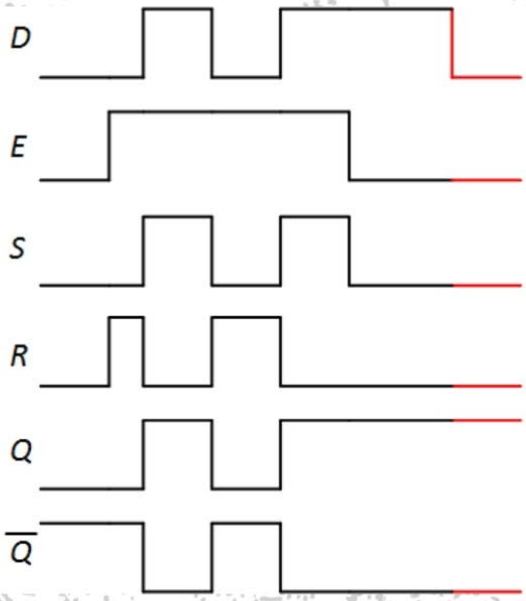
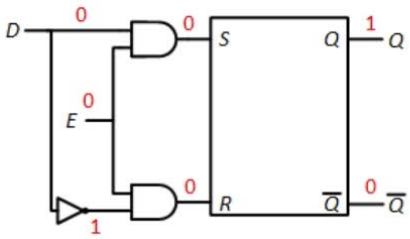
AN ANIMATED INTRODUCTION TO DIGITAL LOGIC DESIGN





JOHN D. CARPINELLI

This page intentionally left blank

An Animated Introduction to Digital Logic Design

First Edition

John D. Carpinelli, Ph.D.

Professor, Helen and John C. Hartmann Department of Electrical and Computer Engineering

New Jersey Institute of Technology, Newark, NJ

NJIT Library
New Jersey Institute of Technology
Newark, NJ, USA

AN ANIMATED INTRODUCTION TO DIGITAL LOGIC DESIGN

Published by NJIT Library, University Heights, Newark, NJ 07102

This open textbook is the first publication of the Faculty Authored Open Textbook (FAOT) program, part of NJIT's Open and Affordable Textbook Initiative, and was supported by NJIT's Sabbatical Leave Award. To learn more about NJIT's Open and Affordable Textbook Program, visit https://researchguides.njit.edu/opentextbooks/.

Book cover image by John D. Carpinelli

Book cover design by Ann D. Hoang

First electronic edition 2023. https://digitalcommons.njit.edu/oat/1/.

ISBN: 978-1-970194-005 (eBook)

DOI: https://doi.org/10.60826/kr55-0k56

To Mary, Maria, and Tony,

and Mom and Dad

About the Author

John D. Carpinelli, Ph.D. is a Professor of Electrical and Computer Engineering in the Helen and John C. Hartmann Department of Electrical and Computer Engineering at the New Jersey Institute of Technology. He holds a Bachelor of Engineering from Stevens Institute of Technology, a Master of Engineering, and a Ph.D. from Rensselaer Polytechnic Institute.

His research interests focus on engineering and STEM education and computer architecture. He has received several awards for his educational service and has been designated a Master Teacher at NJIT.

Use of Open Textbook Request

There is no cost for students and instructors, or anyone, to use this book, either in a formal course or for self-study. If you find this book useful, I encourage you to consider "paying" for it by doing something to help others. Possibilities include a monetary donation to a charity, if your means permit, donating time to a worthy cause, working on a community outreach activity with a campus group, or any other way you can imagine. This is completely optional, and you're welcome to use this book whether or not you choose to do this. To be clear, do not send any money to me personally. I want to state explicitly that I am not asking for, and I will not accept any donations. There are many deserving organizations that would benefit from your assistance.

John Carpinelli

Preface

This book is designed for use in an introductory course on digital logic design, typically offered in computer engineering, electrical engineering, computer science, and other related programs. Such a course is usually offered at the sophomore level. This book makes extensive use of animation to illustrate the flow of data within a digital system and to step through some of the procedures used to design and optimize digital circuits.

My Approach

There are numerous textbooks available that focus on digital logic design, and many of them are quite good. Any printed book, however, is limited in how it can present information. From its inception, I wanted to do something different with this book. Publishing this solely as an electronic book (eBook), I wanted to take advantage of the medium to present material in a way that I could not do in a traditional printed book. For this reason, this book incorporates animation to illustrate the functioning and flow of data within circuits throughout the book.

Over the past few years, I have perceived a shift in the learning styles of my undergraduate students. I believe that more of my students now are visual learners than in my earlier years in academia. The animations are an attempt to address this change and improve student learning.

The animations vary widely in complexity, ranging from showing all possible inputs and outputs of an AND gate to stepping through a function minimization using the Quine-McCluskey algorithm to showing the timing of data values changing within a sequential circuit. The animations are not very fancy; they are meant to convey knowledge with a minimum of distractions. Pixar certainly receives no competition from the animations in this book.

I like to include a lot of design work in my courses. I believe this helps students attain a deeper understanding of the material. For this reason, this book contains numerous design examples to illustrate circuit functions and design methodologies. The end-of-chapter problems are also weighted heavily toward design.

I have found that illustrating concepts using simple examples helps my students learn the material better. Some are absurdly simple. No company manufactures an 8x2 memory chip, but showing how such a chip functions gives students the base knowledge they need to design larger, more realistic memory chips.

Scope of Coverage

This book consists of ten chapters divided into four parts, as described below.

Part I: Background

This part, comprising the first two chapters, provides fundamental background information. The first chapter focuses on number systems, with an emphasis on binary numbers and different formats used to represent numbers. Several representations for nonnumeric data are also presented. This chapter also introduces Gray codes, special sequences of binary values that change only one bit at a time. Later, we use Gray codes to minimize functions and to design sequential circuits.

Chapter 2 introduces Boolean algebra, the mathematical basis for digital logic design. This chapter covers the fundamental Boolean functions and mechanisms used to represent function values, including truth tables, sum of products, product of sums, minterms, and maxterms. Minimizing functions is an essential design practice that reduces the hardware needed to implement the function. Two mechanisms used to minimize functions, Karnaugh maps and the Quine-McCluskey algorithm, are presented. This chapter concludes by examining how to develop and minimize functions when the function value is not specified for some combinations of input values.

Part II: Combinatorial Logic

Chapters 3, 4, and 5 focus on combinatorial logic, circuits that produce output values based solely on the current values of the inputs. Chapter 3 introduces the basic logic gates, digital components that implement the fundamental Boolean functions introduced earlier in this book. This chapter also presents methods to combine these gates to realize more complex functions and the inverse of these functions. Digital components are subject to real-world constraints that may not be reflected in the underlying Boolean algebra models. This chapter discusses fan-in and fan-out, buffers, and propagation delay.

Some functions are used very frequently in digital design. Rather than recreate circuits for these functions every time they are used, engineers have created chips and components to implement these functions. When needed in a circuit, a digital design can simply plug the chip directly into a circuit. This chapter introduces the function and design of these components: decoders, encoders, multiplexers, demultiplexers, comparators, and adders. It also discusses tri-state buffers, a special type of buffer with numerous applications in digital logic design.

Chapter 5 uses some of these components to implement Boolean functions. It presents methodologies for realizing functions using multiplexers and decoders. This chapter also introduces ROMs, read-only memory components that can be used to realize functions by looking up function values. This chapter presents only the very basics of ROMs. They are covered in more detail later in this book.

Part III: Sequential Logic

Unlike combinatorial logic, sequential logic generates outputs based on both its input values and its current state, or current internal conditions or values. Depending on its current state, the same input values may produce different output values and change to different next states.

Chapter 6 introduces the basic model for sequential circuits and the fundamental components. Latches come in several varieties, including the S-R and D latches. This chapter introduces these components and their internal designs. Flip-flops, like latches, store binary values. Unlike latches, however, they load in values only on a clock edge, when the clock value changes from 0 to 1 or from 1 to 0. This chapter presents the internal designs and overall function of the D, J-K, and T flip-flops.

Just as engineers created more complex components for frequently used combinatorial functions, they have done the same for sequential components. This chapter introduces registers, which are used to store multi-bit values. Some registers can perform operations on their data. This chapter introduces several varieties of shift registers and counters and their internal designs.

With this foundational knowledge in place, Chapter 8 focuses on the design of sequential circuits. There are two primary models for sequential circuits: Mealy and Moore machines. Both are introduced and used to design several sequential systems. This chapter also introduces tools used in the design of these systems, including state diagrams and state tables. The designs presented here use fundamental and more complex components; this chapter discusses the methodology used for each of these. It also examines how to refine and optimize designs when some values are not specified, how to identify and combine equivalent states, and how to deal with signal glitches.

Part IV: Advanced Topics

The final two chapters present two advanced topics. Chapter 9 presents asynchronous circuits. Unlike the sequential circuits introduced in the previous chapter, asynchronous circuits do not have a clock signal to coordinate the flow of data within the circuit. This impacts how we analyze and design these circuits. Asynchronous circuit designers must contend with several issues that are not found, or are found much less frequently, in sequential circuits. This chapter presents methods to identify and eliminate issues including unstable circuits, static and dynamic hazards, and race conditions.

The final chapter examines programmable components. Programmable logic devices incorporate numerous logic gates and, for some PLDs, flip-flops. Designers specify the connections between components and between components and system inputs and outputs, to design a circuit within a single chip. This chapter also expands on the read-only memory (ROM) component briefly introduced in Chapter 5. It discusses the different types of memory devices and their internal designs.

Acknowledgments

Although I am the sole author of this book, its creation, and development were influenced and implemented by a large number of people. The greatest influence has been my students. They have given me very useful feedback as to what works best when presenting material. This led me to incorporate animation throughout this book and has also moved me to change how I present material in my courses, for the better, I hope.

Most of this book was written during a sabbatical leave provided by my home university, the New Jersey Institute of Technology, through its Faculty Authored Open Textbook (FAOT) program, which is a part of NJIT's Open and Affordable Textbook Initiative. I am grateful to NJIT for providing this opportunity. I could not have completed this textbook in this timeframe, or possibly at all, without this support.

Through the Open and Affordable Textbook (OAT) Initiative's FAOT program, NJIT provided support for the production of the final version of this book, converting my text and graphics files into a final product. Thanks to Ann Hoang, University Librarian and coordinator of the OAT's FAOT program, and her staff who brought this book to fruition: Matthew Brown, Jennifer A. King, John Kromer, Jill Lagerstrom, Lisa Lakomy, Lisa Weissbard, Tim Valente, and Raymond Vasquez.

My colleagues at NJIT, both in the Helen and John C. Hartmann Department of Electrical and Computer Engineering and throughout the university, have had a great influence on this book, both its material and how it is presented. Special thanks to Professor Jacob Savir, course coordinator for our digital system design course. His advice, and especially his notes on asynchronous systems, were very helpful when preparing this material for this book.

I created most of the figures in this book using Microsoft Visio, with some figures using Word, Excel, and Paint for tables and minor tweaks. Thanks to Texas Instruments for granting permission to reproduce a couple of figures from their publications. I created the animations using the free converters at EZGIF.COM, with a little help from Adobe Photoshop for a couple of the more complex animations. Many thanks to the team at EZGIF for making this tool available to me and everyone.

Finally, last but certainly not least, I thank my family and friends for their love and support throughout the time writing this book, and really throughout my entire life. I especially thank my wife, Mary, my children, Maria and Tony, and my parents Dominick and Nina, to all of whom I dedicate this book.

Table of Contents

PART I: BACKGROUND

PART II: COMBINATORIAL LOGIC

PART III: SEQUENTIAL LOGIC

PART IV: ADVANCED TOPICS

PART I

Introduction and Background

Chapter 1

Digital Systems and Numbers

Chapter 1: Digital Systems and Numbers

Digital systems are everywhere. From consumer appliances to the most advanced supercomputer, including the phone/tablet/computer you're using right now to read this e-book, virtually no aspect of our lives is untouched by these systems. To design these systems, the designer needs to understand what digital systems are (and are not), the mathematical background underlying digital systems, the components used in these systems, and how the digital design process works. We'll look at all of these as we progress through the book. In this chapter, we'll start with the basics.

Digital systems use **binary** values. Mathematically, binary is a base-2 system, where each value within the system is either 0 or 1. We'll look at what this means and how we represent these values in various media.

Just as we represent decimal numbers using more than one digit, we can represent binary numbers using more than one binary digit, or **bit**. In this chapter, we'll introduce several numeric formats used in some digital systems.

Binary values can also be used to represent nonnumeric data, such as alphanumeric characters (letters, numbers, and symbols). We'll look at a couple of the most commonly used formats.

Finally, this chapter introduces **Gray code**, a reflected binary code. We will make use of this code in Chapter 2 as we optimize our first digital designs.

1.1 What Are Digital Systems?

The world is digital! Everything is run by computers! Digital this! Digital that! Blah, blah, blah. You've probably heard all of this and more. But what is digital? What does it mean to be digital? What is something if it isn't digital? And how do you design something that is digital?

These are all fair questions. The first three are pretty straightforward, and we'll discuss them shortly. However, exploring that last question could fill many textbooks.. This book will cover the basics of digital logic design for relatively simple systems. More advanced systems, such as computer systems and microprocessor design, build on the design methodologies presented here, but they are much more complex than those included in this book.

To answer the first question, let's start with a quote.

> *There may be said to be two kinds of people in the world, those who constantly divide everybody into two classes, and those who do not.* – Robert Benchley

In this section, we're going to be the kind of people who divide electrical signals into two classes. **Analog** signals are continuous. They can take on any value within a given range. In the sine wave shown in Figure 1.1 (a), the amplitude (height) of the sine wave can take on any value from -1 to +1. Any value in this range corresponds to the amplitude at some point(s) on the sine wave.

(a) (b) (c)

Figure 1.1: (a) Analog sine wave; (b) Digital representation with nine possible values; (c) Digital representation with nineteen possible values

In contrast, digital signals are **quantized**. They can only take on specific values, and there are gaps between these values. The digital signals can only approximate the equivalent analog values. Consider a digital system that can represent only these nine values: -1, -¾, -½, -¼, 0, ¼, ½, ¾, and 1. Figure 1.1 (b) shows how we could represent the sine wave in this digital system by rounding the value of the amplitude to its nearest digital value.

The digital wave is rather choppy as compared to the analog wave. However, we can make the digital wave more like the analog wave by increasing the number of possible values. The digital wave in Figure 1.1 (c) uses a digital system with 19 possible values: -1, -0.9, -0.8, -0.7, -0.6, -0.5, -0.4, -0.3, -0.2, -0.1, 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, and 1. You can see this wave is not as smooth as the analog wave, but it is much smoother than the previous digital wave.

So, if analog gives us smoother curves and more accurate values, why should we even use digital? There are many reasons to use digital. Digital signals are less susceptible to noise during transmission, resulting in less distortion and better signal quality. Furthermore, transmission across greater distances is possible with digital than with analog. Digital signals can transmit more information (per second) than analog signals. Digital circuits are often cheaper to design and manufacture than analog circuits. There are other reasons as well.

1.2 One and Zero

There are 10 kinds of people in the world, those who know binary, and those who don't. – Anonymous (with apologies to Robert Benchley)

When we examine digital logic, we see that it uses discrete values. Unlike decimal, which has ten digits, digital logic uses **binary** values. Binary is base 2, as opposed to base 10 for decimal, and it has only two digits. These are typically represented as 1 and 0. Each value is called a **bit,** short for binary digit. You can also think of these values as true and false (which can be helpful when we discuss Boolean algebra in Chapter 2), on and off (like a light switch, but only a regular toggle switch, not a dimmer switch), or yes and no. As we design digital circuits, we'll mostly use 1 and 0.

So, why do we use binary? Why not use ternary (base 3) or some other base? As it turns out, it is much simpler to design electronics that generate two possible values than it is to design circuits that can output three or more possible values. If we need more possible values,

we can combine more outputs to represent these values. We'll look at how to do this for numeric values in the next section.

Now back to the electronics. Digital logic gates, the fundamental building blocks of digital systems, are constructed from transistors. Without going too in-depth, a transistor can be used in a circuit that is configured as a switch. When the switch is off, the circuit generates a very low voltage that corresponds to a logic value of 0. When it is on, it outputs a voltage that is interpreted as a logic value of 1. The circuit does not spend any appreciable time in the region between these two extremes, though it does spend some minimal time there as it transitions from one extreme to the other. Figure 1.2 shows these regions for transistor-transistor logic.

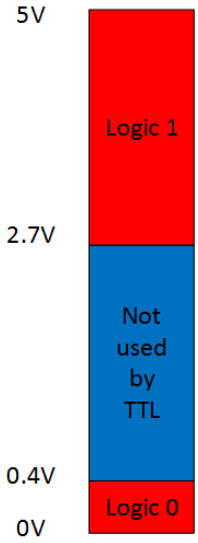


Figure 1.2: Voltage levels for transistor-transistor logic with a 5V source.

There are numerous ways to represent 1 and 0. As we just saw, voltage values are one way to do this. For TTL, transistor-transistor logic, 2.7V to 5V typically represents the logic value 1 and 0V to 0.4V represents logic 0. Values for other technologies are shown in Figure 1.3. Note that one technology, emitter-coupled logic, uses a negative power supply and thus has negative voltage values for its logic levels.

Technology	Source voltage	Logic 1	Logic 0
TTL: transistor-transistor logic	5V	2.7-5V	0-0.4V
CMOS: complementary metal-oxide semiconductor	5V 3.3V	4.44-5V 2.4-3.3V	0-0.5V 0-0.4V
ECL: Emitter-coupled logic	-5.2V	-0.9V	-1.75V
PECL: Positive ECL	5V 3.3V	4.1V 2.27-3.3V	0-3.3V 0-1.68V

Figure 1.3: Voltages corresponding to logic values for several technologies

Memory chips are used to store digital logic values. For the E^2PROM memory chips used in flash drives, SD cards, and solid-state drives, these values are similar to those of TTL chips.

Voltage is not the only way to store digital logic values, though any other method ultimately must be converted to voltages to be used by the rest of a digital system. A traditional (non-SSD) drive stores each bit by magnetizing a portion of the disc in one of two directions. The drive has two heads to read values from the disc. Reading the value induces a current from one head to the other. The current flows in one of two directions, from the first head to the second or vice versa, depending on the magnetization of the disc at that point. Circuitry (connected to the heads of the disc drive) outputs a voltage value corresponding to logic 1 or logic 0 that depends on the direction of current flow.

Optical discs, including CDs, DVDs, and Blu-Ray, encode logic values in one of two ways. For pre-recorded discs, such as a movie you might buy on Blu-Ray (if anyone is still buying discs by the time you read this), the discs are manufactured with areas called **pits** and **lands** that reflect light. The disc drive shines a laser onto a portion of a disc that stores one bit and senses the intensity of the reflected light. A higher intensity corresponds to a logic value of 1 and a lower intensity denotes logic 0. Recordable optical discs, such as DVD RW, write data to the disc by darkening and lightening areas of the disc to achieve the same result. As an aside, note that Blu-Ray discs use lasers of a different frequency than CDs and DVDs which allows them to focus on a smaller area of the disc; this is why they can store more data than the other optical discs.

1.3 Representing Numbers in Binary – Basics

Ultimately, we want to use binary values to represent all types of information. Foremost among these are numbers and alphanumeric characters. In this section, we'll introduce the basics of binary numbers. In the following sections we'll look at more complex numeric formats and formats for characters.

1.3.1 How Binary Numbers Work

To explain how binary numbers work, first let's look at how decimal numbers work, since the general method is similar. The decimal number system has ten digits, 0 to 9, but it wouldn't be very useful if it could not represent values greater than nine. To do this, we use multiple digits in a number. The value of each digit is based on both its value as a digit and its place in the number. Consider, for example, the number 135.79. The 1 is in the hundreds place, so its value is 1 × 100 = 100. Similarly, the 3 has a value of 3 × 10 = 30 because it is in the tens place, and so on for the remaining digits. We could expand this as

$$\begin{aligned} 135.79 &= (1 \times 100) + (3 \times 10) + (5 \times 1) + (7 \times 0.1) + (9 \times 0.01) \\ &= (1 \times 10^2) + (3 \times 10^1) + (5 \times 10^0) + (7 \times 10^{-1}) + (9 \times 10^{-2}) \end{aligned}$$

Binary numbers are created in the same way, except we use powers of 2 instead of powers of 10 since binary is base 2. For example, the binary number 1011.01 can be expressed as follows.

$$1011.01 = (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2})$$
$$= (1 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1) + (0 \times ½) + (1 \times ¼)$$
$$= 11\ ¼$$

You can include more bits to represent larger numbers or numbers with larger denominators.

1.3.2 Octal and Hexadecimal Notations

Properly designed digital circuits can deal with large binary numbers fairly easily. For humans, however, long strings of 0s and 1s can become tedious and error-prone. Consider, for instance, the decimal number 927.625. We would express this in binary as 1110011111.101. To make it simpler for people to understand these numbers, we often use notations that combine several bits into one symbol. It is fairly easy to convert binary notation to another notation that is an exact power of 2, such as base 8 (octal) or base 16 (hexadecimal). It is much more complex to convert from binary to another base, such as base 5.

First, let's look at octal notation. This is base 8. It has eight symbols, 0 to 7. Since $8 = 2^3$, we can combine groups of three binary bits into single octal digits. The key is to start at the right place; that place is the radix point. (Think of this as the decimal point, but in any base/radix.) Going to the left, divide the binary bits into groups of three bits. If the final group has less than three bits, add leading zeroes to complete a group of three bits. Now do the same with the bits to the right of the radix point that represent the fractional portion of the number, grouping the bits into threes going right and adding trailing zeroes if needed to complete the final group. Figure 1.4 shows how we do this for our example value.

927.625 = 1110011111.101 = 001 110 011 111.101
= 1 6 3 7 . 5

Figure 1.4: Converting 927.625 from binary to octal. Zeroes added to complete groups of three bits are shown in red.

WATCH ANIMATED FIGURE 1.4

Hexadecimal is base 16. It has 16 symbols. We can use the decimal symbols 0 to 9 for ten of these symbols, but we still need six symbols for the values corresponding to 10 to 15. Hexadecimal uses the letters A to F for these values, with A=10, B=11, C=12, D=13, E=14, and F=15.

With that out of the way, we can use the same conversion process we used for octal, with one exception. Since $16=2^4$, we divide our binary number into groups of four bits instead of three. Figure 1.5 shows the conversion process for our example value.

927.625 = 1110011111.101 = 0011 1001 1111.1010
= 3 9 F . A

Figure 1.5: Converting 927.625 from binary to hexadecimal. Zeroes added to complete groups of four bits are shown in red.

WATCH ANIMATED FIGURE 1.5

1.3.3 Converting from Decimal to Binary

There's one step I didn't cover explicitly yet: How did I get from 927.625 to 1110011111.101? You may have a calculator app that will do this automatically for you. If so, great, but put it away for a few minutes while we introduce the underlying process. We're actually going to have two separate processes, one for the integer part of the number and one for the fractional part.

First, let's look at the integer part. We divide 927 ÷ 2 to get a quotient of 463 and a remainder of 1. What we're really doing is expressing 927 as 463 × 2 + 1. Next, we divide 463 by 2 to get 463 = 231 × 2 + 1, or

$$927 = 463 \times 2 + 1 = (231 \times 2 + 1) \times 2 + 1 = 231 \times 4 + 1 \times 2 + 1$$

We continue this process until we're down to a final value of 0. The complete process for the integer portion is shown in Figure 1.6.

$927 \div 2 = 463$ R 1	$927 = 463\times2^1+1\times2^0$
$463 \div 2 = 231$ R 1	$927 = (231\times2^1+1)\times2^1+1\times2^0=231\times2^2+1\times2^1+1\times2^0$
$231 \div 2 = 115$ R 1	$927 = (115\times2^1+1)\times2^2+1\times2^1+1\times2^0=115\times2^3+1\times2^2+1\times2^1+1\times2^0$
$115 \div 2 = 57$ R 1	$927 = (57\times2^1+1)\times2^3+1\times2^2+1\times2^1+1\times2^0=57\times2^4+1\times2^3+1\times2^2+1\times2^1+1\times2^0$
$57 \div 2 = 28$ R 1	$927 = (28\times2^1+1)\times2^4+1\times2^3+1\times2^2+1\times2^1+1\times2^0=28\times2^5+1\times2^4+1\times2^3+1\times2^2+1\times2^1+1\times2^0$
$28 \div 2 = 14$ R 0	$927 = (14\times2^1+0)\times2^5+1\times2^4+1\times2^3+1\times2^2+1\times2^1+1\times2^0$
	$= 14\times2^6+0\times2^5+1\times2^4+1\times2^3+1\times2^2+1\times2^1+1\times2^0$
$14 \div 2 = 7$ R 0	$927 = (7\times2^1+0)\times2^6+0\times2^5+1\times2^4+1\times2^3+1\times2^2+1\times2^1+1\times2^0$
	$= 7\times2^7+0\times2^6+0\times2^5+1\times2^4+1\times2^3+1\times2^2+1\times2^1+1\times2^0$
$7 \div 2 = 3$ R 1	$927 = (3\times2^1+1)\times2^7+0\times2^6+0\times2^5+1\times2^4+1\times2^3+1\times2^2+1\times2^1+1\times2^0$
	$= 3\times2^8+1\times2^7+0\times2^6+0\times2^5+1\times2^4+1\times2^3+1\times2^2+1\times2^1+1\times2^0$
$3 \div 2 = 1$ R 1	$927 = (1\times2^1+1)\times2^8+1\times2^7+0\times2^6+0\times2^5+1\times2^4+1\times2^3+1\times2^2+1\times2^1+1\times2^0$
	$= 1\times2^9+1\times2^8+1\times2^7+0\times2^6+0\times2^5+1\times2^4+1\times2^3+1\times2^2+1\times2^1+1\times2^0$
$1 \div 2 = 0$ R 1	$927 = (0\times2^1+1)\times2^9+1\times2^8+1\times2^7+0\times2^6+0\times2^5+1\times2^4+1\times2^3+1\times2^2+1\times2^1+1\times2^0$
	$= 1\times2^9+1\times2^8+1\times2^7+0\times2^6+0\times2^5+1\times2^4+1\times2^3+1\times2^2+1\times2^1+1\times2^0$

Figure 1.6: Conversion of decimal integer to binary

WATCH ANIMATED FIGURE 1.6

The remainders are combined to give us the binary representation of the integer. The first remainder is the least significant bit and the last remainder is the most significant bit. For our value of 927, this is 1110011111.

We calculate the fractional portion using a different but analogous approach. Instead of dividing by 2, we multiply by 2. The bits of the fraction in binary are the integer portion of each product. For our fraction of 0.625, we first multiply by 2 to get 0.625 × 2 =1.25, or 0.625 = 1.25 × 2^{-1}. We keep the 1 and repeat the process on the remaining fractional value, 0.25, and continue until the remaining fraction is 0. (Sometimes a fraction would never end. For example, 1/3 = .010101… in binary. Normally we would decide how many bits to include in the fraction and round the result to that many bits.) Figure 1.7 shows this process for our fractional value.

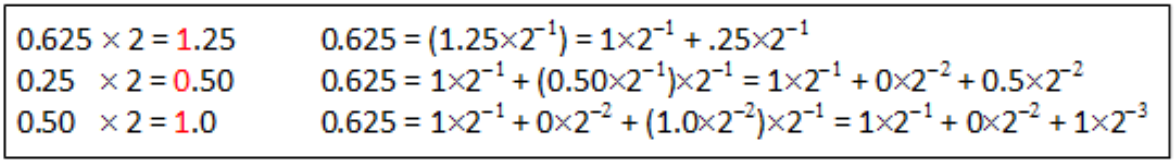


Figure 1.7: Conversion of decimal fraction to binary

WATCH ANIMATED FIGURE 1.7

Here, we combine the bits of the integer part of the product to generate the binary fraction. The first 1, generated by 0.625 × 2, is the most significant bit of the fraction. Unlike the integer procedure, which generated bits from least significant to most significant, the fraction procedure generates the bits in the opposite order, from most significant to least significant. It may help to describe this by saying that both procedures start by generating the bit closest to the radix point and move further away each iteration.

The final step is trivial. We simply concatenate the integer and fraction to produce the final result. For our example, this yields 927.625 = 1110011111.101.

As you might have guessed, you can use this process to convert a number from decimal (base ten) to any arbitrary base. Instead of dividing and multiplying by 2, we divide and multiply by the base we are converting to. Figure 1.8 shows this conversion from 927.625 in decimal to 12202.303 in base 5. Note that the fraction never terminates (it repeats as .303030…), so I chose to truncate it after three digits.

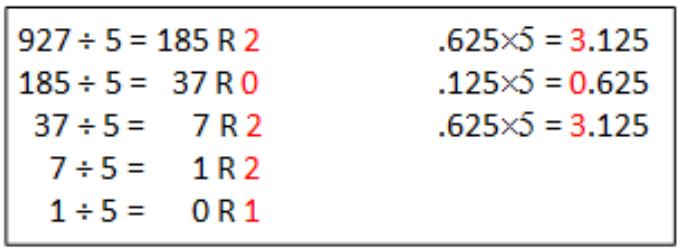


Figure 1.8: Conversion of decimal number to base 5

WATCH ANIMATED FIGURE 1.8

1.4 Numeric Formats

Many digital systems, mostly computers, use several different numeric formats. In this section, we'll look at some of these formats. The first type, unsigned formats, is typically used for integers. In spite of its name, one unsigned format allows for both positive and negative numbers. We'll explain how this works shortly. Next, we'll examine signed formats, which differ from unsigned formats by as little as a single bit. Finally, we'll introduce floating point formats and the IEEE 754 floating point standard that is supported by virtually all computer systems.

1.4.1 Unsigned Formats

There are two unsigned formats we discuss in this section. The first, traditional unsigned numbers, is simply the integer portion of the binary format we introduced in the previous section. Here, an *n*-bit number can take on any integer value from 0 to 2^n-1. Figure 1.9 (a) shows the values for all 4-bit numbers in this format.

The second unsigned format is two's complement. Before we discuss this format, let's take a minute to discuss what complements are and how they are formed.

A complement is the opposite of something. For binary numbers, there are two different types of complements: one's complement and two's complement. To obtain the one's complement of a binary number, you can do one of two things:

- Change all the 1s to 0s and all the 0s to 1s (inverting bits), or
- Subtract the number from a number of equal length that is all 1s.

Figure 1.10 shows both processes for the binary value 0101, or 5 in decimal.

Binary	Unsigned traditional
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

(a)

Binary	Unsigned two's complement
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

(b)

Figure 1.9: Values for all 4-bit binary numbers in (a) traditional and (b) two's complement unsigned notations.

```
0  1  0  1
↓  ↓  ↓  ↓
1  0  1  0
```

(a)

```
  1  1  1  1
- 0  1  0  1
  1  0  1  0
```

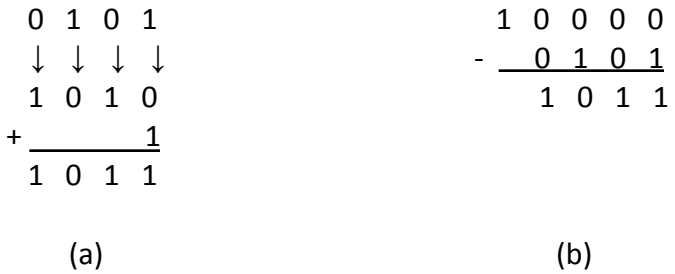
(b)

Figure 1.10: Creating one's complement by (a) inverting bits, and (b) subtraction.

The two's complement is simply the one's complement plus 1. Just as was the case for the one's complement, there are two ways we can produce the two's complement of a binary number:

- Create the one's complement and add 1, or
- Subtract the number from a number that has a leading 1 and as many zeroes as there are bits in the original number. (For example, we would subtract a 4-bit number from 10000, a leading 1 and four zeroes.)

Figure 1.11 shows both methods for our previous value, 0101.



(a) (b)

Figure 1.11: Creating two's complement by (a) creating one's complement and adding 1, and (b) subtraction.

Now, the reason we did all this is because *the two's complement of a number represents the negative of that number in two's complement notation*. For two's complement numbers with *n* bits, the values can range from -2^{n-1} to $+2^{n-1}-1$. When the most significant bit is 1, the number is negative. When it is 0, the number is positive or zero. Figure 1.9 (b) shows the values for all 4-bit numbers in two's-complement format.

Comparing these two unsigned formats, we see that both use the same representation when the leading bit is 0. The difference in value occurs when the leading bit is 1.

Here's one final question. Why is the two's complement notation called unsigned when the values can be either positive or negative? There's a good reason for this, which we'll see in the next subsection.

1.4.2 Signed Notations

Signed notations are similar to unsigned notations, but with one important difference. *Signed notations have a separate bit that is used for the sign*. The remaining bits give the magnitude of the number. The sign bit is set to 1 for negative numbers and 0 for positive numbers and zero.

There are two commonly-used signed notations, both based on the unsigned notations we just introduced. **Signed-magnitude** notation is based on the traditional unsigned notation. In this notation, the magnitude is expressed like a traditional unsigned number. For example, +5 would be represented as a 0 for the sign bit and 0101 for the magnitude. On the other hand, -5 would be represented by a 1 in the sign bit and also 0101 for the magnitude. Figure 1.12 (a) shows some selected values for numbers with a sign bit and four bits for the magnitude.

Sign	Magnitude	Value
0	0000	+0
0	0001	+1
0	1111	+15
1	0001	-1
1	1111	-15

(a)

Sign	Magnitude	Value
0	0000	+0
0	0001	+1
0	1111	+15
1	1111	-1
1	0001	-15

(b)

Figure 1.12: Values for selected binary numbers in (a) signed-magnitude, and (b) signed-two's complement notations.

There is also a format based on the two's complement format called signed-two's complement. It works the same as signed-magnitude format, except the magnitude is stored in two's complement format. Selected values are shown in Figure 1.12 (b). As with the unsigned formats, positive numbers and zero are the same in both formats, but negative numbers are not.

1.4.3 Floating Point Numbers

A floating point number is similar, but not identical to a number expressed in scientific notation. Consider the number +927.625. In scientific notation, we would express this as $+9.27625 \times 10^2$. There are three components to this number: the sign (+), the mantissa (9.27625) and the exponent (2). Of particular note, the mantissa must be at least 1 and less than 10, unless the number is zero.

Floating point works in a similar way, with one difference. Instead of expressing 927.625 as $+9.27625 \times 10^2$, we express it as $+0.927625 \times 10^3$. The mantissa is replaced by the **significand**, which must be at least 0.1 and less than 1, except for special cases, such as zero. This is a **normalized** format; each number is expressed in one unique way.

Digital systems that use floating point numbers typically follow the **IEEE 754 Floating Point Standard** for binary numbers. This standard changes how the significand is defined. Here, the significand is of the form 1.xxxx; that is, it must be at least 1 and less than 2 (10 in binary), again except for zero and other special values. This is much like the mantissa in scientific notation. Since these numbers always have this leading 1, the digital system does not need to include it when it stores the floating point representation; it already knows it is there.

There are two formats specified by IEEE 754: the first is called **single precision**. As shown in Figure 1.13 (a), the single precision format has 32 bits: 1 bit for the sign; 23 bits for the significand; and 8 bits for the exponent. We want to have both positive and negative exponents, so this standard uses a **bias** of 128. That is, we store the exponent 0 as 128 (10000000 in binary), exponent 1 is 129 (10000001), -1 is 127 (01111111), and so on for exponents ranging from -128 (00000000) to +127 (11111111).

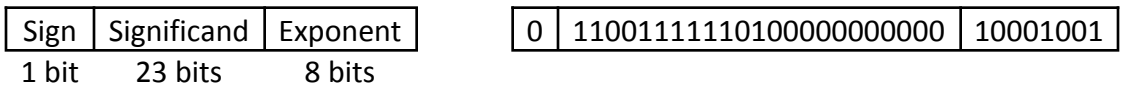


Figure 1.13: IEEE 754 (a) single precision format, and (b) representations of 927.625.

Earlier, we found that +927.625 is equal to +1110011111.101 in binary. For the single precision format, we can express this as $1.110011111101 \times 2^9$. Just as in the signed notations (and floating point is also a signed notation), the sign bit is 0 since we have a positive number. For the significand, we remove the 1 to the left of the radix point and store the remaining 12 bits. Since the significand has 23 bits, we fill in the remaining 11 rightmost bits with zeroes. The exponent is 0, but with a bias of 128 it is stored as 9+128, or 137, or 10001001 in binary. The complete representation is shown in Figure 1.13 (b).

The second IEEE 754 format is called **double precision**. It is similar in structure to single precision, except it has 64 bits, double the number of bits in single precision and hence its name. This format has one sign bit, 52 bits for the significand (again, we don't store the implicit leading 1 for everything except special cases), and 11 bits for the exponent. Double precision uses a bias of 1024 for the exponent. Figure 1.14 shows the double precision format and how +927.625 is stored in this format.

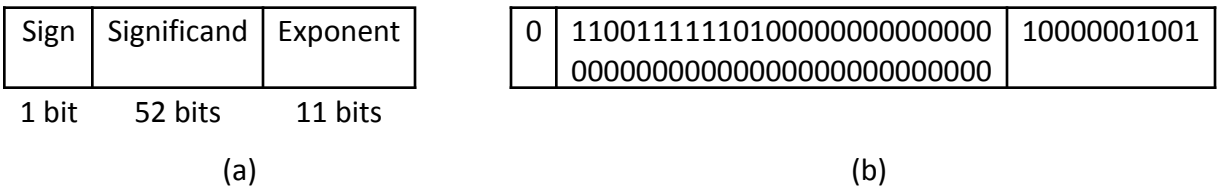


Figure 1.14: IEEE 754 (a) double precision format, and (b) representations of 927.625.

As mentioned already, zero is a special number that needs its own representation. (Spoiler: the significand and exponent are set to all zeroes in both formats.) There are other special numbers as well: ±∞, NaN (not a number), and denormalized numbers. They're beyond the scope of this book and we won't be covering them here. I'll just refer you to my previous book *Computer systems organization & architecture* for more information about these special numbers and how they complicate the digital circuits needed to work with floating point numbers.

1.5 Representing Nonnumeric Data

Not all data is numeric; this sentence is proof of that statement. Much of the data used by digital systems is not numeric, and much of that data consists of letters and other characters. In this section, we'll briefly look at two popular schemes for encoding characters, ASCII and Unicode.

1.5.1 ASCII

As mainframe computers began to find more uses in industry, and more companies designed and manufactured them, it became necessary to standardize some aspects such as the codes used to represent characters. One standard, initiated by IBM engineer Bob Bemer, led to the development of **ASCII**, the American Standard Code for Information Interchange. This standard dates back to the early 1960s, when it was developed by the American Standards Association. When its use was mandated for all systems used by the United States federal government, its success was assured.

ASCII specifies 7-bit codes for 2^7=128 characters. Of these 128 characters, 95 are characters that can be printed, including uppercase and lowercase letters, decimal digits, and punctuation marks. The other 33 are control characters, which controlled the teletype machines of that era, performing functions like line feed to make the teletype machine go to the start of the next line. (Computing history for you, but early computing education for me.)

Figure 1.15 shows the ASCII characters and their binary codes. The character names in italics are the nonprinting control characters.

ASCII solved a lot of problems for computer developers, but it also had some significant drawbacks. It did a great job if you wanted to encode text written in English, but it lacked characters used in other languages, such as ñ in Spanish. If you wanted to write in a language that uses a different alphabet, such as Russian, or a language that uses pictorial characters, such as Japanese or Chinese, ASCII wasn't going to work for you. Several extensions to ASCII were developed. They typically extended the ASCII code to eight bits, which allowed up to 2^8=256 characters. In these extended versions, the first 128 characters were usually the original ASCII characters, and the other 128 were the extended characters. This was a stopgap; clearly a more robust system was needed. One such system is **UNICODE**, described next.

Code	Character
00000000	*NUL*
00000001	*SOH*
00000010	*STX*
00000011	*ETX*
00000100	*EOT*
00000101	*ENQ*
00000110	*ACK*
00000111	*BEL*
00001000	*BS*
00001001	*HT*
00001010	*LF*
00001011	*VT*
00001100	*FF*
00001101	*CR*
00001110	*SO*
00001111	*SI*
00010000	*DLE*
00010001	*DC1*
00010010	*DC2*
00010011	*DC3*
00010100	*DC4*
00010101	*NAK*
00010110	*SYN*
00010111	*ETB*
00011000	*CAN*
00011001	*EM*
00011010	*SUB*
00011011	*ESC*
00011100	*FS*
00011101	*GS*
00011110	*RS*
00011111	*US*

Code	Character
00100000	space
00100001	!
00100010	"
00100011	#
00100100	$
00100101	%
00100110	&
00100111	'
00101000	(
00101001	)
00101010	*
00101011	+
00101100	,
00101101	-
00101110	.
00101111	/
00110000	0
00110001	1
00110010	2
00110011	3
00110100	4
00110101	5
00110110	6
00110111	7
00111000	8
00111001	9
00111010	:
00111011	;
00111100	<
00111101	=
00111110	>
00111111	?

Code	Character
01000000	@
01000001	A
01000010	B
01000011	C
01000100	D
01000101	E
01000110	F
01000111	G
01001000	H
01001001	I
01001010	J
01001011	K
01001100	L
01001101	M
01001110	N
01001111	O
01010000	P
01010001	Q
01010010	R
01010011	S
01010100	T
01010101	U
01010110	V
01010111	W
01011000	X
01011001	Y
01011010	Z
01011011	[
01011100	\
01011101	]
01011110	^
01011111	_

Code	Character
01100000	`
01100001	a
01100010	b
01100011	c
01100100	d
01100101	e
01100110	f
01100111	g
01101000	h
01101001	i
01101010	j
01101011	k
01101100	l
01101101	m
01101110	n
01101111	o
01110000	p
01110001	q
01110010	r
01110011	s
01110100	t
01110101	u
01110110	v
01110111	w
01111000	x
01111001	y
01111010	z
01111011	{
01111100	|
01111101	}
01111110	~
01111111	DEL

Figure 1.15 ASCII Table

1.5.2 UNICODE

As computing expanded its scope and breadth, English was joined by many other languages used for documents stored in digital form. Some of these languages used the same characters as English, sometimes with their own additions, and others used completely different characters. It became clear that having a new encoding that encompassed all these characters, or at least as many as feasible, was desirable. Several systems were developed; the most commonly used is UNICODE.

UNICODE is actually a set of several encoding standards. The most popular are UTF-8 and UTF-16. UTF-8 was designed to maintain compatibility with ASCII. UTF-16 was designed to address the inclusion of non-English characters. UTF-16 supports over one million characters, but we'll focus on the most basic representation, which uses 16 bits to specify one of 2^{16}=65,536 different codes. Most of these codes specify one specific character, though some are used in combination with a second 16-bit code to specify a character. In addition to English-language characters, UTF-16 also encodes Cyrillic, Arabic, Hebrew, Chinese, Japanese, Korean, and many other characters. I won't list all the characters here. See the UNICODE Consortium website or your favorite online search engine for these character encodings.

1.6 Gray Code

Counting in binary is fairly straightforward. Just as we count from 0→1→2→3→4→5→6→7 in decimal, we can count from 000→001→010→011→100→101→110→111 in binary. If all three bits in the binary count change at exactly the same time, this sequence works fine. There are ways to make this happen, often using additional hardware to synchronize these transitions. Without this additional hardware, however, we could run into a problem.

Consider the transition from 011 to 100. If the first bit changes faster than the other two, we could go to 111 instead of 100. Similarly, a faster second bit could bring us to 001, and a faster third bit could send us to 010.

Another way to avoid this issue is to develop sequences in which only one bit changes in each transition. Although not quite as useful when counting, this is very useful when developing state machines (introduced later in this book). It also serves as the basis for Karnaugh maps, which are used to simplify functions and their circuits, and are introduced in the next chapter. This is the reason we're introducing this topic now.

The **Gray code**, named after Bell Labs physicist Frank Gray, is a reflected binary code. To construct this code, we start with a 1-bit sequence of 0 and 1, shown in Figure 1.16 (a). To extend this to a 2-bit sequence, we first reflect the code, as shown in Figure 1.16 (b). We then add a leading 0 to each entry above the line and a leading 1 to every value below the line to give us the complete 2-bit code; see Figure 1.16 (c). (If we were to go through this code in sequence, the last entry goes back to the first entry.) We could repeat this process to generate the 3-bit Gray code shown in Figure 1.16 (d), and we could extend this sequence to produce a Gray code of any length. Here, the 3-bit Gray code follows the sequence 000→001→011→010→110→111→101→100.

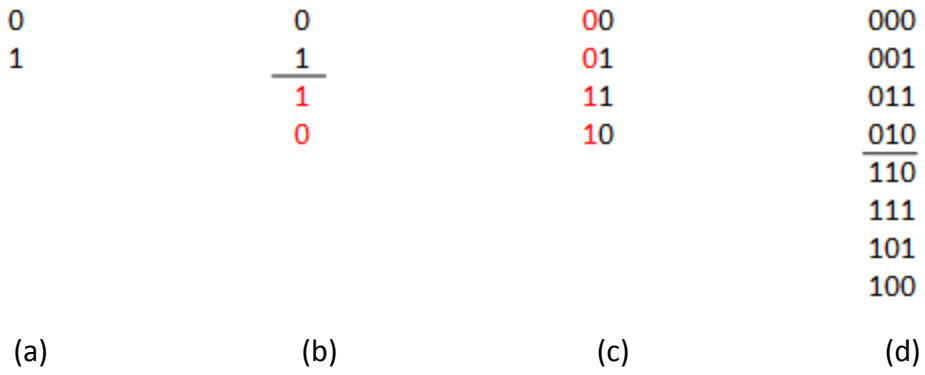


Figure 1.16: Gray codes: (a) 1-bit Gray code; (b) Reflecting the 1-bit code; (c) Adding leading bit to produce the 2-bit Gray code; (d) 3-bit Gray code

WATCH ANIMATED FIG. 1.16

1.7 Summary

Digital systems are ubiquitous. Unlike analog systems, which use continuous values, digital systems have discrete values with gaps between the values. By making the gaps small enough, a digital system can approximate an analog system.

The binary number system is the basis for digital systems. The binary system has only two values, which we call one and zero. The components used to construct digital systems are typically built using transistors or other semiconductor components. Voltage levels represent these two logic values. Some storage devices use magnetization or reflectivity. Ultimately, these are converted to their corresponding voltage values.

Just as with decimal numbers, binary numbers can be extended beyond 0 and 1 by using more binary digits, or bits. Long strings of bits can be difficult to read (for humans), so we often use octal or hexadecimal notation to represent binary numbers. Digital systems, however, always work with binary values. The processes to convert numbers between decimal, binary, octal, and hexadecimal are straightforward.

In addition to traditional binary numbers, there are other ways to represent numbers in binary. These include unsigned and signed notations, as well as floating point. There are also encodings for nonnumeric data, such as characters. ASCII and UNICODE are two commonly used methods. UNICODE specifies several different encodings, of which UTF-8 and UTF-16 are used most frequently.

Gray codes are sequences of binary values in which adjacent values have only one bit that is different. It is useful in designing systems to avoid issues when more than one bit changes its value simultaneously. It is the basis for Karnaugh maps, a method used to simplify binary functions, and hence the digital logic we use to implement these functions.

In the next chapter, we will introduce Boolean algebra. This is the mathematics behind binary numbers and the basis for the digital logic we will use to design digital systems throughout this book and beyond.

Bibliography

Bemer, R. W. (1978, May). Inside ASCII, Part I. *Interface Age, 3*(5), 96–102.

Benchley, R. (1920, February). The most popular book of the month: An extremely literary review of the latest edition of the New York City Telephone Directory. *Vanity Fair, (13)*6, 69.

Carpinelli, J. D. (2001). *Computer systems organization & architecture*. Addison-Wesley.

Hayes, J. P. (1993). *Introduction to Digital Logic Design.* Addison-Wesley.

Johnson, H. (1998). *ECL and PECL.* https://www.sigcon.com/Pubs/news/2_22.htm

Johnson, K., & Gubser, B. (2009, March). *LVPECL, PECL, ECL Logic and Termination*. Mouser Electronics. https://www.mouser.com/pdfDocs/peclclocksandtermination.pdf.

Mano, M. M., & Ciletti, M. (2017). *Digital Design: With an Introduction to the Verilog HDL, VHDL, and SystemVerilog* (6th ed.). Pearson.

Mano, M. M., Kime, C., & Martin, T. (2015). *Logic & Computer Design Fundamentals* (5th ed.). Pearson.

Nelson, V., Nagle, H., Carroll, B., & Irwin, D. (1995). *Digital Logic Circuit Analysis and Design*. Pearson.

Roth, J. C. H., & Kinney, L. L. (2013). *Fundamentals of Logic Design* (7th ed.). Cengage Learning.

Sengupta, A. (2021, August 11). *How Is Information Stored In DVDs?* Science ABC. https://www.scienceabc.com/innovation/how-is-information-stored-in-dvds.html

Sheldon, R. (2020, February 18). Storage 101: Understanding the Hard-Disk Drive. *Simple Talk*. https://www.red-gate.com/simple-talk/databases/sql-server/database-administration-sql-server/storage-101-understanding-the-hard-disk-drive/

Texas Instruments. (2017.) *Logic Guide.* Texas Instruments. https://www.ti.com/lit/sg/sdyu001ab/sdyu001ab.pdf

Unicode Consortium. (1981-2023). *Unicode*. https://home.unicode.org/

Wakerly, J. F. (2018). *Digital design: Principles and practices* (5th ed. with Verilog.). Pearson Education.

Exercises

1. List three analog and three digital systems or devices found in a typical home.
2. Besides those listed at the end of Section 1.1, what are some other advantages that digital systems have over analog systems? What are some advantages that analog systems have over digital systems?
3. Convert 631 from:
    a. Decimal to binary
    b. Binary to octal
    c. Binary to hexadecimal
4. Convert 185.375 from:
    a. Decimal to binary
    b. Binary to octal
    c. Binary to hexadecimal
5. Convert 10111101 from binary to:
    a. Decimal
    b. Octal
    c. Hexadecimal
6. Convert 1011011010.01 from binary to:
    a. Decimal
    b. Octal
    c. Hexadecimal
7. Convert 1537 directly (i.e. without first converting to binary) from decimal to:
    a. Octal
    b. Hexadecimal
8. Convert 753 from octal to hexadecimal.
9. Convert A7F from hexadecimal to octal.
10. Show the decimal number 13 in the following formats:
    a. Traditional unsigned (8 bits)
    b. Unsigned two's complement (8 bits)
    c. Signed-magnitude (8-bit magnitude)
    d. Signed two's complement (8-bit magnitude)

11. Show the decimal number -13 in the following formats:
    a. Unsigned two's complement (8 bits)
    b. Signed-magnitude (8-bit magnitude)
    c. Signed two's complement (8-bit magnitude)
12. A digital system stores floating point numbers using an 8-bit significand and a 4-bit exponent with a bias of 8. Show how it stores the binary number 101.11101.
13. A digital system stores floating point numbers using an 8-bit significand and a 4-bit exponent with a bias of 8. Show how it stores the binary number .00101101.
14. Show how the binary value 101.11101 is stored in IEEE 754:
    a. Single precision format
    b. Double precision format
15. Show how the binary value .00101101 is stored in IEEE 754:
    a. Single precision format
    b. Double precision format
16. Show how your name is encoded in:
    a. ASCII
    b. UTF-8
    c. UTF-16
17. Convert the following ASCII codes to the characters they represent:

    1000011 1101111 1101101 1110000 1110101 1110100 1100101 1110010
18. Convert the following UTF-8 codes to the characters they represent:

    01000100 01101001 01100111 01101011 01110100 0110001 01101100
19. Convert the following UTF-16 codes (shown in hexadecimal format) to the characters they represent
    a. 0043 0069 0072 0063 0075 0069 0074
20. Convert the codes shown in the previous three problems to each of the other formats.
    a. Show the 4-bit Gray code.

Chapter 2

Boolean Algebra

Chapter 2: Boolean Algebra

In the previous chapter, we introduced binary numbers, the base 2 number system. Consisting solely of two digits, 0 and 1, we examined them strictly as they represent numeric values. There is, however, another way to work with binary values, using them to represent logical values rather than numeric values. Here, 1 and 0 can be thought of as TRUE and FALSE. The digital circuits we will introduce and design throughout this book are based on this logical interpretation of binary values.

There is an entire branch of mathematics that focuses on binary values and the logical operations we can perform on them. It is called **Boolean algebra**, named after the English mathematician George Boole. Boolean algebra is the theoretical foundation upon which digital logic design is built. In this chapter, we'll introduce the basics of Boolean algebra and how we can use it to model real-world situations.

First, we introduce the fundamental operations of Boolean algebra and the basic postulates of this algebra. Then we will examine how to combine these operations to specify **functions**, which we will use to model the behavior of real-world actions or the desired behavior of a digital system. Truth tables are a convenient mechanism used to specify the function values for all possible Boolean input values. We'll also look at how to express functions based on truth table values, minterms, and maxterms.

Finally, we'll look at ways to analyze and simplify functions. Karnaugh maps are a graphical method used for this purpose. They make use of the Gray code we introduced in the previous chapter. We'll show why we need to use this code during our discussion of the topic. Karnaugh maps work well when you have up to four inputs. Beyond that, due to the human inability to see beyond three dimensions at any given time, they aren't very useful. We'll introduce another method, the Quine-McCluskey algorithm, which can be used to simplify functions with larger numbers of inputs. Finally, we'll look at incompletely specified functions. For these functions, sometimes we don't care what the output is for certain combinations of input values. We'll look at how to take advantage of this to simplify functions as much as possible.

2.1 Boolean Algebra

Before we can study digital logic design, we need to introduce the fundamentals of Boolean algebra. The most fundamental digital logic components implement basic Boolean operations, and more complex Boolean functions translate directly to digital logic designs.

First, I'll start by noting that, mathematically speaking, there are really many different Boolean algebras, an infinite number of them to be precise. Each of these Boolean algebras has a different number of distinct elements (think of these as digits). Each has 2^n digits, where n is an integer and $n \geq 1$. Fortunately for us, we are concerned only with the Boolean algebra for which $n=1$, which has $2^1=2$ digits, which we denote as 0 and 1. This is also referred to as a **switching algebra**.

We'll introduce Boolean algebra as follows. In the next subsection, we present the basic operations of Boolean algebra. We'll also introduce **truth tables**, a mechanism we will use

throughout this chapter and the rest of this book. The following subsection examines some properties of this Boolean algebra and shows how we know they are correct.

2.1.1 Boolean Algebra Fundamentals

The fundamental components of any algebra are its symbol set and its operations. We have already seen the symbol set for this Boolean algebra, {0, 1}. Every Boolean variable takes on one of these two values.

With that out of the way, let's look at the operations that comprise this Boolean algebra. We'll start with the **AND** operation. The AND function takes two or more Boolean input values and generates a single output. The function output is 1 if and only if every input value is 1. If one or more inputs is 0, the function output is 0. Symbolically, we represent the AND function using the dot ($\cdot$) or $\wedge$ symbol, or by simply concatenating the inputs that are ANDed by the function. For example, if we want to AND Boolean values a and b, we could express this function as $a \cdot b$, $a \wedge b$, or simply ab.

The second function is the OR function. This function also has two or more Boolean inputs, but its output is 1 if *any* of its inputs is 1. It is only 0 when all its inputs are 0. It is denoted using the + or $\vee$ symbol; the OR of a and b is expressed as $a + b$ or $a \vee b$.

Finally, there is the NOT function. Unlike the AND and OR functions, this function takes in only one input. Its output is the opposite of the input value. That is, when the input is 0, the output is 1, and when the input is 1 the output is 0. There are several ways to represent the NOT of an input, say a, including a', $\sim a$, $!a$, $/a$, and $\overline{a}$.

There are several other functions that can be created using these functions, including the XOR (exclusive OR), NAND, NOR, and XNOR functions. They are not an official part of the specification of this Boolean algebra, but they are commonly used in digital logic design. Digital component designers have developed integrated circuit chips that implement the fundamental AND, OR, and NOT functions, as well as the XOR, NAND, NOR, and XNOR functions. We'll see more about this in Chapter 3.

2.1.2 Truth Tables

A truth table is a convenient mechanism to specify the output of any Boolean function, from the most fundamental to the most complex, for all possible combinations of input values. To illustrate how this works, consider the truth table for the AND function, shown in Figure 2.1 (a). Since there are two inputs (a and b), there are $2^2 = 4$ possible combinations of input values: a=0 and b=0; a=0 and b=1; a=1 and b=0; and a=1 and b=1. (If we have three inputs, we would have $2^3 = 8$ possible combinations of input values.) In the table, we list these input values on the left side of the vertical line, one combination of inputs per row. On the right side, we list the value of the function for the input values in each row. Here, the function is only 1 in the last row, where all inputs are 1. Figures 2.1 (b) and 2.1 (c) show the truth tables for the OR and NOT functions.

a	*b*	AND
0	0	0
0	1	0
1	0	0
1	1	1

(a)

a	*b*	OR
0	0	0
0	1	1
1	0	1
1	1	1

(b)

a	NOT
0	1
1	0

(c)

Figure 2.1: Truth tables for the functions (a) AND, (b) OR, and (c) NOT.

Truth tables are a nice way to show how these fundamental functions work, but they are most useful when used to specify the behavior of more complex functions. We'll see how this works later in this chapter.

2.1.3 Boolean Algebra Properties

Mathematically speaking, an algebra is a theory encompassing a set of elements and a set of operations performed on those elements. For our Boolean algebra, we have seen that the set of elements is {0, 1} and the set of operations is {AND, OR, NOT}. An algebra must also have a set of **axioms** (also called **postulates**) that are essentially the most basic rules or properties of the algebra.

American mathematician **Edward Huntington** proposed a set of axioms for our Boolean algebra, collectively called **Huntington's postulates**. In addition, other axioms have been proposed that are true for this Boolean algebra, but are not an official component of the definition of the algebra. Some of Huntington's postulates are solely for mathematical completeness. Others can directly impact the design of digital logic circuits.

The remainder of this section will introduce these postulates and show how they are true for this Boolean algebra. We'll start with Huntington's postulates.

Closure

Closure means that for any fundamental function, if all the function inputs are in the algebra's set of elements, then the function's outputs are also in this set of elements. We can see this is true by looking at the truth tables in Figure 2.1. For each truth table, all possible combinations of input values are listed, and each function output is always equal to either 0 or 1.

Identity Elements

For each function with more than one input, there is an element that causes the output of the function to be the same as the other input. Note that the identity element does not have to be the same for each function. Different functions can have different identity elements.

For our Boolean algebra, we have two functions that need identity elements, AND and OR. We'll look at them individually, and once again we'll use truth tables to justify this postulate. Starting with the AND function, it is straightforward to show that 1 is the identity element for this function. Figure 2.2 (a) shows the truth table for this function, with a as the input and $a \cdot 1$ as the function output. Clearly, $a \cdot 1 = a$.

a	$a \cdot 1$
0	$0 \cdot 1 = 0$
1	$1 \cdot 1 = 1$

(a)

a	$a + 0$
0	$0 + 0 = 0$
1	$1 + 0 = 1$

(b)

Figure 2.2: Truth tables showing the identity elements for the (a) AND, and (b) OR functions.

Using this same approach, we can see that 0 is the identity element for the OR operation. This is shown in Figure 2.2 (b).

Commutativity

This is a fancy way of saying that the order of the inputs does not matter. For this algebra, this means that $a \cdot b = b \cdot a$, and $a + b = b + a$. As before, we use truth tables to verify that this Boolean algebra has this property, as shown in Figure 2.3.

a	b	$a \cdot b$	$b \cdot a$
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

(a)

a	b	$a + b$	$b + a$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	1

(b)

Figure 2.3: Truth tables verifying the commutativity of the (a) AND, and (b) OR functions.

WATCH ANIMATED FIG 2.3.a WATCH ANIMATED FIG 2.3.b

Note that this property does not necessarily apply to all operations in all algebras. In traditional algebra, basic arithmetic, addition and multiplication are commutative but subtraction and division are not. For example 5-3 ≠ 3-5 and 6÷2 ≠ 2÷6.

Distributivity

The distributive property uses both 2-operand functions, with each being distributed over the other. The property for the distribution of the AND function over the OR function can be expressed as

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c)$$

Again using a truth table, we can verify this property is true for the distribution of AND over OR. This table is shown in Figure 2.4.

a	b	c	$b + c$	$a \cdot (b + c)$	$a \cdot b$	$a \cdot c$	$(a \cdot b) + (a \cdot c)$
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

Figure 2.4: Truth table verifying the distributive property of AND over OR.

WATCH ANIMATED FIGURE 2.4

The distribution of OR over AND can be expressed as

$$a + (b \cdot c) = (a + b) \cdot (a + c)$$

The truth table in Figure 2.5 verifies this property.

a	b	c	$b \cdot c$	$a + (b \cdot c)$	$a + b$	$a + c$	$(a + b) \cdot (a + c)$
0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0
0	1	0	0	0	1	0	0
0	1	1	1	1	1	1	1
1	0	0	0	1	1	1	1
1	0	1	0	1	1	1	1
1	1	0	0	1	1	1	1
1	1	1	1	1	1	1	1

Figure 2.5: Truth table verifying the distributive property of OR over AND.

WATCH ANIMATED FIGURE 2.5

Inverse

Each element in the algebra has an inverse such that $a \cdot a' = 0$ and $a + a' = 1$. The truth tables in Figure 2.6 show that this is the case for our Boolean algebra.

a	a'	$a \cdot a'$
0	1	$0 \cdot 1 = 0$
1	0	$1 \cdot 0 = 0$

(a)

a	a'	$a + a'$
0	1	$0 + 1 = 1$
1	0	$1 + 0 = 1$

(b)

Figure 2.6: Truth table verifying the inverse property for the (a) AND, and (b) OR functions.

Distinct Elements

Huntington's postulates specify that the algebra must have at least two distinct elements. This was added just to exclude the trivial case of a Boolean algebra with $2^0 = 1$ element. Our Boolean algebra has two elements, 0 and 1, and thus meets this requirement.

Beyond Huntington's postulates, there are several other axioms that are true for this Boolean algebra. As with Huntington's postulates, some are used widely to minimize functions and the digital logic needed to implement them.

Idempotence

When a function takes one value for both its inputs, the value of the function is equal to the value of the input. Mathematically, this means $a \cdot a = a$, and $a + a = a$ for this algebra. Figure 2.7 shows the truth tables that verify this property.

a	$a \cdot a$
0	$0 \cdot 0 = 0$
1	$1 \cdot 1 = 1$

(a)

a	$a + a$
0	$0 + 0 = 0$
1	$1 + 1 = 1$

(b)

Figure 2.7: Truth table verifying the idempotence property for the (a) AND, and (b) OR functions.

Involution

The inverse of the inverse of an element is the same as the element. You can think of this as a double negative bringing you back to the original value. The truth table in Figure 2.8 verifies that this property is true for this algebra.

a	a'	$(a')'$
0	1	$1' = 0$
1	0	$0' = 1$

Figure 2.8: Truth table verifying the involution property.

WATCH ANIMATED FIGURE 2.8

Absorption

This property has direct use in minimizing functions and digital logic. It has two forms.

$$a + (a \cdot b) = a$$
$$a \cdot (a + b) = a$$

The truth tables in Figure 2.9 verify both forms of this property.

a	b	$a \cdot b$	$a + (a \cdot b)$
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1

(a)

a	b	$a + b$	$a \cdot (a + b)$
0	0	0	0
0	1	1	0
1	0	1	1
1	1	1	1

(b)

Figure 2.9: Truth table verifying both forms of absorption.

WATCH ANIMATED FIGURE 2.9.a WATCH ANIMATED FIGURE 2.9.b

Associativity

When you have only one type of operation occurring in a function, the order in which you perform them does not change the value of the function. Specifically,

$$(a \cdot b) \cdot c = a \cdot (b \cdot c)$$
$$(a + b) + c = a + (b + c)$$

Although Huntington did not include this in his postulates for Boolean algebras, it is indeed true. Figure 2.10 shows the truth table for the AND function and Figure 2.11 shows the truth table for the OR function.

a	b	c	$a \cdot b$	$(a \cdot b) \cdot c$	$b \cdot c$	$a \cdot (b \cdot c)$
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	1	0
1	0	0	0	0	0	0
1	0	1	0	0	0	0
1	1	0	1	0	0	0
1	1	1	1	1	1	1

Figure 2.10: Truth table showing the associative property for the AND function.

WATCH ANIMATED FIGURE 2.10

a	b	c	$a + b$	$(a + b) + c$	$b + c$	$a + (b + c)$
0	0	0	0	0	0	0
0	0	1	0	1	1	1
0	1	0	1	1	1	1
0	1	1	1	1	1	1
1	0	0	1	1	0	1
1	0	1	1	1	1	1
1	1	0	1	1	1	1
1	1	1	1	1	1	1

Figure 2.11: Truth table showing the associative property for the OR function.

WATCH ANIMATED FIGURE 2.11

De Morgan's Laws

Two theorems of logic developed by Augustus De Morgan are fundamental in digital logic design. However, they were not developed for digital circuits, which did not exist when these theorems were first published in 1850. Rather, they were developed as part of De Morgan's efforts as a reformer in the study of logic and propositional calculus. The original theory behind his laws dates to the 14th century. De Morgan's contribution was to develop the logical, mathematical expression of them. In De Morgan's own words, (1851) the theorems are:

> *(1) The negation (or contradictory) of a disjunction is equal to the conjunction of the negative of the alternatives – that is, not(p or q) equals not(p) and not(q), or symbolically* $\sim(p \vee q) = \sim p \cdot \sim q$.
> *(2) The negation of a conjunction is equal to the disjunction of the negation of the original conjuncts – that is, not(p and q) equals not(p) or not(q), or symbolically* $\sim(p \cdot q) = \sim p \vee \sim q$.

To decipher this, it helps to know that a conjunction is the AND function, a disjunction is the OR function, and the negation, contradiction, and ~ symbol are the NOT function.

Well, all of that was quite a mouthful. Fortunately, rewriting the laws in our notation, and using a and b instead of p and q, greatly simplifies them. The first law becomes

$$(a+b)' = a' \cdot b'$$

Once again, a truth table demonstrates the validity of this law. Figure 2.12 shows the truth table for De Morgan's first law.

a	b	$a + b$	$(a + b)'$	a'	b'	$a' \cdot b'$
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

Figure 2.12: Truth table verifying De Morgan's first law.

WATCH ANIMATED FIGURE 2.12

The second De Morgan's law can also be expressed simply in our notation.

$$(a \cdot b)' = a' + b'$$

Figure 2.13 shows the truth table to verify this law.

a	b	$a \cdot b$	$(a \cdot b)'$	a'	b'	$a' + b'$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

Figure 2.13: Truth table verifying De Morgan's first law

WATCH ANIMATED FIGURE 2.13

2.2 Boolean Functions

The fundamental Boolean operations we introduced in the previous section are just that, fundamental. They are fine for some of the things we want to model, and ultimately design and build circuits to implement. In most cases, however, we want to model more complex operations that may be a combination of these operations. We refer to these as **Boolean functions**.

To illustrate this, envision the following scenario based on one of my favorite sports, baseball. We are entering the last day of the regular season, and my favorite team is leading its division by one game over its dreaded archrival. Each team has one game remaining to be played, not against each other. Figure 2.14 shows one such scenario.

Team	W	L	GB
(*a*) Favorite	95	66	---
(*b*) Archrival	94	67	1.0

Figure 2.14: Baseball divisional standing entering the final day of the season.

In order to win its division, my favorite team must have a better record than its archrival. To model the possible last-game results, I'll start by creating two Boolean values.

a = My favorite team wins its final game
b = Dreaded archrival wins its final game

Each of these values is 1 if the statement is true, that is, the team wins its final game, and 0 if it is false and the team loses its final game. There are $2^2 = 4$ possible outcomes: both teams lose; my team loses and the archrival team wins; my team wins and the archrival team loses, and; both teams win. Figure 2.15 shows the final standings for all four scenarios.

Team	W	L	GB
(*a*) Favorite	95	67	---
(*b*) Archrival	94	68	1.0

(a)

Team	W	L	GB
(*a*) Favorite	95	67	---
(*b*) Archrival	95	67	---

(b)

Team	W	L	GB
(*a*) Favorite	96	66	---
(*b*) Archrival	94	68	2.0

(c)

Team	W	L	GB
(*a*) Favorite	96	66	---
(*b*) Archrival	95	67	1.0

(d)

Figure 2.15: Final division standings when (a) both teams lose their final game; (b) my favorite team loses and the archrival team wins; (c) my favorite team wins and the archrival team loses; and (d) both teams win.

As we can see from this figure, my team finishes with a better record and wins the division in three of the four scenarios. In the fourth scenario, the two teams are tied and a tiebreaker criterion determines the division winner. For this example, we'll assume the archrival team would win the tiebreaker, so my team would lose the division in this case. (Otherwise my favorite team would win 100% of the time, which would be great for my team but would make for a lousy example.)

Now let's put all this information into one truth table, shown in Figure 2.16. This truth table has two inputs, *a* and *b*, the two Boolean values we created earlier. It has one output, *W*, that is 1 when my favorite team wins the division and 0 when it does not. The first row, with *a* = 0 and *b* = 0, represents the scenario in which both teams lose their last game. In this case, shown in Figure 2.15 (a), my favorite team wins the division, indicated by a 1 in the *W* (= my team wins the division) column. The second row has *a* = 0 and *b* = 1, the scenario in which my favorite team loses its last game and the archrival team wins its last game. They both end up with the same record, as shown in Figure 2.15 (b). The archrival team holds the tiebreaker, and they win the division. My favorite team does not win the division, so *W* is 0. The third row is the case when my favorite team wins its final game (*a* = 1) and their archrival loses its final game (*b* = 0); this is shown in Figure 2.15 (c). Here, my favorite team wins its division, so *W* = 1. The last row, with *a* = 1 and *b* = 1, occurs when both teams win their final game. As shown in Figure 2.15 (d), my team wins the division, so *W* = 1.

a	*b*	W
0	0	1
0	1	0
1	0	1
1	1	1

Figure 2.16: Truth table for my favorite team winning its division.

Now that we have the truth table, we would next derive a single function for *W* that will produce the correct value for all possible cases. We'll look at two common methods to do this in the following subsections, along with a discussion of how they are related, followed by a couple of more complex examples.

2.2.1 Sum of Products

In the **sum of products** method, we create a product for each row of the truth table. For each product, we have either an input or its complement, for every single input, ANDed together. If a row has a 1 value for an input, we use the original input; if it is 0, we use its complement. By doing this, the product, the AND of these terms, is equal to 1 only for that row's input values.

This is probably a bit confusing, but another truth table may help clear this up. Figure 2.17 (a) shows all four possible ANDed values, $a' \cdot b'$, $a' \cdot b$, $a \cdot b'$, and $a \cdot b$. The first row, with $a = 0$ and $b = 0$, only sets one ANDed term to 1, $a' \cdot b'$. The other terms are also set to 1 for only one combination of values of *a* and *b*. The value set to 1 is called the **minterm** for these input values. Figure 2.17 (b) shows the minterm for each row.

a	b	$a' \cdot b'$	$a' \cdot b$	$a \cdot b'$	$a \cdot b$
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

(a)

a	b	Minterm
0	0	$a' \cdot b'$ (m_0)
0	1	$a' \cdot b$ (m_1)
1	0	$a \cdot b'$ (m_2)
1	1	$a \cdot b$ (m_3)

(b)

Figure 2.17: (a) Values of all minterms for all possible input values; (b) Minterm associated with all possible input values.

Notice the last value in each row of the table in Figure 2.17 (b). One common notation for minterms is to denote each minterm as lowercase *m* with a subscript. If you read the inputs from left to right as a binary number, and then convert that number to decimal, the decimal value is the subscript of *m* for that row. That is, when $a = 0$ and $b = 0$, the inputs are 00 in binary, which is 0 in decimal. The minterm for that row is thus m_0. Similarly, inputs of 01, 10, and 11 give us minterms m_1, m_2, and m_3, respectively.

So, now we have the products. Where's the sum? The sum is an OR operation. We take every minterm in a row that has the function output (*W* in this case) equal to 1, and logically OR them together. For this function, the first, third, and fourth rows have $W = 1$. Their minterms are $a' \cdot b'$, $a \cdot b'$, and $a \cdot b$. We can OR these three terms together to get $W = (a' \cdot b') + (a \cdot b') + (a \cdot b)$, or $W = m_0 + m_2 + m_3$.

2.2.2 Product of Sums

The **product of sums** method is sort of the opposite of the sum of products method. For each sum, we logically OR together either an input or its complement for every single input. Unlike sum of products, however, we use the original term if it is 0 and its complement if it is 1. As shown in Figure 2.18 (a), this gives us only one sum per row that is equal to 0; the rest are 1.

a	b	$a + b$	$a + b'$	$a' + b$	$a' + b'$
0	0	0	1	1	1
0	1	1	0	1	1
1	0	1	1	0	1
1	1	1	1	1	0

(a)

a	b	Maxterm
0	0	$a + b$ (M_0)
0	1	$a + b'$ (M_1)
1	0	$a' + b$ (M_2)
1	1	$a' + b'$ (M_3)

(b)

Figure 2.18: (a) Values of all maxterms for all possible input values; (b) Maxterm associated with all possible input values.

The sums shown in Figure 2.18 (a) are called **maxterms**. Unlike minterms, which have a value of 1 in only one row, maxterms have a value of 0 in only one row. Furthermore, only one maxterm is 0 in each row. The maxterm with a value of 0 is the maxterm for that row. The maxterms for all rows are shown in Figure 2.18 (b). Just as we used lowercase *m* for minterms, we use uppercase *M* with a subscript for maxterms. The subscript for the maxterms is calculated exactly the same as it was calculated for minterms.

Now that we have the sums, we need to combine them to create our final function. We do this by ANDing together the maxterms for all rows that set the function to 0.

For our function, there is only one maxterm that is 0, $a + b'$, or M_1. Thus, this becomes our final function, $W = a + b'$. It is straightforward to verify that this function yields the desired truth table shown in Figure 2.16.

2.2.3 The Relationship between Minterms and Maxterms

There seems to be a sort of symmetry between minterms and maxterms. When we express a function as a sum of products using minterms, we take all the minterms that set a function to 1 and OR them together. Each minterm *adds* one set of input values that sets the function to 1. In our example, there are three cases that set our output to 1, so there are three minterms that are ORed together to give us our final function, $W = (a' \cdot b') + (a \cdot b') + (a \cdot b)$.

In contrast, we express a function as a product of sums by ANDing together all maxterms that set the function to 0. Each maxterm *removes* one set of input values that sets the function to 0. When all the input values that set the function to 0 are removed, we are left with all the input values that set it to 1. This gives us a different, but equivalent, representation of the function, $W = a + b'$ in this example.

Now look at the truth tables we used to generate the *m* and *M* values. In Figure 2.15 (a), we see that each row has exactly one minterm set to 1, whereas Figure 2.18 (a) has exactly one maxterm set to 0. Furthermore, the position of these values is the same in each row. The value of minterm m_i is the opposite of the value of maxterm M_i, or $m_i = M_i'$, or $m_i' = M_i$.

We can show this is true by using De Morgan's laws. Consider $m_0 = a' \cdot b'$. Using De Morgan's second law, we know that

$$m_0' = (a' \cdot b')' = (a')' + (b')' = a + b = M_0$$

You can follow the same procedure to show that this is true for all minterms and maxterms.

We could have gone in the other direction as well, showing that $M_i = m_i'$, or $M_i' = m_i$. Again looking at $M_0 = a + b$, we use De Morgan's first law to show that

$$M_0' = (a + b)' = a' \cdot b' = m_0$$

Finally, we can look at the two equations generated using minterms and maxterms to see another relationship. If we use our *m* and *M* terms, we have expressed our function *W* as

$$W = m_0 + m_2 + m_3$$
and
$$W = M_1$$

For each subscript value, we use its minterm if it yields a value of 1 for the function, or its maxterm if it generates a value of 0. Each index is included in exactly one of the two equations.

2.2.4 Additional Examples

In this subsection, we'll go through a couple of more complex examples. First, we'll start off with the function *q* = *a*'*bc*' + *ac*. This function has three inputs, *a*, *b*, and *c*, so its truth table has 2^3=8 rows. Figure 2.19 shows the truth table for this function and its minterms. To calculate the value of function *q*, we simply plugged in the values of *a*, *b*, and *c* into the equation for each row.

a	*b*	*c*	*q*	Minterm
0	0	0	0	*a*'*b*'*c*' m_0
0	0	1	0	*a*'*b*'*c* m_1
0	1	0	1	*a*'*bc*' m_2
0	1	1	0	*a*'*bc* m_3
1	0	0	0	*ab*'*c*' m_4
1	0	1	1	*ab*'*c* m_5
1	1	0	0	*abc*' m_6
1	1	1	1	*abc* m_7

Figure 2.19: Truth table for *q* = *a*'*bc*' + *ac* with minterms shown.

WATCH ANIMATED FIGURE 2.19

Looking at the table, we can reconstruct the equation as $q = m_2 + m_5 + m_7$, or *q* = *a*'*bc*' + *ab*'*c* + *abc*. This is different from, but equivalent to, our initial equation. In the next section, we'll look at ways to simplify our equations as much as possible.

Now let's do this again, but using maxterms instead of minterms. We have reconstructed the truth table in Figure 2.20, this time showing the maxterm for each row.

a	*b*	*c*	*q*	Maxterm
0	0	0	0	a+*b*+*c* M_0
0	0	1	0	a+*b*+*c*' M_1
0	1	0	1	a+*b*'+*c* M_2
0	1	1	0	a+*b*'+*c*' M_3
1	0	0	0	a'+*b*+*c* M_4
1	0	1	1	a'+*b*+*c*' M_5
1	1	0	0	a'+*b*'+*c* M_6
1	1	1	1	a'+*b*'+*c*' M_7

Figure 2.20: Truth table for $q = a'bc' + ac$ with maxterms shown.

WATCH ANIMATED FIGURE 2.20

ANDing together the maxterms for rows with $q = 0$ gives us the equation

$$q = M_0 \cdot M_1 \cdot M_3 \cdot M_4 \cdot M_6, \text{ or}$$
$$q = (a + b + c) \cdot (a + b + c') \cdot (a + b' + c') \cdot (a' + b + c) \cdot (a' + b' + c).$$

Here is one final example to conclude this section. Instead of starting with an equation, we are given a truth table and must develop the sum of products and product of sums equations for this function. The truth table for this example is given in Figure 2.21. To simplify our work, the minterms and maxterms are also shown.

a	*b*	*c*	*q*	Minterm	Maxterm
0	0	0	0	*a*'*b*'*c*' m_0	a+*b*+*c* M_0
0	0	1	1	*a*'*b*'*c* m_1	a+*b*+*c*' M_1
0	1	0	1	*a*'*bc*' m_2	a+*b*'+*c* M_2
0	1	1	1	*a*'*bc* m_3	a+*b*'+*c*' M_3
1	0	0	1	*ab*'*c*' m_4	a'+*b*+*c* M_4
1	0	1	1	*ab*'*c* m_5	a'+*b*+*c*' M_5
1	1	0	0	*abc*' m_6	a'+*b*'+*c* M_6
1	1	1	1	*abc* m_7	a'+*b*'+*c*' M_7

Figure 2.21: Truth table for the function *q* with minterms and maxterms shown.

WATCH ANIMATED FIGURE 2.21

As before, we create the sum of products form of the equation for q by ORing together all minterms for which $q = 1$. For this truth table, this becomes

$$q = m_1 + m_2 + m_3 + m_4 + m_5 + m_7, \text{ or}$$
$$q = (a' \cdot b' \cdot c) + (a' \cdot b \cdot c') + (a' \cdot b \cdot c) + (a \cdot b' \cdot c') + (a \cdot b' \cdot c) + (a \cdot b \cdot c)$$

Using maxterms, we find the product of sums equation by ANDing the maxterms for which $q = 0$. This gives us the equations

$$q = M_0 \cdot M_6, \text{ or}$$
$$q = (a + b + c) \cdot (a' + b' + c)$$

One thing to note about these equations is that they may have quite a few terms. In general, we want to make our equations as simple as possible. When we design digital circuits to realize functions, simpler functions lead to smaller and simpler circuits. Simpler circuits are highly desirable. In the next section, we'll look at ways to minimize these logic equations.

2.3 Minimizing Functions

As we saw in the previous section, both the sum of products and product of sums methods give us correct equations for desired functions, but these equations may be unnecessarily large. In practice, we want functions to be as simple as possible while still being correct. When we create digital circuits to realize functions, simpler functions require less hardware and wiring, use less power, and usually are faster than circuits for more complex functions. Knowing how to simplify functions is an important skill for digital logic designers to have.

In this section, we introduce **Karnaugh maps**, or **K-maps**, a visual tool that designers can use to minimize functions. Karnaugh maps work well when a function has up to four terms. Beyond that, this method becomes increasingly difficult to use. Another method, the **Quine-McCluskey algorithm**, can be used for those functions; we'll introduce that method as well. Finally, we'll examine how to use these methods for functions that are incompletely specified, that is, we don't know the value of the function for certain values of the inputs, nor do we care. This is not at all uncommon; it typically occurs when we know that certain input values will never occur. We can use this to our advantage to minimize some functions even more than we otherwise would be able to.

2.3.1 Karnaugh Maps

Just as Boolean algebras were developed from a logic perspective, Karnaugh maps also have their foundations in logic theory. **Allan Marquand** developed a method to diagram logic for n elements in the early 1880s. This work was expanded by **Edward Veitch** in 1952 as the basis for his **Veitch diagrams**, which are used to "simplify truth functions." **Maurice Karnaugh** further extended Veitch's work to create the Karnaugh maps we still use today to minimize digital logic circuits.

A Karnaugh map looks similar to a grid. To create this grid, we split up the function inputs into two sets as evenly as possible. If we have four inputs, two are placed in each set; three inputs are split so one set has two and the other has one. The possible values in the two sets become the labels for the rows and columns of the grid. The value placed in the cell of the grid is the value of the function for the values of the inputs for the cell's row and column. There's more to this, but let's stop here for a moment.

2.3.1.1 2-Input Maps

Now that we've dispensed with the formal, perhaps a bit confusing description, let's look at some Karnaugh maps for functions to see how this really works.

Let's start with our baseball function, $W = (a' \cdot b') + (a \cdot b') + (a \cdot b)$. This function has two inputs, *a* and *b*, so it's easy to split them into two equal sets, one with *a* and the other with *b*. Each set has one element which has $2^1 = 2$ possible values, so we need to have a grid that is 2 × 2. I'll arbitrarily assign the set with *a* to the rows and the set with *b* to the columns. This gives us the map shown in Figure 2.22 (a).

*a**b*	0	1
0		
1		

(a)

a	*b*	W
0	0	1
0	1	0
1	0	1
1	1	1

(b)

*a**b*	0	1
0	1	0
1	1	1

(c)

Figure 2.22: (a) A 2 × 2 Karnaugh map; (b) Truth table for the baseball function; (c) Karnaugh map with values for function *W* shown.

WATCH ANIMATED FIGURE 2.22

Notice the label in the upper left corner of the grid, *a**b*. This indicates that the values shown below this square, the labels of the rows, are the possible values of *a*. The values to the right of this, the labels of the columns, are the possible values of *b*.

Next, we need to fill in the rest of the Karnaugh map. Consider the first empty cell, in the row with *a* = 0 and the column with *b* = 0. We go to the truth table we developed earlier for this function, repeated in Figure 2.22 (b), and look up the value of *W* for these input values. Here, *W* = 1, so we enter the value 1 into this cell. We do the same thing for each cell in the Karnaugh map, giving us the completed Karnaugh map shown in Figure 2.22 (c).

So, now we have this Karnaugh map. We didn't create it just to make a nice looking map. We want to use it to help us develop the most simple function equation possible. To do this, we group together adjacent 1s in the map. For this map, look at the bottom row. It has two cells,

both with a value of 1. These cells are adjacent, so we can group them together. We do this by drawing a circle or oval around the two terms, as shown in Figure 2.23 (a). This is called an **implicant**.

(a) (b)

Figure 2.23: Karnaugh map for the function *W*: (a) with *a* terms grouped; (b) with *a* terms and *b*' terms grouped.

There is a second implicant in this Karnaugh map, the two 1s in the column with $b = 0$. Figure 2.23 (b) shows the Karnaugh map with both implicants.

Notice the 1 in the row with $a = 1$ and the column with $b = 0$. That term is in both implicants, and that's OK. *A cell with a 1 can be in more than one implicant*.

Here is another important point. *Every cell with a 1 must be in an implicant*. If it cannot be grouped with any other cell, we just circle it and make it its own implicant.

Mathematically, these implicants are created by performing algebraic operations. Consider the first implicant, which combines the cells with $a = 1$ and $b = 0$, with the cell for which $a = 1$ and $b = 1$. The first cell corresponds to minterm $a \cdot b'$ ($a = 1$, $b = 0$ or $b' = 1$, $1 \cdot 1 = 1$), and the second is minterm $a \cdot b$. Grouping the two cells is essentially a logical OR operation. This group corresponds to the following equation.

$$\begin{aligned} \text{Group 1} &= (a \cdot b') + (a \cdot b) \\ &= a \cdot (b' + b) \\ &= a \cdot 1 \\ &= a \end{aligned}$$

This is the rationale for grouping terms. When we group terms together, we create one term that covers the group's terms. As a bonus, the one term is simpler than any of the terms that are grouped together.

We can follow the same procedure for the second implicant as follows.

$$\begin{aligned} \text{Group 2} &= (a' \cdot b') + (a \cdot b') \\ &= (a' + a) \cdot b' \\ &= 1 \cdot b' \\ &= b' \end{aligned}$$

Finally, we logically OR the two implicant values together to get our final function, $W = a + \cdot b'$.

2.3.1.2 3-Input Maps

Karnaugh maps for two inputs are useful, to a degree, but their value is somewhat limited. If you didn't have the Karnaugh map, you might be able to just look at the truth table and figure out the minimal equation because there are so few terms. Karnaugh maps become more useful when we have functions with more inputs, up to a point. Consider the Karnaugh map for a truth table we saw in the previous section, repeated in Figure 2.24 (a).

a	*b*	*c*	*q*
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

(a)

*a**bc*	00	01	11	10
0	0	0	0	1
1	0	1	1	0

(b)

*a**bc*	00	01	11	10
0	0	0	0	1
1	0	1	1	0

(c)

Figure 2.24: (a) Function truth table; (b) Karnaugh map; (c) Karnaugh map with implicants shown.

To create the Karnaugh map, first we divide the inputs as evenly as possible. This function has three inputs, so we put one (*a*) in one set and the other two (*b* and *c*) in the other set. It doesn't matter which input is in each set; I just chose these arbitrarily. Also, each input has to be completely in one set; you can't have 1½ inputs in each set.

Since the first set has one input, it has $2^1 = 2$ rows, just like the Karnaugh map for the baseball function. The other set, however, has two inputs, so our Karnaugh map has $2^2 = 4$ columns. Figure 2.24 (b) shows the Karnaugh map for this truth table.

There are a couple of really important things I need to explain before we continue. First, notice the top left cell of the Karnaugh map has the text *a**bc*. This means that the two-bit value in the label for each column specifies the values of *b* and *c* in that order. For example, the column labeled 01 has $b = 0$ and $c = 1$.

The other thing to note is the order of the labels. When we have only two rows or columns, the order is always sequential, first 0, then 1. When we have more than two rows or columns, however, this is not the case. The columns are not ordered 00, 01, 10, 11; they are ordered 00, 01, 11, 10. *The order of the labels follows the Gray code*. We do this so we can group adjacent terms. If we ordered the columns sequentially, we could not group together entries in adjacent columns 01 and 10.

Finally, we group terms just as we did before. The 1 in the upper right corner (a = 0, bc = 10, or $a'bc'$) cannot be grouped with any other terms, so we just circle it as its own implicant. The other two terms can be grouped together as follows.

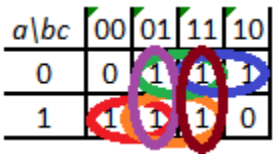
$$(a \cdot b' \cdot c) + (a \cdot b \cdot c) = (a \cdot c) \cdot (b' + b) = (a \cdot c)$$

Thus our final function becomes $a'bc' + ac$.

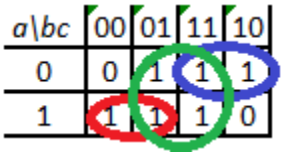
Indicants can have more than two entries of 1, but the total number of entries must always be an exact power of 2, that is, 1, 2, 4, 8, … and so on. To see how this works, let's look at another truth table we saw earlier, repeated in Figure 2.25 (a).

a	b	c	q
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

(a)



(b)

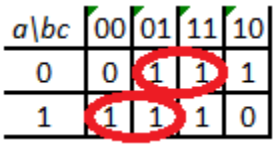


(c)

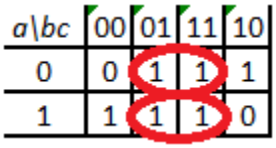
Figure 2.25: (a) Function truth table; (b) Karnaugh map with implicants of size 2; (c) Karnaugh map with implicants prime implicants shown.

We can create the Karnaugh map for this truth table just as we did before, and then we circle each possible implicant of two terms, as shown in Figure 2.25 (b). There are a lot of implicants circled in this figure, six to be precise. Now, we want to see if we can combine any of these implicants of two into larger implicants.

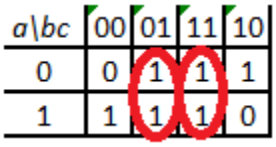
Two implicants can be combined if they are completely adjacent. That is, they can't have only part of the group being adjacent. For this function, consider the two implicants shown in Figure 2.26 (a). The two terms in column 01 are adjacent, but the terms in columns 00 and 11 are not, so these two implicants cannot be combined.



(a)



(b)



(c)

Figure 2.26: (a) Two non-adjacent implicants; (b) and (c) Two adjacent implicants that can be combined.

The two implicants circled in Figure 2.26 (b), however, can be combined. Each term along their common edge is adjacent to a term in the other implicant. The same is true for the two implicants circled in Figure 2.26 (c). The final implicants are shown in Figure 2.25 (c). The three implicants in the figure are all of maximum size; they cannot be combined further. They are called **prime implicants**. Both groupings in Figures 2.26 (b) and (c) group together the same elements and give us the same function.For the first figure, our implicants are

$$(a \cdot b' \cdot c) + (a \cdot b \cdot c) = (a \cdot c), \text{ and}$$
$$(a' \cdot b' \cdot c) + (a' \cdot b \cdot c) = (a' \cdot c)$$

Our final implicant is

$$(a \cdot c) + (a' \cdot c) = c$$

The second figure has the implicants

$$(a' \cdot b' \cdot c) + (a \cdot b' \cdot c) = (b' \cdot c), \text{ and}$$
$$(a' \cdot b \cdot c) + (a \cdot b \cdot c) = (b \cdot c)$$

They are grouped together as

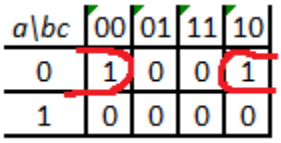
$$(b' \cdot c) + (b \cdot c) = c$$

The other two groups have the functions $(a \cdot b')$ and $(a' \cdot b)$, and our final function is $(a \cdot b') + (a' \cdot b) + c$.

There is another way that terms can be adjacent that is not so obvious. Consider the Karnaugh map shown in Figure 2.27 (a). There are two terms set to 1 in this Karnaugh map, and they appear quite distant from each other. However, they are actually adjacent, because *the rows and columns of the Karnaugh map wrap around*. That is, the last cell in each row/column is actually next to the first cell in that row/column. Traditionally, we group them as shown in Figure 2.27 (b).

a\bc	00	01	11	10
0	1	0	0	1
1	0	0	0	0

(a)



(b)

Figure 2.27: (a) Karnaugh map; (b) Karnaugh map with implicant shown.

Mathematically, the two cells are adjacent if they have only one input that is different, and that is the case here. The left cell has $a = 0$, $b = 0$, and $c = 0$, and corresponds to minterm

$a'b'c'$. The other cell set to 1 has $a = 0$, $b = 1$, and $c = 0$, or minterm $a'bc'$. Grouping these two cells together gives us an implicant with value
$(a' \cdot b' \cdot c') + (a' \cdot b \cdot c') = (a' \cdot c') \cdot (b' \cdot b) = (a' \cdot c')$.

We can only draw the Karnaugh map in two dimensions, as a flat grid. Mathematically speaking, however, it is more like a three-dimensional torus. To see this, get a doughnut and a piece of paper. Draw a Karnaugh map (just the cells, no labels) on a piece of paper, cut it out, and try wrapping it around the doughnut. You may find it helpful to draw the squares as trapezoids since the inner diameter of the doughnut is smaller than the outer diameter. This will show you all the adjacencies within the Karnaugh map. After you have done this, take a picture to document your success, and then feel free to remove your Karnaugh map and eat the doughnut as your reward for a job well done.

2.3.1.3 4-Input Maps

The largest Karnaugh map you can reasonably use has four inputs. This map is shown in Figure 2.28. Notice that both the rows and columns use the Gray code sequence for their labels. It is left as an exercise for the reader to group the terms in this map into prime implicants and obtain an optimal function.

ab\cd	00	01	11	10
00	0	1	0	1
01	1	1	0	0
11	1	1	1	1
10	0	1	0	0

Figure 2.28: A 4-input Karnaugh map with function values shown.

There is one final point I need to share with you regarding implicants. Sometimes, after you have created your Karnaugh map and generated all the prime implicants, you don't need to include all of them in your final equation for the function. As an example, consider the Karnaugh map shown in Figure 2.29 (a). This map has three prime implicants as shown in the figure. The lower prime implicant has the function $a \cdot b'$; the vertical prime implicant has the function $b' \cdot c$; and the upper prime implicant has the function $a' \cdot c$. We could express this overall function as $(a \cdot b') + (b' \cdot c) + (a' \cdot c)$, but we can do better than that. Notice that both cells in $b' \cdot c$ are also included in one of the other two prime implicants, so we can remove it from the overall function, leaving us with $(a \cdot b') + (a' \cdot c)$, as shown in Figure 2.29 (b).

(a) (b)

Figure 2.29: Karnaugh map with (a) all prime implicants; (b) essential prime implicants.

Look at the original map with all prime implicants and find the cell with $a = 1$, $b = 0$, and $c = 0$. This cell is in only one prime implicant. That means any final function must include this prime implicant, otherwise it will not produce the desired function value of 1 for these input values. Such a prime implicant is called an **essential prime implicant**. Using similar reasoning, the upper prime implicant is also an essential prime implicant. Knowing that these two essential prime implicants must be in the final function, we can see that we are already setting the function to 1 for all terms in the vertical prime implicant, so we don't need to include it.

Summarizing, here's the overall procedure to determine the function from a truth table using Karnaugh maps.

- Divide the function inputs into two sets, splitting them up as evenly as possible.
- Create the blank Karnaugh map with these two sets, using the Gray code to create the labels.
- Fill in the cells of the Karnaugh map with the values of the functions from the truth table.
- Group adjacent terms together as implicants.
- Group implicants together to form larger implicants, repeating until each implicant has reached its maximum size. These are the prime implicants for the function.
- Identify cells that are covered by only one prime implicant. These are the essential prime implicants and must be included in the final function.
- Find a cell that is equal to 1 but not in any essential prime implicant. Find one prime implicant that covers this cell and add it to the final function. Repeat this process for other cells that are not covered by any essential prime implicant and the other prime implicants you have added to the final function until all cells are covered.

2.3.2 Quine-McCluskey Algorithm

In theory, you can create a Karnaugh map for any number of function inputs. Realistically, though, Karnaugh maps are difficult to work with when you have more than four function inputs. For n inputs, each cell in the Karnaugh map is adjacent to n other cells. For 5 or more inputs, some of the adjacent cells do not border the cell, which takes away one of the main advantages of Karnaugh maps.

For functions with larger numbers of inputs, we can use the **Quine-McCluskey algorithm**. (This algorithm can also be used for functions with fewer inputs, but Karnaugh maps already work well for these functions.) The original algorithm was developed by **William Quine** in 1952, though the underlying approach was previously developed by **Hugh McColl** in 1878 in the domain of logic. **Edward McCluskey** extended this algorithm in 1956 to give us the algorithm we use today. Computer programs that minimize logic functions are based on this algorithm, not Karnaugh maps.

The Quine-McCluskey algorithm is a tabular method that has two phases. In the first phase, we create a table and manipulate it to determine the prime implicants for the function. This is equivalent to circling adjacent terms in Karnaugh maps and building them up until each has reached its maximum size. Then we determine the function from among these prime implicants in the second phase. We'll describe the steps in this algorithm as we go through a very simple example you have seen before, the baseball example we have used throughout this chapter.

We start with the truth table for this function, shown again in Figure 2.30 (a). We use this table to create the table we will actually use in the first phase of the Quine-McCluskey algorithm. We take each entry in the truth table that sets the function value to 1 and add the input values for that entry to our table. Then we group all the entries based on the number of 1s in the entries. For our truth table, there are three entries that set the function output to 1. They are: a = 0 and b = 0; a = 1 and b = 0; and a = 1 and b = 1. We list these entries as 00, 10, and 11, respectively. Then we divide the entries by the number of 1s. Group 0 has 00; group 1 has 01; and group 2 has 11. These groups are shown in Figure 2.30 (b).

a	*b*	W
0	0	1
0	1	0
1	0	1
1	1	1

(a)

0: 00
1: 10
2: 11

(b)

(c)

Figure 2.30: Determining the equation for the baseball function using the Quine-McCluskey algorithm: (a) Truth table for the function; (b) Table used by the Quine-McCluskey algorithm; (c) Generating prime implicants.

WATCH ANIMATED FIGURE 2.30

Now that we have our table, the next step is to generate the prime implicants. This process is analogous to finding prime implicants in Karnaugh maps. We group together terms until we have grouped as many as possible. To do this, we start with group 0. We take every term in group 0 and compare it to every term in group 1. If a pair of terms has all but one input that are the same, we create an entry in the next column with a dash in place of the input that is different, and all the values that are the same. As shown in Figure 2.30 (c), we combine 00

and 10 to form term -0. Note that we don't have to compare group 0 to group 2. Since group 2 has two terms that are 1 and group 0 has no terms that are 1, there cannot be a pair of terms that vary by only one input.

Next, we go to group 1 and compare each term to every term in group 2. This combines 10 and 11 to form the term 1-. If there are any terms that could not be combined with any other terms, we circle these terms; they are prime implicants. None of our initial terms in this example are prime implicants; each was combined with at least one other term.

We would then group these newly created terms based on the number of 1s and repeat the process, with one very important difference. We only compare terms that have their dashes in the same location. This ensures that we avoid combining terms that are not aligned properly. In our example, our two newly created terms, 1- and -0, have their dashes in different locations, so we cannot compare them. We circle them both and phase 1 of our algorithm is complete. If we had created new terms, we would repeat the process on each column of newly created terms.

All entries in each column have exactly the same number of dashes. Initially, no entries have any dashes. In the next column, each entry has exactly one dash, which makes sense since we replace the one input that is different with a dash. Every newly created column adds one dash to the entries. A function with *n* inputs will create at most *n* columns, usually less.

For all the prime implicants we generated using the first phase of the Quine-McCluskey algorithm, the values without dashes can be used to represent the term as a product of input variables. A 1 represents the original term and a 0 signifies that we use the complement of the term. Terms with a dash are not included. Just as combining terms algebraically ended up removing an input from the combined term, the dash does the same thing. Our two prime implicants, 1- and -0, correspond to *a* and *b*', respectively.

Now, on to phase 2. We need to determine which prime implicants to include in our final function. To do this, we create another table. The columns of this table are the input values that set the function to 1. The rows are the prime implicants we generated in phase 1. Our baseball function has three columns (00, 10, and 11), and two rows (1- and -0). In each entry in the table, we place a check mark if the prime implicant includes the term for that column, or leave it blank if it does not. The initial table is shown in Figure 2.31 (a).

	00	10	11
1-		✔	✔
-0	✔	✔	

(a)

(b)

Figure 2.31: Quine-McCluskey algorithm, phase 2, for the baseball example: (a) Initial table; (b) Table with essential prime implicants circled and rows and columns crossed out.

WATCH ANIMATED FIGURE 2.31

First, we look at this table to see if any column has only one entry checked. If so, the prime implicant for that column is an essential prime implicant. We circle that term and draw a horizontal line through that row. We find all entries in that row and draw a vertical line through their columns. For our example, we have two essential prime implicants. Figure 2.31 (b) shows the table with essential prime implicants circled and rows and columns crossed out.

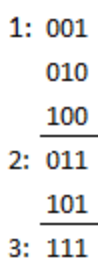
If any columns are not yet crossed out, we still need prime implicants to ensure those terms set the function to 1. We try to choose prime implicants that cover as many of the remaining terms as possible to minimize the overall function. For our example, all terms are covered by the essential prime implicants, and our final function is $W = a + b'$.

Another familiar example

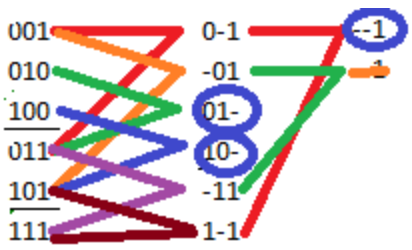
Let's look at another example we've seen before, the truth table repeated in Figure 2.32 (a). This function has six input values that set the function to 1. Grouping these input values as before gives us the table shown in Figure 2.32 (b).

a	*b*	*c*	*q*
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

(a)



(b)



(c)

Figure 2.32: Another Quine-McCluskey example: (a) Truth table for the function; (b) Table used by phase 1; (c) Generating prime implicants.

WATCH ANIMATED FIGURE 2.32

As before, we compare each term in group 1 to each term in group 2, and each term in group 2 to each term in group 3 to generate the second column in Figure 2.32 (c). We repeat the procedure on the second column, remembering to compare only entries that have dashes in the same location. Two entries, 01- and 10-, cannot be combined with any other terms and are circled.

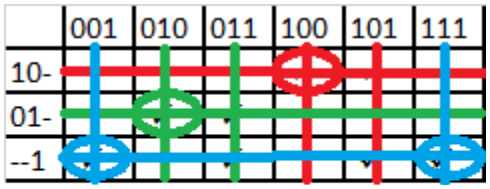
Notice, in the last column, we generated the term --1 twice. Recall that when we used Karnaugh maps to determine this function, we were able to combine two implicants of size 2 to create the same implicant of size 4 in two different ways; see Figure 2.26. That is what's happening here. Since we only need this term once, we simply cross out any duplicates.

Finally, with only one term left, --1, there is nothing to combine it with and it is circled as a prime implicant. Our three prime implicants are 10-, 01-, and --1, or *ab*', *a*'*b*, and *c*.

Now we move on to phase 2. We create the table for this function as before, shown in Figure 2.33 (a). Looking at the table, the columns for inputs 001, 010, 100, and 111 each have only one entry checked. The prime implicants for these entries are essential. We circle them and cross out the rows and columns as before, as shown in Figure 2.33 (b). With all columns covered, this phase is complete. Our final function is $(a \cdot b') + (a' \cdot b) + c$.

	001	010	011	100	101	111
10-				✔	✔	
01-		✔	✔			
--1	✔		✔		✔	✔

(a)



(b)

Figure 2.33: Phase 2 of the Quine-McCluskey algorithm: (a) Initial table; (b) Table with essential prime implicants circled and rows and columns crossed out.

WATCH ANIMATED FIGURE 2.33

A new, more complex example

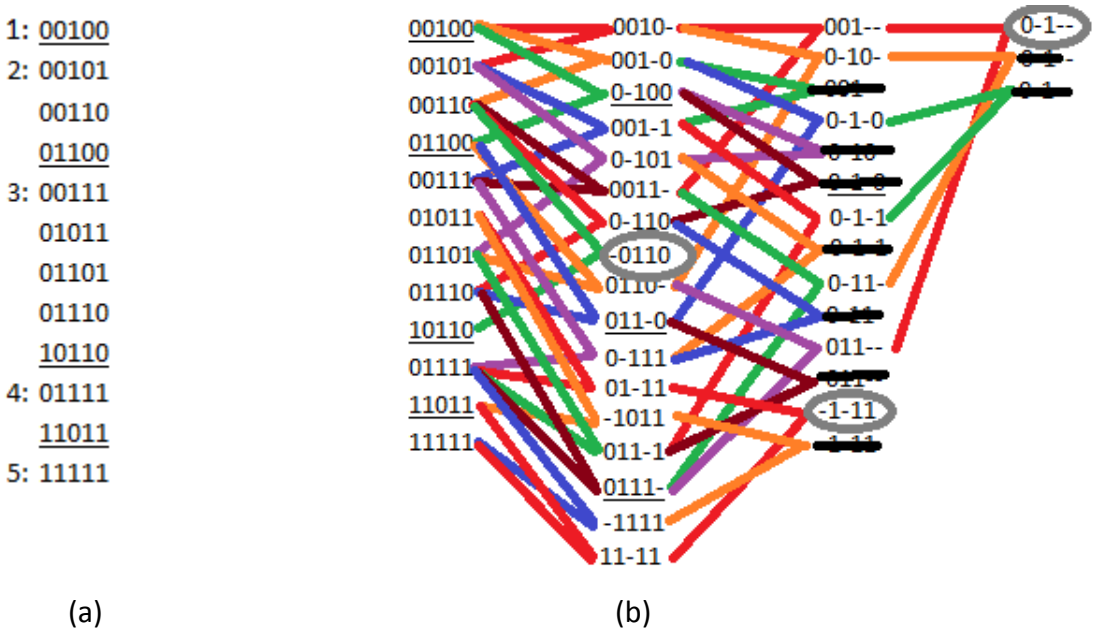
To close out this section, let's look at a more complex example with five inputs. For larger functions, it is not uncommon to express it as a sum of minterms using the little-*m* notation. Our function is $\Sigma m(4, 5, 6, 7, 11, 12, 13, 14, 15, 22, 27, 31)$. The expanded truth table is shown in Figure 2.34.

a	*b*	*c*	*d*	*e*	*q*
0	0	0	0	0	0
0	0	0	0	1	0
0	0	0	1	0	0
0	0	0	1	1	0
0	0	1	0	0	1
0	0	1	0	1	1
0	0	1	1	0	1
0	0	1	1	1	1
0	1	0	0	0	0
0	1	0	0	1	0
0	1	0	1	0	0
0	1	0	1	1	1
0	1	1	0	0	1
0	1	1	0	1	1
0	1	1	1	0	1
0	1	1	1	1	1

a	*b*	*c*	*d*	*e*	*q*
1	0	0	0	0	0
1	0	0	0	1	0
1	0	0	1	0	0
1	0	0	1	1	0
1	0	1	0	0	0
1	0	1	0	1	0
1	0	1	1	0	1
1	0	1	1	1	0
1	1	0	0	0	0
1	1	0	0	1	0
1	1	0	1	0	0
1	1	0	1	1	1
1	1	1	0	0	0
1	1	1	0	1	0
1	1	1	1	0	0
1	1	1	1	1	1

Figure 2.34: Truth table for final example function

As always, we create the table for phase 1 of the algorithm. We can read these values directly from the truth table, or we can simply look at the minterms and list the subscripts as 5-bit values (because we have five inputs for our function). Using either method, we get the initial table shown in Figure 2.35 (a).



(a) (b)

Figure 2.35: Final example phase 1 table: (a) Initial table; (b) Final table with prime implicants circled.

WATCH ANIMATED FIGURE 2.35

Figure 2.35 (b) shows the results obtained from phase 1 of the Quine-McCluskey algorithm. There are three prime implicants: -0110, -1-11, and 0-1--, or *b*'*cde*', *bde*, and *a*'*c*.

With this information, we create the table for phase 2 of the algorithm, shown in Figure 2.36. Once again, all three prime implicants are essential. Our final function is *a*'*c* + *bde* + *b*'*cde*'.

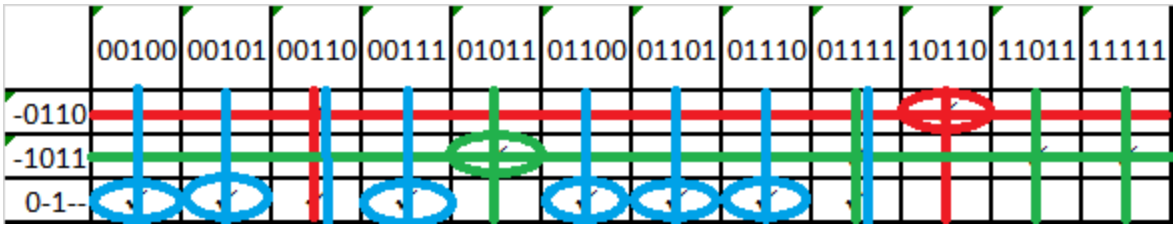


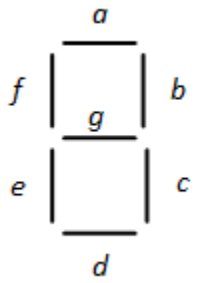
Figure 2.36: Final table for phase 2 of the final example function.

WATCH ANIMATED FIGURE 2.36

2.3.3 Don't Care Values and Incompletely Specified Functions

The functions we've looked at so far are completely specified, that is, they have a specific output value for every possible combination of input values. However, this is not always the case. For some functions, there are times we don't care what output value is generated. The most common reason for this is that certain combinations of input values will never occur. We don't care what output our function generates because we will never input those particular values.

A classic example of this is a BCD to 7-segment decoder. This function has four inputs; together, these inputs represent decimal digits 0 (0000) to 9 (1001). There are actually seven functions, *a* through *g*, that either turn on or turn off one of the seven segments of an LED display. Figure 2.37 shows the segments and the truth table for our ten defined inputs. A value of 1 indicates that the LED for that segment is lit.



$I_3I_2I_1I_0$	*a*	*b*	*c*	*d*	*e*	*f*	*g*
0000	1	1	1	1	1	1	0
0001	0	1	1	0	0	0	0
0010	1	1	0	1	1	0	1
0011	1	1	1	1	0	0	1
0100	0	1	1	0	0	1	1
0101	1	0	1	1	0	1	1
0110	1	0	1	1	1	1	1
0111	1	1	1	0	0	0	0
1000	1	1	1	1	1	1	1
1001	1	1	1	1	0	1	1

$I_3I_2I_1I_0$	*a*	*b*	*c*	*d*	*e*	*f*	*g*
0000	1	1	1	1	1	1	0
0001	0	1	1	0	0	0	0
0010	1	1	0	1	1	0	1
0011	1	1	1	1	0	0	1
0100	0	1	1	0	0	1	1
0101	1	0	1	1	0	1	1
0110	1	0	1	1	1	1	1
0111	1	1	1	0	0	0	0
1000	1	1	1	1	1	1	1
1001	1	1	1	1	0	1	1
1010	X	X	X	X	X	X	X
1011	X	X	X	X	X	X	X
1100	X	X	X	X	X	X	X
1101	X	X	X	X	X	X	X
1110	X	X	X	X	X	X	X
1111	X	X	X	X	X	X	X

(a) (b) (c)

Figure 2.37: (a) 7-segment LED labels; (b) Truth table for defined inputs; (c) Truth table with don't care values shown.

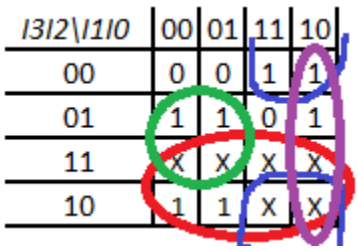
But what happens if we input one of the unspecified values, 1010 through 1111? In practice, we're only displaying decimal digits, so we don't care what it shows because we're never going to input those values anyway. The question is, how can we use this to our advantage to minimize our functions?

This is the role of the **don't care** output. We denote this output as X in the truth table. Figure 2.37 (c) shows the truth table with don't care values included.

To see how we handle don't care values, let's look at the function for segment *g*. Using the last column of the table, we can create the Karnaugh map shown in Figure 2.38 (a). Just as we included outputs set to X in the truth table, we also include X values in the Karnaugh map.

I3I2\I1I0	00	01	11	10
00	0	0	1	1
01	1	1	0	1
11	X	X	X	X
10	1	1	X	X

(a)



(b)

Figure 2.38: Segment *g* of the BCD to 7-segment decoder: (a) Original Karnaugh map; (b) One possible final Karnaugh map with prime implicants shown.

WATCH ANIMATED FIGURE 2.38

When we start to group together terms, we might group together the two 1s in the lower left cells, with inputs 1000 and 1001. But these two cells can be grouped with the two don't care values above it to form a group of four terms. And these four cells can be grouped together with the four cells to their right to form a group of eight cells, which becomes a prime implicant.

We continue the process, identifying other prime implicants and winnowing them down to create a final function. Figure 2.38 (b) shows the Karnaugh map with final prime implicants; this function is $I_3 + (I_2 \cdot I_1') + (I_1 \cdot I_0') + (I_2' \cdot I_1)$. There is another possible function for this segment with one different prime implicant. See if you can find it before continuing with the rest of this section.

The Quine-McCluskey algorithm has an elegant way of handling don't care values. It includes them in phase 1 to minimize the prime implicants, but it excludes them in phase 2 since we don't really have to generate any specific output value for these inputs.

In our compact minterm notation, we would represent this function as

$\sum m(2, 3, 4, 5, 6, 8, 9) + \sum d(10.\,11.\,12.\,13.\,14.\,15)$. The *m* terms are the minterms that set *g* to 1 and the *d* terms are the don't care values. Figure 2.39 shows phase 1 of the algorithm.

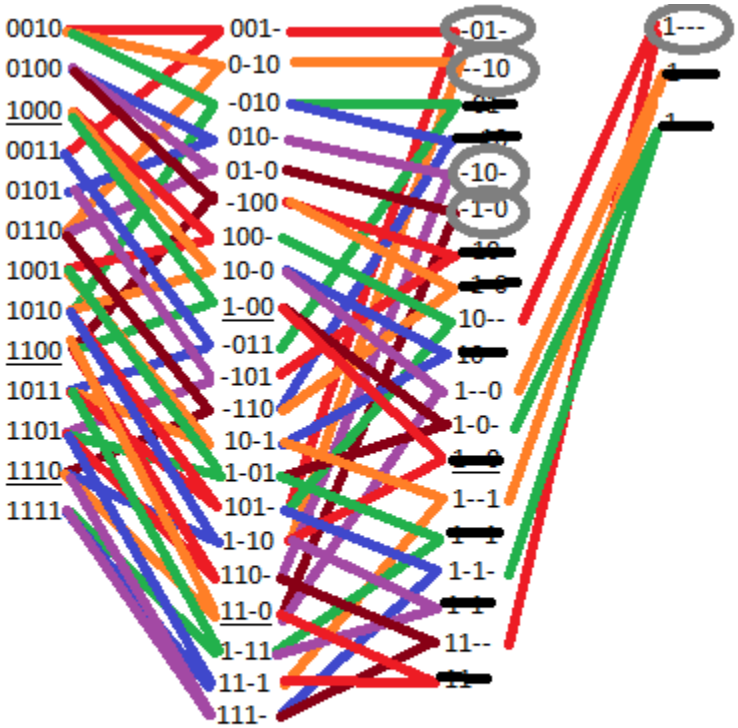


Figure 2.39: Phase 1 of the Quine-McCluskey algorithm to determine the function for *g*.

WATCH ANIMATED FIGURE 2.39

Finally, we create the table for phase 2, this time excluding the don't care values. Figure 2.40 shows this table after essential prime implicants are identified. These three terms, -01-, -10-, and 1---, ($I_2' \cdot I_1$, $I_2 \cdot I_1'$, and I_3) must be included in the final function. You can choose either of the two remaining prime implicants to cover input 0110 and complete the function.

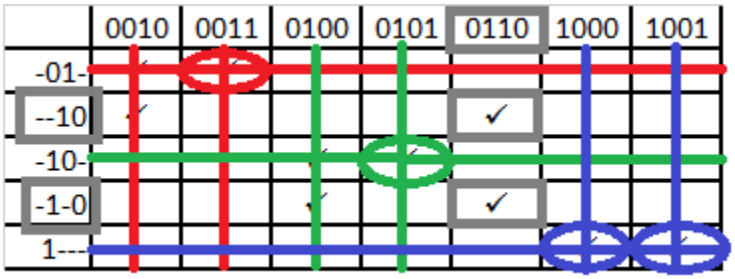


Figure 2.40: Phase 2 of the Quine-McCluskey algorithm to determine the function for *g*.

WATCH ANIMATED FIGURE 2.40

2.4 Summary

Boolean algebra is the mathematical basis for digital logic design. Using 1 and 0 to represent TRUE and FALSE, and incorporating the AND, OR, and NOT operations, we can create a function to realize any desired output values for all combinations of input values. Functions can be used to model real world scenarios. There are an infinite number of Boolean algebras. Fortunately, digital logic design is based on the very simplest Boolean algebra, also called the switching algebra. As with any algebra, our Boolean algebra follows some basic postulates that specify its behavior.

Truth tables are one way to show the behavior of a function. It specifies the value of the function for all possible combinations of input values. The equation for the function can be derived directly from these input values, or the outputs can be treated as minterms or maxterms and then used to specify the function.

Karnaugh maps are a useful tool that allows us to graphically combine terms and minimize function equations. Minimizing equations results in simpler digital logic designs to realize those functions, which has several benefits. Karnaugh maps become difficult to use when a function has more than four inputs. The Quine-McCluskey algorithm can be used for any number of inputs. Some functions do not specify output values for some input values. We can use these don't care outputs to minimize the function equations.

This concludes Part I, the background portion of this book. In the next chapter, we start the design of digital circuits with Part II, Combinatorial Logic. This chapter will introduce the basic digital logic components and show how they can be combined in circuits to realize desired functions.

Bibliography

De Morgan, A. (1851). On the Symbols of Logic, the Theory of the Syllogism, and in particular of the Copula, and the application of the Theory of Probabilities to some questions of Evidence. *Transactions of the Cambridge Philosophical Society*, *9*, 79.

De Morgan, A. (1860). *Syllabus of a proposed system of logic*. Walton and Maberly.

Hayes, J. P. (1993). *Introduction to digital logic design*. Addison-Wesley.

Huntington, E. V. (1904). Sets of Independent Postulates for the Algebra of Logic. *Transactions of the American Mathematical Society*, *5*(3), 288–309. https://doi.org/10.2307/1986459

Karnaugh, M. (1953). The map method for synthesis of combinational logic circuits. *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*, *72*(5), 593–599. https://doi.org/10.1109/TCE.1953.6371932

Mano, M. M., & Ciletti, M. (2017). *Digital design: with an introduction to the Verilog HDL, VHDL, and SystemVerilog* (6th ed.). Pearson.

Mano, M. M., Kime, C., & Martin, T. (2015). *Logic & computer design fundamentals* (5th ed.). Pearson.

Marquand, A. (1881). On logical diagrams for n terms. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, *12*(75), 266–270. https://doi.org/10.1080/14786448108627104

McCluskey, E. J. (1956). Minimization of Boolean functions. *The Bell System Technical Journal*, *35*(6), 1417–1444. https://doi.org/10.1002/j.1538-7305.1956.tb03835.x

McColl, H. (1878). The Calculus of equivalent statements (Third Paper). *Proceedings of the London Mathematical Society*, *s1-10*(1), 16–28. https://doi.org/10.1112/plms/s1-10.1.16

Nelson, V., Nagle, H., Carroll, B., & Irwin, D. (1995). *Digital logic circuit analysis and design*. Pearson.

Quine, W. V. (1952). The Problem of simplifying truth functions. *The American Mathematical Monthly*, *59*(8), 521–531. https://doi.org/10.2307/2308219

Quine, W. V. (1955). A Way to simplify truth functions. *The American Mathematical Monthly*, *62*(9), 627–631. https://doi.org/10.2307/2307285

Roth, J. C. H., Kinney, L. L., & John, E.B. (2020). *Fundamentals of logic design enhanced edition* (7th ed.). Cengage Learning.

Veitch, E. W. (1952). A Chart method for simplifying truth functions. *Proceedings of the 1952 ACM National Meeting (Pittsburgh)*, 127–133. https://doi.org/10.1145/609784.609801

Wakerly, J. F. (2018). *Digital design: Principles and practices* (5th ed.). Pearson.

Exercises

1. Show the truth tables for the 3-input AND and OR functions.
2. Show the truth tables for the 4-input AND and OR functions.
3. Verify the commutativity property of the 3-input AND and OR functions.
4. Verify the distributive property for
    a. $a \cdot (b + c + d)$
    b. $a + (b \cdot c \cdot d)$.
5. Verify associativity for
    a. $(a \cdot b) \cdot (c \cdot d) = a \cdot (b \cdot c) \cdot d$
    b. $(a + b) + (c + d) = a + (b + c) + d$
6. Use De Morgan's law to find the function equivalent to $(a + b')'$.
7. Use De Morgan's law to find the function equivalent to $(a \cdot b')'$.
8. Show De Morgan's laws for three variables and the truth tables to verify them.
9. Create the truth table for the function $a + (a' \cdot b) + (a' \cdot c)$.
10. Specify the function in Problem 9 as a sum of products and as a product of sums.
11. Create the truth table for the function $a + (a \cdot b') + (b' \cdot c) + (a' \cdot c)$.
12. Specify the function in Problem 11 as a sum of products and as a product of sums.
13. Create the truth table for the function $a \cdot (b + c') + (a \cdot b' \cdot c)$.
14. Specify the function in Problem 13 as a sum of products and as a product of sums.
15. Minimize the function of Problem 9 using Karnaugh maps.
16. Minimize the function of Problem 9 using the Quine-McCluskey algorithm.
17. Minimize the function of Problem 11 using Karnaugh maps.
18. Minimize the function of Problem 11 using the Quine-McCluskey algorithm.
19. Minimize the function of Problem 13 using Karnaugh maps.
20. Minimize the function of Problem 13 using the Quine-McCluskey algorithm.

21. Show the truth table corresponding to the following Karnaugh map. Find all prime implicants and essential prime implicants, and give the minimal function for these values.

a\bc	00	01	11	10
0	1	0	1	0
1	1	1	1	1

22. For the truth table you created for Problem 21, use the Quine-McCluskey algorithm to determine the minimal function.

23. Show the truth table corresponding to the following Karnaugh map. Find all prime implicants and essential prime implicants, and give the minimal function for these values.

I3I2\I1I0	00	01	11	10
00	0	1	0	1
01	1	1	0	0
11	1	1	1	1
10	0	1	0	0

24. For the truth table you created for Problem 23, use the Quine-McCluskey algorithm to determine the minimal function.

25. Show the truth table corresponding to the following Karnaugh map. Find all prime implicants and essential prime implicants, and give the minimal function for these values.

I3I2\I1I0	00	01	11	10
00	1	0	1	1
01	0	1	0	1
11	X	X	X	X
10	1	0	X	X

26. For the truth table you created for Problem 25, use the Quine-McCluskey algorithm to determine the minimal function.

27. Use the Quine-McCluskey algorithm to determine the minimum function for

$\sum m(3,\ 11.\ 12.\ 13.\ 14,\ 15,\ 19,\ 27)$. The function has five inputs.

PART II

Combinatorial Logic

Chapter 3

Digital Logic Fundamentals

Chapter 3: Digital Logic Fundamentals

In the previous chapter, we introduced the fundamentals of Boolean algebra and functions. In this chapter, we'll look at digital logic, circuitry developed to realize these functions. Digital logic can be used to implement any Boolean function ranging from the simplest functions to complex computer systems.

We start by introducing logic gates that realize the fundamental functions described in the previous chapter, including AND, OR, XOR, and NOT. Then we will examine how to use these gates to realize more complex functions using the sum of products and product of sums introduced in Chapter 2. We will also look at inverse functions, because sometimes it is easier to design a circuit that does the exact opposite of what you want it to do and then invert its output. Finally, we'll look at some real-world constraints that go beyond the underlying logic that must be accounted for when designing digital logic circuits.

3.1 Basic Logic Gates

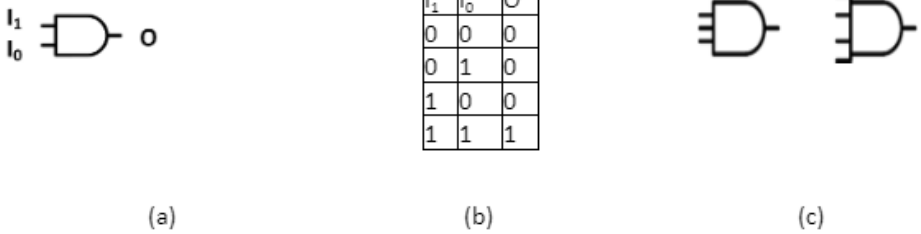
In Chapter 2, we examined several fundamental Boolean operations. In this chapter, we'll move from the algebraic specification of a Boolean function to designing a circuit to realize that function.

Engineers developed integrated circuit chips that can realize these functions. For the AND function, two or more pins on the chip are connected to the inputs of the AND function, which implements the AND function inside the chip. Circuit designers connect these pins to external inputs or other components in their circuits to send data to these inputs. The output of the function is connected to another pin on the chip. Circuit designers can send this value out to other parts of the circuit or send it out from the circuit as needed.

The part of the chip that performs the function is called a logic gate. Just as there are different logic functions, there are different logic gates to realize these functions. Each gate has its own logic symbol that is used in circuit diagrams. Next we'll look at some of the most fundamental logic gates.

Let's start with the AND gate. As its name implies, it implements the AND function: its output is 1 only when all inputs are 1. If any input is 0, its output is 0. In theory, an AND gate could have any number of inputs, but in practice an AND gate has a relatively small number of inputs, usually 2, 3, or 4. There are some physical limitations inherent to digital circuits that must be accounted for beyond just the Boolean logic. We will discuss these more at the end of this chapter and the end of Chapter 4.

Figure 3.1 (a) shows the standard schematic symbol for a 2-input AND gate. Its truth table is shown in Figure 3.1 (b). For AND gates with more than two inputs, we use the same gate symbol, just with more input lines. Three- and four-input AND gates are shown in Figure 3.1 (c). The development of their truth tables are left as exercises for the reader.



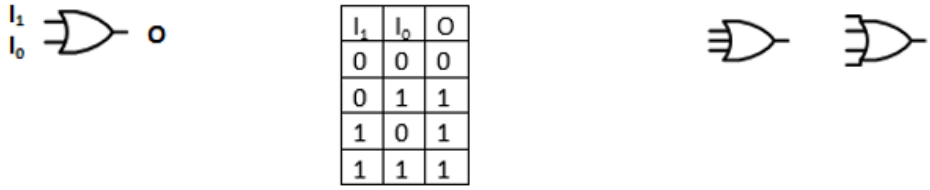
I_1	I_0	O
0	0	0
0	1	0
1	0	0
1	1	1

(a) (b) (c)

Figure 3.1: AND gates: (a) 2-input AND gate; (b) Truth table; (c) 3- and 4-input AND gates

WATCH ANIMATED FIGURE 3.1

The OR gate, like the AND gate, has two or more inputs and one output. The output is 1 if any input is 1, and is 0 only if all inputs are set to 0. The 2-input OR gate is shown in Figure 3.2 (a). Its truth table is given in Figure 3.2 (b), and 3- and 4-input OR gates are shown in Figure 3.2 (c).



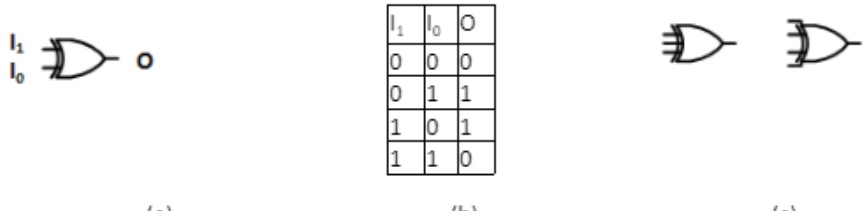
I_1	I_0	O
0	0	0
0	1	1
1	0	1
1	1	1

(a) (b) (c)

Figure 3.2: OR gates: (a) 2-input OR gate; (b) Truth table; (c) 3- and 4-input OR gates

WATCH ANIMATED FIGURE 3.2

The exclusive-or, XOR, gate can have two or more inputs, though it usually has only two. The output is 1 if an odd number of inputs are equal to 1, and it is 0 if the gate has an even number of inputs set to 1. Figure 3.3 shows the symbol for the 2-input XOR gate, its truth table, and the 3- and 4-input XOR gates.



I_1	I_0	O
0	0	0
0	1	1
1	0	1
1	1	0

(a) (b) (c)

Figure 3.3: XOR gates: (a) 2-input XOR gate; (b) Truth table; (c) 3- and 4-input XOR gates

WATCH ANIMATED FIGURE 3.3

Unlike the previous gates, the NOT gate can have only one input. Its output is the complement of the input. That is, when the input is 0 the output is 1, and when the input is 1 the output is 0. Figure 3.4 shows the symbol for the NOT gate and its truth table.

I	O
0	1
1	0

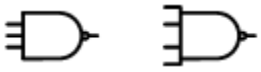
(a) (b)

Figure 3.4: NOT gate: (a) Symbol; (b) Truth table

WATCH ANIMATED FIGURE 3.4

As we saw in the previous chapter, there are also complementary functions for the AND, OR, and XOR functions. The NAND function is 0 when all its inputs are 1 and 1 if any input is 0. The symbol for the 2-input NAND gate, shown in Figure 3.5 (a), is simply the symbol for the 2-input AND gate with a circle at its output, which indicates that it is followed by an inverter. (Actually, internally these gates are designed the other way around; the AND gate is a NAND gate followed by an inverter.) Figure 3.5 (b) and (c) show the truth table for the 2-input NAND gate and the symbol for the 3- and 4-input NAND gates, respectively.

I_1	I_0	O
0	0	1
0	1	1
1	0	1
1	1	0



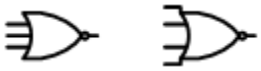
(a) (b) (c)

Figure 3.5: NAND gates: (a) 2-input NAND gate; (b) Truth table; (c) 3- and 4-input NAND gates

WATCH ANIMATED FIGURE 3.5

Continuing on, Figure 3.6 shows the symbols and truth table for the NOR gate, and Figure 3.7 shows this information for the exclusive-NOR (XNOR, not NXOR) gate. Note that the output of the 2-input XNOR gate is 1 when its two inputs are equal. For this reason, it is sometimes referred to as the equivalence function.

I_1	I_0	O
0	0	1
0	1	0
1	0	0
1	1	0

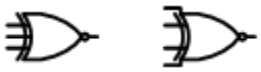


(a) (b) (c)

Figure 3.6: NOR gates: (a) 2-input NOR gate; (b) Truth table; (c) 3- and 4-input NOR gates

WATCH ANIMATED FIGURE 3.6

I_1	I_0	O
0	0	1
0	1	0
1	0	0
1	1	1



(a) (b) (c)

Figure 3.7: XNOR gates: (a) 2-input XNOR gate; (b) Truth table; (c) 3- and 4-input XNOR gates

WATCH ANIMATED FIGURE 3.7

3.2 Implementing Functions

Now that we have seen the fundamental logic gates, we'll spend some time in this section looking at how to use these logic gates to realize logic functions. Using some of the sample functions we examined in Chapter 2, we first implement functions using the least number of gates possible. Then we will design circuits to implement functions using sum of products and product of sums.

3.2.1 Minimizing Logic

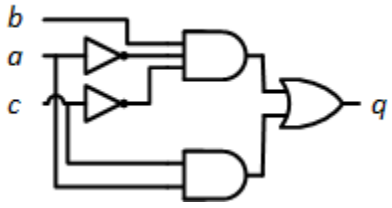
The first steps when designing a digital logic circuit are to determine the value of the output for all possible input values and to create a Boolean equation that generates these outputs. Let's start with an equation we used in Chapter 2, $q = a'bc' + ac$. For your reference, its truth table is shown in Figure 3.8 (a).

a	*b*	*c*	*q*
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

(a)

a	*b*	*c*	*a'*	*c'*	*a'bc'*	*ac*	*q*
0	0	0	1	1	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	1	1	0	1
0	1	1	1	0	0	0	0
1	0	0	0	1	0	0	0
1	0	1	0	0	0	1	1
1	1	0	0	1	0	0	0
1	1	1	0	0	0	1	1

(b)



(c)

Figure 3.8: Implementing the function *q* = *a'bc'* + *ac*: (a) Original truth table; (b) Expanded truth table; (c) Digital logic circuit to implement this function

WATCH ANIMATED FIGURE 3.8

Just by looking at the equation for the function, we can see several of the gates that we will need in our final circuit:

- Two NOT gates to generate *a'* and *c'*
- One 3-input AND gate to generate *a'bc'*
- One 2-input AND gate to generate *ac*
- One 2-input OR gate to combine *a'bc'* and *ac* to produce *q*

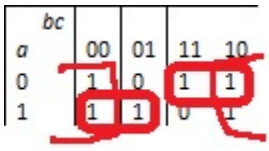
The expanded truth table in Figure 3.8 (b) shows the values of *a'*, *c'*, *a'bc'*, *ac*, and *q* for all values of *a*, *b*, and *c*.

Finally, we implement all of this in digital logic. The sequence of gates is dependent on the availability of their inputs. For example, the OR gate cannot combine *a'bc'* and *ac* until we have generated these functions. In addition, we cannot create *a'bc'*, the output of the 3-input AND gate, until we have generated *a'* and *c'*. Note that *b* is an input, so it can be connected directly to one of the inputs of this gate. Similarly, the 2-input AND gate receives inputs *a* and *c* directly. The complete circuit is shown in Figure 3.8 (c), animated for all possible input values.

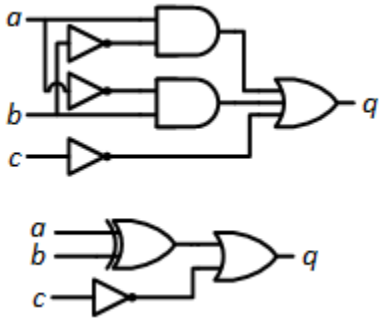
In this example, we were given the final equation for output *q*. Sometimes, however, we may have only the output specified for some or all input values. If this is the case, we will need to develop an equation that generates these outputs and then design a circuit to realize that equation. Consider another function from Chapter 2 with the truth table shown in Figure 3.9 (a)

a	*b*	*c*	*q*
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

(a)



(b)



(c)

Figure 3.9: Another function: (a) Truth table; (b) Karnaugh map; (c) Implementation of $q = ab' + a'b + c'$ and $q = (a \oplus b) + c'$

In Chapter 2, we used a Karnaugh map to generate the function $q = ab' + a'b + c'$, as shown in Figure 3.9 (b). We can create a circuit to realize this function using the same procedure we used for the previous function. We will need three NOT gates to generate a', b', and c', two 2-input AND gates to generate ab' and $a'b$, and one 3-input OR gate to combine ab', $a'b$, and c'. This is shown in the first circuit in Figure 3.9 (c).

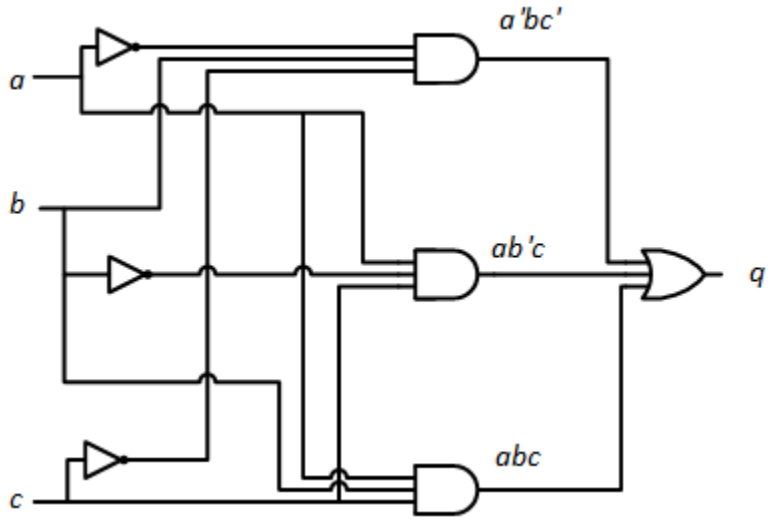
Karnaugh maps are very useful for creating AND/OR-based functions, but sometimes we can reduce the number of gates by going beyond these maps. In this example, $ab' + a'b$ is the same as $a \oplus b$. We could substitute this into our equation, giving us $q = (a \oplus b) + c'$. Then we could replace the two AND gates and the two NOT gates that generate a' and b' with a single XOR gate. In addition, the OR gate would only need two inputs instead of three. A circuit to realize this revised function is also shown in Figure 3.9 (c).

3.2.2 Implementing Functions using Sum of Products

Recall from Chapter 2 that each row of the truth table, or each square in the Karnaugh map, corresponds to one minterm. For each variable, we AND together either the variable or its complement. We AND the variable if its value in the row is 1, or we AND its complement if its value is 0. For example, the minterm for the row with $a = 1$, $b = 0$, and $c = 1$, is $ab'c$. Figure 3.10 (a) shows the truth table for a function we looked at previously, $q = a'bc' + ac$.

a	*b*	*c*	*q*	Minterm
0	0	0	0	*a'b'c'*
0	0	1	1	*a'b'c*
0	1	0	0	*a'bc'*
0	1	1	0	*a'bc*
1	0	0	0	*ab'c'*
1	0	1	1	*ab'c*
1	1	0	0	*abc'*
1	1	1	1	*abc*

(a)



(b)

Figure 3.10: Implementing $q = a'bc' + ac$ using sum of products: (a) Truth table; (b) Circuit to realize *q* using minterms

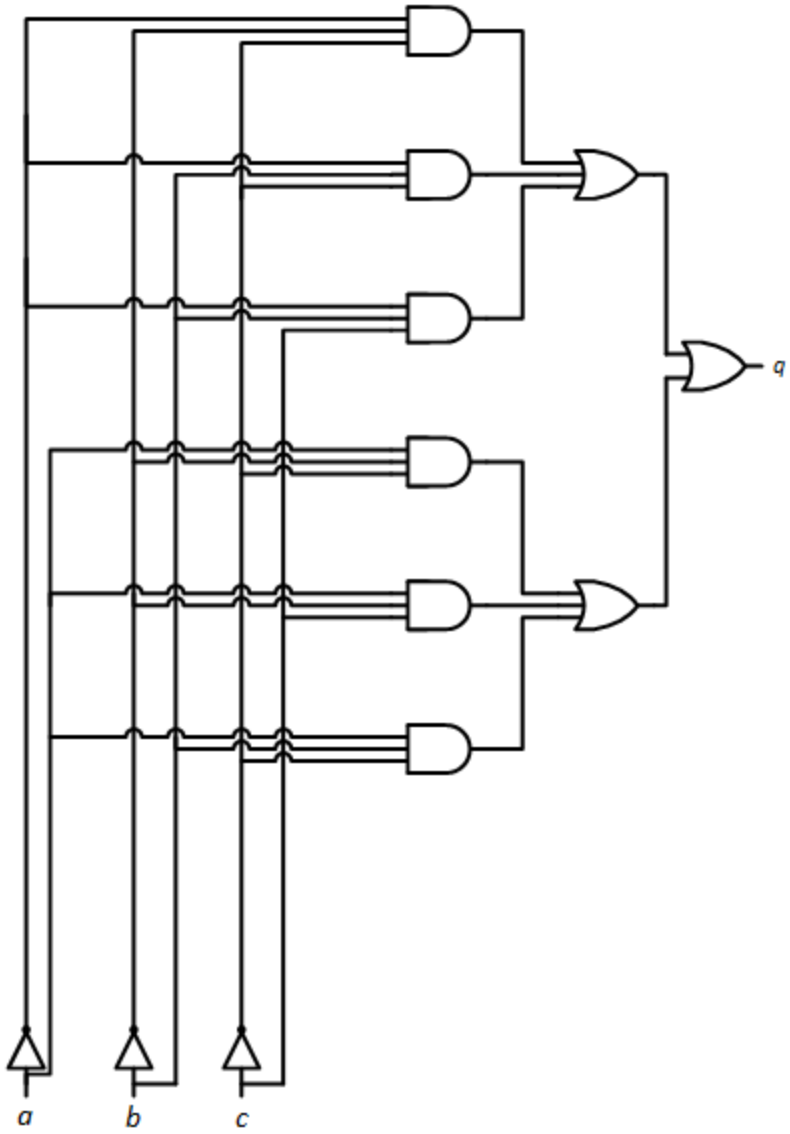
WATCH ANIMATED FIGURE 3.10

To create a sum of products function, we first generate each minterm that sets *q* to 1. In this example, we need three NOT gates to output *a'*, *b'*, and *c'*, three 3-input AND gates to generate *a'bc'*, *ab'c*, and *abc*, and finally a 3-input OR gate to combine these three minterms and output a 1 if any of them is 1. A circuit to realize this implementation is shown in Figure 3.10 (b).

Comparing the two circuits to generate *q*, shown in Figures 3.8 (c) and 3.10 (b), it is clear that the sum of products circuit is larger than the original circuit. So, why should we bother with sum of products? There are several reasons, some of which we'll discuss now and others that we will examine in later chapters. If your goal is to minimize the digital logic needed to realize a function, a very reasonable and frequently used goal, the sum of products can serve as a useful first step in developing your final function. Karnaugh maps are quite useful when you have a limited number of inputs, typically 4 or less, but other methods are used with larger numbers of inputs. These methods often start with minterms and reduce their equations (and thus the circuits that implement these equations) from there. Also, designers have implemented some standard, more complex, digital components that can be used to realize functions. We will introduce some of these components in the next chapter and how they can be used to realize functions in Chapter 5.

Now let's look at the other example from the previous subsection. Figure 3.11 (a) shows the truth table from Figure 3.10 (a) with the addition of the minterms associated with each row. There are six rows that set q to 1, so our sum of products implementation would use six 3-input AND gates to generate these minterms, one 6-input OR gate to combine the minterms, and three NOT gates to produce *a'*, *b'*, and *c'*. The circuit is shown in Figure 3.11 (b).

a	*b*	*c*	*q*	Minterm
0	0	0	1	*a'b'c'*
0	0	1	0	*a'b'c*
0	1	0	1	*a'bc'*
0	1	1	1	*a'bc*
1	0	0	1	*ab'c'*
1	0	1	1	*ab'c*
1	1	0	1	*abc'*
1	1	1	0	*abc*



(a) (b)

Figure 3.11: Implementing $q = a'b'c' + a'bc' + a'bc + ab'c' + ab'c + abc'$ using sum of products: (a) Truth table; (b) Circuit to realize *q* using minterms

There are a couple of things we can generalize from these two examples. First, each circuit has a column of AND gates followed by a single OR gate. For this reason, a sum of products is said to be implemented by an AND-OR circuit. Each AND gate generates one minterm and the OR gate combines them to produce the final output function. This leads to a second generalization: the number of rows in the truth table that set the output to 1 is also the number of AND gates in the first stage, and also the number of inputs to the OR gate. This is true because each row corresponds to one minterm; each row that outputs a 1 is one minterm

we must check. Since each AND gate outputs the value of one minterm, the two numbers must be equal.

One thing that is obvious about the circuit for this function is that it is quite large. We saw much simpler circuits for this function in Figure 3.9 (c). In the next subsection, we'll look at another design methodology that can also produce smaller circuits for some functions.

3.2.3 Implementing Functions using Product of Sums

In the previous example, six of the eight possible minterms set the output to 1, resulting in a circuit that requires a lot of digital logic. Here's another way of approaching this function. Instead of saying the function is 1 if one of the minterms that sets q to 1 is active, we could say the function is 1 if none of the minterms that sets q to 0 is active. That is, if it isn't set to 0, then it must be set to 1.

For our previous function, there are two minterms that set q to 0, $a'b'c$ and abc. If $a = 0$, $b = 0$, and $c = 1$, then minterm $a'b'c$ is active (because $a' = 1$, $b' = 1$, and $c = 1$). Minterm abc is active when $a = 1$, $b = 1$, and $c = 1$. For any other values of a, b, and c, neither of these minterms is active. It doesn't matter which of the other minterms is active; they all set q to 1.

In terms of our function, $q = 1$ if $a'b'c = 0$ and $abc = 0$. In Boolean algebra, we can express this as

$$q = (a'b'c + abc)'$$

Applying De Morgan's Laws, specifically $X + Y = (X'Y')'$, or $(X + Y)' = X'Y'$, with $X = a'b'c$ and $Y = abc$, this becomes

$$q = (a'b'c)'(abc)'$$

We can apply De Morgan's Laws to each individual term, this time using the form $XYZ = (X' + Y' + Z')'$, or $(XYZ)' = X' + Y' + Z'$. For the first term, $a'b'c$, we set $X = a'$, $Y = b'$, and $Z = c$, and this becomes

$$(a'b'c)' = (a + b + c')$$

(Remember that $(a')' = a$ and $(b')' = b$.) Following the same procedure, $(abc)' = (a' + b' + c')$, and our function becomes

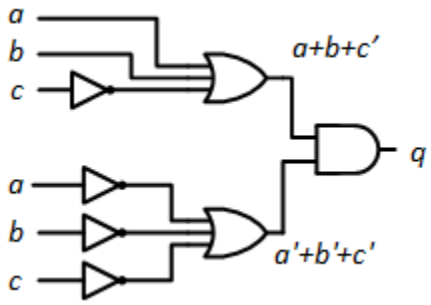
$$q = (a + b + c')(a' + b' + c')$$

The terms $(a + b + c')$ and $(a' + b' + c')$ are called **maxterms**. Each maxterm is always 1 except for one unique combination of input values. The first maxterm, $(a + b + c')$, is one for all possible values of a, b, and c except $a = 0$, $b = 0$, and $c = 1$. The second term, $(a' + b' + c')$, is only zero when $a = 1$, $b = 1$, and $c = 1$. If either of these combinations of values is active, one of the two terms will be 0, and $q = 0 \cdot 1$ (or $1 \cdot 0$) $= 0$. For all other values of a, b, and c, both terms are 1 and $q = 1 \cdot 1 = 1$.

From the structure of the equations for q, we can see that we will need two 3-input OR gates to generate our maxterms and one 2-input AND gate to combine them, as well as three NOT gates to produce a', b', and c'. Figure 3.12 shows the expanded truth table and a circuit to implement the function as a product of sums.

a	b	c	$(a+b+c')$	$(a'+b'+c')$	$(a+b+c')(a'+b'+c)$
0	0	0	1	1	1
0	0	1	0	1	0
0	1	0	1	1	1
0	1	1	1	1	1
1	0	0	1	1	1
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	1	0	0

(a)



(b)

Figure 3.12: Implementing $q = (a + b + c')(a' + b' + c')$ using Product of Sums: (a) Expanded truth table; (b) Circuit to realize q using maxterms

WATCH ANIMATED FIGURE 3.12

Just as the minterm implementation produced a two-stage circuit, the maxterm circuit also has two stages. This time, however, the circuit has OR gates in the first stage and an AND gate in the second stage. This is referred to as an OR-AND circuit.

3.3 Inverse Functions

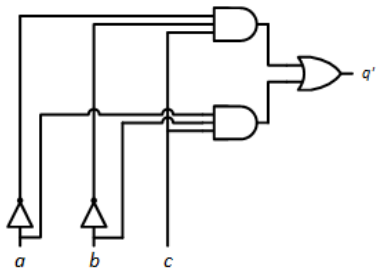
Sometimes it is easier to implement the inverse of a function rather than the function itself, and then just invert it to realize the desired function. In this section, we'll examine how to do this using both minterms and maxterms.

3.3.1 Inverse Functions Using Minterms

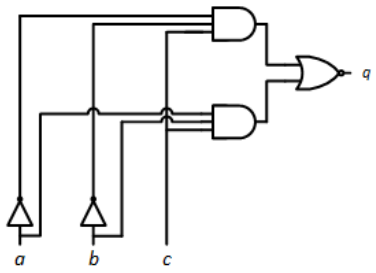
To illustrate how this can be done using minterms, let's start with the function given in Figure 3.11. This table, repeated and expanded in Figure 3.13 (a), has six of its eight minterms set q to 1. We have expanded the table to add a column with the value of q', which is only set to 1 by two minterms. First we will create a circuit to realize q'; then we will invert its output to generate q.

a	*b*	*c*	*q*	*q*'	Minterm
0	0	0	1	0	*a'b'c'*
0	0	1	0	1	*a'b'c*
0	1	0	1	0	*a'bc'*
0	1	1	1	0	*a'bc*
1	0	0	1	0	*ab'c'*
1	0	1	1	0	*ab'c*
1	1	0	1	0	*abc'*
1	1	1	0	1	*abc*

(a)



(b)



(c)

Figure 3.13: Implementing $q = (a'b'c + abc)'$ using Sum of Products for inverse functions: (a) Truth table; (b) Circuit to realize q'; (c) Inverting the output to realize q

WATCH ANIMATED FIGURE 3.13.b WATCH ANIMATED FIGURE 3.13.c

Using only minterms, q' is only equal to 1 when either $a'b'c$ or abc is 1. Just as we did previously, we can use AND gates to generate the minterms and an OR gate to combine them to produce the overall function. This circuit is shown in Figure 3.13 (b).

But we're not quite done. Remember that this circuit generates the inverse of our function, q', and we still need to invert that value to generate our final function, q. We could accomplish this by placing a NOT gate at the output of the OR gate. A simpler method is to replace the OR gate with a NOR gate. The final circuit is shown in Figure 3.13 (c).

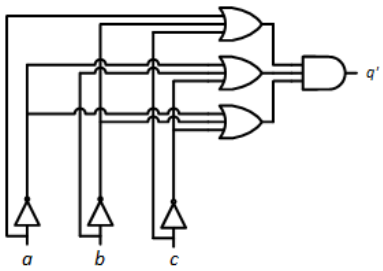
In general, realizing inverse functions using minterms can reduce the hardware needed when the number of minterms is large. In the truth table, this occurs when more than half the entries set the function to 1.

3.3.2 Inverse Functions Using Maxterms

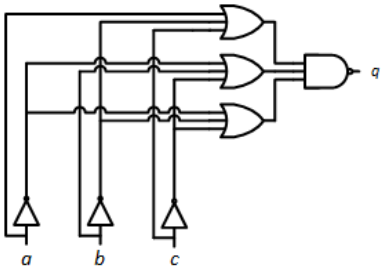
Just as inverse functions can be created more efficiently when the number of minterms is large, inverse functions can also be created using less hardware when the number of maxterms is large. Let's return to another function from the previous section, $q = a'bc' + ac$. Its truth table, with corresponding maxterms, is shown in Figure 3.14 (a).

a	*b*	*c*	*q*	*q'*	Maxterm
0	0	0	0	1	*a*+*b*+*c*
0	0	1	0	1	*a*+*b*+*c'*
0	1	0	1	0	*a*+*b'*+*c*
0	1	1	0	1	*a*+*b'*+*c'*
1	0	0	0	1	*a'*+*b*+*c*
1	0	1	1	0	*a'*+*b*+*c'*
1	1	0	0	1	*a'*+*b'*+*c*
1	1	1	1	0	*a'*+*b'*+*c'*

(a)



(b)



(c)

Figure 3.14: Implementing $q = ((a+b'+c)(a'+b+c')(a'+b'+c'))'$ using product of sums for inverse functions: (a) Truth table; (b) Circuit to realize q'; (c) Inverting the output to realize q

WATCH ANIMATED FIGURE 3.14.b WATCH ANIMATED FIGURE 3.14.c

If we wanted to realize *q* directly as a product of sums, we would generate each minterm that sets *q* to 0 and logically AND them together. For this function, we would need five 3-input OR gates and one 5-input AND gate. In this case, generating the inverse function and then inverting its output produces a simpler design.

Consider how we could generate *q*' as a product of sums. Function *q*' has only three maxterms equal to zero, so our circuit would need only three 3-input NOR gates and one 3-input AND gate. This circuit is shown in Figure 3.14 (b).

Finally, since this circuit generates *q*', not *q*, we need to invert its output. We can do this by replacing the AND gate with a NAND gate. The final circuit is shown in Figure 3.14 (c).

Just as inverse functions can be used to reduce hardware when the number of minterms is large, they can also reduce hardware when the number of maxterms is large. Unlike minterms, however, this is the case when more than half the entries in the truth table set the function to 0.

3.3.3 What Inverse Functions Really Are

I haven't told you the entire story behind inverse functions yet, but I'll do so now. Inverse functions, as presented so far in this section, are not generally called inverse functions. Remember from Chapter 2 that each set of inputs that sets a function output to 1 is a minterm, and each that sets the function to 0 is a maxterm. In reality, the inverse function of minterms is just using the maxterms to set the function, and the inverse function of maxterms is just using the minterms to set the function. Both set it equally well; it's just a matter of selecting which method gives you the simplest circuits for the function you wish to realize.

3.4 Beyond the Logic

In theory, practice and theory are the same. In practice they are not.

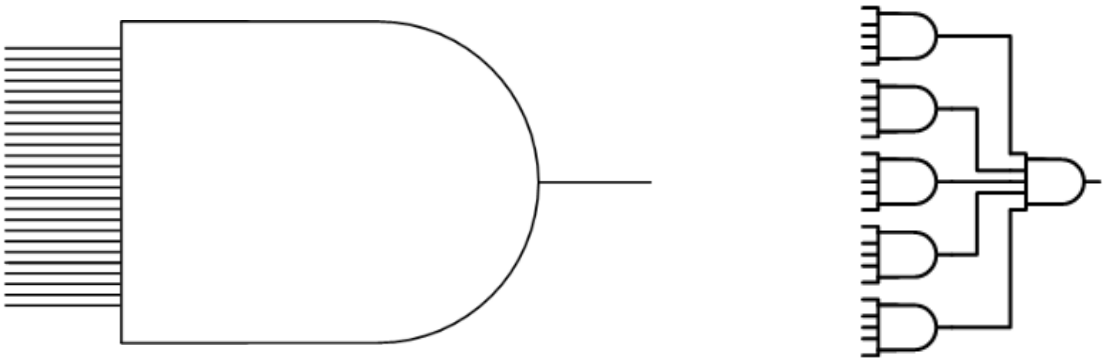
This somewhat famous quote has been attributed to a number of people, including Albert Einstein, Richard Feynman, and Yogi Berra. Here, this quote is applicable to digital logic design. Just because some function is theoretically possible doesn't mean that it can be implemented directly as shown in a Boolean equation. In the real world, there are limitations imposed by constraints such as the amount of current a logic gate can output. Logic gates are quite fast, but not instantaneous. They do take some small amount of time to output their results.

In this section, we will examine some of these constraints: fan-in, fan-out, and propagation delays. We will also introduce the buffer, a component that performs no logical function but is used to overcome constraints in certain cases.

3.4.1 Fan-in

The fan-in of a logic gate is the maximum number of inputs it can support. If you are using TTL or CMOS chips, this has already been decided for you. The chip has some logic gates built into it, and each gate has a fixed number of inputs. When creating new designs, such as a new chip however, this may not be the case.

Consider, for example, an AND function that has 25 inputs. In theory, you could realize that function using a single 25-input AND gate, as shown in Figure 3.15 (a). In practice, however, your AND gate technology may have a fan-in of a much lower number, and you would have to find another way to implement this function.



(a) (b)

Figure 3.15: A 25-input AND function: (a) Using a single AND gate; (b) Using multiple gates with a maximum fan-in of 5

To realize this function, let's assume that your AND gates have a maximum fan-in of five. We could then divide our 25 inputs into five groups of five inputs each. Each group of five inputs could be input to a 5-input AND gate. The output of each gate would be 1 only when its five inputs are 1. Then, the output of each of these AND gates would be input to another 5-input

AND gate. The output of this gate would only be 1 when its five inputs are 1, which only occurs when all 25 inputs to the other AND gates are 1. This circuit is shown in Figure 3.15 (b).

3.4.2 Fan-out

Fan-out is comparable to fan-in, but on the other side of a logic gate. Succinctly put, the fan-out of a logic gate is the maximum number of inputs of other gates it can be connected to. The gate has a limit on how much current it can output. When it sends its output to more than one other component's input, this current is split up among those components. When the current becomes too small, these components may not recognize the input value correctly. A typical TTL component has a fan-out of around 5; it can send its output to up to five other component inputs.

Consider the circuit shown in Figure 3.16. The OR gate sends its output to 25 different AND gates. Although the logic may be correct in theory, in practice this will not work. The maximum fan-out of the OR gate is much less than the number used in this circuit.

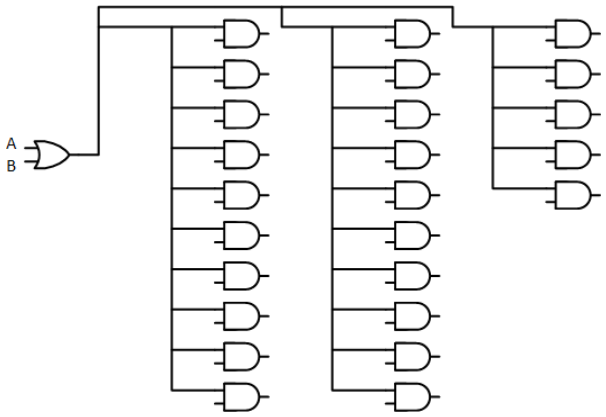


Figure 3.16: A digital circuit that does not meet fan-out requirements

Now let's look at a way to get around fan-out limitations.

3.4.3 Buffers

To boost the output signal, engineers developed a component called a **buffer**. It inputs a signal at a given logic value and outputs a signal at the same logic value. In terms of Boolean logic, a buffer looks like the most useless component imaginable. All it does is output whatever is input to it. A wire would perform the same logical function. Yet, it was invented and is commonly used in digital circuits. Its truth table and logic symbol are shown in Figure 3.17.

Figure 3.17: Buffer logic symbol and truth table

WATCH ANIMATED FIGURE 3.17

Although its logic output has the same logical value as its input, there is an important difference. Its output current is at its maximum value, whereas the input current may be lower if the logic gate that supplies the input also sends it to other components.

To illustrate this, consider the 25-input AND gate in Figure 3.16. We can use buffers to realize this function while still meeting fan-out requirements. One way to do this is shown in Figure 3.18. Here, the OR gate sends its output to five buffers. Each buffer, in turn, sends its output to five AND gates. Since no component sends its output to more than five other inputs, this circuit meets fan-out requirements, assuming each component has a maximum fan-out of at least five.

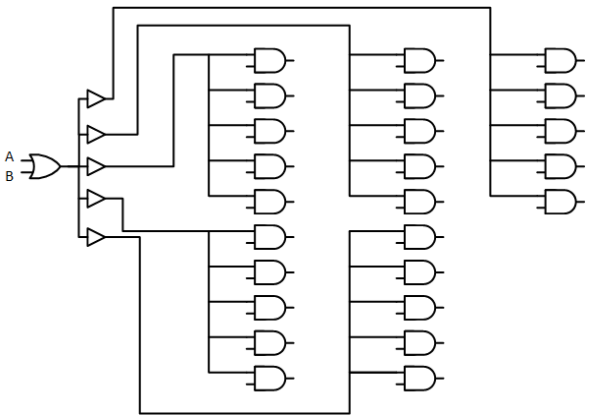


Figure 3.18: Digital circuit that meets fan-out requirements

3.4.4 Propagation Delay

Prior to this section, we have been looking at logic gates in an idealized way. When we change the inputs to a logic circuit, the outputs change to their new values instantaneously. In practice, however, this is not the case. There is a very small, but non-zero amount of time needed for a gate to change its output from 0 to 1 or from 1 to 0. This is called the **propagation delay**. In many circuits, this delay is quite tolerable. Even with the propagation delay, the circuit functions as desired. In some cases, however, this can cause problems.

Consider, for example, the circuit shown in Figure 3.19 (a). When *ToBe* = 1, the output should be 1 because the upper input to the OR gate is 1. When it is 0, the output of the inverter is 1, so the OR gate should again output a value of 1. In theory, *Q* should always be 1, as shown in the truth table in Figure 3.19 (b).

ToBe	*ToBe'*	Q
0	1	0
1	0	1

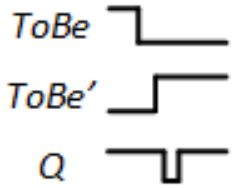
(a)

(b)

Figure 3.19: *ToBe* + *ToBe'*: (a) Circuit diagram; (b) Truth table

Now let's look at a more practical example, the same circuit with non-zero propagation delays. For simplicity, assume that the NOT gate has a propagation delay of 5 nanoseconds (ns), that is, it may not generate the correct value until 5 ns after the input changes. Also assume the OR gate has a propagation delay of 8 ns. As shown in the timing diagram in Figure 3.20, when *ToBe* changes from 1 to 0, it may take 5 ns for the output of the NOT gate to change from 0 to 1. During this time, both inputs to the OR gate will be 0, causing it to output a value of 0. As with the NOT gate, this output is also delayed, this time by 8 ns. After an additional 5 ns, the OR gate has adjusted its output value based on the new value of *ToBe'*. This entire sequence is illustrated in the animation of the circuit and timing diagram in Figure 3.20.

(a)



(b)

Figure 3.20: *ToBe* + *ToBe'*: (a) Circuit diagram; (b) Timing diagram with propagation delays

WATCH ANIMATED FIGURE 3.20

Finally, there's one other thing to consider. The delays within a gate may or may not be symmetric. That is, the propagation delay when an output changes from 0 to 1 may not be the same as when the output changes from 1 to 0. For TTL and CMOS chips, you can refer to the chip's datasheet to find these values, denoted as t_{PLH} (0 to 1) and t_{PHL} (1 to 0).

3.5 Summary

In this chapter, we introduced digital logic components that realize fundamental Boolean functions. The AND gate outputs a logic 1 only if all its inputs are 1. The OR gate, on the other hand, outputs a logic 1 if *any* of its inputs is 1. The XOR, or exclusive OR, gate outputs a 1 if it has an odd number of inputs equal to 1. The NOT gate, unlike these other gates, has only one input. It sets its output to the opposite of its input value. The NAND, NOR, and XNOR gates perform exactly the opposite functions of the AND, OR, and XOR gates, respectively.

Next, this chapter examined how to realize more complex functions with these gates using sum of products and product of sums methods, as well as inverse functions.

Beyond the Boolean algebra used to specify functions, there are some physical limitations to digital logic that must be taken into account when designing a digital circuit. This chapter examined fan-in and fan-out, and the use of buffers to overcome some of these limitations. We also introduced propagation delay, which can inject timing delays into circuits that sometimes produce unintended consequences.

Some functions are used frequently in digital logic design. Engineers have developed integrated circuit chips to realize these functions; using them in digital circuits reduces the amount of power used, wiring needed in the circuit, and other parameters. We'll introduce some of these functions in the next chapter.

Bibliography

Hayes, J. P. (1993). *Introduction to digital logic design*. Addison-Wesley.

Mano, M. M., & Ciletti, M. (2017). *Digital design: with an introduction to the Verilog HDL, VHDL, and SystemVerilog* (6th ed.). Pearson.

Mano, M. M., Kime, C., & Martin, T. (2015). *Logic & computer design fundamentals* (5th ed.). Pearson.

Nelson, V., Nagle, H., Carroll, B., & Irwin, D. (1995). *Digital logic circuit analysis and design*. Pearson.

Roth, J. C. H., Kinney, L. L., & John, E.B. (2020). *Fundamentals of logic design enhanced edition* (7th ed.). Cengage Learning.

Texas Instruments, Inc. (1981). *The TTL Data Book for Design Engineers*. (2nd ed.). Texas Instruments.

Wakerly, J. F. (2018). *Digital design: Principles and practices* (5th ed.). Pearson.

Data sheets for many semiconductor chips are available on numerous websites, including www.ti.com, www.smcelectronics.com, and the sites for most vendors that sell these chips.

Exercises

1. Show the truth tables for the 3- and 4-input AND gates.
2. Show the truth tables for the 3- and 4-input OR gates.
3. Show the truth tables for the 3- and 4-input XOR gates.
4. Show the truth tables for the 3- and 4-input NAND gates.
5. Show the truth tables for the 3- and 4-input NOR gates.
6. Show the truth tables for the 3- and 4-input XNOR gates.
7. Develop a minimal function for the truth table shown below and design a circuit to realize this function.

a	*b*	*c*	*q*
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

8. For the truth table in Problem 7, develop a function consisting solely of minterms and design a circuit to realize this function.
9. For the truth table in Problem 7, develop a function consisting solely of maxterms and design a circuit to realize this function.
10. For the truth table in Problem 7, develop a function to realize q', the inverse of q, and design a circuit to realize q based on this inverse function.

11. Develop a minimal function for the truth table shown below and design a circuit to realize this function.

a	*b*	*c*	*q*
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

12. For the truth table in Problem 11, develop a function consisting solely of minterms and design a circuit to realize this function.

13. For the truth table in Problem 11, develop a function consisting solely of maxterms and design a circuit to realize this function.

14. For the truth table in Problem 11, develop a function to realize q', the inverse of q, and design a circuit to realize q based on this inverse function.

15. Redesign the circuit that implements the 25-input AND function in Figure 3.15 using gates with a maximum fan-in of 3.

16. Redesign the circuit that implements the 25-output function in Figure 3.16 using gates with a maximum fan-out of 3.

17. Figures 3.9, 3.11, 3.12, and 3.14 show four different circuits to realize the same function. For the maximum propagation delays for individual gates shown below, what is the maximum propagation delay for each circuit?

Gate	*TTL IC #*	*Maximum propagation delay*
2-input AND gate	74LS08	20ns
3-input AND gate	74LS11	20ns
3-input NAND gate	74LS10	15ns
2-input OR gate	74LS32	22ns
3-input OR gate	74LS32 (2 gates)	44ns
2-input XOR gate	74LS86	30ns
NOT gate	74LS04	15ns

Chapter 4

More Complex Components

Chapter 4: More Complex Components

When designing digital circuits, there are certain functions that tend to occur frequently. Semiconductor manufacturers have designed dedicated chips that implement these functions, so designers can incorporate them into their circuits directly. This takes up less space on the circuit board, requires less wiring, and uses less power than if the designer had to recreate the function using the fundamental digital logic gates introduced in the previous chapter. Similarly, CAD systems have libraries with these components and functions that designers can use in their designs. In either case, the chips and library components improve the efficiency of the design process. Since the chips and components have been thoroughly tested and verified, they can also increase the reliability of the overall design.

In this chapter, we will examine several of these functions: decoders, encoders, multiplexers, demultiplexers, comparators, and adders. In addition, we will introduce buffers. They do not perform functions like the other components, but they are needed in some circuits to overcome the physical limitations inherent to digital systems.

4.1 Decoders

As its name implies, the primary function of a decoder is to, well, decode. It takes in some binary value and outputs a signal corresponding to that value. Recall from Chapter 1 how several binary bits can represent a numeric value. The decoder performs this function in a single chip or CAD component. Decoders are used inside memory chips to access individual memory locations and in computers to access specific input and output devices.

4.1.1 Decoder Basics

A decoder has n inputs. Together, these bits represent a single n-bit value. The n bits can represent any of 2^n values, ranging from 0 to $2^n - 1$. For example, two bits can represent any of four (2^2) values: 00, 01, 10, 11, or 0, 1, 2, 3, or 0 to 3 (0 to $2^2 - 1$). Three bits can represent any value from 0 (000) to 7 (111), and so on for any value of n. The decoder also has 2^n outputs, each representing one of the 2^n possible values of the inputs. We typically refer to this as an n to 2^n decoder.

Figure 4.1 shows a generic 3 to 8 decoder. Inputs I_2, I_1, and I_0 represent a 3-bit value. I_2 is the most significant bit and I_0 is the least significant bit. For example, I_2=1, I_1=0 and I_0=0 represents the binary value 100, or 4. Outputs O_0 to O_7 correspond to the eight (2^3) possible values represented by these bits. The output corresponding to the input value is set to 1 and the other outputs are set to 0. For input value 100, the decoder sets O_4 to 1 and the other seven outputs to 0. The animation in the figure shows the outputs for each possible input value.

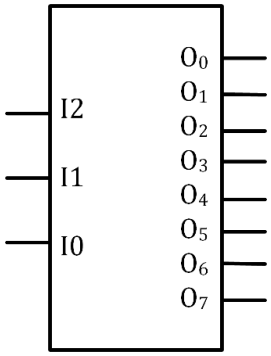


Figure 4.1: Generic 3 to 8 decoder with values shown

WATCH ANIMATED FIGURE 4.1

The decoder just presented, however, is not completely accurate. Whatever value is input to this decoder always generates one specific output. But sometimes we don't want any of the outputs to be active. (Why? We'll get to that shortly.) To do this, a decoder may have an *enable* input. When the enable signal is active (usually 1), the decoder works exactly as described previously. It sets the output corresponding to the inputs to 1 and the remaining outputs to 0. However, if the enable input is 0, all the outputs are set to 0, even the output corresponding to the input value. This is shown in Figure 4.2.

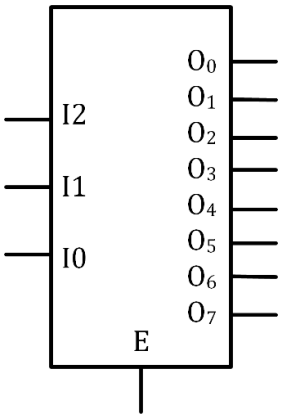


Figure 4.2: Generic 3 to 8 decoder with enable and values shown

WATCH ANIMATED FIGURE 4.2

Now, back to our previous question. Why would we sometimes not want any decoder outputs to be active? That has a lot to do with the limitations of chip manufacturing. When decoder chips were first developed, chip manufacturing technologies had limits on how many pins could be on a chip. Since each input and each output is assigned to one pin, limiting the number of pins limits the number of inputs and outputs, which limits the maximum size of the decoder. For typical chips, the largest decoder is 4 to 16.

You can use the enable signal to combine two or more decoders and make them act like a single, larger decoder. To see how this works, consider the circuit shown in Figure 4.3. We combine two 3 to 8 decoders to create a 4 to 16 decoder. We have a 4-bit value, but each decoder only has three inputs. We take the three lowest order bits and connect them to the three inputs of both decoders. Then we use the remaining bits, only one bit in this case, to generate the values for the enable signals. In this figure, we take the most significant bit of the 4-bit value and use it to generate the enable signals. We pass it through a NOT gate to enable the upper decoder, and we also send it directly to the enable of the lower decoder. When this bit is 0, the NOT gate outputs a 1, the upper decoder is enabled, and the three lower bits cause one of the decoder's outputs to be active. Since it is input directly to the enable of the lower decoder, that decoder is disabled, or not enabled, and all its outputs are set to 0. Conversely, when the most significant bit is 1, the upper decoder is disabled and the lower decoder is enabled. As shown in the figure, the outputs of the upper decoder correspond to input values 0000 (0) to 0111 (7) and those of the lower decoder are activated by input values 1000 (8) to 1111 (15).

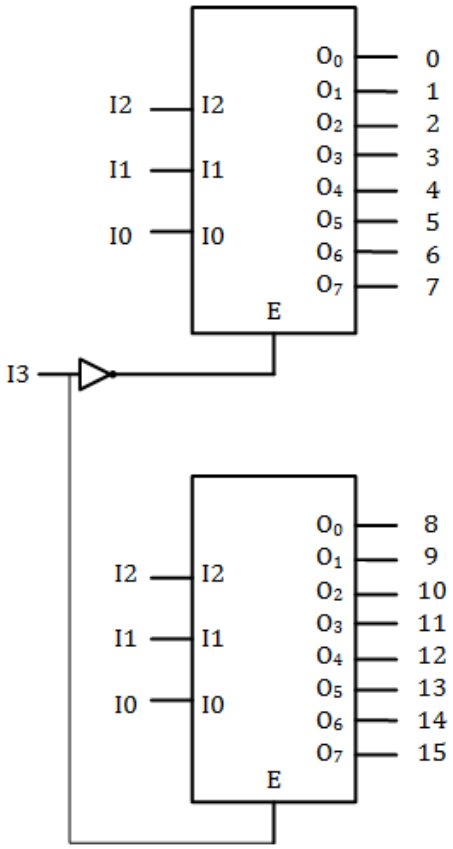


Figure 4.3: 4 to 16 decoder constructed using two 3 to 8 decoders

WATCH ANIMATED FIGURE 4.3

There are other ways to combine decoders. Consider the circuit shown in Figure 4.4. Here, we use four 2 to 4 decoders to generate the outputs of a 4 to 16 decoder. These decoders input the two low-order bits of the input value, and the two remaining bits are used to generate the enable signals. We could use combinatorial logic gates, but instead we use another 2 to 4 decoder. The animation in this figure shows how each of the values generates its corresponding output.

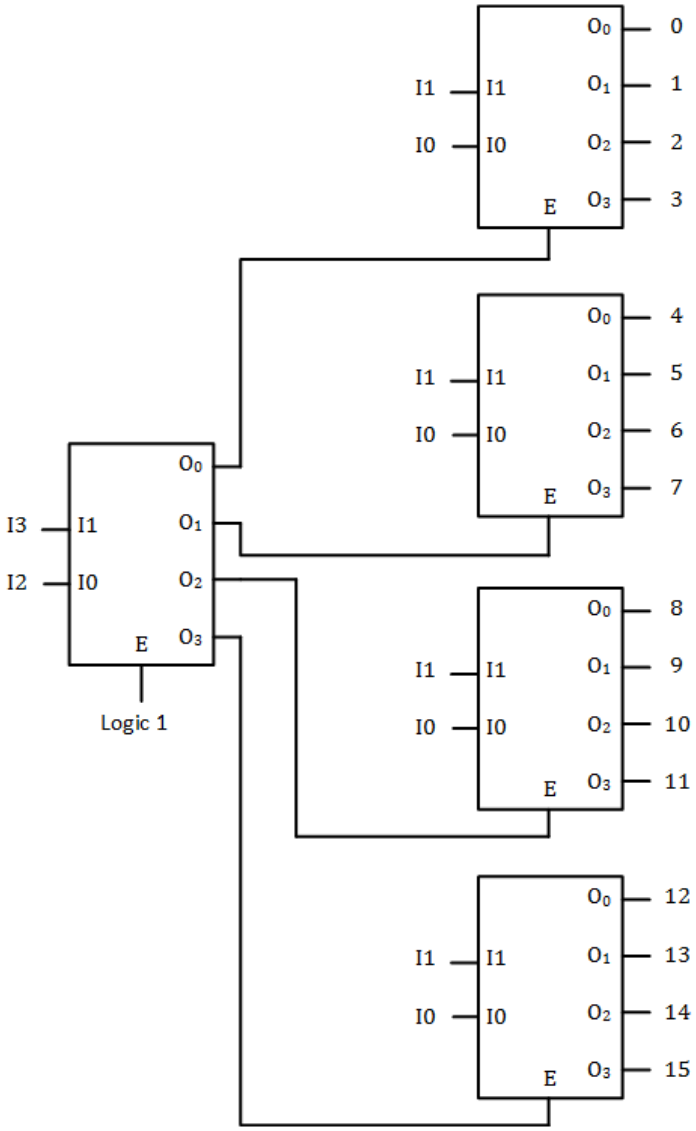


Figure 4.4: 4 to 16 decoder constructed using 2 to 4 decoders

WATCH ANIMATED FIGURE 4.4

4.1.2 Decoder Internal Design

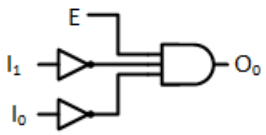
Now that we've seen how decoders work from the outside, the next question is how they work on the inside. What does the circuitry inside a decoder chip look like (as a digital logic circuit) that makes it generate the correct outputs for every possible combination of input values? More succinctly, how can you design a decoder?

One way to do this is to determine the function for each output, and then implement these functions in digital logic, much as we did in the previous chapter.

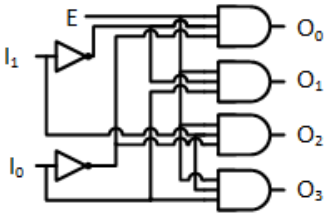
Consider, for example, a 2 to 4 decoder with an enable signal. Its truth table is shown in Figure 4.5 (a). Now consider output O_0. It should only be set to 1 when $I_1 = 0$, $I_0 = 0$, and enable input E = 1. Figure 4.5 (b) shows one circuit to realize this function. Repeating this process for the other outputs produce the final circuit shown in Figure 4.5 (c).

E	I_1	I_0	O_0	O_1	O_2	O_3
0	X	X	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

(a)



(b)



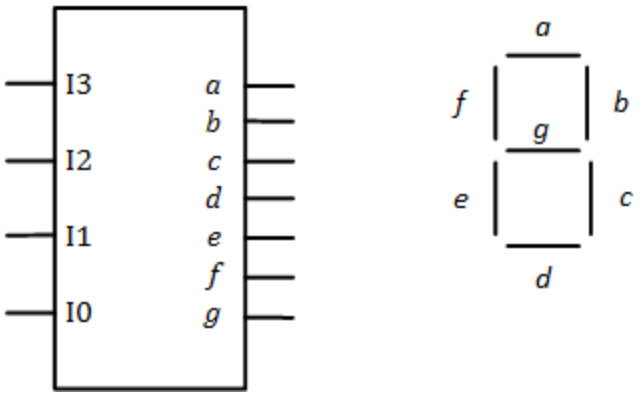
(c)

Figure 4.5: 2 to 4 decoder: (a) Truth table; (b) Circuit to realize output O_0; (c) Complete circuit

4.1.3 BCD to 7-segment Decoder

Although you could design a chip to implement any function, it is only economically feasible to produce chips for commonly used functions. Otherwise, you would not sell enough chips to cover your development and manufacturing costs. One commonly used function is to convert a BCD (binary-coded decimal) value to the signals needed to display it on a 7-segment LED display. This is the function performed by a BCD to 7-segment decoder.

A BCD to 7-segment decoder has a 4-bit input that has the value of a decimal digit ranging from 0 to 9, or 0000 to 1001. These decoders generally assume that they do not receive an invalid input, 1010 to 1111. When designing the decoder, the outputs associated with these inputs are treated as don't care values, which simplifies the design of the chip. This decoder has seven outputs, labeled *a*, *b*, *c*, *d*, *e*, *f*, and *g*, which are connected to the display and cause the segments to light up or remain unlit.



(a) (b)

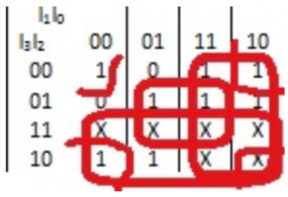
Figure 4.6 (a) Generic BCD to 7-segment decoder; (b) 7-segment display with segments labeled

There are two types of 7-segment displays: common anode and common cathode. In common anode displays, the inputs to the LEDs in the display are connected together, and in the circuit they are connected to the positive voltage of the power supply, or logic 1. The outputs of the LEDs are connected to the outputs of the decoder. When the decoder output is 0, the LED lights up; when it is 1, the LED is off. Common cathode displays do the opposite. The outputs of the LEDs are connected together and connected to the circuit ground, which is logic 0. A decoder output of 1 lights the LED and 0 turns it off.

To design a BCD to 7-segment decoder, we will use the same procedure we've been using all along. First, we create a truth table with four inputs (corresponding to the BCD digit) and seven outputs (one for each segment). We will design a common cathode decoder, so we want each segment's value to be 1 when it should be lit. Figure 4.7 (a) shows the truth table for our decoder.

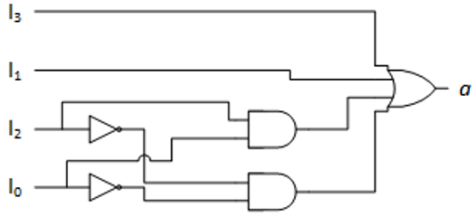
$I_3I_2I_1I_0$	*a*	*b*	*c*	*d*	*e*	*f*	*g*
0000	1	1	1	1	1	1	0
0001	0	1	1	0	0	0	0
0010	1	1	0	1	1	0	1
0011	1	1	1	1	0	0	1
0100	0	1	1	0	0	1	1
0101	1	0	1	1	0	1	1
0110	1	0	1	1	1	1	1
0111	1	1	1	0	0	0	0
1000	1	1	1	1	1	1	1
1001	1	1	1	1	0	1	1
Others	X	X	X	X	X	X	X

(a)



$a = I_3 + I_1 + I_2I_0\,I_2'I_0'$

(b)



(c)

Figure 4.7 BCD to 7-segment decoder: (a) Truth table; (b) Karnaugh Map for segment *a*; (c) Circuit for segment *a*

Next, we create the function for each segment individually. Figure 4.7 (b) shows the Karnaugh Map and function for segment *a*. Finally, we design a circuit to realize the function. One possible circuit is shown in Figure 4.7 (c). The design of the circuits for the remaining segments and for the common anode decoder are left as exercises for the reader.

4.2 Encoders

If the name encoder makes you think it is the opposite of a decoder, you're mostly right. Encoders are used to create an *n*-bit output that corresponds to the one of its 2^n inputs that is active. In this section, we'll look at the basics of encoding and the encoder's internal design, as well as a special encoder, called the priority encoder, which is used when more than one input is active.

4.2.1 Encoder Basics

Whereas a decoder takes an *n*-bit input and sets one of its 2^n outputs to 1, the encoder goes in the other direction. It has 2^n inputs, numbered 0 to $2^n - 1$, and outputs the *n*-bit value corresponding to the single input that is set to 1.

Consider the generic 8 to 3 encoder shown in Figure 4.8. If one of the eight inputs is set to 1, the three output bits indicate which input is set. For example, if input I_6 is 1, the output would be set to $O_2 = 1$, $O_1 = 1$, and $O_0 = 0$, or 110.

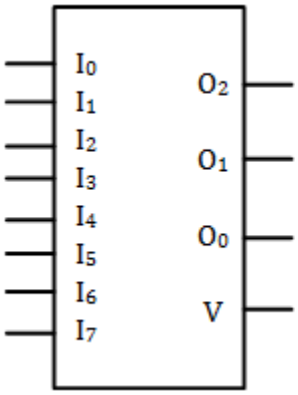


Figure 4.8: Generic 8 to 3 encoder with values shown

WATCH ANIMATED FIGURE 4.8

As long as one input is active, the encoder functions as described. But what does the encoder do if none of the inputs is active? Since the encoder outputs binary values, the O_2, O_1, O_0, will always be set to a value corresponding to one of the inputs, even though that input isn't active. This is the reason this encoder includes another output, *V*. *V* is the *valid* output. The encoder sets *V* to 1 if any of the inputs is equal to 1, or 0 if none of the inputs is 1. When using an encoder, there are two things we need to check. One is the encoded output bits and the

other is the valid bit. If $V = 1$, then the output bits indicate which input is active. However, if $V = 0$, then no input is active, regardless of the value of the output bits. Figure 4.9 shows the truth table for the generic 8 to 3 encoder. For this encoder, O_2, O_1, O_0 is set to 000 when there is no active input.

$I_7I_6I_5I_4I_3I_2I_1I_0$	O_2	O_1	O_0	V
00000000	0	0	0	0
00000001	0	0	0	1
00000010	0	0	1	1
00000100	0	1	0	1
00001000	0	1	1	1
00010000	1	0	0	1
00100000	1	0	1	1
01000000	1	1	0	1
10000000	1	1	1	1

Figure 4.9: Truth table for the generic 8 to 3 encoder

As with decoders, it is possible to combine encoders to handle more inputs. Figure 4.10 shows a 16 to 4 encoder constructed using two 8 to 3 encoders. Let's look at each part of this circuit in more detail.

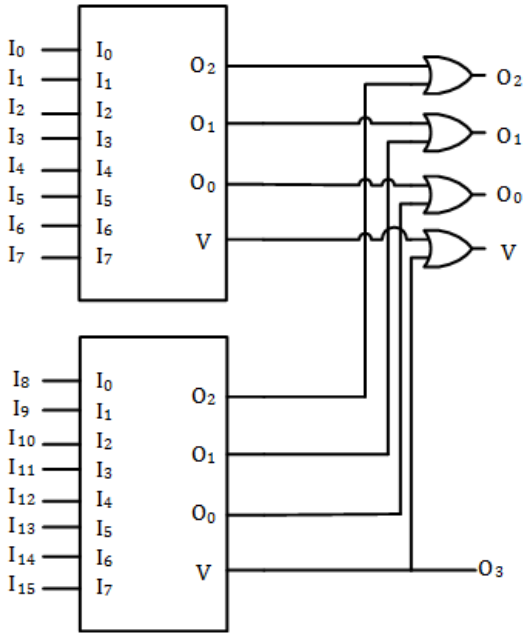


Figure 4.10: 16 to 4 encoder constructed using 8 to 3 encoders

WATCH ANIMATED FIGURE 4.10

The encoder system has 16 inputs. This circuit allocates inputs 0 to 7 (0000 to 0111) to the upper encoder and 8 to 15 (1000 to 1111) to the lower encoder. This becomes important when we generate the data outputs shortly.

If any input from I_0 to I_7 is active, the upper encoder sets its *V* output to 1 and its *O* outputs to the value of the active input, 000 to 111. The same is true for inputs I_8 to I_{15} for the lower decoder. Figure 4.11 (a) shows a partial truth table for this system as it generates outputs O_2, O_1, and O_0. From this table, we can verify that logically ORing together the corresponding outputs (the two O_2s, the two O_1s, and the two O_0s) produces the correct three low-order bits of the encoder output.

	Upper Encoder			Lower Encoder			Overall		
Active Input	O_2	O_1	O_0	O_2	O_1	O_0	O_2	O_1	O_0
None	0	0	0	0	0	0	0	0	0
I_0	0	0	0	0	0	0	0	0	0
I_1	0	0	1	0	0	0	0	0	1
I_2	0	1	0	0	0	0	0	1	0
I_3	0	1	1	0	0	0	0	1	1
I_4	1	0	0	0	0	0	1	0	0
I_5	1	0	1	0	0	0	1	0	1
I_6	1	1	0	0	0	0	1	1	0
I_7	1	1	1	0	0	0	1	1	1
I_8	0	0	0	0	0	0	0	0	0
I_9	0	0	0	0	0	1	0	0	1
I_{10}	0	0	0	0	1	0	0	1	0
I_{11}	0	0	0	0	1	1	0	1	1
I_{12}	0	0	0	1	0	0	1	0	0
I_{13}	0	0	0	1	0	1	1	0	1
I_{14}	0	0	0	1	1	0	1	1	0
I_{15}	0	0	0	1	1	1	1	1	1

(a)

Active Input	Upper Encoder *V*	Lower Encoder *V*	Overall *V* O_3
None	0	0	0 0
I_0	1	0	1 0
I_1	1	0	1 0
I_2	1	0	1 0
I_3	1	0	1 0
I_4	1	0	1 0
I_5	1	0	1 0
I_6	1	0	1 0
I_7	1	0	1 0
I_8	0	1	1 1
I_9	0	1	1 1
I_{10}	0	1	1 1
I_{11}	0	1	1 1
I_{12}	0	1	1 1
I_{13}	0	1	1 1
I_{14}	0	1	1 1
I_{15}	0	1	1 1

(b)

Active Input	O_3	O_2	O_1	O_0	**V**
None	0	0	0	0	0
I_0	0	0	0	0	1
I_1	0	0	0	1	1
I_2	0	0	1	0	1
I_3	0	0	1	1	1
I_4	0	1	0	0	1
I_5	0	1	0	1	1
I_6	0	1	1	0	1
I_7	0	1	1	1	1
I_8	1	0	0	0	1
I_9	1	0	0	1	1
I_{10}	1	0	1	0	1
I_{11}	1	0	1	1	1
I_{12}	1	1	0	0	1
I_{13}	1	1	0	1	1
I_{14}	1	1	1	0	1
I_{15}	1	1	1	1	1

(c)

Figure 4.11: Truth tables for the 16 to 4 encoder: (a) Outputs O_2, O_1, and O_0; (b) O_3 and *V*; (c) Final truth table

Next, let's look at O_3. When any input from I_0 to I_7 is active, our output will be a value from 0 (0000) to 7 (0111). All of these values have $O_3 = 0$. If an input from I_8 to I_{15} is active, then the output will be in the range from 8 (1000) to 15 (1111). All of these values have $O_3 = 1$. Fortunately for our circuit, all of these inputs set the *V* output of the lower encoder to 1, and all other inputs result in that output being set to 0. So, we can use that *V* output to generate O_3 directly.

Finally, we need to generate the final *V* output. This should be 1 whenever any of the 16 inputs is 1. Since *V* of the upper encoder is 1 when any input from I_0 to I_7 is active, and *V* of the lower encoder is 1 when an input from I_8 to I_{15} is active, we can logically OR these two signals together. The resulting signal is a 1 when any input from I_0 to I_{15} is 1, which is exactly what we want.

Figure 4.11 (b) shows the values of *O3* and *V* for all possible inputs. The final truth table for the overall circuit is shown in Figure 4.11 (c).

4.2.2 Encoder Internal Design

The internal design of the encoder is relatively straightforward. To design the 8 to 3 encoder, we start with the truth table shown in Figure 4.9. Then we determine the function for each individual output and create a circuit to realize each function.

First, let's look at output O_2. It should be set to 1 when any of inputs I_4, I_5, I_6, or I_7 is active, or $O_2 = I_4 + I_5 + I_6 + I_7$. (Remember from Chapter 2, + is the logical OR operation.) Following the same process, we come up with these functions for the chip outputs.

$$O_2 = I_4 + I_5 + I_6 + I_7$$
$$O_1 = I_2 + I_3 + I_6 + I_7$$
$$O_0 = I_1 + I_3 + I_5 + I_7$$
$$V = I_0 + I_1 + I_2 + I_3 + I_4 + I_5 + I_6 + I_7$$

Figure 4.12 shows a circuit to realize these functions.

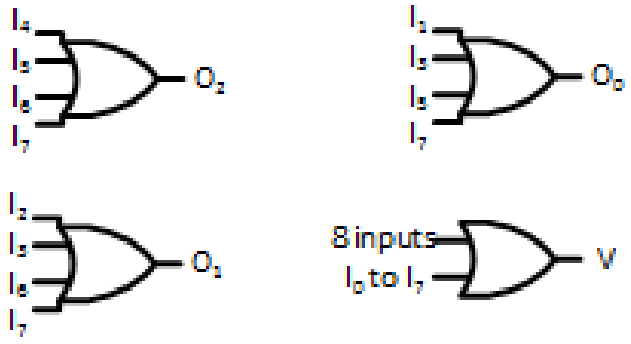


Figure 4.12: Internal design of an 8 to 3 encoder

4.2.3 Priority Encoders

The circuit shown in Figure 4.12 works under most, but not all, circumstances. When exactly one input is active, it outputs the correct output value and sets *V* to 1, indicating that we have a valid input. When no inputs are active, it sets *V* to 0, as required. But what happens when more than one input is active? For example, if I_6 and I_3 are both set to 1, the circuit will set O_2 = 1, O_1 = 1, O_0 = 1, and *V* = 1, which incorrectly indicates that input I_7 is active.

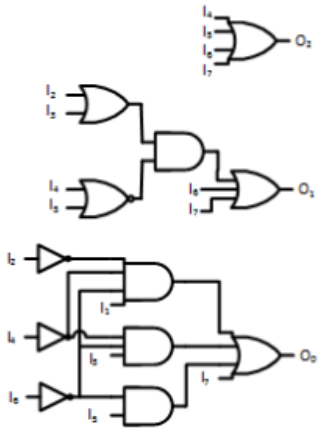
This is not a design flaw, but rather a limitation of our design. At the beginning of this section, I stated that the encoder outputs the value corresponding to the *single* input that is set to 1. This encoder is designed to work only when we can ensure that no more than one input is active, and that is the case for many circuits. Sometimes, however, it is possible for more than one input to be active, and we must choose which one to encode. To do this, we use a special type of encoder called the priority encoder.

A priority encoder acts just like a normal encoder, with one exception. When more than one input is active, a priority encoder outputs the value for the highest numbered active input. That is, each input has priority over all inputs with lower numbers, and all inputs with higher numbers have priority over it. In our earlier example, when I_6 and I_3 are both active, the priority encoder outputs 110, because input I_6 has priority over input I_3.

Figure 4.13 (a) shows the truth table for the 8 to 3 priority encoder. Unlike the regular encoder, which requires that no more than one input is set to 1, the priority encoder has no restrictions on its input values. It simply encodes the highest numbered active input and ignores those below it.

I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	O_2	O_1	O_0	V
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	0	1	X	0	0	1	1
0	0	0	0	0	1	X	X	0	1	0	1
0	0	0	0	1	X	X	X	0	1	1	1
0	0	0	1	X	X	X	X	1	0	0	1
0	0	1	X	X	X	X	X	1	0	1	1
0	1	X	X	X	X	X	X	1	1	0	1
1	X	X	X	X	X	X	X	1	1	1	1

(a)



(b)

Figure 4.13: 8 to 3 priority encoder: (a) Truth table; (b) Circuit design

Naturally, implementing prioritization increases the complexity of the design for a priority encoder as compared to a normal encoder, at least for parts of the circuit. As shown in the circuit for the 8 to 3 priority encoder in Figure 4.13 (b), the circuitry to generate O_2 and *V* is

the same as for the normal encoder. However, this is not the case for the other outputs. Consider output O_1. The normal priority encoder sets this output to 1 if we are encoding input I_2, I_3, I_6, or I_7. These inputs would be encoded as 010, 011, 110, and 111, respectively, all of which have the middle bit set to 1. This would also be the case for the priority encoder, but only if we are encoding one of these inputs. This will always be the case for I_6 and I_7, but not necessarily for I_2 and I_3. For example, if I_2 and I_5 are both active, the priority encoder outputs 101, the value for I_5. Here, O_1 should be 0, not 1.

The design resolves this problem by checking to see if I_4 or I_5 is active. The circuit sets O_1 = 1 if (either I_6 or I_7 is active) or if ((I_2 or I_3 is active) AND (neither I_4 nor I_5 is active)). The circuit to generate O_0 follows this same process, but it is more complex. It sets O_0 to 1 under the following conditions.

(IF I_7 = 1) OR
(IF I_5 = 1 AND I_6 = 0) OR
(IF I_3 = 1 AND I_4 = 0 AND I_6 = 0) OR
(IF I_1 = 1 AND I_2 = 0 AND I_4 = 0 AND I_6 = 0)

THEN O_0 = 1.

4.3 Multiplexers

A multiplexer, or MUX for short, is a digital circuit that has several inputs and allows one of them to pass through to its single output. In addition to these data inputs, it has several control inputs that specify which data input is to pass through to the output. It has many uses in digital circuits, one of which we will see in the next chapter.

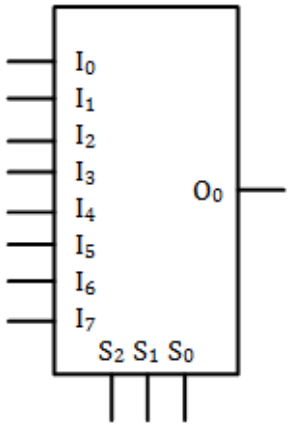
4.3.1 Multiplexer Basics

Although different from a decoder, the multiplexer and decoder employ the same design principles for part, but not all, of their designs. Throughout this subsection, I'll highlight these similarities. Later in this section, we'll design a multiplexer that uses a decoder in its design.

To start, consider the generic 8-input multiplexer and its truth table, both shown in Figure 4.14. The eight data inputs are labeled I_0 to I_7. In a circuit, each input will have a logical 0 or 1 connected to it, usually generated by some other digital logic in the circuit. In many circuits, these input values may change over time.

E	S_2	S_1	S_0	O_0
0	X	X	X	0
1	0	0	0	I_0
1	0	0	1	I_1
1	0	1	0	I_2
1	0	1	1	I_3
1	1	0	0	I_4
1	1	0	1	I_5
1	1	1	0	I_6
1	1	1	1	I_7

(a)



(b)

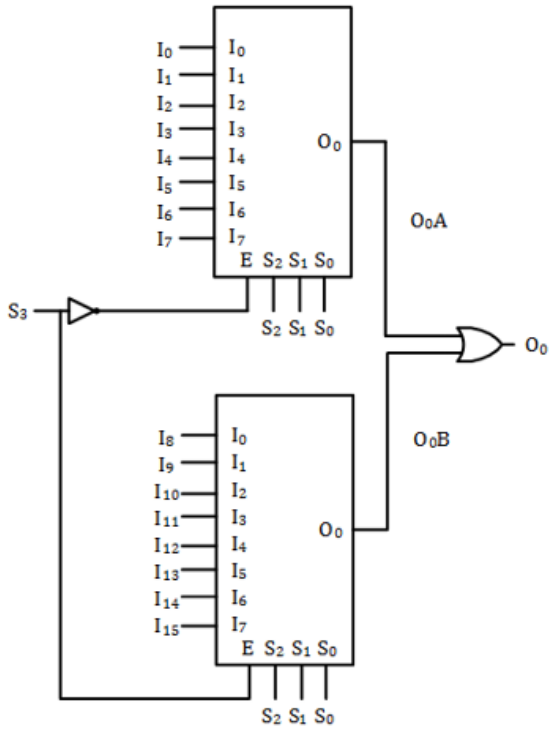
Figure 4.14: 8 to 1 multiplexer: (a) Truth table; (b) Generic representation with values shown

WATCH ANIMATED FIGURE 4.14

The three inputs labeled S_2, S_1, and S_0 select the input to be passed to the output. The three bits are decoded in the same way as they are in the decoder. However, in the decoder these bits set the selected output bit to 1. In the multiplexer, they select an input and send it to the output. For example, if $S_2S_1S_0$ 110 (6), then whatever value is on input I_6 is output via O_0. In the truth table, we list the value of output O_0 as I_6 to indicate that O_0 equals whatever value is being input to I_6. Finally, the multiplexer has an enable input, E. When E = 1, the multiplexer is enabled and acts as described. However, when E = 0, the multiplexer is disabled and outputs the value 0, regardless of the values on the data inputs and select signals.

Also, as is the case for decoders, we can combine two or more multiplexers to act like one larger multiplexer. We follow the same strategy used to combine decoders. We use the low-order select bits to choose an input from each multiplexer and the high-order select bits to enable exactly one of the multiplexers.

Figure 4.15 (a) shows two 8-input multiplexers combined to act like a single 16-input multiplexer.



(a)

S_3	S_2	S_1	S_0	O_0A	O_0B	O_0
1	0	0	0	0	I_8	I_8
1	0	0	1	0	I_9	I_9
1	0	1	0	0	I_{10}	I_{10}
1	0	1	1	0	I_{11}	I_{11}
1	1	0	0	0	I_{12}	I_{12}
1	1	0	1	0	I_{13}	I_{13}
1	1	1	0	0	I_{14}	I_{14}
1	1	1	1	0	I_{15}	I_{15}
0	0	0	0	I_0	0	I_0
0	0	0	1	I_1	0	I_1
0	0	1	0	I_2	0	I_2
0	0	1	1	I_3	0	I_3
0	1	0	0	I_4	0	I_4
0	1	0	1	I_5	0	I_5
0	1	1	0	I_6	0	I_6
0	1	1	1	I_7	0	I_7

(b)

Figure 4.15: (a) 16 to 1 multiplexer constructed using two 8 to 1 multiplexers; (b) Truth table

WATCH ANIMATED FIGURE 4.15

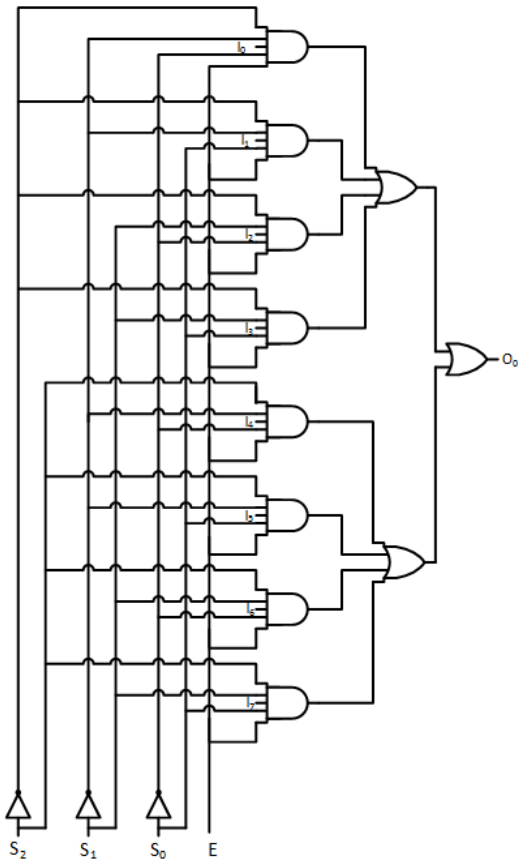
Notice from the truth table in Figure 4.15 (b) that only one multiplexer is enabled at a time, the upper multiplexer when S_3 = 0 and the lower one when S_3 = 1. The multiplexer that is not enabled outputs 0, regardless of its input values, while the enabled multiplexer outputs the value of one of its inputs. Then these values are logically ORed together. Since any value ORed with 0 is its original value, this passes the selected input value directly to the output.

4.3.2 Multiplexer Internal Design

Now that we've seen what a multiplexer looks like from the outside, we need to design the internal logic circuit to make it perform its desired functions. As usual, we'll start with a truth table. Figure 4.16 (a) shows the truth table for the multiplexer, but we've expanded it to include the different data input values.

E	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	S_2	S_1	S_0	O_0
0	X	X	X	X	X	X	X	X	X	X	X	0
1	0	X	X	X	X	X	X	X	0	0	0	0
1	1	X	X	X	X	X	X	X	0	0	0	1
1	X	0	X	X	X	X	X	X	0	0	1	0
1	X	1	X	X	X	X	X	X	0	0	1	1
1	X	X	0	X	X	X	X	X	0	1	0	0
1	X	X	1	X	X	X	X	X	0	1	0	1
1	X	X	X	0	X	X	X	X	0	1	1	0
1	X	X	X	1	X	X	X	X	0	1	1	1
1	X	X	X	X	0	X	X	X	1	0	0	0
1	X	X	X	X	1	X	X	X	1	0	0	1
1	X	X	X	X	X	0	X	X	1	0	1	0
1	X	X	X	X	X	1	X	X	1	0	1	1
1	X	X	X	X	X	X	0	X	1	1	0	0
1	X	X	X	X	X	X	1	X	1	1	0	1
1	X	X	X	X	X	X	X	0	1	1	1	0
1	X	X	X	X	X	X	X	1	1	1	1	1

(a)



(b)

Figure 4.16: 8 to 1 multiplexer (a) Truth table; (b) Internal design

WATCH ANIMATED FIGURE 4.16

For this design, we're going to take a slightly different approach. Instead of creating one large Karnaugh map, we create circuits for each possible value of select signals S_2, S_1, and S_0, and then combine the outputs to produce the final output of the multiplexer.

Consider the case $S_2S_1S_0$ = 000 and E = 1. In the first column of AND gates, only the uppermost AND gate outputs a 1; all others have at least one input equal to 0 and output a 0. In the second column, the uppermost AND gate has one input set to 1 by the previous AND gate and the other equal to the value on input I_0. If I_0 = 0, 1 ^ 0 = 0, and the gate outputs a 0; if it is 1, 1 ^ 1 = 1 and it outputs a 1. In either case, it outputs the value of I_0. The other AND gates in this column all input 0s from their previous AND gates and therefore output 0s, regardless of the value of their data inputs. Finally, we logically OR these values (I_0 and all 0s together. Anything ORed with 0 is equal to its original value (0 + 0 = 0, 1 + 0 = 1), so the output would just be the value of I_0. The reasoning is the same for the other values of the select signals. Of course, if E = 0, all AND gates output a 0 and O_0 becomes 0, as desired when the multiplexer is not enabled. The complete design is shown in Figure 4.16 (b).

Much of this design may look familiar to you. Everything from the first column of AND gates to the left edge of the circuit is just a 3 to 8 decoder with an enable input. This is highlighted in the animation in Figure 4.17, where the circuit is transformed to one using a decoder to replace these components.

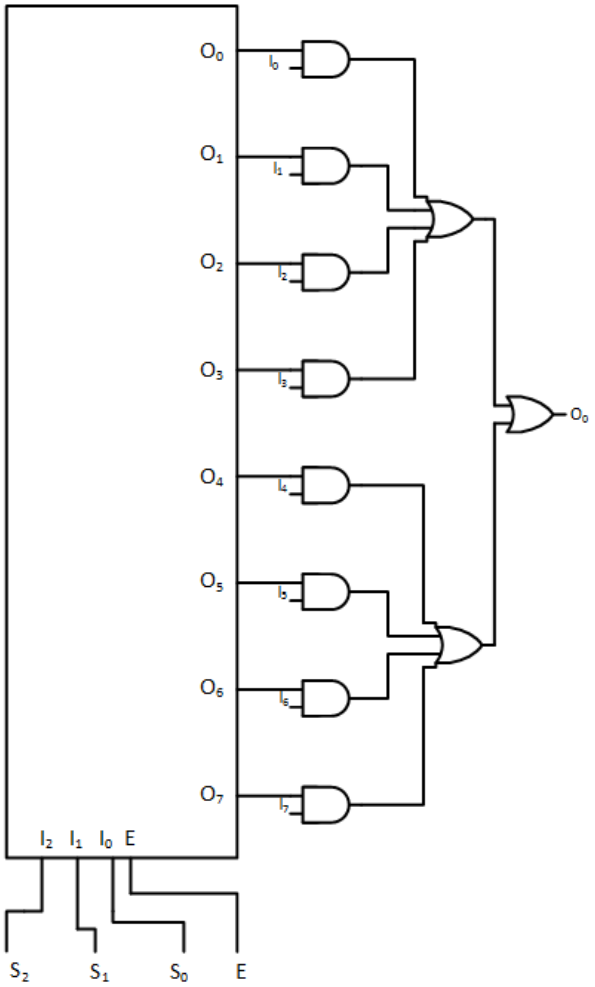


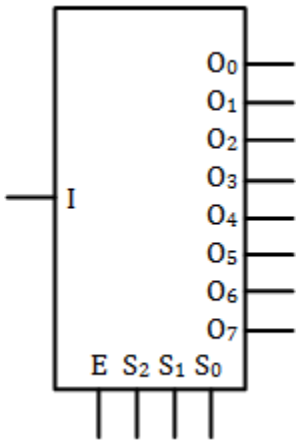
Figure 4.17: 8 to 1 multiplexer constructed using a 3 to 8 decoder

WATCH ANIMATED FIGURE 4.17

4.4 Demultiplexers

The term demultiplexer has two different meanings. In many cases, demultiplexer is just another name for decoder, and the two may be used interchangeably. In the other meaning, a demultiplexer routes a single data input to one of several outputs, with select bits indicating which output receives the data. Since we have already covered decoders in Section 4.2, this section focuses solely on the latter definition.

A demultiplexer, like the decoder, has *n* select inputs and 2^n outputs, numbered 0 to 2^n – 1, and an enable input. In addition, it has a single data input. A generic 1 to 8 demultiplexer is shown in Figure 4.18 (a). As indicated in the truth table in Figure 4.18 (b), when the demultiplexer is enabled, the input value is sent to the output specified by the select bits. For example, when $S_2S_1S_0$ = 110, output O_6 gets the value on the demultiplexer input.



(a)

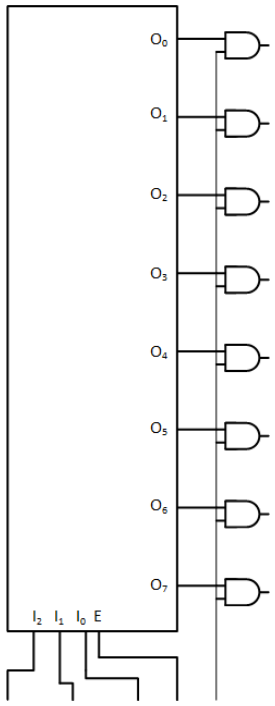
E	S_2	S_1	S_0	O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7
0	X	X	X	0	0	0	0	0	0	0	0
1	0	0	0	I	0	0	0	0	0	0	0
1	0	0	1	0	I	0	0	0	0	0	0
1	0	1	0	0	0	I	0	0	0	0	0
1	0	1	1	0	0	0	I	0	0	0	0
1	1	0	0	0	0	0	0	I	0	0	0
1	1	0	1	0	0	0	0	0	I	0	0
1	1	1	0	0	0	0	0	0	0	I	0
1	1	1	1	0	0	0	0	0	0	0	I

(b)

Figure 4.18: (a) Generic 1 to 8 demultiplexer (b) Truth table

WATCH ANIMATED FIGURE 4.18

To help you understand how a demultiplexer works, visualize the demultiplexer as a decoder with a data input. The demultiplexer decodes the select bits as usual, but then we logically AND the data input with each individual decoder output. This is shown in Figure 4.19 (a). The selected output of the decoder (if enabled) is 1 and all other outputs are 0. *I* ^ 1 = *I*, which is output on the selected output. All other outputs have the value *I* ^ 0 = 0.



(a)

				Decoder Outputs								Demultiplexer Outputs							
E	S_2	S_1	S_0	O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7	O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7
0	X	X	X	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0	0	0	0	I	0	0	0	0	0	0	0
1	0	0	1	0	1	0	0	0	0	0	0	0	I	0	0	0	0	0	0
1	0	1	0	0	0	1	0	0	0	0	0	0	0	I	0	0	0	0	0
1	0	1	1	0	0	0	1	0	0	0	0	0	0	0	I	0	0	0	0
1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	I	0	0	0
1	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	I	0	0
1	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	I	0
1	1	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	I

(b)

Figure 4.19: 1 to 8 demultiplexer constructed using a 3 to 8 decoder: (a) Circuit diagram; (b) Truth table

WATCH ANIMATED FIGURE 4.19

4.5 Comparators

Computers can perform numerous arithmetic operations on numeric data, such as adding, subtracting, multiplying, or dividing two numbers. They use circuits designed specifically to

perform these operations, and these circuits often employ specially designed components. One such component is the comparator.

A comparator does not add or subtract two values, nor does it perform any other arithmetic or logical operation. Rather, it inputs two numbers, let's call them *A* and *B*, and determines which is larger, or if they are equal. Figure 4.20 shows a generic 4-bit comparator.

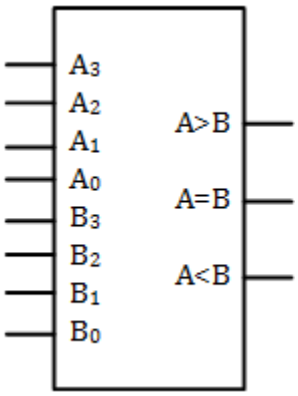


Figure 4.20: Generic 4-bit comparator

For this comparator, *A* and *B* can take on any value from 0000 (0) to 1111 (15). Exactly one of the three outputs will be active, depending on the values of *A* and *B*.

To see how this works, consider how you would compare two 4-digit decimal numbers, say 2021 and 2910. (Just as the comparator has two inputs with the same number of bits, this example also requires two values with the same number of digits.) First, we compare the two digits in the thousands place. For this example, they are both the same, 2, which doesn't tell us anything about which number is larger, so we go to the next digit. The hundreds digit in the first number is 0 and in the second number is 9, so we know the second number must be larger than the first. There is no need to check the remaining digits.

Here's a general rule we can use to help us design a comparator. One number (*A*) is greater than the other number (*B*) if one of these two conditions is met.

1. The digit we are checking in *A* is greater than the digit in *B* AND all previous digits in these two numbers are the same; or
2. A previous digit had already determined that *A* is greater than *B*.

We would start at the most significant digit and work our way down to the least significant digit. If we have not found one number to be greater than the other by the time we have finished comparing all pairs of digits, then the two numbers are equal.

Since we are dealing with binary numbers, the comparison process is relatively straightforward. If we are comparing bits A_i and B_i, one of these three conditions is met:

$A_i > B_i$ if $A_i = 1$ AND $B_i = 0$
$A_i < B_i$ if $A_i = 0$ AND $B_i = 1$
$A_i = B_i$ if $A_i = B_i = 1$ OR $A_i = B_i = 0$

Before you look at the circuit I've created in Figure 4.21, try to design your own circuit to realize these functions.

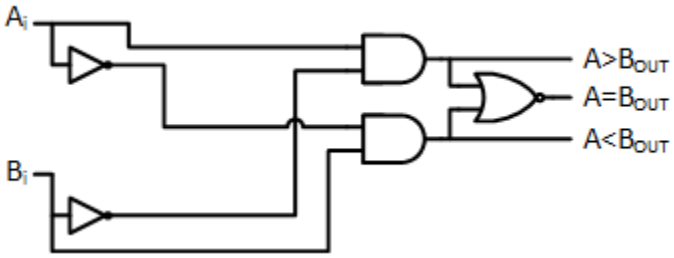


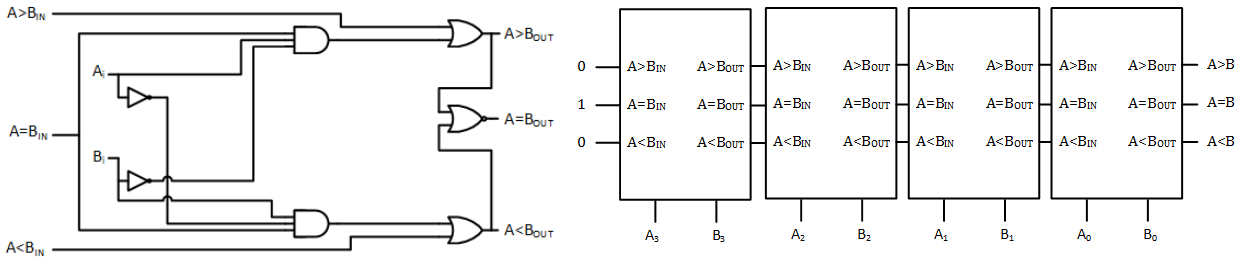
Figure 4.21: A 1-bit comparator

Notice, in my design, I took advantage of the fact that, if A_i is not greater than B_i, and A_i is not less than B_i, then A_i must be equal to B_i. This allows me to use a single NOR gate to generate the $A = B$ signal.

But what if a previous bit had already determined that one number is larger than the other? If we are to construct a comparator using multiple copies of this circuit, one for each bit, we need to make use of the results from previous bits. If we call these previous bits $A>B_{in}$, $A<B_{in}$, and $A=B_{in}$, then we can say

$A>B_{OUT} = 1$ IF $(A>B_{IN} = 1)$ OR $(A=B_{IN} = 1$ AND $A_i = 1$ AND $B_i = 0)$
$A<B_{OUT} = 1$ IF $(A<B_{IN} = 1)$ OR $(A=B_{IN} = 1$ AND $A_i = 0$ AND $B_i = 1)$
$A=B_{OUT} = 1$ IF $(A=B_{IN} = 1)$ AND $(A_i = B_i)$

We can modify the circuit from Figure 4.21 as shown in Figure 4.22 (a) to incorporate these functions. Figure 4.22 (b) shows how this circuit can be replicated and configured to create a 4-bit comparator.



(a) (b)

Figure 4.22: (a) An enhanced 1-bit comparator; (b) Four-bit comparator

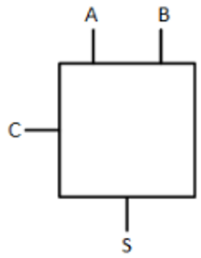
There is another method to perform comparison by subtracting two numbers and examining the result. We will look at this method later in this book.

4.6 Adders

So far we have examined components that perform logical functions on binary values. But we can also design logic circuits that treat their inputs as numeric values and perform arithmetic operations on these values. In this section we examine adders, which, as their name implies, add two numeric inputs to produce a numeric output. We'll start with half adders and full adders, the basic building blocks, and then we'll use these building blocks to design adders that can add any two *n*-bit values.

4.6.1 Half Adders

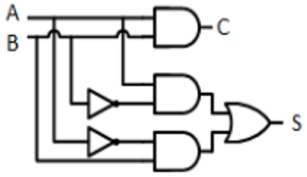
The half adder is the most fundamental arithmetic circuit there is. It inputs two 1-bit values, arithmetically adds them, and outputs the sum. As shown in Figure 4.23 (a), the output can have a value of 0 (0 + 0), 1 (0 + 1 or 1 + 0), or 2 (1 + 1). The last value is represented as 10 in binary, so the half adder must output two bits. The least significant bit is called the *sum*, usually denoted *S*, and the most significant bit is the carry, or *C*. The truth table for the half adder is shown in Figure 4.23 (b).



(a)

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(b)



(c)

Figure 4.23: Half adder: (a) Inputs and outputs; (b) Truth table; (c) Internal design

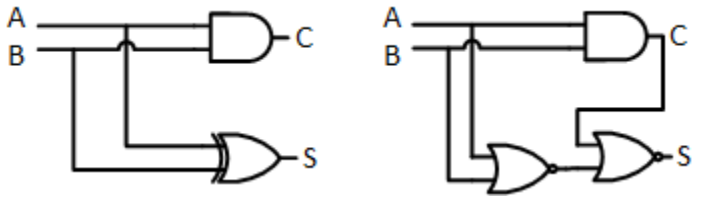
WATCH ANIMATED FIGURE 4.23

So, how can we take these two input values, *A* and *B*, and generate our sum and carry outputs? We do this by treating this as a digital logic design problem, just as we have done when designing the other components in this chapter. We design the logic for *C* and *S* based solely on the value of inputs *A* and *B*.

Carry output *C* is relatively straightforward. It is only set to 1 when *A* = *B* = 1. This is just the AND function, and *C* = *A* ^ *B*. The sum, *S*, is slightly more complicated but also pretty straightforward. It is 1 when either *A* = 1 and *B* = 0 or *A* = 0 and *B* = 1. This is equivalent to *A* = 1 and *B*' = 1 or *A*' = 1 and *B* = 1. In the circuit shown in Figure 4.23 (c), examine the portion of the

circuit that generates output *S*. The upper AND gate outputs a 1 when *A* = 1 and *B*' = 1; the lower gate outputs a 1 when *A*' = 1 and *B* = 1. The OR gate sets *S* to 1 when either AND gate outputs a 1, and 0 otherwise, exactly as desired.

There are other ways you could design the half adder. Everyone is going to come up with the same design for *C*, but there are a few different ways to generate *S*. Looking at the truth table, you may have noticed that *S* is the exclusive OR of *A* and *B*. We could have used an XOR gate to generate *S*, as shown in Figure 4.24 (a). Less obvious, but still valid, we can find all values that set *S* to 0 and NOR them together. That is, if we look at all the combinations that set *S* = 0, and none of them are met, then *S* must be set to 1. One circuit to do this is shown in Figure 4.24 (b). Note that we reduce the size of our circuit by reusing the AND gate that generates *C*. Now it both generates *C* and provides one of the inputs to the NOR gate that generates *S*.



(a) (b)

Figure 4.24: Additional implementations of the half adder: (a) Using an XOR gate; (b) Using *C* to generate *S*

4.6.2 Full adders

Half adders work perfectly when adding numbers that are only one bit each, but they have a problem when adding numbers that are more than one bit. To illustrate this, look at the circuit in Figure 4.25. We want to add two numbers, 3 and 1, and generate the correct result, 4. In binary, this could be expressed as 11 + 01 = 100. We use subscripts to indicate the individual bits of each number as usual, starting with bit 0 as the least significant bit. Here, *A* = 11, so A_1 = 1 and A_0 = 1. Similarly, *B* = 01, or B_1 = 0 and B_0 = 1. We input A_0 and B_0 into the first half adder and A_1 and B_1 into another. The final sum should be *C* of the most significant adder followed by all the sum bits, $C_1S_1S_0$ in this case.

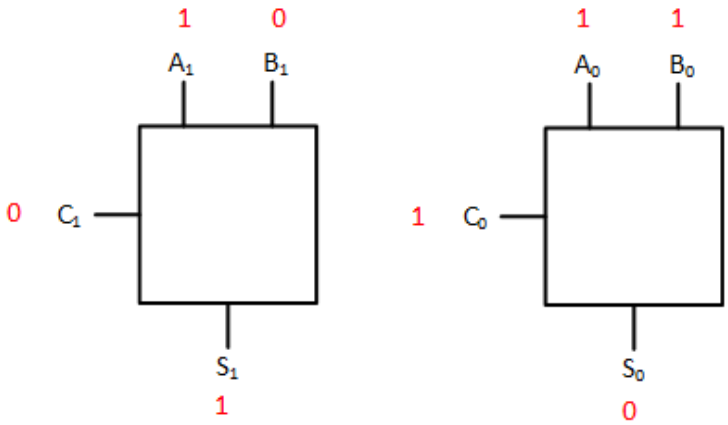


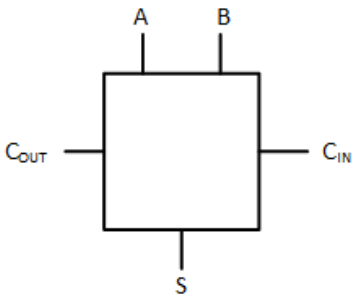
Figure 4.25: Unsuccessful addition using half adders

For these values, this circuit does not work. It has $C_1 = 0$, $S_1 = 1$, and $S_0 = 0$, generating the result 010, or 2. Clearly, $3 + 1 \neq 2$.

The problem has to do with C_0. We generate a value for C_0, but we don't do anything with it when we generate the final sum. The half adder's inputs are all used by inputs *A* and *B*. There are no inputs available to make use of the carry bits when generating the sum. We need a new type of adder to make this work. That adder is the full adder.

The full adder has three inputs. Typically two are used for data, just as we did with the half adder. The third input is used for the carry generated by the previous adder. Since the largest result we can have is 3, or 11, when all three inputs are 1 and we form the sum 1 + 1 + 1, we still need only two outputs.

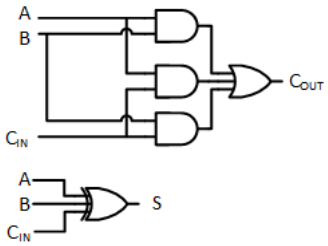
Figure 4.26 (a) shows the full adder and Figure 4.26 (b) shows its truth table. As before, we can design a circuit by treating each output as a separate function and creating a circuit for each function. Figure 4.26 (c) shows one possible circuit; other designs are left as exercises for the reader.



(a)

A	B	C_{IN}	C_{OUT}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

(b)



(c)

Figure 4.26: Full adders: (a) Inputs and outputs; (b) Truth table; (c) Internal design

4.6.3 Ripple adders

Now that we have the full adder, we can create a circuit to add numbers that have more than one bit. The easiest way to do this is to create a ripple adder. This is just a series of full adders with each full adder's carry out bit connected to the carry in of the next higher adder. Figure 4.27 shows how this works when adding 11 + 01. By making use of all the carry bits, this circuit successfully adds these two numbers producing the output 100, or 4.

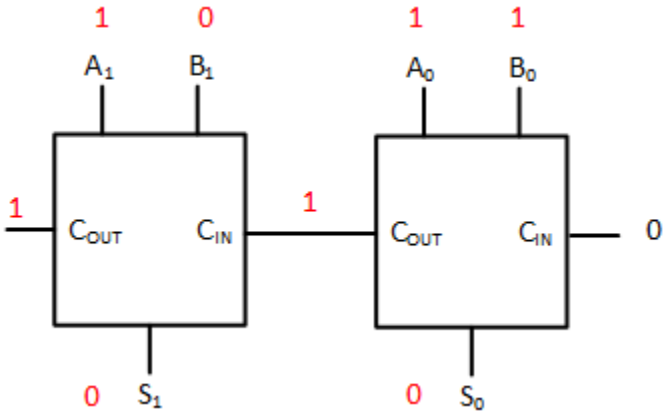


Figure 4.27: Successful addition using full adders

WATCH ANIMATED FIGURE 4.27

There are two things I want to point out about this circuit. First, you can create a ripple adder that adds larger numbers by simply adding more full adders to the circuit and connecting the carry out and carry in signals as we did here. Second, take a look at the least significant bit. There is no other bit to send it a carry input, so this input must be wired to logic 0 to function properly. Alternatively, we could have used a half adder for only this least significant bit of the ripple adder. It is left as an exercise for the reader to verify that both implementations work correctly.

4.6.4 Carry-lookahead Adders

Although ripple adders function properly, they are relatively slow. To see why this is, consider the 4-bit ripple adder shown in Figure 4.28. The animation shows how the signals change their values as it adds 111 + 0001. Initially, each adder adds its two data inputs. The least significant adder generates a carry out value of 1. This is connected to the carry in bit of the next full adder. Its inputs change from $A = 1$, $B = 0$, $C_{in} = 0$ to $A = 1$, $B = 0$, $C_{in} = 1$, and its outputs change from $C_{out} = 0$, $S = 1$ to $C_{out} = 1$, $S = 0$. The same process is repeated for each full adder in sequence. Each adder has to wait for the previous adder to update its carry value before it can generate its final result. Logic gates are fast, but they do take time to output their results. As the number of bits to be added increases, these delays add up.

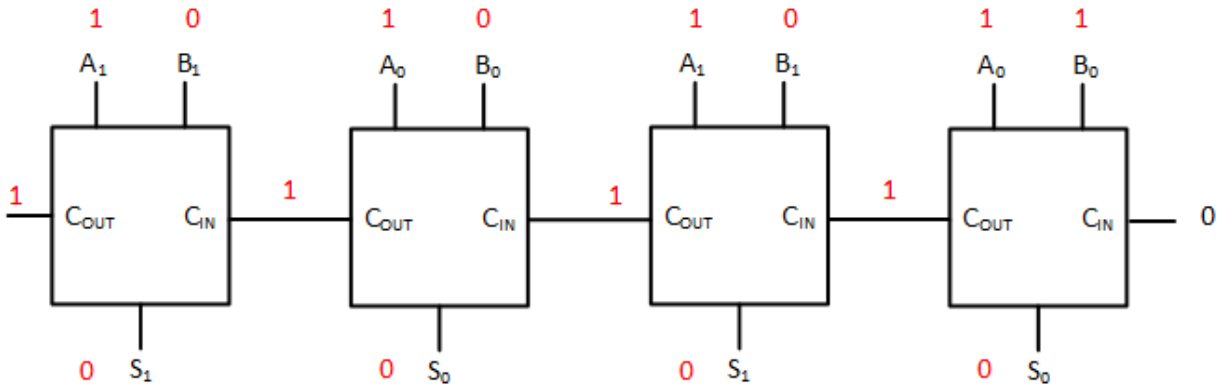


Figure 4.28: 4-bit ripple adder

WATCH ANIMATED FIGURE 4.28

To speed this up, designers developed **carry-lookahead adders**. Like ripple adders, they add two binary numbers. Unlike ripple adders, however, they do not need to wait for carry values to propagate from bit to bit through the adder. Instead, they include additional circuitry that "looks ahead" to see if a carry input will be generated and calculates its results accordingly. As with the ripple adder, carry-lookahead adders incorporate full adders to add two values. The difference lies in how they generate carry output values.

To understand how this works, consider a full adder within a carry-lookahead adder. It has three inputs, two data inputs and a carry in, and two outputs, the sum and carry out. Our goal is to speed up how we generate the carry out signal.

There are two cases that can cause the carry out to be set to 1. One is when the two data inputs are both 1. Regardless of the value of the carry input signal, the carry out will always be 1 in this case. We are adding either 1 + 1 + 0 = 10 or 1 + 1 + 1 = 11, both of which set carry out to 1. We say that the input values *generate* a carry out.

The second case is when one of the input values is 1 and the carry in is 1. Here we add either 0 + 1 + 1 = 10 or 1 + 0 + 1 = 10. The 1 on the carry in is *propagated* to the carry out signal.

Putting these two cases together, the carry out signal is 1 if we generate a carry from the data inputs or if we propagate a 1 from the carry input to the carry output.

For each full adder, we will generate values that indicate if a carry is generated or if it can be propagated. Then we will combine these signals with the carry in to generate the initial logic functions for the carry out signals. But there is one more step. We will expand each function by substituting in the function for the previous carry bit so that each carry output is expressed solely in terms of the data inputs. This is how the adder "looks ahead" when generating the carry bits. To clarify how this works, we'll go through these steps individually for the 4-bit adder. The inputs *A* and *B* have four bits, labeled $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$.

Each full adder generates a carry if its two input bits are both 1. It is easy to check for this by ANDing the two inputs together. We will call this function *g*, for *generate*. For our 4-bit adder, these functions are:

$$g_0 = A_0 \text{ ^ } B_0$$
$$g_1 = A_1 \text{ ^ } B_1$$
$$g_2 = A_2 \text{ ^ } B_2$$
$$g_3 = A_3 \text{ ^ } B_3$$

A full adder propagates its carry in if one of its inputs is 1. We could exclusive OR the two input bits to produce a propagate function, but we're going to do something different. We will logically OR the two together instead. I'll explain why shortly. This gives us the following propagate functions.

$$p_0 = A_0 + B_0$$
$$p_1 = A_1 + B_1$$
$$p_2 = A_2 + B_2$$
$$p_3 = A_3 + B_3$$

Now, the carry out for each full adder is 1 if either the input data generates a carry or if the carry input is 1 and the input data propagates the carry. We can express this as:

$c_0 = g_0 + p_0c_{in}$ (with no carry in this is just g_0)
$c_1 = g_1 + p_1c_0$
$c_2 = g_2 + p_2c_1$
$c_3 = g_3 + p_3c_2$

(Remember that the carry in for each full adder, except the least significant bit, is the carry out of the previous full adder.)

Finally, we can substitute in the carry values until every function is expressed solely in terms of the data inputs, or of the data inputs and the *g* and *p* values, which are constructed solely from the data inputs. Proceeding from the least significant carry bit, we get:

$$c_0 = g_0 + p_0c_{in}$$
$$c_1 = g_1 + p_1c_0 = g_1 + p_1g_0 + p_1p_0c_{in}$$
$$c_2 = g_2 + p_2c_1 = g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_{in}$$
$$c_3 = g_3 + p_3c_2 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_{in}$$

Before we get to the sum outputs, let's go back to why we use the OR function instead of the exclusive OR to implement the propagate functions. OR gates are faster than XOR gates and require less hardware in their internal designs. That alone makes the OR gate a better choice, *but only if the circuit still functions properly.* In this case, it does. Figure 4.29 shows two truth tables. The first (a) generates the carry out using the exclusive OR function. The second (b) uses the OR function. As the tables demonstrate, both produce the same value for the carry out for all possible input values.

A	B	C_{IN}	$g=AB$	$p=A \oplus B$	$C_{OUT}=g+pC_{IN}$
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	0	1	0
1	0	1	0	1	1
1	1	0	1	0	1
1	1	1	1	0	1

(a)

A	B	C_{IN}	$g=AB$	$p=A+B$	$C_{OUT}=g+pC_{IN}$
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	0	1	0
1	0	1	0	1	1
1	1	0	1	1	1
1	1	1	1	1	1

(b)

Figure 4.29: Carry out functions: (a) Using XOR to set *p*; (b) Using OR to set *p*;

The only time *p* is different in the two functions occurs when $A = B = 1$. The XOR sets *p* to 0 and the OR sets it to 1. However, in this case, *g* is 1, which sets the carry out to 1 regardless of the value of pC_{in}. This is why we can use the OR function instead of the XOR function.

Finally, we need to set the sum bits. We can do this as we did before, by exclusive ORing the two data bits and the carry input bit. As an alternative, we can exclusive OR the full adder's *g*, *p*, and carry in bits. Figure 4.30 shows the truth tables for both functions, which verify that both produce the same output for all possible input values.

A	B	C_{IN}	$S=A \oplus B \oplus C_{IN}$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

(a)

A	B	C_{IN}	$g=AB$	$p=A+B$	$S=g \oplus p \oplus C_{IN}$
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	0	1	1
1	0	1	0	1	0
1	1	0	1	1	0
1	1	1	1	1	1

(b)

Figure 4.30: Sum functions: (a) Using data and carry in; (b) Using *g*, *p*, and carry in

The complete design is shown in Figure 4.31. For simplicity, we used the non-expanded functions for the carry bits when designing this circuit.

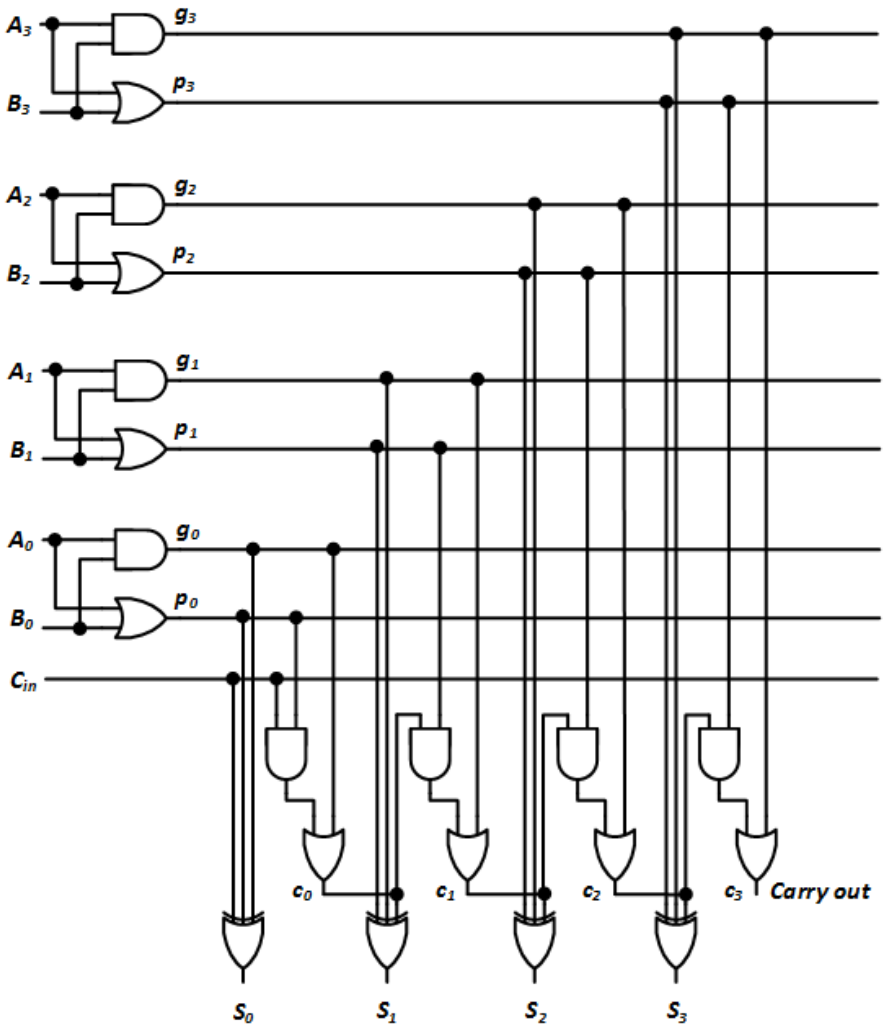


Figure 4.31: 4-bit carry-lookahead adder

4.7 Tri-state Buffers

In Chapter 3, we introduced the buffer, a simple component that outputs whatever value is input to it. In this section, we introduce a special type of buffer that is often used with buses, called a **tri-state buffer**. Before describing these buffers, however, we need to spend a brief amount of time talking about buses.

A bus is simply a wire used to transmit data. Many components can send data onto the bus (though only one at a time), and many can receive data from the bus (this can be more than one at a time, subject to fan-out limitations).

Buses can simplify a circuit by reducing the connections between components. For example, consider a highway with 100 entrances/exits. Each of these has an entrance ramp to go onto the highway, and an exit ramp to leave the highway. You can go from any entrance to any exit using this highway.

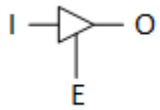
Now consider how to connect these locations without using a highway. In order to travel directly from any one location to any of the others, we would need to build a road directly between each pair of locations. This would total up to 4,950 roads. Furthermore, each location would need 99 entrance and exit ramps, one for each road to/from each destination. In complex digital systems, such as computers or microprocessor chips, this simply is not feasible. For this reason, many digital systems use buses.

To connect components to a bus, you can't just connect, say, the output of an AND gate straight onto the bus. That works fine when the AND gate is sending its output to other components via the bus, but not so well when it is not sending data through the bus. In this case, it would still be putting data onto the bus, even if another component was trying to use it. It would interfere with the data that should be on the bus, and components trying to read that data might not be able to read it correctly. We need a way to connect components to the bus when needed and to isolate them from the bus at all other times. This is the purpose of tri-state buffers.

The tri-state buffer and its truth table are shown in Figure 4.32. In addition to its data input, the tri-state buffer has an enable signal. When enabled, the tri-state buffer acts just like a traditional buffer. When it is not enabled, its output is the high-impedance state, denoted as Hi-Z.

E	I	O
0	X	Z
1	0	0
1	1	1

(a)



(b)

Figure 4.32: Tri-state buffer logic symbol and truth table

WATCH ANIMATED FIGURE 4.32

To understand how high impedance works, we need to go back to Ohm's Law. You may recall this law states that current equals voltage divided by resistance, or $I = V/R$. Well, that is correct when a circuit has only resistive elements, but it does not take into account the impedance introduced by inductive and capacitive elements. In practice, $I = V/Z$, where Z is the overall impedance of the circuit.

When a tri-state buffer is disabled, it has a very high impedance. As Z increases, and the voltage remains the same, the current decreases greatly, typically to some number of micro-Amperes, or millionths of an Ampere. A circuit can usually handle some number of disabled tri-state buffers placing micro-Amperes of current onto the bus and still have components read the value on the bus correctly. Essentially, disabled components act as if they are disconnected from the bus. As long as no more than one tri-state buffer sends data onto the bus at any time, the system should function as desired.

4.8 Summary

In this chapter, we have examined several types of logical functions that are commonly used in digital circuit design. Decoders enable one of their outputs based on the value that is selected, and encoders set an output based on which of its inputs is active. Multiplexers pass the selected input's value through to its output, and demultiplexers send a single input value to its selected output. Comparators and adders treat their inputs as numeric data. Comparators examine the data and set an output to indicate which input value is greater, or if both are the same. Adders output the arithmetic sum of their two inputs. Buffers do not modify their inputs, but can be used to overcome physical limitations inherent to digital electronics.

In the next chapter, we will use some of these components to implement more complex functions than those we saw in Chapter 3. We will also introduce two new components: the lookup ROM and the Programmable Logic Device.

Bibliography

Hayes, J. P. (1993). *Introduction to digital logic design*. Addison-Wesley.

Mano, M. M., & Ciletti, M. (2017). *Digital design: with an introduction to the Verilog HDL, VHDL, and SystemVerilog* (6th ed.). Pearson.

Mano, M. M., Kime, C., & Martin, T. (2015). *Logic & computer design fundamentals* (5th ed.). Pearson.

Nelson, V., Nagle, H., Carroll, B., & Irwin, D. (1995). *Digital logic circuit analysis and design*. Pearson.

Roth, J. C. H., Kinney, L. L., & John, E.B. (2020). *Fundamentals of logic design enhanced edition* (7th ed.). Cengage Learning.

Wakerly, J. F. (2018). *Digital design: Principles and practices* (5th ed.). Pearson.

Exercises

1. Design a 3 to 8 decoder based on the design of the 2 to 4 decoder presented in Figure 4.5(c).
2. The 74138 is a TTL chip that is a 3 to 8 decoder. Unlike the designs presented in this chapter, its outputs are active low. That is, the activated output is 0 and all the other outputs are 1. Modify your design for problem 1 so that it functions like the 74138.
3. Design a 5 to 32 decoder using one 2 to 4 decoder and four 3 to 8 decoders.
4. Design a 5 to 32 decoder using one 3 to 8 decoder and as many 2 to 4 decoders as are necessary. How many 2 to 4 decoders are needed?
5. Design the circuit to generate outputs *b*, *c*, *d*, *e*, *f*, and *g* of the BCD to 7-segment decoder.
6. Design a 16 to 4 encoder based on the design of the 8 to 1 encoder presented in Figure 4.12.
7. Design a 16 to 4 encoder using five 4 to 2 encoders.
8. Design a 32 to 5 encoder using four 8 to 3 encoders and combinatorial logic gates.
9. Design a 32 to 5 encoder using four 8 to 3 encoders and one 4 to 2 encoder.
10. Design a 16 to 4 priority encoder.
11. Design a 16 to 1 multiplexer based on the design of the 8 to 1 multiplexer presented in Figure 4.16.
12. Design a 32 to 1 multiplexer using four 8 to 1 multiplexers and one 4 to 1 multiplexer.
13. Design a 16 to 1 multiplexer using a 4 to 16 decoder.
14. Design a 1 to 8 demultiplexer using only basic logic gates.
15. Design a 1 to 16 demultiplexer using two 1 to 8 demultiplexers.
16. Show the value of each signal in the 4-bit comparator in Figure 4.22(b) as it compares *A*=1011 and *B*=1100.
17. Design a circuit that directly compares two 2-bit values.
18. Show the value of each gate input and output for the half adders in Figure 4.24(a) and (b) as they add *A*=1 and *B*=1.
19. Show the values of each gate input and output for the full adder circuit in Figure 4.26(c) as it adds *A*=1, *B*=0, and C_{in}=1.

20. Show the input and output values of each full adder in Figure 4.27 as it adds *A*=11 and *B*=10.

21. For an 8-bit carry-lookahead adder, give the equations for g_4, g_5, g_6, and g_7; p_4, p_5, p_6, and p_7; and c_4, c_5, c_6, and c_7.

22. If the maximum propagation delays for 2-input AND gates, 3-input OR gates, and 2-input XOR gates are 20ns, 44ns, and 30ns, respectively[1], what is the maximum propagation delay for the full adder circuit in Figure 4.26(c)?

23. Design a circuit to meet the following specification. The circuit has four data inputs labeled *A*, *B*, *C*, and *D*. Each of these is input to a tri-state buffer. The outputs are connected together to generate output *O*. There are two control signals, *E* and *F*, which determine the input value that is sent to output *O*, as follows.

E	*F*	*O*
0	0	*A*
0	1	*B*
1	0	*C*
1	1	*D*

Use one of the components introduced in this chapter to generate the enable signals for the tri-state buffers.

[1] Propagation delays shown are for the 74LS08 (AND), 74LS32 (OR, two gates), and 74LS86 (XOR) gates.

Chapter 5

More Complex Functions

Chapter 5: More Complex Functions

The components introduced in the previous chapter have many real-world applications. Decoders can be used in computer systems to elect one of many memory chips based on the address specified by the microprocessor. Multiplexers can select one of several inputs to be sent to another component. Encoders can be used to specify the row and column of a key that is pressed on a keypad. The possible uses of these components is limited only by the creativity of the engineer using them.

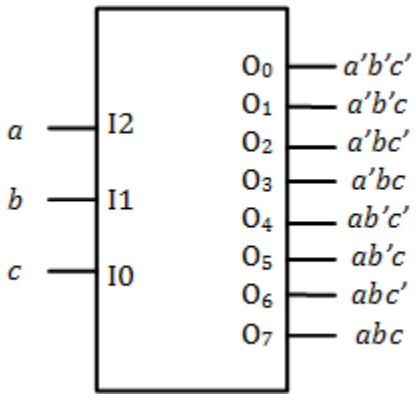
In this chapter, we look at a limited application, realizing Boolean functions. We start by focusing on two components: decoders and multiplexers. Each can be used to realize functions in a relatively straightforward manner. Then we introduce ROMs, read-only memory chips. Although memory chips are most commonly used to store data and programs in computer systems, ROMs can also be configured to realize combinatorial functions by serving as lookup tables.

5.1 Implementing Functions using Decoders

Among their many uses in digital logic design, decoders can be used to realize functions by following a standard procedure. To illustrate how this works, consider the 3 to 8 decoder and its truth table shown in Figure 5.1. Output O_0=1 when a=0, b=0, and c=0. Alternatively, we can say that O_0=$a'b'c'$. We can do this for each output, which gives us the functions shown in the last column of the table.

a	b	c	O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7	Function
0	0	0	1	0	0	0	0	0	0	0	$a'b'c'$
0	0	1	0	1	0	0	0	0	0	0	$a'b'c$
0	1	0	0	0	1	0	0	0	0	0	$a'bc'$
0	1	1	0	0	0	1	0	0	0	0	$a'bc$
1	0	0	0	0	0	0	1	0	0	0	$ab'c'$
1	0	1	0	0	0	0	0	1	0	0	$ab'c$
1	1	0	0	0	0	0	0	0	1	0	abc'
1	1	1	0	0	0	0	0	0	0	1	abc

(a)



(b)

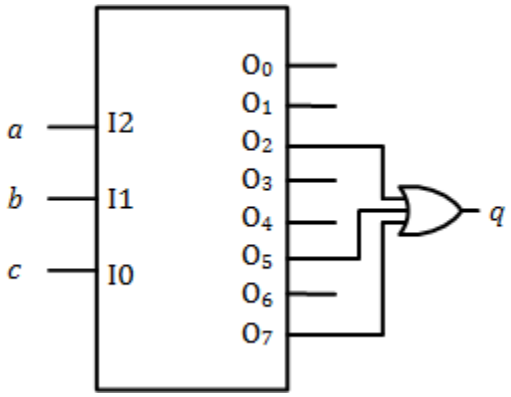
Figure 5.1: 3 to 8 decoder: (a) Truth table; (b) Decoder

Now comes a key point that underlies this whole design methodology. **Each of the outputs represents a minterm of the inputs.** To implement a function as a sum of products, we can input the function inputs to the decoder and logically OR the desired outputs to realize the function.

Let's look at a couple of examples we worked out earlier to illustrate this process. In Figure 5.2 (a), we have the truth table for function *q*. This function is equal to 1 when *a*=0, *b*=1, and *c*=0; *a*=1, *b*=0, and *c*=1; or *a*=1, *b*=1, and *c*=1. Alternatively, *q*=*a'bc'*+*ab'c*+*abc*. In the previous chapter, we combined the last two terms and implemented the function as *q*=*a'bc'*+*ac*, but we can't do that when we use a decoder to realize the function. Since each decoder output represents one minterm, we must express the function solely using minterms. The three decoder outputs corresponding to *a'bc'*, *ab'c*, and *abc* are O_2, O_5, and O_7, respectively. Logically ORing these three outputs together gives us the value for function *q*. This circuit is shown in Figure 5.2 (b).

a	*b*	*c*	*q*
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

(a)



(b)

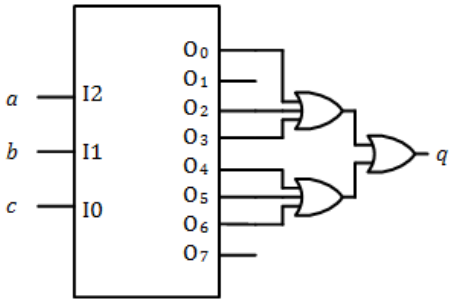
Figure 5.2: Implementing the function *q*=*a'bc'*+*ab'c*+*abc* using a decoder: (a) Truth table; (b) Decoder circuit implementation

WATCH ANIMATED FIGURE 5.2.b

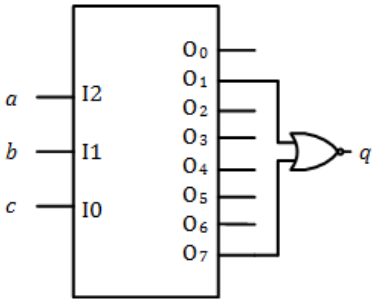
Now consider another example function with the truth table shown in Figure 5.3 (a). There are six minterms which set function *q* to 1, and *q* can be expressed as *q*=*a'b'c'*+*a'bc'*+*a'bc*+*ab'c'*+*ab'c*+*abc'*. We can logically OR the decoder outputs corresponding to these minterms, which are O_0, O_2, O_3, O_4, O_5, and O_6, respectively, to realize function *q*. Figure 5.3 (b) shows this circuit. Since there are no standard TTL chips with 6-input OR gates, this circuit uses OR gates with lower fan-in values to realize the same function.

a	*b*	*c*	*q*
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

(a)



(b)



(c)

Figure 5.3: Implementing the function $q=a'b'c'+a'bc'+a'bc+ab'c'+ab'c+abc'$ using a decoder: (a) Truth table; (b) Decoder circuit implementation; (c) Decoder circuit implementation using inverse function

WATCH ANIMATED FIGURE 5.3.b and c

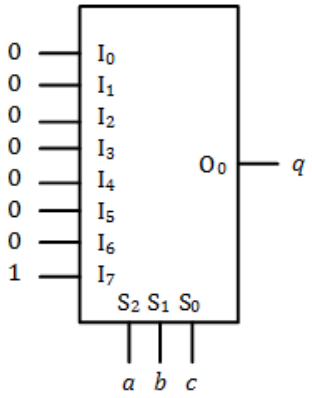
Just as we did in a previous chapter, we can realize the inverse of this function (using maxterms) and then invert the output to realize the original function. For this function, this is equivalent to saying that $q'=1$ if either O_1 ($a'b'c$) or O_7 (abc) is equal to 1. We can logically OR O_1 and O_7 to realize q', and then invert this value to generate q. The simplest way to do this is to NOR O_1 and O_7, which generates q directly. This circuit is shown in Figure 5.3 (c).

5.2 Implementing Functions using Multiplexers

Just as decoders can be used to implement combinatorial logic functions, multiplexers can also be used to implement these functions. To see how this works, let's start with the 3-input AND function. Its truth table is shown in Figure 5.4 (a). Output $q=1$ only when $a=1$, $b=1$, and $c=1$, or $q=abc$.

a	*b*	*c*	*q*
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

(a)



(b)

Figure 5.4: *q*=*abc*: (a) Truth table; (b) Implementation using an 8 to 1 multiplexer

To implement a function using a multiplexer, we connect the function inputs (*a*, *b*, and *c*, for this function) to the select inputs of the multiplexer. Then we connect the value of the function for each possible set of function inputs to the corresponding multiplexer input. For example, consider the circuit shown in Figure 5.4 (b). When *a*=0, *b*=0, and *c*=0, the multiplexer will pass whatever value is at input I_0 directly to output O_0. Since *q*=0 for these values, we want to have 0 at this multiplexer input. In fact, we want to have 0 connected to all inputs except I_7. That input is selected when *a*=1, *b*=1, and *c*=1. For these values, *q*=1, not 0, and we need to set the value of this input to 1.

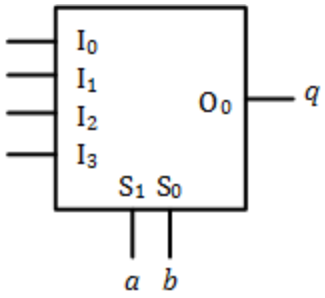
Examining the values of *q* in the truth table, we can see that they are exactly the same as the data inputs to the multiplexer. In essence, this circuit is not computing the value of the function; it is looking up the value by selecting the multiplexer input.

As the number of inputs to a function increases, the size of a multiplexer needed to implement the function increases exponentially. For example, a function of 10 variables would require a multiplexer with 10 select signals and 2^{10}=1024 data inputs. This is simply not feasible. However, we can still use a smaller multiplexer by having some of the function inputs select a multiplexer input and using the other function inputs to generate the multiplexer data inputs. At first, this may seem confusing, but there is a straightforward method to design such a circuit.

To show how this works, let's redesign our circuit to realize the function *q*=*abc* using a 4 to 1 multiplexer. The truth table for this function is repeated in Figure 5.5 (a). The first part of the design is to connect the function inputs to the select signals of the multiplexer. With three function inputs and two select signals, we must choose which function inputs to connect to the select signals, and which to exclude. I chose to connect inputs *a* and *b* to the select signals and to exclude input *c*. You can choose any two inputs and still create a working circuit, but choosing the leftmost bits in the truth table can make your work slightly easier for reasons we'll discuss shortly. The circuit, so far, is shown in Figure 5.5 (b).

a	*b*	*c*	*q*
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

(a)



(b)

Figure 5.5: *q*=*abc*: (a) Truth table; (b) Assignment of function inputs to select signals

Now we have to determine the values of the data inputs to the multiplexer. To do this, we are going to take a divide and conquer approach. We return to the truth table and divide it into several separate tables, one for each multiplexer input. Consider multiplexer input I_0. It is selected and its value is passed through to output O_0 when S_1=0 and S_0=0. For our circuit, this occurs when *a*=0 and *b*=0. We take all the rows of our truth table with *a*=0 and *b*=0 and make them into a separate table. We do this for each multiplexer input. (This is why I used the leftmost bits of the truth table as the select inputs, so that each of these tables would consist of consecutive rows.) The tables for this circuit are shown in Figure 5.6 (a).

I_0:

a	*b*	*c*	*q*
0	0	0	0
0	0	1	0

I_1:

a	*b*	*c*	*q*
0	1	0	0
0	1	1	0

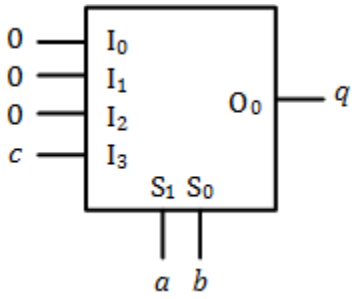
I_2:

a	*b*	*c*	*q*
1	0	0	0
1	0	1	0

I_3:

a	*b*	*c*	*q*
1	1	0	0
1	1	1	1

(a)



(b)

Figure 5.6: *q*=*abc*: (a) Partitioned truth tables; (b) Final design using a 4 to 1 multiplexer

WATCH ANIMATED FIGURE 5.6.b

Finally, we must create and implement a function for each of the multiplexer data inputs. To do this, we express *q* as a function of the function inputs we excluded earlier, that is, all the function inputs that are *not* input to the multiplexer's select signals. We are already taking the select values into account when we partition the tables. For example, let's look at input I_0. This value is only passed through to output O_0 when the select signals are both 0, or *a*=0 and *b*=0. If *a* and *b* have any other values, this multiplexer input will not be passed through

to the output, so we don't care what its value is for any other values of *a* and *b*. We may, however, need to take into account the value of other function inputs, *c* in this example.

For I_0, I_1, and I_2, this is not the case. Each of these functions always sets *q*=0, so we simply hardwire a logic 0 to these three multiplexer inputs. For I_3, however, things are different. When *c*=0, we want to set output *q* to 0, and when *c*=1 we want to set *q* to 1. The simplest way to do this is to set I_3=*c*. When *a*=1, *b*=1, and *c*=0, *a* and *b* select multiplexer data input I_3, and its input value is 0 (I_3=*c*=0). When *a*=1, *b*=1, and *c*=1, we again select I_3, but this time its value is 1. The final design for this circuit is shown in Figure 5.6 (b).

Before we leave this topic, there are a couple of points I want to emphasize. First, each function input is used either as a select signal or to generate the data inputs to the multiplexer, but not both. If it is in both places, your circuit can be simplified.

The second point concerns the choice of function inputs used for the multiplexer select signals. I used the leftmost function inputs in the truth table because that always creates separate tables consisting of lines of the truth table that are next to each other. However, this does not always result in an optimal design (although it will always produce a correct design, sometimes with more hardware than absolutely necessary).

Figure 5.7 (a) shows the truth table for a function. Let's say we implement this function using a 4 to 1 multiplexer and connect *a* and *b* to the multiplexer select inputs. Using the procedure we just demonstrated, we would create the four separate tables shown in Figure 5.7 (b). These give us input functions I_0=0, I_1=*c*′, I_2=*c*, and I_3=*c*′. The circuit to realize the function is shown in Figure 5.7 (c).

a	*b*	*c*	*q*
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

(a)

I_0:

a	*b*	*c*	*q*
0	0	0	0
0	0	1	1

I_1:

a	*b*	*c*	*q*
0	1	0	1
0	1	1	0

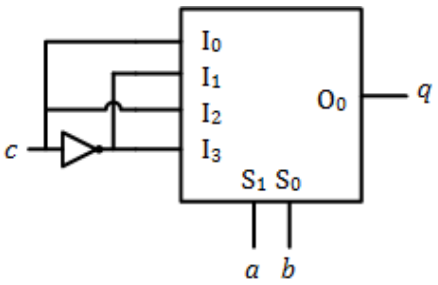
I_2:

a	*b*	*c*	*q*
1	0	0	0
1	0	1	1

I_3:

a	*b*	*c*	*q*
1	1	0	1
1	1	1	0

(b)



(c)

Figure 5.7: Function *q*: (a) Truth table; (b) Partitioned truth tables; (c) Implementation using a 4 to 1 multiplexer

WATCH ANIMATED FIGURE 5.7.c

Now let's design this circuit again, but this time using *b* and *c* as the multiplexer inputs. The truth table, repeated in Figure 5.8 (a) is partitioned as shown in Figure 5.8 (b). For this design, we have I_0=0, I_1=1, I_2=1, and I_3=0. The final design is shown in Figure 5.8 (c).

a	*b*	*c*	*q*
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

(a)

I_0:

a	*b*	*c*	*q*
0	0	0	0
1	0	0	0

I_1:

a	*b*	*c*	*q*
0	0	1	1
1	0	1	1

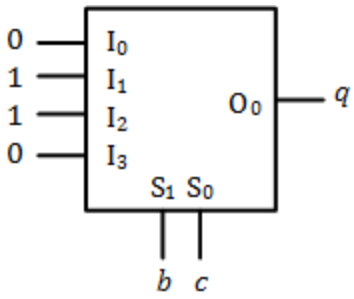
I_2:

a	*b*	*c*	*q*
0	1	0	1
1	1	0	1

I_3:

a	*b*	*c*	*q*
0	1	1	0
1	1	1	0

(b)



(c)

Figure 5.8: Function *q*: (a) Truth table; (b) Partitioned truth tables using *b* and *c* as multiplexer select inputs; (c) Implementation using a 4 to 1 multiplexer

WATCH ANIMATED FIGURE 5.8.c

Comparing these two circuits, we see that some things do not vary between the designs. The multiplexer is always the same size, output O_0 always gives the value of *q*, and two of the function inputs are connected to the multiplexer select inputs. The only thing we can change by choosing which function inputs are connected to the multiplexer select inputs is the functions (and thus the circuits needed to generate the functions) of the multiplexer data inputs. The first design needed one NOT gate to generate these inputs, while the second design needed no logic gates at all. This example was constructed just to illustrate this concept. For more complex functions, the reduction in hardware needed to generate the multiplexer data inputs can be much greater than in this example.

5.3 ROM Lookup Circuits

In the previous section, I mentioned that the multiplexer circuit looks up the value of a function rather than calculating the value. A read-only memory, or ROM, can be used to generate values in much the same way. We'll look at ROMs and other memory components in more detail later in this book. For now, this section introduces just enough information about ROMs for us to use them to realize logic (or arithmetic) functions.

5.3.1 The Very Basics of ROMs

Memory chips are aptly named. They "remember" values that you can retrieve when needed, that is, they store values. In computer systems, memory chips may store the instructions that comprise a program, such as Microsoft Word, or data used by a program, for example, a Word document. If your computer has a solid state hard drive, that drive is constructed from memory chips. Flash drives are also built using memory chips.

Memory chips can be divided into two groups, depending on what happens to their data values when power goes off. **Volatile** memory only keeps its values when power is on. Once power goes off, all the data values it had stored are lost. The most common type of volatile memory is random access memory, or RAM. If a computer manufacturer says their computer has 16GB of memory, or whatever value they say, they are talking about RAM.

ROM (read only memory), on the other hand, is a **non-volatile** memory. When power is off, the data within the memory chip remains unchanged. If you turn power off and later turn it back on, you can access the exact data that was there before you turned the power off, which is why it is useful for things like flash drives, and for lookup tables for logic and arithmetic functions.

To be clear, even though ROM chips store their data when power is off, you can't read this data from the chip unless power is on. The memory chips need to have power in order to output their data. In addition to the memory cells that actually store the data, there are other components used by the chip to access specific memory cells and to perform other functions needed to read data from the memory chip. These components cannot function unless power is on.

So, if a ROM is a read-only memory, how do you write data into it in the first place? There are several types of ROMs, each with its own way to store data. Some chips are fabricated with the data stored in its memory cells, and others use special programming hardware to write values into these cells. We'll talk about the writing process in more detail in Chapter 10. For our ROM lookup circuits, we will assume that, somehow, the data for our lookup function has already been written into the ROM.

5.3.2 Memory Chip Configuration

Regardless of the type of memory chip, its inputs and outputs work in a similar manner. We input an **address** to the memory chip. This is just some number of Boolean inputs that are used to select one unique memory cell, or **location**, within the chip. On a memory chip, each address bit is assigned to a unique pin. The number of address inputs depends on how many memory cells are in the chip. Remember from our discussion of numeric representations in Chapter 1 that n bits can represent 2^n unique values. Here, 2^n is the number of memory locations within a chip and n is the number of address bit inputs. For example, consider a memory chip with 1024 locations. Since $1024=2^{10}$, this chip must have 10 address bit inputs. If the address inputs are 00 0000 0000, we will access memory location 0 within the chip. Address 00 0000 0001 accesses location 1, and so on, up to address 11 1111 1111 accessing location 1023.

There are different ways to label these address inputs. For consistency, we will use a notation called **little Endian** throughout this book. The least significant bit is labeled A_0. The next bit is A_1, and so on. The most significant bit is A_{n-1}. For our 1024-location ROM, $1024=2^{10}$, so n=10 and these bits are A_9 to A_0.

The memory chip may have one bit of data at each memory location, or it may have more bits. Each bit must have its own data bit output. For a memory chip with m data outputs, we again use little Endian notation, labeling them from D_{m-1} (most significant bit) to D_0 (least significant bit). As with the address inputs, each data output is assigned to a unique pin on the memory chip.

For RAMs, these data pins are used both to input and output data. Programming hardware may also use these bits to store data in a ROM. We won't worry about that for now since we are assuming the data is already stored in the ROM.

The size of a memory chip is said to be 2^nxm. For our 1024-location memory chip, if it has eight output bits, we would say it is of size 1024x8.

There are other pins on the memory chips that are important for other uses of these chips. One of these is the chip enable, or chip select, input. Going back to our computer with 16GB of memory, it is unlikely that this computer has a single chip with 16GB locations. It is much more likely that this 16GB of memory is constructed using smaller chips. Different addresses in the 16GB range are assigned to different chips. When we want to get data from a specific location, we need to get the data from one specific chip. We use the chip enable input to choose the correct chip. We also use the chip enable inputs of the other chips to make sure we're not choosing those chips. Fortunately for us, ROM lookup circuits typically include just one single memory chip, so we can always enable the chip in our circuits.

5.3.3 Using ROMs in Lookup Circuits

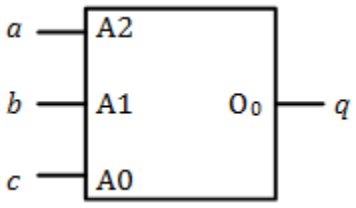
All ROM lookup circuits follow the same basic design procedure, summarized in the following three steps.

1. All function inputs are connected to the address inputs of the ROM.
2. Function values are output on the ROM's data outputs.
3. Each memory location in the ROM stores the value of the function for its input values.

I think the best way to illustrate how this works is to jump right in with an example. Let's start with a 3-input AND function. Its three inputs are *a*, *b*, and *c*, and its output *q* is 1 only when all three inputs are equal to 1. Its truth table is shown in Figure 5.9 (a).

a	*b*	*c*	*q*
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

(a)



(b)

Address	Data
000 (0)	0
001 (1)	0
010 (2)	0
011 (3)	0
100 (4)	0
101 (5)	0
110 (6)	0
111 (7)	1

(c)

Figure 5.9: 3-input AND function *q*=*abc*: (a) Truth table; (b) Lookup ROM circuit; (c) ROM data

WATCH ANIMATED FIGURE 5.9.b

Following the procedure introduced at the beginning of this subsection, first we connect the function inputs, *a*, *b*, and *c*, to the address inputs of the ROM, A_2, A_1, and A_0, respectively. Next, we connect the function output, *q*, to data output D_0. Since we have three address inputs and one data output, *n*=3 and *m*=1, and the size of our ROM is 2^3x1, or 8x1. That is, the ROM has eight memory locations, each with one bit of data. The lookup ROM and its connections are shown in Figure 5.9 (b).

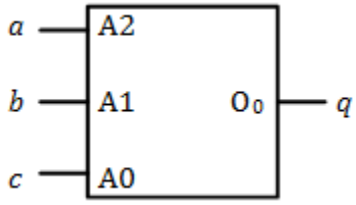
Finally, we must determine the data values to store in every memory location. Let's start with address 0, or 000 in binary. We will read data from this memory location when *a*=0, *b*=0, and *c*=0. When the function inputs have these values, *q*=*abc*=0^0^0=0, so we store a 0 at memory location 0. We follow the same procedure to calculate the values at the other locations. All other locations also have the value 0, except for location 7 (111), which has the value *q*=1^1^1=1. These values are shown in Figure 5.9 (c).

Notice that the two tables in Figure 5.9 are essentially the same. This happens because we connected our function inputs directly to the address inputs of the ROM, that is, A_2=*a*, A_1=*b*, and A_0=*c*.

Now, instead of an AND function, let's say we want to implement an OR function. The truth table for this function is shown in Figure 5.10 (a). Since this function has the same inputs and outputs (though not the same output values) as the AND function, its connections are the same as those for the AND circuit, as shown in Figure 5.10 (b). **This is a really important point and we'll come back to this shortly.** Finally, we determine the values to be stored in each memory location. For this function, location 0 (*a*=0, *b*=0, and *c*=0) has the value 0 and all other locations have the value 1, as shown in Figure 5.10 (c).

a	*b*	*c*	*q*
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

(a)



(b)

Address	Data
000 (0)	0
001 (1)	1
010 (2)	1
011 (3)	1
100 (4)	1
101 (5)	1
110 (6)	1
111 (7)	1

(c)

Figure 5.10: 3-input OR function *q*=*a*+*b*+*c*: (a) Truth table; (b) Lookup ROM circuit; (c) ROM data

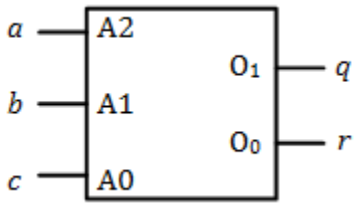
WATCH ANIMATED FIGURE 5.10.b

Now, back to our really important point. Both functions use exactly the same hardware with identical input and output connections. **We change the function of the circuit by changing the contents of the ROM.** By doing so, we can realize any function of the inputs using the same circuit and modifying the data stored in the ROM.

Finally, it is possible to use a lookup ROM to generate more than one function. To do this, you need one data output for each function. After assigning each function to an output, we determine the value for each location for each function and store those values in the ROM. Consider a lookup ROM that generates both the AND and OR functions for three input values. Here, $q=abc$ and $r=a+b+c$. Figure 5.11 (a) shows the combined truth table for the two functions. The circuit diagram is developed just as before. The ROM still has three inputs because the functions each use the same three function inputs, *a*, *b*, and *c*. However, with two functions to generate, the ROM now must have two outputs, so the ROM is of size 8x2. The circuit diagram is shown in Figure 5.11 (b). To complete the design, we determine the values to be output for both functions. These are shown in Figure 5.11 (c).

a	*b*	*c*	*q*	*r*
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(a)



(b)

Address	Data
000 (0)	00 (0)
001 (1)	01 (1)
010 (2)	01 (1)
011 (3)	01 (1)
100 (4)	01 (1)
101 (5)	01 (1)
110 (6)	01 (1)
111 (7)	11 (3)

(c)

Figure 5.11: 3-input AND $q=abc$ and OR $q=a+b+c$ functions: (a) Truth table; (b) Lookup ROM circuit; (c) ROM data

WATCH ANIMATED FIGURE 5.11.b

5.3.4 Arithmetic Lookup ROM Circuits

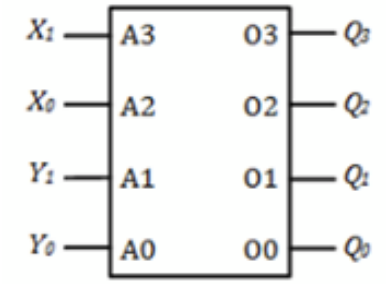
The previous examples used a lookup ROM to generate the output of a logic function. But there's nothing that stops us from using a lookup ROM to output other types of data values. The ROM doesn't know what type of data it outputs. It gets an address and outputs the data stored at that address. By specifying the values input to the address pins, the contents of each memory location within the ROM, and the connections from its outputs, the designer specifies its function and the type and format of its data.

Arithmetic circuits are one type of circuit that can be implemented using a lookup ROM. As in the previous examples, the function inputs are connected to the address pins of the ROM. Instead of being Boolean logic values, however, these bits represent binary numbers. More than one bit may be used to represent a number, and each bit is connected to a separate address bit. Similarly, we may need several output bits to represent a single value.

To see how this works, consider a lookup ROM that multiplies two 2-bit numbers, *X* and *Y*. Each input number has one of four possible values: 0 (00), 1 (01), 2 (10), or 3 (11). We'll represent *X* and *Y* as the two bit values X_1X_0 and Y_1Y_0. The product output by the ROM ranges

from 0 (0x0) to 9 (3x3), or 0000 to 1001 in binary. So, we need four bits to represent the product, which we'll call *Q* and represent as $Q_3Q_2Q_1Q_0$ in binary.

Just as before, we'll connect our function inputs (X_1, X_0, Y_1, and Y_0) to the ROM's address inputs and the product outputs (Q_3, Q_2, Q_1, and Q_0) to its data outputs. The circuit designer can choose which function input and output bits are connected to each address and data pin on the ROM. (Hint: Keeping the bits for each operand input and the product output together and in order will make it easier to calculate the values to be stored in the lookup ROM.) For this example, I decided to connect input value *X* to address bits A_3 (X_1) and A_2 (X_0), and input *Y* to address bits A_1 (Y_1) and A_0 (Y_0). Output *Q* is connected to data outputs D_3 (Q_3), D_2 (Q_2), D_1 (Q_1), and D_0 (Q_0). This is shown in Figure 5.12 (a).



(a)

Address $X_1X_0Y_1Y_0$	Data $Q_3Q_2Q_1Q_0$
0000 (0)	0000 (0)
0001 (1)	0000 (0)
0010 (2)	0000 (0)
0011 (3)	0000 (0)
0100 (4)	0000 (0)
0101 (5)	0001 (1)
0110 (6)	0010 (2)
0111 (7)	0011 (3)
1000 (8)	0000 (0)
1001 (9)	0010 (2)
1010 (10)	0100 (4)
1011 (11)	0110 (6)
1100 (12)	0000 (0)
1101 (13)	0011 (3)
1110 (14)	0110 (6)
1111 (15)	1001 (9)

(b)

Y / X	00	01	10	11
00	0	0	0	0
01	0	1	2	3
10	0	2	4	6
11	0	3	6	9

(c)

Figure 5.12: 2-bit multiplier lookup ROM: (a) Lookup memory circuit; (b) ROM contents; (c) Calculating values using a grid

WATCH ANIMATED FIGURE 5.12.a

To complete the design of this circuit, we must specify the contents of the ROM. We do exactly what we've done all along; we look at each possible set of input values and figure out what output values they should produce. For this circuit, we can start with X=0 and Y=0, or X_1=0, X_0=0, Y_1=0, and Y_0=0. When we multiply these values, we access memory address 0000, or 0. The product of X and Y, 0*0=0, must be stored in that location. Doing this for all possible values of X and Y gives us the data shown in Figure 5.12 (b).

It may be easier to visualize this if we use a grid to show these values, such as the one shown in Figure 5.12 (c). Each row corresponds to a unique value of X and each column represents a specific value of Y. The value at each location in the grid is the product of X and Y for its row and column. We can concatenate X and Y to give the address associated with each entry. For example, the third entry in the fourth row has X=11 and Y=10, and a data value of 0110. This shows that address 1110 has the value 0110, generating the output value 6 when we multiply 3x2.

5.4 Summary

In this chapter, we examined several ways to design combinatorial logic circuits to realize Boolean functions. Decoders can be used to generate the minterms of their inputs, and we can logically OR the desired minterms to create a sum of products realization of the function. A multiplexer can be configured to select the correct value of a function for any set of input values and output it directly. A ROM can be used to look up one or more function values. The function inputs specify the address and the data outputs produce the function values. Lookup ROMs can be used to realize Boolean or arithmetic functions.

In Chapter 10, we'll introduce programmable logic devices. These components have many logic gates on a single chip. The design engineer can configure the connections between gates to realize the desired function. Programmable logic devices can reduce the number of chips in a design, as well as the amount of wiring and power needed in a circuit.

This concluded the material on combinatorial logic. In the next chapter, we begin our discussion of sequential logic. Unlike combinatorial logic, which produces outputs based solely on the current value of its inputs, sequential logic outputs are based on both the current inputs and the current state, which in turn is based on previous values of the inputs. This may seem confusing at first, but it will become clearer as we examine the basic components of sequential logic and how they are used, along with combinatorial logic components, to realize sequential designs.

Bibliography

Hayes, J. P. (1993). *Introduction to digital logic design*. Addison-Wesley.

Mano, M. M., & Ciletti, M. (2017). *Digital design: with an introduction to the Verilog HDL, VHDL, and SystemVerilog* (6th ed.). Pearson.

Mano, M. M., Kime, C., & Martin, T. (2015). *Logic & computer design fundamentals* (5th ed.). Pearson.

Nelson, V., Nagle, H., Carroll, B., & Irwin, D. (1995). *Digital logic circuit analysis and design*. Pearson.

Roth, J. C. H., Kinney, L. L., & John, E.B. (2020). *Fundamentals of logic design enhanced edition* (7th ed.). Cengage Learning.

Skahill, K. (1996). *VHDL for programmable logic*. Addison-Wesley.

Wakerly, J. F. (2018). *Digital design: Principles and practices* (5th ed.). Pearson.

Exercises

1. Design a circuit to realize the following truth table using a decoder and an OR gate.

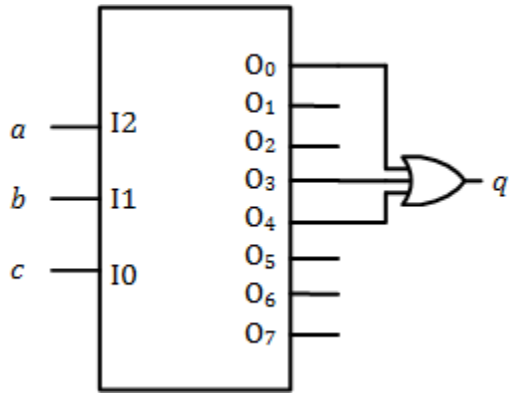
a	*b*	*c*	*q*
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

2. Design a circuit to realize the following truth table using a decoder and an OR gate.

a	*b*	*c*	*q*
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

3. Repeat problem 2, this time using a NOR gate to generate *q* instead of an OR gate.
4. Design circuits to perform the following functions for three inputs using a decoder.
    a. AND
    b. OR
    c. XOR
    d. NAND
    e. NOR
    f. XNOR

5. Show the truth table and minimal function realized by the following circuit.



6. Using a 4 to 16 decoder, design a circuit to generate output *a* of the BCD to 7-segment decoder.

7. Design a circuit to realize the following truth table using a multiplexer.

a	*b*	*c*	*q*
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

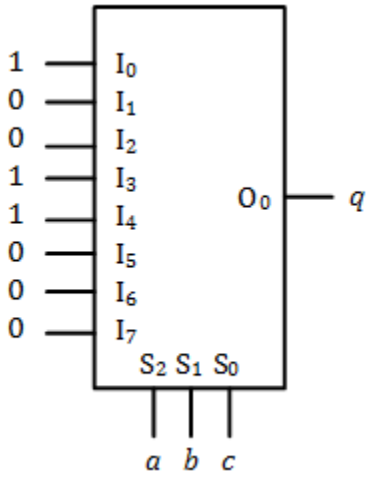
8. Design a circuit to realize the following truth table using a multiplexer.

a	*b*	*c*	*q*
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

9. Design circuits to perform the following functions for three inputs using a multiplexer.

    a. OR
    b. XOR
    c. NAND
    d. NOR
    e. XNOR

10. Show the truth table and minimal function realized by the following circuit.



11. Using a 16 to 1 multiplexer, design a circuit to generate output *a* of the BCD to 7-segment decoder.

12. Design a circuit to realize the following truth table using a ROM. Show the circuit and the contents of the ROM.

a	*b*	*c*	*q*
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

13. Design a circuit to realize the following truth table using a ROM. Show the circuit and the contents of the ROM.

a	b	c	q
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

14. Design a circuit to realize the following truth table using a ROM. Output q_1q_0 is the sum of a, b, and c. Show the circuit and the contents of the ROM.

a	b	c	q_1	q_0
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

15. Design a single circuit to perform the following functions for three inputs using a ROM. Show the circuit and the contents of the ROM.

    a. AND
    b. OR
    c. XOR
    d. NAND
    e. NOR
    f. XNOR

16. Repeat Problem 15 for functions with four inputs.

17. Show the truth table and minimal function realized by the following circuit and contents of memory.

Address	*Data*
000	1
001	0
010	0
011	1
100	1
101	0
110	0
111	0

18. Design a BCD to 7-segment decoder using a ROM and show the contents of memory. Segments *a* through *g* are connected to outputs O_6 to O_0, respectively.

19. A ROM has three address bits and three data bits. The inputs, *a*, *b*, and *c* form a 3-bit binary value and are input to the ROM as in problem 17. The outputs of the ROM realize the function (*abc* + 1) MOD 8. Show the contents of the ROM.

PART III

Sequential Logic

Chapter 6

Sequential Components

Chapter 6: Sequential Components

In Part II of this book, we introduced combinatorial logic. Combinatorial circuits generate outputs that depend *only* on the current values of the inputs. It doesn't matter what value the inputs were set to previously; the circuit only takes the current values into consideration when generating its outputs.

Sequential circuits, in contrast, set their outputs based on both the values of the inputs and its current status, or current state. The same input values could produce different outputs if the circuit is in one state or another.

Consider a classroom filled with students. If the instructor asks the students to add 4 and 5, every student should come up with the same answer, 9. Their calculations depend only on the values of the two inputs (4 and 5) and that we are adding them together. It does not matter what the students were doing before the instructor asked them this question, whether they were adding other numbers, performing a different arithmetic operation on the same or other numbers, or watching online videos.

Now the instructor tells each student to stand up, take one step to the left, and sit back down wherever they are. (For this example, every student has a seat available one step to their left and nobody starts next to a wall on their left.) The students do not end up all sitting in the same seat; if they follow directions properly, each student ends up in a different seat. Each student received the same input (move one step to the left), but because they started in different locations (current states), they ended up in different places.

As you've probably guessed, adding 4+5 is analogous to a combinatorial circuit, whereas stepping to the left is more like a sequential circuit.

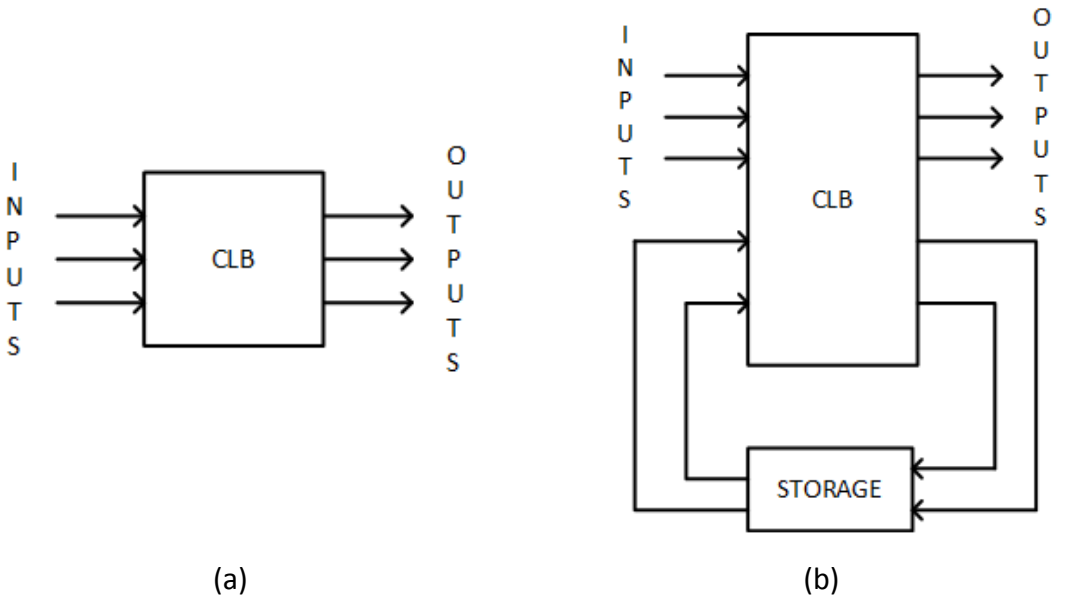
In order to access its current state, a sequential circuit must store this information somewhere. We need digital components that can hold that information and make it available to us as needed. That is largely the focus of this chapter.

The rest of this chapter is organized as follows. First, we examine basic models for sequential circuits. We introduce synchronous circuits, which use a periodic signal called a **clock** to synchronize the flow of data through the circuit. The rest of the chapter introduces two classes of circuits that can store data: latches and flip-flops. We will look at several types of each one, both their internal designs and external functions. We will expand on this in future chapters to create more complex components and more advanced sequential circuits.

6.1 Modeling Sequential Circuits

As already noted, combinatorial circuits generate their outputs based solely on the values of their inputs. Figure 6.1 (a) shows a very generic representation of a combinatorial circuit. The circuit has inputs and combinatorial components, and it generates outputs. The inputs and outputs are set to binary values. The combinatorial logic block may consist of fundamental logic gates, more complex components, and other digital logic. The combinatorial block has one restriction: it cannot contain any components that store information. When the values of the inputs change, the outputs and the values within the combinatorial logic block change to give the correct output for these input values. It doesn't matter what outputs and values within the

logic block were generated previously. The new values depend only on the current input values. Specific input values always produce the same output values.



(a) (b)

Figure 6.1: Generic models: (a) Combinatorial circuit; (b) Sequential circuit.

For sequential circuits, outputs are based on both the current input values and the previous state of the circuit. If a circuit inputs the exact same values at two different times, and the circuit is in a different state at these two times, it may produce different output values. In order to do this, a sequential circuit must have a way to store the current state and to use that state, along with the input values, to generate both the output values of the current state and the value of the new state. Figure 6.1 (b) shows the very generic representation of a sequential circuit.

Everything in this circuit is similar to the combinatorial logic circuit except for the block labeled STORAGE in Figure 6.1 (b). So far, we haven't seen any circuits that can lock in and store data. There are two main classes of circuits we use for this purpose: **latches** and **flip-flops**. We'll look at these in much greater detail in the remaining sections of this chapter. For now, it is enough to know that there is a way to store the state.

This leads us to an obvious question – what is a state? – which we address in the following subsection.

6.1.1 What Is a State?

In its simplest form, a state is just a way to represent the current condition of a circuit. Each condition is its own state. We generate the outputs based on the values of the inputs and the current state, and we also generate the new value of the current state (often called the **next state**) based on these same values. We can model any sequential circuit as a **finite state**

machine. Finite state machines is a large enough topic to warrant its own chapter, Chapter 8 in this book.

To illustrate how this works on a generic level, consider the following specification. We want to design a circuit that receives a single data input. When that input has been equal to 1 for a total of three times, we want to set a single output to 1 and start the whole process again. To do this, we need to check the input to see if it is 1. We also need to keep track of how many 1s we have input so far, otherwise we won't know when to set the output to 1. This is why we need the states in our sequential circuit.

Our system needs three states:

- Zero values of 1 have been input so far.
- One value of 1 has been input so far.
- Two values of 1 have been input so far.

Now let's look at the behavior of our system for each individual state. We'll start with the first state, *Zero values of 1 have been input so far*. If we input a 0, we still have zero 1s input, and we have to do two things. We need to set our output to 0 because we have not read in three 1s. Also, we still have zero 1s input, so we need to remain in this state. If we input a value of 1, we still need to output a 0 since we still have not read in three 1s. However, since we have now input 1 value of 1, we need to change to the state *One value of 1 has been input so far*.

The second state works in a similar way. If the input is 0, we output a 0 and remain in the same state. If it is 1, we still output a 0 (because we now have read in two values of 1, not three) and go to the state *Two values of 1 have been input so far*.

The final state acts like the others when the input is 0. It stays in the same state and sets the output to be 0. If the input is 1, however, our circuit behaves differently. This is our third input value of 1, so we have to set our output to 1. We also need to go to a different state. There is no state called *Three values of 1 have been input so far*. Instead, we do what we said we would do: set the output to 1 and start the whole process again. We do this by going back to the state *Zero values of 1 have been input so far*.

Figure 6.2 shows the generic diagram for this circuit. The animation shows how the state and output change for the input sequence 0, 1, 0, 1, 0, 1.

We're not done yet. If I have the input set to 0, how do I know if that is one 0, two 0s, or just how many 0s it is? The same question applies to the input value of 1.

There is another signal in this circuit labeled CLOCK. We use this signal to tell our circuit when to check the input value. This signal might be used when generating the output value, but it is mainly used to determine when to go from one state to the next. For that reason, it is also passed directly to the portion of the circuit that stores the state. This leads us to our next question: What is a clock?

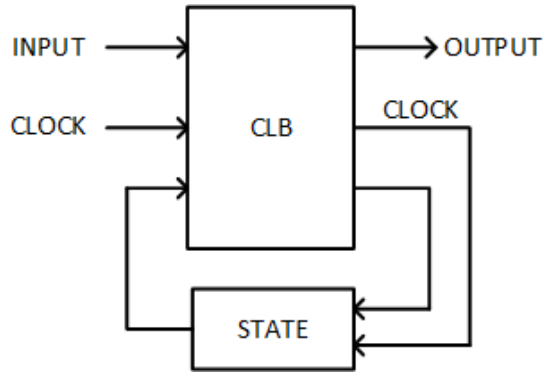


Figure 6.2: Block diagram of a circuit that outputs 1 when three 1's have been input.

WATCH ANIMATED FIGURE 6.2

6.1.2 What Is a Clock?

There are many types of clocks. One that I built a number of years ago is shown in Figure 6.3. In terms of sequential circuits, however, this has nothing to do with the CLOCK signal. This is not our CLOCK.

Figure 6.3: Not our clock.

In digital circuits, a clock is a signal that alternates between high and low, or 1 and 0. In order to minimize the time spent by the signal in between the voltage levels for logic 0 and logic 1, it usually is implemented as a square wave, as shown in Figure 6.4 (a). This is an idealized square wave, which instantaneously changes from 0 to 1 and from 1 to 0. In practice, it does change extremely quickly, but not instantaneously. It does have a small rise and fall time, as shown in Figure 6.4 (b).

For most circuits, the rise and fall times are insignificant. This can be of concern, however, for circuits that run at a very high **frequency**, that is, a large number of clock cycles per second. Modern microprocessors, for example, may have clock frequencies over 3GHz (3 billion clock cycles per second). At this frequency, light travels about 4 inches, or 10 centimeters, in

one clock cycle, which has a **period** (the time for a single clock cycle) of 0.33ns. At this frequency, the rise or fall time is a greater percentage of the clock period.

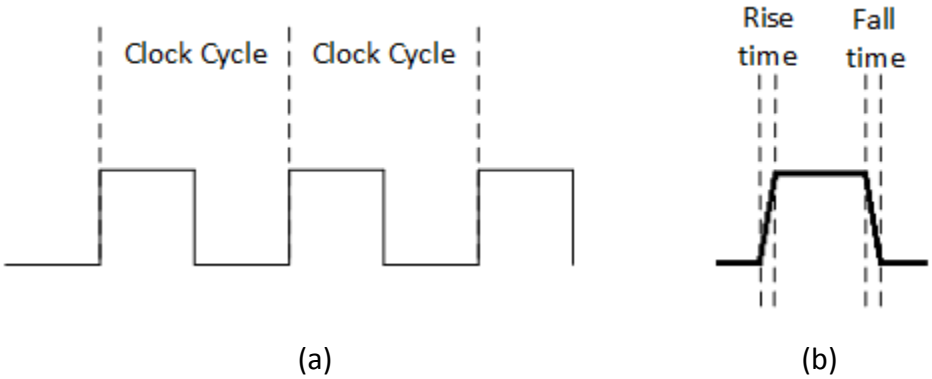


Figure 6.4: Our clock: (a) Idealized with clock cycles shown; (b) Rise and fall times.

As we will see in the following two sections, latches and flip-flops will use the clock signal either as an enable signal to set the new state value when it is at the desired level, or they will make use of its edges to set the new level.

Most sequential circuits are synchronous. They are easier to design and debug when a clock signal can be used to synchronize changes in states within the circuit. As clock frequencies increase, the circuits can do things more quickly, almost as if they aren't even making use of the clock. However, there is another type of sequential circuit that does not use a clock. We'll look at that next.

6.1.3 Asynchronous Sequential Circuits

Whereas synchronous sequential circuits use a clock to synchronize the changing of their states, asynchronous sequential circuits do not use a clock signal. Instead, they typically incorporate timing delays, or unclocked latches, in place of the storage found in synchronous sequential circuits. Figure 6.5 shows the basic configuration of an asynchronous sequential circuit.

Other than replacing storage with time delays, this model is similar to that used for synchronous sequential circuits. The circuit inputs values to a combinatorial logic block that generates outputs and the next state of the circuit. This next state value is fed back to the combinatorial logic block, after a delay, to use as it generates these values. In essence, the asynchronous sequential circuit is just a combinatorial circuit with feedback and delays.

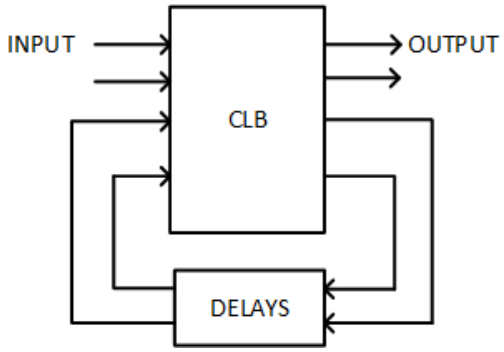


Figure 6.5: Generic model for an asynchronous sequential circuit.

Asynchronous sequential circuits are more difficult to design than synchronous sequential circuits. Being able to synchronize when values change in a circuit is a great advantage that asynchronous sequential circuits, by their nature, simply do not have. In this chapter and in Chapter 8, we will focus on synchronous sequential circuits. However, when we design the latches and flip-flops used in synchronous sequential circuits, we will see that they are actually combinatorial circuits with feedback, much like asynchronous sequential circuits.

6.2 Latches

So far we have alluded to the fact that we need to be able to store the current state of a sequential circuit. So, what are we storing and how do we store it? The answer to the first part of this question is "a binary value." When we want to store the current state of the circuit, we list all the possible states and assign a binary value to each state. The number of bits in these binary values must be the same for all states. From our discussion of binary numbers at the beginning of this book, we know that an n-bit number can have 2^n unique values. We will choose the smallest value possible for n such that:

$$\#\ of\ states \leq 2^n, \text{ or}$$
$$n = \lceil \lg(\#\ of\ states) \rceil$$

where $\lceil\ \rceil$ represents the ceiling function, the smallest integer value that is greater than or equal to the number it encloses, and lg is the base 2 logarithm function.

In the example in the previous section, our circuit has three states, based on the number of 1s read in so far. The smallest possible value of n for this circuit is 2. One possible assignment of 2-bit values to states is shown in Figure 6.6.

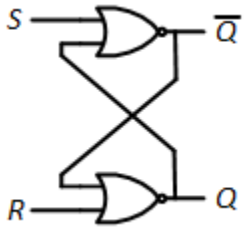
State Value	State
0 0	Zero values of 1 have been read in so far
0 1	One value of 1 has been read in so far
1 0	Two values of 1 have been read in so far

Figure 6.6: One possible assignment of state values to states.

This brings us to the second question: how to store these values. In this section we look at latches. A latch consists of cross-coupled gates that settle into a stable state. Much as the latch on a door keeps the door in place, a latch keeps its outputs at one value, our stable state. In the next subsection, we will look at the S-R latch, and in the following subsection we introduce the D latch.

6.2.1 S-R Latch

The S-R (Set-Reset) latch was the first latch developed for digital circuits. It consists entirely of combinatorial logic, but is connected in such a way that the circuit locks in a binary value, 1 or 0, based on the values of its inputs. Figure 6.7 (a) shows one way to construct the S-R latch using NOR gates. Notice that the output of each NOR gate is fed back to supply one of the inputs of the other NOR gate. This is the cross-coupling mentioned earlier; this is what allows the circuit to lock in a value.



(a)

S	R	Q	$\bar{Q}$
0	0	Q_0	Q0
0	1	0	1
1	0	1	0
1	1	0	0

(b)

Figure 6.7: S-R Latch: (a) Internal configuration using NOR gates; (b) Truth table.

WATCH ANIMATED FIGURE 6.7

The truth table for this circuit is shown in Figure 6.7 (b). When S = 1 and R = 0, we set output Q to 1 and the complemented output, $\overline{Q}$ to 0. When S = 0 and R = 1, we do the opposite: Q = 0 and Q = 1. When S = 1 and R = 1, both NOR gates output a 0. This is an invalid value because it sets $Q = \overline{Q}$, which should never be the case. This combination of input values is not allowed for the S-R latch. The final input values, S = 0 and R = 0, do something different. This keeps whatever value is already stored unchanged. If Q = 0, SR = 00 keeps Q = 0. If Q = 1, they keep Q = 1.

The animation for this figure shows how the internal values change as we go through the following sequence of values for S and R: 00 → 01 → 00 → 10 → 00 → 11. In this animation, we start by setting SR = 01. Since R = 1, the output of the lower NOR gate will be 0, no matter what its other input is. The upper NOR gate has S = 0 as its first input and the output of the second NOR gate, also 0, as its second input. With both inputs equal to 0, its output is 1. This is fed back as an input to the second NOR gate, but does not change its value. The circuit is in a stable state with Q = 0 and $\overline{Q}$ = 1.

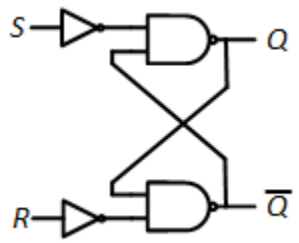
Next, we set *SR* = 00. Since *Q* = 0, both inputs to the upper NOR gate are still 0, and it continues to output a 1. This value is fed back into the lower NOR gate; *R* = 0 is also an input to that gate. With input values of 1 and 0, the NOR gate outputs a 0. The circuit is once again stable, with *Q* = 0 and $\overline{Q}$ = 1.

Then we set *S* = 1 and *R* = 0. Since *S* = 1, the upper NOR gate outputs a 0 regardless of the value of the other input. This 0, and *R* = 0, are input to the lower NOR gate, which outputs a 1. Feeding this back into the first NOR gate does not change its output. In its stable state, the circuit has *Q* = 1 and $\overline{Q}$ = 0.

Now we set *SR* = 00 again. The upper NOR gate has input values *S* = 0 and *Q* = 1, and sets its output to 0. The lower gate inputs $\overline{Q}$ = 0 and *R* = 0, setting its output to 1. The circuit is stable with *Q*= 1 and $\overline{Q}$ = 0. Just as it did earlier, the circuit retains its output values when *SR* = 00.

Finally, we set *SR* = 11. Both NOR gates have an input equal to 1, so both gates set their outputs to 0, giving us the invalid outputs *Q* = 0 and $\overline{Q}$ = 0.

There is another configuration for the S-R latch that uses NAND gates. It is shown in Figure 6.8 (a). Notice that the *S* and *R* inputs are inverted; this is necessary for this circuit to function in the same way as the cross-coupled NOR circuit. Its truth table is shown in Figure 6.8 (b). Notice that there is one difference between this truth table and the truth table for the previous circuit. When we have invalid inputs *SR* = 11, this circuits sets *Q* = 1 and $\overline{Q}$ = 1 instead of setting both to 0.



(a)

S	*R*	*Q*	$\overline{Q}$
0	0	Q_0	$\overline{Q}_0$
0	1	0	1
1	0	1	0
1	1	1	1

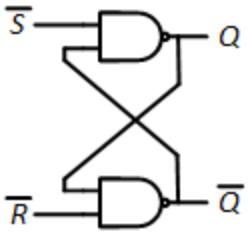
(b)

Figure 6.8: S-R Latch: (a) Internal configuration using NAND gates; (b) Truth table.

WATCH ANIMATED FIGURE 6.8

The animation for this figure takes us through the same input sequence as before: *SR* = 00 ⟶ 01 ⟶ 00 ⟶ 10 ⟶ 00 ⟶ 11.

An alternative design dispenses with the two NOT gates and leaves it to the designer to invert *S* and *R* before inputting them to the circuit. This is sometimes called the $\overline{S}$-$\overline{R}$ latch and is shown in Figure 6.9 (a). Its truth table, with values for $\overline{S}$, $\overline{R}$, *S*, and *R*, is shown in Figure 6.9 (b). The animation shows the internal signals as it goes through the same sequence of values for *S* and *R*, this time shown as $\overline{SR}$ = 11 ⟶ 10 ⟶ 11 ⟶ 01 ⟶ 11 ⟶ 00.



(a)

S	R	$\bar{S}$	$\bar{R}$	Q	$\bar{Q}$
0	0	1	1	Q_0	$\bar{Q}_0$
0	1	1	0	0	1
1	0	0	1	1	0
1	1	0	0	1	1

(b)

Figure 6.9: $\overline{S}$-$\overline{R}$ Latch: (a) Internal configuration using NAND gates; (b) Truth table.

WATCH ANIMATED FIGURE 6.9

6.2.2 S-R Latch with Enable

In Chapter 4, we introduced several more complex components. Some of these components have an enable input. When the enable signal is asserted, usually by setting its value to 1, the component behaves normally, as it is designed to do. When the enable signal is not asserted, however, it does not perform its intended function. As an example, consider the 2 to 4 decoder shown in Section 4.1.2, Figure 4.5. When its enable signal is 0, all the outputs are set to 0, no matter what values the select inputs I_1 and I_0 have.

We can also add an enable signal to an *S*-*R* latch. When the enable signal is 1, the *S*-*R* latch functions just as described in the previous subsection. However, when it is 0, we do not change the outputs regardless of the values of *S* and *R*. Here, we will call it an enable signal. *As we use the S-R latch to create flip-flops in the next section, this signal will morph into a clock.*

We can incorporate this enable signal into the *S*-*R* latch fairly easily by taking advantage of the fact that setting *SR* = 00 does not change the output values. This is exactly what we want to happen when our enable signal is 0. We keep the cross-coupled design of the *S*-*R* latch the same, and combine the enable with the *S* and *R* inputs to create the final *SR* inputs to the latch. This is shown in Figure 6.10.

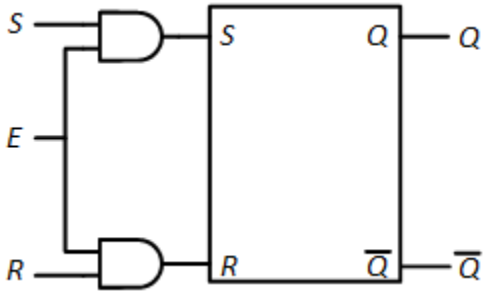


Figure 6.10: S-R Latch with Enable Signal

WATCH ANIMATED FIGURE 6.10

The way this works is fairly straightforward. When $E = 1$, $S \wedge 1 = S$, $R \wedge 1 = R$, and the original S and R values are input to the S-R latch, it functions as it normally would.
When $E = 0$, $S \wedge 0 = 0$, $R \wedge 0 = 0$, then the circuit inputs 00 to the S-R latch which keeps its outputs unchanged.

This methodology, conditioning the inputs and using the rest of the circuit without changes, is used regularly in digital design. The unmodified portion of the circuit, the original S-R latch in this case, may be available only as a component or integrated circuit chip that cannot be modified. There are also other advantages, including that the unmodified portion has already been verified as correct, which simplifies the design cycle. In my other book, we use this methodology to design the arithmetic and logic unit (ALU) portion of a microprocessor.

6.2.3 D Latch

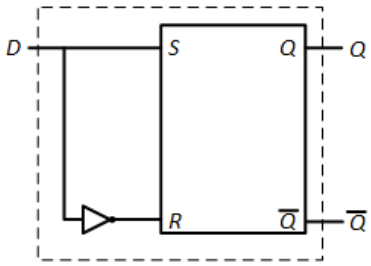
In many instances, we don't want to deal with the latch inputs. We have a value that we want to store in the latch, and we don't want to spend time and construct circuitry to create the latch inputs to store that value. We want to say, "Here's the value; store it in the latch." This led to the creation of the D (Delay) latch. Its truth table is shown in Figure 6.11 (a).

D	Q	$\bar{Q}$
0	0	1
1	1	0

(a)

D	Q	$\bar{Q}$	S	R
0	0	1	0	1
1	1	0	1	0

(b)



(c)

Figure 6.11: D Latch: (a) Truth table; (b) Excitation table with S and R input values; (c) Circuit design.

WATCH ANIMATED FIGURE 6.11

As was the case with the S-R latch, with enable, we design the D latch by starting with an S-R latch and conditioning its inputs. Looking at the truth table, we see that we want Q to be the same as input D. To do this, we need to determine the values of S and R that will accomplish this. In Figure 6.11 (b), we expand the truth table to show the S and R values we need to produce. This is called an **excitation table**. An excitation table is very much like a truth table. Instead of showing outputs, however, it shows the inputs to be generated for the given values of the inputs and the current state/outputs.

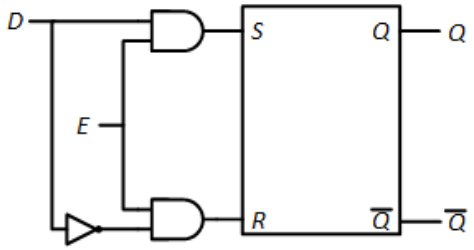
From this table, we can easily see that we must set $S = D$ and $R = D'$. We can connect input D directly to S of the S-R latch, and we pass D through a NOT gate to generate D', which we connect to input R. This circuit is shown in Figure 6.11 (c). Everything inside the dashed box comprises the D latch.

Looking at the behavior of this circuit, you might argue that it is fairly useless. You may be storing some value, but as soon as the value of *D* changes, the latched value changes too. If this is your argument, well, you're quite correct. In this circuit, we always output the same value that we input.

For this reason, D latches always have an enable input, which functions much like the enable signal we added to the S-R latch. When the enable input is 1, we set output *Q* to the value of input *D* (and output $\overline{Q}$ to its complement). When the enable is 0, *Q* retains its previous value, no matter what value input *D* has. The excitation table for the D latch with enable is shown in Figure 6.12 (a), including the values of *S* and *R*.

E	*D*	*Q*	$\bar{Q}$	*S*	*R*
0	0	Q_0	$\bar{Q}_0$	0	0
0	1	Q_0	$\bar{Q}_0$	0	0
1	0	0	1	0	1
1	1	1	0	1	0

(a)



(b)

Figure 6.12: D latch: (a) Excitation table; (b) Circuit design.

WATCH ANIMATED FIGURE 6.12

To implement this, we'll do exactly what we did for the S-R latch with enable. We take the *S* and *R* inputs for the latch without enable and logically AND them with the enable signal. When the enable input is 1, $S = D \wedge 1 = D$ and $R = D' \wedge 1 = D'$. When the enable input is 0, $S = D \wedge 0 = 0$ and $R = D' \wedge 0 = 0$, and the output is unchanged. The circuit to implement the D latch with enable is shown in Figure 6.12 (b).

6.3 Flip-Flops

For the D latch with an enable signal presented in the previous subsection, what happens if the value on the *D* input changes while the enable signal is high? For example, consider the values shown for the *D* input and the enable signal shown in Figure 6.13. What would the waveform for output Q look like? Think this through for a minute before proceeding.

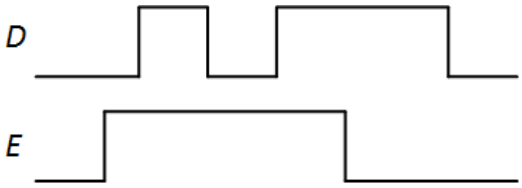


Figure 6.13: D latch input values with *D* changing while *E* = 1.

Hopefully you did take some time to think this through, and to look back at the truth table in Figure 6.12 (a). When *E* = 1, we set *Q* to whatever value is being input on *D*. As *D* goes from 0 to 1, back to 0, and then back to 1 again, the *Q* output does the same because *E* = 1. When *E* goes to 0, the circuit locks in the value of *Q* and does not change it, even when *D* changes from 1 to 0. The timing diagram, with output values, is shown in Figure 6.14. The animation for this figure shows how the inputs, *S* and *R* values, and outputs change over time.

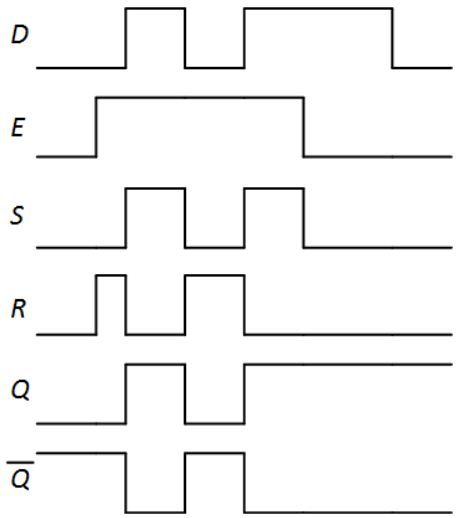


Figure 6.14: D latch input, output, and internal signal values.

WATCH ANIMATED FIGURE 6.14

For many designs, this behavior is undesirable. We want to update our output values and lock them in. When we use a clock to enable our latch, we often want to update these values only once in each clock cycle. This is the reason why most sequential circuits do not use latches.

Instead, engineers designed another type of device to store data values and only update them once per clock cycle, usually on the rising or falling edge of the clock. (The **rising edge** occurs when the clock changes from 0 to 1, and the **falling edge** is when the clock changes from 1 to 0.) These are called **flip-flops**, and that is what we'll be discussing in this section. There are several types of flip-flops, all of which incorporate latches in their design. This is why I spent time writing a section on a component you won't use; we do use it to design the component you actually will use.

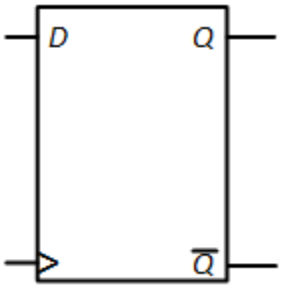
The remainder of this section introduces three flip-flops: the D, J-K, and T flip-flops. We will see how they are constructed from the D and S-R latches introduced in the previous section, their overall functions, and how some of them are extended to include asynchronous signals to preset (set *Q* = 1) and clear (set *Q* = 0) them, regardless of the value of the clock.

6.3.1 D Flip-Flop

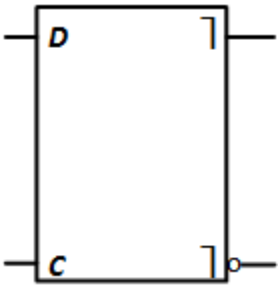
The D flip-flop is functionally similar to the D latch. The data value on the *D* input becomes the value placed on the *Q* output. Unlike the D latch, however, the value is only loaded into the D flip-flop on one of the clock edges. Throughout this section, we will design flip-flops so they load in data on the rising edge of the clock. Figure 6.15 (a) shows the truth table for the D flip-flop. The symbol ↑ indicates the rising edge of the clock.

D	*CLK*	*Q*	$\bar{Q}$
X	not ↑	Q_0	$\bar{Q}_0$
0	↑	0	1
1	↑	1	0

(a)



(b)



(c)

Figure 6.15: Positive edge-triggered D flip-flop: (a) Truth table; (b) Logic symbol (edge-triggered); (c) Logic symbol (pulse triggered).

The standard symbols to represent the D flip-flop are shown in Figures 6.15 (b) and (c). Note that there are two types of flip-flops. The edge-triggered flip-flops use only one edge of the clock internally; the > symbol indicates the clock input and that this is a positive edge-triggered clock. A pulse-triggered flip-flop uses both edges of the clock. From the outside, we see only one edge changing data, but internally there are operations occurring on both edges. The ⌉ symbol on the *Q* and $\bar{Q}$ outputs indicate that the flip-flop is pulse triggered.

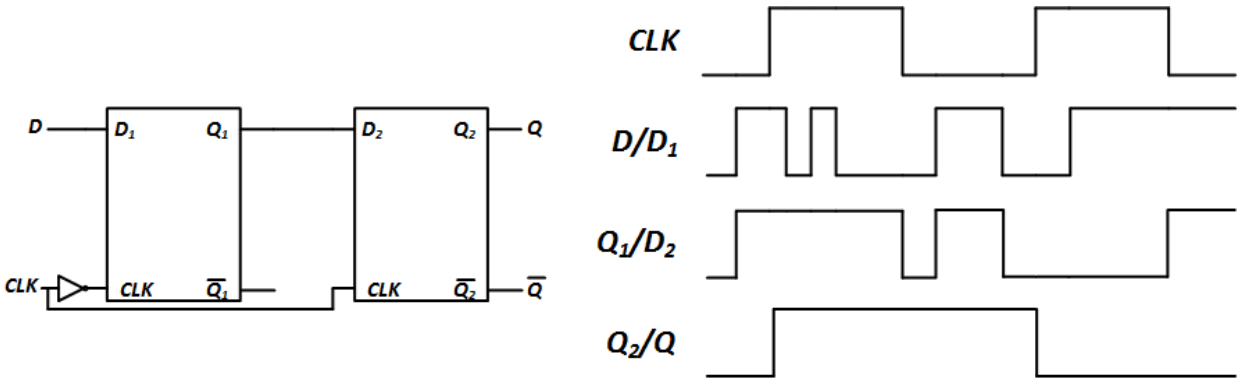
6.3.1.1 Leader-Follower Design

Given all of this, let's look at the internal design of the D flip-flop. The most straightforward design consists of two D latches in what has been called a **master-slave** configuration, but in this book will be referred to as a **leader-follower** configuration. This design is shown in Figure 6.16 (a). It is pulse-triggered, as each edge of the clock enables one of the latches. It works as follows.

1. When *CLK* = 0, the first D latch is enabled and it continuously sets its output *Q* to whatever value *D* has. It passes its *Q* value to the *D* input of the second D latch, but that latch is not enabled, so it ignores that input and keeps its previous value.
2. On the rising edge of the clock, the first D latch is no longer enabled. Whatever value it has at that time is locked in and is held as the *D* input to the second latch. Any changes in the *D* input of the first latch are ignored. At the same time, the second latch is now enabled. It continually passes the value on its *D* input to its *Q* output as long as *CLK* = 1.

This input value is the value that was just locked into the first latch. Since that value does not change, neither does its output.

3. The clock eventually hits its falling edge, transitioning from 1 to 0. This disables the second latch, locking in its value. The first latch is now enabled, and we're back at the first step in this process.



(a) (b)

Figure 6.16: D flip-flop: (a) Leader-follower design; (b) Signal values for sample input sequence.

WATCH ANIMATED FIGURE 6.16

Figure 6.16 (b) shows a timing diagram for a sample *D* and *CLK* input sequence. This sequence is shown in the animation for this figure and briefly described below.

1. Initially, *CLK* = 0. The first D latch is enabled. The value in its *D* input, D_1, is passed through to its output, Q_1. The second latch is disabled; its output remains unchanged.
2. The clock changes from 0 to 1 on its rising edge. The first latch is disabled and locks in its value. It does not change when its *D* input changes as long as *CLK* = 1. The second latch is now enabled. It passes the value on its *D* input, D_2, to its output, Q_2, and to the *Q* output of the flip-flop. Since its input is connected to the output of the first *D* latch, which does not change, its output stays at 1 the entire time that *CLK* = 1.
3. The clock goes back to 0. The second latch is disabled and locks in its output. The first latch is now enabled and passes the value on its input D_1 to its output Q_1.
4. On the next rising edge of *CLK*, *D* = 0, and this value is locked into the first latch; Q_1 = 0. The second latch reads in this value and passes it through to its *Q* output.
5. On the final falling edge, the second latch is disabled and locks in its output value of 0. The first latch is enabled and passes the *D* input to its *Q* output.

6.3.1.2 Preset and Clear

There is another design commonly used for edge-triggered D flip-flops. It uses three $\overline{S}$-$\overline{R}$ latches, a total of six gates. This design, however, includes two additional inputs, $\overline{PRE}$ (preset) and $\overline{CLR}$

(clear). When $\overline{PRE}$=0, it sets output *Q* to 1, regardless of the state of the *D* input. Similarly, $\overline{CLR}$=0 sets *Q* to 0. Unlike the *D* input, which is sent to the output on the rising edge of the clock, $\overline{PRE}$ and $\overline{CLR}$ are asynchronous. When either value goes to 0, the output is changed immediately; it does not have to wait for the rising edge of the clock. The truth table for this flip-flop is shown in Figure 6.17.

D	*CLK*	$\overline{PRE}$	$\overline{CLR}$	*Q*	$\bar{Q}$
X	X	0	1	1	0
X	X	1	0	0	1
0	↑	1	1	0	1
1	↑	1	1	1	0
X	not ↑	1	1	Q_0	$\bar{Q}_0$

Figure 6.17: Truth table for the edge-triggered D flip-flop with preset and clear. $\overline{PRE}$ and $\overline{CLR}$ may not both be set to 0 at the same time.

The complete circuit used to design the 7474 integrated circuit chip that has two edge-triggered D flip-flops is shown in Figure 6.18. Note what happens to the *Q* output when $\overline{PRE}$ = 0. This value is input directly to the NAND gate that generates *Q*. Once it is set to 0, the output of that gate becomes 1, regardless of the values of *D* and the clock. In a similar way, setting $\overline{CLR}$ = 0 causes $\overline{Q}$ to become 1 immediately.

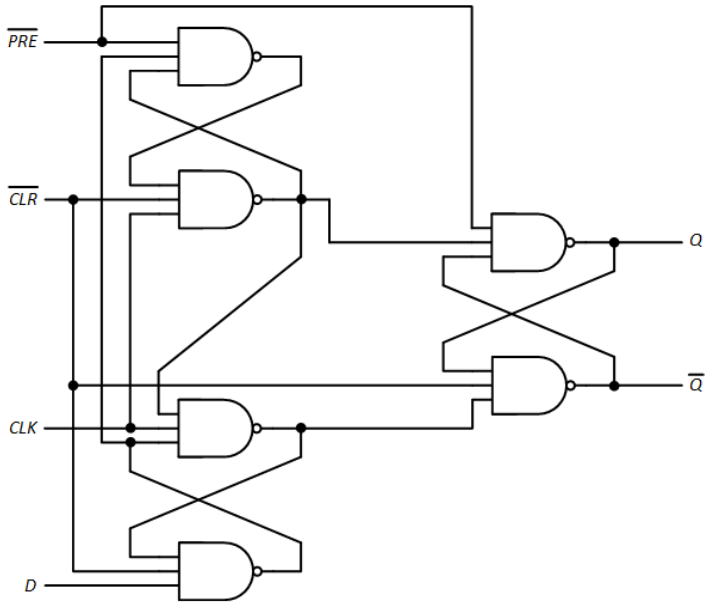


Figure 6.18: Internal design of the positive edge-triggered D flip-flop with preset and clear.

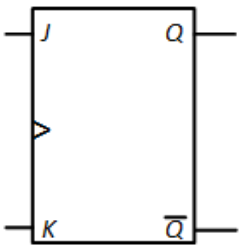
WATCH ANIMATED FIGURE 6.18

6.3.2 J-K Flip-Flop

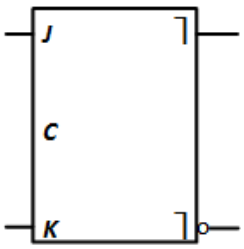
The S-R latch would be a good candidate to extend and create an S-R flip-flop, except for one thing – the invalid *SR* = 11 input values. To address this, digital design engineers developed another type of flip-flop that defines a function to be performed with these input values; it inverts the output values. To avoid confusion with the S-R latch, this is called the J-K flip-flop. Here, *J* replaces *S* and *K* replaces *R*. Its truth table and logic symbols are shown in Figure 6.19.

J	*K*	*CLK*	*Q*	$\bar{Q}$
X	X	not	Q_0	$\bar{Q}_0$
		↑		
0	0	↑	Q_0	$\bar{Q}_0$
0	1	↑	0	1
1	0	↑	1	0
1	1	↑	$\bar{Q}_0$	Q_0

(a)



(b)



(c)

Figure 6.19: J-K flip-flop: (a) Truth table; (b) Logic symbol (edge-triggered); (c) Logic symbol (pulse triggered).

There are a couple of ways to design this flip-flop. We can use the leader-follower methodology we employed for the D flip-flop. Like its counterpart, this is technically considered to be pulse triggered because it uses both the rising and falling clock edges. This design uses two S-R latches as the leader and follower, as shown in Figure 6.20.

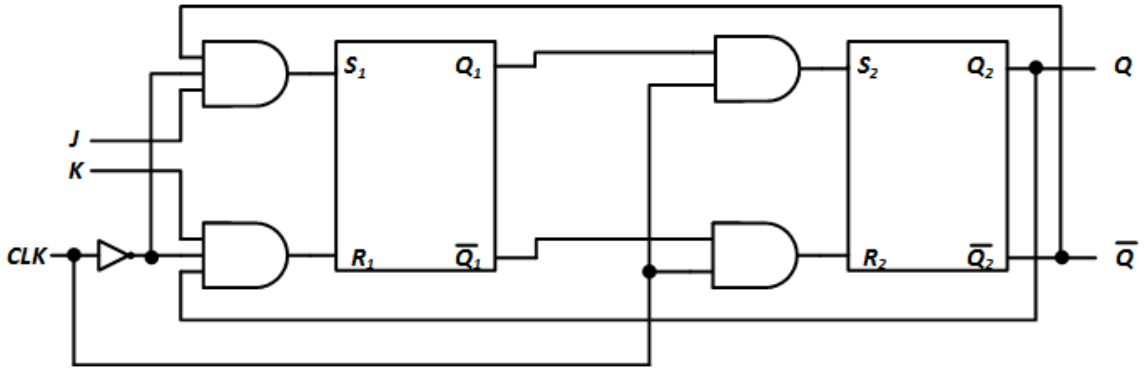


Figure 6.20: J-K flip-flop: Leader-follower design.

WATCH ANIMATED FIGURE 6.20

When the clock input is 0, the first latch has $S = J \wedge Q'$ and $R = K' \wedge Q$. This ensures the latch never has the invalid *SR* = 11 input. Also during this time, the two inputs to the second latch's *S* and *R* inputs are both 0, so it does not change its outputs. When the clock goes from 0

to 1, the *SR* inputs to the first latch become 00 and it locks in its value. Its outputs become the inputs to the second latch, which produces the desired output. The outputs of the first latch are always 10 or 01, so this latch also never receives the invalid 11 input.

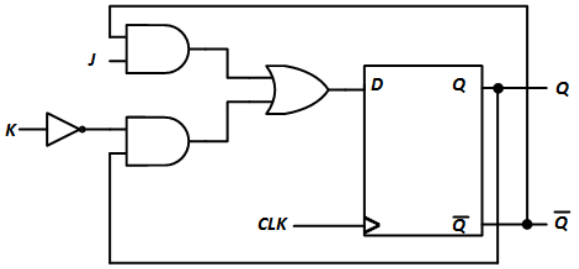
It is also possible to design a J-K edge-triggered flip-flop using the D flip-flop we designed in the previous subsection. We just need to generate the *D* input using *J*, *K*, and the current outputs *Q* and *Q'*. To do this, consider the excitation table shown in Figure 6.21 (a). In this table, note that *Q* is an input, not an output. We are trying to generate *D* and we will use the current output of the J-K flip-flop to do this.

J	*K*	*Q*	*D*
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

(a)

*Q**JK*	00	01	11	10
0	0	0	1	1
1	1	0	0	1

(b)



(c)

Figure 6.21: Creating a J-K flip-flop using a D flip-flop: (a) Excitation table; (b) Karnaugh map to generate *D*; (c) Final circuit.

WATCH ANIMATED FIGURE 6.21

With this excitation table, we can create the Karnaugh map shown in Figure 6.21 (b), identify the essential prime implicants (*JK'* is not essential, nor is it needed to generate *D*), and determine the function to produce the desired value of *D*. The final circuit is shown in Figure 6.21 (c).

Finally, it is possible to incorporate asynchronous preset and clear signals in an edge-triggered J-K flip-flop, just as we did for the D flip-flop. Figure 6.22 shows the design for this circuit. This is mostly the design for the 74112 TTL chip, except the clock is inverted. This changes the design from the negative edge-triggered circuit in this chip to a positive edge-triggered design.

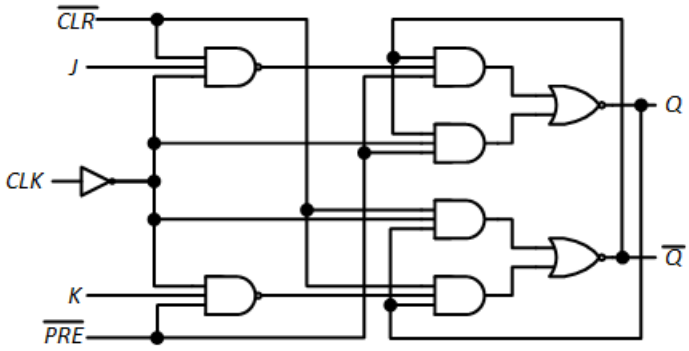


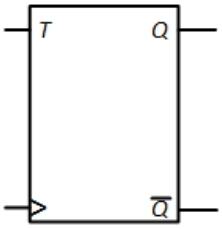
Figure 6.22: Internal design of the positive edge-triggered J-K flip-flop with preset and clear.

6.3.3 T Flip-Flop

The T (toggle) flip-flop, unlike the D and J-K flip-flops, cannot set its output to 0 or 1 explicitly. Rather, it can invert its output or leave it unchanged. It has one input, *T*, and a clock, and the usual *Q* and $\overline{Q}$ outputs. If *T* = 0 on the rising edge of the clock, output *Q* remains unchanged; if *T* = 1, it is inverted. Figure 6.23 shows the truth table and logic symbol for the positive edge-triggered T flip-flop. Note that the clock is not shown in the truth table. We can leave it out of the truth table when it is implicit that all transitions must occur on a clock edge. We could not leave it out of the truth tables for the flip-flops with preset and clear signals, since those signals do not need a clock edge to perform their functions.

T	Q
0	Q_0
1	$\bar{Q}_0$

(a)



(b)

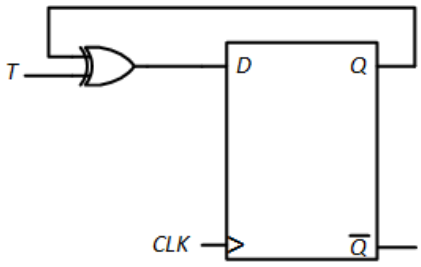
Figure 6.23: T flip-flop: (a) Truth table; (b) Logic symbol.

Several companies manufacture chips with D and J-K flip-flops for designers to use in their circuits. However, no company manufactures a chip with T flip-flops. Nevertheless, they are commonly used, and no company makes them because they are very easy to construct using existing flip-flops. First, let's look at how to construct a T flip-flop using a D flip-flop.

To design a T flip-flop from a D flip-flop, we start by going back to the truth table and using it to generate the excitation table. We expand the table to list all values of *Q* explicitly; this is shown in Figure 6.24 (a).

T	Q	D
0	0	0
0	1	1
1	0	1
1	1	0

(a)



(b)

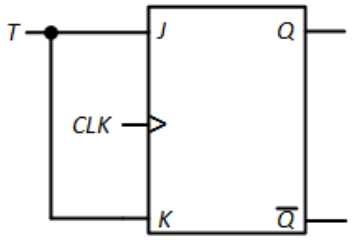
Figure 6.24: T flip-flop constructed using a D flip-flop: (a) Excitation table; (b) Final design.

One way to generate *D* from the *T* and *Q* values is to set $D = T'Q + TQ'$. A simpler way is to XOR *T* and *Q*, giving us $D = T \oplus Q$. Either way is fine; we use the latter in the circuit shown in Figure 6.24 (b).

We can also use a J-K flip-flop to construct a *T* flip-flop. As before, we can generate the excitation table, shown in Figure 6.25 (a). Notice that there are several don't care entries in this table. This occurs because there are two ways to set the new output value. For the first row, for example, we have a current value of $Q = 0$ and we want to end up with the same value. We can set $JK = 00$, which keeps it at its current value of 0, or we can set $JK = 01$, which explicitly sets the value to 0. The second row either sets $Q = 1$ explicitly with $JK = 10$ or inverts the output to 1 if $JK = 11$. The remaining entries are derived in a similar manner.

T	Q	J	K
0	0	0	X
0	1	X	0
1	0	1	X
1	1	X	1

(a)



(b)

Figure 6.25: T flip-flop constructed using a J-K flip-flop: (a) Excitation table; (b) Final design.

With this table, you could construct Karnaugh maps and derive minimal functions for *J* and *K*; for this design, $J = T$ and $K = T$. The final design is shown in Figure 6.25 (b). For this particular design, we could have said "If $T = 0$, we want the output to remain unchanged, so we want $J = K = 0$, and if $T = 1$, we want to invert the output, so we must set $J = K = 1$. Putting them together, we want $J = K = T$." Most designs cannot be completed so easily – good for you if you realized this before reading this paragraph.

6.4 Summary

This chapter begins our study of sequential circuits. Unlike combinatorial circuits, which produce outputs based solely on the current input values, a sequential circuit produces outputs

based on the current input values and its current state. Depending on the state of the circuit, the same input values may generate different output values.

A sequential circuit requires something to store the current state of the circuit. A block of combinatorial logic reads in the state of this circuit from the storage elements and the data inputs, and it generates the circuit outputs; it also sends the new state of the circuit to the storage elements.

Latches are a fundamental storage component. They use cross-coupled logic gates to lock in (or latch) a data value, either 0 or 1. This chapter introduced the S-R latch (and its variant, the $\overline{S}$-$\overline{R}$ latch) and the D latch. We examined their overall functions and their internal configurations.

Latches may have an enable signal. When it is asserted, the output changes whenever the input changes. This can be problematic for many sequential circuit designs. For this reason, designers developed flip-flops. These devices load new data only on the rising edge of the clock, when it changes from 0 to 1, though some can force the output to 1 (preset) or 0 (clear) at any time, without regard for the clock. The design of the D flip-flop is based on the S-R latch. The leader-follower design is pulse-triggered, and the design with asynchronous preset and clear signals uses three modified $\overline{S}$-$\overline{R}$ latches. The J-K flip-flop is created from the edge-triggered D flip-flop by using its inputs and current outputs to generate the next value of *D* to be stored in the flip-flop. The T flip-flop employs the same strategy as it is constructed from either a D or J-K flip-flop.

In the next chapter, we will examine some more complex sequential components. These components are constructed using the flip-flops introduced in this chapter, with the addition of combinatorial logic needed to generate the correct inputs to the flip-flops and the correct component outputs.

Bibliography

Carpinelli, J. D. (2001). *Computer systems organization & architecture*. Addison-Wesley.

Hayes, J. P. (1993). *Introduction to digital logic design*. Addison-Wesley.

Mano, M. M., & Ciletti, M. (2017). *Digital design: with an introduction to the Verilog HDL, VHDL, and SystemVerilog* (6th ed.). Pearson.

Mano, M. M., Kime, C., & Martin, T. (2015). *Logic & computer design fundamentals* (5th ed.). Pearson.

Nelson, V., Nagle, H., Carroll, B., & Irwin, D. (1995). *Digital logic circuit analysis and design*. Pearson.

Roth, J. C. H., Kinney, L. L., & John, E.B. (2020). *Fundamentals of logic design enhanced edition* (7th ed.). Cengage Learning.

Sandige, R. S. (1990). *Modern digital design*. McGraw-Hill.

Skahill, K. (1996). *VHDL for programmable logic*. Addison-Wesley.

Texas Instruments, Inc. (1981). *The TTL Data Book for Design Engineers*. (2nd ed.). Texas Instruments.

Wakerly, J. F. (2018). *Digital design: Principles and practices* (5th ed.). Pearson.

Exercises

1. For the 1s counter sequential system in Section 6.1, show the state and output after each input in the following sequence: 0 → 1 → 1 → 1 → 0 → 1. Initially the system is in state *Zero values of 1 have been input so far*.

2. Repeat Problem 1 for the sequence 1 → 0 → 1 → 1 → 1 → 0.

3. The 1s counter sequential system has the following sequence of inputs and outputs.

Input	0	1	1	1	1	0
Output	0	0	1	0	0	0

   Show its initial and final state.

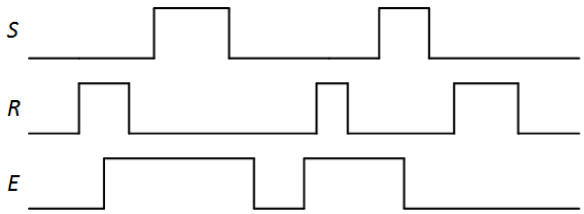
4. Show the outputs of the S-R latch in Figure 6.8 for the input sequence *SR* = 10 → 00 → 01 → 10 → 01 → 00.

5. Repeat Problem 4 for the sequence *SR* = 01 → 10 → 01 → 00 → 10 → 00.

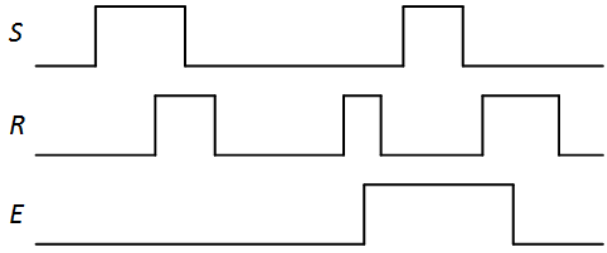
6. Show the outputs of the $\overline{S}$-$\overline{R}$ latch in Figure 6.9 for the input sequence $\overline{SR}$ = 10 → 11 → 01 → 10 → 01 → 11.

7. Repeat Problem 6 for the sequence $\overline{SR}$ = 01 → 10 → 01 → 11 → 10 → 11.

8. For the S-R latch with enable in Figure 6.10, show the values of *Q* and $\overline{Q}$ for the input values in the following timing diagram.

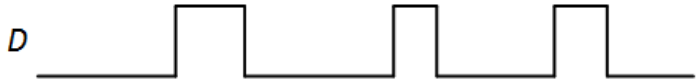


9. Repeat Problem 8 for the input sequence shown in the following timing diagram.



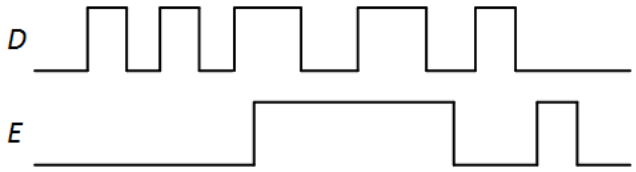
10. For the D latch in Figure 6.11, show the values of *S*, *R*, *Q*, and $\overline{Q}$ for the input sequence *D* = 0 ⟶ 1 ⟶ 1 ⟶ 0 ⟶ 1 ⟶ 0.

11. Repeat Problem 10 for the input sequence shown in the following timing diagram.



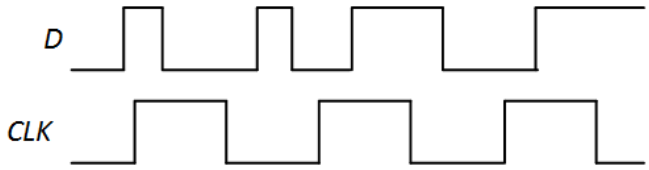
12. For the D latch with enable in Figure 6.12, show the values of *S*, *R*, *Q*, and $\overline{Q}$ for the input sequence *DE* = 01 ⟶ 11 ⟶ 10 ⟶ 00 ⟶ 11 ⟶ 01.

13. Repeat Problem 12 for the input sequence shown in the following timing diagram.

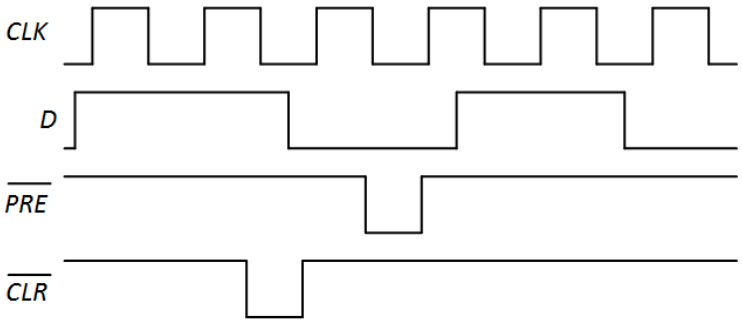


14. For the leader-follower D flip-flop shown in Figure 6.16, show the output of each D latch for the input sequence *D CLK* = 00 ⟶ 10 ⟶ 11 ⟶ 01 ⟶ 00 ⟶ 11.

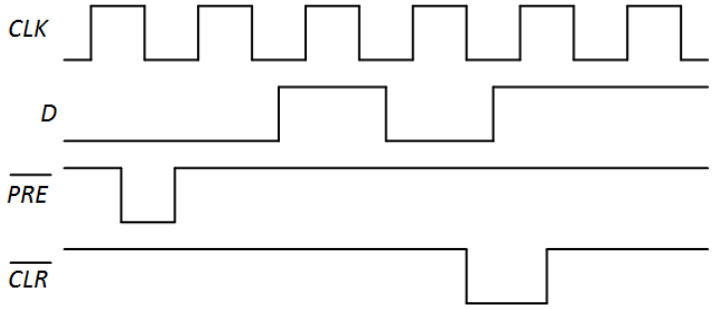
15. Repeat Problem 14 for the input sequence shown in the following timing diagram.



16. For the edge-triggered D flip-flop with preset and clear in Figure 6.18, show the output of each NAND gate for the input values shown in the following timing diagram.

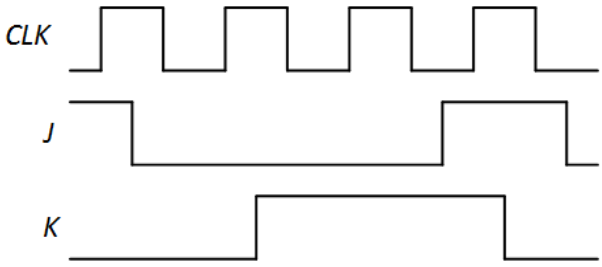


17. Repeat Problem 16 for the input values shown in the following timing diagram.



18. For the leader-follower J-K flip-flop shown in Figure 6.20, show the output of each S-R latch for the input sequence *J K CLK* = 010 → 011 → 000 → 001 → 100 → 101 → 110 → 111.

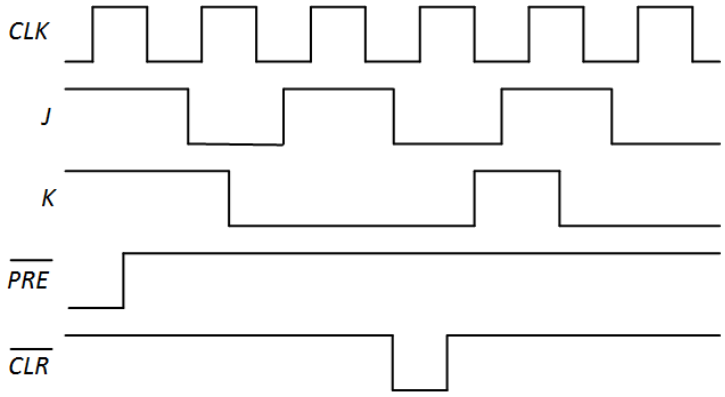
19. Repeat Problem 18 for the input sequence shown in the following timing diagram.



20. Repeat Problem 18 for the positive edge-triggered J-K flip shown in Figure 6.21.

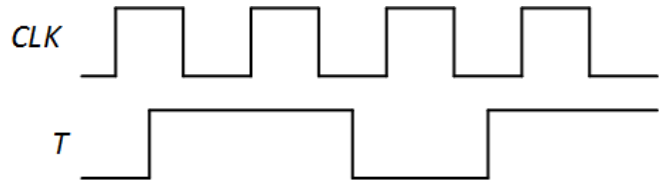
21. Repeat Problem 19 for the positive edge-triggered J-K flip shown in Figure 6.21.

22. For the positive edge-triggered J-K flip-flop with preset and clear shown in Figure 6.22, show the output of each gate for the input values in the following timing diagram.



23. For the T flip-flop shown in Figure 6.24, show the values of *D*, *Q* and $\overline{Q}$ for the input sequence *T CLK* = 00 → 01 → 10 → 111 → 10 → 11.

24. Repeat Problem 23 for the input sequence shown in the following timing diagram.



25. For the T flip-flop shown in Figure 6.25, show the values of *J*, *K*, *Q,* and $\overline{Q}$ for the input sequence shown in Problem 6.23.

26. Repeat Problem 25 for the input sequence shown in Problem 6.24.

27. Design a D flip-flop using a J-K flip-flop. (Hint: Create the excitation table, then determine the functions for *J* and *K*.)

28. Design a D flip-flop using a T flip-flop.

29. Design a J-K flip-flop using a T flip-flop.

Chapter 7

More Complex Sequential Components

Chapter 7: More Complex Sequential Components

Earlier in this book, we saw that there are some combinatorial logic functions that are used so frequently that engineers created components specifically to realize these functions. When a digital designer wants to include a decoder, encoder, multiplexer, or demultiplexers in a design, they can simply use a chip with that function or call up a predefined component in their design software. This is an efficient and cost-effective way to streamline the design process and minimize design errors. The same reasoning can be used to justify the creation of similar components for sequential logic. In this chapter, we examine some of these components, their functions, and their internal designs.

First, we examine **registers**. A flip-flop is sufficient to store a single bit of data, but we often use data consisting of multiple bits, such as the *n*-bit binary numbers we introduced earlier in this book. Registers do just that; they store multi-bit binary values. We will see how they can be designed in a straightforward manner using the flip-flops introduced in the previous chapter.

There are certain types of registers that can not only store data, but can also perform one or more operations on that data. We will look at **shift registers**, which, as the name implies, can shift their bits one place to the right or left. These are particularly useful for data communication, in which a multi-bit value is transmitted one bit at a time. We will introduce several types of shift registers and how they are designed using flip-flops.

Finally, we will present **counters**. They may count up or down, or have the ability to count in either direction. There are counters that count in binary or decimal. A digital circuit may cascade several counters to count a larger number, much as the odometer on an automobile uses several digits to count the number of miles driven. We will show how this works and the signals used by the counters to communicate with each other. We will also examine the methodologies used in the internal designs of these counters.

7.1 Registers

In digital circuits, it is fairly common to store and perform operations on numeric, binary values. In some cases, the value may be a single bit, and a flip-flop is sufficient to store these values. Most of the time, however, the value will have more than one bit. Although we could use multiple flip-flops to store these values, this can become cumbersome, resulting in additional wiring and greater power usage. For this reason, digital circuit designers developed dedicated chips and components just for this purpose. These are called **registers**.

Internally, you can think of a register as several D flip-flops connected in parallel. Each flip-flop receives one bit of a value as its input and outputs that one bit. Consider, for example, the 8-bit register shown in Figure 7.1. An 8-bit value is input to the register via inputs D_7-D_0 and the output is made available at outputs Q_7-Q_0.

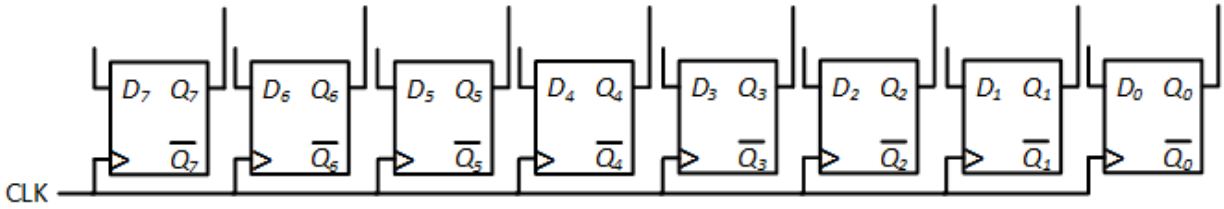


Figure 7.1: 8-bit positive edge-triggered register.

WATCH ANIMATED FIGURE 7.1

Note that all flip-flops within the register use the same clock signal. That is, when we load a value into a register, we load all the bits of the register. Since registers are used to hold a single, multi-bit value, we usually want to load the entire value or nothing at all. You can't load, say, bits 3, 5, and 6 with new values and leave the other bits unchanged. If you need your circuit to be able to load individual bits, you should not use a register; you should use individual flip-flops instead.

In more complex digital systems, such as microprocessors, the system may have a clock signal that is used to synchronize all components within the system. In these systems, good design practice is to use the clock input only for the system clock, and to have a separate signal that indicates when to load the register.

Digital designers have modified the register design shown in Figure 7.1 just for this purpose. They added a separate signal, which we call LOAD. When LOAD = 1, we load the new value into the register on the rising edge of the clock. When LOAD = 0, we do not load in a new value, regardless of the clock value. Figure 7.2 shows a generic 8-bit register with a LOAD signal and a sample timing diagram. The animation shows this progression over time.

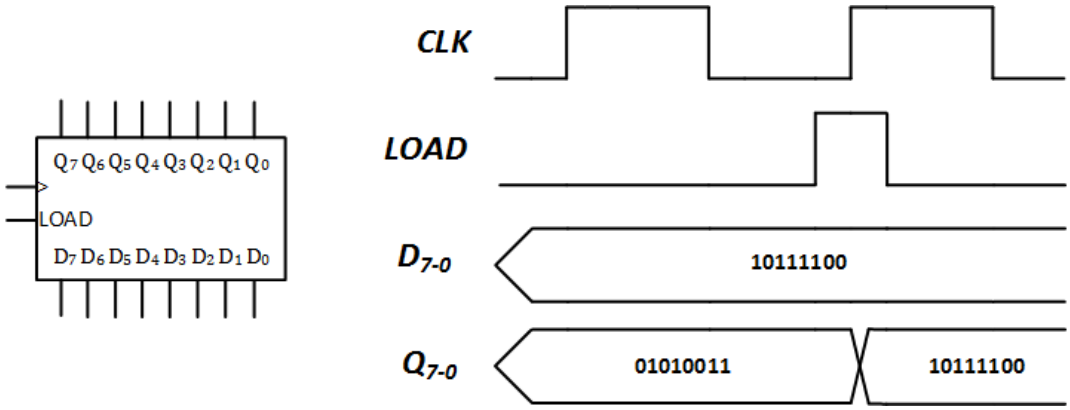


Figure 7.2: 8-bit register with LOAD: (a) Generic diagram showing inputs and outputs; (b) Example timing diagram.

WATCH ANIMATED FIGURE 7.2

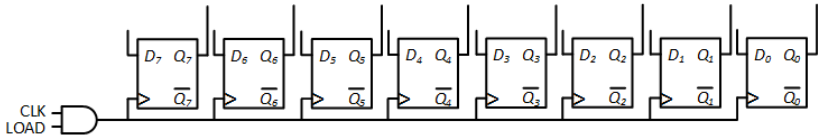
Before you continue reading, think about how you can make the CLK and LOAD signals work with flip-flops that have only a single clock signal.

Hopefully you developed your own method to make the register with CLK and LOAD signals function properly. Here's my solution, which is not the only solution and may or may not match yours. Even if our solutions are not the same, it is still quite possible that your solution is perfectly valid.

I started by creating a truth table, with the LOAD and CLK signals as inputs and the clock signal of the flip-flops as the output. Since we only care about the rising edge of CLK, I use ↑ and not↑ as the value for CLK instead of the usual 0 and 1. This truth table is shown in Figure 7.3 (a).

LOAD	*CLK*	*FF CLOCK*	My choice
0	not ↑	not ↑	0
0	↑	not ↑	0
1	not ↑	not ↑	0
1	↑	↑	↑

(a)



(b)

Figure 7.3: 8-bit register with LOAD: (a) Truth table for flip-flop clock inputs; (b) Final design.

From the table, we see that we must generate a rising edge when LOAD = 1 and CLK = ↑, and any value that is not a rising edge in all other cases. I choose to make this value 0 when LOAD = 0 and the same value as CLK when LOAD = 1. (The latter works both when we do and do not have a rising edge.) To create this value, we can simply AND together the LOAD and CLK signals. Figure 7.3 (b) shows the final design for an 8-bit register with LOAD signal.

7.2 Shift Registers

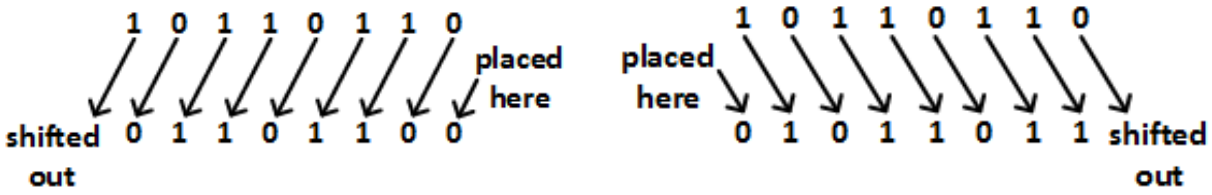
Registers were created to simplify the design process. Instead of wiring up several chips containing individual flip-flops, engineers could use a single chip to store multi-bit values. Registers are very useful, but their functionality is somewhat limited. They can load and store data, but they cannot perform any operations on that data.

For that reason, design engineers created extended versions of registers, components that can load and store data, but also perform specific operations on that data. The designers chose functions that are frequently used by circuit designers. In this section, we look at one class of registers, the **shift register**. First we will introduce the **linear shift** operation, how it works, and the designs of linear shift registers using both D and J-K flip-flops. Then we will discuss ways to combine shift registers for data values with large numbers of bits.

7.2.1 Linear Shifts

Imagine that you are in a small store. There is one line of customers at the checkout. When a customer finishes checking out and leaves, the next customer moves up to the cashier, and everyone else in line moves forward one step. This is essentially how a linear shift operation works.

Instead of customers in a checkout line, consider an 8-bit binary value, for example, 10110110. We can perform a linear shift on this data in one of two ways: linear shift left or linear shift right. First, let's look at the linear shift left. Just as with our line of customers, every bit is shifted one position to the left, as shown in Figure 7.4 (a). This takes care of most of the bits, but it leaves us with two questions. What happens to the most significant bit? And what gets shifted into the least significant bit? Just as the customer who finishes paying leaves the store, the most significant bit also leaves. We say it is shifted out. As for the least significant bit, by definition a linear shift places a 0 in this position.



(a) (b)

Figure 7.4: (a) Linear left shift; (b) Linear right shift.

The linear shift right works in exactly the same way, but in the opposite direction. The least significant bit is shifted out and a 0 is placed in the most significant bit, as shown in Figure 7.4 (b).

7.2.2 Shift Register Design Using D Flip-Flops

Now that we know how linear shift operations work, we can design shift registers to implement these operations. The easiest way to do this is to use flip-flops. First, we will see how to do this using D flip-flops, which is the most straightforward design.

Consider the 8-bit register constructed from eight D flip-flops shown in Figure 7.5. The subscripts 7 to 0 represent the positions of the eight bits, with bit 7 being the most significant and bit 0 being the least significant. When we perform a linear shift right, we want to move the value from bit 7 to bit 6, from bit 6 to bit 5, and so on. More specifically, we want to load the value in bit 7 into bit 6, and so on. To do this, we connect the Q output of bit 7, Q_7, to the D input of bit 6, D_6; we do the same for the other bits. We do not connect Q_0 to anything since that value is shifted out. We connect D_7 to a hard-wired 0 since the linear shift places a 0 in that bit position. Finally, our shift register must have an input signal, which I'll call SHIFT, that causes the shift operation to occur on its rising edge. Putting all of this together gives us the design shown in Figure 7.5.

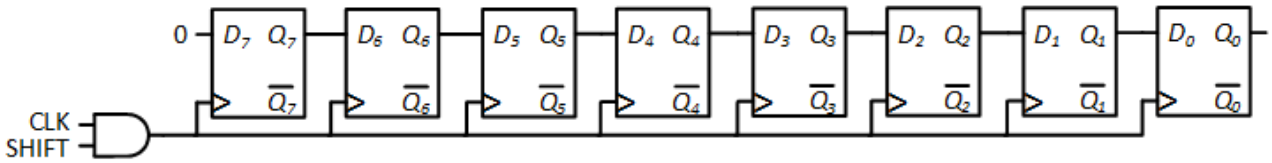


Figure 7.5: 8-bit linear shift right register constructed using D flip-flops.

WATCH ANIMATED FIGURE 7.5

We can use exactly the same design to implement the linear shift left operation. We simply reverse the labels. Instead of labeling the flip-flops from 7 to 0 (left to right), we number them from 0 to 7. It's perfectly acceptable to do this. The flip-flops don't really care what you call them. The only thing that matters to them is how their inputs, outputs, and clock signals are connected. Figure 7.6 shows this configuration.

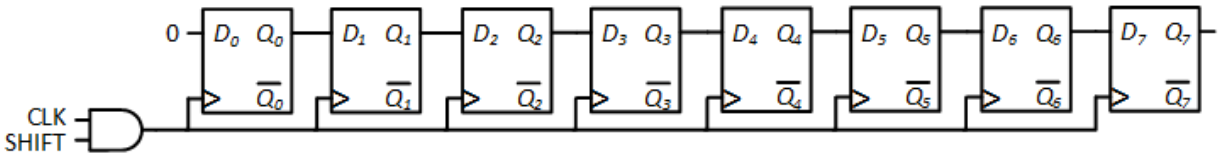


Figure 7.6: 8-bit linear shift register relabeled for linear shift left operation.

WATCH ANIMATED FIGURE 7.6

There is one extremely important point to emphasize when it comes to shift registers. **All the shifts occur at exactly the same time**. The values that are shifted are the values in each flip-flop at the beginning of the shift operation. We don't shift a value from bit 7 to bit 6, and then shift that value from bit 6 to bit 5, and so on. We shift the original value from bit 7 to bit 6, the original value from bit 6 to bit 5, and so on. This ensures that our design implements the linear shift operation properly.

One final point I want to mention concerns the 0 that is loaded into the most significant (linear shift right) or least significant (linear shift left) bit. In most shift register designs, there is a dedicated input called SHIFT_IN that supplies this input. This is particularly useful when we cascade shift registers to accommodate larger numbers; we'll examine this later in this section. For now, we have hard-wired a logic 0 to this input for the designs presented here.

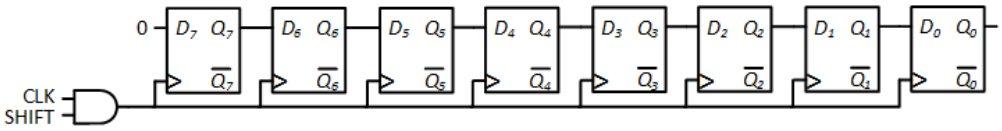
7.2.3 Shift Register Design Using J-K Flip-Flops

We can also design a shift register using J-K flip-flops. Just as before, we use one flip-flop for each bit and use the outputs of each flip-flop to generate the inputs of the next flip-flop. Here, the key is to determine the functions for inputs *J* and *K* for each flip-flop.

Figure 7.7 (a) shows the excitation table for a J-K flip-flop within the shift register. In this table, *Q* is the output of the previous flip-flop, and also the value we want to load into this J-K flip-flop.

Q	J	K
0	0	1
1	1	0

(a)



(b)

Figure 7.7: Linear shift register constructed using J-K flip-flops: (a) Excitation table; (b) Final design.

WATCH ANIMATED FIGURE 7.7

Looking at the excitation table, it is fairly easy to see that $J = Q$ and $K = \overline{Q}$. Since the J-K flip-flop produces both Q and $\overline{Q}$ outputs, they can be connected directly to the inputs of the next flip-flop. We do this for every flip-flop except the first one. Since the linear shift operation must set that bit to 0, we set $J = 0$ and $K = 1$ for that flip-flop. The final design for the register that shifts to the right is shown in Figure 7.7 (b). We can simply relabel the bits as we did for the shift register constructed using D flip-flops to make this shift register perform a shift to the left.

If our register has a SHIFT_IN signal, we would handle the first bit slightly differently. Instead of hard-wiring $J = 0$ and $K = 1$, we would use SHIFT_IN to generate the J and K inputs. When SHIFT_IN = 0, we want to set the flip-flop to 0, which we do by setting $J = 0$ and $K = 1$. If SHIFT_IN = 1, we set the output to 1 by setting $J = 1$ and $K = 0$. Combining these two gives us the functions J = SHIFT_IN and K = SHIFT_IN′. The shift register with a SHIFT_IN signal is shown in Figure 7.8.

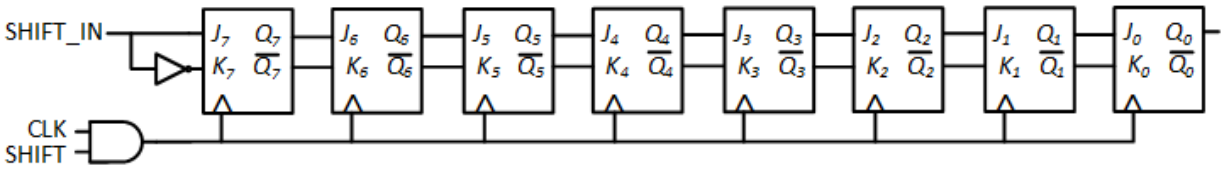


Figure 7.8: Linear shift register with SHIFT_IN signal constructed using J-K flip-flops.

7.2.4 Bidirectional Shift Registers

You may need to design a circuit that can shift its data either left or right. A single input, which we'll call SHIFT_DIR, would indicate the direction to shift the data; we will use SHIFT_DIR = 0 for shift left and SHIFT_DIR = 1 for shift right. You cannot use the designs we've seen so far; they can only shift data in one direction. However, we can modify these designs to create a bidirectional shift register. We'll modify our design using D flip-flops.

The key to this design is to develop functions for each flip-flop input and the circuitry to realize these functions. Let's start by considering one of the flip-flops in the middle of the shift register, say bit 4. If SHIFT_DIR = 1, we want to shift the contents of the register one bit to the

right. Bit 4 must get the value of bit 5, which is available on its output, Q_5. If SHIFT_DIR = 1, $D_4 = Q_5$. If SHIFT_DIR = 0, however, we are shifting the data left by one bit, and bit 4 must get the value of bit 3; when SHIFT_DIR = 0, $D_4 = Q_3$.

We can combine the two cases to generate D_4 by using combinatorial logic gates or a 2 to 1 multiplexer. Figure 7.9 shows one possible design using multiplexers. Notice that the designs for the inputs to the bit 7 and bit 0 flip-flops are slightly different. Rather than getting one of two flip-flop outputs as their *D* input value, they get either one flip-flop value or the SHIFT_IN value.

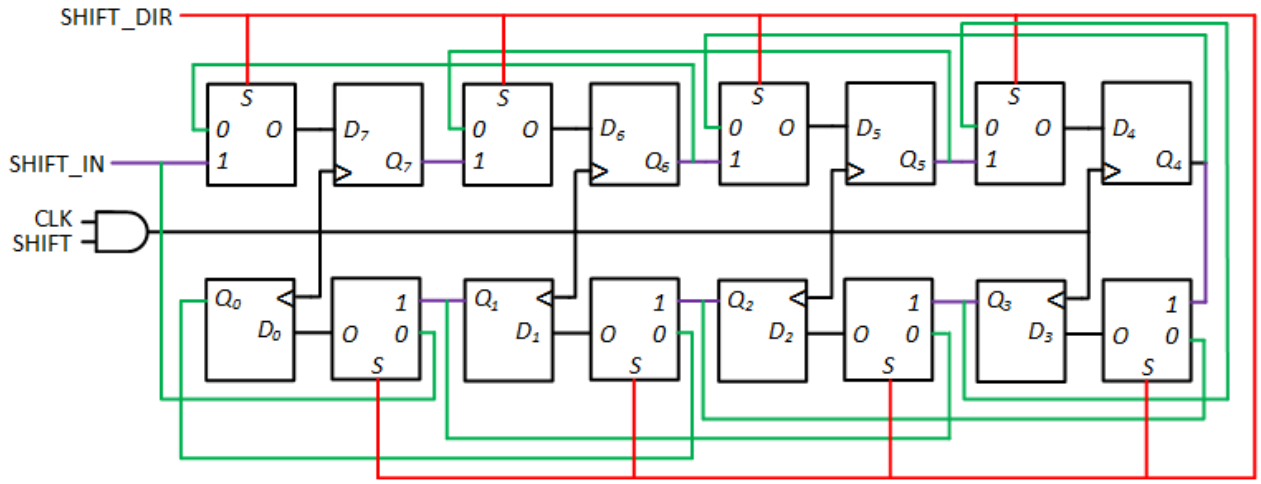


Figure 7.9: Bidirectional shift register constructed using D flip-flops and multiplexers, color coded as follows: RED = direction of shift (0=left, 1=right); GREEN = data paths for left shift; PURPLE = data paths for right shift; BLACK = clock and data path from multiplexers to flip-flops.

WATCH ANIMATED FIGURE 7.9

If we do not have a SHIFT_IN signal on our shift register, we can use the same design to perform a regular linear shift operation. The only change we need to make to our design is to replace SHIFT_IN with a hardwired 0.

7.2.5 Shift Registers with Parallel Load

It is very useful to be able to load a shift register in parallel, that is to load data into all the bits of a shift register simultaneously, just as we load all bits of a traditional register. In computer systems, a microprocessor may send data to a specialized chip called a **Universal Asynchronous Receiver/Transmitter**, or **UART**. The microprocessor sends the data in parallel, and the UART outputs the data to a modem or other serial device one bit at a time. As you may have guessed, UARTs incorporate shift registers designed to perform a parallel load in their designs.

We can modify the designs presented previously in this section to add the ability to load the shift register in parallel. We need to add some input signals to do this. First, we add eight

inputs, I_7 to I_0. When we load data in parallel, the data will be placed on these inputs. We also need to add a signal that tells the shift register to load the data; we'll just call that signal LOAD.

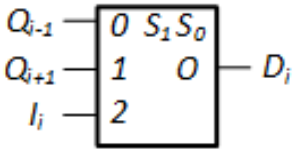
With these additional signals, we can modify any of the shift registers presented so far. As an example, we will modify the bidirectional shift register shown in Figure 7.9. We start by creating the truth table for our design, which is given in Figure 7.10. As shown in this table, one of four things can happen. When LOAD = 0 and SHIFT = 0, we do not want to load in a new value nor shift the current value. We want to keep the value in the register unchanged. If LOAD = 0 and SHIFT = 1, we want to shift the data. The value of SHIFT_DIR determines whether we should shift left (SHIFT_DIR = 0) or right (SHIFT_DIR = 1). Finally, if LOAD = 1 we must load in the data on the data inputs.

Notice what happens when LOAD = 1 and SHIFT = 1. You cannot both load and shift data simultaneously. You must choose one or the other. Since I'm writing this book, I get to choose, and I choose to give LOAD priority over SHIFT. If both signals are high, our register will load data.

LOAD	*SHIFT*	*SHIFT_DIR*	Q
0	0	X	Q_0
0	1	0	Q_{6-0},*SHIFT_IN*
0	1	1	*SHIFT_IN*, Q_{7-1}
1	X	X	I_{7-0}

Figure 7.10: Truth table for the bidirectional shift register with parallel load.

As shown in the truth table, there are three possible values we can load into each flip-flop: the next lower bit or SHIFT_IN (shift left); the next higher bit or SHIFT_IN (shift right); or the value on input *I* (load). To do this, we follow the same design methodology we used for the original bidirectional shift register: we use a multiplexer to select the correct input. Since we have three inputs, we need a larger multiplexer, a 4 to 1 multiplexer. (Multiplexers always have the number of inputs equal to a power of 2. Nobody makes a 3 to 1 multiplexer. We just don't use the extra input and we make sure we never select that input.) Figure 7.11 (a) shows how I chose to assign the possible data values to the multiplexer inputs.



(a)

LOAD	*SHIFT*	*SHIFT_DIR*	S_1	S_0
0	0	X	X	X
0	1	0	0	0
0	1	1	0	1
1	X	X	1	0

(b)

Figure 7.11: Multiplexer to generate D_i: (a) Data connections – replace Q_{i-1} or Q_{i+1} with SHIFT_IN for D_0 and D_7, respectively; (b) Excitation table for multiplexer select signals.

Next, we must determine the functions for S_1 and S_0 to select the correct input. We create the excitation table shown in Figure 7.11 (b) and develop the function for each

multiplexer select signal. We can set S_1 = LOAD and S_0 = LOAD′ ^ SHIFT ^ SHIFT_DIR. There is a simpler function for S_0; finding this function is left as an exercise for the reader.

The last step is to determine when to load data into the flip-flops. We do not want to load new data on every rising edge of the clock; when LOAD = 0 and SHIFT = 0, we want to leave the data unchanged. We only want to change the data when one or both of these signals are 1 and the clock is rising. The function to realize these conditions is (LOAD + SHIFT) ^ CLOCK. Putting all of this together gives us the final design shown in Figure 7.12.

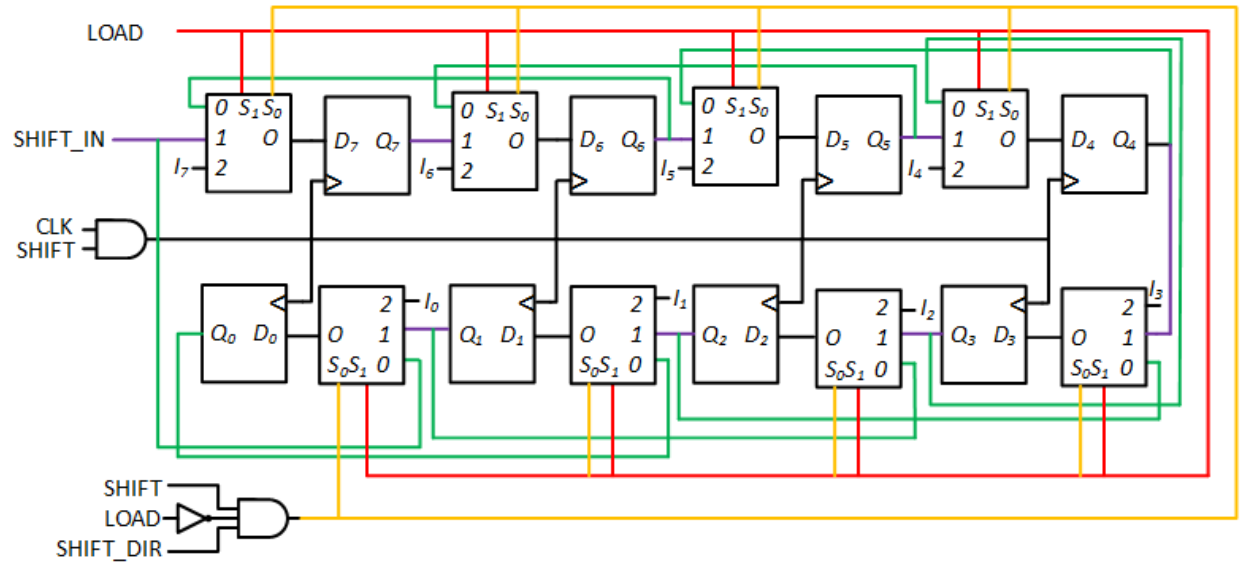


Figure 7.12: Bidirectional shift register with parallel load, color coded as follows: RED and ORANGE = multiplexer select signals; GREEN = data paths for left shift; PURPLE = data paths for right shift; BLACK = clock and data path from multiplexers to flip-flops.

7.2.6 Combining Shift Registers

It is not feasible to design shift registers for all possible applications. Consider, for example, a circuit that, for some reason, needs to shift data that has 128 bits. Designing a single TTL chip to incorporate such a large shift register would not be feasible for several reasons. One has to do with the number of pins on the chip. Shift registers can output their current values, that is, the output of each flip-flop is connected to an output pin on the chip. The only chips with enough pins for all these outputs are the **pin grid array** chips used for microprocessors and advanced programmable gate arrays.

To get around these practical restrictions, designers created shift registers that can be combined to form larger shift registers. To illustrate this, consider how we would create a 16-bit shift right register using two 8-bit shift registers.

The two registers take care of shifting their own data internally, and we can shift in data using the SHIFT_IN signal of one of the registers. But one bit must be shifted out of one register and into the other. The way to do this is to connect the output for the least significant bit of the

first register, the bit that will be shifted out, to the SHIFT_IN input of the second register, as shown in Figure 7.13.

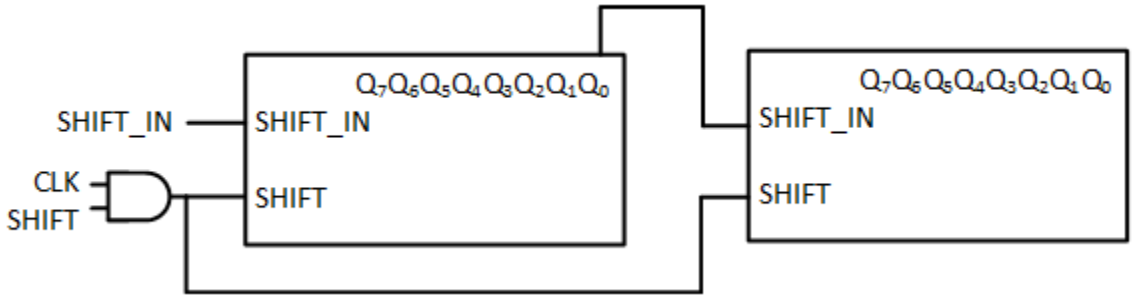


Figure 7.13: 16-bit shift register constructed from two 8-bit shift registers.

WATCH ANIMATED FIGURE 7.13

With the data connections completed, the final task is to set up the control signals, just the SHIFT signal in our design. The way we do this is to connect the SHIFT inputs of the two shift registers together. When the SHIFT input is asserted, this ensures that both registers shift their data at the same time, which is what we want to happen.

To expand this even further, we could combine more registers using the same methodology to create even larger shift registers. However, there is one thing you have to consider. You must ensure that the fan-out of the SHIFT signal is sufficient to supply a valid signal to all of the shift registers. If it is not sufficient, you can use buffers to alleviate this problem, just as we did for the combinatorial circuits in Chapter 3; see Figure 3.16 in subsection 3.4.3.

7.3 Counters

So far in this chapter, we have seen components that store data (registers) and components that move individual data bits (shift registers). There are also components that modify stored data by performing arithmetic operations on the data. One class of this type of components is **counters**.

As the name implies, counters count. That is, they may increment (add 1 to) or decrement (subtract 1 from) their value. There are counters that operate on binary or decimal data, and it is possible to design a counter for any numeric base. In this chapter, we will examine designs for these types of counters.

When we increment a number, we may change more than one bit. For example, incrementing the binary value 0111 gives us 1000, which changes all four bits. As we design the various counters, we will examine ways to tell the correct bits to change their values when incrementing or decrementing. We will also introduce carry and borrow bits and see how they can be used to combine counters for larger numbers of bits.

7.4 Binary Counters – Function

There are many applications that require a circuit to count something. A car odometer and a digital clock are just two examples. The circuit we discussed in Chapter 6, which outputs a 1 when three 1s have been input, can also use a counter to realize its function. More complex circuits, such as microprocessors, often incorporate counters within their designs.

Many (but not all) of these counters are simple binary counters. These counters store a binary value, like a register. When a COUNT signal is asserted, the counter adds 1 to (or subtracts 1 from) that value and stores the result as its new value. For example, a 3-bit **upcounter** (a fancy way of saying a counter that increments, or counts up) will go through the following sequence.

$$000 \rightarrow 001 \rightarrow 010 \rightarrow 011 \rightarrow 100 \rightarrow 101 \rightarrow 110 \rightarrow 111 \rightarrow 000 \rightarrow$$

In decimal, this sequence is $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 0 \rightarrow$

Just like an odometer, when it reaches its largest value, the next increment operation brings the value back to the lowest value – from 111 (7) to 000 (0) in this case. And just like an odometer, this may cause the next digit to be incremented, as when an odometer increments from 19 to 20. If each digit has its own counter, we need a way for one counter to tell the next counter that it has gone from the highest value to the lowest. In the odometer example, the ones digit counter needs to inform the tens digit that it also must be incremented. To do this, counters typically have a **carry out** signal that is set to 1 when this happens. The next counter will use this signal to determine when it must increment its value.

Internally, we must do the same for the bits within the counter. For our 3-bit counting sequence, for example, consider the transition from 001 to 010. The least significant bit changes from its maximum value, 1, to its minimum value, 0. It must signal the next bit so that it changes from 0 to 1, otherwise we will not produce the correct value of 010. In the next section, we will introduce two design methodologies for counters that address this design task in two different ways.

Downcounters work in a similar manner, but in the opposite direction. A 3-bit downcounter goes through the sequence:

$$111 \rightarrow 110 \rightarrow 101 \rightarrow 100 \rightarrow 011 \rightarrow 010 \rightarrow 011 \rightarrow 000 \rightarrow 111 \rightarrow$$

In decimal, this is $7 \rightarrow 6 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 0 \rightarrow 7 \rightarrow$

Just as upcounters have a carry bit to indicate when the counter goes from its maximum value to its minimum value, downcounters have a **borrow** signal that is asserted when it loops back from its minimum value to its maximum value.

There are also counters that can count in either direction called **up-down counters**. Like the bidirectional shift register, these counters have an additional input that determines whether the counter increments or decrements its value. Also like the shift registers, many counters can load data in parallel. Finally, many counters can clear their data, setting all bits to 0. We'll look at these options as we design counters in the remainder of this section.

7.5 Binary Counters – Design

Now that we have established what binary counters do, we can design them. There are two types of designs for these counters based on how one digit rolling over causes the next digit to increment or decrement, as happens with the odometer in the previous section increments from 19 to 20. These two methodologies, **ripple counters** and **synchronous counters**, are described in the two subsections that follow.

Before we look at these methodologies, I want to introduce a couple of things that are common to all binary counters, regardless of which methodology they use. The first thing is the number of flip-flops used to store the value within the counter. An *n*-bit counter must always have *n* flip-flops, one for each bit of the count value. This is the case regardless of which type of flip-flop is used. We'll look at examples of counters using D and J-K flip-flops in the following subsections. This may seem intuitively obvious, but it's best to specify this right up front.

The second commonality is not quite so obvious, but is important to note when designing the counters. Look back at the sequence for the 3-bit count up, which goes from 000 to 111 and then back to 000. For every bit, we either leave the bit unchanged, or change it from 0 to 1 or from 1 to 0. Rephrasing this, we either do not load a new value into the flip-flop for a bit or we complement its value. Thinking of the function in this way will simplify our work when we design our counters.

7.5.1 Ripple Counters

I think the best way to illustrate the design of ripple counters is to jump right into a design. So, let's start by designing a 4-bit ripple counter using positive edge-triggered D flip-flops. There are two steps in the design process: establish the data paths and implement the clock signals. There are two possible functions for every flip-flop in our counter at any given time; either:

- *Do not load a new value into the flip-flop*: If you don't want to load in a new value, don't send a positive edge to the clock. The flip-flop only loads data when the clock goes from 0 to 1. If it stays at 0, or at 1, or it goes from 1 to 0, the flip-flop does not load in data; it retains its previous value.

  Or

- *Complement its value*: When we do want to change the value of a flip-flop, we load the complement of the value. The flip-flop outputs its value on its *Q* output and the complement of its value on its $\overline{Q}$ output. If we connect $\overline{Q}$ to the *D* input for a flip-flop, it will load its complement on the rising edge of its clock input.

Putting these together allows us to design the data paths, the connections used for the data as opposed to the clocks. This partial design is shown in Figure 7.14.

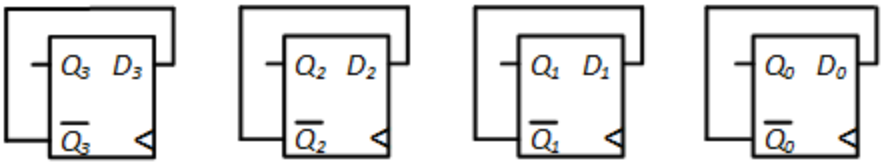


Figure 7.14: Data paths for the 4-bit ripple counter constructed using D flip-flops.

Now we need to tell each flip-flop when to load in a new value. We'll start with the least significant bit, the rightmost flip-flop in this design. Consider the complete sequence for a 4-bit upcounter:

$$000 \rightarrow 001 \rightarrow 010 \rightarrow 011 \rightarrow 100 \rightarrow 101 \rightarrow 110 \rightarrow 111 \rightarrow 000 \rightarrow$$

Whenever our COUNT signal is 1, we need to invert the least significant bit on the rising edge of the CLOCK input. We can do this by ANDing together these two signals. Adding this to our circuit gives us the partial design shown in Figure 7.15.

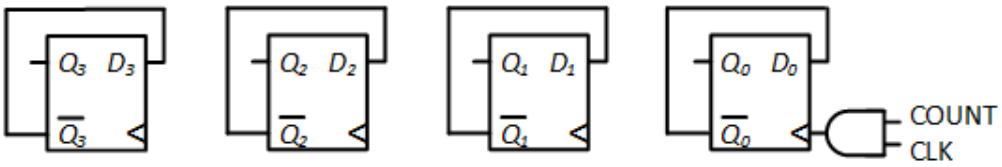


Figure 7.15: 4-bit ripple counter with data paths and clock input for the least significant bit.

Next, let's look at the next bit. Going back to the count sequence, we find that bit 1 changes only when bit 0 changes from 1 to 0. When this happens, Q_0 changes from 1 to 0 and $\overline{Q}_0$ changes from 0 to 1. If we connect $\overline{Q}_0$ to the clock input of the flip-flop for bit 1, it will invert bit 1 exactly when we want it to.

We can repeat this process for bits 2 and 3 to see that $\overline{Q}_1$ is connected to the clock input for bit 2, and $\overline{Q}_2$ serves as the clock input for bit 3. The complete design for the 4-bit ripple counter is shown in Figure 7.16.

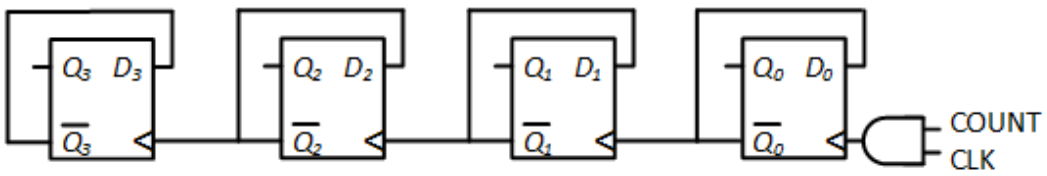


Figure 7.16: 4-bit ripple counter – final design.

WATCH ANIMATED FIGURE 7.16

If we wish to design this counter using J-K flip-flops instead of D flip-flops, we can easily modify our design. Instead of connecting $\overline{Q}$ to our inputs, we simply set $J = K = 1$. When $JK = 11$, we complement the value stored in the J-K flip-flop on the rising edge of the clock. Since we still want to invert our counter bits at the same time as we did in the design using D flip-flops, our clock signals remain the same. This design is shown in Figure 7.17.

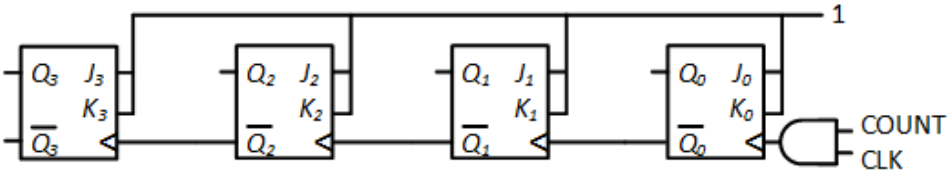


Figure 7.17: 4-bit upcounter constructed using J-K flip-flops.

In addition to the bits that contain the count value, many counters also have one additional output bit called the **carry out**, or **carry bit**, or just **carry**. The counter sets this bit to 1 when its count value changes from all 1s to all 0s, or from 1111 to 0000 for our 4-bit counters. The counter does not use a register for the carry bit; it generates the carry using combinatorial logic. The carry is set to 1 when $Q = 1$ for all flip-flops in the counter and COUNT = 1. The circuitry to add a carry to our 4-bit counters is shown in Figure 7.18 (a). The timing diagram in Figure 7.18 (b) shows how the counter outputs change as its value changes from 1111 to 0000. Note that the carry is 1 until the value changes. Once it does change, the Q values input to the AND gate become 0 and the carry output immediately changes from 1 to 0.

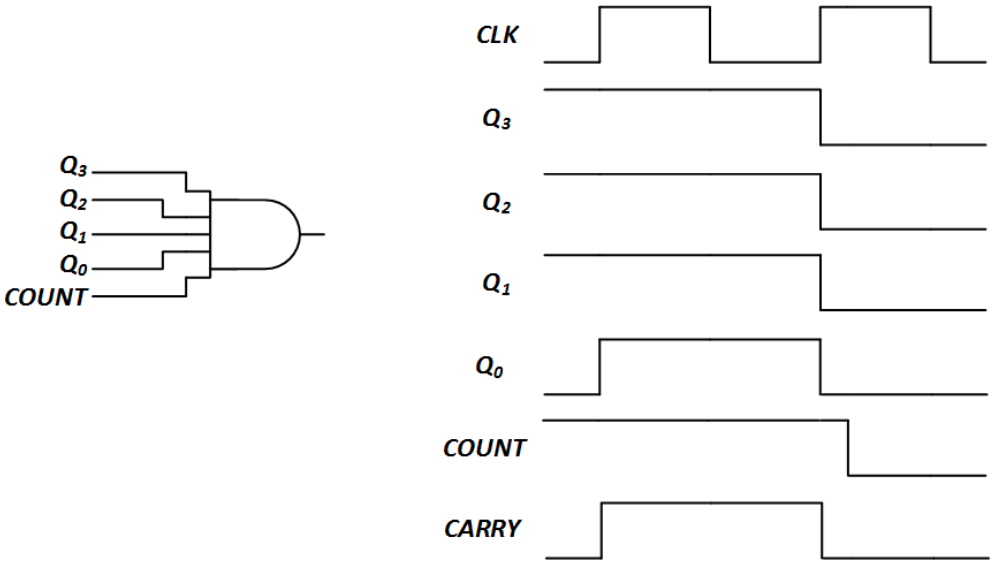


Figure 7.18: Counter carry-out: (a) Circuit to generate CARRY; (b) Sample timing diagram setting CARRY = 1.

WATCH ANIMATED FIGURE 7.18

7.5.1.1 Downcounter

For some applications, you might need to count down instead of up. As it turns out, we can modify our upcounter fairly easily to change it into a **downcounter**. Consider the sequence for a 4-bit downcounter shown here.

$$1111 \rightarrow 1110 \rightarrow 1101 \rightarrow 1100 \rightarrow 1011 \rightarrow 1010 \rightarrow 1011 \rightarrow 1000 \rightarrow$$
$$0111 \rightarrow 0110 \rightarrow 0101 \rightarrow 0100 \rightarrow 0011 \rightarrow 0010 \rightarrow 0011 \rightarrow 0000 \rightarrow 1111 \rightarrow$$

In the upcounter, whenever a bit changed from 1 to 0 we would also need to change the next most significant bit. For the downcounter sequence, the opposite is true. We change the next most significant bit when a bit changes from 0 to 1. For example, when we change the value from 0110 to 0101, the least significant bit changes from 0 to 1, so the next significant bit must also change, from 1 to 0 for these values. We can do this by using *Q* instead of $\overline{Q}$ as our clock signal. This design is shown in Figure 7.19.

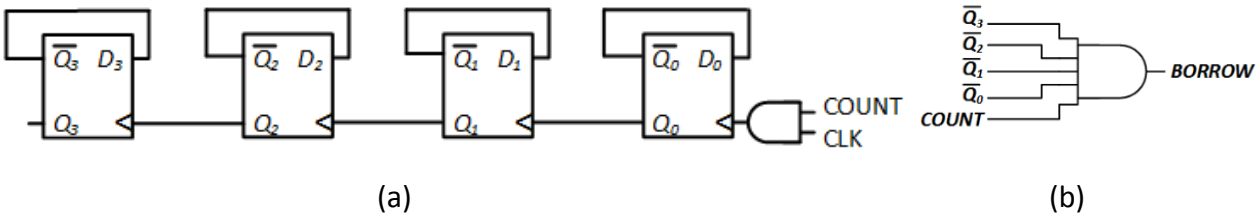


Figure 7.19: Ripple downcounter: (a) Circuit design; (b) Circuit to generate the BORROW signal.

Instead of a CARRY signal, downcounters have a BORROW signal that is set to 1 when going from a value of all 0s to all 1s. It is set to 1 when all the *Q*s are 0 (or all the $\overline{Q}$s are 1) and COUNT = 1. The circuit to generate the BORROW signal is combinatorial, just like the circuit that generates CARRY in the upcounter. This circuit is shown in Figure 7.19 (b).

7.5.1.2 Up/Down Counter

Earlier in this chapter, we combined the designs for registers that shift right and shift left to create a bidirectional shift register. Here, we will follow a similar procedure to create a single counter that can count either up or down. We will add a new signal, $U/\overline{D}$, to indicate whether we count up ($U/\overline{D}$ = 1) or down ($U/\overline{D}$ = 0). This is similar to the SHIFT_DIR signal in the bidirectional shift register.

There is one very important difference in our design procedure. In the shift register design, our data varied depending on the direction of the shift, but our clock signals were always the same. It is exactly the opposite for the up/down counter. The data inputs to the flip-flops are always the same; *D* = $\overline{Q}$ whether we are counting up or down. It is the clock that varies.

With this in mind, we can use a multiplexer to select which value to use for our clock signals. If $U/\overline{D}$ = 0, we should use *Q*; if $U/\overline{D}$ = 1, we use $\overline{Q}$. This circuit is shown in Figure 7.20.

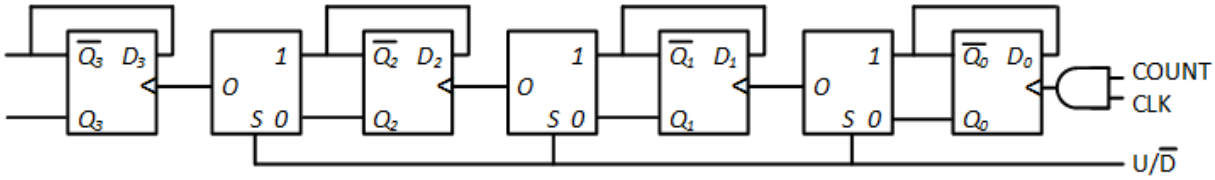


Figure 7.20: 4-bit up/down counter.

WATCH ANIMATED FIGURE 7.20

Bidirectional counters typically combine the CARRY and BORROW signals into a single output signal. This design is left as an exercise for the reader.

7.5.1.3 Additional Signals

It is possible to enhance the capabilities of our counters to include additional functions. For counters, the two most common functions are to load a value into the counter and to clear the counter, that is, set its value to zero. Both modifications are left as exercises for the reader, but here are some hints about how to proceed.

You can add the capability to load data into the counter in the same way that we added this function to the bidirectional shift register in Section 7.2. You will need a LOAD signal that causes the counter to load the data on its inputs I_3 to I_0 into its flip-flops on the rising edge of the clock. You will need to modify both the data inputs and the clock signal of the flip-flops to implement this function.

To enable the counter to clear its value, you will need to add a CLEAR input signal. Using D flip-flops with preset and clear inputs will greatly simplify your design.

7.5.1.4 Why Are They Called Ripple Counters?

Given these designs, why are they called ripple counters? Think about this for half a minute or so, then read on.

The ripple in ripple counters has to do with how the clock signals propagate through the counter, from one flip-flop to the next. Consider the transition from 0111 to 1000. The COUNT signal is high and the CLOCK signal changes from 0 to 1. This generates a rising edge on the clock input to the flip-flop for bit 0. That bit changes from 1 to 0. As it changes, the output for $\overline{Q}_0$ changes from 0 to 1, which sends a rising edge to the clock for bit 1. This bit also changes from 1 to 0, $\overline{Q}_1$ changes from 0 to 1 and it generates a rising edge on the clock for bit 2. Bit 2 does exactly the same thing, sending a rising edge to the flip-flop for bit 3. The flip-flops all toggle their values, but not simultaneously. Bit 0 toggles first, then bit 1, followed by bit 2, and finally

bit 3. The changes move through the counter much like the ripples on a pond after something disturbs the still water.

The ripple design is very efficient in terms of hardware, but it does have a very significant drawback. It takes a relatively long time to set the final value of the counter if you are changing the value of all or most of the bits. We will introduce another design methodology in the next subsection that was developed specifically to address this concern.

7.5.2 Synchronous Counters

In Chapter 4, we introduced carry-lookahead adders. These adders incorporate logic to generate the carry out signals of the full adders within adders without having to wait for carry bits to propagate (ripple) through the full adder. In this section, we will follow the same methodology to speed up our counters. As an example, we will redesign the 4-bit upcounter as a synchronous counter, one that sets all its bits simultaneously. First, let's go back to the sequence for this counter.

$$0000 \rightarrow 0001 \rightarrow 0010 \rightarrow 0011 \rightarrow 0100 \rightarrow 0101 \rightarrow 0110 \rightarrow 0111 \rightarrow 1000 \rightarrow 1001 \rightarrow 1010 \rightarrow 1011 \rightarrow 1100 \rightarrow 1101 \rightarrow 1110 \rightarrow 1111 \rightarrow 0000 \rightarrow$$

Looking at the least significant bit, we can easily see that it changes every time the counter counts up, which occurs when COUNT = 1. The second bit changes when the least significant bit is 1 and we count up, or when $Q_0 = 1$ and COUNT = 1. The third bit changes when the last two bits are both 1 and we count up, or when $Q_1 = 1$, $Q_0 = 1$ and COUNT = 1. You may have noticed a pattern already, or you may simply check the sequence to realize that the most significant bit changes when $Q_2 = 1$, $Q_1 = 1$, $Q_0 = 1$ and COUNT = 1. We can summarize this as follows.

For every bit in the counter, the bit changes on the rising edge of the clock if every less significant bit is 1 and COUNT is 1.

With this in mind, we can create a new design for the 4-bit counter, as shown in Figure 7.21. Our design will be a synchronous counter that loads all bits simultaneously, that is, they will all have the same clock input. For our counter, we AND together the CLK and COUNT inputs.

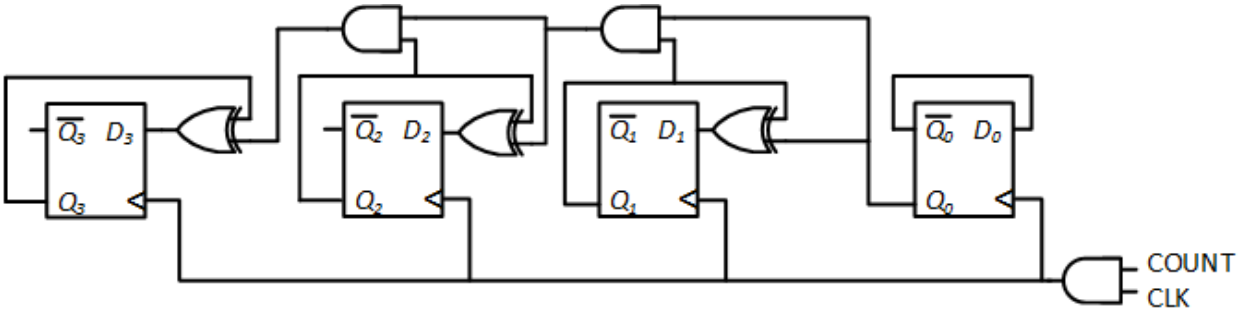


Figure 7.21: 4-bit synchronous upcounter.

WATCH ANIMATED FIGURE 7.21

The data inputs generate one of two values. If all less significant bits are 1, we want to load the counter with the complement of its current value. Otherwise, we want to load in the same value it currently has stored so that its output stays the same. For the least significant bit, there are no less significant bits and we always want to change that bit; inputting $\overline{Q}$ on the D input accomplishes this task.

For the other bits, an XOR gate realizes this function. One input is Q. If the other input is 0, $Q \oplus 0 = Q$ and we reload the current value. However, if the other input is 1, $Q \oplus 1 = \overline{Q}$ and we complement the value in the flip-flop. We want this second input to be 1 when all less significant bits are 1. This is the function we implement for this XOR gate input for each flip-flop. For D_1, we only need to check Q_0. D_2 needs both Q_0 and Q_1 to be 1. And D_2 requires Q_0, Q_1, and Q_2 all to be 1.

The design for the synchronous counter can be derived almost directly from the carry-lookahead adder with $C_{in} = 0$. The carry-lookahead adder from Figure 4.31 is repeated in Figure 7.22.

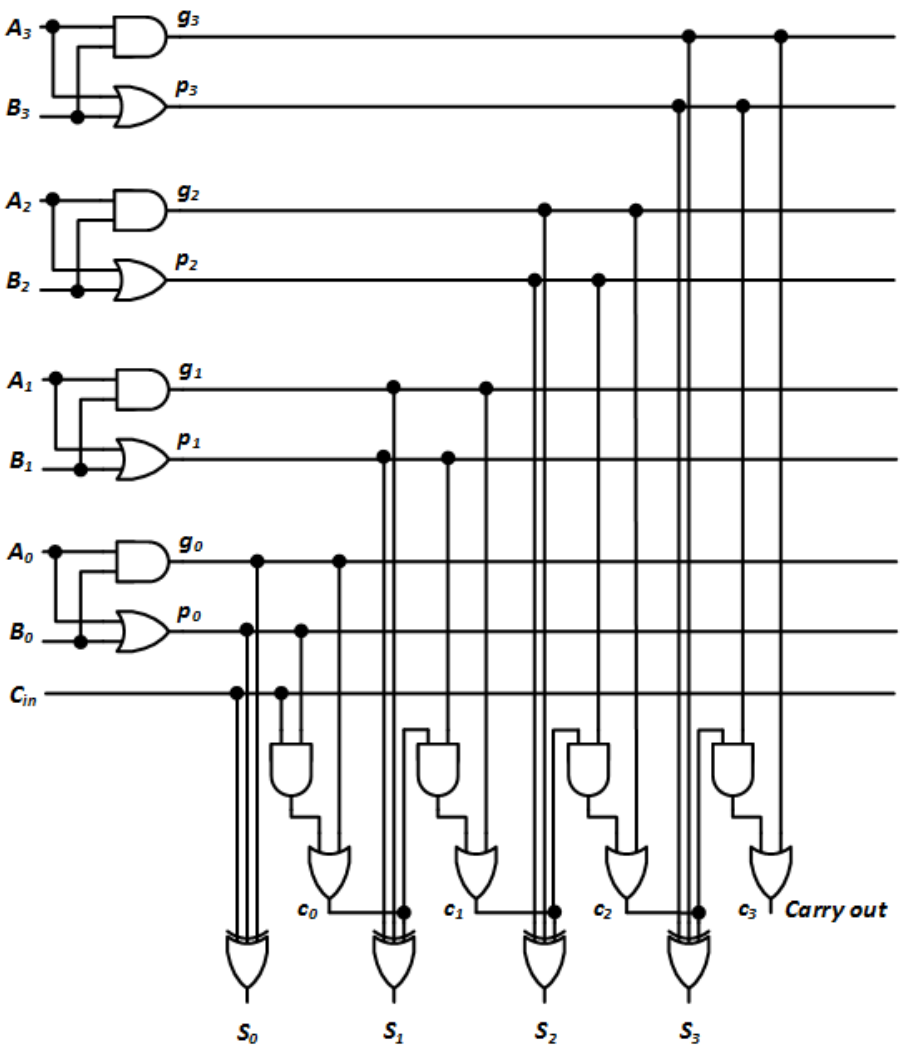


Figure 7.22: 4-bit carry-lookahead adder.

Instead of adding $A_3A_2A_1A_0 + B_3B_2B_1B_0$, however, we will add $A_3A_2A_1A_0 + 1$. This sets our *p* and *g* signals as follows:

$$p_0 = A_0 + B_0 = A_0 + 1 = 1$$
$$g_0 = A_0 \wedge B_0 = A_0 \wedge 1 = A_0$$
$$p_1 = A_1 + B_1 = A_1 + 0 = A_1$$
$$g_1 = A_1 \wedge B_1 = A_1 \wedge 0 = 0$$
$$p_2 = A_2 + B_2 = A_2 + 0 = A_2$$
$$g_2 = A_2 \wedge B_2 = A_2 \wedge 0 = 0$$
$$p_3 = A_3 + B_3 = A_3 + 0 = A_3$$
$$g_3 = A_3 \wedge B_3 = A_3 \wedge 0 = 0$$

In Figure 7.23, we modify the carry-lookahead adder to include these simplified signal functions.

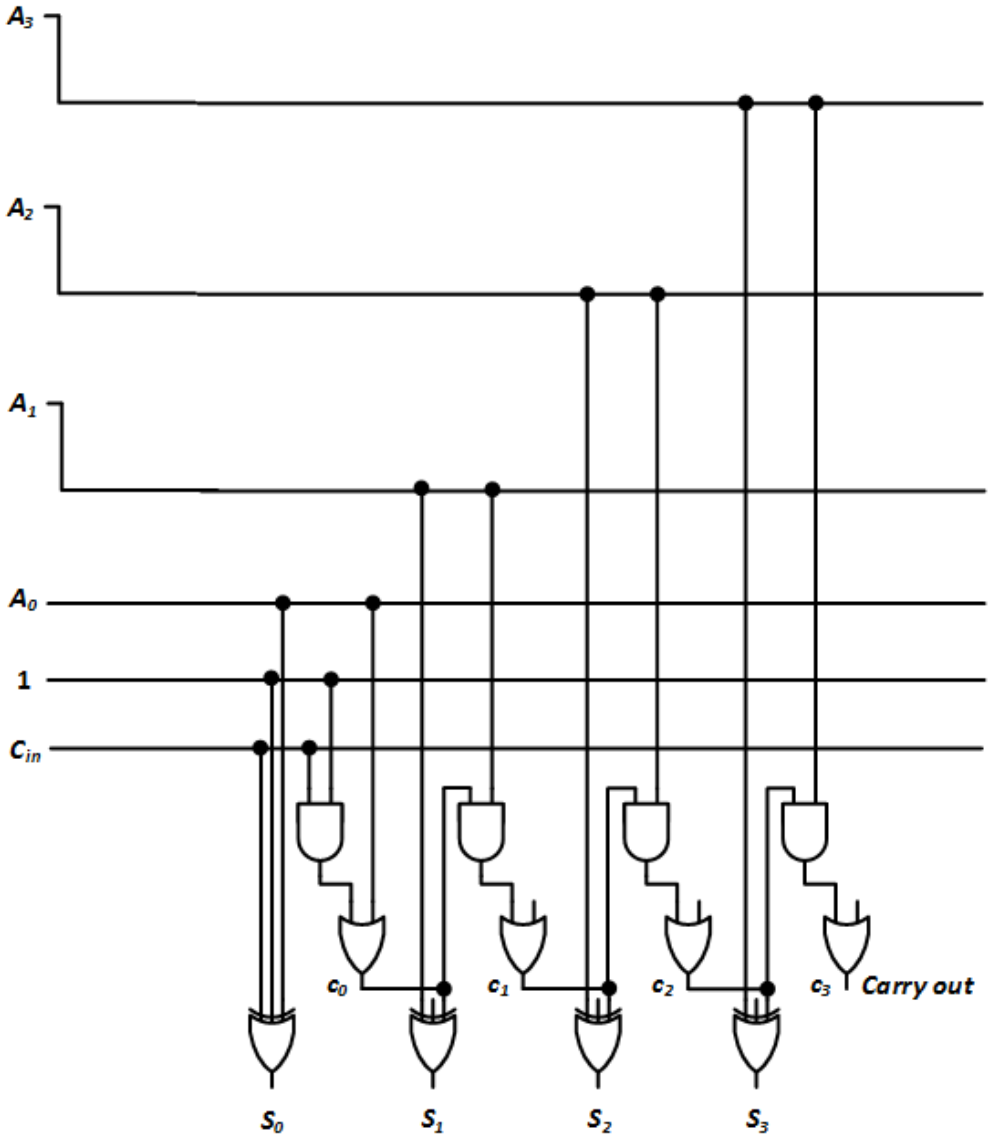


Figure 7.23: 4-bit carry-lookahead adder with simplified *p* and *g* signals when *B* = 0001.

Next, we move to generate the carry signals. Our counter will not have a C_{in} signal, so we set $C_{in} = 0$. When we use the new *p* and *g* values, our carry signals become:

$c_0 = g_0 + p_0c_{in} = A_0 + A_0 \wedge 0 = A_0$
$c_1 = g_1 + p_1c_0 = 0 + A_1 \wedge A_0 = A_1 \wedge A_0$
$c_2 = g_2 + p_2c_1 = g_2 + p_2g_1 + p_2p_1g_0 = 0 + A_2 \wedge 0 + A_2 \wedge A_1 \wedge A_0 = 0 + A_2 \wedge A_1 \wedge A_0$
$c_3 = g_3 + p_3c_2 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 = 0 + A_3 \wedge 0 + A_3 \wedge A_2 \wedge 0 + A_3 \wedge A_2 \wedge A_1 \wedge A_0$
$= A_3 \wedge A_2 \wedge A_1 \wedge A_0$

Figure 7.24 shows our circuit with these changes.

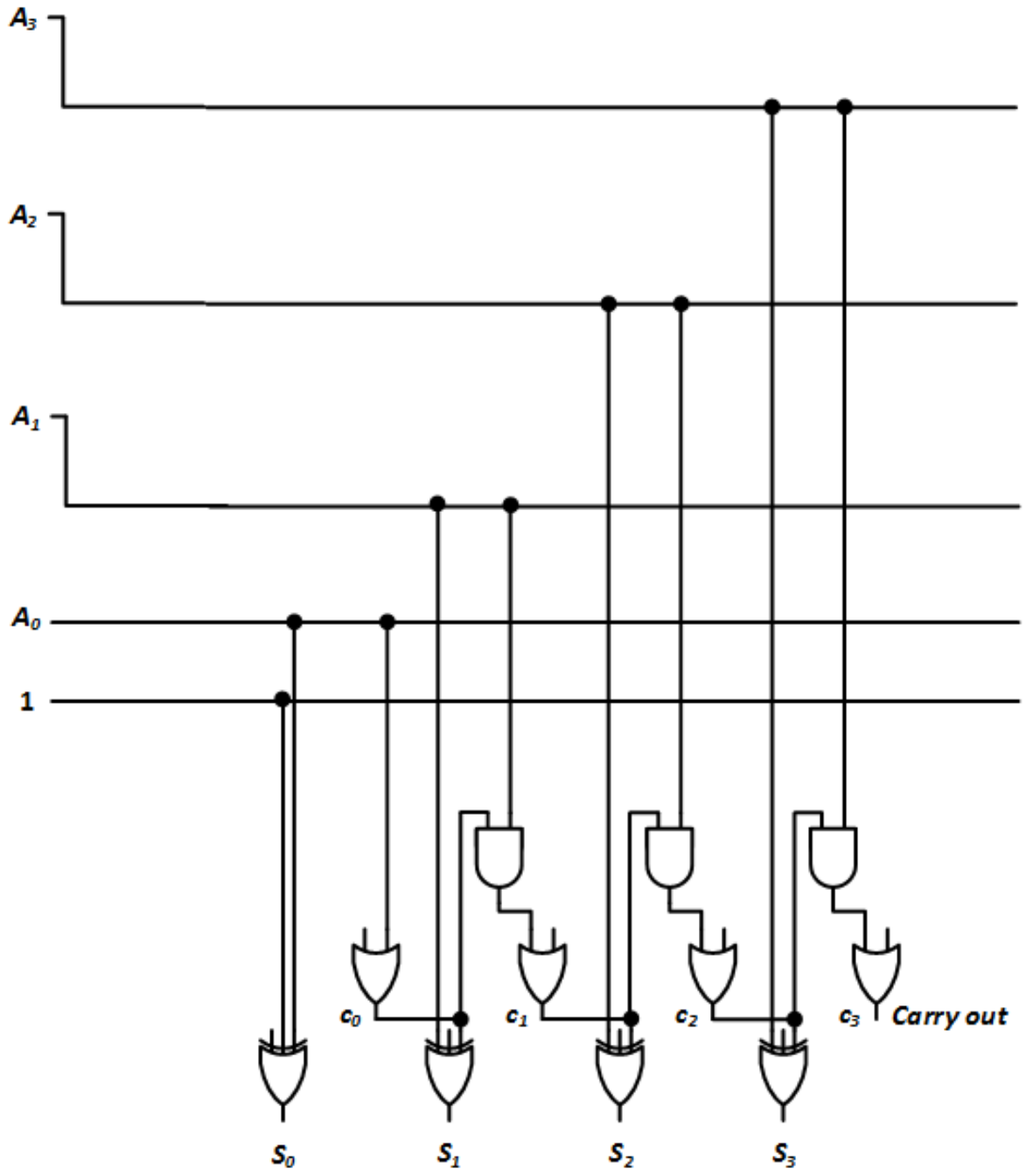


Figure 7.24: 4-bit carry-lookahead adder with simplified *c* signals.

Finally, we get to the XOR gates that generate the sum bits. These are exactly the bits we want to load into the flip-flops of the counter. Removing gate inputs that are equal to zero gives us the circuit shown in Figure 7.25. Comparing this circuit to the circuit in Figure 7.20, we can see that the *S* outputs of the modified carry-lookahead adder are the same as the inputs to the flip-flops.

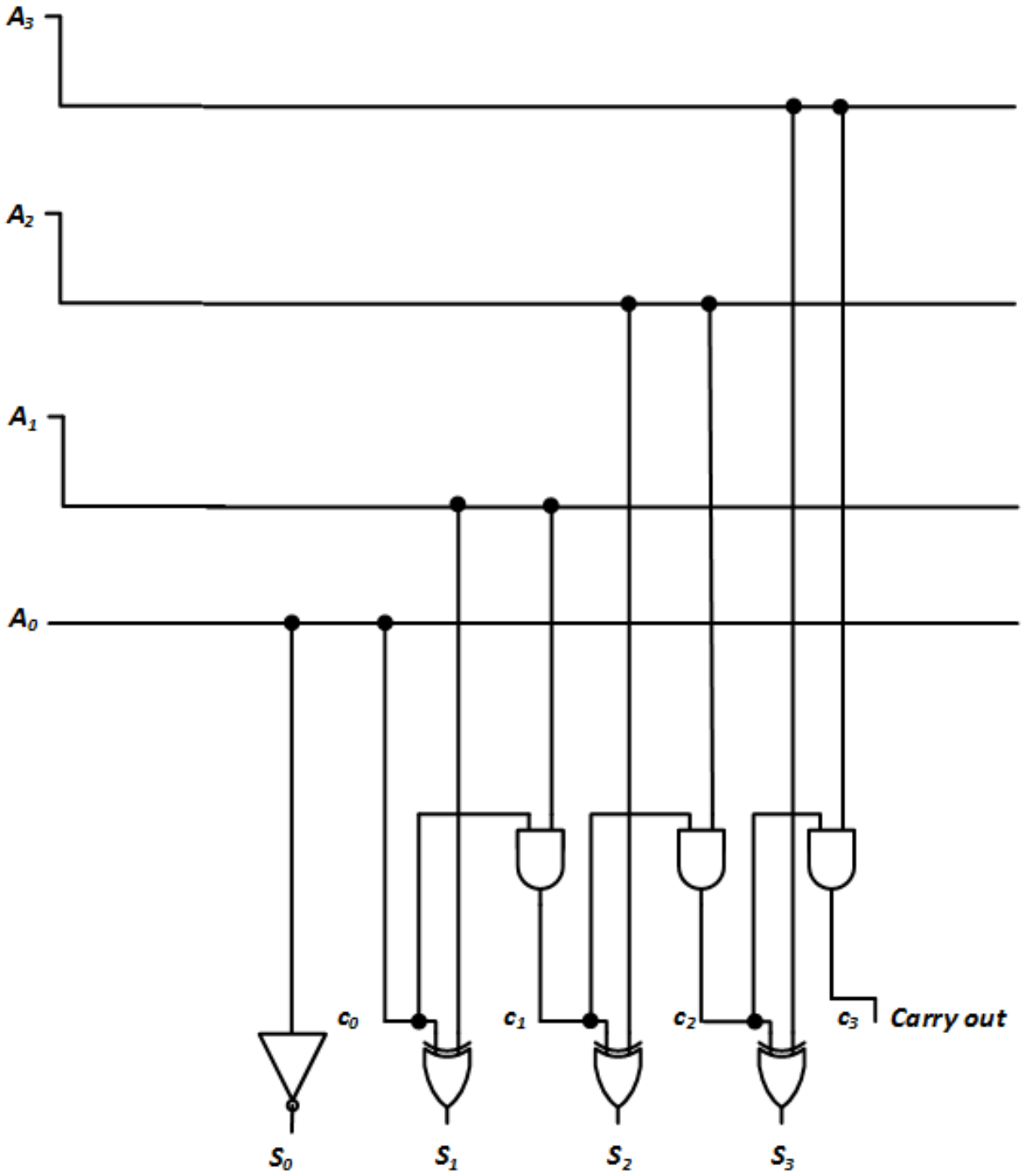


Figure 7.25: Carry-lookahead adder modified to add *A* + 1.

Inputting these sum bits to flip-flops, and adding the clock signal, gives us the final circuit for the synchronous 4-bit upcounter shown in Figure 7.26.

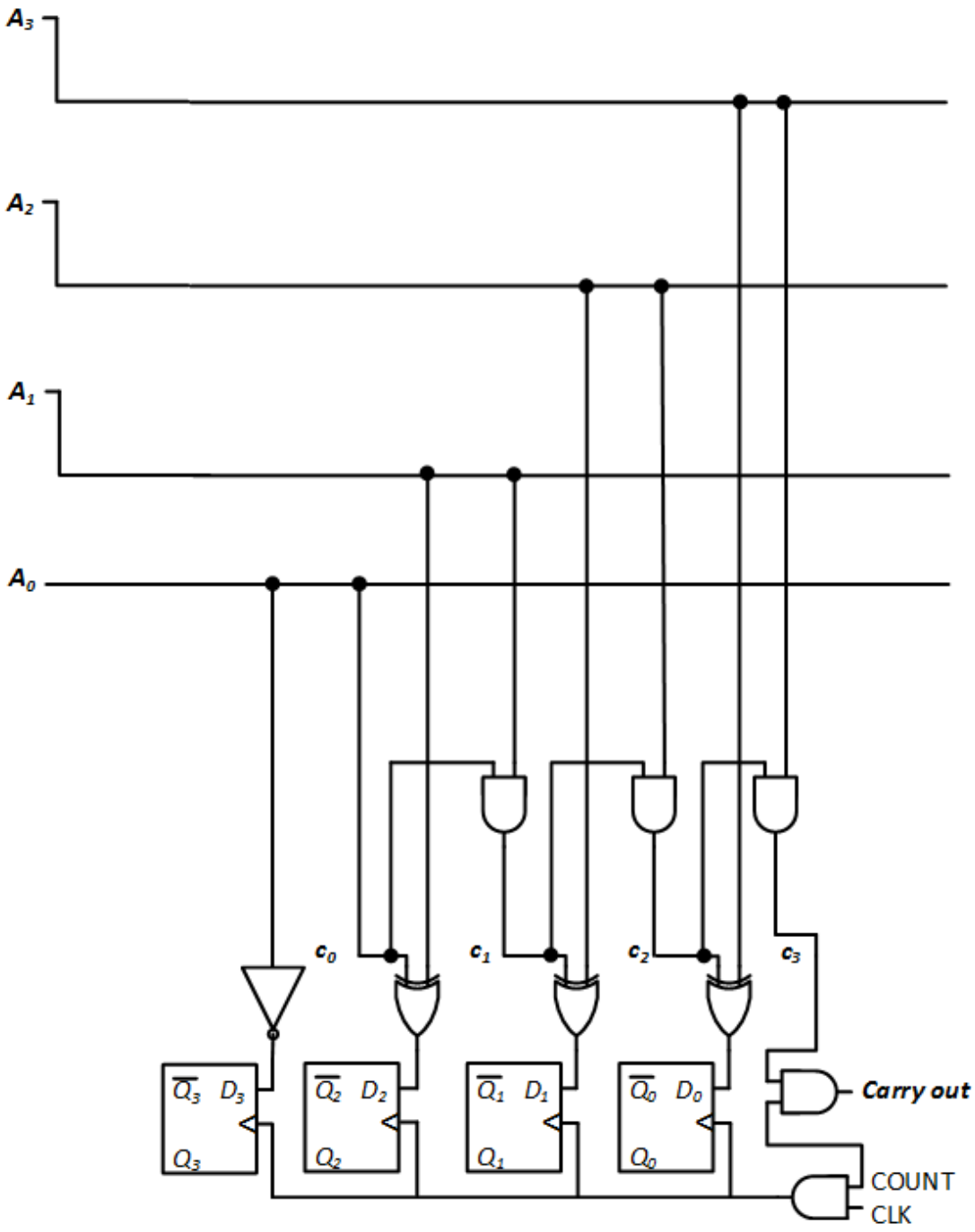


Figure 7.26: Final design for the 4-bit synchronous upcounter.

Figure 7.27 is an animation-only figure that shows this complete sequence.

WATCH ANIMATED FIGURE 7.27

Figure 7.27: Conversion of the 4-bit carry-lookahead adder to the 4-bit synchronous upcounter.

7.6 Cascading Counters

Just as there are practical limitations on the size of shift registers, there are also practical limitations on the size of counters. And just like shift registers, counters have been designed so they can be cascaded, combined to create counters with larger numbers of bits. To do this, the counters we use must have a carry out signal.

A generic 16-bit counter constructed using two 8-bit counters is shown in Figure 7.28. Just as our odometer needed a way to tell the tens digit to increment when the ones digit went back to 0 when the odometer went from 19 to 20, the carry out of the counter on the right (our ones digit) is used to generate the signal that increments the second counter (our tens digit). Note that CLK is only input to the first counter. Internally, it is used to generate the carry out signal for that counter, so it is already accounted for and does not need to be input separately into any other counter.

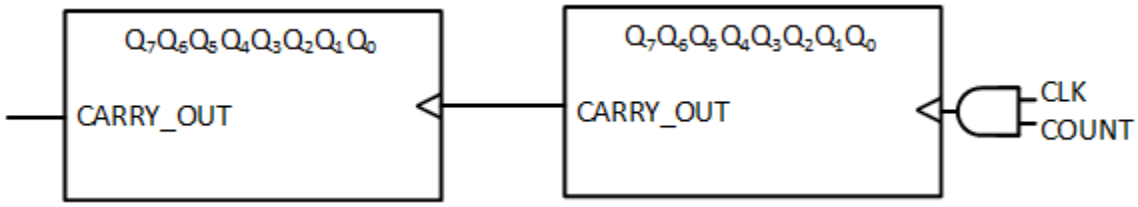


Figure 7.28: 16-bit counter constructed using two 8-bit counters.

7.7 Other Counters

So far we have seen several ways to design counters, but all of the designs presented focus on one type of counter, the binary counter. There are other types of counters that are useful for some applications. One is the **BCD counter**. Instead of counting from 0000 to 1111 (0 to 15), this 4-bit counter counts from 0000 to 1001 (0 to 9). It is particularly useful for applications involving digital displays. We will introduce this counter and its design in this section.

The idea behind a BCD counter can be extended to any **modulo-*n* counter**. For example, the ten-minutes digit on a digital clock could be implemented with a modulo 6 counter that counts from 000 to 101. We will introduce another design that can be used to create a modulo-*n* counter for any value of *n*.

7.7.1 BCD Counters

The **BCD counter**, also called a **decimal counter** or a **decade counter**, is used frequently in digital circuit design. In this subsection, we design this counter using D flip-flops. Our counter must go through the following sequence.

0000 → 0001 → 0010 → 0011 → 0100 → 0101 → 0110 → 0111 → 1000 → 1001 → 0000 →

As with the synchronous binary counter, we will AND together the COUNT and CLK inputs to produce the clock input to the flip-flops. Our remaining design task is to generate the D inputs to the flip-flops.

We then create the excitation table for the BCD counter, as shown in Figure 7.29. Before we continue, there's something I need to explain about this table. Notice the last six rows, with *Q* = 1010 to 1111. These are not valid BCD digits. So, why are they in the table? Think about this for a minute before reading on.

Q_3	Q_2	Q_1	Q_0	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0
1	0	1	0	0	0	0	0
1	0	1	1	0	0	0	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	0	0
1	1	1	0	0	0	0	0
1	1	1	1	0	0	0	0

Figure 7.29: Excitation table for the BCD counter.

Once your counter has a valid BCD value, it will always progress through the decimal counting sequence and will always have a valid BCD value. But what happens if the counter has one of the invalid values? A design flaw could cause this to happen, but the most likely reason for this to occur is that the flip-flops are set to invalid values when the circuit first powers up. To address this concern, we load in the value 0000 whenever the counter has one of the non-BCD values. Once that is done, the counter will function as we want it to.

With this table developed, we can create Karnaugh maps for the individual flip-flop inputs. I did this and derived the following functions.

$$D_0 = \overline{Q_3} \wedge \overline{Q_0} + \overline{Q_2} \wedge \overline{Q_1} \wedge \overline{Q_0}$$

$$D_1 = \overline{Q_3} \wedge (Q_1 \oplus Q_0)$$

$$D_2 = \overline{Q_3} \wedge (Q_2 \oplus (Q_1 \wedge Q_0))$$

$$D_2 = (\overline{Q_3} \wedge Q_2 \wedge Q_1 \wedge Q_0) + (Q3 \wedge \overline{Q_2} \wedge \overline{Q_1} \wedge \overline{Q_0})$$

The final circuit, with these functions implemented using combinatorial logic, is shown in Figure 7.30.

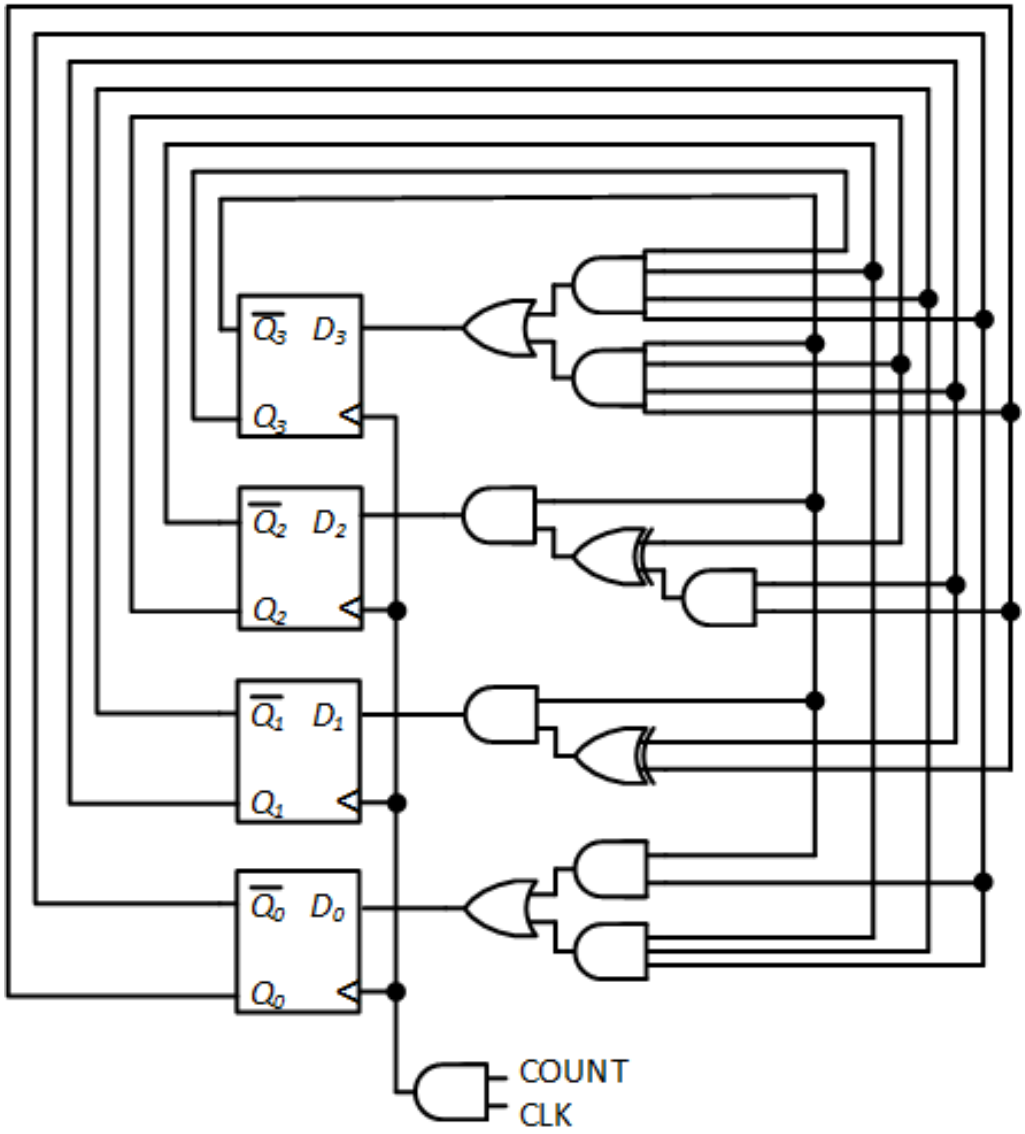


Figure 7.30: Synchronous BCD counter.

WATCH ANIMATED FIGURE 7.30

Now, if we're going to load 0000 into the counter when we reach the end of our sequence or an invalid value, why don't we just use D flip-flops with CLR inputs and clear the counter instead? We'll see what happens if we try to do this in the next subsection.

7.7.2 Modulo-*n* Counters

The BCD counter is a modulo-10 counter. That is, it counts from 0 to 9, 0000 to 1001, and then goes back to 0000, continuing this cycle ad infinitum. In practice, we can design and construct a modulo-*n* counter for any value of $n > 1$. (A modulo-1 counter would just output a 0 at all times, which would be fairly useless.)

Consider our 1s counter from Chapter 6. It counts the number of 1s coming in and outputs a 1 when it has counted three 1s. This is basically a modulo-3 counter; the output would be generated whenever the counter goes back to 0.

We can follow the procedure from the previous section to design this modulo-3 counter, or any modulo-*n* counter. However, there is a simpler way to do this if your circuit can tolerate a momentary glitch in its output.

The basic procedure is to start with a synchronous binary counter constructed using D flip-flops with preset and clear inputs, and with a maximum value that is greater than $n - 1$. If its maximum value is equal to $n - 1$, we just use the binary counter. For example, a modulo-8 counter can just use a 3-bit binary counter which has values ranging from 000 (0) to 111 (7).

We will increment the counter as before, ANDing together the COUNT and CLK signals. Unlike the previous design methodology, however, we do not change the *D* inputs. Instead, we generate a clear signal whenever we have an invalid value. For the BCD counter, we would increment the counter from 1001 (9) to 1010 (10) and then immediately clear the counter back to 0. The CLR input on the D flip-flop is asynchronous, that is, independent of the clock, so this occurs very quickly, on the order of nanoseconds, after the count goes to 1010. The design of a BCD counter using this methodology is shown in Figure 7.31.

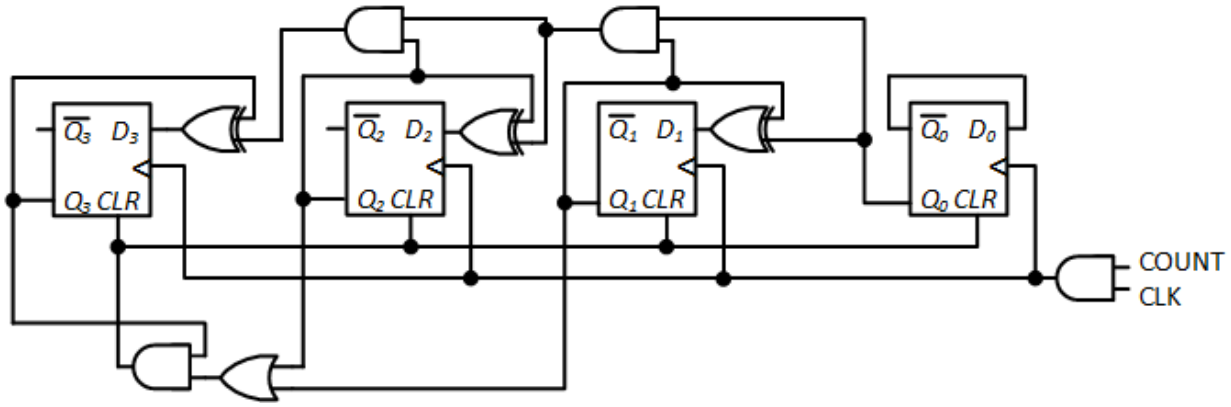


Figure 7.31: BCD counter with transient 1010 output.

For something like a digital clock, it is quite possible that nobody will ever see this invalid value on the display, but for other applications, this may not be the case.

Figure 7.32 shows the timing diagram for this counter as it transitions from 1001 to 1010 to 0000.

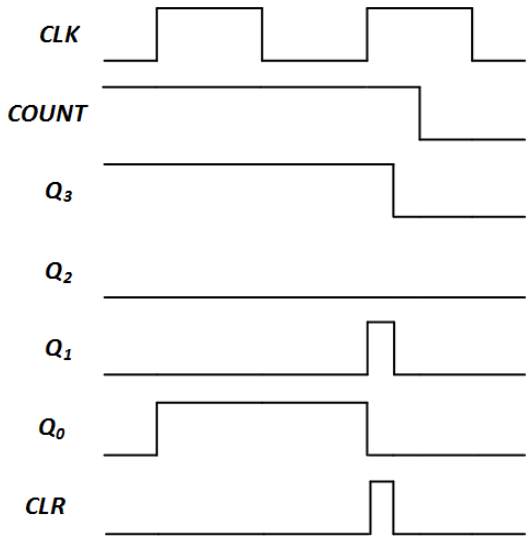


Figure 7.32: Timing diagram for the BCD counter with transient 1010 output.

WATCH ANIMATED FIGURE 7.32

In response to the question I raised at the end of section 7.7.1, the transient 1010 output is the reason I did not use the clear input in the previous design. You can always use the earlier design, but the latter design using the clear input can only be used in circuits that can tolerate the transient 1010 output.

7.8 Summary

Just as with combinatorial logic, digital designers have developed components for frequently used sequential logic. First and foremost among these components are registers. Essentially, these are a series of flip-flops configured in parallel to store multi-bit values. The flip-flops have common control signal inputs so that they act together on the bits of their stored value. Their clock signals are the same, as are other signals they may have, such as a clear input.

Shift registers store data, just like registers, but they can also shift their data. Some shift registers can shift their data in one direction, left or right; others can shift data in either direction. Some shift registers can also load data in parallel. Shift registers can be connected together to store and shift data values with larger numbers of bits.

Counters can store data, but also increment or decrement (or both) their stored values. Ripple counters propagate clock signals from one flip-flop to the next within the counter. This is efficient in terms of hardware, but it can result in slower performance as the clock signals propagate through the counter. Synchronous counters use additional logic gates to speed up the performance of the counters.

Upcounters can generate a carry out bit to indicate that its value has reached its maximum value and looped back to its slowest value, generally 0. Downcounters generate a borrow bit when they loop back from – to their highest value. These bits are particularly useful when we cascade counters to count values with larger numbers of bits.

Traditional counters count in binary; an *n*-bit counter sequences through values in the range from 0 to $2^n - 1$. BCD counters, in contrast, are 4-bit counters that only sequence through the values corresponding to decimal digits, 0 (0000) through 9 (1001). It is also possible to design modulo-*n* counters that can store any of *n* possible values, 0 to $n - 1$. Both the BCD and modulo-*n* counters have a wide variety of applications.

Bibliography

Hayes, J. P. (1993). *Introduction to digital logic design*. Addison-Wesley.

Mano, M. M., & Ciletti, M. (2017). *Digital design: with an introduction to the Verilog HDL, VHDL, and SystemVerilog* (6th ed.). Pearson.

Mano, M. M., Kime, C., & Martin, T. (2015). *Logic & computer design fundamentals* (5th ed.). Pearson.

Nelson, V., Nagle, H., Carroll, B., & Irwin, D. (1995). *Digital logic circuit analysis and design*. Pearson.

Roth, J. C. H., Kinney, L. L., & John, E.B. (2020). *Fundamentals of logic design enhanced edition* (7th ed.). Cengage Learning.

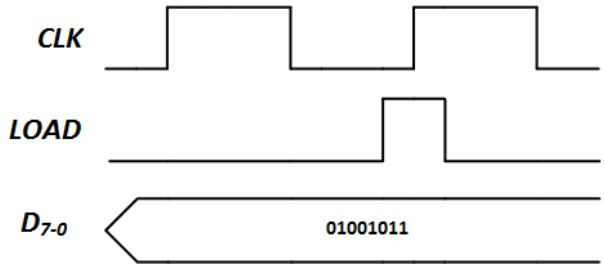
Sandige, R. S. (1990). *Modern digital design*. McGraw-Hill.

Texas Instruments, Inc. (1981). *The TTL Data Book for Design Engineers*. (2nd ed.). Texas Instruments.

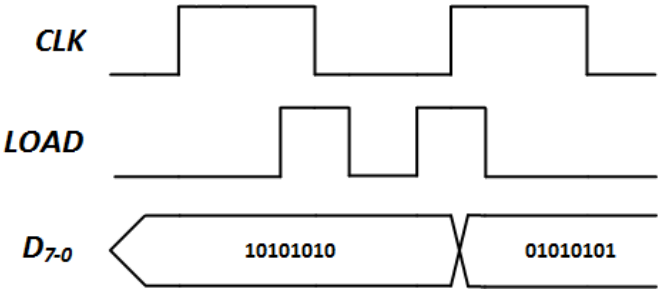
Wakerly, J. F. (2018). *Digital design: Principles and practices* (5th ed.). Pearson.

Exercises

1. Show the outputs of the register in Figure 7.1 for the inputs shown in the following timing diagram.



2. Show the outputs of the register in Figure 7.2 for the inputs shown in the following timing diagram.



3. Show the results of the linear shift left and linear shift right operations on the data value 11010101.
4. Show the values of the *D* inputs and *Q* outputs of the linear shift register after the shift left and shift right operations. The register initially has the value 01101011.
5. Repeat Problem 4 for the linear shift register constructed using J-K flip-flops.
6. Repeat Problem 4 for the bidirectional shift register with
    a. SHIFT_DIR = 0
    b. SHIFT_DIR = 1
7. Design a bidirectional shift register using J-K flip-flops.
8. Redesign the bidirectional shift register using combinatorial logic instead of multiplexers.
9. Modify the bidirectional shift register with parallel load to prioritize SHIFT over LOAD.
10. Minimize the function for S_0 in the bidirectional shift register with parallel load.

11. A circular shift operation works much like a linear shift, except the bit that is shifted out is circulated back and loaded into the bit that receives a 0 value in the linear shift. Modify the existing designs for the linear shift register to implement the circular shift left and circular shift right operations.

12. An arithmetic shift operation is similar to a linear shift, except it leaves the most significant bit unchanged. For the arithmetic shift left, all other bits are shifted left and the next to most significant bit is shifted out. For the arithmetic shift right, the most significant bit is unchanged and is also shifted into the next to most significant bit. Modify the existing designs for the linear shift register to implement the arithmetic shift left and arithmetic shift right operations.

13. Design a 32-bit shift register using 8-bit shift registers.

14. Design a 4-bit upcounter using T flip-flops.

15. Design a downcounter using J-K flip-flops.

16. Design a downcounter using T flip-flops.

17. Simplify the up-down counter of Figure 7.19 by using a single logic gate instead of a multiplexer to generate the clock signal.

18. Design a combined CARRY-BORROW signal for the bidirectional counter in Figure 7.19.

19. Design a 4-bit up-down counter using J-K flip-flops.

20. Design a 4-bit up-down counter using T flip-flops.

21. Modify the up-down counter to include LOAD and CLEAR signals.

22. Design a synchronous 4-bit upcounter using J-K flip-flops.

23. Design a 4-bit synchronous downcounter using:

    a. D flip-flops
    b. J-K flip-flops
    c. T flip-flops

24. Design a 16-bit counter using 4-bit counters.

25. Create Karnaugh maps for the *D* bits in the excitation table in Figure 7.28 and derive functions that do not use XOR gates.

26. Design the 1s counter from Section 6.1.1 with and without transient values

Chapter 8

Sequential Circuits

Chapter 8: Sequential Circuits

In the previous two chapters, we introduced the fundamental components of sequential logic: latches and flip-flops, as well as some more complex components that are designed using these components: registers and counters. In this chapter, we extend this knowledge to design synchronous sequential digital systems.

To design these systems, we will model each system as a finite state machine. In this approach, the system is always in one of its possible states. Depending on the values of the system inputs, the system will produce specific output values and transition to a different state (or remain in the same state). This may sound a bit confusing, but it will become clearer as we progress through this chapter. Using this methodology, we can design any digital system from the simplest controller to the most complex (non-quantum) computer system.

This chapter begins by introducing the basic model of finite state machines and the tools frequently used in their design: state diagrams and state tables. The state tables are not the same as the truth tables we've already seen, but they do have some similarities.

Next, we'll present the two main types of finite state machines, **Mealy machines** and **Moore machines**. Both are useful design models; they differ primarily in how they generate their output values.

With this background, we will then examine the state machine design process using flip-flops. We will go through design examples using both Mealy and Moore machines.

For some state machines, it is possible to use some of the more complex combinatorial and sequential components introduced earlier in this book to simplify our designs. We'll look at when this is helpful and which components to use.

Regardless of the design methodology used, sometimes it is helpful, or even necessary, to refine your design. It may be possible for your design to be in a state that it does not use, or there may be two or more states that are equivalent and can be merged together to simplify your final design. We close out this chapter by examining these and other scenarios.

8.1 Finite State Machines – Basics

When we started our study of sequential logic in Chapter 6, we noted that we can use the finite state machine methodology to model any sequential system. In this section, we introduce some of the fundamental ideas and tools used in finite state machine design. Let's start with perhaps the most fundamental idea of all.

8.1.1 What Is a State?

The Merriam-Webster dictionary has numerous definitions of the word *state*. The definition of interest for finite state machines is *a mode of condition of being*. This sounds really unclear, but it is a succinct and correct definition. This might be explained best by going through a couple of familiar examples.

First, let's look at a very simple example, the D flip-flop. At any given time, this flip-flop stores one of two values, 0 or 1. These two conditions of being, storing 0 and storing 1, are the two states of the D flip-flop. The system that is the D flip-flop has two states.

In this case, the *Q* output of the D flip-flop is 0 when it is in the storing 0 state, and 1 when it is in the storing 1 state. This is not necessarily the case for all finite state machines.

Remember the 1s counter we introduced in Chapter 6. When it has received an input value of 1 three times, it sets its output to 1. As noted in that chapter, this system has three states.

- Zero values of 1 have been input so far.
- One value of 1 has been input so far.
- Two values of 1 have been input so far.

For the first two states, our output is set to 0. Only when the system is in the third state, and we input a third 1, do we set the output to 1.

For some systems, the output values directly correspond to the state. Usually, however, this is not the case, and more than one state may generate the same output values.

In both examples, we performed the first step in the finite state machine design process. We listed all the possible states that can exist in our system. Next, we must determine every possible change that can occur in each state. We'll start with a system model.

8.1.2 System Model

In Chapter 6, we introduced a generic model of a sequential circuit. Figure 6.1 (b) shows this model, which is repeated and enhanced as Figure 8.1. This is also the model for a finite state machine. The state block is a set of flip-flops that stores a value corresponding to the state of the machine; this value is called the **present state**. We combine the value of the present state with the values of the inputs to determine what state we should next have for our system. This value, called the **next state**, is sent back to the state block.

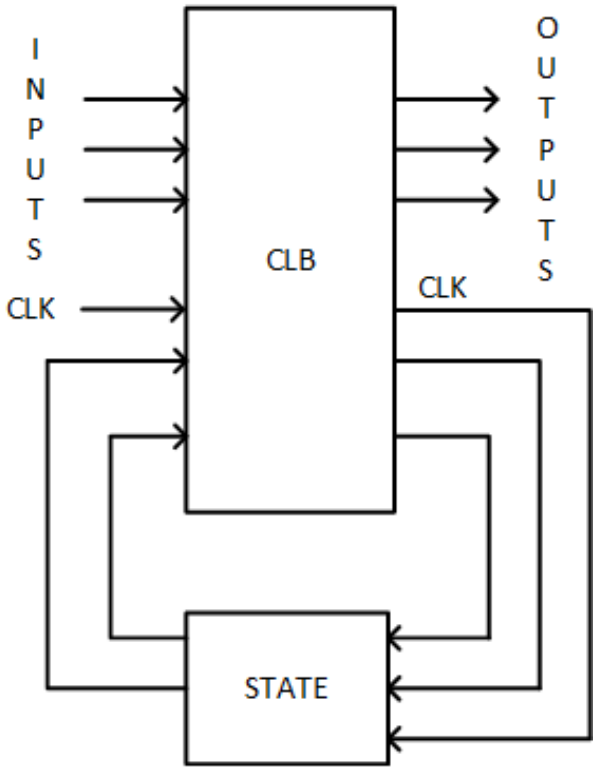


Figure 8.1: Generic finite state machine.

Notice that the clock (*CLK*) is also input to the block with the flip-flops that store the present state. Finite state machines are generally synchronous circuits, that is, they have a system clock that coordinates the flow of data. In our model, the logic within the combinatorial logic block (CLB) will require some amount of time to set its outputs, including the value of the next state. When selecting the frequency of the system clock, we must ensure that it is slow enough to let the CLB set all its values.

The final part of the finite state machine model is its outputs. Depending on the type of finite state machine we use, the outputs will be generated either as a function of the value of the present state and the inputs, or as a function of only the present state. We'll look at this in more detail in Section 8.2.

8.1.3 State Diagrams (Excluding Outputs)

A state diagram is a convenient mechanism used to graphically represent the functioning of the finite state machine. Each state is represented as a circle with the name of the state shown inside the circle. Directed arcs, basically arrows, show the transition from one state to another, or from one state back to itself. The conditions under which each transition occurs are shown on the arc. Outputs are also shown on the state diagram, but each type of finite state machine shows them in a different way. We will not show outputs in this subsection, but we'll come back to them when we introduce our models in Section 8.2.

Consider the D flip-flop. It has only two states, storing 0 and storing 1. We give each state a shorter label that fits more easily inside the state circle; I'll use S_0 for storing 0 and S_1 for storing 1. So far, our state diagram consists of just two states, no transitions yet, as shown in Figure 8.2 (a).

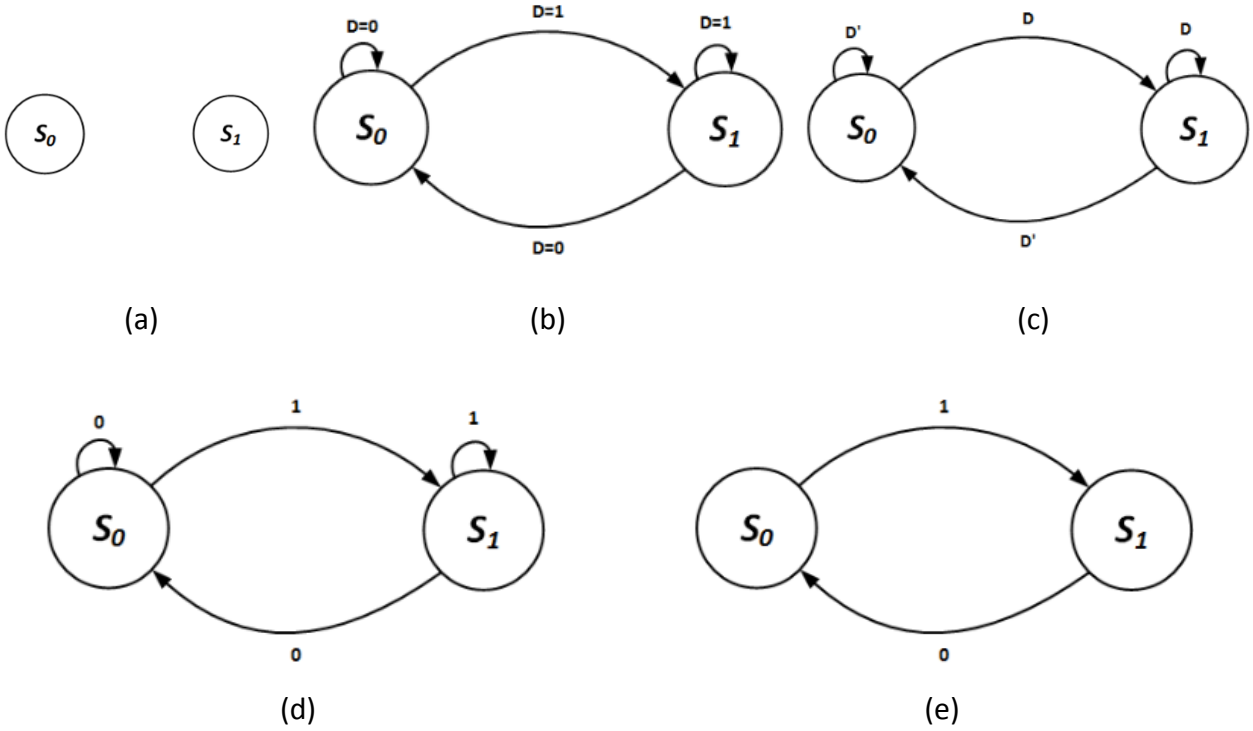


Figure 8.2: D flip-flop state diagram: (a) States; (b) States and all transitions; (c) and (d) Alternate representations of values; (e) Representation with self-arcs removed.

Now, let's see what happens in each individual state for all possible input values, starting with S_0. If $D = 1$, we want our flip-flop to go to S_1, storing 1. It should stay in the same state if $D = 0$. When our system is in state S_1, we want it to go to S_0 if $D = 0$ or to stay in S_1 if $D = 1$. Figure 8.2 (b) shows the state diagram with these state transitions included. The label above each arc indicates the conditions that must be met for the transition to occur.

Figure 8.2 (c) shows an alternative and more common way to represent the conditions. Here, we list the condition that must be true, equal to 1, for the transition to occur. $D = 1$ simply becomes D. $D = 0$ becomes D' because, if $D = 0$, then $D' = 1$. We can also dispense with the Ds and simply show the input values, as in Figure 8.2 (d).

State diagrams can also be represented without the self-arcs, the arcs that go from a state back to itself. By default, if a system is in a state, and the conditions are not met for any of the arcs coming out of the state, then the system just stays in its current state. Figure 8.2 (e) shows the state diagram with self-arcs removed.

All of this brings up an important point regarding the condition values. The conditions on all arcs coming out of a state must be **mutually exclusive**. That is, the conditions for only one arc (or possibly zero arcs if self-arcs are eliminated) can be true at any given time. This ensures that the system only tries to go to one state at a time.

Finally, there is one input signal we did not include in our state diagram, the system clock. The system clock is only used to synchronize the flow of data within the system, generally by causing edge-triggers that store the value of the present state to load in the value of the next state. This is implicit to the finite state machine and is not explicitly shown in the state diagram.

As another example, consider the 1s counter. This system has three states, which we'll call S_0, S_1, and S_2, which correspond to the states with 0, 1, and 2 1s already counted, respectively. In state S_0, we stay in S_0 if the input is 0 or go to S_1 if the input is 1. Similarly, if we are in state S_1, an input of 0 keeps us in S_1 and an input of 1 brings our system to S_2. In S_2, we remain in S_2 when the input is 0. However, if the input is 1, that becomes our third 1 and we reset our count back to 0 by going to S_0. The state diagram for this system is shown without self-arcs in Figure 8.3.

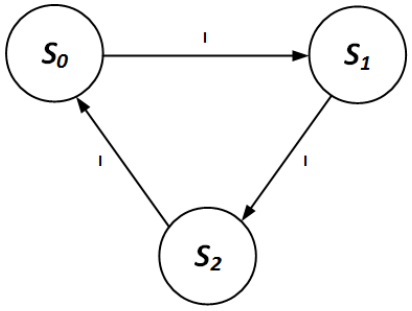


Figure 8.3: State diagram for the 1s counter.

8.1.4 State Tables (Also Excluding Outputs)

Another tool frequently used in finite state machine design is the **state table**. It is quite similar in format to the truth tables we've already seen throughout this book. The input side of the table includes not only the system inputs, but also the present state. The output portion of the table includes both the system outputs and the next state. To save space, we usually denote the present state and next state as *PS* and *NS*, respectively.

To illustrate this, the state table for the D flip-flop, excluding output values, is shown in Figure 8.4. Each row of the table corresponds to one state and one set of input values for the state machine. The animation for this figure shows how each row corresponds to an arc in the state diagram.

PS	D	NS	Q
S_0	0	S_0	
S_0	1	S_1	
S_1	0	S_0	
S_1	1	S_1	

Figure 8.4: D flip-flop state table, excluding output values.

WATCH ANIMATED FIGURE 8.4

Figure 8.5 shows the state table for the 1s counter, also excluding output values.

PS	I	NS	O
S_0	0	S_0	
S_0	1	S_1	
S_1	0	S_1	
S_1	1	S_2	
S_2	0	S_2	
S_2	1	S_0	

Figure 8.5: 1s counter state table, excluding output values.

8.2 Types of Finite State Machines

With this background, we're almost ready to design circuits to realize our finite state machines, except for one thing. As currently designed, our state machines don't generate any outputs. We excluded them so far because we had not yet introduced the models of finite state machines, each of which uses a different method to produce its outputs. In this section, we introduce these two methodologies, **Mealy machines** and **Moore machines**, and we complete the specifications, state diagrams, and state tables we developed for our two examples. In the next section, we will begin the actual design process.

Both Mealy and Moore machines are used frequently in sequential logic design. Neither is superior to the other. I chose to introduce them in alphabetical order, which is also the order in which they were developed.

8.2.1 Mealy Machines

The Mealy machine model for finite state machines was first published by George Mealy in 1955. It makes use of the state diagram and state table introduced in the previous section to indicate the transitions between states and the specification of the next state. It extends what we presented so far to include the output values generated by the state machine. This is where Mealy and Moore machines differ.

In the Mealy machine, outputs are specified as functions of both the present state and the input values. This being the case, they are shown on the arcs in state diagrams. The standard notation lists the inputs, then a slash, followed by the output values.

To illustrate this, let's return to the state diagram for the D flip-flop. We'll use the diagram in Figure 8.2 (d). We look at each arc in the state diagram and determine the output values it will produce, and then we add those values to the diagram. We start with state S_0 and its self-arc. When the flip-flop is in this state and D = 0, we want it to set output Q to 0. We denote this input/output combination as *0/0* on the state diagram. Similarly, when D = 1, the flip-flop should set Q = 1, or *1/1* in our notation. Repeating this process for the two arcs coming out of state S_1 gives us the final Mealy machine state diagram shown in Figure 8.6.

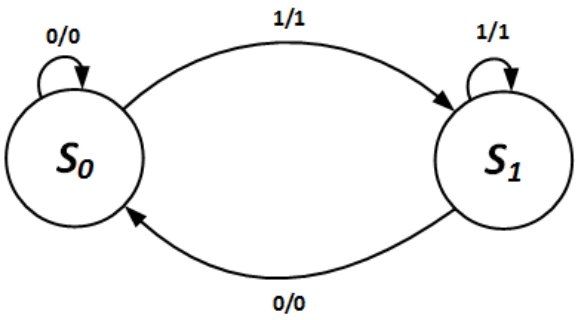


Figure 8.6: Mealy machine state diagram for the D flip-flop.

We modify the state table in the same manner. We determine the outputs generated by the present state and input values for each row in the state table. This is exactly how we generated the outputs for the state diagram. As shown in the state table in Figure 8.7, the state table outputs are identical to the values in the state diagram.

PS	D	NS	Q
S_0	0	S_0	0
S_0	1	S_1	1
S_1	0	S_0	0
S_1	1	S_1	1

Figure 8.7: Mealy machine state table for the D flip-flop.

WATCH ANIMATED FIGURE 8.7

As another example, let's look at the 1s counter. We want to output a 1 when we have counted three 1s, and 0 at all other times. If we add the self-arcs to the state diagram, and include the output values, we get the mostly complete state diagram shown in Figure 8.8.

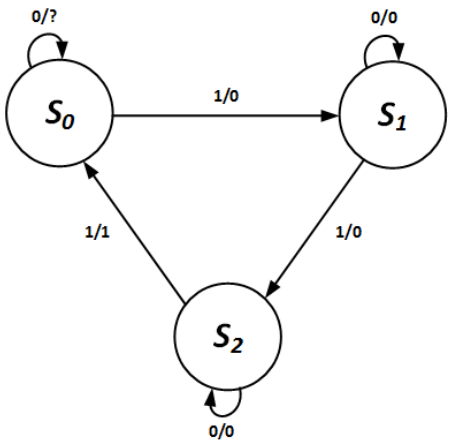


Figure 8.8: Mostly complete Mealy machine state diagram for the 1s counter.

This diagram would be fine if it weren't for the unspecified output on the self-arc at state S_0. The problem is not with the Mealy machine per se, but rather with the original specification. We said that we want the final circuit to set its output to 1 when it inputs a total of three 1s, but we did not say how long this signal should stay set to 1. Do we want it to continue to output a 1 until the next input value of 1, or do we want it to output a 1 for only one clock cycle, regardless of the input values that follow?

In the first case, the unspecified output should be 1. This will keep the output equal to 1 until another 1 is input; when this happens, the state machine transitions from S_0 to S_1 and sets its output to 0.

In the latter case, the unspecified output should be set to 0. The state machine would input its third 1, transition from S_2 to S_0, and set its output to 1. In the next clock cycle, it would either stay in S_0 (if the input value is 0) or go to S_1 (if the input is 1). In either case, it would set its output to 0, resulting in an output of 1 for only one clock cycle. The state diagrams for both cases are shown in Figure 8.9.

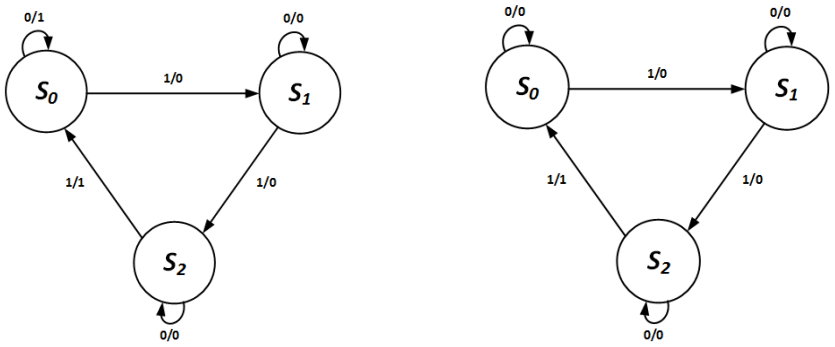


Figure 8.9: Mealy machine state diagram for the 1s counter: (a) Output is 1 until the next sequence begins; (b) Output is 1 for only one clock cycle.

The state tables are constructed just as we did for the previous example. The state tables for both outputs are shown in Figure 8.10.

PS	I	NS	O
S_0	0	S_0	1
S_0	1	S_1	0
S_1	0	S_1	0
S_1	1	S_2	0
S_2	0	S_2	0
S_2	1	S_0	1

(a)

PS	I	NS	O
S_0	0	S_0	0
S_0	1	S_1	0
S_1	0	S_1	0
S_1	1	S_2	0
S_2	0	S_2	0
S_2	1	S_0	1

(b)

Figure 8.10: Mealy machine state table for the 1s counter: (a) Output is 1 until the next sequence begins; (b) Output is 1 for only one clock cycle.

8.2.2 Moore Machines

One year after George Mealy introduced his model for finite state machines, Edward Moore published a different model that is also commonly used today. Unlike Mealy machines, Moore machines generate their outputs based solely on the present state. Moore machines do not use the values of the inputs to generate the output values. The input values do have an indirect role in generating the outputs since they may cause the machine to go to a specific next state, and that state has certain output values. However, when we ultimately develop sequential circuits to realize these state machines, the circuitry that generates the outputs has only the present state as its inputs.

Since a Moore machine always produces the same output values when it is in a given state, we use a different notation in the state diagram than we use for Mealy machines. The output values are included after the label for the state, separated from the state by a forward slash.

Consider once again the D flip-flop state machine. Whenever the machine is in state S_0, we want it to set its Q output to 0. When it is in state S_1, it should output Q = 1. Figure 8.11 shows the state diagram for the Moore machine version of the D flip-flop.

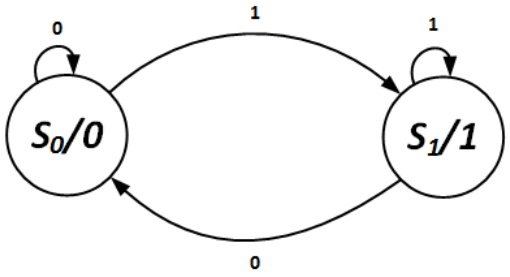


Figure 8.11: Moore machine state diagram for the D flip-flop.

The state table for the Moore machine has exactly the same format as that of the Mealy machine, though its values for the outputs may differ. In the Moore machine, the output value for each state is not changed by the input values. For the D flip-flop, our machine always outputs a 0 when it is in state S_0; in state S_1, it always outputs a 1. The state table for this machine is shown in Figure 8.12.

PS	D	NS	Q
S_0	0	S_0	0
S_0	1	S_1	0
S_1	0	S_0	1
S_1	1	S_1	1

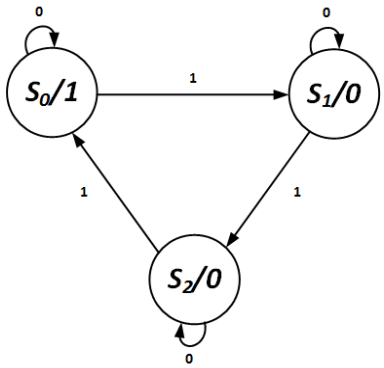
Figure 8.12: Moore machine state table for the D flip-flop.

WATCH ANIMATED FIGURE 8.12

If you compare the state tables for the Mealy and Moore machines for the D flip-flop, you will find that the only difference is the value of output *Q* in the two middle rows. These are the only two rows associated with a change in state, that is, the next state is not the same as the present state. *The Moore machine generates outputs based on the present state, whereas the Mealy machine generates outputs associated with the next state.* In the state table for the Moore machine, $Q = 0$ when the present state is S_0 and $Q = 1$ when it is S_1. The Mealy machine sets $Q = 0$ when the next state is S_0 and $Q = 1$ when it is S_1.

Now let's revisit the 1s counter. Remember that there are two interpretations of the output value: it remains at 1 until the next 1 is input, or it is set to 1 for only one clock cycle. We'll look at both cases in that order.

When the value stays at 1, we can simply set the output for state S_0 to 1 and set it to 0 for the other states. This state diagram is shown in Figure 8.13 (a); its corresponding state table is shown in Figure 8.13 (b).



(a)

PS	I	NS	O
S_0	0	S_0	1
S_0	1	S_1	1
S_1	0	S_1	0
S_1	1	S_2	0
S_2	0	S_2	0
S_2	1	S_0	0

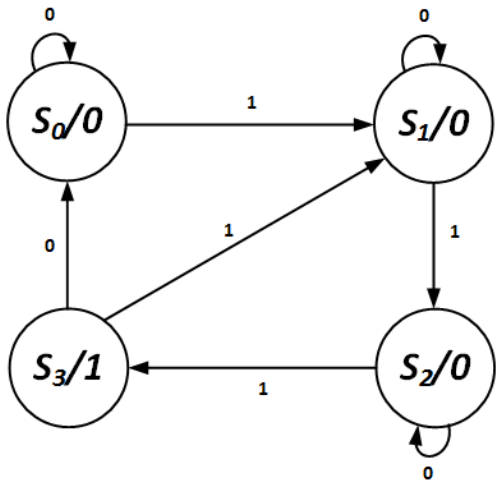
(b)

Figure 8.13: Moore machine for the 1s counter that outputs 1 until the next sequence begins: (a) State diagram; (b) State table.

WATCH ANIMATED FIGURE 8.13

The latter case, which sets the output to 1 for only one clock cycle, is not so simple. When we go to state S_0 from S_2, we want the output to be 1. However, if we then loop back from S_0 to itself (because we input a 0), we want the output to be 0. In Moore machines, we cannot have a state generate different output values at different times; it must always produce the same output values. Before reading on, think about how you would resolve this problem.

Here is one way to handle this issue. I created a new state, S_3. When the machine receives its third input of 1, it transitions from S_2 to S_3 instead of going to S_0. The output is set to 1 in this new state. If we are in state S_3 and receive a 0 input, we go to state S_0 since we have not input any 1s for our new group of three ones. If we instead input a 1, our group has one input of 1 and we go to state S_1. The rest of the state diagram is unchanged. This state diagram and its associated state table are shown in Figure 8.14.



(a)

PS	I	NS	O
S_0	0	S_0	0
S_0	1	S_1	0
S_1	0	S_1	0
S_1	1	S_2	0
S_2	0	S_2	0
S_2	1	S_3	0
S_3	0	S_0	1
S_3	1	S_1	1

(b)

Figure 8.14: Moore machine for the 1s counter that outputs 1 for only one clock cycle: (a) State diagram; (b) State table.

8.2.3 A Word about Infinite State Machines

All of this discussion about finite state machines raises the question, “Are there infinite state machines?” The short answer is yes, at least in theory. Consider, for example, a state machine that recognizes all sequences of inputs that are palindromes, that is, they are the same when read from first to last bit or from last to first. Since the string of inputs can be of any length, even infinite, we would need an infinite number of states in our machine to represent these input strings. The classic Turing machine and pushdown automata can also be considered as infinite state machines.

The problem with having an infinite number of states is that your system needs an infinite amount of storage to represent these states. You can’t really design nor build such a system. For this reason, we’ll note that infinite state machines can be specified mathematically, but since this book focuses on digital logic design, we won’t be discussing them any further.

8.3 Design Process

There are several ways to enumerate the steps in the design process for finite state machines. The process used in this book has seven steps. Other processes may split some of these steps into multiple steps or combine two or more steps into a single step. I can’t say that the process presented here is better or worse than other processes. This is very subjective and may be a matter of personal preference, as all of these design processes should lead to valid final designs.

With that said, here is our seven-step process. Throughout this process, we’ll use the 1s counter as a running example.

Step 1: Specify System Behavior

You can't design something until you know what it is supposed to do. This probably sounds obvious, but a lot of design errors start in this step. Occasionally a designer misrepresents the desired system behavior, but many errors in this step occur because the designer doesn't consider all possibilities. Thinking back to our BCD counter, it is straightforward to specify that the counter must sequence from 0 to 9 and then go back to 0. However, an incomplete specification might neglect to indicate that any invalid values (1010 to 1111) should go to 0 as well.

For the 1s counter, we specify that the system should input a sequence of values and output a 1 when it has received a total of three inputs equal to 1 and then start again.

Step 2: Determine States and Transitions, and Create the State Diagram

What are the possible conditions of being for our system? When we determine this, each becomes a state in our design. Then we look at each state individually, and each possible set of input values, to determine the outputs to generate and the next state to go to. Once this is done, we can create either the Mealy or Moore machine state diagram.

The Mealy machine implementation has three states:

S_0: Zero values of 1 have been input so far.
S_1: One value of 1 has been input so far.
S_2: Two values of 1 have been input so far.

For the Moore machine, we add a fourth state so we can set the output to 1 for only a single clock cycle.

S_3: Three values of 1 have been input so far.

Next, we look at each individual state and each input value to determine the next state and output value, and create the state diagram. We already did this in the previous section. The state diagram for the Mealy machine is shown in Figure 8.9 (a). The Moore machine state diagram is shown in Figure 8.14 (a).

Step 3: Create State Table with State Labels

You've already done most of the work for this in the previous step. The state diagram and state table are equivalent. The diagram illustrates system behavior graphically, whereas the state table enumerates the behaviors explicitly. The state tables for the Mealy and Moore machines are given in Figures 8.10 (b) and 8.14 (b), respectively.

As you can see, the first three steps are things we have already done in the previous section. The remaining steps, however, are new for our design. We present the steps here, and then we use the steps to complete the design of the Mealy and Moore machines in the next two subsections.

Step 4: Assign Binary Values to States

In our final design, we will be using digital logic. We need to store the value of the current state, and we will use flip-flops for this purpose. A flip-flop can only store the values 0 and 1; it can't store a state label such as S_0. So, we need to create a unique binary value for each state. The number of bits in the state value is based on the total number of states in the machine. Mathematically, a machine with n states needs at least $\lceil \lg n \rceil$ bits.

Step 5: Update the State Table with the Binary State Values

We take the state table created in Step 3 and substitute the binary state values developed in Step 4. For example, if state S_0 is represented as binary value 00, we replace S_0 with 00 in the state table wherever it is shown as either a present state or next state.

Step 6: Determine Functions for the Next State Bits and Outputs

At this point, our state table is essentially a truth table. The table inputs are the present state bits and the system inputs, and the table outputs are the next state bits and the system outputs. We can proceed as we have done previously to develop functions for these table outputs. Taking each table output individually, we create a function based solely on the table inputs. For small numbers of table inputs, a Karnaugh map will suffice. Larger numbers of inputs may require the Quine-McCluskey method to determine the final function.

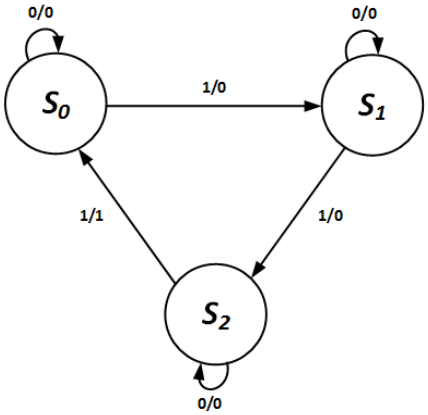
Step 7: Implement the Functions Using Combinatorial Logic

Finally, we create combinatorial logic circuits to implement the functions developed in the previous step. This logic must be combinatorial because the system must transition from the present state to the next state in a single clock cycle. This would not be possible if the functions included sequential logic.

Some parts of this design process are relatively straightforward; others, perhaps less so. In the next two subsections, we illustrate how these steps work as we complete the Mealy and Moore machine designs for the 1s counter that sets the output to 1 for a single clock cycle.

8.3.1 Design Example – Mealy Machine

First, we'll design the 1s counter as a Mealy machine. We developed the state diagram and state table for this machine in Section 8.2, so we've already completed the first three steps of the design process. The state diagram and state table are repeated in Figure 8.15.



(a)

PS	I	NS	O
S_0	0	S_0	0
S_0	1	S_1	0
S_1	0	S_1	0
S_1	1	S_2	0
S_2	0	S_2	0
S_2	1	S_0	1

(b)

Figure 8.15: Mealy machine (a) state diagram, and (b) state table for the 1s counter.

Now we move on to Step 4 of the design process and assign binary values to states. Since there are three states, and $\lceil \lg 3 \rceil = 2$, we need two bits to represent the state values. You can choose any assignment you wish, as long as your design generates the correct output value and next state values. For this example, I assign 00 to S_0, 01 to S_1, and 10 to S_2.

With these assignments made, we proceed to Step 5 and update the state table to include these values for both the present state and the next state. This table is shown in Figure 8.16.

PS	PS_1	PS_0	I	NS	NS_1	NS_0	O
S_0	0	0	0	S_0	0	0	0
S_0	0	0	1	S_1	0	1	0
S_1	0	1	0	S_1	0	1	0
S_1	0	1	1	S_2	1	0	0
S_2	1	0	0	S_2	1	0	0
S_2	1	0	1	S_0	0	0	1

Figure 8.16: Mealy machine state table for the 1s counter with binary state values.

Notice in this table that I have created separate names for each bit of the present state and the next state. As you'll see in the next step, we will create separate functions for the output and each individual bit of the next state value, and each function may use the individual bits of the present state, as well as the value of the input.

Now that everything in the state table is in binary, we can proceed as if it were a truth table. The three inputs, PS_1, PS_0, and I, will be used to create functions to generate NS_1, NS_0, and O. Figure 8.17 shows the Karnaugh maps. Notice the two don't-care values in each K-map. These correspond to present state value 11, which is not used in this design. We will proceed with this for now, but we will revisit this decision in the next section.

NS_1:

$I \backslash PS_1\,PS_0$	00	01	11	10
0	0	0	X	1
1	0	1	X	0

NS_0:

$I \backslash PS_1\,PS_0$	00	01	11	10
0	0	1	X	0
1	1	0	X	0

O:

$I \backslash PS_1\,PS_0$	00	01	11	10
0	0	0	X	0
1	0	0	X	1

Figure 8.17: Karnaugh maps for the next state and output functions for the Mealy machine.

Here are the functions we derived from these Karnaugh maps.

$$NS_1 = PS_0I + PS_1I'$$
$$NS_0 = PS_0I' + PS_1'PS_0'I$$
$$O = PS_1I$$

Finally, in Step 7, we create combinatorial logic circuits to realize these functions, as shown in Figure 8.18. The complete circuit is shown in Figure 8.19.

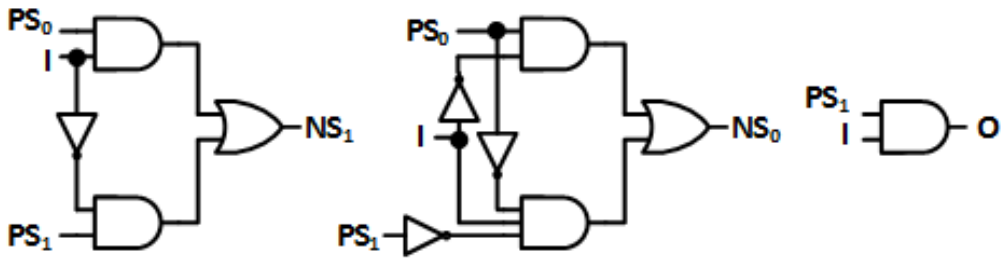


Figure 8.18 Combinatorial logic circuits to generate the next state and output for the Mealy machine.

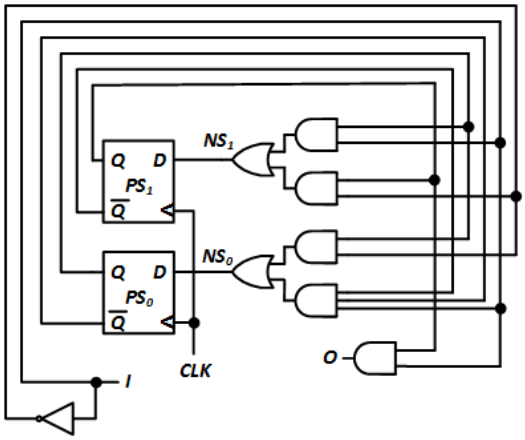
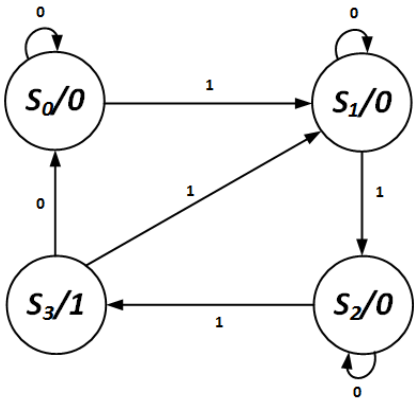


Figure 8.19: Complete circuit for the Mealy machine implementation of the 1s counter.

There is actually one more step in the design process that I did not list explicitly. After you have created the final design, you must verify that the design functions as desired for all possible states and input values. Simulation can be used for much of this work, but your design doesn't really work until you have actually built it and tested the circuit and have shown that it functions as desired.

8.3.2 Design Example – Moore Machine

Now let's create another design for this system based on the Moore machine we previously developed. The first three steps produced the state diagram and state table for this machine. Both are repeated in Figure 8.20.



(a)

PS	I	NS	O
S_0	0	S_0	0
S_0	1	S_1	0
S_1	0	S_1	0
S_1	1	S_2	0
S_2	0	S_2	0
S_2	1	S_3	0
S_3	0	S_0	1
S_3	1	S_1	1

(b)

Figure 8.20: Moore machine (a) state diagram, and (b) state table for the 1s counter.

The Moore machine has one more state than the Mealy machine for the 1s counter. Nevertheless, it still needs only two bits to represent the state since $\lceil \lg 4 \rceil = 2$. As with the Mealy machine, you can choose any assignments you wish; there are 4! = 24 possible state assignments. For this example, I assign 00 to S_0, 01 to S_1, 10 to S_2, and 11 to S_3.

Next, in Step 5, we update the state table to include these values. This is shown in Figure 8.21.

PS	PS_1	PS_0	I	NS	NS_1	NS_0	O
S_0	0	0	0	S_0	0	0	0
S_0	0	0	1	S_1	0	1	0
S_1	0	1	0	S_1	0	1	0
S_1	0	1	1	S_2	1	0	0
S_2	1	0	0	S_2	1	0	0
S_2	1	0	1	S_3	1	1	0
S_3	1	1	0	S_0	0	0	1
S_3	1	1	1	S_1	0	1	1

Figure 8.21: Moore machine state table for the 1s counter with binary state values.

From the table, we create Karnaugh maps for the next state bits and the output and determine functions for each one. These K-maps are shown in Figure 8.22. Notice that the map for output *O* is different from the other maps, and different for output *O* of the Mealy machine. Since this is a Moore machine, outputs are based solely on the present state; input values are not used to generate outputs.

NS_1:

$I\backslash PS_1\,PS_0$	00	01	11	10
0	0	0	0	1
1	0	1	0	1

NS_0:

$I\backslash PS_1\,PS_0$	00	01	11	10
0	0	1	0	0
1	1	0	1	1

O:

$PS_1\backslash PS_0$	00	01
0	0	0
1	0	1

Figure 8.22: Karnaugh maps for the next state and output functions for the Moore machine.

We derive the following functions from the Karnaugh maps.

$$NS_1 = PS_1'PS_0I + PS_1PS_0'$$
$$NS_0 = PS_1'PS_0I' + PS_1I + PS_0'I$$
$$O = PS_1PS_0$$

Next, we create the combinatorial circuit to generate these values. These circuits are shown in Figure 8.23, and the complete circuit is shown in Figure 8.24. Finally, we must build, test, and verify that our circuit functions as desired.

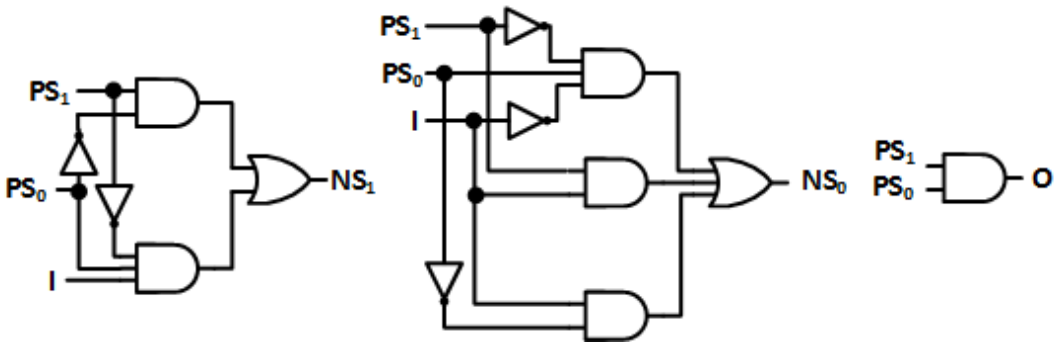


Figure 8.23 Combinatorial logic circuits to generate the next state and output for the Moore machine.

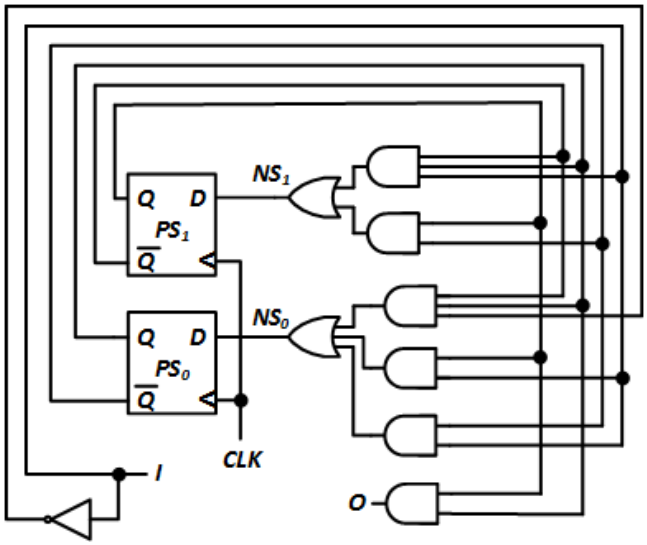


Figure 8.24: Complete circuit for the Moore machine implementation of the 1s counter.

8.3.3 A Brief Word about Flip-Flops

For the Mealy and Moore machines we just presented, we used D flip-flops to store the present state value. This is not strictly necessary; any type of edge-triggered flip-flop could be used instead. D flip-flops, however, do simplify the design process a bit.

When we generate the next state, we are producing the next value to be stored in the present state flip-flops. For D flip-flops, this is exactly the value we want to place on the D inputs of the flip-flops. For other flip-flops, such as J-K or T flip-flops, this is not necessarily the case. For implementations using these flip-flops, we would need to take the state table with the binary values, created in Step 5 of our design process, and create an excitation table for the flip-flops. In the remaining steps of the design process, we would create functions and circuits for the flip-flop inputs rather than the next state. The process of creating functions and circuits to generate the output values would remain the same.

8.3.4 Comparing our Two Designs

Now that we have completed the Mealy and Moore machine designs for the 1s counter circuit, how do the two designs compare? Is one better than the other?

Looking at the state diagrams, one difference is immediately apparent. The Mealy machine has fewer states than the Moore machine. This is not uncommon. It occurs whenever a Mealy machine has a state with two or more arcs going into the state, and those arcs have different output values. In a Moore machine, we need a separate state for each of these output values.

Having more states may result in a design with additional hardware, but this is not always the case. For the 1s counter design, both the Mealy and Moore machine need two flip-flops to store the present state value, since $\lceil \lg 3 \rceil = \lceil \lg 4 \rceil = 2$.

The combinatorial logic to generate the next state and output values is slightly less complex for the Mealy machine for this particular design. This is not always the case, and it is also impacted by other factors, such as the values assigned to each state. Output values for the Moore machine may require less combinatorial logic than those of the Mealy machine simply because they do not use system input values. Moore machine outputs are generated only from the present state. Once again, this is not always the case. For the 1s counter, both machines use a single 2-input AND gate to produce output *O*.

As far as the end user is concerned, both circuits perform the same function. They read in a sequence of bits and output a 1 for one clock cycle whenever it reads in three values set to 1.

8.4 Design Using More Complex Components

All the finite state machine designs we have seen so far in this chapter use edge-triggered flip-flops and basic logic gates in their final circuits. For some state machines, it is possible to use some of the more complex components introduced earlier in this book to simplify our designs. In this section, we'll look at three of these components: counters, decoders, and ROMs, and when and how they can be used in finite state machine design.

8.4.1 Design Using a Counter

Counters are useful when designing finite state machines that always go through a specific sequence of states. You have already seen one example of this: the Mealy machine implementation of the 1s counter. Looking at the state diagram in Figure 8.9 (b), and repeated in Figure 8.15 (a), we see that, for every state, the machine either stays in its current state or goes to one specific next state. Furthermore, this one specific next state is different for every state.

We have already spent a fair amount of time on the 1s counter, so we will illustrate how counters can be used in finite state machines using a different example. In this subsection, we design a 3-bit Gray code sequence generator using a counter; we will design this system as a Moore machine.

Step 1: Specify System Behavior

Our system will produce a 3-bit Gray code. It has a clock input, *CLK*, and a data input *I*. It has a 3-bit output *O*, consisting of bits O_2, O_1, and O_0, that give the current value in the sequence. On the rising edge of the clock, if *I* = 1, *O* changes to the next value in the Gray code sequence; otherwise it keeps its current output value. As we described in Chapter 1, the 3-bit Gray code sequence is

$$000 \rightarrow 001 \rightarrow 011 \rightarrow 010 \rightarrow 110 \rightarrow 111 \rightarrow 101 \rightarrow 100 \rightarrow 000 \rightarrow \ldots$$

Step 2: Determine States and Transitions, and Create the State Diagram

We are designing a Moore machine, and our system has eight possible output values. Therefore, our system must have at least eight states. From each state, we go to one of two states: the state that outputs the next value in the Gray code sequence, or back to itself so that it outputs the same value.

For example, let's say that state S_0 sets the output to 000. On the rising edge of the clock, if *I* = 1, we want the sequence generator to transition to a state that outputs 001; we'll call this state S_1. Otherwise we want to remain in state S_0 and continue to output 000. In S_1, we want to stay in S_1 and output 001 if *I* = 0, or go to S_2 and output 011 if *I* = 1 on the rising edge of the clock. Following this process for the entire sequence gives us the state diagram shown in Figure 8.25; self-loops are not shown.

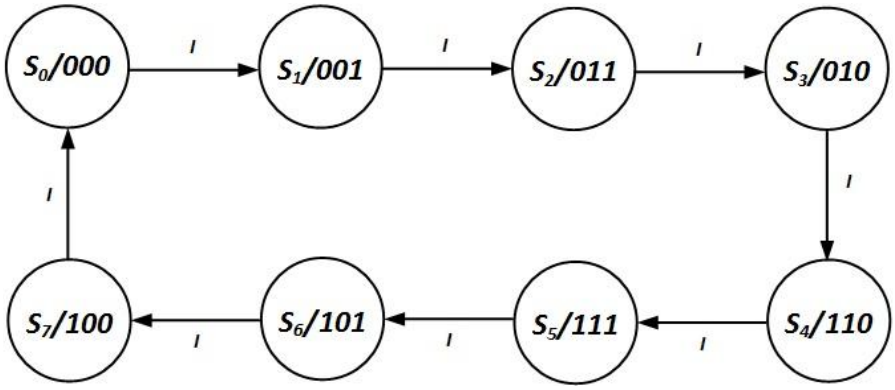


Figure 8.25: Moore machine state diagram for the 3-bit Gray code sequence generator.

Step 3: Create State Table with State Labels

Now we can convert our state diagram into a state table. Although the self-loops are not shown explicitly in the state diagram, they do exist and we must include them in the state table. Following the same procedure we have used throughout this chapter, we develop the state table shown in Figure 8.26.

PS	*I*	NS	O_2	O_1	O_0
S_0	0	S_0	0	0	0
S_0	1	S_1	0	0	0
S_1	0	S_1	0	0	1
S_1	1	S_2	0	0	1
S_2	0	S_2	0	1	1
S_2	1	S_3	0	1	1
S_3	0	S_3	0	1	0
S_3	1	S_4	0	1	0
S_4	0	S_4	1	1	0
S_4	1	S_5	1	1	0
S_5	0	S_5	1	1	1
S_5	1	S_6	1	1	1
S_6	0	S_6	1	0	1
S_6	1	S_7	1	0	1
S_7	0	S_7	1	0	0
S_7	1	S_0	1	0	0

Figure 8.26: State table for the 3-bit Gray code sequence generator.

Step 4: Assign Binary Values to States

When using a counter in a finite state machine design, your work will be much easier if you assign sequential values to the states corresponding to the order in which they are accessed. For example, if we assign 000 to S_0, and we next go to S_1, we should assign 001 to S_1. This allows us to simply increment the counter to transition from one state to the next, and counters already have the ability to increment their values built into their designs. Doing this gives us the following state value assignments.

$S_0 = 000$ $\quad\quad$ $S_4 = 100$
$S_1 = 001$ $\quad\quad$ $S_5 = 101$
$S_2 = 010$ $\quad\quad$ $S_6 = 110$
$S_3 = 011$ $\quad\quad$ $S_7 = 111$

Note that the state values are not the same as the output values, but that's OK. Later in the design process, we will use these values for the present state, along with the value of *I*, to generate the output values O_2, O_1, and O_0. As we'll soon see, we will use (or not use) the counter's increment function to generate the next state.

Step 5: Update the State Table with the Binary State Values

This is pretty straightforward. We take the binary values we just developed and substitute them for the state labels in the state table. This gives us the updated state table shown in Figure 8.27.

PS	PS_2	PS_1	PS_0	I	NS	NS_2	NS_1	NS_0	O_2	O_1	O_0
S_0	0	0	0	0	S_0	0	0	0	0	0	0
S_0	0	0	0	1	S_1	0	0	1	0	0	0
S_1	0	0	1	0	S_1	0	0	1	0	0	1
S_1	0	0	1	1	S_2	0	1	0	0	0	1
S_2	0	1	0	0	S_2	0	1	0	0	1	1
S_2	0	1	0	1	S_3	0	1	1	0	1	1
S_3	0	1	1	0	S_3	0	1	1	0	1	0
S_3	0	1	1	1	S_4	1	0	0	0	1	0
S_4	1	0	0	0	S_4	1	0	0	1	1	0
S_4	1	0	0	1	S_5	1	0	1	1	1	0
S_5	1	0	1	0	S_5	1	0	1	1	1	1
S_5	1	0	1	1	S_6	1	1	0	1	1	1
S_6	1	1	0	0	S_6	1	1	0	1	0	1
S_6	1	1	0	1	S_7	1	1	1	1	0	1
S_7	1	1	1	0	S_7	1	1	1	1	0	0
S_7	1	1	1	1	S_0	0	0	0	1	0	0

Figure 8.27: State table updated with binary state values.

Step 6: Determine Functions for the Next State Bits and Outputs

This is the step where we save ourselves some work by using a counter. If our next state is always either the same as our current state or our current state + 1 (or 000 when our current state is 111), we can use the counter's clock signal to generate our next state. To do this, we'll do something we did in Chapter 7. We AND together the clock signal (*CLK*) and our input signal (*I*) to generate the clock input to the counter, as shown in Figure 8.28. When *I* = 0, the output of the AND gate is always 0, regardless of the value of *CLK*. Since this does not produce a rising edge, the counter does not increment; it keeps its current value. For our system, it stays in its current state. If *I* = 1, however, the output of the AND gate is the same as *CLK*. A rising edge on *CLK* produces a rising edge on the clock input of the counter, causing it to increment its value. For our system, this goes to the next state and outputs the next value in the Gray code sequence.

Figure 8.28: Generating the next state using a counter.

Even though we simplified how we generate the next state, we still must develop functions for our outputs. Since we are using a Moore machine, we only need to consider the present state value. To make it a little easier to visualize this, I've taken our state table and removed everything except the present state and output values; this is shown in Figure 8.29. From this table, you can create Karnaugh maps or simply create the functions by just inspecting the table itself. The final functions for O_2, O_1, and O_0 are also shown in the figure. I chose to use the XOR function, but any valid Boolean function will suffice.

PS_2	PS_1	PS_0	O_2	O_1	O_0
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	1	1	1
1	1	0	1	0	1
1	1	1	1	0	0

$$O_2 = PS_2$$
$$O_1 = PS_2 \oplus PS_1$$
$$O_0 = PS_1 \oplus PS_0$$

Figure 8.29: Reduced state table and functions for system outputs.

WATCH ANIMATED FIGURE 8.29

Step 7: Implement the Functions Using Combinatorial Logic

For this system, we need a total of two 2-input XOR gates to generate our outputs. The complete, final design is shown in Figure 8.30.

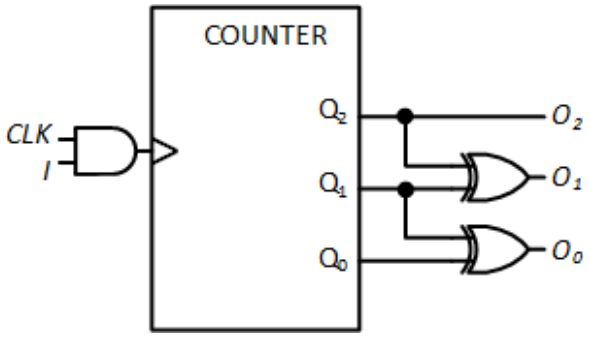


Figure 8.30: Final circuit for the 3-bit Gray code sequence generator using a counter.

WATCH ANIMATED FIGURE 8.30

8.4.2 Design Using a Decoder

As we have seen in earlier chapters, decoders can be used in numerous combinatorial logic circuits. They also can be used in sequential circuits. When used in finite state machines, there is one role they play particularly well: producing signals corresponding to individual states.

To show how this works, we'll redesign the 3-bit Gray code sequence generator to use a decoder. Fortunately for us, we can reuse much of the work we just completed for this state machine in the previous subsection. The state behavior (Step 1), state diagram (Step 2), state table with state labels (Step 3), state assignments (Step 4), and state table with binary values (Step 5) are exactly the same as the previous design using a counter. We will also use a counter in this design, and the logic needed to create the clock signal for the counter is the same as that shown in Figure 8.28. The only thing we will change is the logic to generate the output values.

To do this, we take the output of the counter and send it to the inputs of a 3 to 8 decoder, as shown in Figure 8.31. When the counter value is 000, output O_0 of the decoder is asserted. From our state assignments (Step 4), we know that the value 000 is assigned to state S_0; therefore, decoder output O_0 corresponds to state S_0. The other decoder outputs correspond to the remaining states as shown in the figure.

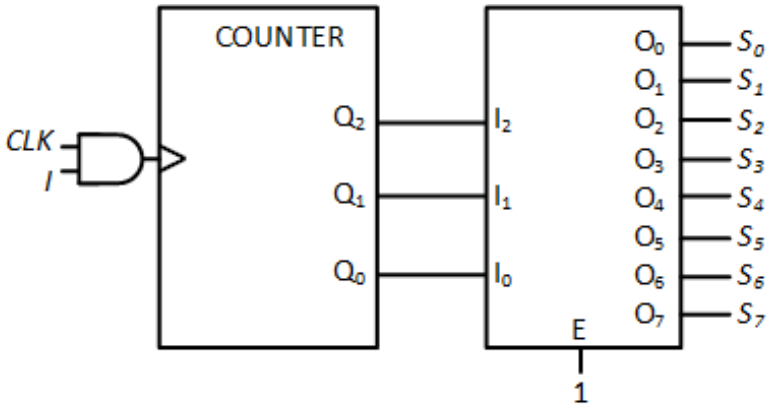


Figure 8.31: Generating state signals using a decoder.

Since we are developing a Moore machine, and outputs are based solely on the present state, we can combine these state signals to generate the system's outputs. We take the state table and remove everything except the present state label and binary output values as shown in Figure 8.32. For each output, we take all the states that set the output to 1 and logically OR them together. Output $O_2 = 1$ when the state machine is in state S_4, S_5, S_6, and S_7, so $O_2 = S_4 + S_5 + S_6 + S_7$. We do the same for O_1 and O_0, giving us the functions shown in the figure.

PS	O_2	O_1	O_0
S_0	0	0	0
S_1	0	0	1
S_2	0	1	1
S_3	0	1	0
S_4	1	1	0
S_5	1	1	1
S_6	1	0	1
S_7	1	0	0

$$O_2 = S_4 + S_5 + S_6 + S_7$$
$$O_1 = S_2 + S_3 + S_4 + S_5$$
$$O_0 = S_1 + S_2 + S_5 + S_6$$

Figure 8.32: Reduced state table and functions for system outputs.

Looking back to the earlier chapters, you may realize that the decoder generates minterms based on the state value. Each output function is expressed as a sum of products.

Finally, we create combinatorial logic circuits to realize these functions. Each can be created using a single 4-input OR gate. The final design is shown in Figure 8.33.

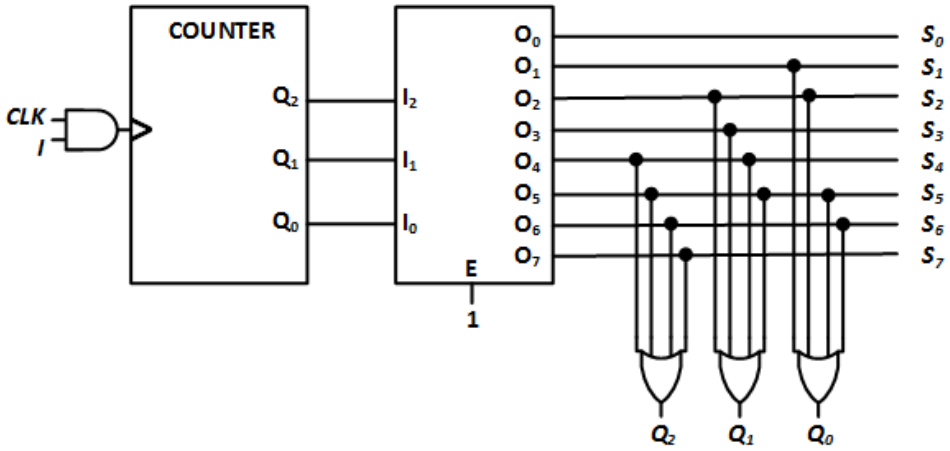


Figure 8.33: Final circuit for the 3-bit Gray code sequence generator using a counter and a decoder.

WATCH ANIMATED FIGURE 8.33

For this machine, using a decoder makes the circuit more complex than the circuit that uses only two XOR gates to generate its outputs. For other circuits, this strategy can reduce the hardware complexity. In either case, it is a viable design methodology that can be employed when appropriate.

8.4.3 Design Using a Lookup ROM

In Chapter 5, we introduced the concept of a lookup ROM. Instead of using combinatorial logic to realize a logical or arithmetic function, we store values in a non-volatile memory chip. The function inputs are connected to the address inputs of the ROM, and the data output pins supply the values of the functions for those input values. We can do something similar for sequential circuits.

Looking at our state tables, we can see that there are two types of information we need in order to use a lookup ROM in a finite state machine. As with the earlier lookup ROM circuits, we need to know the values of the inputs. But for sequential circuits, we also need to know which state we are presently in. With all this information, we can determine our next state and the value of all system outputs.

Figure 8.34 shows a generic configuration for a finite state machine using a lookup ROM. The bits of the present state and the inputs are connected to the address inputs of the lookup ROM. At each location in the ROM, data is stored. Some bits specify the next state of the machine for the present state and input values, and other data bits give the values of the system outputs. The next state bits are loaded into a register every clock cycle. They become the new present state and are fed back to the address bits of the ROM.

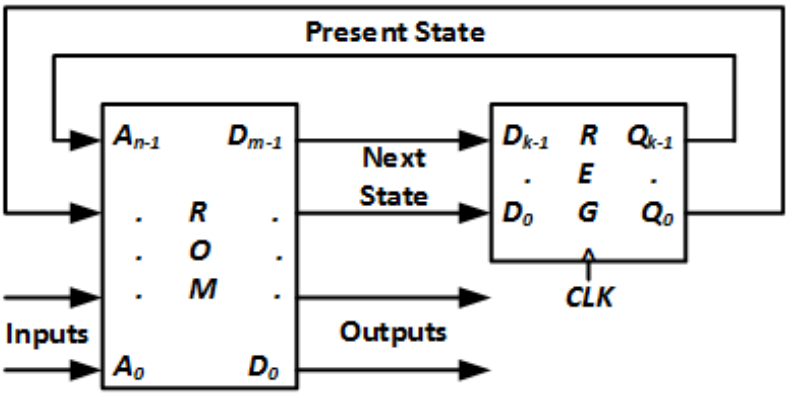


Figure 8.34: Generic state machine constructed using a lookup ROM.

WATCH ANIMATED FIGURE 8.34

To illustrate how this works, let's redesign the 3-bit Gray code sequence generator one final time. This system has the following:

- 3-bit present state (PS_2, PS_1, PS_0)
- 1-bit input (I)
- 3-bit next state (NS_2, NS_1, NS_0)
- 3-bit Gray code output (O_2, O_1, O_0)

We will connect PS_2, PS_1, PS_0, and I to the address bits of the ROM, and the ROM outputs will supply the values of NS_2, NS_1, NS_0, O_2, O_1, and O_0. The three next state bits are connected to the data inputs of a register. The register stores the present state value and its outputs are fed back to the address inputs of the ROM. Our system needs a ROM with four address pins and six data pins, and has a total size of $2^4 = 16$ locations; its size is 16 × 6. Figure 8.35 shows the circuit for this state machine.

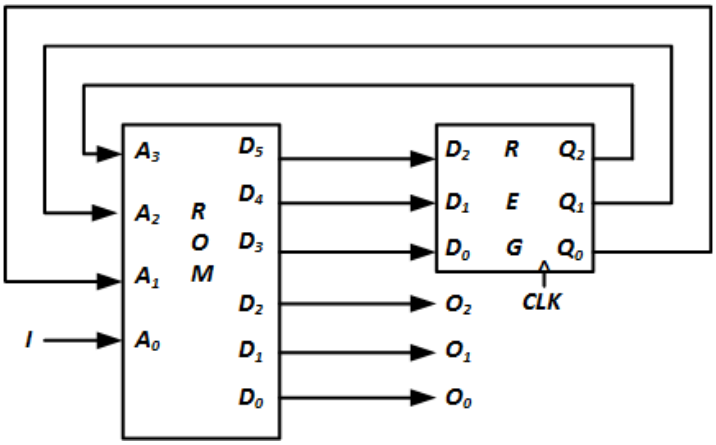


Figure 8.35: Circuit for the 3-bit Gray code sequence generator using a lookup ROM.

To make this circuit function properly, we must determine the data values to be stored in all of the locations in the ROM. We can do this by starting with the state table with binary values, repeated once again in Figure 8.36 (a). Then we see which address pin is connected to each present state bit and input, and we replace these labels in the table with the labels for the corresponding address bits. We do the same for the next state and output bits, this time replacing their labels with those of the corresponding data bits. This gives us the data table shown in Figure 8.36 (b). The left side of the table lists the addresses in the ROM and the right side shows the data stored at each location in the ROM.

PS_2	PS_1	PS_0	I	NS_2	NS_1	NS_0	O_2	O_1	O_0
0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	1	0	0	0
0	0	1	0	0	0	1	0	0	1
0	0	1	1	0	1	0	0	0	1
0	1	0	0	0	1	0	0	1	1
0	1	0	1	0	1	1	0	1	1
0	1	1	0	0	1	1	0	1	0
0	1	1	1	1	0	0	0	1	0
1	0	0	0	1	0	0	1	1	0
1	0	0	1	1	0	1	1	1	0
1	0	1	0	1	0	1	1	1	1
1	0	1	1	1	1	0	1	1	1
1	1	0	0	1	1	0	1	0	1
1	1	0	1	1	1	1	1	0	1
1	1	1	0	1	1	1	1	0	0
1	1	1	1	0	0	0	1	0	0

(a)

A_3	A_2	A_1	A_0	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	1	0	0	0
0	0	1	0	0	0	1	0	0	1
0	0	1	1	0	1	0	0	0	1
0	1	0	0	0	1	0	0	1	1
0	1	0	1	0	1	1	0	1	1
0	1	1	0	0	1	1	0	1	0
0	1	1	1	1	0	0	0	1	0
1	0	0	0	1	0	0	1	1	0
1	0	0	1	1	0	1	1	1	0
1	0	1	0	1	0	1	1	1	1
1	0	1	1	1	1	0	1	1	1
1	1	0	0	1	1	0	1	0	1
1	1	0	1	1	1	1	1	0	1
1	1	1	0	1	1	1	1	0	0
1	1	1	1	0	0	0	1	0	0

(b)

Figure 8.36: (a) State table for the 3-bit Gray code sequence generator; (b) Data table showing the contents of the lookup ROM.

Figure 8.37 shows an animation of the final circuit. To illustrate how this works, consider the case when the circuit is in state S_3 (011) and input I = 1. The 011 state value is stored in the register and fed back to ROM address bits A_3, A_2, and A_1. I = 1 is connected to address bit A_0. Thus, we are accessing address 0111 in the ROM. Now, looking at the data table for the ROM, we see that this location stores the data 100010. The three most significant bits, 100, are output on data pins D_5, D_4, and D_3. These are the next state bits. When we are in state S_3 and I = 1, we want to go to state S_4, which is represented by this value. The three least significant bits, 010, are output on pins D_2, D_1, and D_0. These are the Gray code output bits, and they are the correct output values for our present state, S_3.

WATCH ANIMATED FIGURE 8.37

Figure 8.37: Animation of the ROM-based 3-bit Gray code sequence generator.

On the rising edge of the clock, the next state value, 100, is loaded into the register and becomes our new present state. If input I remains at 1, the ROM address becomes 1001. The ROM then outputs the value 101110, corresponding to next state value 101 (S_5) and the output value for present state S_4, 110. This process repeats for every rising edge of the clock.

This overall configuration is used in many applications, including microprocessor design. It is used there for a type of control unit called a microsequencer. That is beyond the scope of this book, but you can find out more information about microsequencers and microprocessor design in my other book (Carpinelli, J. 2001. *Computer Systems Organization and Architecture*. Chapter 7) and numerous other books about computer architecture.

8.5 Refining Your Designs

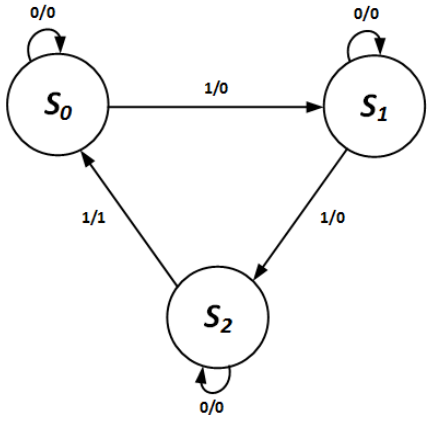
So far in this chapter, we have gone through the entire design process for finite state machines, or so it might appear. Actually, we have gone through the process of creating a first draft of our designs. The designs we have seen so far mostly work properly, but it is often necessary to refine our designs to minimize the hardware needed to implement them and to ensure they work properly under all conditions. In this section, we will look at three design refinements: handling unused states, combining equivalent states, and handling glitches.

8.5.1 Unused States

As we have seen in the designs presented so far in this chapter, finite state machines generally use either flip-flops, a register, or a counter to store the binary value representing the present state of the machine. A machine with n states requires $\lceil \lg n \rceil$ bits to store the state value. When n is an exact power of 2 (2, 4, 8, 16, etc.), this works out fine. Every value that could possibly be stored represents a valid state. When n is not an exact power of 2, however, our state machine will have one or more possible values that do not correspond to valid states. If any of these values ever ends up being stored as the state value, for example, when the system first powers up, our system may fail.

One way to avoid this problem is to create extra states that correspond to these unused values. Each of these states should unconditionally (regardless of the input values) transition to a valid state. Once the machine does this, it should function normally.

Consider again the Mealy machine implementation of the 1s counter that outputs a 1 for only one clock cycle. Its state diagram is repeated in Figure 8.38 (a), and its state table is shown again in Figure 8.38 (b).



(a)

PS	I	NS	O
S_0	0	S_0	0
S_0	1	S_1	0
S_1	0	S_1	0
S_1	1	S_2	0
S_2	0	S_2	0
S_2	1	S_0	1

(b)

Figure 8.38: Mealy machine (a) state diagram, and (b) state table for the 1s counter.

As we went through this design in Section 8.3.1, we decided to represent states S_0, S_1, and S_2 as 00, 01, and 10, respectively. We treated state value 11 as a don't care when developing the functions and digital logic to generate the next state and output values. The functions we developed are

$$NS_1 = PS_0I + PS_1I'$$
$$NS_0 = PS_0I' + PS_1'PS_0'I$$
$$O = PS_1I$$

If our machine ever ends up in state 11, this becomes

$$NS_1 = 1\text{^}I + 1\text{^}I' = 1$$
$$NS_0 = 1\text{^}I' + 0\text{^}0\text{^}I = I'$$
$$O = 1\text{^}I = I$$

In short, if $I = 0$, then we stay in this undefined state, and if $I = 1$ we transition to state S_2 (10) and incorrectly set the output to 1.

To resolve this problem, we can add one extra state to our machine for every unassigned state value. For this machine, there is only one unassigned state value, 11, so we add only one additional state to this machine, which I'll call S_3. I choose to have the machine go from this state to state S_0. The revised state diagram is shown in Figure 8.39 (a). Notice the notation on

the arc from S_3 to S_0. The dash indicates that this transition always occurs, no matter what values the inputs have. I also set the output to 0, as required.

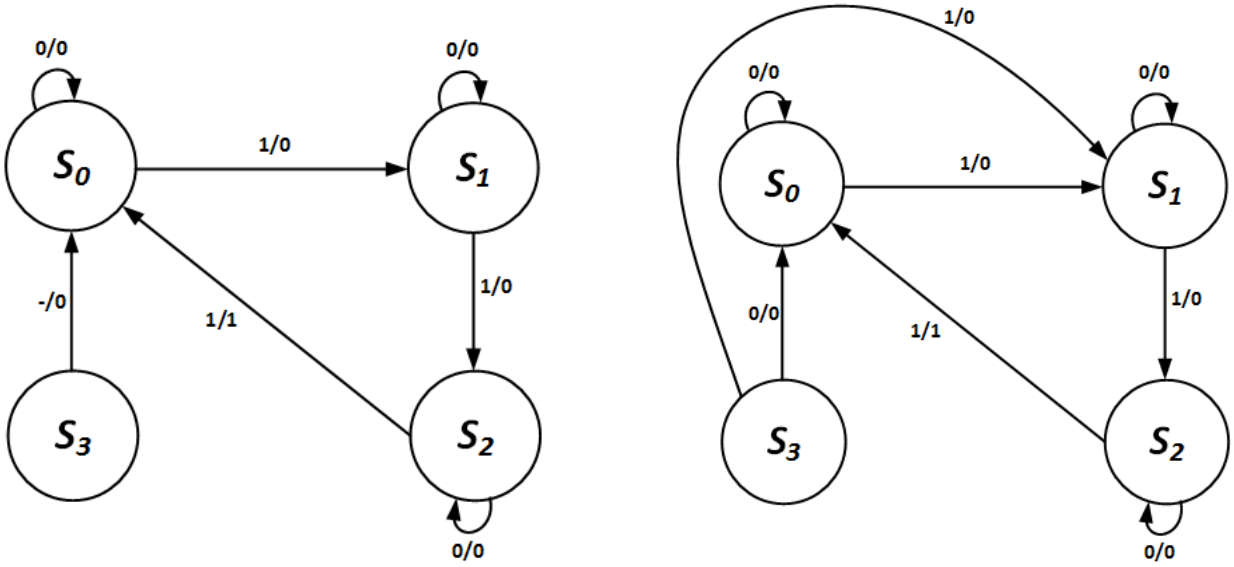


Figure 8.39: Revised state diagram for the Mealy machine implementation of the 1s counter: (a) Excluding the first input bit; (b) Including the first input bit.

This works fine, as long as the first input bit is 0. But what happens if it is 1? In that case, the first 1 takes the machine to state S_0, the next 1 takes it to S_1, and the third brings it to S_2. None of these transitions sets the output to 1. We need a fourth input of 1 to set the output of 1 in this initial sequence. One way to get around this issue is to specify that the state machine ignores the data read in during the first clock cycle. A better way is to modify the state diagram further, so that it goes from S_3 to S_0 when I = 0 and from S_3 to S_1 when I = 1. This is shown in Figure 8.39 (b).

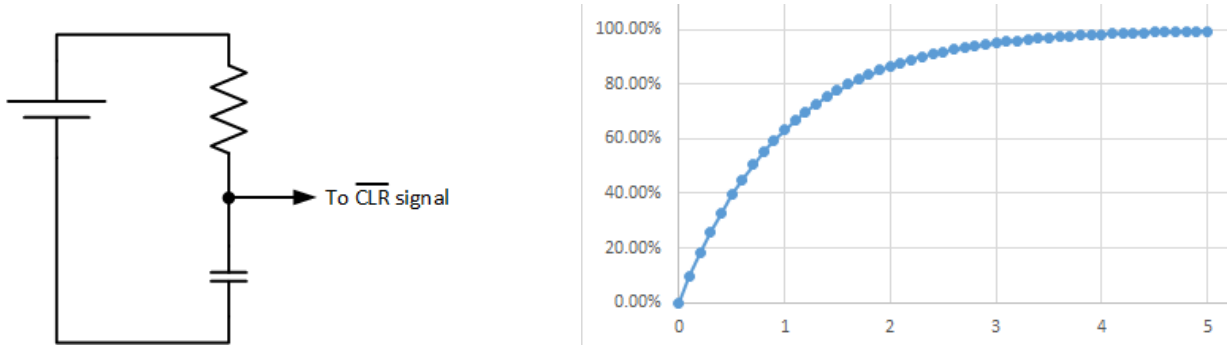
You can proceed through the rest of the design process as before to complete your design. When I say you can do this, I do mean you. This is left as an exercise for the reader.

There is one other strategy that I want to introduce to address this issue. When power is first applied to the state machine, we can asynchronously clear the flip-flops or other components used to store the state value. This forces the state value to become all zeroes, 00 for this machine. As long as this is a valid state, S_0 in this case, this strategy ensures that the machine goes to a valid state when power is applied.

Early microprocessors, such as Intel's 8085, have dedicated reset pins. The reset pin on the 8085 is active low. A logic 0 causes the processor to clear its registers and reset its state, much like we wish to do for the 1s counter.

To do this, designers set up an R-C circuit, as shown in Figure 8.40 (a). When the circuit initially powers up, there is no charge on the capacitor. All the voltage is dropped across the resistor, and the voltage at the point between the resistor and the capacitor is at 0 V, which a digital circuit would recognize as logic 0. The capacitor charges up and the voltage across the

capacitor increases roughly as shown in the voltage curve in Figure 8.40 (b). The time required to reach the maximum voltage will vary, depending on the values of R and C; after five time constants (R × C), it will have reached over 99% of the source voltage value. For the 8085, and for our circuit, this will be on the order of microseconds. When the capacitor reaches its maximum charge, it drops the entire 5 V and there is no voltage drop across the resistor. The voltage level at the point between the resistor and the capacitor is 5 V, which is recognized as a logic 1. The voltage stays at this level as long as the circuit has power.



(a) (b)

Figure 8.40: (a) R-C circuit to clear state value on power-up; (b) Voltage curve for the $\overline{CLR}$ signal.

For our state machine, we can connect the point between the resistor and capacitor to the active-low $\overline{CLR}$ signal of the flip-flops storing the state value. When the circuit first powers up, this signal is logic 0 and the flip-flops set their outputs to 0. Once the capacitor charges, $\overline{CLR}$ becomes logic 1 and does not affect the flip-flop's value any more.

8.5.2 Equivalent States

After creating the initial state diagram and state table for a finite state machine, it might be possible to reduce the number of states. We do this by identifying **equivalent states** and combining them into a single state. Reducing the number of states simplifies the state machine, which can lead to less digital logic needed to implement the machine, as well as other ancillary benefits such as reduced power requirements and faster circuits.

Two states are equivalent if all the conditions for its type of machine are met.

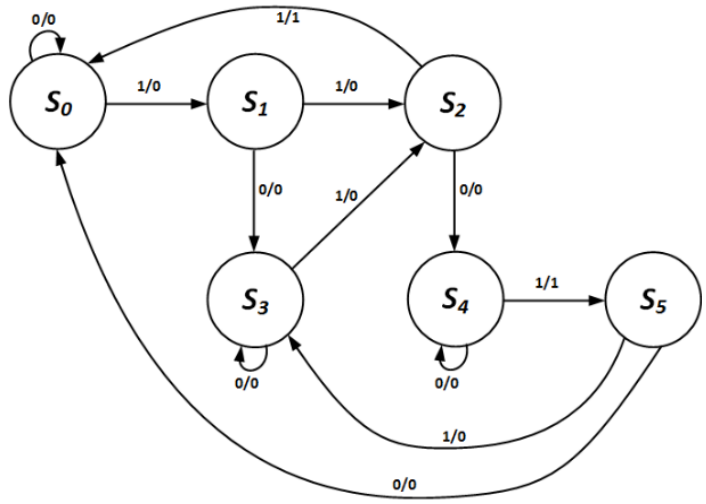
Moore machines:

- The states' output values are all exactly the same.
- For every combination of input values, both states transition to the same next state.

Mealy machines:

- For every combination of input values, both states transition to the same next state and produce **exactly** the same output values.

As one example, consider the state diagram and state table shown in Figure 8.41. To identify equivalent states, we compare every pair of states individually. Some pairs clearly are not equivalent because they generate different output values for the same input values, such as S_0 and S_2 with I = 1.

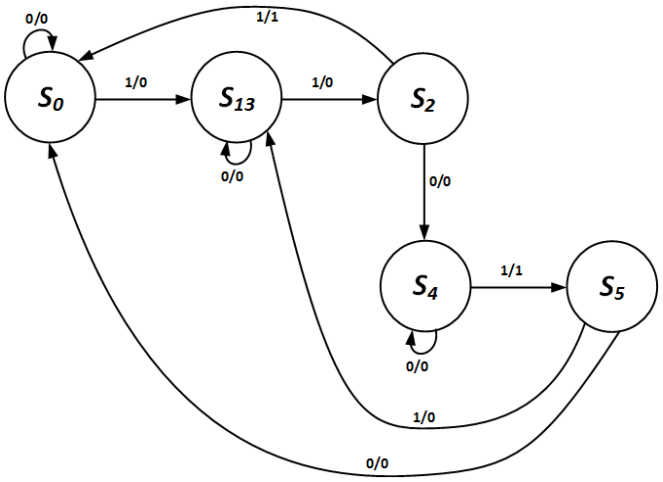


PS	I	NS	O
S_0	0	S_0	0
S_0	1	S_1	0
S_1	0	S_3	0
S_1	1	S_2	0
S_2	0	S_4	0
S_2	1	S_0	1
S_3	0	S_3	0
S_3	1	S_2	0
S_4	0	S_4	0
S_4	1	S_5	1
S_5	0	S_0	0
S_5	1	S_3	0

(a) (b)

Figure 8.41: Initial state diagram (a) and state table (b) for the example state machine.

For this machine, we find that states S_1 and S_3 are equivalent, so we combine them into a single state. I choose to combine them into a state named S_{13}, but any name will do. The arcs coming out of the combined state must be the same as those coming out of S_1 and S_3, which are identical. All arcs going into S_1 and S_3 must be redirected to the combined state. The state table must be updated to replace the two states. The updated state diagram and state table are shown in Figure 8.42.

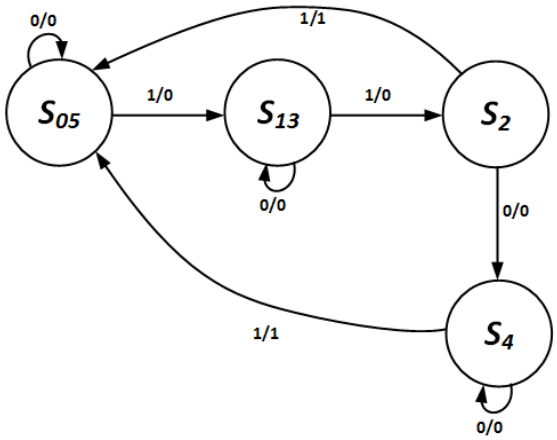


PS	I	NS	O
S_0	0	S_0	0
S_0	1	S_{13}	0
S_{13}	0	S_{13}	0
S_{13}	1	S_2	0
S_2	0	S_4	0
S_2	1	S_0	1
S_4	0	S_4	0
S_4	1	S_5	1
S_5	0	S_0	0
S_5	1	S_{13}	0

(a) (b)

Figure 8.42: State diagram (a) and state table (b) after combining S_1 and S_3.

Now we repeat this process on the reduced state machine, continuing until all equivalent states have been identified and combined. In this example, we find that states S_0 and S_5 are equivalent, and we combine them into a single state, S_{05}. The updated state diagram and state table are shown in Figure 8.43.

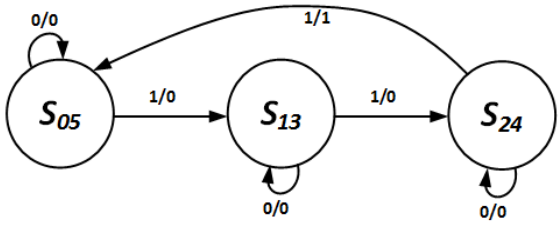


PS	I	NS	O
S_{05}	0	S_{05}	0
S_{05}	1	S_{13}	0
S_{13}	0	S_{13}	0
S_{13}	1	S_2	0
S_2	0	S_4	0
S_2	1	S_{05}	1
S_4	0	S_4	0
S_4	1	S_{05}	1

(a) (b)

Figure 8.43: State diagram (a) and state table (b) after combining S_0 and S_5.

With S_0 and S_5 combined, we see that S_2 and S_4 are now equivalent. We combine them to give us the state diagram and state table shown in Figure 8.44. None of these states are equivalent, so this becomes our final design, in this case the Mealy machine for the 1s counter.



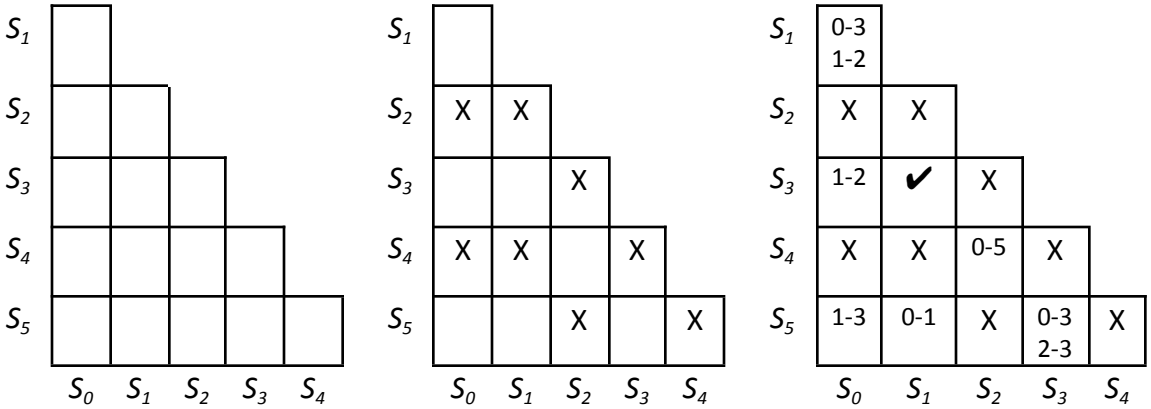
PS	I	NS	O
S_{05}	0	S_{05}	0
S_{05}	1	S_{13}	0
S_{13}	0	S_{13}	0
S_{13}	1	S_{24}	0
S_{24}	0	S_{24}	0
S_{24}	1	S_{05}	1

(a) (b)

Figure 8.44: Final state diagram (a) and state table (b) after combining S_2 and S_4.

WATCH ANIMATED FIGURE 8.44

To formalize this process, designers can use a mechanism called an **implication table**. We start by creating a table with one row per state for all but one state, and one column per state for all but one different state. By convention, we exclude the first state from the rows and the last state from the columns, but any selections will work. Each entry indicates the equivalence of the states for its row and column. We remove the redundant entries; for example, we keep row S_3 column S_2, but we remove row S_2 column S_3. We also remove entries with the same state in the row and column. Doing this for the previous example gives us the empty table shown in Figure 8.45 (a).



(a) (b) (c)

Figure 8.45: Implication table for the state diagram and state table in Figure 8.41: (a) Blank table; (b) Table with states that cannot be equivalent; (c) Completed initial table.

Next, we populate our table. For each cell, we look at the entries in the state table for the states of its row and column. For each input value, we compare the outputs. If they are not the same for any input values, the two states cannot be equivalent. For example, in the state table in Figure 8.41 (b), we see that S_0 outputs a 0 when $I = 0$ and $I = 1$. State S_2, however, outputs a 0 when $I = 0$, but it outputs a 1 when $I = 1$. Therefore, S_0 and S_2 cannot be equivalent, and we place an X at the location in the table for row S_2 and column S_0. Figure 8.45 (b) shows the table with states that cannot be equivalent.

The cells that are still blank may or may not be equivalent. Since we know their outputs are the same, we examine their next states. Sometimes we are fortunate to have a pair of states that go to exactly the same next states for all possible input values. In this state machine, S_1 and S_3 both transition to S_3 when $I = 0$ and S_2 when $I = 1$. We denote this using a ✔ in the table.

For all other cells, the two states are equivalent if their next states are equivalent. We don't know if their next states are equivalent, at least not yet, so we simply list the required equivalences in each cell. As an example, consider the uppermost cell in row S_1 column S_0. If $I = 0$, the next state for S_0 is S_0 and the next state for S_1 is S_3. This means that these two states can only be equivalent if states S_0 and S_3 are equivalent. Also, if $I = 1$, S_0 transitions to S_1 and S_1 goes to S_2, so we must also have S_1 and S_2 be equivalent. We place both entries in the cell since we need both to be true for S_0 and S_1 to be equivalent.

Some entries are self-evident and can be excluded. Consider the cell for states S_0 and S_5. If $I = 0$, both states transition to state S_0. We don't need to include this in the table; S_0 is always equivalent to itself.

Similarly, we don't need to include an equivalency that refers to the cell itself. For example, when $I = 0$, S_0 goes to the next state S_0, and S_3 goes to S_3. We do not need to include this entry in the table. That would be like saying "S_0 and S_3 are equivalent if S_0 and S_3 are equivalent." This is clearly true, and including it in the table doesn't help us find the equivalent states.

Following this process for each cell gives us the completed initial table shown in Figure 8.45 (c).

Next, we update the table based on the pairs of states that we know are or are not equivalent. For example, we know that S_1 and S_2 are not equivalent, so every cell that has this combination as a requirement can be changed to an X. That is, since S_0 and S_3 are equivalent only if S_1 and S_2 are equivalent, and we know that S_1 and S_2 are not equivalent, then S_0 and S_3 are not equivalent.

We also know that S_1 and S_3 are equivalent. Looking at the table, we see that S_0 and S_5 are equivalent only if S_1 and S_3 are equivalent. Since there are no other requirements for S_0 and S_5, these two states are equivalent, and we change the entry for this cell to ✔. Doing this for each table entry gives us the updated table shown in Figure 8.46.

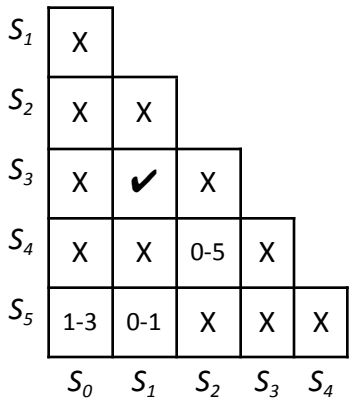


Figure 8.46: Implication table updated for initial entries.

Just as we did in the previous example, we would continue to revise our table to consider the new values created in the previous iteration, repeating until all entries are completed. We only need one more iteration for this example. Since S_0 and S_1 are not equivalent, S_1 and S_5 cannot be equivalent. Also, because S_0 and S_5 are equivalent, S_2 and S_4 must be equivalent. This gives us the completed table shown in Figure 8.47. With this information, we can update the state diagram and state table, and proceed with the rest of the design process.

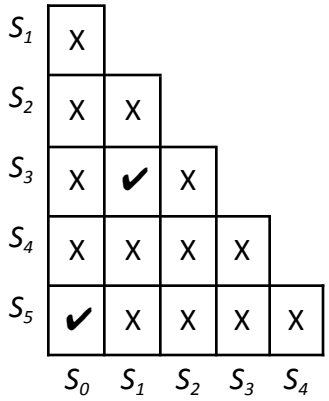


Figure 8.47: Completed implication table.

WATCH ANIMATED FIGURE 8.47

8.5.3 Glitches

Glitches are the bane of digital designers. Often just a momentary change in a signal's value, they can wreak havoc with an otherwise well-designed system. They can be caused by

component timing and propagation delay issues, signal noise, or transmission line effects. They can also be caused by design errors, particularly those that do not take all possibilities into account. We look at one of these errors in this subsection.

To examine this error, we return to the Mealy machine for the 1s counter shown in Figure 8.19. Consider the values for the clock, present state, and input *I* shown in Figure 8.48. Before continuing, take a couple of minutes to sketch out the values of the next state and output *O*, as well as the value of the present state once the rising edge of the clock is reached.

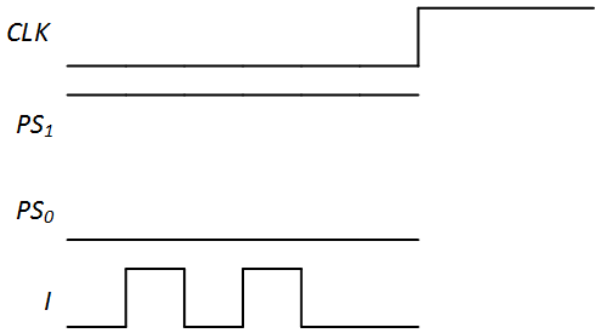


Figure 8.48: Partial timing diagram for the 1s counter.

Hopefully you've completed the timing diagram and found the area of concern. Just so we're all on the same page, we'll work from my completed timing diagram, shown in Figure 8.49. As input *I* changes, both NS_1 and output *O* also change. The change in NS_1 isn't really a problem, since this value is not loaded into its flip-flop until we have a rising edge on the clock signal. The change in *O*, however, is quite visible and is not what we want our system to show to the outside world.

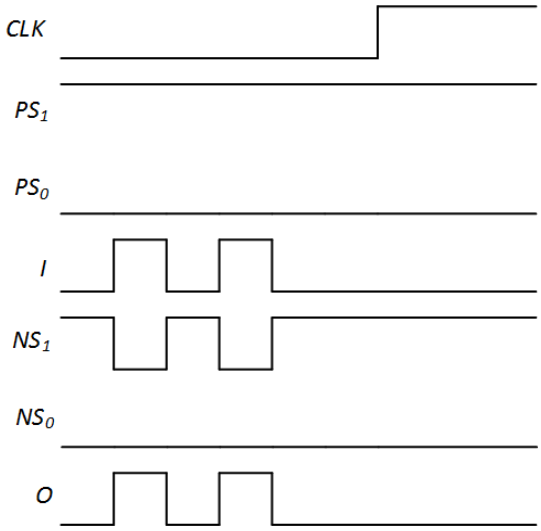


Figure 8.49: Completed timing diagram for the 1s counter.

WATCH ANIMATED FIGURE 8.49

The problem occurs because the state transition is synchronized to the clock, while the output is not. That is, the present state only changes on the rising edge of the clock, but the output can change at any time. In this timing diagram, a change in the value of input *I* changes the value of output *O* immediately.

The present state does not have this issue because it uses an edge-triggered flip-flop, and we can use this same strategy to get rid of this problem with output *O*. We send the value we generate for *O* to an edge-triggered D flip-flop and connect the same clock signal used for the present state flip-flops to the new flip-flop, as shown in Figure 8.50. Now, *O* will also change only on the rising edge of the clock; it will not be impacted by the glitch on input *I.*

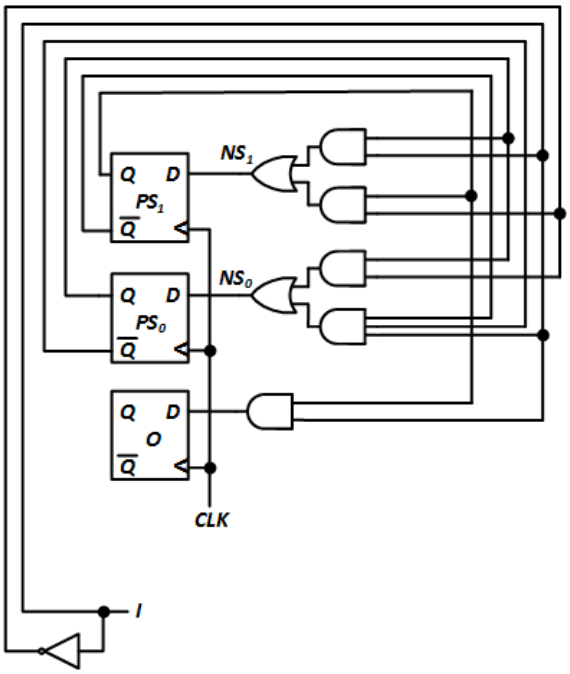


Figure 8.50: Circuit implementing the Mealy machine for the 1s counter with flip-flop for output *O*.

8.6 Summary

Sequential digital systems can be modeled as finite state machines. A finite state machine accepts inputs, and outputs values, just like combinatorial circuits. But unlike combinatorial circuits, the outputs depend on both the input values and the present state of the system. The same input values, when accepted while the system is in different states, may produce different output values.

To design a finite state machine, we determine all possible states. For each state we consider all possible input values to ascertain the next state of the machine and the outputs to be generated. We can represent this information using a state diagram and a state table. After assigning a binary value to each state, we can develop the functions and design the digital logic to generate the next state and output values.

There are two primary types of finite state machines that vary in how they generate their outputs. The outputs of the Mealy machine are a function of both the present state and

the input values. Moore machines generate their outputs based solely on the present state. Both machines use both the present state and input values to produce the next state.

Most finite state machines use flip-flops or registers to store their present state, though it is also possible to use a counter for some systems. Decoders can be used to generate signals corresponding to individual states. A lookup ROM may be used to generate next state and output values in place of traditional combinatorial logic.

Initial designs can often be refined to correct errors, account for unused state values, and simplify the design and resultant hardware. Designers can create additional states that correspond to all unused binary state values. Should the circuit end up in one of these states, such as when the circuit initially powers up, the designer ensures that the machine transitions to a valid state and then functions properly.

Some machines may have states that are equivalent. These states can be merged into a single state, which simplifies the state machine and ultimately the circuit designed to implement the state machine. Implication tables can be used to identify equivalent states.

Glitches can occur for a number of reasons. When designing a circuit to implement a finite state machine, it is important to ensure that output values change only at the desired time. Using flip-flops to store the output value is one way to address this issue.

This completes Part III of this book. In the next chapter, we introduce asynchronous circuits. Although less frequently used than synchronous circuits, they have valid real-world applications. However, they also have issues that must be taken into account during system design, issues beyond those of synchronous systems. We'll look at all of this next.

Bibliography

Carpinelli, J. D. (2001). *Computer systems organization & architecture*. Addison-Wesley.

Hayes, J. P. (1993). *Introduction to digital logic design*. Addison-Wesley.

Intel Corporation. (1983). *The MCS-80/85 family user's manual*. Intel Corporation.

Mano, M. M., & Ciletti, M. (2017). *Digital design: sith an Introduction to the Verilog HDL, VHDL, and SystemVerilog* (6th ed.). Pearson.

Mano, M. M., Kime, C., & Martin, T. (2015). *Logic & computer design fundamentals* (5th ed.). Pearson.

Mealy, G. H. (1955). A method for synthesizing sequential circuits. *The Bell System Technical Journal*, *34*(5), 1045–1079. https://doi.org/10.1002/j.1538-7305.1955.tb03788.x

Merriam-Webster. (n.d.) State. In Merriam-Webster.com dictionary. Retrieved July 27, 2023, from https://www.merriam-webster.com/dictionary/state

Minsky, M. L. (1967). *Computation: finite and infinite machines*. Prentice-Hall.

Moore, E.F. (1956). Gedanken-experiments on sequential machines. *Annals of Mathematics Studies 34*, 129-153.

Nelson, V., Nagle, H., Carroll, B., & Irwin, D. (1995). *Digital logic circuit analysis and design*. Pearson.

Roth, J. C. H., Kinney, L. L., & John, E.B. (2020). *Fundamentals of logic design Enhanced Edition* (7th ed.). Cengage Learning.

Sandige, R. S. (1990). *Modern digital design*. McGraw-Hill.

Texas Instruments, Inc. (1981). *The TTL Data Book for Design Engineers*. (2nd ed.). Texas Instruments.

Turing, A. M. (1937). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, *s2-42*(1), 230–265. https://doi.org/10.1112/plms/s2-42.1.230

Turing, A. M. (1938). On computable numbers, with an application to the Entscheidungsproblem. A Correction. *Proceedings of the London Mathematical Society*, *s2-43*(1), 544–546. https://doi.org/10.1112/plms/s2-43.6.544

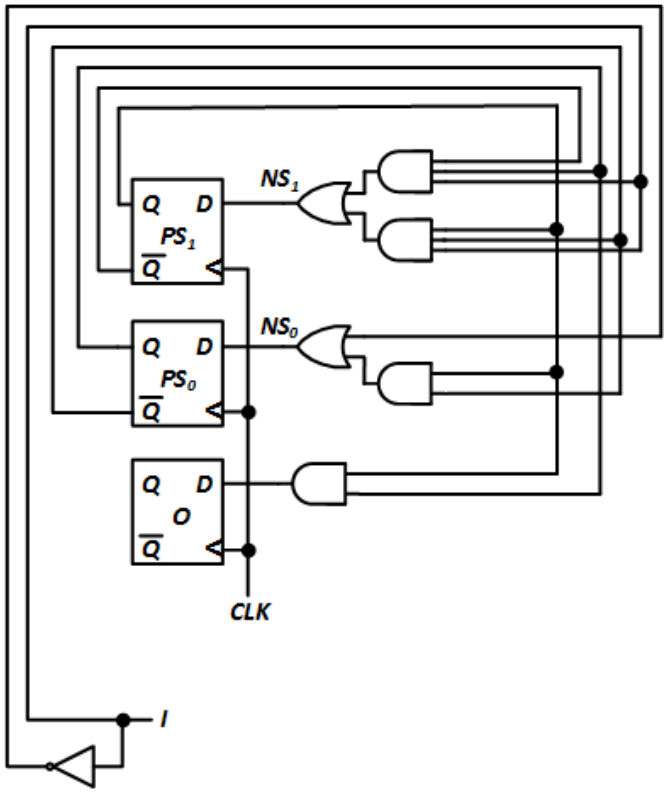
Wakerly, J. F. (2018). *Digital design: principles and practices* (5th ed.). Pearson.

Exercises

1. List the states of a 3-bit binary counter.
2. List the states of a BCD counter.
3. Show the Mealy machine state diagram and state table for a J-K flip-flop.
4. Show the Moore machine state diagram and state table for a J-K flip-flop.
5. Show the Mealy machine state diagram and state table for a T flip-flop.
6. Show the Moore machine state diagram and state table for a T flip-flop.
7. Modify the Mealy machine 1s counter state diagram and state table so it outputs a 1 for one clock cycle after counting five inputs equal to 1.
8. Complete the design of the Mealy machine with the state diagram and state table you developed in the previous problem.
9. Modify the Moore machine 1s counter state diagram and state table so it outputs a 1 for one clock cycle after counting five inputs equal to 1.
10. Complete the design of the Moore machine with the state diagram and state table you developed in the previous problem.
11. Is the following state table for a Mealy or Moore machine? How can you tell?

PS	*I*	*NS*	O_1	O_0
S_0	0	S_0	0	0
S_0	1	S_1	1	0
S_1	0	S_1	1	0
S_1	1	S_2	1	1
S_2	0	S_2	1	1
S_2	1	S_3	0	1
S_3	0	S_3	0	1
S_3	1	S_0	0	0

12. Show the state table and state diagram for the following circuit.



13. Design a circuit that reads in a series of individual bits and outputs a 1 whenever the sequence 1101 is read in. The circuit has a single input, *I*, and a single output, *O*. Design your circuit as a Mealy machine.

14. Design the circuit in the previous problem as a Moore machine.

15. Redesign the Mealy machine for the 1s counter that outputs a 1 for a single clock cycle so that it uses the following state assignments: S_0 = 11; S_1 = 00; S_2 = 01.

16. Redesign the Moore machine for the 1s counter that outputs a 1 for a single clock cycle so that it uses the following state assignments: S_0 = 11; S_1 = 00; S_2 = 01; S_3 = 10.

17. Redesign the Mealy machine for the original 1s counter using a 2-bit counter with a clear input.

18. Redesign the Moore machine for the original 1s counter using a 2-bit counter with a clear input.

19. Design a 3-bit Gray code sequence generator as a Mealy machine using a 3-bit binary counter and combinatorial logic.

20. Design a 4-bit Gray code sequence generator using a 4-bit binary counter and combinatorial logic.

21. Design a 4-bit Gray code sequence generator using a 4-bit binary counter and a 4 to 16 decoder.

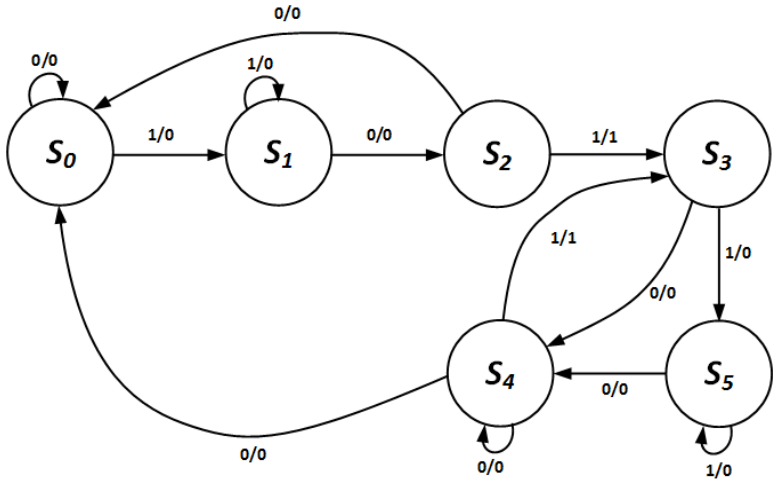
22. Redesign the 1s counter using a lookup ROM.

23. Design a 3-bit Gray code sequence generator as a Moore machine using a lookup ROM.

24. Complete the design of the 1s counter in Section 8.5.1 using the state diagram in Figure 8.39 (b).

25. Revise the design of the BCD counter to incorporate unused states.

26. For the following state diagram (a) and state table (b), simplify this design by combining all equivalent states.



(a)

PS	*I*	*NS*	*O*
S_0	0	S_0	0
S_0	1	S_1	0
S_1	0	S_2	0
S_1	1	S_1	0
S_2	0	S_0	1
S_2	1	S_3	0
S_3	0	S_4	0
S_3	1	S_5	0
S_4	0	S_4	0
S_4	1	S_3	1
S_5	0	S_4	0
S_5	1	S_5	0

(b)

PART IV

Advanced Topics

Chapter 9

Asynchronous Sequential Circuits

Chapter 9: Asynchronous Sequential Circuits

The finite state machines introduced in the last chapter are *synchronous*, that is, they use a clock input to synchronize the flow of data within the circuit. They primarily do this by loading positive-edge-triggered flip-flops to lock in the value of the present state, and possibly also the values of system outputs. Most sequential systems are synchronous, but some are not. This chapter introduces **asynchronous sequential circuits**.

We start by describing the basic characteristics and model of asynchronous sequential circuits. Next, this chapter introduces the asynchronous system design process, first by analyzing an existing asynchronous circuit and then by demonstrating a complete design.

There are specific issues related to asynchronous sequential design that are not present, or are present much less frequently, in synchronous circuits. This chapter introduces oscillation, hazards, and races, and design strategies to resolve these issues in asynchronous sequential circuit design.

9.1 Overview and Model

The clock frequency used in digital circuits has increased dramatically over the past several decades. Whereas early microprocessors used clocks with frequencies around 3 MHz (3 million cycles per second), modern microprocessors use clocks with frequencies over 3 GHz, a thousand-fold increase. Due to improvements in the technology used to create integrated circuit chips, the individual components on these chips produce outputs more quickly, take up less space, and use less power per component. Producing outputs more quickly lowers the propagation delay and is primarily responsible for the faster clock time. But even at such high clock frequencies, there is sometimes a need for circuits that are even faster than these clock frequencies can support. This need can sometimes be met by asynchronous sequential circuits.

Asynchronous sequential circuits were mentioned briefly in Chapter 6. A generic model for these circuits is repeated in Figure 9.1. It is similar to the model for synchronous sequential circuits, with a couple of differences.

- Asynchronous sequential circuits do not use flip-flops or other storage elements to hold the present state. Instead, we model these circuits using a delay. The input to the delay block is the next state of the system. After a delay, this is output as the new present state.
- Since there are no flip-flops, there is no need for a clock signal.

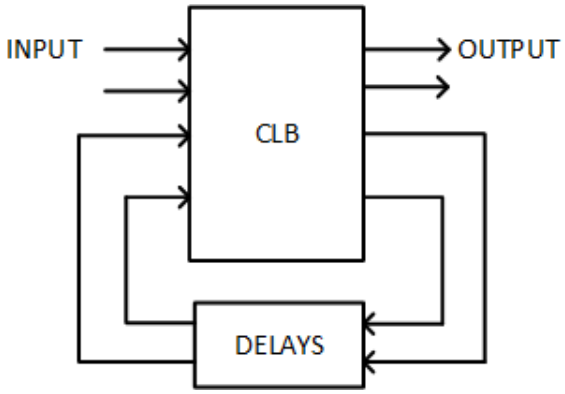


Figure 9.1: Generic model for an asynchronous sequential circuit.

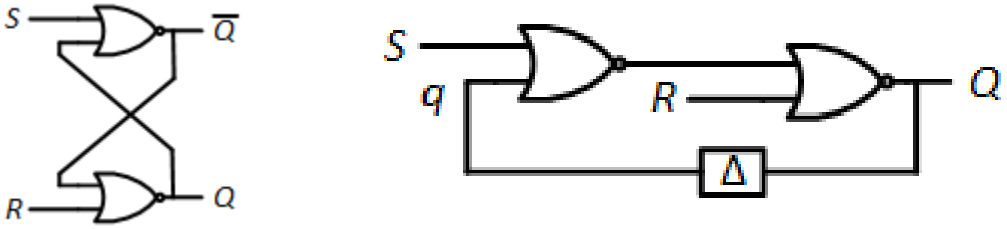
Although there are differences between asynchronous and synchronous sequential circuits, there are also many similarities. Each employs a combinatorial logic block that receives system input values and the present state, and produces system outputs and the next state value. The design processes for the two are also similar, as are the tools used in the design process. In the next section, we'll examine the design process and the mechanisms we use in more detail.

9.2 Design Process

Before getting into the design process for asynchronous sequential systems, this section introduces several tools that we will use in the design process. To do this, we'll analyze a given circuit to illustrate how these tools work. Then we'll go through a complete design from beginning to end.

9.2.1 Analysis Example

To introduce the design tools, we will analyze an asynchronous sequential circuit we have already seen in this book, the S-R latch constructed using NOR gates. For this example, we will only consider the Q output of the latch. We will ignore the $\overline{Q}$ output. Figure 9.2 (a) shows the design for this latch.



(a) (b)

Figure 9.2: Asynchronous sequential circuit: the partial S-R latch: (a) Logic diagram; (b) Redrawn to incorporate delay and feedback.

When modeling asynchronous sequential circuits, it is standard practice to incorporate a delay block between what we would call the present state and the next state. We redraw the logic diagram to include this delay, and we reformat it to emphasize the feedback path in the circuit. This is shown in Figure 9.2 (b). By tracing through the data paths, you can verify that both circuits are the same.

Notice the new label, *q*, associated with the lower input of the leftmost NOR gate. In a steady state, *q* has the same value as *Q*. When *Q* changes, *q* will also change, but only after a delay. In this circuit, *q* is similar to the present state of the circuit; *Q* is both a circuit output and similar to the next state.

From this circuit, we can determine the value of *Q* as a function of *S*, *R*, and *q*, as follows.

$$Q = ((S + q)' + R)'$$

Using DeMorgan's Law of the form $(A + B)' = A'B'$, this becomes

$$\begin{aligned} Q &= ((S + q)')'R' \\ &= (S + q)\, R' \end{aligned}$$

Now that we have our function for *Q*, we can create a **transition table**. This looks very similar to a truth table. For this circuit, the table inputs are the circuit inputs, *S*, *R*, and *q*, and the output is circuit output *Q*. Just as with a truth table, we determine the output value for all possible combinations of input values. The transition table for this circuit is shown in Figure 9.3. Ignore the column labeled *PSD* for now; we'll explain and make use of that shortly.

S	*R*	*q*	*Q*	PSD state
0	0	0	0	*a*
0	0	1	1	*b*
0	1	0	0	*c*
0	1	1	0	*d*
1	0	0	1	*e*
1	0	1	1	*f*
1	1	0	0	*g*
1	1	1	1	*h*

Figure 9.3: Transition table for the S-R latch.

We use the transition table to create the **excitation map**. This looks very much like the Karnaugh maps we have been using throughout this book. As with Karnaugh maps, we set up a grid, with the rows and columns corresponding to all possible input values. Figure 9.4 shows two ways to draw the excitation map for this circuit; they are equivalent.

$q\backslash SR$	00	01	11	10
0	0	0	0	1
1	1	0	0	1

$SR\backslash q$	0	1
00	0	1
01	0	0
11	0	0
10	1	1

Figure 9.4: Excitation maps for the S-R latch. The two maps are equivalent.

There is one very significant difference between the excitation map and a Karnaugh map. Notice that some, but not all of the entries are circled. The entries that are circled are the **stable states** of the circuit. A state is stable when, for its input values, the circuit is to remain in that state. That is, its present state and next state are the same, or $q = Q$.

To illustrate a stable state, consider the case when $S = 0$, $R = 0$, and $Q = 1$. An S-R latch with $SR = 00$ should retain its current value. Let's examine this from the beginning. Initially, $Q = 1$. After a brief delay, $q = 1$. The inputs to the first NOR gate are $S = 0$ and $q = 1$, and its output is $(0 + 1)' = 0$. The second NOR gate inputs this value and $R = 0$, outputting $(0 + 0)' = 1$. This cycle repeats continuously, and Q does not change. This is shown in Figure 9.5 and its animation.

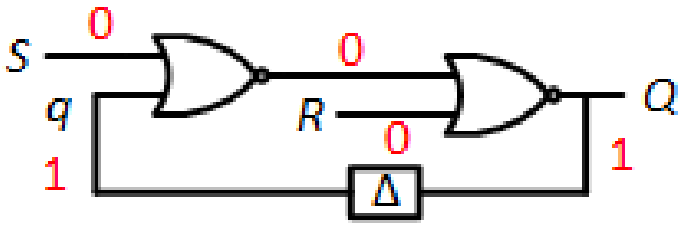


Figure 9.5: S-R latch in a stable state.

WATCH ANIMATED FIGURE 9.5

Not all states, however, are stable. When $Q \neq q$, the state is called a **transient state**. When a circuit enters a transient state, it immediately (after our Δ delay) transitions to another state. Because this occurs so quickly, we model this with our input values not changing while the circuit is in this state. If it goes to another transient state, it transitions out of that state, continuing until it reaches a stable state. A well-designed circuit will always reach a stable state. If it doesn't, well, we'll discuss this more later in this chapter.

As an example, let's say we are in the stable state just discussed, with $S = 0$, $R = 0$, and $Q = q = 1$. Then the R input changes from 0 to 1, giving us $S = 0$, $R = 1$, and $Q = 1$. For an S-R latch, $S = 0$ and $R = 1$ should set $Q = 0$, not 1, so this state is transient, not stable. Tracing through our circuit, the first NOR gate has inputs $S = 0$ and $q = 1$ and outputs a 0. The second NOR gate has this input and $R = 1$, and it also outputs a 0. This sets $Q = 0$ and, after a brief delay, also sets $q =$

0. Now the first NOR gate has inputs $S = 0$ and $q = 0$ and outputs a 1. The second NOR gate has this 1 and $R = 1$ as inputs and outputs a 0. The circuit has transitioned from the transient state with $S = 0$, $R = 1$, and $Q = 1$ to the stable state with $S = 0$, $R = 1$, and $Q = 0$. This is shown in Figure 9.6 and its animation.

Figure 9.6: Transition from (a) a transient state to (b) a stable state.

WATCH ANIMATED FIGURE 9.6

Now, back to our analysis and one final tool, the **primitive state diagram**. This is similar to the state diagrams we have seen earlier in this book, but it takes into account the constraints placed on our system. The primitive state diagram for the S-R latch is shown in Figure 9.7. It uses the values in the *PSD* column of the state transition table in Figure 9.3 to denote the states.

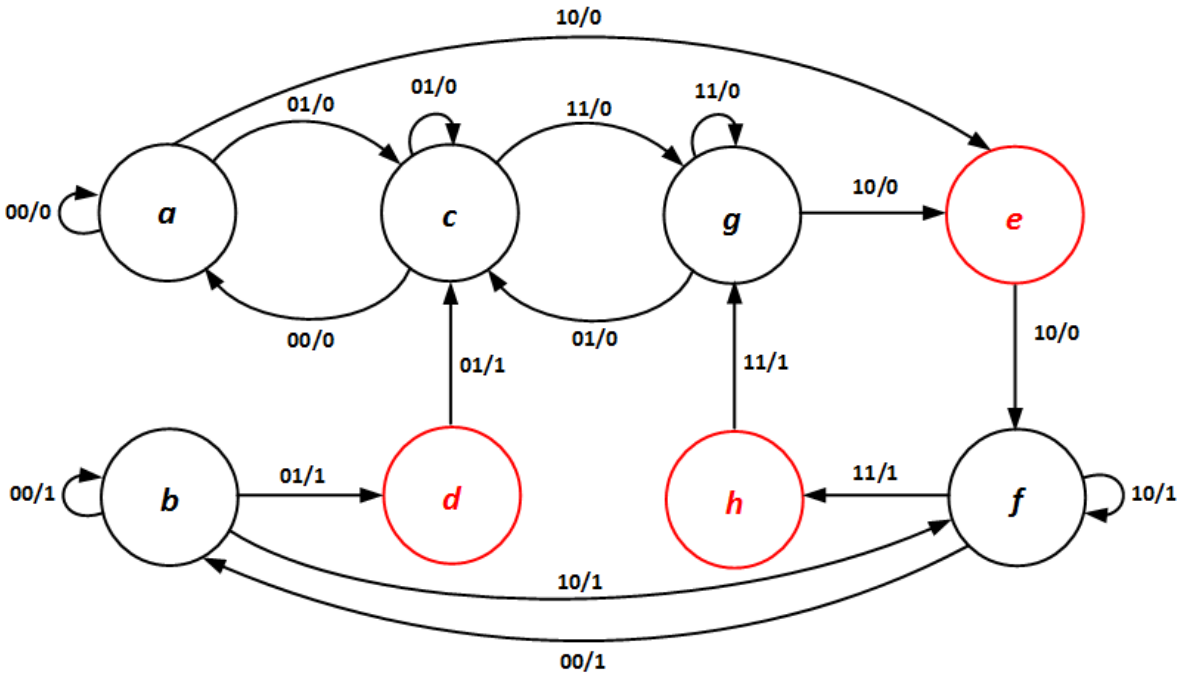


Figure 9.7: Primitive state diagram for the S-R latch. Transient states are shown in red.

We placed two constraints on asynchronous sequential circuits, and both are reflected in this primitive state diagram. First, at most one input value can change at a time. For this reason, arcs are not included for inputs that change two values. For example, state *a* corresponds to *S* = 0 and *R* = 0. The primitive state diagram does not include an arc from state *a* with input values *S* = 1 and *R* = 1. The other three input combinations, *SR* = 00, 01, and 10, are included in the diagram.

The second constraint is that input values do not change during transient states. The transient states in the primitive state diagram, *d*, *e*, and *h*, each have one arc entering and one arc exiting; both have the same input values. Of course, the output value is different. If it were the same, our system would remain in the same state and this state would be stable, not transient.

With these tools, and those we will introduce shortly, we are ready to design an asynchronous sequential system from its initial specification to its final implementation. We'll do that in the next subsection.

9.2.2 Design Example

There are several ways to divide the steps in the design process for an asynchronous sequential system. Here are the steps we will use in this book.

1. Develop system specifications.
2. Define all states.
3. Minimize states.
4. Assign binary values to states.
5. Determine functions and design the circuit.

We will examine each of these steps as we proceed through our example design. In this subsection, we will design the S-R latch with only a *Q* output.

Step 1: Develop System Specifications

We wish to design an asynchronous sequential system with two inputs, *S* and *R*, and one output, *Q*. When *SR* = 00, output *Q* should retain its previous value. If *SR* = 01, the system should set *Q* = 0, and the system should output *Q* = 1 when *SR* = 10. Finally, which *SR* = 11, the system should set output *Q* to 0. Figure 9.8 shows a block diagram of our system.

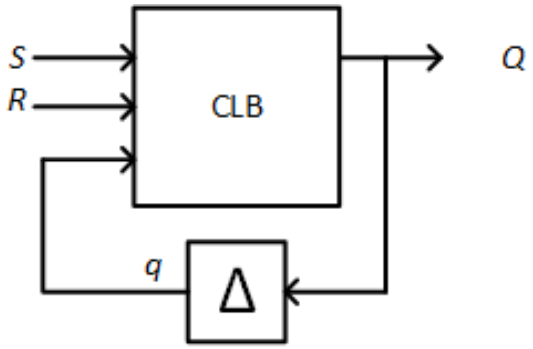


Figure 9.8: Block diagram for the S-R latch.

Step 2: Define States

In the block diagram for this system, there are three inputs to the combinatorial logic block that generate the next state and output values. These three inputs can take on any of $2^3 = 8$ possible values, from 000 to 111, so this system has eight states (before minimization). Some of these states are stable, and the rest are transient. Figure 9.9 shows these states.

State	*S*	*R*	*q*	Condition
a	0	0	0	*S*=0, *R*=0, Previously *Q* = 0
b	0	0	1	*S*=0, *R*=0, Previously *Q* = 1
c	0	1	0	*S*=0, *R*=1, Sets *Q* = 0
d	0	1	1	Transition to state *c*
e	1	0	0	Transition to state *f*
f	1	0	1	*S*=1, *R*=0, Sets *Q* = 1
g	1	1	0	*S*=1, *R*=1, Sets *Q* = 0
h	1	1	1	Transition to state *g*

Figure 9.9: Primitive states for the S-R latch. Transient states are shown in red.

To develop this table, we would go to the specification and identify values corresponding to the specification. These are the stable states of the system. The **Condition** column in the table shows which condition in the specification corresponds to each state.

The remaining values represent transient states. When the system is in a transient state, we want it to transition to a stable state. Its inputs (*S* and *R* for this system) cannot change when the system is in a transient state, so it must go to another state with the same input values but different outputs. The Condition column gives the next state for each transient state.

Step 3: Minimize States

To minimize states, we first create the primitive state diagram and a new mechanism, the **primitive state table**. Let's start with the primitive state diagram. For each state, we list all possible valid input combinations and determine the output value to be generated. We exclude values for which more than one input changes, and those that are not the same inputs as were

set when the system enters a transient state. Figure 9.10 shows the primitive state diagram for this system. This is exactly the state diagram we derived for the S-R latch in Figure 9.7 in the previous subsection.

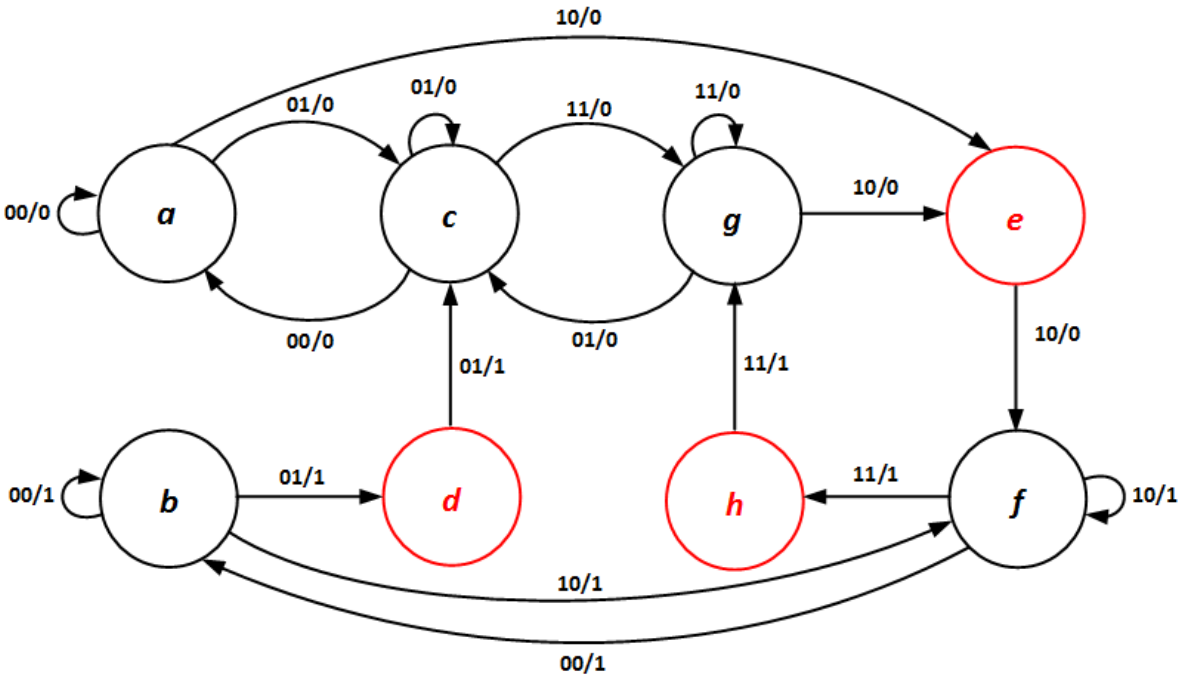


Figure 9.10: Primitive state diagram for the S-R latch.

Notice that some transitions are not made directly. For example, if the system is in state *b* ($S = 0$, $R = 0$, $Q = q = 1$) and R changes to 1, the system should transition to state *c* ($S = 0$, $R = 1$, $Q = q = 0$). Remember that when input values change, Q does not immediately change; there is a slight delay. In the primitive state diagram, I have placed all states with $q = 0$ in the upper row of the diagram, and all states in the lower row have $q = 1$. When input values change, the machine transitions to another state in the same row, that is, the output does not change. Also, the states in each column have the same input values but different output values. It is within these states that the machine changes its output. This can be summarized as follows.

> *When an asynchronous sequential system changes input and output values, it first changes its input values and, after a very brief transient delay, changes its output values.*

Now let's look at the primitive state table. This table is equivalent to the primitive state diagram. To construct this table, we list each state in the first column. We then create one additional column for each possible combination of input values; for this example, the four columns correspond to $SR = 00$, 01, 11, and 10. Then we fill in the table. For each state and input values, we list the next state and output values generated. When an input value is not valid, we place a dash in the table. As you'll see soon, we treat the dashes as don't care values. The complete primitive state table for this system is shown in Figure 9.11.

State	00	01	11	10
a	*a*,0	*c*,0	---	*e*,0
b	*b*,1	*d*,1	---	*f*,1
c	*a*,0	*c*,0	*g*,0	---
d	---	*c*,0	---	---
e	---	---	---	*f*,1
f	*b*,1	---	*h*,1	*f*,1
g	---	*c*,0	*g*,0	*e*,0
h	---	---	*g*,0	---

Figure 9.11: Primitive state table for the S-R latch.

Now we are ready to minimize the states in our system. Since the primitive state diagram does not include the don't care conditions, it may be easier to use the primitive state table. To identify equivalent states and minimize the number of states, we compare every pair of states individually. When comparing two states, they are equivalent only if all the entries are equivalent. Two entries are equivalent if (i) they have the same state and output value, or (ii) one or both are don't cares.

We start by comparing states *a* and *b*. Since their entries for input values 00, 01, and 10 have different outputs, they cannot be equivalent.

Moving on to *a* and *c*, we find that both have *a*,0 for input value 00, and *c*,0 for input value 01. So far so good. In the next column, 11, state *c* has the value *g*,0 and *a* has a don't care. By definition, they also match. Finally, column *a* has *e*,0 and *c* has a don't care, so this matches as well. Since all entries match, *a* and *c* are equivalent.

We continue this process until we have compared every pair of states. Then we can create an implication table for our results. If we find any conditions necessary for equivalence, we include these entries in the table and continue to check these entries until the table is complete. For this example, there are no such conditions, and we get the implication table shown in Figure 9.12.

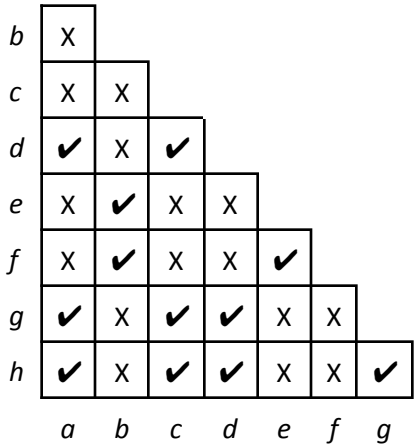


Figure 9.12: Implication table for the primitive state table for the S-R latch.

From this table, we see that there are several pairs of states that are equivalent, namely *a* and *c*, *a* and *d*, *a* and *g*, *a* and *h*, *b* and *e*, *b* and *f*, *c* and *d*, *c* and *g*, *c* and *h*, *d* and *g*, *d* and *h*, *e* and *f*, and *g* and *h*.

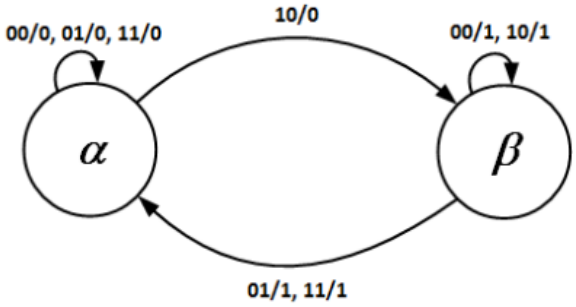
We could choose one equivalence, combine the states, and repeat this process to see if more than two states are equivalent, but there is an easier way to do this. *For any set of states, the states are all equivalent if and only if every pair of states within the set are equivalent.*

Consider states *b*, *e*, and *f*. We have found that *b* and *e* are equivalent, *b* and *f* are equivalent, and *e* and *f* are equivalent. Therefore, states *b*, *e*, and *f* are all equivalent to each other and can be combined. Following this same procedure, we can show that states *a*, *c*, *d*, *g*, and *h* are also all equivalent. Our 8-state system can be reduced to a 2-state system.

To combine equivalent states, we create a single state that has the entries of its combined states. Consider the combined state for *b*, *e*, and *f*, which I'll call β. For inputs 00, *b* and *f* have the entry *b*,1 and *e* has a don't care. Since we don't care what the entry is for *e*, we'll make it *b*,1. Now all three entries match. Since *b* is combined into state β, the entry becomes β,1. We determine the other entries in the same way. We also combine *a*, *c*, *d*, *g*, and *h* into a single state, which I'll call α, and combine their entries in the same way. This gives us the revised state table shown in Figure 9.13 (a). The equivalent, revised state diagram is shown in Figure 9.13 (b).

State	00	01	11	10
α	α,0	α,0	α,0	β, 0
β	β,1	α,1	α,1	β, 1

(a)



(b)

Figure 9.13: Revised (a) state table, and (b) state diagram for the S-R latch.

Step 4: Assign Binary Values To States

Since we have only two states in our system (after minimization), we need only one bit to represent each state. I choose to assign $\alpha = 0$ and $\beta = 1$.

Step 5: Determine Functions and Create Circuit

From here we can create the transition table and excitation map for the revised states. Again using *q* as the present state and *Q* as the next state, and including the binary values for α and β, we get the transition table shown in Figure 9.14 (a).

S	*R*	*q*	*Q*	PSD state
0	0	0	0	*a*
0	0	1	1	*b*
0	1	0	0	*c*
0	1	1	0	*d*
1	0	0	1	*e*
1	0	1	1	*f*
1	1	0	0	*g*
1	1	1	1	*h*

(a)

*SR**q*	0	1
00	0	1
01	0	0
11	0	0
10	1	1

(b)

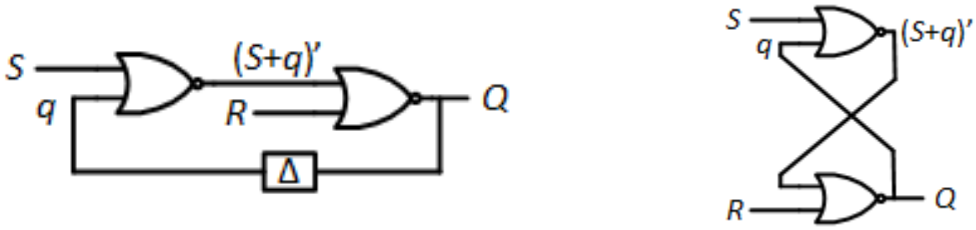
Figure 9.14: (a) State table, and (b) excitation map for the revised S-R latch.

As before, we can derive the excitation map from this state table. The excitation map is shown in Figure 9.14 (b). Both the state table and the excitation map are identical to those derived from the circuit for the S-R latch at the beginning of this section, which makes sense since we are now trying to develop that same circuit.

From this excitation map, it is straightforward to derive the function for *Q*, which is $Q = SR' + R'q$. A bit of logical manipulation transforms it as follows.

$$\begin{aligned} Q &= SR' + R'q \\ &= R'(S + q) \\ &= (R + (S + q)')' \end{aligned}$$

This in turn leads to the implementation using two NOR gates, one to generate $(S + q)'$ and the other to realize the final function. This circuit is shown in Figure 9.15 (a), with a delay block included to show the feedback generating *q* from *Q*. The circuit is redrawn more like a traditional S-R latch in Figure 9.15 (b)



(a) (b)

Figure 9.15: (a) Circuit to generate function *Q*. (b) Redrawn as a traditional S-R latch.

9.3 Unstable Circuits

Stability is an important consideration when designing asynchronous sequential circuits. We have previously introduced stable states, but what makes an entire circuit stable?

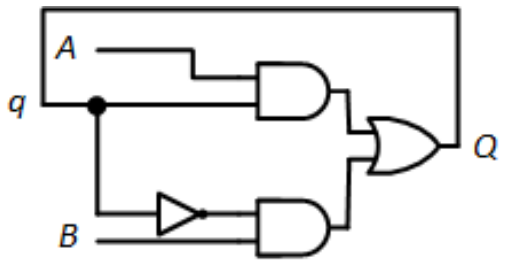
Simply put, an asynchronous sequential circuit is stable if it always reaches a stable state. It may pass through one or more transient states to reach a stable state, but it always gets there, no matter what value the inputs have and what state it is currently in.

This leads to an interesting and necessary condition. For each possible combination of input value, there must be at least one stable state. Furthermore, since input values do not change while the circuit is in a transient state, each transient state must ultimately transition to one of these stable states. If a set of input values does not have any stable states, this cannot happen, and the circuit is unstable.

Consider the circuit shown in Figure 9.16 (a). It is straightforward for us to find that the function for Q can be expressed as

$$Q = Aq + Bq'$$

We can use this function as we create the transition table shown in Figure 9.16 (b).



(a)

A	B	q	Q	PSD state
0	0	0	0	a
0	0	1	0	b
0	1	0	1	c
0	1	1	0	d
1	0	0	0	e
1	0	1	1	f
1	1	0	1	g
1	1	1	1	h

(b)

Figure 9.16: An unstable asynchronous sequential circuit: (a) Logic diagram; (b) Transition table.

You may already see the problem just by looking at the transition table, but it becomes much clearer when we create the excitation map for this circuit. This map is shown in Figure 9.17. Notice that the column for inputs AB = 01 has no stable states. If we input 01 to this circuit, it will constantly transition between two states. When AB = 01 and q = 0, it will set Q = 1. This value is fed back and, after a delay, q = 1. This leaves us with AB = 01 and q = 1, which sets Q = 0. This in turn sets q = 0, bringing us back to the original state. This bouncing back and forth between transient states is an **oscillation**.

q\AB	00	01	11	10
0	0	1	1	0
1	0	0	1	1

or

AB\q	0	1
00	0	0
01	1	0
11	1	1
10	0	1

Figure 9.17: Excitation map for the unstable asynchronous sequential circuit.

Using the state labels in the transition table, we can create the primitive state diagram shown in Figure 9.18. Notice the loop in the graph between states *c* and *d*. This is the same oscillation described in the previous paragraph.

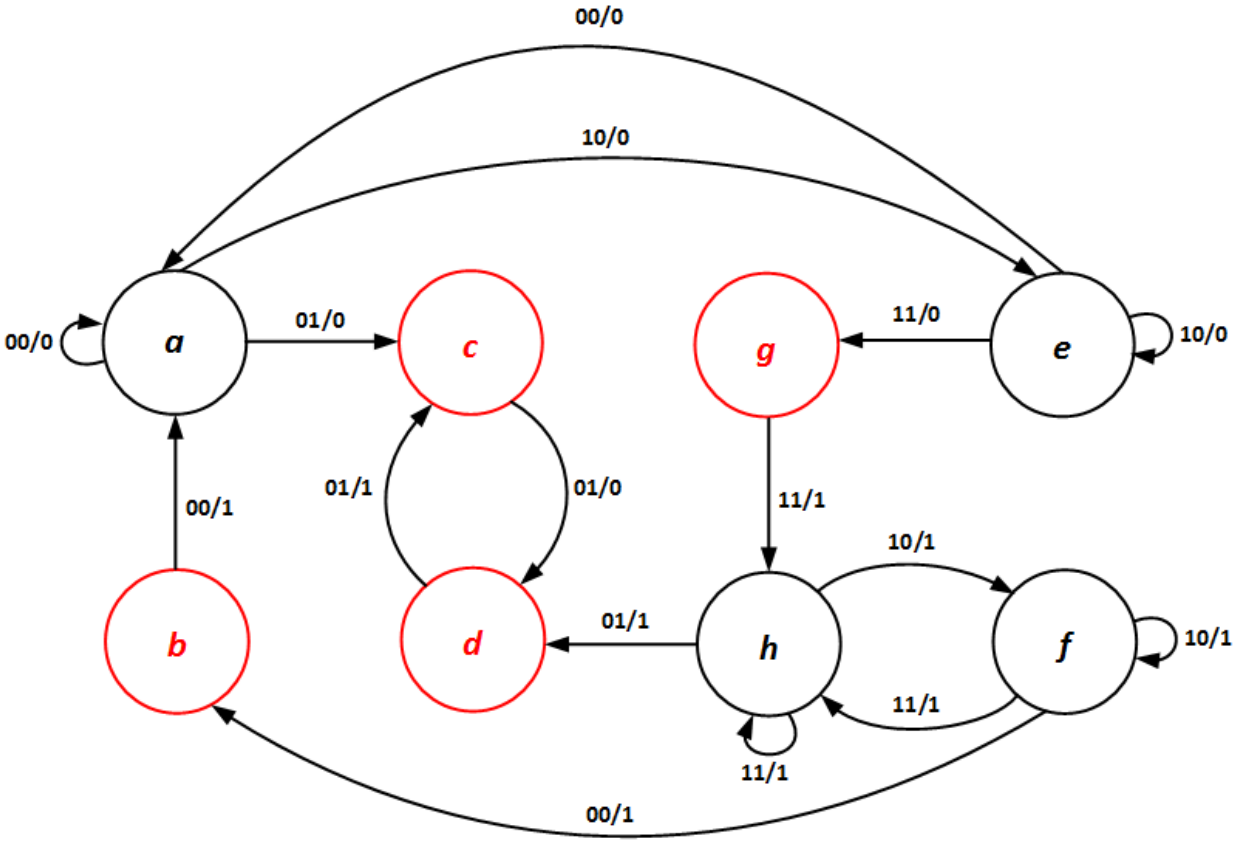


Figure 9.18: Primitive state diagram for the unstable asynchronous sequential circuit.

This process is useful during the analysis of existing circuits, as well as the initial design process. Once a function is determined, we can develop the transition table, excitation map, and primitive state diagram to identify unstable input conditions. However, there is no quick fix to this problem. If a specification is met and the solution leads to an unstable design, it is necessary to revise the initial specification to make the system fully stable.

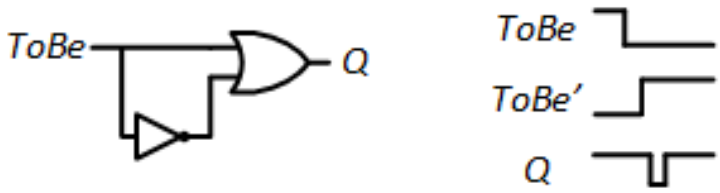
9.4 Hazards

In combinatorial circuits, a **hazard** occurs when a change to a single input value causes a momentary change to an output value that, logically speaking, should not occur. In this section, we'll look at two classes of hazards, how to identify them, and how to design circuits that are hazard-free.

9.4.1 Static Hazards

You have already seen an example of a hazard earlier in this book. See if you can recall this example before continuing.

The example is our Shakespeare-inspired circuit, which comes from the end of Chapter 3 in our discussion of propagation delays. (See *Hamlet*, Act III, Scene 1.) The circuit, its timing diagram, and its animation are repeated in Figure 9.19. When input *ToBe* changes from 1 to 0, inverted output *ToBe'* changes from 0 to 1, but only after the brief propagation delay of the NOT gate. During this delay, the OR gate may see both inputs as 0 and momentarily set its output to 0. This is called a **static-1 hazard**; it occurs when the output should remain at 1, but it glitches to 0.

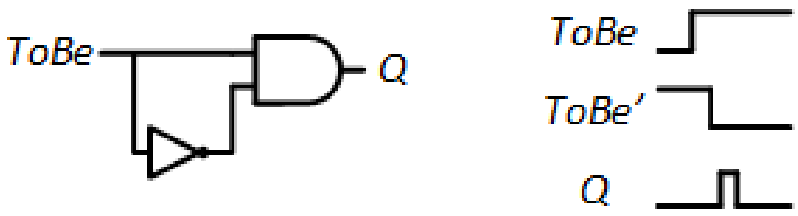


(a) (b)

Figure 9.19: Example of a static-1 hazard: (a) Circuit; (b) Timing diagram.

WATCH ANIMATED FIGURE 9.19

There is an equivalent hazard, called a **static-0 hazard**, that occurs when an output that should remain at 0 glitches to 1. An example circuit and timing diagram are shown in Figure 9.20. In this circuit, when input *A* changes from 0 to 1, the output of the NOT gate changes from 1 to 0 after a slight propagation delay. During the time the NOT gate is changing its output, the AND gate may interpret both inputs as 1, and thus output a 1 very briefly.



(a) (b)

Figure 9.20: Example of a static-0 hazard: (a) Circuit; (b) Timing diagram.

WATCH ANIMATED FIGURE 9.20

In general, static-1 hazards occur most frequently in AND-OR circuits, that is, circuits with several AND gates, each of which sends its output to a single OR gate to generate the circuit output. Static-0 hazards typically occur in OR-AND circuits. These are common circuit configurations. AND-OR circuits are frequently used to implement sum-of-products functions, and OR-AND circuits typically implement product-of-sums functions.

9.4.1.1 Recognizing Static-1 Hazards

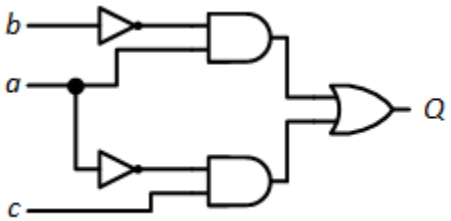
One of the easiest ways to identify static hazards is to examine their Karnaugh maps. Consider, for example, a function with the truth table shown in Figure 9.21 (a). We can use this table to create the Karnaugh map in Figure 9.21 (b). Grouping adjacent 1s as shown in the figure, we derive the function $Q = ab' + a'c$. Finally, we implement it as shown in Figure 9.21 (c). It looks good, but it is not; it has a static-1 hazard.

A	*B*	*q*	*Q*
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

(a)

*a**bc*	00	01	11	10
0	0	1	1	0
1	1	1	0	0

(b)



(c)

Figure 9.21: Example design with a static-1 hazard: (a) Truth table; (b) Karnaugh map; (c) Circuit.

The hazard occurs when $b = 0$, $c = 1$, and a changes from 1 to 0. Under these conditions, the upper AND gate changes from 1 to 0 and the lower gate changes from 0 to 1. However, because a passes through a NOT gate before going into the lower AND gate, the upper gate may

change its output to 0 before the lower gate changes its output to 1. Both gates will send a 0 to the OR gate for a brief time, causing it to generate a glitch output of 0. This sequence is shown in Figure 9.22.

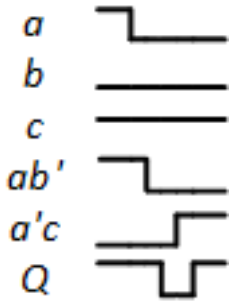


Figure 9.22: Timing diagram illustrating the static-1 hazard.

WATCH ANIMATED FIGURE 9.22

This is a key point in the root cause of static-1 hazards. These hazards occur when the value of two terms in the function change simultaneously, one from 0 to 1 and the other from 1 to 0. Due to different propagation delays, however, the two terms do not actually change at *exactly* the same time. There is a slight difference, and this difference causes the glitch.

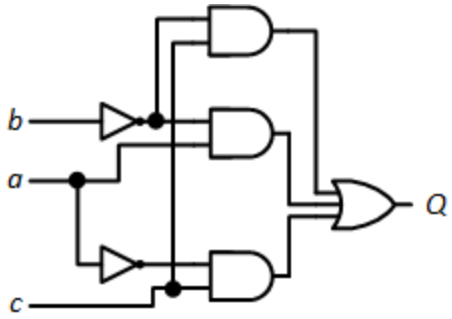
In a Karnaugh map, each circled group corresponds to one of the AND gates. When changing only one input causes the circuit to move from one group to another in the Karnaugh map, we are changing the outputs of two of the AND gates. The AND gate for the group we are leaving changes from 1 to 0, and the AND gate for the group we are entering changes from 0 to 1. So, to find static-1 hazards in a Karnaugh map, look for adjacent 1s that are in different groups. In this map, this occurs in the cells with *abc* = 001 and *abc* = 101.

9.4.1.2 Resolving Static-1 Hazards

Now that we have identified the hazard, how do we get rid of it? Think about this for a minute before reading on.

Hopefully you did think about this, and hopefully you found the solution. The way to get rid of the static-1 hazard is to add another term to our function and another group to the Karnaugh map. The new group should include all the terms that cause the glitch to appear. For our function, this occurs when $b = 0$ and $c = 1$, and a is either 0 or 1. Our additional term is $b'c$. The revised Karnaugh map and circuit are shown in Figure 9.23.

a\bc	00	01	11	10
0	0	1	1	0
1	1	1	0	0



(a) (b)

Figure 9.23: (a) Karnaugh map and (b) circuit updated to remove the static-1 hazard.

Now when the inputs change from *abc* = 001 to *abc* = 101, the output of the new, uppermost AND gate remains at 1. The output of this gate is input to the OR gate, which also keeps its output at 1, removing the glitch. Figure 9.24 shows the updated timing diagram.

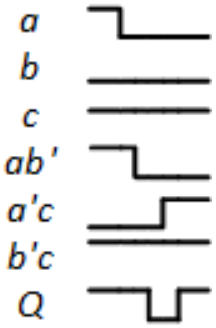


Figure 9.24: Timing diagram showing the static-1 hazard has been resolved.

WATCH ANIMATED FIGURE 9.24

9.4.1.3 Recognizing and Resolving Static-0 Hazards

Just as we can recognize static-1 hazards in a Karnaugh map, we can also recognize static-0 hazards using K-maps. We form groups of the 0 terms; the sum of these is *Q*′. If any of the terms is true, then *Q* = 0 and *Q*′ = 1.

For our previous example, we can group terms as shown in Figure 9.25. The two groups are *ab* and *a*′*c*′.

a\bc	00	01	11	10
0	0	1	1	0
1	1	1	0	0

Figure 9.25: Karnaugh map with zeroes grouped.

We can apply DeMorgan's laws to generate a product of sums formula for Q as follows.

$$
\begin{aligned}
Q' &= ab + a'c' \\
Q &= (ab + a'c')' \\
&= (ab)'(a'c')' \\
&= (a' + b')(a + c)
\end{aligned}
$$

We can implement this function for Q using the circuit shown in Figure 9.26.

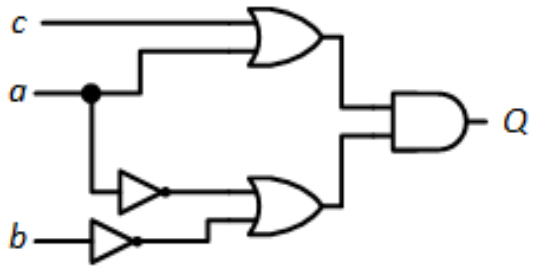


Figure 9.26: Circuit with static-0 hazard.

Static-1 hazards occur when moving from one group to another, and the same is true for static-0 hazards. In this example, this occurs when $b = 1$ and $c = 0$, and a changes from 0 to 1. The output of the upper OR gate changes from 0 to 1, and the output of the lower OR gate changes from 1 to 0, but only after a slight propagation delay due to the NOT gate that outputs a'. During this delay, the AND gate may see both inputs as 1 and very briefly change its output to 1, as shown in the timing diagram in Figure 9.27.

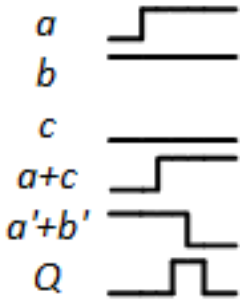


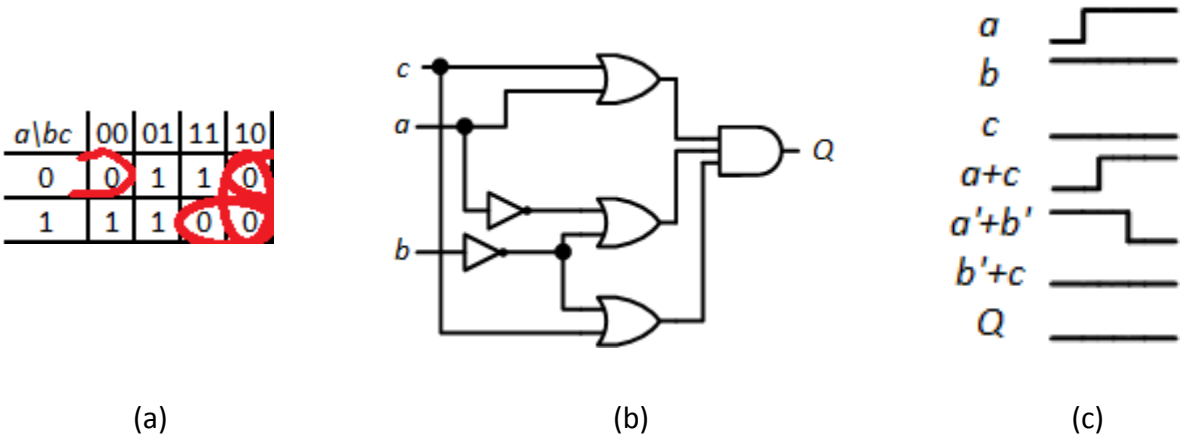
Figure 9.27: Timing diagram showing the static-0 hazard.

WATCH ANIMATED FIGURE 9.27

So far, everything presented seems analogous to the static-1 hazard. The solution continues this trend. To remove the static-0 hazard, we add a term to cover the terms with the hazard, in this case bc'. Our equations become

$$Q' = ab + a'c' + bc'$$
$$Q = (ab + a'c' + bc')'$$
$$= (ab)'(a'c')'(bc')'$$
$$= (a' + b')(a + c)(b' + c)$$

The Karnaugh map, revised circuit, and new timing diagram are shown in Figure 9.28. As we can see in the timing diagram and animation, the additional OR gate outputs a 0 the entire time that the circuit transitions from abc = 010 to abc = 110. This ensures that the output of the AND gate remains at 0, removing the glitch.



(a) (b) (c)

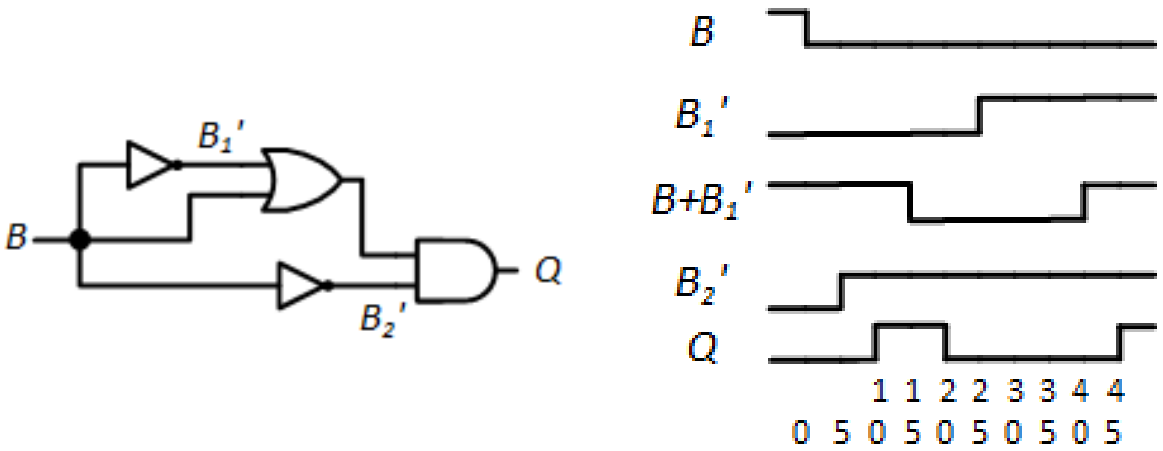
Figure 9.28: (a) Karnaugh map; (b) updated circuit; and (c) timing diagram with static-0 hazard removed.

WATCH ANIMATED FIGURE 9.28

9.4.2 Dynamic Hazards

Static hazards occur when an output that is (logically) not supposed to change experiences a glitch. It is also possible to have a glitch when an output *is* supposed to change. An output that should change from 0 to 1, for example, might have a glitch and change from 0 to 1, back to 0, and finally back to 1. This is called a **dynamic hazard**.

To see how a dynamic hazard can occur, consider the circuit shown in Figure 9.29 (a). The OR gate realizes the function $B + B'$. In theory, this value should always be 1. In practice, however, it has a static-1 hazard and can glitch to 0 when B changes from 1 to 0, as described in the previous subsection. The AND gate realizes the function $(B + B')B'$, or B'.



(a) (b)

Figure 9.29: (a) Circuit with a dynamic hazard, and (b) its timing diagram.

WATCH ANIMATED FIGURE 9.29

To distinguish the outputs of the two inverters, I've labeled them B_1' and B_2'. Both will output the value B', but there is no guarantee that both will take *exactly* the same amount of time to produce this value. In this example, the lower NOT gate generates B_2' more quickly than the upper NOT gate outputs B_1'. The lower NOT gate and the AND gate have 5 ns propagation delays, the OR gate has a propagation delay of 15 ns, and the propagation delay of the upper NOT gate is 25 ns.

The timing diagram in Figure 9.29 (b) and the animation for this figure show the signal values throughout the circuit as B changes from 1 to 0. Initially, $B = 1$, $B_1' = 0$, and the output of the OR gate ($B + B'$) = 1. The output of the lower NOT gate, B_2', is 0 and $Q = 0$.

Then B changes from 1 to 0. 5 ns later, B_2' changes from 0 to 1. This sets both inputs of the AND gate to 1. After an additional 5 ns to accommodate the propagation delay of the AND gate, its output becomes 1.

After another 5 ns, or 15 ns after B changes from 1 to 0, the OR gate sets its output to 0 since its B and B_1' inputs are both 0. This sets one input of the AND gate to 0, and 5 ns later its output goes back to 0.

5 ns later, or 25 ns after B changes, the output of the upper NOT gate, B_1', becomes 1. After its 15 ns propagation delay, the output of the OR gate changes from 0 to 1. Now the inputs to the AND gate are again all set to 1, and 5 ns later the AND gate again outputs the final value of 1. This changing of the output of the AND gate, output Q, from 0 to 1 to 0 to 1 is the dynamic hazard in this circuit.

Now that we know where the dynamic hazard is within the circuit, how do we get rid of it? Think about this for a minute, but here's a hint: you've already seen the solution to this problem.

There is a specific condition of dynamic hazards that greatly simplifies the process of correcting them. *A dynamic hazard can only exist if the circuit also includes at least one static hazard.* In this circuit, the OR gate has a static-1 hazard that causes it to have a glitch in its output. This glitch is input to the AND gate, which causes its output to glitch and generates the dynamic hazard.

Putting this all together, if we get rid of the glitch generated by the static-1 hazard, we no longer have an input glitch to trigger the dynamic hazard. That's the key to mitigating dynamic hazards: get rid of the static hazards in the circuit and the dynamic hazards disappear.

9.5 Race Conditions

The hazards we introduced in the previous section are caused when a single input value changes. Within a circuit, there is another condition that can cause problems. This condition occurs when two (or more) values within a circuit are supposed to change simultaneously. It is very unlikely that the two values will change at *exactly* the same instant in time. In general, one may change slightly more quickly than the others, placing the circuit in a state that it should not be in. This is called a **race condition**.

To illustrate this, consider an asynchronous sequential circuit that continuously outputs two-bit values in the sequence 00 → 01 → 10 → 11 → 00… This circuit has four states with the same values as the outputs. Now look at the point where the state (and outputs) change from 01 to 10. If both bits change at exactly the same time, the circuit functions as desired. But what happens if the most significant bit changes slightly faster than the least significant bit? In this case, the circuit may go from 01 to 11 instead of from 01 to 10. If the least significant bit changes more quickly, the circuit might go from 01 back to 00. This, in a nutshell, is the issue arising from race conditions.

I was careful to specify that this is an *asynchronous* circuit. In synchronous circuits, race conditions typically occur in the combinatorial logic used to generate the inputs to flip-flops within the circuit. To resolve race conditions, we simply wait until all values have changed before loading the flip-flops. We can do this by reducing the frequency (and hence increasing the period) of the clock. For this reason, the remainder of this section will focus solely on race conditions in asynchronous sequential circuits.

Some race conditions don't matter, and some do. In the following two subsections, we'll look at these two cases. Then we'll examine methods to resolve race conditions when necessary.

9.5.1 Non-critical Race Conditions

When analyzing asynchronous sequential circuits, we are primarily concerned with the stable states. If a race condition exists, but all possible paths always lead to the desired stable state, the race condition is called a **non-critical race condition**. To see how this works, consider the example transition table shown in Figure 9.30. The circuit has four states, each represented by a 2-bit value: *a* (00), *b* (01), *c* (10), and *d* (11). Initially, the circuit is in state *a* (00) and the inputs are set to 11; this is a stable state for this input value. Now the input changes to 01 and we want our circuit to transition to state *d* (11). If both bits representing the state change from 0 to 1 at

the same instant in time, the circuit goes directly to state *d* as desired. This is possible but very unlikely.

State	00	01	11	10
00/*a*	*b* ,1	*d* ,1	*a* ,1	*c* ,1
01/*b*	*b* ,0	*d* ,0	*a* ,0	*c* ,0
10/*c*	*b* ,1	*d* ,1	*a* ,1	*c* ,1
11/*d*	*b* ,0	*d* ,0	*a* ,0	*c* ,0

Figure 9.30: Transition table with non-critical race conditions.

WATCH ANIMATED FIGURE 9.30

Let's examine the two cases in which one bit of the state value changes more quickly than the other. If the most significant bit is faster, the circuit goes from state value 00 (*a*) to 10 (*c*). But looking at state *c* for input value 01, we see that the circuit next goes to state *d* (11). Since the most significant bit for both *c* and *d* is 1, it doesn't change any more. At this point, only the least significant bit changes from 0 to 1. With only one bit changing, the state value goes from 10 (*c*) to 11 (*d*), thus ending up in the final, stable state that we wanted it to go to.

If the least significant bit changes more quickly, the circuit instead goes from state *a* (00) to state *b* (01). State *b* also transitions to state *d* (11) with only the most significant bit in its state value changing.

No matter which bit of the state value (if either) changes more quickly, the circuit always ends up in state *d*. This is why this race condition is non-critical. There are also some other non-critical race conditions in this table. Identifying these race conditions is left as an exercise for the reader.

9.5.2 Critical Race Conditions

It would make our lives much easier if all races in asynchronous sequential circuits were non-critical, but that's just not the case. Frequently, races cause the circuit to go into the wrong stable state, and not function as desired. These races are called **critical races**.

The transition table shown in Figure 9.31 includes a critical race. Let's say the circuit is in state *a* (00) and the input value is 0; the circuit is currently in a stable state. Then the input value changes from 0 to 1. The circuit should transition to state *d* (11). If both bits of the state change simultaneously, unlikely but possible, this is exactly what happens. The circuit goes directly from state *a* to state *d*.

State	0	1
00/*a*	*a*,1	*d*,1
01/*b*	*a*,0	*b*,1
10/*c*	*c*,0	*d*,1
11/*d*	*c*,1	*d*,0

Figure 9.31: Transition table with a critical race condition.

WATCH ANIMATED FIGURE 9.31

Next, consider the case in which the most significant bit changes more quickly than the least significant bit. Instead of going directly to state *d* (11), the state value changes from 00 to 10, bringing the circuit to state *c*. Fortunately, when the circuit is in state *c* and the input is 1, it transitions from *c* (10) to *d* (11), which is the stable state we wanted to reach. So far the circuit is functioning as desired.

Finally, we see what happens if the least significant bit changes first. We go from state *a* (00) to state *b* (01). When the input is 1, this is a stable state and the circuit remains in this state. This is not where it should be, and this is a critical race.

We need to modify this system so that all critical races are removed. We look at how to do that in the next subsection.

9.5.3 Resolving Critical Race Conditions

As we've seen in the previous examples, both non-critical and critical race conditions occur when two or more bits of the state value change at the same time, but one may change more quickly than the other(s). When this happens, the circuit may enter an incorrect state, and thus fail to perform as required. The best way to get rid of races in an asynchronous sequential circuit is to assign state values so that each state transition only changes one bit of the state value. Just like a horse race with only one horse,[1] there isn't really a race with only one bit changing; there isn't anything for it to race against.

Going back to the last example, let's swap the values for states *c* and *d*. This makes our state representations *a* = 00, *b* = 01, *c* = 11, and *d* = 10. When our circuit is in state *a* and the input value changes from 00 to 10 instead of from 00 to 11. For the new state values, this transition only changes one bit, which removes the race condition. Equally important, it does not introduce any other race conditions.

It is important to note that changing the state value does not change the output values. The circuit must still generate the outputs as originally specified. If the outputs were derived from the original state values, then we would need to redesign the logic to generate outputs based on the new state values. Remember that whoever is using the circuit does not see the states; they only see the input and output values.

[1] Technically, it is possible to have a horse race with only one horse. This happens when all but one horse is scratched, withdrawn from the race, usually due to illness or injury. A one-horse race is called a walkover.

9.5.3.1 Another Example

We want to design an asynchronous sequential circuit that counts the number of times an input value changes. For simplicity, the circuit outputs a 2-bit value, O_1O_0, and a single input I. When I changes either from 0 to 1 or from 1 to 0, the output value is incremented, progressing through the sequence 00 ⟶ 01 ⟶ 10 ⟶ 11 ⟶ 00…

Since there are four different output values, this circuit needs (at least) four states. For this example, the four states and their state values are *a* (00), *b* (01), *c* (10), and *d* (11). For the initial design, the output values are the same as the state values. Initially, the circuit is in state *a* and input I = 0. When I changes to 1, the circuit goes to state *b*. It stays in *b* until I = 0; then it goes to *c*. When I becomes 1, the circuit transitions to state *d*, and when I is again 0, the circuit moves to state *a*. Before looking at the transition table in Figure 9.32, try to derive this table from the description given in this paragraph.

State	0	1
00/*a*	*a* ,00	*b* ,00
01/*b*	*c* ,01	*b* ,01
10/*c*	*c* ,10	*d* ,10
11/*d*	*a* ,11	*d* ,11

(a)

State	0	1
00/*a*	00,00	01,00
01/*b*	10,01	01,01
10/*c*	10,10	11,10
11/*d*	00,11	11,11

(b)

Figure 9.32: Transition table for the counter: (a) With symbolic state names; (b) With the initial binary state values.

It is relatively easy to determine the functions for the two-bit "next states" Q_1 and Q_0. We can see just by looking at the table that $Q_0 = I$. Q_1 is best determined using an excitation map. Its function is $q_1q_0' + q_1I + q_1'q_0I'$. The circuit to realize these functions, and the entire asynchronous sequential machine, is shown in Figure 9.33. Because we chose these specific state assignments, the outputs can be expressed as $O_1 = Q_1$ and $O_0 = Q_0$.

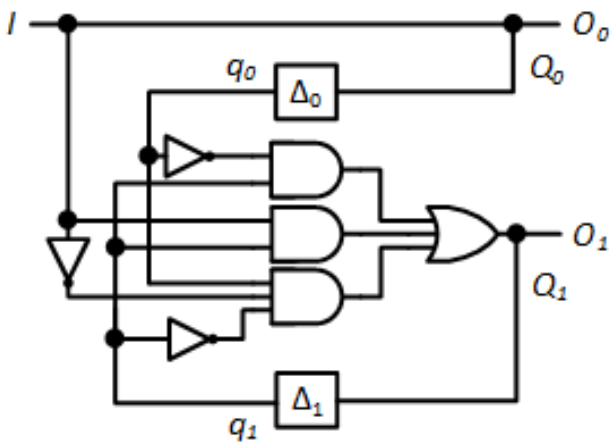


Figure 9.33: Preliminary circuit to realize the counter.

Unfortunately, this circuit has some critical races. Consider the case when the circuit is in state *b* and input *I* changes from 1 to 0. We want the circuit to transition from state *b* to state *c*, with the state value changing from 01 to 10. This might happen if both bits change at exactly the same time. If the least significant bit changes more quickly than the most significant bit, which is realistic since it does not use any logic gates at all, the circuit would transition to state value 00, or state *a*. This is a stable state and the circuit would stay there, not reaching the desired state *c*. If, for some reason, Q_1 changes first, the circuit would go to state *d* with state value 11. From here, it would try to go to state *a* (00), and could go to that state. It could also go to state *b* (01), which is where the circuit just was. There is also a chance that it will go to desired state *c* (10), but this is by no means certain.

Once again, we can remove the race by setting the state values to *a* (00), *b* (01), *c* (11), and *d* (10). Figure 9.34 shows the state table with these values. Using the excitation maps, the development of which is left as an exercise for the reader, we derive the equations for state variables Q_1 and Q_0 as:

$$Q_1 = q_0 I' + q_1' I$$
$$Q_0 = q_0 I' + q_1 I$$

State	0	1
00/*a*	*a*,00	*b*,00
01/*b*	*c*,01	*b*,01
11/*c*	*c*,10	*d*,10
10/*d*	*a*,11	*d*,11

(a)

State	0	1
00/*a*	00,00	01,00
01/*b*	11,01	01,01
11/*c*	11,10	10,10
10/*d*	00,11	10,11

(b)

Figure 9.34: Transition table for the counter: (a) With symbolic state names; (b) With revised binary state values.

Since the outputs are no longer the same as the state values for states *c* and *d*, we need to develop functions for the outputs. The reader can verify that these functions are

$$O_1 = Q_1$$
$$O_0 = Q_1 \oplus Q_0$$

The circuit for the counter with the revised state values is shown in Figure 9.35. This design has no race conditions.

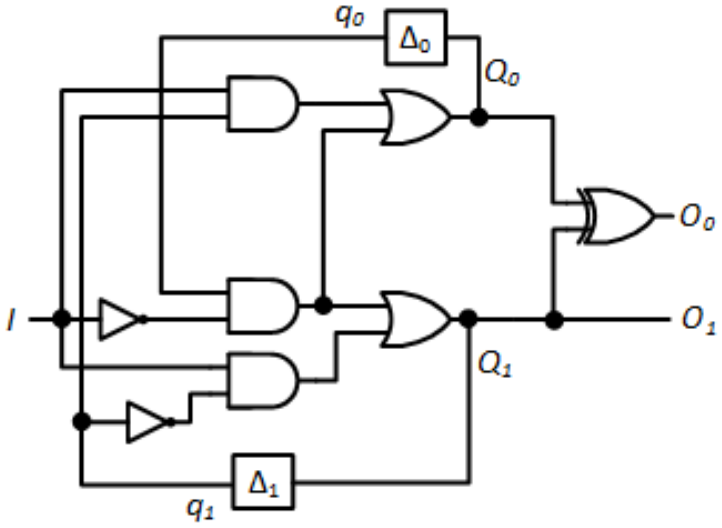


Figure 9.35: Final circuit to realize the counter.

9.5.3.2 How Did We Do That?

So, how did I know that exchanging the state values for *c* and *d* would get rid of the critical races? There is a specific method I used, which I'll explain here. First, I looked at each row of the transition table to see which states are adjacent, that is, which states are involved in a transition. For state *a*, we stay in *a* if *I* = 0 and go to *b* if *I* = 1. Hence, *a* and *b* are adjacent. We ignore the cases when the circuit remains in the same state since the state values do not change. Continuing through the transition table row by row, we find that *b* and *c*, *c* and *d*, and *d* and *a* are also adjacent.

To remove races from the circuit, we want to assign values to each state so that adjacent states' values vary by only one bit. I started by assigning 00 to *a*.

State *a* is adjacent to two states, *b* and *d*. There are two values that are different from the 00 value of state *a* by only one bit, 01 and 10. I assigned 01 to *b* and *10* to *d*.

Now, state *b* is adjacent to states *a* and *c*. We've already assigned 00 to *a*, and there is only one other binary value that differs from the 01 value of state *b* by one bit. We assign this value, 11, to state *c*. Finally, we see that the values of adjacent states *c* (11) and *d* (10) also vary by only one bit.

Notice that the counter progresses from $a \rightarrow b \rightarrow c \rightarrow d \rightarrow a\ldots$, or through state values $00 \rightarrow 01 \rightarrow 11 \rightarrow 10 \rightarrow 00\ldots$ This is the 2-bit reflected Gray code introduced earlier in this book, and this is one of the most common uses of the Gray code, to get rid of races in asynchronous sequential circuits.

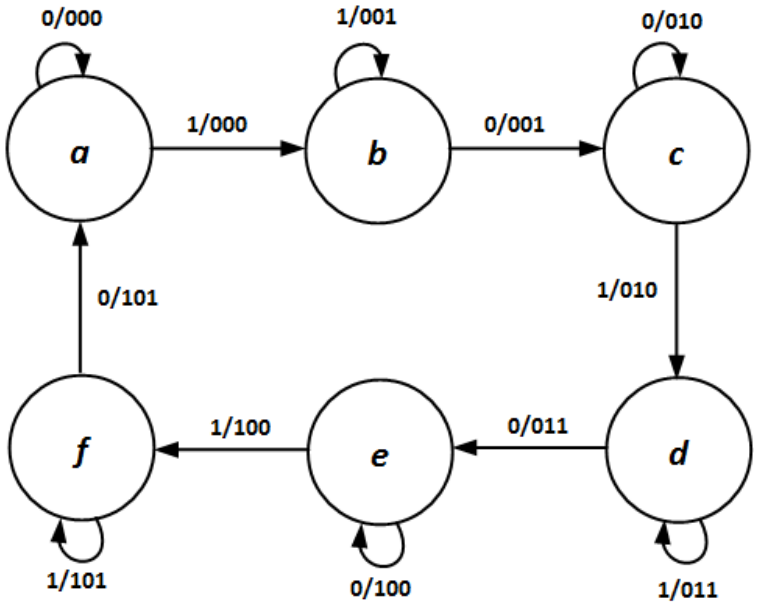
Gray codes work great when the number of states is exactly a power of 2 and you progress through the states sequentially. But that's not always the case. Consider the case when we want to extend this circuit so it counts from 0 to 5 instead of 0 to 3. It will have six states, which we label *a*, *b*, *c*, *d*, *e*, and *f*. We take the first six entries of the 3-bit Gray code and assign them to these states, which gives us *a* = 000, *b* = 001, *c* = 011, *d* = 010, *e* = 110, and *f* = 111. We can construct the transition table in much the same way as we did for the original example. This gives us the transition table and primitive state diagram shown in Figure 9.36.

State	0	1
000/*a*	*a*,000	*b*,000
001/*b*	*c*,001	*b*,001
011/*c*	*c*,010	*d*,010
010/*d*	*e*,011	*d*,011
110/*e*	*e*,100	*f*,100
111/*f*	*a*,101	*f*,101

(a)

State	0	1
000/*a*	000,000	001,000
001/*b*	011,001	001,001
011/*c*	011,010	010,010
010/*d*	110,011	010,011
110/*e*	110,100	111,100
111/*f*	000,101	111,101

(b)



(c)

Figure 9.36: 6-value counter: (a) Primitive transition table; (b) Primitive transition table with state values listed; and (c) Primitive state diagram.

As we transition through the states of this asynchronous sequential system, using the Gray code sequence largely achieves its purpose. Almost all transitions change only one bit of the state value. The only transition that causes a problem occurs in state *f* when input *I* = 0. There, the circuit transitions from state *f* (111) to state *a* (000). All three bits of the state value change, giving the circuit a three-way race.

Before reading on, try to list some ways to resolve this problem.

Hopefully you've developed some alternatives to the design presented here. Below are a few that I found.

1. Change the state value for *f* from 111 to 100.
2. Change the state values for *d*, *e*, and *f* from 010, 110, and 111 to 111, 110, and 100, respectively.
3. Add transient states between states *f* and *a*.

For this example, we'll implement the third option.

The idea behind adding transient states is to have this system change only one bit of the state value for each transition. By incorporating transient states, the system can change one bit of the state value and then transition to another state very quickly, continuing until all bits are changed.

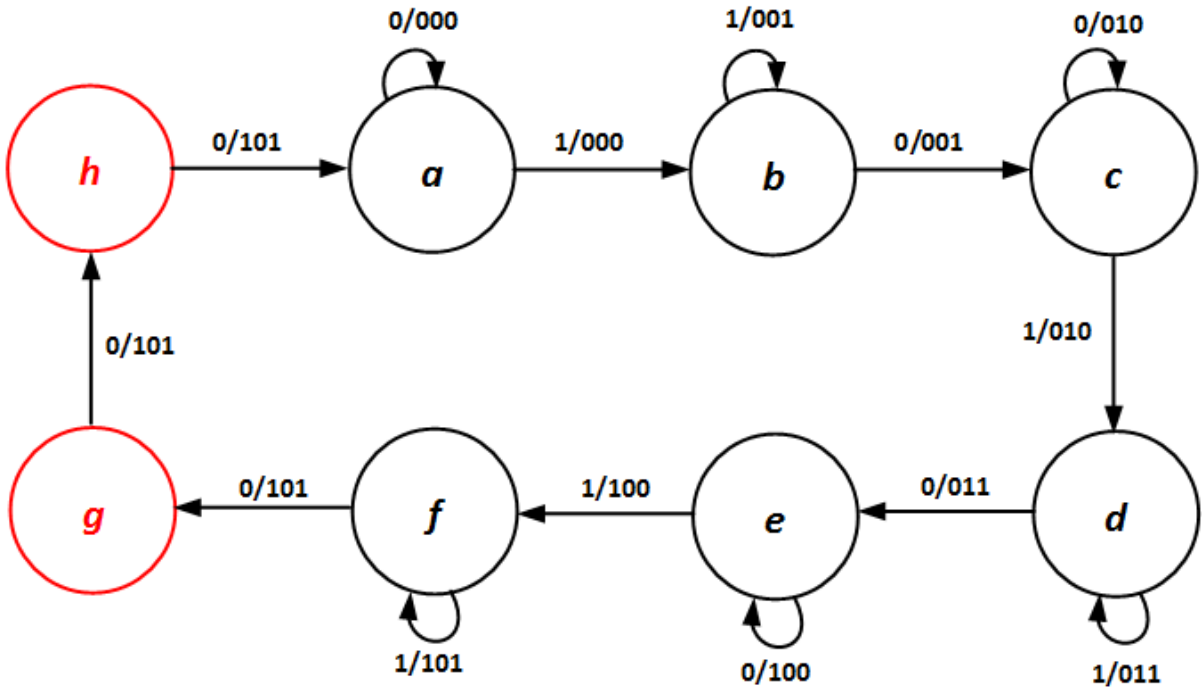
Since three bits are changed as the system goes from state *f* to state *a*, this transition must be divided into three transitions, each of which changes one bit of the state value. To do this, we create two new transient states, *g* and *h*, with state values 101 and 100, respectively. Instead of going from *f* (111) to *a* (000), the system will go from *f* (111) to *g* (101) to *h* (100) to *a* (000). Now, each transition changes only one bit of the state value and the critical race condition is removed. The revised transition table and state diagram are shown in Figure 9.37. Note the two entries for *I* = 1 in states *g* and *h*. Since these are transient states, only the entries for *I* = 0 are defined in this design. The entries for *I* = 1 are treated as don't cares.

State	0	1
000/*a*	*a* ,000	*b* ,000
001/*b*	*c* ,001	*b* ,001
011/*c*	*c* ,010	*d* ,010
010/*d*	*e* ,011	*d* ,011
110/*e*	*e* ,100	*f* ,100
111/*f*	*g* ,101	*f* ,101
101/*g*	*h* ,101	---
100/*h*	*a* ,101	---

(a)

State	0	1
000/*a*	000,000	001,000
001/*b*	011,001	001,001
011/*c*	011,010	010,010
010/*d*	110,011	010,011
110/*e*	110,100	111,100
111/*f*	101,101	111,101
110/*e*	100,100	---
111/*f*	000,101	---

(b)



(c)

Figure 9.37: Revised (a) transition table; (b) transition table with state values listed; and (c) state diagram for the 6-value counter.

With the revised states, we can now design the asynchronous, race-free sequential circuit for this system. The circuit will sequence through the states properly and generate the correct outputs. This is left as an exercise for the reader.

9.6 Summary

The vast majority of sequential circuits are synchronous, that is, they use a clock to coordinate the flow of data within the circuit. Asynchronous sequential circuits, though less frequently used, still play an important role when even the fastest synchronous circuits are not fast enough.

Unlike synchronous circuits, asynchronous sequential circuits do not use storage elements such as flip-flops to store their present state. They incorporate feedback delays as they transition from one state to another.

There are several tools we can use to design and analyze asynchronous sequential circuits. A transition table is similar to a traditional truth table, with system inputs and the present state as inputs and system outputs and the next state as outputs. An excitation map is derived from the transition table. It looks much like a Karnaugh map, with stable entries circled. A primitive state diagram is very much like a traditional state diagram, incorporating constraints placed on the system.

The design process for asynchronous sequential systems begins with the specification of the system and definition of all states. This preliminary specification often results in a large number of states, which can often be combined to reduce the final number of states in the system. We then determine functions and create the final circuit to realize the system specification.

Asynchronous sequential circuits can have several issues that do not occur, or occur less often, in synchronous sequential circuits. A circuit that does not have at least one stable state for every possible combination of input value may oscillate, transitioning continuously between two or more transient states. Static hazards occur when an output value that should stay constant briefly glitches to the opposite value. Static glitches occur when going from one group of terms in a Karnaugh map to another group. Incorporating redundant logic is the usual way to remove static hazards from a circuit. Dynamic hazards occur when a value that is supposed to change once has a glitch and changes several times on its way to its intended value. Dynamic hazards are caused by static hazards. Removing the static hazards also removes the dynamic hazards they cause.

Race conditions occur when two values are supposed to change simultaneously, but the delays associated with the changes are not equal. A race condition is non-critical when the race does not change the overall system behavior; the circuit ultimately reaches its desired state and generates its specified output values. A critical race condition can result in a circuit reaching an incorrect state and outputting the wrong values. We can resolve critical race conditions by assigning binary values to states such that each transition changes only one bit, thus giving that bit nothing to race against. It is also possible to incorporate additional, transient states into the system to limit the number of bits that change to one per transition.

Acknowledgment

Thanks to Professor Jacob Savir for his notes on asynchronous state machines, hazards, and race conditions. They were especially helpful when writing this chapter.

Bibliography

Hayes, J. P. (1993). *Introduction to Digital Logic Design* (First Edition). Addison-Wesley.

Mano, M. M., & Ciletti, M. (2017). *Digital design: with an introduction to the Verilog HDL, VHDL, and SystemVerilog* (6th ed.). Pearson.

Nelson, V., Nagle, H., Carroll, B., & Irwin, D. (1995). *Digital logic circuit analysis and design*. Pearson.

Roth, J. C. H., Kinney, L. L., & John, E.B. (2020). *Fundamentals of logic design enhanced edition* (7th ed.). Cengage Learning.

Savir, J. (2019). *Digital Design: ECE 251 Notes*, New Jersey Institute of Technology.

Shakespeare, William. (1623). *Mr. William Shakespeare's Comedies, Histories, & Tragedies [The First Folio]—Harry Ransom Center Digital Collections*. Retrieved August 1, 2023, from https://hrc.contentdm.oclc.org/digital/collection/p15878coll70

Exercises

For problems 1-5, analyze the S-R latch modified to only output $\overline{Q}$.

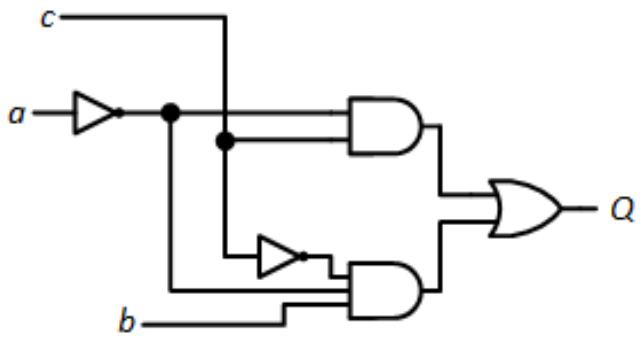
1. Show the circuit model with feedback and delay.
2. Show the transition table.
3. Show the excitation map.
4. Show the primitive state diagram.
5. Show the timing diagram for the S-R latch as inputs transition from 10 to 00 to 01.

For problems 6-9, design the D latch as an asynchronous sequential circuit.

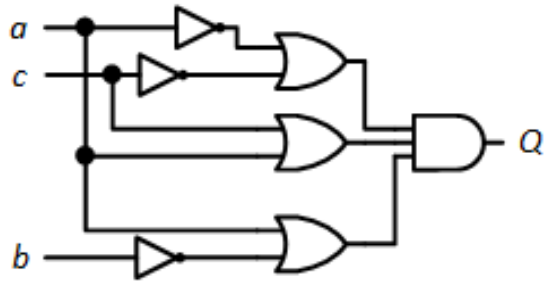
6. Show the transition table and define the primitive states.
7. Minimize the states and show the revised transition table.
8. Show the excitation map and derive the functions.
9. Design the circuit to implement this design.
10. For the transition table below, identify and correct all oscillations.

State	00	01	11	10
a	*b*,0	*d*,1	*a*,1	*c*,0
b	*c*,0	*b*,1	*c*,1	*c*,0
c	*d*,0	*c*,1	*c*,1	*c*,0
d	*d*,1	*a*,1	*a*,1	*b*,0

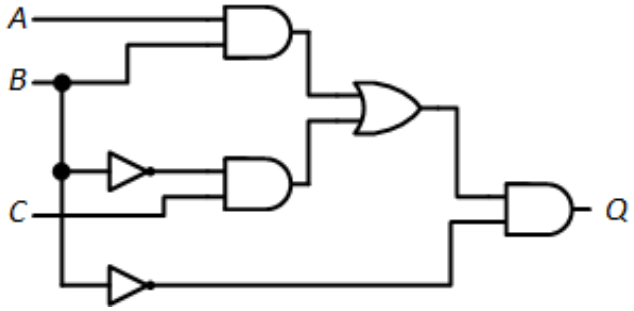
11. For the following circuit, identify and mitigate all static-1 hazards.



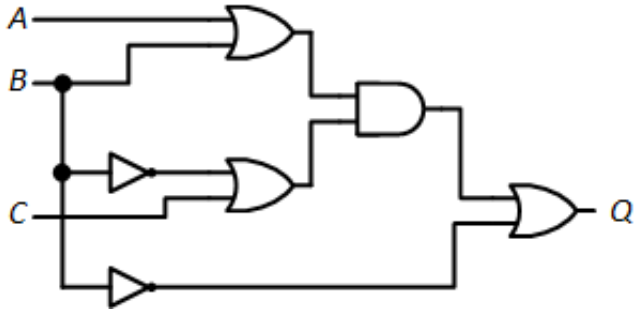
12. For the following circuit, identify and mitigate all static-0 hazards.



13. Find and resolve all dynamic hazards in the following circuit.



14. Find and resolve all dynamic hazards in the following circuit.



15. Find all non-critical race conditions in the transition table shown in Figure 9.30.

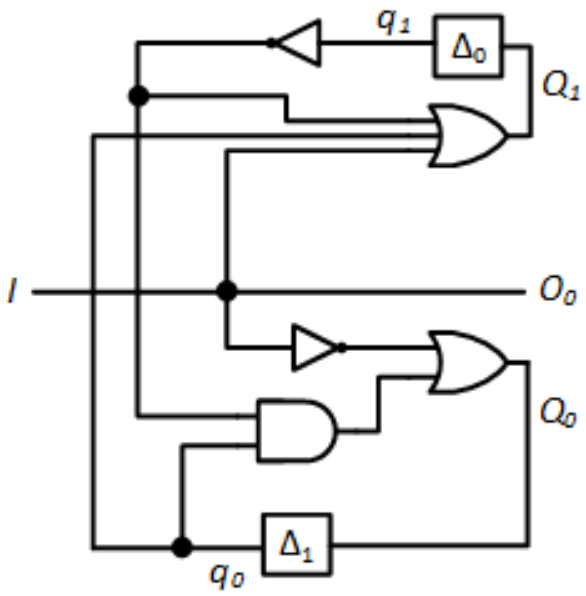
16. Develop the excitation maps for the transition table in Figure 9.34.

17. For the following transition table, identify all race conditions and indicate whether they are critical or non-critical.

State	00	01	11	10
00/*a*	*a*,0	*b*,1	*d*,1	*a*,0
01/*b*	*c*,1	*b*,1	*d*,1	*a*,0
10/*c*	*c*,1	*d*,1	*d*,1	*d*,0
11/*d*	*c*,0	*d*,1	*d*,1	*a*,0

18. Resolve all critical race conditions in the transition table of the previous problem.

19. For the following circuit, identify all race conditions and indicate whether they are critical or non-critical.



20. Resolve all critical race conditions in the circuit of the previous problem.

21. Redesign the six-state circuit in Section 9.5.3 using the first option for reassigning state values.

22. Redesign the six-state circuit in Section 9.5.3 using the second option for reassigning state values.

23. Design the circuit to complete the six-state circuit design at the end of Section 9.5.3.

Chapter 10

Programmable Devices

Chapter 10: Programmable Devices

Almost all of the devices presented so far in this book have static configurations. An AND gate has some set number of inputs and only sets its output to 1 when all its inputs are 1. There are AND gates with different numbers of inputs, but for any particular AND gate the number of inputs does not change. A 2-input AND gate doesn't suddenly become a 3-input AND gate. Nor does it change its function, changing, for example, into a 2-input OR gate. Once the gate is fabricated, it stays that way.

There are a couple of types of components, however, that are not static. These are **programmable devices**. We can program these devices so they output whatever values we want for specific input values, or they realize different functions for different inputs. A designer can change how a circuit functions by programming one of these chips. This chapter begins by examining the programmable device methodology and its advantages and disadvantages.

Next, we introduce **programmable logic devices**, or **PLDs**. Unlike memory devices, which store data values, PLDs incorporate a large number of logic gates (and sometimes flip-flops) on a single chip. We do not program the gates themselves; they are fixed and do not change. Rather, we program the interconnections between gates. By programming these data paths, we connect gate outputs to the inputs of other gates, or chip outputs, and chip inputs to gate inputs. In essence, this is like wiring up our circuit within the chip. Making these connections creates a circuit to realize any desired function.

Finally, we examine programmable memory devices. We saw one type of memory device, read-only memory, when we introduced lookup ROMs in Chapter 5. In this chapter, we introduce several types of memory devices, as well as their internal organization. We discuss some of the uses for these devices in digital logic circuits, computer systems, and inside of a microprocessor.

10.1 Why Programmable Devices?

Programmable logic devices and programmable memory chips offer a number of advantages when used to construct combinatorial and sequential logic circuits, and are used frequently in engineering design. Quite often, a single programmable chip can replace several chips in a circuit. This can lower the cost of the final circuit. Having functions realized using a single chip can also reduce the amount of wiring in the circuit, allowing it to be manufactured using a smaller circuit board, as well as reducing the amount of power required for the circuit. Programmable components can also simplify the process of modifying an existing circuit. For many designs, we can correct design errors or modify our designs to add or change functionality simply by reprograming a chip, without needing to modify the rest of the circuit and its wiring.

Programmable devices also have some drawbacks. Depending on the application, a programmable device may have too many memory locations or components, and the extra logic would be wasted in the circuit design. Another consideration is the speed of the circuit. Memory chips, for example, are generally slower than combinatorial logic gates. Replacing gates with a lookup ROM, as we did in Chapter 5, can increase the amount of time needed to

generate the circuit's outputs. For complex circuits, however, the opposite may be true, depending on how many gates data must pass through to reach the circuit's outputs.

10.2 Programmable Logic Devices

Of all the methods we've looked at so far to realize combinatorial logic functions, the most efficient is the very first method we saw: minimize the function and design a circuit using fundamental logic gates. Decoders, multiplexers, and ROMs all incorporate logic components within their designs that are not needed for specific logic functions. However, building a circuit with different types and sizes of logic gates may require several chips. This means you'll also need a larger circuit board, additional wiring to connect the chips, and more power. This was the impetus behind the creation of **programmable logic devices**, or **PLDs**.

A PLD is a single chip with many basic logic gates built into it. The designer can specify the connections between the gates, and between input and output pins and the gates, to realize the desired function. Some types of PLDs allow only limited connections, while others are more flexible.

The designer "builds" the circuit using a computer program, which converts the design to the connections needed within the chip. The computer has a programmer, a piece of hardware attached to the computer (usually through a USB port), and the program sends the connection information to the programming hardware, which sends it to the PLD chip, thus creating the circuit to realize the desired function.

This section examines several classes of PLDs: **Programmable Array Logic** (**PAL**), **Programmable Logic Array** (**PLA**), **Complex Programmable Logic Device** (**CPLD**), and **Field-Programmable Gate Array** (**FPGA**).

10.2.1 Programmable Array Logic

A Programmable Array Logic, or PAL, chip consists of several AND-OR arrays of gates. That is, each array has several AND gates, and the outputs of the AND gates are connected to the inputs of an OR gate. The function inputs are connected to the input pins of the PAL chip, and the designer selects which function inputs and complements of function inputs are connected to the inputs of each AND gate. The designer also specifies which chip pin each OR gate output is connected to. PALs can typically output the value generated by the OR gate or its complement as specified by the designer. Some PALs have registers on their outputs that can store their values even after the inputs are changed. For now, we'll limit our discussion of all programmable logic devices to those without registers.

A generic PAL array is shown in Figure 10.1. This array is one of several that may be contained within the PAL chip. This chip has four inputs, and each input and its complement can be connected to an input of each AND gate. In this figure, each AND gate is shown with a single input line. This is the convention used for PLDs. This line represents all inputs to the gate. The Xs on this line indicate that a signal is connected to a gate input. In this figure, the uppermost AND gate has a and c as its inputs, and the second AND gate has inputs a', b, and c'. The third gate has no inputs shown. Some functions do not need to use every AND gate. By convention,

showing no connections indicates that the gate's inputs are 0. This causes the AND gate to output a 0, which is then input to the OR gate. Since anything ORed with 0 is just the logical OR of the other inputs, this essentially removes that AND gate from the equation. The output of the OR gate is $ac + a'bc' + 0$, or $ac + a'bc'$, the same function we've seen in previous chapters.

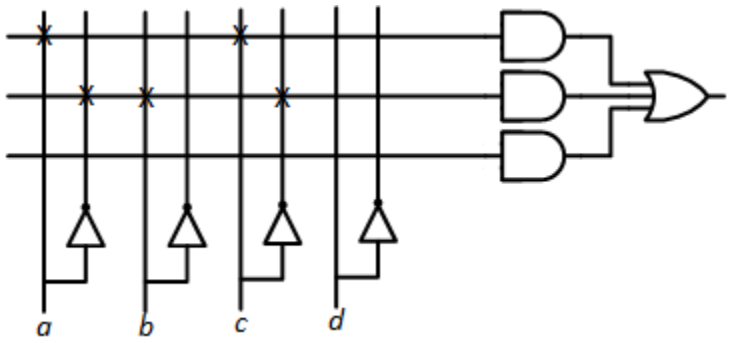


Figure 10.1: Generic PAL cell implementing the function $ac + a'bc'$

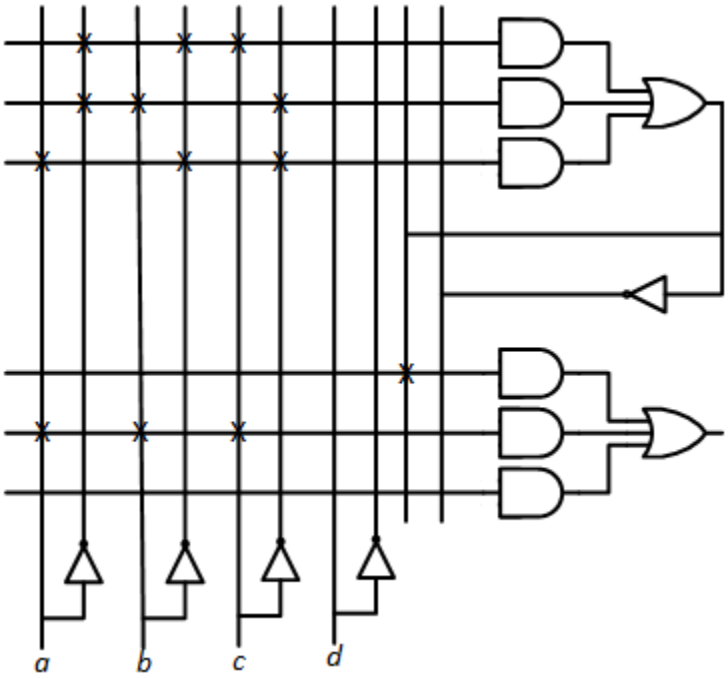
WATCH ANIMATED FIGURE 10.1

A PAL may include a feedback path from the output of some or all of the OR gates, allowing these outputs to be used as inputs to an AND gate in another AND-OR array within the chip. This is useful for functions that have many terms logically ORed together. Consider, for example, the function $a \oplus b \oplus c$; its truth table is shown in Figure 10.2 (a). Since PALs do not incorporate XOR gates, we must express this function solely using AND and OR functions. We must also use NOT gates to produce the complements of the inputs, a', b', and c', but the PAL already does that for us. This function can be expressed as $a \oplus b \oplus c = a'b'c + a'bc' + ab'c' + abc$. If our cells use the configuration shown in Figure 5.13, we could not fit this design into a single cell. We need four AND gates to generate the terms to be logically ORed together, but each cell has only three AND gates. However, if we have feedback paths, we could OR together the first three terms in one cell and feed this result back to a second cell. This cell could OR the feedback value with the fourth term, producing the desired function. This two-cell design is shown in Figure 10.2 (b).

To give you an idea for the size of a real-world PAL, consider the PAL16L8 chip. This chip has 10 dedicated input pins, two dedicated output pins, and six pins that can be configured either as input or output pins. There are eight cells, each of which can output its results to an output or bidirectional pin. Each cell has eight AND gates. The six cells that can drive the bidirectional pins can also feedback their outputs to be used as inputs to AND gates in the other cells.

a	*b*	*c*	*q*
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

(a)



(b)

Figure 10.2: $a \oplus b \oplus c$: (a) Truth table; (b) Implementation using two cells of a PAL. The upper cell generates $a'b'c+a'bc'+ab'c'$ and the lower cell generates the final function.

WATCH ANIMATED FIGURE 10.2

10.2.2 Programmable Logic Arrays

Although its name is similar to that of the PAL, a programmable logic array, or PLA, has one significant difference. In addition to being able to specify the inputs to the AND gate, the designer can also specify which AND gates input their values to each OR gate. In the generic PAL, each OR gate can have up to three AND gates supply its inputs. Here, we do not have this limitation. Also, one AND gate can send its output to more than one OR gate. Figure 10.3 shows a generic PLA cell.

The PLA is more flexible than the PAL, and at first glance it might seem to make sense to always use a PLA instead of a PAL. In fact, the exact opposite is the case. Most designs use PALs instead of PLAs. In spite of their flexibility, PLAs have some shortcomings. This flexibility introduces additional costs to the PLA. Besides being more expensive to fabricate, the PLA is generally slower than a PAL chip that implements the same function. For these reasons, PAL chips are used much more frequently than PLAs.

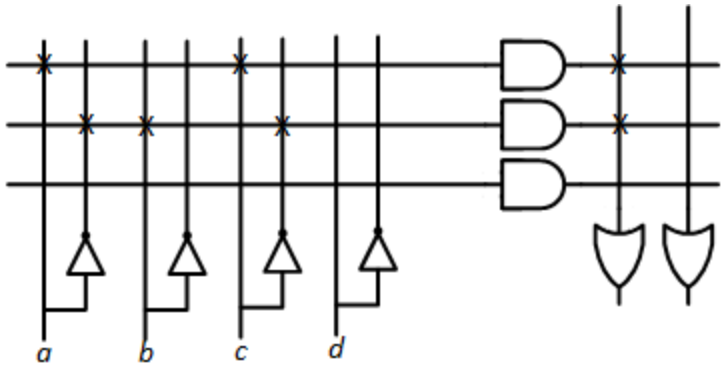


Figure 10.3: Generic PLA cell implementing the function *ac* + *a'bc'*

WATCH ANIMATED FIGURE 10.3

10.2.3 Complex PLDs and Field Programmable Gate Arrays

PALs and PLAs are frequently used in digital logic design. However, as the complexity of digital circuits increased, the need arose for more complex programmable logic devices to hold these designs. In this section, we'll briefly examine two classes of these devices: Complex Programmable Logic Devices (CPLDs) and Field-Programmable Gate Arrays (FPGAs).

PALs and PLAs are classified as **Simple Programmable Logic Devices**, or **SPLDs**. (I'm not sure why developers in this field seem to be more enamored with acronyms than those in other areas, but it is what it is.) Typical SPLDs have on the order of hundreds of gates. The PAL16L8 introduced earlier has about 100 gates. Complex PLDs, in comparison, have thousands of gates.

Beyond the sheer increase in the number of gates, CPLDs have a more complex internal architecture to allow the components to communicate with each other. A CPLD is designed as a series of logic blocks. Each logic block includes a logic array that is similar to a complete array within a PAL or PLA, and a **macrocell**. The macrocell may include registers to store results, and may act as an intermediary between the logic block and rest of the chip. We will refer to all of this as a cell.

The CPLD has an interconnect structure that allows the output of a cell to be sent to other cells. The rationale is much like that of the feedback connections within a PAL or PLA. Doing so allows for the implementation of more complex functions. But beyond this, it allows the designer to partition the design. One cell could be used to generate the value of some function, and the chip could send that value to other cells that perform different actions based on that value. For example, let's say your CPLD inputs some 4-bit value. When that value reaches 0000, you want the chip to cause several things to happen. Maybe you want to light an LED, start another sequence, or perform whatever operations you need to perform. One cell could be programmed to check the input value and generate an output that indicates whether or not it is 0000. It could send that value to other cells via the interconnect structure, and each of those cells could perform one of the necessary operations.

In addition to these interconnects, a CPLD also has logic blocks to coordinate I/O operations, that is, accepting inputs via the chip's input pins and outputting data via its output pins.

Like SPLDs, CPLDs are programmed before they are used in a circuit. Once the power is turned on, they begin to function right away. This may seem pretty obvious, but it is not the case for our next component.

Field-Programmable Gate Arrays are several orders of magnitude more complex than CPLDs. Whereas CPLDs have thousands of gates, FPGAs can have millions of gates. This allows designers to program much more complex designs into the chip than they can using CPLDs. These applications may include digital signal processing designs, and even some complete microprocessors. The OpenCores website has over 200 microprocessor designs that can be downloaded and programmed directly into an FPGA, as well as many designs for other components of computer systems.

Unlike SPLDs and CPLDs, many FPGAs are programmed when power is turned on. Typically, an FPGA circuit will include a ROM with the configuration programming information and any additional circuitry needed to load this information into the FPGA. This must be done every time the circuit's power is turned on, but once this is done the FPGA operates as desired as long as the power remains on.

10.3 Memory Devices

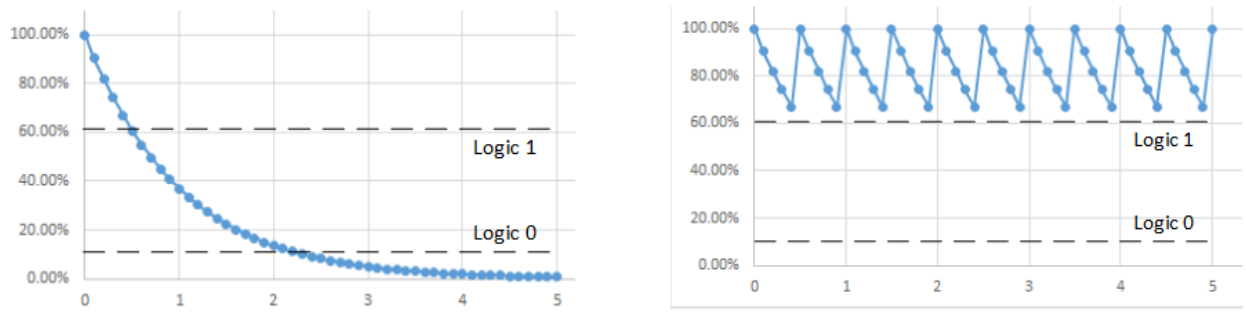
There are several different types of memory components, each with its own applications in digital system design. In Chapter 5, we noted that memory components can be classified as volatile, losing their contents when power is removed, and non-volatile, retaining stored data even when power is disconnected. Within each classification, there are several types of memory devices. We'll look at those in the next two subsections. Then we'll look inside the memory chip to see how it functions.

10.3.1 Volatile Memory Devices

In a typical personal computer, there are several types of memory. Two of these types are classified as volatile memory. One of these is dynamic random access memory, **DRAM**, or **dynamic RAM**. If your PC has 16GB of memory, this refers to its dynamic RAM.

Conceptually, you can think of dynamic RAM as acting like leaky capacitors. You can store charge in a capacitor or discharge a capacitor to remove its charge. These two conditions correspond to logic values 1 and 0. As long as power is supplied to the dynamic RAM, it stores its value. But when power is disconnected, the capacitors lose their charge and the data values are lost.

The capacitors in our model for dynamic RAM are not perfect. Even when power is connected, they leak charge. If we did not do anything, dynamic RAM would leak all its charge and lose its data, whether or not power is on. Figure 10.4 (a) represents the loss of charge over time for a logic value of 1.



(a) (b)

Figure 10.4: Model of dynamic RAM charge: (a) Without refresh; (b) With refresh.

To resolve this problem, dynamic RAM includes **refresh** circuitry. This circuitry reads the contents of memory locations and then writes the same values back to these locations. This does not change the data values, but it does store the maximum charge in the memory locations as shown in Figure 10.4 (b).

As stated earlier, Figure 10.4 shows this process for logic value 1. The same refresh process is used for all bits of dynamic RAM. The bits storing logic value 0, however, remain in a discharged state.

The other type of volatile memory in a computer system is called **static RAM**, or **SRAM**. This type of memory acts more like D flip-flops. Unlike dynamic RAM, static RAM does not leak charge and does not need to be refreshed. It is also faster than dynamic RAM. Unfortunately, it is also much more expensive than dynamic RAM, which is why DRAM is used as the primary physical memory in computer systems. Static RAM is used to construct **cache memory**, a small, high-speed memory found within a microprocessor chip. Cache memory is beyond the scope of this book. I recommend you check out my other book or any book on computer systems for a detailed description of cache memory and how it is used in computer systems.

10.3.2 Non-volatile Memory Devices

When power is removed from a non-volatile memory, it retains the data in its memory cells. This has many practical uses, one of which is the lookup tables introduced in Chapter 5. But this is just one application; in fact, it is one of the less frequently used applications of non-volatile memory. Before reading on, try to think of some devices you have used that include non-volatile memory.

I hope you did stop to think about everyday uses of non-volatile memory before reading this paragraph. If you did, I'd guess the most common (correct) answer that students chose is USB flash drives. You can insert the drive into the USB port of a computer, which gives it power and access to data. After writing a file to the drive, as I'm doing as I write this chapter, you can eject the drive and remove it from the USB port. This removes power from the flash drive. When you reinsert it into the USB port, you can access the file again. The non-volatile memory retained its contents, even when power was removed. An **SD card**, typically used to store data

in devices such as cameras, operates in much the same way, except it uses a different interface than USB.

A typical personal computer includes a non-volatile memory called the **BIOS ROM**, **BIOS**, or **UEFI**. The BIOS (Basic Input/Output System) or UEFI (Unified Extensible Firmware Interface) is used when the computer first starts up to initialize and test system hardware, and then load in software to start up the operating system (Windows, MacOS, Linux, et. al.).

One other place you'll find non-volatile memory in a computer system is inside the microprocessor chip. Many modern microprocessors use a microsequencer as a control unit. (If you think of the microprocessor as the *brain* of a computer, the control unit is the brain of the microprocessor.) A microsequencer stores data that causes it to perform the correct sequence of micro-operations to process instructions. The details are beyond the scope of this book; see my other book or any book on computer architecture for more details about microsequencers and how they work.

Just as there are different types of volatile memory, there are also several types of non-volatile memory.

10.3.2.1 Masked ROM, or ROM

A **masked ROM**, or just a ROM, is a read-only memory that has its data fixed when it is manufactured. This is useful when you're manufacturing a large number of a single product that uses a non-volatile memory that will never change its data. Some consumer appliances, such as washing machines or microwave ovens, which use simple microprocessors may use this type of memory. The memory in the microsequencer of a microprocessor chip would also fall into this category.

10.3.2.2 Programmable ROM

A **programmable read-only memory**, or **PROM**, is a ROM that can be programmed by the user with specialized programming hardware, but only once. The PROM chip contains a fuse for each bit of memory. Programming the PROM chip blows out the fuse or leaves it intact to store a logic 0 or 1 for that bit. Just as with fuses in electrical circuits, once a fuse is blown, it cannot be un-blown. PROMs are useful when prototyping circuits, or when manufacturing a small quantity of a product that might otherwise use a masked ROM.

10.3.2.3 Erasable PROM

Unlike a PROM, an **erasable PROM**, or **EPROM**, can be erased. The EPROM chip is programmed using specialized programming hardware, much like a PROM. Instead of blowing fuses, however, the EPROM stores data in separate cells that set the data for each bit. These cells retain their data under normal circumstances, but leak charge when exposed to ultraviolet light. EPROM chips have a clear window at the top of the chip. To erase the contents of the chip, it is placed in a UV eraser, just a strong ultraviolet light in a completely enclosed case. The EPROM is *cooked*, or exposed to the UV light for approximately 20 minutes to erase its contents so it can be

reprogrammed. When used in a circuit, it is common practice to cover the window with black electrical tape to prevent ultraviolet light from entering the chip and erasing its contents.

10.3.2.4 Electrically Erasable PROM

The **electrically erasable programmable read-only memory**, **EEPROM**, or **E^{2}PROM**, functions like an EPROM, but it can be erased and reprogrammed electrically. The hardware to erase and reprogram an E^{2}PROM is typically built into the circuit that uses the E^{2}PROM, allowing its contents to be modified in place, without removing it from the circuit. The BIOS or UEFI ROM in a computer is constructed using E^{2}PROM technology. Flash drives and SD cards use a type of E^{2}PROM that is block programmable. These devices write and read entire blocks of memory locations, rather than individual locations. They are particularly well suited for storage devices such as these, as well as the solid state drives becoming commonplace in personal computers.

10.3.3 Internal Organization

Internally, there are two primary ways to model a read-only memory. One is to model it as a PLD, with a series of AND gates sending their outputs to a series of OR gates. The other is to model each bit of storage as a D flip-flop, with associated hardware to access the individual bits within the memory chip. We'll look at both, starting with the PLD model.

In the PLD model, the address bits and their complements are sent as inputs to an array of AND gates. Figure 10.5 shows this model for a ROM with eight memory locations, each of which stores a 2-bit value. The connections to the AND gates are fixed, that is, they are always connected as shown in the figure. The uppermost AND gate outputs a 1 when $A_2' = A_1' = A_0' = 1$, or when the address is $A_2A_1A_0 = 000$. The next AND gate sets its output to 1 when the address is 001, and so on, continuing to the last AND gate, which outputs a 1 when the address is 111. This portion of the memory chip acts as a decoder, setting the output of exactly one AND gate to 1.

The outputs of the AND gates are sent to the OR array. If a bit at a given memory address is 1, we connect the output of the AND gate corresponding to that memory location to the input of the OR gate for that bit. In the animation for this figure, address input 4 (100) sets the output of the memory to 2 (10).

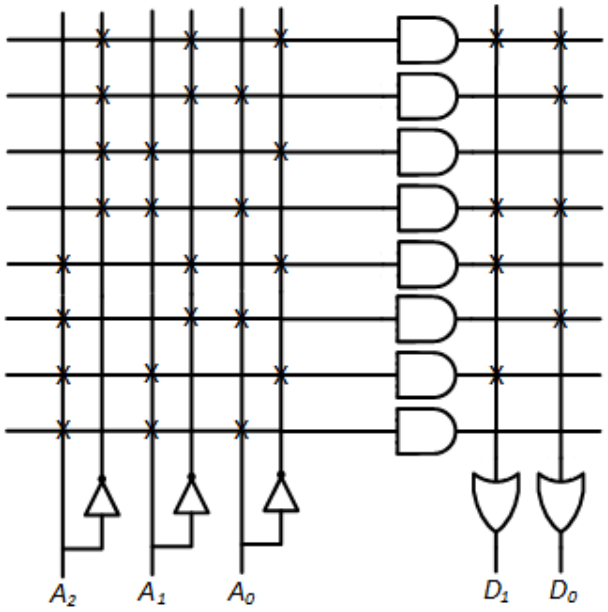


Figure 10.5: PLD model of an 8x2 ROM

WATCH ANIMATED FIGURE 10.5

Now let's look at the model using D flip-flops. Figure 10.6 shows a model for a 16x2 ROM. Here, we show the decoder explicitly rather than construct it using an array of AND gates. Inside the decoder, however, you basically have that AND gate array. Instead of making connections from the AND gates directly to the OR gates to represent the data, this model takes a different approach. The data is stored in an array of D flip-flops. When we access a memory location, we enable the tri-state buffers for the bits of only that memory location. This allows these bits to be output on the data pins of the memory chip.

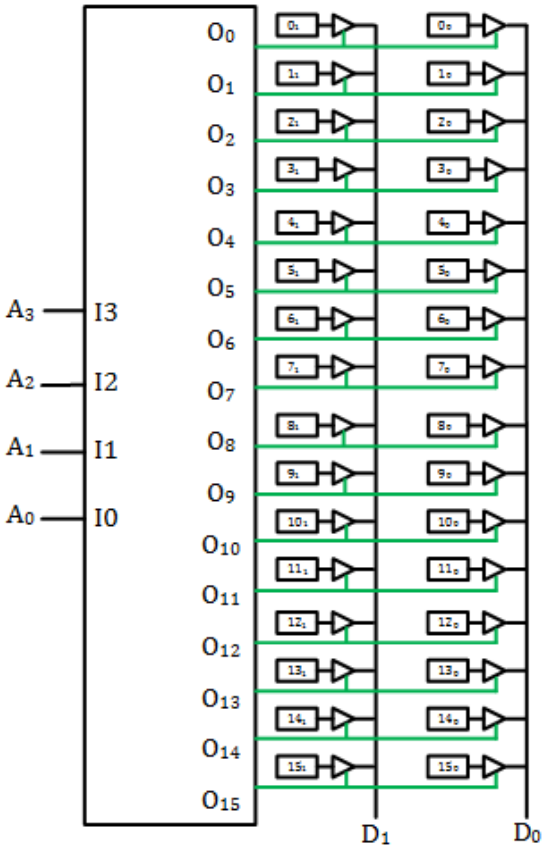


Figure 10.6: D flip-flop model of a 16x2 ROM. Buffer enable signals shown in green.

WATCH ANIMATED FIGURE 10.6

There is one problem with this model. It works fine when our chip has only eight locations, but what happens when it has 8G (2^{33}) locations? Its decoder would have 33 address bits and 2^{33} outputs. The size of a decoder is proportional to the number of outputs, and this decoder would be huge. Instead, memory chips can divide the address bits into different groups and decode each group. They then combine the outputs of the decoders to select individual locations.

Consider a chess board, as shown in Figure 107. In algebraic notation, letters *a* through *h* represent the columns of the board, and numbers 1 to 8 represent the rows. Each square is represented by the labels of its row and column, and each combination of labels corresponds to one unique square of the chessboard.

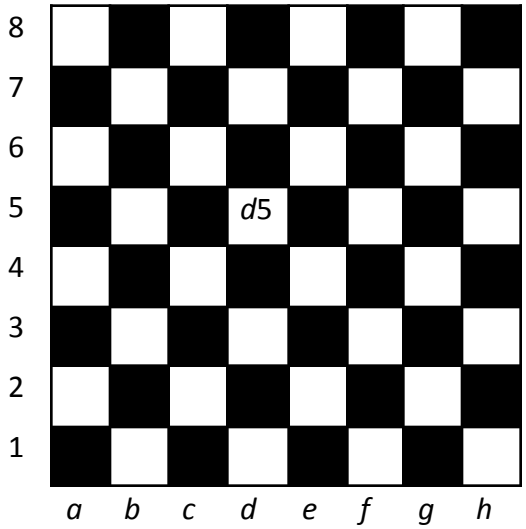


Figure 10.7: Chess board with algebraic notation. Square d5 labeled.

When using two-level decoding, we split up the bits into two groups. In the chess board analogy, one group is decoded to specify the row and the other group is decoded to specify the column. We combine the two decoded values to select a single memory location, or square on the chess board.

Figure 10.8 shows the 16x2 memory chip constructed using two 2 to 4 decoders. The first decoder enables every memory address with the two address bits corresponding to that decoder output. In this figure, address bits A_3 and A_2 are input to the upper decoder. Decoder output 0 is active when $A_3A_2 = 00$, so we want that decoder output to enable all locations having addresses with $A_3A_2 = 00$. For this chip, this is addresses 0000, 0001, 0010, and 0011, or 0, 1, 2, and 3. The other decoder outputs enable the remaining locations. This decoder is equivalent to selecting the row of the chess board.

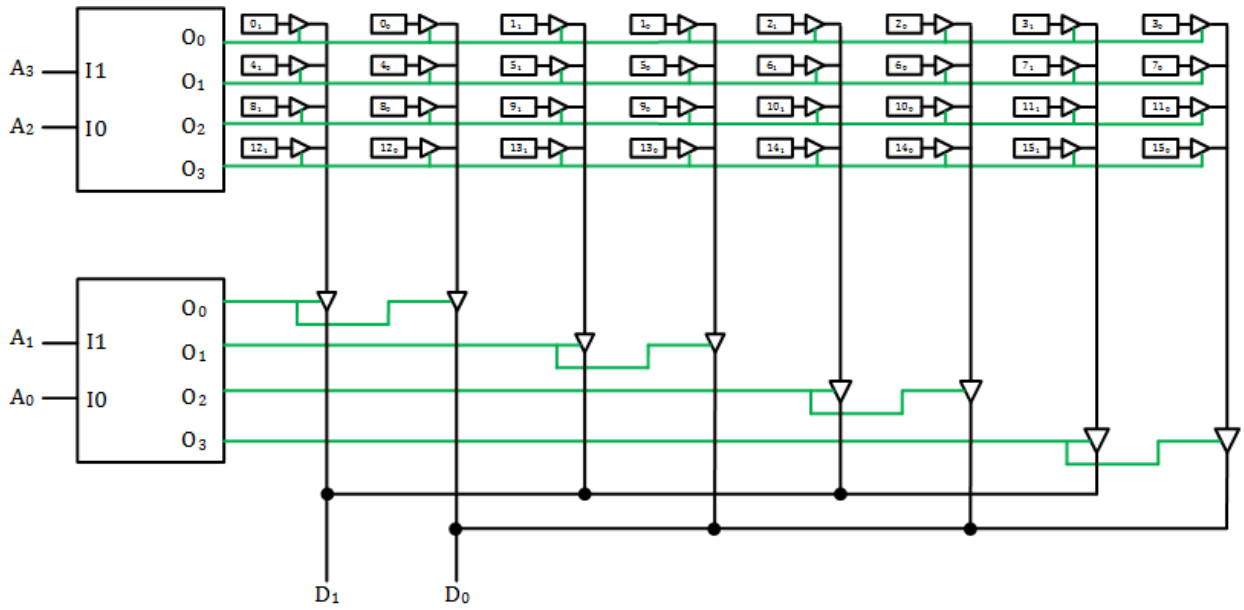


Figure 10.8: Memory chip constructed using two-level decoding. Buffer enable signals shown in green.

The remaining bits are decoded to select all locations with the address bits corresponding to its inputs, A_1 and A_0. When A_1A_0 = 00, the chip enables the tri-state buffers that can receive data from all memory locations with these low-order address bits, 0000 (0), 0100 (4), 1000 (8), and 1100 (12) in this case. The other values of A_1 and A_0 select their corresponding locations. This is equivalent to selecting the column of the chess board.

By selecting one row and one column, the memory chip accesses one memory address within the chip, just as specifying one row and one column specifies one square on the chess board.

By using two smaller decoders, this chip reduces the overall size of the decoders. Since the size of the decoder is proportional to the number of outputs, using two-level decoding reduces the total size of the decoding hardware from O(16) to O(4) + O(4), or O(8), a savings of about 50%. For larger memory chips, this savings can reach over 99%. It is also possible to use more than two levels of decoding to realize even greater reductions in decoding hardware.

The models are meant to give you a basic idea of how a ROM works, but they are incomplete. Real memory chips include an enable signal. When used in a computer system, there may be many ROM chips, as well as RAM chips, and we need to make sure only one memory chip is active at a given time. (If your circuit uses the ROM chip as a lookup table, you can just set this input so the chip is always enabled.) The memory chip may also have a separate output enable signal. In a computer system, you would typically enable the chip first and give it time to decode its address, enable the tri-state buffers, and have the data ready to be output. Then the computer system would enable the outputs, usually through additional tri-state buffers, by asserting the output enable signal. (For lookup ROMs, this signal can always be asserted too.) Figure 10.9 shows how these signals could be used within the memory chip to achieve their desired functions.

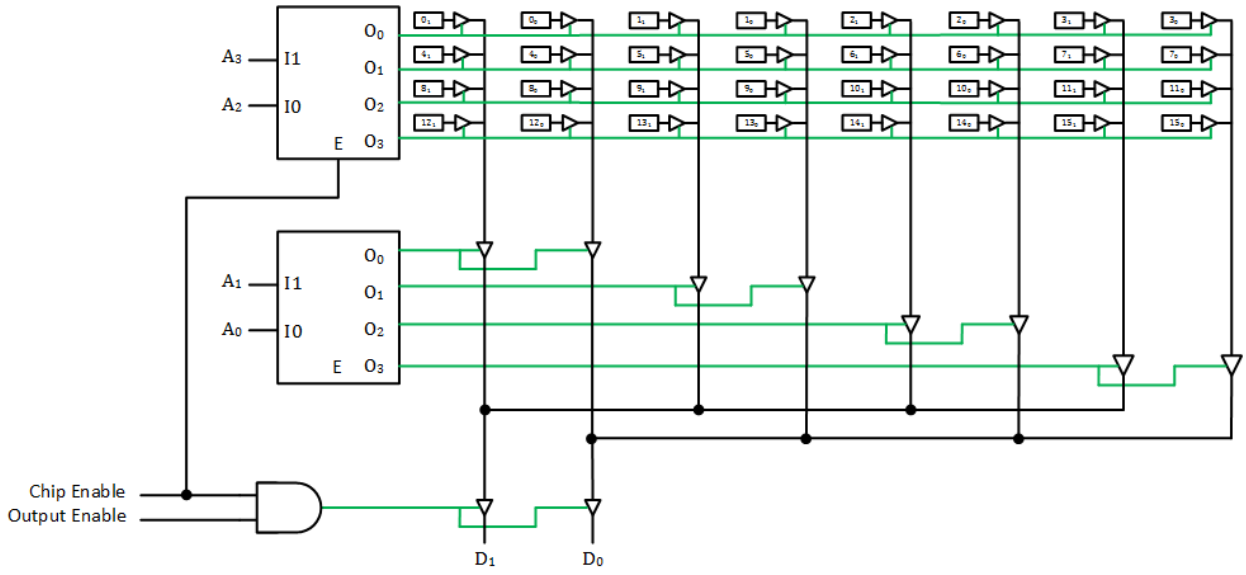


Figure 10.9: D flip-flop model of a 16x2 ROM with chip enable and output enable signals.

WATCH ANIMATED FIGURE 10.9

One final thing that is missing is how we can have the chip write data into a memory location. First, we would include a data path from the data pins of the chip to the D inputs of the flip-flops for each memory location. We would also load the location when its conditions are met. The chip would decode the address just as it does to output data. It would also combine it with a chip enable signal and some signal that indicates we are programming the chip (for ROMs) or just writing data to memory (for RAMs). When using ROMs in digital circuits, we generally don't need to do that once the chip is programmed initially, so we do not cover that here. The reader should consult a book on computer organization and architecture for more information on how this is done.

10.4 Summary

Programmable devices offer designers a number of advantages. Using a single chip to hold several components can reduce circuit size, wiring, power consumption, and propagation delay.

One class of programmable devices is PLDs, or programmable logic devices. A PLD has many combinatorial logic gates within the chip; some, but not all, also include a limited number of flip-flops. By specifying the connections of chip inputs, gate inputs and outputs, and chip outputs, the designer can create an entire circuit within a single chip.

There are several types of PLDs. A PAL, programmable array logic, is composed of multiple AND-OR arrays. The output of the OR gate, or its complement, may be connected to a chip output or to a flip-flop input, with the output of the flip-flop connected to the chip output. Some PALs allow the output of some or all OR gates to be fed back and made available as an input to other AND-OR arrays on the chip. This is especially useful for more complex functions.

A programmable logic array, PLA, is similar to a PAL. Unlike a PAL, however, any AND gate can send its output to any OR gate; the AND-OR array is not fixed. PLAs are more flexible than PALs, but in general they are slower than PALs.

Complex Programmable Logic Devices, CPLDs, have many more gates than the simpler PALs and PLAs, as well as more complex internal architectures. This increases the ability for components within the chip to be interconnected and communicate with each other. CPLDs include multiple logic blocks and macrocells.

Field Programmable Gate Arrays, FPGAs, have several orders of magnitude more gates than CPLDs. Many FPGAs are programmed when power is connected, whereas the other components are programmed before they are added to the circuit. The FPGA typically includes a ROM with configuration information and hardware that uses this information to set up the FPGA.

There are several types of memory devices. Random Access Memory, RAM, is volatile. When power is removed from a RAM chip, its data is lost. Dynamic RAM is the type of memory used in most computer systems. Its contents are constantly refreshed to restore leaked charge. Static RAM is faster but more expensive than DRAM. It is used to construct cache memory within a microprocessor.

Masked ROMs, PROMs, EPROMs, and E^2PROMs are types of read-only memory. They are used in USB flash drives, SD cards, and other devices that are required to retain their data when power is removed. As discussed earlier in this book, they can be used as lookup tables in digital circuits. They are also used to store microcode within many microprocessors. Internally, a ROM can be modeled as a PAL in which the array of AND gates is in a fixed configuration that decodes the input address bits. It can also be modeled as an array of flip-flops with outputs enabled by decoders that explicitly decode the address bits. Multiple decoders can be used to divide the address bits and decrease the total size of the decoding hardware.

Bibliography

Carpinelli, J. D. (2001). *Computer systems organization & architecture*. Addison-Wesley.

Hayes, J. P. (1993). *Introduction to digital logic design*. Addison-Wesley.

Mano, M. M., & Ciletti, M. (2017). *Digital design: with an introduction to the Verilog HDL, VHDL, and SystemVerilog* (6th ed.). Pearson.

Mano, M. M., Kime, C., & Martin, T. (2015). *Logic & computer design fundamentals* (5th ed.). Pearson.

Nelson, V., Nagle, H., Carroll, B., & Irwin, D. (1995). *Digital logic circuit analysis and design*. Pearson.

OpenCores. (2023). Retrieved August 1, 2023, from https://opencores.org/

Robledo, E. (2021, August 4). *What is CPLD (Complex Programmable Logic Device)?* Fusion 360 Blog. https://www.autodesk.com/products/fusion-360/blog/cpld-overview/

Roth, J. C. H., Kinney, L. L., & John, E.B. (2020). *Fundamentals of logic design enhanced edition* (7th ed.). Cengage Learning.

Skahill, K, (1996). *VHDL for Programmable Logic* (2nd ed.). Prentice Hall.

Texas Instruments. (March, 1992). *Standard High-Speed Programmable Array Logic Datasheet*. SRPS016-D2705.

Wakerly, J. F. (2018). *Digital design: Principles and practices* (5th ed.). Pearson.

Exercises

1. List additional advantages of using a programmable memory and PLDs beyond those listed in Section 10.1.

2. List additional advantages of using a programmable memory and PLDs beyond those listed in Section 10.1.

3. List the contents of all memory locations in the memory component shown in Figure 10.5.

4. Show the PLD model for a ROM with four address inputs, four data outputs, and the following values stored in locations 0 to 15, respectively:
   2,5,8,9,0,11,3,14,12,5,7,15,1,6,14,11.

5. Implement the function $ab+ac+bc$ on the PAL cell shown in Figure 10.1.

6. Implement the function $a' \oplus b \oplus c$ on the PAL cells shown in Figure 10.2.

7. Modify the connections of the PAL cell shown in Figure 10.1 to realize the following truth table.

a	*b*	*c*	*q*
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

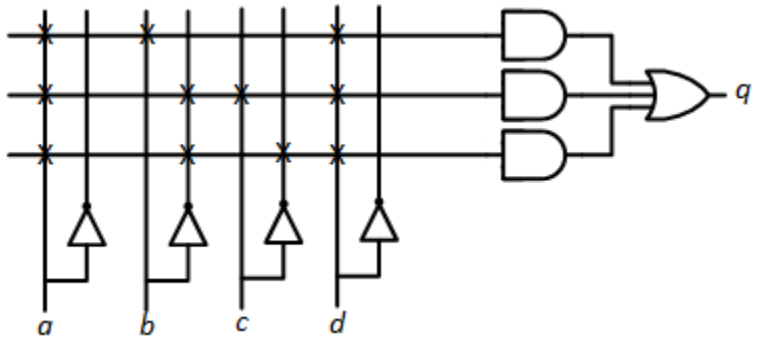
8. Modify the connections of the PAL cell shown in Figure 10.2 to realize the following truth table.

a	*b*	*c*	*q*
0	0	0	1
0	0	1	0
0	1	0	
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

9. Show the connections in Figure 10.1 or 10.2 needed for a PAL to realize the following functions for three variables. Use feedback if necessary.

    a. AND
    b. OR
    c. XOR
    d. NAND
    e. NOR
    f. XNOR

10. Show the truth table and function in minimal form realized by the following PAL cell.



11. Show the connections in Figure 10.3 needed to realize the functions *a*+*b*+*c* and *abc*.

12. Show the connections in Figure 10.3 needed to realize the functions *ab*+*ac* and *ac*+*bc*.

13. Show the connections in Figure 10.1 needed to realize the following truth table.

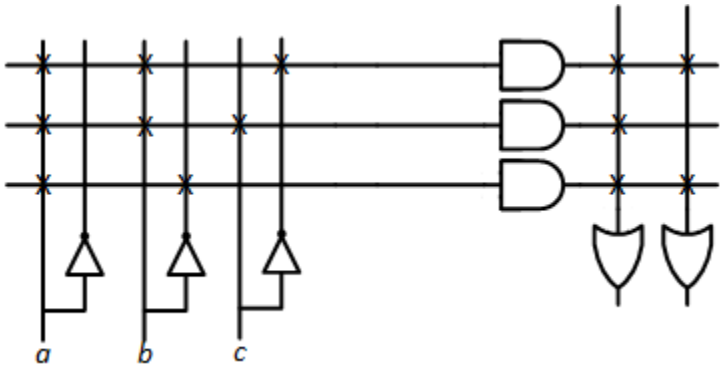
a	*b*	*c*	*q*	*r*
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	0	1
1	0	0	0	0
1	0	1	0	1
1	1	0	1	1
1	1	1	1	1

14. Show the connections in Figure 10.3 needed to realize the following truth table.

a	*b*	*c*	*q*	*r*
0	0	0	0	1
0	0	1	1	1
0	1	0	0	1
0	1	1	1	0
1	0	0	1	1
1	0	1	1	0
1	1	0	0	0
1	1	1	0	0

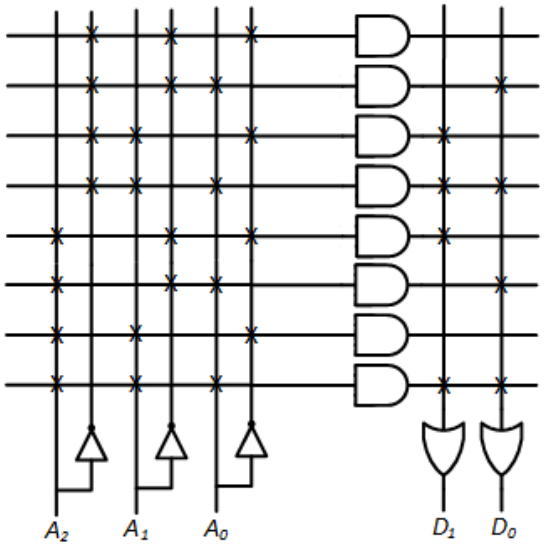
15. A PLA has three inputs, four AND gates, and two OR gates. The OR gate outputs can be fed back as inputs to the AND gates. Show this PLA and the connections needed to realize the functions *a'b+c* and *ab'+ac+a'b+c*.

16. Show the truth table and functions realized by the following PLA. Reprogram the PLA to generate the same truth table using the fewest possible connections.



17. Design an 8x2 memory chip using 3-level decoding.

18. Show the contents of the following ROM.



19. A ROM has three inputs and three outputs. Show the AND-OR configuration of a ROM with the following contents: 7, 6, 5, 4, 3, 2, 1, 0.

20. Repeat Problem 19 for a D flip-flop configuration of the ROM using a single decoder.

21. Repeat Problem 20 for a ROM using two-level decoding.

22. In Figure 10.6, swap A_3 and A_0, and swap A_2 and A_1. Show the address corresponding to each bit in the memory.

23. In Figure 10.8, swap A_3 and A_2, and swap A_1 and A_0. Show the address corresponding to each bit in the memory.

24. Show the 2-bit multiplier lookup ROM in Figure 5.12 as an AND-OR array implementation.