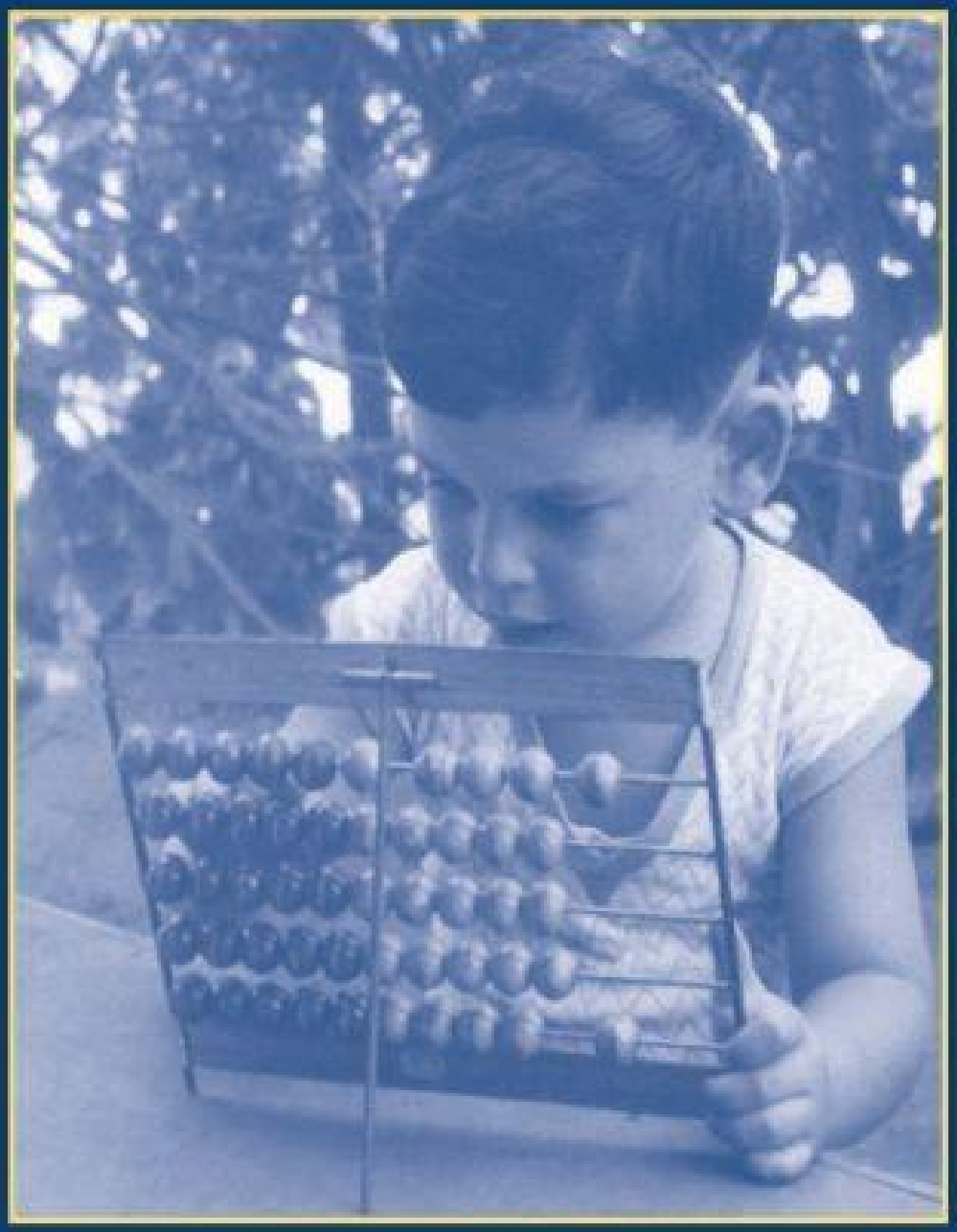
The Elements of Computing Systems

Building a Modern Computer from First Principles



Noam Nisan and Shimon Schocken

Table of Contents

Noam Nisan and Shimon Schocken

The Elements of Computing Systems

Building a Modern Computer from First Principles

The MIT Press
Cambridge, Massachusetts
London, England

This book was set in Times New Roman on 3B2 by Asco Typesetters, Hong Kong.
Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Nisan, Noam.
The elements of computing systems: building a modern computer from first principles / Noam Nisan and Shimon Schocken.
p. cm.
Includes bibliographical references and index.
ISBN 0-262-14087-X (alk. paper)
1. Electronic digital computers. I. Schocken, Shimon. II. Title.
TK7888.3.N57 2005
004.16—dc22
2005042807

10 9 8 7 6 5 4 3 2 1

Note on Software

The book's Web site (http://www.idc.ac.il/tecs) provides the tools and materials necessary to build all the hardware and software systems described in the book. These include a hardware simulator, a CPU emulator, a VM emulator, and executable versions of the assembler, virtual machine, compiler, and operating system described in the book. The Web site also includes all the project materials—about 200 test programs and test scripts, allowing incremental development and unit-testing of each one of the 12 projects. All the supplied software tools and project materials can be used as is on any computer equipped with either Windows or Linux.

To our parents,
For teaching us that less is more.

Preface

What I hear, I forget; What I see, I remember; What I do, I understand.

—Confucius, 551-479 BC

Once upon a time, every computer specialist had a gestalt understanding of how computers worked. The overall interactions among hardware, software, compilers, and the operating system were simple and transparent enough to produce a coherent picture of the computer's operations. As modern computer technologies have become increasingly more complex, this clarity is all but lost: the most fundamental ideas and techniques in computer science—the very essence of the field—are now hidden under many layers of obscure interfaces and proprietary implementations. An inevitable consequence of this complexity has been specialization, leading to computer science curricula of many courses, each covering a single aspect of the field.

We wrote this book because we felt that many computer science students are missing the forest for the trees. The typical student is marshaled through a series of courses in programming, theory, and engineering, without pausing to appreciate the beauty of the picture at large. And the picture at large is such that hardware and software systems are tightly interrelated through a hidden web of abstractions, interfaces, and contract-based implementations. Failure to see this intricate enterprise in the flesh leaves many students and professionals with an uneasy feeling that, well, they don't fully understand what's going on inside computers.

We believe that the best way to understand how computers work is to build one from scratch. With that in mind, we came up with the following concept. Let's specify a simple but sufficiently powerful computer system, and have the students build its hardware platform and software hierarchy from the ground up, starting with nothing more than elementary logic gates. And while we are at it, let's do it right. We say this because building a general-purpose computer from first principles is a huge undertaking. Therefore, we identified a unique educational opportunity not only to build the thing, but also to illustrate, in a hands-on fashion, how to effectively plan and manage large-scale hardware and software development projects. In addition, we sought to demonstrate the ability to construct, through recursive ascent and human reasoning, fantastically complex and useful systems from nothing more than a few primitive building blocks.

Scope

The book exposes students to a significant body of computer science knowledge, gained through a series of hardware and software construction tasks. These tasks demonstrate how theoretical and applied techniques taught in other computer science courses are used in practice. In particular, the following topics are illustrated in a hands-on fashion:

■ *Hardware:* Logic gates, Boolean arithmetic, multiplexors, flip-flops, registers, RAM units, counters, Hardware Description Language (HDL), chip simulation and testing.

■ *Architecture:* ALU/CPU design and implementation, machine code, assembly language programming, addressing modes, memory-mapped input/output (I/O).

■ *Operating systems:* Memory management, math library, basic I/O drivers, screen management, file I/O, high-level language support.

■ *Programming languages:* Object-based design and programming, abstract data types, scoping rules, syntax and semantics, references.

■ *Compilers:* Lexical analysis, top-down parsing, symbol tables, virtual stack-based machine, code generation, implementation of arrays and objects.

■ *Data structures and algorithms:* Stacks, hash tables, lists, recursion, arithmetic algorithms, geometric algorithms, running time considerations.

■ *Software engineering:* Modular design, the interface/implementation paradigm, API design and documentation, proactive test planning, programming at the large, quality assurance.

All these topics are presented with a very clear purpose: building a modern computer from the ground up. In fact, this has been our topic selection rule: The book focuses on the minimal set of topics necessary for building a fully functioning computer system. As it turns out, this set includes many fundamental ideas in applied computer science.

Courses

The book is intended for students of computer science and other engineering disciplines in colleges and universities, at both the undergraduate and graduate levels. A course based on this book is "perpendicular" to the normal computer science curriculum and can be taken at almost any point during the program. Two natural slots are "CS-2"—immediately after learning programming, and "CS-199"—a capstone course coming at the end of the program. The former course can provide a systems-oriented introduction to computer science, and the latter an integrative, project-oriented systems building course. Possible names for such courses may be Constructive Introduction to Computer Science, Elements of Computing Systems, Digital Systems Construction, Computer Construction Workshop, Let's Build a Computer, and the like. The book can support both one- and two-semester courses, depending on topic selection and pace of work.

The book is completely self-contained, requiring only programming (in any language) as a prerequisite. Thus, it lends itself not only to computer science majors, but also to computer-savvy students seeking to gain a hands-on view of hardware architectures, operating systems, and modern software engineering in the framework of one course. The book and the accompanying Web site can also be used as a self-study learning unit, suitable to students from any technical or scientific discipline following a programming course.

Structure

The introduction chapter presents our approach and previews the main hardware and software abstractions discussed in the book. This sets the stage for chapters 1-12, each dedicated to a key hardware or software abstraction, a proposed implementation, and an actual project that builds and tests it. The first five chapters focus on constructing the hardware platform of a simple modern computer. The remaining seven chapters describe the design and implementation of a typical multi-tier software hierarchy, culminating in the construction of an object-based programming language and a simple operating system. The complete game plan is depicted in figure P.1.

The book is based on an abstraction-implementation paradigm. Each chapter starts with a Background section, describing relevant concepts and a generic hardware or software system. The next section is always Specification, which provides a clear statement of the system's abstraction—namely, the various services that it is expected to deliver. Having presented the what, each chapter proceeds to discuss how the abstraction can be implemented, leading to a (proposed) Implementation section. The next section is always Perspective, in which we highlight noteworthy issues left out from the chapter. Each chapter ends with a Project section that provides step-by-step building instructions, testing materials, and software tools for actually building and unit-testing the system described in the chapter.

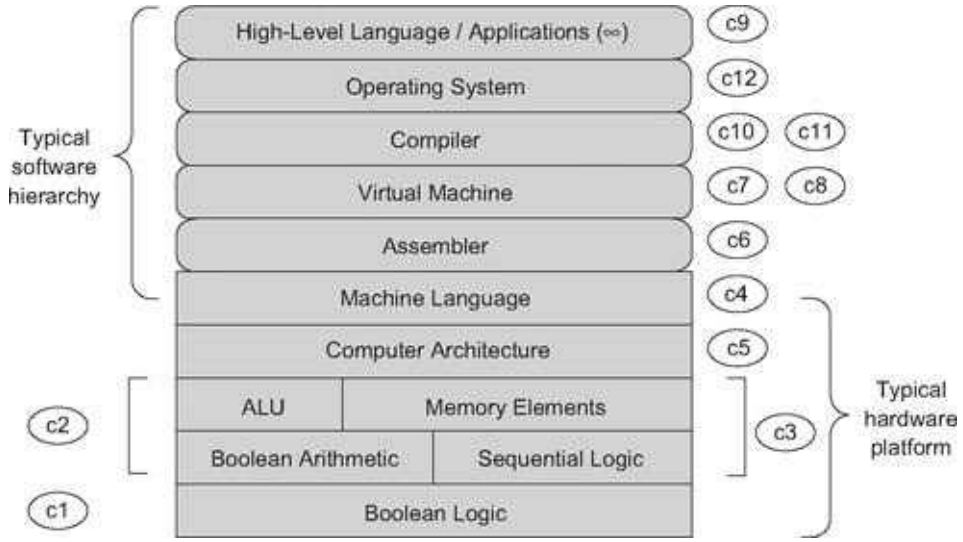


Figure P.1 Book and proposed course map, with chapter numbers in circles.

Projects

The computer system described in the book is *for real*—it can actually be built, and it works! A reader who takes the time and effort to gradually build this computer will gain a level of intimate understanding unmatched by mere reading. Hence, the book is geared toward active readers who are willing to roll up their sleeves and build a computer from the ground up.

Each chapter includes a complete description of a stand-alone hardware or software development project. The four projects that construct the computer platform are built using a simple Hardware Description Language (HDL) and simulated on a hardware simulator supplied with the book. Five of the subsequent software projects (assembler, virtual machine I and II, and compiler I and II) can be written in any modern programming language. The remaining three projects (low-level programming, high-level programming, and the operating system) are written in the assembly language and high-level language implemented in previous projects.

Project Tips There are twelve projects altogether. On average, each project entails a weekly homework load in a typical, rigorous university-level course. The projects are completely self-contained and can be done (or skipped) in any desired order. Of course the "full experience" package requires doing all the projects in their order of appearance, but this is just one option.

When we teach courses based on this book, we normally make two significant concessions. First, except for obvious cases, we pay no attention to optimization, leaving this very important subject to other, more specific courses. Second, when developing the translators suite (assembler, VM implementation, and compiler), we supply error-free test files (source programs), allowing the students to assume that the inputs of these translators are error-free. This eliminates the need to write code for handling errors and exceptions, making the software projects significantly more manageable. Dealing with incorrect input is of course critically important, but once again we assume that students can hone this skill elsewhere, for example, in dedicated programming and software design courses.

Software

The book's Web site (www.idc.ac.il/tecs) provides the tools and materials necessary to build all the hardware and software systems described in the book. These include a hardware simulator, a CPU emulator, a VM emulator, and executable versions of the assembler, virtual machine, compiler, and operating system described in the book. The Web site also includes all the project materials—about two hundred test programs and test scripts, allowing incremental development and unit-testing of each one of the twelve projects. All the supplied software tools and project materials can be used as is on any computer equipped with either Windows or Linux.

Acknowledgments

All the software that accompanies the book was developed by our students at the Efi Arazi School of Computer Science of the Interdisciplinary Center Herzliya, a new Israeli university. The chief software architect was Yaron Ukrainitz, and the developers included Iftach Amit, Nir Rozen, Assaf Gad, and Hadar Rosen-Sior. Working with these student-developers has been a great pleasure, and we feel proud and fortunate to have had the opportunity to play a role in their education. We also wish to thank our teaching assistants, Muawyah Akash, David Rabinowitz, Ran Navok, and Yaron Ukrainitz, who helped us run early versions of the course that led to this book. Thanks also to Jonathan Gross and Oren Baranes, who worked on related projects under the excellent supervision of Dr. Danny Seidner, to Uri Zeira and Oren Cohen, for designing an integrated development environment for the Jack language, to Tal Achituv, for useful advice on open source issues, and to Aryeh Schnall, for careful reading and meticulous editing suggestions.

Writing the book without taking any reduction in our regular professional duties was not simple, and so we wish to thank esti romem, administrative director of the EFI Arazi School of Computer Science, for holding the fort in difficult times. Finally, we are indebted to the many students who endured early versions of this book and helped polish it through numerous bug reports. In the process, we hope, they have learned first-hand that insight of James Joyce, that mistakes are the portals of discovery.

Noam Nisan

Shimon Schocken

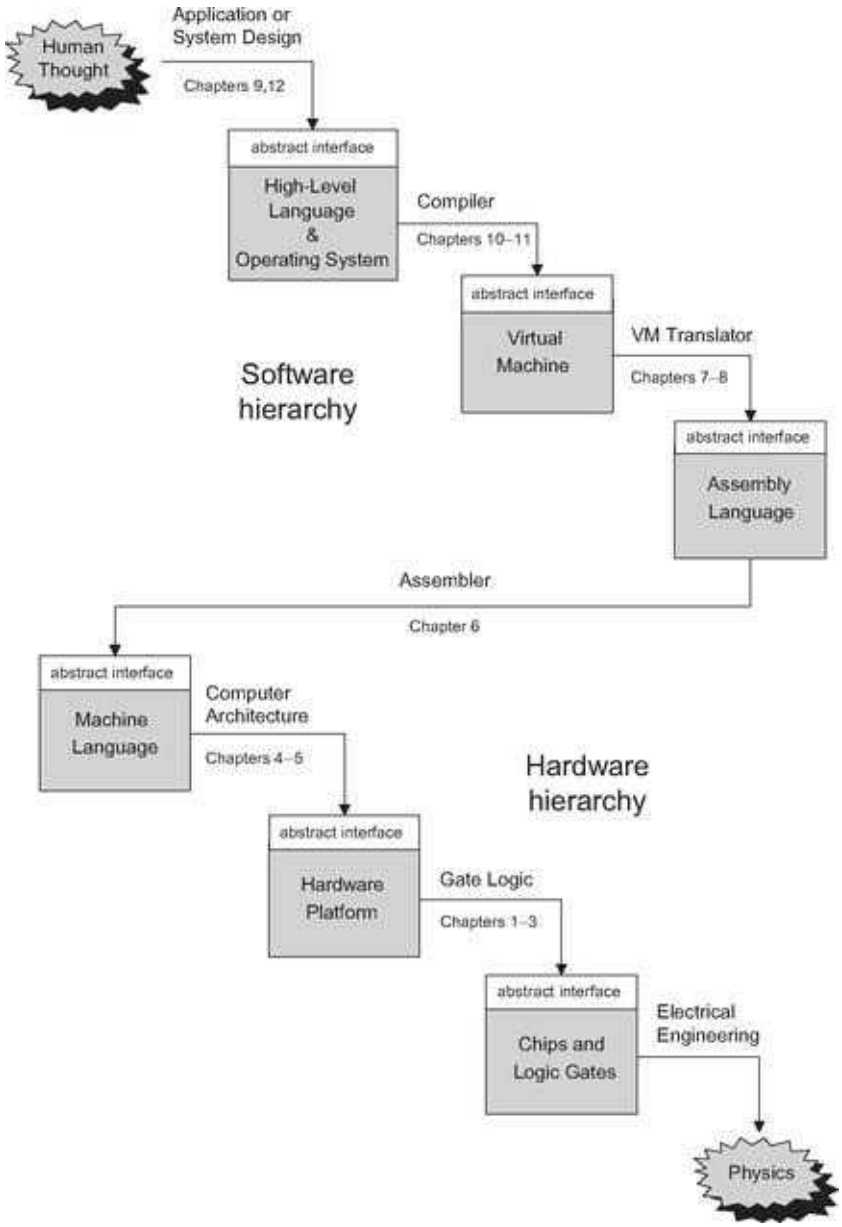


Figure I.1 The major abstractions underlying the design of a typical computing system. The implementation of each level is accomplished using abstract services and building blocks from the level below.

Introduction: Hello, World Below

The true voyage of discovery consists not of going to new places, but of having a new pair of eyes.

—Marcel Proust (1871-1922)

This book is a voyage of discovery. You are about to learn three things: how computers work, how to

break complex problems into manageable modules, and how to develop large-scale hardware and software systems. This will be a hands-on process as you create a complete and working computer system from the ground up. The lessons you will learn, which are far more important and general than the computer itself, will be gained as side effects of this activity. According to the psychologist Carl Rogers, "the only kind of learning which significantly influences behavior is self-discovered or self-appropriated —truth that has been assimilated in experience." This chapter sketches some of the discoveries, truths, and experiences that lie ahead.

The World Above

If you have taken any programming course, you've probably encountered something like the program below early in your education. This particular program is written in *Jack*—a simple high-level language that has a conventional object-based syntax.

```
// First example in Programming 101:
class Main {
  function void main() {
    do Output.printString("Hello World");
    do Output.println(); // New line.
    return;
  }
}
```

Trivial programs like Hello World are deceptively simple. Did you ever think about what it takes to actually run such a program on a computer? Let's look under the hood. For starters, note that the program is nothing more than a bunch of dead characters stored in a text file. Thus, the first thing we must do is parse this text, uncover its semantics, and reexpress it in some low-level language understood by our computer. The result of this elaborate translation process, known as compilation, will be yet another text file, containing machine-level code.

Of course machine language is also an abstraction—an agreed upon set of binary codes. In order to make this abstract formalism concrete, it must be realized by some hardware architecture. And this architecture, in turn, is implemented by a certain chip *set*—registers, memory units, ALU, and so on. Now, every one of these hardware devices is constructed from an integrated package of elementary logic gates. And these gates, in turn, can be built from primitive gates like Nand and Nor. Of course every one of these gates consists of several switching devices, typically implemented by transistors. And each transistor is made of—Well, we won't go further than that, because that's where computer science ends and physics starts.

You may be thinking: "On my computer, compiling and running a program is much easier—all I have to do is click some icons or write some commands!" Indeed, a modern computer system is like a huge iceberg, and most people get to see only the top. Their knowledge of computing systems is sketchy and superficial. If, however, you wish to explore beneath the surface, then lucky you! There's a fascinating world down there, made of some of the most beautiful stuff in computer science. An intimate understanding of this underworld is one of the things that separate naïve programmers from sophisticated developers—people who can create not only application programs, but also complex hardware and software technologies. And the best way to understand how these technologies work—and we mean understand them in the marrow of your bones—is to build a complete computer system from scratch.

Abstractions

You may wonder how it is humanly possible to construct a complete computer system from the ground up, starting with nothing more than elementary logic gates. This must be an enormously complex enterprise! We deal with this complexity by breaking the project into modules, and treating each module separately, in a stand-alone chapter. You might then wonder, how is it possible to describe and construct these modules in isolation? Obviously they are all interrelated! As we will show throughout the book, a good modular design implies just that: You can work on the individual modules independently, while completely ignoring the rest of the system. In fact, you can even build these modules in any desired order!

It turns out that this strategy works well thanks to a special gift unique to humans: our ability to create and use abstractions. The notion of abstraction, central to many arts and sciences, is normally taken to be a mental expression that seeks to separate in thought, and capture in some concise manner, the essence of some entity. In computer science, we take the notion of abstraction very concretely, defining it to be a statement of "what the entity does" and ignoring the details of "how it does it." This functional description must capture all that needs to be known in order to use the entity's services, and nothing more. All the work, cleverness, information, and drama that went into the entity's implementation are concealed from the client who is supposed to use it, since they are simply irrelevant. The articulation, use, and implementation of such abstractions are the bread and butter of our professional practice: Every hardware and software developer is routinely defining abstractions (also called "interfaces") and then implementing them, or asking other people to implement them. The abstractions are often built layer upon layer, resulting in higher and higher levels of capabilities.

Designing good abstractions is a practical art, and one that is best acquired by seeing many examples. Therefore, this book is based on an abstraction-implementation paradigm. Each book chapter presents a key hardware or software abstraction, and a project designed to actually implement it. Thanks to the modular nature of these abstractions, each chapter also entails a stand-alone intellectual unit, inviting the reader to focus on two things only: understanding the given abstraction (a rich world of its own), and then implementing it using abstract services and building blocks from the level below. As you push ahead in this journey, it will be rather thrilling to look back and appreciate the computer that is gradually taking shape in the wake of your efforts.

The World Below

The multi-tier collection of abstractions underlying the design of a computing system can be described top-down, showing how high-level abstractions can be reduced into, or expressed by, simpler ones. This structure can also be described bottom-up, focusing on how lower-level abstractions can be used to construct more complex ones. This book takes the latter approach: We begin with the most basic elements —primitive logic gates—and work our way upward, culminating in the construction of a general-purpose computer system. And if building such a computer is like climbing Mount Everest, then planting a flag on the mountaintop is like having the computer run a program written in some high-level language. Since we are going to ascend this mountain from the ground up, let us survey the book plan in the opposite direction —from the top down—starting in the familiar territory of high-level programming.

Our tour consists of three main legs. We start at the top, where people write and run high-level programs (chapters 9 and 12). We then survey the road down to hardware land, tracking the fascinating twists and curves of translating high-level programs into machine language (chapters 6, 7, 8, 10, 11). Finally, we reach the low grounds of our journey, describing how a typical hardware platform is actually constructed (chapters 1-5).

High-Level Language Land

The topmost abstraction in our journey is the art of programming, where entrepreneurs and programmers dream up applications and write software that implements them. In doing so, they blissfully take for granted the two key tools of their trade: the high-level language in which they work, and the rich library of services that supports it. For example, consider the statement do Output.printString(′ ′Hello World′ ′). This code invokes an abstract service for printing strings—a service that must be implemented somewhere. Indeed, a bit of drilling reveals that this service is usually supplied jointly by the host operating system and the standard language library.

What then is a standard language library? And how does an operating system (OS) work? These questions are taken up in chapter 12. We start by presenting key algorithms relevant to OS services, and then use them to implement various mathematical functions, string operations, memory allocation tasks, and input/output (I/O) routines. The result is a simple operating system, written in the Jack programming language.

Jack is a simple object-based language, designed for a single purpose: to illustrate the key software engineering principles underlying the design and implementation of modern programming languages like Java and C#. Jack is presented in chapter 9, which also illustrates how to build Jack-based applications, for example, computer games. If you have any programming experience with a modern object-oriented language, you can start writing Jack programs right away and watch them execute on the computer platform developed in previous chapters. However, the goal of chapter 9 is not to turn you into a Jack programmer, but rather to prepare you to develop the compiler and operating system described in subsequent chapters.

The Road Down to Hardware Land

Before any program can actually run and do something for real, it must be translated into the machine language of some target computer platform. This compilation process is sufficiently complex to be broken into several layers of abstraction, and these usually involve three translators: a compiler, a virtual machine implementation, and an assembler. We devote five book chapters to this trio, as follows.

The translation task of the compiler is performed in two conceptual stages: syntax analysis and code generation. First, the source text is analyzed and grouped into meaningful language constructs that can be kept in a data structure called a "parse tree." These parsing tasks, collectively known as syntax analysis, are described in chapter 10. This sets the stage for chapter 11, which shows how the parse tree can be recursively processed to yield a program written in an intermediate language. As with Java and C#, the intermediate code generated by the Jack compiler describes a sequence of generic steps operating on a stack-based virtual machine (VM). This classical model, as well as a VM implementation that realizes it on an actual computer, are elaborated in chapters 7-8. Since the output of our VM implementation is a large assembly program, we have to translate it further into binary code. Writing an assembler is a relatively simple task, taken up in chapter 6.

Hardware Land

We have reached the most profound step in our journey—the descent from machine language to the machine itself—the point where software finally meets hardware. This is also the point where Hack enters the picture. Hack is a general-purpose computer system, designed to strike a balance between simplicity and power. On the one hand, the Hack architecture can be built in just a few hours of work, using the guidelines and chip set presented in chapters 1-3. At the same time, Hack is sufficiently general to illustrate the key operating principles and hardware elements underlying the design of any digital computer.

The machine language of the Hack platform is specified in chapter 4, and the computer design itself is discussed and specified in chapter 5. Readers can build this computer as well as all the chips and gates mentioned in the book on their home computers, using the software-based hardware simulator supplied with the book and the Hardware Description Language (HDL) documented in appendix A. All the developed hardware modules can be tested using supplied test scripts, written in a scripting language documented in appendix B.

The computer that emerges from this construction is based on typical components like CPU, RAM, ROM, and simulated screen and keyboard. The computer's registers and memory systems are built in chapter 3, following a brief discussion of sequential logic. The computer's combinational logic, culminating in the Arithmetic Logic Unit (ALU) chip, is built in chapter 2, following a brief discussion of Boolean arithmetic. All the chips presented in these chapters are based on a suite of elementary logic gates, presented and built in chapter 1.

Of course the layers of abstraction don't stop here. Elementary logic gates are built from transistors, using technologies based on solid-state physics and ultimately quantum mechanics. Indeed, this is where the abstractions of the natural world, as studied and formulated by physicists, become the building blocks of the abstractions of the synthetic worlds built and studied by computer scientists.

This marks the end of our grand tour preview—the descent from the high-level peaks of object-based software, all the way down to the bricks and mortar of the hardware platform. This typical modular rendition of a multi-tier system represents not only a powerful engineering paradigm, but also a central dogma in human reasoning, going back at least 2,500 years:

We deliberate not about ends, but about means. For a doctor does not deliberate whether he shall heal, nor an orator whether he shall persuade ... They assume the end and consider how and by what means it is attained, and if it seems easily and best produced thereby; while if it is achieved by other means, they consider how it will be achieved and by what means this will be achieved, until they come to the first cause ... and what is last in the order of analysis seems to be first in the order of becoming. (Aristotles, Nicomachean Ethics, Book III, 3, 1112b)

So here's the plan, in the order of becoming. Starting with the construction of elementary logic gates (chapter 1), we go bottom-up to combinational and sequential chips (chapters 2-3), through the design of a typical computer architecture (chapters 4-5) and a typical software hierarchy (chapters 6-8), all the way to implementing a compiler (chapters 10-11) for a modern object-based language (chapter 9), ending with the design and implementation of a simple operating system (chapter 12). We hope that the reader has gained a general idea of what lies ahead and is eager to push forward on this grand tour of discovery. So, assuming that you are ready and set, let the countdown start: 1, 0, Go!

1

Boolean Logic

Such simple things, And we make of them something so complex it defeats us, Almost.

—John Ashbery (b. 1927), American poet

Every digital device—be it a personal computer, a cellular telephone, or a network router—is based on a set of chips designed to store and process information. Although these chips come in different shapes and forms, they are all made from the same building blocks: Elementary logic gates. The gates can be physically implemented in many different materials and fabrication technologies, but their logical behavior is consistent across all computers. In this chapter we start out with one primitive logic gate—Nand—and build all the other logic gates from it. The result is a rather standard set of gates, which will be later used to construct our computer's processing and storage chips. This will be done in chapters 2 and 3, respectively.

All the hardware chapters in the book, beginning with this one, have the same structure. Each chapter focuses on a well-defined task, designed to construct or integrate a certain family of chips. The prerequisite knowledge needed to approach this task is provided in a brief Background section. The next section provides a complete Specification of the chips' abstractions, namely, the various services that they should deliver, one way or another. Having presented the what, a subsequent Implementation section proposes guidelines and hints about how the chips can be actually implemented. A Perspective section rounds up the chapter with concluding comments about important topics that were left out from the discussion. Each chapter ends with a technical Project section. This section gives step-by-step instructions for actually building the chips on a personal computer, using the hardware simulator supplied with the book.

This being the first hardware chapter in the book, the Background section is somewhat lengthy, featuring a special section on hardware description and simulation tools.

1.1 Background

This chapter focuses on the construction of a family of simple chips called Boolean gates. Since Boolean gates are physical implementations of Boolean functions, we start with a brief treatment of Boolean algebra. We then show how Boolean gates implementing simple Boolean functions can be interconnected to deliver the functionality of more complex chips. We conclude the background section with a description of how hardware design is actually done in practice, using software simulation tools.

1.1.1 Boolean Algebra

Boolean algebra deals with Boolean (also called binary) values that are typically labeled true/false, 1/0, yes/no, on/off, and so forth. We will use 1 and 0. A Boolean function is a function that operates on binary inputs and returns binary outputs. Since computer hardware is based on the representation and manipulation of binary values, Boolean functions play a central role in the specification, construction, and optimization of hardware architectures. Hence, the ability to formulate and analyze Boolean functions is the first step toward constructing computer architectures.

Truth Table Representation The simplest way to specify a Boolean function is to enumerate all the possible values of the function's input variables, along with the function's output for each set of inputs. This is called the truth table representation of the function, illustrated in figure 1.1.

The first three columns of figure 1.1 enumerate all the possible binary values of the function's variables. For each one of the 2^n possible tuples $v_1 \ldots v_n$ (here $n = 3$), the last column gives the value of $f(v_1 \ldots v_n)$.

Boolean Expressions In addition to the truth table specification, a Boolean function can also be specified using Boolean operations over its input variables. The basic Boolean operators that are typically used are "And" (x And y is 1 exactly when both x and y are 1) "Or" (x Or y is 1 exactly when either x or y or both are 1), and "Not" (Not x is 1 exactly when x is 0). We will use a common arithmetic-like notation for these operations: $x \cdot y$ (or xy) means x And y, $x + y$ means x Or y, and $\bar{x}$ means Not x.

To illustrate, the function defined in figure 1.1 is equivalently given by the Boolean expression $f(x, y, z) = (x + y) \cdot \bar{z}$. For example, let us evaluate this expression on the inputs $x = 0$, $y = 1$, $z = 0$ (third row in the table). Since y is 1, it follows that x + y = 1 and thus $\bar{0}$. The complete verification of the equivalence between the expression and the truth table is achieved by evaluating the expression on each of the eight possible input combinations, verifying that it yields the same value listed in the table's right column.

x	y	z	$f(x, y, z)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Figure 1.1 Truth table representation of a Boolean function (example).

Canonical Representation As it turns out, every Boolean function can be expressed using at least one

Boolean expression called the *canonical representation*. Starting with the function's truth table, we focus on all the rows in which the function has value 1. For each such row, we construct a term created by And-ing together literals (variables or their negations) that fix the values of all the row's inputs. For example, let us focus on the third row in figure 1.1, where the function's value is 1. Since the variable values in this row are $x = 0, y = 1, z = 0$, we construct the term $\bar{x}y\bar{z}$. Following the same procedure, we construct the terms $x\bar{y}\bar{z}$ and $xy\bar{z}$ for rows 5 and 7. Now, if we Or-together all these terms (for all the rows where the function has value 1), we get a Boolean expression that is equivalent to the given truth table. Thus the canonical representation of the Boolean function shown in figure 1.1 is $f(x, y, z) = \bar{x}y\bar{z} + x\bar{y}\bar{z} + xy\bar{z}$. This construction leads to an important conclusion: Every Boolean function, no matter how complex, can be expressed using three Boolean operators only: And, Or, and Not.

Two-Input Boolean Functions An inspection of figure 1.1 reveals that the number of Boolean functions that can be defined over n binary variables is 2^{2^n}. For example, the sixteen Boolean functions spanned by two variables are listed in figure 1.2. These functions were constructed systematically, by enumerating all the possible 4-wise combinations of binary values in the four right columns. Each function has a conventional name that seeks to describe its underlying operation. Here are some examples: The name of the Nor function is shorthand for Not-Or: Take the Or of *x* and y, then negate the result. The Xor function —shorthand for "exclusive or"—returns 1 when its two variables have opposing truth-values and 0 otherwise. Conversely, the Equivalence function returns 1 when the two variables have identical truth-values. The If-*x*-then-*y* function (also known as $x \rightarrow$ y, or "*x* Implies *y*") returns 1 when *x* is 0 or when both *x* and *y* are 1. The other functions are self-explanatory.

Function	*x*	0	0	1	1
	y	0	1	0	1
Constant 0	0	0	0	0	0
And	$x \cdot y$	0	0	0	1
x And Not *y*	$x \cdot \bar{y}$	0	0	1	0
x	*x*	0	0	1	1
Not *x* And *y*	$\bar{x} \cdot y$	0	1	0	0
y	*y*	0	1	0	1
Xor	$x \cdot \bar{y} + \bar{x} \cdot y$	0	1	1	0
Or	$x + y$	0	1	1	1
Nor	$\overline{x + y}$	1	0	0	0
Equivalence	$x \cdot y + \bar{x} \cdot \bar{y}$	1	0	0	1
Not *y*	$\bar{y}$	1	0	1	0
If *y* then *x*	$x + \bar{y}$	1	0	1	1
Not *x*	$\bar{x}$	1	1	0	0
If *x* then *y*	$\bar{x} + y$	1	1	0	1
Nand	$\overline{x \cdot y}$	1	1	1	0
Constant 1	1	1	1	1	1

Figure 1.2 All the Boolean functions of two variables.

The Nand function (as well as the Nor function) has an interesting theoretical property: Each one of the operations And, Or, and Not can be constructed from it, and it alone (e.g., *x* Or *y* = (*x* Nand x) Nand (*y* Nand y). And since every Boolean function can be constructed from And, Or, and Not operations using the

canonical representation method, it follows that every Boolean function can be constructed from Nand operations alone. This result has far-reaching practical implications: Once we have in our disposal a physical device that implements Nand, we can use many copies of this device (wired in a certain way) to implement in hardware any Boolean function.

1.1.2 Gate Logic

A *gate* is a physical device that implements a Boolean function. If a Boolean function f operates on n variables and returns *m* binary results (in all our examples so far, m was 1), the gate that implements f will have n input pins and m output pins. When we put some values v_1 ...v_n in the gate's input pins, the gate's "logic"—its internal structure—should compute and output $f(v_1 \ldots v_n)$. And just like complex Boolean functions can be expressed in terms of simpler functions, complex gates are composed from more elementary gates. The simplest gates of all are made from tiny switching devices, called transistors, wired in a certain topology designed to effect the overall gate functionality.

Although most digital computers today use electricity to represent and transmit binary data from one gate to another, any alternative technology permitting switching and conducting capabilities can be employed. Indeed, during the last fifty years, researchers have built many hardware implementations of Boolean functions, including magnetic, optical, biological, hydraulic, and pneumatic mechanisms. Today, most gates are implemented as transistors etched in silicon, packaged as chips. In this book we use the words chip and gate interchangeably, tending to use the term gates for simple chips.

The availability of alternative switching technology options, on the one hand, and the observation that Boolean algebra can be used to abstract the behavior of any such technology, on the other, is extremely important. Basically, it implies that computer scientists don't have to worry about physical things like electricity, circuits, switches, relays, and power supply. Instead, computer scientists can be content with the abstract notions of Boolean algebra and gate logic, trusting that someone else (the physicists and electrical engineers—bless their souls) will figure out how to actually realize them in hardware. Hence, a primitive gate (see figure 1.3) can be viewed as a black box device that implements an elementary logical operation in one way or another—we don't care how. A hardware designer starts from such primitive gates and designs more complicated functionality by interconnecting them, leading to the construction of composite gates.

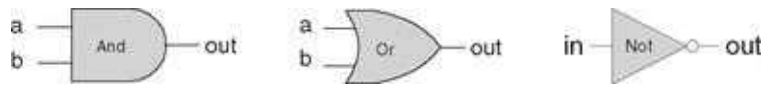


Figure 1.3 Standard symbolic notation of some elementary logic gates.

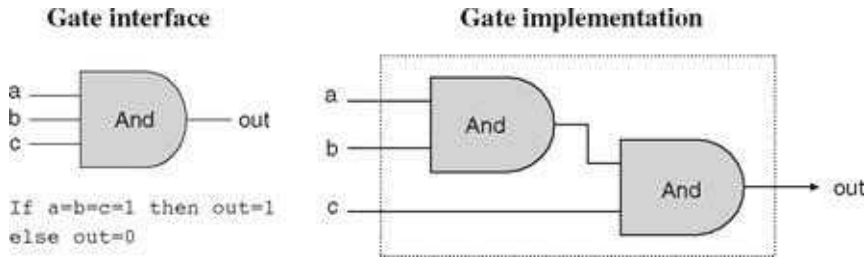


Figure 1.4 Composite implementation of a three-way And gate. The rectangle on the right defines the

conceptual boundaries of the gate interface.

Primitive and Composite Gates Since all logic gates have the same input and output semantics (0's and 1's), they can be chained together, creating composite gates of arbitrary complexity. For example, suppose we are asked to implement the 3-way Boolean function And(*a, b, c*). Using Boolean algebra, we can begin by observing that $a \cdot b \cdot c = (a \cdot b) \cdot c$, or, using prefix notation, And(*a*, b, c) = And(And(*a, b*), *c*). Next, we can use this result to construct the composite gate depicted in figure 1.4.

The construction described in figure 1.4 is a simple example of gate logic, also called logic design. Simply put, logic design is the art of interconnecting gates in order to implement more complex functionality, leading to the notion of composite gates. Since composite gates are themselves realizations of (possibly complex) Boolean functions, their "outside appearance" (e.g., left side of figure 1.4) looks just like that of primitive gates. At the same time, their internal structure can be rather complex.

We see that any given logic gate can be viewed from two different perspectives: external and internal. The right-hand side of figure 1.4 gives the gate's internal architecture, or implementation, whereas the left side shows only the gate interface, namely, the input and output pins that it exposes to the outside world. The former is relevant only to the gate designer, whereas the latter is the right level of detail for other designers who wish to use the gate as an abstract off-the-shelf component, without paying attention to its internal structure.

Let us consider another logic design example—that of a Xor gate. As discussed before, Xor(*a*, b) is 1 exactly when either a is 1 and *b* is 0, or when a is 0 and *b* is 1. Said otherwise, Xor(*a, b*) = Or(And(*a*, Not(*b*)), And(Not(*a*), *b*)). This definition leads to the logic design shown in figure 1.5.

Note that the gate interface is unique: There is only one way to describe it, and this is normally done using a truth table, a Boolean expression, or some verbal specifica-tion. This interface, however, can be realized using many different implementations, some of which will be better than others in terms of cost, speed, and simplicity. For example, the Xor function can be implemented using four, rather than five, And, Or, and Not gates. Thus, from a functional standpoint, the fundamental requirement of logic design is that *the gate implementation will realize its stated interface, in one way or another*. From an efficiency standpoint, the general rule is to try to do more with less, that is, use as few gates as possible.

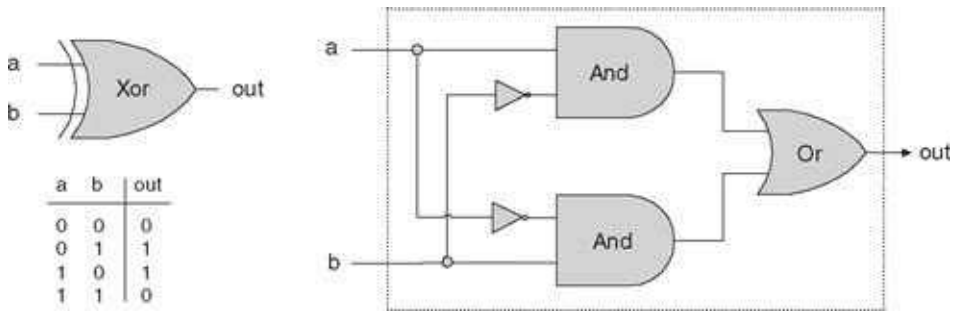


Figure 1.5 Xor gate, along with a possible implementation.

To sum up, the art of logic design can be described as follows: Given a gate specification (interface), find an efficient way to implement it using other gates that were already implemented. This, in a nutshell, is what we will do in the rest of this chapter.

1.1.3 Actual Hardware Construction

Having described the logic of composing complex gates from simpler ones, we are now in a position to discuss how gates are actually built. Let us start with an intentionally naïve example.

Suppose we open a chip fabrication shop in our home garage. Our first contract is to build a hundred Xor gates. Using the order's downpayment, we purchase a soldering gun, a roll of copper wire, and three bins labeled "And gates," "Or gates," and "Not gates," each containing many identical copies of these elementary logic gates. Each of these gates is sealed in a plastic casing that exposes some input and output pins, as well as a power supply plug. To get started, we pin figure 1.5 to our garage wall and proceed to realize it using our hardware. First, we take two And gates, two Not gates, and one Or gate, and mount them on a board according to the figure's layout. Next, we connect the chips to one another by running copper wires among them and by soldering the wire ends to the respective input/output pins. Now, if we follow the gate diagram carefully, we will end up having three exposed wire ends. We then solder a pin to each one of these wire ends, seal the entire device (except for the three pins) in a plastic casing, and label it "Xor." We can repeat this assembly process many times over. At the end of the day, we can store all the chips that we've built in a new bin and label it "Xor gates." If we (or other people) are asked to construct some other chips in the future, we'll be able to use these Xor gates as elementary building blocks, just as we used the And, Or, and Not gates before.

As the reader has probably sensed, the garage approach to chip production leaves much to be desired. For starters, there is no guarantee that the given chip diagram is correct. Although we can prove correctness in simple cases like Xor, we cannot do so in many realistically complex chips. Thus, we must settle for empirical testing: Build the chip, connect it to a power supply, activate and deactivate the input pins in various configurations, and hope that the chip outputs will agree with its specifications. If the chip fails to deliver the desired outputs, we will have to tinker with its physical structure—a rather messy affair. Further, even if we will come up with the right design, replicating the chip assembly process many times over will be a time-consuming and error-prone affair. There must be a better way!

1.1.4 Hardware Description Language (HDL)

Today, hardware designers no longer build anything with their bare hands. Instead, they plan and optimize the chip architecture on a computer workstation, using structured modeling formalisms like Hardware Description Language, or HDL (also known as VHDL, where V stands for *Virtual*). The designer specifies the chip structure by writing an HDL program, which is then subjected to a rigorous battery of tests. These tests are carried out virtually, using computer simulation: A special software tool, called a hardware simulator, takes the HDL program as input and builds an image of the modeled chip in memory. Next, the designer can instruct the simulator to test the virtual chip on various sets of inputs, generating simulated chip outputs. The outputs can then be compared to the desired results, as mandated by the client who ordered the chip built.

In addition to testing the chip's correctness, the hardware designer will typically be interested in a variety of parameters such as speed of computation, energy consumption, and the overall cost implied by the chip design. All these parameters can be simulated and quantified by the hardware simulator, helping the designer optimize the design until the simulated chip delivers desired cost/performance levels.

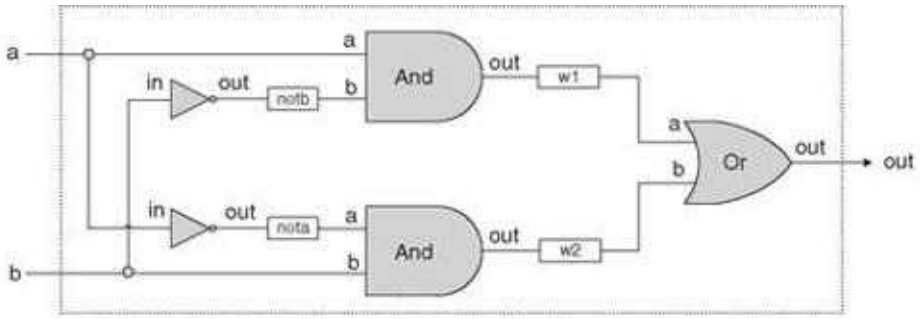
Thus, using HDL, one can completely plan, debug, and optimize the entire chip before a single penny is spent on actual production. When the HDL program is deemed complete, that is, when the performance of the simulated chip satisfies the client who ordered it, the HDL program can become the blueprint from which many copies of the physical chip can be stamped in silicon. This final step in the chip life cycle—from an optimized HDL program to mass production—is typically out-sourced to companies that specialize in chip fabrication, using one switching technology or another.

Example: Building a Xor Gate As we have seen in figures 1.2 and 1.5, one way to define exclusive or is Xor(*a*, *b*) = Or(And(*a*, Not(*b*)), And(Not(*a*), *b*)). This logic can be expressed either graphically, as a gate diagram, or textually, as an HDL program. The latter program is written in the HDL variant used throughout this book, defined in appendix A. See figure 1.6 for the details.

Explanation An HDL definition of a chip consists of a header section and a parts section. The header section specifies the chip interface, namely the chip name and the names of its input and output pins. The parts section describes the names and topology of all the lower-level parts (other chips) from which this chip is constructed. Each part is represented by a statement that specifies the part name and the way it is connected to other parts in the design. Note that in order to write such statements legibly, the HDL programmer must have a complete documentation of the underlying parts' interfaces. For example, figure 1.6 assumes that the input and output pins of the Not gate are labeled in and out, and those of And and Or are labeled a, b and out. This API-type information is not obvious, and one must have access to it before one can plug the chip parts into the present code.

Inter-part connections are described by creating and connecting internal pins, as needed. For example, consider the bottom of the gate diagram, where the output of a Not gate is piped into the input of a subsequent And gate. The HDL code describes this connection by the pair of statements Not(...,out=nota) and And(a=nota,...). The first statement creates an internal pin (outbound wire) named nota, feeding out into it. The second statement feeds the value of nota into the a input of an And gate. Note that pins may have an unlimited fan out. For example, in figure 1.6, each input is simultaneously fed into two gates. In

gate diagrams, multiple connections are described using forks. In HDL, the existence of forks is implied by the code.



HDL program (Xor.hdl)

```
/* Xor (exclusive or) gate:
   If a<>b out=1 else out=0. */
CHIP Xor {
   IN a, b;
   OUT out;
   PARTS:
   Not(in=a, out=nota);
   Not(in=b, out=notb);
   And(a=a, b=notb, out=w1);
   And(a=nota, b=b, out=w2);
   Or(a=w1, b=w2, out=out);
}
```

Test script (Xor.tst)

```
load Xor.hdl,
output-list a, b, out;
set a 0, set b 0,
eval, output;
set a 0, set b 1,
eval, output;
set a 1, set b 0,
eval, output;
set a 1, set b 1,
eval, output;
```

Output file (Xor.out)

```
a | b | out
- - - - - - -
0 | 0 | 0
0 | 1 | 1
1 | 0 | 1
1 | 1 | 0
```

Figure 1.6 HDL implementation of a Xor gate.

Testing Rigorous quality assurance mandates that chips be tested in a specific, replicable, and well-documented fashion. With that in mind, hardware simulators are usually designed to run test scripts, written in some scripting language. For example, the test script in figure 1.6 is written in the scripting language understood by the hardware simulator supplied with the book. This scripting language is described fully in appendix B.

Let us give a brief description of the test script from figure 1.6. The first two lines of the test script instruct the simulator to load the Xor.hdl program and get ready to print the values of selected variables. Next, the script lists a series of testing scenarios, designed to simulate the various contingencies under which the Xor chip will have to operate in "real-life" situations. In each scenario, the script instructs the simulator to bind the chip inputs to certain data values, compute the resulting output, and record the test results in a designated output file. In the case of simple gates like Xor, one can write an exhaustive test script that enumerates all the possible input values of the gate. The resulting output file (right side of figure 1.6) can then be viewed as a complete empirical proof that the chip is well designed. The luxury of such certitude is not feasible in more complex chips, as we will see later.

1.1.5 Hardware Simulation

Since HDL is a hardware construction language, the process of writing and debugging HDL programs is quite similar to software development. The main difference is that instead of writing code in a language like Java, we write it in HDL, and instead of using a compiler to translate and test the code, we use a hardware simulator. The hardware simulator is a computer program that knows how to parse and interpret HDL code, turn it into an executable representation, and test it according to the specifications of a given test script. There exist many commercial hardware simulators on the market, and these vary greatly in terms of cost, complexity, and ease of use. Together with this book we provide a simple (and free!) hardware simulator that is sufficiently powerful to support sophisticated hardware design projects. In particular, the simulator provides all the necessary tools for building, testing, and integrating all the chips presented in the book, leading to the construction of a general-purpose computer. Figure 1.7 illustrates a typical chip simulation session.

1.2 Specification

This section specifies a typical set of gates, each designed to carry out a common Boolean operation. These gates will be used in the chapters that follow to construct the full architecture of a typical modern computer. Our starting point is a single primitive Nand gate, from which all other gates will be derived recursively. Note that we provide only the gates' specifications, or interfaces, delaying implementation details until a subsequent section. Readers who wish to construct the specified gates in HDL are encouraged to do so, referring to appendix A as needed. All the gates can be built and simulated on a personal computer, using the hardware simulator supplied with the book.

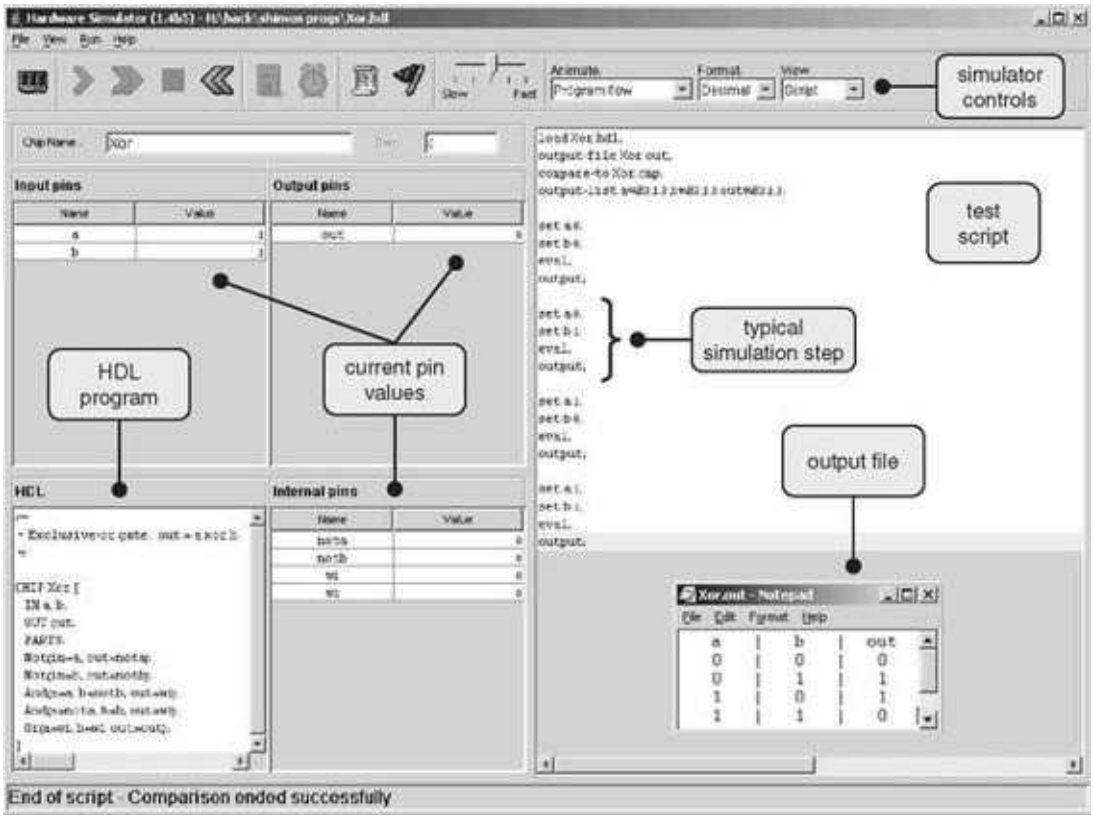


Figure 1.7 A screen shot of simulating an Xor chip on the hardware simulator. The simulator state is shown just after the test script has completed running. The pin values correspond to the last simulation step ($a = b = 1$). Note that the output file generated by the simulation is consistent with the Xor truth table, indicating that the loaded HDL program delivers a correct Xor functionality. The compare file, not shown in the figure and typically specified by the chip's client, has exactly the same structure and contents as that of the output file. The fact that the two files agree with each other is evident from the status message displayed at the bottom of the screen.

1.2.1 The Nand Gate

The starting point of our computer architecture is the Nand gate, from which all other gates and chips are built. The Nand gate is designed to compute the following Boolean function:

a	b	Nand(a,b)
0	0	1
0	1	1
1	0	1
1	1	0

Throughout the book, we use "chip API boxes" to specify chips. For each chip, the API specifies the chip name, the names of its input and output pins, the function or operation that the chip effects, and an optional comment.

```
Chip name: Nand
Inputs:    a, b
Outputs:   out
Function:  If a=b=1 then out=0 else out=1
Comment:   This gate is considered primitive and thus there is
           no need to implement it.
```

1.2.2 Basic Logic Gates

Some of the logic gates presented here are typically referred to as "elementary" or "basic." At the same time, every one of them can be composed from Nand gates alone. Therefore, they need not be viewed as primitive.

Not The single-input Not gate, also known as "converter," converts its input from 0 to 1 and vice versa. The gate API is as follows:

```
Chip name: Not
Inputs:    in
Outputs:   out
Function:  If in=0 then out=1 else out=0.
```

And The And function returns 1 when both its inputs are 1, and 0 otherwise.

```
Chip name: And
Inputs:    a, b
Outputs:   out
Function:  If a=b=1 then out=1 else out=0.
```

Or The Or function returns 1 when at least one of its inputs is 1, and 0 otherwise.

```
Chip name: Or
Inputs:    a, b
Outputs:   out
Function:  If a=b=0 then out=0 else out=1.
```

Xor The Xor function, also known as "exclusive or," returns 1 when its two inputs have opposing values, and 0 otherwise.

```
Chip name: Xor
Inputs:    a, b
Outputs:   out
Function:  If a≠b then out=1 else out=0.
```

Multiplexor A multiplexor (figure 1.8) is a three-input gate that uses one of the inputs, called "selection bit," to select and output one of the other two inputs, called "data bits." Thus, a better name for this device might have been selector. The name multiplexor was adopted from communications systems, where similar devices are used to serialize (multiplex) several input signals over a single output wire.

```
Chip name: Mux
Inputs:    a, b, sel
Outputs:   out
Function:  If sel=0 then out=a else out=b.
```

a	b	sel	out
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	1
0	0	1	0
0	1	1	1
1	0	1	0
1	1	1	1

sel	out
0	a
1	b

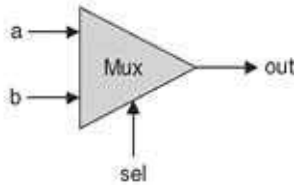


Figure 1.8 Multiplexor. The table at the top right is an abbreviated version of the truth table on the left.

sel	a	b
0	in	0
1	0	in

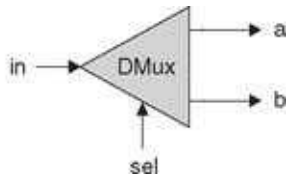


Figure 1.9 Demultiplexor.

```
Chip name: DMux
Inputs:    in, sel
Outputs:   a, b
Function:  If sel=0 then {a=in, b=0} else {a=0, b=in}.
```

1.2.3 Multi-Bit Versions of Basic Gates

Computer hardware is typically designed to operate on multi-bit arrays called "buses." For example, a basic requirement of a 32-bit computer is to be able to compute (bit-wise) an And function on two given 32-bit buses. To implement this operation, we can build an array of 32 binary And gates, each operating separately on a pair of bits. In order to enclose all this logic in one package, we can encapsulate the gates array in a single chip interface consisting of two 32-bit input buses and one 32-bit output bus.

This section describes a typical set of such multi-bit logic gates, as needed for the construction of a typical 16-bit computer. We note in passing that the architecture of *n*-bit logic gates is basically the same irrespective of *n*'s value.

When referring to individual bits in a bus, it is common to use an array syntax. For example, to refer to individual bits in a 16-bit bus named data, we use the notation data [0], data [1],..., data[15].

Multi-Bit Not An *n*-bit Not gate applies the Boolean operation Not to every one of the bits in its *n*-bit input bus:

```
Chip name: Not16
Inputs:    in[16] // a 16-bit pin
Outputs:   out[16]
Function:  For i=0..15 out[i]=Not(in[i]).
```

Multi-Bit And An *n*-bit And gate applies the Boolean operation And to every one of the n bit-pairs arrayed in its two *n*-bit input buses:

```
Chip name: And16
Inputs:    a[16], b[16]
Outputs:   out[16]
Function:  For i=0..15 out[i]=And(a[i],b[i]).
```

Multi-Bit Or An *n*-bit Or gate applies the Boolean operation Or to every one of the n bit-pairs arrayed in its two *n*-bit input buses:

```
Chip name: Or16
Inputs:    a[16], b[16]
Outputs:   out[16]
Function:  For i=0..15 out[i]=Or(a[i],b[i]).
```

Multi-Bit Multiplexor An *n*-bit multiplexor is exactly the same as the binary multiplexor described in figure 1.8, except that the two inputs are each *n*-bit wide; the selector is a single bit.

```
Chip name: Mux16
Inputs:    a[16], b[16], sel
Outputs:   out[16]
Function:  If sel=0 then for i=0..15 out[i]=a[i]
           else for i=0..15 out[i]=b[i].
```

1.2.4 Multi-Way Versions of Basic Gates

Many 2-way logic gates that accept two inputs have natural generalization to multi-way variants that accept an arbitrary number of inputs. This section describes a set of multi-way gates that will be used subsequently in various chips in our computer architecture. Similar generalizations can be developed for other architectures, as needed.

Multi-Way Or An *n*-way Or gate outputs 1 when at least one of its n bit inputs is 1, and 0 otherwise. Here is the 8-way variant of this gate:

```
Chip name: Or8Way
Inputs:    in[8]
Outputs:   out
Function:  out=Or(in[0],in[1],...,in[7]).
```

Multi-Way/Multi-Bit Multiplexor An *m*-way *n*-bit multiplexor selects one of m n-bit input buses and outputs it to a single *n*-bit output bus. The selection is specified by a set of *k* control bits, where $k = \log_2 m$. Figure 1.10 depicts a typical example.

The computer platform that we develop in this book requires two variations of this chip: A 4-way 16-bit multiplexor and an 8-way 16-bit multiplexor:

sel[1]	sel[0]	out
0	0	a
0	1	b
1	0	c
1	1	d

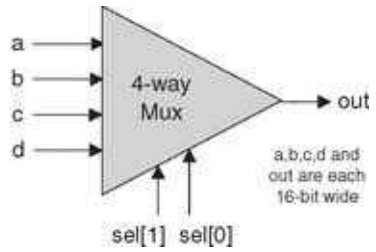


Figure 1.10 4-way multiplexor. The width of the input and output buses may vary.

```
Chip name: Mux4Way16
Inputs:    a[16], b[16], c[16], d[16], sel[2]
Outputs:   out[16]
Function:  If sel=00 then out=a else if sel=01 then out=b
           else if sel=10 then out=c else if sel=11 then out=d
Comment:   The assignment operations mentioned above are all
           16-bit. For example, "out=a" means "for i=0..15
           out[i]=a[i]".
```

```
Chip name: Mux8Way16
Inputs:    a[16],b[16],c[16],d[16],e[16],f[16],g[16],h[16],
           sel[3]
Outputs:   out[16]
Function:  If sel=000 then out=a else if sel=001 then out=b
           else if sel=010 out=c ... else if sel=111 then out=h
Comment:   The assignment operations mentioned above are all
           16-bit. For example, "out=a" means "for i=0..15
           out[i]=a[i]".
```

Multi-Way/Multi-Bit Demultiplexor An *m*-way *n*-bit demultiplexor (figure 1.11) channels a single *n*-bit input into one of *m* possible *n*-bit outputs. The selection is specified by a set of *k* control bits, where $k = \log_2 m$.

The specific computer platform that we will build requires two variations of this chip: A 4-way 1-bit demultiplexor and an 8-way 1-bit multiplexor, as follows.

sel[1]	sel[0]	a	b	c	d
0	0	in	0	0	0
0	1	0	in	0	0
1	0	0	0	in	0
1	1	0	0	0	in

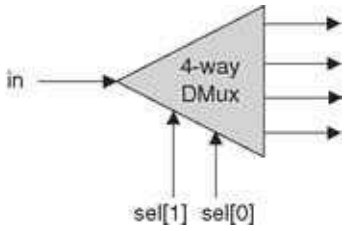


Figure 1.11 4-way demultiplexor.

```
Chip name: DMux4Way
Inputs:    in, sel[2]
Outputs:   a, b, c, d
Function:  If sel=00 then      {a=in, b=c=d=0}
           else if sel=01 then {b=in, a=c=d=0}
           else if sel=10 then {c=in, a=b=d=0}
           else if sel=11 then {d=in, a=b=c=0}.
```

```
Chip name: DMux8Way
Inputs:    in, sel[3]
Outputs:   a, b, c, d, e, f, g, h
Function:  If sel=000 then      {a=in, b=c=d=e=f=g=h=0}
           else if sel=001 then {b=in, a=c=d=e=f=g=h=0}
           else if sel=010 ...
           ...
           else if sel=111 then {h=in, a=b=c=d=e=f=g=0}.
```

1.3 Implementation

Similar to the role of axioms in mathematics, primitive gates provide a set of elementary building blocks from which everything else can be built. Operationally, primitive gates have an "off-the-shelf" implementation that is supplied externally. Thus, they can be used in the construction of other gates and chips without worrying about their internal design. In the computer architecture that we are now beginning to build, we have chosen to base all the hardware on one primitive gate only: Nand. We now turn to outlining the first stage of this bottom-up hardware construction project, one gate at a time.

Our implementation guidelines are intentionally partial, since we want you to discover the actual gate architectures yourself. We reiterate that each gate can be implemented in more than one way; the simpler the implementation, the better.

Not: The implementation of a unary Not gate from a binary Nand gate is simple. Tip: Think positive.

And: Once again, the gate implementation is simple. Tip: Think negative.

Or/Xor: These functions can be defined in terms of some of the Boolean functions implemented previously, using some simple Boolean manipulations. Thus, the respective gates can be built using previously built gates.

Multiplexor/ Demultiplexor: Likewise, these gates can be built using previously built gates.

Multi-Bit Not/And/Or Gates: Since we already know how to implement the elementary versions of these gates, the implementation of their *n*-ary versions is simply a matter of constructing arrays of n elementary gates, having each gate operate separately on its bit inputs. This implementation task is rather boring, but it will carry its weight when these multi-bit gates are used in more complex chips, as described in subsequent chapters.

Multi-Bit Multiplexor: The implementation of an *n*-ary multiplexor is simply a matter of feeding the same selection bit to every one of *n* binary multiplexors. Again, a boring task resulting in a very useful chip.

Multi-Way Gates: Implementation tip: Think forks.

1.4 Perspective

This chapter described the first steps taken in an applied digital design project. In the next chapter we will build more complicated functionality using the gates built here. Although we have chosen to use Nand as our basic building block, other approaches are possible. For example, one can build a complete computer platform using Nor gates alone, or, alternatively, a combination of And, Or, and Not gates. These constructive approaches to logic design are theoretically equivalent, just as all theorems in geometry can be founded on different sets of axioms as alternative points of departure. The theory and practice of such constructions are covered in standard textbooks about digital design or logic design.

Throughout the chapter, we paid no attention to efficiency considerations such as the number of elementary gates used in constructing a composite gate or the number of wire crossovers implied by the design. Such considerations are critically important in practice, and a great deal of computer science and electrical engineering expertise focuses on optimizing them. Another issue we did not address at all is the physical implementation of gates and chips using the laws of physics, for example, the role of transistors embedded in silicon. There are of course several such implementation options, each having its own characteristics (speed, power requirements, production cost, etc.). Any nontrivial coverage of these issues requires some background in electronics and physics.

1.5 Project

Objective Implement all the logic gates presented in the chapter. The only building blocks that you can use are primitive Nand gates and the composite gates that you will gradually build on top of them.

Resources The only tool that you need for this project is the hardware simulator supplied with the book. All the chips should be implemented in the HDL language specified in appendix A. For each one of the chips mentioned in the chapter, we provide a skeletal .hdl program (text file) with a missing implementation part. In addition, for each chip we provide a .tst script file that tells the hardware simulator how to test it, along with the correct output file that this script should generate, called .cmp or "compare file." Your job is to complete the missing implementation parts of the supplied .hdl programs.

Contract When loaded into the hardware simulator, your chip design (modified .hdl program), tested on the supplied .tst file, should produce the outputs listed in the supplied .cmp file. If that is not the case, the simulator will let you know.

Tips The Nand gate is considered primitive and thus there is no need to build it: Whenever you use Nand in one of your HDL programs, the simulator will automatically invoke its built-in tools/builtIn/Nand.hdl implementation. We recommend implementing the other gates in this project in the order in which they appear in the chapter. However, since the builtIn directory features working versions of all the chips described in the book, you can always use these chips without defining them first: The simulator will automatically use their built-in versions.

For example, consider the skeletal Mux.hdl program supplied in this project. Suppose that for one reason or another you did not complete this program's implementation, but you still want to use Mux gates as internal parts in other chip designs. This is not a problem, thanks to the following convention. If our simulator fails to find a Mux.hdl file in the current directory, it automatically invokes a built-in Mux implementation, pre-supplied with the simulator's software. This built-in implementation—a Java class stored in the built In directory—has the same interface and functionality as those of the Mux gate described in the book. Thus, if you want the simulator to ignore one or more of your chip implementations, simply move the corresponding .hdl files out of the current directory.

Steps We recommend proceeding in the following order:

0. The *hardware simulator* needed for this project is available in the tools directory of the book's software suite.

1. Read appendix A, sections A1-A6 only.

2. Go *through the hardware simulator tutorial,* parts I, II, and III only.

3. Build and simulate all the chips specified in the projects/01 directory.