THE ELEMENTS OF COMPUTING SYSTEMS

second edition

BUILDING A MODERN COMPUTER FROM FIRST PRINCIPLES

NOAM NISAN AND SHIMON SCHOCKEN

Noam Nisan and Shimon Schocken

The Elements of Computing Systems

Building a Modern Computer from First Principles

Second Edition

The MIT Press
Cambridge, Massachusetts
London, England

Library of Congress Cataloging-in-Publication Data

Names: Nisan, Noam, author. | Schocken, Shimon, author.
Title: The elements of computing systems : building a modern computer from first principles / Noam Nisan and Shimon Schocken.
Description: Second edition. | Cambridge, Massachusetts : The MIT Press, [2021] | Includes index.
Identifiers: LCCN 2020002671 | ISBN 9780262539807 (paperback)
Subjects: LCSH: Electronic digital computers.
Classification: LCC TK7888.3 .N57 2020 | DDC 004.16—dc23
LC record available at https://lccn.loc.gov/2020002671

d_r0

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

Contents

List of Figures

Figure 1.9
Multiplexer. The table at the top right is an abbreviated version of the truth table.

Figure 1.10
Demultiplexer.

Figure 2.1
Two's complement representation of signed numbers, in a 4-bit binary system.

Figure 2.2
Half-adder, designed to add 2 bits.

Figure 2.3
Full-adder, designed to add 3 bits.

Figure 2.4
16-bit adder, designed to add two 16-bit numbers, with an example of addition action (on the left).

Figure 2.5a
The Hack ALU, designed to compute the eighteen arithmetic-logical functions shown on the right (the symbols !, &, and | represent, respectively, the 16-bit operations Not, And, and Or). For now, ignore the zr and ng output bits.

Figure 2.5b
Taken together, the values of the six control bits zx, nx, zy, ny, f, and no cause the ALU to compute one of the functions listed in the rightmost column.

Figure 2.5c
The Hack ALU API.

Figure 3.1
The memory hierarchy built in this chapter.

Figure 3.2
Discrete time representation: State changes (input and output values) are observed only during cycle transitions. Within cycles, changes are ignored.

Figure 3.3
The data flip-flop (top) and behavioral example (bottom). In the first cycle the previous input is unknown, so the DFF's output is undefined. In every subsequent time unit, the DFF outputs the input from the previous time unit. Following gate diagramming conventions, the clock input is marked by a small triangle, drawn at the bottom of the gate icon.

Figure 3.4
Sequential logic design typically involves DFF gates that feed from, and connect to, combinational chips. This gives sequential chips the ability to respond to current as well as to previous inputs and outputs.

Figure 3.5
1-bit register. Stores and emits a 1-bit value until instructed to load a new value.

Figure 3.6
16-bit Register. Stores and emits a 16-bit value until instructed to load a new value.

Figure 3.7
A RAM chip, consisting of *n* 16-bit Register chips that can be selected and manipulated separately.

The register addresses (0 to $n-1$) are not part of the chip hardware. Rather, they are realized by a gate logic implementation that will be discussed in the next section.

Figure 3.8
Program Counter (PC): To use it properly, at most one of the load, inc, or reset bits should be asserted.

Figure 3.9
The Bit (1-bit register) implementation: invalid (left) and correct (right) solutions.

Figure 4.2
Conceptual model of the Hack memory system. Although the actual architecture is wired somewhat differently (as described in chapter 5), this model helps understand the semantics of Hack programs.

Figure 4.3
Hack assembly code examples.

Figure 4.4
A Hack assembly program (example). Note that RAM[0] and RAM[1] can be referred to as R0 and R1.

Figure 4.5
The Hack instruction set, showing symbolic mnemonics and their corresponding binary codes.

Figure 4.6
A Hack assembly program that computes a simple arithmetic expression.

Figure 4.7
Array processing example, using pointer-based access to array elements.

Figure 4.8
The CPU emulator, with a program loaded in the instruction memory (ROM) and some data in the data memory (RAM). The figure shows a snapshot taken during the program's execution.

Figure 5.1
A generic von Neumann computer architecture.

Figure 5.2
The Hack Central Processing Unit (CPU) interface.

Figure 5.3
The Hack instruction memory interface.

Figure 5.4
The Hack Screen chip interface.

Figure 5.5
The Hack Keyboard chip interface.

Figure 5.6
The Hack data memory interface. Note that the decimal values 16384 and 24576 are 4000 and 6000 in hexadecimal.

Figure 5.7
Interface of Computer, the topmost chip in the Hack hardware platform.

Figure 5.8
Proposed implementation of the Hack CPU, showing an incoming 16-bit instruction. We use the instruction notation *cccccccccccccccc* to emphasize that in the case of a *C*-instruction, the instruction is treated as a capsule of control bits, designed to control different CPU chip-parts. In this diagram, every *c* symbol entering a chip-part stands for some control bit extracted from the instruction (in the case of the ALU, the *c*'s input stands for the six control bits that instruct the ALU what to compute). Taken together, the distributed behavior induced by these control bits ends up executing the instruction. We don't specify which bits go where, since we want readers to answer these questions themselves.

Figure 5.9
Proposed implementation of Computer, the topmost chip in the Hack platform.

Figure 5.10
Testing the Computer chip on the supplied hardware simulator. The stored program is Rect, which draws a rectangle of RAM[0] rows of 16 pixels each, all black, at the top-left of the screen.

Figure 6.1
Assembly code, translated to binary code using a symbol table. The line numbers, which are not part of the code, are listed for reference.

Figure 6.2
The Hack instruction set, showing both symbolic mnemonics and their corresponding binary codes.

Figure 6.3
Testing the assembler's output using the supplied assembler.

Figure II.1
Manipulating points on a plane: example and Jack code.

Figure II.2
Jack implementation of the Point abstraction.

Figure II.3
Road map of part II (the assembler belongs to part I and is shown here for completeness). The road map describes a translation hierarchy, from a high-level, object-based, multi-class program to VM code, to assembly code, to executable binary code. The numbered circles stand for the projects that implement the compiler, the VM translator, the assembler, and the operating system. Project 9 focuses on writing a Jack application in order to get acquainted with the language.

Figure 7.1
The virtual machine framework, using Java as an example. High-level programs are compiled into intermediate VM code. The same VM code can be shipped to, and executed on, any hardware platform equipped with a suitable *JVM implementation*. These VM implementations are typically realized as client-side programs that translate the VM code into the machine languages of the target devices.

Figure 7.2
Stack processing example, illustrating the two elementary operations push and pop. The setting consists of two data structures: a RAM-like memory segment and a stack. Following convention, the stack is drawn as if it grows downward. The location just following the stack's top value is referred to by a pointer called *sp*, or *stack pointer*. The *x* and *y* symbols refer to two arbitrary memory locations.

Figure 7.3a
Stack-based evaluation of arithmetic expressions.

Figure 7.3b
Stack-based evaluation of logical expressions.

Figure 7.4
Virtual memory segments.

Figure 7.5
The arithmetic-logical commands of the VM language.

Figure 7.6
The VM emulator supplied with the Nand to Tetris software suite.

Figure 8.1
Branching commands action. (The VM code on the right uses symbolic variable names instead of virtual memory segments, to make it more readable.)

Figure 8.2
Run-time snapshots of selected stack and segment states during the execution of a three-function program. The line numbers are not part of the code and are given for reference only.

Figure 8.3
The global stack, shown when the callee is running. Before the callee terminates, it pushes a return value onto the stack (not shown). When the VM implementation handles the return command, it copies the return value onto argument 0, and sets SP to point to the address just following it. This effectively frees the global stack area below the new value of SP. Thus, when the caller resumes its execution, it sees the return value at the top of its working stack.

Figure 8.4
Several snapshots of the *global stack*, taken during the execution of the main function, which calls factorial to compute 3!. The running function sees only its working stack, which is the unshaded area at the tip of the global stack; the other unshaded areas in the global stack are the working stacks of functions up the calling chain, waiting for the currently running function to return. Note that the shaded areas are not "drawn to scale," since each frame consists of five words, as shown in figure 8.3.

Figure 8.5
Implementation of the function commands of the VM language. All the actions described on the right are realized by generated Hack assembly instructions.

Figure 8.6
The naming conventions described above are designed to support the translation of multiple .vm files and functions into a single .asm file, ensuring that the generated assembly symbols will be unique within the file.

Figure 9.1
Hello World, written in the Jack language.

Figure 9.2
Typical procedural programming and simple array handling. Uses the services of the OS classes Array, Keyboard, and Output.

Figure 9.3a
Fraction API (top) and sample Jack class that uses it for creating and manipulating Fraction objects.

Figure 9.3b
A Jack implementation of the Fraction abstraction.

Figure 9.4
Linked list implementation in Jack (left and top right) and sample usage (bottom right).

Figure 9.5
Operating system services (summary). The complete OS API is given in appendix 6.

Figure 9.6
The syntax elements of the Jack language.

Figure 9.7
Variable kinds in the Jack language. Throughout the table, *subroutine* refers to either a *function*, a *method*, or a *constructor*.

Figure 9.8
Statements in the Jack language.

Figure 9.9
Screenshots of Jack applications running on the Hack computer.

Figure 10.1
Staged development plan of the Jack compiler.

Figure 10.2
Definition of the Jack lexicon, and lexical analysis of a sample input.

Figure 10.3
A subset of the Jack language grammar, and Jack code segments that are either accepted or rejected by the grammar.

Figure 10.4a
Parse tree of a typical code segment. The parsing process is driven by the grammar rules.

Figure 10.4b
Same parse tree, in XML.

Figure 10.5
The Jack grammar.

Figure 11.1
The Point class. This class features all the possible variable kinds (field, static, local, and argument) and subroutine kinds (constructor, method, and function), as well as subroutines that return primitive types, object types, and void subroutines. It also illustrates function calls, constructor calls, and method calls on the current object (this) and on other objects.

Figure 11.2
Symbol table examples. The this row in the subroutine-level table is discussed later in the chapter.

Figure 11.3
Infix and postfix renditions of the same semantics.

Figure 11.4
A VM code generation algorithm for expressions, and a compilation example. The algorithm assumes that the input expression is valid. The final implementation of this algorithm should replace the emitted symbolic variables with their corresponding symbol table mappings.

Figure 11.5
Expressions in the Jack language.

Figure 11.6
Compiling if and while statements. The L1 and L2 labels are generated by the compiler.

Figure 11.7
Object construction from the caller's perspective. In this example, the caller declares two object variables and then calls a class constructor for constructing the two objects. The constructor works its magic, allocating memory blocks for representing the two objects. The calling code then makes the two object variables refer to these memory blocks.

Figure 11.8
Object construction: the constructor's perspective.

Figure 11.9
Compiling method calls: the caller's perspective.

Figure 11.10
Compiling methods: the callee's perspective.

Figure 11.11
Array access using VM commands.

Figure 11.12
Basic compilation strategy for arrays, and an example of the bugs that it can generate. In this particular case, the value stored in pointer 1 is overridden, and the address of a[i] is lost.

Figure 12.1
Multiplication algorithm.

Figure 12.2
Division algorithm.

Figure 12.3
Square root algorithm.

Figure 12.4
String-integer conversions. (appendChar, length, and charAt are String class methods.)

Figure 12.5a
Memory allocation algorithm (basic).

Figure 12.5b
Memory allocation algorithm (improved).

Figure 12.6
Drawing a pixel.

Figure 12.7
Line-drawing algorithm: basic version (bottom, left) and improved version (bottom, right).

Figure 12.8
Circle-drawing algorithm.

Figure 12.9
Example of a character bitmap.

Figure 12.10
Handling input from the keyboard.

Figure 12.11
A trapdoor enabling complete control of the host RAM from Jack.

Figure 12.12
Logical view (left) and physical implementation (right) of a linked list that supports dynamic memory allocation.

Figure A1.1
Synthesizing a Boolean function from a truth table (example).

Figure A2.1
HDL program example.

Figure A2.2
Buses in action (example).

Figure A2.3
Built-in chip definition example.

Figure A2.4
DFF definition.

Figure A2.5
A chip that activates GUI-empowered chip-parts.

Figure A2.6
GUI-empowered chips demo. Since the loaded HDL program uses GUI-empowered chip-parts (step 1), the simulator renders their respective GUI images (step 2). When the user changes the values of the chip input pins (step 3), the simulator reflects these changes in the respective GUIs (step 4).

Figure A3.1
Test script and compare file (example).

Figure A3.2
Variables and methods of key built-in chips in Nand to Tetris.

Figure A3.3
Testing the topmost Computer chip.

Figure A3.4
Testing a machine language program on the CPU emulator.

Figure A3.5
Testing a VM program on the VM emulator.

Preface

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

—Confucius (551–479 B.C.)

It is commonly argued that enlightened people of the twenty-first century ought to familiarize themselves with the key ideas underlying BANG: Bits, Atoms, Neurons, and Genes. Although science has been remarkably successful in uncovering their basic operating systems, it is quite possible that we will never fully grasp how atoms, neurons, and genes actually work. Bits, however, and computing systems at large, entail a consoling exception: in spite of their fantastic complexity, one can completely understand how modern computers work, and how they are built. So, as we gaze with awe at the BANG around us, it is a pleasing thought that at least one field in this quartet can be fully laid bare to human comprehension.

Indeed, in the early days of computers, any curious person who cared to do so could gain a gestalt understanding of how the machine works. The interactions between hardware and software were simple and transparent enough to produce a coherent picture of the computer's operations. Alas, as digital 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: the study of applied computer science became a pursuit of many niche 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 learner 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, software, and application 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 learners and professionals with an uneasy feeling that, well, they don't fully understand what's going on inside computers. This is unfortunate, since computers are the most important machines in the twenty-first century.

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 idea: Let's specify a simple but sufficiently powerful computer system, and invite learners to build its hardware platform and software hierarchy from the ground up. And while we are at it, let's do it right. We are saying this because building a general-purpose computer from first principles is a huge enterprise.

Therefore, we identified a unique educational opportunity to not only build the thing, but also 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 thrill of constructing, through careful reasoning and modular planning, fantastically complex and useful systems from first principles.

The outcome of this effort became what is now known colloquially as *Nand to Tetris*: a hands-on journey that starts with the most elementary logic gate, called Nand, and ends up, twelve projects later, with a general-purpose computer system capable of running Tetris, as well as any other program that comes to your mind. After designing, building, redesigning, and rebuilding the computer system several times ourselves, we wrote this book, explaining how any learner can do the same. We also launched the www.nand2tetris.org website, making all our project materials and software tools freely available to anyone who wants to learn, or teach, Nand to Tetris courses.

We are gratified to say that the response has been overwhelming. Today, Nand to Tetris courses are taught in numerous universities, high schools, coding boot camps, online platforms, and hacker clubs around the world. The book and our online courses became highly popular, and thousands of

learners—ranging from high school students to Google engineers—routinely post reviews describing Nand to Tetris as their best educational experience ever. As Richard Feynman famously said: "What I cannot create, I do not understand." Nand to Tetris is all about understanding through creation. Apparently, people connect passionately to this maker mentality.

Since the publication of the book's first edition, we received numerous questions, comments, and suggestions. As we addressed these issues by modifying our online materials, a gap developed between the web-based and the book-based versions of Nand to Tetris. In addition, we felt that many book sections could benefit from more clarity and a better organization. So, after delaying this surgery as much as we could, we decided to roll up our sleeves and write a second edition, leading to the present book. The remainder of this preface describes this new edition, ending with a section that compares it to the previous one.

Scope

The book exposes learners to a significant body of computer science knowledge, gained through a series of hardware and software construction tasks. In particular, the following topics are illustrated in the context of hands-on projects:

- *Hardware:* Boolean arithmetic, combinational logic, sequential logic, design and implementation of logic gates, multiplexers, flip-flops, registers, RAM units, counters, Hardware Description Language (HDL), chip simulation, verification and testing.
- *Architecture:* ALU/CPU design and implementation, clocks and cycles, addressing modes, fetch and execute logic, instruction set, memory-mapped input/output.
- *Low-level languages:* Design and implementation of a simple machine language (binary and symbolic), instruction sets, assembly programming, assemblers.

- *Virtual machines:* Stack-based automata, stack arithmetic, function call and return, handling recursion, design and implementation of a simple VM language.
- *High-level languages:* Design and implementation of a simple object-based, Java-like language: abstract data types, classes, constructors, methods, scoping rules, syntax and semantics, references.
- *Compilers:* Lexical analysis, parsing, symbol tables, code generation, implementation of arrays and objects, two-tier compilation.
- *Programming:* Implementation of an assembler, virtual machine, and compiler, following supplied APIs. Can be done in any programming language.
- *Operating systems:* Design and implementation of memory management, math library, input/output drivers, string processing, textual output, graphical output, high-level language support.
- *Data structures and algorithms:* Stacks, hash tables, lists, trees, arithmetic algorithms, geometric algorithms, running time considerations.
- *Software engineering:* Modular design, the interface/implementation paradigm, API design and documentation, unit testing, proactive test planning, quality assurance, programming at the large.

A unique feature of Nand to Tetris is that all these topics are presented cohesively, with a clear, over-arching purpose: building a modern computer system from the ground up. In fact, this has been our topic selection criterion: the book focuses on the minimal set of topics necessary for building a general-purpose computer system, capable of running programs written in a high-level, object-based language. As it turns out, this critical set includes most of the fundamental concepts and techniques, as well as some of the most beautiful ideas, in applied computer science.

Courses

Nand to Tetris courses are typically cross-listed for both undergraduate and graduate students, and are highly popular among self-learners. Courses based on this book are "perpendicular" to the typical computer science

curriculum and can be taken at almost any point during the program. Two natural slots are CS-2—an introductory yet post-programming course—and CS-99—a synthesis course coming at the end of the program. The former course entails a forward-looking, systems-oriented introduction to applied computer science, while the latter is an integrative, project-based course that fills gaps left by previous courses.

Another increasingly popular slot is a course that combines, in one framework, key topics from traditional computer architecture courses and compilation courses. Whichever purpose they are made to serve, Nand to Tetris courses go by many names, including Elements of Computing Systems, Digital Systems Construction, Computer Organization, Let's Build a Computer, and, of course, Nand to Tetris.

The book and the projects are highly modular, starting from the top division into Part I: Hardware and Part II: Software, each comprising six chapters and six projects. Although we recommend going through the full experience, it is entirely possible to learn each of the two parts separately. The book and the projects can support two independent courses, each six to seven weeks long, a typical semester-long course, or two semester-long courses, depending on topic selection and pace of study.

The book is completely self-contained: all the necessary knowledge for building the hardware and software systems described in the book is given in its chapters and projects. Part I: Hardware requires no prerequisite knowledge, making projects 1–6 accessible to any student and self-learner. Part II: Software and projects 7–12 require programming (in any high-level language) as a prerequisite.

Nand to Tetris courses are not restricted to computer science majors. Rather, they lend themselves to learners from any discipline who seek to gain a hands-on understanding of hardware architectures, operating systems, compilation, and software engineering—all in one course. Once again, the only prerequisite (for part II) is programming. Indeed, many Nand to Tetris students are nonmajors who took an introduction to computer science course and now wish to learn more computer science without committing themselves to a multicourse program. Many other learners are software developers who wish to "go below," understand how the enabling technologies work, and become better high-level programmers.

Following the acute shortage of developers in the hardware and software industries, there is a growing demand for compact and focused programs in applied computer science. These often take the form of coding boot camps and clusters of online courses designed to prepare learners for the job market without going through the full gamut of an academic degree. Any such solid program must offer, at minimum, working knowledge of programming, algorithms, and systems. Nand to Tetris is uniquely positioned to cover the systems element of such programs, in the framework of one course. Further, the Nand to Tetris projects provide an attractive means for synthesizing, and putting to practice, much of the algorithmic and programmatic knowledge learned in other courses.

Resources

All the necessary tools for building the hardware and software systems described in the book are supplied freely in the Nand to Tetris software suite. These include a hardware simulator, a CPU emulator, a VM emulator (all in open source), tutorials, and executable versions of the assembler, virtual machine, compiler, and operating system described in the book. In addition, the www.nand2tetris.org website 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. The software tools and project materials can be used as is on any computer running Windows, Linux, or macOS.

Structure

Part I: Hardware consists of chapters 1–6. Following an introduction to Boolean algebra, chapter 1 starts with the elementary Nand gate and builds a set of elementary logic gates on top of it. Chapter 2 presents combinational logic and builds a set of adders, leading up to an ALU. Chapter 3 presents sequential logic and builds a set of registers and memory devices, leading up to a RAM. Chapter 4 discusses low-level programming and specifies a machine language in both its symbolic and binary forms.

Chapter 5 integrates the chips built in chapters 1–3 into a hardware architecture capable of executing programs written in the machine language presented in chapter 4. Chapter 6 discusses low-level program translation, culminating in the construction of an assembler.

Part II: Software consists of chapters 7–12 and requires programming background (in any language) at the level of introduction to computer science courses. Chapters 7–8 present stack-based automata and describe the construction of a JVM-like virtual machine. Chapter 9 presents an object-based, Java-like high-level language. Chapters 10–11 discuss parsing and code generation algorithms and describe the construction of a two-tier compiler. Chapter 12 presents various memory management, algebraic, and geometric algorithms and describes the implementation of an operating system that puts them to practice. The OS is designed to close gaps between the high-level language implemented in part II and the hardware platform built in part I.

The book is based on an *abstraction-implementation* paradigm. Each chapter starts with an Introduction describing relevant concepts and a generic hardware or software system. The next section is always Specification, describing the system's abstraction, that is, the various services that it is expected to deliver, one way or another. Having presented the *what*, each chapter proceeds to discuss *how* the abstraction can be realized, leading to a proposed Implementation section. The next section is always Project, providing step-by-step guidelines, testing materials, and software tools for building and unit-testing the system described in the chapter. The closing Perspective section highlights noteworthy issues left out from the chapter.

Projects

The computer system described in the book is *for real*. It is designed to be built, and it works! The book is geared toward active readers who are willing to get their hands dirty and build the computer from the ground up. If you'll take the time and effort to do so, you will gain a depth of understanding and a sense of accomplishment unmatched by mere reading.

The hardware devices built in projects 1, 2, 3, and 5 are implemented using a simple Hardware Description Language (HDL) and simulated on a supplied software-based hardware simulator, which is exactly how hardware architects work in industry. Projects 6, 7, 8, 10, and 11 (assembler, virtual machine I + II, and compiler I + II) can be written in any programming language. Project 4 is written in the computer's assembly language, and projects 9 and 12 (a simple computer game and a basic operating system) are written in Jack—the Java-like high-level language for which we build a compiler in chapters 10 and 11.

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

Is it possible to cover so much ground in a one-semester course? The answer is yes, and the proof is in the pudding: more than 150 universities teach semester-long Nand to Tetris courses. The student satisfaction is exceptional, and Nand to Tetris MOOCs are routinely listed at the top of the top-rated lists of online course. One reason why learners respond to our methodology is *focus*. Except for obvious cases, we pay no attention to optimization, leaving this important subject to other, more specific courses. In addition, we allow students to assume error-free inputs. This eliminates the need to write code for handling exceptions, making the software projects significantly more focused and manageable. Dealing with incorrect input is of course critically important, but this skill can be honed elsewhere, for example, in project extensions and in dedicated programming and software design courses.

The Second Edition

Although Nand to Tetris was always structured around two themes, the second edition makes this structure explicit: The book is now divided into two distinct and standalone parts, Part I: Hardware and Part II: Software. Each part consists of six chapters and six projects and begins with a newly written introduction that sets the stage for the part's chapters. Importantly,

the two parts are independent of each other. Thus, the new book structure lends itself well to quarter-long as well as semester-long courses.

In addition to the two new introduction chapters, the second edition features four new appendices. Following the requests of many learners, these new appendices give focused presentations of various technical topics that, in the first edition, were scattered across the chapters. Another new appendix provides a formal proof that any Boolean function can be built from Nand operators, adding a theoretical perspective to the applied hardware construction projects. Many new sections, figures, and examples were added.

All the chapters and project materials were rewritten with an emphasis on separating abstraction from implementation—a major theme in Nand to Tetris. We took special care to add examples and sections that address the thousands of questions that were posted over the years in Nand to Tetris Q&A forums.

Acknowledgments

The software tools that accompany the book were developed by our students at IDC Herzliya and at the Hebrew University. The two chief software architects were Yaron Ukrainitz and Yannai Gonczarowski, and the developers included Iftach Ian Amit, Assaf Gad, Gal Katzhendler, Hadar Rosen-Sior, and Nir Rozen. Oren Baranes, Oren Cohen, Jonathan Gross, Golan Parashi, and Uri Zeira worked on other aspects of the tools. Working with these student-developers has been a great pleasure, and we are proud to have had the opportunity to play a role in their education.

We also thank our teaching assistants, Muawyah Akash, Philip Hendrix, Eytan Lifshitz, Ran Navok, and David Rabinowitz, who helped run early versions of the course that led to this book. Tal Achituv, Yong Bakos, Tali Gutman and Michael Schröder provided great help with various aspects of the course materials, and Aryeh Schnall, Tomasz Różański and Rudolf Adamkovič gave meticulous editing suggestions. Rudolf's comments were particularly enlightening, for which we are very grateful.

Many people around the world got involved in Nand to Tetris, and we cannot possibly thank them individually. We do take one exception. Mark

Armbrust, a software and firmware engineer from Colorado, became the guarding angel of Nand to Tetris learners. Volunteering to manage our global Q&A forum, Mark answered numerous questions with great patience and graceful style. His answers never gave away the solutions; rather, he guided learners how to apply themselves and see the light on their own. In doing so, Mark has gained the respect and admiration of numerous learners around the world. Serving at the forefront of Nand to Tetris for more than ten years, Mark wrote 2,607 posts, discovered dozens of bugs, and wrote corrective scripts and fixes. Doing all this in addition to his regular day job, Mark became a pillar of the Nand to Tetris community, and the community became his second home. Mark died in March 2019, following a struggle with heart disease that lasted several months. During his hospitalization, Mark received a daily stream of hundreds of emails from Nand to Tetris students. Young men and women from all over the world thanked Mark for his boundless generosity and shared the impact that he has had on their lives.

In recent years, computer science education became a powerful driver of personal growth and economic mobility. Looking back, we feel fortunate that we decided, early on, to make all our teaching resources freely available, in open source. Quite simply, any person who wants to do so can not only learn but also teach Nand to Tetris courses, without any restrictions. All you have to do is go to our website and take what you need, so long as you operate in a nonprofit setting. This turned Nand to Tetris into a readily available vehicle for disseminating high-quality computer science education, freely and equitably. The result became a vast educational ecosystem, fueled by endless supplies of good will. We thank the many people around the world who helped us make it happen.

I Hardware

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 computer systems work, how to break complex problems into manageable modules, and how to build large-scale hardware and software systems. This will be a hands-on journey, as you create a complete and working computer system from the ground up. The lessons you will learn, which are far more important than the computer itself, will be gained as side effects of these constructions. 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 introduction chapter sketches some of the discoveries, truths, and experiences that lie ahead.

Hello, World Below

If you have some programming experience, you've probably encountered something like the program below early in your training. And if you haven't, you can still guess what the program is doing: it displays the text `Hello World` and terminates. This particular program is written in Jack—a simple, Java-like high-level language:

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

Trivial programs like Hello World are deceptively simple. Did you ever stop to 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 sequence of plain characters, stored in a text file. This abstraction is a complete mystery for the computer, which understands only instructions written in machine language. Thus, if we want to execute this program, the first thing we must do is parse the string of characters of which the high-level code is made, uncover its semantics—figure out what the program seeks to do—and then generate low-level code that reexpresses this semantics using the machine language of the target computer. The result of this elaborate translation process, known as *compilation*, will be an executable sequence of machine language instructions.

Of course, machine language is also an abstraction—an agreed upon set of binary codes. To make this abstraction concrete, it must be realized by some *hardware architecture*. And this architecture, in turn, is implemented by a certain set of chips—registers, memory units, adders, and so on. Now, every one of these hardware devices is constructed from lower-level, *elementary logic gates*. And these gates, in turn, can be built from primitive gates like *Nand* and *Nor*. These primitive gates are very low in the hierarchy, but they, too, are made 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 programs is much easier—all I have to do is click this icon or write that command!" Indeed, a modern computer system is like a submerged iceberg: most people get to see only the top, and 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 what separates naïve programmers from sophisticated developers—people who can create 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 the ground up.

Nand to Tetris

Assuming that we want to build a computer system from the ground up, which specific computer should we build? As it turns out, every general-purpose computer—every PC, smartphone, or server—is a Nand to Tetris machine. First, all computers are based, at bottom, on elementary logic gates, of which Nand is the most widely used in industry (we'll explain what exactly is a Nand gate in chapter 1). Second, every general-purpose computer can be programmed to run a Tetris game, as well as any other program that tickles your fancy. Thus, there is nothing unique about either Nand or Tetris. It is the word *to* in *Nand to Tetris* that turns this book into the magical journey that you are about to undertake: going all the way from a heap of barebone switching devices to a machine that engages the mind with text, graphics, animation, music, video, analysis, simulation, artificial intelligence, and all the capabilities that we came to expect from general-purpose computers. Therefore, it doesn't really matter which specific hardware platform and software hierarchy we will build, so long as they will be based on the same ideas and techniques that characterize *all* computing systems out there.

Figure I.1 describes the key milestones in the Nand to Tetris road map. Starting at the bottom tier of the figure, any general-purpose computer has an architecture that includes a ALU (Arithmetic Logic Unit) and a RAM (Random Access Memory). All ALU and RAM devices are made of elementary logic gates. And, surprisingly and fortunately, as we will soon see, all logic gates can be made from Nand gates alone. Focusing on the software hierarchy, all high-level languages rely on a suite of translators (compiler/interpreter, virtual machine, assembler) for reducing high-level code all the way down to machine-level instructions. Some high-level

languages are interpreted rather than compiled, and some don't use a virtual machine, but the big picture is essentially the same. This observation is a manifestation of a fundamental computer science principle, known as the *Church-Turing conjecture*: at bottom, all computers are essentially equivalent.

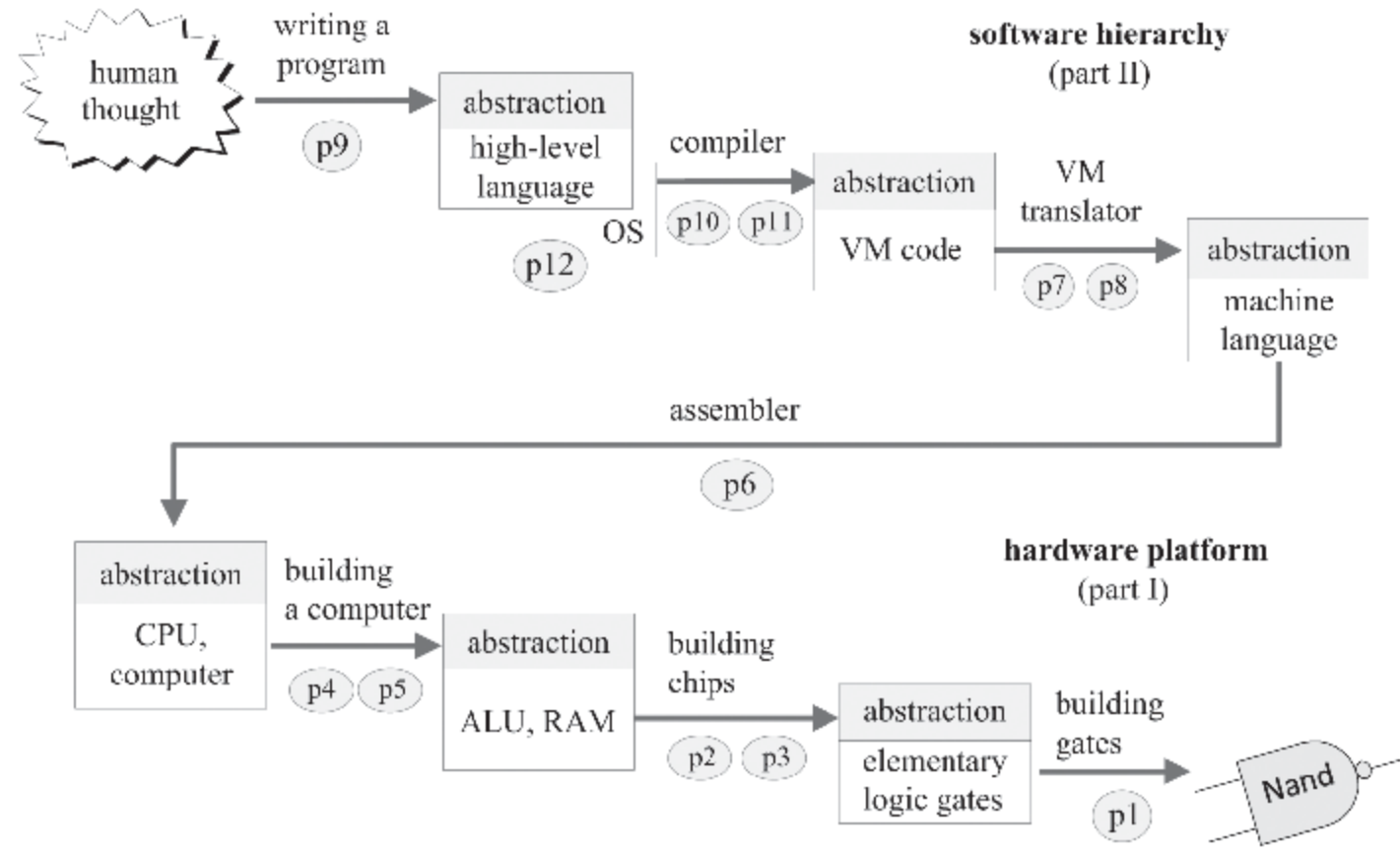


Figure I.1 Major modules of a typical computer system, consisting of a hardware platform and a software hierarchy. Each module has an *abstract view* (also called the module's *interface*) and an *implementation*. The right-pointing arrows signify that each module is implemented using abstract building blocks from the level below. Each circle represents a Nand to Tetris project and chapter—twelve projects and chapters altogether.

We make these observations in order to emphasize the generality of our approach: the challenges, insights, tips, tricks, techniques, and terminology that you will encounter in this book are exactly the same as those encountered by practicing hardware and software engineers. In that respect, Nand to Tetris is a form of initiation: if you'll manage to complete the journey, you will gain an excellent basis for becoming a hardcore computer professional yourself.

So, which specific hardware platform, and which specific high-level language, shall we build in Nand to Tetris? One possibility is building an industrial-strength, widely used computer model and writing a compiler for a popular high-level language. We opted against these choices, for three reasons. First, computer models come and go, and hot programming

languages give way to new ones. Therefore, we didn't want to commit to any particular hardware/software configuration. Second, the computers and languages that are used in practice feature numerous details that have little instructive value, yet take ages to implement. Finally, we sought a hardware platform and a software hierarchy that could be easily controlled, understood, and extended. These considerations led to the creation of Hack, the computer platform built in part I of the book, and Jack, the high-level language implemented in part II.

Typically, computer systems are described *top-down*, showing how high-level abstractions can be reduced to, or realized by, simpler ones. For example, we can describe how binary machine instructions executing on the computer architecture are broken into micro-codes that travel through the architecture's wires and end up manipulating the lower-level ALU and RAM chips. Alternatively, we can go *bottom-up*, describing how the ALU and RAM chips are judiciously designed to execute micro-codes that, taken together, form binary machine instructions. Both the top-down and the bottom-up approaches are enlightening, each giving a different perspective on the system that we are about to build.

In figure I.1, the direction of the arrows suggests a top-down orientation. For any given pair of modules, there is a right-pointing arrow connecting the higher module with the lower one. The meaning of this arrow is precise: it implies that the higher-level module is implemented using abstract building blocks from the level below. For example, a high-level program is implemented by translating each high-level statement into a set of abstract VM commands. And each VM command, in turn, is translated further into a set of abstract machine language instructions. And so it goes. The distinction between *abstraction* and *implementation* plays a major role in systems design, as we now turn to discuss.

Abstraction and Implementation

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 a humongous enterprise! We deal with this complexity by breaking the system into *modules*. Each module is

described separately, in a dedicated chapter, and built separately, in a standalone project. You might then wonder, how is it possible to describe and construct these modules in isolation? Surely they are interrelated! As we will demonstrate 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, if the system is well designed, you can build these modules in any desired order, and even in parallel, if you work in a team.

The cognitive ability to "divide and conquer" a complex system into manageable modules is empowered by yet another cognitive gift: our ability to discern between the *abstraction* and the *implementation* of each module. In computer science, we take these words concretely: abstraction describes what the module does, and implementation describes how it does it. With this distinction in mind, here is the most important rule in system engineering: when using a module as a building block—*any module*—you are to focus exclusively on the module's abstraction, ignoring completely its implementation details.

For example, let's focus on the bottom tier of figure I.1, starting at the "computer architecture" level. As seen in the figure, the implementation of this architecture uses several building blocks from the level below, including a Random Access Memory. The RAM is a remarkable device. It may contain billions of registers, yet any one of them can be accessed directly, and almost instantaneously. Figure I.1 informs us that the computer architect should use this direct-access device abstractly, without paying any attention to how it is actually realized. All the work, cleverness, and drama that went into implementing the direct-access RAM magic—the *how*—should be completely ignored, since this information is irrelevant in the context of *using* the RAM for its effect.

Going one level downward in figure I.1, we now find ourselves in the position of having to build the RAM chip. How should we go about it? Following the right-pointing arrow, we see that the RAM implementation will be based on elementary logic gates and chips from the level below. In particular, the RAM storage and direct-access capabilities will be realized using *registers* and *multiplexers*, respectively. And once again, the same abstraction-implementation principle kicks in: we will use these chips as abstract building blocks, focusing on their interfaces, and caring naught

about *their* implementations. And so it goes, all the way down to the Nand level.

To recap, whenever your implementation uses a lower-level hardware or software module, you are to treat this module as an off-the-shelf, black box abstraction: all you need is the documentation of the module's interface, describing *what* it can do, and off you go. You are to pay no attention whatsoever to *how* the module performs what its interface advertises. This abstraction-implementation paradigm helps developers manage complexity and maintain sanity: by dividing an overwhelming system into well-defined modules, we create manageable chunks of implementation work and localize error detection and correction. This is the most important design principle in hardware and software construction projects.

Needless to say, everything in this story hinges on the intricate art of *modular design*: the human ability to separate the problem at hand into an elegant collection of well-defined modules, each having a clear interface, each representing a reasonable chunk of standalone implementation work, each lending itself to an independent unit-testing program. Indeed, modular design is the bread and butter of applied computer science: every system architect routinely defines abstractions, sometimes referred to as *modules* or *interfaces*, and then implements them, or asks other people to implement them. The abstractions are often built layer upon layer, resulting in higher and higher levels of functionality. If the system architect designs a good set of modules, the implementation work will flow like clear water; if the design is slipshod, the implementation will be doomed.

Modular design is an acquired art, honed by seeing and implementing many well-designed abstractions. That's exactly what you are about to experience in Nand to Tetris: you will learn to appreciate the elegance and functionality of hundreds of hardware and software abstractions. You will then be guided how to implement each one of these abstractions, one step at a time, creating bigger and bigger chunks of functionality. As you push ahead in this journey, going from one chapter to the next, it will be thrilling to look back and appreciate the computer system that is gradually taking shape in the wake of your efforts.

Methodology

The Nand to Tetris journey entails building a hardware platform and a software hierarchy. The hardware platform is based on a set of about thirty logic gates and chips, built in part I of the book. Every one of these gates and chips, including the topmost computer architecture, will be built using a *Hardware Description Language*. The HDL that we will use is documented in appendix 2 and can be learned in about one hour. You will test the correctness of your HDL programs using a software-based hardware simulator running on your PC. This is exactly how hardware engineers work in practice: they build and test chips using software-based simulators. When they are satisfied with the simulated performance of the chips, they ship their specifications (HDL programs) to a fabrication company. Following optimization, the HDL programs become the input of robotic arms that build the hardware in silicon.

Moving up on the Nand to Tetris journey, in part II of the book we will build a software stack that includes an assembler, a virtual machine, and a compiler. These programs can be implemented in any high-level programming language. In addition, we will build a basic operating system, written in Jack.

You may wonder how it is possible to develop these ambitious projects in the scope of one course or one book. Well, in addition to modular design, our secret sauce is reducing design uncertainty to an absolute minimum. We'll provide elaborate scaffolding for each project, including detailed APIs, skeletal programs, test scripts, and staged implementation guidelines.

All the software tools that are necessary for completing projects 1–12 are available in the Nand to Tetris software suite, which can be downloaded freely from www.nand2tetris.org. These include a hardware simulator, a CPU emulator, a VM emulator, and executable versions of the hardware chips, assembler, compiler, and OS. Once you download the software suite to your PC, all these tools will be at your fingertips.

The Road Ahead

The Nand to Tetris journey entails twelve hardware and software construction projects. The general direction of development *across* these projects, as well as the book's table of contents, imply a bottom-up journey: we start with elementary logic gates and work our way upward, leading to a high-level, object-based programming language. At the same time, the direction of development *within* each project is top-down. In particular, whenever we present a hardware or software module, we will always start with an abstract description of *what* the module is designed to do and why it is needed. Once you understand the module's abstraction (a rich world in its own right), you'll proceed to implement it, using abstract building blocks from the level below.

So here, finally, is the grand plan of part I of our tour de force. In chapter 1 we start with a single logic gate—Nand—and build from it a set of elementary and commonly used logic gates like And, Or, Xor, and so on. In chapters 2 and 3 we use these building blocks for constructing an Arithmetic Logic Unit and memory devices, respectively. In chapter 4 we pause our hardware construction journey and introduce a low-level machine language in both its symbolic and binary forms. In chapter 5 we use the previously built ALU and memory units for building a Central Processing Unit (CPU) and a Random Access Memory (RAM). These devices will then be integrated into a hardware platform capable of running programs written in the machine language presented in chapter 4. In chapter 6 we describe and build an *assembler*, which is a program that translates low-level programs written in symbolic machine language into executable binary code. This will complete the construction of the hardware platform. This platform will then become the point of departure for part II of the book, in which we'll extend the barebone hardware with a modern software hierarchy consisting of a virtual machine, a compiler, and an operating system.

We hope that we managed to convey what lies ahead, and that you are eager to get started on this grand voyage 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 (1927–2017)

Every digital device—be it a personal computer, a cell phone, or a network router—is based on a set of chips designed to store and process binary information. Although these chips come in different shapes and forms, they are all made of the same building blocks: elementary *logic gates*. The gates can be physically realized using many different hardware technologies, but their logical behavior, or *abstraction*, is consistent across all implementations.

In this chapter we start out with one primitive logic gate—Nand—and build all the other logic gates that we will need from it. In particular, we will build Not, And, Or, and Xor gates, as well as two gates named *multiplexer* and *demultiplexer* (the function of all these gates is described below). Since our target computer will be designed to operate on 16-bit values, we will also build 16-bit versions of the basic gates, like Not16, And16, and so on. The result will be a rather standard set of logic gates, which will be later used to construct our computer's processing and memory chips. This will be done in chapters 2 and 3, respectively.

The chapter starts with the minimal set of theoretical concepts and practical tools needed for designing and implementing logic gates. In particular, we introduce Boolean algebra and Boolean functions and show how Boolean functions can be realized by logic gates. We then describe how logic gates can be implemented using a Hardware Description Language (HDL) and how these designs can be tested using hardware simulators. This introduction will carry its weight throughout part I of the

book, since Boolean algebra and HDL will come into play in every one of the forthcoming hardware chapters and projects.

1.1 Boolean Algebra

Boolean algebra manipulates two-state 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 representing and manipulating binary values, Boolean functions play a central role in the specification, analysis, and optimization of hardware architectures.

Boolean operators: Figure 1.1 presents three commonly used Boolean functions, also known as *Boolean operators*. These functions are named And, Or, and Not, also written using the notation $x \cdot y$, $x + y$, and $\bar{x}$, or $x \wedge y, x \vee y$, and $\neg x$, respectively. Figure 1.2 gives the definition of *all* the possible Boolean functions that can be defined over two variables, along with their common names. These functions were constructed systematically by enumerating all the possible combinations of values spanned by two binary variables. Each operator has a conventional name that seeks to describe its underlying semantics. For example, the name of the Nand operator is shorthand for *Not-And*, coming from the observation that Nand (x, y) is equivalent to Not (And (x, y)). The Xor operator—shorthand for *exclusive or*—evaluates to 1 when exactly one of its two variables is 1. The Nor gate derives its name from *Not-Or*. All these gate names are not terribly important.

x	y	x And y
0	0	0
0	1	0
1	0	0
1	1	1

x	y	x Or y
0	0	0
0	1	1
1	0	1
1	1	1

x	Not x
0	1
1	0

Figure 1.1 Three elementary Boolean functions.

	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	$x \cdot \overline{y}$	0	0	1	0
x	x	0	0	1	1
Not x And y	$\overline{x} \cdot y$	0	1	0	0
y	y	0	1	0	1
Xor	$x \cdot \overline{y} + \overline{x} \cdot y$	0	1	1	0
Or	$x + y$	0	1	1	1
Nor	$\overline{x+y}$	1	0	0	0
Equivalence	$x \cdot y + \overline{x} \cdot \overline{y}$	1	0	0	1
Not y	$\overline{y}$	1	0	1	0
If y then x	$x + \overline{y}$	1	0	1	1
Not x	$\overline{x}$	1	1	0	0
If x then y	$\overline{x} + y$	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 binary variables. In general, the number of Boolean functions spanned by n binary variables (here $n = 2$) is 2^{2^n} (that's a lot of Boolean functions).

Figure 1.2 begs the question: What makes And, Or, and Not more interesting, or privileged, than any other subset of Boolean operators? The short answer is that indeed there is nothing special about And, Or, and Not. A deeper answer is that various subsets of logical operators can be used for expressing *any* Boolean function, and {And, Or, Not} is one such subset. If you find this claim impressive, consider this: any one of these three basic operators can be expressed using yet another operator—Nand. Now, *that's* impressive! It follows that any Boolean function can be realized using Nand

gates only. Appendix 1, which is an optional reading, provides a proof of this remarkable claim.

Boolean Functions

Every Boolean function can be defined using two alternative representations. First, we can define the function using a *truth table*, as we do in figure 1.3. For each one of the 2^n possible tuples of variable values $v_1, \ldots, v_n$ (here $n = 3$), the table lists the value of $f(v_1, \ldots, v_n)$. In addition to this data-driven definition, we can also define Boolean functions using Boolean expressions, for example, $f(x, y, z) = (x \text{ Or } y) \text{ And Not } (z)$.

x	y	z	$f(x, y, z) = (x \text{ Or } y) \text{ And Not}(z)$
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 1.3 Truth table and functional definitions of a Boolean function (example).

How can we verify that a given Boolean expression is equivalent to a given truth table? Let's use figure 1.3 as an example. Starting with the first row, we compute $f(0, 0, 0)$, which is (0 Or 0) And Not (0). This expression evaluates to 0, the same value listed in the truth table. So far so good. A similar equivalence test can be applied to every row in the table—a rather tedious affair. Instead of using this laborious bottom-up proof technique, we can prove the equivalence top-down, by analyzing the Boolean expression (x Or y) And Not (z). Focusing on the left-hand side of the And operator, we observe that the overall expression evaluates to 1 only when ((x is 1) Or (y

is 1)). Turning to the right-hand side of the And operator, we observe that the overall expression evaluates to 1 only when (*z* is 0). Putting these two observations together, we conclude that the expression evaluates to 1 only when (((*x* is 1) Or (*y* is 1)) And (*z* is 0)). This pattern of 0's and 1's occurs only in rows 3, 5, and 7 of the truth table, and indeed these are the only rows in which the table's rightmost column contains a 1.

Truth Tables and Boolean Expressions

Given a Boolean function of *n* variables represented by a Boolean expression, we can always construct from it the function's truth table. We simply compute the function for every set of values (row) in the table. This construction is laborious, and obvious. At the same time, the dual construction is not obvious at all: Given a truth table representation of a Boolean function, can we always synthesize from it a Boolean expression for the underlying function? The answer to this intriguing question is *yes*. A proof can be found in appendix 1.

When it comes to building computers, the truth table representation, the Boolean expression, and the ability to construct one from the other are all highly relevant. For example, suppose that we are called to build some hardware for sequencing DNA data and that our domain expert biologist wants to describe the sequencing logic using a truth table. Our job is to realize this logic in hardware. Taking the given truth table data as a point of departure, we can synthesize from it a Boolean expression that represents the underlying function. After simplifying the expression using Boolean algebra, we can proceed to implement it using logic gates, as we'll do later in the chapter. To sum up, a truth table is often a convenient means for describing some states of nature, whereas a Boolean expression is a convenient formalism for realizing this description in silicon. The ability to move from one representation to the other is one of the most important practices of hardware design.

We note in passing that although the truth table representation of a Boolean function is unique, every Boolean function can be represented by many different yet equivalent Boolean expressions, and some will be shorter and easier to work with. For example, the expression (Not (*x* And *y*) And (Not (*x*) Or *y*) And (Not (*y*) Or *y*)) is equivalent to the expression Not

(x). We see that the ability to simplify a Boolean expression is the first step toward hardware optimization. This is done using Boolean algebra and common sense, as illustrated in appendix 1.

1.2 Logic Gates

A *gate* is a physical device that implements a simple Boolean function. Although most digital computers today use electricity to realize gates and represent binary data, any alternative technology permitting switching and conducting capabilities can be employed. Indeed, over the years, many hardware implementations of Boolean functions were created, including magnetic, optical, biological, hydraulic, pneumatic, quantum-based, and even domino-based mechanisms (many of these implementations were proposed as whimsical "can do" feats). Today, gates are typically implemented as transistors etched in silicon, packaged as *chips*. In Nand to Tetris we use the words *chip* and *gate* interchangeably, tending to use the latter for simple instances of the former.

The availability of alternative switching technologies, on the one hand, and the observation that Boolean algebra can be used to abstract the behavior of logic gates, on the other, is extremely important. Basically, it implies that computer scientists don't have to worry about physical artifacts like electricity, circuits, switches, relays, and power sources. Instead, computer scientists are content with the abstract notions of Boolean algebra and gate logic, trusting blissfully that someone else—physicists and electrical engineers—will figure out how to actually realize them in hardware. Hence, primitive gates like those shown in figure 1.4 can be viewed as black box devices that implement elementary logical operations in one way or another—we don't care how. The use of Boolean algebra for analyzing the abstract behavior of logic gates was articulated in 1937 by Claude Shannon, leading to what is sometimes described as the most important M.Sc. thesis in computer science.

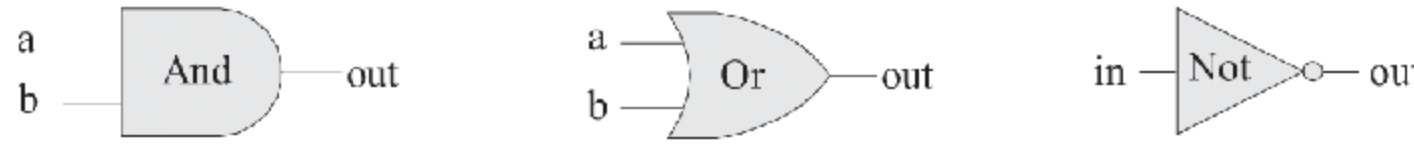


Figure 1.4 Standard gate diagrams of three elementary logic gates.

Primitive and Composite Gates

Since all logic gates have the same input and output data types (0's and 1's), they can be combined, creating *composite gates* of arbitrary complexity. For example, suppose we are asked to implement the three-way Boolean function And (*a*, *b*, *c*), which returns 1 when every one of its inputs is 1, and 0 otherwise. 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 1.5.

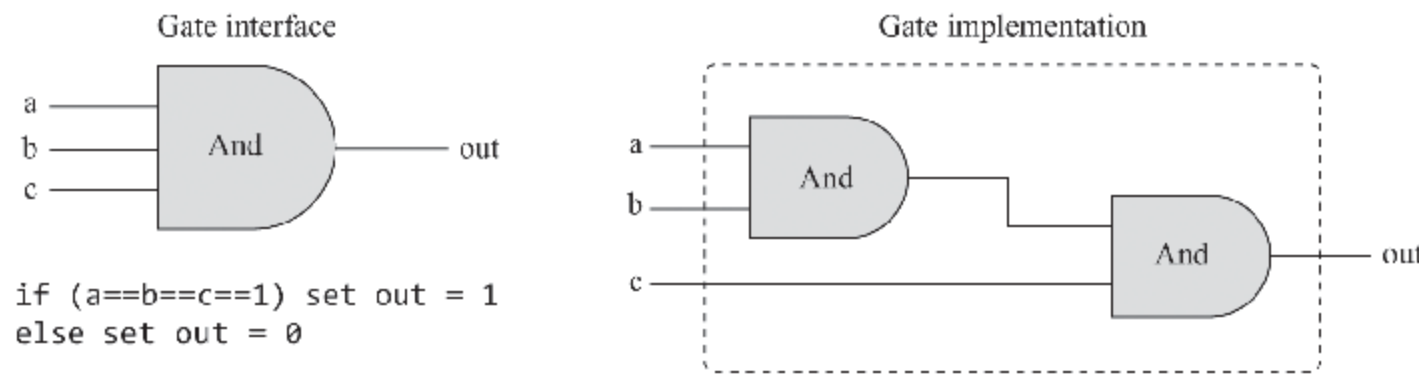


Figure 1.5 Composite implementation of a three-way And gate. The rectangular dashed outline defines the boundary of the gate interface.

We see that any given logic gate can be viewed from two different perspectives: internal and external. The right side of figure 1.5 gives the gate's internal architecture, or *implementation*, whereas the left side shows the gate *interface*, namely, its input and output pins and the behavior that it exposes to the outside world. The internal view is relevant only to the gate builder, whereas the external view is the right level of detail for designers who wish to use the gate as an abstract, off-the-shelf component, without paying attention to its implementation.

Let us consider another logic design example: Xor. By definition, Xor (*a*, *b*) is 1 exactly when either *a* is 1 and *b* is 0 or *a* is 0 and *b* is 1. Said otherwise, Xor (a, b) = Or(And(*a*, Not(*b*)), And (Not (*a*), *b*)). This definition is implemented in the logic design shown in figure 1.6.

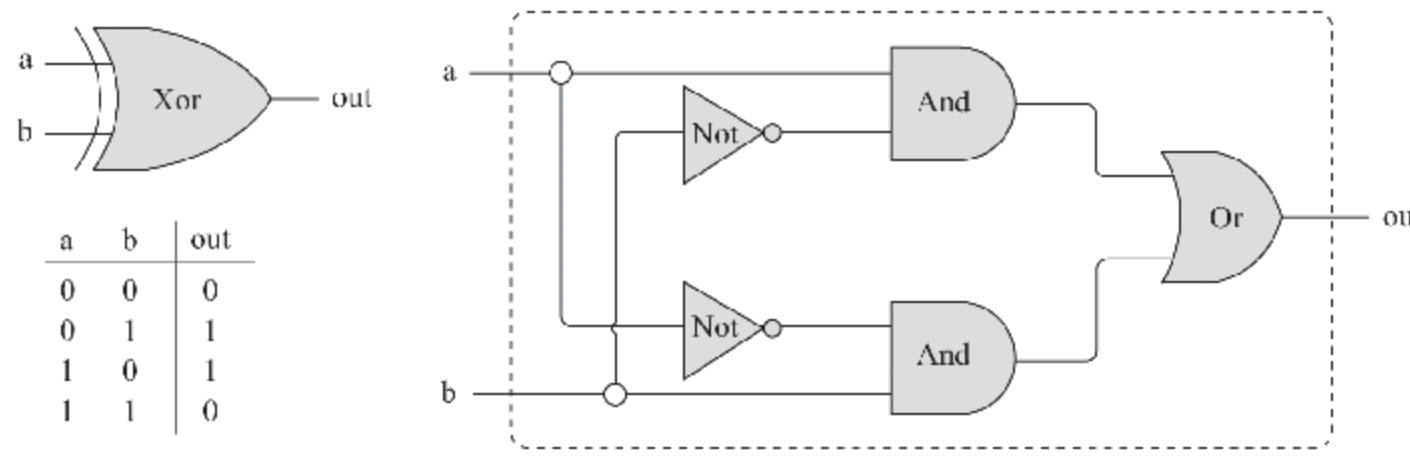


Figure 1.6 Xor gate interface (left) and a possible implementation (right).

Note that the *interface* of any given gate is unique: there is only one way to specify it, and this is normally done using a truth table, a Boolean expression, or a verbal specification. This interface, however, can be realized in many different ways, and some will be more elegant and efficient than others. For example, the Xor implementation shown in figure 1.6 is one possibility; there are more efficient ways to realize Xor, using less logic gates and less inter-gate connections. Thus, from a functional standpoint, the fundamental requirement of logic design is that *the gate implementation will realize its stated interface, one way or another*. From an efficiency standpoint, the general rule is to try to use as few gates as possible, since fewer gates imply less cost, less energy, and faster computation.

To sum up, the art of logic design can be described as follows: Given a gate abstraction (also referred to as *specification*, or *interface*), find an efficient way to implement it using other gates that were already implemented.

1.3 Hardware Construction

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 down payment, 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 port. Our goal is to realize the gate diagram shown in figure 1.6 using this hardware.

We begin by taking two And gates, two Not gates, and one Or gate and mounting them on a board, according to the figure's layout. Next, we connect the chips to one another by running wires among them and 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 wish to construct some other chips in the future, we'll be able to use these Xor gates as black box building blocks, just as we used the And, Or, and Not gates before.

As you have 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's input/output behavior delivers the desired specification. If the chip fails to do so, we will have to tinker with its physical structure—a rather messy affair. Further, even if we do come up with a correct and efficient design, replicating the chip assembly process many times over will be a time-consuming and error-prone affair. There must be a better way!

1.3.1 Hardware Description Language

Today, hardware designers no longer build anything with their bare hands. Instead, they design the chip architecture using a formalism called *Hardware Description Language*, or *HDL*. The designer specifies the chip logic by writing an *HDL program*, which is then subjected to a rigorous battery of tests. The tests are carried out virtually, using computer simulation: A special software tool, called a *hardware simulator*, takes the HDL program as input and creates a software representation of the chip logic. Next, the designer can instruct the simulator to test the virtual chip on

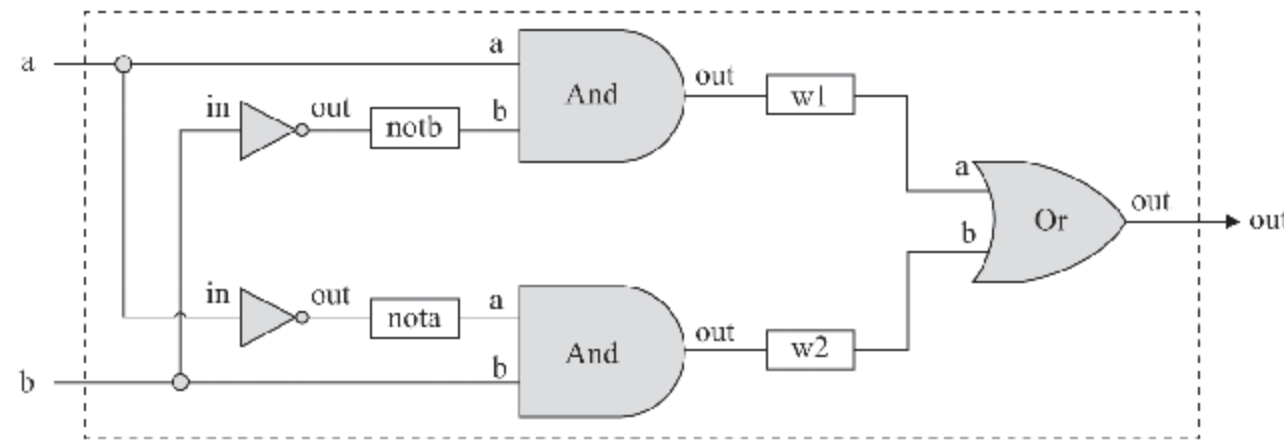
various sets of inputs. The simulator computes the chip outputs, which are then compared to the desired outputs, as mandated by the client who ordered the chip built.

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 proposed chip implementation. 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 an entire chip before a single penny is spent on physical production. When the performance of the simulated chip satisfies the client who ordered it, an optimized version of 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 design process—from an optimized HDL program to mass production—is typically outsourced to companies that specialize in robotic chip fabrication, using one switching technology or another.

Example: Building an Xor Gate: The remainder of this section gives a brief introduction to HDL, using an Xor gate example; a detailed HDL specification can be found in appendix 2.

Let us focus on the bottom left of figure 1.7. An HDL definition of a chip consists of a *header* section and a *parts* section. The header section specifies the chip *interface*, listing the chip name and the names of its input and output pins. The PARTS section describes the chip-parts from which the chip architecture is made. Each chip-part is represented by a single *statement* that specifies the part name, followed by a parenthetical expression that specifies how it is connected to other parts in the design. Note that in order to write such statements, the HDL programmer must have access to the interfaces of all the underlying chip-parts: the names of their input and output pins, as well as their intended operation. For example, the programmer who wrote the HDL program listed in figure 1.7 must have known that the input and output pins of the Not gate are named in and out and that those of the And and Or gates are named a, b, and out. (The APIs of all the chips used in Nand to Tetris are listed in appendix 4).



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 1.7 Gate diagram and HDL implementation of the Boolean function Xor (a, b) = Or (And (a, Not (b)), And (Not (a), b)), used as an example. A test script and an output file generated by the test are also shown. Detailed descriptions of HDL and the testing language are given in appendices 2 and 3, respectively.

Inter-part connections are specified 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 connection) named `nota` and pipes the value of the `out` pin into it. The second statement pipes the value of `nota` into the `a` input of an And gate. Two comments are in order here. First, internal pins are created "automatically" the first time they appear in an HDL program. Second, pins may have an unlimited fan-out. For example, in figure 1.7, each input is simultaneously fed into two gates. In gate diagrams, multiple connections are described by drawing them, creating forked patterns. In HDL programs, the existence of forks is inferred from the code.

The HDL that we use in Nand to Tetris has a similar look and feel to industrial strength HDLs but is much simpler. Our HDL syntax is mostly self-explanatory and can be learned by seeing a few examples and consulting appendix 2, as needed.

Testing

Rigorous quality assurance mandates that chips be tested in a specific, replicable, and well-documented fashion. With that in mind, hardware simulators are typically designed to run *test scripts*, written in a scripting language. The test script listed in figure 1.7 is written in the scripting language understood by the Nand to Tetris hardware simulator.

Let us give a brief overview of this test script. The first two lines 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. In each scenario, the script instructs the simulator to bind the chip inputs to selected 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 input values that the gate can possibly get. In this case, the resulting output file (right side of figure 1.7) provides a complete empirical test that the chip is well behaving. The luxury of such certitude is not feasible in more complex chips, as we will see later.

Readers who plan to build the Hack computer will be pleased to know that all the chips that appear in the book are accompanied by skeletal HDL programs and supplied test scripts, available in the Nand to Tetris software suite. Unlike HDL, which must be learned in order to complete the chip specifications, there is no need to learn our testing language. At the same time, you have to be able to read and understand the supplied test scripts. The scripting language is described in appendix 3, which can be consulted on a need-to-know basis.

1.3.2 Hardware Simulation

Writing and debugging HDL programs is similar to conventional software development. The main difference is that instead of writing code in a high-level language, we write it in HDL, and instead of compiling and running the code, we use a *hardware simulator* to test it. 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 supplied test scripts. There exist many such commercial hardware simulators in the

market. The Nand to Tetris software suite includes a simple hardware simulator that provides all the necessary tools for building, testing, and integrating all the chips presented in the book, leading up to the construction of a general-purpose computer. Figure 1.8 illustrates a typical chip simulation session.

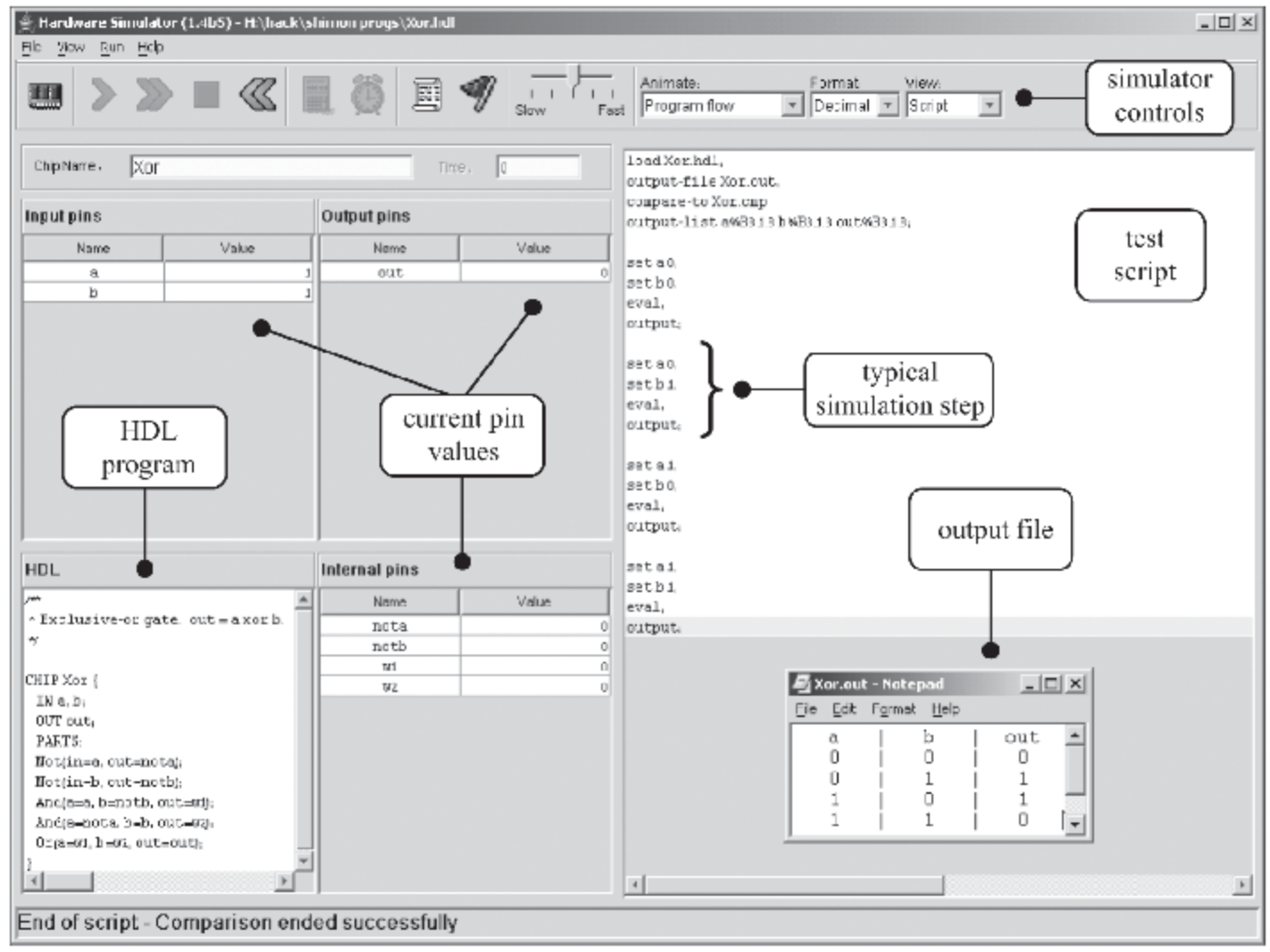


Figure 1.8 A screenshot of simulating an Xor chip in the supplied hardware simulator (other versions of this simulator may have a slightly different GUI). The simulator state is shown just after the test script has completed running. The pin values correspond to the last simulation step (a=b=1). Not shown in this screenshot is a *compare file* that lists the expected output of the simulation specified by this particular test script. Like the test script, the compare file is typically supplied by the client who wants the chip built. In this particular example, the output file generated by the simulation (bottom right of the figure) is identical to the supplied compare file.

1.4 Specification

We now turn to specify a set of logic gates that will be needed for building the chips of our computer system. These gates are ordinary, each designed to carry out a common Boolean operation. For each gate, we'll focus on the gate interface (*what* the gate is supposed to do), delaying implementation details (*how* to build the gate's functionality) to a later section.

1.4.1 Nand

The starting point of our computer architecture is the Nand gate, from which all other gates and chips will be built. The Nand gate realizes the following Boolean function:

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

Or, using API style:

```
Chip name: Nand
Input:     a, b
Output:    out
Function:  if ((a==1) and (b==1)) then out = 0, else out = 1
```

Throughout the book, chips are specified using the API style shown above. For each chip, the API specifies the chip name, the names of its input and output pins, the chip's intended function or operation, and optional comments.

1.4.2 Basic Logic Gates

The logic gates that we present here are typically referred to as *basic*, since they come into play in the construction of more complex chips. The Not, And, Or, and Xor gates implement classical logical operators, and the multiplexer and demultiplexer gates provide means for controlling flows of information.

Not: Also known as *inverter*, this gate outputs the opposite value of its input's value. Here is the API:

```
Chip name: Not
Input:     in
Output:    out
Function:  if (in==0) then out = 1, else out = 0
```

And: Returns 1 when both its inputs are 1, and 0 otherwise:

```
Chip name: And
Input:     a, b
Output:    out
Function:  if ((a==1) and (b==1)) then out = 1, else out = 0
```

Or: Returns 1 when at least one of its inputs is 1, and 0 otherwise:

```
Chip name: Or
Input:     a, b
Output:    out
Function:  if ((a==0) and (b==0)) then out = 0, else out = 1
```

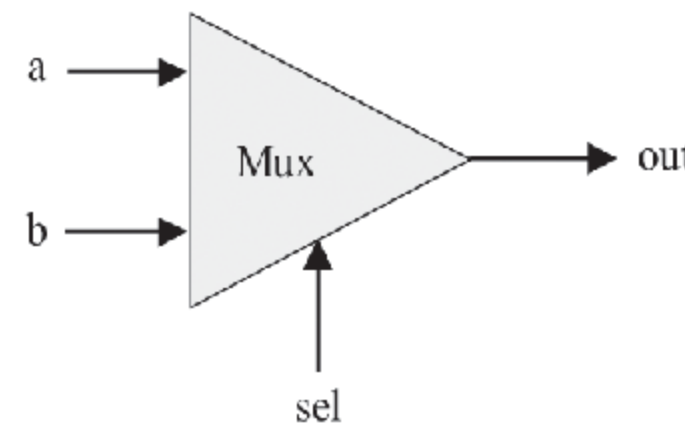
Xor: Also known as *exclusive or*, this gate returns 1 when exactly one of its inputs is 1, and 0 otherwise:

```
Chip name: Xor
Input:     a, b
Output:    out
Function:  if (a!=b) then out = 1, else out = 0
```

Multiplexer: A multiplexer is a three-input gate (see figure 1.9). Two input bits, named a and b, are interpreted as *data bits*, and a third input bit, named sel, is interpreted as a *selection bit*. The multiplexer uses sel to select and output the value of either a or b. Thus, a sensible name for this device could have been *selector*. The name *multiplexer* was adopted from communications systems, where extended versions of this device are used for serializing (multiplexing) several input signals over a single communications channel.

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

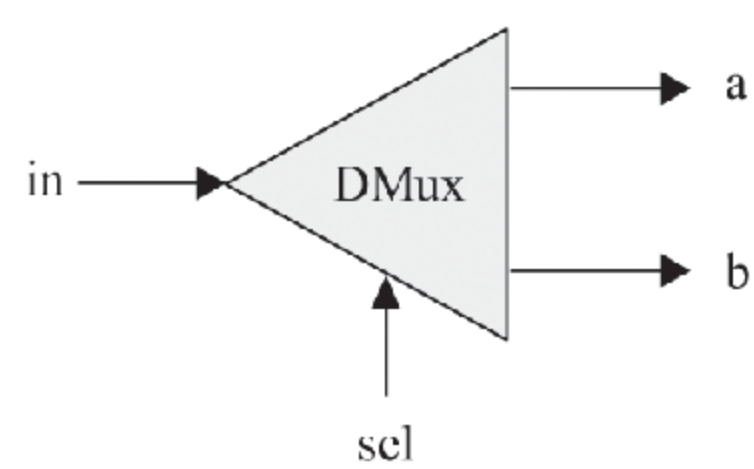



```
Chip name: Mux
Input:     a, b, sel
Output:    out
Function:  if (sel==0) then out = a, else out = b
```

Figure 1.9 Multiplexer. The table at the top right is an abbreviated version of the truth table.

Demultiplexer: A demultiplexer performs the opposite function of a multiplexer: it takes a single input value and routes it to one of two possible outputs, according to a selector bit that selects the destination output. The other output is set to 0. Figure 1.10 gives the API.

sel	a	b
0	in	0
1	0	in




```
Chip name: DMux
Input:     in, sel
Output:    a, b
Function:  if (sel==0) then {a, b} = {in, 0},
           else             {a, b} = {0, in}
```

Figure 1.10 Demultiplexer.

1.4.3 Multi-Bit Versions of Basic Gates

Computer hardware is often designed to process multi-bit values—for example, computing a bitwise And function on two given 16-bit inputs. This section describes several 16-bit logic gates that will be needed for constructing our target computer platform. We note in passing that the logical architecture of these *n*-bit gates is the same, irrespective of *n*'s value (e.g., 16, 32, or 64 bits). HDL programs treat multi-bit values like single-bit values, except that the values can be indexed in order to access individual bits. For example, if `in` and `out` represent 16-bit values, then `out[3]=in[5]` sets the 3rd bit of `out` to the value of the 5th bit of `in`. The bits are indexed from right to left, the rightmost bit being the 0'th bit and the leftmost bit being the 15'th bit (in a 16-bit setting).

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

```
Chip name: Not16
Input:     in[16]
Output:    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 respective pair in its two *n*-bit inputs:

```
Chip name: And16
Input:     a[16], b[16]
Output:    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 respective pair in its two *n*-bit inputs:

```
Chip name: Or16
Input:     a[16], b[16]
Output:    out[16]
Function:  for i = 0..15 out[i] = Or(a[i], b[i])
```

Multi-bit multiplexer: An *n*-bit multiplexer operates exactly the same as a basic multiplexer, except that its inputs and output are *n*-bits wide:

```
Chip name: Mux16
Input:     a[16], b[16], sel
Output:    out[16]
Function:  if (sel==0) then for i = 0..15 out[i] = a[i],
           else for i = 0..15 out[i] = b[i]
```

1.4.4 Multi-Way Versions of Basic Gates

Logic gates that operate on one or two inputs have natural generalization to multi-way variants that operate on more than two inputs. This section describes a set of multi-way gates that will be used subsequently in various chips in our computer architecture.

Multi-way Or: An *m*-way Or gate outputs 1 when at least one of its *m* input bits is 1, and 0 otherwise. We will need an 8-way variant of this gate:

```
Chip name: Or8Way
Input:     in[8]
Output:    out
Function:  out = Or(in[0], in[1],… ,in[7])
```

Multi-way/Multi-bit multiplexer: An *m*-way *n*-bit multiplexer selects one of its *m* *n*-bit inputs, and outputs it to its *n*-bit output. The selection is specified by a set of *k* selection bits, where $k = log_2 m$. Here is the API of a 4-way multiplexer: