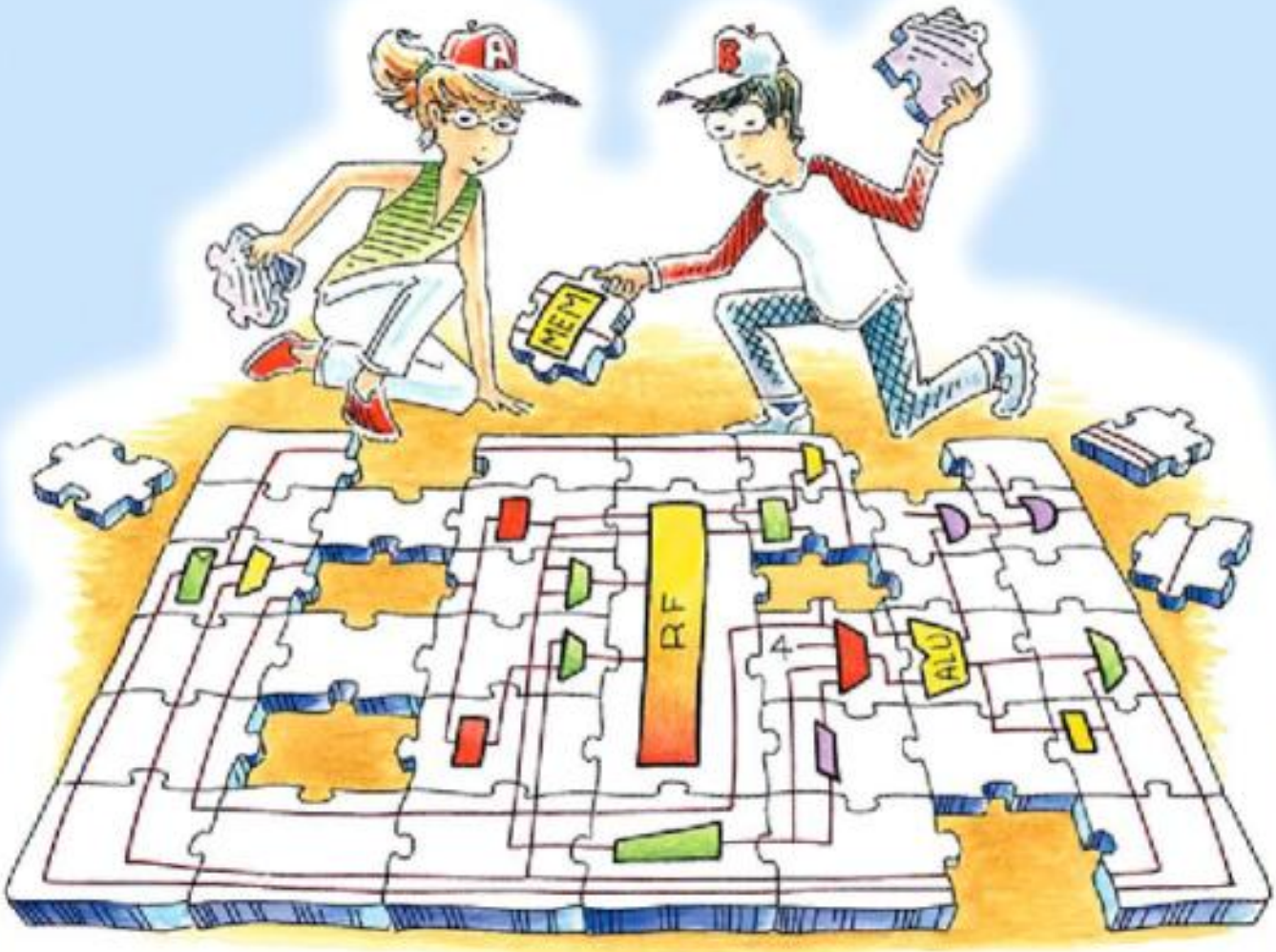
Digital Design and Computer Architecture



David Money Harris & Sarah L. Harris

MORGAN KAUFMANN

About the Authors

David Money Harris is an associate professor of engineering at Harvey Mudd College. He received his Ph.D. in electrical engineering from Stanford University and his M.Eng. in electrical engineering and computer science from MIT. Before attending Stanford, he worked at Intel as a logic and circuit designer on the Itanium and Pentium II processors. Since then, he has consulted at Sun Microsystems, Hewlett-Packard, Evans & Sutherland, and other design companies.

David's passions include teaching, building chips, and exploring the outdoors. When he is not at work, he can usually be found hiking, mountaineering, or rock climbing. He particularly enjoys hiking with his son, Abraham, who was born at the start of this book project. David holds about a dozen patents and is the author of three other textbooks on chip design, as well as two guidebooks to the Southern California mountains.

Sarah L. Harris is an assistant professor of engineering at Harvey Mudd College. She received her Ph.D. and M.S. in electrical engineering from Stanford University. Before attending Stanford, she received a B.S. in electrical and computer engineering from Brigham Young University. Sarah has also worked with Hewlett-Packard, the San Diego Supercomputer Center, Nvidia, and Microsoft Research in Beijing.

Sarah loves teaching, exploring and developing new technologies, traveling, wind surfing, rock climbing, and playing the guitar. Her recent exploits include researching sketching interfaces for digital circuit design, acting as a science correspondent for a National Public Radio affiliate, and learning how to kite surf. She speaks four languages and looks forward to adding a few more to the list in the near future.

Preface

Why publish yet another book on digital design and computer architecture? There are dozens of good books in print on digital design. There are also several good books about computer architecture, especially the classic texts of Patterson and Hennessy. This book is unique in its treatment in that it presents digital logic design from the perspective of computer architecture, starting at the beginning with 1's and 0's, and leading students through the design of a MIPS microprocessor.

We have used several editions of Patterson and Hennessy's *Computer Organization and Design* (*COD*) for many years at Harvey Mudd College. We particularly like their coverage of the MIPS architecture and microarchitecture because MIPS is a commercially successful microprocessor architecture, yet it is simple enough to clearly explain and build in an introductory class. Because our class has no prerequisites, the first half of the semester is dedicated to digital design, which is not covered by *COD*. Other universities have indicated a need for a book that combines digital design and computer architecture. We have undertaken to prepare such a book.

We believe that building a microprocessor is a special rite of passage for engineering and computer science students. The inner workings of a processor seem almost magical to the uninitiated, yet prove to be straightforward when carefully explained. Digital design in itself is a powerful and exciting subject. Assembly language programming unveils the inner language spoken by the processor. Microarchitecture is the link that brings it all together.

This book is suitable for a rapid-paced, single-semester introduction to digital design and computer architecture or for a two-quarter or two-semester sequence giving more time to digest the material and experiment in the lab. The only prerequisite is basic familiarity with a high-level programming language such as C, C++, or Java. The material is usually taught at the sophomore- or junior-year level, but may also be accessible to bright freshmen who have some programming experience.

FEATURES

This book offers a number of special features.

Side-by-Side Coverage of Verilog and VHDL

Hardware description languages (HDLs) are at the center of modern digital design practices. Unfortunately, designers are evenly split between the two dominant languages, Verilog and VHDL. This book introduces HDLs in Chapter 4 as soon as combinational and sequential logic design has been covered. HDLs are then used in Chapters 5 and 7 to design larger building blocks and entire processors. Nevertheless, Chapter 4 can be skipped and the later chapters are still accessible for courses that choose not to cover HDLs.

This book is unique in its side-by-side presentation of Verilog and VHDL, enabling the reader to quickly compare and contrast the two languages. Chapter 4 describes principles applying to both HDLs, then provides language-specific syntax and examples in adjacent columns. This side-by-side treatment makes it easy for an instructor to choose either HDL, and for the reader to transition from one to the other, either in a class or in professional practice.

Classic MIPS Architecture and Microarchitecture

Chapters 6 and 7 focus on the MIPS architecture adapted from the treatment of Patterson and Hennessy. MIPS is an ideal architecture because it is a real architecture shipped in millions of products yearly, yet it is streamlined and easy to learn. Moreover, hundreds of universities around the world have developed pedagogy, labs, and tools around the MIPS architecture.

Real-World Perspectives

Chapters 6, 7, and 8 illustrate the architecture, microarchitecture, and memory hierarchy of Intel IA-32 processors. These real-world perspective chapters show how the concepts in the chapter relate to the chips found in most PCs.

Accessible Overview of Advanced Microarchitecture

Chapter 7 includes an overview of modern high-performance microarchitectural features including branch prediction, superscalar and out-of-order operation, multithreading, and multicore processors. The treatment is accessible to a student in a first course and shows how the microarchitectures in the book can be extended to modern processors.

End-of-Chapter Exercises and Interview Questions

The best way to learn digital design is to do it. Each chapter ends with numerous exercises to practice the material. The exercises are followed by a set of interview questions that our industrial colleagues have asked students applying for work in the field. These questions provide a helpful

glimpse into the types of problems job applicants will typically encounter during the interview process. (Exercise solutions are available via the book's companion and instructor Web pages. For more details, see the next section, Online Supplements.)

ONLINE SUPPLEMENTS

Supplementary materials are available online at *textbooks.elsevier.com/9780123704979*. This companion site (accessible to all readers) includes:

- Solutions to odd-numbered exercises
- Links to professional-strength computer-aided design (CAD) tools from Xilinx® and Synplicity®
- Link to PCSPIM, a Windows-based MIPS simulator
- Hardware description language (HDL) code for the MIPS processor
- Xilinx Project Navigator helpful hints
- Lecture slides in PowerPoint (PPT) format
- Sample course and lab materials
- List of errata

The instructor site (linked to the companion site and accessible to adopters who register at *textbooks.elsevier.com*) includes:

- Solutions to even-numbered exercises
- Links to professional-strength computer-aided design (CAD) tools from Xilinx® and Synplicity®. (Instructors from qualified universities can access *free* Synplicity tools for use in their classroom and laboratories. More details are available at the instructor site.)
- Figures from the text in JPG and PPT formats

Additional details on using the Xilinx, Synplicity, and PCSPIM tools in your course are provided in the next section. Details on the sample lab materials are also provided here.

HOW TO USE THE SOFTWARE TOOLS IN A COURSE

Xilinx ISE WebPACK

Xilinx ISE WebPACK is a free version of the professional-strength Xilinx ISE Foundation FPGA design tools. It allows students to enter their digital designs in schematic or using either the Verilog or VHDL hardware description language (HDL). After entering the design, students can

simulate their circuits using ModelSim MXE III Starter, which is included in the Xilinx WebPACK. Xilinx WebPACK also includes XST, a logic synthesis tool supporting both Verilog and VHDL.

The difference between WebPACK and Foundation is that WebPACK supports a subset of the most common Xilinx FPGAs. The difference between ModelSim MXE III Starter and ModelSim commercial versions is that Starter degrades performance for simulations with more than 10,000 lines of HDL.

Synplify Pro

Synplify Pro® is a high-performance, sophisticated logic synthesis engine for FPGA and CPLD designs. Synplify Pro also contains HDL Analyst, a graphical interface tool that generates schematic views of the HDL source code. We have found that this is immensely useful in the learning and debugging process.

Synplicity has generously agreed to donate Synplify Pro to qualified universities and will provide as many licenses as needed to fill university labs. Instructors should visit the instructor Web page for this text for more information on how to request Synplify Pro licenses. For additional information on Synplicity and its other software, visit *www.synplicity.com/university*.

PCSPIM

PCSPIM, also called simply SPIM, is a Windows-based MIPS simulator that runs MIPS assembly code. Students enter their MIPS assembly code into a text file and run it using PCSPIM. PCSPIM displays the instructions, memory, and register values. Links to the user's manual and an example file are available at the companion site (textbooks.elsevier.com/9780123704979).

LABS

The companion site includes links to a series of labs that cover topics from digital design through computer architecture. The labs teach students how to use the Xilinx WebPACK or Foundation tools to enter, simulate, synthesize, and implement their designs. The labs also include topics on assembly language programming using the PCSPIM simulator.

After synthesis, students can implement their designs using the Digilent Spartan 3 Starter Board or the XUP-Virtex 2 Pro (V2Pro) Board. Both of these powerful and competitively priced boards are available from *www.digilentinc.com*. The boards contain FPGAs that can be programmed to implement student designs. We provide labs that describe how to implement a selection of designs using Digilent's Spartan 3 Board using

WebPACK. Unfortunately, Xilinx WebPACK does not support the huge FPGA on the V2Pro board. Qualified universities may contact the Xilinx University Program to request a donation of the full Foundation tools.

To run the labs, students will need to download and install the Xilinx WebPACK, PCSPIM, and possibly Synplify Pro. Instructors may also choose to install the tools on lab machines. The labs include instructions on how to implement the projects on the Digilent's Spartan 3 Starter Board. The implementation step may be skipped, but we have found it of great value. The labs will also work with the XST synthesis tool, but we recommend using Synplify Pro because the schematics it produces give students invaluable feedback.

We have tested the labs on Windows, but the tools are also available for Linux.

BUGS

As all experienced programmers know, any program of significant complexity undoubtedly contains bugs. So too do books. We have taken great care to find and squash the bugs in this book. However, some errors undoubtedly do remain. We will maintain a list of errata on the book's Web page.

Please send your bug reports to *ddcabugs@onehotlogic.com*. The first person to report a substantive bug with a fix that we use in a future printing will be rewarded with a $1 bounty! (Be sure to include your mailing address.)

ACKNOWLEDGMENTS

First and foremost, we thank David Patterson and John Hennessy for their pioneering MIPS microarchitectures described in their *Computer Organization and Design* textbook. We have taught from various editions of their book for many years. We appreciate their gracious support of this book and their permission to build on their microarchitectures.

Duane Bibby, our favorite cartoonist, labored long and hard to illustrate the fun and adventure of digital design. We also appreciate the enthusiasm of Denise Penrose, Nate McFadden, and the rest of the team at Morgan Kaufmann who made this book happen. Jeff Somers at Graphic World Publishing Services has ably guided the book through production.

Numerous reviewers have substantially improved the book. They include John Barr (Ithaca College), Jack V. Briner (Charleston Southern University), Andrew C. Brown (SK Communications), Carl Baumgaertner (Harvey Mudd College), A. Utku Diril (Nvidia Corporation), Jim Frenzel (University of Idaho), Jaeha Kim (Rambus, Inc.), Phillip King

(ShotSpotter, Inc.), James Pinter-Lucke (Claremont McKenna College), Amir Roth, Z. Jerry Shi (University of Connecticut), James E. Stine (Oklahoma State University), Luke Teyssier, Peiyi Zhao (Chapman University), and an anonymous reviewer. Simon Moore was a wonderful host during David's sabbatical visit to Cambridge University, where major sections of this book were written.

We also appreciate the students in our course at Harvey Mudd College who have given us helpful feedback on drafts of this textbook. Of special note are Casey Schilling, Alice Clifton, Chris Acon, and Stephen Brawner.

I, David, particularly thank my wife, Jennifer, who gave birth to our son Abraham at the beginning of the project. I appreciate her patience and loving support through yet another project at a busy time in our lives.

From Zero to One

1

1.1 THE GAME PLAN

Microprocessors have revolutionized our world during the past three decades. A laptop computer today has far more capability than a room-sized mainframe of yesteryear. A luxury automobile contains about 50 microprocessors. Advances in microprocessors have made cell phones and the Internet possible, have vastly improved medicine, and have transformed how war is waged. Worldwide semiconductor industry sales have grown from US $21 billion in 1985 to $227 billion in 2005, and microprocessors are a major segment of these sales. We believe that microprocessors are not only technically, economically, and socially important, but are also an intrinsically fascinating human invention. By the time you finish reading this book, you will know how to design and build your own microprocessor. The skills you learn along the way will prepare you to design many other digital systems.

We assume that you have a basic familiarity with electricity, some prior programming experience, and a genuine interest in understanding what goes on under the hood of a computer. This book focuses on the design of digital systems, which operate on 1's and 0's. We begin with digital logic gates that accept 1's and 0's as inputs and produce 1's and 0's as outputs. We then explore how to combine logic gates into more complicated modules such as adders and memories. Then we shift gears to programming in assembly language, the native tongue of the microprocessor. Finally, we put gates together to build a microprocessor that runs these assembly language programs.

A great advantage of digital systems is that the building blocks are quite simple: just 1's and 0's. They do not require grungy mathematics or a profound knowledge of physics. Instead, the designer's challenge is to combine these simple blocks into complicated systems. A microprocessor may be the first system that you build that is too complex to fit in your

head all at once. One of the major themes weaved through this book is how to manage complexity.

1.2 THE ART OF MANAGING COMPLEXITY

One of the characteristics that separates an engineer or computer scientist from a layperson is a systematic approach to managing complexity. Modern digital systems are built from millions or billions of transistors. No human being could understand these systems by writing equations describing the movement of electrons in each transistor and solving all of the equations simultaneously. You will need to learn to manage complexity to understand how to build a microprocessor without getting mired in a morass of detail.

1.2.1 Abstraction

The critical technique for managing complexity is *abstraction*: hiding details when they are not important. A system can be viewed from many different levels of abstraction. For example, American politicians abstract the world into cities, counties, states, and countries. A county contains multiple cities and a state contains many counties. When a politician is running for president, the politician is mostly interested in how the state as a whole will vote, rather than how each county votes, so the state is the most useful level of abstraction. On the other hand, the Census Bureau measures the population of every city, so the agency must consider the details of a lower level of abstraction.

Figure 1.1 illustrates levels of abstraction for an electronic computer system along with typical building blocks at each level. At the lowest level of abstraction is the physics, the motion of electrons. The behavior of electrons is described by quantum mechanics and Maxwell's equations. Our system is constructed from electronic *devices* such as transistors (or vacuum tubes, once upon a time). These devices have well-defined connection points called *terminals* and can be modeled by the relationship between voltage and current as measured at each terminal. By abstracting to this device level, we can ignore the individual electrons. The next level of abstraction is *analog circuits*, in which devices are assembled to create components such as amplifiers. Analog circuits input and output a continuous range of voltages. *Digital circuits* such as logic gates restrict the voltages to discrete ranges, which we will use to indicate 0 and 1. In logic design, we build more complex structures, such as adders or memories, from digital circuits.

Microarchitecture links the logic and architecture levels of abstraction. The *architecture* level of abstraction describes a computer from the programmer's perspective. For example, the Intel IA-32 architecture used by microprocessors in most *personal computers* (*PCs*) is defined by a set of

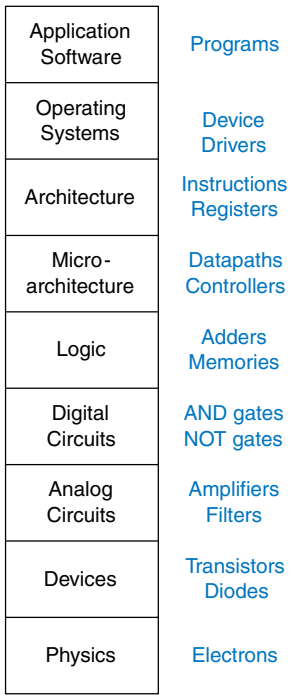


Figure 1.1 Levels of abstraction for electronic computing system

instructions and registers (memory for temporarily storing variables) that the programmer is allowed to use. Microarchitecture involves combining logic elements to execute the instructions defined by the architecture. A particular architecture can be implemented by one of many different microarchitectures with different price/performance/power trade-offs. For example, the Intel Core 2 Duo, the Intel 80486, and the AMD Athlon all implement the IA-32 architecture with different microarchitectures.

Moving into the software realm, the operating system handles low-level details such as accessing a hard drive or managing memory. Finally, the application software uses these facilities provided by the operating system to solve a problem for the user. Thanks to the power of abstraction, your grandmother can surf the Web without any regard for the quantum vibrations of electrons or the organization of the memory in her computer.

This book focuses on the levels of abstraction from digital circuits through computer architecture. When you are working at one level of abstraction, it is good to know something about the levels of abstraction immediately above and below where you are working. For example, a computer scientist cannot fully optimize code without understanding the architecture for which the program is being written. A device engineer cannot make wise trade-offs in transistor design without understanding the circuits in which the transistors will be used. We hope that by the time you finish reading this book, you can pick the level of abstraction appropriate to solving your problem and evaluate the impact of your design choices on other levels of abstraction.

1.2.2 Discipline

Discipline is the act of intentionally restricting your design choices so that you can work more productively at a higher level of abstraction. Using interchangeable parts is a familiar application of discipline. One of the first examples of interchangeable parts was in flintlock rifle manufacturing. Until the early 19th century, rifles were individually crafted by hand. Components purchased from many different craftsmen were carefully filed and fit together by a highly skilled gunmaker. The discipline of interchangeable parts revolutionized the industry. By limiting the components to a standardized set with well-defined tolerances, rifles could be assembled and repaired much faster and with less skill. The gunmaker no longer concerned himself with lower levels of abstraction such as the specific shape of an individual barrel or gunstock.

In the context of this book, the digital discipline will be very important. Digital circuits use discrete voltages, whereas analog circuits use continuous voltages. Therefore, digital circuits are a subset of analog circuits and in some sense must be capable of less than the broader class of analog circuits. However, digital circuits are much simpler to design. By limiting

ourselves to digital circuits, we can easily combine components into sophisticated systems that ultimately outperform those built from analog components in many applications. For example, digital televisions, compact disks (CDs), and cell phones are replacing their analog predecessors.

1.2.3 The Three -Y's

In addition to abstraction and discipline, designers use the three "-y's" to manage complexity: hierarchy, modularity, and regularity. These principles apply to both software and hardware systems.

- *Hierarchy* involves dividing a system into modules, then further subdividing each of these modules until the pieces are easy to understand.
- *Modularity* states that the modules have well-defined functions and interfaces, so that they connect together easily without unanticipated side effects.
- *Regularity* seeks uniformity among the modules. Common modules are reused many times, reducing the number of distinct modules that must be designed.

To illustrate these "-y's" we return to the example of rifle manufacturing. A flintlock rifle was one of the most intricate objects in common use in the early 19th century. Using the principle of hierarchy, we can break it into components shown in Figure 1.2: the lock, stock, and barrel.

The barrel is the long metal tube through which the bullet is fired. The lock is the firing mechanism. And the stock is the wooden body that holds the parts together and provides a secure grip for the user. In turn, the lock contains the trigger, hammer, flint, frizzen, and pan. Each of these components could be hierarchically described in further detail.

Modularity teaches that each component should have a well-defined function and interface. A function of the stock is to mount the barrel and lock. Its interface consists of its length and the location of its mounting pins. In a modular rifle design, stocks from many different manufacturers can be used with a particular barrel as long as the stock and barrel are of the correct length and have the proper mounting mechanism. A function of the barrel is to impart spin to the bullet so that it travels more accurately. Modularity dictates that there should be no side effects: the design of the stock should not impede the function of the barrel.

Regularity teaches that interchangeable parts are a good idea. With regularity, a damaged barrel can be replaced by an identical part. The barrels can be efficiently built on an assembly line, instead of being painstakingly hand-crafted.

We will return to these principles of hierarchy, modularity, and regularity throughout the book.

Captain Meriwether Lewis of the Lewis and Clark Expedition was one of the early advocates of interchangeable parts for rifles. In 1806, he explained:

> The guns of Drewyer and Sergt. Pryor were both out of order. The first was repared with a new lock, the old one having become unfit for use; the second had the cock screw broken which was replaced by a duplicate which had been prepared for the lock at Harpers Ferry where she was manufactured. But for the precaution taken in bringing on those extra locks, and parts of locks, in addition to the ingenuity of John Shields, most of our guns would at this moment been entirely unfit for use; but fortunately for us I have it in my power here to record that they are all in good order.

See Elliott Coues, ed., *The History of the Lewis and Clark Expedition*... (4 vols), New York: Harper, 1893; reprint, 3 vols, New York: Dover, 3:817.

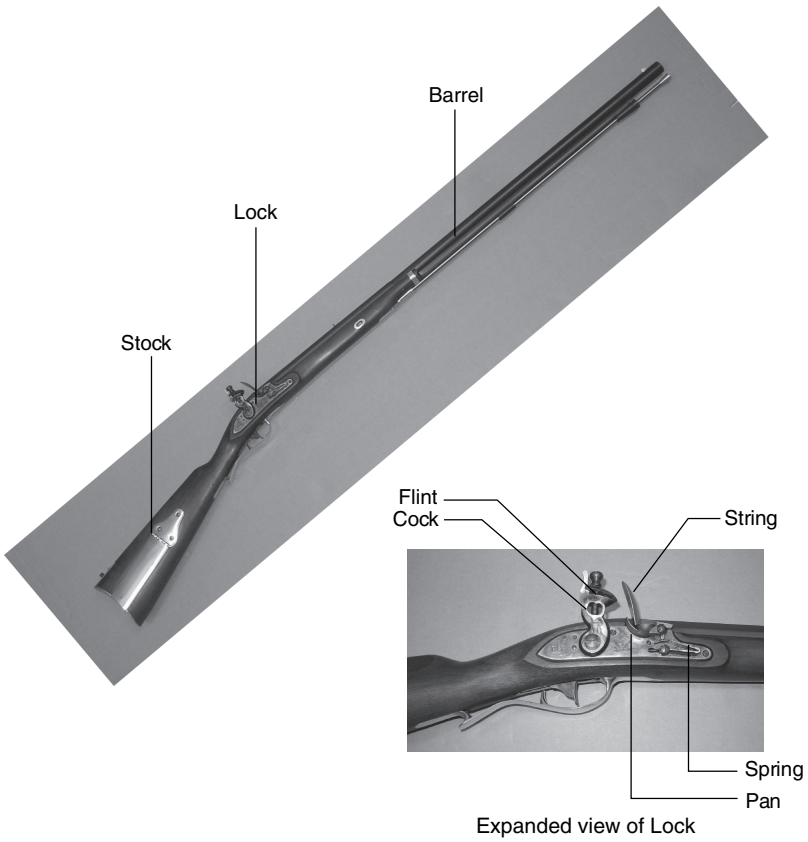
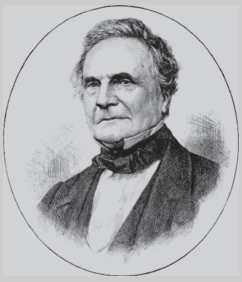


Figure 1.2 Flintlock rifle with a close-up view of the lock (Image by Euroams Italia. www.euroarms.net © 2006).



Charles Babbage, 1791–1871. Attended Cambridge University and married Georgiana Whitmore in 1814. Invented the Analytical Engine, the world's first mechanical computer. Also invented the cowcatcher and the universal postage rate. Interested in lock-picking, but abhorred street musicians (image courtesy of Fourmilab Switzerland, www.fourmilab.ch).

1.3 THE DIGITAL ABSTRACTION

Most physical variables are continuous. For example, the voltage on a wire, the frequency of an oscillation, or the position of a mass are all continuous quantities. Digital systems, on the other hand, represent information with *discrete-valued variables*—that is, variables with a finite number of distinct values.

An early digital system using variables with ten discrete values was Charles Babbage's Analytical Engine. Babbage labored from 1834 to 1871,[1] designing and attempting to build this mechanical computer. The Analytical Engine used gears with ten positions labeled 0 through 9, much like a mechanical odometer in a car. Figure 1.3 shows a prototype

[1] And we thought graduate school was long!

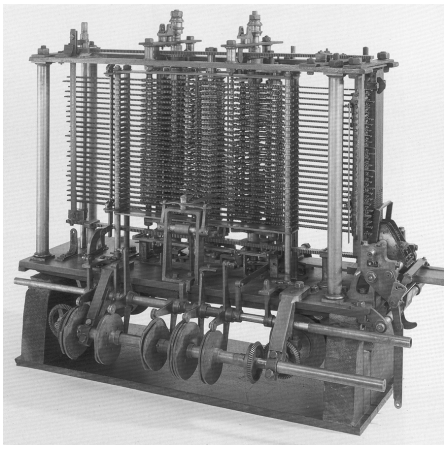


Figure 1.3 Babbage's Analytical Engine, under construction at the time of his death in 1871 (image courtesy of Science Museum/Science and Society Picture Library).

of the Analytical Engine, in which each row processes one digit. Babbage chose 25 rows of gears, so the machine has 25-digit precision.

Unlike Babbage's machine, most electronic computers use a binary (two-valued) representation in which a high voltage indicates a '1' and a low voltage indicates a '0,' because it is easier to distinguish between two voltages than ten.

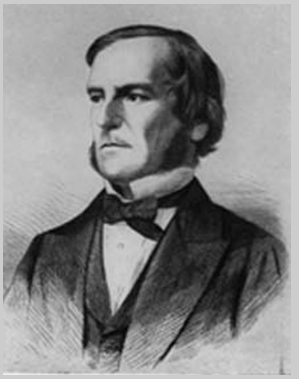
The *amount of information* D in a discrete valued variable with N distinct states is measured in units of *bits* as

$$D = \log_2 N \text{ bits} \tag{1.1}$$

A binary variable conveys $\log_2 2 = 1$ bit of information. Indeed, the word bit is short for *b*inary dig*it*. Each of Babbage's gears carried $\log_2 10 = 3.322$ bits of information because it could be in one of $2^{3.322} = 10$ unique positions. A continuous signal theoretically contains an infinite amount of information because it can take on an infinite number of values. In practice, noise and measurement error limit the information to only 10 to 16 bits for most continuous signals. If the measurement must be made rapidly, the information content is lower (e.g., 8 bits).

This book focuses on digital circuits using binary variables: 1's and 0's. George Boole developed a system of logic operating on binary variables that is now known as *Boolean logic*. Each of Boole's variables could be TRUE or FALSE. Electronic computers commonly use a positive voltage to represent '1' and zero volts to represent '0'. In this book, we will use the terms '1,' TRUE, and HIGH synonymously. Similarly, we will use '0,' FALSE, and LOW interchangeably.

The beauty of the *digital abstraction* is that digital designers can focus on 1's and 0's, ignoring whether the Boolean variables are physically represented with specific voltages, rotating gears, or even hydraulic



George Boole, 1815–1864. Born to working-class parents and unable to afford a formal education, Boole taught himself mathematics and joined the faculty of Queen's College in Ireland. He wrote *An Investigation of the Laws of Thought* (1854), which introduced binary variables and the three fundamental logic operations: AND, OR, and NOT (image courtesy of xxx).

fluid levels. A computer programmer can work without needing to know the intimate details of the computer hardware. On the other hand, understanding the details of the hardware allows the programmer to optimize the software better for that specific computer.

An individual bit doesn't carry much information. In the next section, we examine how groups of bits can be used to represent numbers. In later chapters, we will also use groups of bits to represent letters and programs.

1.4 NUMBER SYSTEMS

You are accustomed to working with decimal numbers. In digital systems consisting of 1's and 0's, binary or hexadecimal numbers are often more convenient. This section introduces the various number systems that will be used throughout the rest of the book.

1.4.1 Decimal Numbers

In elementary school, you learned to count and do arithmetic in *decimal*. Just as you (probably) have ten fingers, there are ten decimal digits, 0, 1, 2, ..., 9. Decimal digits are joined together to form longer decimal numbers. Each column of a decimal number has ten times the weight of the previous column. From right to left, the column weights are 1, 10, 100, 1000, and so on. Decimal numbers are referred to as *base 10*. The base is indicated by a subscript after the number to prevent confusion when working in more than one base. For example, Figure 1.4 shows how the decimal number 9742_{10} is written as the sum of each of its digits multiplied by the weight of the corresponding column.

An N-digit decimal number represents one of 10^N possibilities: 0, 1, 2, 3, ..., 10^{N-1}. This is called the *range* of the number. For example, a three-digit decimal number represents one of 1000 possibilities in the range of 0 to 999.

1.4.2 Binary Numbers

Bits represent one of two values, 0 or 1, and are joined together to form *binary numbers*. Each column of a binary number has twice the weight

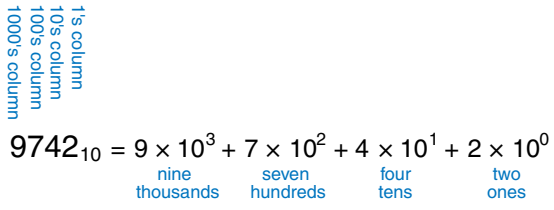


Figure 1.4 Representation of a decimal number

of the previous column, so binary numbers are *base 2*. In binary, the column weights (again from right to left) are 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, and so on. If you work with binary numbers often, you'll save time if you remember these powers of two up to 2^{16}.

An N-bit binary number represents one of 2^N possibilities: 0, 1, 2, 3, ..., 2^{N-1}. Table 1.1 shows 1, 2, 3, and 4-bit binary numbers and their decimal equivalents.

Example 1.1 BINARY TO DECIMAL CONVERSION

Convert the binary number 10110_2 to decimal.

Solution: Figure 1.5 shows the conversion.

Table 1.1 Binary numbers and their decimal equivalent

1-Bit Binary Numbers	2-Bit Binnary Numbers	3-Bit Binary Numbers	4-Bit Binary Numbers	Decimal Equivalents
0	00	000	0000	0
1	01	001	0001	1
	10	010	0010	2
	11	011	0011	3
		100	0100	4
		101	0101	5
		110	0110	6
		111	0111	7
			1000	8
			1001	9
			1010	10
			1011	11
			1100	12
			1101	13
			1110	14
			1111	15

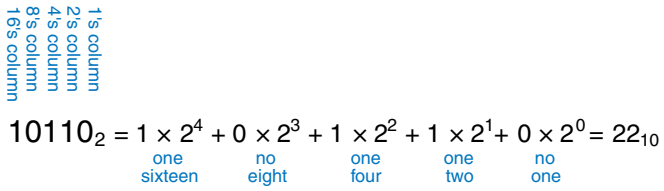


Figure 1.5 Conversion of a binary number to decimal

Example 1.2 DECIMAL TO BINARY CONVERSION

Convert the decimal number 84_{10} to binary.

Solution: Determine whether each column of the binary result has a 1 or a 0. We can do this starting at either the left or the right column.

Working from the left, start with the largest power of 2 less than the number (in this case, 64). $84 \geq 64$, so there is a 1 in the 64's column, leaving $84 - 64 = 20$. $20 < 32$, so there is a 0 in the 32's column. $20 \geq 16$, so there is a 1 in the 16's column, leaving $20 - 16 = 4$. $4 < 8$, so there is a 0 in the 8's column. $4 \geq 4$, so there is a 1 in the 4's column, leaving $4 - 4 = 0$. Thus there must be 0's in the 2's and 1's column. Putting this all together, $84_{10} = 1010100_2$.

Working from the right, repeatedly divide the number by 2. The remainder goes in each column. $84/2 = 42$, so 0 goes in the 1's column. $42/2 = 21$, so 0 goes in the 2's column. $21/2 = 10$ with a remainder of 1 going in the 4's column. $10/2 = 5$, so 0 goes in the 8's column. $5/2 = 2$ with a remainder of 1 going in the 16's column. $2/2 = 1$, so 0 goes in the 32's column. Finally $1/2 = 0$ with a remainder of 1 going in the 64's column. Again, $84_{10} = 1010100_2$

1.4.3 Hexadecimal Numbers

Writing long binary numbers becomes tedious and prone to error. A group of four bits represents one of $2^4 = 16$ possibilities. Hence, it is sometimes more convenient to work in *base 16*, called *hexadecimal*. Hexadecimal numbers use the digits 0 to 9 along with the letters A to F, as shown in Table 1.2. Columns in base 16 have weights of 1, 16, 16^2 (or 256), 16^3 (or 4096), and so on.

"Hexadecimal," a term coined by IBM in 1963, derives from the Greek *hexi* (six) and Latin *decem* (ten). A more proper term would use the Latin *sexa* (six), but *sexidecimal* sounded too risqué.

Example 1.3 HEXADECIMAL TO BINARY AND DECIMAL CONVERSION

Convert the hexadecimal number $2ED_{16}$ to binary and to decimal.

Solution: Conversion between hexadecimal and binary is easy because each hexadecimal digit directly corresponds to four binary digits. $2_{16} = 0010_2$, $E_{16} = 1110_2$ and $D_{16} = 1101_2$, so $2ED_{16} = 001011101101_2$. Conversion to decimal requires the arithmetic shown in Figure 1.6.

Table 1.2 Hexadecimal number system

Hexadecimal Digit	Decimal Equivalent	Binary Equivalent
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

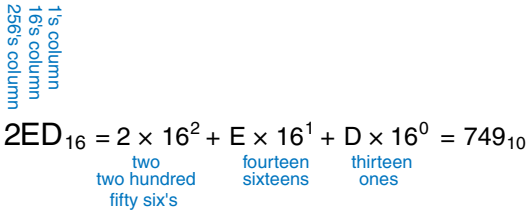


Figure 1.6 Conversion of hexadecimal number to decimal

Example 1.4 BINARY TO HEXADECIMAL CONVERSION

Convert the binary number 1111010_2 to hexadecimal.

Solution: Again, conversion is easy. Start reading from the right. The four least significant bits are $1010_2 = A_{16}$. The next bits are $111_2 = 7_{16}$. Hence $1111010_2 = 7A_{16}$.

Example 1.5 DECIMAL TO HEXADECIMAL AND BINARY CONVERSION

Convert the decimal number 333_{10} to hexadecimal and binary.

Solution: Like decimal to binary conversion, decimal to hexadecimal conversion can be done from the left or the right.

Working from the left, start with the largest power of 16 less than the number (in this case, 256). 256 goes into 333 once, so there is a 1 in the 256's column, leaving $333 - 256 = 77$. 16 goes into 77 four times, so there is a 4 in the 16's column, leaving $77 - 16 \times 4 = 13$. $13_{10} = D_{16}$, so there is a D in the 1's column. In summary, $333_{10} = 14D_{16}$. Now it is easy to convert from hexadecimal to binary, as in Example 1.3. $14D_{16} = 101001101_2$.

Working from the right, repeatedly divide the number by 16. The remainder goes in each column. 333/16 = 20 with a remainder of $13_{10} = D_{16}$ going in the 1's column. 20/16 = 1 with a remainder of 4 going in the 16's column. 1/16 = 0 with a remainder of 1 going in the 256's column. Again, the result is $14D_{16}$.

1.4.4 Bytes, Nibbles, and All That Jazz

A group of eight bits is called a *byte*. It represents one of $2^8 = 256$ possibilities. The size of objects stored in computer memories is customarily measured in bytes rather than bits.

A group of four bits, or half a byte, is called a *nibble*. It represents one of $2^4 = 16$ possibilities. One hexadecimal digit stores one nibble and two hexadecimal digits store one full byte. Nibbles are no longer a commonly used unit, but the term is cute.

Microprocessors handle data in chunks called *words*. The size of a word depends on the architecture of the microprocessor. When this chapter was written in 2006, most computers had 32-bit processors, indicating that they operate on 32-bit words. At the time, computers handling 64-bit words were on the verge of becoming widely available. Simpler microprocessors, especially those used in gadgets such as toasters, use 8- or 16-bit words.

Within a group of bits, the bit in the 1's column is called the *least significant bit* (*lsb*), and the bit at the other end is called the *most significant bit* (*msb*), as shown in Figure 1.7(a) for a 6-bit binary number. Similarly, within a word, the bytes are identified as *least significant byte* (*LSB*) through *most significant byte* (*MSB*), as shown in Figure 1.7(b) for a four-byte number written with eight hexadecimal digits.

A *microprocessor* is a processor built on a single chip. Until the 1970's, processors were too complicated to fit on one chip, so mainframe processors were built from boards containing many chips. Intel introduced the first 4-bit microprocessor, called the 4004, in 1971. Now, even the most sophisticated supercomputers are built using microprocessors. We will use the terms microprocessor and processor interchangeably throughout this book.

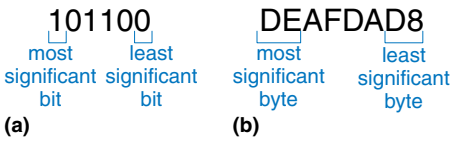


Figure 1.7 Least and most significant bits and bytes

By handy coincidence, $2^{10} = 1024 \approx 10^3$. Hence, the term *kilo* (Greek for thousand) indicates 2^{10}. For example, 2^{10} bytes is one kilobyte (1 KB). Similarly, *mega* (million) indicates $2^{20} \approx 10^6$, and *giga* (billion) indicates $2^{30} \approx 10^9$. If you know $2^{10} \approx 1$ thousand, $2^{20} \approx 1$ million, $2^{30} \approx 1$ billion, and remember the powers of two up to 2^9, it is easy to estimate any power of two in your head.

Example 1.6 ESTIMATING POWERS OF TWO

Find the approximate value of 2^{24} without using a calculator.

Solution: Split the exponent into a multiple of ten and the remainder. $2^{24} = 2^{20} \times 2^4$. $2^{20} \approx 1$ million. $2^4 = 16$. So $2^{24} \approx 16$ million. Technically, $2^{24} = 16{,}777{,}216$, but 16 million is close enough for marketing purposes.

1024 bytes is called a *kilobyte* (KB). 1024 bits is called a *kilobit* (Kb or Kbit). Similarly, MB, Mb, GB, and Gb are used for millions and billions of bytes and bits. Memory capacity is usually measured in bytes. Communication speed is usually measured in bits/sec. For example, the maximum speed of a dial-up modem is usually 56 Kbits/sec.

1.4.5 Binary Addition

Binary addition is much like decimal addition, but easier, as shown in Figure 1.8. As in decimal addition, if the sum of two numbers is greater than what fits in a single digit, we *carry* a 1 into the next column. Figure 1.8 compares addition of decimal and binary numbers. In the right-most column of Figure 1.8(a), 7 + 9 = 16, which cannot fit in a single digit because it is greater than 9. So we record the 1's digit, 6, and carry the 10's digit, 1, over to the next column. Likewise, in binary, if the sum of two numbers is greater than 1, we carry the 2's digit over to the next column. For example, in the right-most column of Figure 1.8(b), the sum

Figure 1.8 Addition examples showing carries: (a) decimal (b) binary

$1 + 1 = 2_{10} = 10_2$ cannot fit in a single binary digit. So we record the 1's digit (0) and carry the 2's digit (1) of the result to the next column. In the second column, the sum is $1 + 1 + 1 = 3_{10} = 11_2$. Again, we record the 1's digit (1) and carry the 2's digit (1) to the next column. For obvious reasons, the bit that is carried over to the neighboring column is called the *carry bit*.

Example 1.7 BINARY ADDITION

Compute $0111_2 + 0101_2$.

Solution: Figure 1.9 shows that the sum is 1100_2. The carries are indicated in blue. We can check our work by repeating the computation in decimal. $0111_2 = 7_{10}$. $0101_2 = 5_{10}$. The sum is $12_{10} = 1100_2$.

```
  111
  0111
+ 0101
 1100
```

Figure 1.9 Binary addition example

Digital systems usually operate on a fixed number of digits. Addition is said to *overflow* if the result is too big to fit in the available digits. A 4-bit number, for example, has the range [0, 15]. 4-bit binary addition overflows if the result exceeds 15. The fifth bit is discarded, producing an incorrect result in the remaining four bits. Overflow can be detected by checking for a carry out of the most significant column.

Example 1.8 ADDITION WITH OVERFLOW

Compute $1101_2 + 0101_2$. Does overflow occur?

Solution: Figure 1.10 shows the sum is 10010_2. This result overflows the range of a 4-bit binary number. If it must be stored as four bits, the most significant bit is discarded, leaving the incorrect result of 0010_2. If the computation had been done using numbers with five or more bits, the result 10010_2 would have been correct.

```
 11 1
  1101
+ 0101
 10010
```

Figure 1.10 Binary addition example with overflow

1.4.6 Signed Binary Numbers

So far, we have considered only *unsigned* binary numbers that represent positive quantities. We will often want to represent both positive and negative numbers, requiring a different binary number system. Several schemes exist to represent *signed* binary numbers; the two most widely employed are called sign/magnitude and two's complement.

Sign/Magnitude Numbers

Sign/magnitude numbers are intuitively appealing because they match our custom of writing negative numbers with a minus sign followed by the magnitude. An N-bit sign/magnitude number uses the most significant bit

The $7 billion Ariane 5 rocket, launched on June 4, 1996, veered off course 40 seconds after launch, broke up, and exploded. The failure was caused when the computer controlling the rocket overflowed its 16-bit range and crashed.

The code had been extensively tested on the Ariane 4 rocket. However, the Ariane 5 had a faster engine that produced larger values for the control computer, leading to the overflow.

(Photograph courtesy ESA/CNES/ ARIANESPACE-Service Optique CS6.)

as the sign and the remaining $N - 1$ bits as the magnitude (absolute value). A sign bit of 0 indicates positive and a sign bit of 1 indicates negative.

Example 1.9 **SIGN/MAGNITUDE NUMBERS**

Write 5 and -5 as 4-bit sign/magnitude numbers

Solution: Both numbers have a magnitude of $5_{10} = 101_2$. Thus, $5_{10} = 0101_2$ and $-5_{10} = 1101_2$.

Unfortunately, ordinary binary addition does not work for sign/magnitude numbers. For example, using ordinary addition on $-5_{10} + 5_{10}$ gives $1101_2 + 0101_2 = 10010_2$, which is nonsense.

An N-bit sign/magnitude number spans the range $[-2^{N-1} + 1, 2^{N-1} - 1]$. Sign/magnitude numbers are slightly odd in that both $+0$ and -0 exist. Both indicate zero. As you may expect, it can be troublesome to have two different representations for the same number.

Two's Complement Numbers

Two's complement numbers are identical to unsigned binary numbers except that the most significant bit position has a weight of -2^{N-1} instead of 2^{N-1}. They overcome the shortcomings of sign/magnitude numbers: zero has a single representation, and ordinary addition works.

In two's complement representation, zero is written as all zeros: $00...000_2$. The most positive number has a 0 in the most significant position and 1's elsewhere: $01...111_2 = 2^{N-1} - 1$. The most negative number has a 1 in the most significant position and 0's elsewhere: $10...000_2 = -2^{N-1}$. And -1 is written as all ones: $11...111_2$.

Notice that positive numbers have a 0 in the most significant position and negative numbers have a 1 in this position, so the most significant bit can be viewed as the sign bit. However, the remaining bits are interpreted differently for two's complement numbers than for sign/magnitude numbers.

The sign of a two's complement number is reversed in a process called *taking the two's complement*. The process consists of inverting all of the bits in the number, then adding 1 to the least significant bit position. This is useful to find the representation of a negative number or to determine the magnitude of a negative number.

Example 1.10 **TWO'S COMPLEMENT REPRESENTATION OF A NEGATIVE NUMBER**

Find the representation of -2_{10} as a 4-bit two's complement number.

Solution: Start with $+2_{10} = 0010_2$. To get -2_{10}, invert the bits and add 1. Inverting 0010_2 produces 1101_2. $1101_2 + 1 = 1110_2$. So -2_{10} is 1110_2.

Example 1.11 VALUE OF NEGATIVE TWO'S COMPLEMENT NUMBERS

Find the decimal value of the two's complement number 1001_2.

Solution: 1001_2 has a leading 1, so it must be negative. To find its magnitude, invert the bits and add 1. Inverting $1001_2 = 0110_2$. $0110_2 + 1 = 0111_2 = 7_{10}$. Hence, $1001_2 = -7_{10}$.

Two's complement numbers have the compelling advantage that addition works properly for both positive and negative numbers. Recall that when adding N-bit numbers, the carry out of the Nth bit (i.e., the $N + 1^{\text{th}}$ result bit), is discarded.

Example 1.12 ADDING TWO'S COMPLEMENT NUMBERS

Compute (a) $-2_{10} + 1_{10}$ and (b) $-7_{10} + 7_{10}$ using two's complement numbers.

Solution: (a) $-2_{10} + 1_{10} = 1110_2 + 0001_2 = 1111_2 = -1_{10}$. (b) $-7_{10} + 7_{10} = 1001_2 + 0111_2 = 10000_2$. The fifth bit is discarded, leaving the correct 4-bit result 0000_2.

Subtraction is performed by taking the two's complement of the second number, then adding.

Example 1.13 SUBTRACTING TWO'S COMPLEMENT NUMBERS

Compute (a) $5_{10} - 3_{10}$ and (b) $3_{10} - 5_{10}$ using 4-bit two's complement numbers.

Solution: (a) $3_{10} = 0011_2$. Take its two's complement to obtain $-3_{10} = 1101_2$. Now add $5_{10} + (-3_{10}) = 0101_2 + 1101_2 = 0010_2 = 2_{10}$. Note that the carry out of the most significant position is discarded because the result is stored in four bits. (b) Take the two's complement of 5_{10} to obtain $-5_{10} = 1011$. Now add $3_{10} + (-5_{10}) = 0011_2 + 1011_2 = 1110_2 = -2_{10}$.

The two's complement of 0 is found by inverting all the bits (producing $11...111_2$) and adding 1, which produces all 0's, disregarding the carry out of the most significant bit position. Hence, zero is always represented with all 0's. Unlike the sign/magnitude system, the two's complement system has no separate -0. Zero is considered positive because its sign bit is 0.

Like unsigned numbers, N-bit two's complement numbers represent one of 2^N possible values. However the values are split between positive and negative numbers. For example, a 4-bit unsigned number represents 16 values: 0 to 15. A 4-bit two's complement number also represents 16 values: -8 to 7. In general, the range of an N-bit two's complement number spans $[-2^{N-1}, 2^{N-1} - 1]$. It should make sense that there is one more negative number than positive number because there is no -0. The most negative number $10...000_2 = -2^{N-1}$ is sometimes called the *weird number*. Its two's complement is found by inverting the bits (producing $01...111_2$ and adding 1, which produces $10...000_2$, the weird number, again). Hence, this negative number has no positive counterpart.

Adding two N-bit positive numbers or negative numbers may cause overflow if the result is greater than $2^{N-1} - 1$ or less than -2^{N-1}. Adding a positive number to a negative number never causes overflow. Unlike unsigned numbers, a carry out of the most significant column does not indicate overflow. Instead, overflow occurs if the two numbers being added have the same sign bit and the result has the opposite sign bit.

Example 1.14 ADDING TWO'S COMPLEMENT NUMBERS WITH OVERFLOW

Compute (a) $4_{10} + 5_{10}$ using 4-bit two's complement numbers. Does the result overflow?

Solution: (a) $4_{10} + 5_{10} = 0100_2 + 0101_2 = 1001_2 = -7_{10}$. The result overflows the range of 4-bit positive two's complement numbers, producing an incorrect negative result. If the computation had been done using five or more bits, the result $01001_2 = 9_{10}$ would have been correct.

When a two's complement number is extended to more bits, the sign bit must be copied into the most significant bit positions. This process is called *sign extension*. For example, the numbers 3 and -3 are written as 4-bit two's complement numbers 0011 and 1101, respectively. They are sign-extended to seven bits by copying the sign bit into the three new upper bits to form *000*0011 and *111*1101, respectively.

Comparison of Number Systems

The three most commonly used binary number systems are unsigned, two's complement, and sign/magnitude. Table 1.3 compares the range of N-bit numbers in each of these three systems. Two's complement numbers are convenient because they represent both positive and negative integers and because ordinary addition works for all numbers.

Table 1.3 Range of *N*-bit numbers

System	Range
Unsigned	$[0, 2^N - 1]$
Sign/Magnitude	$[-2^{N-1} + 1, 2^{N-1} - 1]$
Two's Complement	$[-2^{N-1}, 2^{N-1} - 1]$

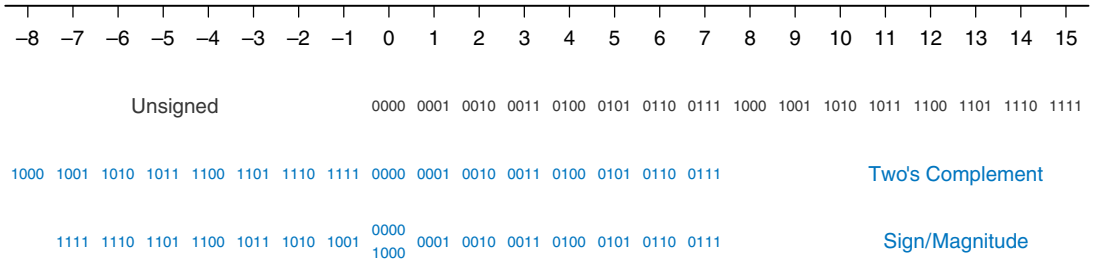


Figure 1.11 Number line and 4-bit binary encodings

Subtraction is performed by negating the second number (i.e., taking the two's complement), and then adding. Unless stated otherwise, assume that all signed binary numbers use two's complement representation.

Figure 1.11 shows a number line indicating the values of 4-bit numbers in each system. Unsigned numbers span the range [0, 15] in regular binary order. Two's complement numbers span the range [−8, 7]. The nonnegative numbers [0, 7] share the same encodings as unsigned numbers. The negative numbers [−8, −1] are encoded such that a larger unsigned binary value represents a number closer to 0. Notice that the weird number, 1000, represents −8 and has no positive counterpart. Sign/magnitude numbers span the range [−7, 7]. The most significant bit is the sign bit. The positive numbers [1, 7] share the same encodings as unsigned numbers. The negative numbers are symmetric but have the sign bit set. 0 is represented by both 0000 and 1000. Thus, *N*-bit sign/magnitude numbers represent only $2^N - 1$ integers because of the two representations for 0.

1.5 LOGIC GATES

Now that we know how to use binary variables to represent information, we explore digital systems that perform operations on these binary variables. *Logic gates* are simple digital circuits that take one or more binary inputs and produce a binary output. Logic gates are drawn with a symbol showing the input (or inputs) and the output. Inputs are

usually drawn on the left (or top) and outputs on the right (or bottom). Digital designers typically use letters near the beginning of the alphabet for gate inputs and the letter Y for the gate output. The relationship between the inputs and the output can be described with a truth table or a Boolean equation. A *truth table* lists inputs on the left and the corresponding output on the right. It has one row for each possible combination of inputs. A *Boolean equation* is a mathematical expression using binary variables.

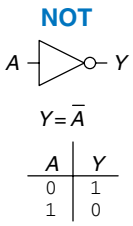


Figure 1.12 NOT gate

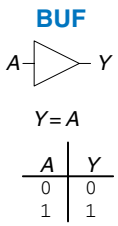


Figure 1.13 Buffer

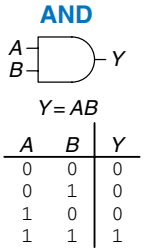


Figure 1.14 AND gate

According to Larry Wall, inventor of the Perl programming language, "the three principal virtues of a programmer are Laziness, Impatience, and Hubris."

1.5.1 NOT Gate

A *NOT gate* has one input, A, and one output, Y, as shown in Figure 1.12. The NOT gate's output is the inverse of its input. If A is FALSE, then Y is TRUE. If A is TRUE, then Y is FALSE. This relationship is summarized by the truth table and Boolean equation in the figure. The line over A in the Boolean equation is pronounced *NOT*, so $Y = \bar{A}$ is read "Y equals NOT A." The NOT gate is also called an *inverter*.

Other texts use a variety of notations for NOT, including $Y = A'$, $Y = \neg A$, $Y = !A$ or $Y = \sim A$. We will use $Y = \bar{A}$ exclusively, but don't be puzzled if you encounter another notation elsewhere.

1.5.2 Buffer

The other one-input logic gate is called a *buffer* and is shown in Figure 1.13. It simply copies the input to the output.

From the logical point of view, a buffer is no different from a wire, so it might seem useless. However, from the analog point of view, the buffer might have desirable characteristics such as the ability to deliver large amounts of current to a motor or the ability to quickly send its output to many gates. This is an example of why we need to consider multiple levels of abstraction to fully understand a system; the digital abstraction hides the real purpose of a buffer.

The triangle symbol indicates a buffer. A circle on the output is called a *bubble* and indicates inversion, as was seen in the NOT gate symbol of Figure 1.12.

1.5.3 AND Gate

Two-input logic gates are more interesting. The *AND gate* shown in Figure 1.14 produces a TRUE output, Y, if and only if both A *and* B are TRUE. Otherwise, the output is FALSE. By convention, the inputs are listed in the order 00, 01, 10, 11, as if you were counting in binary. The Boolean equation for an AND gate can be written in several ways: $Y = A \bullet B$, $Y = AB$, or $Y = A \cap B$. The $\cap$ symbol is pronounced "intersection" and is preferred by logicians. We prefer $Y = AB$, read "Y equals A and B," because we are lazy.

1.5.4 OR Gate

The *OR gate* shown in Figure 1.15 produces a TRUE output, *Y*, if either *A* or *B* (or both) are TRUE. The Boolean equation for an OR gate is written as $Y = A + B$ or $Y = A \cup B$. The $\cup$ symbol is pronounced union and is preferred by logicians. Digital designers normally use the + notation, $Y = A + B$ is pronounced "*Y* equals *A* or *B*".

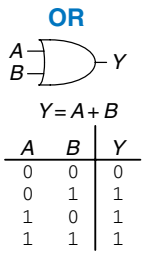


Figure 1.15 OR gate

1.5.5 Other Two-Input Gates

Figure 1.16 shows other common two-input logic gates. *XOR* (exclusive OR, pronounced "ex-OR") is TRUE if *A* or *B*, but not both, are TRUE. Any gate can be followed by a bubble to invert its operation. The *NAND gate* performs NOT AND. Its output is TRUE unless both inputs are TRUE. The *NOR gate* performs NOT OR. Its output is TRUE if neither *A* nor *B* is TRUE.

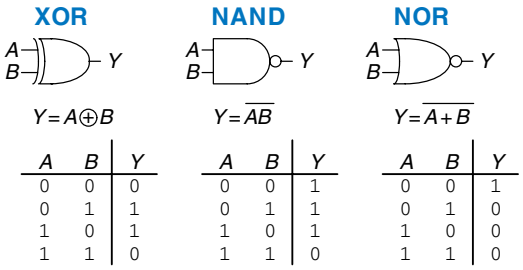
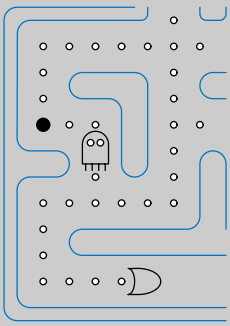


Figure 1.16 More two-input logic gates

A silly way to remember the OR symbol is that it's input side is curved like Pacman's mouth, so the gate is hungry and willing to eat any TRUE inputs it can find!



Example 1.15 XNOR GATE

Figure 1.17 shows the symbol and Boolean equation for a two-input *XNOR gate* that performs the inverse of an XOR. Complete the truth table.

Solution: Figure 1.18 shows the truth table. The XNOR output is TRUE if both inputs are FALSE or both inputs are TRUE. The two-input XNOR gate is sometimes called an *equality* gate because its output is TRUE when the inputs are equal.

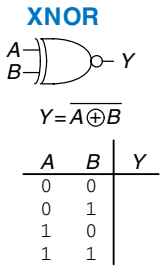


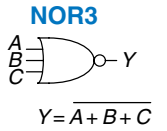
Figure 1.17 XNOR gate

1.5.6 Multiple-Input Gates

Many Boolean functions of three or more inputs exist. The most common are AND, OR, XOR, NAND, NOR, and XNOR. An *N*-input AND gate produces a TRUE output when all *N* inputs are TRUE. An *N*-input OR gate produces a TRUE output when at least one input is TRUE.

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

Figure 1.18 XNOR truth table



A	B	C	Y
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Figure 1.19 Three-input NOR gate

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Figure 1.20 Three-input NOR truth table

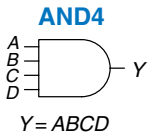


Figure 1.21 Four-input AND gate

An N-input XOR gate is sometimes called a *parity* gate and produces a TRUE output if an odd number of inputs are TRUE. As with two-input gates, the input combinations in the truth table are listed in counting order.

Example 1.16 THREE-INPUT NOR GATE

Figure 1.19 shows the symbol and Boolean equation for a three-input NOR gate. Complete the truth table.

Solution: Figure 1.20 shows the truth table. The output is TRUE only if none of the inputs are TRUE.

Example 1.17 FOUR-INPUT AND GATE

Figure 1.21 shows the symbol and Boolean equation for a four-input AND gate. Create a truth table.

Solution: Figure 1.22 shows the truth table. The output is TRUE only if all of the inputs are TRUE.

1.6 BENEATH THE DIGITAL ABSTRACTION

A digital system uses discrete-valued variables. However, the variables are represented by continuous physical quantities such as the voltage on a wire, the position of a gear, or the level of fluid in a cylinder. Hence, the designer must choose a way to relate the continuous value to the discrete value.

For example, consider representing a binary signal A with a voltage on a wire. Let 0 volts (V) indicate $A = 0$ and 5 V indicate $A = 1$. Any real system must tolerate some noise, so 4.97 V probably ought to be interpreted as $A = 1$ as well. But what about 4.3 V? Or 2.8 V? Or 2.500000 V?

1.6.1 Supply Voltage

Suppose the lowest voltage in the system is 0 V, also called *ground* or GND. The highest voltage in the system comes from the power supply and is usually called V_{DD}. In 1970's and 1980's technology, V_{DD} was generally 5 V. As chips have progressed to smaller transistors, V_{DD} has dropped to 3.3 V, 2.5 V, 1.8 V, 1.5 V, 1.2 V, or even lower to save power and avoid overloading the transistors.

1.6.2 Logic Levels

The mapping of a continuous variable onto a discrete binary variable is done by defining *logic levels*, as shown in Figure 1.23. The first gate is called the *driver* and the second gate is called the *receiver*. The output of

the driver is connected to the input of the receiver. The driver produces a LOW (0) output in the range of 0 to V_{OL} or a HIGH (1) output in the range of V_{OH} to V_{DD}. If the receiver gets an input in the range of 0 to V_{IL}, it will consider the input to be LOW. If the receiver gets an input in the range of V_{IH} to V_{DD}, it will consider the input to be HIGH. If, for some reason such as noise or faulty components, the receiver's input should fall in the *forbidden zone* between V_{IL} and V_{IH}, the behavior of the gate is unpredictable. V_{OH}, V_{OL}, V_{IH}, and V_{IL} are called the output and input high and low logic levels.

A	C	B	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Figure 1.22 Four-input AND truth table

1.6.3 Noise Margins

If the output of the driver is to be correctly interpreted at the input of the receiver, we must choose $V_{OL} < V_{IL}$ and $V_{OH} > V_{IH}$. Thus, even if the output of the driver is contaminated by some noise, the input of the receiver will still detect the correct logic level. The *noise margin* is the amount of noise that could be added to a worst-case output such that the signal can still be interpreted as a valid input. As can be seen in Figure 1.23, the low and high noise margins are, respectively

$$NM_L = V_{IL} - V_{OL} \quad (1.2)$$
$$NM_H = V_{OH} - V_{IH} \quad (1.3)$$

Example 1.18

Consider the inverter circuit of Figure 1.24. V_{O1} is the output voltage of inverter I1, and V_{I2} is the input voltage of inverter I2. Both inverters have the following characteristics: $V_{DD} = 5$ V, $V_{IL} = 1.35$ V, $V_{IH} = 3.15$ V, $V_{OL} = 0.33$ V, and $V_{OH} = 3.84$ V. What are the inverter low and high noise margins? Can the circuit tolerate 1 V of noise between V_{O1} and V_{I2}?

Solution: The inverter noise margins are: $NM_L = V_{IL} - V_{OL} = (1.35\text{ V} - 0.33\text{ V}) = 1.02$ V, $NM_H = V_{OH} - V_{IH} = (3.84\text{ V} - 3.15\text{ V}) = 0.69$ V. The circuit can tolerate 1 V of noise when the output is LOW ($NM_L = 1.02$ V) but not when the output is HIGH ($NM_H = 0.69$ V). For example, suppose the driver, I1, outputs its worst-case HIGH value, $V_{O1} = V_{OH} = 3.84$ V. If noise causes the voltage to droop by 1 V before reaching the input of the receiver, $V_{I2} = (3.84\text{ V} - 1\text{ V}) = 2.84$ V. This is less than the acceptable input HIGH value, $V_{IH} = 3.15$ V, so the receiver may not sense a proper HIGH input.

1.6.4 DC Transfer Characteristics

To understand the limits of the digital abstraction, we must delve into the analog behavior of a gate. The *DC transfer characteristics* of a gate describe the output voltage as a function of the input voltage when the

V_{DD} stands for the voltage on the *drain* of a metal-oxide-semiconductor transistor, used to build most modern chips. The power supply voltage is also sometimes called V_{CC}, standing for the voltage on the *collector* of a bipolar transistor used to build chips in an older technology. Ground is sometimes called V_{SS} because it is the voltage on the *source* of a metal-oxide-semiconductor transistor. See Section 1.7 for more information on transistors.

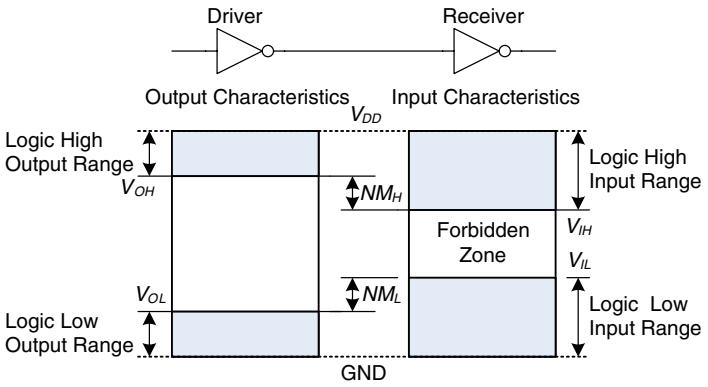


Figure 1.23 Logic levels and noise margins

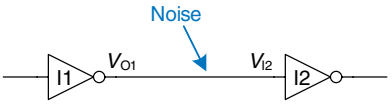


Figure 1.24 Inverter circuit

input is changed slowly enough that the output can keep up. They are called transfer characteristics because they describe the relationship between input and output voltages.

An ideal inverter would have an abrupt switching threshold at $V_{DD}/2$, as shown in Figure 1.25(a). For $V(A) < V_{DD}/2$, $V(Y) = V_{DD}$. For $V(A) > V_{DD}/2$, $V(Y) = 0$. In such a case, $V_{IH} = V_{IL} = V_{DD}/2$. $V_{OH} = V_{DD}$ and $V_{OL} = 0$.

A real inverter changes more gradually between the extremes, as shown in Figure 1.25(b). When the input voltage $V(A)$ is 0, the output voltage $V(Y) = V_{DD}$. When $V(A) = V_{DD}$, $V(Y) = 0$. However, the transition between these endpoints is smooth and may not be centered at exactly $V_{DD}/2$. This raises the question of how to define the logic levels.

A reasonable place to choose the logic levels is where the slope of the transfer characteristic $dV(Y)/dV(A)$ is -1. These two points are called the *unity gain points*. Choosing logic levels at the unity gain points usually maximizes the noise margins. If V_{IL} were reduced, V_{OH} would only increase by a small amount. But if V_{IL} were increased, V_{OH} would drop precipitously.

DC indicates behavior when an input voltage is held constant or changes slowly enough for the rest of the system to keep up. The term's historical root comes from *direct current*, a method of transmitting power across a line with a constant voltage. In contrast, the *transient response* of a circuit is the behavior when an input voltage changes rapidly. Section 2.9 explores transient response further.

1.6.5 The Static Discipline

To avoid inputs falling into the forbidden zone, digital logic gates are designed to conform to the *static discipline*. The static discipline requires that, given logically valid inputs, every circuit element will produce logically valid outputs.

By conforming to the static discipline, digital designers sacrifice the freedom of using arbitrary analog circuit elements in return for the

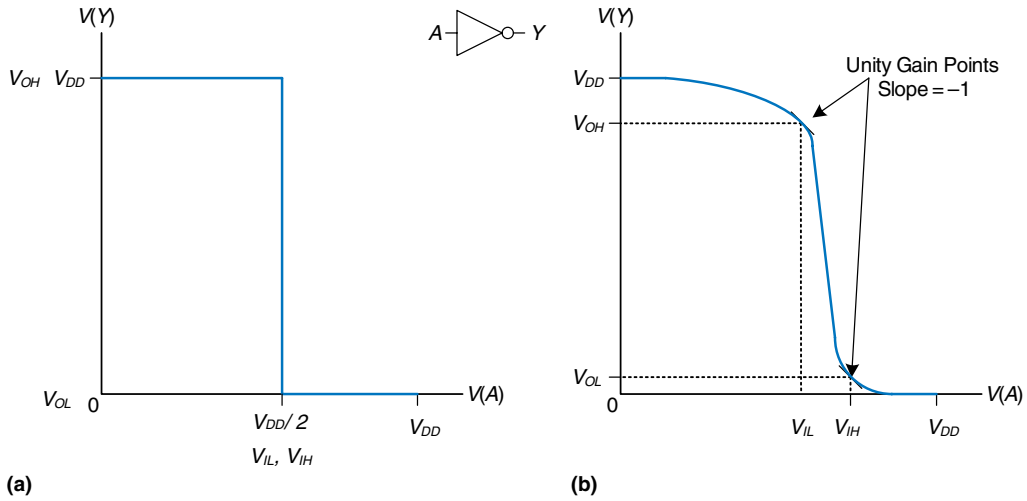


Figure 1.25 DC transfer characteristics and logic levels

simplicity and robustness of digital circuits. They raise the level of abstraction from analog to digital, increasing design productivity by hiding needless detail.

The choice of V_{DD} and logic levels is arbitrary, but all gates that communicate must have compatible logic levels. Therefore, gates are grouped into *logic families* such that all gates in a logic family obey the static discipline when used with other gates in the family. Logic gates in the same logic family snap together like Legos in that they use consistent power supply voltages and logic levels.

Four major logic families that predominated from the 1970's through the 1990's are *Transistor-Transistor Logic* (*TTL*), *Complementary Metal-Oxide-Semiconductor Logic* (*CMOS*, pronounced *sea-moss*), *Low Voltage TTL Logic* (*LVTTL*), and *Low Voltage CMOS Logic* (*LVCMOS*). Their logic levels are compared in Table 1.4. Since then, logic families have balkanized with a proliferation of even lower power supply voltages. Appendix A.6 revisits popular logic families in more detail.

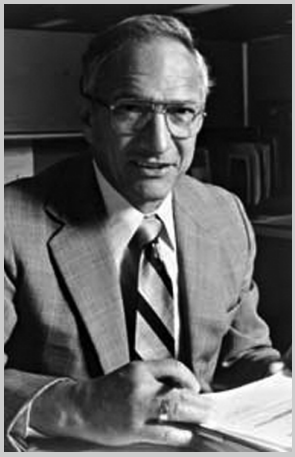
Table 1.4 Logic levels of 5 V and 3.3 V logic families

Logic Family	V_{DD}	V_{IL}	V_{IH}	V_{OL}	V_{OH}
TTL	5 (4.75–5.25)	0.8	2.0	0.4	2.4
CMOS	5 (4.5–6)	1.35	3.15	0.33	3.84
LVTTL	3.3 (3–3.6)	0.8	2.0	0.4	2.4
LVCMOS	3.3 (3–3.6)	0.9	1.8	0.36	2.7

Table 1.5 Compatibility of logic families

		Receiver			
		TTL	CMOS	LVTTL	LVCMOS
Driver	TTL	OK	NO: $V_{OH} < V_{IH}$	MAYBE[a]	MAYBE[a]
	CMOS	OK	OK	MAYBE[a]	MAYBE[a]
	LVTTL	OK	NO: $V_{OH} < V_{IH}$	OK	OK
	LVCMOS	OK	NO: $V_{OH} < V_{IH}$	OK	OK

[a] As long as a 5 V HIGH level does not damage the receiver input



Robert Noyce, 1927–1990. Born in Burlington, Iowa. Received a B.A. in physics from Grinnell College and a Ph.D. in physics from MIT. Nicknamed "Mayor of Silicon Valley" for his profound influence on the industry.

Cofounded Fairchild Semiconductor in 1957 and Intel in 1968. Coinvented the integrated circuit. Many engineers from his teams went on to found other seminal semiconductor companies (© 2006, Intel Corporation. Reproduced by permission).

Example 1.19 LOGIC FAMILY COMPATIBILITY

Which of the logic families in Table 1.4 can communicate with each other reliably?

Solution: Table 1.5 lists which logic families have compatible logic levels. Note that a 5 V logic family such as TTL or CMOS may produce an output voltage as HIGH as 5 V. If this 5 V signal drives the input of a 3.3 V logic family such as LVTTL or LVCMOS, it can damage the receiver, unless the receiver is specially designed to be "5-volt compatible."

1.7 CMOS TRANSISTORS*

This section and other sections marked with a * are optional and are not necessary to understand the main flow of the book.

Babbage's Analytical Engine was built from gears, and early electrical computers used relays or vacuum tubes. Modern computers use transistors because they are cheap, small, and reliable. *Transistors* are electrically controlled switches that turn ON or OFF when a voltage or current is applied to a control terminal. The two main types of transistors are *bipolar transistors* and *metal-oxide-semiconductor field effect transistors* (*MOSFETs* or *MOS transistors,* pronounced "moss-fets" or "M-O-S", respectively).

In 1958, Jack Kilby at Texas Instruments built the first integrated circuit containing two transistors. In 1959, Robert Noyce at Fairchild Semiconductor patented a method of interconnecting multiple transistors on a single silicon chip. At the time, transistors cost about $10 each.

Thanks to more than three decades of unprecedented manufacturing advances, engineers can now pack roughly one billion MOSFETs onto a 1 cm^2 chip of silicon, and these transistors cost less than 10 microcents apiece. The capacity and cost continue to improve by an order of magnitude every 8 years or so. MOSFETs are now the building blocks of

almost all digital systems. In this section, we will peer beneath the digital abstraction to see how logic gates are built from MOSFETs.

1.7.1 Semiconductors

MOS transistors are built from silicon, the predominant atom in rock and sand. Silicon (Si) is a group IV atom, so it has four electrons in its valence shell and forms bonds with four adjacent atoms, resulting in a crystalline *lattice*. Figure 1.26(a) shows the lattice in two dimensions for ease of drawing, but remember that the lattice actually forms a cubic crystal. In the figure, a line represents a covalent bond. By itself, silicon is a poor conductor because all the electrons are tied up in covalent bonds. However, it becomes a better conductor when small amounts of impurities, called *dopant* atoms, are carefully added. If a group V dopant such as arsenic (As) is added, the dopant atoms have an extra electron that is not involved in the bonds. The electron can easily move about the lattice, leaving an ionized dopant atom (As^+) behind, as shown in Figure 1.26(b). The electron carries a negative charge, so we call arsenic an *n-type* dopant. On the other hand, if a group III dopant such as boron (B) is added, the dopant atoms are missing an electron, as shown in Figure 1.26(c). This missing electron is called a *hole*. An electron from a neighboring silicon atom may move over to fill the missing bond, forming an ionized dopant atom (B^-) and leaving a hole at the neighboring silicon atom. In a similar fashion, the hole can migrate around the lattice. The hole is a lack of negative charge, so it acts like a positively charged particle. Hence, we call boron a *p-type* dopant. Because the conductivity of silicon changes over many orders of magnitude depending on the concentration of dopants, silicon is called a *semiconductor*.

1.7.2 Diodes

The junction between p-type and n-type silicon is called a *diode*. The p-type region is called the *anode* and the n-type region is called the *cathode*, as illustrated in Figure 1.27. When the voltage on the anode rises above the voltage on the cathode, the diode is *forward biased,* and

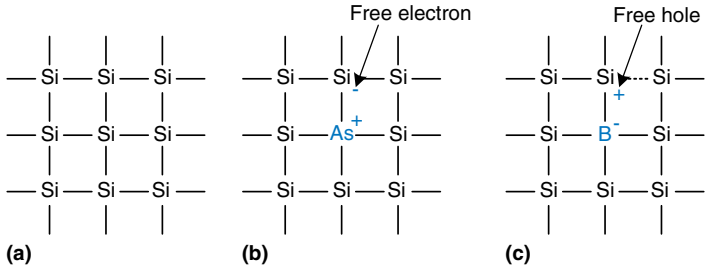


Figure 1.26 Silicon lattice and dopant atoms

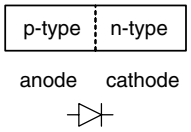
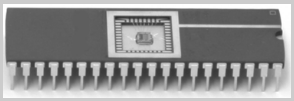


Figure 1.27 The p-n junction diode structure and symbol

Figure 1.28 Capacitor symbol

Technicians in an Intel clean room wear Gore-Tex bunny suits to prevent particulates from their hair, skin, and clothing from contaminating the microscopic transistors on silicon wafers (© 2006, Intel Corporation. Reproduced by permission).



A 40-pin dual-inline package (DIP) contains a small chip (scarcely visible) in the center that is connected to 40 metal pins, 20 on a side, by gold wires thinner than a strand of hair (photograph by Kevin Mapp. © Harvey Mudd College).

current flows through the diode from the anode to the cathode. But when the anode voltage is lower than the voltage on the cathode, the diode is *reverse biased*, and no current flows. The diode symbol intuitively shows that current only flows in one direction.

1.7.3 Capacitors

A *capacitor* consists of two conductors separated by an insulator. When a voltage V is applied to one of the conductors, the conductor accumulates electric *charge* Q and the other conductor accumulates the opposite charge $-Q$. The *capacitance* C of the capacitor is the ratio of charge to voltage: $C = Q/V$. The capacitance is proportional to the size of the conductors and inversely proportional the distance between them. The symbol for a capacitor is shown in Figure 1.28.

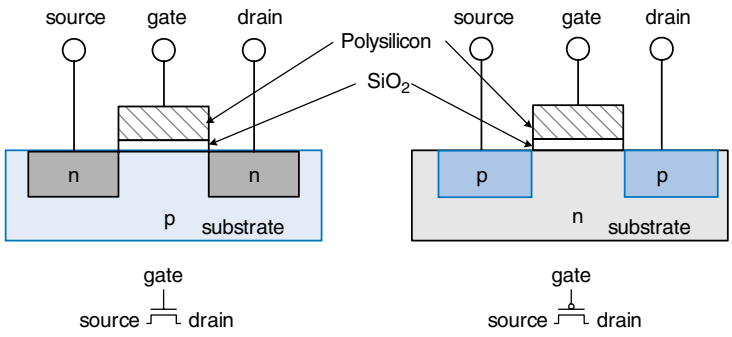
Capacitance is important because charging or discharging a conductor takes time and energy. More capacitance means that a circuit will be slower and require more energy to operate. Speed and energy will be discussed throughout this book.

1.7.4 nMOS and pMOS Transistors

A MOSFET is a sandwich of several layers of conducting and insulating materials. MOSFETs are built on thin flat *wafers* of silicon of about 15 to 30 cm in diameter. The manufacturing process begins with a bare wafer. The process involves a sequence of steps in which dopants are implanted into the silicon, thin films of silicon dioxide and silicon are grown, and metal is deposited. Between each step, the wafer is *patterned* so that the materials appear only where they are desired. Because transistors are a fraction of a micron[2] in length and the entire wafer is processed at once, it is inexpensive to manufacture billions of transistors at a time. Once processing is complete, the wafer is cut into rectangles called *chips* or *dice* that contain thousands, millions, or even billions of transistors. The chip is tested, then placed in a plastic or ceramic *package* with metal pins to connect it to a circuit board.

The MOSFET sandwich consists of a conducting layer called the *gate* on top of an insulating layer of *silicon dioxide* (SiO_2) on top of the silicon wafer, called the *substrate*. Historically, the gate was constructed from metal, hence the name metal-oxide-semiconductor. Modern manufacturing processes use polycrystalline silicon for the gate, because it does not melt during subsequent high-temperature processing steps. Silicon dioxide is better known as glass and is often simply called *oxide* in the semiconductor industry. The metal-oxide-semiconductor sandwich forms a capacitor, in which a thin layer of insulating oxide called a *dielectric* separates the metal and semiconductor plates.

[2] 1 μm = 1 micron = 10^{-6} m.



(a) nMOS **(b)** pMOS

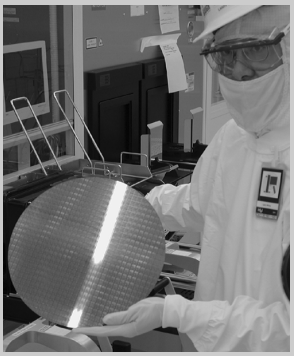
Figure 1.29 nMOS and pMOS transistors

The source and drain terminals are physically symmetric. However, we say that charge flows from the source to the drain. In an nMOS transistor, the charge is carried by electrons, which flow from negative voltage to positive voltage. In a pMOS transistor, the charge is carried by holes, which flow from positive voltage to negative voltage. If we draw schematics with the most positive voltage at the top and the most negative at the bottom, the source of (negative) charges in an nMOS transistor is the bottom terminal and the source of (positive) charges in a pMOS transistor is the top terminal.

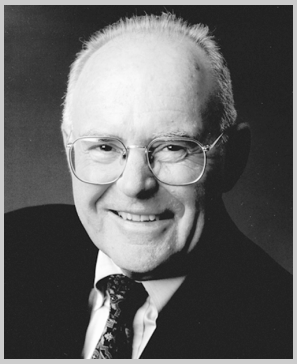
There are two flavors of MOSFETs: nMOS and pMOS (pronounced "n-moss" and "p-moss"). Figure 1.29 shows cross-sections of each type, made by sawing through a wafer and looking at it from the side. The n-type transistors, called *nMOS*, have regions of n-type dopants adjacent to the gate called the *source* and the *drain* and are built on a p-type semiconductor substrate. The *pMOS* transistors are just the opposite, consisting of p-type source and drain regions in an n-type *substrate*.

A MOSFET behaves as a voltage-controlled switch in which the gate voltage creates an electric field that turns ON or OFF a connection between the source and drain. The term *field effect transistor* comes from this principle of operation. Let us start by exploring the operation of an nMOS transistor.

The substrate of an nMOS transistor is normally tied to GND, the lowest voltage in the system. First, consider the situation when the gate is also at 0 V, as shown in Figure 1.30(a). The diodes between the source or drain and the substrate are reverse biased because the source or drain voltage is nonnegative. Hence, there is no path for current to flow between the source and drain, so the transistor is OFF. Now, consider when the gate is raised to V_{DD}, as shown in Figure 1.30(b). When a positive voltage is applied to the top plate of a capacitor, it establishes an electric field that attracts positive charge on the top plate and negative charge to the bottom plate. If the voltage is sufficiently large, so much negative charge is attracted to the underside of the gate that the region *inverts* from p-type to effectively become n-type. This inverted region is called the *channel*. Now the transistor has a continuous path from the n-type source through the n-type channel to the n-type drain, so electrons can flow from source to drain. The transistor is ON. The gate voltage required to turn on a transistor is called the *threshold voltage*, V_t, and is typically 0.3 to 0.7 V.



A technician holds a 12-inch wafer containing hundreds of microprocessor chips (© 2006, Intel Corporation. Reproduced by permission).



Gordon Moore, 1929–. Born in San Francisco. Received a B.S. in chemistry from UC Berkeley and a Ph.D. in chemistry and physics from Caltech. Cofounded Intel in 1968 with Robert Noyce. Observed in 1965 that the number of transistors on a computer chip doubles every year. This trend has become known as *Moore's Law*. Since 1975, transistor counts have doubled every two years.

A corollary of Moore's Law is that microprocessor performance doubles every 18 to 24 months. Semiconductor sales have also increased exponentially. Unfortunately, power consumption has increased exponentially as well (© 2006, Intel Corporation. Reproduced by permission).

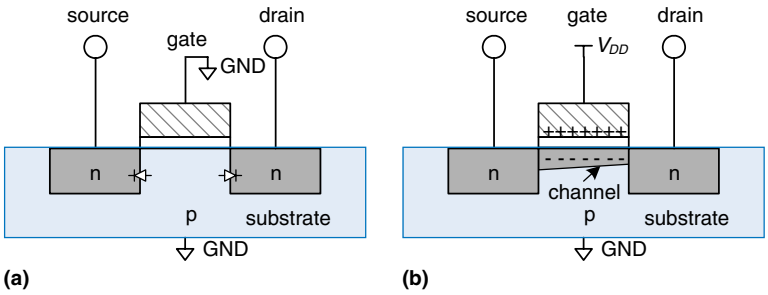


Figure 1.30 nMOS transistor operation

pMOS transistors work in just the opposite fashion, as might be guessed from the bubble on their symbol. The substrate is tied to V_{DD}. When the gate is also at V_{DD}, the pMOS transistor is OFF. When the gate is at GND, the channel inverts to p-type and the pMOS transistor is ON.

Unfortunately, MOSFETs are not perfect switches. In particular, nMOS transistors pass 0's well but pass 1's poorly. Specifically, when the gate of an nMOS transistor is at V_{DD}, the drain will only swing between 0 and $V_{DD} - V_t$. Similarly, pMOS transistors pass 1's well but 0's poorly. However, we will see that it is possible to build logic gates that use transistors only in their good mode.

nMOS transistors need a p-type substrate, and pMOS transistors need an n-type substrate. To build both flavors of transistors on the same chip, manufacturing processes typically start with a p-type wafer, then implant n-type regions called *wells* where the pMOS transistors should go. These processes that provide both flavors of transistors are called Complementary MOS or *CMOS*. CMOS processes are used to build the vast majority of all transistors fabricated today.

In summary, CMOS processes give us two types of electrically controlled switches, as shown in Figure 1.31. The voltage at the gate (g) regulates the flow of current between the source (s) and drain (d). nMOS transistors are OFF when the gate is 0 and ON when the gate is 1.

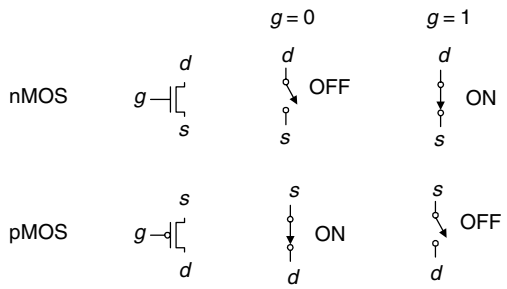


Figure 1.31 Switch models of MOSFETs

pMOS transistors are just the opposite: ON when the gate is 0 and OFF when the gate is 1.

1.7.5 CMOS NOT Gate

Figure 1.32 shows a schematic of a NOT gate built with CMOS transistors. The triangle indicates GND, and the flat bar indicates V_{DD}; these labels will be omitted from future schematics. The nMOS transistor, *N1*, is connected between GND and the *Y* output. The pMOS transistor, *P1*, is connected between V_{DD} and the *Y* output. Both transistor gates are controlled by the input, *A*.

If $A = 0$, *N1* is OFF and *P1* is ON. Hence, *Y* is connected to V_{DD} but not to GND, and is pulled up to a logic 1. *P1* passes a good 1. If $A = 1$, *N1* is ON and *P1* is OFF, and *Y* is pulled down to a logic 0. *N1* passes a good 0. Checking against the truth table in Figure 1.12, we see that the circuit is indeed a NOT gate.

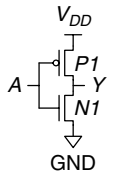


Figure 1.32 NOT gate schematic

1.7.6 Other CMOS Logic Gates

Figure 1.33 shows a schematic of a two-input NAND gate. In schematic diagrams, wires are always joined at three-way junctions. They are joined at four-way junctions only if a dot is shown. The nMOS transistors *N1* and *N2* are connected in series; both nMOS transistors must be ON to pull the output down to GND. The pMOS transistors *P1* and *P2* are in parallel; only one pMOS transistor must be ON to pull the output up to V_{DD}. Table 1.6 lists the operation of the pull-down and pull-up networks and the state of the output, demonstrating that the gate does function as a NAND. For example, when $A = 1$ and $B = 0$, *N1* is ON, but *N2* is OFF, blocking the path from *Y* to GND. *P1* is OFF, but *P2* is ON, creating a path from V_{DD} to *Y*. Therefore, *Y* is pulled up to 1.

Figure 1.34 shows the general form used to construct any inverting logic gate, such as NOT, NAND, or NOR. nMOS transistors are good at passing 0's, so a pull-down network of nMOS transistors is placed between the output and GND to pull the output down to 0. pMOS transistors are

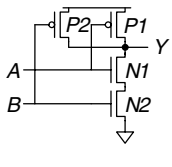


Figure 1.33 Two-input NAND gate schematic

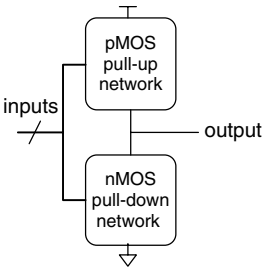


Figure 1.34 General form of an inverting logic gate

Table 1.6 NAND gate operation

A	*B*	Pull-Down Network	Pull-Up Network	*Y*
0	0	OFF	ON	1
0	1	OFF	ON	1
1	0	OFF	ON	1
1	1	ON	OFF	0

Experienced designers claim that electronic devices operate because they contain *magic smoke*. They confirm this theory with the observation that if the magic smoke is ever let out of the device, it ceases to work.

good at passing 1's, so a pull-up network of pMOS transistors is placed between the output and V_{DD} to pull the output up to 1. The networks may consist of transistors in series or in parallel. When transistors are in parallel, the network is ON if either transistor is ON. When transistors are in series, the network is ON only if both transistors are ON. The slash across the input wire indicates that the gate may receive multiple inputs.

If both the pull-up and pull-down networks were ON simultaneously, a *short circuit* would exist between V_{DD} and GND. The output of the gate might be in the forbidden zone and the transistors would consume large amounts of power, possibly enough to burn out. On the other hand, if both the pull-up and pull-down networks were OFF simultaneously, the output would be connected to neither V_{DD} nor GND. We say that the output *floats*. Its value is again undefined. Floating outputs are usually undesirable, but in Section 2.6 we will see how they can occasionally be used to the designer's advantage.

In a properly functioning logic gate, one of the networks should be ON and the other OFF at any given time, so that the output is pulled HIGH or LOW but not shorted or floating. We can guarantee this by using the rule of *conduction complements*. When nMOS transistors are in series, the pMOS transistors must be in parallel. When nMOS transistors are in parallel, the pMOS transistors must be in series.

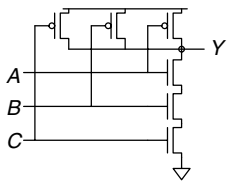


Figure 1.35 Three-input NAND gate schematic

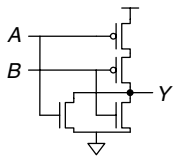


Figure 1.36 Two-input NOR gate schematic

Example 1.20 THREE-INPUT NAND SCHEMATIC

Draw a schematic for a three-input NAND gate using CMOS transistors.

Solution: The NAND gate should produce a 0 output only when all three inputs are 1. Hence, the pull-down network should have three nMOS transistors in series. By the conduction complements rule, the pMOS transistors must be in parallel. Such a gate is shown in Figure 1.35; you can verify the function by checking that it has the correct truth table.

Example 1.21 TWO-INPUT NOR SCHEMATIC

Draw a schematic for a two-input NOR gate using CMOS transistors.

Solution: The NOR gate should produce a 0 output if either input is 1. Hence, the pull-down network should have two nMOS transistors in parallel. By the conduction complements rule, the pMOS transistors must be in series. Such a gate is shown in Figure 1.36.

Example 1.22 TWO-INPUT AND SCHEMATIC

Draw a schematic for a two-input AND gate.

Solution: It is impossible to build an AND gate with a single CMOS gate. However, building NAND and NOT gates is easy. Thus, the best way to build an AND gate using CMOS transistors is to use a NAND followed by a NOT, as shown in Figure 1.37.

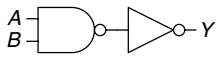


Figure 1.37 Two-input AND gate schematic

1.7.7 Transmission Gates

At times, designers find it convenient to use an ideal switch that can pass both 0 and 1 well. Recall that nMOS transistors are good at passing 0 and pMOS transistors are good at passing 1, so the parallel combination of the two passes both values well. Figure 1.38 shows such a circuit, called a *transmission gate* or *pass gate*. The two sides of the switch are called A and B because a switch is bidirectional and has no preferred input or output side. The control signals are called *enable*s, EN and $\overline{EN}$. When $EN = 0$ and $\overline{EN} = 1$, both transistors are OFF. Hence, the transmission gate is OFF or disabled, so A and B are not connected. When $EN = 1$ and $\overline{EN} = 0$, the transmission gate is ON or enabled, and any logic value can flow between A and B.

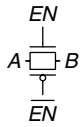


Figure 1.38 Transmission gate

1.7.8 Pseudo-nMOS Logic

An N-input CMOS NOR gate uses N nMOS transistors in parallel and N pMOS transistors in series. Transistors in series are slower than transistors in parallel, just as resistors in series have more resistance than resistors in parallel. Moreover, pMOS transistors are slower than nMOS transistors because holes cannot move around the silicon lattice as fast as electrons. Therefore the parallel nMOS transistors are fast and the series pMOS transistors are slow, especially when many are in series.

Pseudo-nMOS logic replaces the slow stack of pMOS transistors with a single weak pMOS transistor that is always ON, as shown in Figure 1.39. This pMOS transistor is often called a *weak pull-up*. The physical dimensions of the pMOS transistor are selected so that the pMOS transistor will pull the output, Y, HIGH weakly—that is, only if none of the nMOS transistors are ON. But if any nMOS transistor is ON, it overpowers the weak pull-up and pulls Y down close enough to GND to produce a logic 0.

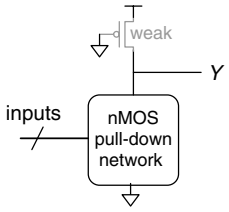


Figure 1.39 Generic pseudo-nMOS gate

The advantage of pseudo-nMOS logic is that it can be used to build fast NOR gates with many inputs. For example, Figure 1.40 shows a pseudo-nMOS four-input NOR. Pseudo-nMOS gates are useful for certain memory and logic arrays discussed in Chapter 5. The disadvantage is that a short circuit exists between V_{DD} and GND when the output is LOW; the weak pMOS and nMOS transistors are both ON. The short circuit draws continuous power, so pseudo-nMOS logic must be used sparingly.

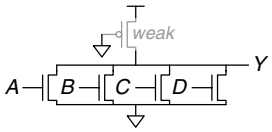


Figure 1.40 Pseudo-nMOS four-input NOR gate

Pseudo-nMOS gates got their name from the 1970's, when manufacturing processes only had nMOS transistors. A weak nMOS transistor was used to pull the output HIGH because pMOS transistors were not available.

1.8 POWER CONSUMPTION*

Power consumption is the amount of energy used per unit time. Power consumption is of great importance in digital systems. The battery life of portable systems such as cell phones and laptop computers is limited by power consumption. Power is also significant for systems that are plugged in, because electricity costs money and because the system will overheat if it draws too much power.

Digital systems draw both *dynamic* and *static* power. Dynamic power is the power used to charge capacitance as signals change between 0 and 1. Static power is the power used even when signals do not change and the system is idle.

Logic gates and the wires that connect them have capacitance. The energy drawn from the power supply to charge a capacitance C to voltage V_{DD} is CV_{DD}^2. If the voltage on the capacitor switches at frequency f (i.e., f times per second), it charges the capacitor $f/2$ times and discharges it $f/2$ times per second. Discharging does not draw energy from the power supply, so the dynamic power consumption is

$$P_{\text{dynamic}} = \frac{1}{2} C V_{DD}^2 f \tag{1.4}$$

Electrical systems draw some current even when they are idle. When transistors are OFF, they leak a small amount of current. Some circuits, such as the pseudo-nMOS gate discussed in Section 1.7.8, have a path from V_{DD} to GND through which current flows continuously. The total static current, I_{DD}, is also called the *leakage current* or the *quiescent supply current* flowing between V_{DD} and GND. The static power consumption is proportional to this static current:

$$P_{\text{static}} = I_{DD} V_{DD} \tag{1.5}$$

Example 1.23 POWER CONSUMPTION

A particular cell phone has a 6 watt-hour (W-hr) battery and operates at 1.2 V. Suppose that, when it is in use, the cell phone operates at 300 MHz and the average amount of capacitance in the chip switching at any given time is 10 nF (10^{-8} Farads). When in use, it also broadcasts 3 W of power out of its antenna. When the phone is not in use, the dynamic power drops to almost zero because the signal processing is turned off. But the phone also draws 40 mA of quiescent current whether it is in use or not. Determine the battery life of the phone (a) if it is not being used, and (b) if it is being used continuously.

Solution: The static power is P_{static} = (0.040 A)(1.2 V) = 48 mW. If the phone is not being used, this is the only power consumption, so the battery life is (6 W-hr)/(0.048 W) = 125 hours (about 5 days). If the phone is being used, the dynamic power is $P_{dynamic} = (0.5)(10^{-8}\text{ F})(1.2\text{ V})^2(3 \times 10^8\text{ Hz}) = 2.16$ W. Together with the static and broadcast power, the total active power is 2.16 W + 0.048 W + 3 W = 5.2 W, so the battery life is 6 W-hr/5.2 W = 1.15 hours. This example somewhat oversimplifies the actual operation of a cell phone, but it illustrates the key ideas of power consumption.

1.9 SUMMARY AND A LOOK AHEAD

There are 10 kinds of people in this world: those who can count in binary and those who can't.

This chapter has introduced principles for understanding and designing complex systems. Although the real world is analog, digital designers discipline themselves to use a discrete subset of possible signals. In particular, binary variables have just two states: 0 and 1, also called FALSE and TRUE or LOW and HIGH. Logic gates compute a binary output from one or more binary inputs. Some of the common logic gates are:

- **NOT:** TRUE when input is FALSE
- **AND:** TRUE when all inputs are TRUE
- **OR:** TRUE when any inputs are TRUE
- **XOR:** TRUE when an odd number of inputs are TRUE

Logic gates are commonly built from CMOS transistors, which behave as electrically controlled switches. nMOS transistors turn ON when the gate is 1. pMOS transistors turn ON when the gate is 0.

In Chapters 2 through 5, we continue the study of digital logic. Chapter 2 addresses *combinational logic*, in which the outputs depend only on the current inputs. The logic gates introduced already are examples of combinational logic. You will learn to design circuits involving multiple gates to implement a relationship between inputs and outputs specified by a truth table or Boolean equation. Chapter 3 addresses *sequential logic,* in which the outputs depend on both current and past inputs. *Registers* are common sequential elements that remember their previous input. *Finite state machines*, built from registers and combinational logic, are a powerful way to build complicated systems in a systematic fashion. We also study timing of digital systems to analyze how fast the systems can operate. Chapter 4 describes hardware description languages (HDLs). HDLs are related to conventional programming languages but are used to simulate and build hardware rather than software. Most digital systems today are designed with HDLs. Verilog

and VHDL are the two prevalent languages, and they are covered side-by-side in this book. Chapter 5 studies other combinational and sequential building blocks such as adders, multipliers, and memories.

Chapter 6 shifts to computer architecture. It describes the MIPS processor, an industry-standard microprocessor used in consumer electronics, some Silicon Graphics workstations, and many communications systems such as televisions, networking hardware and wireless links. The MIPS architecture is defined by its registers and assembly language instruction set. You will learn to write programs in assembly language for the MIPS processor so that you can communicate with the processor in its native language.

Chapters 7 and 8 bridge the gap between digital logic and computer architecture. Chapter 7 investigates microarchitecture, the arrangement of digital building blocks, such as adders and registers, needed to construct a processor. In that chapter, you learn to build your own MIPS processor. Indeed, you learn three microarchitectures illustrating different trade-offs of performance and cost. Processor performance has increased exponentially, requiring ever more sophisticated memory systems to feed the insatiable demand for data. Chapter 8 delves into memory system architecture and also describes how computers communicate with peripheral devices such as keyboards and printers.

Exercises

Exercise 1.1 Explain in one paragraph at least three levels of abstraction that are used by

a) biologists studying the operation of cells.

b) chemists studying the composition of matter.

Exercise 1.2 Explain in one paragraph how the techniques of hierarchy, modularity, and regularity may be used by

a) automobile designers.

b) businesses to manage their operations.

Exercise 1.3 Ben Bitdiddle is building a house. Explain how he can use the principles of hierarchy, modularity, and regularity to save time and money during construction.

Exercise 1.4 An analog voltage is in the range of 0–5 V. If it can be measured with an accuracy of $\pm$ 50 mV, at most how many bits of information does it convey?

Exercise 1.5 A classroom has an old clock on the wall whose minute hand broke off.

a) If you can read the hour hand to the nearest 15 minutes, how many bits of information does the clock convey about the time?

b) If you know whether it is before or after noon, how many additional bits of information do you know about the time?

Exercise 1.6 The Babylonians developed the *sexagesimal* (base 60) number system about 4000 years ago. How many bits of information is conveyed with one sexagesimal digit? How do you write the number 4000_{10} in sexagesimal?

Exercise 1.7 How many different numbers can be represented with 16 bits?

Exercise 1.8 What is the largest unsigned 32-bit binary number?

Exercise 1.9 What is the largest 16-bit binary number that can be represented with

a) unsigned numbers?

b) two's complement numbers?

c) sign/magnitude numbers?

Exercise 1.10 What is the smallest (most negative) 16-bit binary number that can be represented with

a) unsigned numbers?

b) two's complement numbers?

c) sign/magnitude numbers?

Exercise 1.11 Convert the following unsigned binary numbers to decimal.

a) 1010_2

b) 110110_2

c) 11110000_2

d) 0001100010100111_2

Exercise 1.12 Repeat Exercise 1.11, but convert to hexadecimal.

Exercise 1.13 Convert the following hexadecimal numbers to decimal.

a) $A5_{16}$

b) $3B_{16}$

c) $FFFF_{16}$

d) $D0000000_{16}$

Exercise 1.14 Repeat Exercise 1.13, but convert to unsigned binary.

Exercise 1.15 Convert the following two's complement binary numbers to decimal.

a) 1010_2

b) 110110_2

c) 01110000_2

d) 10011111_2

Exercise 1.16 Repeat Exercise 1.15, assuming the binary numbers are in sign/magnitude form rather than two's complement representation.

Exercise 1.17 Convert the following decimal numbers to unsigned binary numbers.

a) 42_{10}

b) 63_{10}

c) 229_{10}

d) 845_{10}

Exercise 1.18 Repeat Exercise 1.17, but convert to hexadecimal.

Exercise 1.19 Convert the following decimal numbers to 8-bit two's complement numbers or indicate that the decimal number would overflow the range.

a) 42_{10}

b) -63_{10}

c) 124_{10}

d) -128_{10}

e) 133_{10}

Exercise 1.20 Repeat Exercise 1.19, but convert to 8-bit sign/magnitude numbers.

Exercise 1.21 Convert the following 4-bit two's complement numbers to 8-bit two's complement numbers.

a) 0101_2

b) 1010_2

Exercise 1.22 Repeat Exercise 1.21 if the numbers are unsigned rather than two's complement.

Exercise 1.23 Base 8 is referred to as *octal*. Convert each of the numbers from Exercise 1.17 to octal.

Exercise 1.24 Convert each of the following octal numbers to binary, hexadecimal, and decimal.

a) 42_8

b) 63_8

c) 255_8

d) 3047_8

Exercise 1.25 How many 5-bit two's complement numbers are greater than 0? How many are less than 0? How would your answers differ for sign/magnitude numbers?

Exercise 1.26 How many bytes are in a 32-bit word? How many nibbles are in the word?

Exercise 1.27 How many bytes are in a 64-bit word?

Exercise 1.28 A particular DSL modem operates at 768 kbits/sec. How many bytes can it receive in 1 minute?

Exercise 1.29 Hard disk manufacturers use the term "megabyte" to mean 10^6 bytes and "gigabyte" to mean 10^9 bytes. How many real GBs of music can you store on a 50 GB hard disk?

Exercise 1.30 Estimate the value of 2^{31} without using a calculator.

Exercise 1.31 A memory on the Pentium II microprocessor is organized as a rectangular array of bits with 2^8 rows and 2^9 columns. Estimate how many bits it has without using a calculator.

Exercise 1.32 Draw a number line analogous to Figure 1.11 for 3-bit unsigned, two's complement, and sign/magnitude numbers.

Exercise 1.33 Perform the following additions of unsigned binary numbers. Indicate whether or not the sum overflows a 4-bit result.

a) $1001_2 + 0100_2$

b) $1101_2 + 1011_2$

Exercise 1.34 Perform the following additions of unsigned binary numbers. Indicate whether or not the sum overflows an 8-bit result.

a) $10011001_2 + 01000100_2$

b) $11010010_2 + 10110110_2$

Exercise 1.35 Repeat Exercise 1.34, assuming that the binary numbers are in two's complement form.

Exercise 1.36 Convert the following decimal numbers to 6-bit two's complement binary numbers and add them. Indicate whether or not the sum overflows a 6-bit result.

a) $16_{10} + 9_{10}$

b) $27_{10} + 31_{10}$

c) $-4_{10} + 19_{10}$

d) $3_{10} + -32_{10}$

e) $-16_{10} + -9_{10}$

f) $-27_{10} + -31_{10}$

Exercise 1.37 Perform the following additions of unsigned hexadecimal numbers. Indicate whether or not the sum overflows an 8-bit (two hex digit) result.

a) $7_{16} + 9_{16}$

b) $13_{16} + 28_{16}$

c) $AB_{16} + 3E_{16}$

d) $8F_{16} + AD_{16}$

Exercise 1.38 Convert the following decimal numbers to 5-bit two's complement binary numbers and subtract them. Indicate whether or not the difference overflows a 5-bit result.

a) $9_{10} - 7_{10}$

b) $12_{10} - 15_{10}$

c) $-6_{10} - 11_{10}$

d) $4_{10} - -8_{10}$

Exercise 1.39 In a *biased* N-bit binary number system with bias B, positive and negative numbers are represented as their value plus the bias B. For example, for 5-bit numbers with a bias of 15, the number 0 is represented as 01111, 1 as 10000, and so forth. Biased number systems are sometimes used in floating point mathematics, which will be discussed in Chapter 5. Consider a biased 8-bit binary number system with a bias of 127_{10}.

a) What decimal value does the binary number 10000010_2 represent?

b) What binary number represents the value 0?

c) What is the representation and value of the most negative number?

d) What is the representation and value of the most positive number?

Exercise 1.40 Draw a number line analogous to Figure 1.11 for 3-bit biased numbers with a bias of 3 (see Exercise 1.39 for a definition of biased numbers).

Exercise 1.41 In a *binary coded decimal* (BCD) system, 4 bits are used to represent a decimal digit from 0 to 9. For example, 37_{10} is written as 00110111_{BCD}.

a) Write 289_{10} in BCD.

b) Convert 100101010001_{BCD} to decimal.

c) Convert 01101001_{BCD} to binary.

d) Explain why BCD might be a useful way to represent numbers.

Exercise 1.42 A flying saucer crashes in a Nebraska cornfield. The FBI investigates the wreckage and finds an engineering manual containing an equation in the Martian number system: 325 + 42 = 411. If this equation is correct, how many fingers would you expect Martians have?

Exercise 1.43 Ben Bitdiddle and Alyssa P. Hacker are having an argument. Ben says, "All integers greater than zero and exactly divisible by six have exactly two 1's in their binary representation." Alyssa disagrees. She says, "No, but all such numbers have an even number of 1's in their representation." Do you agree with Ben or Alyssa or both or neither? Explain.

Exercise 1.44 Ben Bitdiddle and Alyssa P. Hacker are having another argument. Ben says, "I can get the two's complement of a number by subtracting 1, then inverting all the bits of the result." Alyssa says, "No, I can do it by examining each bit of the number, starting with the least significant bit. When the first 1 is found, invert each subsequent bit." Do you agree with Ben or Alyssa or both or neither? Explain.

Exercise 1.45 Write a program in your favorite language (e.g., C, Java, Perl) to convert numbers from binary to decimal. The user should type in an unsigned binary number. The program should print the decimal equivalent.

Exercise 1.46 Repeat Exercise 1.45 but convert from decimal to hexadecimal.

Exercise 1.47 Repeat Exercise 1.45 but convert from an arbitrary base b_1 to another base b_2, as specified by the user. Support bases up to 16, using the letters of the alphabet for digits greater than 9. The user should enter b_1, b_2, and then the number to convert in base b_1. The program should print the equivalent number in base b_2.

Exercise 1.48 Draw the symbol, Boolean equation, and truth table for

a) a three-input OR gate.

b) a three-input exclusive OR (XOR) gate.

c) a four-input XNOR gate

Exercise 1.49 A *majority gate* produces a TRUE output if and only if more than half of its inputs are TRUE. Complete a truth table for the three-input majority gate shown in Figure 1.41.

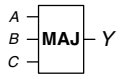


Figure 1.41 Three-input majority gate

Exercise 1.50 A three-input *AND-OR* (*AO*) *gate* shown in Figure 1.42 produces a TRUE output if both *A* and *B* are TRUE, or if *C* is TRUE. Complete a truth table for the gate.

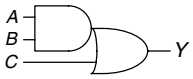


Figure 1.42 Three-input AND-OR gate

Exercise 1.51 A three-input *OR-AND-INVERT* (*OAI*) *gate* shown in Figure 1.43 produces a FALSE input if *C* is TRUE and *A* or *B* is TRUE. Otherwise it produces a TRUE output. Complete a truth table for the gate.

Figure 1.43 Three-input OR-AND-INVERT gate

Exercise 1.52 There are 16 different truth tables for Boolean functions of two variables. List each truth table. Give each one a short descriptive name (such as OR, NAND, and so on).

Exercise 1.53 How many different truth tables exist for Boolean functions of *N* variables?

Exercise 1.54 Is it possible to assign logic levels so that a device with the transfer characteristics shown in Figure 1.44 would serve as an inverter? If so, what are the input and output low and high levels (V_{IL}, V_{OL}, V_{IH}, and V_{OH}) and noise margins (NM_L and NM_H)? If not, explain why not.

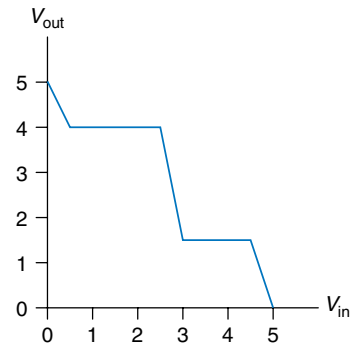


Figure 1.44 DC transfer characteristics

Exercise 1.55 Repeat Exercise 1.54 for the transfer characteristics shown in Figure 1.45.

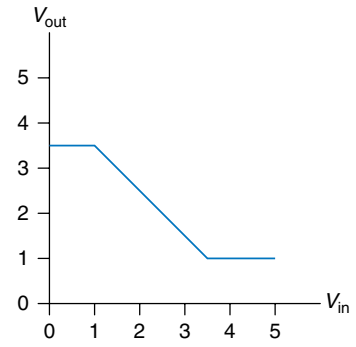


Figure 1.45 DC transfer characteristics

Exercise 1.56 Is it possible to assign logic levels so that a device with the transfer characteristics shown in Figure 1.46 would serve as a buffer? If so, what are the input and output low and high levels (V_{IL}, V_{OL}, V_{IH}, and V_{OH}) and noise margins (NM_L and NM_H)? If not, explain why not.

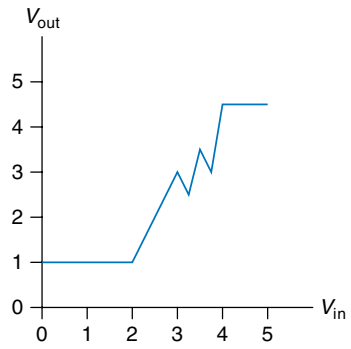


Figure 1.46 DC transfer characteristics

Exercise 1.57 Ben Bitdiddle has invented a circuit with the transfer characteristics shown in Figure 1.47 that he would like to use as a buffer. Will it work? Why or why not? He would like to advertise that it is compatible with LVCMOS and LVTTL logic. Can Ben's buffer correctly receive inputs from those logic families? Can its output properly drive those logic families? Explain.

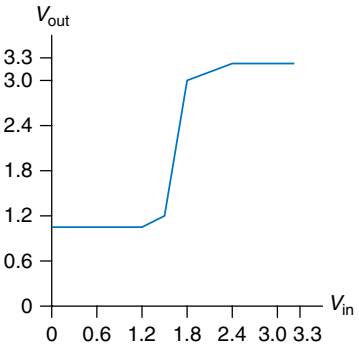


Figure 1.47 Ben's buffer DC transfer characteristics

Exercise 1.58 While walking down a dark alley, Ben Bitdiddle encounters a two-input gate with the transfer function shown in Figure 1.48. The inputs are *A* and *B* and the output is *Y*.

a) What kind of logic gate did he find?

b) What are the approximate high and low logic levels?

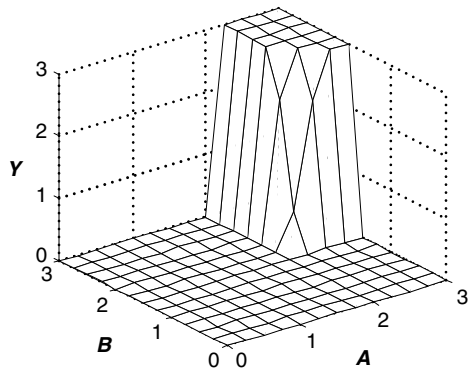


Figure 1.48 **Two-input DC transfer characteristics**

Exercise 1.59 Repeat Exercise 1.58 for Figure 1.49.

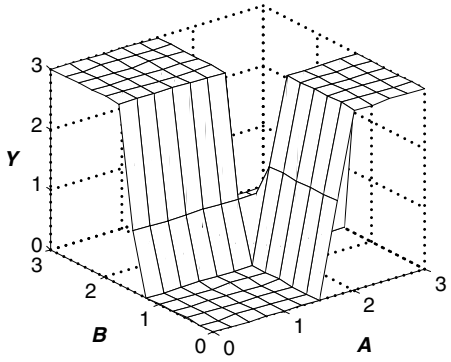


Figure 1.49 **Two-input DC transfer characteristics**

Exercise 1.60 Sketch a transistor-level circuit for the following CMOS gates. Use a minimum number of transistors.

a) A four-input NAND gate.

b) A three-input OR-AND-INVERT gate (see Exercise 1.51).

c) A three-input AND-OR gate (see Exercise 1.50).

Exercise 1.61 A *minority gate* produces a TRUE output if and only if fewer than half of its inputs are TRUE. Otherwise it produces a FALSE output. Sketch a transistor-level circuit for a CMOS minority gate. Use a minimum number of transistors.

Exercise 1.62 Write a truth table for the function performed by the gate in Figure 1.50. The truth table should have two inputs, *A* and *B*. What is the name of this function?

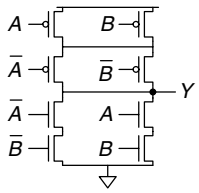


Figure 1.50 Mystery schematic

Exercise 1.63 Write a truth table for the function performed by the gate in Figure 1.51. The truth table should have three inputs, *A*, *B*, and *C*.

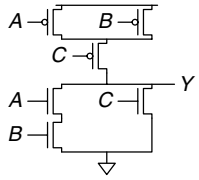


Figure 1.51 Mystery schematic

Exercise 1.64 Implement the following three-input gates using **only** pseudo-nMOS logic gates. Your gates receive three inputs, *A*, *B*, and *C*. Use a minimum number of transistors.

a) three-input NOR gate

b) three-input NAND gate

a) three-input AND gate

Exercise 1.65 *Resistor-Transistor Logic* (*RTL*) uses nMOS transistors to pull the gate output LOW and a weak resistor to pull the output HIGH when none of the paths to ground are active. A NOT gate built using RTL is shown in Figure 1.52. Sketch a three-input RTL NOR gate. Use a minimum number of transistors.

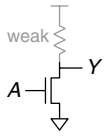


Figure 1.52 RTL NOT gate

Interview Questions

These questions have been asked at interviews for digital design jobs.

Question 1.1 Sketch a transistor-level circuit for a CMOS four-input NOR gate.

Question 1.2 The king receives 64 gold coins in taxes but has reason to believe that one is counterfeit. He summons you to identify the fake coin. You have a balance that can hold coins on each side. How many times do you need to use the balance to find the lighter, fake coin?

Question 1.3 The professor, the teaching assistant, the digital design student, and the freshman track star need to cross a rickety bridge on a dark night. The bridge is so shakey that only two people can cross at a time. They have only one flashlight among them and the span is too long to throw the flashlight, so somebody must carry it back to the other people. The freshman track star can cross the bridge in 1 minute. The digital design student can cross the bridge in 2 minutes. The teaching assistant can cross the bridge in 5 minutes. The professor always gets distracted and takes 10 minutes to cross the bridge. What is the fastest time to get everyone across the bridge?

Combinational Logic Design

2

2.1 INTRODUCTION

In digital electronics, a *circuit* is a network that processes discrete-valued variables. A circuit can be viewed as a black box, shown in Figure 2.1, with

- one or more discrete-valued *input terminals*
- one or more discrete-valued *output terminals*
- a *functional specification* describing the relationship between inputs and outputs
- a *timing specification* describing the delay between inputs changing and outputs responding.

Peering inside the black box, circuits are composed of nodes and elements. An *element* is itself a circuit with inputs, outputs, and a specification. A *node* is a wire, whose voltage conveys a discrete-valued variable. Nodes are classified as *input*, *output*, or *internal*. Inputs receive values from the external world. Outputs deliver values to the external world. Wires that are not inputs or outputs are called internal nodes. Figure 2.2

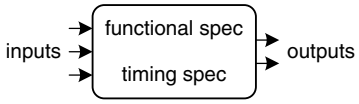


Figure 2.1 Circuit as a black box with inputs, outputs, and specifications

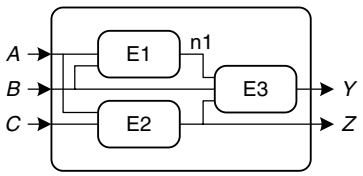
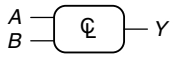


Figure 2.2 Elements and nodes

illustrates a circuit with three elements, E1, E2, and E3, and six nodes. Nodes *A*, *B*, and *C* are inputs. *Y* and *Z* are outputs. n1 is an internal node between E1 and E3.

Digital circuits are classified as *combinational* or *sequential*. A combinational circuit's outputs depend only on the current values of the inputs; in other words, it combines the current input values to compute the output. For example, a logic gate is a combinational circuit. A sequential circuit's outputs depend on both current and previous values of the inputs; in other words, it depends on the input sequence. A combinational circuit is *memoryless*, but a sequential circuit has *memory*. This chapter focuses on combinational circuits, and Chapter 3 examines sequential circuits.

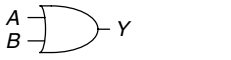
The functional specification of a combinational circuit expresses the output values in terms of the current input values. The timing specification of a combinational circuit consists of lower and upper bounds on the delay from input to output. We will initially concentrate on the functional specification, then return to the timing specification later in this chapter.



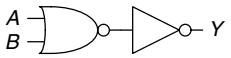
$Y = F(A, B) = A + B$

Figure 2.3 Combinational logic circuit

Figure 2.3 shows a combinational circuit with two inputs and one output. On the left of the figure are the inputs, *A* and *B*, and on the right is the output, *Y*. The symbol inside the box indicates that it is implemented using only combinational logic. In this example, the function, *F*, is specified to be OR: $Y = F(A, B) = A + B$. In words, we say the output, *Y*, is a function of the two inputs, *A* and *B*, namely *Y* = *A* OR *B*.



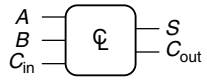
(a)



(b)

Figure 2.4 Two OR implementations

Figure 2.4 shows two possible *implementations* for the combinational logic circuit in Figure 2.3. As we will see repeatedly throughout the book, there are often many implementations for a single function. You choose which to use given the building blocks at your disposal and your design constraints. These constraints often include area, speed, power, and design time.



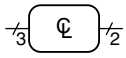
$S = A \oplus B \oplus C_{in}$

$C_{out} = AB + AC_{in} + BC_{in}$

Figure 2.5 Multiple-output combinational circuit

Figure 2.5 shows a combinational circuit with multiple outputs. This particular combinational circuit is called a *full adder* and we will revisit it in Section 5.2.1. The two equations specify the function of the outputs, *S* and C_{out}, in terms of the inputs, *A*, *B*, and C_{in}.

To simplify drawings, we often use a single line with a slash through it and a number next to it to indicate a *bus*, a bundle of multiple signals. The number specifies how many signals are in the bus. For example, Figure 2.6(a) represents a block of combinational logic with three inputs and two outputs. If the number of bits is unimportant or obvious from the context, the slash may be shown without a number. Figure 2.6(b) indicates two blocks of combinational logic with an arbitrary number of outputs from one block serving as inputs to the second block.



(a)

(b)

Figure 2.6 Slash notation for multiple signals

The rules of *combinational composition* tell us how we can build a large combinational circuit from smaller combinational circuit

elements. A circuit is combinational if it consists of interconnected circuit elements such that

- Every circuit element is itself combinational.
- Every node of the circuit is either designated as an input to the circuit or connects to exactly one output terminal of a circuit element.
- The circuit contains no cyclic paths: every path through the circuit visits each circuit node at most once.

The rules of combinational composition are sufficient but not strictly necessary. Certain circuits that disobey these rules are still combinational, so long as the outputs depend only on the current values of the inputs. However, determining whether oddball circuits are combinational is more difficult, so we will usually restrict ourselves to combinational composition as a way to build combinational circuits.

Example 2.1 COMBINATIONAL CIRCUITS

Which of the circuits in Figure 2.7 are combinational circuits according to the rules of combinational composition?

Solution: Circuit (a) is combinational. It is constructed from two combinational circuit elements (inverters I1 and I2). It has three nodes: n1, n2, and n3. n1 is an input to the circuit and to I1; n2 is an internal node, which is the output of I1 and the input to I2; n3 is the output of the circuit and of I2. (b) is not combinational, because there is a cyclic path: the output of the XOR feeds back to one of its inputs. Hence, a cyclic path starting at n4 passes through the XOR to n5, which returns to n4. (c) is combinational. (d) is not combinational, because node n6 connects to the output terminals of both I3 and I4. (e) is combinational, illustrating two combinational circuits connected to form a larger combinational circuit. (f) does not obey the rules of combinational composition because it has a cyclic path through the two elements. Depending on the functions of the elements, it may or may not be a combinational circuit.

Large circuits such as microprocessors can be very complicated, so we use the principles from Chapter 1 to manage the complexity. Viewing a circuit as a black box with a well-defined interface and function is an

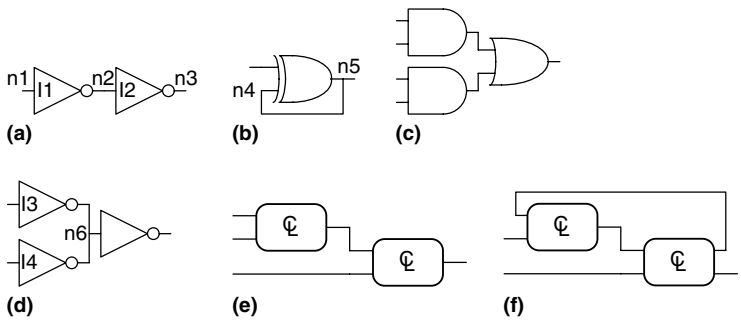


Figure 2.7 Example circuits

application of abstraction and modularity. Building the circuit out of smaller circuit elements is an application of hierarchy. The rules of combinational composition are an application of discipline.

The functional specification of a combinational circuit is usually expressed as a truth table or a Boolean equation. In the next sections, we describe how to derive a Boolean equation from any truth table and how to use Boolean algebra and Karnaugh maps to simplify equations. We show how to implement these equations using logic gates and how to analyze the speed of these circuits.

2.2 BOOLEAN EQUATIONS

Boolean equations deal with variables that are either TRUE or FALSE, so they are perfect for describing digital logic. This section defines some terminology commonly used in Boolean equations, then shows how to write a Boolean equation for any logic function given its truth table.

2.2.1 Terminology

The *complement* of a variable, A, is its inverse, $\overline{A}$. The variable or its complement is called a *literal*. For example, A, $\overline{A}$, B, and $\overline{B}$ are literals. We call A the *true form* of the variable and $\overline{A}$ the *complementary form;* "true form" does not mean that A is TRUE, but merely that A does not have a line over it.

The AND of one or more literals is called a *product* or an *implicant*. $\overline{A}B$, $A\overline{B}\overline{C}$, and B are all implicants for a function of three variables. A *minterm* is a product involving all of the inputs to the function. $A\overline{B}\overline{C}$ is a minterm for a function of the three variables A, B, and C, but $\overline{A}B$ is not, because it does not involve C. Similarly, the OR of one or more literals is called a *sum*. A *maxterm* is a sum involving all of the inputs to the function. $A + \overline{B} + C$ is a maxterm for a function of the three variables A, B, and C.

The *order of operations* is important when interpreting Boolean equations. Does $Y = A + BC$ mean $Y = (A$ OR $B)$ AND C or $Y = A$ OR $(B$ AND $C)$? In Boolean equations, NOT has the highest *precedence,* followed by AND, then OR. Just as in ordinary equations, products are performed before sums. Therefore, the equation is read as $Y = A$ OR $(B$ AND $C)$. Equation 2.1 gives another example of order of operations.

$$\overline{A}B + BC\overline{D} = ((\overline{A})B) + (BC(\overline{D})) \tag{2.1}$$

2.2.2 Sum-of-Products Form

A truth table of N inputs contains 2^N rows, one for each possible value of the inputs. Each row in a truth table is associated with a minterm

that is TRUE for that row. Figure 2.8 shows a truth table of two inputs, A and B. Each row shows its corresponding minterm. For example, the minterm for the first row is $\overline{A}\,\overline{B}$ because $\overline{A}\,\overline{B}$ is TRUE when $A = 0$, $B = 0$.

We can write a Boolean equation for any truth table by summing each of the minterms for which the output, Y, is TRUE. For example, in Figure 2.8, there is only one row (or minterm) for which the output Y is TRUE, shown circled in blue. Thus, $Y = \overline{A}B$. Figure 2.9 shows a truth table with more than one row in which the output is TRUE. Taking the sum of each of the circled minterms gives $Y = \overline{A}B + AB$.

This is called the *sum-of-products canonical form* of a function because it is the sum (OR) of products (ANDs forming minterms). Although there are many ways to write the same function, such as $Y = B\overline{A} + BA$, we will sort the minterms in the same order that they appear in the truth table, so that we always write the same Boolean expression for the same truth table.

A	B	Y	minterm
0	0	0	$\overline{A}\,\overline{B}$
0	1	1	$\overline{A}\,B$
1	0	0	$A\,\overline{B}$
1	1	0	$A\,B$

Figure 2.8 Truth table and minterms

Canonical form is just a fancy word for standard form. You can use the term to impress your friends and scare your enemies.

A	B	Y	minterm
0	0	0	$\overline{A}\,\overline{B}$
0	1	1	$\overline{A}\,B$
1	0	0	$A\,\overline{B}$
1	1	1	$A\,B$

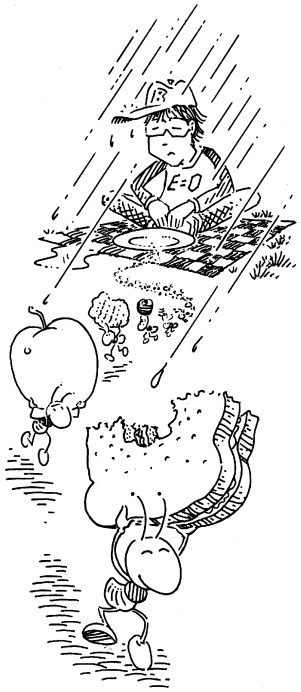
Figure 2.9 Truth table with multiple TRUE minterms

Example 2.2 SUM-OF-PRODUCTS FORM

Ben Bitdiddle is having a picnic. He won't enjoy it if it rains or if there are ants. Design a circuit that will output TRUE *only* if Ben enjoys the picnic.

Solution: First define the inputs and outputs. The inputs are A and R, which indicate if there are ants and if it rains. A is TRUE when there are ants and FALSE when there are no ants. Likewise, R is TRUE when it rains and FALSE when the sun smiles on Ben. The output is E, Ben's enjoyment of the picnic. E is TRUE if Ben enjoys the picnic and FALSE if he suffers. Figure 2.10 shows the truth table for Ben's picnic experience.

Using sum-of-products form, we write the equation as: $E = \overline{A}\,\overline{R}$. We can build the equation using two inverters and a two-input AND gate, shown in Figure 2.11(a). You may recognize this truth table as the NOR function from Section 1.5.5: $E = A$ NOR $R = \overline{A + R}$. Figure 2.11(b) shows the NOR implementation. In Section 2.3, we show that the two equations, $\overline{A}\,\overline{R}$ and $\overline{A + R}$, are equivalent.



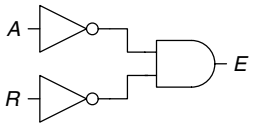
The sum-of-products form provides a Boolean equation for any truth table with any number of variables. Figure 2.12 shows a random three-input truth table. The sum-of-products form of the logic function is

$$Y = \overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C \qquad (2.2)$$

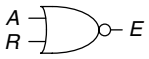
Unfortunately, sum-of-products form does not necessarily generate the simplest equation. In Section 2.3 we show how to write the same function using fewer terms.

A	R	E
0	0	1
0	1	0
1	0	0
1	1	0

Figure 2.10 Ben's truth table



(a)



(b)

Figure 2.11 Ben's circuit

A	B	C	Y
0	0	0	1
0	0	1	0
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 2.12 Random three-input truth table

A	B	Y	maxterm
0	0	0	A + B
0	1	1	A + $\overline{B}$
1	0	0	$\overline{A}$ + B
1	1	1	$\overline{A}$ + $\overline{B}$

Figure 2.13 Truth table with multiple FALSE maxterms

2.2.3 Product-of-Sums Form

An alternative way of expressing Boolean functions is the *product-of-sums canonical form*. Each row of a truth table corresponds to a maxterm that is FALSE for that row. For example, the maxterm for the first row of a two-input truth table is $(A + B)$ because $(A + B)$ is FALSE when $A = 0$, $B = 0$. We can write a Boolean equation for any circuit directly from the truth table as the AND of each of the maxterms for which the output is FALSE.

Example 2.3 PRODUCT-OF-SUMS FORM

Write an equation in product-of-sums form for the truth table in Figure 2.13.

Solution: The truth table has two rows in which the output is FALSE. Hence, the function can be written in product-of-sums form as $Y = (A + B)(\overline{A} + B)$. The first maxterm, $(A + B)$, guarantees that $Y = 0$ for $A = 0$, $B = 0$, because any value AND 0 is 0. Likewise, the second maxterm, $(\overline{A} + B)$, guarantees that $Y = 0$ for $A = 1$, $B = 0$. Figure 2.13 is the same truth table as Figure 2.9, showing that the same function can be written in more than one way.

Similarly, a Boolean equation for Ben's picnic from Figure 2.10 can be written in product-of-sums form by circling the three rows of 0's to obtain $E = (A + \overline{R})(\overline{A} + R)(\overline{A} + \overline{R})$. This is uglier than the sum-of-products equation, $E = \overline{A}\,\overline{R}$, but the two equations are logically equivalent.

Sum-of-products produces the shortest equations when the output is TRUE on only a few rows of a truth table; product-of-sums is simpler when the output is FALSE on only a few rows of a truth table.

2.3 BOOLEAN ALGEBRA

In the previous section, we learned how to write a Boolean expression given a truth table. However, that expression does not necessarily lead to the simplest set of logic gates. Just as you use algebra to simplify mathematical equations, you can use *Boolean algebra* to simplify Boolean equations. The rules of Boolean algebra are much like those of ordinary algebra but are in some cases simpler, because variables have only two possible values: 0 or 1.

Boolean algebra is based on a set of axioms that we assume are correct. Axioms are unprovable in the sense that a definition cannot be proved. From these axioms, we prove all the theorems of Boolean algebra. These theorems have great practical significance, because they teach us how to simplify logic to produce smaller and less costly circuits.

Table 2.1 Axioms of Boolean algebra

	Axiom		Dual	Name
A1	$B = 0$ if $B \neq 1$	A1′	$B = 1$ if $B \neq 0$	Binary field
A2	$\overline{0} = 1$	A2′	$\overline{1} = 0$	NOT
A3	$0 \bullet 0 = 0$	A3′	$1 + 1 = 1$	AND/OR
A4	$1 \bullet 1 = 1$	A4′	$0 + 0 = 0$	AND/OR
A5	$0 \bullet 1 = 1 \bullet 0 = 0$	A5′	$1 + 0 = 0 + 1 = 1$	AND/OR

Axioms and theorems of Boolean algebra obey the principle of *duality*. If the symbols 0 and 1 and the operators • (AND) and + (OR) are interchanged, the statement will still be correct. We use the prime (′) symbol to denote the *dual* of a statement.

2.3.1 Axioms

Table 2.1 states the axioms of Boolean algebra. These five axioms and their duals define Boolean variables and the meanings of NOT, AND, and OR. Axiom A1 states that a Boolean variable B is 0 if it is not 1. The axiom's dual, A1′, states that the variable is 1 if it is not 0. Together, A1 and A1′ tell us that we are working in a Boolean or binary field of 0's and 1's. Axioms A2 and A2′ define the NOT operation. Axioms A3 to A5 define AND; their duals, A3′ to A5′ define OR.

2.3.2 Theorems of One Variable

Theorems T1 to T5 in Table 2.2 describe how to simplify equations involving one variable.

The *identity* theorem, T1, states that for any Boolean variable B, B AND $1 = B$. Its dual states that B OR $0 = B$. In hardware, as shown in Figure 2.14, T1 means that if one input of a two-input AND gate is always 1, we can remove the AND gate and replace it with a wire connected to the variable input (B). Likewise, T1′ means that if one input of a two-input OR gate is always 0, we can replace the OR gate with a wire connected to B. In general, gates cost money, power, and delay, so replacing a gate with a wire is beneficial.

The *null element theorem,* T2, says that B AND 0 is always equal to 0. Therefore, 0 is called the *null* element for the AND operation, because it nullifies the effect of any other input. The dual states that B OR 1 is always equal to 1. Hence, 1 is the null element for the OR operation. In hardware, as shown in Figure 2.15, if one input of an AND gate is 0, we can replace the AND gate with a wire that is tied

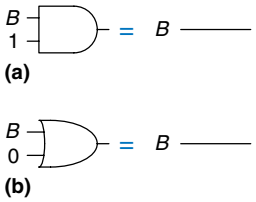


Figure 2.14 Identity theorem in hardware: (a) T1, (b) T1′

The null element theorem leads to some outlandish statements that are actually true! It is particularly dangerous when left in the hands of advertisers: YOU WILL GET A MILLION DOLLARS or we'll send you a toothbrush in the mail. (You'll most likely be receiving a toothbrush in the mail.)

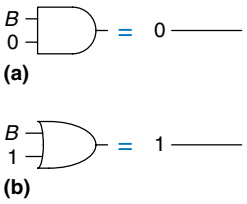


Figure 2.15 Null element theorem in hardware: (a) T2, (b) T2′

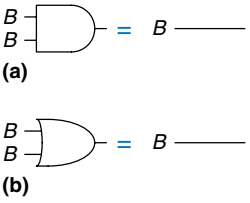


Figure 2.16 Idempotency theorem in hardware: (a) T3, (b) T3′

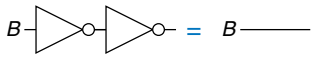


Figure 2.17 Involution theorem in hardware: T4

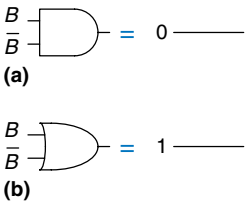


Figure 2.18 Complement theorem in hardware: (a) T5, (b) T5′

Table 2.2 Boolean theorems of one variable

	Theorem		Dual	Name
T1	$B \bullet 1 = B$	T1′	$B + 0 = B$	Identity
T2	$B \bullet 0 = 0$	T2′	$B + 1 = 1$	Null Element
T3	$B \bullet B = B$	T3′	$B + B = B$	Idempotency
T4		$\overline{\overline{B}} = B$		Involution
T5	$B \bullet \overline{B} = 0$	T5′	$B + \overline{B} = 1$	Complements

LOW (to 0). Likewise, if one input of an OR gate is 1, we can replace the OR gate with a wire that is tied HIGH (to 1).

Idempotency, T3, says that a variable AND itself is equal to just itself. Likewise, a variable OR itself is equal to itself. The theorem gets its name from the Latin roots: *idem* (same) and *potent* (power). The operations return the same thing you put into them. Figure 2.16 shows that idempotency again permits replacing a gate with a wire.

Involution, T4, is a fancy way of saying that complementing a variable twice results in the original variable. In digital electronics, two wrongs make a right. Two inverters in series logically cancel each other out and are logically equivalent to a wire, as shown in Figure 2.17. The dual of T4 is itself.

The *complement theorem,* T5 (Figure 2.18), states that a variable AND its complement is 0 (because one of them has to be 0). And, by duality, a variable OR its complement is 1 (because one of them has to be 1).

2.3.3 Theorems of Several Variables

Theorems T6 to T12 in Table 2.3 describe how to simplify equations involving more than one Boolean variable.

Commutativity and *associativity,* T6 and T7, work the same as in traditional algebra. By commutativity, the *order* of inputs for an AND or OR function does not affect the value of the output. By associativity, the specific groupings of inputs do not affect the value of the output.

The *distributivity theorem,* T8, is the same as in traditional algebra, but its dual, T8′, is not. By T8, AND distributes over OR, and by T8′, OR distributes over AND. In traditional algebra, multiplication distributes over addition but addition does not distribute over multiplication, so that $(B + C) \times (B + D) \neq B + (C \times D)$.

The *covering, combining,* and *consensus* theorems, T9 to T11, permit us to eliminate redundant variables. With some thought, you should be able to convince yourself that these theorems are correct.

Table 2.3 Boolean theorems of several variables

	Theorem		Dual	Name
T6	$B \bullet C = C \bullet B$	T6′	$B + C = C + B$	Commutativity
T7	$(B \bullet C) \bullet D = B \bullet (C \bullet D)$	T7′	$(B + C) + D = B + (C + D)$	Associativity
T8	$(B \bullet C) + (B \bullet D) = B \bullet (C + D)$	T8′	$(B + C) \bullet (B + D) = B + (C \bullet D)$	Distributivity
T9	$B \bullet (B + C) = B$	T9′	$B + (B \bullet C) = B$	Covering
T10	$(B \bullet C) + (B \bullet \overline{C}) = B$	T10′	$(B + C) \bullet (B + \overline{C}) = B$	Combining
T11	$(B \bullet C) + (\overline{B} \bullet D) + (C \bullet D)$ $= B \bullet C + \overline{B} \bullet D$	T11′	$(B + C) \bullet (\overline{B} + D) \bullet (C + D)$ $= (B + C) \bullet (\overline{B} + D)$	Consensus
T12	$\overline{B_0 \bullet B_1 \bullet B_2 \ldots}$ $= (\overline{B_0} + \overline{B_1} + \overline{B_2} \ldots)$	T12′	$\overline{B_0 + B_1 + B_2 \ldots}$ $= (\overline{B_0} \bullet \overline{B_1} \bullet \overline{B_2})$	De Morgan's Theorem

De Morgan's Theorem, T12, is a particularly powerful tool in digital design. The theorem explains that the complement of the product of all the terms is equal to the sum of the complement of each term. Likewise, the complement of the sum of all the terms is equal to the product of the complement of each term.

According to De Morgan's theorem, a NAND gate is equivalent to an OR gate with inverted inputs. Similarly, a NOR gate is equivalent to an AND gate with inverted inputs. Figure 2.19 shows these *De Morgan equivalent gates* for NAND and NOR gates. The two symbols shown for each function are called *duals*. They are logically equivalent and can be used interchangeably.

Augustus De Morgan, died 1871. A British mathematician, born in India. Blind in one eye. His father died when he was 10. Attended Trinity College, Cambridge, at age 16, and was appointed Professor of Mathematics at the newly founded London University at age 22. Wrote widely on many mathematical subjects, including logic, algebra, and paradoxes. De Morgan's crater on the moon is named for him. He proposed a riddle for the year of his birth: "I was x years of age in the year x^2."

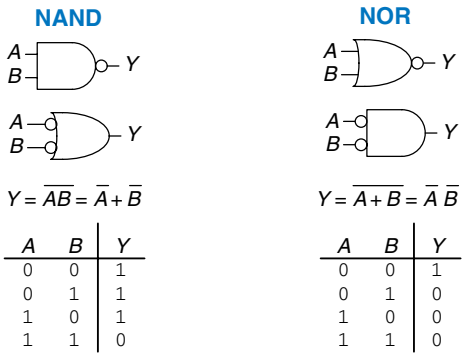


Figure 2.19 De Morgan equivalent gates

The inversion circle is called a *bubble*. Intuitively, you can imagine that "pushing" a bubble through the gate causes it to come out at the other side and flips the body of the gate from AND to OR or vice versa. For example, the NAND gate in Figure 2.19 consists of an AND body with a bubble on the output. Pushing the bubble to the left results in an OR body with bubbles on the inputs. The underlying rules for bubble pushing are

- Pushing bubbles backward (from the output) or forward (from the inputs) changes the body of the gate from AND to OR or vice versa.
- Pushing a bubble from the output back to the inputs puts bubbles on all gate inputs.
- Pushing bubbles on *all* gate inputs forward toward the output puts a bubble on the output.

Section 2.5.2 uses bubble pushing to help analyze circuits.

A	B	Y	$\overline{Y}$
0	0	0	1
0	1	0	1
1	0	1	0
1	1	1	0

FIGURE 2.20 Truth table showing Y and $\overline{Y}$

A	B	Y	$\overline{Y}$	minterm
0	0	0	**1**	$\overline{A}\,\overline{B}$
0	1	0	**1**	$\overline{A}\,B$
1	0	1	**0**	$A\,\overline{B}$
1	1	1	**0**	$A\,B$

Figure 2.21 Truth table showing minterms for $\overline{Y}$

Example 2.4 DERIVE THE PRODUCT-OF-SUMS FORM

Figure 2.20 shows the truth table for a Boolean function, Y, and its complement, $\overline{Y}$. Using De Morgan's Theorem, derive the product-of-sums canonical form of Y from the sum-of-products form of $\overline{Y}$.

Solution: Figure 2.21 shows the minterms (circled) contained in $\overline{Y}$. The sum-of-products canonical form of $\overline{Y}$ is

$$\overline{Y} = \overline{A}B + A\overline{B} \tag{2.3}$$

Taking the complement of both sides and applying De Morgan's Theorem twice, we get:

$$\overline{\overline{Y}} = Y = \overline{\overline{A}B + A\overline{B}} = (\overline{\overline{A}B})(\overline{A\overline{B}}) = (A + B)(\overline{A} + \overline{B}) \tag{2.4}$$

2.3.4 The Truth Behind It All

The curious reader might wonder how to prove that a theorem is true. In Boolean algebra, proofs of theorems with a finite number of variables are easy: just show that the theorem holds for all possible values of these variables. This method is called *perfect induction* and can be done with a truth table.

Example 2.5 PROVING THE CONSENSUS THEOREM

Prove the consensus theorem, T11, from Table 2.3.

Solution: Check both sides of the equation for all eight combinations of B, C, and D. The truth table in Figure 2.22 illustrates these combinations. Because $BC + \overline{B}D + CD = BC + \overline{B}D$ for all cases, the theorem is proved.

B	C	D	$BC + \overline{B}D + CD$	$BC + \overline{B}D$
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	1	1
1	0	0	0	0
1	0	1	0	0
1	1	0	1	1
1	1	1	1	1

Figure 2.22 Truth table proving T11

2.3.5 Simplifying Equations

The theorems of Boolean algebra help us simplify Boolean equations. For example, consider the sum-of-products expression from the truth table of Figure 2.9: $Y = \overline{A}\,\overline{B} + A\overline{B}$. By Theorem T10, the equation simplifies to $Y = \overline{B}$. This may have been obvious looking at the truth table. In general, multiple steps may be necessary to simplify more complex equations.

The basic principle of simplifying sum-of-products equations is to combine terms using the relationship $PA + P\overline{A} = P$, where P may be any implicant. How far can an equation be simplified? We define an equation in sum-of-products form to be *minimized* if it uses the fewest possible implicants. If there are several equations with the same number of implicants, the minimal one is the one with the fewest literals.

An implicant is called a *prime implicant* if it cannot be combined with any other implicants to form a new implicant with fewer literals. The implicants in a minimal equation must all be prime implicants. Otherwise, they could be combined to reduce the number of literals.

Example 2.6 EQUATION MINIMIZATION

Minimize Equation 2.2: $\overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}\,C$.

Solution: We start with the original equation and apply Boolean theorems step by step, as shown in Table 2.4.

Have we simplified the equation completely at this point? Let's take a closer look. From the original equation, the minterms $\overline{A}\,\overline{B}\,\overline{C}$ and $A\overline{B}\,\overline{C}$ differ only in the variable A. So we combined the minterms to form $\overline{B}\,\overline{C}$. However, if we look at the original equation, we note that the last two minterms $A\overline{B}\,\overline{C}$ and $A\overline{B}C$ also differ by a single literal (C and $\overline{C}$). Thus, using the same method, we could have combined these two minterms to form the minterm $A\overline{B}$. We say that implicants $\overline{B}\,\overline{C}$ and $A\overline{B}$ *share* the minterm $A\overline{B}\,\overline{C}$.

So, are we stuck with simplifying only one of the minterm pairs, or can we simplify both? Using the idempotency theorem, we can duplicate terms as many times as we want: $B = B + B + B + B \ldots$. Using this principle, we simplify the equation completely to its two prime implicants, $\overline{B}\,\overline{C} + A\overline{B}$, as shown in Table 2.5.

Table 2.4 Equation minimization

Step	Equation	Justification
	$\overline{A}B\overline{C} + A\overline{B}\overline{C} + A\overline{B}C$	
1	$\overline{B}\overline{C}\,(\overline{A} + A) + A\overline{B}C$	T8: Distributivity
2	$\overline{B}\overline{C}\,(1) + A\overline{B}C$	T5: Complements
3	$\overline{B}\overline{C} + A\overline{B}C$	T1: Identity

Table 2.5 Improved equation minimization

Step	Equation	Justification
	$\overline{A}B\overline{C} + A\overline{B}\overline{C} + A\overline{B}C$	
1	$\overline{A}B\overline{C} + A\overline{B}\overline{C} + A\overline{B}\overline{C} + A\overline{B}C$	T3: Idempotency
2	$\overline{B}\overline{C}(\overline{A} + A) + A\overline{B}(\overline{C} + C)$	T8: Distributivity
3	$\overline{B}\overline{C}(1) + A\overline{B}(1)$	T5: Complements
4	$\overline{B}\overline{C} + A\overline{B}$	T1: Identity

Although it is a bit counterintuitive, *expanding* an implicant (for example, turning AB into $ABC + AB\overline{C}$) is sometimes useful in minimizing equations. By doing this, you can repeat one of the expanded minterms to be combined (shared) with another minterm.

You may have noticed that completely simplifying a Boolean equation with the theorems of Boolean algebra can take some trial and error. Section 2.7 describes a methodical technique called Karnaugh maps that makes the process easier.

Why bother simplifying a Boolean equation if it remains logically equivalent? Simplifying reduces the number of gates used to physically implement the function, thus making it smaller, cheaper, and possibly faster. The next section describes how to implement Boolean equations with logic gates.

2.4 FROM LOGIC TO GATES

A *schematic* is a diagram of a digital circuit showing the elements and the wires that connect them together. For example, the schematic in Figure 2.23 shows a possible hardware implementation of our favorite logic function, Equation 2.2:

$$Y = \overline{A}B\overline{C} + A\overline{B}\overline{C} + A\overline{B}C.$$

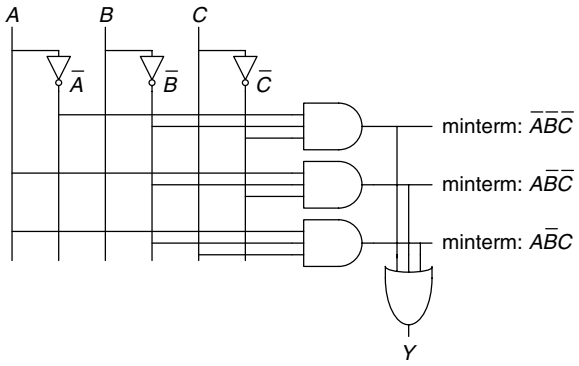


Figure 2.23 Schematic of $Y = \bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}C$

By drawing schematics in a consistent fashion, we make them easier to read and debug. We will generally obey the following guidelines:

- Inputs are on the left (or top) side of a schematic.
- Outputs are on the right (or bottom) side of a schematic.
- Whenever possible, gates should flow from left to right.
- Straight wires are better to use than wires with multiple corners (jagged wires waste mental effort following the wire rather than thinking of what the circuit does).
- Wires always connect at a T junction.
- A dot where wires cross indicates a connection between the wires.
- Wires crossing *without* a dot make no connection.

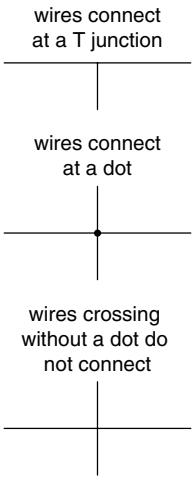


Figure 2.24 Wire connections

The last three guidelines are illustrated in Figure 2.24.

Any Boolean equation in sum-of-products form can be drawn as a schematic in a systematic way similar to Figure 2.23. First, draw columns for the inputs. Place inverters in adjacent columns to provide the complementary inputs if necessary. Draw rows of AND gates for each of the minterms. Then, for each output, draw an OR gate connected to the minterms related to that output. This style is called a *programmable logic array (PLA)* because the inverters, AND gates, and OR gates are arrayed in a systematic fashion. PLAs will be discussed further in Section 5.6.

Figure 2.25 shows an implementation of the simplified equation we found using Boolean algebra in Example 2.6. Notice that the simplified circuit has significantly less hardware than that of Figure 2.23. It may also be faster, because it uses gates with fewer inputs.

We can reduce the number of gates even further (albeit by a single inverter) by taking advantage of inverting gates. Observe that $\bar{B}\bar{C}$ is an

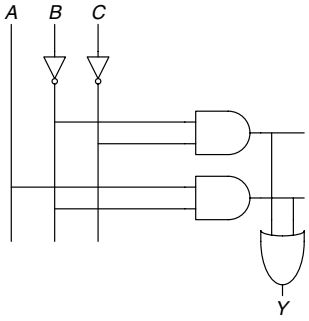


Figure 2.25 Schematic of $Y = \bar{B}\bar{C} + A\bar{B}$

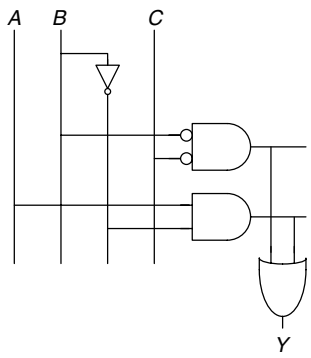


Figure 2.26 Schematic using fewer gates

AND with inverted inputs. Figure 2.26 shows a schematic using this optimization to eliminate the inverter on C. Recall that by De Morgan's theorem the AND with inverted inputs is equivalent to a NOR gate. Depending on the implementation technology, it may be cheaper to use the fewest gates or to use certain types of gates in preference to others. For example, NANDs and NORs are preferred over ANDs and ORs in CMOS implementations.

Many circuits have multiple outputs, each of which computes a separate Boolean function of the inputs. We can write a separate truth table for each output, but it is often convenient to write all of the outputs on a single truth table and sketch one schematic with all of the outputs.

Example 2.7 MULTIPLE-OUTPUT CIRCUITS

The dean, the department chair, the teaching assistant, and the dorm social chair each use the auditorium from time to time. Unfortunately, they occasionally conflict, leading to disasters such as the one that occurred when the dean's fundraising meeting with crusty trustees happened at the same time as the dorm's BTB[1] party. Alyssa P. Hacker has been called in to design a room reservation system.

The system has four inputs, $A_3, \ldots, A_0$, and four outputs, $Y_3, \ldots, Y_0$. These signals can also be written as $A_{3:0}$ and $Y_{3:0}$. Each user asserts her input when she requests the auditorium for the next day. The system asserts at most one output, granting the auditorium to the highest priority user. The dean, who is paying for the system, demands highest priority (3). The department chair, teaching assistant, and dorm social chair have decreasing priority.

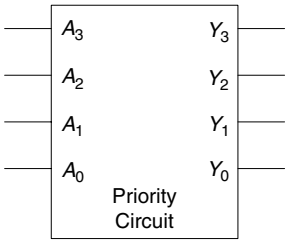
Write a truth table and Boolean equations for the system. Sketch a circuit that performs this function.

Solution: This function is called a four-input *priority circuit*. Its symbol and truth table are shown in Figure 2.27.

We could write each output in sum-of-products form and reduce the equations using Boolean algebra. However, the simplified equations are clear by inspection from the functional description (and the truth table): Y_3 is TRUE whenever A_3 is asserted, so $Y_3 = A_3$. Y_2 is TRUE if A_2 is asserted and A_3 is not asserted, so $Y_2 = \overline{A}_3 A_2$. Y_1 is TRUE if A_1 is asserted and neither of the higher priority inputs is asserted: $Y_1 = \overline{A}_3 \overline{A}_2 A_1$. And Y_0 is TRUE whenever A_0 and no other input is asserted: $Y_0 = \overline{A}_3 \overline{A}_2 \overline{A}_1 A_0$. The schematic is shown in Figure 2.28. An experienced designer can often implement a logic circuit by inspection. Given a clear specification, simply turn the words into equations and the equations into gates.

---

[1] Black light, twinkies, and beer.



A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	0
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	1	0	0	0
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	0
1	1	1	0	1	0	0	0
1	1	1	1	1	0	0	0

Figure 2.27 Priority circuit

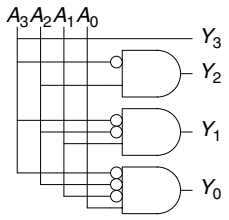


Figure 2.28 Priority circuit schematic

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	X	0	0	1	0
0	1	X	X	0	1	0	0
1	X	X	X	1	0	0	0

Figure 2.29 Priority circuit truth table with don't cares (X's)

Notice that if A_3 is asserted in the priority circuit, the outputs *don't care* what the other inputs are. We use the symbol X to describe inputs that the output doesn't care about. Figure 2.29 shows that the four-input priority circuit truth table becomes much smaller with don't cares. From this truth table, we can easily read the Boolean equations in sum-of-products form by ignoring inputs with X's. Don't cares can also appear in truth table outputs, as we will see in Section 2.7.3.

X is an overloaded symbol that means "don't care" in truth tables and "contention" in logic simulation (see Section 2.6.1). Think about the context so you don't mix up the meanings. Some authors use D or ? instead for "don't care" to avoid this ambiguity.

2.5 MULTILEVEL COMBINATIONAL LOGIC

Logic in sum-of-products form is called *two-level logic* because it consists of literals connected to a level of AND gates connected to a level of

OR gates. Designers often build circuits with more than two levels of logic gates. These multilevel combinational circuits may use less hardware than their two-level counterparts. Bubble pushing is especially helpful in analyzing and designing multilevel circuits.

2.5.1 Hardware Reduction

Some logic functions require an enormous amount of hardware when built using two-level logic. A notable example is the XOR function of multiple variables. For example consider building a three-input XOR using the two-level techniques we have studied so far.

Recall that an N-input XOR produces a TRUE output when an odd number of inputs are TRUE. Figure 2.30 shows the truth table for a three-input XOR with the rows circled that produce TRUE outputs. From the truth table, we read off a Boolean equation in sum-of-products form in Equation 2.5. Unfortunately, there is no way to simplify this equation into fewer implicants.

$$Y = \overline{A}\,\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\,\overline{C} + ABC \tag{2.5}$$

On the other hand, $A \oplus B \oplus C = (A \oplus B) \oplus C$ (prove this to yourself by perfect induction if you are in doubt). Therefore, the three-input XOR can be built out of a cascade of two-input XORs, as shown in Figure 2.31.

Similarly, an eight-input XOR would require 128 eight-input AND gates and one 128-input OR gate for a two-level sum-of-products implementation. A much better option is to use a tree of two-input XOR gates, as shown in Figure 2.32.

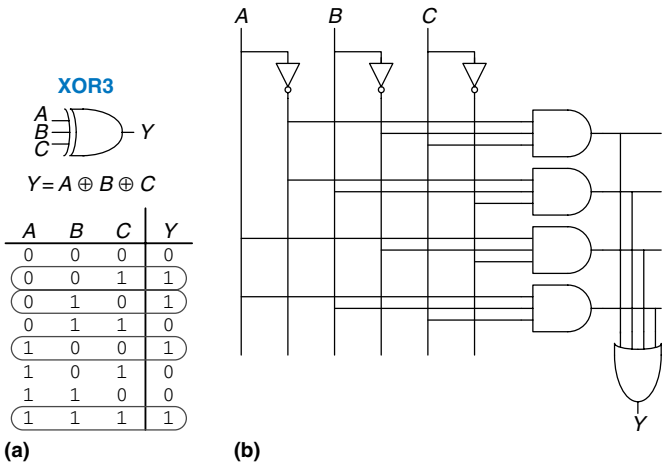


Figure 2.30 Three-input XOR: (a) functional specification and (b) two-level logic implementation

Selecting the best multilevel implementation of a specific logic function is not a simple process. Moreover, "best" has many meanings: fewest gates, fastest, shortest design time, least cost, least power consumption. In Chapter 5, you will see that the "best" circuit in one technology is not necessarily the best in another. For example, we have been using ANDs and ORs, but in CMOS, NANDs and NORs are more efficient. With some experience, you will find that you can create a good multilevel design by inspection for most circuits. You will develop some of this experience as you study circuit examples through the rest of this book. As you are learning, explore various design options and think about the trade-offs. Computer-aided design (CAD) tools are also available to search a vast space of possible multilevel designs and seek the one that best fits your constraints given the available building blocks.

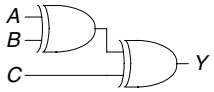


Figure 2.31 Three-input XOR using two two-input XORs

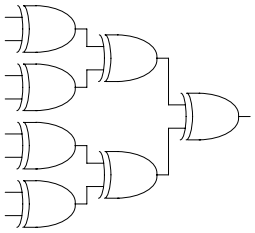


Figure 2.32 Eight-input XOR using seven two-input XORs

2.5.2 Bubble Pushing

You may recall from Section 1.7.6 that CMOS circuits prefer NANDs and NORs over ANDs and ORs. But reading the equation by inspection from a multilevel circuit with NANDs and NORs can get pretty hairy. Figure 2.33 shows a multilevel circuit whose function is not immediately clear by inspection. Bubble pushing is a helpful way to redraw these circuits so that the bubbles cancel out and the function can be more easily determined. Building on the principles from Section 2.3.3, the guidelines for bubble pushing are as follows:

- Begin at the output of the circuit and work toward the inputs.
- Push any bubbles on the final output back toward the inputs so that you can read an equation in terms of the output (for example, Y) instead of the complement of the output ($\overline{Y}$).
- Working backward, draw each gate in a form so that bubbles cancel. If the current gate has an input bubble, draw the preceding gate with an output bubble. If the current gate does not have an input bubble, draw the preceding gate without an output bubble.

Figure 2.34 shows how to redraw Figure 2.33 according to the bubble pushing guidelines. Starting at the output, Y, the NAND gate has a bubble on the output that we wish to eliminate. We push the output bubble back to form an OR with inverted inputs, shown in

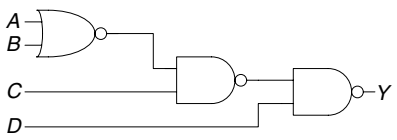


Figure 2.33 Multilevel circuit using NANDs and NORs

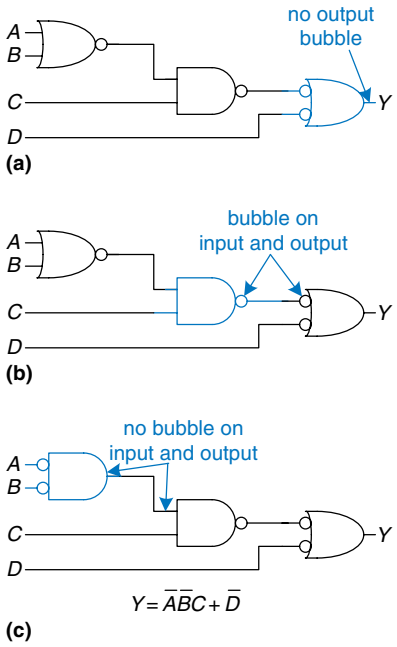


Figure 2.34 Bubble-pushed circuit

Figure 2.34(a). Working to the left, the rightmost gate has an input bubble that cancels with the output bubble of the middle NAND gate, so no change is necessary, as shown in Figure 2.34(b). The middle gate has no input bubble, so we transform the leftmost gate to have no output bubble, as shown in Figure 2.34(c). Now all of the bubbles in the circuit cancel except at the inputs, so the function can be read by inspection in terms of ANDs and ORs of true or complementary inputs: $Y = \overline{A}\overline{B}C + \overline{D}$.

For emphasis of this last point, Figure 2.35 shows a circuit logically equivalent to the one in Figure 2.34. The functions of internal nodes are labeled in blue. Because bubbles in series cancel, we can ignore the bubble on the output of the middle gate and the input of the rightmost gate to produce the logically equivalent circuit of Figure 2.35.

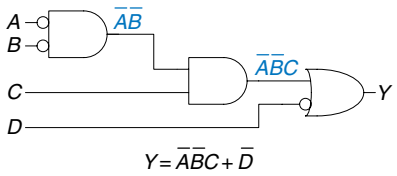


Figure 2.35 Logically equivalent bubble-pushed circuit

Example 2.8 BUBBLE PUSHING FOR CMOS LOGIC

Most designers think in terms of AND and OR gates, but suppose you would like to implement the circuit in Figure 2.36 in CMOS logic, which favors NAND and NOR gates. Use bubble pushing to convert the circuit to NANDs, NORs, and inverters.

Solution: A brute force solution is to just replace each AND gate with a NAND and an inverter, and each OR gate with a NOR and an inverter, as shown in Figure 2.37. This requires eight gates. Notice that the inverter is drawn with the bubble on the front rather than back, to emphasize how the bubble can cancel with the preceding inverting gate.

For a better solution, observe that bubbles can be added to the output of a gate and the input of the next gate without changing the function, as shown in Figure 2.38(a). The final AND is converted to a NAND and an inverter, as shown in Figure 2.38(b). This solution requires only five gates.

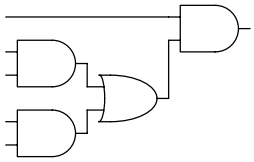


Figure 2.36 Circuit using ANDs and ORs

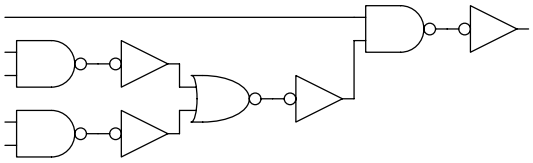


Figure 2.37 Poor circuit using NANDs and NORs

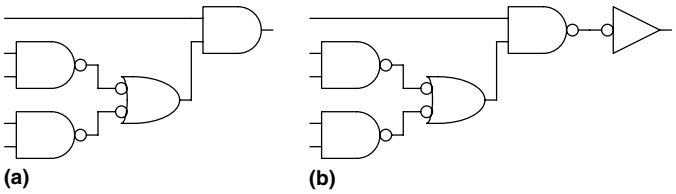


Figure 2.38 Better circuit using NANDs and NORs

2.6 X'S AND Z'S, OH MY

Boolean algebra is limited to 0's and 1's. However, real circuits can also have illegal and floating values, represented symbolically by X and Z.

2.6.1 Illegal Value: X

The symbol X indicates that the circuit node has an *unknown* or *illegal* value. This commonly happens if it is being driven to both 0 and 1 at the same time. Figure 2.39 shows a case where node *Y* is driven both HIGH and LOW. This situation, called *contention,* is considered to be an error

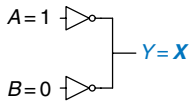


Figure 2.39 Circuit with contention

and must be avoided. The actual voltage on a node with contention may be somewhere between 0 and V_{DD}, depending on the relative strengths of the gates driving HIGH and LOW. It is often, but not always, in the forbidden zone. Contention also can cause large amounts of power to flow between the fighting gates, resulting in the circuit getting hot and possibly damaged.

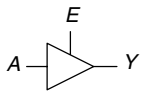
X values are also sometimes used by circuit simulators to indicate an uninitialized value. For example, if you forget to specify the value of an input, the simulator may assume it is an X to warn you of the problem.

As mentioned in Section 2.4, digital designers also use the symbol X to indicate "don't care" values in truth tables. Be sure not to mix up the two meanings. When X appears in a truth table, it indicates that the value of the variable in the truth table is unimportant. When X appears in a circuit, it means that the circuit node has an unknown or illegal value.

2.6.2 Floating Value: Z

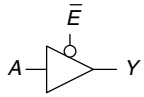
The symbol Z indicates that a node is being driven neither HIGH nor LOW. The node is said to be *floating, high impedance,* or *high Z*. A typical misconception is that a floating or undriven node is the same as a logic 0. In reality, a floating node might be 0, might be 1, or might be at some voltage in between, depending on the history of the system. A floating node does not always mean there is an error in the circuit, so long as some other circuit element does drive the node to a valid logic level when the value of the node is relevant to circuit operation.

One common way to produce a floating node is to forget to connect a voltage to a circuit input, or to assume that an unconnected input is the same as an input with the value of 0. This mistake may cause the circuit to behave erratically as the floating input randomly changes from 0 to 1. Indeed, touching the circuit may be enough to trigger the change by means of static electricity from the body. We have seen circuits that operate correctly only as long as the student keeps a finger pressed on a chip.



E	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

Figure 2.40 Tristate buffer



$\bar{E}$	A	Y
0	0	0
0	1	1
1	0	Z
1	1	Z

Figure 2.41 Tristate buffer with active low enable

The *tristate buffer,* shown in Figure 2.40, has three possible output states: HIGH (1), LOW (0), and floating (Z). The tristate buffer has an input, *A,* an output, *Y,* and an *enable, E*. When the enable is TRUE, the tristate buffer acts as a simple buffer, transferring the input value to the output. When the enable is FALSE, the output is allowed to float (Z).

The tristate buffer in Figure 2.40 has an *active high* enable. That is, when the enable is HIGH (1), the buffer is enabled. Figure 2.41 shows a tristate buffer with an *active low* enable. When the enable is LOW (0),

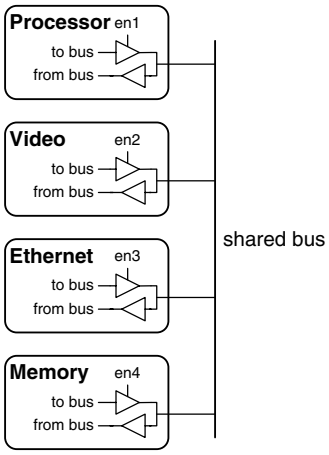


Figure 2.42 Tristate bus connecting multiple chips

the buffer is enabled. We show that the signal is active low by putting a bubble on its input wire. We often indicate an active low input by drawing a bar over its name, $\overline{E}$, or appending the word "bar" after its name, *Ebar*.

Tristate buffers are commonly used on *busses* that connect multiple chips. For example, a microprocessor, a video controller, and an Ethernet controller might all need to communicate with the memory system in a personal computer. Each chip can connect to a shared memory bus using tristate buffers, as shown in Figure 2.42. Only one chip at a time is allowed to assert its enable signal to drive a value onto the bus. The other chips must produce floating outputs so that they do not cause contention with the chip talking to the memory. Any chip can read the information from the shared bus at any time. Such tristate busses were once common. However, in modern computers, higher speeds are possible with *point-to-point links*, in which chips are connected to each other directly rather than over a shared bus.

2.7 KARNAUGH MAPS

After working through several minimizations of Boolean equations using Boolean algebra, you will realize that, if you're not careful, you sometimes end up with a completely *different* equation instead of a simplified equation. *Karnaugh maps* (*K-maps*) are a graphical method for simplifying Boolean equations. They were invented in 1953 by Maurice Karnaugh, a telecommunications engineer at Bell Labs. K-maps work well for problems with up to four variables. More important, they give insight into manipulating Boolean equations.

Maurice Karnaugh, 1924–. Graduated with a bachelor's degree in physics from the City College of New York in 1948 and earned a Ph.D. in physics from Yale in 1952. Worked at Bell Labs and IBM from 1952 to 1993 and as a computer science professor at the Polytechnic University of New York from 1980 to 1999.

Gray codes were patented (U.S. Patent 2,632,058) by Frank Gray, a Bell Labs researcher, in 1953. They are especially useful in mechanical encoders because a slight misalignment causes an error in only one bit.

Gray codes generalize to any number of bits. For example, a 3-bit Gray code sequence is:

000, 001, 011, 010, 110, 111, 101, 100

Lewis Carroll posed a related puzzle in *Vanity Fair* in 1879.

> "The rules of the Puzzle are simple enough. Two words are proposed, of the same length; and the puzzle consists of linking these together by interposing other words, each of which shall differ from the next word in one letter only. That is to say, one letter may be changed in one of the given words, then one letter in the word so obtained, and so on, till we arrive at the other given word."

For example, SHIP to DOCK:

SHIP, SLIP, SLOP, SLOT, SOOT, LOOT, LOOK, LOCK, DOCK.

Can you find a shorter sequence?

Recall that logic minimization involves combining terms. Two terms containing an implicant, P, and the true and complementary forms of some variable, A, are combined to eliminate A: $PA + P\overline{A} = P$. Karnaugh maps make these combinable terms easy to see by putting them next to each other in a grid.

Figure 2.43 shows the truth table and K-map for a three-input function. The top row of the K-map gives the four possible values for the A and B inputs. The left column gives the two possible values for the C input. Each square in the K-map corresponds to a row in the truth table and contains the value of the output, Y, for that row. For example, the top left square corresponds to the first row in the truth table and indicates that the output value $Y = 1$ when $ABC = 000$. Just like each row in a truth table, each square in a K-map represents a single minterm. For the purpose of explanation, Figure 2.43(c) shows the minterm corresponding to each square in the K-map.

Each square, or minterm, differs from an adjacent square by a change in a single variable. This means that adjacent squares share all the same literals except one, which appears in true form in one square and in complementary form in the other. For example, the squares representing the minterms $\overline{A}\overline{B}\overline{C}$ and $\overline{A}\overline{B}C$ are adjacent and differ only in the variable C. You may have noticed that the A and B combinations in the top row are in a peculiar order: 00, 01, 11, 10. This order is called a *Gray code*. It differs from ordinary binary order (00, 01, 10, 11) in that adjacent entries differ only in a single variable. For example, 01 : 11 only changes A from 0 to 1, while 01 : 10 would change A from 1 to 0 and B from 0 to 1. Hence, writing the combinations in binary order would not have produced our desired property of adjacent squares differing only in one variable.

The K-map also "wraps around." The squares on the far right are effectively adjacent to the squares on the far left, in that they differ only in one variable, A. In other words, you could take the map and roll it into a cylinder, then join the ends of the cylinder to form a torus (i.e., a donut), and still guarantee that adjacent squares would differ only in one variable.

2.7.1 Circular Thinking

In the K-map in Figure 2.43, only two minterms are present in the equation, $\overline{A}\overline{B}\overline{C}$ and $\overline{A}\overline{B}C$, as indicated by the 1's in the left column. Reading the minterms from the K-map is exactly equivalent to reading equations in sum-of-products form directly from the truth table.

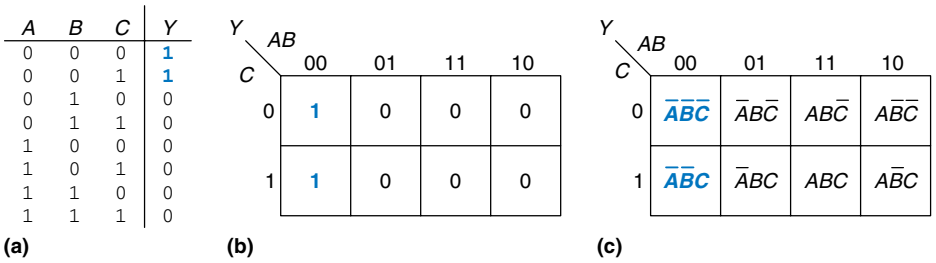


Figure 2.43 Three-input function: (a) truth table, (b) K-map, (c) K-map showing minterms

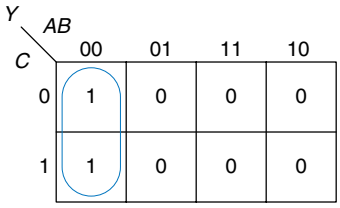


Figure 2.44 K-map minimization

As before, we can use Boolean algebra to minimize equations in sum-of-products form.

$$Y = \overline{A}\overline{B}\overline{C} + \overline{A}\overline{B}C = \overline{A}\overline{B}(\overline{C} + C) = \overline{A}\overline{B} \tag{2.6}$$

K-maps help us do this simplification graphically by *circling* 1's in adjacent squares, as shown in Figure 2.44. For each circle, we write the corresponding implicant. Remember from Section 2.2 that an implicant is the product of one or more literals. Variables whose true *and* complementary forms are both in the circle are excluded from the implicant. In this case, the variable C has both its true form (1) and its complementary form (0) in the circle, so we do not include it in the implicant. In other words, Y is TRUE when $A = B = 0$, independent of C. So the implicant is $\overline{A}\overline{B}$. This K-map gives the same answer we reached using Boolean algebra.

2.7.2 Logic Minimization with K-Maps

K-maps provide an easy visual way to minimize logic. Simply circle all the rectangular blocks of 1's in the map, using the fewest possible number of circles. Each circle should be as large as possible. Then read off the implicants that were circled.

More formally, recall that a Boolean equation is minimized when it is written as a sum of the fewest number of prime implicants. Each circle on the K-map represents an implicant. The largest possible circles are prime implicants.

For example, in the K-map of Figure 2.44, $\overline{A}\overline{B}\overline{C}$ and $\overline{A}\overline{B}C$ are implicants, but *not* prime implicants. Only $\overline{A}\overline{B}$ is a prime implicant in that K-map. Rules for finding a minimized equation from a K-map are as follows:

- Use the fewest circles necessary to cover all the 1's.
- All the squares in each circle must contain 1's.
- Each circle must span a rectangular block that is a power of 2 (i.e., 1, 2, or 4) squares in each direction.
- Each circle should be as large as possible.
- A circle may wrap around the edges of the K-map.
- A 1 in a K-map may be circled multiple times if doing so allows fewer circles to be used.

Example 2.9 MINIMIZATION OF A THREE-VARIABLE FUNCTION USING A K-MAP

Suppose we have the function $Y = F(A, B, C)$ with the K-map shown in Figure 2.45. Minimize the equation using the K-map.

Solution: Circle the 1's in the K-map using as few circles as possible, as shown in Figure 2.46. Each circle in the K-map represents a prime implicant, and the dimension of each circle is a power of two (2×1 and 2×2). We form the prime implicant for each circle by writing those variables that appear in the circle only in true or only in complementary form.

For example, in the 2×1 circle, the true and complementary forms of B are included in the circle, so we *do not* include B in the prime implicant. However, only the true form of $A(A)$ and complementary form of $C(\overline{C})$ are in this circle, so we include these variables in the prime implicant $A\overline{C}$. Similarly, the 2×2 circle covers all squares where $B = 0$, so the prime implicant is $\overline{B}$.

Notice how the top-right square (minterm) is covered twice to make the prime implicant circles as large as possible. As we saw with Boolean algebra techniques, this is equivalent to sharing a minterm to reduce the size of the

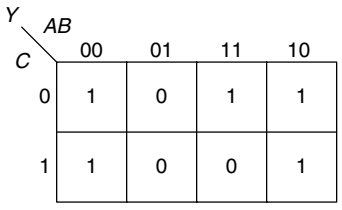


Figure 2.45 K-map for Example 2.9

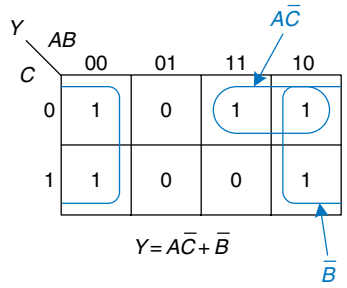


Figure 2.46 Solution for Example 2.9

implicant. Also notice how the circle covering four squares wraps around the sides of the K-map.

Example 2.10 SEVEN-SEGMENT DISPLAY DECODER

A *seven-segment display decoder* takes a 4-bit data input, $D_{3:0}$, and produces seven outputs to control light-emitting diodes to display a digit from 0 to 9. The seven outputs are often called segments *a* through *g*, or S_a–S_g, as defined in Figure 2.47. The digits are shown in Figure 2.48. Write a truth table for the outputs, and use K-maps to find Boolean equations for outputs S_a and S_b. Assume that illegal input values (10–15) produce a blank readout.

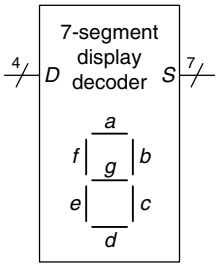


Figure 2.47 Seven-segment display decoder icon

Solution: The truth table is given in Table 2.6. For example, an input of 0000 should turn on all segments except S_g.

Each of the seven outputs is an independent function of four variables. The K-maps for outputs S_a and S_b are shown in Figure 2.49. Remember that adjacent squares may differ in only a single variable, so we label the rows and columns in Gray code order: 00, 01, 11, 10. Be careful to also remember this ordering when *entering* the output values into the squares.

Next, circle the prime implicants. Use the fewest number of circles necessary to cover all the 1's. A circle can wrap around the edges (vertical *and* horizontal), and a 1 may be circled more than once. Figure 2.50 shows the prime implicants and the simplified Boolean equations.

Note that the minimal set of prime implicants is not unique. For example, the 0000 entry in the S_a K-map was circled along with the 1000 entry to produce the $\bar{D}_2\bar{D}_1\bar{D}_0$ minterm. The circle could have included the 0010 entry instead, producing a $\bar{D}_3\bar{D}_2\bar{D}_0$ minterm, as shown with dashed lines in Figure 2.51.

Figure 2.52 illustrates (see page 78) a common error in which a nonprime implicant was chosen to cover the 1 in the upper left corner. This minterm, $\bar{D}_3\bar{D}_2\bar{D}_1\bar{D}_0$, gives a sum-of-products equation that is *not* minimal. The minterm could have been combined with either of the adjacent ones to form a larger circle, as was done in the previous two figures.

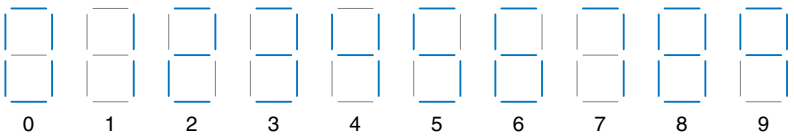


Figure 2.48 Seven-segment display digits

Table 2.6 Seven-segment display decoder truth table

$D_{3:0}$	S_a	S_b	S_c	S_d	S_e	S_f	S_g
0000	1	1	1	1	1	1	0
0001	0	1	1	0	0	0	0
0010	1	1	0	1	1	0	1
0011	1	1	1	1	0	0	1
0100	0	1	1	0	0	1	1
0101	1	0	1	1	0	1	1
0110	1	0	1	1	1	1	1
0111	1	1	1	0	0	0	0
1000	1	1	1	1	1	1	1
1001	1	1	1	0	0	1	1
others	0	0	0	0	0	0	0

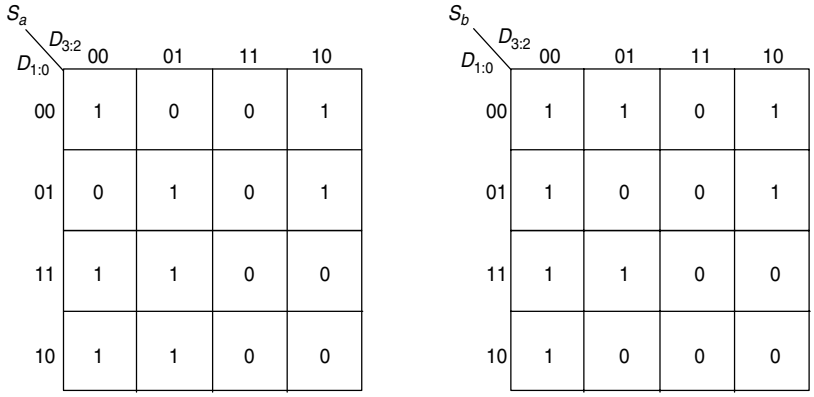


Figure 2.49 Karnaugh maps for S_a and S_b

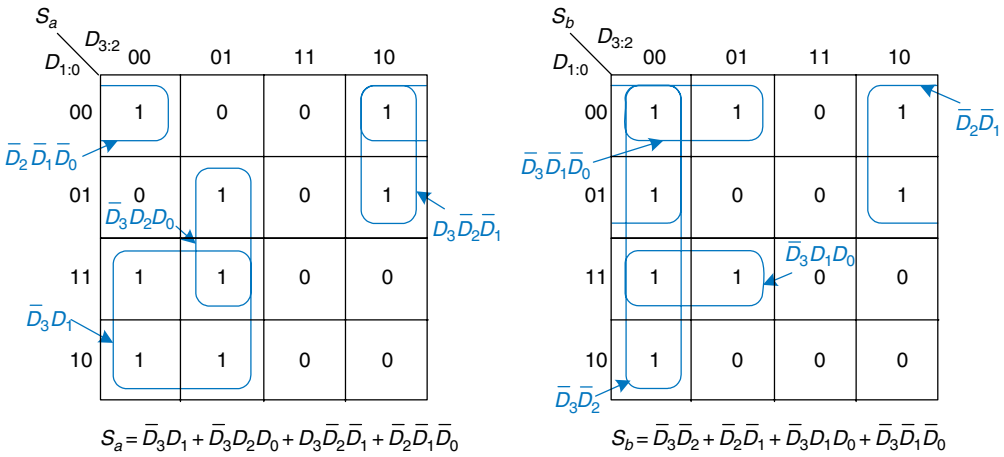


Figure 2.50 K-map solution for Example 2.10

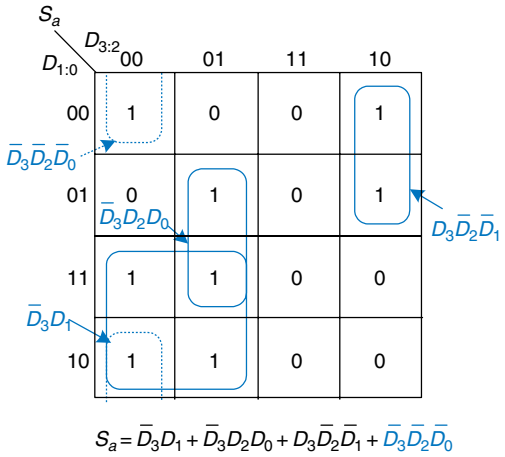


Figure 2.51 Alternative K-map for S_a showing different set of prime implicants

2.7.3 Don't Cares

Recall that "don't care" entries for truth table inputs were introduced in Section 2.4 to reduce the number of rows in the table when some variables do not affect the output. They are indicated by the symbol X, which means that the entry can be either 0 or 1.

Don't cares also appear in truth table outputs where the output value is unimportant or the corresponding input combination can never happen. Such outputs can be treated as either 0's or 1's at the designer's discretion.

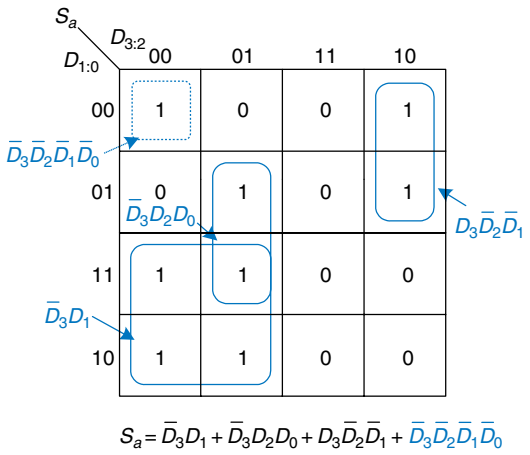


Figure 2.52 **Alternative K-map for S_a showing incorrect nonprime implicant**

In a K-map, X's allow for even more logic minimization. They can be circled if they help cover the 1's with fewer or larger circles, but they do not have to be circled if they are not helpful.

Example 2.11 SEVEN-SEGMENT DISPLAY DECODER WITH DON'T CARES

Repeat Example 2.10 if we don't care about the output values for illegal input values of 10 to 15.

Solution: The K-map is shown in Figure 2.53 with X entries representing don't care. Because don't cares can be 0 or 1, we circle a don't care if it allows us to cover the 1's with fewer or bigger circles. Circled don't cares are treated as 1's, whereas uncircled don't cares are 0's. Observe how a 2×2 square wrapping around all four corners is circled for segment S_a. Use of don't cares simplifies the logic substantially.

2.7.4 The Big Picture

Boolean algebra and Karnaugh maps are two methods for logic simplification. Ultimately, the goal is to find a low-cost method of implementing a particular logic function.

In modern engineering practice, computer programs called *logic synthesizers* produce simplified circuits from a description of the logic function, as we will see in Chapter 4. For large problems, logic synthesizers are much more efficient than humans. For small problems, a human with a bit of experience can find a good solution by inspection. Neither of the authors has ever used a Karnaugh map in real life to

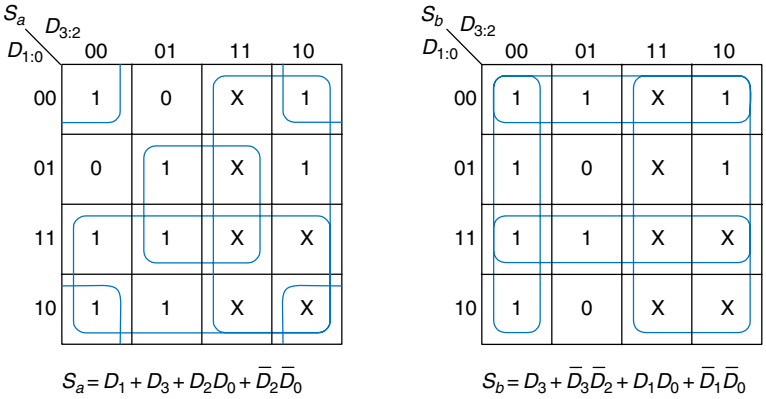


Figure 2.53 K-map solution with don't cares

solve a practical problem. But the insight gained from the principles underlying Karnaugh maps is valuable. And Karnaugh maps often appear at job interviews!

2.8 COMBINATIONAL BUILDING BLOCKS

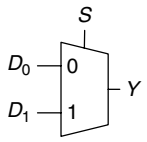
Combinational logic is often grouped into larger building blocks to build more complex systems. This is an application of the principle of abstraction, hiding the unnecessary gate-level details to emphasize the function of the building block. We have already studied three such building blocks: full adders (from Section 2.1), priority circuits (from Section 2.4), and seven-segment display decoders (from Section 2.7). This section introduces two more commonly used building blocks: multiplexers and decoders. Chapter 5 covers other combinational building blocks.

2.8.1 Multiplexers

Multiplexers are among the most commonly used combinational circuits. They choose an output from among several possible inputs based on the value of a *select* signal. A multiplexer is sometimes affectionately called a *mux*.

2:1 Multiplexer

Figure 2.54 shows the schematic and truth table for a 2:1 multiplexer with two data inputs, D_0 and D_1, a select input, S, and one output, Y. The multiplexer chooses between the two data inputs based on the select: if $S = 0$, $Y = D_0$, and if $S = 1$, $Y = D_1$. S is also called a *control signal* because it controls what the multiplexer does.



S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figure 2.54 2:1 multiplexer symbol and truth table

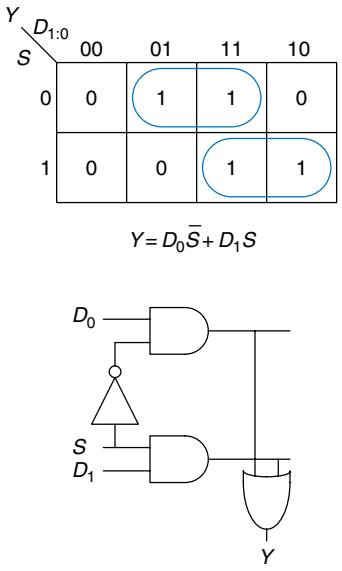


Figure 2.55 2:1 multiplexer implementation using two-level logic

Shorting together the outputs of multiple gates technically violates the rules for combinational circuits given in Section 2.1. But because exactly one of the outputs is driven at any time, this exception is allowed.

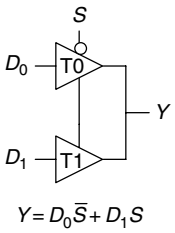


Figure 2.56 Multiplexer using tristate buffers

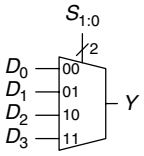


Figure 2.57 4:1 multiplexer

A 2:1 multiplexer can be built from sum-of-products logic as shown in Figure 2.55. The Boolean equation for the multiplexer may be derived with a Karnaugh map or read off by inspection (Y is 1 if $S = 0$ AND D_0 is 1 OR if $S = 1$ AND D_1 is 1).

Alternatively, multiplexers can be built from tristate buffers, as shown in Figure 2.56. The tristate enables are arranged such that, at all times, exactly one tristate buffer is active. When $S = 0$, tristate T0 is enabled, allowing D_0 to flow to Y. When $S = 1$, tristate T1 is enabled, allowing D_1 to flow to Y.

Wider Multiplexers

A 4:1 multiplexer has four data inputs and one output, as shown in Figure 2.57. Two select signals are needed to choose among the four data inputs. The 4:1 multiplexer can be built using sum-of-products logic, tristates, or multiple 2:1 multiplexers, as shown in Figure 2.58.

The product terms enabling the tristates can be formed using AND gates and inverters. They can also be formed using a decoder, which we will introduce in Section 2.8.2.

Wider multiplexers, such as 8:1 and 16:1 multiplexers, can be built by expanding the methods shown in Figure 2.58. In general, an N:1 multiplexer needs $\log_2 N$ select lines. Again, the best implementation choice depends on the target technology.

Multiplexer Logic

Multiplexers can be used as *lookup tables* to perform logic functions. Figure 2.59 shows a 4:1 multiplexer used to implement a two-input

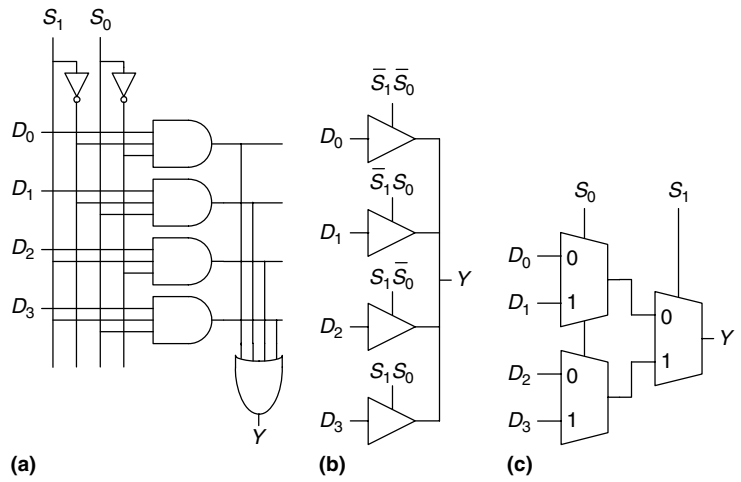


Figure 2.58 4:1 multiplexer implementations: (a) two-level logic, (b) tristates, (c) hierarchical

AND gate. The inputs, A and B, serve as select lines. The multiplexer data inputs are connected to 0 or 1 according to the corresponding row of the truth table. In general, a 2^N-input multiplexer can be programmed to perform any N-input logic function by applying 0's and 1's to the appropriate data inputs. Indeed, by changing the data inputs, the multiplexer can be reprogrammed to perform a different function.

With a little cleverness, we can cut the multiplexer size in half, using only a 2^{N-1}-input multiplexer to perform any N-input logic function. The strategy is to provide one of the literals, as well as 0's and 1's, to the multiplexer data inputs.

To illustrate this principle, Figure 2.60 shows two-input AND and XOR functions implemented with 2:1 multiplexers. We start with an ordinary truth table, and then combine pairs of rows to eliminate the rightmost input variable by expressing the output in terms of this variable. For example, in the case of AND, when $A = 0$, $Y = 0$, regardless of B. When $A = 1$, $Y = 0$ if $B = 0$ and $Y = 1$ if $B = 1$, so $Y = B$. We then use the multiplexer as a lookup table according to the new, smaller truth table.

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

$Y = AB$

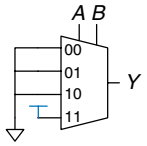


Figure 2.59 4:1 multiplexer implementation of two-input AND function

---

Example 2.12 LOGIC WITH MULTIPLEXERS

Alyssa P. Hacker needs to implement the function $Y = A\overline{B} + \overline{B}\overline{C} + \overline{A}BC$ to finish her senior project, but when she looks in her lab kit, the only part she has left is an 8:1 multiplexer. How does she implement the function?

Solution: Figure 2.61 shows Alyssa's implementation using a single 8:1 multiplexer. The multiplexer acts as a lookup table where each row in the truth table corresponds to a multiplexer input.

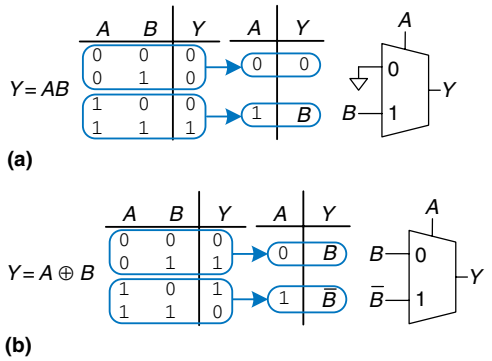


Figure 2.60 Multiplexer logic using variable inputs

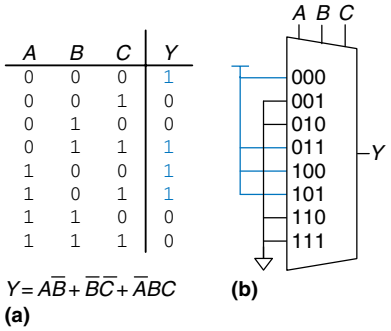


Figure 2.61 Alyssa's circuit: (a) truth table, (b) 8:1 multiplexer implementation

Example 2.13 LOGIC WITH MULTIPLEXERS, REPRISED

Alyssa turns on her circuit one more time before the final presentation and blows up the 8:1 multiplexer. (She accidently powered it with 20 V instead of 5 V after not sleeping all night.) She begs her friends for spare parts and they give her a 4:1 multiplexer and an inverter. Can she build her circuit with only these parts?

Solution: Alyssa reduces her truth table to four rows by letting the output depend on C. (She could also have chosen to rearrange the columns of the truth table to let the output depend on A or B.) Figure 2.62 shows the new design.

2.8.2 Decoders

A decoder has N inputs and 2^N outputs. It asserts exactly one of its outputs depending on the input combination. Figure 2.63 shows a 2:4 decoder. When $A_{1:0} = 00$, Y_0 is 1. When $A_{1:0} = 01$, Y_1 is 1. And so forth. The outputs are called *one-hot*, because exactly one is "hot" (HIGH) at a given time.

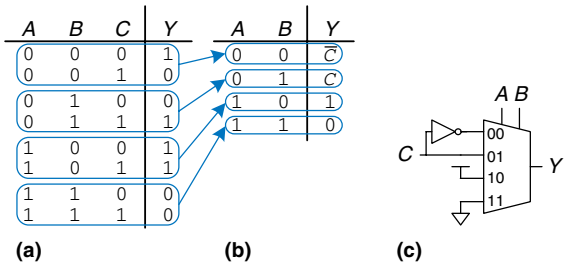
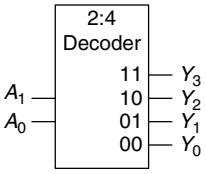


Figure 2.62 Alyssa's new circuit

Example 2.14 DECODER IMPLEMENTATION

Implement a 2:4 decoder with AND, OR, and NOT gates.

Solution: Figure 2.64 shows an implementation for the 2:4 decoder using four AND gates. Each gate depends on either the true or the complementary form of each input. In general, an N:2^N decoder can be constructed from 2^N N-input AND gates that accept the various combinations of true or complementary inputs. Each output in a decoder represents a single minterm. For example, Y_0 represents the minterm $\overline{A}_1\overline{A}_0$. This fact will be handy when using decoders with other digital building blocks.



A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Figure 2.63 2:4 decoder

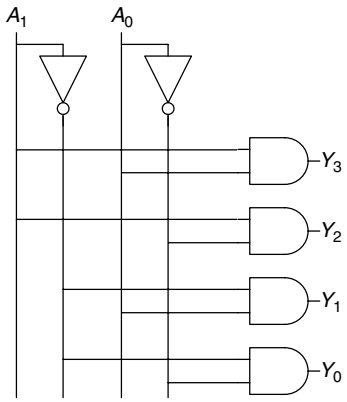


Figure 2.64 2:4 decoder implementation

Decoder Logic

Decoders can be combined with OR gates to build logic functions. Figure 2.65 shows the two-input XNOR function using a 2:4 decoder and a single OR gate. Because each output of a decoder represents a single minterm, the function is built as the OR of all the minterms in the function. In Figure 2.65, $Y = \overline{A}\overline{B} + AB = \overline{A \oplus B}$.

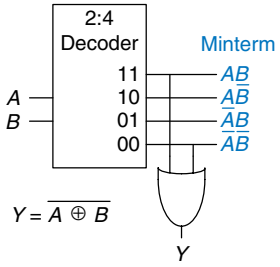


Figure 2.65 Logic function using decoder

When using decoders to build logic, it is easiest to express functions as a truth table or in canonical sum-of-products form. An N-input function with M 1's in the truth table can be built with an $N{:}2^N$ decoder and an M-input OR gate attached to all of the minterms containing 1's in the truth table. This concept will be applied to the building of Read Only Memories (ROMs) in Section 5.5.6.

2.9 TIMING

In previous sections, we have been concerned primarily with whether the circuit works—ideally, using the fewest gates. However, as any seasoned circuit designer will attest, one of the most challenging issues in circuit design is *timing:* making a circuit run fast.

An output takes time to change in response to an input change. Figure 2.66 shows the *delay* between an input change and the subsequent output change for a buffer. The figure is called a *timing diagram;* it portrays the *transient response* of the buffer circuit when an input changes. The transition from LOW to HIGH is called the *rising edge*. Similarly, the transition from HIGH to LOW (not shown in the figure) is called the *falling edge*. The blue arrow indicates that the rising edge of Y is caused by the rising edge of A. We measure delay from the *50% point* of the input signal, A, to the 50% point of the output signal, Y. The 50% point is the point at which the signal is half-way (50%) between its LOW and HIGH values as it transitions.

When designers speak of calculating the *delay* of a circuit, they generally are referring to the worst-case value (the propagation delay), unless it is clear otherwise from the context.

2.9.1 Propagation and Contamination Delay

Combinational logic is characterized by its *propagation delay* and *contamination delay*. The propagation delay, t_{pd}, is the maximum time from when an input changes until the output or outputs reach their final value. The contamination delay, t_{cd}, is the minimum time from when an input changes until any output starts to change its value.

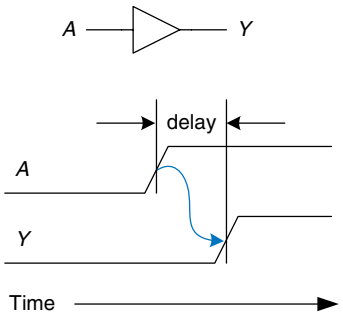


Figure 2.66 Circuit delay

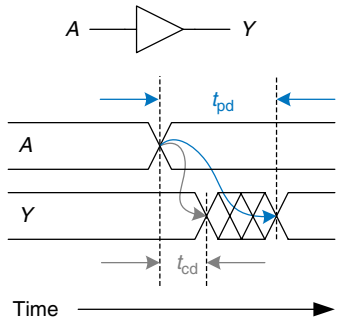


Figure 2.67 Propagation and contamination delay

Figure 2.67 illustrates a buffer's propagation delay and contamination delay in blue and gray, respectively. The figure shows that A is initially either HIGH or LOW and changes to the other state at a particular time; we are interested only in the fact that it changes, not what value it has. In response, Y changes some time later. The arcs indicate that Y may start to change t_{cd} after A transitions and that Y definitely settles to its new value within t_{pd}.

The underlying causes of delay in circuits include the time required to charge the capacitance in a circuit and the speed of light. t_{pd} and t_{cd} may be different for many reasons, including

- different rising and falling delays
- multiple inputs and outputs, some of which are faster than others
- circuits slowing down when hot and speeding up when cold

Circuit delays are ordinarily on the order of picoseconds (1 ps = 10^{-12} seconds) to nanoseconds (1 ns = 10^{-9} seconds). Trillions of picoseconds have elapsed in the time you spent reading this sidebar.

Calculating t_{pd} and t_{cd} requires delving into the lower levels of abstraction beyond the scope of this book. However, manufacturers normally supply data sheets specifying these delays for each gate.

Along with the factors already listed, propagation and contamination delays are also determined by the *path* a signal takes from input to output. Figure 2.68 shows a four-input logic circuit. The *critical path*, shown in blue, is the path from input A or B to output Y. It is the longest, and therefore the slowest, path, because the input travels

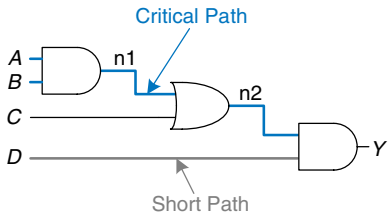


Figure 2.68 Short path and critical path

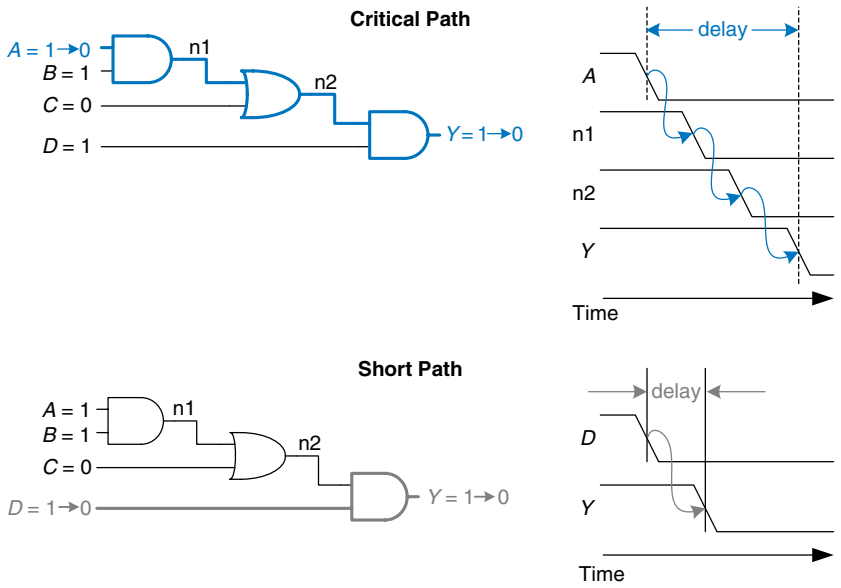


Figure 2.69 **Critical and short path waveforms**

through three gates to the output. This path is critical because it limits the speed at which the circuit operates. The *short path* through the circuit, shown in gray, is from input D to output Y. This is the shortest, and therefore the fastest, path through the circuit, because the input travels through only a single gate to the output.

The propagation delay of a combinational circuit is the sum of the propagation delays through each element on the critical path. The contamination delay is the sum of the contamination delays through each element on the short path. These delays are illustrated in Figure 2.69 and are described by the following equations:

$$t_{pd} = 2t_{pd\_\mathrm{AND}} + t_{pd\_\mathrm{OR}} \tag{2.7}$$

$$t_{cd} = t_{cd\_\mathrm{AND}} \tag{2.8}$$

Although we are ignoring wire delay in this analysis, digital circuits are now so fast that the delay of long wires can be as important as the delay of the gates. The speed of light delay in wires is covered in Appendix A.

Example 2.15 FINDING DELAYS

Ben Bitdiddle needs to find the propagation delay and contamination delay of the circuit shown in Figure 2.70. According to his data book, each gate has a propagation delay of 100 picoseconds (ps) and a contamination delay of 60 ps.

Solution: Ben begins by finding the critical path and the shortest path through the circuit. The critical path, highlighted in blue in Figure 2.71, is from input A

or B through three gates to the output, Y. Hence, t_{pd} is three times the propagation delay of a single gate, or 300 ps.

The shortest path, shown in gray in Figure 2.72, is from input C, D, or E through two gates to the output, Y. There are only two gates in the shortest path, so t_{cd} is 120 ps.

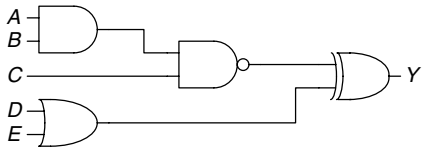


Figure 2.70 Ben's circuit

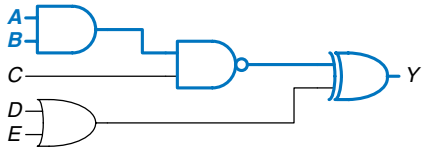


Figure 2.71 Ben's critical path

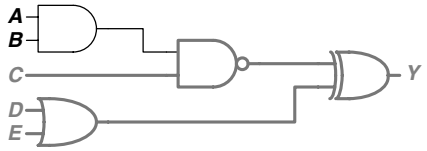


Figure 2.72 Ben's shortest path

Example 2.16 MULTIPLEXER TIMING: CONTROL-CRITICAL VS. DATA-CRITICAL

Compare the worst-case timing of the three four-input multiplexer designs shown in Figure 2.58 in Section 2.8.1. Table 2.7 lists the propagation delays for the components. What is the critical path for each design? Given your timing analysis, why might you choose one design over the other?

Solution: One of the critical paths for each of the three design options is highlighted in blue in Figures 2.73 and 2.74. t_{pd_sy} indicates the propagation delay from input S to output Y; t_{pd_dy} indicates the propagation delay from input D to output Y; t_{pd} is the worst of the two: $\max(t_{pd\_sy}, t_{pd\_dy})$.

For both the two-level logic and tristate implementations in Figure 2.73, the critical path is from one of the control signals, S, to the output, Y: $t_{pd} = t_{pd\_sy}$. These circuits are *control critical*, because the critical path is from the control signals to the output. Any additional delay in the control signals will add directly to the worst-case delay. The delay from D to Y in Figure 2.73(b) is only 50 ps, compared with the delay from S to Y of 125 ps.

Figure 2.74 shows the hierarchical implementation of the 4:1 multiplexer using two stages of 2:1 multiplexers. The critical path is from any of the D inputs to the output. This circuit is *data critical*, because the critical path is from the data input to the output: ($t_{pd} = t_{pd\_dy}$).

If data inputs arrive well before the control inputs, we would prefer the design with the shortest control-to-output delay (the hierarchical design in Figure 2.74). Similarly, if the control inputs arrive well before the data inputs, we would prefer the design with the shortest data-to-output delay (the tristate design in Figure 2.73(b)).

The best choice depends not only on the critical path through the circuit and the input arrival times, but also on the power, cost, and availability of parts.

Table 2.7 Timing specifications for multiplexer circuit elements

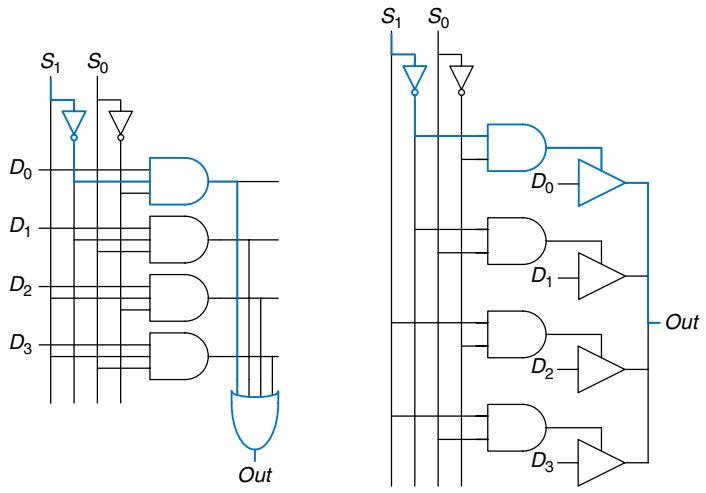
Gate	t_{pd} (ps)
NOT	30
2-input AND	60
3-input AND	80
4-input OR	90
tristate (A to Y)	50
tristate (enable to Y)	35

2.9.2 Glitches

So far we have discussed the case where a single input transition causes a single output transition. However, it is possible that a single input transition can cause *multiple* output transitions. These are called *glitches* or *hazards*. Although glitches usually don't cause problems, it is important to realize that they exist and recognize them when looking at timing diagrams. Figure 2.75 shows a circuit with a glitch and the Karnaugh map of the circuit.

Hazards have another meaning related to microarchitecture in Chapter 7, so we will stick with the term *glitches* for multiple output transitions to avoid confusion.

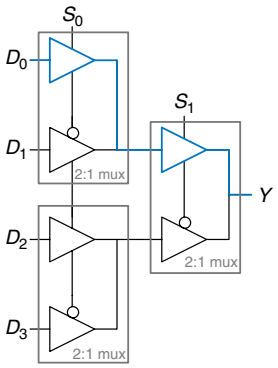
The Boolean equation is correctly minimized, but let's look at what happens when $A = 0$, $C = 1$, and B transitions from 1 to 0. Figure 2.76 (see page 90) illustrates this scenario. The short path (shown in gray) goes through two gates, the AND and OR gates. The critical path (shown in blue) goes through an inverter and two gates, the AND and OR gates.



(a)
$t_{pd\_sy} = t_{pd\_INV} + t_{pd\_AND3} + t_{pd\_OR4}$
$= 30\text{ ps} + 80\text{ ps} + 90\text{ ps}$
$= \mathbf{200\text{ ps}}$
$t_{pd\_dy} = t_{pd\_AND3} + t_{pd\_OR4}$
$= \mathbf{170\text{ ps}}$

(b)
$t_{pd\_sy} = t_{pd\_INV} + t_{pd\_AND2} + t_{pd\_TRI\_SY}$
$= 30\text{ ps} + 60\text{ ps} + 35\text{ ps}$
$= \mathbf{125\text{ ps}}$
$t_{pd\_dy} = t_{pd\_TRI\_AY}$
$= \mathbf{50\text{ ps}}$

Figure 2.73 4:1 multiplexer propagation delays: (a) two-level logic, (b) tristate



$t_{pd\_s0y} = t_{pd\_TRLSY} + t_{pd\_TRI\_AY} = \mathbf{85\text{ ns}}$
$t_{pd\_dy} = 2\ t_{pd\_TRI\_AY} = \mathbf{100\text{ ns}}$

Figure 2.74 4:1 multiplexer propagation delays: hierarchical using 2:1 multiplexers

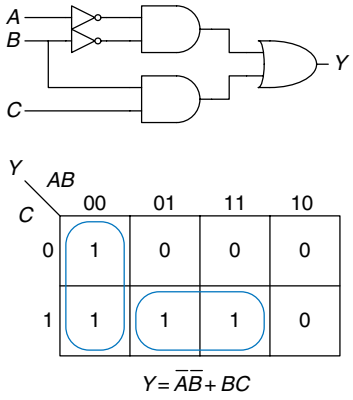
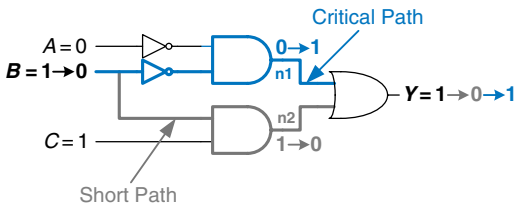


Figure 2.75 Circuit with a glitch

As B transitions from 1 to 0, n2 (on the short path) falls before n1 (on the critical path) can rise. Until n1 rises, the two inputs to the OR gate are 0, and the output Y drops to 0. When n1 eventually rises, Y returns to 1. As shown in the timing diagram of Figure 2.76, Y starts at 1 and ends at 1 but momentarily glitches to 0.



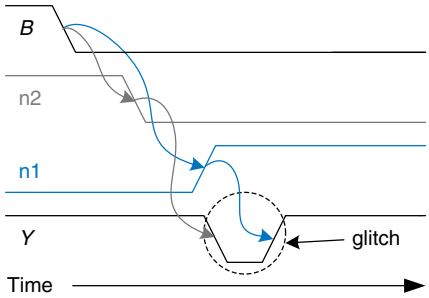


Figure 2.76 Timing of a glitch

As long as we wait for the propagation delay to elapse before we depend on the output, glitches are not a problem, because the output eventually settles to the right answer.

If we choose to, we can avoid this glitch by adding another gate to the implementation. This is easiest to understand in terms of the K-map. Figure 2.77 shows how an input transition on B from $ABC = 001$ to $ABC = 011$ moves from one prime implicant circle to another. The transition across the boundary of two prime implicants in the K-map indicates a possible glitch.

As we saw from the timing diagram in Figure 2.76, if the circuitry implementing one of the prime implicants turns *off* before the circuitry of the other prime implicant can turn *on*, there is a glitch. To fix this, we add another circle that *covers* that prime implicant boundary, as shown in Figure 2.78. You might recognize this as the consensus theorem, where the added term, $\overline{A}\,\overline{C}$, is the consensus or redundant term.

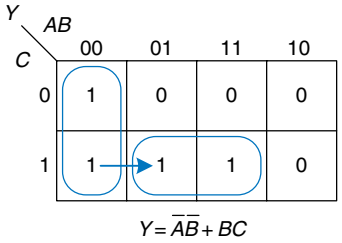


Figure 2.77 Input change crosses implicant boundary

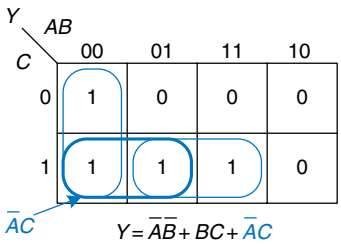


Figure 2.78 K-map without glitch

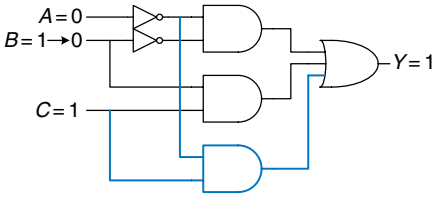


Figure 2.79 Circuit without glitch

Figure 2.79 shows the glitch-proof circuit. The added AND gate is highlighted in blue. Now a transition on B when $A = 0$ and $C = 1$ does not cause a glitch on the output, because the blue AND gate outputs 1 throughout the transition.

In general, a glitch can occur when a change in a single variable crosses the boundary between two prime implicants in a K-map. We can eliminate the glitch by adding redundant implicants to the K-map to cover these boundaries. This of course comes at the cost of extra hardware.

However, simultaneous transitions on multiple variables can also cause glitches. These glitches cannot be fixed by adding hardware. Because the vast majority of interesting systems have simultaneous (or near-simultaneous) transitions on multiple variables, glitches are a fact of life in most circuits. Although we have shown how to eliminate one kind of glitch, the point of discussing glitches is not to eliminate them but to be aware that they exist. This is especially important when looking at timing diagrams on a simulator or oscilloscope.

2.10 SUMMARY

A digital circuit is a module with discrete-valued inputs and outputs and a specification describing the function and timing of the module. This chapter has focused on combinational circuits, circuits whose outputs depend only on the current values of the inputs.

The function of a combinational circuit can be given by a truth table or a Boolean equation. The Boolean equation for any truth table can be obtained systematically using sum-of-products or product-of-sums form. In sum-of-products form, the function is written as the sum (OR) of one or more implicants. Implicants are the product (AND) of literals. Literals are the true or complementary forms of the input variables.

Boolean equations can be simplified using the rules of Boolean algebra. In particular, they can be simplified into minimal sum-of-products form by combining implicants that differ only in the true and complementary forms of one of the literals: $PA + P\overline{A} = P$. Karnaugh maps are a visual tool for minimizing functions of up to four variables. With practice, designers can usually simplify functions of a few variables by inspection. Computer-aided design tools are used for more complicated functions; such methods and tools are discussed in Chapter 4.

Logic gates are connected to create combinational circuits that perform the desired function. Any function in sum-of-products form can be built using two-level logic with the literals as inputs: NOT gates form the complementary literals, AND gates form the products, and OR gates form the sum. Depending on the function and the building blocks available, multilevel logic implementations with various types of gates may be more efficient. For example, CMOS circuits favor NAND and NOR gates because these gates can be built directly from CMOS transistors without requiring extra NOT gates. When using NAND and NOR gates, bubble pushing is helpful to keep track of the inversions.

Logic gates are combined to produce larger circuits such as multiplexers, decoders, and priority circuits. A multiplexer chooses one of the data inputs based on the select input. A decoder sets one of the outputs HIGH according to the input. A priority circuit produces an output indicating the highest priority input. These circuits are all examples of combinational building blocks. Chapter 5 will introduce more building blocks, including other arithmetic circuits. These building blocks will be used extensively to build a microprocessor in Chapter 7.

The timing specification of a combinational circuit consists of the propagation and contamination delays through the circuit. These indicate the longest and shortest times between an input change and the consequent output change. Calculating the propagation delay of a circuit involves identifying the critical path through the circuit, then adding up the propagation delays of each element along that path. There are many different ways to implement complicated combinational circuits; these ways offer trade-offs between speed and cost.

The next chapter will move to sequential circuits, whose outputs depend on previous as well as current values of the inputs. In other words, sequential circuits have *memory* of the past.

Exercises

Exercise 2.1 Write a Boolean equation in sum-of-products canonical form for each of the truth tables in Figure 2.80.

(a)

A	B	Y
0	0	1
0	1	0
1	0	1
1	1	1

(b)

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

(c)

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

(d)

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

(e)

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Figure 2.80 Truth tables

Exercise 2.2 Write a Boolean equation in product-of-sums canonical form for the truth tables in Figure 2.80.

Exercise 2.3 Minimize each of the Boolean equations from Exercise 2.1.

Exercise 2.4 Sketch a reasonably simple combinational circuit implementing each of the functions from Exercise 2.3. Reasonably simple means that you are not wasteful of gates, but you don't waste vast amounts of time checking every possible implementation of the circuit either.

Exercise 2.5 Repeat Exercise 2.4 using only NOT gates and AND and OR gates.

Exercise 2.6 Repeat Exercise 2.4 using only NOT gates and NAND and NOR gates.

Exercise 2.7 Simplify the following Boolean equations using Boolean theorems. Check for correctness using a truth table or K-map.

(a) $Y = AC + \overline{A}\overline{B}C$

(b) $Y = \overline{A}\overline{B} + \overline{A}B\overline{C} + \overline{(A + \overline{C})}$

(c) $Y = \overline{A}\overline{B}\overline{C}\overline{D} + A\overline{B}\overline{C} + A\overline{B}C\overline{D} + ABD + \overline{A}\overline{B}C\overline{D} + B\overline{C}D + \overline{A}$

Exercise 2.8 Sketch a reasonably simple combinational circuit implementing each of the functions from Exercise 2.7.

Exercise 2.9 Simplify each of the following Boolean equations. Sketch a reasonably simple combinational circuit implementing the simplified equation.

(a) $Y = BC + \overline{A}\,\overline{B}\,\overline{C} + B\overline{C}$

(b) $Y = \overline{A + \overline{A}B + \overline{A}\,\overline{B}} + \overline{A + \overline{B}}$

(c) $Y = ABC + ABD + ABE + ACD + ACE + (\overline{A + D + E}) + \overline{B}\,\overline{C}D + \overline{B}\,\overline{C}E + \overline{B}\,\overline{D}\,\overline{E} + \overline{C}\,\overline{D}\,\overline{E}$

Exercise 2.10 Give an example of a truth table requiring between 3 billion and 5 billion rows that can be constructed using fewer than 40 (but at least 1) two-input gates.

Exercise 2.11 Give an example of a circuit with a cyclic path that is nevertheless combinational.

Exercise 2.12 Alyssa P. Hacker says that any Boolean function can be written in minimal sum-of-products form as the sum of all of the prime implicants of the function. Ben Bitdiddle says that there are some functions whose minimal equation does not involve all of the prime implicants. Explain why Alyssa is right or provide a counterexample demonstrating Ben's point.

Exercise 2.13 Prove that the following theorems are true using perfect induction. You need not prove their duals.

(a) The idempotency theorem (T3)

(b) The distributivity theorem (T8)

(c) The combining theorem (T10)

Exercise 2.14 Prove De Morgan's Theorem (T12) for three variables, B_2, B_1, B_0, using perfect induction.

Exercise 2.15 Write Boolean equations for the circuit in Figure 2.81. You need not minimize the equations.

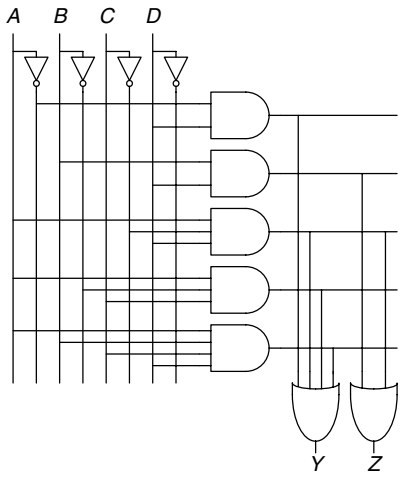


Figure 2.81 Circuit schematic

Exercise 2.16 Minimize the Boolean equations from Exercise 2.15 and sketch an improved circuit with the same function.

Exercise 2.17 Using De Morgan equivalent gates and bubble pushing methods, redraw the circuit in Figure 2.82 so that you can find the Boolean equation by inspection. Write the Boolean equation.

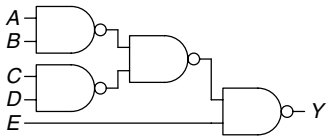


Figure 2.82 Circuit schematic

Exercise 2.18 Repeat Exercise 2.17 for the circuit in Figure 2.83.

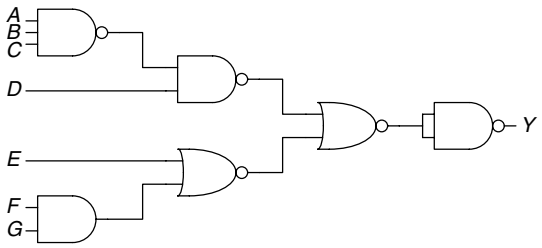


Figure 2.83 Circuit schematic

Exercise 2.19 Find a minimal Boolean equation for the function in Figure 2.84. Remember to take advantage of the don't care entries.

A	B	C	D	Y
0	0	0	0	X
0	0	0	1	X
0	0	1	0	X
0	0	1	1	0
0	1	0	0	0
0	1	0	1	X
0	1	1	0	0
0	1	1	1	X
1	0	0	0	1
1	0	0	1	0
1	0	1	0	X
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	X
1	1	1	1	1

Figure 2.84 Truth table

Exercise 2.20 Sketch a circuit for the function from Exercise 2.19.

Exercise 2.21 Does your circuit from Exercise 2.20 have any potential glitches when one of the inputs changes? If not, explain why not. If so, show how to modify the circuit to eliminate the glitches.

Exercise 2.22 Ben Bitdiddle will enjoy his picnic on sunny days that have no ants. He will also enjoy his picnic any day he sees a hummingbird, as well as on days where there are ants and ladybugs. Write a Boolean equation for his enjoyment (E) in terms of sun (S), ants (A), hummingbirds (H), and ladybugs (L).

Exercise 2.23 Complete the design of the seven-segment decoder segments S_c through S_g (see Example 2.10):

(a) Derive Boolean equations for the outputs S_c through S_g assuming that inputs greater than 9 must produce blank (0) outputs.

(b) Derive Boolean equations for the outputs S_c through S_g assuming that inputs greater than 9 are don't cares.

(c) Sketch a reasonably simple gate-level implementation of part (b). Multiple outputs can share gates where appropriate.

Exercise 2.24 A circuit has four inputs and two outputs. The inputs, $A_{3:0}$, represent a number from 0 to 15. Output P should be TRUE if the number is prime (0 and 1 are not prime, but 2, 3, 5, and so on, are prime). Output D should be TRUE if the number is divisible by 3. Give simplified Boolean equations for each output and sketch a circuit.

Exercise 2.25 A *priority encoder* has 2^N inputs. It produces an N-bit binary output indicating the most significant bit of the input that is TRUE, or 0 if none of the inputs are TRUE. It also produces an output *NONE* that is TRUE if none of the input bits are TRUE. Design an eight-input priority encoder with inputs $A_{7:0}$ and outputs $Y_{2:0}$ and *NONE*. For example, if the input is 00100000, the output Y should be 101 and *NONE* should be 0. Give a simplified Boolean equation for each output, and sketch a schematic.

Exercise 2.26 Design a modified priority encoder (see Exercise 2.25) that receives an 8-bit input, $A_{7:0}$, and produces two 3-bit outputs, $Y_{2:0}$ and $Z_{2:0}$. Y indicates the most significant bit of the input that is TRUE. Z indicates the second most significant bit of the input that is TRUE. Y should be 0 if none of the inputs are TRUE. Z should be 0 if no more than one of the inputs is TRUE. Give a simplified Boolean equation for each output, and sketch a schematic.

Exercise 2.27 An M-bit *thermometer code* for the number k consists of k 1's in the least significant bit positions and $M - k$ 0's in all the more significant bit positions. A *binary-to-thermometer code converter* has N inputs and 2^N-1 outputs. It produces a 2^N-1 bit thermometer code for the number specified by the input. For example, if the input is 110, the output should be 0111111. Design a 3:7 binary-to-thermometer code converter. Give a simplified Boolean equation for each output, and sketch a schematic.

Exercise 2.28 Write a minimized Boolean equation for the function performed by the circuit in Figure 2.85.

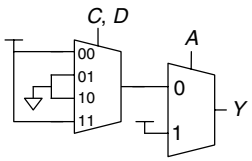


Figure 2.85 Multiplexer circuit

Exercise 2.29 Write a minimized Boolean equation for the function performed by the circuit in Figure 2.86.

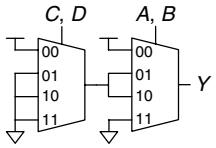


Figure 2.86 Multiplexer circuit

Exercise 2.30 Implement the function from Figure 2.80(b) using

(a) an 8:1 multiplexer

(b) a 4:1 multiplexer and one inverter

(c) a 2:1 multiplexer and two other logic gates

Exercise 2.31 Implement the function from Exercise 2.9(a) using

(a) an 8:1 multiplexer

(b) a 4:1 multiplexer and no other gates

(c) a 2:1 multiplexer, one OR gate, and an inverter

Exercise 2.32 Determine the propagation delay and contamination delay of the circuit in Figure 2.83. Use the gate delays given in Table 2.8.

Table 2.8 Gate delays for Exercises 2.32–2.35

Gate	t_{pd} (ps)	t_{cd} (ps)
NOT	15	10
2-input NAND	20	15
3-input NAND	30	25
2-input NOR	30	25
3-input NOR	45	35
2-input AND	30	25
3-input AND	40	30
2-input OR	40	30
3-input OR	55	45
2-input XOR	60	40

Exercise 2.33 Sketch a schematic for a fast 3:8 decoder. Suppose gate delays are given in Table 2.8 (and only the gates in that table are available). Design your decoder to have the shortest possible critical path, and indicate what that path is. What are its propagation delay and contamination delay?

Exercise 2.34 Redesign the circuit from Exercise 2.24 to be as fast as possible. Use only the gates from Table 2.8. Sketch the new circuit and indicate the critical path. What are its propagation delay and contamination delay?

Exercise 2.35 Redesign the priority encoder from Exercise 2.25 to be as fast as possible. You may use any of the gates from Table 2.8. Sketch the new circuit and indicate the critical path. What are its propagation delay and contamination delay?

Exercise 2.36 Design an 8:1 multiplexer with the shortest possible delay from the data inputs to the output. You may use any of the gates from Table 2.7 on page 88. Sketch a schematic. Using the gate delays from the table, determine this delay.

Interview Questions

The following exercises present questions that have been asked at interviews for digital design jobs.

Question 2.1 Sketch a schematic for the two-input XOR function using only NAND gates. How few can you use?

Question 2.2 Design a circuit that will tell whether a given month has 31 days in it. The month is specified by a 4-bit input, $A_{3:0}$. For example, if the inputs are 0001, the month is January, and if the inputs are 1100, the month is December. The circuit output, Y, should be HIGH only when the month specified by the inputs has 31 days in it. Write the simplified equation, and draw the circuit diagram using a minimum number of gates. (Hint: Remember to take advantage of don't cares.)

Question 2.3 What is a tristate buffer? How and why is it used?

Question 2.4 A gate or set of gates is universal if it can be used to construct any Boolean function. For example, the set {AND, OR, NOT} is universal.

(a) Is an AND gate by itself universal? Why or why not?

(b) Is the set {OR, NOT} universal? Why or why not?

(c) Is a NAND gate by itself universal? Why or why not?

Question 2.5 Explain why a circuit's contamination delay might be less than (instead of equal to) its propagation delay.

Sequential Logic Design

3

3.1 INTRODUCTION

In the last chapter, we showed how to analyze and design combinational logic. The output of combinational logic depends only on current input values. Given a specification in the form of a truth table or Boolean equation, we can create an optimized circuit to meet the specification.

In this chapter, we will analyze and design *sequential* logic. The outputs of sequential logic depend on both current and prior input values. Hence, sequential logic has memory. Sequential logic might explicitly remember certain previous inputs, or it might distill the prior inputs into a smaller amount of information called the *state* of the system. The state of a digital sequential circuit is a set of bits called *state variables* that contain all the information about the past necessary to explain the future behavior of the circuit.

The chapter begins by studying latches and flip-flops, which are simple sequential circuits that store one bit of state. In general, sequential circuits are complicated to analyze. To simplify design, we discipline ourselves to build only synchronous sequential circuits consisting of combinational logic and banks of flip-flops containing the state of the circuit. The chapter describes finite state machines, which are an easy way to design sequential circuits. Finally, we analyze the speed of sequential circuits and discuss parallelism as a way to increase clock speed.

3.2 LATCHES AND FLIP-FLOPS

The fundamental building block of memory is a *bistable* element, an element with two stable states. Figure 3.1(a) shows a simple bistable element consisting of a pair of inverters connected in a loop. Figure 3.1(b) shows the same circuit redrawn to emphasize the symmetry. The inverters are *cross-coupled,* meaning that the input of I1 is the output of I2 and vice versa. The circuit has no inputs, but it does have two outputs,

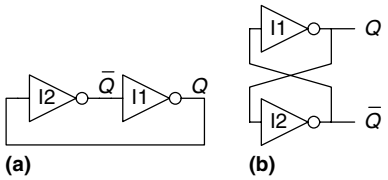


Figure 3.1 Cross-coupled inverter pair

Just as Y is commonly used for the output of combinational logic, Q is commonly used for the output of sequential logic.

Q and $\overline{Q}$. Analyzing this circuit is different from analyzing a combinational circuit because it is cyclic: Q depends on $\overline{Q}$, and $\overline{Q}$ depends on Q.

Consider the two cases, Q is 0 or Q is 1. Working through the consequences of each case, we have:

- *Case I:* $Q = 0$
  As shown in Figure 3.2(a), I2 receives a FALSE input, Q, so it produces a TRUE output on $\overline{Q}$. I1 receives a TRUE input, $\overline{Q}$, so it produces a FALSE output on Q. This is consistent with the original assumption that $Q = 0$, so the case is said to be *stable*.
- *Case II:* $Q = 1$
  As shown in Figure 3.2(b), I2 receives a TRUE input and produces a FALSE output on $\overline{Q}$. I1 receives a FALSE input and produces a TRUE output on Q. This is again stable.

Because the cross-coupled inverters have two stable states, $Q = 0$ and $Q = 1$, the circuit is said to be bistable. A subtle point is that the circuit has a third possible state with both outputs approximately halfway between 0 and 1. This is called a *metastable* state and will be discussed in Section 3.5.4.

An element with N stable states conveys $\log_2 N$ bits of information, so a bistable element stores one bit. The state of the cross-coupled inverters is contained in one binary state variable, Q. The value of Q tells us everything about the past that is necessary to explain the future behavior of the circuit. Specifically, if $Q = 0$, it will remain 0 forever, and if $Q = 1$, it will remain 1 forever. The circuit does have another node, $\overline{Q}$, but $\overline{Q}$ does not contain any additional information because if Q is known, $\overline{Q}$ is also known. On the other hand, $\overline{Q}$ is also an acceptable choice for the state variable.

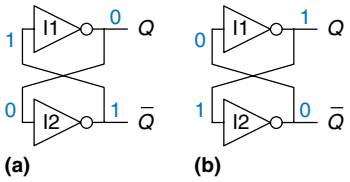


Figure 3.2 Bistable operation of cross-coupled inverters

When power is first applied to a sequential circuit, the initial state is unknown and usually unpredictable. It may differ each time the circuit is turned on.

Although the cross-coupled inverters can store a bit of information, they are not practical because the user has no inputs to control the state. However, other bistable elements, such as *latches* and *flip-flops*, provide inputs to control the value of the state variable. The remainder of this section considers these circuits.

3.2.1 SR Latch

One of the simplest sequential circuits is the *SR latch*, which is composed of two cross-coupled NOR gates, as shown in Figure 3.3. The latch has two inputs, S and R, and two outputs, Q and $\overline{Q}$. The SR latch is similar to the cross-coupled inverters, but its state can be controlled through the S and R inputs, which *set* and *reset* the output Q.

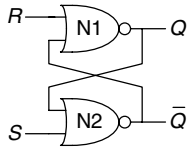


Figure 3.3 SR latch schematic

A good way to understand an unfamiliar circuit is to work out its truth table, so that is where we begin. Recall that a NOR gate produces a FALSE output when either input is TRUE. Consider the four possible combinations of R and S.

- *Case I:* $R = 1, S = 0$
  N1 sees at least one TRUE input, R, so it produces a FALSE output on Q. N2 sees both Q and S FALSE, so it produces a TRUE output on $\overline{Q}$.

- *Case II:* $R = 0, S = 1$
  N1 receives inputs of 0 and $\overline{Q}$. Because we don't yet know $\overline{Q}$, we can't determine the output Q. N2 receives at least one TRUE input, S, so it produces a FALSE output on $\overline{Q}$. Now we can revisit N1, knowing that both inputs are FALSE, so the output Q is TRUE.

- *Case III:* $R = 1, S = 1$
  N1 and N2 both see at least one TRUE input (R or S), so each produces a FALSE output. Hence Q and $\overline{Q}$ are both FALSE.

- *Case IV:* $R = 0, S = 0$
  N1 receives inputs of 0 and $\overline{Q}$. Because we don't yet know $\overline{Q}$, we can't determine the output. N2 receives inputs of 0 and Q. Because we don't yet know Q, we can't determine the output. Now we are stuck. This is reminiscent of the cross-coupled inverters. But we know that Q must either be 0 or 1. So we can solve the problem by checking what happens in each of these subcases.

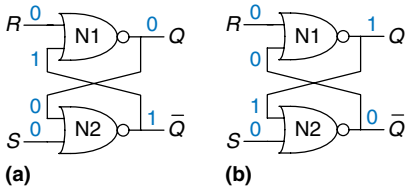


Figure 3.4 Bistable states of SR latch

- *Case IVa:* $Q = 0$
  Because S and Q are FALSE, N2 produces a TRUE output on $\overline{Q}$, as shown in Figure 3.4(a). Now N1 receives one TRUE input, $\overline{Q}$, so its output, Q, is FALSE, just as we had assumed.
- *Case IVb:* $Q = 1$
  Because Q is TRUE, N2 produces a FALSE output on $\overline{Q}$, as shown in Figure 3.4(b). Now N1 receives two FALSE inputs, R and $\overline{Q}$, so its output, Q, is TRUE, just as we had assumed.

Putting this all together, suppose Q has some known prior value, which we will call Q_{prev}, before we enter Case IV. Q_{prev} is either 0 or 1, and represents the state of the system. When R and S are 0, Q will remember this old value, Q_{prev}, and $\overline{Q}$ will be its complement, $\overline{Q}_{prev}$. This circuit has memory.

Case	S	R	Q	$\overline{Q}$
IV	0	0	Q_{prev}	$\overline{Q}_{prev}$
I	0	1	0	1
II	1	0	1	0
III	1	1	0	0

Figure 3.5 SR latch truth table

The truth table in Figure 3.5 summarizes these four cases. The inputs S and R stand for *Set* and *Reset*. To *set* a bit means to make it TRUE. To *reset* a bit means to make it FALSE. The outputs, Q and $\overline{Q}$, are normally complementary. When R is asserted, Q is reset to 0 and $\overline{Q}$ does the opposite. When S is asserted, Q is set to 1 and $\overline{Q}$ does the opposite. When neither input is asserted, Q remembers its old value, Q_{prev}. Asserting both S and R simultaneously doesn't make much sense because it means the latch should be set and reset at the same time, which is impossible. The poor confused circuit responds by making both outputs 0.

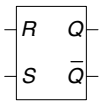


Figure 3.6 SR latch symbol

The SR latch is represented by the symbol in Figure 3.6. Using the symbol is an application of abstraction and modularity. There are various ways to build an SR latch, such as using different logic gates or transistors. Nevertheless, any circuit element with the relationship specified by the truth table in Figure 3.5 and the symbol in Figure 3.6 is called an SR latch.

Like the cross-coupled inverters, the SR latch is a bistable element with one bit of state stored in Q. However, the state can be controlled through the S and R inputs. When R is asserted, the state is reset to 0. When S is asserted, the state is set to 1. When neither is asserted, the state retains its old value. Notice that the entire history of inputs can be

accounted for by the single state variable Q. No matter what pattern of setting and resetting occurred in the past, all that is needed to predict the future behavior of the SR latch is whether it was most recently set or reset.

3.2.2 D Latch

The SR latch is awkward because it behaves strangely when both S and R are simultaneously asserted. Moreover, the S and R inputs conflate the issues of *what* and *when*. Asserting one of the inputs determines not only *what* the state should be but also *when* it should change. Designing circuits becomes easier when these questions of what and when are separated. The D latch in Figure 3.7(a) solves these problems. It has two inputs. The *data* input, D, controls what the next state should be. The *clock* input, CLK, controls when the state should change.

Again, we analyze the latch by writing the truth table, given in Figure 3.7(b). For convenience, we first consider the internal nodes $\overline{D}$, S, and R. If $CLK = 0$, both S and R are FALSE, regardless of the value of D. If $CLK = 1$, one AND gate will produce TRUE and the other FALSE, depending on the value of D. Given S and R, Q and $\overline{Q}$ are determined using Figure 3.5. Observe that when $CLK = 0$, Q remembers its old value, Q_{prev}. When $CLK = 1$, $Q = D$. In all cases, $\overline{Q}$ is the complement of Q, as would seem logical. The D latch avoids the strange case of simultaneously asserted R and S inputs.

Putting it all together, we see that the clock controls when data flows through the latch. When $CLK = 1$, the latch is *transparent*. The data at D flows through to Q as if the latch were just a buffer. When $CLK = 0$, the latch is *opaque*. It blocks the new data from flowing through to Q, and Q retains the old value. Hence, the D latch is sometimes called a *transparent latch* or a *level-sensitive* latch. The D latch symbol is given in Figure 3.7(c).

The D latch updates its state continuously while $CLK = 1$. We shall see later in this chapter that it is useful to update the state only at a specific instant in time. The D flip-flop described in the next section does just that.

Some people call a latch open or closed rather than transparent or opaque. However, we think those terms are ambiguous—does *open* mean transparent like an open door, or opaque, like an open circuit?

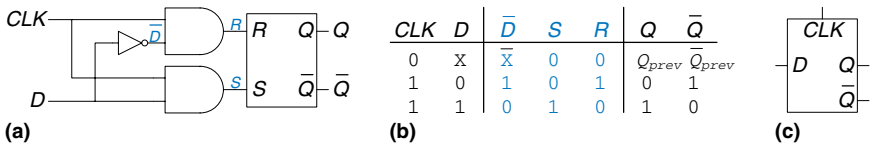


Figure 3.7 D latch: (a) schematic, (b) truth table, (c) symbol

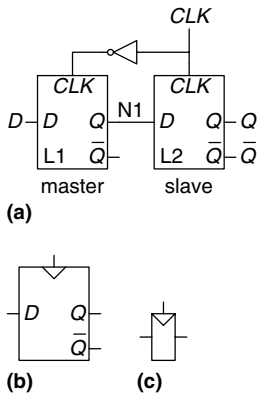


Figure 3.8 D flip-flop: (a) schematic, (b) symbol, (c) condensed symbol

The precise distinction between *flip-flops* and *latches* is somewhat muddled and has evolved over time. In common industry usage, a flip-flop is *edge-triggered*. In other words, it is a bistable element with a *clock* input. The state of the flip-flop changes only in response to a clock edge, such as when the clock rises from 0 to 1. Bistable elements without an edge-triggered clock are commonly called latches.

The term flip-flop or latch by itself usually refers to a *D flip-flop* or *D latch*, respectively, because these are the types most commonly used in practice.

3.2.3 D Flip-Flop

A *D flip-flop* can be built from two back-to-back D latches controlled by complementary clocks, as shown in Figure 3.8(a). The first latch, L1, is called the *master*. The second latch, L2, is called the *slave*. The node between them is named N1. A symbol for the D flip-flop is given in Figure 3.8(b). When the $\overline{Q}$ output is not needed, the symbol is often condensed as in Figure 3.8(c).

When $CLK = 0$, the master latch is transparent and the slave is opaque. Therefore, whatever value was at D propagates through to N1. When $CLK = 1$, the master goes opaque and the slave becomes transparent. The value at N1 propagates through to Q, but N1 is cut off from D. Hence, whatever value was at D immediately before the clock rises from 0 to 1 gets copied to Q immediately after the clock rises. At all other times, Q retains its old value, because there is always an opaque latch blocking the path between D and Q.

In other words, *a D flip-flop copies D to Q on the rising edge of the clock, and remembers its state at all other times*. Reread this definition until you have it memorized; one of the most common problems for beginning digital designers is to forget what a flip-flop does. The rising edge of the clock is often just called the *clock edge* for brevity. The D input specifies what the new state will be. The clock edge indicates when the state should be updated.

A D flip-flop is also known as a *master-slave flip-flop*, an *edge-triggered flip-flop*, or a *positive edge-triggered flip-flop*. The triangle in the symbols denotes an edge-triggered clock input. The $\overline{Q}$ output is often omitted when it is not needed.

Example 3.1 FLIP-FLOP TRANSISTOR COUNT

How many transistors are needed to build the D flip-flop described in this section?

Solution: A NAND or NOR gate uses four transistors. A NOT gate uses two transistors. An AND gate is built from a NAND and a NOT, so it uses six transistors. The SR latch uses two NOR gates, or eight transistors. The D latch uses an SR latch, two AND gates, and a NOT gate, or 22 transistors. The D flip-flop uses two D latches and a NOT gate, or 46 transistors. Section 3.2.7 describes a more efficient CMOS implementation using transmission gates.

3.2.4 Register

An N-bit register is a bank of N flip-flops that share a common CLK input, so that all bits of the register are updated at the same time.

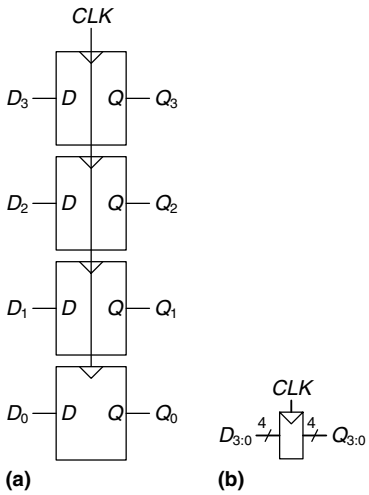


Figure 3.9 A 4-bit register: (a) schematic and (b) symbol

Registers are the key building block of most sequential circuits. Figure 3.9 shows the schematic and symbol for a four-bit register with inputs $D_{3:0}$ and outputs $Q_{3:0}$. $D_{3:0}$ and $Q_{3:0}$ are both 4-bit busses.

3.2.5 Enabled Flip-Flop

An *enabled flip-flop* adds another input called *EN* or ENABLE to determine whether data is loaded on the clock edge. When *EN* is TRUE, the enabled flip-flop behaves like an ordinary D flip-flop. When *EN* is FALSE, the enabled flip-flop ignores the clock and retains its state. Enabled flip-flops are useful when we wish to load a new value into a flip-flop only some of the time, rather than on every clock edge.

Figure 3.10 shows two ways to construct an enabled flip-flop from a D flip-flop and an extra gate. In Figure 3.10(a), an input multiplexer chooses whether to pass the value at *D*, if *EN* is TRUE, or to recycle the old state from *Q*, if *EN* is FALSE. In Figure 3.10(b), the clock is *gated*.

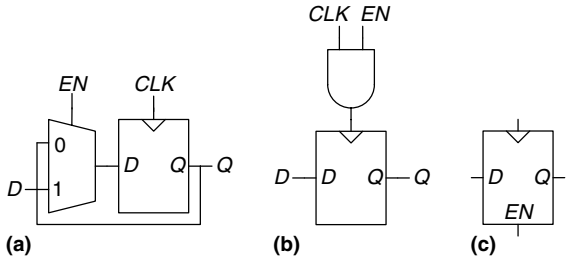


Figure 3.10 Enabled flip-flop: (a, b) schematics, (c) symbol

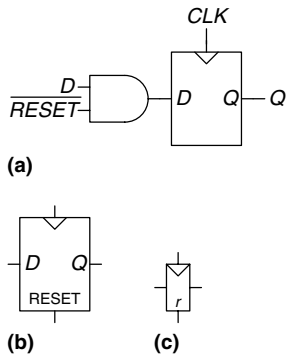


Figure 3.11 **Synchronously resettable flip-flop: (a) schematic, (b, c) symbols**

If *EN* is TRUE, the *CLK* input to the flip-flop toggles normally. If *EN* is FALSE, the *CLK* input is also FALSE and the flip-flop retains its old value. Notice that *EN* must not change while *CLK* = 1, lest the flip-flop see a clock *glitch* (switch at an incorrect time). Generally, performing logic on the clock is a bad idea. Clock gating delays the clock and can cause timing errors, as we will see in Section 3.5.3, so do it only if you are sure you know what you are doing. The symbol for an enabled flip-flop is given in Figure 3.10(c).

3.2.6 Resettable Flip-Flop

A *resettable flip-flop* adds another input called RESET. When RESET is FALSE, the resettable flip-flop behaves like an ordinary D flip-flop. When RESET is TRUE, the resettable flip-flop ignores *D* and resets the output to 0. Resettable flip-flops are useful when we want to force a known state (i.e., 0) into all the flip-flops in a system when we first turn it on.

Such flip-flops may be *synchronously* or *asynchronously resettable*. Synchronously resettable flip-flops reset themselves only on the rising edge of *CLK*. Asynchronously resettable flip-flops reset themselves as soon as RESET becomes TRUE, independent of *CLK*.

Figure 3.11(a) shows how to construct a synchronously resettable flip-flop from an ordinary D flip-flop and an AND gate. When $\overline{\text{RESET}}$ is FALSE, the AND gate forces a 0 into the input of the flip-flop. When $\overline{\text{RESET}}$ is TRUE, the AND gate passes *D* to the flip-flop. In this example, $\overline{\text{RESET}}$ is an *active low* signal, meaning that the reset signal performs its function when it is 0, not 1. By adding an inverter, the circuit could have accepted an active high RESET signal instead. Figures 3.11(b) and 3.11(c) show symbols for the resettable flip-flop with active high RESET.

Asynchronously resettable flip-flops require modifying the internal structure of the flip-flop and are left to you to design in Exercise 3.10; however, they are frequently available to the designer as a standard component.

As you might imagine, settable flip-flops are also occasionally used. They load a 1 into the flip-flop when SET is asserted, and they too come in synchronous and asynchronous flavors. Resettable and settable flip-flops may also have an enable input and may be grouped into *N*-bit registers.

3.2.7 Transistor-Level Latch and Flip-Flop Designs*

Example 3.1 showed that latches and flip-flops require a large number of transistors when built from logic gates. But the fundamental role of a

latch is to be transparent or opaque, much like a switch. Recall from Section 1.7.7 that a transmission gate is an efficient way to build a CMOS switch, so we might expect that we could take advantage of transmission gates to reduce the transistor count.

A compact D latch can be constructed from a single transmission gate, as shown in Figure 3.12(a). When $CLK = 1$ and $\overline{CLK} = 0$, the transmission gate is ON, so D flows to Q and the latch is transparent. When $CLK = 0$ and $\overline{CLK} = 1$, the transmission gate is OFF, so Q is isolated from D and the latch is opaque. This latch suffers from two major limitations:

- *Floating output node:* When the latch is opaque, Q is not held at its value by any gates. Thus Q is called a *floating* or *dynamic* node. After some time, noise and charge leakage may disturb the value of Q.
- *No buffers:* The lack of buffers has caused malfunctions on several commercial chips. A spike of noise that pulls D to a negative voltage can turn on the nMOS transistor, making the latch transparent, even when $CLK = 0$. Likewise, a spike on D above V_{DD} can turn on the pMOS transistor even when $CLK = 0$. And the transmission gate is symmetric, so it could be driven backward with noise on Q affecting the input D. The general rule is that neither the input of a transmission gate nor the state node of a sequential circuit should ever be exposed to the outside world, where noise is likely.

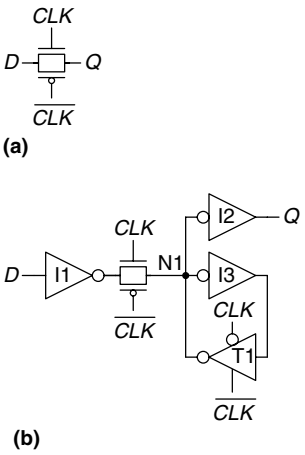


Figure 3.12 D latch schematic

Figure 3.12(b) shows a more robust 12-transistor D latch used on modern commercial chips. It is still built around a clocked transmission gate, but it adds inverters I1 and I2 to buffer the input and output. The state of the latch is held on node N1. Inverter I3 and the tristate buffer, T1, provide feedback to turn N1 into a *static node*. If a small amount of noise occurs on N1 while $CLK = 0$, T1 will drive N1 back to a valid logic value.

Figure 3.13 shows a D flip-flop constructed from two static latches controlled by $\overline{CLK}$ and CLK. Some redundant internal inverters have been removed, so the flip-flop requires only 20 transistors.

This circuit assumes CLK and $\overline{CLK}$ are both available. If not, two more transistors are needed for a CLK inverter.

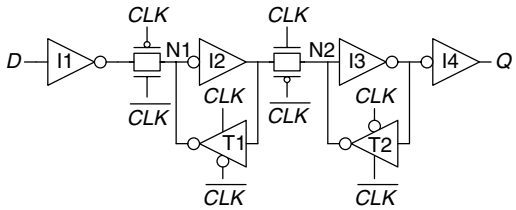


Figure 3.13 D flip-flop schematic

3.2.8 Putting It All Together

Latches and flip-flops are the fundamental building blocks of sequential circuits. Remember that a D latch is level-sensitive, whereas a D flip-flop is edge-triggered. The D latch is transparent when $CLK = 1$, allowing the input D to flow through to the output Q. The D flip-flop copies D to Q on the rising edge of CLK. At all other times, latches and flip-flops retain their old state. A register is a bank of several D flip-flops that share a common CLK signal.

Example 3.2 FLIP-FLOP AND LATCH COMPARISON

Ben Bitdiddle applies the D and CLK inputs shown in Figure 3.14 to a D latch and a D flip-flop. Help him determine the output, Q, of each device.

Solution: Figure 3.15 shows the output waveforms, assuming a small delay for Q to respond to input changes. The arrows indicate the cause of an output change. The initial value of Q is unknown and could be 0 or 1, as indicated by the pair of horizontal lines. First consider the latch. On the first rising edge of CLK, $D = 0$, so Q definitely becomes 0. Each time D changes while $CLK = 1$, Q also follows. When D changes while $CLK = 0$, it is ignored. Now consider the flip-flop. On each rising edge of CLK, D is copied to Q. At all other times, Q retains its state.

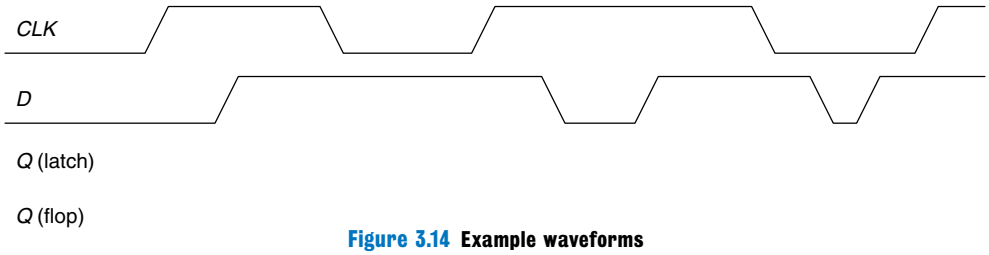


Figure 3.14 Example waveforms

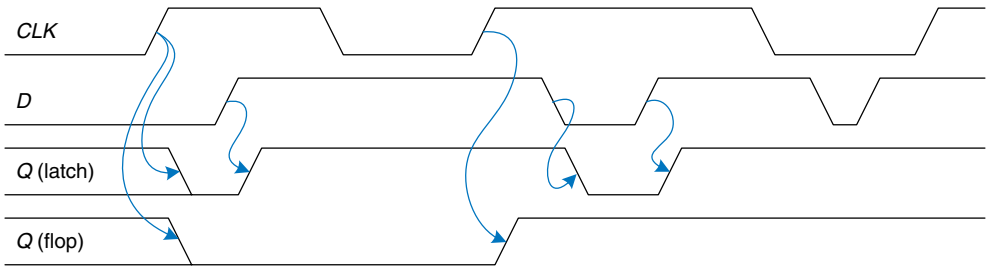


Figure 3.15 Solution waveforms

3.3 SYNCHRONOUS LOGIC DESIGN

In general, sequential circuits include all circuits that are not combinational—that is, those whose output cannot be determined simply by looking at the current inputs. Some sequential circuits are just plain kooky. This section begins by examining some of those curious circuits. It then introduces the notion of synchronous sequential circuits and the dynamic discipline. By disciplining ourselves to synchronous sequential circuits, we can develop easy, systematic ways to analyze and design sequential systems.

3.3.1 Some Problematic Circuits

Example 3.3 ASTABLE CIRCUITS

Alyssa P. Hacker encounters three misbegotten inverters who have tied themselves in a loop, as shown in Figure 3.16. The output of the third inverter is *fed back* to the first inverter. Each inverter has a propagation delay of 1 ns. Determine what the circuit does.

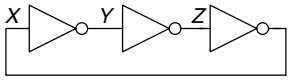


Figure 3.16 Three-inverter loop

Solution: Suppose node X is initially 0. Then $Y = 1$, $Z = 0$, and hence $X = 1$, which is inconsistent with our original assumption. The circuit has no stable states and is said to be *unstable* or *astable*. Figure 3.17 shows the behavior of the circuit. If X rises at time 0, Y will fall at 1 ns, Z will rise at 2 ns, and X will fall again at 3 ns. In turn, Y will rise at 4 ns, Z will fall at 5 ns, and X will rise again at 6 ns, and then the pattern will repeat. Each node oscillates between 0 and 1 with a *period* (repetition time) of 6 ns. This circuit is called a *ring oscillator*.

The period of the ring oscillator depends on the propagation delay of each inverter. This delay depends on how the inverter was manufactured, the power supply voltage, and even the temperature. Therefore, the ring oscillator period is difficult to accurately predict. In short, the ring oscillator is a sequential circuit with zero inputs and one output that changes periodically.

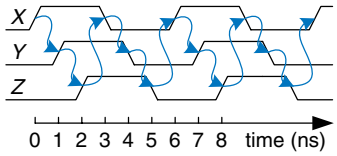


Figure 3.17 Ring oscillator waveforms

Example 3.4 RACE CONDITIONS

Ben Bitdiddle designed a new D latch that he claims is better than the one in Figure 3.17 because it uses fewer gates. He has written the truth table to find

CLK	D	Q_{prev}	Q
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

$$Q = CLK \cdot D + \overline{CLK} \cdot Q_{prev}$$

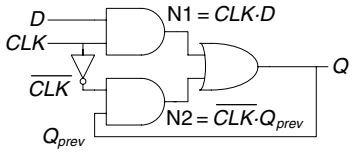


Figure 3.18 An improved (?) D latch

the output, Q, given the two inputs, D and CLK, and the old state of the latch, Q_{prev}. Based on this truth table, he has derived Boolean equations. He obtains Q_{prev} by feeding back the output, Q. His design is shown in Figure 3.18. Does his latch work correctly, independent of the delays of each gate?

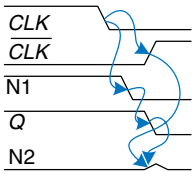


Figure 3.19 Latch waveforms illustrating race condition

Solution: Figure 3.19 shows that the circuit has a *race condition* that causes it to fail when certain gates are slower than others. Suppose $CLK = D = 1$. The latch is transparent and passes D through to make $Q = 1$. Now, CLK falls. The latch should remember its old value, keeping $Q = 1$. However, suppose the delay through the inverter from CLK to $\overline{CLK}$ is rather long compared to the delays of the AND and OR gates. Then nodes N1 and Q may both fall before $\overline{CLK}$ rises. In such a case, N2 will never rise, and Q becomes stuck at 0.

This is an example of *asynchronous* circuit design in which outputs are directly fed back to inputs. Asynchronous circuits are infamous for having race conditions where the behavior of the circuit depends on which of two paths through logic gates is fastest. One circuit may work, while a seemingly identical one built from gates with slightly different delays may not work. Or the circuit may work only at certain temperatures or voltages at which the delays are just right. These mistakes are extremely difficult to track down.

3.3.2 Synchronous Sequential Circuits

The previous two examples contain loops called *cyclic paths*, in which outputs are fed directly back to inputs. They are sequential rather than combinational circuits. Combinational logic has no cyclic paths and no races. If inputs are applied to combinational logic, the outputs will always settle to the correct value within a propagation delay. However, sequential circuits with cyclic paths can have undesirable races or unstable behavior. Analyzing such circuits for problems is time-consuming, and many bright people have made mistakes.

To avoid these problems, designers break the cyclic paths by inserting registers somewhere in the path. This transforms the circuit into a collection of combinational logic and registers. The registers contain the state of the system, which changes only at the clock edge, so we say the state is *synchronized* to the clock. If the clock is sufficiently slow, so that the inputs to all registers settle before the next clock edge, all races are eliminated. Adopting this discipline of always using registers in the feedback path leads us to the formal definition of a synchronous sequential circuit.

Recall that a circuit is defined by its input and output terminals and its functional and timing specifications. A sequential circuit has a finite set of discrete *states* $\{S_0, S_1, \ldots, S_{k-1}\}$. A *synchronous sequential circuit* has a clock input, whose rising edges indicate a sequence of times at which state transitions occur. We often use the terms *current state* and *next state* to distinguish the state of the system at the present from the state to which it will enter on the next clock edge. The functional specification details the next state and the value of each output for each possible combination of current state and input values. The timing specification consists of an upper bound, t_{pcq}, and a lower bound, t_{ccq}, on the time from the rising edge of the clock until the *output* changes, as well as *setup* and *hold* times, t_{setup} and t_{hold}, that indicate when the *inputs* must be stable relative to the rising edge of the clock.

t_{pcq} stands for the time of propagation from clock to Q, where Q indicates the output of a synchronous sequential circuit. t_{ccq} stands for the time of contamination from clock to Q. These are analogous to t_{pd} and t_{cd} in combinational logic.

The rules of s*ynchronous sequential circuit composition* teach us that a circuit is a synchronous sequential circuit if it consists of interconnected circuit elements such that

- Every circuit element is either a register or a combinational circuit
- At least one circuit element is a register
- All registers receive the same clock signal
- Every cyclic path contains at least one register.

This definition of a synchronous sequential circuit is sufficient, but more restrictive than necessary. For example, in high-performance microprocessors, some registers may receive delayed or gated clocks to squeeze out the last bit of performance or power. Similarly, some microprocessors use latches instead of registers. However, the definition is adequate for all of the synchronous sequential circuits covered in this book and for most commercial digital systems.

Sequential circuits that are not synchronous are called *asynchronous*.

A flip-flop is the simplest synchronous sequential circuit. It has one input, D, one clock, CLK, one output, Q, and two states, $\{0, 1\}$. The functional specification for a flip-flop is that the next state is D and that the output, Q, is the current state, as shown in Figure 3.20.

We often call the current state variable S and the next state variable S'. In this case, the prime after S indicates next state, not inversion. The timing of sequential circuits will be analyzed in Section 3.5.

Two other common types of synchronous sequential circuits are called finite state machines and pipelines. These will be covered later in this chapter.

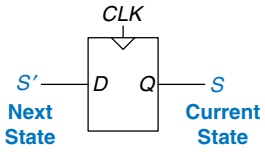


Figure 3.20 Flip-flop current state and next state

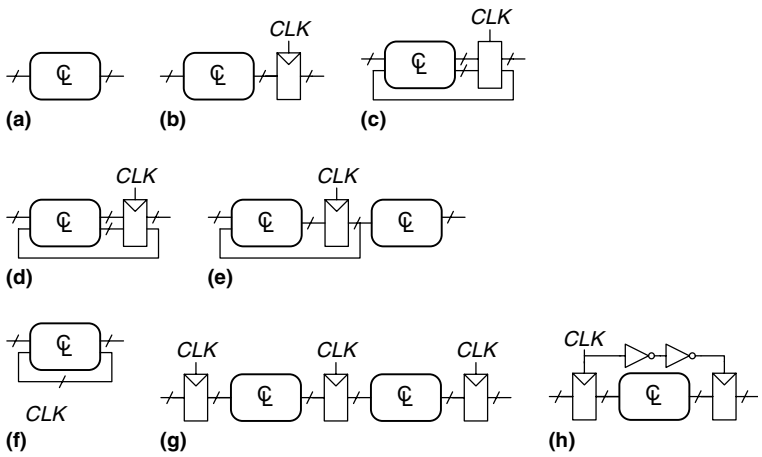


Figure 3.21 Example circuits

Example 3.5 SYNCHRONOUS SEQUENTIAL CIRCUITS

Which of the circuits in Figure 3.21 are synchronous sequential circuits?

Solution: Circuit (a) is combinational, not sequential, because it has no registers. (b) is a simple sequential circuit with no feedback. (c) is neither a combinational circuit nor a synchronous sequential circuit, because it has a latch that is neither a register nor a combinational circuit. (d) and (e) are synchronous sequential logic; they are two forms of finite state machines, which are discussed in Section 3.4. (f) is neither combinational nor synchronous sequential, because it has a cyclic path from the output of the combinational logic back to the input of the same logic but no register in the path. (g) is synchronous sequential logic in the form of a pipeline, which we will study in Section 3.6. (h) is not, strictly speaking, a synchronous sequential circuit, because the second register receives a different clock signal than the first, delayed by two inverter delays.

3.3.3 Synchronous and Asynchronous Circuits

Asynchronous design in theory is more general than synchronous design, because the timing of the system is not limited by clocked registers. Just as analog circuits are more general than digital circuits because analog circuits can use any voltage, asynchronous circuits are more general than synchronous circuits because they can use any kind of feedback. However, synchronous circuits have proved to be easier to design and use than asynchronous circuits, just as digital are easier than analog circuits. Despite decades of research on asynchronous circuits, virtually all digital systems are essentially synchronous.

Of course, asynchronous circuits are occasionally necessary when communicating between systems with different clocks or when receiving inputs at arbitrary times, just as analog circuits are necessary when communicating with the real world of continuous voltages. Furthermore, research in asynchronous circuits continues to generate interesting insights, some of which can improve synchronous circuits too.

3.4 FINITE STATE MACHINES

Synchronous sequential circuits can be drawn in the forms shown in Figure 3.22. These forms are called *finite state machines* (*FSMs*). They get their name because a circuit with k registers can be in one of a finite number (2^k) of unique states. An FSM has M inputs, N outputs, and k bits of state. It also receives a clock and, optionally, a reset signal. An FSM consists of two blocks of combinational logic, *next state logic* and *output logic*, and a register that stores the state. On each clock edge, the FSM advances to the next state, which was computed based on the current state and inputs. There are two general classes of finite state machines, characterized by their functional specifications. In *Moore machines*, the outputs depend only on the current state of the machine. In *Mealy machines*, the outputs depend on both the current state and the current inputs. Finite state machines provide a systematic way to design synchronous sequential circuits given a functional specification. This method will be explained in the remainder of this section, starting with an example.

Moore and Mealy machines are named after their promoters, researchers who developed *automata theory*, the mathematical underpinnings of state machines, at Bell Labs.

Edward F. Moore (1925–2003), not to be confused with Intel founder Gordon Moore, published his seminal article, *Gedanken-experiments on Sequential Machines* in 1956. He subsequently became a professor of mathematics and computer science at the University of Wisconsin.

George H. Mealy published *A Method of Synthesizing Sequential Circuits* in 1955. He subsequently wrote the first Bell Labs operating system for the IBM 704 computer. He later joined Harvard University.

3.4.1 FSM Design Example

To illustrate the design of FSMs, consider the problem of inventing a controller for a traffic light at a busy intersection on campus. Engineering students are moseying between their dorms and the labs on Academic Ave. They are busy reading about FSMs in their favorite

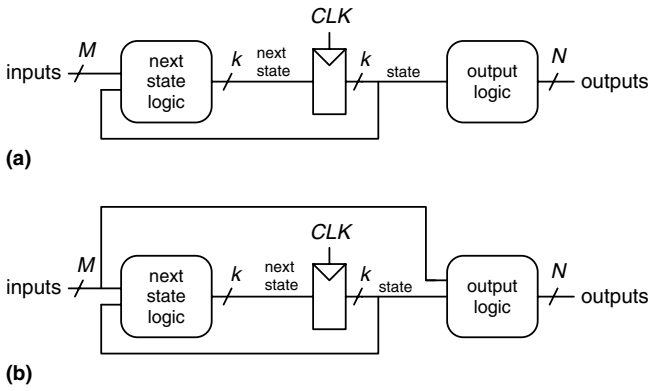


Figure 3.22 Finite state machines: (a) Moore machine, (b) Mealy machine

textbook and aren't looking where they are going. Football players are hustling between the athletic fields and the dining hall on Bravado Boulevard. They are tossing the ball back and forth and aren't looking where they are going either. Several serious injuries have already occurred at the intersection of these two roads, and the Dean of Students asks Ben Bitdiddle to install a traffic light before there are fatalities.

Ben decides to solve the problem with an FSM. He installs two traffic sensors, T_A and T_B, on Academic Ave. and Bravado Blvd., respectively. Each sensor indicates TRUE if students are present and FALSE if the street is empty. He also installs two traffic lights, L_A and L_B, to control traffic. Each light receives digital inputs specifying whether it should be green, yellow, or red. Hence, his FSM has two inputs, T_A and T_B, and two outputs, L_A and L_B. The intersection with lights and sensors is shown in Figure 3.23. Ben provides a clock with a 5-second period. On each clock tick (rising edge), the lights may change based on the traffic sensors. He also provides a reset button so that Physical Plant technicians can put the controller in a known initial state when they turn it on. Figure 3.24 shows a black box view of the state machine.

Ben's next step is to sketch the *state transition diagram*, shown in Figure 3.25, to indicate all the possible states of the system and the transitions between these states. When the system is reset, the lights are green on Academic Ave. and red on Bravado Blvd. Every 5 seconds, the controller examines the traffic pattern and decides what to do next. As long

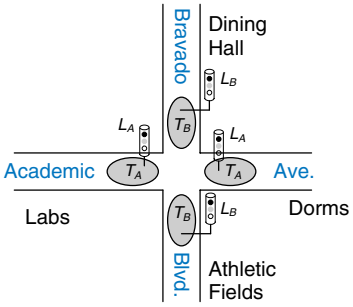


Figure 3.23 Campus map

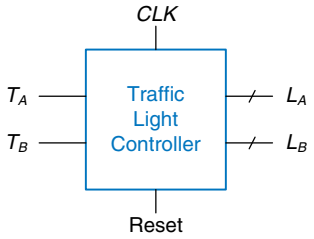


Figure 3.24 Black box view of finite state machine

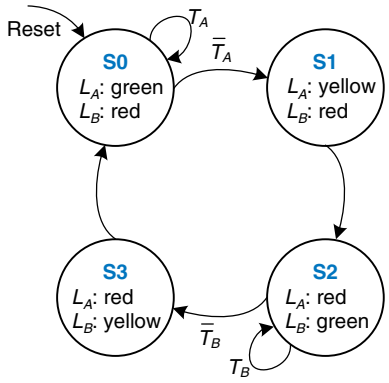


Figure 3.25 State transition diagram

as traffic is present on Academic Ave., the lights do not change. When there is no longer traffic on Academic Ave., the light on Academic Ave. becomes yellow for 5 seconds before it turns red and Bravado Blvd.'s light turns green. Similarly, the Bravado Blvd. light remains green as long as traffic is present on the boulevard, then turns yellow and eventually red.

In a state transition diagram, circles represent states and arcs represent transitions between states. The transitions take place on the rising edge of the clock; we do not bother to show the clock on the diagram, because it is always present in a synchronous sequential circuit. Moreover, the clock simply controls when the transitions should occur, whereas the diagram indicates which transitions occur. The arc labeled Reset pointing from outer space into state S0 indicates that the system should enter that state upon reset, regardless of what previous state it was in. If a state has multiple arcs leaving it, the arcs are labeled to show what input triggers each transition. For example, when in state S0, the system will remain in that state if T_A is TRUE and move to S1 if T_A is FALSE. If a state has a single arc leaving it, that transition always occurs regardless of the inputs. For example, when in state S1, the system will always move to S2. The value that the outputs have while in a particular state are indicated in the state. For example, while in state S2, L_A is red and L_B is green.

Ben rewrites the state transition diagram as a *state transition table* (Table 3.1), which indicates, for each state and input, what the next state, S', should be. Note that the table uses don't care symbols (X) whenever the next state does not depend on a particular input. Also note that Reset is omitted from the table. Instead, we use resettable flip-flops that always go to state S0 on reset, independent of the inputs.

The state transition diagram is abstract in that it uses states labeled {S0, S1, S2, S3} and outputs labeled {red, yellow, green}. To build a real circuit, the states and outputs must be assigned *binary encodings*. Ben chooses the simple encodings given in Tables 3.2 and 3.3. Each state and each output is encoded with two bits: $S_{1:0}$, $L_{A1:0}$, and $L_{B1:0}$.

Table 3.1 State transition table

Current State S	Inputs T_A	Inputs T_B	Next State S'
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0

Table 3.2 State encoding

State	Encoding $S_{1:0}$
S0	00
S1	01
S2	10
S3	11

Table 3.3 Output encoding

Output	Encoding $L_{1:0}$
green	00
yellow	01
red	10

Ben updates the state transition table to use these binary encodings, as shown in Table 3.4. The revised state transition table is a truth table specifying the next state logic. It defines next state, S', as a function of the current state, S, and the inputs. The revised output table is a truth table specifying the output logic. It defines the outputs, L_A and L_B, as functions of the current state, S.

From this table, it is straightforward to read off the Boolean equations for the next state in sum-of-products form.

$$S_1' = \overline{S}_1 S_0 + S_1 \overline{S}_0 \overline{T}_B + S_1 \overline{S}_0 T_B$$
$$S_0' = \overline{S}_1 \overline{S}_0 \overline{T}_A + S_1 \overline{S}_0 \overline{T}_B \quad (3.1)$$

The equations can be simplified using Karnaugh maps, but often doing it by inspection is easier. For example, the T_B and $\overline{T}_B$ terms in the S'_1 equation are clearly redundant. Thus S'_1 reduces to an XOR operation. Equation 3.2 gives the *next state equations*.

Table 3.4 State transition table with binary encodings

Current State S_1	Current State S_0	Inputs T_A	Inputs T_B	Next State S'_1	Next State S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

Table 3.5 Output table

Current State		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

$$\begin{aligned} S_1' &= S_1 \oplus S_0 \\ S_0' &= \overline{S}_1 \overline{S}_0 \overline{T}_A + S_1 \overline{S}_0 \overline{T}_B \end{aligned} \tag{3.2}$$

Similarly, Ben writes an *output table* (Table 3.5) indicating, for each state, what the output should be in that state. Again, it is straightforward to read off and simplify the Boolean equations for the outputs. For example, observe that L_{A1} is TRUE only on the rows where S_1 is TRUE.

$$\begin{aligned} L_{A1} &= S_1 \\ L_{A0} &= \overline{S}_1 S_0 \\ L_{B1} &= \overline{S}_1 \\ L_{B0} &= S_1 S_0 \end{aligned} \tag{3.3}$$

Finally, Ben sketches his Moore FSM in the form of Figure 3.22(a). First, he draws the 2-bit state register, as shown in Figure 3.26(a). On each clock edge, the state register copies the next state, $S'_{1:0}$, to become the state, $S_{1:0}$. The state register receives a synchronous or asynchronous reset to initialize the FSM at startup. Then, he draws the next state logic, based on Equation 3.2, which computes the next state, based on the current state and inputs, as shown in Figure 3.26(b). Finally, he draws the output logic, based on Equation 3.3, which computes the outputs based on the current state, as shown in Figure 3.26(c).

Figure 3.27 shows a timing diagram illustrating the traffic light controller going through a sequence of states. The diagram shows *CLK*, Reset, the inputs T_A and T_B, next state S', state S, and outputs L_A and L_B. Arrows indicate causality; for example, changing the state causes the outputs to change, and changing the inputs causes the next state to change. Dashed lines indicate the rising edge of *CLK* when the state changes.

The clock has a 5-second period, so the traffic lights change at most once every 5 seconds. When the finite state machine is first turned on, its state is unknown, as indicated by the question marks. Therefore, the system should be reset to put it into a known state. In this timing diagram,

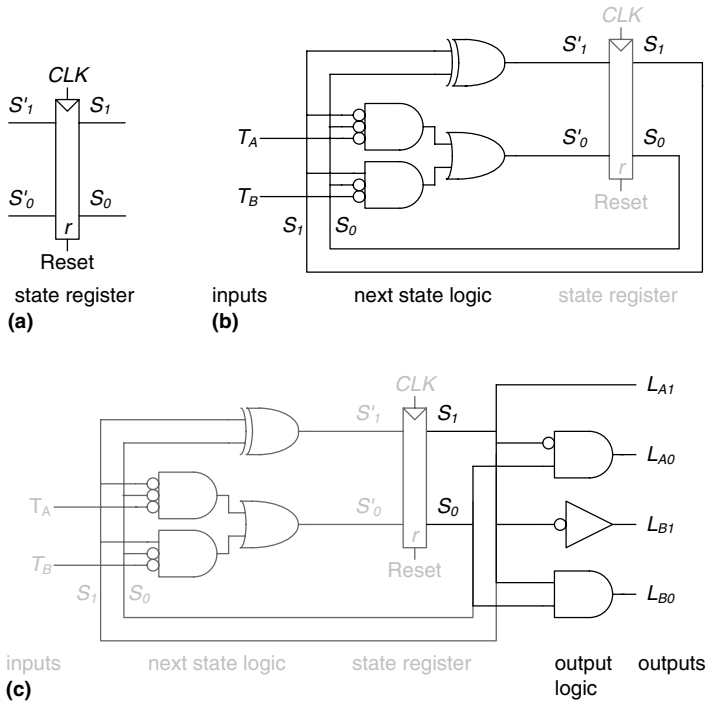


Figure 3.26 State machine circuit for traffic light controller

This schematic uses some AND gates with bubbles on the inputs. They might be constructed with AND gates and input inverters, with NOR gates and inverters for the non-bubbled inputs, or with some other combination of gates. The best choice depends on the particular implementation technology.

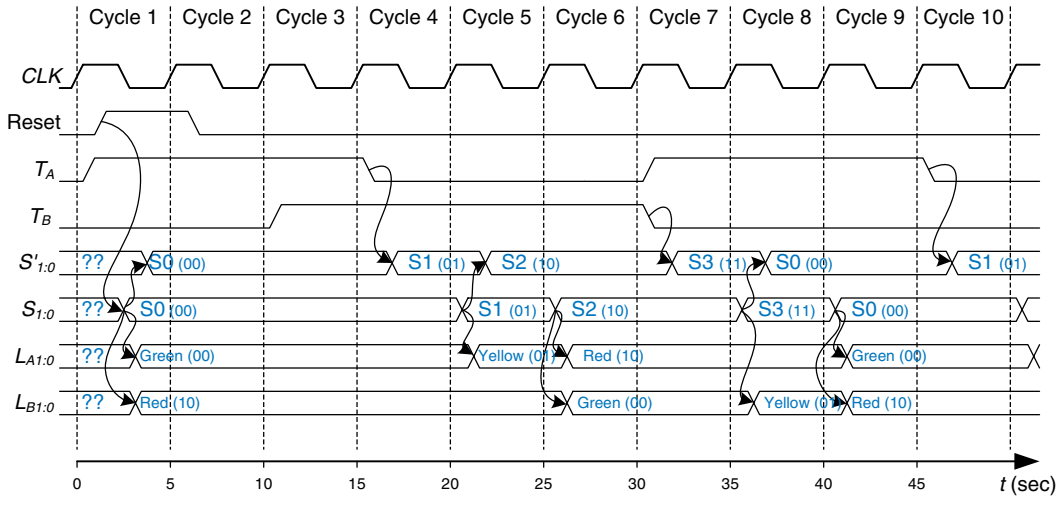
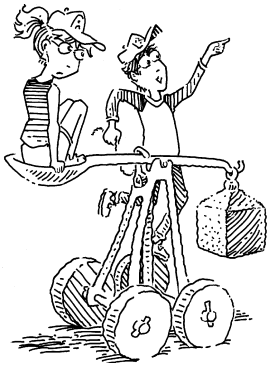


Figure 3.27 Timing diagram for traffic light controller

S immediately resets to S0, indicating that asynchronously resettable flip-flops are being used. In state S0, light L_A is green and light L_B is red.

In this example, traffic arrives immediately on Academic Ave. Therefore, the controller remains in state S0, keeping L_A green even though traffic arrives on Bravado Blvd. and starts waiting. After 15 seconds, the traffic on Academic Ave. has all passed through and T_A falls. At the following clock edge, the controller moves to state S1, turning L_A yellow. In another 5 seconds, the controller proceeds to state S2 in which L_A turns red and L_B turns green. The controller waits in state S2 until all the traffic on Bravado Blvd. has passed through. It then proceeds to state S3, turning L_B yellow. 5 seconds later, the controller enters state S0, turning L_B red and L_A green. The process repeats.

Despite Ben's best efforts, students don't pay attention to traffic lights and collisions continue to occur. The Dean of Students next asks him to design a catapult to throw engineering students directly from their dorm roofs through the open windows of the lab, bypassing the troublesome intersection all together. But that is the subject of another textbook.



3.4.2 State Encodings

In the previous example, the state and output encodings were selected arbitrarily. A different choice would have resulted in a different circuit. A natural question is how to determine the encoding that produces the circuit with the fewest logic gates or the shortest propagation delay. Unfortunately, there is no simple way to find the best encoding except to try all possibilities, which is infeasible when the number of states is large. However, it is often possible to choose a good encoding by inspection, so that related states or outputs share bits. Computer-aided design (CAD) tools are also good at searching the set of possible encodings and selecting a reasonable one.

One important decision in state encoding is the choice between binary encoding and one-hot encoding. With *binary encoding,* as was used in the traffic light controller example, each state is represented as a binary number. Because *K* binary numbers can be represented by $\log_2 K$ bits, a system with *K* states only needs $\log_2 K$ bits of state.

In *one-hot encoding,* a separate bit of state is used for each state. It is called one-hot because only one bit is "hot" or TRUE at any time. For example, a one-hot encoded FSM with three states would have state encodings of 001, 010, and 100. Each bit of state is stored in a flip-flop, so one-hot encoding requires more flip-flops than binary encoding. However, with one-hot encoding, the next-state and output logic is often simpler, so fewer gates are required. The best encoding choice depends on the specific FSM.

Example 3.6 FSM STATE ENCODING

A *divide-by-N counter* has one output and no inputs. The output *Y* is HIGH for one clock cycle out of every *N*. In other words, the output divides the frequency of the clock by *N*. The waveform and state transition diagram for a divide-by-3 counter is shown in Figure 3.28. Sketch circuit designs for such a counter using binary and one-hot state encodings.

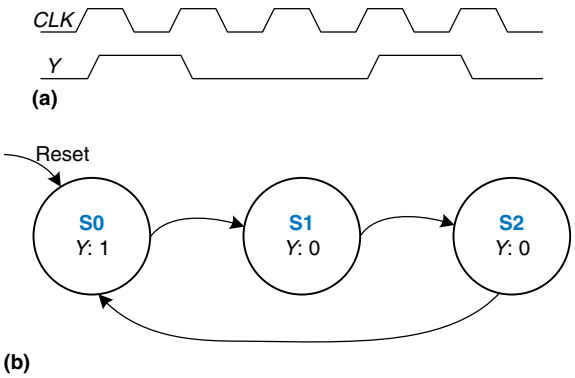


Figure 3.28 **Divide-by-3 counter (a) waveform and (b) state transition diagram**

Table 3.6 **Divide-by-3 counter state transition table**

Current State	Next State
S0	S1
S1	S2
S2	S0

Table 3.7 **Divide-by-3 counter output table**

Current State	Output
S0	1
S1	0
S2	0

Solution: Tables 3.6 and 3.7 show the abstract state transition and output tables before encoding.

Table 3.8 compares binary and one-hot encodings for the three states.

The binary encoding uses two bits of state. Using this encoding, the state transition table is shown in Table 3.9. Note that there are no inputs; the next state depends only on the current state. The output table is left as an exercise to the reader. The next-state and output equations are:

$$\begin{aligned} S_1' &= \overline{S}_1 S_0 \\ S_0' &= \overline{S}_1 \overline{S}_0 \end{aligned} \tag{3.4}$$

$$Y = \overline{S}_1 \overline{S}_0 \tag{3.5}$$

The one-hot encoding uses three bits of state. The state transition table for this encoding is shown in Table 3.10 and the output table is again left as an exercise to the reader. The next-state and output equations are as follows:

$$\begin{aligned} S_2' &= S_1 \\ S_1' &= S_0 \\ S_0' &= S_2 \end{aligned} \tag{3.6}$$

$$Y = S_0 \tag{3.7}$$

Figure 3.29 shows schematics for each of these designs. Note that the hardware for the binary encoded design could be optimized to share the same gate for Y and S'_0. Also observe that the one-hot encoding requires both settable (s) and resettable (r) flip-flops to initialize the machine to S0 on reset. The best implementation choice depends on the relative cost of gates and flip-flops, but the one-hot design is usually preferable for this specific example.

A related encoding is the *one-cold* encoding, in which K states are represented with K bits, exactly one of which is FALSE.

Table 3.8 Binary and one-hot encodings for divide-by-3 counter

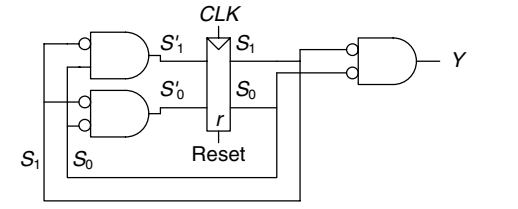
State	Binary Encoding S_2	S_1	S_0	One-Hot Encoding S_1	S_0
S0	0	0	1	0	1
S1	0	1	0	1	0
S2	1	0	0	0	0

Table 3.9 State transition table with binary encoding

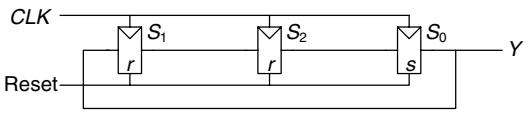
Current State S_1	S_0	Next State S'_1	S'_0
0	0	0	1
0	1	1	0
1	0	0	0

Table 3.10 State transition table with one-hot encoding

Current State S_2	S_1	S_0	Next State S'_2	S'_1	S'_0
0	0	1	0	1	0
0	1	0	1	0	0
1	0	0	0	0	1



(a)



(b)

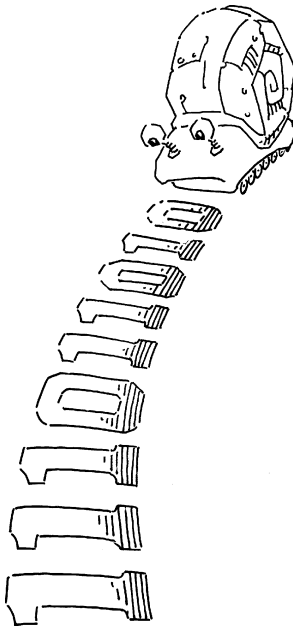
Figure 3.29 Divide-by-3 circuits for (a) binary and (b) one-hot encodings

3.4.3 Moore and Mealy Machines

An easy way to remember the difference between the two types of finite state machines is that a Moore machine typically has *more* states than a Mealy machine for a given problem.

So far, we have shown examples of Moore machines, in which the output depends only on the state of the system. Hence, in state transition diagrams for Moore machines, the outputs are labeled in the circles. Recall that Mealy machines are much like Moore machines, but the outputs can depend on inputs as well as the current state. Hence, in state transition diagrams for Mealy machines, the outputs are labeled on the arcs instead of in the circles. The block of combinational logic that computes the outputs uses the current state and inputs, as was shown in Figure 3.22(b).

Example 3.7 MOORE VERSUS MEALY MACHINES



Alyssa P. Hacker owns a pet robotic snail with an FSM brain. The snail crawls from left to right along a paper tape containing a sequence of 1's and 0's. On each clock cycle, the snail crawls to the next bit. The snail smiles when the last four bits that it has crawled over are, from left to right, 1101. Design the FSM to compute when the snail should smile. The input A is the bit underneath the snail's antennae. The output Y is TRUE when the snail smiles. Compare Moore and Mealy state machine designs. Sketch a timing diagram for each machine showing the input, states, and output as your snail crawls along the sequence 111011010.

Solution: The Moore machine requires five states, as shown in Figure 3.30(a). Convince yourself that the state transition diagram is correct. In particular, why is there an arc from S4 to S2 when the input is 1?

In comparison, the Mealy machine requires only four states, as shown in Figure 3.30(b). Each arc is labeled as A/Y. A is the value of the input that causes that transition, and Y is the corresponding output.

Tables 3.11 and 3.12 show the state transition and output tables for the Moore machine. The Moore machine requires at least three bits of state. Consider using a binary state encoding: S0 = 000, S1 = 001, S2 = 010, S3 = 011, and S4 = 100. Tables 3.13 and 3.14 rewrite the state transition and output tables with these encodings (These four tables follow on page 128).

From these tables, we find the next state and output equations by inspection. Note that these equations are simplified using the fact that states 101, 110, and 111 do not exist. Thus, the corresponding next state and output for the nonexistent states are don't cares (not shown in the tables). We use the don't cares to minimize our equations.

$$
\begin{aligned}
S_2' &= S_1 S_0 A \\
S_1' &= \overline{S}_1 S_0 A + S_1 \overline{S}_0 + S_2 A \\
S_0' &= \overline{S}_2 \overline{S}_1 \overline{S}_0 A + S_1 \overline{S}_0 \overline{A}
\end{aligned}
\tag{3.8}
$$

$$Y = S_2 \tag{3.9}$$

Table 3.15 shows the combined state transition and output table for the Mealy machine. The Mealy machine requires at least two bits of state. Consider using a binary state encoding: S0 = 00, S1 = 01, S2 = 10, and S3 = 11. Table 3.16 rewrites the state transition and output table with these encodings.

From these tables, we find the next state and output equations by inspection.

$$\begin{aligned} S_1' &= S_1\overline{S}_0 + \overline{S}_1 S_0 A \\ S_0' &= \overline{S}_1\overline{S}_0 A + S_1\overline{S}_0\overline{A} + S_1 S_0 A \end{aligned} \tag{3.10}$$

$$Y = S_1 S_0 A \tag{3.11}$$

The Moore and Mealy machine schematics are shown in Figure 3.31(a) and 3.31(b), respectively.

The timing diagrams for the Moore and Mealy machines are shown in Figure 3.32 (see page 131). The two machines follow a different sequence of states. Moreover, the Mealy machine's output rises a cycle sooner because it responds to the input rather than waiting for the state change. If the Mealy output were delayed through a flip-flop, it would match the Moore output. When choosing your FSM design style, consider when you want your outputs to respond.

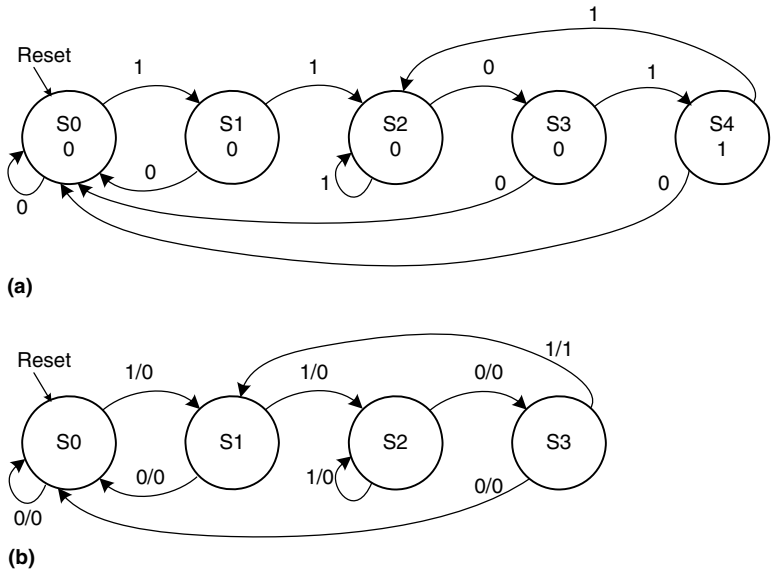


Figure 3.30 **FSM state transition diagrams: (a) Moore machine, (b) Mealy machine**

Table 3.11 Moore state transition table

Current State S	Input A	Next State S'
S0	0	S0
S0	1	S1
S1	0	S0
S1	1	S2
S2	0	S3
S2	1	S2
S3	0	S0
S3	1	S4
S4	0	S0
S4	1	S2

Table 3.12 Moore output table

Current State S	Output Y
S0	0
S1	0
S2	0
S3	0
S4	1

Table 3.13 Moore state transition table with state encodings

Current State S_2	S_1	S_0	Input A	Next State S'_2	S'_1	S'_0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	0	0
0	0	1	1	0	1	0
0	1	0	0	0	1	1
0	1	0	1	0	1	0
0	1	1	0	0	0	0
0	1	1	1	1	0	0
1	0	0	0	0	0	0
1	0	0	1	0	1	0

Table 3.14 Moore output table with state encodings

Current State S_2	S_1	S_0	Output Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1

Table 3.15 Mealy state transition and output table

Current State S	Input A	Next State S'	Output Y
S0	0	S0	0
S0	1	S1	0
S1	0	S0	0
S1	1	S2	0
S2	0	S3	0
S2	1	S2	0
S3	0	S0	0
S3	1	S1	1

Table 3.16 Mealy state transition and output table with state encodings

Current State S_1	Current State S_0	Input A	Next State S'_1	Next State S'_0	Output Y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	1	1	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	1	0	1	1

3.4.4 Factoring State Machines

Designing complex FSMs is often easier if they can be broken down into multiple interacting simpler state machines such that the output of some machines is the input of others. This application of hierarchy and modularity is called *factoring* of state machines.

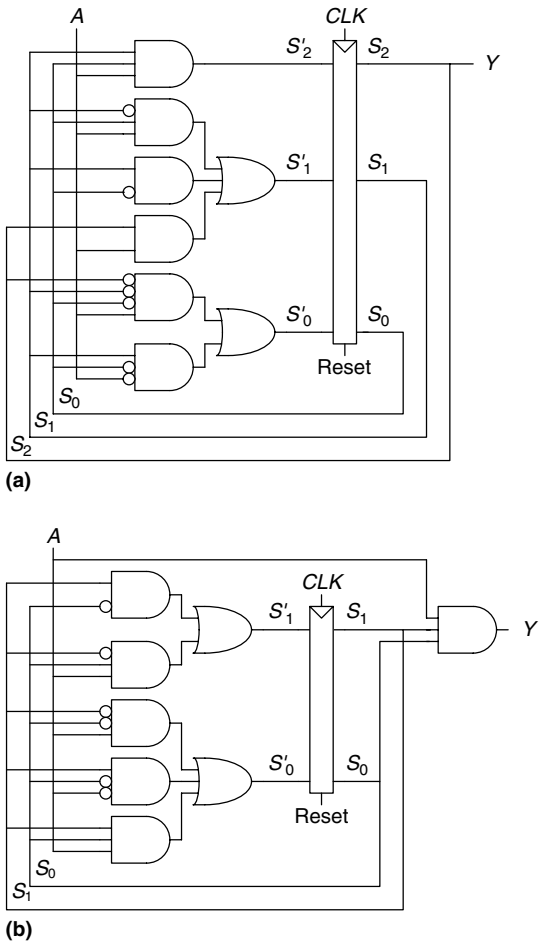


Figure 3.31 FSM schematics for (a) Moore and (b) Mealy machines

Example 3.8 UNFACTORED AND FACTORED STATE MACHINES

Modify the traffic light controller from Section 3.4.1 to have a parade mode, which keeps the Bravado Boulevard light green while spectators and the band march to football games in scattered groups. The controller receives two more inputs: P and R. Asserting P for at least one cycle enters parade mode. Asserting R for at least one cycle leaves parade mode. When in parade mode, the controller proceeds through its usual sequence until L_B turns green, then remains in that state with L_B green until parade mode ends.

First, sketch a state transition diagram for a single FSM, as shown in Figure 3.33(a). Then, sketch the state transition diagrams for two interacting FSMs, as

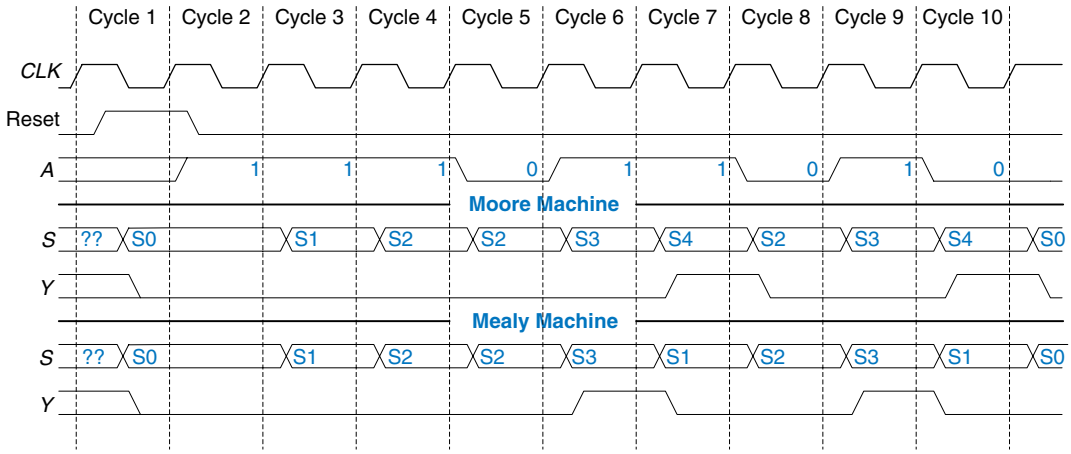


Figure 3.32 Timing diagrams for Moore and Mealy machines

shown in Figure 3.33(b). The Mode FSM asserts the output M when it is in parade mode. The Lights FSM controls the lights based on M and the traffic sensors, T_A and T_B.

Solution: Figure 3.34(a) shows the single FSM design. States S0 to S3 handle normal mode. States S4 to S7 handle parade mode. The two halves of the diagram are almost identical, but in parade mode, the FSM remains in S6 with a green light on Bravado Blvd. The P and R inputs control movement between these two halves. The FSM is messy and tedious to design. Figure 3.34(b) shows the factored FSM design. The mode FSM has two states to track whether the lights are in normal or parade mode. The Lights FSM is modified to remain in S2 while M is TRUE.

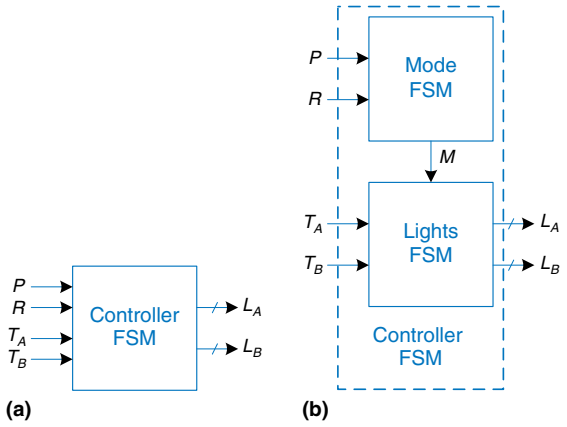


Figure 3.33 (a) single and (b) factored designs for modified traffic light controller FSM

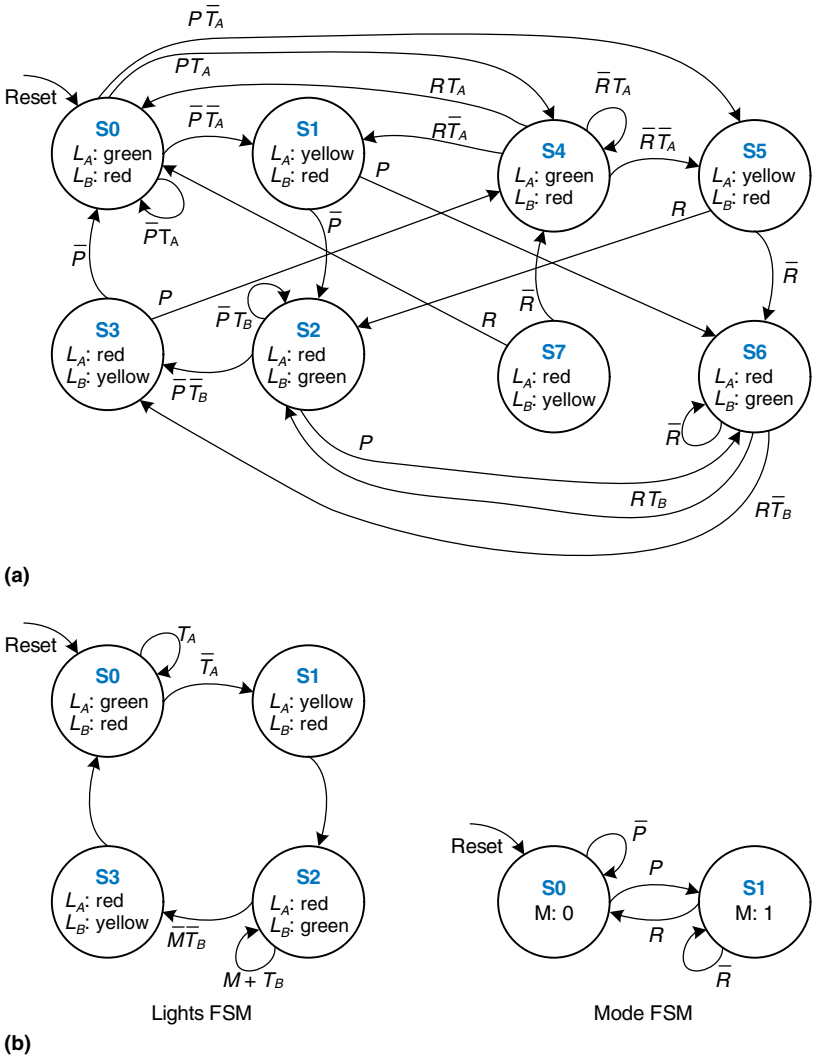


Figure 3.34 **State transition diagrams: (a) unfactored, (b) factored**

3.4.5 FSM Review

Finite state machines are a powerful way to systematically design sequential circuits from a written specification. Use the following procedure to design an FSM:

- Identify the inputs and outputs.
- Sketch a state transition diagram.

- For a Moore machine:
  - Write a state transition table.
  - Write an output table.
- For a Mealy machine:
  - Write a combined state transition and output table.
- Select state encodings—your selection affects the hardware design.
- Write Boolean equations for the next state and output logic.
- Sketch the circuit schematic.

We will repeatedly use FSMs to design complex digital systems throughout this book.

3.5 TIMING OF SEQUENTIAL LOGIC

Recall that a flip-flop copies the input D to the output Q on the rising edge of the clock. This process is called *sampling* D on the clock edge. If D is *stable* at either 0 or 1 when the clock rises, this behavior is clearly defined. But what happens if D is changing at the same time the clock rises?

This problem is similar to that faced by a camera when snapping a picture. Imagine photographing a frog jumping from a lily pad into the lake. If you take the picture before the jump, you will see a frog on a lily pad. If you take the picture after the jump, you will see ripples in the water. But if you take it just as the frog jumps, you may see a blurred image of the frog stretching from the lily pad into the water. A camera is characterized by its *aperture time*, during which the object must remain still for a sharp image to be captured. Similarly, a sequential element has an aperture time around the clock edge, during which the input must be stable for the flip-flop to produce a well-defined output.

The aperture of a sequential element is defined by a *setup* time and a *hold* time, before and after the clock edge, respectively. Just as the static discipline limited us to using logic levels outside the forbidden zone, the *dynamic discipline* limits us to using signals that change outside the aperture time. By taking advantage of the dynamic discipline, we can think of time in discrete units called clock cycles, just as we think of signal levels as discrete 1's and 0's. A signal may glitch and oscillate wildly for some bounded amount of time. Under the dynamic discipline, we are concerned only about its final value at the end of the clock cycle, after it has settled to a stable value. Hence, we can simply write $A[n]$, the value of signal A at the end of the n^{th} clock cycle, where n is an integer, rather than $A(t)$, the value of A at some instant t, where t is any real number.

The clock period has to be long enough for all signals to settle. This sets a limit on the speed of the system. In real systems, the clock does not

reach all flip-flops at precisely the same time. This variation in time, called clock skew, further increases the necessary clock period.

Sometimes it is impossible to satisfy the dynamic discipline, especially when interfacing with the real world. For example, consider a circuit with an input coming from a button. A monkey might press the button just as the clock rises. This can result in a phenomenon called metastability, where the flip-flop captures a value partway between 0 and 1 that can take an unlimited amount of time to resolve into a good logic value. The solution to such asynchronous inputs is to use a synchronizer, which has a very small (but nonzero) probability of producing an illegal logic value.

We expand on all of these ideas in the rest of this section.

3.5.1 The Dynamic Discipline

So far, we have focused on the functional specification of sequential circuits. Recall that a synchronous sequential circuit, such as a flip-flop or FSM, also has a timing specification, as illustrated in Figure 3.35. When the clock rises, the output (or outputs) may start to change after the clock-to-Q *contamination delay*, t_{ccq}, and must definitely settle to the final value within the clock-to-Q *propagation delay*, t_{pcq}. These represent the fastest and slowest delays through the circuit, respectively. For the circuit to sample its input correctly, the input (or inputs) must have stabilized at least some *setup time*, t_{setup}, before the rising edge of the clock and must remain stable for at least some *hold time*, t_{hold}, after the rising edge of the clock. The sum of the setup and hold times is called the *aperture time* of the circuit, because it is the total time for which the input must remain stable.

The *dynamic discipline* states that the inputs of a synchronous sequential circuit must be stable during the setup and hold aperture time around the clock edge. By imposing this requirement, we guarantee that the flip-flops sample signals while they are not changing. Because we are concerned only about the final values of the inputs at the time they are sampled, we can treat signals as discrete in time as well as in logic levels.

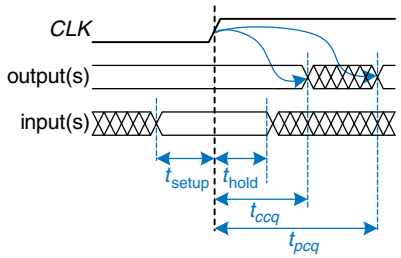


Figure 3.35 Timing specification for synchronous sequential circuit

3.5.2 System Timing

In the three decades from when one of the authors' families bought an Apple II+ computer to the present time of writing, microprocessor clock frequencies have increased from 1 MHz to several GHz, a factor of more than 1000. This speedup partially explains the revolutionary changes computers have made in society.

The *clock period* or *cycle time*, T_c, is the time between rising edges of a repetitive clock signal. Its reciprocal, $f_c = 1/T_c$, is the *clock frequency*. All else being the same, increasing the clock frequency increases the work that a digital system can accomplish per unit time. Frequency is measured in units of Hertz (Hz), or cycles per second: 1 megahertz (MHz) = 10^6 Hz, and 1 gigahertz (GHz) = 10^9 Hz.

Figure 3.36(a) illustrates a generic path in a synchronous sequential circuit whose clock period we wish to calculate. On the rising edge of the clock, register R1 produces output (or outputs) $Q1$. These signals enter a block of combinational logic, producing $D2$, the input (or inputs) to register R2. The timing diagram in Figure 3.36(b) shows that each output signal may start to change a contamination delay after its input change and settles to the final value within a propagation delay after its input settles. The gray arrows represent the contamination delay through R1 and the combinational logic, and the blue arrows represent the propagation delay through R1 and the combinational logic. We analyze the timing constraints with respect to the setup and hold time of the second register, R2.

Setup Time Constraint

Figure 3.37 is the timing diagram showing only the maximum delay through the path, indicated by the blue arrows. To satisfy the setup time of R2, $D2$ must settle no later than the setup time before the next clock edge.

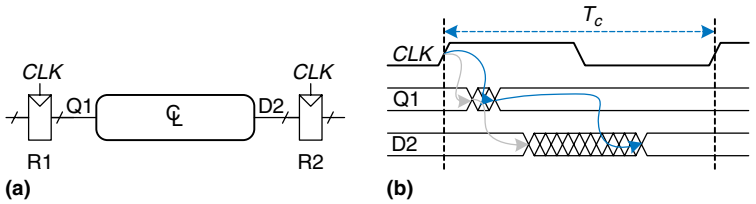


Figure 3.36 Path between registers and timing diagram

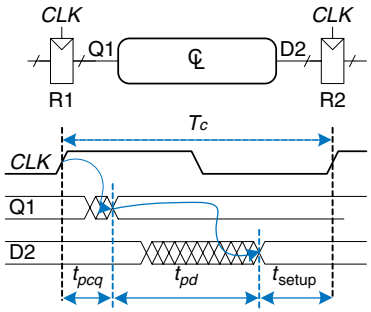


Figure 3.37 Maximum delay for setup time constraint

Hence, we find an equation for the minimum clock period:

$$T_c \geq t_{pcq} + t_{pd} + t_{\text{setup}} \tag{3.12}$$

In commercial designs, the clock period is often dictated by the Director of Engineering or by the marketing department (to ensure a competitive product). Moreover, the flip-flop clock-to-Q propagation delay and setup time, t_{pcq} and t_{setup}, are specified by the manufacturer. Hence, we rearrange Equation 3.12 to solve for the maximum propagation delay through the combinational logic, which is usually the only variable under the control of the individual designer.

$$t_{pd} \leq T_c - (t_{pcq} + t_{\text{setup}}) \tag{3.13}$$

The term in parentheses, $t_{pcq} + t_{\text{setup}}$, is called the *sequencing overhead*. Ideally, the entire cycle time, T_c, would be available for useful computation in the combinational logic, t_{pd}. However, the sequencing overhead of the flip-flop cuts into this time. Equation 3.13 is called the *setup time constraint* or *max-delay constraint*, because it depends on the setup time and limits the maximum delay through combinational logic.

If the propagation delay through the combinational logic is too great, $D2$ may not have settled to its final value by the time R2 needs it to be stable and samples it. Hence, R2 may sample an incorrect result or even an illegal logic level, a level in the forbidden region. In such a case, the circuit will malfunction. The problem can be solved by increasing the clock period or by redesigning the combinational logic to have a shorter propagation delay.

Hold Time Constraint

The register R2 in Figure 3.36(a) also has a *hold time constraint*. Its input, $D2$, must not change until some time, t_{hold}, after the rising edge of the clock. According to Figure 3.38, $D2$ might change as soon as $t_{ccq} + t_{cd}$ after the rising edge of the clock.

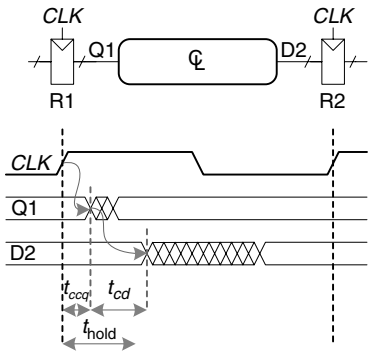


Figure 3.38 Minimum delay for hold time constraint

Hence, we find

$$t_{ccq} + t_{cd} \geq t_{\text{hold}} \tag{3.14}$$

Again, t_{ccq} and t_{hold} are characteristics of the flip-flop that are usually outside the designer's control. Rearranging, we can solve for the minimum contamination delay through the combinational logic:

$$t_{cd} \geq t_{\text{hold}} - t_{ccq} \tag{3.15}$$

Equation 3.15 is also called the *min-delay constraint* because it limits the minimum delay through combinational logic.

We have assumed that any logic elements can be connected to each other without introducing timing problems. In particular, we would expect that two flip-flops may be directly cascaded as in Figure 3.39 without causing hold time problems.

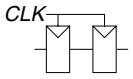


Figure 3.39 Back-to-back flip-flops

In such a case, $t_{cd} = 0$ because there is no combinational logic between flip-flops. Substituting into Equation 3.15 yields the requirement that

$$t_{\text{hold}} \leq t_{ccq} \tag{3.16}$$

In other words, a reliable flip-flop must have a hold time shorter than its contamination delay. Often, flip-flops are designed with $t_{\text{hold}} = 0$, so that Equation 3.16 is always satisfied. Unless noted otherwise, we will usually make that assumption and ignore the hold time constraint in this book.

Nevertheless, hold time constraints are critically important. If they are violated, the only solution is to increase the contamination delay through the logic, which requires redesigning the circuit. Unlike setup time constraints, they cannot be fixed by adjusting the clock period. Redesigning an integrated circuit and manufacturing the corrected design takes months and millions of dollars in today's advanced technologies, so *hold time violations* must be taken extremely seriously.

Putting It All Together

Sequential circuits have setup and hold time constraints that dictate the maximum and minimum delays of the combinational logic between flip-flops. Modern flip-flops are usually designed so that the minimum delay through the combinational logic is 0—that is, flip-flops can be placed back-to-back. The maximum delay constraint limits the number of consecutive gates on the critical path of a high-speed circuit, because a high clock frequency means a short clock period.

Example 3.9 TIMING ANALYSIS

Ben Bitdiddle designed the circuit in Figure 3.40. According to the data sheets for the components he is using, flip-flops have a clock-to-Q contamination delay

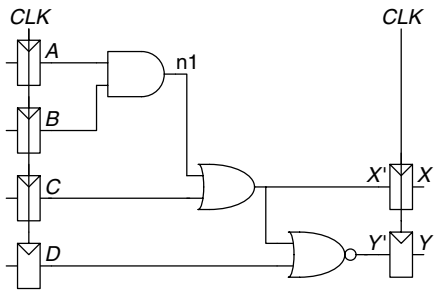


Figure 3.40 Sample circuit for timing analysis

of 30 ps and a propagation delay of 80 ps. They have a setup time of 50 ps and a hold time of 60 ps. Each logic gate has a propagation delay of 40 ps and a contamination delay of 25 ps. Help Ben determine the maximum clock frequency and whether any hold time violations could occur. This process is called *timing analysis*.

Solution: Figure 3.41(a) shows waveforms illustrating when the signals might change. The inputs, A to D, are registered, so they change shortly after CLK rises only.

The critical path occurs when $B = 1$, $C = 0$, $D = 0$, and A rises from 0 to 1, triggering n1 to rise, X' to rise and Y' to fall, as shown in Figure 3.41(b). This path involves three gate delays. For the critical path, we assume that each gate requires its full propagation delay. Y' must setup before the next rising edge of the CLK. Hence, the minimum cycle time is

$$T_c \geq t_{pcq} + 3\,t_{pd} + t_{\text{setup}} = 80 + 3 \times 40 + 50 = 250 \text{ ps} \qquad (3.17)$$

The maximum clock frequency is $f_c = 1/T_c = 4$ GHz.

A short path occurs when $A = 0$ and C rises, causing X' to rise, as shown in Figure 3.41(c). For the short path, we assume that each gate switches after only a contamination delay. This path involves only one gate delay, so it may occur after $t_{ccq} + t_{cd} = 30 + 25 = 55$ ps. But recall that the flip-flop has a hold time of 60 ps, meaning that X' must remain stable for 60 ps after the rising edge of CLK for the flip-flop to reliably sample its value. In this case, $X' = 0$ at the first rising edge of CLK, so we want the flip-flop to capture $X = 0$. Because X' did not hold stable long enough, the actual value of X is unpredictable. The circuit has a hold time violation and may behave erratically at any clock frequency.

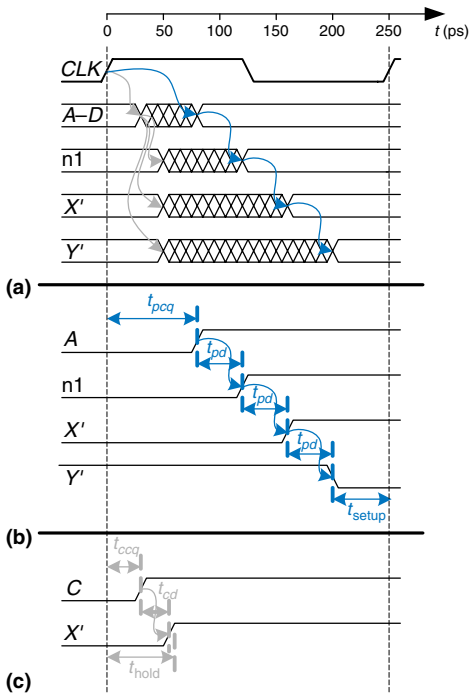


Figure 3.41 Timing diagram: (a) general case, (b) critical path, (c) short path

Example 3.10 FIXING HOLD TIME VIOLATIONS

Alyssa P. Hacker proposes to fix Ben's circuit by adding buffers to slow down the short paths, as shown in Figure 3.42. The buffers have the same delays as other gates. Determine the maximum clock frequency and whether any hold time problems could occur.

Solution: Figure 3.43 shows waveforms illustrating when the signals might change. The critical path from A to Y is unaffected, because it does not pass through any buffers. Therefore, the maximum clock frequency is still 4 GHz. However, the short paths are slowed by the contamination delay of the buffer. Now X' will not change until $t_{ccq} + 2t_{cd} = 30 + 2 \times 25 = 80$ ps. This is after the 60 ps hold time has elapsed, so the circuit now operates correctly.

This example had an unusually long hold time to illustrate the point of hold time problems. Most flip-flops are designed with $t_{\text{hold}} < t_{ccq}$ to avoid such problems. However, several high-performance microprocessors, including the Pentium 4, use an element called a *pulsed latch* in place of a flip-flop. The pulsed latch behaves like a flip-flop but has a short clock-to Q delay and a long hold time. In general, adding buffers can usually, but not always, solve hold time problems without slowing the critical path.

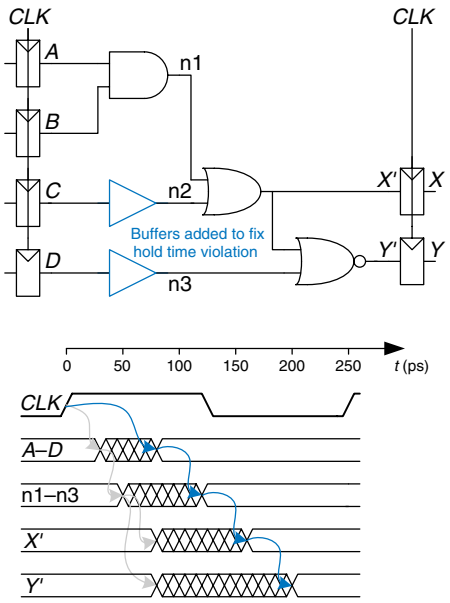


Figure 3.42 Corrected circuit to fix hold time problem

Figure 3.43 Timing diagram with buffers to fix hold time problem

3.5.3 Clock Skew*

In the previous analysis, we assumed that the clock reaches all registers at exactly the same time. In reality, there is some variation in this time. This variation in clock edges is called *clock skew*. For example, the wires from the clock source to different registers may be of different lengths, resulting in slightly different delays, as shown in Figure 3.44. Noise also results in different delays. Clock gating, described in Section 3.2.5, further delays the clock. If some clocks are gated and others are not, there will be substantial skew between the gated and ungated clocks. In

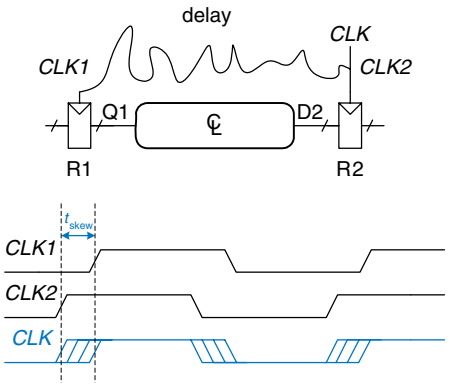


Figure 3.44 Clock skew caused by wire delay

Figure 3.44, *CLK*2 is *early* with respect to *CLK*1, because the clock wire between the two registers follows a scenic route. If the clock had been routed differently, *CLK*1 might have been early instead. When doing timing analysis, we consider the worst-case scenario, so that we can guarantee that the circuit will work under all circumstances.

Figure 3.45 adds skew to the timing diagram from Figure 3.36. The heavy clock line indicates the latest time at which the clock signal might reach any register; the hashed lines show that the clock might arrive up to t_{skew} earlier.

First, consider the setup time constraint shown in Figure 3.46. In the worst case, R1 receives the latest skewed clock and R2 receives the earliest skewed clock, leaving as little time as possible for data to propagate between the registers.

The data propagates through the register and combinational logic and must setup before R2 samples it. Hence, we conclude that

$$T_c \geq t_{pcq} + t_{pd} + t_{\text{setup}} + t_{\text{skew}} \tag{3.18}$$

$$t_{pd} \leq T_c - (t_{pcq} + t_{\text{setup}} + t_{\text{skew}}) \tag{3.19}$$

Next, consider the hold time constraint shown in Figure 3.47. In the worst case, R1 receives an early skewed clock, *CLK*1, and R2 receives a late skewed clock, *CLK*2. The data zips through the register

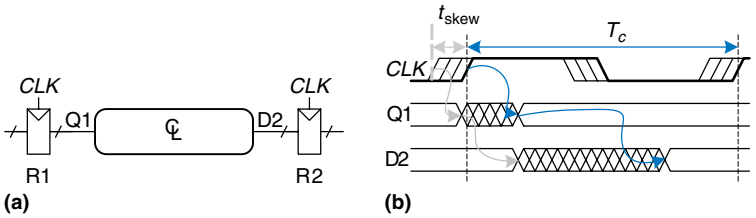


Figure 3.45 Timing diagram with clock skew

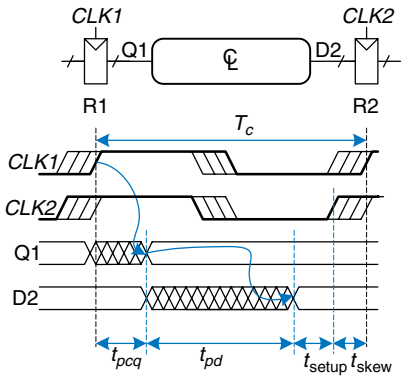


Figure 3.46 Setup time constraint with clock skew

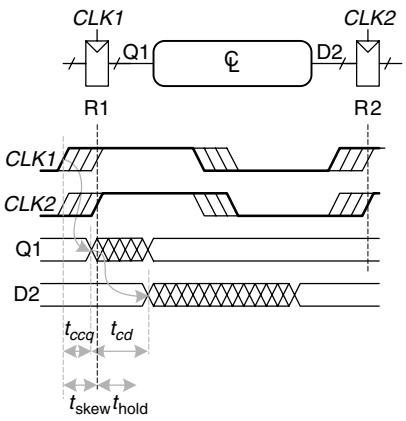


Figure 3.47 Hold time constraint with clock skew

and combinational logic but must not arrive until a hold time after the late clock. Thus, we find that

$$t_{ccq} + t_{cd} \geq t_{\text{hold}} + t_{\text{skew}} \tag{3.20}$$

$$t_{cd} \geq t_{\text{hold}} + t_{\text{skew}} - t_{ccq} \tag{3.21}$$

In summary, clock skew effectively increases both the setup time and the hold time. It adds to the sequencing overhead, reducing the time available for useful work in the combinational logic. It also increases the required minimum delay through the combinational logic. Even if $t_{\text{hold}} = 0$, a pair of back-to-back flip-flops will violate Equation 3.21 if $t_{\text{skew}} > t_{ccq}$. To prevent serious hold time failures, designers must not permit too much clock skew. Sometimes flip-flops are intentionally designed to be particularly slow (i.e., large t_{ccq}), to prevent hold time problems even when the clock skew is substantial.

Example 3.11 TIMING ANALYSIS WITH CLOCK SKEW

Revisit Example 3.9 and assume that the system has 50 ps of clock skew.

Solution: The critical path remains the same, but the setup time is effectively increased by the skew. Hence, the minimum cycle time is

$$\begin{aligned} T_c &\geq t_{pcq} + 3t_{pd} + t_{\text{setup}} + t_{\text{skew}} \\ &= 80 + 3 \times 40 + 50 + 50 = 300 \text{ ps} \end{aligned} \tag{3.22}$$

The maximum clock frequency is $f_c = 1/T_c = 3.33$ GHz.

The short path also remains the same at 55 ps. The hold time is effectively increased by the skew to 60 + 50 = 110 ps, which is much greater than 55 ps. Hence, the circuit will violate the hold time and malfunction at any frequency. The circuit violated the hold time constraint even without skew. Skew in the system just makes the violation worse.

Example 3.12 FIXING HOLD TIME VIOLATIONS

Revisit Example 3.10 and assume that the system has 50 ps of clock skew.

Solution: The critical path is unaffected, so the maximum clock frequency remains 3.33 GHz.

The short path increases to 80 ps. This is still less than $t_{hold} + t_{skew} = 110$ ps, so the circuit still violates its hold time constraint.

To fix the problem, even more buffers could be inserted. Buffers would need to be added on the critical path as well, reducing the clock frequency. Alternatively, a better flip-flop with a shorter hold time might be used.

3.5.4 Metastability

As noted earlier, it is not always possible to guarantee that the input to a sequential circuit is stable during the aperture time, especially when the input arrives from the external world. Consider a button connected to the input of a flip-flop, as shown in Figure 3.48. When the button is not pressed, $D = 0$. When the button is pressed, $D = 1$. A monkey presses the button at some random time relative to the rising edge of *CLK*. We want to know the output Q after the rising edge of *CLK*. In Case I, when the button is pressed much before *CLK*, $Q = 1$.

In Case II, when the button is not pressed until long after *CLK*, $Q = 0$. But in Case III, when the button is pressed sometime between t_{setup} before *CLK* and t_{hold} after *CLK*, the input violates the dynamic discipline and the output is undefined.

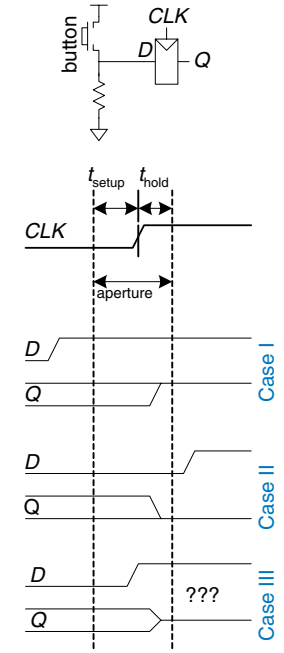


Figure 3.48 Input changing before, after, or during aperture

Metastable State

In reality, when a flip-flop samples an input that is changing during its aperture, the output Q may momentarily take on a voltage between 0 and V_{DD} that is in the forbidden zone. This is called a *metastable* state. Eventually, the flip-flop will resolve the output to a *stable state* of either 0 or 1. However, the *resolution time* required to reach the stable state is unbounded.

The metastable state of a flip-flop is analogous to a ball on the summit of a hill between two valleys, as shown in Figure 3.49. The two valleys are stable states, because a ball in the valley will remain there as long as it is not disturbed. The top of the hill is called metastable because the ball would remain there if it were perfectly balanced. But because nothing is perfect, the ball will eventually roll to one side or the other. The time required for this change to occur depends on how nearly well balanced the ball originally was. Every bistable device has a metastable state between the two stable states.

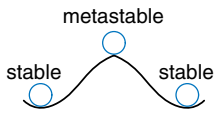


Figure 3.49 Stable and metastable states

Resolution Time

If a flip-flop input changes at a random time during the clock cycle, the resolution time, t_{res}, required to resolve to a stable state is also a random variable. If the input changes outside the aperture, then $t_{res} = t_{pcq}$. But if the input happens to change within the aperture, t_{res} can be substantially longer. Theoretical and experimental analyses (see Section 3.5.6) have shown that the probability that the resolution time, t_{res}, exceeds some arbitrary time, t, decreases exponentially with t:

$$P(t_{res} > t) = \frac{T_0}{T_c} e^{-\frac{t}{\tau}} \tag{3.23}$$

where T_c is the clock period, and T_0 and τ are characteristic of the flip-flop. The equation is valid only for t substantially longer than t_{pcq}.

Intuitively, T_0/T_c describes the probability that the input changes at a bad time (i.e., during the aperture time); this probability decreases with the cycle time, T_c. τ is a time constant indicating how fast the flip-flop moves away from the metastable state; it is related to the delay through the cross-coupled gates in the flip-flop.

In summary, if the input to a bistable device such as a flip-flop changes during the aperture time, the output may take on a metastable value for some time before resolving to a stable 0 or 1. The amount of time required to resolve is unbounded, because for any finite time, t, the probability that the flip-flop is still metastable is nonzero. However, this probability drops off exponentially as t increases. Therefore, if we wait long enough, much longer than t_{pcq}, we can expect with exceedingly high probability that the flip-flop will reach a valid logic level.

3.5.5 Synchronizers

Asynchronous inputs to digital systems from the real world are inevitable. Human input is asynchronous, for example. If handled carelessly, these asynchronous inputs can lead to metastable voltages within the system, causing erratic system failures that are extremely difficult to track down and correct. The goal of a digital system designer should be to ensure that, given asynchronous inputs, the probability of encountering a metastable voltage is sufficiently small. "Sufficiently" depends on the context. For a digital cell phone, perhaps one failure in 10 years is acceptable, because the user can always turn the phone off and back on if it locks up. For a medical device, one failure in the expected life of the universe (10^{10} years) is a better target. To guarantee good logic levels, all asynchronous inputs should be passed through *synchronizers*.

A synchronizer, shown in Figure 3.50, is a device that receives an asynchronous input, D, and a clock, CLK. It produces an output, Q, within a bounded amount of time; the output has a valid logic level with extremely high probability. If D is stable during the aperture, Q should take on the same value as D. If D changes during the aperture, Q may take on either a HIGH or LOW value but must not be metastable.

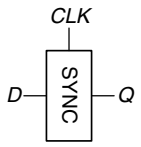


Figure 3.50 Synchronizer symbol

Figure 3.51 shows a simple way to build a synchronizer out of two flip-flops. F1 samples D on the rising edge of CLK. If D is changing at that time, the output D2 may be momentarily metastable. If the clock period is long enough, D2 will, with high probability, resolve to a valid logic level before the end of the period. F2 then samples D2, which is now stable, producing a good output Q.

We say that a synchronizer *fails* if Q, the output of the synchronizer, becomes metastable. This may happen if $D2$ has not resolved to a valid level by the time it must setup at F2—that is, if $t_{res} > T_c - t_{\text{setup}}$. According to Equation 3.23, the probability of failure for a single input change at a random time is

$$P(\text{failure}) = \frac{T_0}{T_c} e^{-\frac{T_c - t_{\text{setup}}}{\tau}} \tag{3.24}$$

The probability of failure, P(failure), is the probability that the output, Q, will be metastable upon a single change in D. If D changes once per second, the probability of failure per second is just P(failure). However, if D changes N times per second, the probability of failure per second is N times as great:

$$P(\text{failure})/\text{sec} = N \frac{T_0}{T_c} e^{-\frac{T_c - t_{\text{setup}}}{\tau}} \tag{3.25}$$

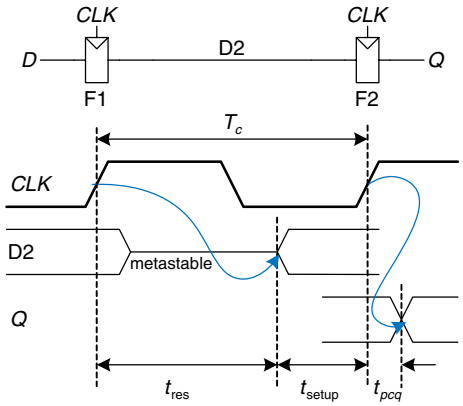


Figure 3.51 Simple synchronizer

System reliability is usually measured in *mean time between failures* (*MTBF*). As the name might suggest, MTBF is the average amount of time between failures of the system. It is the reciprocal of the probability that the system will fail in any given second

$$MTBF = \frac{1}{P(\text{failure})/\text{sec}} = \frac{T_c \, e^{\frac{T_c - t_{\text{setup}}}{\tau}}}{NT_0} \tag{3.26}$$

Equation 3.26 shows that the MTBF improves exponentially as the synchronizer waits for a longer time, T_c. For most systems, a synchronizer that waits for one clock cycle provides a safe MTBF. In exceptionally high-speed systems, waiting for more cycles may be necessary.

Example 3.13 SYNCHRONIZER FOR FSM INPUT

The traffic light controller FSM from Section 3.4.1 receives asynchronous inputs from the traffic sensors. Suppose that a synchronizer is used to guarantee stable inputs to the controller. Traffic arrives on average 0.2 times per second. The flip-flops in the synchronizer have the following characteristics: $\tau = 200$ ps, $T_0 = 150$ ps, and $t_{\text{setup}} = 500$ ps. How long must the synchronizer clock period be for the MTBF to exceed 1 year?

Solution: 1 year $\approx \pi \times 10^7$ seconds. Solve Equation 3.26.

$$\pi \times 10^7 = \frac{T_c e^{\frac{T_c - 500\times10^{-12}}{200\times10^{-12}}}}{(0.2)(150\times10^{-12})} \tag{3.27}$$

This equation has no closed form solution. However, it is easy enough to solve by guess and check. In a spreadsheet, try a few values of T_c and calculate the MTBF until discovering the value of T_c that gives an MTBF of 1 year: $T_c = 3.036$ ns.

3.5.6 Derivation of Resolution Time*

Equation 3.23 can be derived using a basic knowledge of circuit theory, differential equations, and probability. This section can be skipped if you are not interested in the derivation or if you are unfamiliar with the mathematics.

A flip-flop output will be metastable after some time, t, if the flip-flop samples a changing input (causing a metastable condition) and the output does not resolve to a valid level within that time after the clock edge. Symbolically, this can be expressed as

$$P(t_{res} > t) = P(\text{samples changing input}) \times P(\text{unresolved}) \tag{3.28}$$

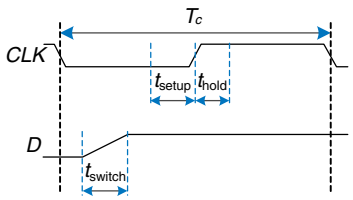


Figure 3.52 Input timing

We consider each probability term individually. The asynchronous input signal switches between 0 and 1 in some time, t_{switch}, as shown in Figure 3.52. The probability that the input changes during the aperture around the clock edge is

$$P(\text{samples changing input}) = \frac{t_{switch} + t_{setup} + t_{hold}}{T_c} \tag{3.29}$$

If the flip-flop does enter metastability—that is, with probability P(samples changing input)—the time to resolve from metastability depends on the inner workings of the circuit. This resolution time determines P(unresolved), the probability that the flip-flop has not yet resolved to a valid logic level after a time t. The remainder of this section analyzes a simple model of a bistable device to estimate this probability.

A bistable device uses storage with positive feedback. Figure 3.53(a) shows this feedback implemented with a pair of inverters; this circuit's behavior is representative of most bistable elements. A pair of inverters behaves like a buffer. Let us model it as having the symmetric DC transfer characteristics shown in Figure 3.53(b), with a slope of G. The buffer can deliver only a finite amount of output current; we can model this as an output resistance, R. All real circuits also have some capacitance, C, that must be charged up. Charging the capacitor through the resistor causes an RC delay, preventing the buffer from switching instantaneously. Hence, the complete circuit model is shown in Figure 3.53(c), where $v_{out}(t)$ is the voltage of interest conveying the state of the bistable device.

The metastable point for this circuit is $v_{out}(t) = v_{in}(t) = V_{DD}/2$; if the circuit began at exactly that point, it would remain there indefinitely in the

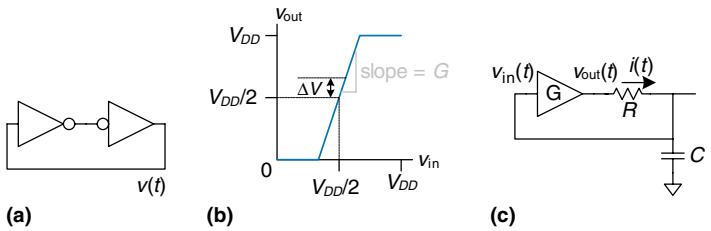


Figure 3.53 Circuit model of bistable device

absence of noise. Because voltages are continuous variables, the chance that the circuit will begin at exactly the metastable point is vanishingly small. However, the circuit might begin at time 0 near metastability at $v_{out}(0) = V_{DD}/2 + \Delta V$ for some small offset ΔV. In such a case, the positive feedback will eventually drive $v_{out}(t)$ to V_{DD} if $\Delta V > 0$ and to 0 if $\Delta V < 0$. The time required to reach V_{DD} or 0 is the resolution time of the bistable device.

The DC transfer characteristic is nonlinear, but it appears linear near the metastable point, which is the region of interest to us. Specifically, if $v_{in}(t) = V_{DD}/2 + \Delta V/G$, then $v_{out}(t) = V_{DD}/2 + \Delta V$ for small ΔV. The current through the resistor is $i(t) = (v_{out}(t) - v_{in}(t))/R$. The capacitor charges at a rate $dv_{in}(t)/dt = i(t)/C$. Putting these facts together, we find the governing equation for the output voltage.

$$\frac{dv_{out}(t)}{dt} = \frac{(G-1)}{RC}\left[v_{out}(t) - \frac{V_{DD}}{2}\right] \tag{3.30}$$

This is a linear first-order differential equation. Solving it with the initial condition $v_{out}(0) = V_{DD}/2 + \Delta V$ gives

$$v_{out}(t) = \frac{V_{DD}}{2} + \Delta V e^{\frac{(G-1)t}{RC}} \tag{3.31}$$

Figure 3.54 plots trajectories for $v_{out}(t)$ given various starting points. $v_{out}(t)$ moves exponentially away from the metastable point $V_{DD}/2$ until it saturates at V_{DD} or 0. The output voltage eventually resolves to 1 or 0. The amount of time this takes depends on the initial voltage offset (ΔV) from the metastable point ($V_{DD}/2$).

Solving Equation 3.31 for the resolution time t_{res}, such that $v_{out}(t_{res}) = V_{DD}$ or 0, gives

$$|\Delta V| e^{\frac{(G-1)t_{res}}{RC}} = \frac{V_{DD}}{2} \tag{3.32}$$

$$t_{res} = \frac{RC}{G-1} \ln \frac{V_{DD}}{2|\Delta V|} \tag{3.33}$$

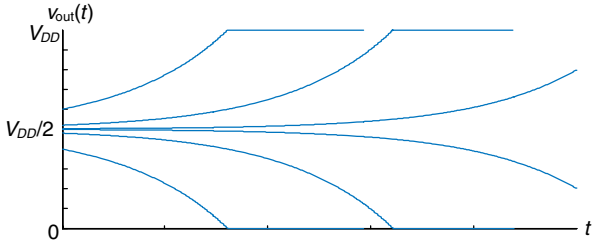


Figure 3.54 Resolution trajectories

In summary, the resolution time increases if the bistable device has high resistance or capacitance that causes the output to change slowly. It decreases if the bistable device has high *gain*, G. The resolution time also increases logarithmically as the circuit starts closer to the metastable point ($\Delta V \to 0$).

Define τ as $\frac{RC}{G-1}$. Solving Equation 3.33 for ΔV finds the initial offset, ΔV_{res}, that gives a particular resolution time, t_{res}:

$$\Delta V_{res} = \frac{V_{DD}}{2} e^{-t_{res}/\tau} \tag{3.34}$$

Suppose that the bistable device samples the input while it is changing. It measures a voltage, $v_{\text{in}}(0)$, which we will assume is uniformly distributed between 0 and V_{DD}. The probability that the output has not resolved to a legal value after time t_{res} depends on the probability that the initial offset is sufficiently small. Specifically, the initial offset on v_{out} must be less than ΔV_{res}, so the initial offset on v_{in} must be less than $\Delta V_{res}/G$. Then the probability that the bistable device samples the input at a time to obtain a sufficiently small initial offset is

$$P(\text{unresolved}) = P\left(\left| v_{\text{in}}(0) - \frac{V_{DD}}{2} \right| < \frac{\Delta V_{res}}{G}\right) = \frac{2\Delta V_{res}}{G V_{DD}} \tag{3.35}$$

Putting this all together, the probability that the resolution time exceeds some time, t, is given by the following equation:

$$P\left(t_{res} > t\right) = \frac{t_{\text{switch}} + t_{\text{setup}} + t_{\text{hold}}}{G T_c} e^{-\frac{t}{\tau}} \tag{3.36}$$

Observe that Equation 3.36 is in the form of Equation 3.23, where $T_0 = (t_{\text{switch}} + t_{\text{setup}} + t_{\text{hold}})/G$ and $\tau = RC/(G - 1)$. In summary, we have derived Equation 3.23 and shown how T_0 and τ depend on physical properties of the bistable device.

3.6 PARALLELISM

The speed of a system is measured in latency and throughput of tokens moving through a system. We define a *token* to be a group of inputs that are processed to produce a group of outputs. The term conjures up the notion of placing subway tokens on a circuit diagram and moving them around to visualize data moving through the circuit. The *latency* of a system is the time required for one token to pass through the system from start to end. The *throughput* is the number of tokens that can be produced per unit time.

Example 3.14 COOKIE THROUGHPUT AND LATENCY

Ben Bitdiddle is throwing a milk and cookies party to celebrate the installation of his traffic light controller. It takes him 5 minutes to roll cookies and place them on his tray. It then takes 15 minutes for the cookies to bake in the oven. Once the cookies are baked, he starts another tray. What is Ben's throughput and latency for a tray of cookies?

Solution: In this example, a tray of cookies is a token. The latency is 1/3 hour per tray. The throughput is 3 trays/hour.

As you might imagine, the throughput can be improved by processing several tokens at the same time. This is called *parallelism,* and it comes in two forms: spatial and temporal. With *spatial parallelism,* multiple copies of the hardware are provided so that multiple tasks can be done at the same time. With *temporal parallelism,* a task is broken into stages, like an assembly line. Multiple tasks can be spread across the stages. Although each task must pass through all stages, a *different* task will be in each stage at any given time so multiple tasks can overlap. Temporal parallelism is commonly called *pipelining*. Spatial parallelism is sometimes just called parallelism, but we will avoid that naming convention because it is ambiguous.

Example 3.15 COOKIE PARALLELISM

Ben Bitdiddle has hundreds of friends coming to his party and needs to bake cookies faster. He is considering using spatial and/or temporal parallelism.

Spatial Parallelism: Ben asks Alyssa P. Hacker to help out. She has her own cookie tray and oven.

Temporal Parallelism: Ben gets a second cookie tray. Once he puts one cookie tray in the oven, he starts rolling cookies on the other tray rather than waiting for the first tray to bake.

What is the throughput and latency using spatial parallelism? Using temporal parallelism? Using both?

Solution: The latency is the time required to complete one task from start to finish. In all cases, the latency is 1/3 hour. If Ben starts with no cookies, the latency is the time needed for him to produce the first cookie tray.

The throughput is the number of cookie trays per hour. With spatial parallelism, Ben and Alyssa each complete one tray every 20 minutes. Hence, the throughput doubles, to 6 trays/hour. With temporal parallelism, Ben puts a new tray in the oven every 15 minutes, for a throughput of 4 trays/hour. These are illustrated in Figure 3.55.

If Ben and Alyssa use both techniques, they can bake 8 trays/hour.

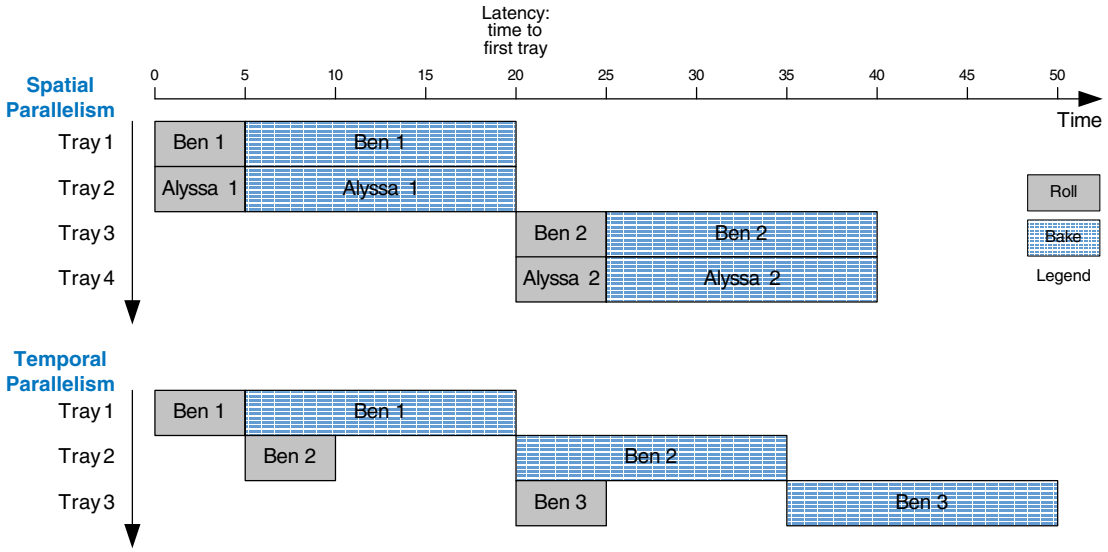


Figure 3.55 Spatial and temporal parallelism in the cookie kitchen

Consider a task with latency L. In a system with no parallelism, the throughput is $1/L$. In a spatially parallel system with N copies of the hardware, the throughput is N/L. In a temporally parallel system, the task is ideally broken into N steps, or stages, of equal length. In such a case, the throughput is also N/L, and only one copy of the hardware is required. However, as the cookie example showed, finding N steps of equal length is often impractical. If the longest step has a latency L_1, the pipelined throughput is $1/L_1$.

Pipelining (temporal parallelism) is particularly attractive because it speeds up a circuit without duplicating the hardware. Instead, registers are placed between blocks of combinational logic to divide the logic into shorter stages that can run with a faster clock. The registers prevent a token in one pipeline stage from catching up with and corrupting the token in the next stage.

Figure 3.56 shows an example of a circuit with no pipelining. It contains four blocks of logic between the registers. The critical path passes through blocks 2, 3, and 4. Assume that the register has a clock-to-Q propagation delay of 0.3 ns and a setup time of 0.2 ns. Then the cycle time is $T_c = 0.3 + 3 + 2 + 4 + 0.2 = 9.5$ ns. The circuit has a latency of 9.5 ns and a throughput of 1/9.5 ns = 105 MHz.

Figure 3.57 shows the same circuit partitioned into a two-stage pipeline by adding a register between blocks 3 and 4. The first stage has a minimum clock period of $0.3 + 3 + 2 + 0.2 = 5.5$ ns. The second

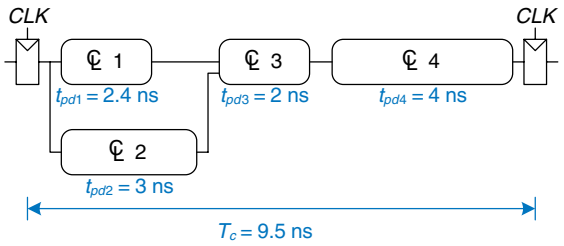


Figure 3.56 Circuit with no pipelining

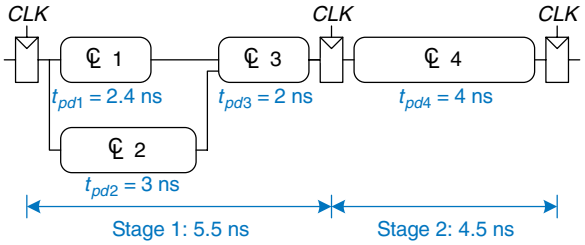


Figure 3.57 Circuit with two-stage pipeline

stage has a minimum clock period of 0.3 + 4 + 0.2 = 4.5 ns. The clock must be slow enough for all stages to work. Hence, T_c = 5.5 ns. The latency is two clock cycles, or 11 ns. The throughput is 1/5.5 ns = 182 MHz. This example shows that, in a real circuit, pipelining with two stages almost doubles the throughput and slightly increases the latency. In comparison, ideal pipelining would exactly double the throughput at no penalty in latency. The discrepancy comes about because the circuit cannot be divided into two exactly equal halves and because the registers introduce more sequencing overhead.

Figure 3.58 shows the same circuit partitioned into a three-stage pipeline. Note that two more registers are needed to store the results of blocks 1 and 2 at the end of the first pipeline stage. The cycle time is now limited by the third stage to 4.5 ns. The latency is three cycles, or 13.5 ns. The throughput is 1/4.5 ns = 222 MHz. Again, adding a pipeline stage improves throughput at the expense of some latency.

Although these techniques are powerful, they do not apply to all situations. The bane of parallelism is *dependencies*. If a current task is

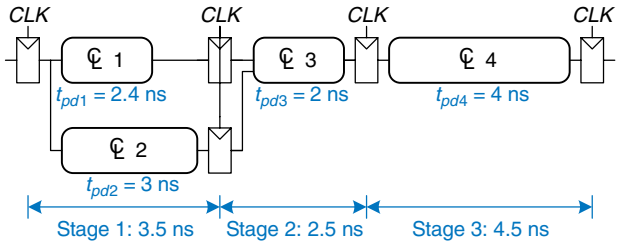


Figure 3.58 Circuit with three-stage pipeline

dependent on the result of a prior task, rather than just prior steps in the current task, the task cannot start until the prior task has completed. For example, if Ben wants to check that the first tray of cookies tastes good before he starts preparing the second, he has a dependency that prevents pipelining or parallel operation. Parallelism is one of the most important techniques for designing high-performance microprocessors. Chapter 7 discusses pipelining further and shows examples of handling dependencies.

3.7 SUMMARY

Anyone who could invent logic whose outputs depend on future inputs would be fabulously wealthy!

This chapter has described the analysis and design of sequential logic. In contrast to combinational logic, whose outputs depend only on the current inputs, sequential logic outputs depend on both current and prior inputs. In other words, sequential logic remembers information about prior inputs. This memory is called the state of the logic.

Sequential circuits can be difficult to analyze and are easy to design incorrectly, so we limit ourselves to a small set of carefully designed building blocks. The most important element for our purposes is the flip-flop, which receives a clock and an input, D, and produces an output, Q. The flip-flop copies D to Q on the rising edge of the clock and otherwise remembers the old state of Q. A group of flip-flops sharing a common clock is called a register. Flip-flops may also receive reset or enable control signals.

Although many forms of sequential logic exist, we discipline ourselves to use synchronous sequential circuits because they are easy to design. Synchronous sequential circuits consist of blocks of combinational logic separated by clocked registers. The state of the circuit is stored in the registers and updated only on clock edges.

Finite state machines are a powerful technique for designing sequential circuits. To design an FSM, first identify the inputs and outputs of the machine and sketch a state transition diagram, indicating the states and the transitions between them. Select an encoding for the states, and rewrite the diagram as a state transition table and output table, indicating the next state and output given the current state and input. From these tables, design the combinational logic to compute the next state and output, and sketch the circuit.

Synchronous sequential circuits have a timing specification including the clock-to-Q propagation and contamination delays, t_{pcq} and t_{ccq}, and the setup and hold times, t_{setup} and t_{hold}. For correct operation, their inputs must be stable during an aperture time that starts a setup time before the rising edge of the clock and ends a hold time after the rising edge of the clock. The minimum cycle time, T_c, of the system is equal to the propagation delay, t_{pd}, through the combinational logic plus

$t_{pcq} + t_{\text{setup}}$ of the register. For correct operation, the contamination delay through the register and combinational logic must be greater than t_{hold}. Despite the common misconception to the contrary, hold time does not affect the cycle time.

Overall system performance is measured in latency and throughput. The latency is the time required for a token to pass from start to end. The throughput is the number of tokens that the system can process per unit time. Parallelism improves the system throughput.

Exercises

Exercise 3.1 Given the input waveforms shown in Figure 3.59, sketch the output, Q, of an SR latch.

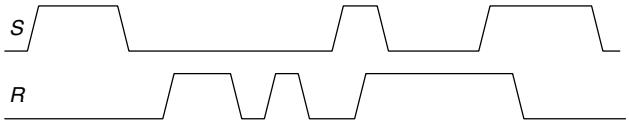


Figure 3.59 Input waveform of SR latch

Exercise 3.2 Given the input waveforms shown in Figure 3.60, sketch the output, Q, of a D latch.

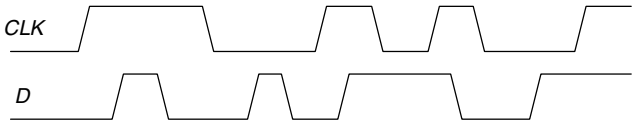


Figure 3.60 Input waveform of D latch or flip-flop

Exercise 3.3 Given the input waveforms shown in Figure 3.60, sketch the output, Q, of a D flip-flop.

Exercise 3.4 Is the circuit in Figure 3.61 combinational logic or sequential logic? Explain in a simple fashion what the relationship is between the inputs and outputs. What would you call this circuit?

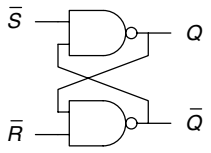


Figure 3.61 Mystery circuit

Exercise 3.5 Is the circuit in Figure 3.62 combinational logic or sequential logic? Explain in a simple fashion what the relationship is between the inputs and outputs. What would you call this circuit?

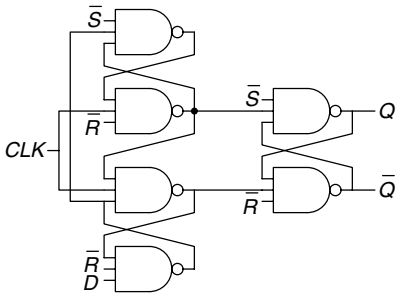


Figure 3.62 Mystery circuit

Exercise 3.6 The *toggle* (*T*) *flip-flop* has one input, *CLK*, and one output, *Q*. On each rising edge of *CLK*, *Q* toggles to the complement of its previous value. Draw a schematic for a T flip-flop using a D flip-flop and an inverter.

Exercise 3.7 A *JK flip-flop* receives a clock and two inputs, *J* and *K*. On the rising edge of the clock, it updates the output, *Q*. If *J* and *K* are both 0, *Q* retains its old value. If only *J* is 1, *Q* becomes 1. If only *K* is 1, *Q* becomes 0. If both *J* and *K* are 1, *Q* becomes the opposite of its present state.

(a) Construct a JK flip-flop using a D flip-flop and some combinational logic.

(b) Construct a D flip-flop using a JK flip-flop and some combinational logic.

(c) Construct a T flip-flop (see Exercise 3.6) using a JK flip-flop.

Exercise 3.8 The circuit in Figure 3.63 is called a *Muller C-element*. Explain in a simple fashion what the relationship is between the inputs and output.

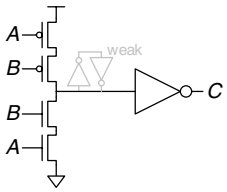


Figure 3.63 Muller C-element

Exercise 3.9 Design an asynchronously resettable D latch using logic gates.

Exercise 3.10 Design an asynchronously resettable D flip-flop using logic gates.

Exercise 3.11 Design a synchronously settable D flip-flop using logic gates.

Exercise 3.12 Design an asynchronously settable D flip-flop using logic gates.

Exercise 3.13 Suppose a ring oscillator is built from N inverters connected in a loop. Each inverter has a minimum delay of t_{cd} and a maximum delay of t_{pd}. If N is odd, determine the range of frequencies at which the oscillator might operate.

Exercise 3.14 Why must N be odd in Exercise 3.13?

Exercise 3.15 Which of the circuits in Figure 3.64 are synchronous sequential circuits? Explain.

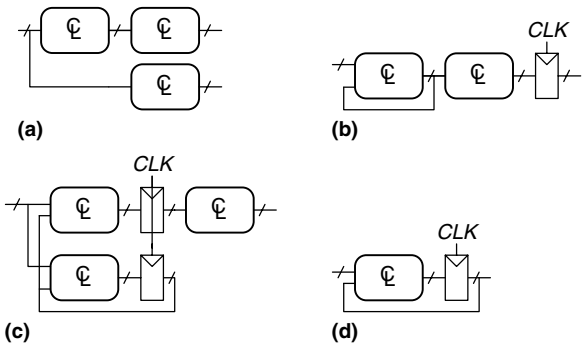


Figure 3.64 Circuits

Exercise 3.16 You are designing an elevator controller for a building with 25 floors. The controller has two inputs: *UP* and *DOWN*. It produces an output indicating the floor that the elevator is on. There is no floor 13. What is the minimum number of bits of state in the controller?

Exercise 3.17 You are designing an FSM to keep track of the mood of four students working in the digital design lab. Each student's mood is either HAPPY (the circuit works), SAD (the circuit blew up), BUSY (working on the circuit), CLUELESS (confused about the circuit), or ASLEEP (face down on the circuit board). How many states does the FSM have? What is the minimum number of bits necessary to represent these states?

Exercise 3.18 How would you factor the FSM from Exercise 3.17 into multiple simpler machines? How many states does each simpler machine have? What is the minimum total number of bits necessary in this factored design?

Exercise 3.19 Describe in words what the state machine in Figure 3.65 does. Using binary state encodings, complete a state transition table and output table for the FSM. Write Boolean equations for the next state and output and sketch a schematic of the FSM.

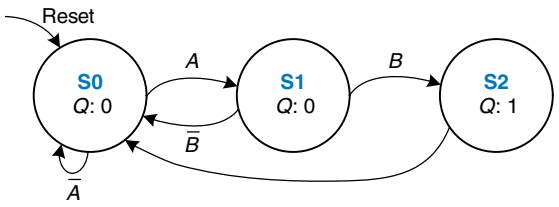


Figure 3.65 State transition diagram

Exercise 3.20 Describe in words what the state machine in Figure 3.66 does. Using binary state encodings, complete a state transition table and output table for the FSM. Write Boolean equations for the next state and output and sketch a schematic of the FSM.

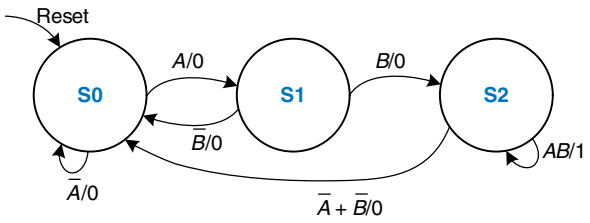


Figure 3.66 State transition diagram

Exercise 3.21 Accidents are still occurring at the intersection of Academic Avenue and Bravado Boulevard. The football team is rushing into the intersection the moment light *B* turns green. They are colliding with sleep-deprived CS majors who stagger into the intersection just before light *A* turns red. Extend the traffic light controller from Section 3.4.1 so that both lights are red for 5 seconds before either light turns green again. Sketch your improved Moore machine state transition diagram, state encodings, state transition table, output table, next state and output equations, and your FSM schematic.

Exercise 3.22 Alyssa P. Hacker's snail from Section 3.4.3 has a daughter with a Mealy machine FSM brain. The daughter snail smiles whenever she slides over the pattern 1101 or the pattern 1110. Sketch the state transition diagram for this happy snail using as few states as possible. Choose state encodings and write a

combined state transition and output table using your encodings. Write the next state and output equations and sketch your FSM schematic.

Exercise 3.23 You have been enlisted to design a soda machine dispenser for your department lounge. Sodas are partially subsidized by the student chapter of the IEEE, so they cost only 25 cents. The machine accepts nickels, dimes, and quarters. When enough coins have been inserted, it dispenses the soda and returns any necessary change. Design an FSM controller for the soda machine. The FSM inputs are *Nickel*, *Dime*, and *Quarter*, indicating which coin was inserted. Assume that exactly one coin is inserted on each cycle. The outputs are *Dispense*, *ReturnNickel*, *ReturnDime*, and *ReturnTwoDimes*. When the FSM reaches 25 cents, it asserts *Dispense* and the necessary *Return* outputs required to deliver the appropriate change. Then it should be ready to start accepting coins for another soda.

Exercise 3.24 Gray codes have a useful property in that consecutive numbers differ in only a single bit position. Table 3.17 lists a 3-bit Gray code representing the numbers 0 to 7. Design a 3-bit modulo 8 Gray code counter FSM with no inputs and three outputs. (A modulo N counter counts from 0 to $N-1$, then repeats. For example, a watch uses a modulo 60 counter for the minutes and seconds that counts from 0 to 59.) When reset, the output should be 000. On each clock edge, the output should advance to the next Gray code. After reaching 100, it should repeat with 000.

Table 3.17 3-bit Gray code

Number	Gray code		
0	0	0	0
1	0	0	1
2	0	1	1
3	0	1	0
4	1	1	0
5	1	1	1
6	1	0	1
7	1	0	0

Exercise 3.25 Extend your modulo 8 Gray code counter from Exercise 3.24 to be an UP/DOWN counter by adding an *UP* input. If $UP = 1$, the counter advances to the next number. If $UP = 0$, the counter retreats to the previous number.

Exercise 3.26 Your company, Detect-o-rama, would like to design an FSM that takes two inputs, A and B, and generates one output, Z. The output in cycle n, Z_n, is either the Boolean AND or OR of the corresponding input A_n and the previous input A_{n-1}, depending on the other input, B_n:

$$Z_n = A_n A_{n-1} \quad \text{if } B_n = 0$$
$$Z_n = A_n + A_{n-1} \text{ if } B_n = 1$$

(a) Sketch the waveform for Z given the inputs shown in Figure 3.67.

(b) Is this FSM a Moore or a Mealy machine?

(c) Design the FSM. Show your state transition diagram, encoded state transition table, next state and output equations, and schematic.

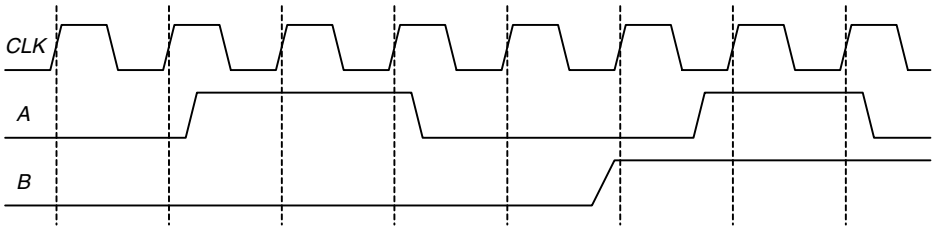


Figure 3.67 FSM input waveforms

Exercise 3.27 Design an FSM with one input, A, and two outputs, X and Y. X should be 1 if A has been 1 for at least three cycles altogether (not necessarily consecutively). Y should be 1 if A has been 1 for at least two consecutive cycles. Show your state transition diagram, encoded state transition table, next state and output equations, and schematic.

Exercise 3.28 Analyze the FSM shown in Figure 3.68. Write the state transition and output tables and sketch the state transition diagram. Describe in words what the FSM does.

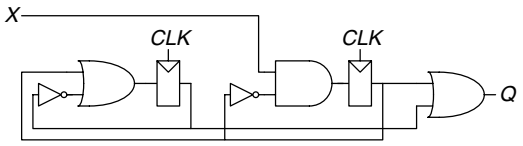


Figure 3.68 FSM schematic

Exercise 3.29 Repeat Exercise 3.28 for the FSM shown in Figure 3.69. Recall that the *r* and *s* register inputs indicate set and reset, respectively.

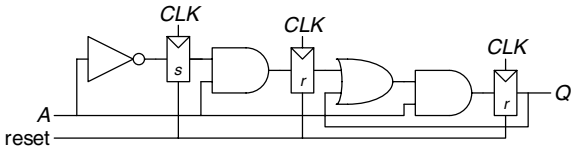


Figure 3.69 FSM schematic

Exercise 3.30 Ben Bitdiddle has designed the circuit in Figure 3.70 to compute a registered four-input XOR function. Each two-input XOR gate has a propagation delay of 100 ps and a contamination delay of 55 ps. Each flip-flop has a setup time of 60 ps, a hold time of 20 ps, a clock-to-Q maximum delay of 70 ps, and a clock-to-Q minimum delay of 50 ps.

(a) If there is no clock skew, what is the maximum operating frequency of the circuit?

(b) How much clock skew can the circuit tolerate if it must operate at 2 GHz?

(c) How much clock skew can the circuit tolerate before it might experience a hold time violation?

(d) Alyssa P. Hacker points out that she can redesign the combinational logic between the registers to be faster *and* tolerate more clock skew. Her improved circuit also uses three two-input XORs, but they are arranged differently. What is her circuit? What is its maximum frequency if there is no clock skew? How much clock skew can the circuit tolerate before it might experience a hold time violation?

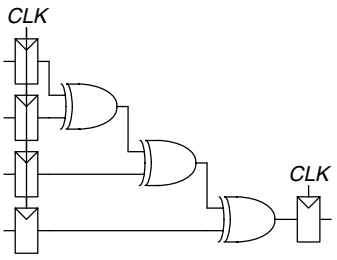


Figure 3.70 Registered four-input XOR circuit

Exercise 3.31 You are designing an adder for the blindingly fast 2-bit RePentium Processor. The adder is built from two full adders such that the carry out of the first adder is the carry in to the second adder, as shown in Figure 3.71. Your adder has input and output registers and must complete the

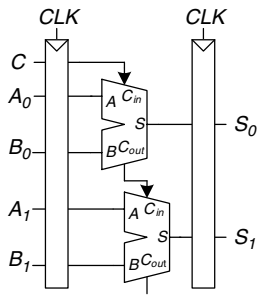


Figure 3.71 2-bit adder schematic

addition in one clock cycle. Each full adder has the following propagation delays: 20 ps from C_{in} to C_{out} or to *Sum* (*S*), 25 ps from *A* or *B* to C_{out}, and 30 ps from *A* or *B* to *S*. The adder has a contamination delay of 15 ps from C_{in} to either output and 22 ps from *A* or *B* to either output. Each flip-flop has a setup time of 30 ps, a hold time of 10 ps, a clock-to-*Q* propagation delay of 35 ps, and a clock-to-*Q* contamination delay of 21 ps.

(a) If there is no clock skew, what is the maximum operating frequency of the circuit?

(b) How much clock skew can the circuit tolerate if it must operate at 8 GHz?

(c) How much clock skew can the circuit tolerate before it might experience a hold time violation?

Exercise 3.32 A *field programmable gate array* (*FPGA*) uses *configurable logic blocks* (*CLBs*) rather than logic gates to implement combinational logic. The Xilinx Spartan 3 FPGA has propagation and contamination delays of 0.61 and 0.30 ns, respectively for each CLB. It also contains flip-flops with propagation and contamination delays of 0.72 and 0.50 ns, and setup and hold times of 0.53 and 0 ns, respectively.

(a) If you are building a system that needs to run at 40 MHz, how many consecutive CLBs can you use between two flip-flops? Assume there is no clock skew and no delay through wires between CLBs.

(b) Suppose that all paths between flip-flops pass through at least one CLB. How much clock skew can the FPGA have without violating the hold time?

Exercise 3.33 A synchronizer is built from a pair of flip-flops with $t_{setup} = 50$ ps, $T_0 = 20$ ps, and $\tau = 30$ ps. It samples an asynchronous input that changes 10^8 times per second. What is the minimum clock period of the synchronizer to achieve a mean time between failures (MTBF) of 100 years?

Exercise 3.34 You would like to build a synchronizer that can receive asynchronous inputs with an MTBF of 50 years. Your system is running at 1 GHz, and you use sampling flip-flops with $\tau = 100$ ps, $T_0 = 110$ ps, and $t_{\text{setup}} = 70$ ps. The synchronizer receives a new asynchronous input on average 0.5 times per second (i.e., once every 2 seconds). What is the required probability of failure to satisfy this MTBF? How many clock cycles would you have to wait before reading the sampled input signal to give that probability of error?

Exercise 3.35 You are walking down the hallway when you run into your lab partner walking in the other direction. The two of you first step one way and are still in each other's way. Then you both step the other way and are still in each other's way. Then you both wait a bit, hoping the other person will step aside. You can model this situation as a metastable point and apply the same theory that has been applied to synchronizers and flip-flops. Suppose you create a mathematical model for yourself and your lab partner. You start the unfortunate encounter in the metastable state. The probability that you remain in this state after t seconds is $e^{-\frac{t}{\tau}}$. τ indicates your response rate; today, your brain has been blurred by lack of sleep and has $\tau = 20$ seconds.

(a) How long will it be until you have 99% certainty that you will have resolved from metastability (i.e., figured out how to pass one another)?

(b) You are not only sleepy, but also ravenously hungry. In fact, you will starve to death if you don't get going to the cafeteria within 3 minutes. What is the probability that your lab partner will have to drag you to the morgue?

Exercise 3.36 You have built a synchronizer using flip-flops with $T_0 = 20$ ps and $\tau = 30$ ps. Your boss tells you that you need to increase the MTBF by a factor of 10. By how much do you need to increase the clock period?

Exercise 3.37 Ben Bitdiddle invents a new and improved synchronizer in Figure 3.72 that he claims eliminates metastability in a single cycle. He explains that the circuit in box *M* is an analog "metastability detector" that produces a HIGH output if the input voltage is in the forbidden zone between V_{IL} and V_{IH}. The metastability detector checks to determine

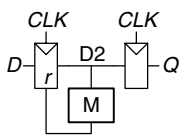


Figure 3.72 "New and improved" synchronizer

whether the first flip-flop has produced a metastable output on $D2$. If so, it asynchronously resets the flip-flop to produce a good 0 at $D2$. The second flip-flop then samples $D2$, always producing a valid logic level on Q. Alyssa P. Hacker tells Ben that there must be a bug in the circuit, because eliminating metastability is just as impossible as building a perpetual motion machine. Who is right? Explain, showing Ben's error or showing why Alyssa is wrong.

Interview Questions

The following exercises present questions that have been asked at interviews for digital design jobs.

Question 3.1 Draw a state machine that can detect when it has received the serial input sequence 01010.

Question 3.2 Design a serial (one bit at a time) two's complementer FSM with two inputs, *Start* and *A*, and one output, *Q*. A binary number of arbitrary length is provided to input *A*, starting with the least significant bit. The corresponding bit of the output appears at *Q* on the same cycle. *Start* is asserted for one cycle to initialize the FSM before the least significant bit is provided.

Question 3.3 What is the difference between a latch and a flip-flop? Under what circumstances is each one preferable?

Question 3.4 Design a 5-bit counter finite state machine.

Question 3.5 Design an edge detector circuit. The output should go HIGH for one cycle after the input makes a $0 \rightarrow 1$ transition.

Question 3.6 Describe the concept of pipelining and why it is used.

Question 3.7 Describe what it means for a flip-flop to have a negative hold time.

Question 3.8 Given signal *A*, shown in Figure 3.73, design a circuit that produces signal *B*.

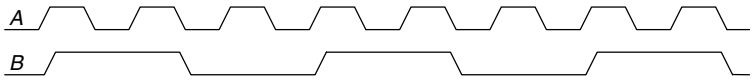


Figure 3.73 Signal waveforms

Question 3.9 Consider a block of logic between two registers. Explain the timing constraints. If you add a buffer on the clock input of the receiver (the second flip-flop), does the setup time constraint get better or worse?

Hardware Description Languages

4

4.1 INTRODUCTION

Thus far, we have focused on designing combinational and sequential digital circuits at the schematic level. The process of finding an efficient set of logic gates to perform a given function is labor intensive and error prone, requiring manual simplification of truth tables or Boolean equations and manual translation of finite state machines (FSMs) into gates. In the 1990's, designers discovered that they were far more productive if they worked at a higher level of abstraction, specifying just the logical function and allowing a *computer-aided design* (*CAD*) tool to produce the optimized gates. The specifications are generally given in a *hardware description language* (*HDL*). The two leading hardware description languages are *Verilog* and *VHDL*.

Verilog and VHDL are built on similar principles but have different syntax. Discussion of these languages in this chapter is divided into two columns for literal side-by-side comparison, with Verilog on the left and VHDL on the right. When you read the chapter for the first time, focus on one language or the other. Once you know one, you'll quickly master the other if you need it.

Subsequent chapters show hardware in both schematic and HDL form. If you choose to skip this chapter and not learn one of the HDLs, you will still be able to master the principles of computer organization from the schematics. However, the vast majority of commercial systems are now built using HDLs rather than schematics. If you expect to do digital design at any point in your professional life, we urge you to learn one of the HDLs.

4.1.1 Modules

A block of hardware with inputs and outputs is called a *module*. An AND gate, a multiplexer, and a priority circuit are all examples of hardware modules. The two general styles for describing module functionality are

behavioral and *structural*. Behavioral models describe what a module does. Structural models describe how a module is built from simpler pieces; it is an application of hierarchy. The Verilog and VHDL code in HDL Example 4.1 illustrate behavioral descriptions of a module that computes the Boolean function from Example 2.6, $y = \bar{a}\bar{b}\bar{c} + a\bar{b}\bar{c} + a\bar{b}c$. In both languages, the module is named `sillyfunction` and has three inputs, `a`, `b`, and `c`, and one output, `y`.

HDL Example 4.1 COMBINATIONAL LOGIC

Verilog

```
module sillyfunction(input a, b, c,
                     output y);

  assign y = ~a & ~b & ~c |
             a & ~b & ~c |
             a & ~b &  c;

endmodule
```

A Verilog module begins with the module name and a listing of the inputs and outputs. The `assign` statement describes combinational logic. `~` indicates NOT, `&` indicates AND, and `|` indicates OR.

Verilog signals such as the inputs and outputs are Boolean variables (0 or 1). They may also have floating and undefined values, as discussed in Section 4.2.8.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity sillyfunction is
  port(a, b, c: in  STD_LOGIC;
       y:       out STD_LOGIC);
end;

architecture synth of sillyfunction is
begin
  y <= ((not a) and (not b) and (not c)) or
       (a and (not b) and (not c)) or
       (a and (not b) and c);
end;
```

VHDL code has three parts: the `library` use clause, the `entity` declaration, and the `architecture` body. The `library` use clause is required and will be discussed in Section 4.2.11. The `entity` declaration lists the module name and its inputs and outputs. The `architecture` body defines what the module does.

VHDL signals, such as inputs and outputs, must have a *type declaration*. Digital signals should be declared to be `STD_LOGIC` type. `STD_LOGIC` signals can have a value of '0' or '1', as well as floating and undefined values that will be described in Section 4.2.8. The `STD_LOGIC` type is defined in the `IEEE.STD_LOGIC_1164` library, which is why the library must be used.

VHDL lacks a good default order of operations, so Boolean equations should be parenthesized.

A module, as you might expect, is a good application of modularity. It has a well defined interface, consisting of its inputs and outputs, and it performs a specific function. The particular way in which it is coded is unimportant to others that might use the module, as long as it performs its function.

4.1.2 Language Origins

Universities are almost evenly split on which of these languages is taught in a first course, and industry is similarly split on which language is preferred. Compared to Verilog, VHDL is more verbose and cumbersome,

Verilog

Verilog was developed by Gateway Design Automation as a proprietary language for logic simulation in 1984. Gateway was acquired by Cadence in 1989 and Verilog was made an open standard in 1990 under the control of Open Verilog International. The language became an IEEE standard[1] in 1995 (IEEE STD 1364) and was updated in 2001.

VHDL

VHDL is an acronym for the *VHSIC Hardware Description Language*. VHSIC is in turn an acronym for the *Very High Speed Integrated Circuits* program of the US Department of Defense.

VHDL was originally developed in 1981 by the Department of Defense to describe the structure and function of hardware. Its roots draw from the Ada programming language. The IEEE standardized it in 1987 (IEEE STD 1076) and has updated the standard several times since. The language was first envisioned for documentation but was quickly adopted for simulation and synthesis.

as you might expect of a language developed by committee. U.S. military contractors, the European Space Agency, and telecommunications companies use VHDL extensively.

Both languages are fully capable of describing any hardware system, and both have their quirks. The best language to use is the one that is already being used at your site or the one that your customers demand. Most CAD tools today allow the two languages to be mixed, so that different modules can be described in different languages.

4.1.3 Simulation and Synthesis

The two major purposes of HDLs are logic *simulation* and *synthesis*. During simulation, inputs are applied to a module, and the outputs are checked to verify that the module operates correctly. During synthesis, the textual description of a module is transformed into logic gates.

Simulation

Humans routinely make mistakes. Such errors in hardware designs are called *bugs*. Eliminating the bugs from a digital system is obviously important, especially when customers are paying money and lives depend on the correct operation. Testing a system in the laboratory is time-consuming. Discovering the cause of errors in the lab can be extremely difficult, because only signals routed to the chip pins can be observed. There is no way to directly observe what is happening inside a chip. Correcting errors after the system is built can be devastatingly expensive. For example, correcting a mistake in a cutting-edge integrated circuit costs more than a million dollars and takes several months. Intel's infamous FDIV (floating point division) bug in the Pentium processor forced the company to recall chips after they had shipped, at a total cost of $475 million. Logic simulation is essential to test a system before it is built.

The term "bug" predates the invention of the computer. Thomas Edison called the "little faults and difficulties" with his inventions "bugs" in 1878.

The first real computer bug was a moth, which got caught between the relays of the Harvard Mark II electromechanical computer in 1947. It was found by Grace Hopper, who logged the incident, along with the moth itself and the comment "first actual case of bug being found."

Source: Notebook entry courtesy Naval Historical Center, US Navy; photo No. NII 96566-KN

[1] The Institute of Electrical and Electronics Engineers (IEEE) is a professional society responsible for many computing standards including WiFi (802.11), Ethernet (802.3), and floating-point numbers (754) (see Chapter 5).

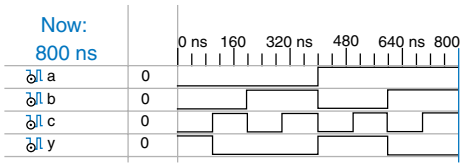


Figure 4.1 Simulation waveforms

Figure 4.1 shows waveforms from a simulation[2] of the previous `sillyfunction` module demonstrating that the module works correctly. `y` is TRUE when `a`, `b`, and `c` are 000, 100, or 101, as specified by the Boolean equation.

Synthesis

Logic synthesis transforms HDL code into a *netlist* describing the hardware (e.g., the logic gates and the wires connecting them). The logic synthesizer might perform optimizations to reduce the amount of hardware required. The netlist may be a text file, or it may be drawn as a schematic to help visualize the circuit. Figure 4.2 shows the results of synthesizing the `sillyfunction` module.[3] Notice how the three three-input AND gates are simplified into two two-input AND gates, as we discovered in Example 2.6 using Boolean algebra.

Circuit descriptions in HDL resemble code in a programming language. However, you must remember that the code is intended to represent hardware. Verilog and VHDL are rich languages with many commands. Not all of these commands can be synthesized into hardware. For example, a command to print results on the screen during simulation does not translate into hardware. Because our primary interest is

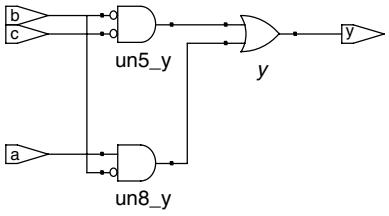


Figure 4.2 Synthesized circuit

[2] The simulation was performed with the Xilinx ISE Simulator, which is part of the Xilinx ISE 8.2 software. The simulator was selected because it is used commercially, yet is freely available to universities.

[3] Synthesis was performed with Synplify Pro from Synplicity. The tool was selected because it is the leading commercial tool for synthesizing HDL to field-programmable gate arrays (see Section 5.6.2) and because it is available inexpensively for universities.

to build hardware, we will emphasize a *synthesizable subset* of the languages. Specifically, we will divide HDL code into *synthesizable* modules and a *testbench*. The synthesizable modules describe the hardware. The testbench contains code to apply inputs to a module, check whether the output results are correct, and print discrepancies between expected and actual outputs. Testbench code is intended only for simulation and cannot be synthesized.

One of the most common mistakes for beginners is to think of HDL as a computer program rather than as a shorthand for describing digital hardware. If you don't know approximately what hardware your HDL should synthesize into, you probably won't like what you get. You might create far more hardware than is necessary, or you might write code that simulates correctly but cannot be implemented in hardware. Instead, think of your system in terms of blocks of combinational logic, registers, and finite state machines. Sketch these blocks on paper and show how they are connected before you start writing code.

In our experience, the best way to learn an HDL is by example. HDLs have specific ways of describing various classes of logic; these ways are called *idioms*. This chapter will teach you how to write the proper HDL idioms for each type of block and then how to put the blocks together to produce a working system. When you need to describe a particular kind of hardware, look for a similar example and adapt it to your purpose. We do not attempt to rigorously define all the syntax of the HDLs, because that is deathly boring and because it tends to encourage thinking of HDLs as programming languages, not shorthand for hardware. The IEEE Verilog and VHDL specifications, and numerous dry but exhaustive textbooks, contain all of the details, should you find yourself needing more information on a particular topic. (See Further Readings section at back of the book.)

4.2 COMBINATIONAL LOGIC

Recall that we are disciplining ourselves to design synchronous sequential circuits, which consist of combinational logic and registers. The outputs of combinational logic depend only on the current inputs. This section describes how to write behavioral models of combinational logic with HDLs.

4.2.1 Bitwise Operators

Bitwise operators act on single-bit signals or on multi-bit busses. For example, the `inv` module in HDL Example 4.2 describes four inverters connected to 4-bit busses.

HDL Example 4.2 INVERTERS

Verilog

```
module inv(input  [3:0] a,
           output [3:0] y);

  assign y = ~a;
endmodule
```

`a[3:0]` represents a 4-bit bus. The bits, from most significant to least significant, are `a[3]`, `a[2]`, `a[1]`, and `a[0]`. This is called *little-endian* order, because the least significant bit has the smallest bit number. We could have named the bus `a[4:1]`, in which case `a[4]` would have been the most significant. Or we could have used `a[0:3]`, in which case the bits, from most significant to least significant, would be `a[0]`, `a[1]`, `a[2]`, and `a[3]`. This is called *big-endian* order.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity inv is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of inv is
begin
  y <= not a;
end;
```

VHDL uses `STD_LOGIC_VECTOR`, to indicate busses of `STD_LOGIC`. `STD_LOGIC_VECTOR (3 downto 0)` represents a 4-bit bus. The bits, from most significant to least significant, are 3, 2, 1, and 0. This is called *little-endian* order, because the least significant bit has the smallest bit number. We could have declared the bus to be `STD_LOGIC_VECTOR (4 downto 1)`, in which case bit 4 would have been the most significant. Or we could have written `STD_LOGIC_VECTOR (0 to 3)`, in which case the bits, from most significant to least significant, would be 0, 1, 2, and 3. This is called *big-endian* order.

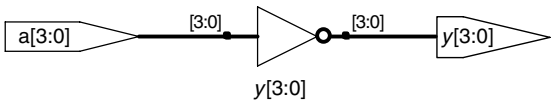


Figure 4.3 `inv` **synthesized circuit**

The endianness of a bus is purely arbitrary. (See the sidebar in Section 6.2.2 for the origin of the term.) Indeed, endianness is also irrelevant to this example, because a bank of inverters doesn't care what the order of the bits are. Endianness matters only for operators, such as addition, where the sum of one column carries over into the next. Either ordering is acceptable, as long as it is used consistently. We will consistently use the little-endian order, `[N-1:0]` in Verilog and `(N-1 downto 0)` in VHDL, for an *N*-bit bus.

After each code example in this chapter is a schematic produced from the Verilog code by the Synplify Pro synthesis tool. Figure 4.3 shows that the `inv` module synthesizes to a bank of four inverters, indicated by the inverter symbol labeled `y[3:0]`. The bank of inverters connects to 4-bit input and output busses. Similar hardware is produced from the synthesized VHDL code.

The `gates` module in HDL Example 4.3 demonstrates bitwise operations acting on 4-bit busses for other basic logic functions.

HDL Example 4.3 LOGIC GATES

Verilog

```
module gates(input  [3:0] a, b,
             output [3:0] y1, y2,
                          y3, y4, y5);

  /* Five different two-input logic
     gates acting on 4 bit busses */
  assign y1 = a & b;    // AND
  assign y2 = a | b;    // OR
  assign y3 = a ^ b;    // XOR
  assign y4 = ~(a & b); // NAND
  assign y5 = ~(a | b); // NOR
endmodule
```

~, ^, and | are examples of Verilog *operators*, whereas a, b, and y1 are *operands*. A combination of operators and operands, such as a & b, or ~(a | b), is called an *expression*. A complete command such as assign y4 = ~(a & b); is called a *statement*.

assign out = in1 op in2; is called a *continuous assignment statement*. Continuous assignment statements end with a semicolon. Anytime the inputs on the right side of the = in a continuous assignment statement change, the output on the left side is recomputed. Thus, continuous assignment statements describe combinational logic.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity gates is
  port(a, b: in  STD_LOGIC_VECTOR(3 downto 0);
       y1, y2, y3, y4,
       y5:   out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of gates is
begin
  -- Five different two-input logic gates
  --acting on 4 bit busses
  y1 <= a and b;
  y2 <= a or b;
  y3 <= a xor b;
  y4 <= a nand b;
  y5 <= a nor b;
end;
```

not, xor, and or are examples of VHDL *operators*, whereas a, b, and y1 are *operands*. A combination of operators and operands, such as a and b, or a nor b, is called an *expression*. A complete command such as y4 <= a nand b; is called a *statement*.

out <= in1 op in2; is called a *concurrent signal assignment statement*. VHDL assignment statements end with a semicolon. Anytime the inputs on the right side of the <= in a concurrent signal assignment statement change, the output on the left side is recomputed. Thus, concurrent signal assignment statements describe combinational logic.

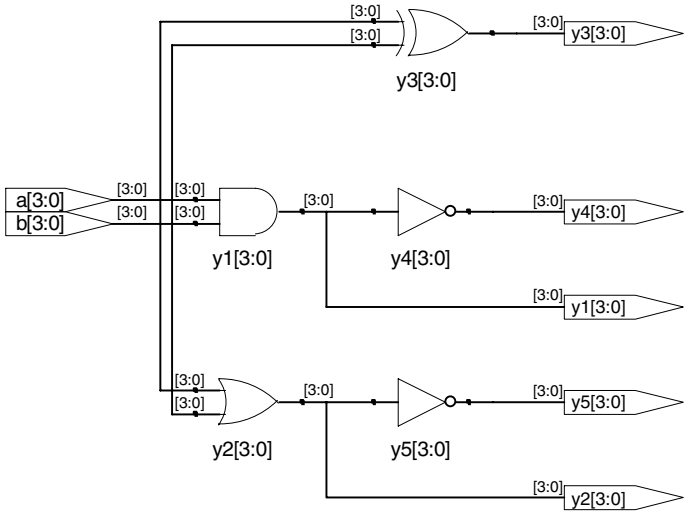


Figure 4.4 gates **synthesized circuit**

4.2.2 Comments and White Space

The `gates` example showed how to format comments. Verilog and VHDL are not picky about the use of white space (i.e., spaces, tabs, and line breaks). Nevertheless, proper indenting and use of blank lines is helpful to make nontrivial designs readable. Be consistent in your use of capitalization and underscores in signal and module names. Module and signal names must not begin with a digit.

Verilog

Verilog comments are just like those in C or Java. Comments beginning with /* continue, possibly across multiple lines, to the next */. Comments beginning with // continue to the end of the line.

Verilog is case-sensitive. `y1` and `Y1` are different signals in Verilog.

VHDL

VHDL comments begin with -- and continue to the end of the line. Comments spanning multiple lines must use -- at the beginning of each line.

VHDL is not case-sensitive. `y1` and `Y1` are the same signal in VHDL. However, other tools that may read your file might be case sensitive, leading to nasty bugs if you blithely mix upper and lower case.

4.2.3 Reduction Operators

Reduction operators imply a multiple-input gate acting on a single bus. HDL Example 4.4 describes an eight-input AND gate with inputs a_7, $a_6, \ldots, a_0$.

HDL Example 4.4 EIGHT-INPUT AND

Verilog

```
module and8(input [7:0] a,
            output      y);

  assign y = &a;

  // &a is much easier to write than
  // assign y = a[7] & a[6] & a[5] & a[4] &
  //            a[3] & a[2] & a[1] & a[0];
endmodule
```

As one would expect, |, ^, ~&, and ~ | reduction operators are available for OR, XOR, NAND, and NOR as well. Recall that a multi-input XOR performs parity, returning TRUE if an odd number of inputs are TRUE.

VHDL

VHDL does not have reduction operators. Instead, it provides the `generate` command (see Section 4.7). Alternatively, the operation can be written explicitly, as shown below.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity and8 is
  port(a: in  STD_LOGIC_VECTOR(7 downto 0);
       y: out STD_LOGIC);
end;

architecture synth of and8 is
begin
  y <= a(7) and a(6) and a(5) and a(4) and
       a(3) and a(2) and a(1) and a(0);
end;
```

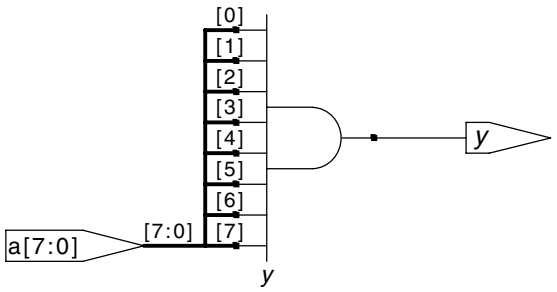


Figure 4.5 `and8` **synthesized circuit**

4.2.4 Conditional Assignment

Conditional assignments select the output from among alternatives based on an input called the *condition*. HDL Example 4.5 illustrates a 2:1 multiplexer using conditional assignment.

HDL Example 4.5 2:1 MULTIPLEXER

Verilog

The *conditional operator* `?:` chooses, based on a first expression, between a second and third expression. The first expression is called the *condition*. If the condition is 1, the operator chooses the second expression. If the condition is 0, the operator chooses the third expression.

`?:` is especially useful for describing a multiplexer because, based on the first input, it selects between two others. The following code demonstrates the idiom for a 2:1 multiplexer with 4-bit inputs and outputs using the conditional operator.

```
module mux2 (input  [3:0] d0, d1,
             input        s,
             output [3:0] y);

   assign y = s ? d1 : d0;
endmodule
```

If `s` is 1, then `y` = `d1`. If `s` is 0, then `y` = `d0`.

`?:` is also called a *ternary operator*, because it takes three inputs. It is used for the same purpose in the C and Java programming languages.

VHDL

Conditional signal assignments perform different operations depending on some condition. They are especially useful for describing a multiplexer. For example, a 2:1 multiplexer can use conditional signal assignment to select one of two 4-bit inputs.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is
  port (d0, d1: in  STD_LOGIC_VECTOR (3 downto 0);
        s:      in  STD_LOGIC;
        y:      out STD_LOGIC_VECTOR (3 downto 0));
end;

architecture synth of mux2 is
begin
  y <= d0 when s = '0' else d1;
end;
```

The conditional signal assignment sets `y` to `d0` if `s` is 0. Otherwise it sets `y` to `d1`.

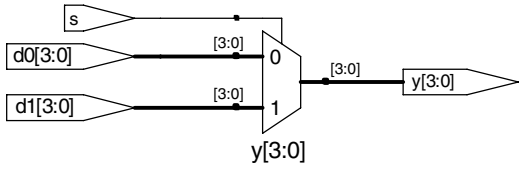


Figure 4.6 `mux2` **synthesized circuit**

HDL Example 4.6 shows a 4:1 multiplexer based on the same principle as the 2:1 multiplexer in HDL Example 4.5.

Figure 4.7 shows the schematic for the 4:1 multiplexer produced by Synplify Pro. The software uses a different multiplexer symbol than this text has shown so far. The multiplexer has multiple data (`d`) and one-hot enable (`e`) inputs. When one of the enables is asserted, the associated data is passed to the output. For example, when `s[1] = s[0] = 0`, the bottom AND gate, `un1_s_5`, produces a 1, enabling the bottom input of the multiplexer and causing it to select `d0[3:0]`.

HDL Example 4.6 4:1 MULTIPLEXER

Verilog

A 4:1 multiplexer can select one of four inputs using nested conditional operators.

```
module mux4 (input  [3:0] d0, d1, d2, d3,
             input  [1:0] s,
             output [3:0] y);

  assign y = s[1] ? (s[0] ? d3 : d2)
                  : (s[0] ? d1 : d0);
endmodule
```

If `s[1]` is 1, then the multiplexer chooses the first expression, `(s[0] ? d3 : d2)`. This expression in turn chooses either `d3` or `d2` based on `s[0]` (`y = d3` if `s[0]` is 1 and `d2` if `s[0]` is 0). If `s[1]` is 0, then the multiplexer similarly chooses the second expression, which gives either `d1` or `d0` based on `s[0]`.

VHDL

A 4:1 multiplexer can select one of four inputs using multiple `else` clauses in the conditional signal assignment.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux4 is
  port (d0, d1,
        d2, d3: in  STD_LOGIC_VECTOR (3 downto 0);
        s:      in  STD_LOGIC_VECTOR (1 downto 0);
        y:      out STD_LOGIC_VECTOR (3 downto 0));
end;

architecture synth1 of mux4 is
begin
  y <= d0 when s = "00" else
       d1 when s = "01" else
       d2 when s = "10" else
       d3;
end;
```

VHDL also supports *selected signal assignment statements* to provide a shorthand when selecting from one of several possibilities. This is analogous to using a `case` statement in place of multiple `if/else` statements in some programming languages. The 4:1 multiplexer can be rewritten with selected signal assignment as follows:

```
architecture synth2 of mux4 is
begin
  with a select y <=
    d0 when "00",
    d1 when "01",
    d2 when "10",
    d3 when others;
end;
```

4.2.5 Internal Variables

Often it is convenient to break a complex function into intermediate steps. For example, a full adder, which will be described in Section 5.2.1,

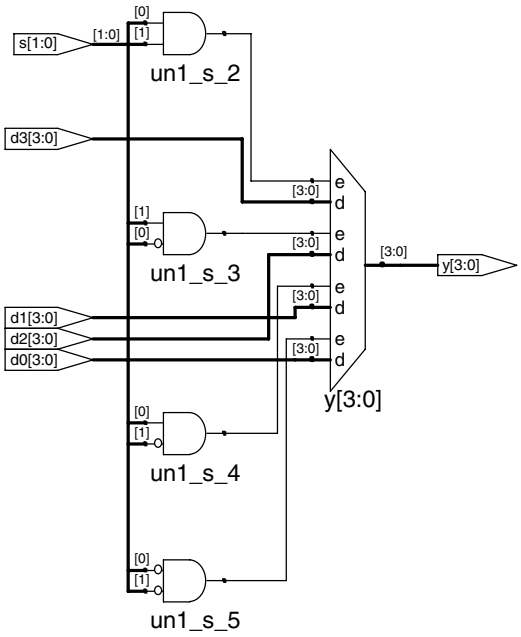


Figure 4.7 `mux4` **synthesized circuit**

is a circuit with three inputs and two outputs defined by the following equations:

$$\begin{aligned} S &= A \oplus B \oplus C_{\text{in}} \\ C_{\text{out}} &= AB + AC_{\text{in}} + BC_{\text{in}} \end{aligned} \tag{4.1}$$

If we define intermediate signals, P and G,

$$\begin{aligned} P &= A \oplus B \\ G &= AB \end{aligned} \tag{4.2}$$

we can rewrite the full adder as follows:

$$\begin{aligned} S &= P \oplus C_{\text{in}} \\ C_{\text{out}} &= G + PC_{\text{in}} \end{aligned} \tag{4.3}$$

Check this by filling out the truth table to convince yourself it is correct.

P and G are called *internal variables*, because they are neither inputs nor outputs but are used only internal to the module. They are similar to local variables in programming languages. HDL Example 4.7 shows how they are used in HDLs.

HDL assignment statements (`assign` in Verilog and <= in VHDL) take place concurrently. This is different from conventional programming languages such as C or Java, in which statements are evaluated in the order in which they are written. In a conventional language, it is

HDL Example 4.7 FULL ADDER

Verilog

In Verilog, *wires* are used to represent internal variables whose values are defined by `assign` statements such as `assign p = a ^ b;` Wires technically have to be declared only for multibit busses, but it is good practice to include them for all internal variables; their declaration could have been omitted in this example.

```
module fulladder(input  a, b, cin,
                 output s, cout);

  wire p, g;

  assign p = a ^ b;
  assign g = a & b;

  assign s = p ^ cin;
  assign cout = g  |  (p & cin);
endmodule
```

VHDL

In VHDL, *signals* are used to represent internal variables whose values are defined by *concurrent signal assignment statements* such as `p <= a xor b;`

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fulladder is
  port(a, b, cin: in  STD_LOGIC;
       s, cout:   out STD_LOGIC);
end;

architecture synth of fulladder is
  signal p, g: STD_LOGIC;
begin
  p <= a xor b;
  g <= a and b;

  s <= p xor cin;
  cout <= g or (p and cin);
end;
```

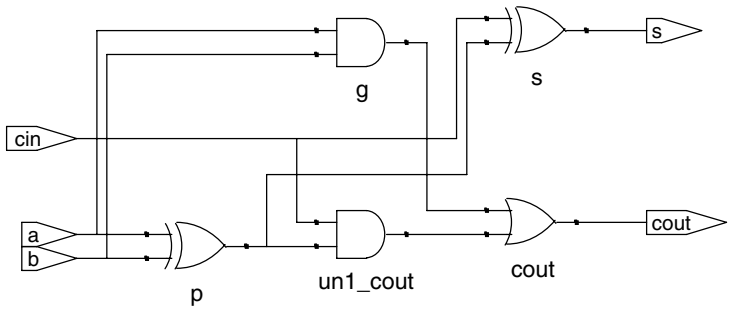


Figure 4.8 `fulladder` **synthesized circuit**

important that $S = P \oplus C_{in}$ comes after $P = A \oplus B$, because statements are executed sequentially. In an HDL, the order does not matter. Like hardware, HDL assignment statements are evaluated any time the inputs, signals on the right hand side, change their value, regardless of the order in which the assignment statements appear in a module.

4.2.6 Precedence

Notice that we parenthesized the `cout` computation in HDL Example 4.7 to define the order of operations as $C_{out} = G + (P \cdot C_{in})$, rather than $C_{out} = (G + P) \cdot C_{in}$. If we had not used parentheses, the default operation order is defined by the language. HDL Example 4.8 specifies operator precedence from highest to lowest for each language. The tables include arithmetic, shift, and comparison operators that will be defined in Chapter 5.

HDL Example 4.8 OPERATOR PRECEDENCE

Verilog

Table 4.1 Verilog operator precedence

	Op	Meaning
Highest	~	NOT
	*, /, %	MUL, DIV, MOD
	+, –	PLUS, MINUS
	<<, >>	Logical Left/Right Shift
	<<<, >>>	Arithmetic Left/Right Shift
	<, <=, >, >=	Relative Comparison
	==, !=	Equality Comparison
	&, ~&	AND, NAND
	^, ~^	XOR, XNOR
Lowest	|, ~|	OR, NOR
	?:	Conditional

The operator precedence for Verilog is much like you would expect in other programming languages. In particular, AND has precedence over OR. We could take advantage of this precedence to eliminate the parentheses.

```
assign cout = g  |  p & cin;
```

VHDL

Table 4.2 VHDL operator precedence

	Op	Meaning
Highest	not	NOT
	*, /, mod, rem	MUL, DIV, MOD, REM
	+, –, &	PLUS, MINUS, CONCATENATE
	rol, ror, srl, sll, sra, sla	Rotate, Shift logical, Shift arithmetic
	=, /=, <, <=, >, >=	Comparison
Lowest	and, or, nand, nor, xor	Logical Operations

Multiplication has precedence over addition in VHDL, as you would expect. However, unlike Verilog, all of the logical operations (`and`, `or`, etc.) have equal precedence, unlike what one might expect in Boolean algebra. Thus, parentheses are necessary; otherwise `cout <= g or p and cin` would be interpreted from left to right as `cout <= (g or p) and cin`.

4.2.7 Numbers

Numbers can be specified in a variety of bases. Underscores in numbers are ignored and can be helpful in breaking long numbers into more readable chunks. HDL Example 4.9 explains how numbers are written in each language.

4.2.8 Z's and X's

HDLs use `z` to indicate a floating value. `z` is particularly useful for describing a tristate buffer, whose output floats when the enable is 0. Recall from Section 2.6 that a bus can be driven by several tristate buffers, exactly one of which should be enabled. HDL Example 4.10 shows the idiom for a tristate buffer. If the buffer is enabled, the output is the same as the input. If the buffer is disabled, the output is assigned a floating value (`z`).

HDL Example 4.9 NUMBERS

Verilog

Verilog numbers can specify their base and size (the number of bits used to represent them). The format for declaring constants is `N'Bvalue`, where `N` is the size in bits, `B` is the base, and `value` gives the value. For example `9'h25` indicates a 9-bit number with a value of $25_{16} = 37_{10} = 000100101_2$. Verilog supports `'b` for binary (base 2), `'o` for octal (base 8), `'d` for decimal (base 10), and `'h` for hexadecimal (base 16). If the base is omitted, the base defaults to decimal.

If the size is not given, the number is assumed to have as many bits as the expression in which it is being used. Zeros are automatically padded on the front of the number to bring it up to full size. For example, if `w` is a 6-bit bus, `assign w = 'b11` gives `w` the value 000011. It is better practice to explicitly give the size.

Table 4.3 Verilog numbers

Numbers	Bits	Base	Val	Stored
3'b101	3	2	5	101
'b11	?	2	3	000 ... 0011
8'b11	8	2	3	00000011
8'b1010_1011	8	2	171	10101011
3'd6	3	10	6	110
6'o42	6	8	34	100010
8'hAB	8	16	171	10101011
42	?	10	42	00 ... 0101010

VHDL

In VHDL, STD_LOGIC numbers are written in binary and enclosed in single quotes: '0' and '1' indicate logic 0 and 1.

STD_LOGIC_VECTOR numbers are written in binary or hexadecimal and enclosed in double quotation marks. The base is binary by default and can be explicitly defined with the prefix `X` for hexadecimal or `B` for binary.

Table 4.4 VHDL numbers

Numbers	Bits	Base	Val	Stored
"101"	3	2	5	101
B"101"	3	2	5	101
X"AB"	8	16	161	10101011

HDL Example 4.10 TRISTATE BUFFER

Verilog

```
module tristate(input  [3:0] a,
                input        en,
                output [3:0] y);

  assign y = en ? a : 4'bz;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity tristate is
  port(a:  in  STD_LOGIC_VECTOR(3 downto 0);
       en: in  STD_LOGIC;
       y:  out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of tristate is
begin
  y <= "ZZZZ" when en = '0' else a;
end;
```

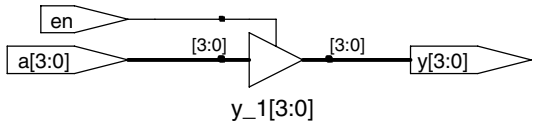


Figure 4.9 `tristate` **synthesized circuit**

Similarly, HDLs use `x` to indicate an invalid logic level. If a bus is simultaneously driven to 0 and 1 by two enabled tristate buffers (or other gates), the result is `x`, indicating contention. If all the tristate buffers driving a bus are simultaneously OFF, the bus will float, indicated by `z`.

At the start of simulation, state nodes such as flip-flop outputs are initialized to an unknown state (`x` in Verilog and `u` in VHDL). This is helpful to track errors caused by forgetting to reset a flip-flop before its output is used.

If a gate receives a floating input, it may produce an `x` output when it can't determine the correct output value. Similarly, if it receives an illegal or uninitialized input, it may produce an `x` output. HDL Example 4.11 shows how Verilog and VHDL combine these different signal values in logic gates.

HDL Example 4.11 TRUTH TABLES WITH UNDEFINED AND FLOATING INPUTS

Verilog

Verilog signal values are `0`, `1`, `z`, and `x`. Verilog constants starting with `z` or `x` are padded with leading `z`'s or `x`'s (instead of 0's) to reach their full length when necessary.

Table 4.5 shows a truth table for an AND gate using all four possible signal values. Note that the gate can sometimes determine the output despite some inputs being unknown. For example `0 & z` returns `0` because the output of an AND gate is always 0 if either input is 0. Otherwise, floating or invalid inputs cause invalid outputs, displayed as `x` in Verilog.

Table 4.5 Verilog AND gate truth table with `z` and `x`

&		A			
		0	1	z	x
B	0	0	0	0	0
	1	0	1	x	x
	z	0	x	x	x
	x	0	x	x	x

VHDL

VHDL `STD_LOGIC` signals are '`0`', '`1`', '`z`', '`x`', and '`u`'.

Table 4.6 shows a truth table for an AND gate using all five possible signal values. Notice that the gate can sometimes determine the output despite some inputs being unknown. For example, '`0`' `and` '`z`' returns '`0`' because the output of an AND gate is always '`0`' if either input is '`0`.' Otherwise, floating or invalid inputs cause invalid outputs, displayed as '`x`' in VHDL. Uninitialized inputs cause uninitialized outputs, displayed as '`u`' in VHDL.

Table 4.6 VHDL AND gate truth table with `z`, `x`, and `u`

AND		A				
		0	1	z	x	u
B	0	0	0	0	0	0
	1	0	1	x	x	u
	z	0	x	x	x	u
	x	0	x	x	x	u
	u	0	u	u	u	u

HDL Example 4.12 BIT SWIZZLING

Verilog

```
assign y = {c[2:1], {3{d[0]}}, c[0], 3'b101};
```

The {} operator is used to concatenate busses. {3{d[0]}} indicates three copies of d[0].

Don't confuse the 3-bit binary constant 3'b101 with a bus named b. Note that it was critical to specify the length of 3 bits in the constant; otherwise, it would have had an unknown number of leading zeros that might appear in the middle of y.

If y were wider than 9 bits, zeros would be placed in the most significant bits.

VHDL

```
y <= c(2 downto 1) & d(0) & d(0) & d(0) &
     c(0) & "101";
```

The & operator is used to concatenate busses. y must be a 9-bit STD_LOGIC_VECTOR. Do not confuse & with the and operator in VHDL.

Seeing x or u values in simulation is almost always an indication of a bug or bad coding practice. In the synthesized circuit, this corresponds to a floating gate input, uninitialized state, or contention. The x or u may be interpreted randomly by the circuit as 0 or 1, leading to unpredictable behavior.

4.2.9 Bit Swizzling

Often it is necessary to operate on a subset of a bus or to concatenate (join together) signals to form busses. These operations are collectively known as *bit swizzling*. In HDL Example 4.12, y is given the 9-bit value $c_2c_1d_0d_0d_0c_0101$ using bit swizzling operations.

4.2.10 Delays

HDL statements may be associated with delays specified in arbitrary units. They are helpful during simulation to predict how fast a circuit will work (if you specify meaningful delays) and also for debugging purposes to understand cause and effect (deducing the source of a bad output is tricky if all signals change simultaneously in the simulation results). These delays are ignored during synthesis; the delay of a gate produced by the synthesizer depends on its t_{pd} and t_{cd} specifications, not on numbers in HDL code.

HDL Example 4.13 adds delays to the original function from HDL Example 4.1, $y = \bar{a}\bar{b}\bar{c} + a\bar{b}\bar{c} + a\bar{b}c$. It assumes that inverters have a delay of 1 ns, three-input AND gates have a delay of 2 ns, and three-input OR gates have a delay of 4 ns. Figure 4.10 shows the simulation waveforms, with y lagging 7 ns after the inputs. Note that y is initially unknown at the beginning of the simulation.

HDL Example 4.13 LOGIC GATES WITH DELAYS

Verilog

```
`timescale 1ns/1ps

module example(input  a, b, c,
               output y);
  wire ab, bb, cb, n1, n2, n3;

  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```

Verilog files can include a timescale directive that indicates the value of each time unit. The statement is of the form `timescale unit/precision. In this file, each unit is 1 ns, and the simulation has 1 ps precision. If no timescale directive is given in the file, a default unit and precision (usually 1 ns for both) is used. In Verilog, a # symbol is used to indicate the number of units of delay. It can be placed in assign statements, as well as non-blocking (<=) and blocking (=) assignments, which will be discussed in Section 4.5.4.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity example is
  port(a, b, c: in  STD_LOGIC;
       y:       out STD_LOGIC);
end;

architecture synth of example is
signal ab, bb, cb, n1, n2, n3: STD_LOGIC;
begin
  ab <= not a after 1 ns;
  bb <= not b after 1 ns;
  cb <= not c after 1 ns;
  n1 <= ab and bb and cb after 2 ns;
  n2 <= a and bb and cb after 2 ns;
  n3 <= a and bb and c after 2 ns;
  y <= n1 or n2 or n3 after 4 ns;
end;
```

In VHDL, the after clause is used to indicate delay. The units, in this case, are specified as nanoseconds.

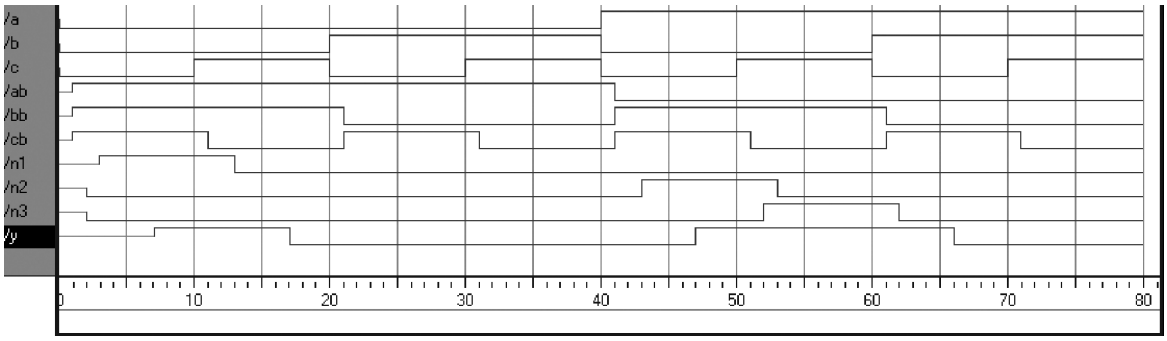


Figure 4.10 Example simulation waveforms with delays (from the ModelSim simulator)

4.2.11 VHDL Libraries and Types*

(This section may be skipped by Verilog users.) Unlike Verilog, VHDL enforces a strict data typing system that can protect the user from some errors but that is also clumsy at times.

Despite its fundamental importance, the STD_LOGIC type is not built into VHDL. Instead, it is part of the IEEE.STD_LOGIC_1164 library. Thus, every file must contain the library statements shown in the previous examples.

Moreover, IEEE.STD_LOGIC_1164 lacks basic operations such as addition, comparison, shifts, and conversion to integers for the STD_LOGIC_VECTOR data. Most CAD vendors have adopted yet more libraries containing these functions: IEEE.STD_LOGIC_UNSIGNED and IEEE.STD_LOGIC_SIGNED. See Section 1.4 for a discussion of unsigned and signed numbers and examples of these operations.

VHDL also has a BOOLEAN type with two values: true and false. BOOLEAN values are returned by comparisons (such as the equality comparison, s = '0') and are used in conditional statements such as when. Despite the temptation to believe a BOOLEAN true value should be equivalent to a STD_LOGIC '1' and BOOLEAN false should mean STD_LOGIC '0', these types are not interchangeable. Thus, the following code is illegal:

```
y <= d1 when s else d0;
q <= (state = S2);
```

Instead, we must write

```
y <= d1 when (s = '1') else d0;
q <= '1' when (state = S2) else '0';
```

Although we do not declare any signals to be BOOLEAN, they are automatically implied by comparisons and used by conditional statements.

Similarly, VHDL has an INTEGER type that represents both positive and negative integers. Signals of type INTEGER span at least the values -2^{31} to $2^{31} - 1$. Integer values are used as indices of busses. For example, in the statement

```
y <= a(3) and a(2) and a(1) and a(0);
```

0, 1, 2, and 3 are integers serving as an index to choose bits of the a signal. We cannot directly index a bus with a STD_LOGIC or STD_LOGIC_VECTOR signal. Instead, we must convert the signal to an INTEGER. This is demonstrated in HDL Example 4.14 for an 8:1 multiplexer that selects one bit from a vector using a 3-bit index. The CONV_INTEGER function is defined in the IEEE.STD_LOGIC_UNSIGNED library and performs the conversion from STD_LOGIC_VECTOR to INTEGER for positive (unsigned) values.

VHDL is also strict about out ports being exclusively for output. For example, the following code for two and three-input AND gates is illegal VHDL because v is an output and is also used to compute w.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity and23 is
  port(a, b, c: in  STD_LOGIC;
       v, w:    out STD_LOGIC);
end;
```

HDL Example 4.14 8:1 MULTIPLEXER WITH TYPE CONVERSION

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity mux8 is
  port(d: in  STD_LOGIC_VECTOR(7 downto 0);
       s: in  STD_LOGIC_VECTOR(2 downto 0);
       y: out STD_LOGIC);
end;

architecture synth of mux8 is
begin
  y <= d(CONV_INTEGER(s));
end;
```

Figure follows on next page.

```
architecture synth of and23 is
begin
  v <= a and b;
  w <= v and c;
end;
```

VHDL defines a special port type, `buffer`, to solve this problem. A signal connected to a `buffer` port behaves as an output but may also be used within the module. The corrected entity definition follows. Verilog does not have this limitation and does not require buffer ports.

```
entity and23 is
  port(a, b, c: in     STD_LOGIC;
       v:       buffer STD_LOGIC;
       w:       out    STD_LOGIC);
end;
```

VHDL supports *enumeration* types as an abstract way of representing information without assigning specific binary encodings. For example, the divide-by-3 FSM described in Section 3.4.2 uses three states. We can give the states names using the enumeration type rather than referring to them by binary values. This is powerful because it allows VHDL to search for the best state encoding during synthesis, rather than depending on an arbitrary encoding specified by the user.

```
type statetype is (S0, S1, S2);
signal state, nextstate: statetype;
```

4.3 STRUCTURAL MODELING

The previous section discussed *behavioral* modeling, describing a module in terms of the relationships between inputs and outputs. This section

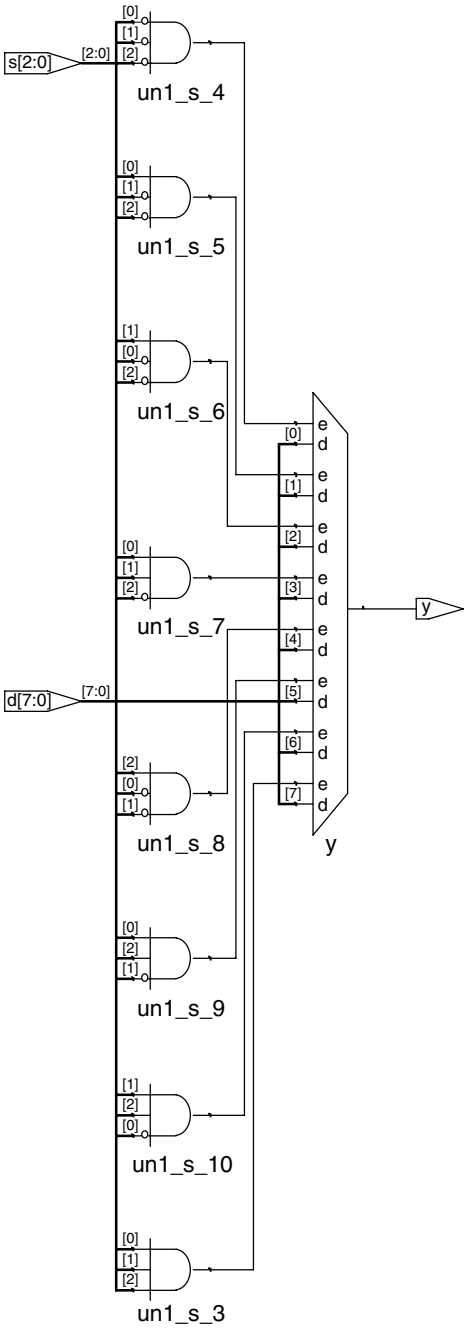


Figure 4.11 `mux8` **synthesized circuit**

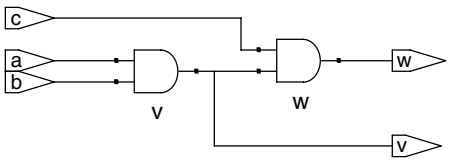


Figure 4.12 `and23` **synthesized circuit**

examines *structural* modeling, describing a module in terms of how it is composed of simpler modules.

For example, HDL Example 4.15 shows how to assemble a 4:1 multiplexer from three 2:1 multiplexers. Each copy of the 2:1 multiplexer

---

HDL Example 4.15 STRUCTURAL MODEL OF 4:1 MULTIPLEXER

Verilog

```
module mux4(input  [3:0] d0, d1, d2, d3,
            input  [1:0] s,
            output [3:0] y);

  wire [3:0] low, high;

  mux2 lowmux(d0, d1, s[0], low);
  mux2 highmux(d2, d3, s[0], high);
  mux2 finalmux(low, high, s[1], y);
endmodule
```

The three `mux2` instances are called `lowmux`, `highmux`, and `finalmux`. The `mux2` module must be defined elsewhere in the Verilog code.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux4 is
  port(d0, d1,
       d2, d3: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC_VECTOR(1 downto 0);
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture struct of mux4 is
  component mux2
    port(d0,
         d1: in  STD_LOGIC_VECTOR(3 downto 0);
         s:  in  STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(3 downto 0));
  end component;
  signal low, high: STD_LOGIC_VECTOR(3 downto 0);
begin
  lowmux:   mux2 port map(d0, d1, s(0), low);
  highmux:  mux2 port map(d2, d3, s(0), high);
  finalmux: mux2 port map(low, high, s(1), y);
end;
```

The architecture must first declare the `mux2` ports using the `component` declaration statement. This allows VHDL tools to check that the component you wish to use has the same ports as the entity that was declared somewhere else in another entity statement, preventing errors caused by changing the entity but not the instance. However, component declaration makes VHDL code rather cumbersome.

Note that this architecture of `mux4` was named `struct`, whereas architectures of modules with behavioral descriptions from Section 4.2 were named `synth`. VHDL allows multiple architectures (implementations) for the same entity; the architectures are distinguished by name. The names themselves have no significance to the CAD tools, but `struct` and `synth` are common. Synthesizable VHDL code generally contains only one architecture for each entity, so we will not discuss the VHDL syntax to configure which architecture is used when multiple architectures are defined.

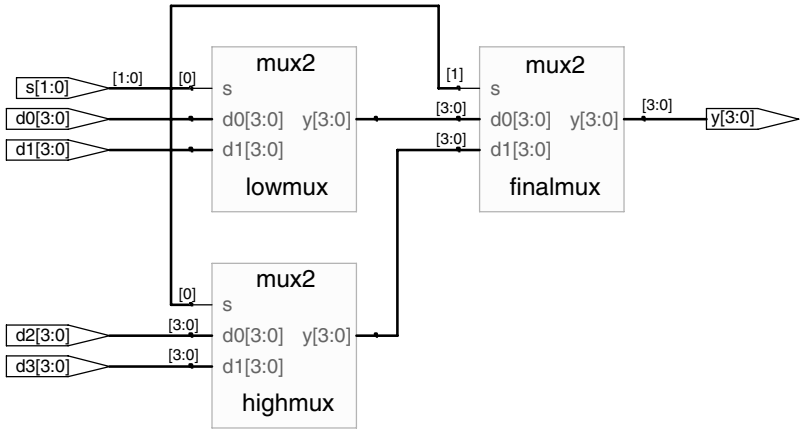


Figure 4.13 `mux4` **synthesized circuit**

is called an *instance*. Multiple instances of the same module are distinguished by distinct names, in this case `lowmux`, `highmux`, and `finalmux`. This is an example of regularity, in which the 2:1 multiplexer is reused many times.

HDL Example 4.16 uses structural modeling to construct a 2:1 multiplexer from a pair of tristate buffers.

HDL Example 4.16 STRUCTURAL MODEL OF 2:1 MULTIPLEXER

Verilog

```
module mux2(input  [3:0] d0, d1,
            input        s,
            output [3:0] y);

  tristate t0(d0, ~s, y);
  tristate t1(d1, s, y);
endmodule
```

In Verilog, expressions such as `~s` are permitted in the port list for an instance. Arbitrarily complicated expressions are legal but discouraged because they make the code difficult to read.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is
  port(d0, d1: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture struct of mux2 is
  component tristate
    port(a:  in  STD_LOGIC_VECTOR(3 downto 0);
         en: in  STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(3 downto 0));
  end component;
  signal sbar: STD_LOGIC;
begin
  sbar <= not s;
  t0: tristate port map(d0, sbar, y);
  t1: tristate port map(d1, s, y);
end;
```

In VHDL, expressions such as `not s` are not permitted in the port map for an instance. Thus, `sbar` must be defined as a separate signal.

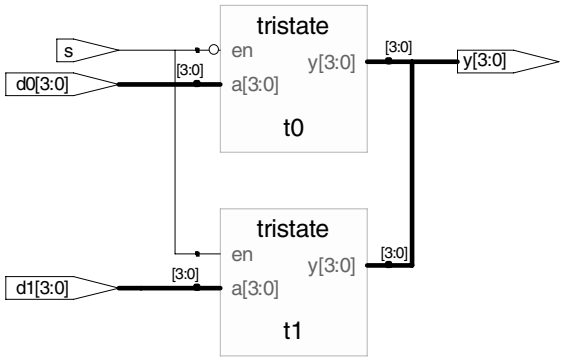


Figure 4.14 `mux2` **synthesized circuit**

HDL Example 4.17 shows how modules can access part of a bus. An 8-bit wide 2:1 multiplexer is built using two of the 4-bit 2:1 multiplexers already defined, operating on the low and high nibbles of the byte.

In general, complex systems are designed *hierarchically*. The overall system is described structurally by instantiating its major components. Each of these components is described structurally from its building blocks, and so forth recursively until the pieces are simple enough to describe behaviorally. It is good style to avoid (or at least to minimize) mixing structural and behavioral descriptions within a single module.

HDL Example 4.17 ACCESSING PARTS OF BUSSES

Verilog

```
module mux2_8(input  [7:0] d0, d1,
              input        s,
              output [7:0] y);

  mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);
  mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2_8 is
  port(d0, d1: in  STD_LOGIC_VECTOR(7 downto 0);
       s:      in  STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture struct of mux2_8 is
  component mux2
    port(d0, d1: in   STD_LOGIC_VECTOR(3
                                  downto 0);
         s:       in  STD_LOGIC;
         y:       out STD_LOGIC_VECTOR(3 downto 0));
  end component;
begin

  lsbmux:  mux2
    port map(d0(3 downto 0), d1(3 downto 0),
             s, y(3 downto 0));
  msbhmux: mux2
    port map(d0(7 downto 4), d1(7 downto 4),
             s, y(7 downto 4));
end;
```

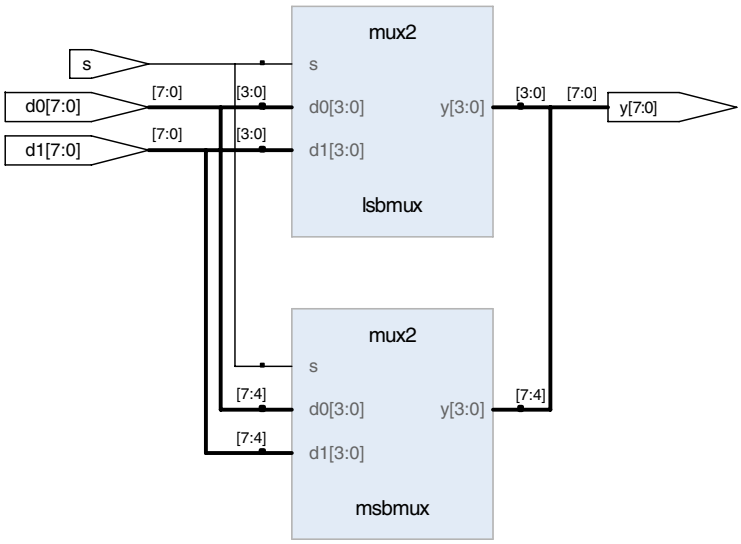


Figure 4.15 `mux2_8` **synthesized circuit**

4.4 SEQUENTIAL LOGIC

HDL synthesizers recognize certain idioms and turn them into specific sequential circuits. Other coding styles may simulate correctly but synthesize into circuits with blatant or subtle errors. This section presents the proper idioms to describe registers and latches.

4.4.1 Registers

The vast majority of modern commercial systems are built with registers using positive edge-triggered D flip-flops. HDL Example 4.18 shows the idiom for such flip-flops.

In Verilog `always` statements and VHDL `process` statements, signals keep their old value until an event in the sensitivity list takes place that explicitly causes them to change. Hence, such code, with appropriate sensitivity lists, can be used to describe sequential circuits with memory. For example, the flip-flop includes only `clk` in the sensitive list. It remembers its old value of `q` until the next rising edge of the `clk`, even if `d` changes in the interim.

In contrast, Verilog continuous assignment statements (`assign`) and VHDL concurrent assignment statements (<=) are reevaluated anytime any of the inputs on the right hand side changes. Therefore, such code necessarily describes combinational logic.

HDL Example 4.18 REGISTER

Verilog

```
module flop(input             clk,
            input       [3:0] d,
            output reg [3:0] q);

  always @ (posedge clk)
    q <= d;
endmodule
```

A Verilog `always` statement is written in the form

```
always @ (sensitivity list)
  statement;
```

The `statement` is executed only when the event specified in the `sensitivity list` occurs. In this example, the statement is `q <= d` (pronounced "q gets d"). Hence, the flip-flop copies `d` to `q` on the positive edge of the clock and otherwise remembers the old state of `q`.

`<=` is called a *nonblocking assignment*. Think of it as a regular `=` sign for now; we'll return to the more subtle points in Section 4.5.4. Note that `<=` is used instead of `assign` inside an `always` statement.

All signals on the left hand side of `<=` or `=` in an `always` statement must be declared as `reg`. In this example, `q` is both an output and a `reg`, so it is declared as `output reg [3:0] q`. Declaring a signal as `reg` does not mean the signal is actually the output of a register! All it means is that the signal appears on the left hand side of an assignment in an `always` statement. We will see later examples of `always` statements describing combinational logic in which the output is declared `reg` but does not come from a flip-flop.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flop is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC_VECTOR(3 downto 0);
       q:   out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of flop is
begin
  process(clk) begin
    if clk'event and clk = '1' then
      q <= d;
    end if;
  end process;
end;
```

A VHDL `process` is written in the form

```
process(sensitivity list) begin
  statement;
end process;
```

The statement is executed when any of the variables in the `sensitivity list` change. In this example, the `if` statement is executed when `clk` changes, indicated by `clk'event`. If the change is a rising edge (`clk = '1'` after the event), then `q <= d` (pronounced "q gets d"). Hence, the flip-flop copies `d` to `q` on the positive edge of the clock and otherwise remembers the old state of `q`.

An alternative VHDL idiom for a flip-flop is

```
process(clk) begin
  if RISING_EDGE(clk) then
    q <= d;
  end if;
end process;
```

`RISING_EDGE(clk)` is synonymous with `clk'event` and `clk = 1`.

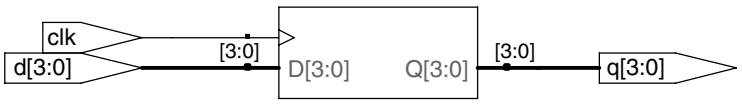


Figure 4.16 `flop` **synthesized circuit**

4.4.2 Resettable Registers

When simulation begins or power is first applied to a circuit, the output of a flop or register is unknown. This is indicated with `x` in Verilog and '`u`' in VHDL. Generally, it is good practice to use resettable registers so that on powerup you can put your system in a known state. The reset may be

either asynchronous or synchronous. Recall that asynchronous reset occurs immediately, whereas synchronous reset clears the output only on the next rising edge of the clock. HDL Example 4.19 demonstrates the idioms for flip-flops with asynchronous and synchronous resets. Note that distinguishing synchronous and asynchronous reset in a schematic can be difficult. The schematic produced by Synplify Pro places asynchronous reset at the bottom of a flip-flop and synchronous reset on the left side.

HDL Example 4.19 RESETTABLE REGISTER

Verilog

```
module flopr (input            clk,
              input            reset,
              input      [3:0] d,
              output reg [3:0] q);

  // asynchronous reset
  always @ (posedge clk, posedge reset)
    if (reset) q <= 4'b0;
    else       q <= d;
endmodule

module flopr (input            clk,
              input            reset,
              input      [3:0] d,
              output reg [3:0] q);

  // synchronous reset
  always @ (posedge clk)
    if (reset) q <= 4'b0;
    else       q <= d;
endmodule
```

Multiple signals in an `always` statement sensitivity list are separated with a comma or the word `or`. Notice that `posedge reset` is in the sensitivity list on the asynchronously resettable flop, but not on the synchronously resettable flop. Thus, the asynchronously resettable flop immediately responds to a rising edge on `reset`, but the synchronously resettable flop responds to `reset` only on the rising edge of the clock.

Because the modules have the same name, `flopr`, you may include only one or the other in your design.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flopr is
  port (clk,
        reset: in  STD_LOGIC;
        d:     in  STD_LOGIC_VECTOR (3 downto 0);
        q:     out STD_LOGIC_VECTOR (3 downto 0));
end;

architecture asynchronous of flopr is
begin
  process (clk, reset) begin
    if reset = '1' then
      q <= "0000";
    elsif clk' event and clk = '1' then
      q <= d;
    end if;
  end process;
end;

architecture synchronous of flopr is
begin
  process (clk) begin
    if clk'event and clk = '1' then
      if reset = '1' then
        q <= "0000";
      else q <= d;
      end if;
    end if;
  end process;
end;
```

Multiple signals in a `process` sensitivity list are separated with a comma. Notice that `reset` is in the sensitivity list on the asynchronously resettable flop, but not on the synchronously resettable flop. Thus, the asynchronously resettable flop immediately responds to a rising edge on `reset`, but the synchronously resettable flop responds to `reset` only on the rising edge of the clock.

Recall that the state of a flop is initialized to '`u`' at startup during VHDL simulation.

As mentioned earlier, the name of the architecture (`asynchronous` or `synchronous`, in this example) is ignored by the VHDL tools but may be helpful to the human reading the code. Because both architectures describe the entity `flopr`, you may include only one or the other in your design.

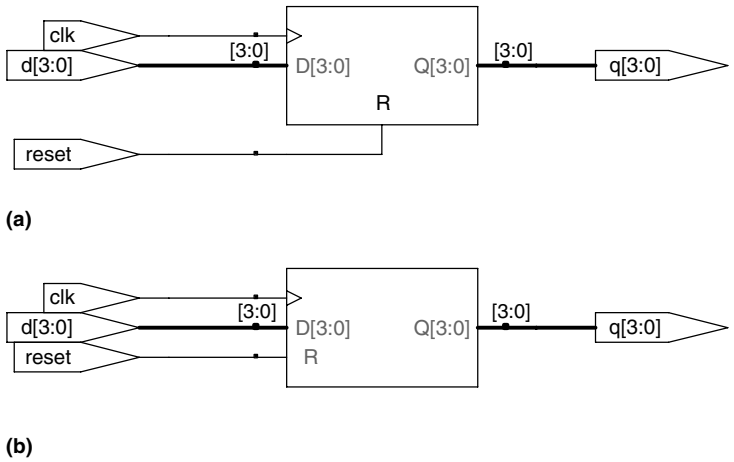


Figure 4.17 `flopr` **synthesized circuit (a) asynchronous reset, (b) synchronous reset**

4.4.3 Enabled Registers

Enabled registers respond to the clock only when the enable is asserted. HDL Example 4.20 shows an asynchronously resettable enabled register that retains its old value if both `reset` and `en` are FALSE.

HDL Example 4.20 RESETTABLE ENABLED REGISTER

Verilog

```
module flopenr (input            clk,
                input            reset,
                input            en,
                input      [3:0] d,
                output reg [3:0] q);

  // asynchronous reset
  always @ (posedge clk, posedge reset)
    if (reset) q <= 4'b0;
    else if (en) q <= d;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flopenr is
  port (clk,
        reset,
        en: in  STD_LOGIC;
        d:  in  STD_LOGIC_VECTOR (3 downto 0);
        q:  out STD_LOGIC_VECTOR (3 downto 0));
end;

architecture asynchronous of flopenr is
-- asynchronous reset
begin
  process (clk, reset) begin
    if reset = '1' then
      q <= "0000";
    elsif clk'event and clk = '1' then
      if en = '1' then
        q <= d;
      end if;
    end if;
  end process;
end;
```

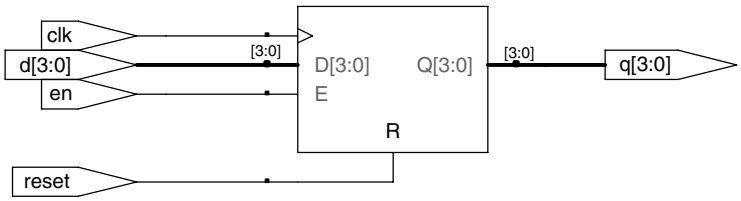


Figure 4.18 `flopenr` **synthesized circuit**

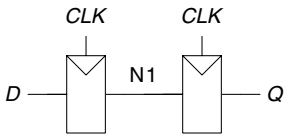


Figure 4.19 **Synchronizer circuit**

4.4.4 Multiple Registers

A single `always/process` statement can be used to describe multiple pieces of hardware. For example, consider the synchronizer from Section 3.5.5 made of two back-to-back flip-flops, as shown in Figure 4.19. HDL Example 4.21 describes the synchronizer. On the rising edge of `clk`, `d` is copied to `n1`. At the same time, `n1` is copied to `q`.

HDL Example 4.21 SYNCHRONIZER

Verilog

```
module sync(input      clk,
            input      d,
            output reg q);

  reg n1;

  always @ (posedge clk)
    begin
      n1 <= d;
      q <= n1;
    end
endmodule
```

`n1` must be declared as a `reg` because it is an internal signal used on the left hand side of <= in an `always` statement. Also notice that the `begin/end` construct is necessary because multiple statements appear in the `always` statement. This is analogous to { } in C or Java. The `begin/end` was not needed in the `flopr` example because `if/else` counts as a single statement.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity sync is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC;
       q:   out STD_LOGIC);
end;

architecture good of sync is
  signal n1: STD_LOGIC;
begin
  process(clk) begin
    if clk'event and clk = '1' then
      n1 <= d;
      q <= n1;
    end if;
  end process;
end;
```

`n1` must be declared as a `signal` because it is an internal signal used in the module.

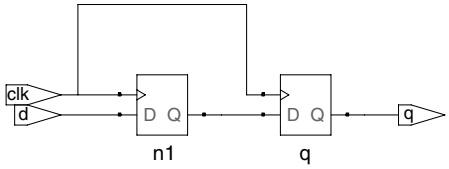


Figure 4.20 `sync` **synthesized circuit**

4.4.5 Latches

Recall from Section 3.2.2 that a D latch is transparent when the clock is HIGH, allowing data to flow from input to output. The latch becomes opaque when the clock is LOW, retaining its old state. HDL Example 4.22 shows the idiom for a D latch.

Not all synthesis tools support latches well. Unless you know that your tool does support latches and you have a good reason to use them, avoid them and use edge-triggered flip-flops instead. Furthermore, take care that your HDL does not imply any unintended latches, something that is easy to do if you aren't attentive. Many synthesis tools warn you when a latch is created; if you didn't expect one, track down the bug in your HDL.

4.5 MORE COMBINATIONAL LOGIC

In Section 4.2, we used assignment statements to describe combinational logic behaviorally. Verilog `always` statements and VHDL `process` statements are used to describe sequential circuits, because they remember the old state when no new state is prescribed. However, `always`/`process`

HDL Example 4.22 D LATCH

Verilog

```
module latch(input            clk,
             input      [3:0] d,
             output reg [3:0] q);

  always @(clk, d)
    if (clk) q <= d;
endmodule
```

The sensitivity list contains both `clk` and `d`, so the `always` statement evaluates any time `clk` or `d` changes. If `clk` is HIGH, `d` flows through to `q`.

`q` must be declared to be a `reg` because it appears on the left hand side of <= in an `always` statement. This does not always mean that `q` is the output of a register.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity latch is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC_VECTOR(3 downto 0);
       q:   out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of latch is
begin
  process(clk, d) begin
    if clk = '1' then q <= d;
    end if;
  end process;
end;
```

The sensitivity list contains both `clk` and `d`, so the `process` evaluates anytime `clk` or `d` changes. If `clk` is HIGH, `d` flows through to `q`.

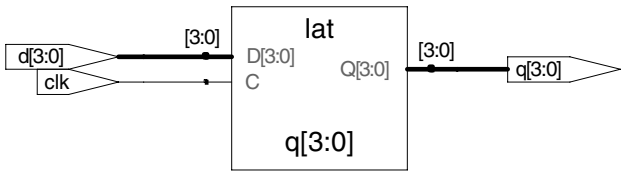


Figure 4.21 `latch` **synthesized circuit**

HDL Example 4.23 INVERTER USING `always/process`

Verilog

```
module inv(input      [3:0] a,
           output reg [3:0] y);

  always @ (*)
    y = ~a;
endmodule
```

`always @ (*)` reevaluates the statements inside the `always` statement any time any of the signals on the right hand side of <= or = inside the always statement change. Thus, `@ (*)` is a safe way to model combinational logic. In this particular example, `@ (a)` would also have sufficed.

The = in the always statement is called a *blocking assignment*, in contrast to the <= nonblocking assignment. In Verilog, it is good practice to use blocking assignments for combinational logic and nonblocking assignments for sequential logic. This will be discussed further in Section 4.5.4.

Note that `y` must be declared as `reg` because it appears on the left hand side of a <= or = sign in an `always` statement. Nevertheless, `y` is the output of combinational logic, not a register.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity inv is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture proc of inv is
begin
  process(a) begin
    y <= not a;
  end process;
end;
```

The `begin` and `end process` statements are required in VHDL even though the `process` contains only one assignment.

statements can also be used to describe combinational logic behaviorally if the sensitivity list is written to respond to changes in all of the inputs and the body prescribes the output value for every possible input combination. HDL Example 4.23 uses `always/process` statements to describe a bank of four inverters (see Figure 4.3 for the synthesized circuit).

HDLs support *blocking* and *nonblocking assignments* in an `always/process` statement. A group of blocking assignments are evaluated in the order in which they appear in the code, just as one would expect in a standard programming language. A group of nonblocking assignments are evaluated concurrently; all of the statements are evaluated before any of the signals on the left hand sides are updated.

HDL Example 4.24 defines a full adder using intermediate signals `p` and `g` to compute `s` and `cout`. It produces the same circuit from Figure 4.8, but uses `always/process` statements in place of assignment statements.

These two examples are poor applications of `always/process` statements for modeling combinational logic because they require more lines than the equivalent approach with assignment statements from Section 4.2.1. Moreover, they pose the risk of inadvertently implying sequential logic if the inputs are left out of the sensitivity list. However, `case` and `if` statements are convenient for modeling more complicated combinational logic. `case` and `if` statements must appear within `always/process` statements and are examined in the next sections.

Verilog

In a Verilog `always` statement, = indicates a blocking assignment and <= indicates a nonblocking assignment (also called a concurrent assignment).

Do not confuse either type with continuous assignment using the `assign` statement. `assign` statements must be used outside `always` statements and are also evaluated concurrently.

VHDL

In a VHDL `process` statement, := indicates a blocking assignment and <= indicates a nonblocking assignment (also called a concurrent assignment). This is the first section where := is introduced.

Nonblocking assignments are made to outputs and to signals. Blocking assignments are made to variables, which are declared in `process` statements (see HDL Example 4.24).

<= can also appear outside `process` statements, where it is also evaluated concurrently.

HDL Example 4.24 FULL ADDER USING `always/process`

Verilog

```
module fulladder(input      a, b, cin,
                 output reg s, cout);
  reg p, g;

  always @ (*)
    begin
      p = a ^ b;               // blocking
      g = a & b;               // blocking

      s = p ^ cin;             // blocking
      cout = g | (p & cin);    // blocking
    end
endmodule
```

In this case, an `@(a, b, cin)` would have been equivalent to `@(*)`. However, `@(*)` is better because it avoids common mistakes of missing signals in the stimulus list.

For reasons that will be discussed in Section 4.5.4, it is best to use blocking assignments for combinational logic. This example uses blocking assignments, first computing `p`, then `g`, then `s`, and finally `cout`.

Because `p` and `g` appear on the left hand side of an assignment in an `always` statement, they must be declared to be `reg`.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fulladder is
  port(a, b, cin:  in  STD_LOGIC;
       s, cout:    out STD_LOGIC);
end;

architecture synth of fulladder is
begin
  process(a, b, cin)
    variable p, g: STD_LOGIC;
  begin
    p := a xor b; -- blocking
    g := a and b; -- blocking

    s <= p xor cin;
    cout <= g or (p and cin);
  end process;
end;
```

The `process` sensitivity list must include `a`, `b`, and `cin` because combinational logic should respond to changes of any input.

For reasons that will be discussed in Section 4.5.4, it is best to use blocking assignments for intermediate variables in combinational logic. This example uses blocking assignments for `p` and `g` so that they get their new values before being used to compute `s` and `cout` that depend on them.

Because `p` and `g` appear on the left hand side of a blocking assignment (:=) in a `process` statement, they must be declared to be `variable` rather than `signal`. The variable declaration appears before the `begin` in the process where the variable is used.

4.5.1 Case Statements

A better application of using the `always/process` statement for combinational logic is a seven-segment display decoder that takes advantage of the `case` statement that must appear inside an `always/process` statement.

As you might have noticed in Example 2.10, the design process for large blocks of combinational logic is tedious and prone to error. HDLs offer a great improvement, allowing you to specify the function at a higher level of abstraction, and then automatically synthesize the function into gates. HDL Example 4.25 uses `case` statements to describe a seven-segment display decoder based on its truth table. (See Example 2.10 for a description of the seven-segment display decoder.) The `case`

HDL Example 4.25 SEVEN-SEGMENT DISPLAY DECODER

Verilog

```
module sevenseg(input      [3:0] data,
                output reg [6:0] segments);
  always @ (*)
    case(data)
      //               abc_defg
      0: segments = 7'b111_1110;
      1: segments = 7'b011_0000;
      2: segments = 7'b110_1101;
      3: segments = 7'b111_1001;
      4: segments = 7'b011_0011;
      5: segments = 7'b101_1011;
      6: segments = 7'b101_1111;
      7: segments = 7'b111_0000;
      8: segments = 7'b111_1111;
      9: segments = 7'b111_1011;
      default: segments = 7'b000_0000;
    endcase
endmodule
```

The `case` statement checks the value of `data`. When `data` is 0, the statement performs the action after the colon, setting `segments` to 1111110. The `case` statement similarly checks other `data` values up to 9 (note the use of the default base, base 10).

The `default` clause is a convenient way to define the output for all cases not explicitly listed, guaranteeing combinational logic.

In Verilog, `case` statements must appear inside `always` statements.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity seven_seg_decoder is
  port(data:     in  STD_LOGIC_VECTOR(3 downto 0);
       segments: out STD_LOGIC_VECTOR(6 downto 0));
end;

architecture synth of seven_seg_decoder is
begin
  process(data) begin
    case data is
      --                          abcdefg
      when X"0" => segments <= "1111110";
      when X"1" => segments <= "0110000";
      when X"2" => segments <= "1101101";
      when X"3" => segments <= "1111001";
      when X"4" => segments <= "0110011";
      when X"5" => segments <= "1011011";
      when X"6" => segments <= "1011111";
      when X"7" => segments <= "1110000";
      when X"8" => segments <= "1111111";
      when X"9" => segments <= "1111011";
      when others => segments <= "0000000";
    end case;
  end process;
end;
```

The `case` statement checks the value of `data`. When `data` is 0, the statement performs the action after the `=>`, setting `segments` to 1111110. The `case` statement similarly checks other `data` values up to 9 (note the use of `X` for hexadecimal numbers). The `others` clause is a convenient way to define the output for all cases not explicitly listed, guaranteeing combinational logic.

Unlike Verilog, VHDL supports selected signal assignment statements (see HDL Example 4.6), which are much like `case` statements but can appear outside processes. Thus, there is less reason to use processes to describe combinational logic.

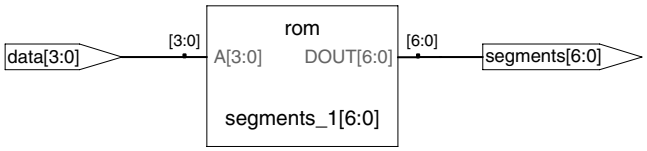


Figure 4.22 `sevenseg` **synthesized circuit**

statement performs different actions depending on the value of its input. A `case` statement implies combinational logic if all possible input combinations are defined; otherwise it implies sequential logic, because the output will keep its old value in the undefined cases.

Synplify Pro synthesizes the seven-segment display decoder into a *read-only memory* (*ROM*) containing the 7 outputs for each of the 16 possible inputs. ROMs are discussed further in Section 5.5.6.

If the `default` or `others` clause were left out of the `case` statement, the decoder would have remembered its previous output anytime data were in the range of 10–15. This is strange behavior for hardware.

Ordinary decoders are also commonly written with `case` statements. HDL Example 4.26 describes a 3:8 decoder.

4.5.2 If Statements

`always/process` statements may also contain `if` statements. The `if` statement may be followed by an `else` statement. If all possible input

HDL Example 4.26 3:8 DECODER

Verilog

```
module decoder3_8 (input      [2:0] a,
                   output reg [7:0] y);
  always @ (*)
    case (a)
      3'b000: y = 8'b00000001;
      3'b001: y = 8'b00000010;
      3'b010: y = 8'b00000100;
      3'b011: y = 8'b00001000;
      3'b100: y = 8'b00010000;
      3'b101: y = 8'b00100000;
      3'b110: y = 8'b01000000;
      3'b111: y = 8'b10000000;
    endcase
endmodule
```

No `default` statement is needed because all cases are covered.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity decoder3_8 is
  port (a: in  STD_LOGIC_VECTOR (2 downto 0);
        y: out STD_LOGIC_VECTOR (7 downto 0));
end;

architecture synth of decoder3_8 is
begin
  process (a) begin
    case a is
      when "000" => y <= "00000001";
      when "001" => y <= "00000010";
      when "010" => y <= "00000100";
      when "011" => y <= "00001000";
      when "100" => y <= "00010000";
      when "101" => y <= "00100000";
      when "110" => y <= "01000000";
      when "111" => y <= "10000000";
    end case;
  end process;
end;
```

No `others` clause is needed because all cases are covered.

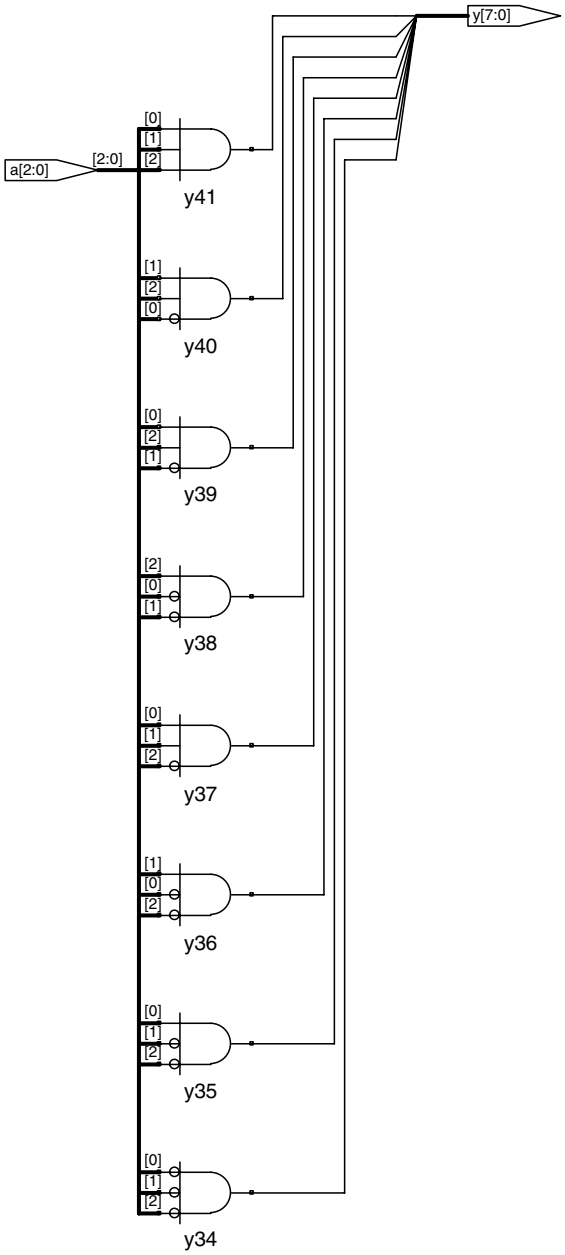


Figure 4.23 `decoder3_8` **synthesized circuit**

HDL Example 4.27 PRIORITY CIRCUIT

Verilog

```
module priority(input      [3:0] a,
                output reg [3:0] y);

  always @ (*)
    if      (a[3]) y = 4'b1000;
    else if (a[2]) y = 4'b0100;
    else if (a[1]) y = 4'b0010;
    else if (a[0]) y = 4'b0001;
    else           y = 4'b0000;
endmodule
```

In Verilog, `if` statements must appear inside of `always` statements.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity priority is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of priority is
begin
  process(a) begin
    if    a(3) = '1' then y <= "1000";
    elsif a(2) = '1' then y <= "0100";
    elsif a(1) = '1' then y <= "0010";
    elsif a(0) = '1' then y <= "0001";
    else                  y <= "0000";
    end if;
  end process;
end;
```

Unlike Verilog, VHDL supports conditional signal assignment statements (see HDL Example 4.6), which are much like `if` statements but can appear outside processes. Thus, there is less reason to use processes to describe combinational logic. (Figure follows on next page.)

combinations are handled, the statement implies combinational logic; otherwise, it produces sequential logic (like the latch in Section 4.4.5).

HDL Example 4.27 uses `if` statements to describe a priority circuit, defined in Section 2.4. Recall that an *N*-input priority circuit sets the output TRUE that corresponds to the most significant input that is TRUE.

4.5.3 Verilog `casez`*

(This section may be skipped by VHDL users.) Verilog also provides the `casez` statement to describe truth tables with don't cares (indicated with `?` in the `casez` statement). HDL Example 4.28 shows how to describe a priority circuit with `casez`.

Synplify Pro synthesizes a slightly different circuit for this module, shown in Figure 4.25, than it did for the priority circuit in Figure 4.24. However, the circuits are logically equivalent.

4.5.4 Blocking and Nonblocking Assignments

The guidelines on page 203 explain when and how to use each type of assignment. If these guidelines are not followed, it is possible to write

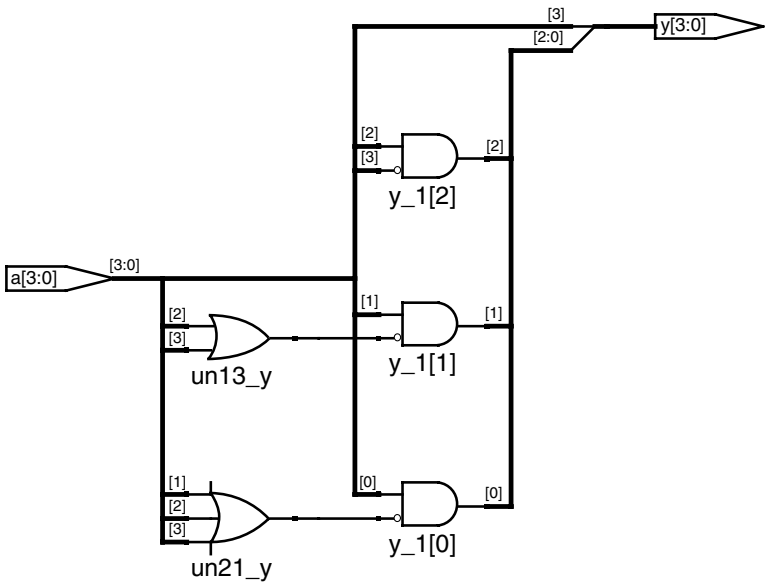


Figure 4.24 `priority` **synthesized circuit**

HDL Example 4.28 PRIORITY CIRCUIT USING `casez`

```
module priority_casez(input      [3:0] a,
                      output reg [3:0] y);

  always @ (*)
    casez (a)
      4'b1???: y = 4'b1000;
      4'b01??: y = 4'b0100;
      4'b001?: y = 4'b0010;
      4'b0001: y = 4'b0001;
      default: y = 4'b0000;
  endcase
endmodule
```

(See figure 4.25.)

code that appears to work in simulation but synthesizes to incorrect hardware. The optional remainder of this section explains the principles behind the guidelines.

Combinational Logic*

The full adder from HDL Example 4.24 is correctly modeled using blocking assignments. This section explores how it operates and how it would differ if nonblocking assignments had been used.

Imagine that `a`, `b`, and `cin` are all initially 0. `p`, `g`, `s`, and `cout` are thus 0 as well. At some time, `a` changes to 1, triggering the `always/process` statement. The four blocking assignments evaluate in

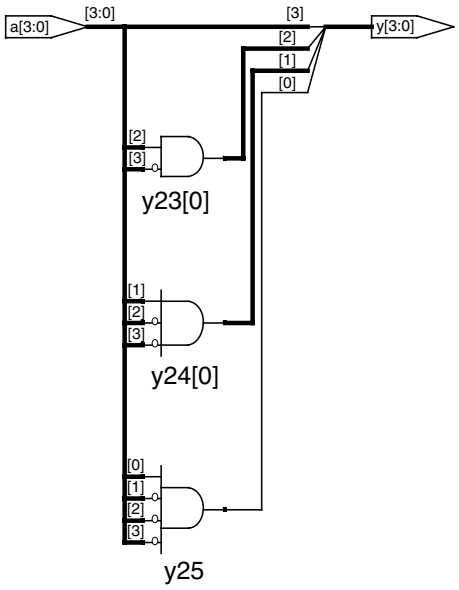


Figure 4.25 `priority_casez` **synthesized circuit**

BLOCKING AND NONBLOCKING ASSIGNMENT GUIDELINES

Verilog

1. Use `always @ (posedge clk)` and nonblocking assignments to model synchronous sequential logic.

```
always @ (posedge clk)
  begin
    n1 <= d; // nonblocking
    q <= n1; // nonblocking
  end
```

2. Use continuous assignments to model simple combinational logic.

```
assign y = s ? d1 : d0;
```

3. Use `always @ (*)` and blocking assignments to model more complicated combinational logic where the `always` statement is helpful.

```
always @ (*)
  begin
    p = a ^ b; // blocking
    g = a & b; // blocking
    s = p ^ cin;
    cout = g | (p & cin);
  end
```

4. Do not make assignments to the same signal in more than one `always` statement or continuous assignment statement.

VHDL

1. Use `process (clk)` and nonblocking assignments to model synchronous sequential logic.

```
process (clk) begin
  if clk'event and clk = '1' then
    n1 <= d; -- nonblocking
    q <= n1; -- nonblocking
  end if;
end process;
```

2. Use concurrent assignments outside `process` statements to model simple combinational logic.

```
y <= d0 when s = '0' else d1;
```

3. Use `process (in1, in2, ... )` to model more complicated combinational logic where the `process` is helpful. Use blocking assignments for internal variables.

```
process (a, b, cin)
  variable p, g: STD_LOGIC;
begin
  p := a xor b; -- blocking
  g := a and b; -- blocking
  s <= p xor cin;
  cout <= g or (p and cin);
end process;
```

4. Do not make assignments to the same variable in more than one `process` or concurrent assignment statement.

the order shown here. (In the VHDL code, s and cout are assigned concurrently.) Note that p and g get their new values before s and cout are computed because of the blocking assignments. This is important because we want to compute s and cout using the new values of p and g.

1. p ← 1 ⊕ 0 = 1
2. g ← 1 • 0 = 0
3. s ← 1 ⊕ 0 = 1
4. cout ← 0 + 1 • 0 = 0

In contrast, HDL Example 4.29 illustrates the use of nonblocking assignments.

Now consider the same case of a rising from 0 to 1 while b and cin are 0. The four nonblocking assignments evaluate concurrently:

p ← 1 ⊕ 0 = 1 g ← 1 • 0 = 0 s ← 0 ⊕ 0 = 0 cout ← 0 + 0 • 0 = 0

Observe that s is computed concurrently with p and hence uses the old value of p, not the new value. Therefore, s remains 0 rather than

HDL Example 4.29 FULL ADDER USING NONBLOCKING ASSIGNMENTS

Verilog

```
// nonblocking assignments (not recommended)
module fulladder (input      a, b, cin,
                  output reg s, cout);
  reg p, g;

  always @ (*)
    begin
      p <= a ^ b; // nonblocking
      g <= a & b; // nonblocking

      s <= p ^ cin;
      cout <= g | (p & cin);
    end
endmodule
```

Because p and g appear on the left hand side of an assignment in an always statement, they must be declared to be reg.

VHDL

```
-- nonblocking assignments (not recommended)
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fulladder is
  port (a, b, cin: in  STD_LOGIC;
        s, cout:   out STD_LOGIC);
end;

architecture nonblocking of fulladder is
  signal p, g: STD_LOGIC;
begin
  process (a, b, cin, p, g) begin
    p <= a xor b; -- nonblocking
    g <= a and b; -- nonblocking

    s <= p xor cin;
    cout <= g or (p and cin);
  end process;
end;
```

Because p and g appear on the left hand side of a nonblocking assignment in a process statement, they must be declared to be signal rather than variable. The signal declaration appears before the begin in the architecture, not the process.

becoming 1. However, p does change from 0 to 1. This change triggers the always/process statement to evaluate a second time, as follows:

p ← 1 ⊕ 0 = 1 g ← 1 • 0 = 0 s ← 1 ⊕ 0 = 1 cout ← 0 + 1 • 0 = 0

This time, p is already 1, so s correctly changes to 1. The nonblocking assignments eventually reach the right answer, but the always/process statement had to evaluate twice. This makes simulation slower, though it synthesizes to the same hardware.

Another drawback of nonblocking assignments in modeling combinational logic is that the HDL will produce the wrong result if you forget to include the intermediate variables in the sensitivity list.

Verilog	VHDL
If the sensitivity list of the always statement in HDL Example 4.29 were written as always @ (a, b, cin) rather than always @ (*), then the statement would not reevaluate when p or g changes. In the previous example, s would be incorrectly left at 0, not 1.	If the sensitivity list of the process statement in HDL Example 4.29 were written as process (a, b, cin) rather than process (a, b, cin, p, g), then the statement would not reevaluate when p or g changes. In the previous example, s would be incorrectly left at 0, not 1.

Worse yet, some synthesis tools will synthesize the correct hardware even when a faulty sensitivity list causes incorrect simulation. This leads to a mismatch between the simulation results and what the hardware actually does.

Sequential Logic*

The synchronizer from HDL Example 4.21 is correctly modeled using nonblocking assignments. On the rising edge of the clock, d is copied to n1 at the same time that n1 is copied to q, so the code properly describes two registers. For example, suppose initially that d = 0, n1 = 1, and q = 0. On the rising edge of the clock, the following two assignments occur concurrently, so that after the clock edge, n1 = 0 and q = 1.

n1 ← d = 0 q ← n1 = 1

HDL Example 4.30 tries to describe the same module using blocking assignments. On the rising edge of clk, d is copied to n1. Then this new value of n1 is copied to q, resulting in d improperly appearing at both n1 and q. The assignments occur one after the other so that after the clock edge, q = n1 = 0.

1. n1 ← d = 0
2. q ← n1 = 0

HDL Example 4.30 BAD SYNCHRONIZER WITH BLOCKING ASSIGNMENTS

Verilog

```
// Bad implementation using blocking assignments

module syncbad (input      clk,
                input      d,
                output reg q);

  reg n1;

  always @ (posedge clk)
    begin
      n1 = d; // blocking
      q = n1; // blocking
    end
endmodule
```

VHDL

```
-- Bad implementation using blocking assignment

library IEEE; use IEEE.STD_LOGIC_1164.all;

entity syncbad is
  port (clk: in  STD_LOGIC;
        d:   in  STD_LOGIC;
        q:   out STD_LOGIC);
end;

architecture bad of syncbad is
begin
  process (clk)
    variable n1: STD_LOGIC;
  begin
    if clk'event and clk = '1' then
      n1 := d; -- blocking
      q <= n1;
    end if;
  end process;
end;
```

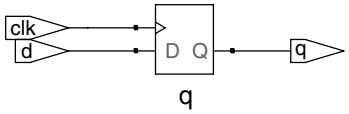


Figure 4.26 `syncbad` **synthesized circuit**

Because `n1` is invisible to the outside world and does not influence the behavior of `q`, the synthesizer optimizes it away entirely, as shown in Figure 4.26.

The moral of this illustration is to exclusively use nonblocking assignment in `always` statements when modeling sequential logic. With sufficient cleverness, such as reversing the orders of the assignments, you could make blocking assignments work correctly, but blocking assignments offer no advantages and only introduce the risk of unintended behavior. Certain sequential circuits will not work with blocking assignments no matter what the order.

4.6 FINITE STATE MACHINES

Recall that a finite state machine (FSM) consists of a state register and two blocks of combinational logic to compute the next state and the output given the current state and the input, as was shown in Figure 3.22. HDL descriptions of state machines are correspondingly divided into three parts to model the state register, the next state logic, and the output logic.

HDL Example 4.31 DIVIDE-BY-3 FINITE STATE MACHINE

Verilog

```
module divideby3FSM(input  clk,
                    input  reset,
                    output y);

  reg [1:0] state, nextstate;

  parameter S0 = 2'b00;
  parameter S1 = 2'b01;
  parameter S2 = 2'b10;

  // state register
  always @ (posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always @ (*)
    case (state)
      S0: nextstate = S1;
      S1: nextstate = S2;
      S2: nextstate = S0;
      default: nextstate = S0;
    endcase

  // output logic
  assign y = (state == S0);
endmodule
```

The `parameter` statement is used to define constants within a module. Naming the states with parameters is not required, but it makes changing state encodings much easier and makes the code more readable.

Notice how a `case` statement is used to define the state transition table. Because the next state logic should be combinational, a `default` is necessary even though the state `2'b11` should never arise.

The output, `y`, is 1 when the state is `S0`. The *equality comparison* `a == b` evaluates to 1 if `a` equals `b` and 0 otherwise. The *inequality comparison* `a != b` does the inverse, evaluating to 1 if `a` does not equal `b`.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity divideby3FSM is
  port(clk, reset: in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of divideby3FSM is
  type statetype is (S0, S1, S2);
  signal state, nextstate: statetype;
begin
  -- state register
  process (clk, reset) begin
    if reset = '1' then state <= S0;
    elsif clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;

  -- next state logic
  nextstate <= S1 when state = S0 else
               S2 when state = S1 else
               S0;

  -- output logic
  y <= '1' when state = S0 else '0';
end;
```

This example defines a new *enumeration* data type, `statetype`, with three possibilities: `S0`, `S1`, and `S2`. `state` and `nextstate` are `statetype` signals. By using an enumeration instead of choosing the state encoding, VHDL frees the synthesizer to explore various state encodings to choose the best one.

The output, `y`, is 1 when the `state` is `S0`. The inequality-comparison uses `/=`. To produce an output of 1 when the state is anything but `S0`, change the comparison to `state /= S0`.

HDL Example 4.31 describes the divide-by-3 FSM from Section 3.4.2. It provides an asynchronous reset to initialize the FSM. The state register uses the ordinary idiom for flip-flops. The next state and output logic blocks are combinational.

The Synplify Pro Synthesis tool just produces a block diagram and state transition diagram for state machines; it does not show the logic gates or the inputs and outputs on the arcs and states. Therefore, be careful that you have specified the FSM correctly in your HDL code. The state transition diagram in Figure 4.27 for the divide-by-3 FSM is analogous to the diagram in Figure 3.28(b). The double circle indicates that

Notice that the synthesis tool uses a 3-bit encoding (*Q*[2:0]) instead of the 2-bit encoding suggested in the Verilog code.

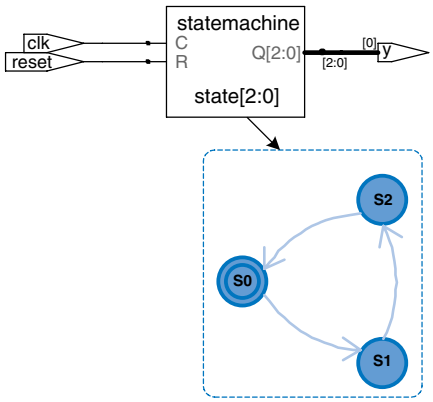


Figure 4.27 `divideby3fsm` **synthesized circuit**

S0 is the reset state. Gate-level implementations of the divide-by-3 FSM were shown in Section 3.4.2.

If, for some reason, we had wanted the output to be HIGH in states S0 and S1, the output logic would be modified as follows.

Verilog

```
// Output Logic
assign y = (state == S0  |  state == S1);
```

VHDL

```
-- output logic
y <= '1' when (state = S0 or state = S1) else '0';
```

The next two examples describe the snail pattern recognizer FSM from Section 3.4.3. The code shows how to use `case` and `if` statements to handle next state and output logic that depend on the inputs as well as the current state. We show both Moore and Mealy modules. In the Moore machine (HDL Example 4.32), the output depends only on the current state, whereas in the Mealy machine (HDL Example 4.33), the output logic depends on both the current state and inputs.

HDL Example 4.32 PATTERN RECOGNIZER MOORE FSM

Verilog

```
module patternMoore(input  clk,
                    input  reset,
                    input  a,
                    output y);

  reg [2:0] state, nextstate;

  parameter S0 = 3'b000;
  parameter S1 = 3'b001;
  parameter S2 = 3'b010;
  parameter S3 = 3'b011;
  parameter S4 = 3'b100;

  // state register
  always @ (posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always @ (*)
    case (state)
      S0: if (a) nextstate = S1;
          else   nextstate = S0;
      S1: if (a) nextstate = S2;
          else   nextstate = S0;
      S2: if (a) nextstate = S2;
          else   nextstate = S3;
      S3: if (a) nextstate = S4;
          else   nextstate = S0;
      S4: if (a) nextstate = S2;
          else   nextstate = S0;
      default:   nextstate = S0;
    endcase

  // output logic
  assign y = (state == S4);
endmodule
```

Note how nonblocking assignments (<=) are used in the state register to describe sequential logic, whereas blocking assignments (=) are used in the next state logic to describe combinational logic.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity patternMoore is
  port(clk, reset: in  STD_LOGIC;
       a:          in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of patternMoore is
  type statetype is (S0, S1, S2, S3, S4);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset = '1' then state <= S0;
    elsif clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;

  -- next state logic
  process(state, a) begin
    case state is
      when S0 => if a = '1' then
                       nextstate <= S1;
                 else nextstate <= S0;
                 end if;
      when S1 => if a = '1' then
                       nextstate <= S2;
                 else nextstate <= S0;
                 end if;
      when S2 => if a = '1' then
                       nextstate <= S2;
                 else nextstate <= S3;
                 end if;
      when S3 => if a = '1' then
                       nextstate <= S4;
                 else nextstate <= S0;
                 end if;
      when S4 => if a = '1' then
                       nextstate <= S2;
                 else nextstate <= S0;
                 end if;
      when others =>   nextstate <= S0;
    end case;
  end process;

  -- output logic
  y <= '1' when state = S4 else '0';
end;
```

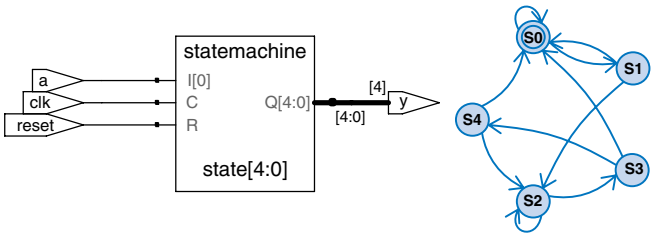


Figure 4.28 `patternMoore` **synthesized circuit**

HDL Example 4.33 PATTERN RECOGNIZER MEALY FSM

Verilog

```
module patternMealy(input  clk,
                    input  reset,
                    input  a,
                    output y);

  reg [1:0] state, nextstate;

  parameter S0 = 2'b00;
  parameter S1 = 2'b01;
  parameter S2 = 2'b10;
  parameter S3 = 2'b11;

  // state register
  always @ (posedge clk, posedge reset)
    if(reset) state <= S0;
    else      state <= nextstate;

  // next state logic
  always @ (*)
    case (state)
      S0: if (a) nextstate = S1;
          else   nextstate = S0;
      S1: if (a) nextstate = S2;
          else   nextstate = S0;
      S2: if (a) nextstate = S2;
          else   nextstate = S3;
      S3: if (a) nextstate = S1;
          else   nextstate = S0;
      default:   nextstate = S0;
    endcase

  // output logic
  assign y = (a & state == S3);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity patternMealy is
  port(clk, reset: in  STD_LOGIC;
       a:          in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of patternMealy is
  type statetype is (S0, S1, S2, S3);
  signal state, nextstate: statetype;
begin
  -- state register
  process(clk, reset) begin
    if reset = '1' then state <= S0;
    elsif clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;

  -- next state logic
  process(state, a) begin
    case state is
      when S0 => if a = '1' then
                      nextstate <= S1;
                 else nextstate <= S0;
         end if;
      when S1 => if a = '1' then
                      nextstate <= S2;
                 else nextstate <= S0;
         end if;
      when S2 => if a = '1' then
                      nextstate <= S2;
                 else nextstate <= S3;
         end if;
      when S3 => if a = '1' then
                      nextstate <= S1;
                 else nextstate <= S0;
         end if;
      when others =>  nextstate <= S0;
    end case;
  end process;

  -- output logic
  y <= '1' when (a = '1' and state = S3) else '0';
end;
```

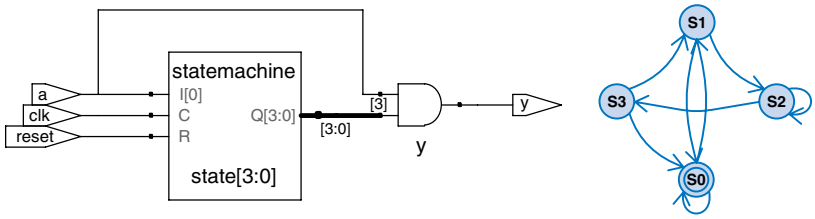


Figure 4.29 `patternMealy` **synthesized circuit**

4.7 PARAMETERIZED MODULES*

So far all of our modules have had fixed-width inputs and outputs. For example, we had to define separate modules for 4- and 8-bit wide 2:1 multiplexers. HDLs permit variable bit widths using parameterized modules.

HDL Example 4.34 PARAMETERIZED *N*-BIT MULTIPLEXERS

Verilog

```
module mux2
  #(parameter width = 8)
   (input  [width-1:0] d0, d1,
    input              s,
    output [width-1:0] y);

   assign y = s ? d1 : d0;
endmodule
```

Verilog allows a `#(parameter ...)` statement before the inputs and outputs to define parameters. The `parameter` statement includes a default value (8) of the parameter, `width`. The number of bits in the inputs and outputs can depend on this parameter.

```
module mux4_8(input  [7:0] d0, d1, d2, d3,
              input  [1:0] s,
              output [7:0] y);

  wire [7:0] low, hi;

  mux2 lowmux(d0, d1, s[0], low);
  mux2 himux(d2, d3, s[1], hi);
  mux2 outmux(low, hi, s[1], y);
endmodule
```

The 8-bit 4:1 multiplexer instantiates three 2:1 multiplexers using their default widths.

In contrast, a 12-bit 4:1 multiplexer, `mux4_12`, would need to override the default width using `#()` before the instance name, as shown below.

```
module mux4_12(input  [11:0] d0, d1, d2, d3,
               input  [1:0]  s,
               output [11:0] y);

  wire [11:0] low, hi;

  mux2 #(12) lowmux(d0, d1, s[0], low);
  mux2 #(12) himux(d2, d3, s[1], hi);
  mux2 #(12) outmux(low, hi, s[1], y);
endmodule
```

Do not confuse the use of the # sign indicating delays with the use of `#(...)` in defining and overriding parameters.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is
  generic(width: integer := 8);
  port(d0,
       d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
       s:  in  STD_LOGIC;
       y:  out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture synth of mux2 is
begin
  y <= d0 when s = '0' else d1;
end;
```

The `generic` statement includes a default value (8) of `width`. The value is an integer.

```
entity mux4_8 is
  port(d0, d1, d2,
       d3: in  STD_LOGIC_VECTOR(7 downto 0);
       s:  in  STD_LOGIC_VECTOR(1 downto 0);
       y:  out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture struct of mux4_8 is
  component mux2
    generic(width: integer);
  port(d0,
       d1: in  STD_LOGIC_VECTOR(width-1 downto 0) ;
       s:  in  STD_LOGIC;
       y:  out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  signal low, hi: STD_LOGIC_VECTOR(7 downto 0);
begin
  lowmux: mux2 port map(d0, d1, s(0), low);
  himux:  mux2 port map(d2, d3, s(0), hi);
  outmux: mux2 port map(low, hi, s(1), y);
end;
```

The 8-bit 4:1 multiplexer, `mux4_8`, instantiates three 2:1 multiplexers using their default widths.

In contrast, a 12-bit 4:1 multiplexer, `mux4_12`, would need to override the default width using `generic map`, as shown below.

```
lowmux: mux2 generic map(12)
             port map(d0, d1, s(0), low);
himux:  mux2 generic map(12)
             port map(d2, d3, s(0), hi);
outmux: mux2 generic map(12)
             port map(low, hi, s(1), y);
```

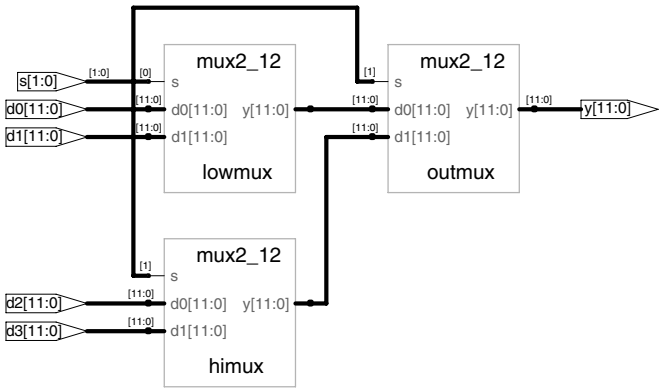


Figure 4.30 `mux4_12` **synthesized circuit**

HDL Example 4.35 PARAMETERIZED $N{:}2^N$ DECODER

Verilog

```
module decoder #(parameter   N = 3)
                (input       [N-1:0]    a,
                 output reg  [2**N-1:0] y);

  always @ (*)
    begin
      y = 0;
      y[a] = 1;
    end
endmodule
```

`2**N` indicates 2^N.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use IEEE.STD_LOGIC_ARITH.all;

entity decoder is
  generic (N: integer := 3);
  port (a: in  STD_LOGIC_VECTOR (N-1 downto 0);
        y: out STD_LOGIC_VECTOR (2**N-1 downto 0));
end;

architecture synth of decoder is
begin
  process (a)
    variable tmp: STD_LOGIC_VECTOR (2**N-1 downto 0);
  begin
    tmp := CONV_STD_LOGIC_VECTOR(0, 2**N);
    tmp (CONV_INTEGER(a)) := '1';
    y <= tmp;
  end process;
end;
```

`2**N` indicates 2^N.

`CONV_STD_LOGIC_VECTOR(0, 2**N)` produces a `STD_LOGIC_VECTOR` of length 2^N containing all 0's. It requires the `STD_LOGIC_ARITH` library. The function is useful in other parameterized functions, such as resettable flip-flops that need to be able to produce constants with a parameterized number of bits. The bit index in VHDL must be an integer, so the `CONV_INTEGER` function is used to convert `a` from a `STD_LOGIC_VECTOR` to an integer.

HDL Example 4.34 declares a parameterized 2:1 multiplexer with a default width of 8, then uses it to create 8- and 12-bit 4:1 multiplexers.

HDL Example 4.35 shows a decoder, which is an even better application of parameterized modules. A large $N{:}2^N$ decoder is cumbersome to

specify with `case` statements, but easy using parameterized code that simply sets the appropriate output bit to 1. Specifically, the decoder uses blocking assignments to set all the bits to 0, then changes the appropriate bit to 1.

HDLs also provide `generate` statements to produce a variable amount of hardware depending on the value of a parameter. `generate` supports `for` loops and `if` statements to determine how many of what types of hardware to produce. HDL Example 4.36 demonstrates how to use `generate` statements to produce an *N*-input AND function from a cascade of two-input AND gates.

Use `generate` statements with caution; it is easy to produce a large amount of hardware unintentionally!

HDL Example 4.36 PARAMETERIZED *N*-INPUT AND GATE

Verilog

```
module andN
  #(parameter width = 8)
    (input [width-1:0] a,
     output            y);

  genvar i;
  wire [width-1:1] x;

  generate
    for (i=1; i<width; i=i+1) begin:forloop
      if (i == 1)
        assign x[1] = a[0] & a[1];
      else
        assign x[i] = a[i] & x[i-1];
    end
  endgenerate
  assign y = x[width-1];
endmodule
```

The `for` statement loops through `i` = 1, 2, . . . , `width-1` to produce many consecutive AND gates. The `begin` in a `generate for` loop must be followed by a : and an arbitrary label (`forloop`, in this case).

Of course, writing `assign y = &a` would be much easier!

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity andN is
  generic (width: integer := 8);
  port (a: in  STD_LOGIC_VECTOR (width-1 downto 0);
        y: out STD_LOGIC);
end;

architecture synth of andN is
  signal i: integer;
  signal x: STD_LOGIC_VECTOR (width-1 downto 1);
begin
  AllBits: for i in 1 to width-1 generate
    LowBit: if i = 1 generate
      A1: x(1) <= a(0) and a(1);
    end generate;
    OtherBits: if i /= 1 generate
      Ai: x(i) <= a(i) and x(i-1);
    end generate;
  end generate;
  y <= x(width-1);
end;
```

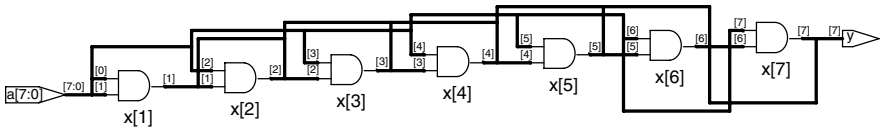


Figure 4.31 `andN` **synthesized circuit**

4.8 TESTBENCHES

Some tools also call the module to be tested the *unit under test* (*UUT*).

A *testbench* is an HDL module that is used to test another module, called the *device under test* (*DUT*). The testbench contains statements to apply inputs to the DUT and, ideally, to check that the correct outputs are produced. The input and desired output patterns are called *test vectors*.

Consider testing the `sillyfunction` module from Section 4.1.1 that computes $y = \bar{a}\bar{b}\bar{c} + a\bar{b}\bar{c} + a\bar{b}c$. This is a simple module, so we can perform exhaustive testing by applying all eight possible test vectors.

HDL Example 4.37 demonstrates a simple testbench. It instantiates the DUT, then applies the inputs. Blocking assignments and delays are used to apply the inputs in the appropriate order. The user must view the results of the simulation and verify by inspection that the correct outputs are produced. Testbenches are simulated the same as other HDL modules. However, they are not synthesizeable.

HDL Example 4.37 TESTBENCH

Verilog

```
module testbench1();
  reg a, b, c;
  wire y;

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // apply inputs one at a time
    initial begin
    a = 0; b = 0; c = 0; #10;
    c = 1;               #10;
    b = 1; c = 0;        #10;
    c = 1;               #10;
    a = 1; b = 0; c = 0; #10;
    c = 1;               #10;
    b = 1; c = 0;        #10;
    c = 1;               #10;
  end
endmodule
```

The `initial` statement executes the statements in its body at the start of simulation. In this case, it first applies the input pattern 000 and waits for 10 time units. It then applies 001 and waits 10 more units, and so forth until all eight possible inputs have been applied. `initial` statements should be used only in testbenches for simulation, not in modules intended to be synthesized into actual hardware. Hardware has no way of magically executing a sequence of special steps when it is first turned on.

Like signals in `always` statements, signals in `initial` statements must be declared to be `reg`.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity testbench1 is -- no inputs or outputs
end;

architecture sim of testbench1 is
  component sillyfunction
    port(a, b, c: in  STD_LOGIC;
         y:       out STD_LOGIC);
  end component;
  signal a, b, c, y: STD_LOGIC;
begin
  -- instantiate device under test
  dut: sillyfunction port map(a, b, c, y);

  -- apply inputs one at a time
  process begin
    a <= '0'; b <= '0'; c <= '0'; wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    b <= '1'; c <= '0';           wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    a <= '1'; b <= '0'; c <= '0'; wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    b <= '1'; c <= '0';           wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    wait; -- wait forever
  end process;
end;
```

The `process` statement first applies the input pattern 000 and waits for 10 ns. It then applies 001 and waits 10 more ns, and so forth until all eight possible inputs have been applied.

At the end, the process waits indefinitely; otherwise, the process would begin again, repeatedly applying the pattern of test vectors.

Checking for correct outputs is tedious and error-prone. Moreover, determining the correct outputs is much easier when the design is fresh in your mind; if you make minor changes and need to retest weeks later, determining the correct outputs becomes a hassle. A much better approach is to write a self-checking testbench, shown in HDL Example 4.38.

HDL Example 4.38 SELF-CHECKING TESTBENCH

Verilog

```
module testbench2();
  reg a, b, c;
  wire y;

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // apply inputs one at a time
  // checking results
  initial begin
    a = 0; b = 0; c = 0; #10;
    if(y !== 1) $display("000 failed.");
    c = 1;               #10;
    if(y !== 0) $display("001 failed.");
    b = 1; c = 0;        #10;
    if(y !== 0) $display("010 failed.");
    c = 1;               #10;
    if(y !== 0) $display("011 failed.");
    a = 1; b = 0; c = 0; #10;
    if(y !== 1) $display("100 failed.");
    c = 1;               #10;
    if(y !== 1) $display("101 failed.");
    b = 1; c = 0;        #10;
    if(y !== 0) $display("110 failed.");
    c = 1;               #10;
    if(y !== 0) $display("111 failed.");
  end
endmodule
```

This module checks `y` against expectations after each input test vector is applied. In Verilog, comparison using `==` or `!=` is effective between signals that do not take on the values of `x` and `z`. Testbenches use the `===` and `!==` operators for comparisons of equality and inequality, respectively, because these operators work correctly with operands that could be `x` or `z`. It uses the `$display` *system task* to print a message on the simulator console if an error occurs. `$display` is meaningful only in simulation, not synthesis.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity testbench2 is -- no inputs or outputs
end;

architecture sim of testbench2 is
  component sillyfunction
    port(a, b, c: in  STD_LOGIC;
         y:       out STD_LOGIC);
  end component;
  signal a, b, c, y: STD_LOGIC;
begin
  -- instantiate device under test
  dut: sillyfunction port map(a, b, c, y);

  -- apply inputs one at a time
  -- checking results
  process begin
    a <= '0'; b <= '0'; c <= '0'; wait for 10 ns;
      assert y = '1' report "000 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '0' report "001 failed.";
    b <= '1'; c <= '0';           wait for 10 ns;
      assert y = '0' report "010 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '0' report "011 failed.";
    a <= '1'; b <= '0'; c <= '0'; wait for 10 ns;
      assert y = '1' report "100 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '1' report "101 failed.";
    b <= '1'; c <= '0';           wait for 10 ns;
      assert y = '0' report "110 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '0' report "111 failed.";
    wait; -- wait forever
  end process;
end;
```

The `assert` statement checks a condition and prints the message given in the `report` clause if the condition is not satisfied. `assert` is meaningful only in simulation, not in synthesis.

Writing code for each test vector also becomes tedious, especially for modules that require a large number of vectors. An even better approach is to place the test vectors in a separate file. The testbench simply reads the test vectors from the file, applies the input test vector to the DUT, waits, checks that the output values from the DUT match the output vector, and repeats until reaching the end of the test vectors file.

HDL Example 4.39 demonstrates such a testbench. The testbench generates a clock using an `always/process` statement with no stimulus list, so that it is continuously reevaluated. At the beginning of the simulation, it reads the test vectors from a text file and pulses `reset` for two cycles. `example.tv` is a text file containing the inputs and expected output written in binary:

```
000_1
001_0
010_0
011_0
100_1
101_1
110_0
111_0
```

HDL Example 4.39 TESTBENCH WITH TEST VECTOR FILE

Verilog

```
module testbench3();
  reg        clk, reset;
  reg        a, b, c, yexpected;
  wire       y;
  reg [31:0] vectornum, errors;
  reg [3:0]  testvectors[10000:0];

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // generate clock
  always
    begin
      clk = 1; #5; clk = 0; #5;
    end

  // at start of test, load vectors
  // and pulse reset
  initial
    begin
      $readmemb("example.tv", testvectors);
      vectornum = 0; errors = 0;
      reset = 1; #27; reset = 0;
    end

  // apply test vectors on rising edge of clk
  always @(posedge clk)
    begin
      #1; {a, b, c, yexpected} =
           testvectors[vectornum];
    end

  // check results on falling edge of clk
  always @(negedge clk)
    if(~reset) begin // skip during reset
      if(y !== yexpected) begin
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use STD.TEXTIO.all;

entity testbench3 is -- no inputs or outputs
end;

architecture sim of testbench3 is
  component sillyfunction
    port(a, b, c: in  STD_LOGIC;
         y:       out STD_LOGIC);
  end component;
  signal a, b, c, y: STD_LOGIC;
  signal clk, reset: STD_LOGIC;
  signal yexpected:  STD_LOGIC;
  constant MEMSIZE: integer := 10000;
  type tvarray is array(MEMSIZE downto 0) of
    STD_LOGIC_VECTOR(3 downto 0);
  signal testvectors: tvarray;
  shared variable vectornum, errors: integer;
begin
  -- instantiate device under test
  dut: sillyfunction port map(a, b, c, y);

  -- generate clock
  process begin
    clk <= '1'; wait for 5 ns;
    clk <= '0'; wait for 5 ns;
  end process;

  -- at start of test, load vectors
  -- and pulse reset
  process is
    file tv: TEXT;
    variable i, j: integer;
    variable L: line;
    variable ch: character;
```

```
      $display ("Error: inputs = %b", {a, b, c});
      $display (" outputs = %b (%b expected)",
                y, yexpected);
      errors = errors + 1;
    end
    vectornum = vectornum + 1;
    if (testvectors[vectornum] === 4'bx) begin
      $display ("%d tests completed with %d errors",
                vectornum, errors);
      $finish;
    end
  end
endmodule
```

`$readmemb` reads a file of binary numbers into the `testvectors` array. `$readmemh` is similar but reads a file of hexadecimal numbers.

The next block of code waits one time unit after the rising edge of the clock (to avoid any confusion if clock and data change simultaneously), then sets the three inputs and the expected output based on the four bits in the current test vector. The next block of code checks the output of the DUT at the negative edge of the clock, after the inputs have had time to propagate through the DUT to produce the output, `y`. The testbench compares the generated output, `y`, with the expected output, `yexpected`, and prints an error if they don't match. `%b` and `%d` indicate to print the values in binary and decimal, respectively. For example, `$display ("%b %b", y, yexpected);` prints the two values, `y` and `yexpected`, in binary. `%h` prints a value in hexadecimal.

This process repeats until there are no more valid test vectors in the `testvectors` array. `$finish` terminates the simulation.

Note that even though the Verilog module supports up to 10,001 test vectors, it will terminate the simulation after executing the eight vectors in the file.

```
begin
  -- read file of test vectors
  i := 0;
  FILE_OPEN (tv, "example.tv", READ_MODE);
  while not endfile (tv) loop
    readline (tv, L);
    for j in 0 to 3 loop
      read (L, ch);
      if (ch = '_') then read (L, ch);
      end if;
      if (ch = '0') then
        testvectors (i) (j) <= '0';
      else testvectors (i) (j) <= '1';
      end if;
    end loop;
    i := i + 1;
  end loop;

  vectornum := 0; errors := 0;
  reset <= '1'; wait for 27 ns; reset <= '0';
  wait;
end process;

-- apply test vectors on rising edge of clk
process (clk) begin
  if (clk'event and clk = '1') then

    a <= testvectors (vectornum) (0) after 1 ns;
    b <= testvectors (vectornum) (1) after 1 ns;
    c <= testvectors (vectornum) (2) after 1 ns;
    yexpected <= testvectors (vectornum) (3)
      after 1 ns;
  end if;
end process;

-- check results on falling edge of clk
process (clk) begin
  if (clk'event and clk = '0' and reset = '0') then
    assert y = yexpected
      report "Error: y = " & STD_LOGIC'image(y);
    if (y /= yexpected) then
      errors := errors + 1;
    end if;
    vectornum := vectornum + 1;
    if (is_x (testvectors(vectornum))) then
      if (errors = 0) then
        report "Just kidding --" &
               integer'image (vectornum) &
               "tests completed successfully."
               severity failure;
      else
        report integer'image (vectornum) &
               "tests completed, errors = " &
               integer'image (errors)
               severity failure;
      end if;
    end if;
  end if;
end process;
end;
```

The VHDL code is rather ungainly and uses file reading commands beyond the scope of this chapter, but it gives the sense of what a self-checking testbench looks like.

New inputs are applied on the rising edge of the clock, and the output is checked on the falling edge of the clock. This clock (and `reset`) would also be provided to the DUT if sequential logic were being tested. Errors are reported as they occur. At the end of the simulation, the testbench prints the total number of test vectors applied and the number of errors detected.

The testbench in HDL Example 4.39 is overkill for such a simple circuit. However, it can easily be modified to test more complex circuits by changing the `example.tv` file, instantiating the new DUT, and changing a few lines of code to set the inputs and check the outputs.

4.9 SUMMARY

Hardware description languages (HDLs) are extremely important tools for modern digital designers. Once you have learned Verilog or VHDL, you will be able to specify digital systems much faster than if you had to draw the complete schematics. The debug cycle is also often much faster, because modifications require code changes instead of tedious schematic rewiring. However, the debug cycle can be much *longer* using HDLs if you don't have a good idea of the hardware your code implies.

HDLs are used for both simulation and synthesis. Logic simulation is a powerful way to test a system on a computer before it is turned into hardware. Simulators let you check the values of signals inside your system that might be impossible to measure on a physical piece of hardware. Logic synthesis converts the HDL code into digital logic circuits.

The most important thing to remember when you are writing HDL code is that you are describing real hardware, not writing a computer program. The most common beginner's mistake is to write HDL code without thinking about the hardware you intend to produce. If you don't know what hardware you are implying, you are almost certain not to get what you want. Instead, begin by sketching a block diagram of your system, identifying which portions are combinational logic, which portions are sequential circuits or finite state machines, and so forth. Then write HDL code for each portion, using the correct idioms to imply the kind of hardware you need.

Exercises

The following exercises may be done using your favorite HDL. If you have a simulator available, test your design. Print the waveforms and explain how they prove that it works. If you have a synthesizer available, synthesize your code. Print the generated circuit diagram, and explain why it matches your expectations.

Exercise 4.1 Sketch a schematic of the circuit described by the following HDL code. Simplify the schematic so that it shows a minimum number of gates.

Verilog

```
module exercise1(input  a, b, c,
                 output y, z);

  assign y = a & b & c  |  a & b & ~c  |  a & ~b & c;
  assign z = a & b  |  ~a & ~b;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity exercise1 is
  port(a, b, c: in  STD_LOGIC;
       y, z:    out STD_LOGIC);
end;

architecture synth of exercise1 is
begin
  y <= (a and b and c) or (a and b and (not c)) or
       (a and (not b) and c);
  z <= (a and b) or ((not a) and (not b));
end;
```

Exercise 4.2 Sketch a schematic of the circuit described by the following HDL code. Simplify the schematic so that it shows a minimum number of gates.

Verilog

```
module exercise2(input      [3:0] a,
                 output reg [1:0] y);

  always @(*)
    if     (a[0]) y = 2'b11;
    else if(a[1]) y = 2'b10;
    else if(a[2]) y = 2'b01;
    else if(a[3]) y = 2'b00;
    else          y = a[1:0];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity exercise2 is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(1 downto 0));
end;

architecture synth of exercise2 is
begin
  process(a) begin
    if    a(0) = '1' then y <= "11";
    elsif a(1) = '1' then y <= "10";
    elsif a(2) = '1' then y <= "01";
    elsif a(3) = '1' then y <= "00";
    else                  y <= a(1 downto 0);
    end if;
  end process;
end;
```

Exercise 4.3 Write an HDL module that computes a four-input XOR function. The input is $a_{3:0}$, and the output is y.

Exercise 4.4 Write a self-checking testbench for Exercise 4.3. Create a test vector file containing all 16 test cases. Simulate the circuit and show that it works. Introduce an error in the test vector file and show that the testbench reports a mismatch.

Exercise 4.5 Write an HDL module called `minority`. It receives three inputs, `a`, `b`, and `c`. It produces one output, `y`, that is TRUE if at least two of the inputs are FALSE.

Exercise 4.6 Write an HDL module for a hexadecimal seven-segment display decoder. The decoder should handle the digits A, B, C, D, E, and F as well as 0–9.

Exercise 4.7 Write a self-checking testbench for Exercise 4.6. Create a test vector file containing all 16 test cases. Simulate the circuit and show that it works. Introduce an error in the test vector file and show that the testbench reports a mismatch.

Exercise 4.8 Write an 8:1 multiplexer module called `mux8` with inputs $s_{2:0}$, `d0`, `d1`, `d2`, `d3`, `d4`, `d5`, `d6`, `d7`, and output `y`.

Exercise 4.9 Write a structural module to compute the logic function, $y = a\bar{b} + \bar{b}\,\bar{c} + \bar{a}bc$, using multiplexer logic. Use the 8:1 multiplexer from Exercise 4.8.

Exercise 4.10 Repeat Exercise 4.9 using a 4:1 multiplexer and as many NOT gates as you need.

Exercise 4.11 Section 4.5.4 pointed out that a synchronizer could be correctly described with blocking assignments if the assignments were given in the proper order. Think of a simple sequential circuit that cannot be correctly described with blocking assignments, regardless of order.

Exercise 4.12 Write an HDL module for an eight-input priority circuit.

Exercise 4.13 Write an HDL module for a 2:4 decoder.

Exercise 4.14 Write an HDL module for a 6:64 decoder using three instances of the 2:4 decoders from Exercise 4.13 and a bunch of three-input AND gates.

Exercise 4.15 Write HDL modules that implement the Boolean equations from Exercise 2.7.

Exercise 4.16 Write an HDL module that implements the circuit from Exercise 2.18.

Exercise 4.17 Write an HDL module that implements the logic function from Exercise 2.19. Pay careful attention to how you handle don't cares.

Exercise 4.18 Write an HDL module that implements the functions from Exercise 2.24.

Exercise 4.19 Write an HDL module that implements the priority encoder from Exercise 2.25.

Exercise 4.20 Write an HDL module that implements the binary-to-thermometer code converter from Exercise 2.27.

Exercise 4.21 Write an HDL module implementing the days-in-month function from Question 2.2.

Exercise 4.22 Sketch the state transition diagram for the FSM described by the following HDL code.

Verilog

```
module fsm2(input  clk, reset,
            input  a, b,
            output y);

  reg [1:0] state, nextstate;

  parameter S0 = 2'b00;
  parameter S1 = 2'b01;
  parameter S2 = 2'b10;
  parameter S3 = 2'b11;

  always @ (posedge clk, posedge reset)
    if(reset) state <= S0;
    else      state <= nextstate;

  always @ (*)
    case(state)
      S0: if(a ^ b)  nextstate = S1;
          else       nextstate = S0;
      S1: if(a & b)  nextstate = S2;
          else       nextstate = S0;
      S2: if(a | b)  nextstate = S3;
          else       nextstate = S0;
      S3: if(a | b)  nextstate = S3;
          else       nextstate = S0;
    endcase

  assign y = (state == S1) | (state == S2);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fsm2 is
  port(clk, reset: in  STD_LOGIC;
       a, b:       in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of fsm2 is
  type statetype is (S0, S1, S2, S3);
  signal state, nextstate: statetype;
begin
  process(clk, reset) begin
    if reset = '1' then state <= S0;
    elsif clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;

  process(state, a, b) begin
    case state is
      when S0 => if(a xor b) = '1' then
                   nextstate <= S1;
                 else nextstate <= S0;
                 end if;
      when S1 => if(a and b) = '1' then
                   nextstate <= S2;
                 else nextstate <= S0;
                 end if;
      when S2 => if(a or b) = '1' then
                   nextstate <= S3;
                 else nextstate <= S0;
                 end if;
      when S3 => if(a or b) = '1' then
                   nextstate <= S3;
                 else nextstate <= S0;
                 end if;
    end case;
  end process;

  y <= '1' when((state = S1) or(state = S2))
       else '0';
end;
```

Exercise 4.23 Sketch the state transition diagram for the FSM described by the following HDL code. An FSM of this nature is used in a branch predictor on some microprocessors.

Verilog

```
module fsm1(input  clk, reset,
            input  taken, back,
            output predicttaken);

  reg [4:0] state, nextstate;

  parameter S0 = 5'b00001;
  parameter S1 = 5'b00010;
  parameter S2 = 5'b00100;
  parameter S3 = 5'b01000;
  parameter S4 = 5'b10000;

  always @(posedge clk, posedge reset)
    if(reset) state <= S2;
    else      state <= nextstate;

  always @(*)
    case(state)
      S0: if(taken) nextstate = S1;
          else      nextstate = S0;
      S1: if(taken) nextstate = S2;
          else      nextstate = S0;
      S2: if(taken) nextstate = S3;
          else      nextstate = S1;
      S3: if(taken) nextstate = S4;
          else      nextstate = S2;
      S4: if(taken) nextstate = S4;
          else      nextstate = S3;
      default:      nextstate = S2;
    endcase

  assign predicttaken = (state == S4)  |  |
                        (state == S3)  |  |
                        (state == S2 && back);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fsm1 is
  port(clk, reset:   in  STD_LOGIC;
       taken, back:  in  STD_LOGIC;
       predicttaken: out STD_LOGIC);
end;

architecture synth of fsm1 is
  type statetype is (S0, S1, S2, S3, S4);
  signal state, nextstate: statetype;
begin
  process(clk, reset) begin
    if reset = '1' then state <= S2;
    elsif clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;

  process(state, taken) begin
    case state is
      when S0 => if taken = '1' then
                      nextstate <= S1;
                 else nextstate <= S0;
                 end if;
      when S1 => if taken = '1' then
                      nextstate <= S2;
                 else nextstate <= S0;
                 end if;
      when S2 => if taken = '1' then
                      nextstate <= S3;
                 else nextstate <= S1;
                 end if;
      when S3 => if taken = '1' then
                      nextstate <= S4;
                 else nextstate <= S2;
                 end if;
      when S4 => if taken = '1' then
                      nextstate <= S4;
                 else nextstate <= S3;
                 end if;
      when others => nextstate <= S2;
    end case;
  end process;

  -- output logic
  predicttaken <= '1' when
                  ((state = S4) or (state = S3) or
                  (state = S2 and back = '1'))
    else '0';
end;
```

Exercise 4.24 Write an HDL module for an SR latch.

Exercise 4.25 Write an HDL module for a *JK flip-flop*. The flip-flop has inputs, *clk*, *J*, and *K*, and output *Q*. On the rising edge of the clock, *Q* keeps its old value if $J = K = 0$. It sets *Q* to 1 if $J = 1$, resets *Q* to 0 if $K = 1$, and inverts *Q* if $J = K = 1$.

Exercise 4.26 Write an HDL module for the latch from Figure 3.18. Use one assignment statement for each gate. Specify delays of 1 unit or 1 ns to each gate. Simulate the latch and show that it operates correctly. Then increase the inverter delay. How long does the delay have to be before a race condition causes the latch to malfunction?

Exercise 4.27 Write an HDL module for the traffic light controller from Section 3.4.1.

Exercise 4.28 Write three HDL modules for the factored parade mode traffic light controller from Example 3.8. The modules should be called `controller`, `mode`, and `lights`, and they should have the inputs and outputs shown in Figure 3.33(b).

Exercise 4.29 Write an HDL module describing the circuit in Figure 3.40.

Exercise 4.30 Write an HDL module for the FSM with the state transition diagram given in Figure 3.65 from Exercise 3.19.

Exercise 4.31 Write an HDL module for the FSM with the state transition diagram given in Figure 3.66 from Exercise 3.20.

Exercise 4.32 Write an HDL module for the improved traffic light controller from Exercise 3.21.

Exercise 4.33 Write an HDL module for the daughter snail from Exercise 3.22.

Exercise 4.34 Write an HDL module for the soda machine dispenser from Exercise 3.23.

Exercise 4.35 Write an HDL module for the Gray code counter from Exercise 3.24.

Exercise 4.36 Write an HDL module for the UP/DOWN Gray code counter from Exercise 3.25.

Exercise 4.37 Write an HDL module for the FSM from Exercise 3.26.

Exercise 4.38 Write an HDL module for the FSM from Exercise 3.27.

Exercise 4.39 Write an HDL module for the serial two's complementer from Question 3.2.

Exercise 4.40 Write an HDL module for the circuit in Exercise 3.28.

Exercise 4.41 Write an HDL module for the circuit in Exercise 3.29.

Exercise 4.42 Write an HDL module for the circuit in Exercise 3.30.

Exercise 4.43 Write an HDL module for the circuit in Exercise 3.31. You may use the full adder from Section 4.2.5.

Verilog Exercises

The following exercises are specific to Verilog.

Exercise 4.44 What does it mean for a signal to be declared `reg` in Verilog?

Exercise 4.45 Rewrite the `syncbad` module from HDL Example 4.30. Use nonblocking assignments, but change the code to produce a correct synchronizer with two flip-flops.

Exercise 4.46 Consider the following two Verilog modules. Do they have the same function? Sketch the hardware each one implies.

```
module code1(input      clk, a, b, c,
             output reg y);
  reg x;

  always @ (posedge clk) begin
    x <= a & b;
    y <= x | c;
  end
endmodule

module code2(input      a, b, c, clk,
             output reg y);
  reg x;

  always @ (posedge clk) begin
    y <= x | c;
    x <= a & b;
  end
endmodule
```

Exercise 4.47 Repeat Exercise 4.46 if the <= is replaced by = in every assignment.

Exercise 4.48 The following Verilog modules show errors that the authors have seen students make in the laboratory. Explain the error in each module and show how to fix it.

(a)
```
module latch(input            clk,
             input      [3:0] d,
             output reg [3:0] q);

    always @ (clk)
      if(clk)q <= d;
endmodule
```

(b)
```
module gates(input      [3:0] a, b,
             output reg [3:0] y1, y2, y3, y4, y5);

    always @ (a)
      begin
        y1 = a & b;
        y2 = a | b;
        y3 = a ^ b;
        y4 = ~(a & b);
        y5 = ~(a | b);
      end
endmodule
```

(c)
```
module mux2(input      [3:0] d0, d1,
            input            s,
            output reg [3:0] y);

    always @ (posedge s)
      if(s)y <= d1;
      else  y <= d0;
endmodule
```

(d)
```
module twoflops(input      clk,
                input      d0, d1,
                output reg q0, q1);

    always @ (posedge clk)
      q1 = d1;
      q0 = d0;
endmodule
```

(e)
```
module FSM(input      clk,
           input      a,
           output reg out1, out2);

    reg state;

    // next state logic and register (sequential)
    always @ (posedge clk)
       if(state == 0)begin
          if(a)  state <= 1;
       end else begin
          if(~a) state <= 0;
       end

    always @ (*)// output logic (combinational)
       if(state == 0)out1 = 1;
       else          out2 = 1;
endmodule
```

(f)
```
module priority(input      [3:0] a,
                output reg [3:0] y);

    always @ (*)
       if      (a[3])y = 4'b1000;
       else if(a[2])y = 4'b0100;
       else if(a[1])y = 4'b0010;
       else if(a[0])y = 4'b0001;
endmodule
```

(g)
```
module divideby3FSM(input  clk,
                    input  reset,
                    output out);

    reg [1:0] state, nextstate;

    parameter S0 = 2'b00;
    parameter S1 = 2'b01;
    parameter S2 = 2'b10;

    // State Register
    always @ (posedge clk, posedge reset)
       if(reset) state <= S0;
       else      state <= nextstate;

    // Next State Logic
    always @ (state)
       case(state)
         S0: nextstate = S1;
         S1: nextstate = S2;
          2: nextstate = S0;
       endcase

    // Output Logic
    assign out =(state == S2);
endmodule
```

(h)
```
module mux2tri(input  [3:0] d0, d1,
               input        s,
               output [3:0] y);

   tristate t0(d0, s, y);
   tristate t1(d1, s, y);
endmodule
```

(i)
```
module floprsen(input            clk,
                input            reset,
                input            set,
                input      [3:0] d,
                output reg [3:0] q);

   always @ (posedge clk, posedge reset)
      if(reset) q <= 0;
      else      q <= d;

   always @ (set)
      if(set)   q <= 1;
endmodule
```

(j)
```
module and3(input      a, b, c,
            output reg y);

   reg tmp;

   always @ (a, b, c)
   begin
      tmp <= a & b;
      y   <= tmp & c;
   end
endmodule
```

VHDL Exercises

The following exercises are specific to VHDL.

Exercise 4.49 In VHDL, why is it necessary to write

```
q <= '1' when state = S0 else '0';
```

rather than simply

```
q <= (state = S0);
```

Exercise 4.50 Each of the following VHDL modules contains an error. For brevity, only the architecture is shown; assume that the `library` use clause and `entity` declaration are correct. Explain the error and show how to fix it.

(a)
```
architecture synth of latch is
begin
  process (clk) begin
     if clk = '1' then q <= d;
     end if;
  end process;
end;
```

(b)
```
architecture proc of gates is
begin
  process (a) begin
     y1 <= a and b;
     y2 <= a or b;
     y3 <= a xor b;
     y4 <= a nand b;
     y5 <= a nor b;
  end process;
end;
```

(c)
```
architecture synth of flop is
begin
  process (clk)
    if clk'event and clk = '1' then
      q <= d;
end;
```

(d)
```
architecture synth of priority is
begin
  process (a) begin
    if    a (3) = '1' then y <= "1000";
    elsif a (2) = '1' then y <= "0100";
    elsif a (1) = '1' then y <= "0010";
    elsif a (0) = '1' then y <= "0001";
    end if;
  end process;
end;
```

(e)
```
architecture synth of divideby3FSM is
  type statetype is (S0, S1, S2);
  signal state, nextstate: statetype;
begin
  process (clk, reset) begin
     if reset = '1' then state <= S0;
     elsif clk'event and clk = '1' then
       state <= nextstate;
     end if;
  end process;
```

```
    process (state) begin
     case state is
        when S0 => nextstate <= S1;
        when S1 => nextstate <= S2;
        when S2 => nextstate <= S0;
     end case;
    end process;

  q <= '1' when state = S0 else '0';
end;
```

(f)
```
architecture struct of mux2 is
  component tristate
    port (a:  in  STD_LOGIC_VECTOR (3 downto 0);
          en: in  STD_LOGIC;
          y:  out STD_LOGIC_VECTOR (3 downto 0));
  end component;
begin
  t0: tristate port map (d0, s, y);
  t1: tristate port map (d1, s, y);
end;
```

(g)
```
architecture asynchronous of flopr is
begin
  process (clk, reset) begin
    if reset = '1' then
      q <= '0';
    elsif clk'event and clk = '1' then
      q <= d;
    end if;
  end process;

  process (set) begin
    if set = '1' then
      q <= '1';
    end if;
  end process;
end;
```

(h)
```
architecture synth of mux3 is
begin
  y <= d2 when s(1) else
       d1 when s(0) else d0;
end;
```

Interview Questions

The following exercises present questions that have been asked at interviews for digital design jobs.

Question 4.1 Write a line of HDL code that gates a 32-bit bus called `data` with another signal called `sel` to produce a 32-bit `result`. If `sel` is TRUE, `result = data`. Otherwise, `result` should be all 0's.

Question 4.2 Explain the difference between blocking and nonblocking assignments in Verilog. Give examples.

Question 4.3 What does the following Verilog statement do?

```
result = | (data[15:0] & 16'hC820);
```