To my family, Amy, Eric, Kelsi, and Maya; and to all engineers who apply their skills to improve the human condition.
Contents

Preface ix
To Students About To Study Digital Design ix
To Instructors of Digital Design ix
How to Use This Book xii
    RTL-Focused Approach xii
    Traditional Approach with Some Reordering xii
    Traditional Approach xii
Acknowledgements xiii
About the Cover xiv
About the Author xiv
Reviewers and Evaluators xv

CHAPTER 1
Introduction 1
1.1 Digital Systems in the World Around Us 1
1.2 The World of Digital Systems 4
    Digital versus Analog 4
    Digital Encodings and Binary Numbers—0s and 1s 9
1.3 Implementing Digital Systems:
    Microprocessors versus Digital Circuits 22
    Software on Microprocessors: The Digital
    Workhorse 22
    Digital Design—When Microprocessors Aren’t
    Good Enough 26
1.4 About this Book 28
1.5 Exercises 29

CHAPTER 2
Combinational Logic Design 35
2.1 Introduction 35
2.2 Switches 36
    Electronics 101 36
    The Amazing Shrinking Switch 37
2.3 The CMOS Transistor 40
2.4 Boolean Logic Gates—Building Blocks for
    Digital Circuits 43
    Boolean Algebra and its Relation to Digital
    Circuits 43
    AND, OR, & NOT Gates 46
    Building Simple Circuits Using Gates 49
2.5 Boolean Algebra 52
    Notation and Terminology 53
    Some Properties of Boolean Algebra 55
    Complementing a Function 60
2.6 Representations of Boolean Functions 61
    Equations 62
    Circuits 62
    Truth Tables 62
    Converting among Boolean Function
    Representations 64
    Standard Representation and Canonical Form 68
    Multiple-Output Combinational Circuits 71
2.7 Combinational Logic Design Process 73
2.8 More Gates 80
    NAND & NOR 80
    XOR & XNOR 81
    Interesting Uses of these Additional Gates 82
    Completeness of NAND and of NOR 82
    Number of Possible Logic Gates 83
2.9 Decoders and Muxes 84
    Decoders 84
    Multiplexers (Muxes) 86
2.10 Additional Considerations 91
    Nonideal Gate Behavior—Delay 91
    Active Low Inputs 92
    Demultiplexers and Encoders 93
    Schematic Capture and Simulation 93
2.11 Combinational Logic Optimizations
    and Tradeoffs (See Section 6.2) 95
2.12 Combinational Logic Description Using
    Hardware Description Languages (See
    Section 9.2) 95
2.13 Chapter Summary 96
2.14 Exercises 96
CHAPTER 5
Register-Transfer Level (RTL) Design 247

5.1 Introduction 247
5.2 High-Level State Machines 248
5.3 RTL Design Process 255
   Step 2A—Creating a Datapath using Components from a Library 259
   Step 2B—Connecting the Datapath to a Controller 262
   Step 2C—Deriving the Controller’s FSM 263
5.4 More RTL Design 264
   Additional Datapath Components for the Library 264
   RTL Design Involving Register Files or Memories 265
   RTL Design Pitfall Involving Storage Updates 271
   RTL Design Involving a Timer 272
   A Data-Dominated RTL Design Example 275
5.5 Determining Clock Frequency 278
5.6 Behavioral-Level Design: C to Gates (Optional) 281
5.7 Memory Components 285
   Random Access Memory (RAM) 286
   Bit Storage in a RAM 288
   Using a RAM 290
   Read-Only Memory (ROM) 292
   ROM Types 294
   Using a ROM 297
   The Blurring of the Distinction between RAM and ROM 299
5.8 Queues (FIFOs) 299
5.9 Multiple Processors 303
5.10 Hierarchy—A Key Design Concept 305
    Managing Complexity 305
    Abstraction 306
    Composing a Larger Component from Smaller Versions of the Same Component 307
5.11 RTL Design Optimizations and Tradeoffs (See Section 6.5) 309
5.12 RTL Design Using Hardware Description Languages (See Section 9.5) 310

5.13 Product Profile: Cell Phone 310
    Cells and Basesstations 310
    How Cellular Phone Calls Work 311
    Inside a Cell Phone 312
5.14 Chapter Summary 316
5.15 Exercises 317

CHAPTER 6
Optimizations and Tradeoffs 325

6.1 Introduction 325
6.2 Combinational Logic Optimizations and Tradeoffs 327
   Two-Level Size Optimization Using Algebraic Methods 327
   Don’t Care Input Combinations 336
   Automating Two-Level Logic-Size Optimization 339
   Multilevel Logic Optimization—Performance and Size Tradeoffs 348
6.3 Sequential Logic Optimizations and Tradeoffs 351
   State Reduction 351
   State Encoding 354
   Moore versus Mealy FSMs 360
6.4 Datapath Component Tradeoffs 365
   Faster Adders 365
   Smaller Multiplier—Sequential (Shift-and-Add) Style 375
6.5 RTL Design Optimizations and Tradeoffs 377
   Pipelining 377
   Concurrency 380
   Component Allocation 381
   Operator Binding 382
   Operator Scheduling 383
   Moore versus Mealy High-Level State Machines 386
6.6 More on Optimizations and Tradeoffs 386
   Serial versus Concurrent Computation 386
   Optimizations and Tradeoffs at Higher versus Lower Levels of Design 387
   Algorithm Selection 388
   Power Optimization 389
Contents

6.7 Product Profile: Digital Video Player/Recorder 393
   Digital Video Overview 393
   DVD—One Form of Digital Video Storage 393
   MPEG-2 Video Encoding—Sending Frame Differences Using I-, P-, and B-Frames 395
   Transforming to the Frequency Domain for Further Compression 396
6.8 Chapter Summary 402
6.9 Exercises 403

CHAPTER 7
Physical Implementation on ICs 413
7.1 Introduction 413
7.2 Manufactured IC Types 414
   Full-Custom Integrated Circuits 414
   Semicustom (Application-Specific) Integrated Circuits—ASICs 415
7.3 Off-the-Shelf Programmable IC Type—FPGA 423
   Lookup Tables 424
   Mapping a Circuit among Multiple Lookup Tables 426
   Programmable Interconnects (Switch Matrices) 432
   Configurable Logic Block 434
   Overall FPGA Architecture 436
7.4 Other Off-the-Shelf IC Types 438
   Off-the-Shelf Logic (SSI) IC 438
   Simple Programmable Logic Device (SPLD) 441
   Complex Programmable Logic Device (CPLD) 445
   FPGA-to-Structured-ASIC Flows 445
7.5 IC Tradeoffs, Trends, and Comparisons 446
   Tradeoffs Among IC Types 447
   IC Technology Trend—Moore’s Law 448
   Relative Popularity of IC Types 450
   ASSPs 450
   IC Types versus Processor Varieties 451
   FPGAs alongside Microprocessors 452
7.6 Product Profile: Giant LED-Based Video Display with FPGAs 453
7.7 Chapter Summary 457
7.8 Exercises 457

CHAPTER 8
Programmable Processors 461
8.1 Introduction 461
8.2 Basic Architecture 462
   Basic Datapath 462
   Basic Control Unit 465
8.3 A Three-Instruction Programmable Processor 469
   A First Instruction Set with Three Instructions 469
   Control Unit and Datapath for the Three-Instruction Processor 471
8.4 A Six-Instruction Programmable Processor 475
   Extending the Instruction Set 475
   Extending the Control Unit and Datapath 476
8.5 Example Assembly and Machine Programs 478
8.6 Further Extensions to the Programmable Processor 480
   Instruction Set Extensions 480
   Input/Output Extensions 481
   Performance Extensions 481
8.7 Chapter Summary 482
8.8 Exercises 483

CHAPTER 9
Hardware Description Languages 487
9.1 Introduction 487
9.2 Combinational Logic Description Using Hardware Description Languages 489
   Structure 489
   Combinational Behavior 494
   Testbenches 498
9.3 Sequential Logic Description Using Hardware Description Languages 501
   Register 501
   Oscillator 503
   Controllers 505
9.4 Datapath Component Description Using Hardware Description Languages 509
   Full-Adders 509
9.5 RTL Design Using Hardware Description Languages 517
   High-Level State Machine of the Laser-Based Distance Measurer 517
   Controller and Datapath of the Laser-Based Distance Measurer 523
9.6 Chapter Summary 532
9.7 Exercises 532

APPENDIX A
Boolean Algebras 537
A.1 Boolean Algebra 537
A.2 Switching Algebra 538
A.3 Important Theorems in Boolean Algebra 540
A.4 Other Examples of Boolean Algebras 545
A.5 Further Readings 545

APPENDIX B
Additional Topics in Binary Number Systems 547
B.1 Introduction 547
B.2 Real Number Representation 547
B.3 Fixed Point Arithmetic 550
B.4 Floating Point Representation 551
   The IEEE 754-1985 Standard 552
B.5 Exercises 556

APPENDIX C
Extended RTL Design Example 557
C.1 Introduction 557
C.2 Designing the Soda Dispenser Controller 558
C.3 Understanding the Behavior of the Soda Dispenser Controller and Datapath 562
image not available
1.2 THE WORLD OF DIGITAL SYSTEMS

Digital versus Analog

A digital signal, also known as a discrete signal, is a signal that at any time can have one of a finite set of possible values. In contrast, an analog signal can have one of an infinite number of possible values, and is also known as a continuous signal. A signal is just some physical phenomenon that has a unique value at every instant of time. An everyday example of an analog signal is the temperature outside, because physical temperature is a continuous value—the temperature may be 92.356666... degrees. An everyday example of a digital signal is the number of fingers you hold up, because the value must be either 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, or 10—a finite set of values. In fact, the term “digital” comes from the Latin word for “digit” (digitus), meaning finger.

In computing systems, the most common digital signals are those that can have one of only two possible values, like on or off (often represented as 1 or 0). Such a two-valued representation is known as a binary representation. A digital system is a system that takes digital inputs and generates digital outputs. A digital circuit is a connection of digital components that together comprise a digital system. In this textbook, the term “digital” will refer to systems with binary-valued signals. A single binary signal is known as a binary digit, or bit for short (binary digit). Digital electronics became extremely popular in the mid-1900s after the invention of the transistor, an electric switch that can be turned on or off using another electric signal. We’ll describe transistors further in the next chapter.

Digital Circuits are the Basis for Computers

The most well-known use of digital circuits in the world around us is probably to build the microprocessors that serve as the brain of general-purpose computers, like the personal computer or laptop computer that you might have at home, illustrated in Figure 1.1(a). General-purpose computers are also used as servers, which operate behind the scenes to implement banking, airline reservation, web search, payroll, and similar such systems. General-purpose computers take digital input data, such as letters and numbers received from files or keyboards, and output new digital data, such as new letters and numbers stored in files or displayed on a monitor. Learning about digital design is therefore useful in understanding how computers work “under the hood,” and hence has been required learning for most computing and electrical engineering majors for decades. Based on material in upcoming chapters, we’ll design a simple computer in Chapter 8.

Digital Circuits are the Basis for Much More

Increasingly, digital circuits are being used for much more than implementing general-purpose computers. More and more new applications convert analog signals to digital ones, and run those digital signals through customized digital circuits, to achieve numerous benefits. Such applications, such as those in Figure 1.1(b), include cell phones, automobile engine controllers, TV set-top boxes, music instruments, digital cameras and camcorders, video game consoles, and so on. Digital circuits found inside applications other than general-purpose computers are often called embedded systems, because those digital systems are embedded inside another electronic device.
1 Introduction

Figure 1.4 More and more analog products are becoming primarily digital.

Figure 1.4, over the past decade numerous products that were previously analog have converted primarily to digital technology. Portable music players, for example, switched from cassette tapes to digital CDs in the middle 1990s, and recently to MP3s and other digital formats. Early cell phones used analog communication, but in the late 1990s digital communication, similar in idea to that shown in Figure 1.3, became dominant. In the early 2000s, analog VHS video players gave way to digital video disc (DVD) players, and then to hard-drive-based digital video recorders (DVRs). Portable video cameras have begun to digitize video before storing the video onto tape or a hard drive, while still picture cameras have eliminated film and store photos on digital cards. Musical instruments are increasingly digitized-based, with electronic drums, keyboards, and electric guitars including more digital processing. Analog TV is also giving way to digital TV. Hundreds of other devices have converted from analog to digital in past decades, such as clocks and watches, household thermostats, human temperature thermometers (which now work in the ear rather than under the tongue or other places), car engine controllers, gasoline pumps, hearing aids, and more. Many other devices were never analog, instead being introduced in digital form from the very start. For example, video games have been digital since their inception.

The above devices use digitization, and digitization requires that phenomena be encoded into 1s and 0s. Computations using digital circuits also require that numbers be digitized into 1s and 0s. The next section describes how to encode items digitally.

**THE TELEPHONE.**

The telephone, patented by Alexander Graham Bell in the late 1800s (though invented by Antonio Meucci), operates using the electromagnetic principle described earlier—your speech creates sound waves that move a membrane, which moves a magnet, which creates current on a nearby wire. Run that wire to somewhere far away, put a magnet connected to a membrane near that wire, and the membrane will move, producing sound waves that sound like you talking. Much of the telephone system today digitizes the audio to improve quality and quantity of audio transmissions over long distances. A couple of interesting facts about the telephone:

- Believe it or not, Western Union actually turned down Bell’s initial proposal to develop the telephone, perhaps thinking that the then-popular telegraph was all people needed.
- Bell and his assistant Watson disagreed on how to answer the phone: Watson wanted “Hello,” which won, but Bell wanted “Hoy hoy” instead. (Fans of the TV show *The Simpsons* may have noticed that Homer’s boss, Mr. Burns, answers the phone with a “hoy hoy.”

(Source of some of the above material: www.pbs.org, transcript of “The Telephone”).

An early-style telephone.
image not available
12  Introduction

**NAMES IN BASE TEN.**

English speakers use names for various quantities in base ten, names that are useful but can hamper gaining an intuitive understanding of base ten. $10^0$ has its own name: *hundred*. $10^1$ has the name *thousand*. There is no name (in American English) for $10^2$ or $10^3$. $10^4$ has the name *million*, and subsequent groups that are multiples of 1,000 have the names *billion*, *trillion*, *quadrillion*, etc. English speakers also use abbreviated names for groups of tens—the numbers 10, 20, 30, ..., 90 could be called one ten, two ten, up to nine ten, but instead have abbreviated names: one ten as just “ten,” two ten is “twenty,” up to nine ten being “ninety.” You can see how “ninety” is a shortening of “nine ten.” Special names are also used for the numbers between 10 and 20. 11 could be “one ten one,” but is instead “eleven,” while 19 could be “one ten nine” but is instead “nineteen.” Table 1.1 indicates how one might count in base ten without these various names, to emphasize the nature of base ten. 523 might be spoken as “five hundred two ten three” rather than “five hundred twenty-three.” Kids may have a harder time learning math because of the arbitrary base ten names—for example, carrying a one from the ones column to the tens column makes more sense if the ones column sums to “one ten seven” rather than to “seventeen”—“one ten seven” obviously adds one to the tens column. Likewise, learning binary may be slightly harder for some students due to a lack of a solid understanding of base ten. To help remedy the situation, perhaps when a store clerk tells you “That will be ninety-nine cents,” you might say “You mean nine ten nine cents.” If enough of us do this, perhaps it will catch on?

Table 1.1 Counting in base ten without the abbreviated or short names.

<table>
<thead>
<tr>
<th></th>
<th>As usual: “zero,” “one,” “two,” ..., “nine.”</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 9</td>
<td>10, 11, 12, ..., 19: “one ten,” “one ten one,” “one ten two,” ..., “one ten nine”</td>
</tr>
<tr>
<td>10 to 99</td>
<td>20, 21, 22, ..., 29: “two ten,” “two ten one,” “two ten two,” ..., “two ten nine”</td>
</tr>
<tr>
<td></td>
<td>30, 31, 32, ..., 39: “three ten,” “four ten,” ..., “nine ten”</td>
</tr>
<tr>
<td>100 to 900</td>
<td>As usual: “one hundred,” “two hundred,” ..., “nine hundred.” Even clearer would be to replace the word “hundred” by “ten to the power of 2.”</td>
</tr>
<tr>
<td>1000 and up</td>
<td>As usual. Even clearer for understanding bases: replace “thousand” by “ten to the (power of) 3,” “ten thousand” by “ten to the 4,” etc., eliminating the various names.</td>
</tr>
</tbody>
</table>

---

**Example 1.1 Using digital data in a digital system**

A digital system is desired that reads the value of a temperature sensor and shows the letter “F” (for “freezing”) on a display if the temperature is 32 degrees Fahrenheit or below, shows “N” (for “normal”) if the temperature is between 32 and 212 degrees, and shows the letter “B” (for “boiling”) if the temperature is 212 or greater. The temperature sensor has an 8-bit output representing the temperature as a binary number between 0 and 255. The display has a 7-bit input that accepts an ASCII bit encoding and displays the corresponding symbol.

Figure 1.13 shows the temperature sensor output connected to the input of the desired digital system. Each wire can have

![Digital System Diagram](image-url)
image not available
image not available
Figure 1.17 Decimal to binary conversion for a DIP switch: (a) ceiling fan with remote control, both having DIP switches to set their communication channel, (b) setting the fan’s channel to “73” requires first converting 73 to binary, then setting the DIP switch to represent that binary value, (c) ceiling fan module only outputs 1 if the received channel matches DIP switch setting.

Instead, he/she can open the ceiling fan module and the remote controller of one pair, and simply change the DIP switch settings for that pair, ensuring that both DIP switches match after the change.

While this section introduced the addition method for converting from decimal to binary, many books and web resources introduce the subtraction method, wherein we start by setting a current number to the desired decimal number, put a 1 in the highest binary number place that doesn’t exceed the current number, subtract that place’s weight from the current number, and repeat until the current number reaches zero. The two methods are fundamentally the same; the addition method may be more intuitive when converting by hand.

Hexadecimal and Octal Numbers
Base sixteen numbers, known as hexadecimal numbers or just hex, are also popular in digital design, mainly because one base sixteen digit is equivalent to four base two digits, making hexadecimal numbers a nice shorthand representation for binary numbers. In base sixteen, the first digit represents up to fifteen ones—the sixteen symbols commonly used are 0, 1, 2, ..., 9, A, B, C, D, E, F (so A = ten, B = eleven, C = twelve, D = thirteen, E = fourteen, and F = fifteen). The next digit represents the number of groups of 16¹, the next digit the number of groups of 16², etc., as shown in Figure 1.18. So 8AF₁₆ equals 8*16² + 10*16¹ + 15*16⁰, or 2223₁₀
image not available
1110. Checking the answer shows that 1110 is correct: $1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 8 + 4 + 2 + 0 = 14$.

To convert 99 to binary, the process is the same but naturally takes more steps: $99/2 = 49$ remainder 1, $49/2 = 24$, remainder 1, $24/2 = 12$, remainder 0, $12/2 = 6$, remainder 0, $6/2 = 3$, remainder 0, $3/2 = 1$, remainder 1, $1/2 = 0$, remainder 1. Combining the remainders together gives us the binary number 1100011. We know from Example 1.3 that this is the correct answer.

We can use the same basic method to convert a base 10 number to a number in any base. To convert a number from base 10 to base $n$, we repeatedly divide the number by $n$ and place the remainder in the new base $n$ number, starting from the least significant digit. The method is called the divide-by-$n$ method.

**Example 1.9 Decimal to arbitrary bases using the divide-by-$n$ method**

Convert the number 3439 to base 10 and to base 7.

We know the number 3439 is 3439 in base 10, but let’s use the divide-by-$n$ method (where $n$ is 10) to illustrate that the method works for any base. We start by dividing 3439 by 10: $3439/10 = 343$, remainder 9. We then divide the quotient by 10: $343/10 = 34$, remainder 3. We do the same with the new quotient: $34/3 = 3$, remainder 4. Finally, we divide 3 by 10: $3/10 = 0$, remainder 3. Combining the remainders, least significant digit first, gives us the base 10 number 3439.

To convert 3439 to base 7, the approach is similar, except we now divide by 7. We begin by dividing 3439 by 7: $3439/7 = 491$, remainder 2. Continuing our calculations, we get $491/7 = 70$, remainder 1, $70/7 = 10$, remainder 0, $10/7 = 1$, remainder 3, $1/7 = 0$, remainder 1. Thus, 3439 in base 7 is 13012. Checking the answer verifies that we have the correct result: $1 \cdot 7^3 + 3 \cdot 7^2 + 0 \cdot 7^1 + 1 \cdot 7^0 = 2401 + 1029 + 7 + 2 = 3439$.

Conversion between any two bases can be done by first converting to base ten, then converting the base ten number to the desired base using the divide-by-$n$ method.
image not available
image not available
Figure 1.25 shows an example of the values of signals \( a \), \( b \), and \( F \) over time, with time proceeding to the right. As time proceeds, each signal may be either 0 or 1, illustrated by each signal’s associated line being either low or high. We made \( a \) equal to 0 until time 7:05, when we made \( a \) become 1. We made \( a \) stay 1 until 7:06, when we made \( a \) return back to 0. We made \( a \) stay 0 until 9:00, when we made \( a \) become 1 again, and then we made \( a \) become 0 at 9:01. On the other hand, we made \( b \) start as 0, and then become 1 between 7:06 and 9:00. The diagram shows what the value of \( F \) would be given the C program executing on the microprocessor—when \( a \) is 1 and \( b \) is 0 (from 7:05 to 7:06), \( F \) will be 1. A diagram with time proceeding to the right, and the values of digital signals shown by high or low lines, is known as a timing diagram. We draw the input lines (\( a \) and \( b \)) to be whatever values we want, but then the output line (\( F \)) must describe the behavior of the digital system.

---

**Example 1.10 Outdoor motion notifier using a microprocessor**

Let’s use the basic microprocessor of Figure 1.23 to implement a system that sounds a buzzer when motion is detected at any of the three motion sensors outside a house. We connect the motion sensors to microprocessor input pins \( I0, I1, \) and \( I2 \), and connect output pin \( P0 \) to a buzzer (Figure 1.26). (We assume the motion sensors and buzzers have appropriate electronic interfaces to the microprocessor pins.) We can then write the following C program:

```c
void main()
{
    while (1) {
        P0 = I0 || I1 || I2;
    }
}
```

The program executes the statement inside the while loop repeatedly. That statement will set \( P0 \) to 1 if \( I0 \) is 1 or \( I1 \) or \( I2 \) is 1, otherwise the statement sets \( P0 \) to 0.
image not available
image not available
1.22 Convert the following hexadecimal numbers to binary:
   (a) 4F5E
   (b) 3FAD
   (c) 3E2A
   (d) DEED

1.23 Convert the following hexadecimal numbers to binary:
   (a) B0C4
   (b) 1EF03
   (c) F002
   (d) BEEF

1.24 Convert the following hexadecimal numbers to decimal:
   (a) FF
   (b) F0A2
   (c) 0F100
   (d) 100

1.25 Convert the following hexadecimal numbers to decimal:
   (a) 10
   (b) 4E3
   (c) FF0
   (d) 200

1.26 Convert the decimal number 128 to the following number systems:
   (a) binary
   (b) hexadecimal
   (c) base three
   (d) base five
   (e) base fifteen

1.27 Compare the number of digits necessary to represent the following decimal numbers in binary, octal, decimal, and hexadecimal representations. You need not determine the actual representations—just the number of required digits. For example, representing the decimal number 12 requires four digits in binary (1100 is the actual representation), two digits in octal (14), two digits in decimal (12), and one digit in hexadecimal (C).
   (a) 8
   (b) 60
   (c) 300
   (d) 1000
   (e) 999,999

1.28 Determine the decimal number ranges that can be represented in binary, octal, decimal, and hexadecimal using the following numbers of digits. For example, 2 digits can represent decimal number range 0 through 3 in binary (00 through 11), 0 through 63 in octal (00 through 77), 0 through 99 in decimal (00 through 99), and 0 through 255 in hexadecimal (00 through FF).
   (a) 1
   (b) 3
   (c) 6
   (d) 8
image not available
HOW TRANSISTORS ARE MADE SO SMALL—USING PHOTOGRAPHIC METHODS

If you took a pencil and made the smallest dot that you could on a sheet of paper, that dot's area would hold many thousands of transistors on a modern silicon chip. How can chip makers create such tiny transistors? The key lies in photographic methods. Chip makers lay a special chemical onto the chip—special because the chemical changes when exposed to light. Chip makers then shine light through a lens that focuses the light down to extremely small regions on the chip—similar to how a microscope’s lens lets us see tiny things by focusing light, but in reverse. The chemical in the small illuminated region changes, and then a solvent washes away the chemical—but some regions stay because of the light that changed that region. Those remaining regions form parts of transistors. Repeating this process over and over again, with different chemicals at different steps, results not only in transistors, but also wires connecting the transistors, and insulators preventing crossing wires from touching.

Photograph of a Pentium processor’s silicon chip, having millions of transistors. Actual size is about 1 cm each side.

Atom or Celeron processor, requires only about 50 million transistors, and the processor in a cell phone, like an ARM processor, may have only a few million transistors. Many of today’s high-end chips, like chips inside Internet routers, contain tens or hundreds of such microprocessors, and can conceivably contain thousands of even smaller microprocessors (or just a few very big microprocessors).

IC density has been doubling roughly every 18 months since the 1960s. The doubling of IC density every 18 months is widely known as Moore’s Law, named after Gordon Moore, a co-founder of Intel Corporation, who made predictions back in 1965 that the number of components per IC would double every year or so. At some point, chip makers won’t be able to shrink transistors any further. After all, the transistor has to at least be wide enough to let electrons pass through. People have been predicting the end of Moore’s Law for two decades now, but transistors keep shrinking, though in 2009 many observers noted a slowdown.

Not only do smaller transistors and wires provide for more functionality in a chip, but they also provide for faster circuits, in part because electrons need not travel as far to get from one transistor to the next. This increased speed is the main reason why personal computer clock speeds have improved so drastically over the past few decades, from kilohertz frequencies in the 1970s to gigahertz frequencies in the 2000s.

2.3 THE CMOS TRANSISTOR

The most popular type of IC transistor is the CMOS transistor. A detailed explanation of how a CMOS transistor works is beyond the scope of this book, but nevertheless a simplified explanation may satisfy much curiosity.

A chip is made primarily from the element silicon. A chip, also known as an integrated circuit, or IC, is typically about the size of a fingernail. Even if you open up a computer or other chip-based device, you would not actually see the silicon chip, since chips are actually inside a larger, usually black, protective package. But you certainly
image not available
2 Combinational Logic Design

“ab=01” is shorthand for “a=0, b=1.”

- OR returns 1 if either or both of its operands are 1. So the result of a OR b is 1 in any of the following cases: ab=01, ab=10, ab=11. Thus, the only time a OR b is 0 is when ab=00.
- NOT returns 1 if its operand is 0. So NOT(a) returns 1 if a is 0, and returns 0 if a is 1.

We use Boolean logic operators frequently in everyday thought, such as in the statement “I’ll go to lunch if Mary goes OR John goes, AND Sally does not go.” To represent this using Boolean concepts, let F represent my going to lunch (F=1 means I’ll go to lunch, F=0 means I won’t go). Let Boolean variables m, j, and s represent Mary, John, and Sally each going to lunch (so s=1 would represent Sally going to lunch, else s=0). Then we can translate the above English sentence into the Boolean equation:

\[ F = (m \text{ OR } j) \text{ AND NOT(} s\text{)} \]

So F will equal 1 if either m or j is 1, and s is 0. Now that we’ve translated the English sentence into a Boolean equation, we can perform several mathematical activities with that equation. One thing we can do is determine the value of F for different values of m, j, and s:

- m=1, j=0, s=1 \rightarrow F = (1 \text{ OR } 0) \text{ AND NOT(1)} = 1 \text{ AND } 0 = 0
- m=1, j=1, s=0 \rightarrow F = (1 \text{ OR } 1) \text{ AND NOT(0)} = 1 \text{ AND } 1 = 1

In the first case, I don’t go to lunch; in the second, I do.

A second thing we could do is apply some algebraic rules (discussed later) to modify the original equation to the equivalent equation:

\[ F = (m \text{ and NOT(} s\text{)}) \text{ OR (} j \text{ and NOT(} s\text{)}) \]

In other words, I’ll go to lunch if Mary goes AND Sally does not go, OR if John goes AND Sally does not go. That statement, as different as it may look from the earlier statement, is nevertheless equivalent to the earlier statement.

A third thing we could do is formally prove properties about the equation. For example, we could prove that if Sally goes to lunch (s=1), then I don’t go to lunch (F=0) no matter who else goes, using the equation:

\[ F = (m \text{ OR } j) \text{ AND NOT(1)} = (m \text{ OR } j) \text{ AND } 0 = 0 \]

No matter what the values of m and j, F will equal 0.

Noting all the mathematical activities we can do using Boolean equations, you can start to see what Boole was trying to accomplish in formalizing human reasoning.

**Example 2.1** Converting a problem statement to a Boolean equation

Convert the following problem statements to Boolean equations using AND, OR, and NOT operators. F should equal 1 only if:

1. a is 1 and b is 1. Answer: \( F = a \text{ AND } b \)
2. either a or b is 1. Answer: \( F = a \text{ OR } b \)
Example 2.8 Seat belt warning light with initial illumination

Let's further extend the previous example. Automobiles typically light up all their warning lights when you first turn the key, so that you can check that all the warning lights are working. Assume that the system receives an input \( t \) that is 1 for the first 5 seconds after a key is inserted into the ignition, and 0 afterward (don't worry about who or what sets \( t \) in that way). So the system should set \( w = 1 \) when \( p = 1 \) and \( s = 0 \) and \( k = 1 \), OR when \( t = 1 \). Note that when \( t = 1 \), the circuit should illuminate the light, regardless of the values of \( p, s, \) and \( k \). The new circuit equation is:

\[
  w = (p \text{ AND NOT}(s) \text{ AND } k) \text{ OR } t
\]

The circuit is shown in Figure 2.25.

Some circuit drawing rules and conventions

There are some rules and conventions that designers commonly follow when drawing circuits of logic gates, as shown in Figure 2.26.

- Logic gates have one or more inputs and one output, but each input and output is typically not labeled. Remember: the order of the inputs into a gate doesn't affect the gate's logical behavior.
- Each wire has an implicit direction, from one gate's output to another gate's input, but we typically don't draw arrows showing each direction.
- A single wire can be branched out into two (or more) wires going to multiple gate inputs—the branches have the same value as the single wire. But two wires can NOT be merged into one wire—what would be the value of that one wire if the incoming two wires had different values?

2.5 BOOLEAN ALGEBRA

Logic gates are useful for implementing circuits, but equations are better for manipulating circuits. The algebraic tools of Boolean algebra enable us to manipulate Boolean equations so we can do things like simplify the equations, check whether two equations are equivalent, find the inverse of an equation, prove properties about the equations, etc. Since a Boolean equation consisting of AND, OR, and NOT operations can be straightforwardly transformed into a circuit of AND, OR, and NOT gates, manipulating Boolean equations can be considered as manipulating digital circuits. We'll informally introduce some of the most useful algebraic tools of Boolean algebra. Appendix A provides a formal definition of Boolean algebra.
image not available
2 Combinational Logic Design

- **Identity**
  
  \[
  0 + a = a + 0 = a \\
  1 * a = a * 1 = a
  \]
  
  This one should be intuitive. ORing a with 0 (a+0) just means that the result will be whatever a is. After all, 1+0 is 1, while 0+0 is 0. Likewise, ANDing a with 1 (a*1) results in a. 1*1 is 1, while 0*1 is 0.

- **Complement**
  
  \[
  a + a' = 1 \\
  a * a' = 0
  \]
  
  This also makes intuitive sense. Regardless of the value of a, a’ is the opposite, so you get a 0 and a 1, or you get a 1 and a 0. One of (a, a’) will always be a 1, so ORing them (a+a’) must yield a 1. Likewise, one of (a, a’) will always be a 0, so ANDing them (a*a’) must yield a 0.

The following examples apply these basic properties to some digital design examples to see how the properties can help.

**Example 2.11 Applying the basic properties of Boolean algebra**

Use the properties of Boolean algebra for the following problems:

- Show that abc’ is equivalent to c’ba.
  
  The commutative property allows swapping the operands being ANDed, so a*b*c’ = a*c’*b = c’*a*b = c’*b*a = c’ba.

- Show that abc + abc’ = ab.
  
  The first distributive property allows factoring out the ab term: abc + abc’ = ab(c+c’). Then, the complement property allows replacing the c+c’ by 1: ab(c+c’) = ab(1). Finally, the identity property allows removal of the 1 from the AND term: ab(1) = ab*1 = ab.

- Show that the equation x + x’z is equivalent to x + z.
  
  The second distributive property (the tricky one) allows replacing x+x’z by (x+x’)’(x+z). The complement property allows replacing (x+x’) by 1, and the identity property allows replacing 1*(x+z) by x+z.

- Show that (a+a’)bc is just bc.
  
  The complement property states that (a+a’) is 1, yielding 1*bc. The identity property then results in bc.

- Multiply out (w + x) (y + z) into sum-of-products form.
  
  First writing (w + x) as A will make clear that the distributive property can be applied: A(y+z). The first distributive property yields Ay + Az. Expanding A back yields (w+x)y + (w+x)z. Applying the first distributive property again yields wy + xy + wz + xz, which is in sum-of-products form.
image not available
image not available
output for every possible input. Thus, notice that truth tables were used in Figure 2.8 to
describe in an intuitive manner the behavior of basic logic gates.

A drawback of truth tables is that for a large number of inputs, the number of truth
table rows can be very large. Given a function with \( n \) inputs, the number of input combi-
nations is \( 2^n \). A function with 10 inputs would have \( 2^{10} = 1024 \) possible input
combinations—you can’t easily see much of anything in a table having 1024 rows. A
function with 16 inputs would have 65,536 rows in its truth table.

**Example 2.18 Capturing a function as a truth table**

```
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Create a truth table describing a function that detects whether a
three-bit inputs’ value, representing a binary number, is 5 or
greater. Table 2.2 shows a truth table for the function. We first
list all possible combinations of the three input bits, which
we’ve labeled a, b, and c. We then enter a 1 in the output row
if the inputs represent 5, 6, or 7 in binary, meaning the last
three rows. We enter 0s in all the other rows.

**Converting among Boolean Function Representations**

Given the above representations, converting from
one representation to another is sometimes necessary
or useful. For the three representations discussed so
far (equations, circuits, and truth tables), there are
six possible conversions from one representation to
another, as shown in Figure 2.36, which will now be
described.

1. **Equations to Circuits**
   Converting an equation to a circuit can be done
   straightforwardly by using an AND gate for every
   AND operator, an OR gate for every OR operator, and a NOT gate for every NOT oper-
   ator. Several examples of such conversions appear in Section 2.4.

2. **Circuits to Equations**
   Converting a circuit into an equation can be done by starting from the circuit’s inputs, and
   then writing the output of each gate as an expression involving the gate’s inputs. The
   expression of the last gate before the output represents the expression for the circuit’s
   function.
2 Combinational Logic Design

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 2.42 Truth table for the circuit's equation.

From the Boolean equation, we can now construct the truth table for the combinational circuit. Since our circuit has three inputs—a, b, and c—there are \(2^3 = 8\) possible combinations of inputs (i.e., \(abc = 000, 001, 010, 011, 100, 101, 110, 111\)), so the truth table has the eight rows shown in Figure 2.42. For each input, we compute the value of \(F\) and fill in the corresponding entry in the truth table. For example, when \(a=0, b=0,\) and \(c=0\), \(F\) is \((00)' \cdot 0' = (0)' \cdot 1 = 1\). We compute the circuit’s output for the remaining combinations of inputs using a truth table with intermediate values, shown in Figure 2.42.

Standard Representation and Canonical Form

Standard Representation—Truth Tables

As stated earlier, although there are many equation representations and circuit representations of a Boolean function, there is only one possible truth table representation of a Boolean function. Truth tables therefore represent a standard representation of a function—for any function, there may be many possible equations, and many possible circuits, but there is only one truth table. The truth table representation is unique.

One use of a standard representation of a Boolean function is for comparing two functions to see if they are equivalent. Suppose you wanted to check whether two Boolean equations represented the same function. One way would be to try to manipulate one equation to be the same as the other equation, like we did in the automatic sliding door example of Example 2.13. But suppose we were not successful in getting them to be the same—is that because they really are not the same, or because we just didn’t manipulate the equation enough? How do we really know the two equations do not represent the same function?

A conclusive way to check whether two items represent the same function is to create a truth table for each, and then check whether the truth tables are identical. So to determine whether \(F = ab + a'\) is equivalent to \(F = a'b' + a'b + ab\), we could generate truth tables for each, using the method described earlier of evaluating the function for each output row, as Figure 2.43.

![Figure 2.43 Truth tables showing equivalence.](image_url)
Example 2.23 Binary number to seven-segment display converter

Many electronic appliances display a number for us to read. Example appliances include a clock, a microwave oven, and a telephone answering machine. A popular and simple device for displaying a single digit number is a seven-segment display, illustrated in Figure 2.46.

![Seven-segment display](image)

**Figure 2.46** Seven-segment display: (a) connections of inputs to segments, (b) input values for numbers 0, 1, and 2, and (c) a pair of real seven-segment display components.

The display consists of seven light segments, each of which can be illuminated independently of the others. A desired digit can be displayed by setting the signals a, b, c, d, e, f, and g appropriately. So to display the digit 8, all seven signals must be set to 1. To display the digit 1, b and c are each set to 1. A few letters can be displayed too, like a lower case “b.”

Commonly, a microprocessor outputs a 4-bit binary number intended to be shown on a seven-segment display as a decimal (base ten) digit. Outputting four rather than seven signals conserves scarce pins on the microprocessor. Thus, a useful combinational circuit converts four bits w, x, y, and z of a binary number to the seven-segment display signals a-g, as in Figure 2.47.

![Binary to seven-segment converter](image)

**Figure 2.47** Binary to seven-segment converter.

The desired circuit behavior is easily captured as a table, shown in Table 2.4. In case the microprocessor outputs a number greater than 9, no segments are activated.

We can create a custom logic circuit to implement the converter. Note that Table 2.4 is in the form of a truth table having multiple outputs (a through g). We can treat each output separately, designing a circuit for a, then for b, etc. Summing the terms corresponding to the 1s in the a column (as was done in Figure 2.40) leads to the following equation for a:

\[
\begin{align*}
a &= w'x'y'z' + w'x'yz' + w'x'y'z + w'xy'z + w'xyz' \\
&
+ w'xyz + wx'y'z' + wx'y'z
\end{align*}
\]

Likewise, summing the terms for the 1s in the b column leads to the following equation for b:

\[
\begin{align*}
b &= w'x'y'z' + w'x'y'z + w'x'y'z' + w'x'yz + w'xy'z' \\
&
+ w'xyz + wx'y'z' + wx'y'z
\end{align*}
\]

Equations could similarly be created for the remaining outputs c through g. Finally, a circuit could be created for a having 8 4-input AND gates and an 8-input OR gate, another circuit for b having 8 4-input AND gates and an 8-input OR gate, and so on for c through g. We could, of course, have minimized the logic for each equation before creating each of the circuits.

You may notice that the equations for a and b have several terms in common. For example, the term \(w'x'y'z'\) appears in both equations. So it would make sense for both outputs to share one
image not available
image not available
Step 2A: Create equations. We create equations (as was done in Figure 2.40) for each output as follows:

\[ y = a'b'c + ab'c + abc + abc' + abc \]
\[ z = a'b'c + a'bc' + ab'c' + abc \]

We can simplify the first equation algebraically:

\[ y = a'b'c + ab'c + ab|c' + c) = a'b'c + ab'c + ab \]

Step 2B: Implement as a gate-based circuit. We then create the final circuits for the two outputs, as shown in Figure 2.49.

![Diagram of circuit](image)

Figure 2.49 Number-of-1's counter gate-based circuit.

Simplifying Circuit Notations

Some new simplifying notations were used in the circuits in the previous example. One simplifying notation is to list the inputs multiple times, as in Figure 2.50(a). Such listing reduces lines in a drawing crossing one another. An input listed multiple times is assumed to have been branched from the same input.

![Diagram of circuit](image)

Figure 2.50 Simplifying circuit notations: (a) listing inputs multiple times to reduce drawing of crossing wires, (b) using inversion bubbles or complemented input to reduce NOT gates drawn.

Another simplifying notation is the use of an inversion bubble at the input of a gate, rather than the use of an inverter, as in Figure 2.50(b). An inversion bubble is a small circle drawn at the input of a gate as shown, indicating that the signal is inverted. An external input that has inversion bubbles at many gates is assumed to feed through a single inverter that is then branched out to those gates. An alternative simplification is to simply list the input as complemented, like \( b' \) shown in the figure.
image
not available
image not available
2.8 MORE GATES

Designers use several other types of gates beyond just AND, OR, and NOT. Those gates include NAND, NOR, XOR, and XNOR.

**NAND & NOR**

A NAND gate (short for “not AND”) has the opposite output of an AND gate, outputting a 0 only when all inputs are 1, and outputting a 1 otherwise (meaning at least one input is 0). A NAND gate has the same behavior as an AND gate followed by a NOT gate. Figure 2.54(a) illustrates a NAND gate.

A NOR gate (“not OR”) has the opposite output as an OR gate, outputting a 0 if at least one input is 1, and outputting 1 if all inputs are 0. A NOR gate has the same behavior as an OR gate followed by a NOT gate. Figure 2.54(b) shows a NOR gate.

Whereas Boolean algebra has the symbols “¬” and “+” for the AND and OR operations, no such commonly-used operator symbols exist for NAND and NOR. Instead, the NAND operation on variables `a` and `b` would be written as `(a^b)` or just `(ab)^`, and the NOR operation would be written as `(a + b)^`.

![NAND NOR XOR XNOR Truth Tables and Circuits](image)

Figure 2.54 Additional gates: (a) NAND, (b) NOR, (c) XOR, (d) XNOR.

Section 2.4 warned that the shown CMOS transistor implementations of AND and OR gates were not realistic. The reason is because pMOS transistors don’t conduct 0s very well, but they conduct 1s just fine. Likewise, nMOS transistors don’t conduct 1s well, but they conduct 0s just fine. The reasons for these asymmetries are beyond this book’s scope. The implications are that the AND and OR gates in Figure 2.8 are not feasible, as they rely on pMOS transistors to conduct 0s (but pMOS conducts 0s poorly) and nMOS transistors to conduct 1s (but nMOS conducts 1s poorly). However, if we switch the locations of power and ground in the AND and OR circuits of Figure 2.8, the results are the NAND and NOR gate circuits shown in Figure 2.54(a) and Figure 2.54(b).
image not available
image not available
2 Combinational Logic Design

N-variable function will have \(2^N\) rows in its truth table. Then, note that each row can output one of two possible values. Thus, the number of possible functions will be \(2 \times 2 \times 2 \times \cdots \times 2^N\) times. Therefore, the total number of functions is:

\[
2^{2^N}
\]

So there are \(2^3 = 2^8 = 256\) possible Boolean functions of 3 variables, and \(2^4 = 2^{16} = 65536\) possible functions of 4 variables.

2.9 DECODERS AND MUXES

Two additional components, a decoder and a multiplexer, are also commonly used as digital circuit building blocks, though they themselves can be built from logic gates.

Decoders

A decoder is a higher-level building block commonly used in digital circuits. A decoder decodes an input n-bit binary number by setting exactly one of the decoder's \(2^n\) outputs to 1. For example, a 2-input decoder, illustrated in Figure 2.62(a), would have \(2^2 = 4\) outputs, \(d_3, d_2, d_1, d_0\). If the two inputs \(i_1i_0\) are 00, \(d_0\) would be 1 and the remaining outputs would be 0. If \(i_1i_0 = 01, d_1\) would be 1. If \(i_1i_0 = 10, d_2\) would be 1. If \(i_1i_0 = 11, d_3\) would be 1. One and only one output of a decoder will ever be 1 at a given time, corresponding to the particular current value of the inputs, as shown in Figure 2.62(a).

The internal design of a decoder is straightforward. Consider a 2x4 decoder. Each output \(d_0, d_1, d_2,\) and \(d_3\) is a distinct function. \(d_0\) should be 1 only when \(i_1 = 0\) and \(i_0 = 0\), so \(d_0 = i_1' i_0'\). Likewise, \(d_1 = i_1' i_0, d_2 = i_1 i_0',\) and \(d_3 = i_1 i_0.\) Thus, we build the decoder with one AND gate for each output, connecting the true or complemented values of \(i_1\) and \(i_0\) to each gate, as shown in Figure 2.62.

![2x4 decoder](image)

Figure 2.62 2x4 decoder: (a) outputs for possible input combinations, (b) internal design.
image not available
image not available
likely to be correct. However, sometimes designers start with an equation rather than a truth table, as in Example 2.24. A designer can reverse engineer the circuit to an equation, but that equation may be different than the original equation, especially if the designer algebraically manipulated the original equation when designing the circuit. Furthermore, checking that two equations are equivalent may require converting to canonical form (sum-of-minterms), which may result in huge equations if the function has a large number of inputs.

In fact, even if a designer didn't make any mistakes in converting a mental understanding of the desired function into a truth table or equation, how does the designer know that the original understanding was correct?

A commonly used method for checking that a circuit works as expected is called simulation. Simulation of a circuit is the process of providing sample inputs to the circuit and running a computer program that computes the circuit's output for the given inputs. A designer can then check that the output matches what is expected. The computer program that performs simulation is called a simulator.

To use simulation to check a circuit, a designer must describe the circuit using a method that enables computer programs to read the circuit. One method of describing a circuit is to draw the circuit using a schematic capture tool. A schematic capture tool allows a user to place logic gates on a computer screen and to draw wires connecting those gates. The tool allows users to save their circuit drawings as computer files. All the circuit drawings in this chapter have represented examples of schematics—for example, the circuit drawing in Figure 2.62(b), which showed a 2x4 decoder, was an example of a schematic. Figure 2.75 shows a schematic for the same design, drawn using a popular commercial schematic capture tool. Schematic capture is used not only to capture circuits for simulator tools, but also for tools that map our circuits to physical implementations, which will be discussed in Chapter 7.

Once a designer has created a circuit using schematic capture, the designer must provide the simulator with a set of inputs that will be used to check for proper output. One way of providing the inputs is by drawing waveforms for the circuit's inputs. An input's waveform is a line that goes from left to right, representing the value of the input as time proceeds to the right. The line is drawn high to represent 1 and low to represent 0.

![Figure 2.75 Display snapshot of a commercial schematic capture tool.](image)
image not available
image not available
2.24 For the function \( F = a'd' + a'c + b'cd' + cd \):
(a) List all the variables.
(b) List all the literals.
(c) List all the product terms.

2.25 Let variables \( T \) represent being tall, \( H \) being heavy, and \( F \) being fast. Let's consider anyone who is not tall as short, not heavy as light, and not fast as slow. Write a Boolean equation to represent each of the following:
(a) You may ride a particular amusement park ride only if you are either tall and light, or short and heavy.
(b) You may NOT ride an amusement park ride if you are either tall and light, or short and heavy. Use algebra to simplify the equation to sum of products.
(c) You are eligible to play on a particular basketball team if you are tall and fast, or tall and slow. Simplify this equation.
(d) You are NOT eligible to play on a particular football team if you are short and slow, or if you are light. Simplify to sum-of-products form.
(e) You are eligible to play on both the basketball and football teams above, based on the above criteria. Hint: combine the two equations into one equation by ANDing them.

2.26 Let variables \( S \) represent a package being small, \( H \) being heavy, and \( E \) being expensive. Let's consider a package that is not small as big, not heavy as light, and not expensive as inexpensive. Write a Boolean equation to represent each of the following:
(a) Your company specializes in delivering packages that are both small and inexpensive (a package must be small AND inexpensive for us to deliver it); you'll also deliver packages that are big but only if they are expensive.
(b) A particular truck can be loaded with packages only if the packages are small and light, small and heavy, or big and light. Simplify the equation.
(c) Your above-mentioned company buys the above-mentioned truck. Write an equation that describes the packages your company can deliver. Hint: Appropriately combine the equations from the above two parts.

2.27 Use algebraic manipulation to convert the following equation to sum-of-products form:
\[ F = a(b' + c)(d') + ac'(b + d) \]

2.28 Use algebraic manipulation to convert the following equation to sum-of-products form:
\[ F = a'b(c + d') + a(b' + c) + a(b + d)c \]

2.29 Use DeMorgan’s Law to find the inverse of the following equation: \( F = abc + a'b' \). Reduce to sum-of-products form. Hint: Start with \( F' = (abc + a'b')' \)

2.30 Use DeMorgan’s Law to find the inverse of the following equation: \( F = ac' + ab'd' + acd \). Reduce to sum-of-products form.

SECTION 2.6: REPRESENTATIONS OF BOOLEAN FUNCTIONS

2.31 Convert the following Boolean equations to a digital circuit:
(a) \( F(a, b, c) = a'bc + ab \)
(b) \( F(a, b, c) = a'b \)
(c) \( F(a, b, c) = abc + ab + a + b + c \)
(d) \( F(a, b, c) = c' \)
2.32 Create a Boolean equation representation of the digital circuit in Figure 2.78.
2.33 Create a Boolean equation representation for the digital circuit in Figure 2.79.
2.34 Convert each of the Boolean equations in Exercise 2.31 to a truth table.
2.35 Convert each of the following Boolean equations to a truth table:
   (a) \( F(a, b, c) = a' + bc' \)
   (b) \( F(a, b, c) = (ab)' + ac' + bc \)
   (c) \( F(a, b, c) = ab + ac + ab'c' + c' \)
   (d) \( F(a, b, c, d) = a'b + bc + d' \)
2.36 Fill in Table 2.8’s columns for the equation: \( F = ab + b' \)
2.37 Convert the function \( F \) shown in the truth table in Table 2.9 to an equation. Don’t minimize the equation.
2.38 Use algebraic manipulation to minimize the equation obtained in Exercise 2.37.
2.39 Convert the function \( F \) shown in the truth table in Table 2.10 to an equation. Don’t minimize the equation.
2.40 Use algebraic manipulation to minimize the equation obtained in Exercise 2.39.
2.41 Convert the function \( F \) shown in the truth table in Table 2.11 to an equation. Don’t minimize the equation.
2.42 Use algebraic manipulation to minimize the equation obtained in Exercise 2.41.
2.43 Create a truth table for the circuit of Figure 2.78.
2.44 Create a truth table for the circuit of Figure 2.79.
2.45 Convert the function \( F \) shown in the truth table in Table 2.9 to a digital circuit.
2.46 Convert the function \( F \) shown in the truth table in Table 2.10 to a digital circuit.
2.47 Convert the function \( F \) shown in the truth table in Table 2.11 to a digital circuit.
2.48 Convert the following Boolean equations to canonical sum-of-minterms form:
   (a) \( F(a, b, c) = a'b + bc \)
   (b) \( F(a, b, c) = a'b \)
   (c) \( F(a, b, c) = ab + ab + a + b + c \)
   (d) \( F(a, b, c) = c' \)
2.49 Determine whether the Boolean functions \( F = (a + b)' \cdot a \) and \( G = a + b' \) are equivalent, using (a) algebraic manipulation and (b) truth tables.

2.50 Determine whether the Boolean functions \( F = ab' \) and \( G = (a' + ab)’ \) are equivalent, using (a) algebraic manipulation and (b) truth tables.

2.51 Determine whether the Boolean function \( G = a'b'c + ab'c + abc' + abc \) is equivalent to the function represented by the circuit in Figure 2.80.

2.52 Determine whether the two circuits in Figure 2.81 are equivalent circuits, using (a) algebraic manipulation and (b) truth tables.

![Figure 2.80 Combinational circuit for H.](image)

2.53 * Figure 2.82 shows two circuits whose inputs are unlabeled.
(a) Determine whether the two circuits are equivalent. Hint: Try all possible labelings of the inputs for both circuits.
(b) How many circuit comparisons would need to be performed to determine whether two circuits with 10 unlabeled inputs are equivalent?

![Figure 2.81 Combinational circuits for F and G.](image)

![Figure 2.82 Combinational circuits for F and G.](image)

**SECTION 2.7: COMBINATIONAL LOGIC DESIGN PROCESS**

2.54 A museum has three rooms, each with a motion sensor (\( m_0, m_1, \) and \( m_2 \)) that outputs 1 when motion is detected. At night, the only person in the museum is one security guard who walks from room to room. Create a circuit that sounds an alarm (by setting an output \( A \) to 1) if motion is ever detected in more than one room at a time (i.e., in two or three rooms), meaning there must be one or more intruders in the museum. Start with a truth table.

2.55 Create a circuit for the museum of Exercise 2.54 that detects whether the guard is properly patrolling the museum, detected by exactly one motion sensor being 1. (If no motion sensor is 1, the guard may be sitting, sleeping, or absent.)
2.56 Consider the museum security alarm function of Exercise 2.54, but for a museum with 10 rooms. A truth table is not a good starting point (too many rows), nor is an equation describing when the alarm should sound (too many terms). However, the inverse of the alarm function can be straightforwardly captured as an equation. Design the circuit for the 10-room security system by designing the inverse of the function, and then just adding an inverter before the circuit’s output.

2.57 A network router connects multiple computers together and allows them to send messages to each other. If two or more computers send messages simultaneously, the messages “collide” and must be re-sent. Using the combinational design process of Table 2.5, create a collision detection circuit for a router that connects 4 computers. The circuit has 4 inputs labeled M0 through M3 that are 1 when the corresponding computer is sending a message and 0 otherwise. The circuit has one output labeled C that is 1 when a collision is detected and 0 otherwise.

2.58 Using the combinational design process of Table 2.5, create a 4-bit prime number detector. The circuit has four inputs—N3, N2, N1, and N0—that correspond to a 4-bit number (N3 is the most significant bit) and one output P that is 1 when the input is a prime number and that is 0 otherwise.

2.59 A car has a fuel-level detector that outputs the current fuel-level as a 3-bit binary number, with 000 meaning empty and 111 meaning full. Create a circuit that illuminates a “low fuel” indicator light (by setting an output L to 1) when the fuel level drops below level 3.

2.60 A car has a low-tire-pressure sensor that outputs the current tire pressure as a 5-bit binary number. Create a circuit that illuminates a “low tire pressure” indicator light (by setting an output T to 1) when the tire pressure drops below 16. Hint: you might find it easier to create a circuit that detects the inverse function. You can then just append an inverter to the output of that circuit.

SECTION 2.8: MORE GATES

2.61 Show the conduction paths and output value of the NAND gate transistor circuit in Figure 2.54 when: (a) $x = 1$ and $y = 0$, (b) $x = 1$ and $y = 1$.

2.62 Show the conduction paths and output value of the NOR gate transistor circuit in Figure 2.54 when: (a) $x = 1$ and $y = 0$, (b) $x = 0$ and $y = 0$.

2.63 Show the conduction paths and output value of the AND gate transistor circuit in Figure 2.55 when: (a) $x = 1$ and $y = 1$, (b) $x = 0$ and $y = 1$.

2.64 Two people, denoted using variables A and B, want to ride with you on your motorcycle. Write a Boolean equation that indicates that exactly one of the two people can come ($A=1$ means A can come; $A=0$ means A can’t come). Then use XOR to simplify your equation.

2.65 Simplify the following equation by using XOR wherever possible: $F = a'b + ab' + c'd + ac$.

2.66 Use 2-input XOR gates to create a circuit that outputs a 1 when the number of 1s on inputs $a$, $b$, $c$, $d$ is odd.

2.67 Use 2-input XOR or XNOR gates to create a circuit that detects whether an even number of the inputs $a$, $b$, $c$, $d$ are 1s.

SECTION 2.9: DECODERS AND MUXES

2.68 Design a 3x8 decoder using AND, OR, and NOT gates.

2.69 Design a 4x16 decoder using AND, OR, and NOT gates.

2.70 Design a 3x8 decoder with enable using AND, OR, and NOT gates.

2.71 Design an 8x1 multiplexer using AND, OR, and NOT gates.
2.72 Design a 16x1 multiplexer using AND, OR, and NOT gates.

2.73 Design a 4-bit 4x1 multiplexer using four 4x1 multiplexers.

2.74 A house has four external doors, each with a sensor that outputs 1 if its door is open. Inside the house is a single LED that a homeowner wishes to use to indicate whether a door is open or closed. Because the LED can only show the status of one sensor, the homeowner buys a switch that can be set to 0, 1, 2, or 3 and that has a 2-bit output representing the switch position in binary. Create a circuit to connect the four sensors, the switch, and the LED. Use at least one mux (a single mux or an N-bit mux) or decoder. Use block symbols, each with a clearly defined function, such as “2x1 mux,” “8-bit 2x1 mux,” or “3x8 decoder”; do not show the internal design of a mux or decoder.

2.75 A video system can accept video from one of two video sources, but can only display one source at a given time. Each source outputs a stream of digitized video on its own 8-bit output. A switch with a single-bit output chooses which of the two 8-bit streams will be passed on a display’s single 8-bit input. Create a circuit to connect the two video sources, the switch, and the display. Use at least one mux (a single mux or an N-bit mux) or decoder. Use block symbols, each with a clearly defined function, such as “2x1 mux,” “8-bit 2x1 mux,” or “3x8 decoder”; do not show the internal design of a mux or decoder.

2.76 A store owner wishes to be able to indicate to customers that the items in one of the store’s eight aisles are temporarily discounted (“on sale”). The store owner thus mounts a light above each aisle, and each light has a single-bit input that turns on the light when 1. The store owner has a switch that can be set to 0, 1, 2, 3, 4, 5, 6, or 7, and that has a 3-bit output representing the switch position in binary. A second switch can be set up or down and has a single-bit output that is 1 when the switch is up; the store owner can set this switch down if no aisles are currently discounted. Use at least one mux (a single mux or an N-bit mux) or decoder. Use block symbols, each with a clearly defined function, such as “2x1 mux,” “8-bit 2x1 mux,” or “3x8 decoder”; do not show the internal design of a mux or decoder.

SECTION 2.10: ADDITIONAL CONSIDERATIONS

2.77 Determine the critical path of the following specified circuits. Assume that each AND and OR gate has a delay of 1 ns, each NOT gate has a delay of 0.75 ns, and each wire has a delay of 0.5 ns.
   (a) The circuit of Figure 2.37.
   (b) The circuit of Figure 2.41.

2.78 Design a 1x4 demultiplexer using AND, OR, and NOT gates.

2.79 Design an 8x3 encoder using AND, OR, and NOT gates. Assume that only one input will be 1 at any given time.

2.80 Design a 4x2 priority encoder using AND, OR, and NOT gates. If every input is 0, the encoder output should be 00.
DESIGNER PROFILE

Samson enjoyed physics and math in college, and focused his advanced studies on integrated circuit (IC) design, believing the industry to have a great future. Years later, he realizes his belief was true: “Looking back 20 years in high tech, we have experienced four major revolutions: the PC revolution, digital revolution, communication revolution, and Internet revolution—all four enabled by the IC industry. The impact of these revolutions to our daily life is profound.”

He has found his job to be “very challenging, interesting, and exciting. I continually learn new skills to keep up, and to do my job more efficiently.”

One of Samson’s key design projects was for digital television, namely, high-definition TV (HDTV), involving companies like Zenith, Philips, and Intel. In particular, he led the 12-person design team that built Intel’s first liquid crystal on silicon (LCOS) chip for rear-projection HDTV. “Traditional LCOS chips are analog. They apply different analog voltages on each pixel of the display chip so it can produce an image. But analog LCOS is very sensitive to noise and temperature variation. We used digital signals to do pulse width modulation on each pixel.” Samson is quite proud of his team’s accomplishments: “Our HDTV picture quality was much better.”

Samson also worked on the 200-member design team for Intel’s Pentium II processor. That was a very different experience. “For the smaller team project, each person had more responsibility, and overall efficiency was high. For the large team project, each person worked on a specific part of the project—the chip was divided into clusters, each cluster into units, and each unit had a leader. We relied heavily on design flows and methodologies.”

Samson has seen the industry’s peaks and valleys during the past two decades: “Like any industry, the IC job market has its ups and downs.” He believes the industry survives the low points in large part due to innovation. “Brand names sell products, but without innovation, markets go elsewhere. So we have to be very innovative, creating new products so that we are always ahead in the global competition.”

But “innovation doesn’t grow on trees,” Samson points out. “There are two kinds of innovations. The first is invention, which requires a good understanding of the physics behind technology. For example, to make an analog TV into a digital TV, we must know how human eyes perceive video images, which parts can be digitized, how digital images can be produced on a silicon chip, etc. The second kind of innovation reuses existing technology for a new application. For example, we can reuse advanced space technologies in a new non-space product serving a bigger market. eBay is another example—it reused Internet technology for online auctions. Innovations lead to new products, and thus new jobs for many years.”

Thus, Samson points out that “The industry is counting on new engineers from college to be innovative, so they can continue to drive the high-tech industry forward. When you graduate from college, it’s up to you to make things better.”
3

Sequential Logic Design: Controllers

3.1 Introduction

The output of a combinational circuit is a function of the circuit’s present inputs. A combinational circuit has no memory—the feature of a circuit storing new bits and retaining those bits over time for later use. Combinational circuits alone are of limited usefulness. Designers typically use combinational circuits as part of larger circuits called sequential circuits—circuits that have memory. A sequential circuit is a circuit whose output depends not only on the circuit’s present inputs, but also on the circuit’s present state, which is all the bits currently stored in the circuit. The circuit’s state in turn depends on the past sequence of the circuit’s input values.

An everyday sequential system example is a lamp that toggles (changes from off to on, or from on to off) when its button is pressed, as was shown in Figure 2.1(c). After plugging in the lamp, push the lamp’s button (the input) a first time, and the lamp turns on. Push the button a second time, and the lamp turns off. Push the button a third time, and the lamp turns on again. The system’s output (whether the lamp is on or off) depends on the input and on whether the system is currently in the state of the lamp being on or off. That state in turn depends on the past sequence of input values since the system was initially powered on. In contrast, an everyday combinational system example is a basic doorbell, as was shown in Figure 2.1(a). Push the button (the input) now, and the bell (the output) rings. Push the button again, and the bell rings again. Push the button tomorrow and the bell rings the same each time. A basic doorbell has no state—its output value (whether the bell rings or not) depends solely on its present input value (whether the button is pressed or not).

Most digital systems with which you are likely familiar involve sequential circuits. A calculator contains a sequential circuit to store the numbers you enter, in order to operate on those numbers. A digital camera stores pictures. A traffic light controller stores information indicating which light is presently green. A kitchen timer that counts down from a set time to zero stores the present count value, to know what the next value should be.

This chapter describes sequential circuit building blocks called flip-flops and registers, which can store bits. It then introduces a sequential circuit design process in which a designer first captures desired behavior, and then converts that behavior to a type of sequential circuit known as a controller, comprised of a register and combinational logic.
3.2 Storing One Bit—Flip-Flops

Sequential circuit design is aided by a building block that enables storing of a bit, much like combinational circuit design was aided by the AND, OR, and NOT gate building blocks. Storing a bit means that we can save either a 0 or a 1 in the block and later come back to see what was saved. For example, consider designing the flight attendant call-button system in Figure 3.1. An airline passenger can push the Call button to turn on a small blue light above the passenger’s seat, indicating to a flight attendant that the passenger needs service. The light stays on even after the call button is released. The light can be turned off by pressing the Cancel button. Because the light must stay on even after the call button is released, a mechanism is needed to “remember” that the call button was pressed. That mechanism can be a bit storage block, in which a 1 will be stored when the call button is pressed, and a 0 stored when the cancel button is pressed. The inputs of this bit storage block will be connected to the call and cancel buttons, and the output to the blue light, as in Figure 3.1. The light illuminates when the block’s output is 1.

This section introduces the internal design of such a bit storage block by introducing several increasingly complex circuits able to store a bit—a basic SR latch, a level-sensitive SR latch, a level-sensitive D latch, and an edge-triggered D flip-flop. The D flip-flop will then be used to create a block capable of storing multiple bits, known as a register, which will serve as the main bit storage block in the rest of the book. Each successive circuit eliminates some problem of the previous one. Be aware that designers today rarely use bit storage blocks other than D flip-flops. We introduce the other blocks to provide the reader with an underlying intuition of the D flip-flop’s internal design.

Feedback—The Basic Storage Method

The basic method used to store a bit in a digital circuit is feedback. You’ve surely experienced feedback in the form of audio feedback, when someone talking into a microphone stood in front of the speaker, causing a loud continuous humming sound to come out of the speakers (in turn causing everyone to cover their ears and snicker). The talker generated a sound that was picked up by the microphone, came out of the speakers (amplified), was picked up again by the microphone, came out the speakers again (amplified even more), etc. That’s feedback.

Feedback in audio systems is annoying, but in digital systems is extremely useful. Intuitively, we know that somehow the output of a logic gate must feed back into the gate itself, so that the stored bit ends up looping around and around, like a dog chasing its own tail. We might try the circuit in Figure 3.2.

Suppose initially Q is 0 and S is 0. At some point, suppose we set S to 1. That causes Q to become 1, and that 1 feeds back into the OR gate, causing Q to be 1, etc. So even when S returns to 0, the output Q will remain at 1, which is exactly what we want.
to 0, Q stays 1. Unfortunately, Q stays 1 from then on, and we have no way of resetting Q to 0. But hopefully you understand the basic idea of feedback now—we did successfully store a 1 using feedback, but we couldn’t store a 0 again.

Figure 3.3 shows the timing diagram for the feedback circuit of Figure 3.2. Initially, we set both OR gate inputs to 0 (Figure 3.3(a)). Then we set S to 1 (Figure 3.3(b)), which causes Q to become 1 slightly later (Figure 3.3(c)), assuming the OR gate has a small delay as discussed in Section 2.10. Q becoming 1 causes t to become 1 slightly later (Figure 3.3(d)), assuming the wire has a small delay too. Q will stay at 1. Finally, when we change S back to 0 (Figure 3.3(e)), Q will continue to stay 1 because t is 1. The first curved line with an arrow indicates that the event of S changing from 0 to 1 causes the event of Q changing from 0 to 1. An event is any change on a bit signal from 0 to 1 or from 1 to 0. The second curved line with an arrow indicates that the event of Q changing from 0 to 1 in turn causes the event of t changing from 0 to 1. That 1 then continues to loop around, forever, with no way for S to reset Q to 0.

**Basic SR Latch**

It turns out that the simple circuit in Figure 3.4, called a basic SR latch, implements the bit storage building block that we seek. The circuit consists of a pair of cross-coupled NOR gates. Making the circuit’s S input equal to 1 causes Q to become 1, while making R equal to 1 causes Q to become 0. Making both S and R equal to 0 causes Q’s current value to keep looping around. In other words, S “sets” the latch to 1, and R “resets” the latch to 0—hence the letters S (for set) and R (for reset).

Let’s see why the basic SR latch works as it does. Recall that a NOR gate outputs 1 only when all the gate’s inputs equal 0, as shown in Figure 3.5; if at least one input equals 1, the NOR gate outputs 0.
Suppose we make $S=0$ and $R=1$, as in the SR latch circuit of Figure 3.6, and that the values of $Q$ and $t$ are initially unknown. Because the bottom gate of the circuit has at least one input equal to 1 ($R$), the gate outputs 0—in the timing diagram, $R$ becomes 1 causes $Q$ to become 0. In the circuit, $Q$’s 0 feeds back to the top NOR gate, which will have both its inputs equal to 0, and thus its output will be 1. In the timing diagram, $Q$ becoming 0 causes $t$ to become 1. In the circuit, $t$ feeds back to the bottom NOR gate, which has at least one input (actually, both) equal to 1, so the bottom gate continues to output 0. Thus the output $Q$ equals 0, and all values are stable, meaning the values won’t change as long as no external input changes.

Now suppose we keep $S=0$ and change $R$ from 1 back to 0, as in Figure 3.7. The bottom gate still has at least one input equal to 1 (the input coming from the top gate), so the bottom gate continues to output 0. The top gate continues to have both inputs equal to 0 and continues to output 1. The output $Q$ will thus still be 0. Therefore, the earlier $R=1$ stored a 0 into the SR latch, also known as resetting the latch, and that 0 remains stored even when $R$ is changed back to 0. Note that $R=1$ will reset the latch regardless of the initial value of $Q$.

Consider making $S=1$ and $R=0$, as in Figure 3.8. The top gate in the circuit now has one input equal to 1, so the top gate outputs a 0—the timing diagram shows the change of $S$ from 0 to 1, causing $t$ to change from 1 to 0. The top gate’s 0 output feeds back to the bottom gate, which now has both inputs equal to 0 and thus outputs 1—the timing diagram shows the change of $t$ from 1 to 0, causing $Q$ to change from 0 to 1. The bottom gate’s 1 output ($Q$) feeds back to the top gate, which has at least one input (actually, both of its inputs) equal to 1, so the top gate continues to output 0. The output $Q$ therefore equals 1, and all values are stable.
Next, consider making $S = 0$ and $R = 0$ again, as in Figure 3.9. The top gate still has at least one input equal to 1 (the input coming from the bottom gate), so the top gate continues to output 0. The bottom gate continues to have both inputs equal to 0 and continues to output 1. The output $Q$ is still 1. Thus, the earlier $S = 1$ stored a 1 into the SR latch, also known as setting the latch, and that 1 remains stored even when we return $S$ to 0. Note that $S = 1$ will set the latch regardless of the initial value of $Q$.

The basic SR latch can be used to implement the flight attendant call-button system as shown in Figure 3.10, by connecting the call button to $S$, the cancel button to $R$, and $Q$ to the light. Pressing the call button sets $Q$ to 1, thus turning on the light. $Q$ stays 1 even when the call button is released. Pressing the cancel button resets $Q$ to 0, thus turning off the light. $Q$ stays 0 even when the cancel button is released.

**Problem when SR=11 in a Basic SR latch**

A problem with the basic SR latch is that if $S$ and $R$ are both 1, undefined behavior results—the latch might store a 1, it might store a 0, or its output might oscillate, changing from 1 to 0 to 1 to 0, and so on. In particular, if $S = 1$ and $R = 1$ (written as “SR=11” for short), both the NOR gates have at least one input equal to 1, and thus both gates output 0, as in Figure 3.11(a). A problem occurs when $S$ and $R$ are made 0 again. Suppose $S$ and $R$ return to 0 at the same time. Then both gates will have 0s at all their inputs, so each gate’s output will change from 0 to 1, as in Figure 3.11(b). Those 1s feed back to the gates’ inputs, causing the gates to output 0s, as in Figure 3.11(c). Those 0s feed back to the gate inputs again, causing the gates to output 1s. And so on. Going from 1 to 0 to 1 to 0 repeatedly is called oscillation. Oscillation is not a desirable feature of a bit storage block.

**Figure 3.10** Flight attendant call-button system using a basic SR latch.

**Figure 3.11** The situation of $S=1$ and $R=1$ causes problems—$Q$ oscillates when $SR$ return to 00.
In a real circuit, the delays of the upper and lower gates and wires would be slightly different from one another. So after some time of oscillation, one of the gates will get ahead of the other, outputting a 1 before the other does, then a 0 before the other does, until it gets far enough ahead to cause the circuit to enter a stable situation of either \( Q = 0 \) or \( Q = 1 \). Which situation will happen is unknown beforehand. A situation in which the final output of a sequential circuit depends on the delays of gates and wires is a race condition. Figure 3.12 shows a race condition involving oscillation but ending with a stable situation of \( Q = 1 \).

Therefore, \( S \) and \( R \) must never be allowed to simultaneously equal 1 in an SR latch. A designer using an SR latch should add a circuit external to the SR latch that strives to ensure that \( S \) and \( R \) never simultaneously equal 1. For example, in the flight attendant call-button system of Figure 3.10, a passenger pushing both buttons at the same time might result in oscillation in the SR latch and hence a blinking light. The SR latch will eventually settle to 1 or 0, and thus the light will end up either on or off. A designer might therefore decide that if both buttons are pressed then the call button should take priority so that \( SR \) won’t both be 1. Such behavior can be achieved using a combinational circuit in front of \( S \) and \( R \), as shown in Figure 3.13. \( S \) should be 1 if the call button (denoted as \( Call \)) is pressed and either the cancel button (\( Cncl \)) is pressed or not pressed, so \( S = Call \cdot Cncl + Call \cdot Cncl' = Call \cdot Cncl \cdot Cncl' = Call \). \( R \) should be 1 only if the cancel button is pressed and the call button is not pressed, meaning \( R = Cncl \cdot Call' \). The circuit in Figure 3.13 is derived directly from these equations.

Even with such an external circuit, \( S \) and \( R \) could still inadvertently both become 1 due to the delay of real gates (see Section 2.10). Assume the AND and NOT gates in Figure 3.13 have delays of 1 ns each (ignore wire delays for now). Suppose the cancel button is being pressed and hence \( SR = 01 \), as in Figure 3.14, and then the call button is also pressed. \( S \) will change from 0 to 1 almost immediately, but \( R \) will remain at 1 for 2 ns longer, due to the AND and NOT gate delays, before changing to 0. \( SR \) would therefore be 11 for 2 ns. A temporary unintended signal value caused by circuit delays is called a glitch.
Significantly, glitches can also cause an unintended latch set or reset. Assume that the wire connecting the cancel button to the AND gate in Figure 3.13 has a delay of 4 ns (perhaps the wire is very long), in addition to the 1 ns AND and NOT gate delays. Suppose both buttons are pressed, so SR=10, and then the buttons are both released—SR should become 00. S will indeed change to 0 almost immediately. The top input of the AND gate will become 1 after the 1 ns delay of the NOT gate. The bottom input of that AND gate will remain 1 for 3 ns more, due to the 4 ns wire delay, thus causing R to change 1. After that bottom input finally changes to 0, yet another 1 ns will pass due to the AND gate delay before R returns to 0. Thus, R experienced a 4 ns glitch, which resets the latch to 0—yet a reset is clearly not what the designer intended.

**Level-Sensitive SR Latch**

A partial solution to the glitch problem is to extend the SR latch to have an enable input C as in Figure 3.16. When C=1, the S and R signals pass through the two AND gates to the S1 and R1 inputs of the basic SR latch, because S*1=S and R*1=R. The latch is enabled. But when C=0, the two AND gates cause S1 and R1 to be 0, regardless of the values of S and R. The latch is disabled. The enable input can be set to 0 when S and R might change so that glitches won’t propagate through to S1 and R1, and then set to 1 only when S and R are stable. The question then remains of when to set the enable input to 1. That question will be answered in the upcoming sections.

Figure 3.17 shows the call button system from Figure 3.13, this time using an SR latch with an enable input C. The timing diagram shows that if Cncl is 1 and then Call is changed to 1, a glitch of SR=11 occurs, as was already shown in Figure 3.14. However, because C=0, S1R1 stay at 00. When we later set the enable input to 1, the stable SR values propagate through to S1R1. An SR latch with an enable is called a **level-sensitive SR latch**, because the latch is only sensitive to its S and R inputs when the level of the enable input is 1. It is also called a **transparent SR latch**, because setting the enable input to 1 makes the internal SR latch transparent to the S and R inputs. It is also sometimes called a **gated SR latch**.
Notice that the top NOR gate of an SR latch outputs the opposite value as the bottom NOR gate that outputs $Q$. Thus, an output $Q'$ can be included on an SR latch almost for free, just by connecting the top gate to an output named $Q'$. Most latches come with both $Q$ and $Q'$ outputs. The symbol for a level-sensitive SR latch with such dual outputs is shown in Figure 3.18.

**Level-Sensitive D Latch—A Basic Bit Store**

A designer using a level-sensitive SR latch has the burden of ensuring that $S$ and $R$ are never simultaneously 1 when the enable input is 1. One way to relieve designers of this burden is to introduce another type of latch, called a **level-sensitive D latch** (also known as a **transparent D latch** or **gated D latch**), shown in Figure 3.19. Internally, the latch's $D$ input connects directly to the $S$ input of a level-sensitive SR latch, and connects through an inverter to the $R$ input of the SR latch. The D latch is thus either setting (when $D=1$) or resetting (when $D=0$) its internal basic SR latch when the enable input $C$ is 1.
A level-sensitive D latch thus stores whatever value is present at the latch's D input when \( C = 1 \), and remembers that value when \( C = 0 \). Figure 3.20 shows a timing diagram of a D latch for sample input values on D and C; arrows indicate which signal changes cause other signals to change. When D is 1 and C is 1, the latch is set to 1, because \( S_1 \) is 1 and \( R_1 \) is 0. When D is 0 and C is 1, the latch is reset to 0, because \( R_1 \) is 1 and \( S_1 \) is 0. By making \( R \) the opposite of \( S \), the D latch ensures that \( S \) and \( R \) won't both be 1 at the same time, as long as \( D \) is only changed when \( C = 0 \) (even if changed when \( C = 1 \), the inverter's delay could cause \( S \) and \( R \) to both be 1 briefly, but for too short of time to cause a problem).

The symbol for a D latch with dual-outputs (Q and \( Q' \)) is shown in Figure 3.21.

**Edge-Triggered D Flip-Flop—A Robust Bit Store**

The D latch still has a problem that can cause unpredictable circuit behavior—namely, signals can propagate from a latch output to another latch's input while the clock signal is 1. For example, consider the circuit in Figure 3.22 and the pulsing enable signals—a **pulse** is a change from 0 to 1 and back to 0, and a pulsing enable signal is called a **clock** signal. When \( \text{Clk} = 1 \), the value on Y will be loaded into the first latch and appear at that latch's output. If \( \text{Clk} \) still equals 1, then that value will also get loaded into the second latch. The value will keep propagating through the latches until \( \text{Clk} \) returns to 0. Through how many latches will the value propagate for a pulse on \( \text{Clk} \)? It's hard to say—we would have to know the precise timing delay information of each latch.

![Figure 3.22 A problem with latches—through how many latches will Y propagate for each pulse of Clk_A? For Clk_B?](image)

Figure 3.23 illustrates this propagation problem in more detail. Suppose D1 is initially 0 for a long time, changes to 1 long enough to be stable, and then \( \text{Clk} \) becomes 1. Q1 will thus change from 0 to 1 after about three gate delays, and thus D2 will also change from 0 to 1, as shown in the left timing diagram. If \( \text{Clk} \) is still 1, then that new value for D2 will propagate through the AND gates of the second latch, causing \( S_2 \) to change from 0 to 1 and \( R_2 \) from 1 to 0, thus changing Q2 from 0 to 1, as shown in the left timing diagram.
You might suggest making the clock signal such that the clock is 1 only for a short amount of time, so there’s not enough time for the new output of a latch to propagate to the next latch’s inputs. But how short is short enough? 50 ns? 10 ns? 1 ns? 0.1 ns? And if we make the clock’s time at 1 too short, that time may not be long enough for the bit at a latch’s D input to stabilize in the latch’s feedback circuit, and we might therefore not successfully store the bit, as illustrated in Figure 3.23(c).

A good solution is to design a more robust block for storing a bit—a block that stores the bit at the D input at the instant that the clock rises from 0 to 1. Note that we didn’t say that the block stores the bit instantly. Rather, the bit that will eventually get stored into the block is the bit that was stable at D at the instant that the clock rises from 0 to 1. Such a block is called an edge-triggered D flip-flop. The word “edge” refers to the vertical part of the line representing the clock signal, when the signal transitions from 0 to 1. Figure 3.24 shows three cycles of a clock signal, and indicates the three rising clock edges of those cycles.

**Figure 3.24** Rising clock edges.

**Edge-Triggered D Flip-Flop Using a Master-Servant Design.** One way to design an edge-triggered D flip-flop is to use two D latches, as shown in Figure 3.25.

The first D latch, known as the **master**, is enabled (can store new values on Dm) when C1k is 0 (due to the inverter), while the second D latch, known as the **servant**, is enabled...
Figure 3.25 A D flip-flop implementing an edge-triggered bit storage block, internally using two D latches in a master-servant arrangement. The master D latch stores its Dm input while Clk = 0, but the new value appearing at Qm, and hence at Ds, does not get stored into the servant latch, because the servant latch is disabled when Clk = 0. When Clk becomes 1, the servant D latch becomes enabled and thus gets loaded with whatever value was in the master latch at the instant that Clk changed from 0 to 1.

When Clk is 1, thus, while Clk is 0, the bit on D is stored into the master latch, and hence Qm and Ds are updated—but the servant latch does not store this new bit, because the servant latch is not enabled since Clk is not 1. When Clk becomes 1, the master latch becomes disabled, thus holding whatever bit was at the D input just before the clock changed from 0 to 1. Also, when Clk is 1, the servant latch becomes enabled, thus storing the bit that the master is storing, and that bit is the bit that was at the D input just before Clk changed from 0 to 1. The two latches thus implement an edge-triggered storage block—the bit that was at the input when Clk changed from 0 to 1 gets stored.

The edge-triggered block using two internal latches thus prevents the stored bit from propagating through more than one flip-flop when the clock is 1. Consider the chain of flip-flops in Figure 3.26, which is similar to the chain in Figure 3.22 but with D flip-flops in place of D latches.

We know that Y will propagate through exactly one flip-flop on each clock cycle.

The drawback of a master-servant approach is that two D latches are needed to store one bit. Figure 3.26 shows four flip-flops, but there are two latches inside each flip-flop, for a total of eight latches.

Many alternative methods exist other than the master-servant method for designing an edge-triggered flip-flop. In fact, there are hundreds of different designs for latches and flip-flops beyond the designs shown above, with those designs differing in terms of their size, speed, power, etc. When using an edge-triggered flip-flop, a designer usually doesn’t consider whether the flip-flop achieves edge-triggering using the master-servant method or using some other method. The designer need only know that the flip-flop is edge-trig-
Triggered, meaning the data value present when the clock edge is rising is the value that gets loaded into the flip-flop and that will appear at the flip-flop’s output some time later.

The above discussion is for what is known as **positive or rising** edge-triggered flip-flops, which are triggered by the clock signal changing from 0 to 1. There are also flip-flops known as **negative or falling** edge-triggered flip-flops, which are triggered by the clock changing from 1 to 0. A negative edge-triggered D flip-flop can be built using a master-servant design where the second flip-flop’s clock input is inverted, rather than the first flip-flop’s.

Positive edge-triggered flip-flops are drawn using a small triangle at the clock input, and negative edge-triggered flip-flops are drawn using a small triangle along with an inversion bubble, as shown in Figure 3.27. Because those symbols identify the clock input, those inputs typically are not given a name.

Bear in mind that although the master-servant design doesn’t change the output until the falling clock edge, the flip-flop is still positive edge-triggered, because the flip-flop stored the value that was at the D input at the instant that the clock edge was rising.

**Latches versus Flip-Flops:** Various textbooks define the terms latch and flip-flop differently. We’ll use what seems to be the most common convention among designers, namely:

- A **latch** is level-sensitive, and
- A **flip-flop** is edge-triggered.

So saying “edge-triggered flip-flop” would be redundant, since flip-flops are, by this definition, edge-triggered. Likewise, saying “level-sensitive latch” is redundant, since latches are by definition level-sensitive.

Figure 3.28 uses a timing diagram to illustrate the difference between level-sensitive (latch) and edge-triggered (flip-flop) bit storage blocks. The figure provides an example of a clock signal and a value on a signal D. The next signal trace is for the Q output of a D latch, which is level-sensitive. The latch ignores the first pulse on D (labeled as 3 in the figure) because Clk is low. However, when Clk becomes high (1), the latch output follows the D input, so when D changes from 0 to 1 (4), so does the latch output (7). The latch ignores the next changes on D when Clk is low (5), but then follows D again when Clk is high (6, 8).

---

*Designers commonly refer to flip-flops as just “flops.”*
The circuit’s desired behavior can be captured as the truth table in Table 3.1. If Call=0 and Cnc1=0 (the first two rows), D equals Q’s value. If Call=0 and Cnc1=1 (the next two rows), D=0. If Call=1 and Cnc1=0 (the next two rows), D=1. And if both Call=1 and Cnc1=1 (the last two rows), the Call button gets priority, so D=1.

After some algebraic simplification, we obtain the following equation for D:

\[ D = \text{Cnc1}'Q + \text{Call} \]

We can then convert the equation to the circuit shown in Figure 3.34(b). That circuit is more robust than the earlier circuit using an SR latch in Figure 3.10. But it is still not as good as it could be: Section 3.5 will explain why we might want to add additional flip-flops at the Call and Cnc1 inputs. Furthermore, our design process in this example was ad hoc; the following two sections will introduce better methods for capturing desired behavior and converting to a circuit.

The above sections went through several intermediate bit storage block designs before arriving at the robust D flip-flop design. Figure 3.35 summarizes those designs, including features and problems of each. Notice that the D flip-flop relies on an internal SR latch to maintain a stored bit between clock edges, and relies on the designer to introduce feedback outside the D flip-flop to maintain a stored bit across clock edges.

Figure 3.35 Increasingly better bit storage blocks, leading to the D flip-flop.
A BIT OF HISTORY—RS, JK, T, AND D LATCHES AND FLIP-FLOPS.

Many textbooks, especially those with origins in the 1970s and 1980s, introduce several types of latches and flip-flops and use many pages to describe how to design sequential circuits using those different types. In the 1980s, transistors on ICs were more costly and scarcer than today. The D flip-flop-based design for the call-button system in Figure 3.34(b) uses more transistors than the SR-latch-based design in Figure 3.10—not only does a D flip-flop contain more transistors internally, but it may require more external logic to set D to the appropriate value. Other flip-flop types included a JK flip-flop that acts like an SR flip-flop plus the behavior that the flip-flop toggles if both inputs are 1 (toggle means to change from 0 to 1, or from 1 to 0), and a T flip-flop with a single input T that toggles the flip-flop when 1. For a given desired behavior, using a particular flip-flop type could save transistors. Designing sequential circuits for any flip-flop type was a challenging task, involving something called “excitation tables” and comparison of different designs, and was helpful for reducing circuit transistors. But today, in the era of billion-transistor ICs, the savings of such flip-flops are trivial. Nearly all modern sequential circuits use D flip-flops and hence are created using the more straightforward design process introduced in this chapter.

Basic Register—Storing Multiple Bits

A register is a sequential component that can store multiple bits. A basic register can be built simply by using multiple D flip-flops as shown in Figure 3.36. That register can hold four bits. When the clock rises, all four flip-flops get loaded with inputs I_0, I_1, I_2, and I_3 simultaneously.

![Figure 3.36 A basic 4-bit register: (a) internal design, (b) block symbol.](image)

Such a register, made simply from multiple flip-flops, is the most basic form of a register—so basic that some companies refer to such a register simply as a “4-bit D flip-flop.” Chapter 4 introduces more advanced registers having additional features and operations.

Example 3.2 Temperature history display using registers

We want to design a system that records the outside temperature every hour and displays the last three recorded temperatures, so that an observer can see the temperature trend. An architecture of the system is shown in Figure 3.37.

A timer generates a pulse on signal C every hour. A temperature sensor outputs the present temperature as a 5-bit binary number ranging from 0 to 31, corresponding to those temperatures in Celsius. Three displays convert their 5-bit binary inputs into a numerical display.
3.2 Storing One Bit—Flip-Flops

![Diagram of a temperature history display system](image)

**Figure 3.37** Temperature history display system. (In practice, we would actually avoid connecting the timer output \( C \) to a clock input, instead only connecting an oscillator output to a clock input.)

We can implement the *TemperatureHistoryStorage* component using three 5-bit registers, as shown in Figure 3.38. Each pulse on signal \( C \) loads \( Ra \) with the present temperature on inputs \( x_4 \ldots x_0 \) (by loading the 5 flip-flops inside \( Ra \) with the 5 input bits). At the same time that register \( Ra \) gets loaded with that present temperature, register \( Rb \) gets loaded with the value that was in \( Ra \). Likewise, \( Rc \) gets loaded with \( Rb \)'s value. All three loads happen at the same time, namely, on the rising edge of \( C \). The effect is that the values that were in \( Ra \) and \( Rb \) just before the clock edge are shifted into \( Rb \) and \( Rc \), respectively.

![Diagram of the internal design of the Temperature History Storage component](image)

**Figure 3.38** Internal design of the *Temperature History Storage* component.

Figure 3.39 shows sample values in the registers for several clock cycles, assuming all the registers initially held 0s, and assuming that as time proceeds the inputs \( x_4 \ldots x_0 \) have the values shown at the top of the timing diagram.

![Example of values in the TemperatureHistoryStorage registers](image)

**Figure 3.39** Example of values in the *TemperatureHistoryStorage* registers. One particular data item, 18, is shown moving through the registers on each clock cycle.
This example demonstrates one of the desirable aspects of synchronous circuits built from edge-triggered flip-flops—many things happen at once, yet we need not be concerned about signals propagating too fast through a register to another register. The reason we need not be concerned is because registers only get loaded on the rising clock edge, which effectively is an infinitely small period of time, so by the time signals propagate through a register to a second register, it’s too late—that second register is no longer paying attention to its data inputs.

In practice, designers avoid connecting anything but an oscillator’s output to the clock input of a register. A key reason is so that automated tools that analyze a circuit’s timing characteristics can work properly; such tools are beyond the scope of this book. We connected a timer’s output, which pulsed once per hour, in the above example for the purpose of an intuitive introduction to registers. A better implementation would instead have an oscillator connected to the clock input, and then use the “load” input of a register when the timer output pulsed. The load input of a register will be introduced in Chapter 4.

### 3.3 FINITE-STATE MACHINES (FSMS)

Registers store bits in a digital circuit. Stored bits mean the circuit has memory resulting in sequential circuits. A circuit’s state is the value of all a circuit’s stored bits. While a register storing bits happens to result in a circuit with state, state can be intentionally used to design circuits that have a specific behavior over time. For example, we can specifically design a circuit that outputs a 1 for exactly three cycles whenever a button is pressed. We could design a circuit that blinks lights in a specific pattern. We could design a circuit that detects if three buttons get pushed in a particular sequence and then unlocks a door. All these cases make use of state to create specific time-ordered behavior for a circuit.

**Example 3.3 Three-cycles-high laser timer—a poorly done first design**

Consider the design of a part of a laser surgery system, such as a system for scar removal or corrective vision. Such systems work by turning on a laser for a precise amount of time (see “How does it work?—Laser surgery” on page 123). A general architecture of such a system is shown in Figure 3.40.

A surgeon activates the laser by pressing the button. Assume that the laser should then stay on for exactly 30 ns. Assume that the system’s clock period is 10 ns, so that 3 clock cycles last 30 ns. Assume that b from the button is synchronized with the clock and stays high for exactly 1 clock cycle. We need to design a controller component that, once detecting that b = 1, holds x high for exactly 3 clock cycles, thus turning on the laser for 30 ns.

This is one example for which a microprocessor solution may not work. Using a microprocessor’s programming statements that read input ports and write output ports may not provide a way to hold an output port high for exactly 30 ns—for example, when the microprocessor clock frequency is not fast enough.
Let’s try to create a sequential circuit implementation for the system. After thinking about the problem for a while, we might come up with the (bad) implementation in Figure 3.41.

Knowing the output should be held high for three clock cycles, we used three flip-flops, with the idea being that we’ll shift a 1 through those three flip-flops, taking three clock cycles for the bit to move through all three flip-flops. We ORed the flip-flop outputs to generate signal $x$, so that if any flip-flop contains a 1, the laser will be on. We made $b$ the input to the first flip-flop, so when $b=1$, the first flip-flop stores a 1 on the next rising clock edge. One clock cycle later, the second flip-flop will get loaded with 1, and assuming $b$ has now returned to 0, the first flip-flop will get loaded with 0. One clock cycle later, the third flip-flop will get loaded with 1, and the second flip-flop with 0. One clock cycle later, the third flip-flop will get loaded with 0. Thus, the circuit held the output $x$ at 1 for three clock cycles after the button was pressed.

We did a poor job implementing this system. First, what happens if the surgeon presses the button a second time before the three cycles are completed? Such a situation could cause the laser to stay on too long. Is there a simple way to fix our circuit to account for this behavior? Second, we didn’t use any orderly process for designing the circuit—we came up with the ORing of flip-flop outputs, but how did we come up with that? Will that method work for all time-ordered behavior that needs to be designed?

Two things are required to do a better job at designing circuits having time-ordered behavior: (1) a way to explicitly capture the desired time-ordered behavior, and (2) a technique for converting such behavior to a sequential circuit.

---

**HOW DOES IT WORK?—LASER SURGERY.**

Laser surgery has become very popular in the past two decades, and has been enabled due to digital systems. Lasers, invented in the early 1960s, generate an intense narrow beam of coherent light, with photons having a single wavelength and being in phase (like being in rhythm) with one another. In contrast, a regular light’s photons fly out in all directions, with a diversity of wavelengths. Think of a laser as a platoon of soldiers marching in synch, while a regular light is more like kids running out of school at the end-of-the-day bell. A laser’s light can be so intense as to even cut steel. The ability of a digital circuit to carefully control the location, intensity, and duration of the laser is what makes lasers so useful for surgery.

One popular use of lasers for surgery is for scar removal. The laser is focused on the damaged cells slightly below the surface, causing those cells to be vaporized. The laser can also be used to vaporize skin cells that form bumps on the skin, due to scars or moles. Similarly, lasers can reduce wrinkles by smoothing the skin around the wrinkle to make the crevices more gradual and hence less obvious, or by stimulating tissue under the skin to stimulate new collagen growth.

Another popular use of lasers for surgery is for correcting vision. In one popular laser eye surgery method, the surgeon uses a laser to cut open a flap on the surface of the cornea, and then uses a laser to reshape the cornea by thinning the cornea in a particular pattern, with such thinning accomplished through vaporizing cells.

A digital system controls the laser’s location, energy, and duration, based on programmed information of the desired procedure. The availability of lasers, combined with low-cost high-speed digital circuits, makes such precise and useful surgery now possible.
The above examples illustrate that a finite-state machine or FSM is a mathematical formalism consisting of several items:

- A set of states. The above example had four states: \( \text{On1, On2, On3, Off} \).
- A set of inputs and a set of outputs. The example had one input: \( \{b\} \), and one output: \( \{x\} \).
- An initial state: the state in which to start when the system is first powered on. An FSM’s initial state can be shown graphically by a directed edge (an edge with an arrow at one end) starting from no state and pointing to the initial state. An FSM can only have one initial state. The example’s initial state was the state named \( \text{Off} \). Note that \( \text{Off} \) is just a name, and does not suggest that the system’s power is off (rather, it suggests that the laser is off).
- A set of transitions: An indication of the next state based on the current state and the current values of the inputs. The example used directed edges with associated input conditions, which is a Boolean expression of input variables, to indicate the next state. Those edges with conditions are called transitions. The example had several transitions, such as the edge with condition \( b \cdot \text{clk}\).
- A description of what output values to assign in each state. The example assigns a value to \( x \) in every state. Assigning an output in an FSM is known as an action.

After being defined, an FSM can then be executed (even if just mentally)—what computer programmers might call “running” the FSM. The FSM starts with the current state being the initial state and then transitions to a different state based on the current state and input values, continuing as time proceeds. In each state, the FSM sets output values. Mentally executing an FSM is akin to mentally evaluating a Boolean equation for sample input values.

The FSM in Figure 3.45 would be interpreted as follows. The system starts in the initial state \( \text{Off} \). The system stays in state \( \text{Off} \) until one of the state’s two outgoing transitions has a true condition. One of those transitions has the condition of \( b \cdot \text{clk}\)—in that case, the system transitions right back to state \( \text{Off} \). The other transition has the condition of \( b \cdot \text{clk}\)—in that case, the system transitions to state \( \text{On1} \). The system stays in state \( \text{On1} \) until its only outgoing transition’s condition \( \text{clk} \) becomes true—in which case the system transitions to state \( \text{On2} \). Likewise, the system stays in \( \text{On2} \) until the next rising clock edge, transitioning to \( \text{On3} \). The system stays in \( \text{On3} \) until the next rising clock edge, transitioning back to state \( \text{Off} \). State \( \text{Off} \) has associated the action of setting \( x = 0 \), while the states \( \text{On1, On2, and On3} \) each set \( x = 1 \).

**“STATE” I UNDERSTAND, BUT WHY THE TERMS “FINITE” AND “MACHINE?”**

Finite-state machines, or FSMs, have a rather awkward name that sometimes causes confusion. The term “finite” is there to contrast FSMs with a similar representation used in mathematics that can have an infinite number of states; that representation is not very useful in digital design. FSMs, in contrast, have a limited, or finite, number of states. The term “machine” is used in its mathematical or computer science sense, being a conceptual object that can execute an abstract language—specifically, that sense of machine is not hardware. Finite-state machines are also known as finite-state automata. FSMs are used for many things other than just digital design.
The FSM in Figure 3.45 precisely describes the desired time-ordered behavior of the laser timer system from Example 3.3.

It is interesting to examine the behavior of this FSM if the button is pressed a second time while the laser is on. Notice that the transitions among the On states are independent of the value of \( b \). So this system will always turn the laser on for exactly three cycles and then return to the Off state to await another press of the button.

**Simplifying FSM Notation: Making the Rising Clock Implicit**

Thus far the rising clock edge (\( \text{clk}^* \)) has appeared in the condition of every FSM transition, because this book only considers the design of sequential circuits that are synchronous and that use rising edge-triggered flip-flops to store bits. Synchronous circuits with edge-triggered flip-flops make up the majority of sequential circuits in modern practice. As such, to make state diagrams more readable, most textbooks and designers follow the convention shown in Figure 3.46 wherein every FSM transition is *implicitly* ANDed with a rising clock edge. For example, a transition labeled “\( a^* \)" actually means “\( a^* \text{clk}^* \).” Subsequent state diagrams will not include the rising clock edge in transition conditions, instead following the convention that *every* transition is implicitly ANDed with a rising clock edge. Figure 3.47 illustrates the laser timer state diagram from Figure 3.45, redrawn using implicit rising clock edges.

A transition with no associated condition as in Figure 3.48 simply transitions on the next rising clock edge, because of the implicit rising clock edge.

Following are more examples showing how FSMs can describe time-ordered behavior.

![Figure 3.46 Simplifying notation: implicit rising clock edge on every transition.](image)

**Example 3.4 Secure car key**

Have you noticed that the keys for many new automobiles have a thicker plastic head than in the past (see Figure 3.49)? The reason is that, believe it or not, there is a computer chip inside the head of the key, implementing a secure car key. In a basic version of such a secure car key, when the driver turns the key in the ignition, the car’s computer (which is under the hood and communicates using what’s called the *base station*) sends out a radio signal asking the car key’s chip to respond by sending an identifier via a radio signal. The chip in the key then responds by sending the identifier

![Figure 3.49 Why are the heads of car keys getting thicker? Note that the key on the right is thicker than the key on the left. The key on the right has a computer chip inside that sends an identifier to the car’s computer, thus helping to reduce car thefts.](image)
(ID), using what’s known as a transponder (a transponder “transmits” in “response” to a request). If the basestation does not receive a response or the key’s response has an ID different than the ID programmed into the car’s computer, the computer shuts down and the car won’t start.

Let’s design the controller for such a key, having an ID of 1011 (real IDs are typically 32 bits long or more, not just 4 bits). Assume the controller has an input \( a \) that is 1 when the car’s computer requests the key’s ID. Thus the controller initially waits for the input \( a \) to become 1. The key should then send its ID (1011) serially, starting with the rightmost bit, on an output \( z \); the key sends 1 on the first clock cycle, 1 on the second cycle, 0 on the third cycle, and finally 1 on the fourth cycle. The FSM for the controller is shown in Figure 3.50. Note that the FSM sends the bits starting from the bit on the right, which is known as the least significant bit (LSB).

The computer chip in the car key has circuitry that converts radio signals to bits and vice versa.

Figure 3.51 provides a timing diagram for the FSM for a particular situation. When we set \( a = 1 \), the FSM enters state \( K1 \) and outputs \( r = 1 \). The FSM then proceeds through \( K2, K3, \) and \( K4 \), outputting \( r = 1, 0, \) and 1, respectively, even though we returned input \( a \) to 0.

Timing diagrams represent a particular situation defined by how we set the inputs. What would have happened if we had held \( a = 1 \) for many more clock cycles? The timing diagram in Figure 3.52 illustrates that situation. Notice how in that case the FSM, after returning to state \( K1 \), proceeds to state \( K1 \) again on the next cycle.

“So my car key may someday need its batteries replaced?” you might ask. Actually, no—those chips in keys draw their power, as well as their clock, from the magnetic component of the radio-frequency field generated from the computer basestation, as in RFID chips. The extremely low power requirement makes custom digital circuitry, rather than instructions on a microprocessor, a preferred implementation.

Computer chip keys make stealing cars a lot harder—no more “hot-wiring” to start a car, since the car’s computer won’t work unless it also receives the correct identifier. And the method above is actually an overly simplistic method—many cars have more sophisticated communication between the computer and the key, involving several communications in both directions, even using encrypted communication—making fooling the car’s computer even harder. A drawback of secure car keys is that you...
can’t just run down to the local hardware store and copy those keys for $5 any longer—copying keys requires special tools that today can run $50-$100. A common problem while computer chip keys were becoming popular was that low-cost locksmiths didn’t realize the keys had chips in them, so copies were made and the car owners went home and later couldn’t figure out why their car wouldn’t start, even though the key fit in the ignition slot and turned.

Example 3.5 Flight-attendant call button

This example uses an FSM to describe the desired behavior of the flight-attendant call button system from Figure 3.1. The FSM has inputs Call and Cncl for the call and cancel buttons, and output L to control the light. Call will be given priority if both buttons are pressed. The FSM has two states, LightOff, which sets L to 0, and LightOn, which sets L to 1, as shown in Figure 3.53. Light-Off is the initial state. The FSM stays in that state until Call is 1, which causes a transition to LightOn. If Call is 0, the FSM stays in LightOff. In state LightOn, the only way to transition back to LightOff is if Cncl is 1 and Call is 0 (because the call button has priority), meaning Cncl·Call. If that condition is false, i.e., (Cncl·Call) is true, the FSM stays in LightOn.

Notice how clearly the FSM captures the behavior of the flight-attendant call button system. Once you understand FSMs, an FSM description is likely to be more concise and precise than an English description.

How to Capture Desired System Behavior as an FSM

The previous section showed FSM examples, but how were those FSMs originally created? Creating an FSM that captures desired system behavior can be a challenging task for a designer. Using the following method can help:

- List states: First list all possible states of the system, giving each a meaningful name, and denoting the initial state. Optionally add some transitions if they help indicate the purpose of each state.
- Create transitions: For each state, define all possible transitions leaving that state.
- Refine the FSM: Execute the FSM mentally and make any needed improvements.

The method described above is just a guide. Capturing behavior as an FSM may require some creativity and trial-and-error, as is the case in some other engineering tasks, like computer programming. For a complex system, a designer may at first list a few states, and then upon defining transitions the designer may decide that more states are required. While creating an FSM, the preciseness of the FSM may cause the designer to realize that the system’s behavior should be different than originally anticipated. Note also that many different FSMs could be created that describe the same desired behavior; one FSM may be easier to understand while another FSM may have fewer states, for example. Experience can help greatly in creating correct and easy-to-understand FSMs that capture desired system behavior.
image not available
the FSM should go to state Red1; we’d already added that transition \((a \cdot \overline{r})\). If a button is pressed and that button is not the red button \((a \cdot \overline{r})\), then the FSM should somehow enter a “fail” mode and not unlock the door. At this point, we might consider adding another state called Fail. Instead, we decide that the FSM should go back to the Wait state and just wait for the start button to be pressed again, so we add such a transition with condition \(a \cdot \overline{r}\) as shown.

The pattern of three transitions for state Start can be replicated for states Red1, Blue, and Green, modified to detect the correct colored button press as shown in Figure 3.57. Finally, we must decide what the FSM should do after the FSM reaches state Red2 and unlocks the door. For simplicity of this example, we decide to have the FSM just return to state Wait, which locks the door again; a real system would keep the door unlocked for a fixed period of time before locking it again.

**Refine the FSM:** We can now mentally execute the FSM to see if it behaves as desired:

- The FSM begins in the Wait state. As long as the start button is not pressed \((a \cdot \overline{r})\), the FSM stays in Wait; when the start button \(a\) is pressed (and a rising clock edge arrives, of course), the FSM goes to the Start state.
- Being in the Start state means the FSM is now ready to detect the sequence red, blue, green, red. If no button is pressed \((a \cdot \overline{r})\), the FSM stays in Start. If a button is pressed AND that button is the red button \((a \cdot \overline{r})\), the FSM goes to state Red1. Instead, if a button is pressed AND that button is not the red button \((a \cdot \overline{r})\), the FSM returns to the Wait state—note that when in the Wait state, further presses of the colored buttons would be ignored, until the start button is pressed again.
- The FSM stays in state Red1 as long as no button is pressed \((a \cdot \overline{r})\). If a button is pressed AND that button is blue \((ab)\), the FSM goes to state Blue; if that button is not blue \((ab \cdot \overline{r})\), the FSM returns to state Wait. At this point, we detect a potential problem—what if the red button is still being pressed as part of the first button press when the next rising clock edge arrives? The FSM would go to state Wait, which is not what we want. One solution is to add another state, Red1_Release, that the FSM transitions to after Red1, and in which the FSM stays until \(a \cdot \overline{r}\) is not pressed. For simplicity, we’ll instead assume that each button has a special circuit that synchronizes the button with the clock signal. That circuit sets its output to \(1\) for exactly one clock cycle for each unique press of the button. This is necessary to ensure that the current state doesn’t inadvertently change to another state if a button press lasts longer than a single clock cycle. We’ll design such a synchronization circuit in Example 3.9.
- Likewise, the FSM stays in state Blue as long as no button is pressed \((a \cdot \overline{r})\), and goes to state Green on condition \(a \cdot \overline{r}\), and state Wait on condition \(a \cdot \overline{r}\).
- Finally, the FSM stays in Green if no button is pressed, and goes to state Red2 on condition \(a \cdot \overline{r}\), and to state Wait on condition \(a \cdot \overline{r}\).
- If the FSM makes it to state Red2, that means that the user pressed the buttons in the correct sequence—Red2 will set \(u = 1\), thus unlocking the door. Note that all other states set \(u = 0\). The FSM then returns to state Wait.
The FSM works well for normal button presses, but let’s mentally execute the FSM for unusual cases. What happens if the user presses the start button and then presses all three colored buttons simultaneously, four times in a row? The way the FSM is defined, the door would unlock! A solution to this undesirable situation is to modify the transitions between the states that detect correct colored button presses, to detect not only the correct colored button press, but also that the other colored buttons are not pressed. For example, for the transition leaving state Start with condition $ar$, the condition should instead be $a \langle rb' g' \rangle$. That change also means that the transition going back to state Wait should have the condition $a \langle rb' g' \rangle 1$. The intuitive meaning of that condition is that a button was pressed, but it was not just the red button. Similar changes can be made to the other transition conditions too, resulting in the improved FSM of Figure 3.58.

### 3.4 CONTROLLER DESIGN

#### Standard Controller Architecture for Implementing an FSM as a Sequential Circuit

The previous section provided examples of capturing sequential behavior using FSMs. This section defines a process to convert an FSM to a sequential circuit. The sequential circuit that implements an FSM is commonly called a controller. Converting an FSM to a controller is quite straightforward when a standard pattern, commonly called a standard architecture, is used for the controller. Other ways exist for implementing an FSM, but using the standard architecture results in a straightforward design process.

A standard controller architecture for an FSM consists of a register and combinational logic. For example, the standard controller architecture for the laser timer FSM of Figure 3.45 is shown in Figure 3.59. The controller’s register stores the current FSM state and is thus called a state register. Each state is represented as a unique bit encoding. For example, the laser timer’s Off state could be encoded as 00, On1 as 01, On2 as 10, and On3 as 11, the four states thus requiring a 2-bit state register.

The combinational logic computes the output values for the present state, and also computes the next state based on the current state and current input values. Its inputs are thus the state register bits ($s_1$ and $s_0$ in the example of Figure 3.59) and the FSM’s external inputs ($b$ for the example). The combinational logic’s outputs are the outputs of
We then obtain the sequential circuit in Figure 3.62, implementing the FSM.

![Figure 3.62 Final implementation of the three-cycles-high laser timer controller.](image)

Many textbooks use different table organizations from that in Table 3.3. However, we intentionally organized the table so that it serves both as a state table, which is a tabular representation of an FSM, and as a truth table that can be used to design the combinational logic of the controller.

**Example 3.8 Understanding the laser timer controller’s behavior**

To aid in understanding how a controller implements an FSM, this example traces through the behavior of the three-cycles-high laser timer controller. Assume the system is initially in state 00 (s1s0=00), b is 0, and the clock is currently low. As shown in Figure 3.63(a), based on the combinational logic, x will be 0 (the desired output in state 00), n1 will be 0, and n0 will be 0, meaning the value 00 will be waiting at the state register’s inputs. Thus, on the next clock edge, 00 will be loaded into the state register, meaning the system stays in state 00—which is correct.

Now suppose b becomes 1. As shown in Figure 3.63(b), x will still be 0, as desired. n1 will be 0, but n0 will be 1, meaning the value 01 will be waiting at the state register’s inputs. Thus, on the next clock edge, 01 will be loaded into the state register, as desired.

As in Figure 3.63(c), soon after 01 is loaded into the state register, x will become 1 (after the register is loaded, there’s a slight delay as the new values for s1 and s0 propagate through the combinational logic gates). That output is correct—the system should output x=1 when in state 01. Also, n1 will become v and n0 will equal 0, meaning the value 10 will be waiting at the state register inputs. Thus, on the next clock edge, 10 will be loaded into the state register, as desired.

After 10 is loaded into the state register, x will stay 1, and n1n0 becomes 11. When another clock edge comes, 11 will be loaded into the register, x will stay 1, and n1n0 becomes 00.

When another clock edge comes, 00 will be loaded into the register. Soon after, x will become 0, and if b is 0, n1n0 will stay 00; if b is 1, n1n0 will become 01. Notice that the system is back in the state where it started.

Understanding how a state register and combinational logic implement a state machine can take a while, since in a particular state (indicated by the value presently in the state register), we generate
bo=0 again, so that bo was 1 for just one cycle, as desired. The FSM goes from B to A if bi returned to 0. If bi is still 1, the FSM goes to state C, where the FSM waits for bi to return 0, causing a transition back to state A.

**Step 2A: Set up the architecture.** Because the FSM has three states, the architecture has a two-bit state register, as in Figure 3.65(b).

**Step 2B: Encode the states.** The three states can be straightforwardly encoded as 00, 01, and 10, as in Figure 3.65(c).

**Step 2C: Fill in the truth table.** We convert the FSM with encoded states to a truth table for the controller’s combinational logic, as shown in Figure 3.65(d). For the unused state 11, we have chosen to output bo=0 and return to state 00.

**Step 2D: Implement the combinational logic.** We derive the equations for each combinational logic output, as shown in Figure 3.65(e), and then create the final circuit as shown.

**Figure 3.65** Button press synchronizer design steps: (a) initial FSM, (b) controller architecture, (c) FSM with encoded states, (d) truth table for the combinational logic, (e) final controller with implemented combinational logic.
Example 3.10  Sequence generator

This example designs a sequential circuit with four outputs: \( w, x, y, \) and \( z \). The circuit should generate the following sequence of output patterns: \( 0001, 0011, 1100, \) and \( 1000 \), one per clock cycle. After \( 1000 \), the circuit should repeat the sequence. Sequence generators are common in a variety of systems, such as a system that blinks a set of four lights in a particular pattern for a festive lights display. Another example is a system that rotates an electric motor a fixed number of degrees each clock cycle by powering magnets around the motor in a specific sequence to attract the magnetized motor to the next position in the rotation—known as a **stepper motor**, because the motor rotates in steps.

The sequence generator controller can be designed using the controller design process:

**Step 1:** Capture the FSM. Figure 3.66 shows an FSM having four states labeled \( A, B, C, \) and \( D \) (though any other four unique names would do just fine) to generate the desired sequence.

**Step 2A:** Set up the architecture. The standard controller architecture for the sequence generator will have a 2-bit state register to represent the four possible states, no inputs to the logic, and outputs \( w, x, y, z \) from the logic, along with outputs \( n_1 \) and \( n_0 \), as shown in Figure 3.67.

**Step 2B:** Encode the states. The states can be encoded as follows—\( A: 00, B: 01, C: 10, D: 11 \). Any other encoding with a unique code for each state would also be fine.

**Step 2C:** Fill in the truth table. Table 3.4 shows the table for the FSM with encoded states.

**Step 2D:** Implement the combinational logic. An equation can be derived for each output of the combinational logic directly from the truth table. After some algebraic simplification, the equations are those shown below. The final circuit is shown in Figure 3.68.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_1 s_0 )</td>
<td>( w \ x \ y \ z \ n_1 n_0 )</td>
</tr>
<tr>
<td>( A )</td>
<td>0 0 0 0 1 0 1</td>
</tr>
<tr>
<td>( B )</td>
<td>0 1 0 1 1 1 0</td>
</tr>
<tr>
<td>( C )</td>
<td>1 0 1 1 0 0 1 1</td>
</tr>
<tr>
<td>( D )</td>
<td>1 1 1 0 0 0 0 0</td>
</tr>
</tbody>
</table>

**Figure 3.66** Sequence generator FSM.

**Figure 3.67** Sequence generator controller architecture.

**Figure 3.68** Sequence generator controller with implemented combinational logic.
Example 3.11 Secure car key controller (continued)

Let’s complete the design for the secure car key controller from Example 3.4. We already carried out Step 1: Capture the FSM, shown in Figure 3.50. The remaining steps are as follows.

Step 2A: Set up the architecture. The FSM has five states, and thus requires a 3-bit state register, which can represent eight states; three states will be unused. The inputs to the combinational logic are $a$ and the three state bits $s_2$, $s_1$, and $s_0$, while the outputs are signal $r$ and next state outputs $n_2,n_1,n_0$. The architecture is shown in Figure 3.69.

Step 2B: Encode the states. Let’s encode the states using a straightforward binary encoding of 000 through 100. The FSM with state encodings is shown in Figure 3.70.

Step 2C: Fill in the truth table. The FSM converted to a truth table for the logic is shown in Table 3.5. For the unused states, we have chosen to set $x = 0$ and the next state to 000.

![Figure 3.69 Secure car key controller architecture.](image)

![Figure 3.70 Secure car key FSM with encoded states.](image)

### Table 3.5 Truth table for secure car key controller’s combinational logic.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_2$ $s_1$ $s_0$ $a$</td>
<td>$r$ $n_2$ $n_1$ $n_0$</td>
</tr>
<tr>
<td>Wait</td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>0 0 0 1 0 0 0 1</td>
<td></td>
</tr>
<tr>
<td>K1</td>
<td>0 0 1 0 1 0 1 0</td>
</tr>
<tr>
<td>0 0 1 1 1 0 1 0</td>
<td></td>
</tr>
<tr>
<td>K2</td>
<td>0 1 0 0 1 0 1 1</td>
</tr>
<tr>
<td>0 1 0 1 1 0 1 1</td>
<td></td>
</tr>
<tr>
<td>K3</td>
<td>0 1 1 0 0 1 0 0</td>
</tr>
<tr>
<td>0 1 1 1 0 1 0 0</td>
<td></td>
</tr>
<tr>
<td>K4</td>
<td>1 0 0 0 1 0 0 0</td>
</tr>
<tr>
<td>1 0 0 1 1 0 0 0</td>
<td></td>
</tr>
<tr>
<td>Unused</td>
<td>1 0 1 0 0 0 0 0</td>
</tr>
<tr>
<td>1 0 1 1 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>1 1 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>1 1 0 1 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>1 1 1 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>1 1 1 1 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>
Converting a Circuit to an FSM (Reverse Engineering)

We showed in Section 2.6 that a circuit, truth table, and equation were all forms able to represent the same combinational function. Similarly, a circuit, state table, and FSM are all forms able to represent the same sequential function.

The process in Table 3.2 for converting an FSM to a circuit can be applied in reverse to convert a circuit to an FSM. In general, converting a circuit to an equation or FSM is known as reverse engineering the behavior of the circuit. Not only is reverse engineering useful to help develop a better understanding of sequential circuit design, but it can also be used to understand the behavior of a previously-designed circuit such as a circuit created by a designer who is no longer at a company, and also to check that a circuit we designed has the correct behavior.

**Example 3.12 Converting a sequential circuit to an FSM**

Given the sequential circuit in Figure 3.71, find an equivalent FSM. We start from step 2D in Table 3.2. The combinational circuit already exists. Step 2C fills in a truth table. The combinational logic in the controller architecture has three inputs: two inputs \( s_1 \) and \( s_0 \) represent the contents of the state register, and \( x \) is an external input. Thus the truth table will have 8 rows because there are \( 2^3 = 8 \) possible combinations of inputs. After listing the truth table and enumerating all combinations of inputs (e.g., \( s_1 s_0 x = 000, ..., s_1 s_0 x = 111 \)), the techniques in Section 2.6 can be used to fill in the values of the outputs. Consider the output \( y \). The combinational circuit shows that \( y = s_1 \). Knowing this, we place a 1 in the \( y \) column of the truth table in every row where \( s_1 = 0 \), and place a 0 in the remaining spaces in the \( y \) column. Consider \( n_0 \), which the circuit shows as having the Boolean equation \( n_0 = s_1 ' s_0 ' x \). Accordingly, we set \( n_0 = 1 \) when \( s_1 = 0 \) and \( s_0 = 0 \) and \( x = 1 \). We fill in the columns for \( z \) and \( n_1 \) using a similar analysis and move on to the next step.

Step 2B encodes the states. The states have already been encoded, so this step in reverse assigns a name to encoded state. We arbitrarily choose the names \( A, B, C, \) and \( D \), seen in Table 3.6.

Step 2A sets up the standard controller architecture. This step requires no work since the controller architecture was already defined.

Finally, step 1 captures the FSM. Initially, we can set up an FSM diagram with the four states whose names were given in step 2A, shown in Figure 3.72(a). Next, we list the values of the FSM outputs \( y \) and \( z \) next to each state as defined by the truth table.
A designer can verify the above two properties using Boolean algebra. The exclusive transitions property can be verified by ensuring that the AND of every pair of conditions on a state’s transitions always results in 0. For example, if a state has two transitions, one with condition $x$ and the other with condition $x\, y$, transformations of Boolean algebra can be used as follows:

\[
\begin{align*}
x \cdot x\, y &= (x \cdot x') \cdot y \\
&= 0 \cdot y \\
&= 0
\end{align*}
\]

If a state has three transitions with conditions $C1$, $C2$, and $C3$, the designer can verify that $C1 \cdot C2 = 0$, $C1 \cdot C3 = 0$, and finally that $C2 \cdot C3 = 0$, thus verifying that every pair yields 0. Note that verifying that $C1 \cdot C2 \cdot C3 = 0$ does not verify that the transitions are exclusive; for example, if $C1$ and $C2$ were exclusive but $C2$ and $C3$ were not, $C1 \cdot C2 \cdot C3$ would still equal 0 because $0 \cdot C3 = 0$.

The second property of complete transitions can be verified by checking that the OR of all the conditions on a state’s transitions results in 1. Considering the same example of a state that has two transitions, one with condition $x$ and the other with condition $x\, y$, transformations of Boolean algebra can be applied as follows:

\[
\begin{align*}
x + x\, y &= x + (1 + y) \cdot x\, y \\
&= x + xy + x\, y \\
&= x + (x + x') y \\
&= x + y
\end{align*}
\]

The OR of those two conditions is not 1 but rather $x + y$. If $x$ and $y$ were both 0, neither condition would be true, and so the next state would not be specified in the FSM. Figure 3.75(b) fixed this problem by adding another transition, $x\, y'$. Checking these transitions yields:

\[
\begin{align*}
x + x\, y + x\, y' &= x + x' (y + y') \\
&= x + x' \cdot 1 \\
&= x + x' \\
&= 1
\end{align*}
\]

If a state has three transitions with conditions $C1$, $C2$, and $C3$, the designer can verify that $C1 + C2 + C3 = 1$.

Proving the properties for the transitions of every state can be time-consuming. A good FSM capture tool will verify the above two properties automatically and inform the designer of any problems.
image not available
3.5 MORE ON FLIP-FLOPS AND CONTROLLERS

Non-Ideal Flip-Flop Behavior

When first learning digital design we assume ideal behavior for logic gates and flip-flops, just like when first learning physics of motion we assume there’s no friction or wind resistance. However, there is a non-ideal behavior of flip-flops—metastability—that is such a common problem in the practice of real digital design, we feel obliged to discuss the issue briefly here. Digital designers in practice should study metastability and possible solutions quite thoroughly before doing serious designs. Metastability comes from failing to meet flip-flop setup or hold times, which are now introduced.

Setup Times and Hold Times

Flip-flops are built from wires and logic gates, and wires and logic gates have delays. Thus, a real flip-flop imposes some restrictions on when the flip-flop’s inputs can change relative to the clock edge, in order to ensure correct operation despite those delays. Two important restrictions are:

- **Setup time:** The inputs of a flip-flop (e.g., the D input) must be stable for a minimum amount of time, known as the setup time, before a clock edge arrives. This intuitively makes sense—the input values must have time to propagate through any flip-flop internal logic and be waiting at the internal gates’ inputs before the clock pulse arrives.

- **Hold time:** The inputs of a flip-flop must remain stable for a minimum amount of time, known as the hold time, after a clock edge arrives. This also makes intuitive sense—the clock signal must have time to propagate through the internal gates to create a stable feedback situation.

A related restriction is on the minimum clock pulse width—the pulse must be wide enough to ensure that the correct values propagate through the internal logic and create a stable feedback situation.

A flip-flop typically comes with a datasheet describing setup times, hold times, and minimum clock pulse widths. A **datasheet** is a document that tells a designer what a component does and how to properly use that component.

Figure 3.79 illustrates an example of a setup time violation. D changed to 0 too close to the rising clock. The result is that R was not 1 long enough to create a stable feedback situation in the cross-coupled NOR gates with Q being 0. Instead, Q glitches to 0 briefly. That glitch feeds back to the top NOR gate, causing Q' to glitch to 1 briefly. That glitch feeds back to the bottom NOR gate, and so on. The oscillation would likely continue until a race condition caused the circuit to settle into a stable situation of Q = 0 or Q = 1—or the circuit could enter a metastable state, which we now describe.
image not available
image not available
image not available
image
not available
image not available
image not available
Index

Event 107, 167
event (in VHDl) 502
Exact algorithm 339
Excitation table 120
Exclusive (transitions) 142
Exclusive OR 81
Execute 466
Expand operation 346
Expanding (a term) 340
Expression 62

F
Fabrication plant (fab) 415
Factoring 350
Fahrenheit 213
Falling edge-triggered flip-flop 116
Fanout 225
Feature size 449
Field programmable gate array 424
FIFO (first-in-first-out) 300
FIFO queue 300
Filter 278
Filtering 313
Finite Impulse Response filter 276
Finite-state automata 126
Finite-state machine 124, 126, 505
FIR filter 276, 313, 384, 391
FIR filter (pipelined) 379
First-in-first-out (FIFO) 300
Fixed-point arithmetic 550
Flash memory 296
Floating point number 552
Flop 116
Flow-of-control instruction 480
FPGA 424, 445, 446, 448
Frame 393
Frame (video) 267
Frequency 117
FSM 124, 126
FSM with data 253
FSMD 253
Full-adder 184
Full-custom IC 414
Full-subtractor 197
Function cover 340
Functions (of a register) 175
Fuse-based PLD 441
Fuse-based ROM 294

G
GAL (generic array logic) 444
Gate 46
Gate (of a transistor) 41
Gate array 424
Gate array ASIC 417, 448
Gate delay 91
Gated D latch 112
Gated SR latch 111
Gbyte 21
General-purpose computer 4
General-purpose processor 461
Generate 370
Generic array logic (GAL) 444
Gigabyte 21
Glitching 151
Global register 303
Global signal 303
Google 12
Ground 36, 439

H
Hauß’s Law 453
Half-adder 184, 190, 417
Hard core 452
Hardware description language (HDL) 489
HDL x, 95, 489
Hertz 118, 278
Heuristic 339
Hex 16
Hexadecimal 16
Hierarchical carry-lookahead adder 374
Hierarchy 305
High-level state machine 248
HLSM 248
Hold time 146
Hz (Hertz) 118

I
IC 39, 40, 413
IC capacity 448
IC type 413
Idempotent 541
Idempotent law 58
Identity comparator 191
Identity element 538
Identity property 56
If-then statement 282
If-then-else statement 282
Implicant 340
Implication table 353
Increment 215
Incrementer 190
Indirect addressing 480
initial procedure (in Verilog) 499
Initial state 176, 150
Input (in Verilog) 491
Instance 259
Instantiate 259
Instruction memory 465
Instruction register 466
Sensitive (in Verilog) 496
Sensitive (in VHDL) 495
sensitive_pos (in SystemC) 503
Sensitivity list (in Verilog) 496, 504
Sensitivity list (in VHDL) 495, 504
Sensor 9
Sequential circuit 35
Sequential multiplier 375
Serial 386
Serial communication 175, 179
Serializing 383
Servant 114
Set (a latch) 109
Set (flip-flop input) 149
Setup time 146
Seven-segment display 72
Shannon, Claude 45
Shared bus 227
Shift 210
Shift register 173, 436
Shifter 211, 214, 259
Shifting (a register) 173
Shifts and adds 212
Shockley, William 38
SHR (in VHDL) 519
Sign bit 202
Signal (in VHDL) 490
Signed number 202, 212
Signed-magnitude representation 200
Significant 552
Silicon 40, 42
Silicon dioxide 41
Silicon Valley 42
Simple PLD 444
Simulation 94
Simulator 24
Single precision 552
Single-purpose processor 461
Size 326
Sliding average 211
Small-scale integration 439
SOC 450
Soda machine dispenser 248
Soft core (on an FPGA) 451
Software 22
Solid state 39
Source (of a transistor) 41
Spin (in IC fabrication) 415
SPLD 444
Spurious value 187
SRAM 288
SSI 39
SSI chip 439
Stable 108
Stage (of a pipeline) 378

Stages 466
Standard architecture (for sequential circuit) 132
Standard cell ASIC 416, 448
Standard representation 68
State 122
State diagram 124
State encoding 354
State minimization 351
State reduction 351
State register 132, 168, 505
State table 135
Static 289
Static RAM 288
std_logic (in VHDL) 489
std_logic_1164 (in VHDL) 489
std_logic_arith (in VHDL) 517
std_logic_unsigned (in VHDL) 517
std_logic_vector (in VHDL) 501
Stepper motor 138
Store operation 465
Storing (in a register) 168
Storing a bit 106
Stream 275
Structure 489
Structured ASIC 418, 446
Subtraction method 16
Subtractor 196
Sum 53
Sum of absolute differences (SAD) 281
Sum of minterms 71
Sum of absolute differences (SAD) 267
Sum-of-products 69, 340
Sum-of-products 55, 69, 74, 327, 422
Superscalar processor 482
Switch 36
Switch matrix 432
Switching algebra 537, 538
Synchronized processors 304
Synchronous circuit 117
Synchronous clear 180, 216
Synchronous reset 150
Synchronous set 150, 180
Synthesis 282
System-on-a-chip (SOC) 450

T
T flip-flop 120
Tabular method 341
Tap (in FIR filter) 314
Thyristor 21
Technology mapping 417, 427
Telephone 8
Terabyte 21
Term 55
Terminal count 215