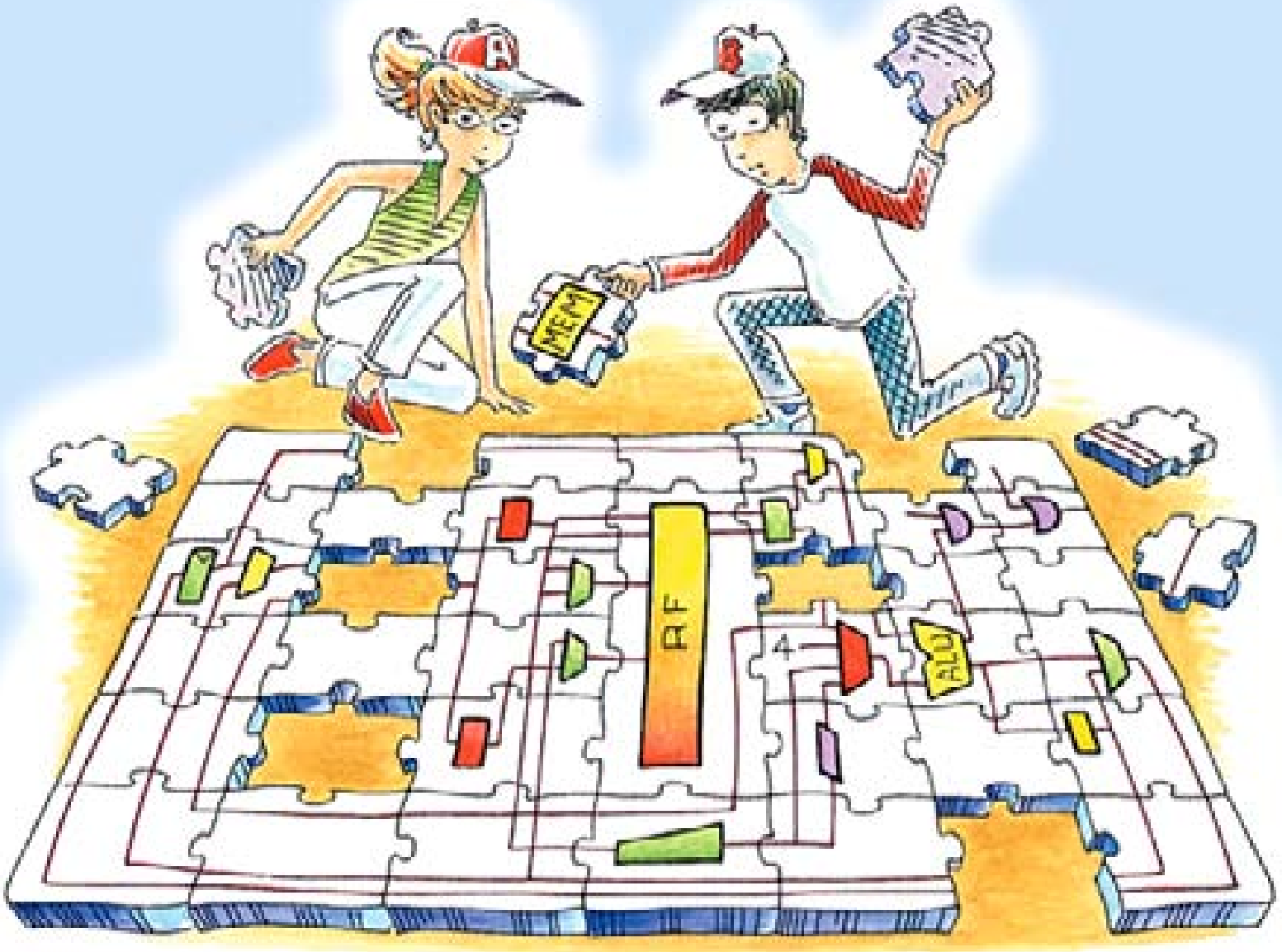
Digital Design and Computer Architecture



David Money Harris & Sarah L. Harris

In Praise of *Digital Design and Computer Architecture*

Harris and Harris have taken the popular pedagogy from Computer Organization and Design *to the next level of refinement, showing in detail how to build a MIPS microprocessor in both Verilog and VHDL. Given the exciting opportunity that students have to run large digital designs on modern FGPAs, the approach the authors take in this book is both informative and enlightening.*

David A. Patterson University of California, Berkeley

Digital Design and Computer Architecture *brings a fresh perspective to an old discipline. Many textbooks tend to resemble overgrown shrubs, but Harris and Harris have managed to prune away the deadwood while preserving the fundamentals and presenting them in a contemporary context. In doing so, they offer a text that will benefit students interested in designing solutions for tomorrow's challenges.*

Jim Frenzel University of Idaho

Harris and Harris have a pleasant and informative writing style. Their treatment of the material is at a good level for introducing students to computer engineering with plenty of helpful diagrams. Combinational circuits, microarchitecture, and memory systems are handled particularly well.

James Pinter-Lucke Claremont McKenna College

Harris and Harris have written a book that is very clear and easy to understand. The exercises are well-designed and the real-world examples are a nice touch. The lengthy and confusing explanations often found in similar textbooks are not seen here. It's obvious that the authors have devoted a great deal of time and effort to create an accessible text. I strongly recommend Digital Design and Computer Architecture.

Peiyi Zhao Chapman University

Harris and Harris have created the first book that successfully combines digital system design with computer architecture. Digital Design and Computer Architecture *is a much-welcomed text that extensively explores digital systems designs and explains the MIPS architecture in fantastic detail. I highly recommend this book.*

James E. Stine, Jr., Oklahoma State University

Digital Design and Computer Architecture *is a brilliant book. Harris and Harris seamlessly tie together all the important elements in microprocessor design—transistors, circuits, logic gates, finite state machines, memories, arithmetic units—and conclude with computer architecture. This text is an excellent guide for understanding how complex systems can be flawlessly designed.*

Jaeha Kim Rambus, Inc.

Digital Design and Computer Architecture *is a very well-written book that will appeal to both young engineers who are learning these subjects for the first time and also to the experienced engineers who want to use this book as a reference. I highly recommend it.*

A. Utku Diril Nvidia Corporation

Digital Design and Computer Architecture

About the Authors

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

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

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

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

Digital Design and Computer Architecture

David Money Harris
Sarah L. Harris

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann Publishers is an imprint of Elsevier

Publisher: Denise E. M. Penrose
Senior Developmental Editor: Nate McFadden
Publishing Services Manager: George Morrison
Project Manager: Marilyn E Rash
Assistant Editor: Mary E. James
Editorial Assistant: Kimberlee Honjo
Cover and Editorial Illustrations: Duane Bibby
Interior Design: Frances Baca Design
Composition: Integra
Technical Illustrations: Harris and Harris/Integra
Production Services: Graphic World Inc.
Interior Printer: Courier-Westford
Cover Printer: Phoenix Color Corp.

Morgan Kaufmann Publishers is an imprint of Elsevier.
500 Sansome Street, Suite 400, San Francisco, CA 94111

This book is printed on acid-free paper.

Library of Congress Cataloging-in-Publication Data

Harris, David Money.
Digital design and computer architecture / David Money Harris and
Sarah L. Harris.—1st ed.
p. cm.
Includes bibliographical references and index.
ISBN 13: 978-0-12-370497-9 (alk. paper)
ISBN 10: 0-12-370497-9
1. Digital electronics. 2. Logic design. 3. Computer architecture. I. Harris, Sarah L. II. Title.
TK7868.D5H298 2007
621.381—dc22

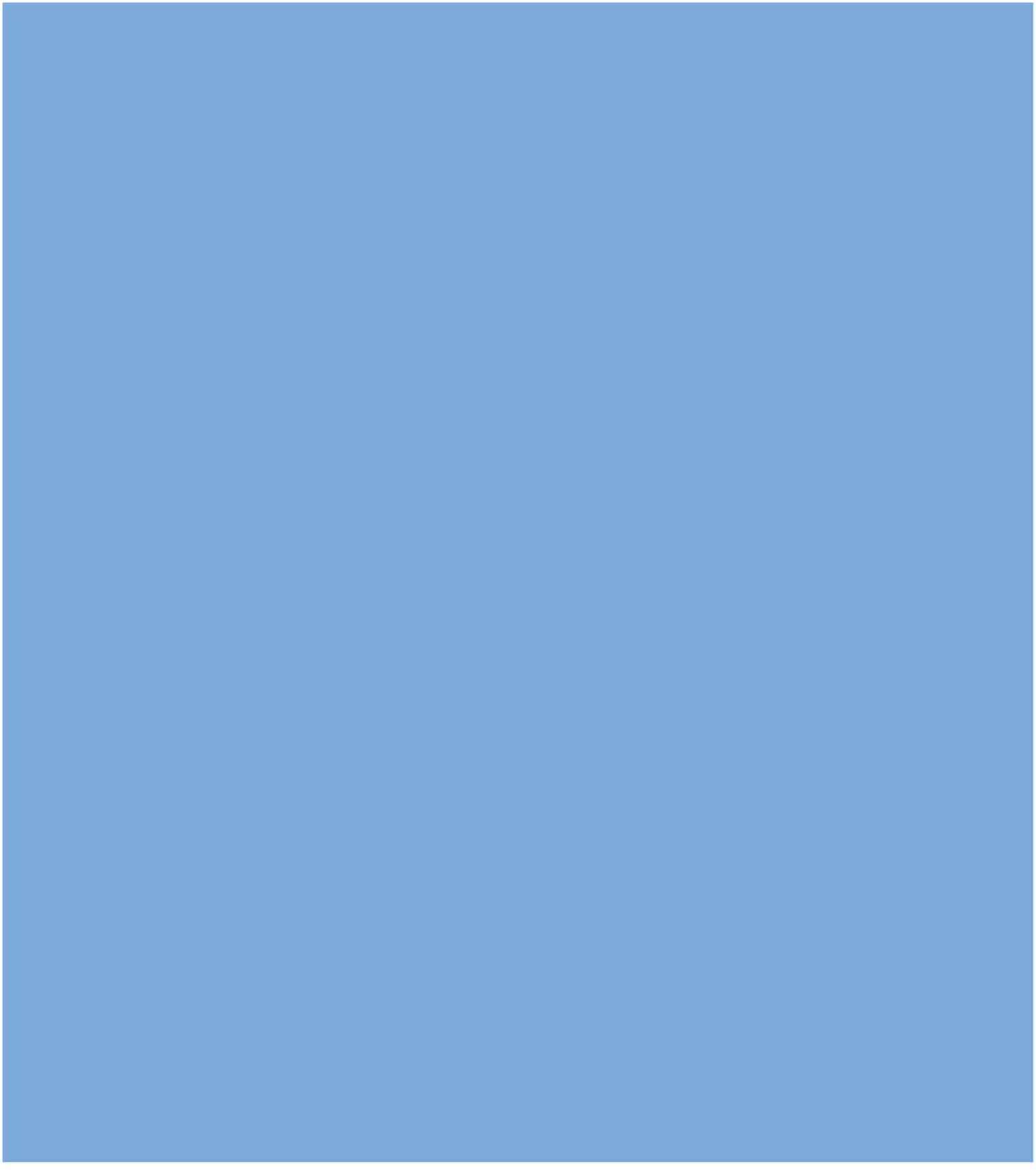
2006030554

For information on all Morgan Kaufmann publications,
visit our Web site at *www.mkp.com*

Printed in the United States of America
07 08 09 10 11 5 4 3 2 1

To my family, Jennifer and Abraham
– DMH

To my sisters, Lara and Jenny
– SLH



Contents

Preface

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

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

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

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

FEATURES

This book offers a number of special features.

Side-by-Side Coverage of Verilog and VHDL

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

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

Classic MIPS Architecture and Microarchitecture

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

Real-World Perspectives

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

Accessible Overview of Advanced Microarchitecture

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

End-of-Chapter Exercises and Interview Questions

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

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

ONLINE SUPPLEMENTS

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

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

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

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

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

HOW TO USE THE SOFTWARE TOOLS IN A COURSE

Xilinx ISE WebPACK

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

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

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

Synplify Pro

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

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

PCSPIM

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

LABS

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

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

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

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

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

BUGS

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

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

ACKNOWLEDGMENTS

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

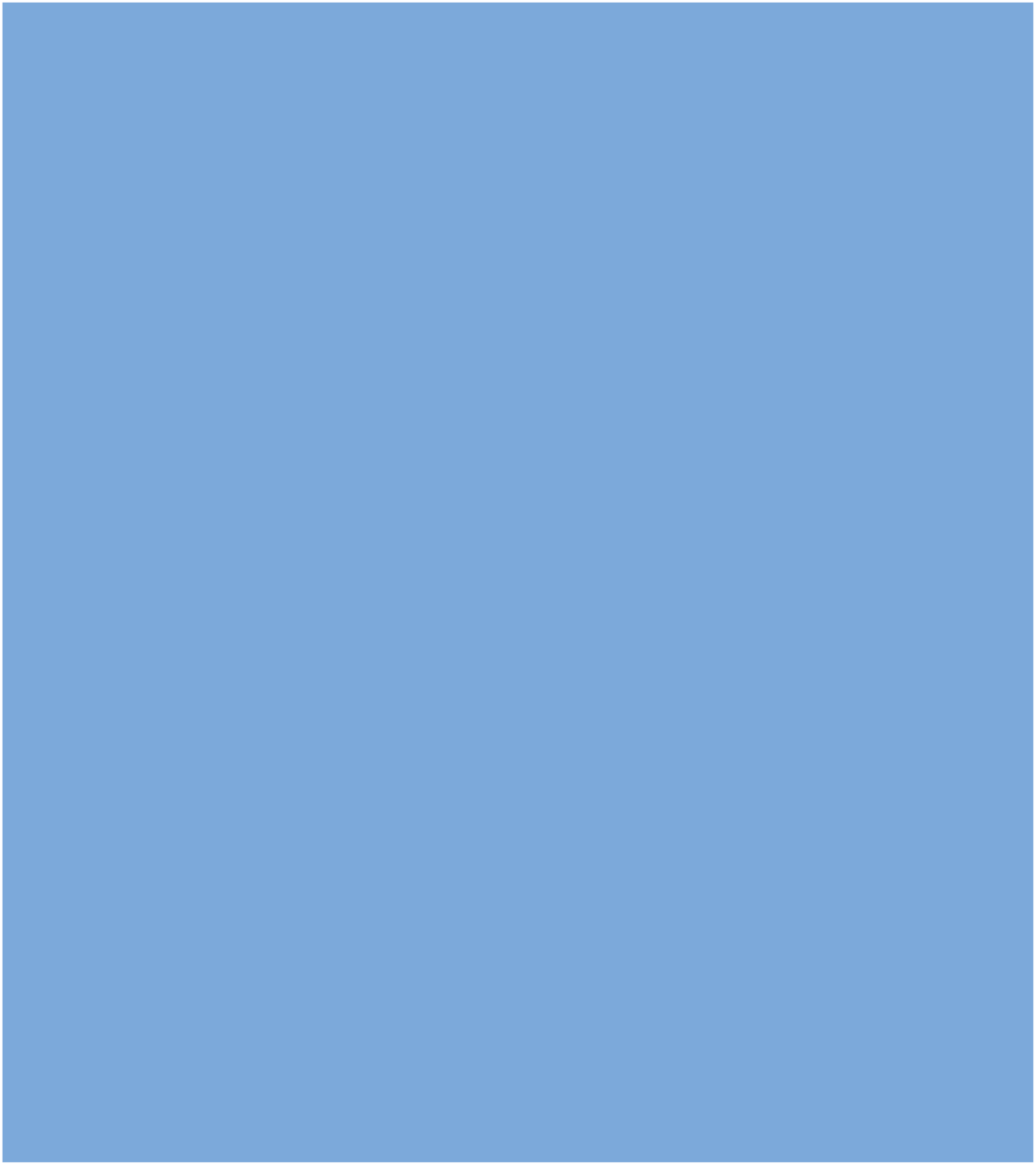
Duane Bibby, our favorite cartoonist, labored long and hard to illustrate the fun and adventure of digital design. We also appreciate the enthusiasm of Denise Penrose, Nate McFadden, and the rest of the team at Morgan Kaufmann who made this book happen. Jeff Somers at Graphic World Publishing Services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

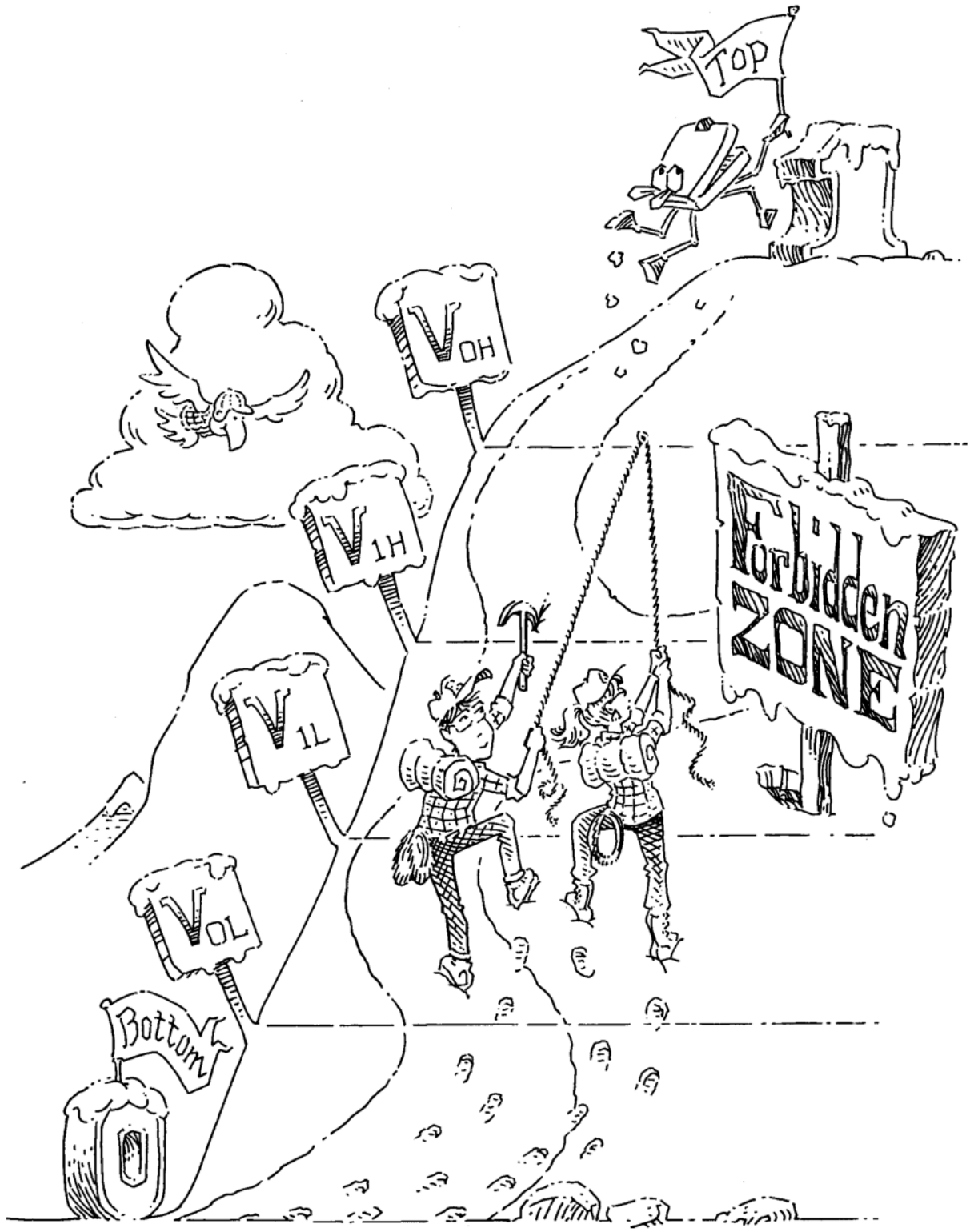
Numerous reviewers have substantially improved the book. They include John Barr (Ithaca College), Jack V. Briner (Charleston Southern University), Andrew C. Brown (SK Communications), Carl Baumgaertner (Harvey Mudd College), A. Utku Diril (Nvidia Corporation), Jim Frenzel (University of Idaho), Jaeha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

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

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

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




Top
V_{OH}
V_{1H}
V_{1L}
V_{OL}
Bottom
Forbidden ZONE

From Zero to One

1

1.1 THE GAME PLAN

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

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

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

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

1.2 THE ART OF MANAGING COMPLEXITY

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

1.2.1 Abstraction

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

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

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

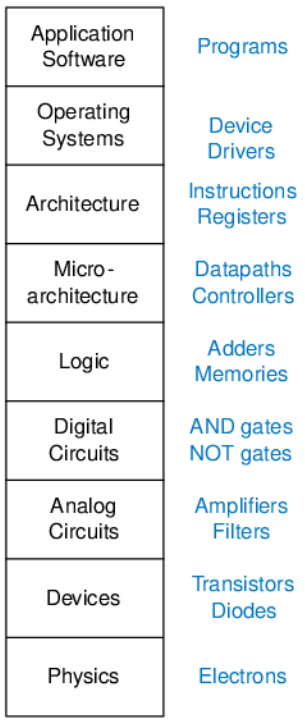


Figure 1.1 Levels of abstraction for electronic computing system

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

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

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

1.2.2 Discipline

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

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

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

1.2.3 The Three -Y's

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

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

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

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

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

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

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

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

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

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

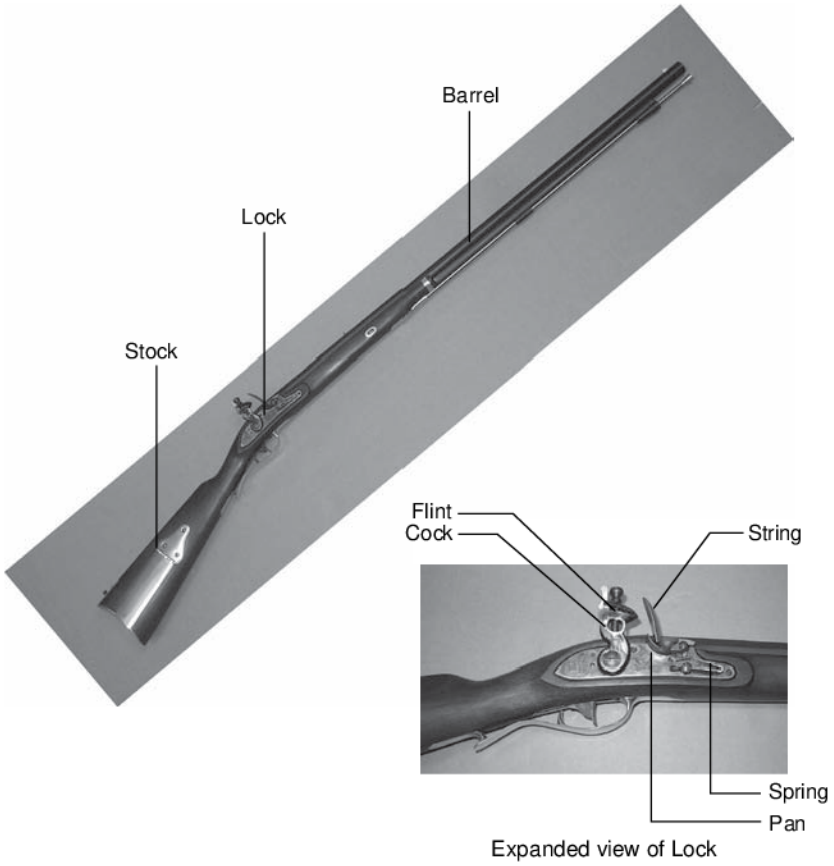


Figure 1.2 Flintlock rifle with a close-up view of the lock (Image by Euroams Italia. www.euroarms.net © 2006).

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

1.3 THE DIGITAL ABSTRACTION

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

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

[1] And we thought graduate school was long!

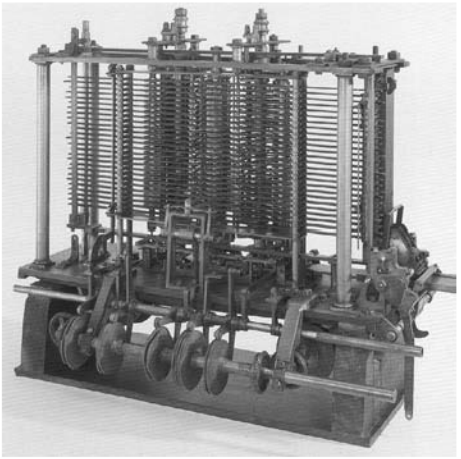


Figure 1.3 Babbage's Analytical Engine, under construction at the time of his death in 1871 (image courtesy of Science Museum/Science and Society Picture Library).

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

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

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

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

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

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

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

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

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

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

1.4 NUMBER SYSTEMS

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

1.4.1 Decimal Numbers

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

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

1.4.2 Binary Numbers

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

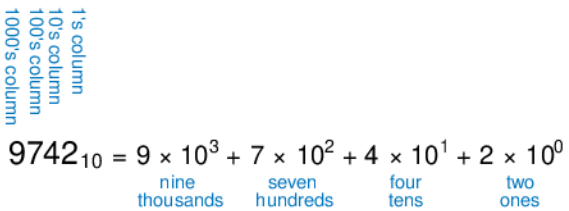


Figure 1.4 Representation of a decimal number

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

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

Example 1.1 BINARY TO DECIMAL CONVERSION

Convert the binary number 10110_2 to decimal.

Solution: Figure 1.5 shows the conversion.

Table 1.1 Binary numbers and their decimal equivalent

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

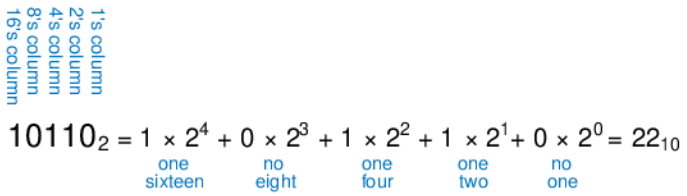


Figure 1.5 Conversion of a binary number to decimal

Example 1.2 DECIMAL TO BINARY CONVERSION

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

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

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

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

1.4.3 Hexadecimal Numbers

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

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

Example 1.3 HEXADECIMAL TO BINARY AND DECIMAL CONVERSION

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

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

Table 1.2 Hexadecimal number system

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

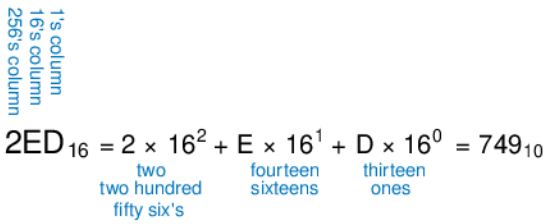


Figure 1.6 Conversion of hexadecimal number to decimal

Example 1.4 BINARY TO HEXADECIMAL CONVERSION

Convert the binary number 1111010_2 to hexadecimal.

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

Example 1.5 DECIMAL TO HEXADECIMAL AND BINARY CONVERSION

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

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

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

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

1.4.4 Bytes, Nibbles, and All That Jazz

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

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

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

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

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

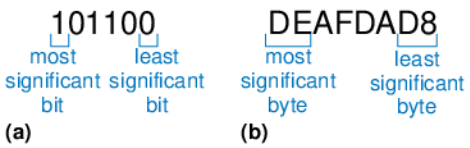


Figure 1.7 Least and most significant bits and bytes

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

Example 1.6 ESTIMATING POWERS OF TWO

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

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

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

1.4.5 Binary Addition

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

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

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

Example 1.7 BINARY ADDITION

Compute $0111_2 + 0101_2$.

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

```
  111
  0111
+ 0101
------
 1100
```

Figure 1.9 Binary addition example

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

Example 1.8 ADDITION WITH OVERFLOW

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

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

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

Figure 1.10 Binary addition example with overflow

1.4.6 Signed Binary Numbers

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

Sign/Magnitude Numbers

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

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

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

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

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

Example 1.9 SIGN/MAGNITUDE NUMBERS

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

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

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

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

Two's Complement Numbers

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

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

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

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

Example 1.10 TWO'S COMPLEMENT REPRESENTATION OF A NEGATIVE NUMBER

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

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

Example 1.11 VALUE OF NEGATIVE TWO'S COMPLEMENT NUMBERS

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

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

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

Example 1.12 ADDING TWO'S COMPLEMENT NUMBERS

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

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

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

Example 1.13 SUBTRACTING TWO'S COMPLEMENT NUMBERS

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

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

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

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

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

Example 1.14 ADDING TWO'S COMPLEMENT NUMBERS WITH OVERFLOW

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

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

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

Comparison of Number Systems

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

Table 1.3 Range of *N*-bit numbers

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

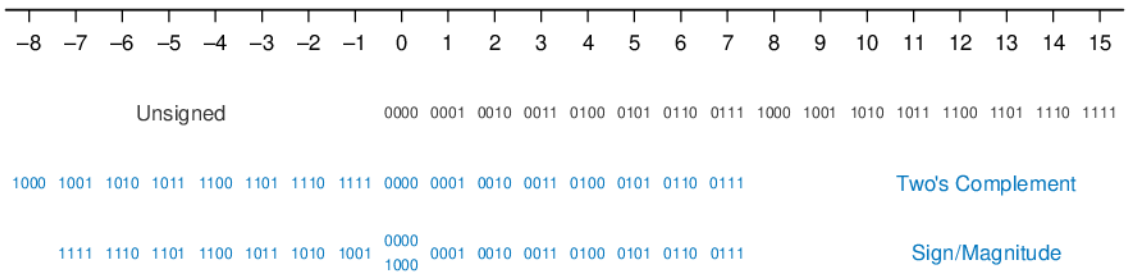


Figure 1.11 Number line and 4-bit binary encodings

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

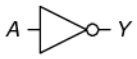
Figure 1.11 shows a number line indicating the values of 4-bit numbers in each system. Unsigned numbers span the range [0, 15] in regular binary order. Two's complement numbers span the range [−8, 7]. The nonnegative numbers [0, 7] share the same encodings as unsigned numbers. The negative numbers [−8,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

1.5 LOGIC GATES

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

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

NOT

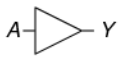


$Y = \overline{A}$

A	Y
0	1
1	0

Figure 1.12 NOT gate

BUF



$Y = A$

A	Y
0	0
1	1

Figure 1.13 Buffer

AND

$Y = AB$

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

Figure 1.14 AND gate

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

1.5.1 NOT Gate

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

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

1.5.2 Buffer

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

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

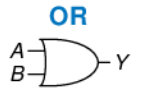
The triangle symbol indicates a buffer. A circle on the output is called a *bubble* and indicates inversion, as was seen in the NOT gate symbol of Figure 1.12.

1.5.3 AND Gate

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

1.5.4 OR Gate

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



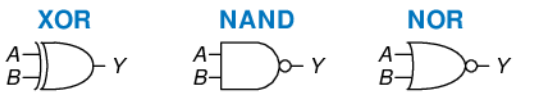
$Y = A + B$

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

Figure 1.15 OR gate

1.5.5 Other Two-Input Gates

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



$Y = A \oplus B$

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

$Y = \overline{AB}$

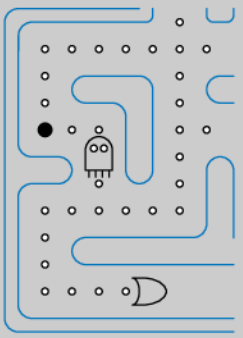
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

$Y = \overline{A + B}$

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

Figure 1.16 More two-input logic gates

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



Example 1.15 XNOR GATE

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

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

$Y = \overline{A \oplus B}$

A	B	Y
0	0	
0	1	
1	0	
1	1	

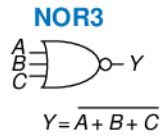
Figure 1.17 XNOR gate

1.5.6 Multiple-Input Gates

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

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

Figure 1.18 XNOR truth table



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

Figure 1.19 Three-input NOR gate

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

Figure 1.20 Three-input NOR truth table

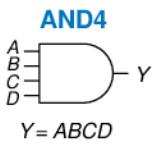


Figure 1.21 Four-input AND gate

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

Example 1.16 THREE-INPUT NOR GATE

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

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

Example 1.17 FOUR-INPUT AND GATE

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

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

1.6 BENEATH THE DIGITAL ABSTRACTION

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

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

1.6.1 Supply Voltage

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

1.6.2 Logic Levels

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

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

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

Figure 1.22 Four-input AND truth table

1.6.3 Noise Margins

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

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

Example 1.18

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

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

1.6.4 DC Transfer Characteristics

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

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

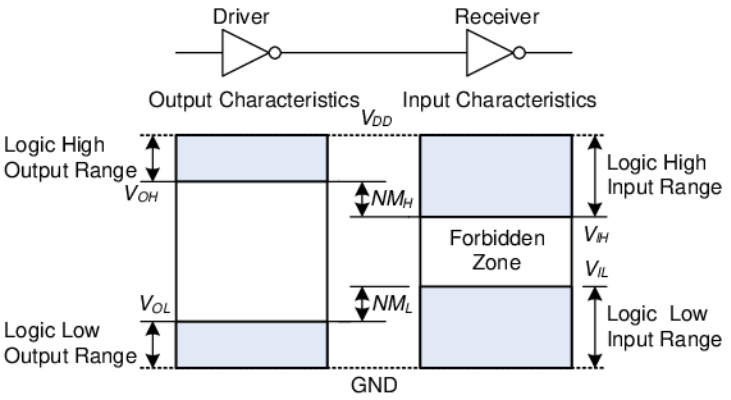


Figure 1.23 Logic levels and noise margins

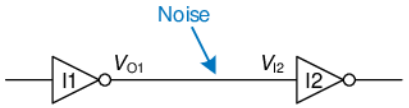


Figure 1.24 Inverter circuit

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

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

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

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

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

1.6.5 The Static Discipline

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

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

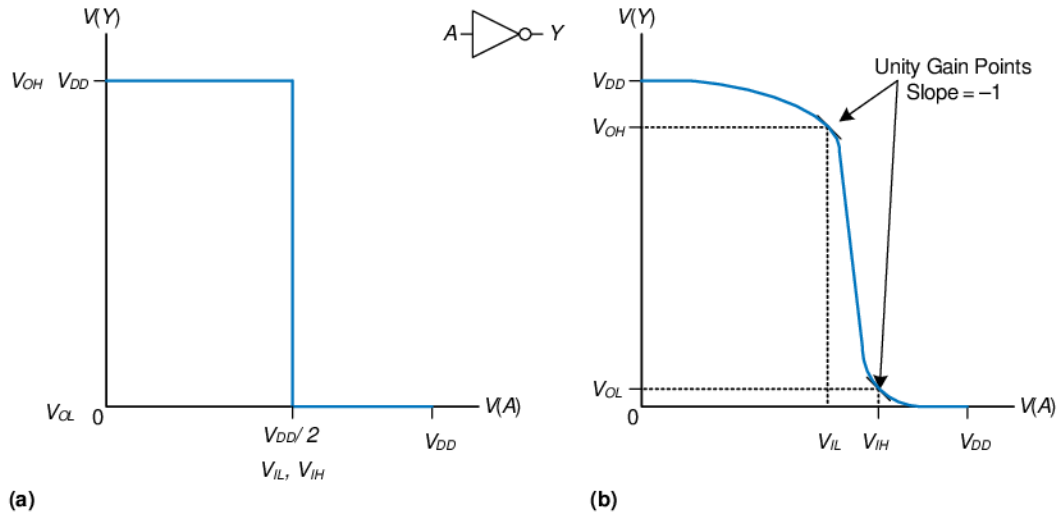


Figure 1.25 DC transfer characteristics and logic levels

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

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

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

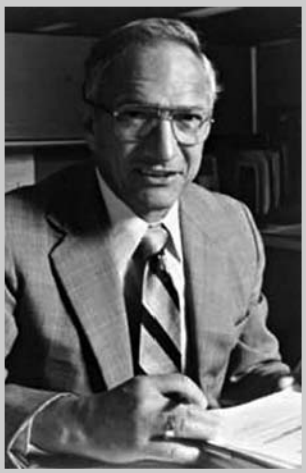
Table 1.4 Logic levels of 5 V and 3.3 V logic families

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

Table 1.5 Compatibility of logic families

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

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



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

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

Example 1.19 LOGIC FAMILY COMPATIBILITY

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

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

1.7 CMOS TRANSISTORS*

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

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

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

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

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

1.7.1 Semiconductors

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

1.7.2 Diodes

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

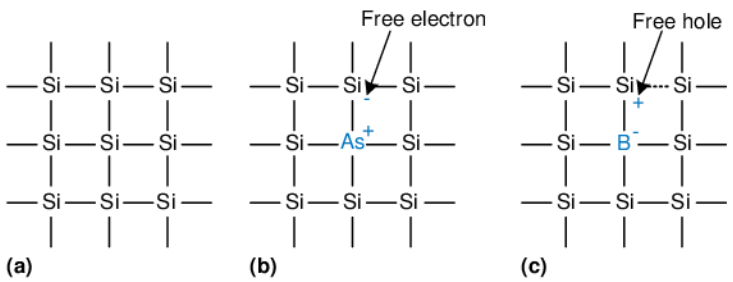


Figure 1.26 Silicon lattice and dopant atoms