array must be idle when a read is followed by a write or vice versa.

turn-around penalty

zero-bus-turn-around (ZBT) SSRAM

The turn-around penalty is eliminated in so-called *zero-bus-turn-around (ZBT) SSRAMs*. The timing for a *ZBT SSRAM with flow-through outputs* is shown in Figure 15-25. The type of operation (read or write) is selected by a control signal R/$\overline{\text{W}}$ that is sampled at the same clock edge as the address. Regardless of whether the operation is a read or a write, the DIO bus is used during the next clock period to transfer the read or write data. As a result, there is no data-bus-usage conflict, as long as OE is controlled properly to avoid bus-fighting between successive cycles. However, if a write is followed by a read, both operations would like to use the SRAM array during the same clock period. To avoid

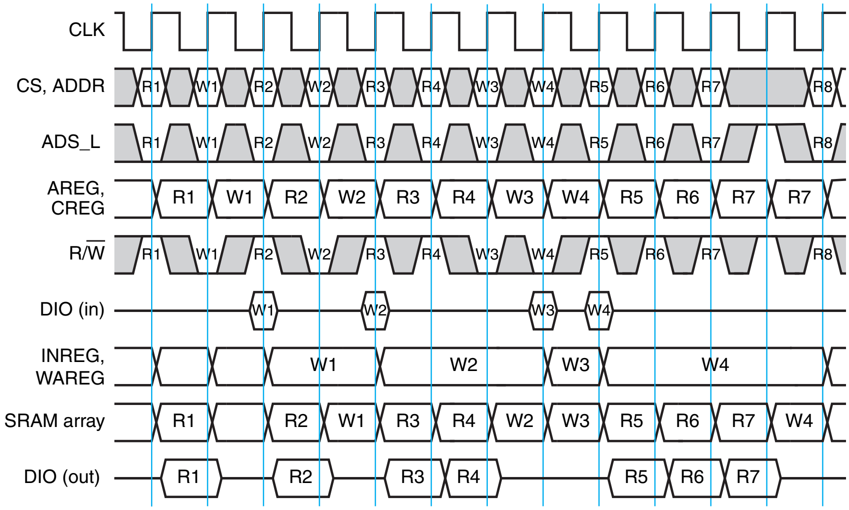


Figure 15-25
Timing behavior for a ZBT SSRAM with flow-through outputs.

this resource conflict, the write operation is deferred until the next available SRAM cycle. This opportunity occurs when either another write operation or no operation is initiated on the address and control lines.

Although a ZBT SSRAM can access the internal SRAM array on every clock cycle, this performance improvement is not without a price. While a write operation is pending, the write address and related information must be stored in another register, WAREG, since AREG is reused by other operations; this may increase the on-chip memory access time slightly since the address source for the array must now go through a multiplexer. More significantly for some applications, a write operation may be deferred indefinitely if it is immediately followed by a continuous series of read operations. This anomaly may require tricky controller design to detect the case where one of these read operations attempts to access the address that was just written, since the value stored in the SRAM array is "stale"!

ZBT SSRAM with pipelined outputs

A *ZBT SSRAM with pipelined outputs* adds OUTREG to the read data path but is otherwise similar to the previous device. In this device, both reads and writes use the DIO bus during the second clock period following the clock edge in which the operation was initiated. As in the previous device, writes to the internal SRAM array are deferred until an available cycle, so that reads can take precedence. Timing is shown in Figure 15-26. As implied by the timing, two levels of internal registers are needed for write address and data, since up to two writes may be deferred while a sequence of reads occurs.

Among the four styles of SSRAM that we described, no single one is the "best." The best SSRAM is the one that best fits the bus protocol and other requirements of the system in which it is used. SSRAM access protocols are very

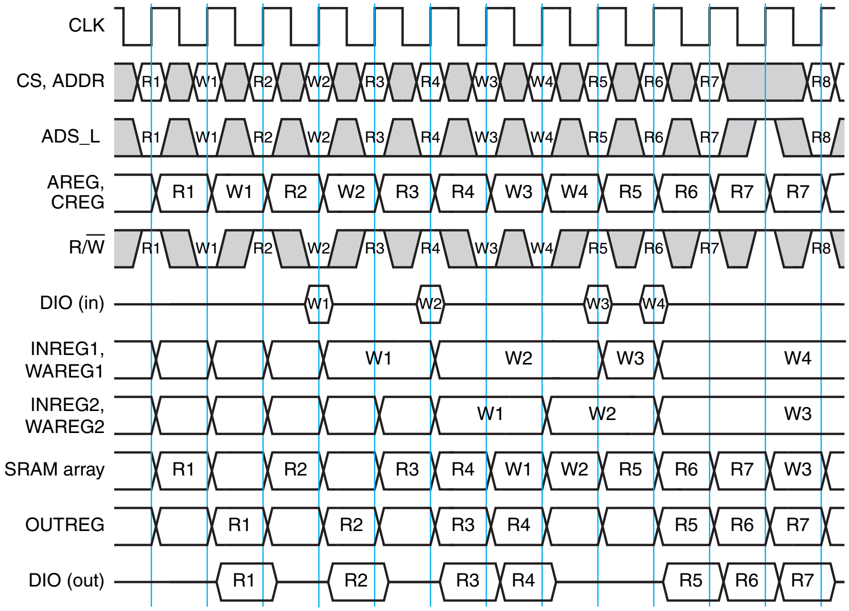

CLK
CS, ADDR
R1 W1 R2 W2 R3 R4 W3 W4 R5 R6 R7 R8
ADS_L
R1 W1 R2 W2 R3 R4 W3 W4 R5 R6 R7 R8
AREG, CREG
R1 W1 R2 W2 R3 R4 W3 W4 R5 R6 R7 R7
R/W
R1 W1 R2 W2 R3 R4 W3 W4 R5 R6 R7 R8
DIO (in)
W1 W2 W3 W4
INREG1, WAREG1
W1 W2 W3 W4
INREG2, WAREG2
W1 W2 W3
SRAM array
R1 R2 R3 R4 W1 W2 R5 R6 R7 W3
OUTREG
R1 R2 R3 R4 R5 R6 R7
DIO (out)
R1 R2 R3 R4 R5 R6 R7

Figure 15-26
Timing behavior for a ZBT SSRAM with pipelined outputs.

beneficial in high-speed systems. For example, address, control, and write inputs can be applied with more-or-less conventional setup and hold times with respect to the system clock, and read data on pipelined output pins is available for almost a complete clock cycle. Very importantly, the designer does not have to worry about the tricky circuits and timing paths that are otherwise needed to enable conventional SRAM latch-style operation.

quad-data-rate (QDR) SSRAM

The latest *quad-data-rate (QDR) SSRAMs* make timing tricky again, by transferring data on both edges of the clock. But they also simplify operations and eliminate the turn-around penalty by using separate input and output buses (hence the claim that they are "four" times as fast). QDR devices were available in 2017 with sizes up to 144 Mbits and clock frequencies as high as 1066 MHz.

SRAM IN FPGAS

While the use of standalone SRAM chips has declined, SRAM usage in FPGAs has soared. Today's FPGAs have built-in blocks of SRAM that the designer can interconnect with other programmable logic on the chip. For example, each Xilinx 7-series FPGA has up to about 2,000 36-Kb "block RAMs" that can be individually configured as *n* words by 1 to 72 bits (e.g., *n*=512 at width 72). To support high-speed operation, each RAM's interface to the rest of the FPGA is fully synchronous. As Xilinx puts it in their documentation, "Nothing happens without a clock."

15.4 Dynamic RAM

The basic memory cell in an SRAM, a D latch, requires four gates in a discrete design, and four to six transistors in a custom-designed SRAM VLSI chip. In order to build RAMs with higher density (more bits per chip), chip designers invented memory cells that use as little as one transistor per bit.

15.4.1 Dynamic-RAM Structure

dynamic RAM (DRAM)

It is not possible to build a bistable element with just one transistor. Instead, the memory cells in a *dynamic RAM (DRAM)* store information on a tiny capacitor that is accessed through a MOS transistor. Figure 15-27 shows the storage cell for one bit of a DRAM, which is accessed by setting the word line to a HIGH voltage. To store a 1, the bit is accessed and a HIGH voltage is placed on the bit

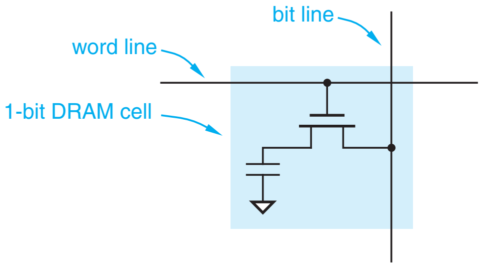


Figure 15-27
Storage cell for one bit in a DRAM.

line, which charges the capacitor through the "on" transistor. To store a 0, a LOW voltage placed on the bit line discharges the capacitor.

precharge

To read a DRAM cell, the bit line is first *precharged* to a voltage halfway between HIGH and LOW, and then the word line is set HIGH. Depending on whether the capacitor voltage is HIGH or LOW, the precharged bit line is pulled slightly higher or slightly lower. A *sense amplifier* detects this small change and recovers a 1 or 0 accordingly. Note that reading a cell destroys the original voltage stored on the capacitor, so that the recovered data must be written back into the cell after reading.

sense amplifier

The capacitor in a DRAM cell has a very small capacitance, but the MOS transistor that accesses it has a very high impedance. Therefore, it takes a relatively long time (100 milliseconds or more) for a HIGH voltage to discharge to the point that it looks more like a LOW voltage. In the meantime, the capacitor stores one bit of information.

Naturally, using a computer would be no fun if you had to reboot every 100 milliseconds because its memory contents disappeared (the behavior of some computers notwithstanding). Therefore, DRAM-based memory systems use *refresh cycles* to update every memory cell periodically, typically once every 64 milliseconds. This involves sequentially reading the somewhat degraded contents of each cell into a D latch and then writing back a nice solid LOW or HIGH value from the latch. Figure 15-28 illustrates the electrical state of a cell after a write and a sequence of refresh operations.

refresh cycle

The first DRAMs, introduced in the early 1970s, contained only 1024 bits, but modern DRAMs are available containing 16 *gigabits* or more. If you had to refresh every cell, one at a time, in 64 milliseconds, you'd have a problem—that works out to about 1 picosecond per cell, and includes no time for useful read and write operations. But like other memories, as we'll show, DRAMs are organized using two-dimensional arrays, and a single operation refreshes an entire row of the array. Early DRAM arrays had 256 rows, requiring 256 refresh operations every four milliseconds, or one about every 15.6 µsec. The newest arrays have 8192 rows but need to be refreshed only once every 64 ms, which works out to one row per 7.8 µsec. A refresh operation typically takes only a few tens of nanoseconds, and can often be "hidden" during times when the DRAM would otherwise be idle, so the DRAM is available for useful read and write operations well over 99% of the time.

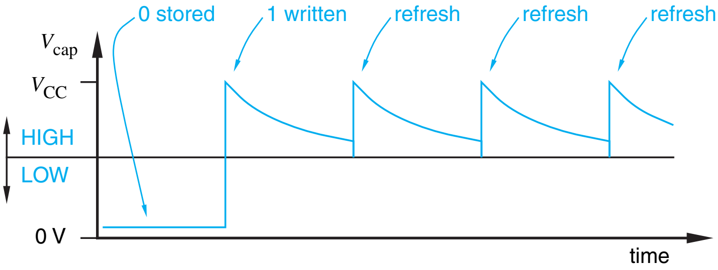


Figure 15-28 Voltage stored in a DRAM cell after writing and refresh operations.

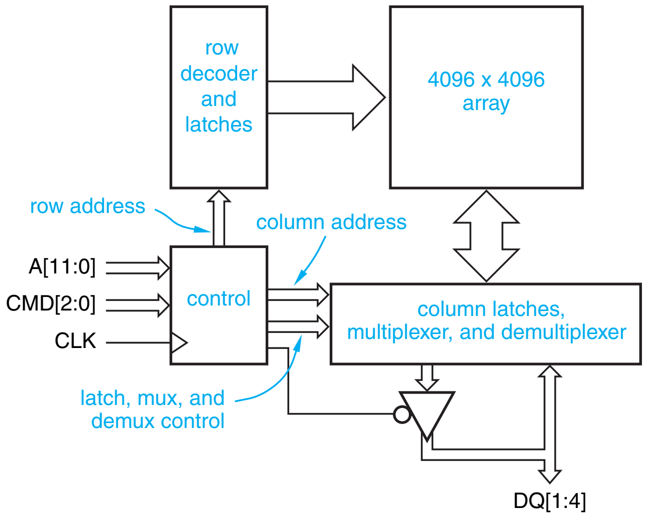


Figure 15-29
SDRAM internal structure.

For simplicity, we'll describe DRAM operations in terms of a relatively small, generic 4M × 4-bit DRAM. Larger DRAMs typically contain multiple arrays (called *banks*) with sizes that may be different from what is shown here.

DRAM bank

Figure 15-29 is a block diagram of the internal structure of our example 4M × 4 DRAM. This device is called a *synchronous DRAM (SDRAM)* because its control and data operations are all referenced to a common clock signal, CLK. Older DRAMs had asynchronous control signals; for more information, see the third edition of this book.

synchronous DRAM (SDRAM)

The logical array in Figure 15-29 has 4M × 4 bits, but the physical array is square, containing 4096 × 4096 bits. Many commercial DRAM chips have non-square individual arrays, but multiple nonsquare arrays (banks) are arranged to yield an overall chip that is close to square.

In the earliest SDRAMs, the clock signal CLK ran at 100 MHz; the newest SDRAMs run at 1067 MHz or more, and transfer data on both edges of the clock (at a so-called *double data rate [DDR]*). Various commands, explained shortly, can be applied to the device on the 3-bit CMD bus at each rising edge of CLK.

double data rate (DDR)

Although the example SDRAM has 4M (2^{22}) locations, the chip has only 12 *multiplexed address inputs* A[11:0]. A complete 22-bit address is presented to the chip in two steps at two clock ticks, as determined by operation codes on the CMD bus. Multiplexing the address inputs saves pins, important for compact design of memory systems, and also fits quite naturally with the two-step SDRAM access methods that we'll describe shortly.

multiplexed address inputs

One advantage of having multiple banks in larger SDRAMs is to ease the electrical and physical design problems that would occur with a single, very large memory array. But even more important is the parallelism that can occur when there are multiple banks. As we'll see in the next subsection, SDRAM operation is much more complicated than SRAM operation. Taking advantage of the multiple banks in larger, high-speed SDRAMs, a modern SDRAM memory controller can perform several operations in parallel—for example, completing a write operation in one bank while initiating a read operation in another. This increases the effective throughput of the memory.

15.4.2 SDRAM Timing

There are many different timing scenarios for different SDRAM types and operations. In this subsection, we'll describe the most common cycles for conventional SDRAMs and relate them to the internal structure of the device.

As we mentioned previously, the 3-bit CMD bus is used to give a command to the SDRAM at each clock tick. Typical SDRAMs have four or more banks; additional input bits select the bank to which the command is applied. Most commands take several clock cycles to complete, and multiple commands are required to perform a single read or write operation. Table 15-2 gives the names and descriptions of the most commonly used commands.

To perform a *read cycle*, several steps and commands are required:

read cycle

1. Select the bank containing the desired address and issue the PRE command. This precharges all of the bit lines in the bank to a voltage halfway between HIGH and LOW.
2. Wait a few clock ticks (as specified by the SDRAM manufacturer) until the precharge operation has completed.
3. Once again select the desired bank, apply the high-order bits of the desired address (the *row address*) to the A[11:0] inputs, and issue the ACTV command. The row address is stored in an internal *row-address register* and the word line for the selected row is activated, so the entire row can be read and stored in an internal 4096-bit *row latch*.
4. Wait a few clock ticks (the so-called *RAS-CAS delay*) for the 4096-bit word that was just read to stabilize internally.

row address

row-address register

row latch

RAS-CAS delay

Command Name	***Description***
NOP	No operation
ACTV	Row-address strobe and activate bank
READ	Column address and read command
READA	Read with auto-precharge
WRIT	Column address and write command
WRITA	Write with auto-precharge
REF	Auto refresh
PRE	Precharge

Table 15-2
Commonly used SDRAM commands.

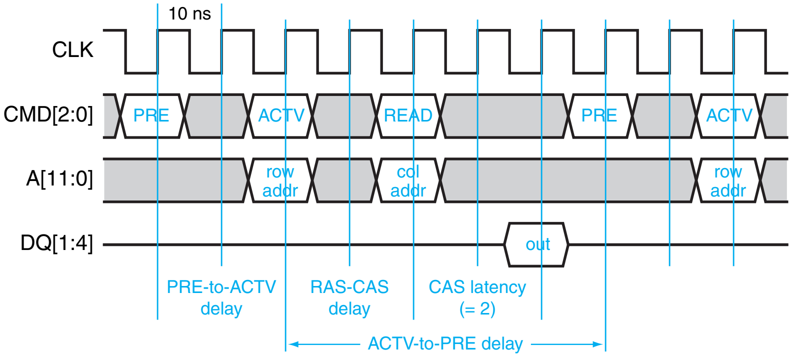


Figure 15-30 SDRAM read-cycle timing.

column address

column-address register

5. Apply the low-order bits of the desired address (the *column address*) to the A[11:0] inputs, and issue the READ command. The column address is stored in an internal *column-address register* and is applied to the column multiplexer to select four bits out of the 4096-bit row latch to be delivered to the DQ[1:4] output pins. (Only pins A[9:0] are used for the 10-bit column address in this example.)

CAS latency

6. Wait a few more clock ticks (the so-called *CAS latency*) for the addressed four bits to propagate through the column multiplexer and to the DQ[1:4] outputs. During this time, the 4096-bit row latch is also written back into the selected row. (Remember, all the capacitors in the row were discharged by the read operation back in step 3.)
7. Finally, read the data on the DQ[1:4] data input/output pins.

These operations are illustrated in Figure 15-30 assuming a fairly generic SDRAM with a 100-MHz clock and a CAS latency of 2. A memory controller or microprocessor applies commands and addresses on the CMD and A buses in order to be valid at the rising edge of CLK with a few nanoseconds of setup and hold time. The SDRAM eventually places read data on the DQ bus shortly (typically 5 ns) after a rising edge of CLK, so that the memory controller or microprocessor can read it reliably on the next rising edge.

WHEN PRE BECOMES POST

The need to precharge a DRAM's bit lines prior to a read or write adds significant delay to these operations. Therefore, most DRAM controllers are designed to precharge the bit lines *after* each operation completes. That way, there's a good chance that the precharge will be completed by the time a new operation to the bank is requested. In SDRAMs, the READA and WRITA commands automatically perform a precharge on the just-used bank as a read or write operation is completed, and no separate PRE command needs to be given.

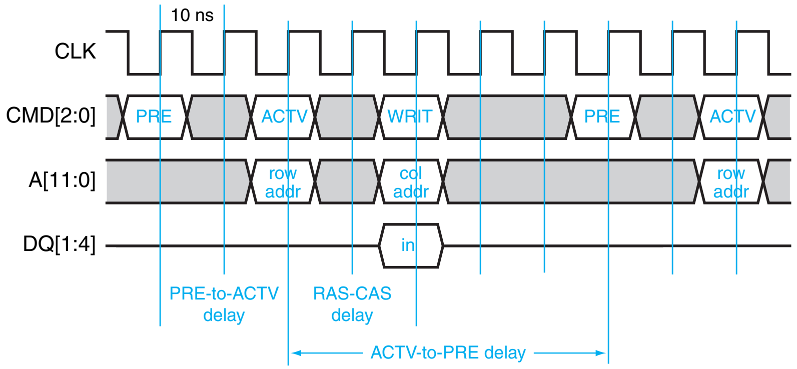


Figure 15-31
SDRAM write-cycle timing.

SDRAMs have all sorts of complicated timing requirements that are specified by the manufacturer. For example, once a read cycle has completed, it may or may not be possible to request a new precharge cycle immediately—first, an active-to-precharge delay must elapse, as shown in the figure. The timing requirements also vary depending on whether read and write operations are interleaved, and whether successive operations are going to the same bank or different banks. This makes the design of SDRAM memory controllers very challenging, but ripe with opportunities for increasing efficiency.

write cycle

As shown in Figure 15-31, SDRAM *write cycles* are like read operations, with one main difference. The memory controller or microprocessor drives the write-data onto the DQ bus at the same time that it issues the WRIT command. During the next few clock cycles (step 6 in our previous description of read operations), the SDRAM merges the write-data into the addressed column in the row latch and then writes the entire, updated 4096-bit value back into selected row.

As you can see from the timing diagrams, it takes a lot of time and effort to read or write a single location in an SDRAM—a total of seven clock cycles in our read and write examples just to obtain one clock cycle with an actual data transfer. Compare with SSRAMs (see Section 15.3.5), which can perform a data transfer on every clock cycle.

SDRAMs can achieve higher data transfer rates when multiple locations in the same row of the internal memory array are accessed successively. After all, in our example SDRAM, the entire 4096-bit row is stored in the row latch during a read operation, and it is a relatively simple matter to bring out a different 4-bit word to the DQ bus on successive clock ticks.

burst-read cycle

Thus, Figure 15-32 shows the timing for a *burst-read cycle*, assuming a burst length of four. The first 4-bit word is delivered at the same time as in a normal read cycle, and additional words from successive locations are delivered during the next three clock cycles. A typical SDRAM can support burst lengths of 1, 2, 4, or 8 words, or the entire row latch (also called a *page*, 1024 words in this example).

page

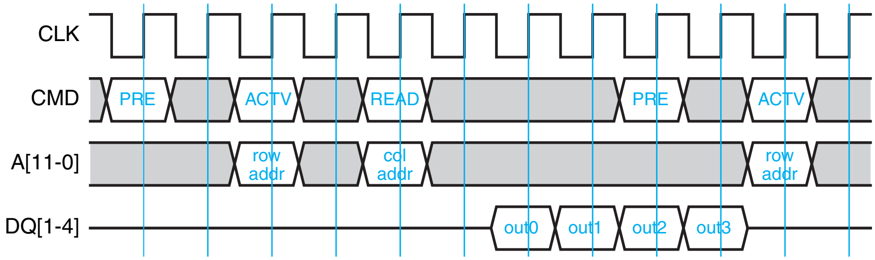


Figure 15-32
SDRAM burst-read-cycle timing.

configuration register

The burst length is not specified on a per-operation basis; rather, it is specified statically, using a few bits in an internal *configuration register*. The memory controller programs the configuration register when the system begins operation. Other static operating parameters, such as CAS latency (2 or 3) and the method for generating successive burst addresses (sequential or interleaved), are also programmed in the configuration register.

burst-write cycle

As shown in Figure 15-33, operations for a *burst-write cycle* are similar, with the memory controller or microprocessor delivering one word to the DQ bus per cycle, starting in the same clock cycle as the WRIT command. A new precharge command cannot be given until two ticks after the last write-data appears on the DQ bus. This provides time for the updated row latch to be written back into the selected row.

auto-refresh cycle

refresh counter

Before we forget. we should discuss one other very important SDRAM operation—the *auto-refresh cycle*. In this cycle, initiated by the REF command, the SDRAM reads one row of each bank's internal array into its row latch, and writes it back. No address is applied to the A bus; instead, the SDRAM uses the value of an internal 12-bit *refresh counter* as the row address, and increments it after the refresh operation. A total of 4096 refresh operations must be performed every 64 ms to prevent data loss. (The banks in larger chips have 8192 rows and need twice as many refresh operations.) All banks must be precharged before a REF command is given, and the command precharges the banks as it completes.

15.4.3 DDR SDRAMs

double-data-rate (DDR) SDRAM

Double-data-rate (DDR) SDRAMs do just that—they double the data-transfer rate of an SDRAM by transferring data on both edges of the clock, rising and falling. Note that address and command operations still require one clock cycle,

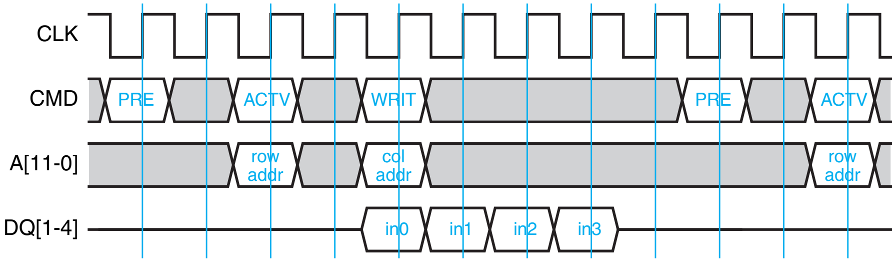


Figure 15-33
SDRAM burst-write-cycle timing.

the same as in a conventional SDRAM. So, the actual data-transfer rate is increased only for burst operations.

DDR operation is "simple" from a functional point of view. For example, just imagine eight words coming out in the same interval that four are shown in the SDRAM burst cycles in Figures 15-32 and 15-33, with even-numbered words referenced to the rising edge of the clock, and odd-numbered to the falling edge.

But DDR operation is very tricky from a timing and analog implementation point of view. Remember, the whole point of DDR is to go *fast*. To maintain precise timing, DDR SDRAMs use differential clock inputs—complementary versions of the clock signal with very little timing skew between them. An on-chip analog *delay-locked loop (DLL)* locks onto this clock signal and generates internal and external signals, including output data and input and output latch enables, with precise delays relative to this clock.

delay-locked loop (DLL)

Board-level designers must be very careful to balance the delays, minimize the skew, and optimize the quality of all signals going to and from the DDR SDRAM. Even after all this work, DDR operation provides faster data transfers only during burst-mode operations, so the benefit is application dependent.

15.5 Field-Programmable Gate Arrays (FPGAs)

In Chapter 1, we introduced basic FPGA architecture, an array of *configurable logic blocks (CLBs)* embedded in a sea of programmable interconnect. Leading FPGA manufacturers include Xilinx, Altera (now part of Intel), and Lattice Semiconductor. The internal architectures of FPGAs differ among different manufacturers and even among different families from the same manufacturer; in this book we use the Xilinx 7 series as the running example.

configurable logic block (CLB)

In Sections 6.1.3, 8.1.10, and 10.7, we looked at the architectural features of the configurable logic blocks in 7-series FPGAs. In this section, we'll start at an even higher level, showing how multiple "regions" of logic are put together on a single chip. Then we'll revisit CLBs and other elements, drill down into some detail on the family's programmable I/O features, and end with a peek at the device's massive programmable interconnect structure.

15.5.1 Xilinx 7-Series FPGA Family

The Xilinx 7-series FPGA family was designed to span a broad range of device sizes and applications. The smallest devices have only about 2,800 LUTs, while the largest have over 600,000.

An important question must be answered to scale this or any other FPGA architecture—should larger devices simply increase the dimensions and parameters of the basic architecture, or should they add another layer of hierarchy? Another important question exists at all sizes. Because of the nature of design and redesign for FPGAs, it is often necessary to migrate a design from one

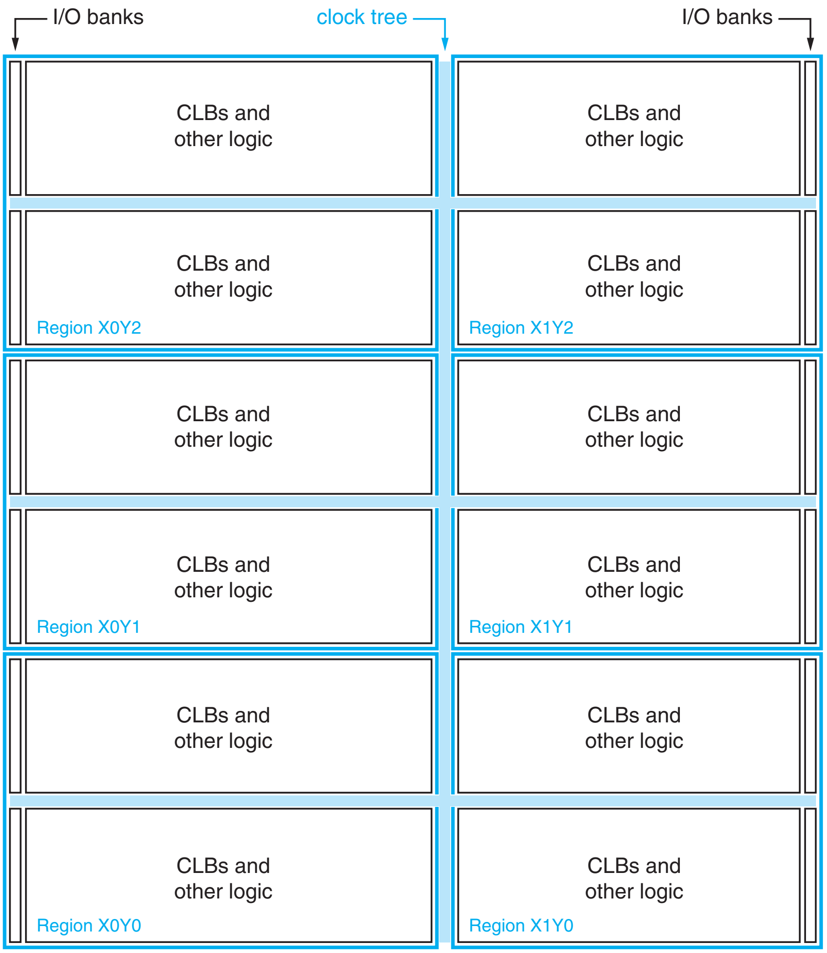


Figure 15-34 Xilinx 7-series FPGA with six regions.

device to a larger one, or in rare cases, to a smaller one—how can that be done without reworking the design for different device structural parameters?

The 7 series uses the approach in Figure 15-34, which shows the physical layout for one FPGA chip (die). This chip is divided into six regions, outlined in dark color, which are stacked three high and two across. All regions in this and in all 7-series FPGA are exactly the same height. However, the stack height (number of regions) may vary in different chips, and the left and right regions may have different widths, even in the same chip. The largest chips may have a third stack of regions. These variables provided ample opportunity for Xilinx to offer a wide range of device sizes, some of which are listed in Table 15-3. We'll describe resources listed there in the next subsection.

A few other aspects of Figure 15-34 are noteworthy. Dedicated resources are provided for clock distribution in a tree-like fashion that may remind you of our clock-skew discussion in Section 13.3.1. The main "trunk" of the tree goes

Device	*Slices*	*LUTs*	*18 Kb BRAMs*	*DSPs*	*CMTs*	*Max. user I/O*
XC7S6	938	3,752	10	10	2	100
XC7S15	2,000	8,000	20	20	2	100
XC7S50	8,150	32,600	150	120	5	250
XC7S100	16,000	64,000	240	160	8	400
XC7A12T	2,000	8,000	40	40	3	150
XC7A35T	5,200	20,800	100	90	5	250
XC7A75T	11,800	47,200	210	180	6	300
XC7A200T	33,650	134,600	730	740	10	500
XC7K70T	10,250	41,000	270	240	6	300
XC7K160T	25,350	101,400	650	600	8	400
XC7K325T	50,950	203,800	890	840	10	500
XC7K355T	55,650	222,600	1,430	1,440	6	300
XC7K410T	63,550	253,800	1,590	1,540	10	500
XC7K480T	74,650	297,400	1,910	1,920	8	400
XC7VX415T	64,400	257,600	1,760	2,160	12	600
XC7V585T	91,050	364,200	1,590	1,260	17	850
XC7VX690T	108,300	433,200	2,940	3,600	20	1,000
XC7VX980T	153,000	612,000	3,000	3,600	18	900

Table 15-3
Resources in Xilinx 7-series FPGAs

down the center of the chip, between the left and right regions, with horizontal branches extending across at the middle of each region. From the branches, additional branches (not shown) extend upward and downward to reach all of the CLBs in the region. Thus, clock-signal lengths are well-balanced, and clocks are distributed and "managed" on a regional basis. Each region has local clocks as well as access to global (chip-wide) clocks. Even clocks that are generated off-chip are brought to the trunk before being distributed to the regions. A special logic resource called a "Clock Management Tile" (CMT) in each region, located near the trunk, selects, adjusts, and distributes the clocks in that region.

7-series I/O bank

A second interesting aspect is the vertical *I/O bank* on one side of each region. Each I/O bank contains 50 I/O pads and related circuitry (an "I/O block"), which can be programmed to support various electrical I/O standards on a bank-by-bank basis, as we describe in more detail in Section 15.5.3. So, even if a design grows or shrinks enough to require different resources, including more or fewer regions, its I/Os and clocks can normally stay in their originally assigned region, thereby avoiding I/O-dependent "fitting" problems.

I/O PAD LAYOUT

In our original FPGA diagram in Chapter 1, we showed the I/O pads in a so-called "pad ring" around the circumference of the chip, for wired connections to the IC package's pins. In Figure 15-34, the pads are on the two sides only. In larger chips, a third column of pads has to go somewhere in the middle of the chip. But that all still works with the "flip-chip" technology used in the 7 series.

In the flip-chip approach, the FPGA die is manufactured with solder bumps on each I/O pad. A special carrier is made with a pattern of contacts that match the bumps on the FPGA die when the die is "flipped" and placed face down on the carrier. The whole assembly is heated, melting the solder to mechanically and electrically bond the FPGA chip to the carrier. The carrier has its own internal metal routing layers that connect the FPGA's I/O pads to whatever the carrier uses to connect to a printed-circuit board on the other side, like pins in a ceramic pin-grid array (PGA), or the carrier's solder balls in a ball-grid array (BGA). Figure 15-35 illustrates the BGA, omitting the typical lid over the FPGA chip.

Consistent with this aspect, notice in the last two columns of Table 15-3 that the maximum number of user I/Os is always exactly 50 times the number of CMTs, which is also the number of regions. Also notice the letter S, A, K, or V after "XC7" in each device number. The letter designates one of four different 7-series subfamilies with different packaging and price-performance points.

Yet another aspect of modern FPGA architectures is the ability to augment their I/O structures with specialized elements for particular applications. In the Xilinx 7 series, many of the devices in Table 15-3 have one or more regions containing different I/O pads and circuits to support high-speed serial interfaces.

For example, all of the devices except the "S" subfamily have dedicated, hardwired logic to support at least one PCI Express interface with up to eight lanes of 5 Gbps serial I/O. For even higher speed serial I/O, some devices have specialized I/O transceivers, phase-locked loops, and serial-parallel conversion circuits to support rates up to 28 Gbps per transceiver, typically used to connect with other devices in the same system. One of the largest devices in Table 15-3, the XC7VX690T, has 80 such transceivers capable of up to 13 Gbps each, in addition to its 1,000 pins of "low-speed" user I/O.

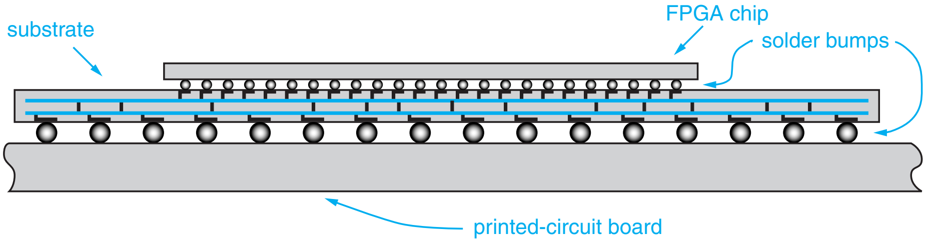


Figure 15-35 Flip-chip FPGA bonded to the substrate for a ball-grid array.

A final aspect that has gained popularity is the deployment of FPGAs together with microprocessors to create *systems-on-a-chip (SoCs)* for embedded applications, like medical instruments, machine vision, and professional video equipment. Instead of combining the FPGA's customized hardware with an off-chip microprocessor subsystem, the designer can now use a microprocessor that is already integrated on the same chip with the FPGA itself.

system on a chip (SoC)

Thus, Xilinx created the "Zynq" family of devices that have the same base architecture as 7-series FPGAs, but replace one or more regions or parts of them with a microprocessor subsystem having the following major elements:

- A microprocessor and associated cache memory.
- Boot ROM and a modest amount of on-chip SRAM (256 KB).
- Interfaces to external NOR flash, NAND flash, SRAM, and DRAM.
- Direct Memory Access (DMA) interface.
- Two gigabit Ethernet and two USB interfaces.
- Up to 54 general-purpose I/O ports assignable to external I/O pins.
- Industry-standard on-chip I/O bus to interface with customized elements created using the FPGA's programmable logic.

Thus, using the FPGA's programmable hardware resources, a designer can customize the circuit for a particular application, and control it with customized software using the on-chip microprocessor subsystem and off-chip resources like flash and DRAM that are also tailored to the application.

Like the base 7-series FPGA family, the Zynq family is offered at several different size and price-performance points, with programmable-logic options ranging from 3,600 to 69,100 slices, 100–1,510 BRAMs, 66–2,020 DSPs, and 54–400 user I/O pins in addition to ones used by the microprocessor subsystem for memory interfaces and the like.

15.5.2 CLBs and Other Logic Resources

Since an FPGA can have lots and lots of CLBs, it's important that we understand them! We showed the position of CLBs within the 7-series FPGA chip hierarchy in Section 10.7, and for reference the illustration is repeated in Figure 15-36. The basic configurable logic element contains four LUTs, eight flip-flops, and a CARRY4 element, and is called a *slice*. Two slices are paired to form a CLB which is embedded as a unit into the sea of interconnect. However, note that like the Xilinx literature, our Table 15-3 counts slices, not CLBs.

slice

We saw the basic capabilities of a slice elsewhere as we introduced each of its various elements, but for reference they are summarized here:

- Each slice has four 6-input LUTs. A LUT can perform any combinational logic function of six variables, or it can be split to perform any two functions of the same five variables (see Section 6.1.3).

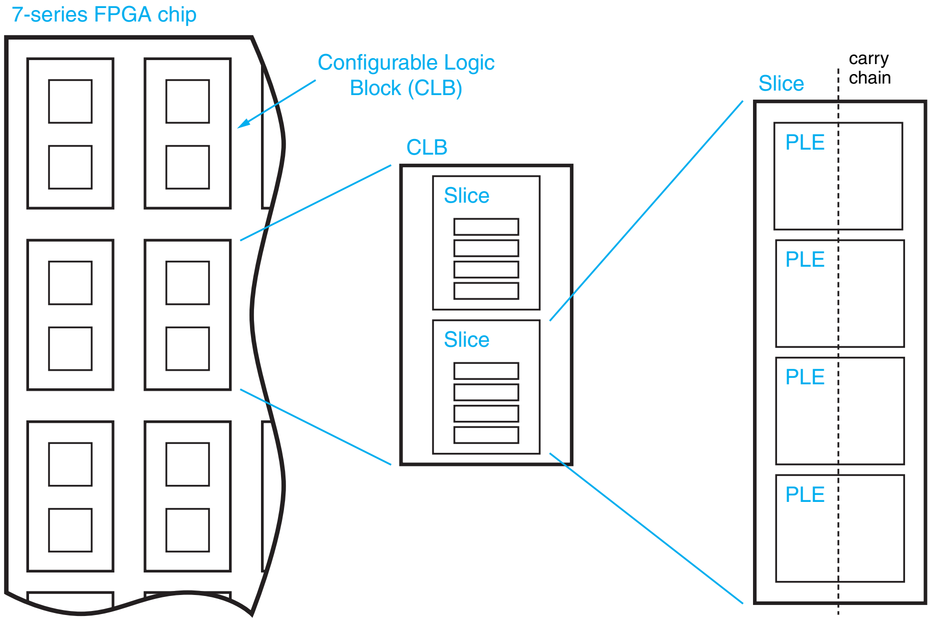


Figure 15-36 CLBs in a Xilinx 7-series FPGA.

- A D flip-flop and a second 1-bit storage element, programmable to be a D flip-flop or latch, are paired with each LUT, for a total of eight flip-flops/latches in the slice (see Section 10.7).
- A specialized CARRY4 element provides a fast carry path for 4 bits of addition, and carries into and out of the slice are cascaded with the slices above and below (see Section 8.1.10).
- Special multiplexers (F7MUX and F8MUX) combine LUT outputs to implement 7- and 8-input combinational functions (see box on page 245).
- A LUT's 64-bit "ROM" may be used to create a 32-bit shift register instead of storing a truth table (see References on page 596).
- The "ROM" in one or more LUTs may be used to create a 32 × 1 or larger "distributed" SRAM or ROM (see References on page 863).

Figure 15-37 shows the physical layout of one region in a Xilinx 7-series FPGA. The I/O blocks and the CLBs are the same height, and the height of a region is always exactly 50 of these elements. Two other elements that we'll describe next, BRAMs and DSPs, are somewhat taller, so their columns in a region are 20 elements high. The widths of the left and right regions may be different, may have more than one BRAM or DSP column, and may also may vary among family members.

block RAM

The BRAM elements in Table 15-3 and Figure 15-37 are *block RAMs*. Each BRAM has 18K bits of SRAM that can be used as an independent 16K × 1, 8K × 2, 4K × 4, 2K × 9, 1K × 18 or 512 × 36 memory. Pairs of BRAMs are laid

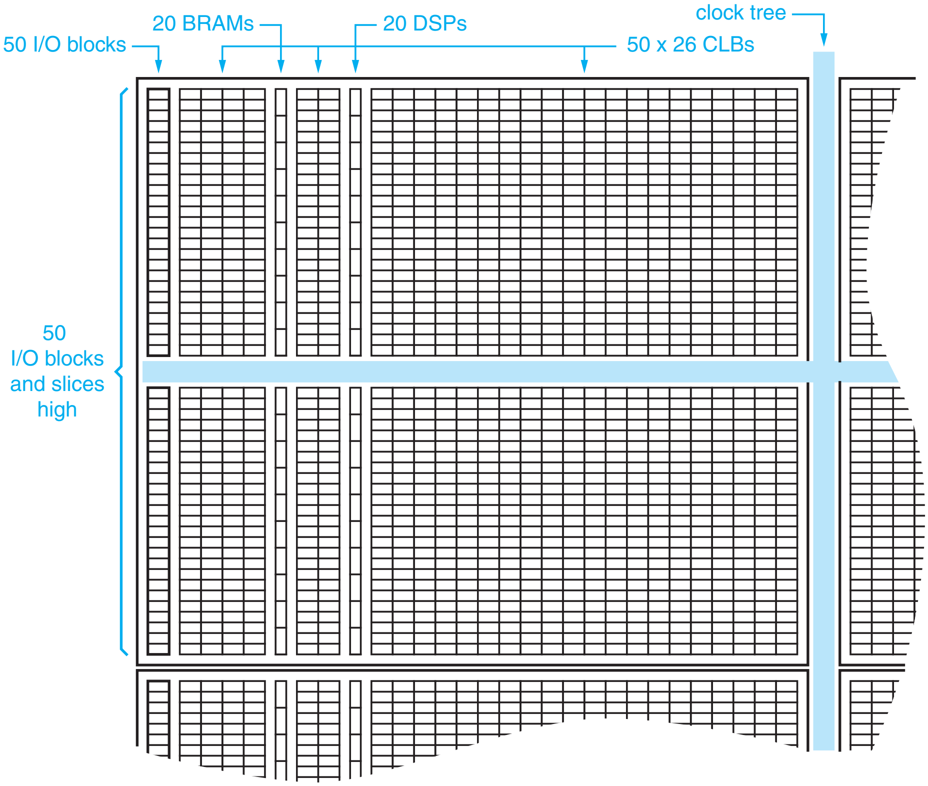


Figure 15-37 Physical layout of a Xilinx 7-series region.

out vertically in 36-Kb blocks, supporting block configurations of 64K × 1, 32K × 1, 16K × 2, 8K × 4, 4K × 9, 2K × 18, 1K × 36, and 512 × 72. Some of these configurations have multiple ports, which means that the memory can be read or written simultaneously at two or more different addresses. Writing the BRAM is always synchronous, while reading may be synchronous or asynchronous. Synthesis tools can create wider and deeper memories using multiple BRAMs and CLBs as needed for additional control and multiplexing logic.

The BRAM's 36-Kb block size may seem unusual until you see that it leads to a configuration with a 72-bit width. This is exactly what's needed to support an extended Hamming code that can correct one error and detect double errors in 64-bit data (see Section 2.15.3). In fact, each 36-Kb BRAM block has built-in Hamming encoding and decoding logic to support such a code. Each BRAM can also be configured as a FIFO memory with the same (synchronous) or different (asynchronous) read and write clocks.

DSP slice

The last fundamental element in Table 15-3 and Figure 15-37 is the *DSP slice*, which is an integrated datapath for performing digital-signal-processing (DSP) operations. The DSP slice has four data inputs and corresponding registers, an 18-bit adder/subtractor and a 48-bit ALU, an 18 × 25-bit multiplier, inputs like clock enables and multiplexer selects to control various operations, and a 48-bit registered data output. The designer must build a controller using CLBs to specify how datapath resources are used at each clock tick to execute a desired DSP algorithm. For DSP-intensive applications, like video processing, the largest 7-series FPGAs have up to 3,600 DSP slices. The DSP slices are laid out on the FPGA chip near BRAMs, which can store arrays of operands that are used in various algorithms.

15.5.3 Input/Output Block

We introduced the Xilinx logical three-state input/output buffer component IOBUF in Section 7.1.4. Each I/O pad on a 7-series device can be an input, an output, or both.

The physical I/O buffers in 7-series FPGAs have many different electrical configuration options. These options are selected through a combination of programmable bits that are loaded with all the others at chip initialization, and power supplies that are connected to the installed chip on a printed circuit board.

Recall from Section 14.7 that there are several standard voltage standards for low-voltage CMOS logic families. The 7-series FPGAs aim to support all of these and more. The power supply for 7-series internal logic is a mere 1.0 V, but level translation is provided to support higher voltages for I/O. Each I/O bank has its own supply pins to power its 50 output drivers, which swing between 0 V and V_{CCO}; V_{CCO} may be as high as 3.3 V. Each I/O bank also has its own "input reference voltage" V_{REF}, which establishes the threshold voltage between LOW and HIGH inputs, normally half of the signal swing. The reference voltage can be from an external source using an I/O pin, or for 1.2-V to 1.8-V standards it can be generated internally. Using these options, a single 7-series device can connect using multiple I/O standards, but voltage-compatible standards must be used within each 50-pin I/O bank.

A greatly simplified logic/block diagram for one bit of 7-series I/O is shown in Figure 15-38. The full capabilities are so extensive that Xilinx documentation actually separates this diagram into five parts, the *input/output block (IOB)* and separate logic and delay blocks for both the input and the output signal paths. Let's look at the IOB starting on the right-hand side of the figure.

input/output block (IOB)

The I/O pad (pin) can be configured to have a weak pull-up or pull-down resistor, or a bus-holder (a.k.a. "keeper") circuit of the kind we described in Section 10.5.2 to hold the last value on a three-state bus when it is not actively driven. The IOB's input buffer and output driver support various I/O standards, as described previously, and the output driver has two other "analog" options:

- *Slew rate*. Transitions can be configured to be "fast" or "slow," as required for external devices or for trading off board-level signal speed versus noise.
- *Drive strength*. The maximum current sourcing and sinking capability can be adjusted between 2 and 24 mA, as needed for the connected load.

Moving now to "logic resources" on the left-hand side of Figure 15-38, both the input path and output path can be configured with storage elements in the path, each selected by a multiplexer. Placing storage elements "up close" to the device I/O pins is especially useful in FPGAs. On output, relatively long delays from internal CLB flip-flop outputs to the IOBs can make it difficult to connect to external synchronous systems at very high clock rates. On input, long delays from the I/O pins to CLB flip-flop inputs can make it difficult to meet external system setup and hold times. Of course, using IOB storage elements is possible only if the FPGA application's interfacing requirements for external devices allow such "pipelining" of inputs and outputs.

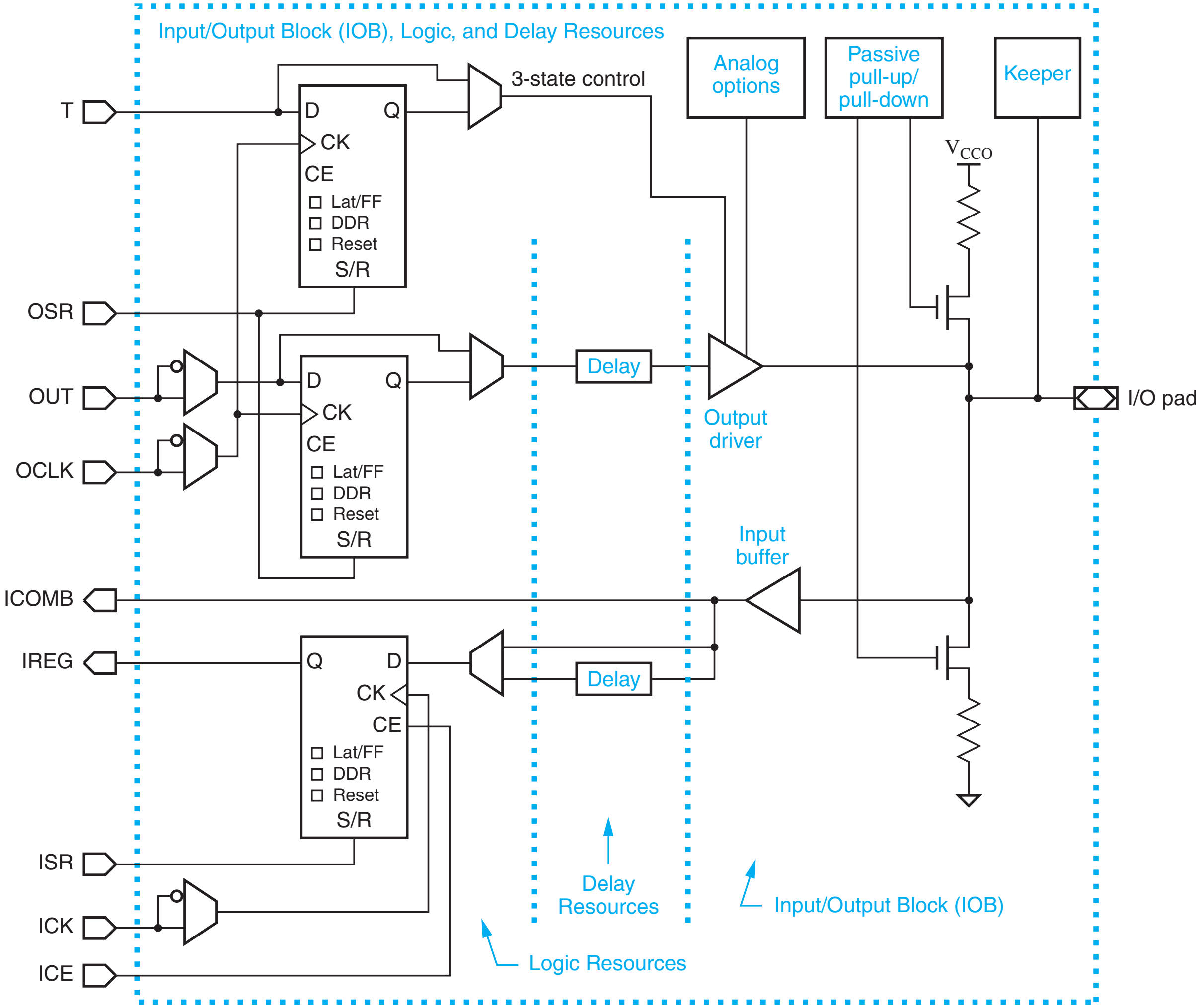


Figure 15-38 Xilinx 7-series I/O block, I/O logic, and I/O delay resources.

The storage elements in the input and output paths are configured individually with options similar to the ones in the CLBs: register or latch, positive or negative clock polarity, and synchronous or asynchronous preset or clear. Note that in the output path, a storage element is provided for the three-state enable as well as for the data bit. The "DDR" options are discussed shortly. The logic resources also include a *serializer/deserializer (SERDES)*, not shown, to convert very high speed serial inputs and outputs (up to 1.6 Gbps) to and from a parallel format up to 14 bits wide for lower-speed processing by the FPGA's logic.

serializer/deserializer (SERDES)

Between the logic resources and the IOB, the input and output paths have "delay resources" that allow the designer to delay the signal path by up to 32 steps as short as 39 ps. This block can be used for various purposes, including compensating for board-level path delays and spreading out the transitions in otherwise simultaneously switching outputs to mitigate board-level noise.

Several other important facilities for high-speed interfaces are not shown in the figure. First, the input and output paths are configurable to support double-data-rate (DDR) operation, used in many memory interfaces to double the data bandwidth by sending and receiving data on both edges of the reference clock. The I/O logic has extra storage elements and control signals to convert each DDR signal to or from a pair of internal signals to process using just a single edge of an internal clock. Second, a pair of input buffers or a pair of output drivers can be configured and used together to accomplish differential signaling, which we described in Section 14.8. DDR and differential signaling configurations can be used independently or together. Finally, the I/O pads may be configured with various resistive transmission-line termination networks to reduce the reflections that occur in high-speed board-level connections.

15.5.4 Programmable Interconnect

Well, we saved the best for last. The 7-series programmable interconnect architecture is a fine example of a structure that provides massive programmable connectivity in a commercially feasible silicon area.

In Chapter 1, we showed a general FPGA architecture in which CLBs are individually embedded in the interconnect structure, but the 7 series instead embeds back-to-back pairs of CLBs to get, according to Xilinx, better routing and almost certainly slightly lower chip area (and cost) for a given amount of connectivity.

As shown on the left-hand side of Figure 15-39, the two slices in a CLB are connected horizontally to a *switch box* (described shortly) which itself connects horizontally to a much larger switch box, and the whole structure is mirrored on the right-hand side of the figure. The mirrored structure is embedded between horizontal interconnect lines above and below it, and there are vertical interconnect lines between the switch boxes. The two large switch boxes connect to both vertical and horizontal interconnect, as well as to a vertical spine coming from the region's horizontal clock-distribution branch shown in Figure 15-37.

switch box

The interconnect itself has more structure and subtlety than is conveyed by Figure 15-39's thick gray lines. Wires are classified as "singles," "doubles," "quads," or longer, depending on whether they connect switch boxes that are adjacent or 2, 4, or more switch boxes apart. Each switch box has a variety of vertical and horizontal interconnect lengths connected to it, providing many potential ways to connect to other switch boxes. Connections that are more than two switch boxes apart may require multiple hops through intermediate boxes.

Each switch box programmably makes connections among the wires going into it. Connections are made by CMOS transmission gates—switches—whose open/closed states are determined by configuration memory (SRAM) that is loaded when the FPGA chip is initialized. Each site of a potential connection is called a *programmable interconnection point (PIP)*.

programmable interconnection point (PIP)

The leftmost, smaller switch box in Figure 15-39 has only about 150 PIPs. All of the inputs and outputs of the CLB's two slices enter this switch box on its lefthand side. During normal operation, the switch box can connect these wires to corresponding dedicated wires on its righthand side, and these wires connect to the larger switch box on to its right. However, before Xilinx ships the FPGA to its customer, these wires may be connected to non-customer-visible resources for factory-testing purposes.

The real action occurs in the larger switch boxes, one per CLB, and each one containing about 3,700 PIPs. The numbers in color show roughly how many wires enter each side of the switch box. Inside the switch box, transmission-gate-based multiplexers provide a rich set of connection opportunities, all con-

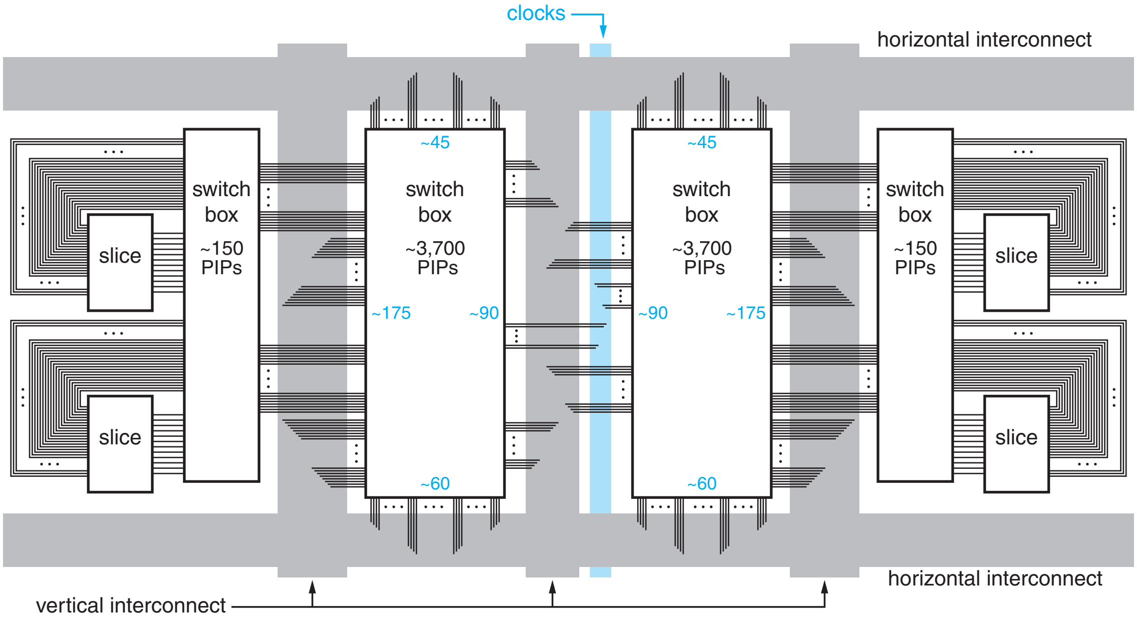


Figure 15-39 Xilinx 7-series interconnect for a pair of CLBs.

CONVENIENCE STORY

OK, I lied. The small, 150-PIP switch boxes in Figure 15-39 don’t really exist, even though you can see them if you look deeply into the “Device” view of a 7-series FPGA using Vivado tools. According to Xilinx, this switch box is merely a graphical convenience to better see what’s going on here and with other interconnected FPGA elements, such as BRAMs. However, its functionality *does* exist; it’s just physically integrated with the larger 3,700-PIP switch box.

trolled by configuration memory. Basically, any CLB output can drive any type of interconnect resource—singles, doubles, quads, or longer—or one of its own inputs (for example, when a register output drives a LUT input in the same CLB). CLB inputs can be driven only by singles, doubles, other CLB inputs, and clocks which, as shown, also come into the switchbox.

The switch box has other capabilities. It can connect interconnect wires to other interconnect wires. Thus, signal routes can “turn a corner” when a switch box connects a vertical wire to a horizontal wire. It can also fan out signals: an incoming signal can connect to multiple outputs. Signal paths through the switch box may be buffered within the switch box, which is especially important to reduce delay on signals arriving from or going to longer interconnect wires, which are themselves passive.

We mentioned previously that longer connections may require hops through multiple switch boxes. This increases “cost” in two ways: more switch box resources are used, reducing their availability for other connections; and delay is increased. The placement and routing tools use well-developed algorithms to optimize connections based on these and other considerations. For example, to connect CLBs that are six apart, it’s better to use a quad and a double rather than six singles or three doubles.

FPGAs are judged not only by the capabilities of their logic resources but also by the consistency of the results obtained from a fitter after small design changes are made. There’s nothing more frustrating than making a small change to a large design and finding that it no longer meets timing requirements, or worse, cannot be routed. Thus, FPGA manufacturers have learned to provide “extra” resources in their architectures to help ensure consistent results.

GOOD PRACTICE

Placement and routing is actually a pretty well understood problem, because it is the major part of the “back end” of any custom chip design. Thus, the same kind of tools and the same tool vendors are involved with placement and routing for both FPGAs and ASICS. So, you might like to consider any FPGA experience that you get to be good practice for ASIC design!

References

Manufacturers publish individual data sheets and application notes for their devices on their websites. A good source for information on smaller, “legacy” ROMS and SRAMs is Renesas (`www.renesas.com`), while Micron Technology (`www.micron.com`) has information on large NAND flash devices and just about every variety of DRAM. Interfacing standards for NAND flash devices are developed and published by an industry group, Open NAND Flash Interface (ONFI, `www.onfi.org`). Synchronous SRAMs are offered by Integrated Device Technology (IDT, `www.idt.com`) and Cypress Semiconductor (`www.cypress.com`).

In addition to the memories discussed in this chapter, several types of “specialty” memory devices have widespread use. Probably the most common are *first-in, first-out (FIFO) memories*; these are typically used to transfer data from one processor or clock domain to another. The websites of IDT and Texas Instruments (`www.ti.com`) are good sources of information on FIFOs.

first-in, first-out (FIFO) memories

Another type of specialty memory is the dual-port memory, which has two independent sets of address, data, and control lines and allows independent operations to be performed on both ports simultaneously. IDT is a leading source for these devices; in addition to data sheets, their website has an excellent set of application notes on the devices.

The “ROMs” used in the LUTs in many FPGAs, including the Xilinx 7 series, are actually small read/write memories that are simply loaded with truth tables at initialization to perform combinational logic functions. However, they can also be optionally configured to small read/write memories; for example, a 7-series LUT is just a 64×1 or 32×2 SRAM. Typical tools also allow the designer to create even larger SRAMs using multiple LUTs, in a “distributed SRAM” organization. In fact, they can also create “distributed ROMs” using LUTs that are loaded at initialization and have no active inputs to allow further write operations. For examples, see Xilinx publication UG474, *7 Series FPGA Configurable Logic Block*, for information and options.

FPGA vendors provide comprehensive user guides for various aspects of their devices, and the latest version is always available on the vendor’s website. For the Xilinx 7 series, important references include DS180, *7 Series FPGAs Data Sheet: Overview*; UG474 *Configurable Logic Block*; UG473, *Memory Resources*; UG472, *Clocking*; and UG471, *SelectIO Resources*.

Little is published about the programmable interconnection architectures of modern FPGAs, since vendors consider this to be part of the “secret sauce” that makes their products great. Also, the architectures are so complex that most designers would not attempt to override the tools’ programming decisions even if they were fully documented for the user. However, for more details on an older FPGA architecture, the Xilinx XC4000 family, see the fourth edition of the book you’re reading, or the documentation upon which that presentation was based, the *XC4000E Product Specification* (May, 1999), available online.

Drill Problems

15.1 Determine the ROM size needed to realize the combinational logic functions in each of the following figures: 6-19, 6-33, 7-13, and 7-27.

15.2 Determine the ROM size needed to realize the combinational logic functions in each of the following figures: 7-31, 8-6, 8-17, and 8-21.

15.3 How many ROM bits would be required to build a 16-bit adder/subtractor with mode control, carry input, carry output, and two's-complement overflow output? Be more specific than "billions and billions," and explain your answer.

15.4 Draw a logic symbol for and determine the size of a ROM that realizes an 8×8 combinational multiplier.

15.5 Show how to design a 2M × 8 SRAM using 512K × 8 SRAMs and an MSI device from Chapter 6 as building blocks.

Exercises

15.6 Describe the logic function of seven variables that is performed by the 128 × 1 ROM in Figure 15-3. Starting with the ROM pattern, one way to describe the logic function is to write the corresponding truth table and canonical sum. However, since the canonical sum has 64 seven-variable product terms, you might want to look for a simple but precise word description of the function.

15.7 Show how, using additional gates and building-block logic, a 512K × 8 ROM can be used as a 4M × 1 ROM. What is the access time of the 4M × 1 ROM?

15.8 Draw a logic diagram for a ROM-based circuit that performs combinational multiplication of a pair of 8-bit unsigned or signed-magnitude integers. Signed vs. unsigned operation should be selected by a single input, SIGNED. You may use use as many discrete gates as you can easily draw, and you may use any number of any of the ROMs in Figure 15-7, but minimize the number of ROM bits used.

15.9 Write and test a program in C or another programming language to generate the contents of the ROM(s) in Exercise 15.8.

15.10 Write a program in C or another language that generates a 256K × 4 ROM that computes the next move in a Tic-Tac-Toe game, using the input and output encodings of Section 7.5. Your program must be smart enough to pick a winning move whenever possible.

15.11 Repeat Exercise 15.10 using a 32K × 4 ROM. To accomplish this, the board state must be encoded in only 15 bits. Explain your coding algorithm, and write functions in C or another programming language to translate a cell number in either direction between your encoding and the encoding of Section 7.5.

15.12 While so-called "landlines" in the public switched telephone network (PSTN) are still analog, the switching systems at a local office as well as the long-distance network are now all digital. Each landline terminates in a device at the local office that samples the analog signals 8,000 times per second and converts each sample's voltage into a digital representation with a dynamic range of approximately 14 bits, representing analog values between approximately $-2^{13} \cdot k$ and $+2^{13} \cdot k$, where k is an arbitrary scale factor. However, only 8 bits are used in the encoding, called µ-law (“mu-law”) PCM, which has a sort of floating-point format shown in Figure X15.12. The analog value V, 14 bits including sign, of a signal encoded in this system can be given by a formula,

$$V = (1-2S)\cdot[(2^{E})\cdot(2M+33)-32]$$

The range of values of V is ± 8032, and the smallest possible difference between successive coded values is 2, when $E=0$.

Sometimes in a phone system, it is necessary to process the “linear” equivalents of µ-law-coded bytes: for example, to raise or lower the amplitude of the corresponding analog signal, or to add two signal streams in a conference bridge. Given any 8-bit encoded value U[7:0], it is possible to obtain a corresponding linear 14-bit two’s-complement value using the formula above.

Draw the logic diagram of a ROM-based circuit that converts a µ-law input U[7:0] to a corresponding two’s-complement value LIN[13:0]. What ROM size is required, and how many bits total are in the ROM? *Optional*: Write a program in your favorite language to generate the contents of the ROM.

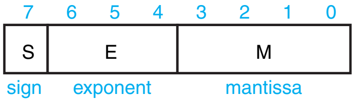


Figure X15.12

15.13 Repeat Exercise 15.12 for a ROM-based circuit that converts a two’s-complement input value LIN[13:0] into a corresponding µ-law output U[7:0]. *Optional*: Write a program in your favorite language to generate the contents of the ROM. If a linear input value falls between two linear values that correspond exactly with a µ-law-coded value, as most will, ensure that your program selects the µ-law value that corresponds most closely.

15.14 Based on the description in Exercise 15.12, write a Verilog module `Vru2lin` that converts a µ-law input value to a two’s-complement output value by performing the operations in the formula in real time, that is, not using the ROM-based approach. Target your design to an FPGA and synthesize it. How many LUTs does it use, and how many LUT “ROM” bits does it use in total? How does this compare with the ROM-based realization of Exercise 15.12?

15.15 Read the documentation for your FPGA synthesis tools and learn how to create a ROM whose bits are distributed among multiple LUTs. In your targeted FPGA family, how many “LUTs as ROM” would you expect to need to perform the µ-law to two’s-complement conversion of Exercise 15.12?

15.16 Continuing from Exercise 15.15, write a Verilog module `Vru2lin_rom` that does the conversion using distributed LUTs as ROM. Your module should include an `initial` block to generate the ROM contents using the conversion formula. Target your module to an FPGA and synthesize it.

How many LUTs does your module actually require, and how many bits of LUT “ROM” is that? Explain any discrepancy with what you expected (you may be able to look at the synthesized schematic for clues). *Optional*: Compare the worst-case delay of this module and the one in Exercise 15.12, in terms of both the worst-case number of LUT levels and the tool’s predicted propagation delays.

15.17 Read the documentation for your FPGA synthesis tools and learn how to create a ROM whose bits are distributed among multiple LUTs. In your targeted FPGA family, how many "LUTs as ROM" would you expect to need to perform the two's-complement to µ-law conversion of Exercise 15.13?

15.18 Continuing from Exercise 15.17, write a Verilog module `Vrlin2u_rom` that does the conversion using distributed LUTs as ROM. Your module may include an `initial` block to generate the ROM contents, or you may write a program in another language to generate the ROM contents. Either way, if a linear input value falls between two linear values that correspond exactly with a µ-law-coded value, as most will, ensure that your code selects the µ-law value that corresponds most closely with the input value. Use the generated µ-law values to initialize the ROM in your Verilog module, target it to an FPGA, and synthesize it.

How many LUTs does your module actually require, and how many bits of LUT "ROM" is that? Explain any discrepancy with what you expected (you may be able to look at the synthesized schematic for clues). *Optional*: Compare the worst-case delay of this module and the one in Exercise 15.13, in terms of both the worst-case number of LUT levels and the tool's predicted propagation delays.

15.19 Continuing from Exercise 15.18, reduce the size of the module and eliminate over half of the ROM, by including ROM only to perform linear to µ-law conversion for positive values. Provide additional code to handle negative values. Compare the size of the new module (number of LUTs) with the original. Also compare the delay in terms of both the worst-case number of LUT levels and the tool's predicted propagation delays.

15.20 In the style of Figure 15-23, draw the timing diagram for a late-write SSRAM with flow-through outputs for a series of interleaved reads and writes in the pattern R-R-W-W-R-R-W-W. Run the individual cycles as close together as possible, but be sure to account for resource conflicts that prevent back-to-back cycles. What is the average utilization of the SRAM array if the SSRAM is presented with a continuous stream of such requests?

15.21 Repeat the preceding exercise for a late-write SSRAM with pipelined outputs.

15.22 The Xilinx 7-series FPGAs scale in one other dimension, not discussed in the text or shown in Figure 15-34, to create even larger devices. Investigate this online, write a paragraph or two explaining it, and give the characteristics of at least three devices that utilize this dimension, in the style of Table 15-3.

15.23 Modify the multiplier module of Program 8-16 so it loads inputs `X` and `Y` into 8-bit registers on the rising edge of an external clock `CLK`, and places the resulting product `P` into a 16-bit register on the next rising edge of `CLK`. Read the appropriate parts of the Xilinx 7-series and Vivado documentation and determine how to force a Verilog design that has registered inputs and outputs to use the registers in the IOBs. Target your multiplier design to a 7-series FPGA, and apply your knowledge to use IOB registers for `X`, `Y`, and `P`.

INDEX

Note: Page numbers for defining references are given in color.