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.

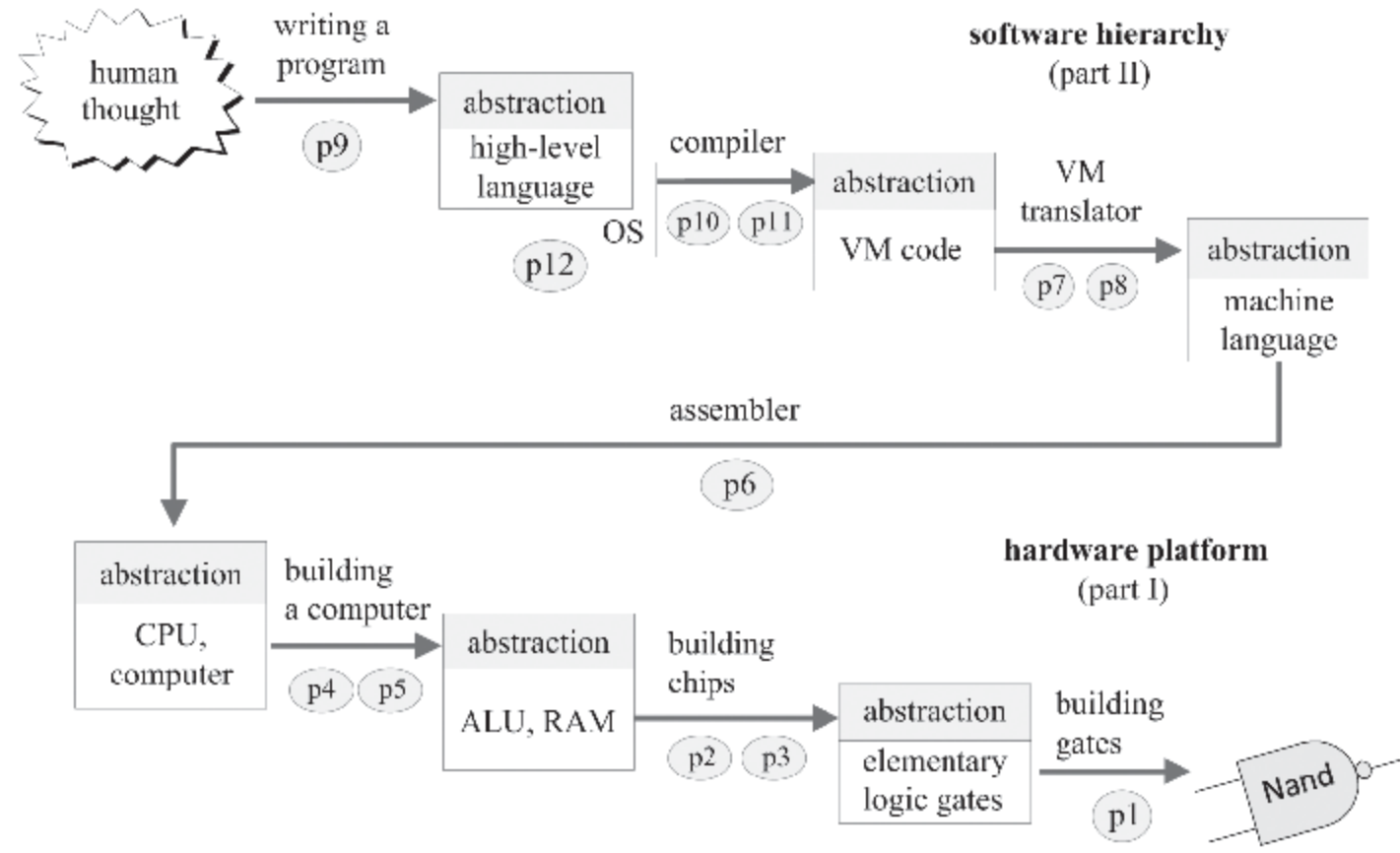


Figure I.1 Major modules of a typical computer system, consisting of a hardware platform and a software hierarchy. Each module has an *abstract view* (also called the module's *interface*) and an *implementation*. The right-pointing arrows signify that each module is implemented using abstract building 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.

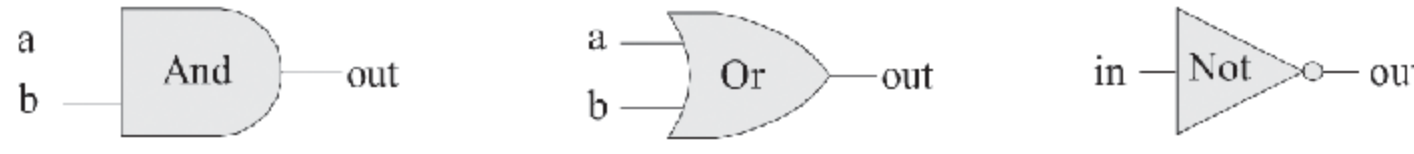


Figure 1.4 Standard gate diagrams of three elementary logic gates.

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.

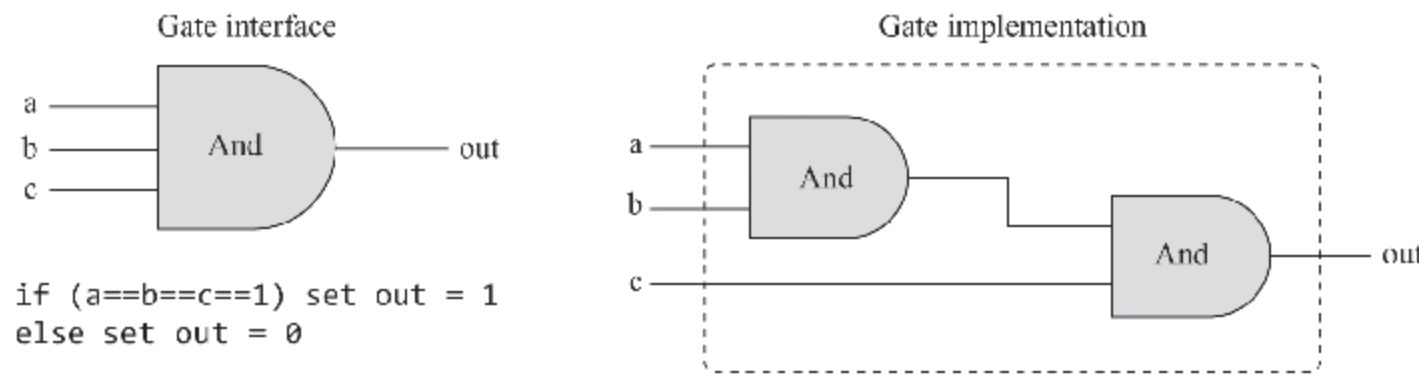


Figure 1.5 Composite implementation of a three-way And gate. The rectangular dashed outline defines the boundary of the gate interface.

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.

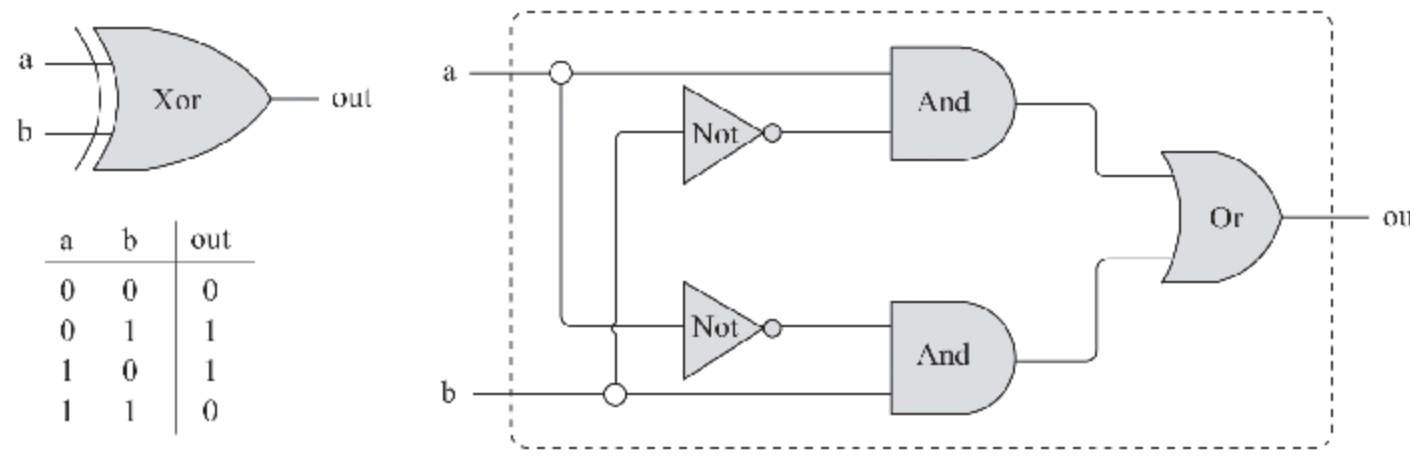


Figure 1.6 Xor gate interface (left) and a possible implementation (right).

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

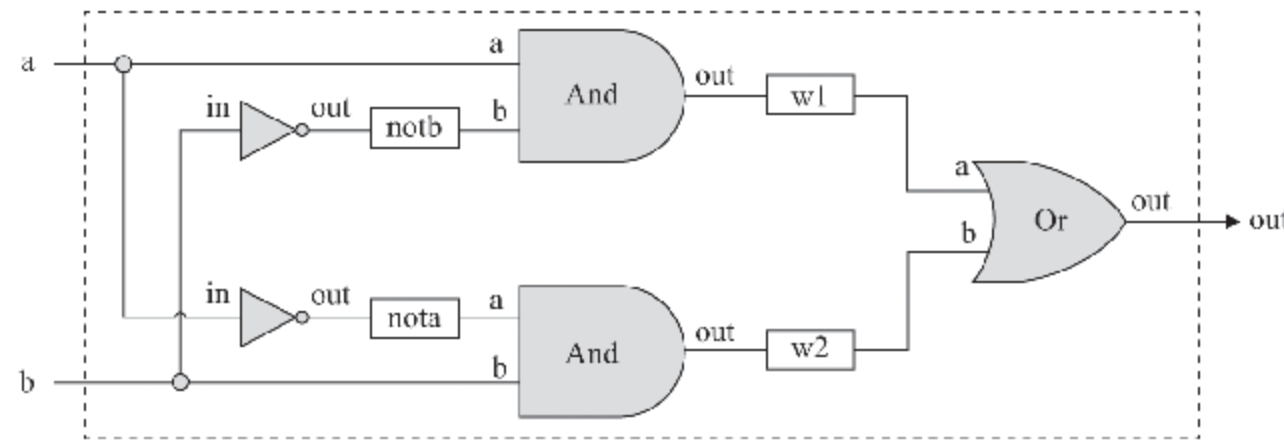
various sets of inputs. The simulator computes the chip outputs, which are then compared to the desired outputs, as mandated by the client who ordered the chip built.

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.

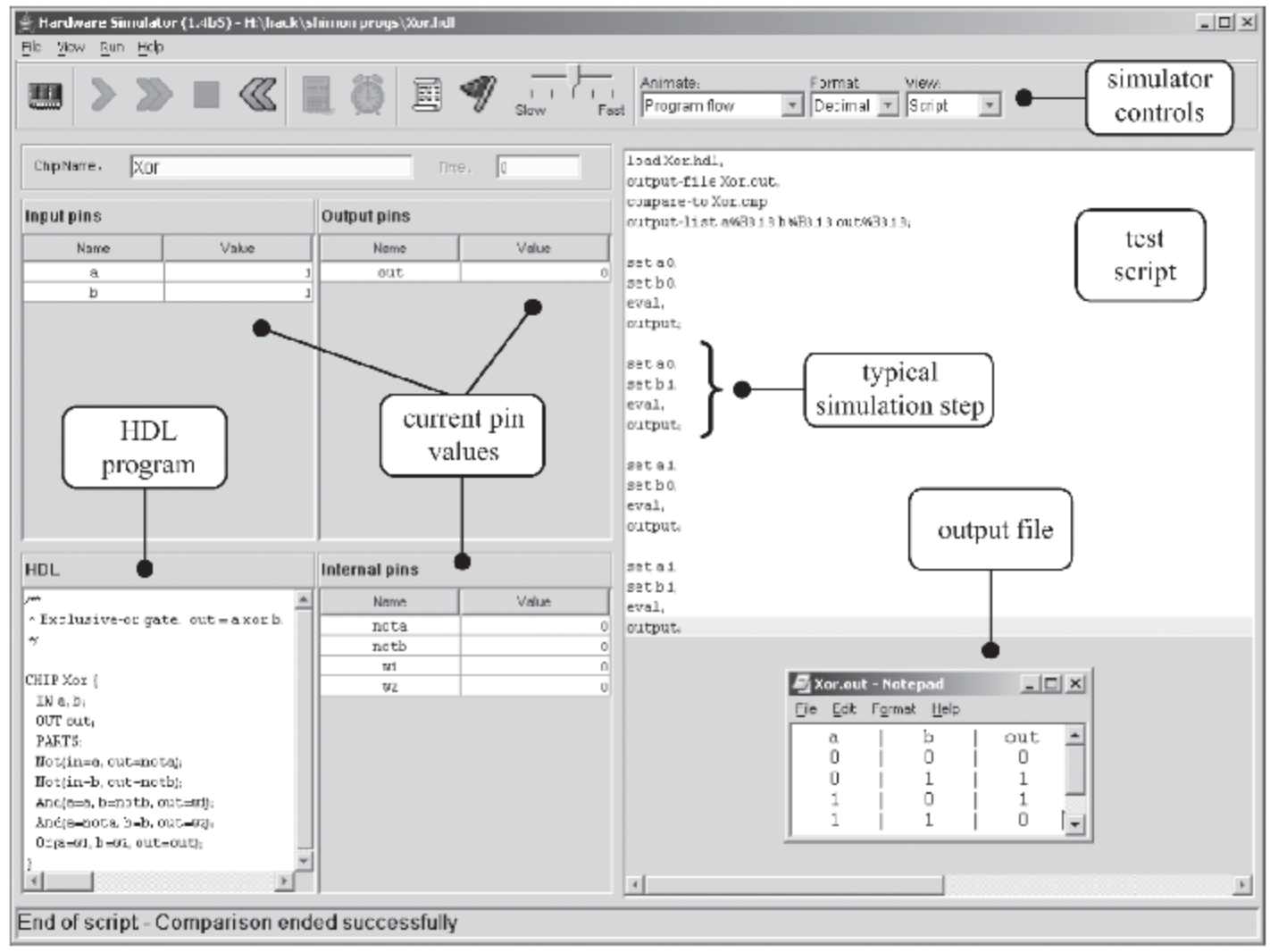


Figure 1.8 A screenshot of simulating an Xor chip in the supplied hardware simulator (other versions of this simulator may have a slightly different GUI). The simulator state is shown just after the test script has completed running. The pin values correspond to the last simulation step (a=b=1). Not 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
```

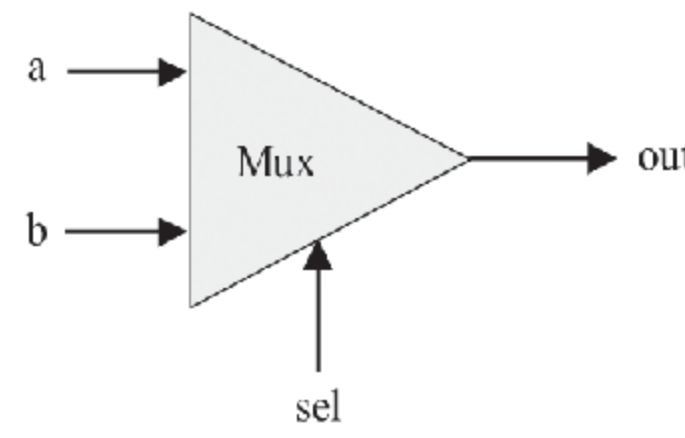
Xor: Also known as *exclusive or*, this gate returns 1 when exactly one of its inputs is 1, and 0 otherwise:

```
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

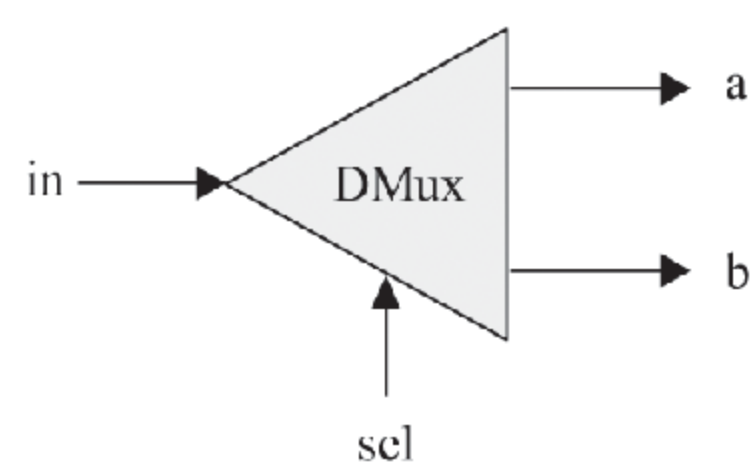



```
Chip name: Mux
Input:     a, b, sel
Output:    out
Function:  if (sel==0) then out = a, else out = b
```

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

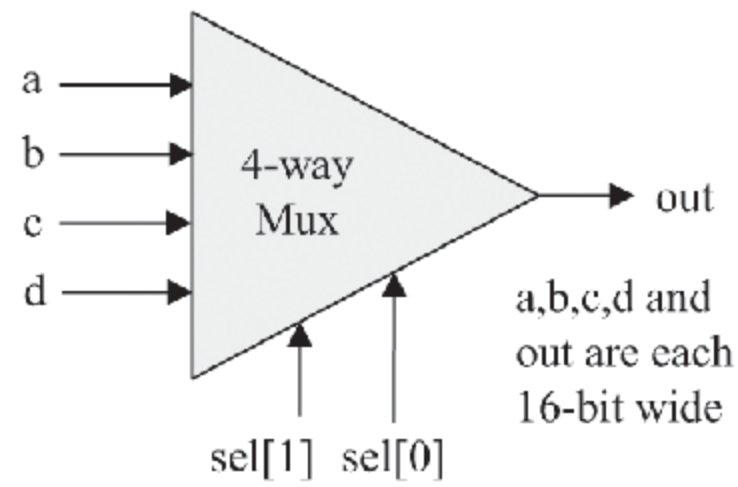
Logic gates that operate on one or two inputs have natural generalization to multi-way variants that operate on more than two inputs. This section describes a set of multi-way gates that will be used subsequently in various chips in our computer architecture.

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:

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



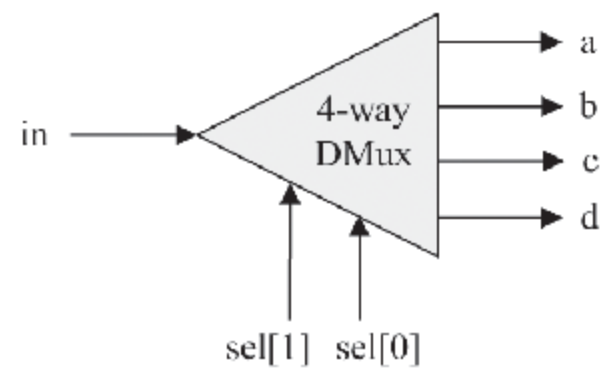
Our target computer platform requires two variants of this chip: a 4-way 16-bit multiplexer and an 8-way 16-bit multiplexer:

```
Chip name: Mux4Way16
Input:     a[16],b[16],c[16],d[16],sel[2]
Output:    out[16]
Function:  if (sel==00,01,10, or 11) then out = a,b,c, or d
Comment:   The assignment is a 16-bit operation.
           For example, "out = a" means "for i = 0..15 out[i] = a[i]".

Chip name: Mux8Way16
Input:     a[16],b[16],c[16],d[16],e[16],f[16],g[16],h[16], sel[3]
Output:    out[16]
Function:  if (sel==000,001,010, ..., or 111)
           then out = a,b,c,d, ..., or h
Comment:   The assignment is a 16-bit operation.
           For example, "out = a" means "for i = 0..15 out[i] = a[i]".
```

Multi-way/Multi-bit demultiplexer: An *m*-way *n*-bit demultiplexer routes its single *n*-bit input to one of its *m* *n*-bit outputs. The other outputs are set to 0. The selection is specified by a set of *k* selection bits, where $k = log_2 m$. Here is the API of a 4-way demultiplexer:

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



Our target computer platform requires two variants of this chip: a 4-way 1-bit demultiplexer and an 8-way 1-bit demultiplexer:

```
Chip name: DMux4Way
Input:     in, sel[2]
Output:    a, b, c, d
Function:  if (sel==00)      then {a,b,c,d} = {1,0,0,0},
           else if (sel==01) then {a,b,c,d} = {0,1,0,0},
           else if (sel==10) then {a,b,c,d} = {0,0,1,0},
           else if (sel==11) then {a,b,c,d} = {0,0,0,1}

Chip name: Dmux8Way
Input:     in, sel[3]
Output:    a, b, c, d, e, f, g, h
Function:  if (sel==000)      then {a,b,c,…, h} = {1,0,0,0,0,0,0,0},
           else if (sel==001) then {a,b,c,…, h} = {0,1,0,0,0,0,0,0},
           else if (sel==010) then {a,b,c,…, h} = {0,0,1,0,0,0,0,0},
           …
           else if (sel==111) then {a,b,c,…, h} = {0,0,0,0,0,0,0,1}
```

1.5 Implementation

The previous section described the specifications, or interfaces, of a family of basic logic gates. Having described the *what*, we now turn to discuss the *how*. In particular, we'll focus on two general approaches to implementing logic gates: *behavioral simulation* and *hardware implementation*. Both approaches play important roles in all our hardware construction projects.

1.5.1 Behavioral Simulation

The chip descriptions presented thus far are strictly abstract. It would have been nice if we could experiment with these abstractions hands-on, before setting out to build them in HDL. How can we possibly do so?

Well, if all we want to do is interact with the chips' behavior, we don't have to go through the trouble of building them in HDL. Instead, we can opt for a much simpler implementation, using conventional programming. For example, we can use some object-oriented language to create a set of classes, each implementing a generic chip. We can write class constructors

for creating chip instances and *eval* methods for evaluating their logic, and we can have the classes interact with each other so that high-level chips can be defined in terms of lower-level ones. We could then add a nice graphical user interface that enables putting different values in the chip inputs, evaluating their logic, and observing the chip outputs. This software-based technique, called *behavioral simulation*, makes a lot of sense. It enables experimenting with chip interfaces before starting the laborious process of building them in HDL.

The Nand to Tetris *hardware simulator* provides exactly such a service. In addition to simulating the behavior of HDL programs, which is its main purpose, the simulator features built-in software implementations of all the chips built in the Nand to Tetris hardware projects. The built-in version of each chip is implemented as an executable software module, invoked by a skeletal HDL program that provides the chip interface. For example, here is the HDL program that implements the built-in version of the Xor chip:

```
/* Xor (exclusive or) gate:
   If a!=b out=1 else out=0. */
CHIP Xor {
    IN  a, b;
    OUT out;
    BUILTIN Xor;
}
```

Compare this to the HDL program listed in figure 1.7. First, note that regular chips and built-in chips have precisely the same interface. Thus, they provide exactly the same functionality. In the built-in implementation though, the PARTS section is replaced with the single statement BUILTIN Xor. This statement informs the simulator that the chip is implemented by Xor.class. This class file, like all the Java class files that implement built-in chips, is located in the folder nand2tetris/tools/builtIn.

We note in passing that realizing logic gates using high-level programming is not difficult, and that's another virtue of behavioral simulation: it's inexpensive and quick. At some point, of course, hardware engineers must do the real thing, which is implementing the chips not as

software artifacts but rather as HDL programs that can be committed to silicon. That's what we'll do next.

1.5.2 Hardware Implementation

This section gives guidelines on how to implement the fifteen logic gates described in this chapter. As a rule in this book, our implementation guidelines are intentionally brief. We give just enough insights to get started, leaving you the pleasure of discovering the rest of the gate implementations yourself.

Nand: Since we decided to base our hardware on elementary Nand gates, we treat Nand as a primitive gate whose functionality is given externally. The supplied hardware simulator features a built-in implementation of Nand, and thus there is no need to implement it.

Not: Can be implemented using a single Nand gate. *Tip:* Inspect the Nand truth table, and ask yourself how the Nand inputs can be arranged so that a single input signal, 0, will cause the Nand gate to output 1, and a single input signal, 1, will cause it to output 0.

And: Can be implemented from the two previously discussed gates.

Or / Xor: The Boolean function Or can be defined using the Boolean functions And and Not. The Boolean function Xor can be defined using And, Not, and Or.

Multiplexer / Demultiplexer: Can be implemented using previously built gates.

Multi-bit Not / And / Or gates: Assuming that you've already built the basic versions of these gates, the implementation of their *n*-ary versions is a matter of arranging arrays of *n* basic gates and having each gate operate separately on its single-bit inputs. The resulting HDL code will be somewhat boring and repetitive (using copy-paste), but it will carry its weight when these multi-bit gates are used in the construction of more complex chips, later in the book.

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

Multi-way gates: Implementation tip: Think forks.

1.5.3 Built-In Chips

As we pointed out when we discussed behavioral simulation, our hardware simulator provides software-based, built-in implementations of most of the chips described in the book. In Nand to Tetris, the most celebrated built-in chip is of course Nand: whenever you use a Nand chip-part in an HDL program, the hardware simulator invokes the built-in tools/builtIn/Nand.hdl implementation. This convention is a special case of a more general chip invocation strategy: whenever the hardware simulator encounters a chip-part, say, *Xxx*, in an HDL program, it looks up the file *Xxx*.hdl in the current folder; if the file is found, the simulator evaluates its underlying HDL code. If the file is not found, the simulator looks it up in the tools/builtIn folder. If the file is found there, the simulator executes the chip's built-in implementation; otherwise, the simulator issues an error message and terminates the simulation.

This convention comes in handy. For example, suppose you began implementing a Mux.hdl program, but, for some reason, you did not complete it. This could be an annoying setback, since, in theory, you cannot continue building chips that use Mux as a chip-part. Fortunately, and actually by design, this is where built-in chips come to the rescue. All you have to do is rename your partial implementation Mux1.hdl, for example. Each time the hardware simulator is called to simulate the functionality of a Mux chip-part, it will fail to find a Mux.hdl file in the current folder. This will cause behavioral simulation to kick in, forcing the simulator to use the built-in Mux version instead. Exactly what we want! At a later stage you may want to go back to Mux1.hdl and resume working on its implementation. At this point you can restore its original file name, Mux.hdl, and continue from where you left off.

1.6 Project

This section describes the tools and resources needed for completing project 1 and gives recommended implementation steps and tips.

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: We assume that you've already downloaded the Nand to Tetris zip file, containing the book's software suite, and that you've extracted it into a folder named nand2tetris on your computer. If that is the case, then the nand2tetris/tools folder on your computer contains the hardware simulator discussed in this chapter. This program, along with a plain text editor, are the only tools needed for completing project 1 as well as all the other hardware projects described in the book.

The fifteen chips mentioned in this chapter, except for Nand, should be implemented in the HDL language described in appendix 2. For each chip *Xxx*, we provide a skeletal *Xxx*.hdl program (sometimes called a *stub file*) with a missing implementation part. In addition, for each chip we provide an *Xxx*.tst script that tells the hardware simulator how to test it, along with an *Xxx*.cmp compare file that lists the correct output that the supplied test is expected to generate. All these files are available in your nand2tetris/projects/01 folder. Your job is to complete and test all the *Xxx*.hdl files in this folder. These files can be written and edited using any plain text editor.

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 the actual outputs generated by the simulator disagree with the desired outputs, the simulator will stop the simulation and produce an error message.

Steps: We recommend proceeding in the following order:

0. The *hardware simulator* needed for this project is available in nand2tetris/tools.
1. Consult appendix 2 (HDL), as needed.
2. Consult the Hardware Simulator Tutorial (available at www.nand2tetris.org), as needed.
3. Build and simulate all the chips listed in nand2tetris/projects/01.

General Implementation Tips

(We use the terms *gate* and *chip* interchangeably.)

- Each gate can be implemented in more than one way. The simpler the implementation, the better. As a general rule, strive to use as few chip-parts as possible.
- Although each chip can be implemented directly from Nand gates only, we recommend always using composite gates that were already implemented. See the previous tip.
- There is no need to build "helper chips" of your own design. Your HDL programs should use only the chips mentioned in this chapter.
- Implement the chips in the order in which they appear in the chapter. If, for some reason, you don't complete the HDL implementation of some chip, you can still use it as a chip-part in other HDL programs. Simply rename the chip file, or remove it from the folder, causing the simulator to use its built-in version instead.

A web-based version of project 1 is available at www.nand2tetris.org.

1.7 Perspective

This chapter specified a set of basic logic gates that are widely used in computer architectures. In chapters 2 and 3 we will use these gates for building our processing and storage chips, respectively. These chips, in turn, will be later used for constructing the central processing unit and the memory devices of our computer.

Although we have chosen to use Nand as our basic building block, other logic gates can be used as possible points of departure. For example, you can build a complete computer platform using Nor gates only or, alternatively, a combination of And, Or, and Not gates. These constructive approaches to logic design are theoretically equivalent, just like the same geometry can be founded on alternative sets of agreed-upon axioms. In principle, if electrical engineers or physicists can come up with efficient and low-cost implementations of logic gates using any technology that they see fit, we will happily use them as primitive building blocks. The reality, though, is that most computers are built from either Nand or Nor gates.

Throughout the chapter, we paid no attention to efficiency and cost considerations, such as energy consumption or the number of wire crossovers implied by our HDL programs. Such considerations are critically important in practice, and a great deal of computer science and technology expertise focuses on optimizing them. Another issue we did not address is physical aspects, for example, how primitive logic gates can be built from transistors embedded in silicon or from other switching technologies. There are of course several such implementation options, each having its own characteristics (speed, energy consumption, production cost, and so on). Any nontrivial coverage of these issues requires venturing into areas outside computer science, like electrical engineering and solid-state physics.

The next chapter describes how bits can be used to represent binary numbers and how logic gates can be used to realize arithmetic operations. These capabilities will be based on the elementary logic gates built in this chapter.

2 Boolean Arithmetic

Counting is the religion of this generation, its hope and salvation.
—Gertrude Stein (1874–1946)

In this chapter we build a family of chips designed to represent numbers and perform arithmetic operations. Our starting point is the set of logic gates built in chapter 1, and our ending point is a fully functional *Arithmetic Logic Unit*. The ALU will later become the computational centerpiece of the *Central Processing Unit* (CPU)—the chip that executes all the instructions handled by the computer. Hence, building the ALU is an important milestone in our Nand to Tetris journey.

As usual, we approach this task gradually, starting with a background section that describes how binary codes and Boolean arithmetic can be used, respectively, to represent and add signed integers. The Specification section presents a succession of *adder chips* designed to add two bits, three bits, and pairs of *n*-bit binary numbers. This sets the stage for the ALU specification, which is based on a surprisingly simple logic design. The Implementation and Project sections provide tips and guidelines on how to build the adder chips and the ALU using HDL and the supplied hardware simulator.

2.1 Arithmetic Operations

General-purpose computer systems are required to perform at least the following arithmetic operations on signed integers:

- addition
- sign conversion
- subtraction
- comparison
- multiplication
- division

We'll start by developing gate logic that carries out addition and sign conversion. Later, we will show how the other arithmetic operations can be implemented from these two building blocks.

In mathematics as well as in computer science, *addition* is a simple operation that runs deep. Remarkably, all the functions performed by digital computers—not only arithmetic operations—can be reduced to adding binary numbers. Therefore, constructive understanding of binary addition holds the key to understanding many fundamental operations performed by the computer's hardware.

2.2 Binary Numbers

When we are told that a certain code, say, 6083, represents a number using the *decimal system*, then, by convention, we take this number to be:

$$(6083)_{10} = 6 \cdot 10^3 + 0 \cdot 10^2 + 8 \cdot 10^1 + 3 \cdot 10^0 = 6083$$

Each digit in the decimal code contributes a value that depends on the *base* 10 and on the digit's position in the code. Suppose now that we are told that the code 10011 represents a number using base 2, or *binary* representation. To compute the value of this number, we follow exactly the same procedure, using base 2 instead of base 10:

$$(10011)_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 19$$

Inside computers, *everything* is represented using binary codes. For example, when we press the keyboard keys labeled 1, 9, and Enter in response to "Give an example of a prime number," what ends up stored in the computer's memory is the binary code 10011. When we ask the computer to display this value on the screen, the following process ensues. First, the computer's operating system calculates the decimal value that 10011 represents, which happens to be 19. After converting this integer value to the two characters 1 and 9, the OS looks up the current font and gets the two bitmap images used for rendering these characters on the screen. The OS then causes the screen driver to turn on and off the relevant pixels, and, don't hold your breath—the whole thing lasts a tiny fraction of a second—we finally see the image 19 appear on the screen.

In chapter 12 we'll develop an operating system that carries out such rendering operations, among many other low-level services. For now, suffice it to observe that the decimal representation of numbers is a human indulgence explained by the obscure fact that, at some point in ancient history, humans decided to represent quantities using their ten fingers, and the habit stuck. From a mathematical perspective, the number ten is utterly uninteresting, and, as far as computers go, is a complete nuisance. Computers handle *everything* in binary and care naught about decimal. Yet since humans insist on dealing with numbers using decimal codes, computers have to work hard behind the scenes to carry out binary-to-decimal and decimal-to-binary conversions whenever humans want to see, or supply, numeric information. At all other times, computers stick to binary.

Fixed word size: Integer numbers are of course unbounded: for any given number x there are integers that are less than x and integers that are greater than x. Yet computers are finite machines that use a fixed word size for representing numbers. *Word size* is a common hardware term used for specifying the number of bits that computers use for representing a basic chunk of information—in this case, integer values. Typically, 8-, 16-, 32-, or 64-bit registers are used for representing integers.[1] The fixed word size implies that there is a limit on the number of values that these registers can represent.

For example, suppose we use 8-bit registers for representing integers. This representation can code $2^8 = 256$ different things. If we wish to represent only nonnegative integers, we can assign 00000000 for representing 0, 00000001 for representing 1, 00000010 for representing 2, 00000011 for representing 3, all the way up to assigning 11111111 for representing 255. In general, using n bits we can represent all the nonnegative integers ranging from 0 to $2^n - 1$.

What about representing negative numbers using binary codes? Later in the chapter we'll present a technique that meets this challenge in a most elegant and satisfying way.

And what about representing numbers that are greater than, or less than, the maximal and minimal values permitted by the fixed register size? Every high-level language provides abstractions for handling numbers that are as large or as small as we can practically want. These abstractions are typically implemented by lashing together as many n-bit registers as necessary for representing the numbers. Since executing arithmetic and logical operations on multi-word numbers is a slow affair, it is recommended to use this practice only when the application requires processing extremely large or extremely small numbers.

2.3 Binary Addition

A pair of binary numbers can be added bitwise from right to left, using the same decimal addition algorithm learned in elementary school. First, we add the two rightmost bits, also called the *least significant bits* (LSB) of the two binary numbers. Next, we add the resulting carry bit to the sum of the next pair of bits. We continue this lockstep process until the two left *most significant bits* (MSB) are added. Here is an example of this algorithm in action, assuming that we use a fixed word size of 4 bits:

```
  0   0   0   1          (carry)        1   1   1   1
      1   0   0   1          x              1   0   1   1
+     0   1   0   1          y          +   0   1   1   1
  0   1   1   1   0      x + y          1   0   0   1   0
      no overflow                           overflow
```

If the most significant bitwise addition generates a carry of 1, we have what is known as *overflow*. What to do with overflow is a matter of decision, and ours is to ignore it. Basically, we are content to guarantee that the result of adding any two n-bit numbers will be correct up to n bits. We note in passing that ignoring things is perfectly acceptable as long as one is clear and forthcoming about it.

2.4 Signed Binary Numbers

An n-bit binary system can code 2^n different things. If we have to represent signed (positive and negative) numbers in binary code, a natural solution is to split the available code space into two subsets: one for representing nonnegative numbers, and the other for representing negative numbers. Ideally, the coding scheme should be chosen such that the introduction of signed numbers would complicate the hardware implementation of arithmetic operations as little as possible.

Over the years, this challenge has led to the development of several coding schemes for representing signed numbers in binary code. The solution used today in almost all computers is called the *two's complement* method, also known as *radix complement*. In a binary system that uses a word size of n bits, the two's complement binary code that represents negative x is taken to be the code that represents $2^n - x$. For example, in a 4-bit binary system, -7 is represented using the binary code associated with $2^4 - 7 = 9$, which happens to be 1001. Recalling that +7 is represented by 0111, we see that $1001 + 0111 = 0000$ (ignoring the overflow bit). Figure 2.1 lists all the signed numbers represented by a 4-bit system using the two's complement method.

0000:	0	
0001:	1	
0010:	2	
0011:	3	
0100:	4	
0101:	5	
0110:	6	
0111:	7	
1000:	–8	(16 – 8)
1001:	–7	(16 – 7)
1010:	–6	(16 – 6)
1011:	–5	(16 – 5)
1100:	–4	(16 – 4)
1101:	–3	(16 – 3)
1110:	–2	(16 – 2)
1111:	–1	(16 – 1)

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

An inspection of figure 2.1 suggests that an n-bit binary system with two's complement representation has the following attractive properties:

- The system codes 2^n signed numbers, ranging from $-(2^{n-1})$ to $2^{n-1} - 1$.
- The code of any nonnegative number begins with a 0.
- The code of any negative number begins with a 1.
- To obtain the binary code of $-x$ from the binary code of x, leave all the least significant 0-bits and the first least significant 1-bit of x intact, and flip all the remaining bits (convert 0's to 1's and vice versa). Alternatively, flip all the bits of x and add 1 to the result.

A particularly attractive feature of the two's complement representation is that *subtraction* is handled as a special case of addition. To illustrate, consider $5-7$. Noting that this is equivalent to $5+(-7)$, and following figure 2.1, we proceed to compute `0101 + 1001`. The result is `1110`, which indeed is the binary code of -2. Here is another example: To compute $(-2)+(-3)$, we add `1110 + 1101`, obtaining the sum `11011`. Ignoring the overflow bit, we get `1011`, which is the binary code of -5.

We see that the two's complement method enables adding and subtracting signed numbers using nothing more than the hardware required for adding nonnegative numbers. As we will see later in the book, every arithmetic operation, from multiplication to division to square root, can be implemented reductively using binary addition. So, on the one hand, we observe that a huge range of computer capabilities rides on top of binary addition, and on the other hand, we observe that the two's complement method obviates the need for special hardware for adding and subtracting signed numbers. Taking these two observations together, we are compelled to conclude that the two's complement method is one of the most remarkable and unsung heroes of applied computer science.

2.5 Specification

We now turn to specifying a hierarchy of chips, starting with simple adders and culminating with an Arithmetic Logic Unit (ALU). As usual in this book, we focus first on the abstract (*what* the chips are designed to), delaying implementation details (*how* they do it) to the next section. We cannot resist reiterating, with pleasure, that thanks to the two's complement method we don't have to say anything special about handling signed numbers. All the arithmetic chips that we'll present work equally well on nonnegative, negative, and mixed-sign numbers.

2.5.1 Adders

We'll focus on the following hierarchy of *adders*:

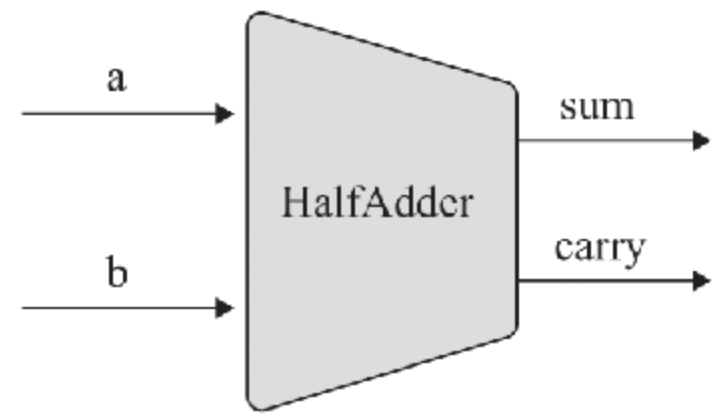
- *Half-adder*: designed to add two bits

- *Full-adder*: designed to add three bits
- *Adder*: designed to add two *n*-bit numbers

We'll also specify a special-purpose adder, called an *incrementer*, designed to add 1 to a given number. (The names *half-adder* and *full-adder* derive from the implementation detail that a full-adder chip can be realized from two half-adders, as we'll see later in the chapter.)

Half-adder: The first step on our road to adding binary numbers is adding two bits. Let us treat the result of this operation as a 2-bit number, and call its right and left bits sum and carry, respectively. Figure 2.2 presents a chip that carries out this addition operation.

inputs		*outputs*	
a	b	carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

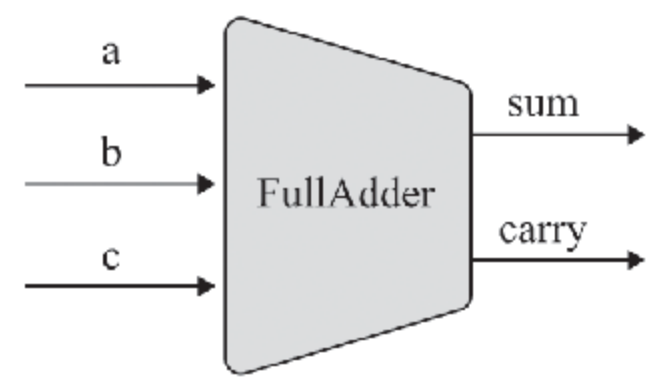



```
Chip name: HalfAdder
Input:     a, b
Output:    sum, carry
Function:  sum   = LSB of a + b
           carry = MSB of a + b
```

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

Full-adder: Figure 2.3 presents a *full-adder* chip, designed to add three bits. Like the half-adder, the full-adder chip outputs two bits that, taken together, represent the addition of the three input bits.

a	b	c	carry	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

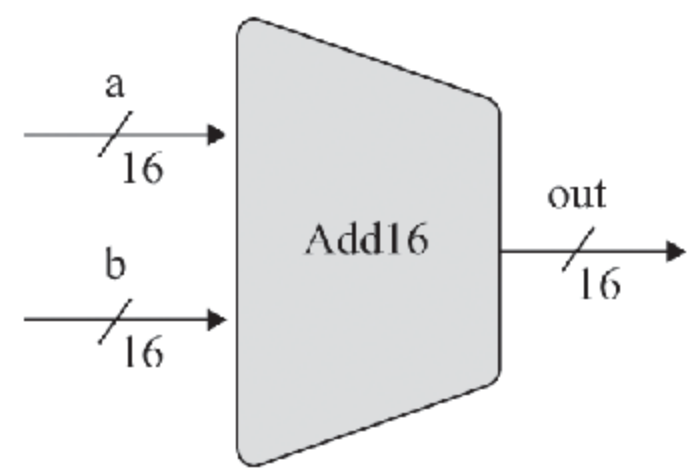



```
Chip name: FullAdder
Input:     a, b, c
Output:    sum, carry
Function:  sum   = LSB of a + b + c
           carry = MSB of a + b + c
```

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

Adder: Computers represent integer numbers using a fixed word size like 8, 16, 32, or 64 bits. The chip whose job is to add two such *n*-bit numbers is called *adder*. Figure 2.4 presents a 16-bit adder.

```
      0  1  0
      1  0  1  1   a
    + 0  0  1  0   b
...   ------------
...   1  1  0  1   out
```




```
Chip name: Add16
Input:     a[16], b[16]
Output:    out[16]
Function:  Adds two 16-bit numbers.
           The overflow bit is ignored.
```

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

We note in passing that the logic design for adding 16-bit numbers can be easily extended to implement any *n*-bit adder chip, irrespective of *n*.

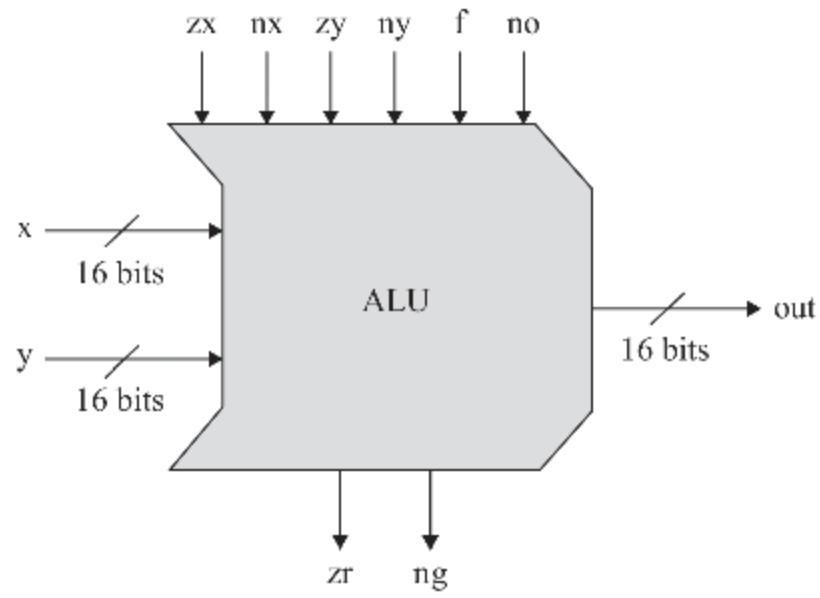
Incrementer: When we later design our computer architecture, we will need a chip that adds 1 to a given number (*Spoiler*: This will enable fetching the next instruction from memory, after executing the current one). Although the $x+1$ operation can be realized by our general-propose Adder chip, a dedicated *incrementer* chip can do it more efficiently. Here is the chip interface:

```
Chip name: Inc16
Input:     in[16]
Output:    out[16]
Function:  out = in + 1
Comment:   The overflow bit is ignored.
```

2.5.2 The Arithmetic Logic Unit

All the adder chips presented so far are generic: any computer that performs arithmetic operations uses such chips, one way or another. Building on these chips, we now turn to describe an *Arithmetic Logic Unit*, a chip that will later become the computational centerpiece of our CPU. Unlike the generic gates and chips discussed thus far, the ALU design is unique to the computer built in Nand to Tetris, named Hack. That said, the design principles underlying the Hack ALU are general and instructive. Further, our ALU architecture achieves a great deal of functionality using a minimal set of internal parts. In that respect, it provides a good example of an efficient and elegant logic design.

As its name implies, an Arithmetic Logic Unit is a chip designed to compute a set of arithmetic and logic operations. Exactly *which* operations an ALU should feature is a design decision derived from cost-effectiveness considerations. In the case of the Hack platform, we decided that (i) the ALU will perform only integer arithmetic (and not, for example, floating point arithmetic) and (ii) the ALU will compute the set of eighteen arithmetic-logical functions shown in figure 2.5a.




```
out(x, y, zx,nx,zy,ny,f,no) =
    0, 1, -1,
    x, y, !x, !y, -x, -y,
    x+1, y+1, x-1, y-1,
    x+y, x-y, y-x,
    x&y, x|y
```

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.

As seen in figure 2.5a, the Hack ALU operates on two 16-bit two's complement integers, denoted x and y, and on six 1-bit inputs, called *control bits*. These control bits "tell" the ALU which function to compute. The exact specification is given in figure 2.5b.

pre-setting the x input		pre-setting the y input		computing + or &	post-setting the output	resulting ALU output
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x&y	if no then out=!out	out(x,y) =
zx	**nx**	**zy**	**ny**	**f**	**no**	**out**
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

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.

To illustrate the ALU logic, suppose we wish to compute the function $x-1$, for $x=27$. To get started, we feed the 16-bit binary code of 27 into the x input. In this particular example we don't care about y's value, since it has no impact on the required calculation. Now, looking up x-1 in figure 2.5b, we set the ALU's control bits to 001110. According to the specification, this setting should cause the ALU to output the binary code representing 26.

Is that so? To find out, let's delve deeper, and reckon how the Hack ALU performs its magic. Focusing on the top row of figure 2.5b, note that each one of the six control bits is associated with a standalone, conditional

micro-action. For example, the zx bit is associated with "if (zx==1) then set x to 0". These six directives are to be performed in order: first, we either set the x and y inputs to 0, or not; next, we either negate the resulting values, or not; next, we compute either + or & on the preprocessed values; and finally, we either negate the resulting value, or not. All these settings, negations, additions, and conjunctions are 16-bit operations.

With this logic in mind, let us revisit the row associated with x-1 and verify that the micro-operations coded by the six control bits will indeed cause the ALU to compute $x-1$. Going left to right, we see that the zx and nx bits are 0, so we neither zero nor negate the x input—we leave it as is. The zy and ny bits are 1, so we first zero the y input and then negate the result, yielding the 16-bit value 1111111111111111. Since this binary code happens to represent -1 in two's complement, we see that the two data inputs of the ALU are now x's value and -1. Since the f bit is 1, the selected operation is *addition*, causing the ALU to compute $x+(-1)$. Finally, since the no bit is 0, the output is not negated. To conclude, we've illustrated that if we feed the ALU with x and y values and set the six control bits to 001110, the ALU will compute $x-1$, as specified.

What about the other seventeen functions listed in figure 2.5b? Does the ALU actually compute them as well? To verify that this is indeed the case, you are invited to focus on other rows in the table, go through the same process of carrying out the micro-actions coded by the six control bits, and figure out for yourself what the ALU will output. Or, you can believe us that the ALU works as advertised.

Note that the ALU actually computes a total of sixty-four functions, since six control bits code that many possibilities. We've decided to focus on, and document, only eighteen of these possibilities, since these will suffice for supporting the instruction set of our target computer system. The curious reader may be intrigued to know that some of the undocumented ALU operations are quite meaningful. However, we've opted not to exploit them in the Hack system.

The Hack ALU interface is given in figure 2.5c. Note that in addition to computing the specified function on its two inputs, the ALU also computes the two output bits zr and ng. These bits, which flag whether the ALU output is zero or negative, respectively, will be used by the future CPU of our computer system.

```
Chip name: ALU
Input:     x[16], y[16], // Two 16-bit data inputs
           zx,           // Zero the x input
           nx,           // Negate the x input
           zy,           // Zero the y input
           ny,           // Negate the y input
           f,            // if f==1 out=add(x,y) else out=and(x,y)
           no            // Negate the out output
Output:    out[16],      // 16-bit output
           zr,           // if out==0 zr=1 else zr=0
           ng            // if out<0  ng=1 else ng=0
Function:
           if zx x=0      // 16-bit zero constant
           if nx x=!x     // Bit-wise negation
           if zy y=0      // 16-bit zero constant
           if ny y=!y     // Bit-wise negation
           if f out=x+y   // Integer two's complement addition
           else out=x&y   // Bit-wise And
           if no out=!out // Bit-wise negation
           if out==0 zr=1 else zr=0  // 16-bit equality comparison
           if out<0 ng=1  else ng=0  // two's complement comparison
Comment:   The overflow bit is ignored.
```

Figure 2.5c The Hack ALU API.

It may be instructive to describe the thought process that led to the design of our ALU. First, we made a tentative list of the primitive operations that we wanted our computer to perform (right column of figure 2.5b). Next, we used backward reasoning to figure out how x, y, and out can be manipulated in binary fashion in order to carry out the desired operations. These processing requirements, along with our objective to keep the ALU logic as simple as possible, have led to the design decision to use six control bits, each associated with a straightforward operation that can be easily implemented with basic logic gates. The resulting ALU is simple and elegant. And in the hardware business, simplicity and elegance carry the day.

2.6 Implementation

Our implementation guidelines are intentionally minimal. We already gave many implementation tips along the way, and now it is your turn to discover the missing parts in the chip architectures.

Throughout this section, when we say "build/implement a logic design that …," we expect you to (i) figure out the logic design (e.g., by sketching a gate diagram), (ii) write the HDL code that realizes the design, and (iii) test and debug your design using the supplied test scripts and hardware simulator. More details are given in the next section, which describes project 2.

Half-adder: An inspection of the truth table in figure 2.2 reveals that the outputs sum(a,b) and carry(a,b) happen to be identical to those of two simple Boolean functions discussed and implemented in project 1. Therefore, the half-adder implementation is straightforward.

Full-adder: A full-adder chip can be implemented from two half-adders and one additional gate (and that's why these adders are called *half* and *full)*. Other implementations are possible, including direct approaches that don't use half-adders.

Adder: The addition of two *n*-bit numbers can be done bitwise, from right to left. In step 0, the least significant pair of bits is added, and the resulting carry bit is fed into the addition of the next significant pair of bits. The process continues until the pair of the most significant bits is added. Note that each step involves the addition of three bits, one of which is propagated from the "previous" addition.

Readers may wonder how we can add pairs of bits "in parallel" before the carry bit has been computed by the previous pair of bits. The answer is that these computations are sufficiently fast as to complete and stabilize within one clock cycle. We'll discuss clock cycles and synchronization in the next chapter; for now, you can ignore the time element completely, and write HDL code that computes the addition operation by acting on all the bit-pairs simultaneously.

Incrementer: An *n*-bit incrementer can be implemented easily in a number of different ways.

ALU: Our ALU was carefully planned to effect all the desired ALU operations *logically*, using the simple Boolean operations implied by the six

control bits. Therefore, the *physical* implementation of the ALU can be reduced to implementing these simple operations, following the pseudocode specifications listed at the top of figure 2.5b. Your first step will likely be creating a logic design for zeroing and negating a 16-bit value. This logic can be used for handling the x and y inputs as well as the out output. Chips for bitwise And-ing and addition have already been built in projects 1 and 2, respectively. Thus, what remains is building logic that selects between these two operations, according to the f control bit (this selection logic was also implemented in project 1). Once this main ALU functionality works properly, you can proceed to implement the required functionality of the single-bit zr and ng outputs.

2.7 Project

Objective: Implement all the chips presented in this chapter. The only building blocks that you need are some of the gates described in chapter 1 and the chips that you will gradually build in this project.

Built-in chips: As was just said, the chips that you will build in this project use, as chip-parts, some of the chips described in chapter 1. Even if you've built these lower-level chips successfully in HDL, we recommend using their built-in versions instead. As best-practice advice pertaining to all the hardware projects in Nand to Tetris, always prefer using built-in chip-parts instead of their HDL implementations. The built-in chips are guaranteed to operate to specification and are designed to speed up the operation of the hardware simulator.

There is a simple way to follow this best-practice advice: Don't add to the project folder nand2tetris/projects/02 any .hdl file from project 1. Whenever the hardware simulator will encounter in your HDL code a reference to a chip-part from project 1, for example, And16, it will check whether there is an And16.hdl file in the current folder. Failing to find it, the hardware simulator will resort by default to using the built-in version of this chip, which is exactly what we want.

The remaining guidelines for this project are identical to those of project 1. In particular, remember that good HDL programs use as few chip-parts as

possible, and there is no need to invent and implement any "helper chips"; your HDL programs should use only chips that were specified in chapters 1 and 2.

A web-based version of project 2 is available at www.nand2tetris.org.

2.8 Perspective

The construction of the multi-bit adder presented in this chapter was standard, although no attention was paid to efficiency. Indeed, our suggested adder implementation is inefficient, due to the delays incurred while the carry bits propagate throughout the *n*-bit addends. This computation can be accelerated by logic circuits that effect so-called *carry lookahead* heuristics. Since addition is the most prevalent operation in computer architectures, any such low-level improvement can result in dramatic performance gains throughout the system. Yet in this book we focus mostly on functionality, leaving chip optimization to more specialized hardware books and courses.[2]

The overall functionality of any hardware/software system is delivered jointly by the CPU and the operating system that runs on top of the hardware platform. Thus, when designing a new computer system, the question of how to allocate the desired functionality between the ALU and the OS is essentially a cost/performance dilemma. As a rule, direct hardware implementations of arithmetic and logical operations are more efficient than software implementations but make the hardware platform more expensive.

The trade-off that we have chosen in Nand to Tetris is to design a basic ALU with minimal functionality, and use system software to implement additional mathematical operations as needed. For example, our ALU features neither multiplication nor division. In part II of the book, when we discuss the operating system (chapter 12), we'll implement elegant and efficient bitwise algorithms for multiplication and division, along with other mathematical operations. These OS routines can then be used by compilers of high-level languages operating on top of the Hack platform. Thus, when a high-level programmer writes an expression like, say, `x * 12 + sqrt(y)`, then, following compilation, some parts of the expression will be evaluated directly by the ALU and some by the OS, yet the high-level programmer will be completely oblivious to this low-level division of work. Indeed, one of the key roles of an operating system is closing gaps between the high-level language abstractions that programmers use and the barebone hardware on which they are realized.

---

1. Which correspond, respectively, to the typical high-level data types *byte*, *short*, *int*, and *long*. For example, when reduced to machine-level instructions, *short* variables can be handled by 16-bit registers. Since 16-bit arithmetic is four times faster than 64-bit arithmetic, programmers are advised to always use the most compact data type that satisfies the application's requirements.
2. A technical reason for not using carry look-ahead techniques in our adder chips is that their hardware implementation requires cyclical pin connections, which are not supported by the Nand to Tetris hardware simulator.

3 Memory

It's a poor sort of memory that only works backward.
—Lewis Carroll (1832–1898)

Consider the high-level operation `x=y+17`. In chapter 2 we showed how logic gates can be utilized for representing numbers and for computing simple arithmetic expressions like `y+17`. We now turn to discuss how logic gates can be used to *store values over time*—in particular, how a variable like `x` can be set to "contain" a value and persist it until we set it to another value. To do so, we'll develop a new family of *memory chips*.

So far, all the chips that we built in chapters 1 and 2, culminating with the ALU, were time independent. Such chips are sometimes called *combinational*: they respond to different combinations of their inputs without delay, except for the time it takes their inner chip-parts to complete the computation. In this chapter we introduce and build *sequential* chips. Unlike combinational chips, which are oblivious to time, the outputs of sequential chips depend not only on the inputs in the current time but also on inputs and outputs that have been processed previously.

Needless to say, the notions of *current* and *previous* go hand in hand with the notion of *time*: you remember now what was committed to memory before. Thus, before we start talking about memory, we must first figure out how to use logic to model the progression of time. This can be done using a *clock* that generates an ongoing train of binary signals that we call *tick* and *tock*. The time between the beginning of a tick and the end of the subsequent tock is called a *cycle*, and these cycles will be used to regulate the operations of all the memory chips used by the computer.

Following a brief, user-oriented introduction to memory devices, we will present the art of sequential logic, which we will use for building time-dependent chips. We will then set out to build registers, RAM devices, and counters. These memory devices, along with the arithmetic devices built in chapter 2, comprise all the chips needed for building a complete, general-purpose computer system—a challenge that we will take up in chapter 5.

3.1 Memory Devices

Computer programs use variables, arrays, and objects—abstractions that persist data over time. Hardware platforms support this ability by offering memory devices that know how to *maintain state*. Because evolution gave humans a phenomenal electro-chemical memory system, we tend to take for granted the ability to remember things over time. However, this ability is hard to implement in classical logic, which is aware of neither time nor state. Thus, to get started, we must first find a way to model the progression of time and endow logic gates with the ability to maintain state and respond to time changes.

We will approach this challenge by introducing a clock and an elementary, time-dependent logic gate that can flip and flop between two stable states: representing 0 and representing 1. This gate, called *data flip-flop* (DFF), is the fundamental building block from which all memory devices will be built. In spite of its central role, though, the DFF is a low-profile, inconspicuous gate: unlike registers, RAM devices, and counters, which play prominent roles in computer architectures, DFFs are used implicitly, as low-level chip-parts embedded deep within other memory devices.

The fundamental role of the DFF is seen clearly in figure 3.1, where it serves as the foundation of the memory hierarchy that we are about to build. We will show how DFFs can be used to create 1-bit registers and how *n* such registers can be lashed together to create an *n*-bit register. Next, we'll construct a RAM device containing an arbitrary number of such registers. Among other things, we'll develop a means for *addressing*, that is, accessing by address, any randomly chosen register from the RAM directly and instantaneously.

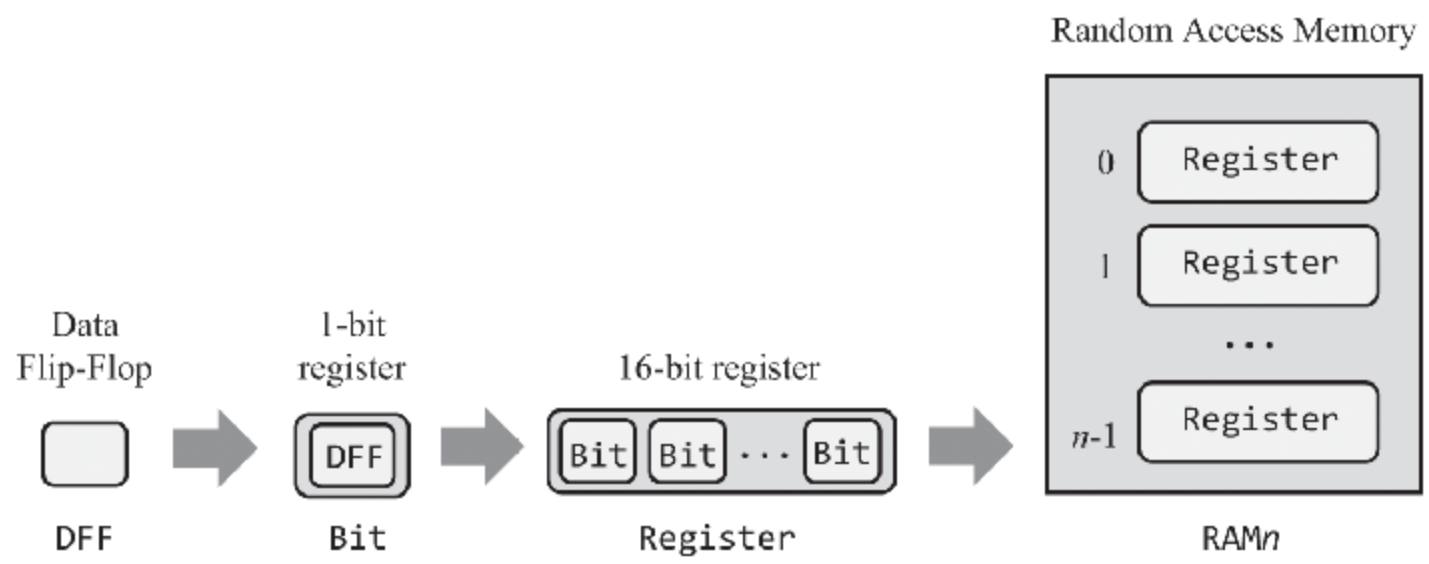


Figure 3.1 The memory hierarchy built in this chapter.

Before setting out to build these chips, though, we'll present a methodology and tools that enable modeling the progression of time and maintaining state over time.

3.2 Sequential Logic

All the chips discussed in chapters 1 and 2 were based on classical logic, which is time independent. In order to develop memory devices, we need to extend our gate logic with the ability to respond not only to input changes but also to the ticking of a clock: we remember the meaning of the word *dog* in time t since we remembered it in time $t-1$, all the way back to the point of time when we first committed it to memory. In order to develop this temporal ability to *maintain state*, we must extend our computer architecture with a time dimension and build tools that handle time using Boolean functions.

3.2.1 Time Matters

So far in our Nand to Tetris journey, we have assumed that chips respond to their inputs instantaneously: you input 7, 2, and "subtract" into the ALU, and … *poof!* the ALU output becomes 5. In reality, outputs are always delayed, due to at least two reasons. First, the inputs of the chips don't appear out of thin air; rather, the signals that represent them travel from the outputs of other chips, and this travel takes time. Second, the computations that chips perform also take time; the more chip-parts the chip has—the

more elaborate its logic—the more time it will take for the chip's outputs to emerge fully formed from the chip's circuitry.

Thus *time* is an issue that must be dealt with. As seen at the top of figure 3.2, time is typically viewed as a metaphorical arrow that progresses relentlessly forward. This progression is taken to be continuous: between every two time-points there is another time-point, and changes in the world can be infinitesimally small. This notion of time, which is popular among philosophers and physicists, is too deep and mysterious for computer scientists. Thus, instead of viewing time as a continuous progression, we prefer to break it into fixed-length intervals, called *cycles*. This representation is discrete, resulting in cycle 1, cycle 2, cycle 3, and so on. Unlike the continuous arrow of time, which has an infinite granularity, the cycles are atomic and indivisible: changes in the world occur only during cycle transitions; within cycles, the world stands still.

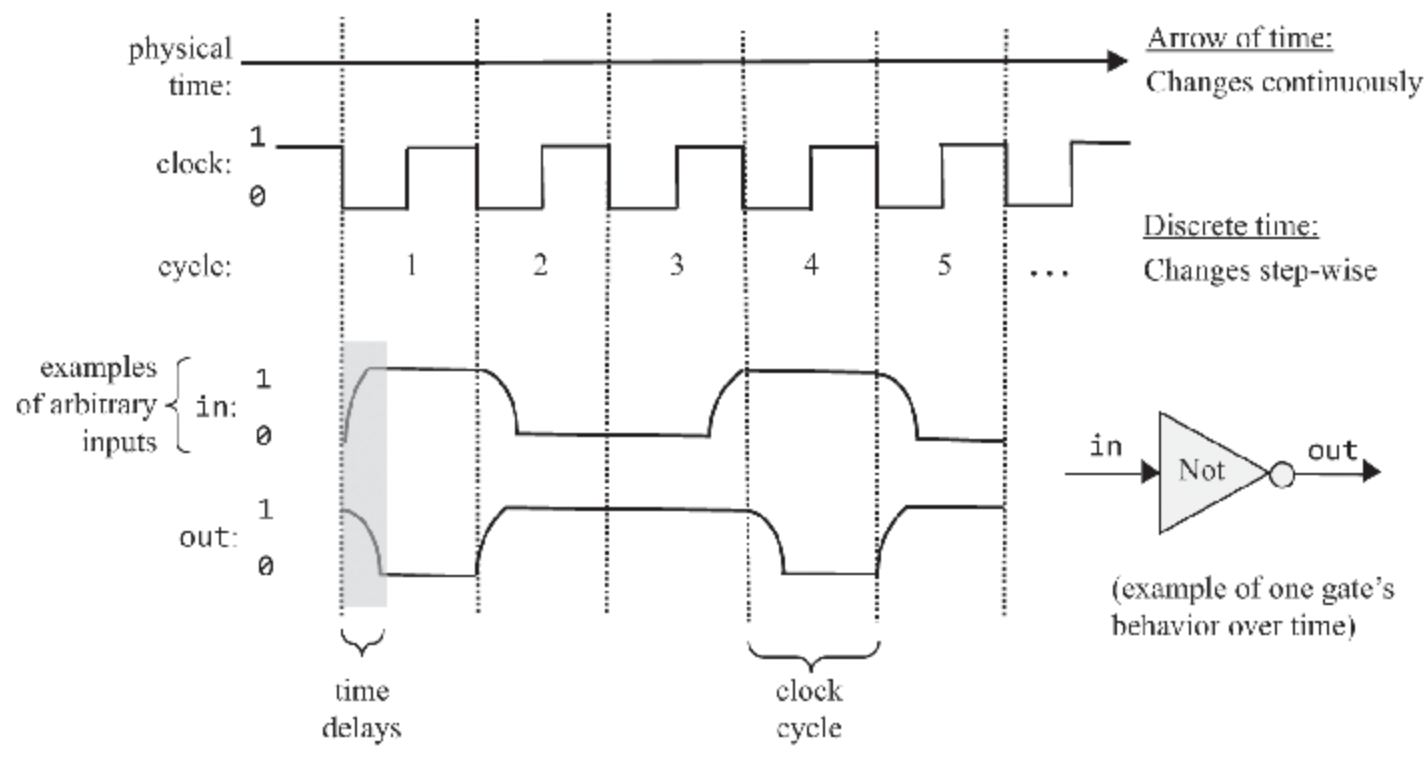


Figure 3.2 Discrete time representation: State changes (input and output values) are observed only during cycle transitions. Within cycles, changes are ignored.

Of course the world never stands still. However, by treating time discretely, we make a conscious decision to ignore continuous change. We are content to know the state of the world in cycle n, and then in cycle $n+1$, but *during* each cycle the state is assumed to be—well, we don't care. When it comes to building computer architectures, this discrete view of time serves two important design objectives. First, it can be used for neutralizing the randomness associated with communications and

computation time delays. Second, it can be used for synchronizing the operations of different chips across the system, as we'll see later.

To illustrate, let's focus on the bottom part of figure 3.2, which tracks how a Not gate (used as an example) responds to arbitrarily chosen inputs. When we feed the gate with 1, it takes a while before the gate's output stabilizes on 0. However, since the cycle duration is—*by design*—longer than the time delay, when we reach the cycle's end, the gate output has already stabilized on 0. Since we probe the state of the world only at cycle ends, we don't get to see the interim time delays; rather, it appears as if we fed the gate with 0, and *poof!* the gate responded with 1. If we make the same observations at the end of each cycle, we can generalize that when a Not gate is fed with some binary input *x*, it instantaneously outputs Not (*x*).

Thoughtful readers have probably noticed that for this scheme to work, the *cycle's length* must be longer than the maximal time delays that can occur in the system. Indeed, cycle length is one of the most important design parameters of any hardware platform: When planning a computer, the hardware engineer chooses a cycle length that meets two design objectives. On the one hand, the cycle should be sufficiently long to contain, and neutralize, any possible time delay; on the other hand, the shorter the cycle, the faster the computer: if things happen only during cycle transitions, then obviously things happen faster when the cycles are shorter. To sum up, the cycle length is chosen to be slightly longer than the maximal time delay in any chip in the system. Following the tremendous progress in switching technologies, we are now able to create cycles as tiny as a billionth of a second, achieving remarkable computer speed.

Typically, the cycles are realized by an oscillator that alternates continuously between two phases labeled 0−1, *low-high*, or *ticktock* (as seen in figure 3.2). The elapsed time between the beginning of a tick and the end of the subsequent tock is called a *cycle*, and each cycle is taken to model one discrete time unit. The current clock phase (*tick* or *tock*) is represented by a binary signal. Using the hardware's circuitry, the same master clock signal is simultaneously broadcast to every memory chip in the system. In every such chip, the clock input is funneled to the lower-level DFF gates, where it serves to ensure that the chip will commit to a new state, and output it, only at the end of the clock cycle.

3.2.2 Flip-Flops

Memory chips are designed to "remember", or store, information over time. The low-level devices that facilitate this storage abstraction are named *flip-flop* gates, of which there are several variants. In Nand to Tetris we use a variant named *data flip-flop*, or DFF, whose interface includes a single-bit data input and a single-bit data output (see top of figure 3.3). In addition, the DFF has a clock input that feeds from the master clock's signal. Taken together, the data input and the clock input enable the DFF to implement the simple time-based behavior $\text{out}(t) = \text{in}(t-1)$, where in and out are the gate's input and output values, and t is the current time unit (from now on, we'll use the terms "time unit" and "cycle" interchangeably). Let us not worry how this behavior is actually implemented. For now, we simply observe that at the end of each time unit, the DFF outputs the input value from the previous time unit.

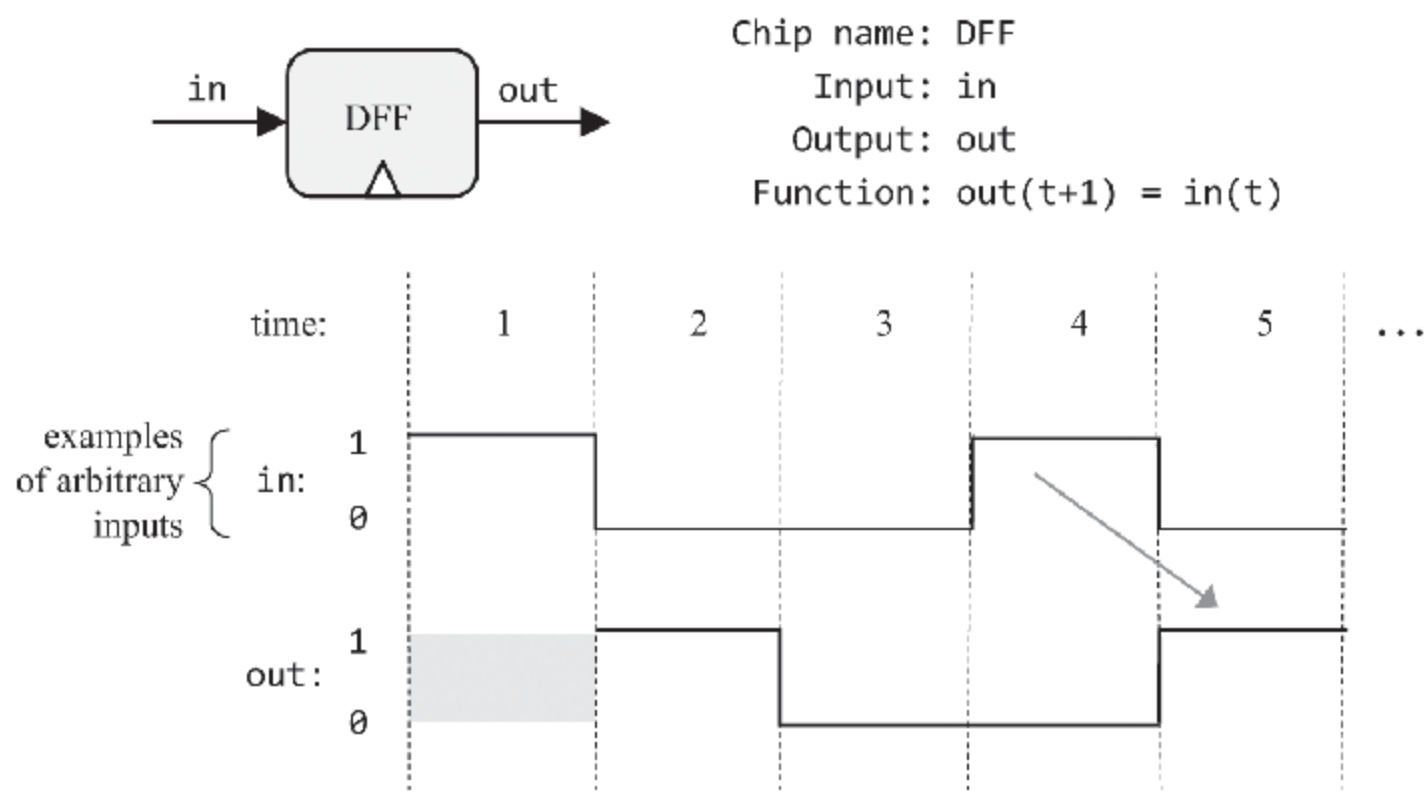


Figure 3.3 The data flip-flop (top) and behavioral example (bottom). In the first cycle the previous input is unknown, so the DFF's output is undefined. In every subsequent time unit, the DFF outputs the input from the previous time unit. Following gate diagramming conventions, the clock input is marked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

Like Nand gates, DFF gates lie deep in the hardware hierarchy. As shown in figure 3.1, all the memory chips in the computer—registers, RAM units, and counters—are based, at bottom, on DFF gates. All these DFFs are connected to the same master clock, forming a huge distributed "chorus

line." At the end of each clock cycle, the outputs of *all* the DFFs in the computer commit to their inputs from the previous cycle. At all other times, the DFFs are *latched*, meaning that changes in their inputs have no immediate effect on their outputs. This conduction operation effects any one of the system's numerous DFF gates many times per second (depending on the computer's clock frequency).

Hardware implementations realize the time dependency using a dedicated clock bus that feeds the master clock signal simultaneously to all the DFF gates in the system. Hardware simulators emulate the same effect in software. In particular, the Nand to Tetris hardware simulator features a clock icon, enabling the user to advance the clock interactively, as well as `tick` and `tock` commands that can be used programmatically, in test scripts.

3.2.3 Combinational and Sequential Logic

All the chips that were developed in chapters 1 and 2, starting with the elementary logic gates and culminating with the ALU, were designed to respond only to changes that occur during the current clock cycle. Such chips are called *time-independent* chips, or *combinational* chips. The latter name alludes to the fact that these chips respond only to different combinations of their input values, while paying no attention to the progression of time.

In contrast, chips that are designed to respond to changes that occurred during previous time units (and possibly during the current time unit as well) are called *sequential*, or *clocked*. The most fundamental sequential gate is the DFF, and any chip that includes it, either directly or indirectly, is also said to be sequential. Figure 3.4 depicts a typical sequential logic configuration. The main element in this configuration is a set of one or more chips that include DFF chip-parts, either directly or indirectly. As shown in the figure, these sequential chips may also interact with combinational chips. The feedback loop enables the sequential chip to respond to inputs and outputs from the previous time unit. In combinational chips, where time is neither modeled nor recognized, the introduction of feedback loops is problematic: the output of the chip would depend on its input, which itself would depend on the output, and thus the output would depend on itself. Note, however, that there is no difficulty in feeding

outputs back into inputs, as long as the feedback loop goes through a DFF gate: the DFF introduces an inherent time delay so that the output at time t does not depend on itself but rather on the output at time $t-1$.

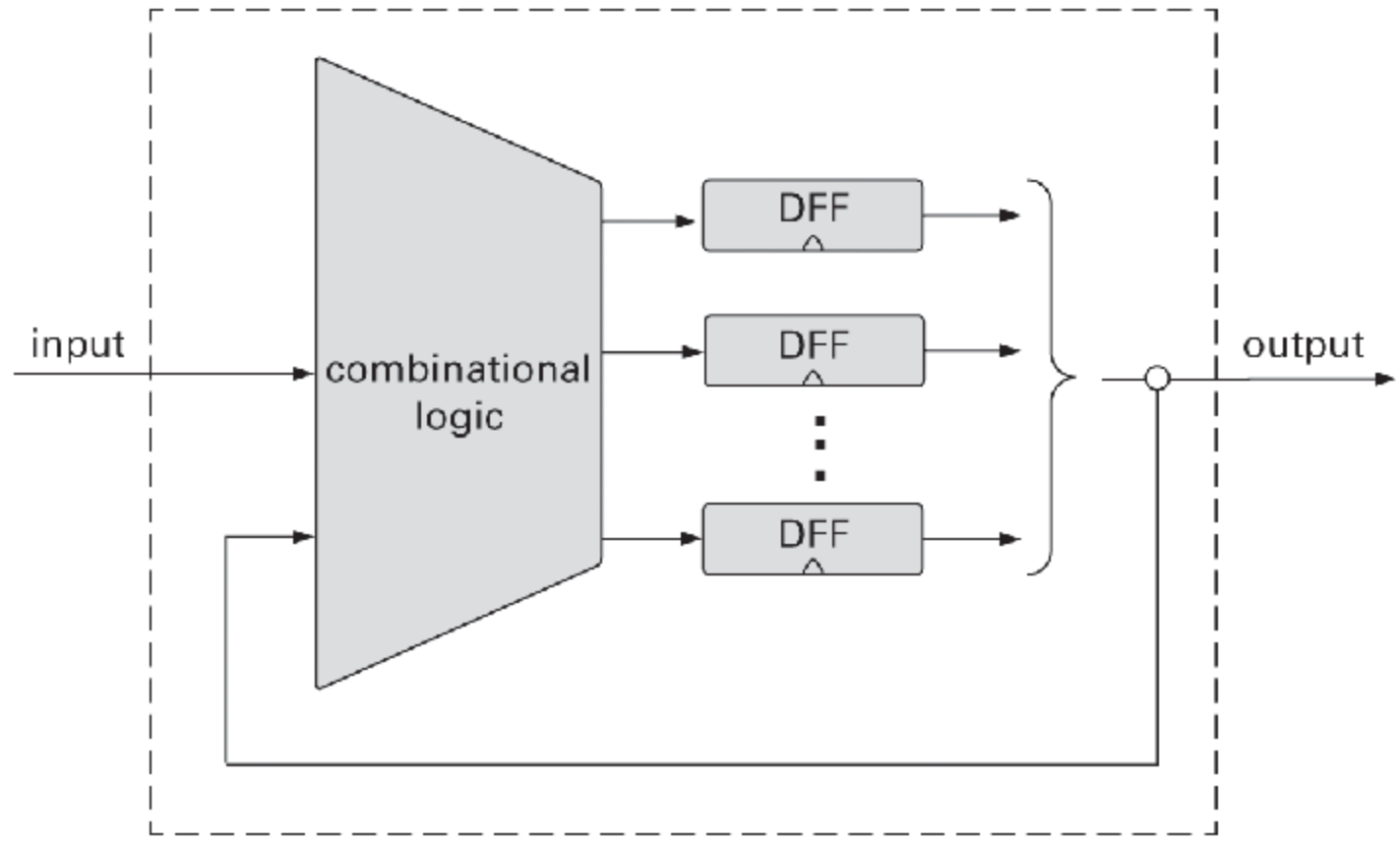


Figure 3.4 Sequential logic design typically involves DFF gates that feed from, and connect to, combinational chips. This gives sequential chips the ability to respond to current as well as to previous inputs and outputs.

The time dependency of sequential chips has an important side effect that serves to synchronize the overall computer architecture. To illustrate, suppose we instruct the ALU to compute $x+y$, where x is the output of a nearby register, and y is the output of a remote RAM register. Because of physical constraints like distance, resistance, and interference, the electric signals representing x and y will likely arrive at the ALU at different times. However, being a combinational chip, the ALU is insensitive to the concept of time—it continuously and happily adds up whichever data values happen to lodge at its inputs. Thus, it will take some time before the ALU's output stabilizes to the correct $x+y$ result. Until then, the ALU will generate garbage.

How can we overcome this difficulty? Well, if we use a discrete representation of time, *we simply don't care*. All we have to do is ensure, when we build the computer's clock, that the duration of the clock cycle will be slightly longer than the time it takes a bit to travel the longest distance from one chip to another, plus the time it takes to complete the

most time-consuming within-chip calculation. This way, we are guaranteed that by the end of the clock cycle, the ALU's output will be valid. This, in a nutshell, is the trick that turns a set of standalone hardware components into a well-synchronized system. We will have more to say about this master orchestration when we build the computer architecture in chapter 5.

3.3 Specification

We now turn to specify the memory chips that are typically used in computer architectures:

- data flip-flops (DFFs)
- registers (based on DFFs)
- RAM devices (based on registers)
- counters (based on registers)

As usual, we describe these chips *abstractly*. In particular, we focus on each chip's interface: inputs, outputs, and function. How the chips deliver this functionality will be discussed in the Implementation section.

3.3.1 Data Flip-Flop

The most elementary sequential device that we will use—the basic component from which all other memory chips will be constructed—is the *data flip-flop*. A DFF gate has a single-bit data input, a single-bit data output, a clock input, and a simple time-dependent behavior: $\mathtt{out}(t) = \mathtt{in}(t-1)$.

Usage: If we put a one-bit value in the DFF's input, the DFF's state will be set to this value, and the DFF's output will emit it in the next time unit (see figure 3.3). This humble operation will prove most useful in the implementation of registers, which is described next.

3.3.2 Registers

We present a single-bit register, named Bit, and a 16-bit register, named Register. The Bit chip is designed to store a single bit of information—0 or 1

—over time. The chip interface consists of an in input that carries a data bit, a load input that enables the register for writes, and an out output that emits the current state of the register. The Bit API and input/output behavior are described in figure 3.5.

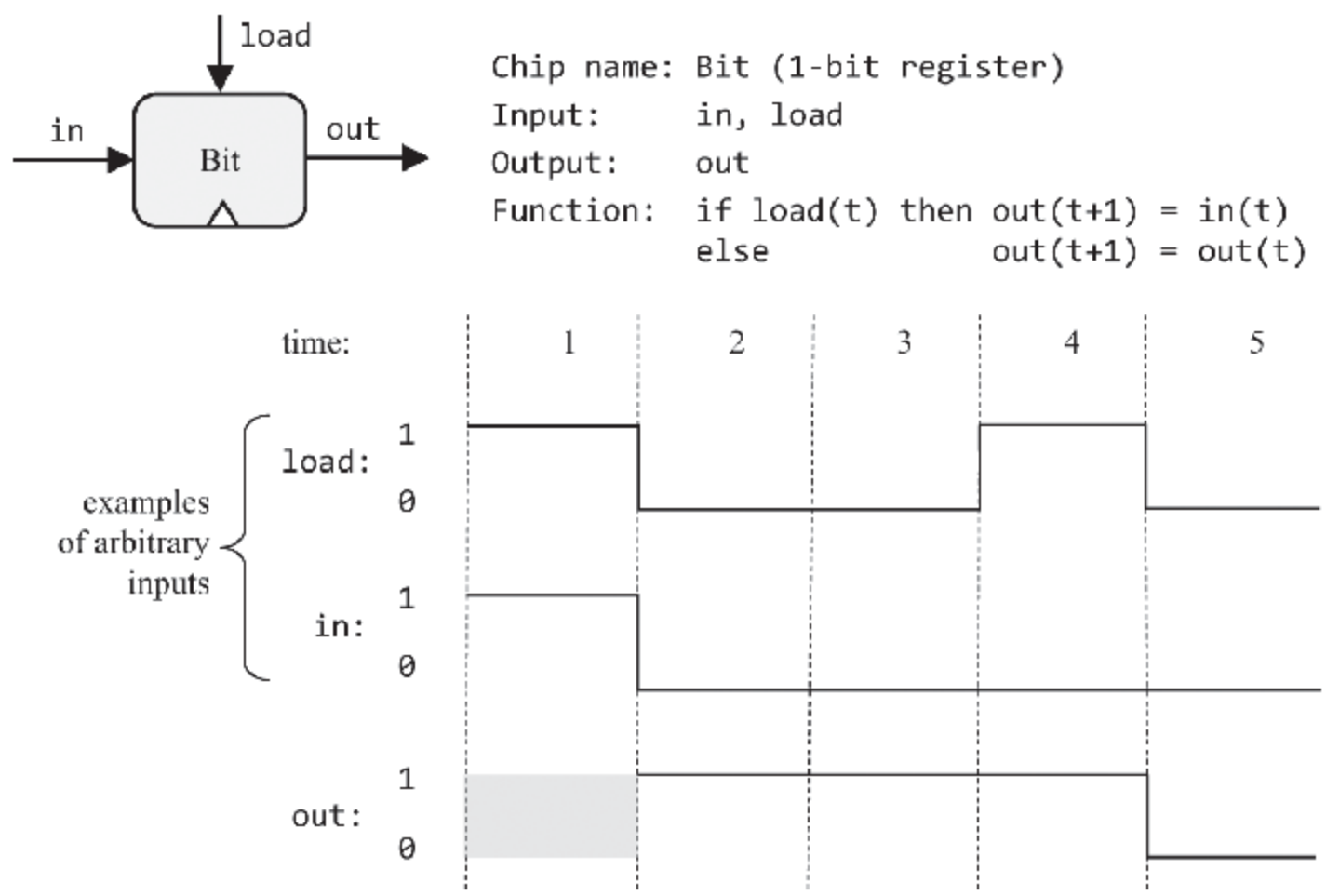


Figure 3.5 1-bit register. Stores and emits a 1-bit value until instructed to load a new value.

Figure 3.5 illustrates how the single-bit register behaves over time, responding to arbitrary examples of in and load inputs. Note that irrespective of the input value, as long as the load bit is not asserted, the register is latched, maintaining its current state.

The 16-bit Register chip behaves exactly the same as the Bit chip, except that it is designed to handle 16-bit values. Figure 3.6 gives the details.

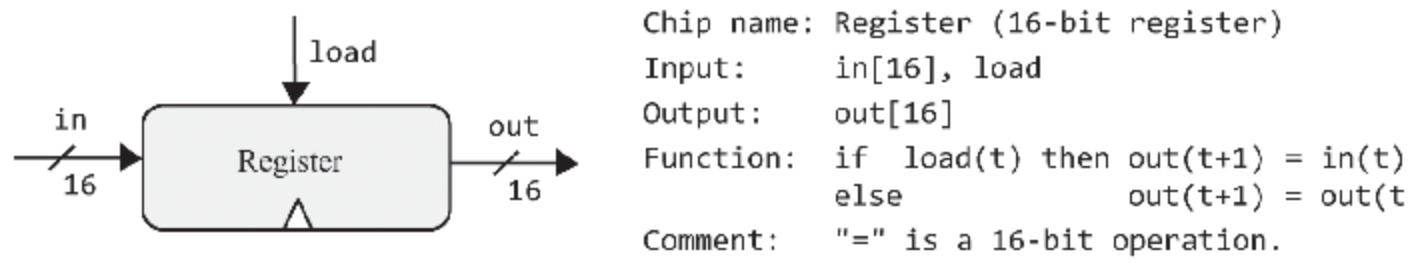
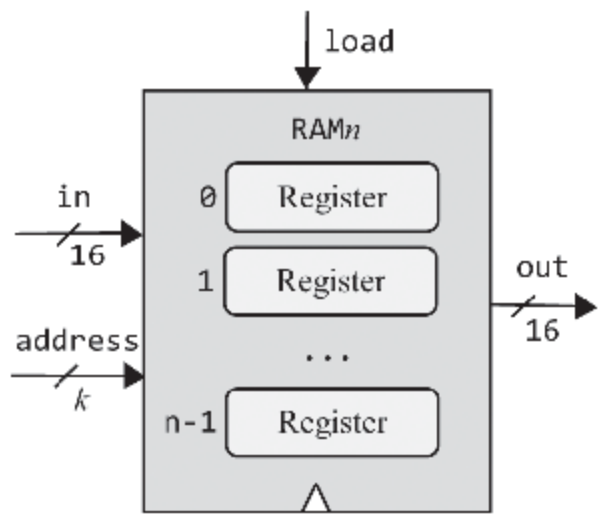


Figure 3.6 16-bit Register. Stores and emits a 16-bit value until instructed to load a new value.

Usage: The Bit register and the 16-bit Register are used identically. To read the state of the register, probe the value of out. To set the register's state to *v*, put *v* in the in input, and assert (put 1 into) the load bit. This will set the register's state to *v*, and, from the next time unit onward, the register will commit to the new value, and its out output will start emitting it. We see that the Register chip fulfills the classical function of a memory device: it remembers and emits the last value that was written into it, until we set it to another value.

3.3.3 Random Access Memory

A direct-access memory unit, also called *Random Access Memory*, or RAM, is an aggregate of n Register chips. By specifying a particular address (a number between 0 to $n-1$), each register in the RAM can be selected and made available for read/write operations. Importantly, the access time to any randomly selected memory register is instantaneous and independent of the register's address and the size of the RAM. That's what makes RAM devices so remarkably useful: even if they contain billions of registers, we can still access and manipulate each selected register directly, in the same instantaneous access time. The RAM API is given in figure 3.7.



Chip name: RAM*n*
Input: in[16], load, address[*k*] (*k* = $\log_2 n$)
Output: out[16]
Function:
Out emits the value stored at the memory location (register) specified by address. If load==1, then the memory location specified by address is set to the value of in. The loaded value will be emitted by out from the next time step onward.

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.

Usage: To read the contents of register number *m*, set the address input to *m*. This action will select register number *m*, and the RAM's output will emit its value. To write a new value *v* into register number *m*, set the address

input to *m*, set the in input to *v*, and assert the load bit (set it to 1). This action will select register number *m*, enable it for writing, and set its value to *v*. In the next time unit, the RAM's output will start emitting *v*.

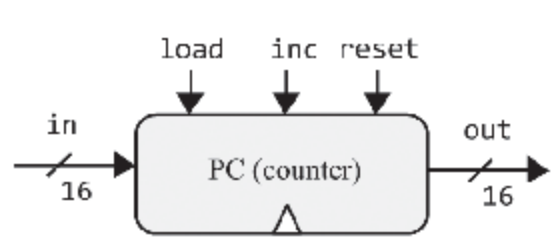
The net result is that the RAM device behaves exactly as required: a bank of addressable registers, each of which can be accessed, and operated upon, independently. In the case of a read operation (load==0), the RAM's output immediately emits the value of the selected register. In the case of a write operation (load==1), the selected memory register is set to the input value, and the RAM's output will start emitting it from the next time unit onward.

Importantly, the RAM implementation must ensure that the access time to *any* register in the RAM will be nearly instantaneous. If this were not the case, we would not be able to fetch instructions and manipulate variables in a reasonable time, making computers impractically slow. The magic of instantaneous access time will be unfolded shortly, in the Implementation section.

3.3.4 Counter

The Counter is a chip that knows how to increment its value by 1 each time unit. When we build our computer architecture in chapter 5, we will call this chip Program Counter, or PC, so that's the name that we will use here also.

The interface of our PC chip is identical to that of a register, except that it also has control bits labeled inc and reset. When inc==1, the counter increments its state in every clock cycle, effecting the operation PC++. If we want to reset the counter to 0, we assert the reset bit; if we want to set the counter to the value *v*, we put *v* in the in input and assert the load bit, as we normally do with registers. The details are given in figure 3.8.




```
Chip name: PC
Input:     in[16], load, inc, reset
Output:    out[16]
Function:  16-bit counter
if reset(t)      out(t+1) = 0
else if load(t)  out(t+1) = in(t)
else if inc(t)   out(t+1) = out(t) + 1 (integer addition)
else             out(t+1) = out(t)
```

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

Usage: To read the current contents of the PC, probe the out pin. To reset the PC, assert the reset bit and set the other control bits to 0. To have the PC increment by 1 in each time unit until further notice, assert the inc bit and set the other control bits to 0. To set the PC to the value *v*, set the in input to *v*, assert the load bit, and set the other control bits to 0.

3.4 Implementation

The previous section presented a family of memory chip abstractions, focusing on their interface and functionality. This section focuses on how these chips can be realized, using simpler chips that were already built. As usual, our implementation guidelines are intentionally suggestive; we want to give you enough hints to complete the implementation yourself, using HDL and the supplied hardware simulator.

3.4.1 Data Flip-Flop

A DFF gate is designed to be able to "flip-flop" between two stable states, representing 0 and representing 1. This functionality can be implemented in several different ways, including ones that use Nand gates only. The Nand-based DFF implementations are elegant, yet intricate and impossible to model in our hardware simulator since they require feedback loops among combinational gates. Wishing to abstract away this complexity, we will treat the DFF as a primitive building block. In particular, the Nand to Tetris hardware simulator provides a built-in DFF implementation that can be readily used by other chips, as we now turn to describe.

3.4.2 Registers

Register chips are memory devices: they are expected to implement the basic behavior $\texttt{out}(t+1) = \texttt{out}(t)$, remembering and emitting their state over time. This looks similar to the DFF behavior, which is $\texttt{out}(t+1) = \texttt{in}(t)$. If we could only feed the DFF output back into its input, this could be a good starting point for implementing the one-bit Bit register. This solution is shown on the left of figure 3.9.

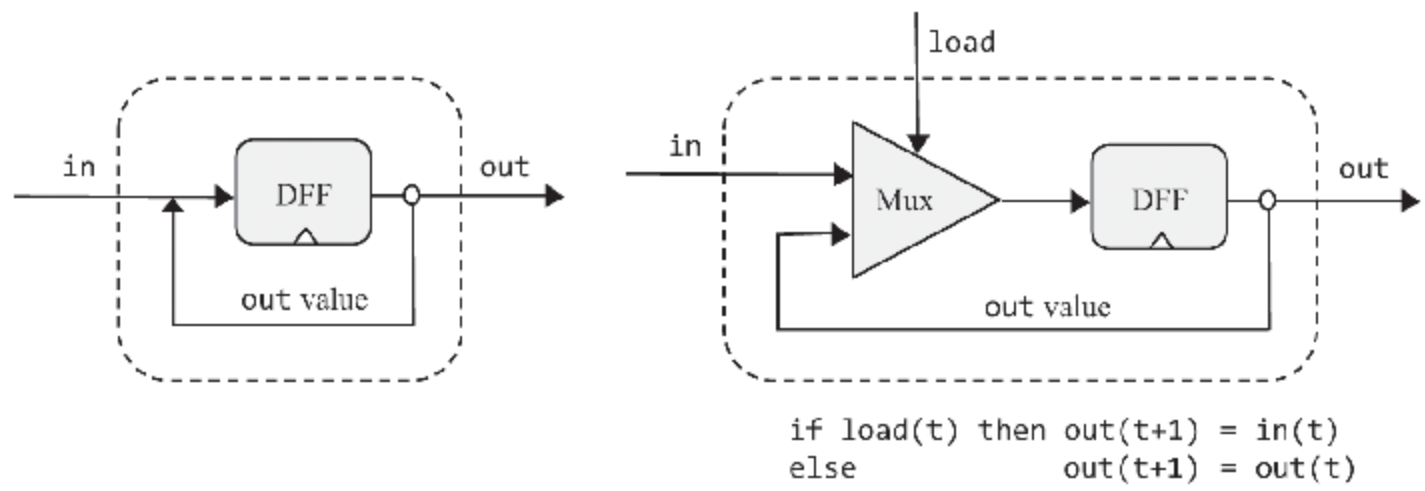


Figure 3.9 The Bit (1-bit register) implementation: invalid (left) and correct (right) solutions.

It turns out that the implementation shown on the left of figure 3.9 is invalid, for several related reasons. First, the implementation does not expose a load bit, as required by the register's interface. Second, there is no way to tell the DFF chip-part when to draw its input from the in wire and when from the incoming out value. Indeed, HDL programming rules forbid feeding a pin from more than one source.

The good thing about this invalid design is that it leads us to the correct implementation, shown on the right of figure 3.9. As the chip diagram shows, a natural way to resolve the input ambiguity is introducing a multiplexer into the design. The *load bit* of the overall register chip can then be funneled to the *select bit* of the inner multiplexer: If we set this bit to 1, the multiplexer will feed the in value into the DFF; if we set the load bit to 0, the multiplexer will feed the DFF's previous output. This will yield the behavior "if load, set the register to a new value, else set it to the previously stored value"—exactly how we want a register to behave.

Note that the feedback loop just described does not entail cyclical *data race* problems: the loop goes through a DFF gate, which introduces a time delay. In fact, the Bit design shown in figure 3.9 is a special case of the general sequential logic design shown in figure 3.4.

Once we've completed the implementation of the single-bit Bit register, we can move on to constructing a *w*-bit register. This can be achieved by forming an array of *w* Bit chips (see figure 3.1). The basic design parameter of such a register is *w*—the number of bits that it is supposed to hold—for example, 16, 32, or 64. Since the Hack computer will be based on a 16-bit hardware platform, our Register chip will be based on sixteen Bit chip-parts.

The Bit register is the only chip in the Hack architecture that uses a DFF gate directly; all the higher-level memory devices in the computer use DFF chips indirectly, by virtue of using Register chips made of Bit chips. Note that the inclusion of a DFF gate in the design of any chip—directly or indirectly—turns the latter chip, as well as all the higher-level chips that use it as a chip-part, into time-dependent chips.

3.4.3 RAM

The Hack hardware platform requires a RAM device of 16K (16384) 16-bit registers, so that's what we have to implement. We propose the following gradual implementation roadmap:

chip	n	k	built from:
RAM8	8	3	8 Register chips
RAM64	64	6	8 RAM8 chips
RAM512	512	9	8 RAM64 chips
RAM4K	4096	12	8 RAM512 chips
RAM16K	16384	14	4 RAM4K chips

All these memory chips have precisely the same RAMn API given in figure 3.7. Each RAM chip has n registers, and the width of its address input is $k = log_2 n$ bits. We now describe how these chips can be implemented, starting with RAM8.

A RAM8 chip features 8 registers, as shown in figure 3.7, for $n = 8$. Each register can be selected by setting the RAM8's 3-bit address input to a value between 0 and 7. The act of *reading* the value of a selected register can be described as follows: Given some address (a value between 0 and 7), how can we "select" register number address and pipe its output to the RAM8's output? *Hint:* We can do it using one of the combinational chips built in project 1. That's why reading the value of a selected RAM register is achieved nearly instantaneously, independent of the clock and of the number of registers in the RAM. In a similar way, the act of *writing* a value into a selected register can be described as follows: Given an address value, a load value (1), and a 16-bit in value, how can we set the value of register number address to in? *Hint:* The 16-bit in data can be fed simultaneously to

the in inputs of all eight Register chips. Using another combinational chip developed in project 1, along with the address and load inputs, you can ensure that only one of the registers will accept the incoming in value, while all the other seven registers will ignore it.

We note in passing that the RAM registers are not marked with addresses in any physical sense. Rather, the logic described above is capable of, and sufficient for, selecting individual registers according to their address, and this is done by virtue of using combinational chips. Now here is a crucially important observation: since combinational logic is time independent, the access time to any individual register will be nearly instantaneous.

Once we've implemented the RAM8 chip, we can move on to implementing a RAM64 chip. The implementation can be based on eight RAM8 chip-parts. To select a particular register from the RAM64 memory, we use a 6-bit address, say *xxxyyy*. The *xxx* bits can be used to select one of the RAM8 chips, and the *yyy* bits can be used to select one of the registers within the selected RAM8. This hierarchical addressing scheme can be effected by gate logic. The same implementation idea can guide the implementation of the remaining RAM512, RAM4K, and RAM16K chips.

To recap, we take an aggregate of an unlimited number of registers, and impose on it a combinational superstructure that permits direct access to any individual register. We hope that the beauty of this solution does not escape the reader's attention.

3.4.4 Counter

A counter is a memory device that can increment its value in every time unit. In addition, the counter can be set to 0 or some other value. The basic storage and counting functionalities of the counter can be implemented, respectively, by a Register chip and by the incrementer chip built in project 2. The logic that selects between the counter's inc, load, and reset modes can be implemented using some of the multiplexers built in project 1.

3.5 Project

Objective: Build all the chips described in the chapter. The building blocks that you can use are primitive DFF gates, chips that you will build on top of them, and the gates built in previous chapters.

Resources: The only tool that you need for this project is the Nand to Tetris hardware simulator. All the chips should be implemented in the HDL language specified in appendix 2. As usual, for each chip we supply a skeletal .hdl program with a missing implementation part, a .tst script file that tells the hardware simulator how to test it, and a .cmp compare file that defines the expected results. 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.

Tip: The data flip-flop (DFF) gate is considered primitive; thus there is no need to build it. When the simulator encounters a DFF chip-part in an HDL program, it automatically invokes the tools/builtIn/DFF.hdl implementation.

Folders structure of this project: When constructing RAM chips from lower-level RAM chip-parts, we recommend using built-in versions of the latter. Otherwise, the simulator will recursively generate numerous memory-resident software objects, one for each of the many chip-parts that make up a typical RAM unit. This may cause the simulator to run slowly or, worse, run out of the memory of the host computer on which the simulator is running.

To avert this problem, we've partitioned the RAM chips built in this project into two subfolders. The RAM8.hdl and RAM64.hdl programs are stored in projects/03/a, and the other, higher-level RAM chips are stored in projects/03/b. This partitioning is done for one purpose only: when evaluating the RAM chips stored in the b folder, the simulator will be forced to use built-in implementations of the RAM64 chip-parts, because RAM64.hdl cannot be found in the b folder.

Steps: We recommend proceeding in the following order:

1. The hardware simulator needed for this project is available in nand2tetris/tools.
2. Consult appendix 2 and the Hardware Simulator Tutorial, as needed.
3. Build and simulate all the chips specified in the projects/03 folder.

A web-based version of project 3 is available at www.nand2tetris.org.

3.6 Perspective

The cornerstone of all the memory systems described in this chapter is the flip-flop, which we treated abstractly as a primitive, built-in gate. The usual approach is to construct flip-flops from elementary combinational gates (e.g., Nand gates) connected in feedback loops. The standard construction begins by building a non-clocked flip-flop which is bi-stable, that is, can be set to be in one of two states (storing 0, and storing 1). Then a clocked flip-flop is obtained by cascading two such non-clocked flip-flops, the first being set when the clock *ticks* and the second when the clock *tocks*. This master-slave design endows the overall flip-flop with the desired clocked synchronization functionality.

Such flip-flop implementations are both elegant and intricate. In this book we have chosen to abstract away these low-level considerations by treating the flip-flop as a primitive gate. Readers who wish to explore the internal structure of flip-flop gates can find detailed descriptions in most logic design and computer architecture textbooks.

One reason not to dwell on flip-flop esoterica is that the lowest level of the memory devices used in modern computers is not necessarily constructed from flip-flop gates. Instead, modern memory chips are carefully optimized, exploiting the unique physical properties of the underlying storage technology. Many such alternative technologies are available today to computer designers; as usual, which technology to use is a cost-performance issue. Likewise, the recursive ascent method that we used to build the RAM chips is elegant but not necessarily efficient. More efficient implementations are possible.

Aside from these physical considerations, all the chip constructions described in this chapter—the registers, the counter, and the RAM chips—are standard, and versions of them can be found in every computer system.

In chapter 5, we will use the register chips built in this chapter, along with the ALU built in chapter 2, to build a Central Processing Unit. The CPU will then be augmented with a RAM device, leading up to a general-purpose computer architecture capable of executing programs written in a machine language. This machine language is discussed in the next chapter.

4 Machine Language

Works of imagination should be written in very plain language; the more purely imaginative they are, the more necessary it is to be plain.

—Samuel Taylor Coleridge (1772–1834)

In chapters 1–3 we built processing and memory chips that can be integrated into the hardware platform of a general-purpose computer. Before we set out to complete this construction, let's pause and ask: What exactly is the *purpose* of this computer? As the architect Louis Sullivan famously observed, "Form follows function." If you wish to understand a system, or build one, start by studying the function that the system is supposed to serve. With that in mind, before we set out to complete the construction of our hardware platform, we'll devote this chapter to studying the *machine language* that this platform is designed to realize. After all, executing programs written in machine language efficiently is the ultimate function of any general-purpose computer.

A machine language is an agreed-upon formalism designed to code machine instructions. Using these instructions, we can instruct the computer's processor to perform arithmetic and logical operations, read and write values from and to the computer's memory, test Boolean conditions, and decide which instruction to fetch and execute next. Unlike high-level languages, whose design goals are cross-platform compatibility and power of expression, machine languages are designed to effect direct execution in, and total control of, a specific hardware platform. Of course, generality, elegance, and power of expression are still desired, but only to the extent that they support the basic requirement of direct and efficient execution in hardware.

Machine language is the most profound interface in the computer enterprise—the fine line where hardware meets software. This is the point where the abstract designs of humans, as manifested in high-level programs, are finally reduced to physical operations performed in silicon. Thus, a machine language can be construed as both a programming artifact and an integral part of the hardware platform. In fact, just as we say that the machine language is designed to control a particular hardware platform, we can say that the hardware platform is designed to execute instructions written in a particular machine language.

The chapter begins with a general introduction to low-level programming in machine language. Next, we give a detailed specification of the *Hack machine language*, covering both its binary and symbolic versions. The project that ends the chapter focuses on writing some machine language programs. This provides a hands-on appreciation of low-level programming and sets the stage for completing the construction of the computer hardware in the next chapter.

Although programmers rarely write programs directly in machine language, the study of low-level programming is a prerequisite to a complete and rigorous understanding of how computers work. Further, an intimate understanding of low-level programming helps the programmer write better and more efficient high-level programs. Finally, it is rather fascinating to observe, hands-on, how the most sophisticated software systems are, at bottom, streams of simple instructions, each specifying a primitive bitwise operation on the underlying hardware.

4.1 Machine Language: Overview

This chapter focuses not on the machine but rather on the *language* used to control the machine. Therefore, we will abstract away the hardware platform, focusing on the minimal subset of hardware elements that are mentioned explicitly in machine language instructions.

4.1.1 Hardware Elements

A machine language can be viewed as an agreed-upon formalism designed to manipulate a *memory* using a *processor* and a set of *registers*.

Memory: The term *memory* refers loosely to the collection of hardware devices that store data and instructions in a computer. Functionally speaking, a memory is a continuous sequence of cells, also referred to as *locations* or *memory registers*, each having a unique *address*. An individual memory register is accessed by supplying its address.

Processor: The processor, normally called the *Central Processing Unit*, or *CPU*, is a device capable of performing a fixed set of primitive operations. These include arithmetic and logical operations, memory access operations, and control (also called *branching*) operations. The processor draws its inputs from selected registers and memory locations and writes its outputs to selected registers and memory locations. It consists of an ALU, a set of registers, and gate logic that enables it to parse and execute binary instructions.

Registers: The processor and the memory are implemented as two separate, standalone chips, and moving data from one to the other is a relatively slow affair. For this reason, processors are normally equipped with several on-board registers, each capable of holding a single value. Located inside the processor's chip, the registers serve as a high-speed local memory, allowing the processor to manipulate data and instructions without having to venture outside the chip.

The CPU-resident registers fall into two categories: *data registers*, designed to hold data values, and *address registers*, designed to hold values that can be interpreted either as data values or as memory addresses. The computer architecture is configured in such a way that placing a particular value, say *n*, in an address register, causes the memory location whose address is *n* to become *selected* instantaneously.[1] This sets the stage for a subsequent operation on the selected memory location.

4.1.2 Languages

Machine language programs can be written in two alternative, but equivalent, ways: *binary* and *symbolic*. For example, consider the abstract operation "set R1 to the value of R1+R2". As language designers, we can decide to represent the addition operation using the 6-bit code 101011, and registers R1 and R2 using the codes 00001 and 00010, respectively. Assembling these codes left to right, we can decide to use the 16-bit instruction 1010110001000001 as the binary version of "set R1 to the value of R1+R2".

In the early days of computer systems, computers were programmed manually: When proto-programmers wanted to issue the instruction "set R1 to the value of R1+R2", they pushed up and down mechanical switches that stored a binary code like 1010110001000001 in the computer's instruction memory. And if the program was a hundred instructions long, they had to go through this ordeal a hundred times. Of course debugging such programs was a perfect nightmare. This led programmers to invent and use symbolic codes as a convenient way for documenting and debugging programs *on paper*, before entering them into the computer. For example, the symbolic format add R2,R1 could be chosen for representing the semantics "set R1 to the value of R1+R2" and the binary instruction 1010110001000001.

It didn't take long before several people hit on the same idea: Symbols like R, 1, 2, and + can also be represented using agreed-upon binary codes. Why not use symbolic instructions for writing programs, and then use another program—a *translator*—for translating the symbolic instructions into executable binary code? This innovation liberated programmers from the tedium of writing binary code, paving the way for the subsequent onslaught of high-level programming languages. For reasons that will become clear in chapter 6, symbolic machine languages are called *assembly languages*, and the programs that translate them into binary code are called *assemblers*.

Unlike the syntax of high-level languages, which is portable and hardware independent, the syntax of an assembly language is tightly related to the low-level details of the target hardware: the available ALU operations, number and type of registers, memory size, and so on. Since different computers vary greatly in terms of any one of these parameters, there is a Tower of Babel of machine languages, each with its obscure syntax, each designed to control a particular family of CPUs. Irrespective of

this variety, though, all machine languages are theoretically equivalent, and all of them support similar sets of generic tasks, as we now turn to describe.

4.1.3 Instructions

In what follows, we assume that the computer's processor is equipped with a set of registers denoted R0, R1, R2, … The exact number and type of these registers are irrelevant to our present discussion.

Arithmetic and logical operations: Every machine language features instructions for performing basic arithmetic operations like addition and subtraction, as well as basic logical operations like And, Or, Not. For example, consider the following code segments:

```
// Adds up two numbers:
load R1,17      // R1 ← 17
load R2,4       // R2 ← 4
add  R1,R1,R2   // R1 ← R1 + R2
```

```
// Computes a logical operation:
load R1,true    // R1 ← binary representation of true
load R2,false   // R2 ← binary representation of false
and  R1,R1,R2   // R1 ← R1 And R2 (bit-wise And)
```

For such symbolic instructions to execute on a computer, they must first be translated into binary code. The translation is done by a program named *assembler*, which we'll develop in chapter 6. For now, we assume that we have access to such an assembler and that we can use it as needed.

Memory access: Every machine language features means for accessing, and then manipulating, selected memory locations. This is typically done using an *address register*, let's call it A. For example, suppose we wish to set memory location 17 to the value 1. We can decide to do so using the two instructions load A,17 followed by load M,1, where, by convention, M stands for the memory register selected by A (namely, the memory register whose address is the current value of A). With that in mind, suppose we wish to set the fifty memory locations 200, 201, 202, …, 249 to 1. This can be done by

executing the instruction load A,200 and then entering a loop that executes the instructions load M,1 and add A,A,1 fifty times.

Flow control: While computer programs execute by default sequentially, one instruction after another, they also include occasional *jumps* to locations other than the next instruction. To facilitate such branching actions, machine languages feature several variants of conditional and unconditional *goto* instructions, as well as label declaration statements that mark the goto destinations. Figure 4.1 illustrates a simple branching action using machine language.

Using physical addresses

```
...
// Sets R1 to 0+1+2, ...
12: load R1,0
13: add R1,R1,1
...
27: goto 13
...
```

Using symbolic addresses

```
...
// Sets R1 to 0+1+2, ...
   load R1,0
(LOOP)
   add R1,R1,1
   ...
   goto LOOP
   ...
```

Figure 4.1 Two versions of the same low-level code (it is assumed that the code includes some loop termination logic, not shown here).

Symbols: Both code versions in figure 4.1 are written in assembly language; thus, both must be translated into binary code before they can be executed. Also, both versions perform exactly the same logic. However, the code version that uses symbolic references is much easier to write, debug, and maintain.

Further, unlike the code that uses physical addresses, the translated binary version of the code that uses symbolic references can be loaded into, and executed from, any memory segment that happens to be available in the computer's memory. Therefore, low-level code that mentions no physical addresses is said to be *relocatable*. Clearly, relocatable code is essential in computer systems like PCs and cell phones, which routinely load and execute multiple apps dynamically and simultaneously. Thus, we see that

symbolic references are not just a matter of cosmetics—they are used to liberate the code from unnecessary physical attachments to the host memory.

This ends our brief introduction to some machine language essentials. The next section gives a formal description of one specific machine language—the native code of the Hack computer.

4.2 The Hack Machine Language

Programmers who write low-level code (or programmers who write compilers and interpreters that generate low-level code) interact with the computer abstractly, through its *interface*, which is the computer's machine language. Although programmers don't have to be aware of all the details of the underlying computer architecture, they should be familiar with the hardware elements that come to play in their low-level programs.

With that in mind, we begin the discussion of the Hack machine language with a conceptual description of the Hack computer. Next, we give an example of a complete program written in the Hack assembly language. This sets the stage for the remainder of this section, in which we give a formal specification of the Hack language instructions.

4.2.1 Background

The design of the Hack computer, which will be presented in the next chapter, follows a widely used hardware paradigm known as the *von Neumann architecture*, named after the computing pioneer John von Neumann. Hack is a 16-bit computer, meaning that the CPU and the memory units are designed to process, move, and store, chunks of 16-bit values.

Memory: As seen in figure 4.2, the Hack platform uses two distinct memory units: a *data memory* and an *instruction memory*. The data memory stores the binary values that programs manipulate. The instruction memory stores the program's instructions, also represented as binary values. Both memories are 16-bit wide, and each has a 15-bit address space. Thus the

maximum addressable size of each memory unit is 2^{15} or 32K 16-bit words (the symbol *K*, abbreviated from *kilo*—the Greek word for *thousand*—is commonly used to stand for the number $2^{10} = 1024$). It is convenient to think about each memory unit as a linear sequence of addressable memory registers, with addresses ranging from 0 to 32K–1.

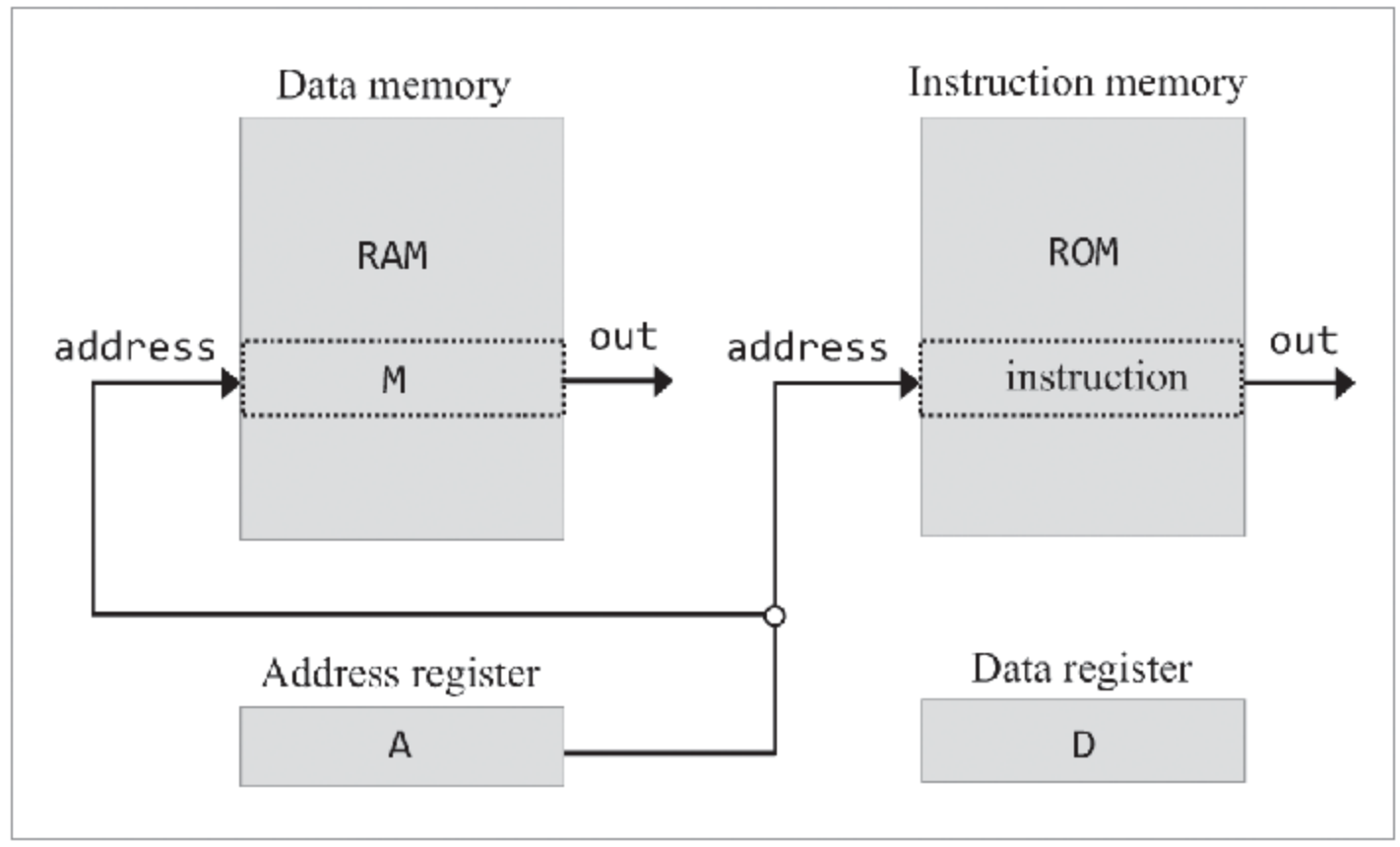


Figure 4.2 Conceptual model of the Hack memory system. Although the actual architecture is wired somewhat differently (as described in chapter 5), this model helps understand the semantics of Hack programs.

The *data memory* (which we also call RAM) is a read/write device. Hack instructions can read data from, and write data to, selected RAM registers. An individual register is selected by supplying its address. Since the memory's `address` input always contains some value, there is always one selected register, and this register is referred to in Hack instructions as `M`. For example, the Hack instruction `M=0` sets the selected RAM register to `0`.

The *instruction memory* (which we also call ROM) is a read-only device, and programs are loaded into it using some exogenous means (more about this in chapter 5). Just like with the RAM, the instruction memory's `address` input always contains some value; thus, there is always one selected instruction memory register. The value of this register is referred to as the *current instruction*.

Registers: Hack instructions are designed to manipulate three 16-bit registers: a *data register*, denoted `D`, an *address register*, denoted `A`, and a selected data memory register, denoted `M`. Hack instructions have a self-explanatory syntax. For example: `D=M`, `M=D+1`, `D=0`, `D=M-1`, and so on.

The role of the data register `D` is straightforward: it serves to store a 16-bit value. The second register, `A`, serves as both an address register and a data register. If we wish to store the value 17 in the `A` register, we use the Hack instruction `@17` (the reason for this syntax will become clear soon). In fact, this is the only way to get a constant into the Hack computer. For example, if we wish to set the `D` register to 17, we use the two instructions `@17`, followed by `D=A`. In addition to serving as a second data register, the hard-working `A` register is also used for addressing the data memory and the instruction memory, as we now turn to discuss.

Addressing: The Hack instruction `@` *xxx* sets the `A` register to the value *xxx*. In addition, the `@` *xxx* instruction has two side effects. First, it makes the RAM register whose address is *xxx* the selected memory register, denoted `M`. Second, it makes the value of the ROM register whose address is *xxx* the selected instruction. Therefore, setting `A` to some value has the simultaneous effect of preparing the stage, potentially, for one of two very different subsequent actions: manipulating the selected data memory register or doing something with the selected instruction. Which action to pursue (and which to ignore) is determined by the subsequent Hack instruction.

To illustrate, suppose we wish to set the value of `RAM[100]` to 17. This can be done using the Hack instructions `@17`, `D=A`, `@100`, `M=D`. Note that in the first pair of instructions, `A` serves as a data register; in the second pair of instructions, it serves as an address register. Here is another example: To set `RAM[100]` to the value of `RAM[200]`, we can use the Hack instructions `@200`, `D=M`, `@100`, `M=D`.

In both of these scenarios, the `A` register also selected registers in the instruction memory—an action which the two scenarios ignored. The next section discusses the opposite scenario: using `A` for selecting instructions while ignoring its effect on the data memory.

Branching: The code examples thus far imply that a Hack program is a sequence of instructions, to be executed one after the other. This indeed is

the default flow of control, but what happens if we wish to branch to executing not the next instruction but, say, instruction number 29 in the program? In the Hack language, this can be done using the Hack instruction @29, followed by the Hack instruction 0;JMP. The first instruction selects the ROM[29] register (it also selects RAM[29], but we don't care about it). The subsequent 0;JMP instruction realizes the Hack version of *unconditional branching*: go to execute the instruction addressed by the A register (we'll explain the ;0 prefix later). Since the ROM is assumed to contain the program that we are presently executing, starting at address 0, the two instructions @29 and 0;JMP end up making the value of ROM[29] the next instruction to be executed.

The Hack language also features *conditional branching*. For example, the logic if D==0 goto 52 can be implemented using the instruction @52, followed by the instruction D;JEQ. The semantics of the second instruction is "evaluate D; if the value equals zero, jump to execute the instruction stored in the address selected by A". The Hack language features several such *conditional branching* commands, as we'll explain later in the chapter.

To recap: The A register serves two simultaneous, yet very different, addressing functions. Following an @*xxx* instruction, we either focus on the selected data memory register (M) and ignore the selected instruction, or we focus on the selected instruction and ignore the selected data memory register. This duality is a bit confusing, but note that we got away with using one address register to control two separate memory devices (see figure 4.2). The result is a simpler computer architecture and a compact machine language. As usual in our business, simplicity and thrift reign supreme.

Variables: The *xxx* in the Hack instruction @*xxx* can be either a constant or a symbol. If the instruction is @23, the A register is set to the value 23. If the instruction is @x, where x is a symbol that is bound to some value, say 513, the instruction sets the A register to 513. The use of symbols endows Hack assembly programs with the ability to use *variables* rather than physical memory addresses. For example, the typical high-level assignment statement let x=17 can be implemented in the Hack language as @17, D=A, @x, M=D. The semantics of this code is "select the RAM register whose address is the value that is bound to the symbol x, and set this register to 17". Here we

assume that there is an agent who knows how to bind the symbols found in high-level languages, like `x`, to sensible and consistent addresses in the data memory. This agent is the assembler.

Thanks to the assembler, variables like `x` can be named and used in Hack programs at will, and as needed. For example, suppose we wish to write code that increments some counter. One option is to keep this counter in, say, `RAM[30]`, and increment it using the instructions `@30`, `M=M+1`. A more sensible approach is to use `@count`, `M=M+1`, and let the assembler worry about where to put this variable in memory. We don't care about the specific address so long as the assembler will always resolve the symbol to that address. In chapter 6 we'll learn how to develop an assembler that implements this useful mapping operation.

In addition to the symbols that can be introduced into Hack assembly programs as needed, the Hack language features sixteen built-in symbols named `R0`, `R1`, `R2`, …, `R15`. These symbols are always bound by the assembler to the values 0, 1, 2, …, 15. Thus, for example, the two Hack instructions `@R3`, `M=0` will end up setting `RAM[3]` to `0`. In what follows, we sometimes refer to `R0`, `R1`, `R2`, …, `R15` as *virtual registers*.

Before going on, we suggest you review, and make sure you fully understand, the code examples shown in figure 4.3 (some of which were already discussed).

Memory access examples

```
// D = 17
@17
D=A

// RAM[100] = 17
@17
D=A
@100
M=D

// RAM[100] = RAM[200]
@200
D=M
@100
M=D
```

Branching examples

```
// goto 29
@29
0;JMP

// if D>0 goto 63
@63
D;JGT
```

Variables use examples

```
// x = -1
@x
M=-1

// count = count - 1
@count
M=M-1

// sum = sum + x
@sum
D=M
@x
D=D+M
@sum
M=D
```

Figure 4.3 Hack assembly code examples.

4.2.2 Program Example

Jumping into the cold water, let's review a complete Hack assembly program, deferring a formal description of the Hack language to the next section. Before we do so, a word of caution: Most readers will probably be mystified by the obscure style of this program. To which we say: Welcome to machine language programming. Unlike high-level languages, machine languages are not designed to please programmers. Rather, they are designed to control a hardware platform, efficiently and plainly.

Suppose we wish to compute the sum $1+2+3+\cdots+n$, for a given value n. To operationalize things, we'll put the input n in RAM[0] and the output sum in RAM[1]. The program that computes this sum is listed in figure 4.4. Beginning with the pseudocode, note that instead of utilizing the well-known formula for computing the sum of an arithmetic series, we use brute-force addition. This is done for illustrating conditional and iterative processing in the Hack machine language.

Pseudocode

```
// Program: Sum1ToN
// Computes RAM[1]=1+2+3+...+RAM[0]
// Usage: put a value>=1 in RAM[0]
   i = 1
   sum = 0
LOOP:
   if (i > R0) goto STOP
   sum = sum + i
   i = i + 1
   goto LOOP
STOP:
   R1 = sum
```

Hack assembly code

```
// File: Sum1ToN.asm
// Computes RAM[1]=1+2+3+...+RAM[0]
// Usage: put a value>=1 in RAM[0]
   // i = 1
   @i
   M=1
   // sum = 0
   @sum
   M=0
(LOOP)
   // if (i > R0) goto STOP
   @i
   D=M
   @R0
   D=D-M
   @STOP
   D;JGT
   // sum = sum + i
   @sum
   D=M
   @i
   D=D+M
   @sum
   M=D
   // i = i + 1
   @i
   M=M+1
   // goto LOOP
   @LOOP
   0;JMP
(STOP)
   // R1 = sum
   @sum
   D=M
   @R1
   M=D
(END)
   @END
   0;JMP
```

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

Later in the chapter you will understand this program completely. For now, we suggest ignoring the details, and observing instead the following pattern: In the Hack language, every operation involving a memory location entails two instructions. The first instruction, @*addr*, is used to select a target memory address; the subsequent instruction specifies what to do at this address. To support this logic, the Hack language features two generic instructions, several examples of which we have already seen: an *address instruction*, also called *A*-instruction (the instructions that start with @), and a *compute instruction*, also called *C*-instruction (all the other instructions).

Each instruction has a symbolic representation, a binary representation, and an effect on the computer, as we now turn to describe.

4.2.3 The Hack Language Specification

The Hack machine language consists of two instructions, specified in figure 4.5.

A-instruction:

Symbolic: `@`*xxx* (*xxx* is a decimal value ranging from 0 to 32767, or a symbol bound to such a decimal value)

Binary: `0` *vvvvvvvvvvvvvvv* (*vv ... v* = 15-bit value of *xxx*)

C-instruction:

Symbolic: *dest* = *comp* ; *jump* (*comp* is mandatory. If *dest* is empty, the = is omitted; If *jump* is empty, the ; is omitted)

Binary: `111`*accccccdddjjj*

comp		*c*	*c*	*c*	*c*	*c*	*c*
0		1	0	1	0	1	0
1		1	1	1	1	1	1
-1		1	1	1	0	1	0
D		0	0	1	1	0	0
A	M	1	1	0	0	0	0
!D		0	0	1	1	0	1
!A	!M	1	1	0	0	0	1
-D		0	0	1	1	1	1
-A	-M	1	1	0	0	1	1
D+1		0	1	1	1	1	1
A+1	M+1	1	1	0	1	1	1
D-1		0	0	1	1	1	0
A-1	M-1	1	1	0	0	1	0
D+A	D+M	0	0	0	0	1	0
D-A	D-M	0	1	0	0	1	1
A-D	M-D	0	0	0	1	1	1
D&A	D&M	0	0	0	0	0	0
D|A	D|M	0	1	0	1	0	1

a == 0 *a* == 1

dest	*d*	*d*	*d*	Effect: store *comp* in:
null	0	0	0	the value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register (reg)
DM	0	1	1	D reg and RAM[A]
A	1	0	0	A reg
AM	1	0	1	A reg and RAM[A]
AD	1	1	0	A reg and D reg
ADM	1	1	1	A reg, D reg, and RAM[A]

jump	*j*	*j*	*j*	Effect:
null	0	0	0	no jump
JGT	0	0	1	if *comp* > 0 jump
JEQ	0	1	0	if *comp* = 0 jump
JGE	0	1	1	if *comp* ≥ 0 jump
JLT	1	0	0	if *comp* < 0 jump
JNE	1	0	1	if *comp* ≠ 0 jump
JLE	1	1	0	if *comp* ≤ 0 jump
JMP	1	1	1	unconditional jump

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

The *A*-instruction

The *A*-instruction sets the A register to some 15-bit value. The binary version consists of two fields: an operation code, also known as *op-code*, which is 0 (the leftmost bit), followed by fifteen bits that code a nonnegative binary number. For example, the symbolic instruction @5, whose binary version is 0000000000000101, stores the binary representation of 5 in the A register.

The *A*-instruction is used for three different purposes. First, it provides the only way to enter a constant into the computer under program control.

Second, it sets the stage for a subsequent *C*-instruction that manipulates a selected RAM register, referred to as `M`, by first setting `A` to the address of that register. Third, it sets the stage for a subsequent *C*-instruction that specifies a jump by first setting `A` to the address of the jump destination.

The *C*-instruction

The *C*-instruction answers three questions: what to compute (an ALU operation, denoted *comp*), where to store the computed value (*dest*), and what to do next (*jump*). Along with the *A*-instruction, the *C*-instruction specifies all the possible operations of the computer.

In the binary version, the leftmost bit is the *C*-instruction's op-code, which is `1`. The next two bits are not used, and are set by convention to `1`. The next seven bits are the binary representation of the *comp* field. The next three bits are the binary representation of the *dest* field. The rightmost three bits are the binary representation of the *jump* field. We now describe the syntax and semantics of these three fields.

Computation specification (*comp*): The Hack ALU is designed to compute one out of a fixed set of functions on two given 16-bit inputs. In the Hack computer, the two ALU data inputs are wired as follows. The first ALU input feeds from the `D` register. The second ALU input feeds either from the `A` register (when the `a`-bit is 0) or from `M`, the selected data memory register (when the `a`-bit is 1). Taken together, the computed function is specified by the `a`-bit and the six `c`-bits comprising the instruction's *comp* field. This 7-bit pattern can potentially code 128 different calculations, of which only the twenty-eight listed in figure 4.5 are documented in the language specification.

Recall that the format of the *C*-instruction is `111accccccdddjjj`. Suppose we want to compute the value of the `D` register, minus 1. According to figure 4.5, this can be done using the symbolic instruction `D-1`, which is `1110001110000000` in binary (the relevant 7-bit *comp* field is underlined for emphasis). To compute a bitwise Or between the values of the `D` and `M` registers, we use the instruction `D|M` (in binary: `1111010101000000`). To compute the constant –1, we use the instruction `-1` (in binary: `1110111010000000`), and so on.

Destination specification (*dest*): The ALU output can be stored in zero, one, two, or three possible destinations, simultaneously. The first and second d-bits code whether to store the computed value in the A register and in the D register, respectively. The third d-bit codes whether to store the computed value in M, the currently selected memory register. One, more than one, or none of these three bits may be asserted.

Recall that the format of the *C*-instruction is 111accccccdddjjj. Suppose we wish to increment the value of the memory register whose address is 7 and also to store the new value in the D register. According to figure 4.5, this can be accomplished using the two instructions:

```
0000000000000111 // @7
1111110111011000 // DM=M+1
```

Jump directive (*jump*): The *jump* field of the *C*-instruction specifies what to do next. There are two possibilities: fetch and execute the next instruction in the program, which is the default, or fetch and execute some other, designated instruction. In the latter case, we assume that the A register was already set to the address of the target instruction.

During run-time, whether or not to jump is determined jointly by the three j-bits of the instruction's *jump* field and by the ALU output. The first, second, and third j-bits specify whether to jump in case the ALU output is negative, zero, or positive, respectively. This gives eight possible jump conditions, listed at the bottom right of figure 4.5. The convention for specifying an unconditional goto instruction is 0;JMP (since the *comp* field is mandatory, the convention is to compute 0—an arbitrarily chosen ALU operation—which is ignored).

Preventing conflicting uses of the *A* register: The Hack computer uses one address register for addressing both the RAM and the ROM. Thus, when we execute the instruction @*n*, we select both RAM[*n*] and ROM[*n*]. This is done in order to set the stage for either a subsequent *C*-instruction that operates on the selected data memory register, M, or a subsequent *C*-instruction that specifies a jump. To make sure that we perform exactly one of these two operations, we issue the following best-practice advice: A *C*-

instruction that contains a reference to M should specify no jump, and vice versa: a *C*-instruction that specifies a jump should make no reference to M.

4.2.4 Symbols

Assembly instructions can specify memory locations (addresses) using either constants or symbols. The symbols fall into three functional categories: *predefined symbols*, representing special memory addresses; *label symbols*, representing destinations of goto instructions; and *variable symbols*, representing variables.

Predefined symbols: There are several kinds of predefined symbols, designed to promote consistency and readability of low-level Hack programs.

R0, **R1**, …, **R15**: These symbols are bound to the values 0 to 15. This predefined binding helps make Hack programs more readable. To illustrate, consider the following code segment:

```
// Sets RAM[3] to 7:
@7
D=A
@R3
M=D
```

The instruction @7 sets the A register to 7, and @R3 sets the A register to 3. Why do we use R in the latter and not in the former? Because it makes the code more self-explanatory. In the instruction @7, the syntax hints that A is used as a *data register*, ignoring the side effect of also selecting RAM[7]. In the instruction @R3, the syntax hints that A is used *to select a data memory address*. In general, the predefined symbols R0, R1, …, R15 can be viewed as ready-made working variables, sometimes referred to as *virtual registers*.

SP, **LCL**, **ARG**, **THIS**, **THAT**: These symbols are bound to the values 0, 1, 2, 3, and 4, respectively. For example, address 2 can be selected using either @2, @R2, or @ARG. The symbols SP, LCL, ARG, THIS, and THAT will be used in part II of the book, when we implement the compiler and the virtual

machine that run on top of the Hack platform. These symbols can be completely ignored for now; we specify them for completeness.

SCREEN, KBD: Hack programs can read data from a keyboard and display data on a screen. The screen and the keyboard interface with the computer via two designated memory blocks known as *memory maps*. The symbols SCREEN and KBD are bound, respectively, to the values 16384 and 24576 (in hexadecimal: 4000 and 6000), which are the agreed-upon base addresses of the *screen memory map* and the *keyboard memory map*, respectively. These symbols are used by Hack programs that manipulate the screen and the keyboard, as we'll see in the next section.

Label symbols: Labels can appear anywhere in a Hack assembly program and are declared using the syntax (*xxx*). This directive binds the symbol *xxx* to the address of the next instruction in the program. Goto instructions that make use of label symbols can appear anywhere in the program, even before the label has been declared. By convention, label symbols are written using uppercase letters. The program listed in figure 4.4 uses three label symbols: LOOP, STOP and END.

Variable symbols: Any symbol *xxx* appearing in a Hack assembly program that is not predefined and is not declared elsewhere using (*xxx*) is treated as a variable and is bound to a unique running number starting at 16. By convention, variable symbols are written using lowercase letters. For example, the program listed in figure 4.4 uses two variables: i and sum. These symbols are bound by the assembler to 16 and 17, respectively. Therefore, following translation, instructions like @i and @sum end up selecting memory addresses 16 and 17, respectively. The beauty of this contract is that the assembly program is completely oblivious of the physical addresses. The assembly program uses symbols only, trusting that the assembler will know how to resolve them into actual addresses.

4.2.5 Input/Output Handling

The Hack hardware platform can be connected to two peripheral I/O devices: a screen and a keyboard. Both devices interact with the computer

platform through *memory maps*.

Drawing pixels on the screen is done by writing binary values into a designated memory segment associated with the screen, and listening to the keyboard is done by reading a designated memory location associated with the keyboard. The physical I/O devices and their memory maps are synchronized via continuous refresh loops that are external to the main hardware platform.

Screen: The Hack computer interacts with a black-and-white screen organized as 256 rows of 512 pixels per row. The screen's contents are represented by a memory map, stored in an 8K memory block of 16-bit words, starting at RAM address 16384 (in hexadecimal: 4000), also referred to by the predefined symbol SCREEN. Each row in the physical screen, starting at the screen's top-left corner, is represented in the RAM by thirty-two consecutive 16-bit words. Following convention, the screen origin is the top-left corner, which is considered row 0 and column 0. With that in mind, the pixel at row *row* and column *col* is mapped onto the *col* % 16 bit (counting from LSB to MSB) of the word located at RAM[SCREEN $+ row \cdot 32 + col/16$]. This pixel can be either read (probing whether it is black or white), made black by setting it to 1, or made white by setting it to 0. For example, consider the following code segment, which blackens the first 16 pixels at the top left of the screen:

```
// Sets the A register to the address of the RAM register that represents
// the 16 left-most pixels at the screen's top row:
@SCREEN
// Sets this RAM register to 1111111111111111:
M=-1
```

Note that Hack instructions cannot access individual pixels/bits directly. Instead, we must fetch a complete 16-bit word from the memory map, figure out which bit or bits we wish to manipulate, carry out the manipulation using arithmetic/logical operations (without touching the other bits), and then write the modified 16-bit word to the memory. In the example given above, we got away with not doing bit-specific

manipulations since the task could be implemented using one bulk manipulation.

Keyboard: The Hack computer can interact with a standard physical keyboard via a single-word memory map located at RAM address 24576 (in hexadecimal: 6000), also referred to by the predefined symbol KBD. The contract is as follows: When a key is pressed on the physical keyboard, its 16-bit character code appears at RAM[KBD]. When no key is pressed, the code 0 appears. The Hack character set is listed in appendix 5.

By now, readers with programming experience have probably noticed that manipulating input/output devices using assembly language is a tedious affair. That's because they are accustomed to using high-level statements like write ("hello") or draw Circle (x,y, radius). As you can now appreciate, there is a considerable gap between these abstract, high-level I/O statements and the bit-by-bit machine instructions that end up realizing them in silicon. One of the agents that closes this gap is the *operating system*—a program that knows, among many other things, how to render text and draw graphics using pixel manipulations. We will discuss and write one such OS in part II of the book.

4.2.7 Syntax Conventions and File Formats

Binary code files: By convention, programs written in the binary Hack language are stored in text files with a hack extension, for example, Prog.hack. Each line in the file codes a single binary instruction, using a sequence of sixteen 0 and 1 characters. Taken together, all the lines in the file represent a machine language program. The contract is as follows: When a machine language program is loaded into the computer's instruction memory, the binary code appearing in the file's *n*th line is stored at address *n* of the instruction memory. The counts of program lines, instructions, and memory addresses start at 0.

Assembly language files: By convention, programs written in the symbolic Hack assembly language are stored in text files with an asm extension, for example, Prog.asm. An assembly language file is composed of text lines,

each being an *A*-instruction, a *C*-instruction, a label declaration, or a comment.

A label declaration is a text line of the form (*symbol*). The assembler handles such a declaration by binding *symbol* to the address of the next instruction in the program. This is the only action that the assembler takes when handling a label declaration; no binary code is generated. That's why label declarations are sometimes referred to as *pseudo-instructions*: they exist only at the symbolic level, generating no code.

Constants and symbols: These are the *xxx*'s in *A*-instructions of the form @*xxx*. *Constants* are nonnegative values from 0 to $2^{15}-1$, and are written in decimal notation. A *symbol* is any sequence of letters, digits, underscore (_), dot (.), dollar sign ($), and colon (:) that does not begin with a digit.

Comments: A text line beginning with two slashes (//) and ending at the end of the line is considered a comment and is ignored.

White space: Leading space characters and empty lines are ignored.

Case conventions: All the assembly mnemonics (figure 4.5) must be written in uppercase. By convention, label symbols are written in uppercase, and variable symbols in lowercase. See figure 4.4 for examples.

4.3 Hack Programming

We now turn to present three examples of low-level programming, using the Hack assembly language. Since project 4 focuses on writing Hack assembly programs, it will serve you well to carefully read and understand these examples.

Example 1: Figure 4.6 shows a program that adds up the values of the first two RAM registers, adds 17 to the sum, and stores the result in the third RAM register. Before running the program, the user (or a test script) is expected to put some values in RAM[0] and RAM[1].

```
// Program: Add.asm
// Computes: RAM[2] = RAM[0] + RAM[1] + 17
// Usage: put values in RAM[0] and in RAM[1]
   // D = RAM[0]
   @R0
   D=M
   // D = D + RAM[1]
   @R1
   D=D+M
   // D = D + 17
   @17
   D=D+A
   // RAM[2] = D
   @R2
   M=D
(END)
   @END
   0;JMP
```

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

Among other things, the program illustrates how the so-called virtual registers R0, R1, R2, … can be used as working variables. The program also illustrates the recommended way of terminating Hack programs, which is staging and entering an infinite loop. In the absence of this infinite loop, the CPU's fetch-execute logic (explained in the next chapter) will merrily glide forward, trying to execute whatever instructions are stored in the computer's memory following the last instruction in the current program. This may lead to unpredictable and potentially hazardous consequences. The deliberate infinite loop serves to control and contain the CPU's operation after completing the program's execution.

Example 2: The second example computes the sum $1+2+3+\cdots+n$, where n is the value of the first RAM register, and puts the sum in the second RAM

register. This program is shown in figure 4.4, and now we have what it takes to understand it fully.

Among other things, this program illustrates the use of symbolic variables—in this case i and sum. The example also illustrates our recommended practice for low-level program development: instead of writing assembly code directly, start by writing goto-oriented pseudocode. Next, test your pseudocode on paper, tracing the values of key variables. When convinced that the program's logic is correct, and that it does what it's supposed to do, proceed to express each pseudo-instruction as one or more assembly instructions.

The virtues of writing and debugging symbolic (rather than physical) instructions were observed by the gifted mathematician and writer Augusta Ada King-Noel, Countess of Lovelace, back in 1843. This important insight has contributed to her lasting fame as history's first programmer. Before Ada Lovelace, proto-programmers who worked with early mechanical computers were reduced to tinkering with machine operations directly, and coding was hard and error prone. What was true in 1843 about symbolic and physical programming is equally true today about pseudo and assembly programming: When it comes to nontrivial programs, writing and testing pseudocode and then translating it into assembly instructions is easier and safer than writing assembly code directly.

Example 3: Consider the high-level array processing idiom for `i = 0 ... n` {do something with arr[i]}. If we wish to express this logic in assembly, then our first challenge is that the array abstraction does not exist in machine language. However, if we know the base address of the array in the RAM, we can readily implement this logic in assembly, using pointer-based access to the array elements.

To illustrate the notion of a pointer, suppose that variable x contains the value 523, and consider the two possible pseudo-instructions `x=17` and `*x=17` (of which we execute only one). The first instruction sets the value of x to 17. The second instruction informs that x is to be treated as a *pointer*, that is, a variable whose value is interpreted as a memory address. Hence, the instruction ends up setting RAM[523] to 17, leaving the value of x intact.

The program in figure 4.7 illustrates pointer-based array processing in the Hack machine language. The key instructions of interest are `A=D+M`,

followed by `M=-1`. In the Hack language, the basic pointer-processing idiom is implemented by an instruction of the form `A=...`, followed by a *C*-instruction that operates on `M` (which stands for `RAM[A]`, the memory location selected by A). As we will see when we write the compiler in the second part of the book, this humble low-level programming idiom enables implementing, in Hack assembly, any array access or object-based get/set operation expressed in any high-level language.

Pseudocode

```
// Program: PointerDemo
// Starting at base address R0,
// sets the first R1 words to -1
   n = 0
LOOP:
   if (n == R1) goto END
   *(R0 + n) = -1
   n = n + 1
   goto LOOP
END:
```

Hack assembly code

```
// Program: PointerDemo.asm
// Starting at base address R0,
// sets the first R1 words to -1
    // n = 0
    @n
    M=0
(LOOP)
    // if (n == R1) goto END
    @n
    D=M
    @R1
    D=D-M
    @END
    D;JEQ
    // *(R0 + n) = -1
    @R0
    D=M
    @n
    A=D+M
    M=-1
    // n = n + 1
    @n
    M=M+1
    // goto LOOP
    @LOOP
    0;JMP
(END)
    @END
    0;JMP
```

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

4.4 Project

Objective: Acquire a taste of low-level programming, and get acquainted with the Hack computer system. This will be done by writing and executing two low-level programs, written in the Hack assembly language.

Resources: The only resources required to complete the project are the Hack *CPU emulator*, available in `nand2tetris/tools`, and the test scripts

described below, available in the projects/04 folder.

Contract: Write and test the two programs described below. When executed on the supplied CPU emulator, your programs should realize the described behaviors.

Multiplication (Mult.asm): The inputs of this program are the values stored in R0 and R1 (RAM[0] and RAM[1]). The program computes the product R0 * R1 and stores the result in R2. Assume that R0 ≥ 0, R1 ≥ 0, and R0 * R1 < 32768 (your program need not test these assertions). The supplied Mult.tst and Mult.cmp scripts are designed to test your program on representative data values.

I/O handling (Fill.asm): This program runs an infinite loop that listens to the keyboard. When a key is pressed (any key), the program blackens the screen by writing *black* in every pixel. When no key is pressed, the program clears the screen by writing *white* in every pixel. You may choose to blacken and clear the screen in any spatial pattern, as long as pressing a key continuously for long enough will result in a fully blackened screen, and not pressing any key for long enough will result in a cleared screen. This program has a test script (Fill.tst) but no compare file—it should be checked by visibly inspecting the simulated screen in the CPU emulator.

CPU emulator: This program, available in nand2tetris/tools, provides a visual simulation of the Hack computer (see figure 4.8). The program's GUI shows the current states of the computer's instruction memory (ROM), data memory (RAM), the two registers A and D, the program counter PC, and the ALU. It also displays the current state of the computer's screen and allows entering inputs through the keyboard.

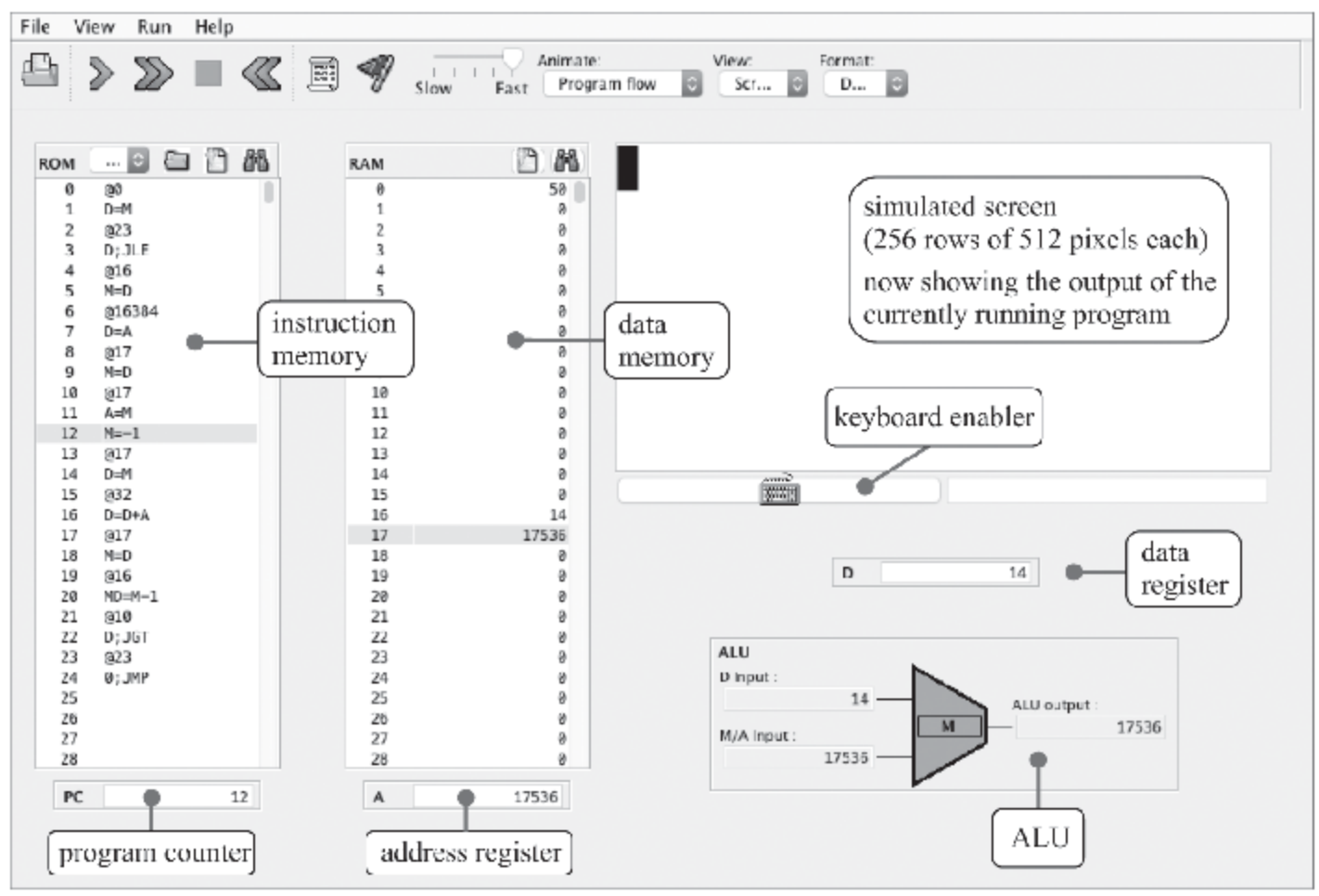


Figure 4.8 The CPU emulator, with a program loaded in the instruction memory (ROM) and some data in the data memory (RAM). The figure shows a snapshot taken during the program's execution.

The typical way to use the CPU emulator is to load a machine language program into the ROM, execute the code, and observe its impact on the simulated hardware elements. Importantly, the CPU emulator enables loading binary.hack files as well as symbolic .asm files, written in the Hack assembly language. In the latter case, the emulator translates the assembly program into binary code on the fly. Conveniently, the loaded code can be viewed in both its binary and symbolic representations.

Since the supplied CPU emulator features a built-in assembler, there is no need to use a standalone Hack assembler in this project.

Steps: We recommend proceeding as follows:

0. The supplied CPU emulator is available in the nand2tetris/tools folder. If you need help, consult the tutorial available at www.nand2tetris.org.
1. Write/edit the Mult.asm program using a plain text editor. Start with the skeletal program stored in projects/04/mult/Mult.asm.
2. Load Mult.asm into the CPU emulator. This can be done either interactively or by loading and executing the supplied Mult.tst script.
3. Run the script. If you get any translation or run-time errors, go to step 1.

Follow steps 1–3 for writing the second program, using the projects/04/fill folder.

Debugging tip: The Hack language is case-sensitive. A common assembly programming error occurs when one writes, say, @foo and @Foo in different parts of the program, thinking that both instructions refer to the same symbol. In fact, the assembler will generate and manage two variables that have nothing in common.

A web-based version of project 4 is available at www.nand2tetris.org.

4.5 Perspective

The Hack machine language is basic. Typical machine languages feature more operations, more data types, more registers, and more instruction formats. In terms of syntax, we have chosen to give Hack a lighter look and feel than that of conventional assembly languages. In particular, we have chosen a friendly syntax for the *C*-instruction, for example, `D=D+M`, instead of the more common prefix syntax `add M,D` used in many machine languages. The reader should note, however, that this is just a syntax effect. For example, the + character in the operation code `D+M` plays no algebraic role whatsoever. Rather, the three-character string `D+M`, taken as a whole, is treated as a single assembly mnemonic, designed to code a single ALU operation.

One of the main characteristics that gives machine languages their particular flavor is the number of memory addresses that can be squeezed into a single instruction. In this respect, the austere Hack language may be described as a 1/2 *address* machine language: Since there is no room to pack both an instruction code and a 15-bit address in a single 16-bit instruction, operations involving memory access require two Hack instructions: one for specifying the address on which we wish to operate, and one for specifying the operation. In comparison, many machine languages can specify an operation and at least one address in every machine instruction.

Indeed, Hack assembly code typically ends up being mostly an alternating sequence of *A*- and *C*-instructions: `@sum` followed by `M=0`, `@LOOP` followed by `0;JMP`, and so on. If you find this coding style tedious or peculiar, you should note that friendlier *macro-instructions* like `sum=0` and `goto LOOP` can be easily introduced into the language, making Hack assembly code shorter and more readable. The trick is to extend the assembler to translate each macro-instruction into the two Hack instructions that it entails—a relatively simple tweak.

The *assembler*, mentioned many times in this chapter, is the program responsible for translating symbolic assembly programs into executable programs written in binary code. In addition, the assembler is responsible for managing all the system- and user-defined symbols found in the assembly program and for resolving them into physical memory addresses that are injected into the generated binary code. We will return to this translation task in chapter 6, which is dedicated to understanding and building assemblers.

---

1. By *instantaneously* we mean within the same clock cycle, or time unit.

5 Computer Architecture

Make everything as simple as possible, but not simpler.
—Albert Einstein (1879–1955)

This chapter is the pinnacle of the hardware part of our journey. We are now ready to take the chips that we built in chapters 1–3 and integrate them into a general-purpose computer system, capable of running programs written in the machine language presented in chapter 4. The specific computer that we will build, named Hack, has two important virtues. On the one hand, Hack is a simple machine that can be constructed in a few hours, using previously built chips and the supplied hardware simulator. On the other hand, Hack is sufficiently powerful to illustrate the key operating principles and hardware elements of any general-purpose computer. Therefore, building it will give you a hands-on understanding of how modern computers work, and how they are built.

Section 5.1 begins with an overview of the *von Neumann architecture*—a central dogma in computer science underlying the design of almost all modern computers. The Hack platform is a von Neumann machine variant, and section 5.2 gives its exact hardware specification. Section 5.3 describes how the Hack platform can be implemented from previously built chips, in particular the ALU built in project 2 and the registers and memory devices built in project 3. Section 5.4 describes the project in which you will build the computer. Section 5.5 provides perspective. In particular, we compare the Hack machine to industrial-strength computers and emphasize the critical role that optimization plays in the latter.

The computer that will emerge from this effort will be as simple as possible, but not simpler. On the one hand, the computer will be based on a

minimal and elegant hardware configuration. On the other hand, the resulting configuration will be sufficiently powerful for executing programs written in a Java-like programming language, presented in part II of the book. This language will enable developing interactive computer games and applications involving graphics and animation, delivering a solid performance and a satisfying user experience. In order to realize these high-level applications on the barebone hardware platform, we will need to build a compiler, a virtual machine, and an operating system. This will be done in part II. For now, let's complete part I by integrating the chips that we've built so far into a complete, general-purpose hardware platform.

5.1 Computer Architecture Fundamentals

5.1.1 The Stored Program Concept

Compared to all the machines around us, the most remarkable feature of the digital computer is its amazing versatility. Here is a machine with a finite and fixed hardware that can perform an infinite number of tasks, from playing games to typesetting books to driving cars. This remarkable versatility—a boon that we have come to take for granted—is the fruit of a brilliant early idea called the *stored program* concept. Formulated independently by several scientists and engineers in the 1930s, the stored program concept is still considered the most profound invention in, if not the very foundation of, modern computer science.

Like many scientific breakthroughs, the basic idea is simple. The computer is based on a fixed hardware platform capable of executing a fixed repertoire of simple instructions. At the same time, these instructions can be combined like building blocks, yielding arbitrarily sophisticated programs. Moreover, the logic of these programs is not embedded in the hardware, as was customary in mechanical computers predating 1930. Instead, the program's code is temporarily stored in the computer's memory, *like data*, becoming what is known as *software*. Since the computer's operation manifests itself to the user through the currently executing software, the same hardware platform can be made to behave completely differently each time it is loaded with a different program.

5.1.2 The von Neumann Architecture

The stored program concept is the key element of both abstract and practical computer models, most notably the *Turing machine* (1936) and the *von Neumann machine* (1945). The Turing machine—an abstract artifact describing a deceptively simple computer—is used mainly in theoretical computer science for analyzing the logical foundations of computation. In contrast, the von Neumann machine is a practical model that informs the construction of almost all computer platforms today.

The von Neumann architecture, shown in figure 5.1, is based on a *Central Processing Unit* (CPU), interacting with a *memory* device, receiving data from some *input* device, and emitting data to some *output* device. At the heart of this architecture lies the *stored program* concept: the computer's memory stores not only the data that the computer manipulates but also the instructions that tell the computer what to do. Let us explore this architecture in some detail.

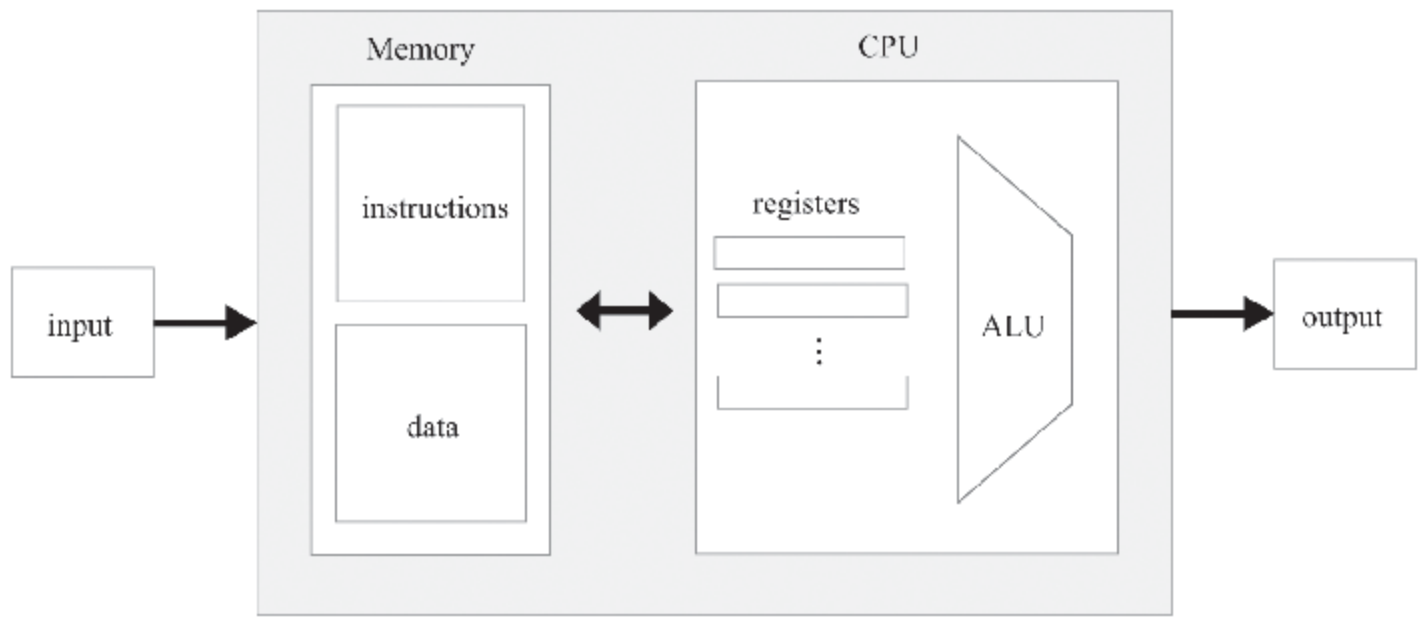


Figure 5.1 A generic von Neumann computer architecture.

5.1.3 Memory

The computer's *Memory* can be discussed from both physical and logical perspectives. Physically, the memory is a linear sequence of addressable, fixed-size *registers*, each having a unique address and a value. Logically, this address space serves two purposes: storing data and storing instructions. Both the "instruction words" and the "data words" are implemented exactly the same way—as sequences of bits.

All the memory registers—irrespective of their roles—are handled the same way: to access a particular memory register, we supply the register's address. This action, also referred to as *addressing*, provides an immediate access to the register's data. The term *Random Access Memory* derives from the important requirement that each randomly selected memory register can be reached instantaneously, that is, within the same cycle (or time step), irrespective of the memory size and the register's location. This requirement clearly carries its weight in memory units that have billions of registers. Readers who built the RAM devices in project 3 know that we've already satisfied this requirement.

In what follows, we'll refer to the memory area dedicated to data as *data memory* and to the memory area dedicated to instructions as *instruction memory*. In some variants of the von Neumann architecture, the data memory and the instruction memory are allocated and managed dynamically, as needed, within the same physical address space. In other variants, the data memory and the instruction memory are kept in two physically separate memory units, each having its own distinct address space. Both variants have pros and cons, as we'll discuss later.

Data memory: High-level programs are designed to manipulate abstract artifacts like variables, arrays, and objects. Yet at the hardware level, these data abstractions are realized by binary values stored in memory registers. In particular, following translation to machine language, abstract array processing and get/set operations on objects are reduced to *reading* and *writing* selected memory registers. To read a register, we supply an address and probe the value of the selected register. To write to a register, we supply an address and store a new value in the selected register, overriding its previous value.

Instruction memory: Before a high-level program can be executed on a target computer, it must first be translated into the machine language of the target computer. Each high-level statement is translated into one or more low-level instructions, which are then written as binary values to a file called the *binary*, or *executable*, version of the program. Before running a program, we must first load its binary version from a mass storage device, and serialize its instructions into the computer's *instruction memory*.

From the pure focus of computer architecture, *how* a program is loaded into the computer's memory is considered an external issue. What's important is that when the CPU is called upon to execute a program, the program's code will already reside in the computer's memory.

5.1.4 Central Processing Unit

The Central Processing Unit (CPU)—the centerpiece of the computer's architecture—is in charge of executing the instructions of the currently running program. Each instruction tells the CPU which computation to perform, which registers to access, and which instruction to fetch and execute next. The CPU executes these tasks using three main elements: An Arithmetic Logic Unit (ALU), a set of registers, and a control unit.

Arithmetic Logic Unit: The ALU chip is built to perform all the low-level arithmetic and logical operations featured by the computer. A typical ALU can add two given values, compute their bitwise And, compare them for equality, and so on. How much functionality the ALU should feature is a design decision. In general, any function not supported by the ALU can be realized later, using system software running on top of the hardware platform. The trade-off is simple: hardware implementations are typically more efficient but result in more expensive hardware, while software implementations are inexpensive and less efficient.

Registers: In the course of performing computations, the CPU is often required to store interim values temporarily. In theory, we could have stored these values in memory registers, but this would entail long-distance trips between the CPU and the RAM, which are two separate chips. These delays would frustrate the CPU-resident ALU, which is an ultra-fast combinational calculator. The result will be a condition known as *starvation*, which is what happens when a fast processor depends on a sluggish data store for supplying its inputs and consuming its outputs.

In order to avert starvation and boost performance, we normally equip the CPU with a small set of high-speed (and relatively expensive) *registers*, acting as the processor's immediate memory. These registers serve various purposes: *data registers* store interim values, *address registers* store values

that are used to address the RAM, the *program counter* stores the address of the instruction that should be fetched and executed next, and the *instruction register* stores the current instruction. A typical CPU uses a few dozen such registers, but our frugal Hack computer will need only three.

Control: A computer instruction is a structured package of agreed-upon micro-codes, that is, sequences of one or more bits designed to signal different devices what to do. Thus, before an instruction can be executed, it must first be decoded into its micro-codes. Next, each micro-code is routed to its designated hardware device (ALU, registers, memory) within the CPU, where it tells the device how to partake in the overall execution of the instruction.

Fetch-Execute: In each step (cycle) of the program's execution, the CPU fetches a binary machine instruction from the instruction memory, decodes it, and executes it. As a side effect of the instruction's execution, the CPU also figures out which instruction to fetch and execute next. This repetitive process is sometimes referred to as the *fetch-execute cycle*.

5.1.5 Input and Output

Computers interact with their external environments using a great variety of input and output (I/O) devices: screens, keyboards, storage devices, printers, microphones, speakers, network interface cards, and so on, not to mention the bewildering array of sensors and activators embedded in automobiles, cameras, hearing aids, alarm systems, and all the gadgets around us. There are two reasons why we don't concern ourselves with these I/O devices. First, every one of them represents a unique piece of machinery, requiring a unique knowledge of engineering. Second, for that very same reason, computer scientists have devised clever schemes for abstracting away this complexity and making all I/O devices look exactly the same to the computer. The key element in this abstraction is called *memory-mapped* I/O.

The basic idea is to create a binary emulation of the I/O device, making it appear to the CPU as if it were a regular linear memory segment. This is done by allocating, for each I/O device, a designated area in the computer's

memory that acts as its memory map. In the case of an input device like a keyboard, the memory map is made to continuously reflect the physical state of the device: when the user presses a key on the keyboard, a binary code representing that key appears in the keyboard's memory map. In the case of an output device like a screen, the screen is made to continuously reflect the state of its designated memory map: when we write a bit in the screen's memory map, a respective pixel is turned on or off on the screen.

The I/O devices and the memory maps are refreshed, or synchronized, many times per second, so the response time from the user's perspective appears to be instantaneous. Programmatically, the key implication is that low-level computer programs can access any I/O device by manipulating its designated memory map.

The memory map convention is based on several agreed-upon contracts. First, the data that drives each I/O device must be serialized, or mapped, onto the computer's memory, hence the name *memory map*. For example, the screen, which is a two-dimensional grid of pixels, is mapped on a one-dimensional block of fixed-size memory registers. Second, each I/O device is required to support an agreed-upon interaction protocol so that programs will be able to access it in a predictable manner. For example, it should be decided which binary codes should represent which keys on the keyboard. Given the multitude of computer platforms, I/O devices, and different hardware and software vendors, one can appreciate the crucial role that agreed-upon industry-wide *standards* play in realizing these low-level interaction contracts.

The practical implications of memory-mapped I/O are significant: The computer system is totally independent of the number, nature, or make of the I/O devices that interact, or may interact, with it. Whenever we want to connect a new I/O device to the computer, all we have to do is allocate to it a new memory map and take note of the map's base address (these onetime configurations are carried out by the so-called *installer* programs). Another necessary element is a *device driver* program, which is added to the computer's operating system. This program bridges the gap between the I/O device's memory map data and the way this data is actually rendered on, or generated by, the physical I/O device.

5.2 The Hack Hardware Platform: Specification

The architectural framework described thus far is characteristic of any general-purpose computer system. We now turn to describe one specific variant of this architecture: the Hack computer. As usual in Nand to Tetris, we start with the abstraction, focusing on *what* the computer is designed to do. The computer's implementation—*how* it does it—is described later.

5.2.1 Overview

The Hack platform is a 16-bit von Neumann machine designed to execute programs written in the Hack machine language. In order to do so, the Hack platform consists of a *CPU*, two separate memory modules serving as *instruction memory* and *data memory*, and two memory-mapped I/O devices: a *screen* and a *keyboard*.

The Hack computer executes programs that reside in an instruction memory. In physical implementations of the Hack platform, the instruction memory can be implemented as a ROM (Read-Only Memory) chip that is preloaded with the required program. Software-based emulators of the Hack computer support this functionality by providing means for loading the instruction memory from a text file containing a program written in the Hack machine language.

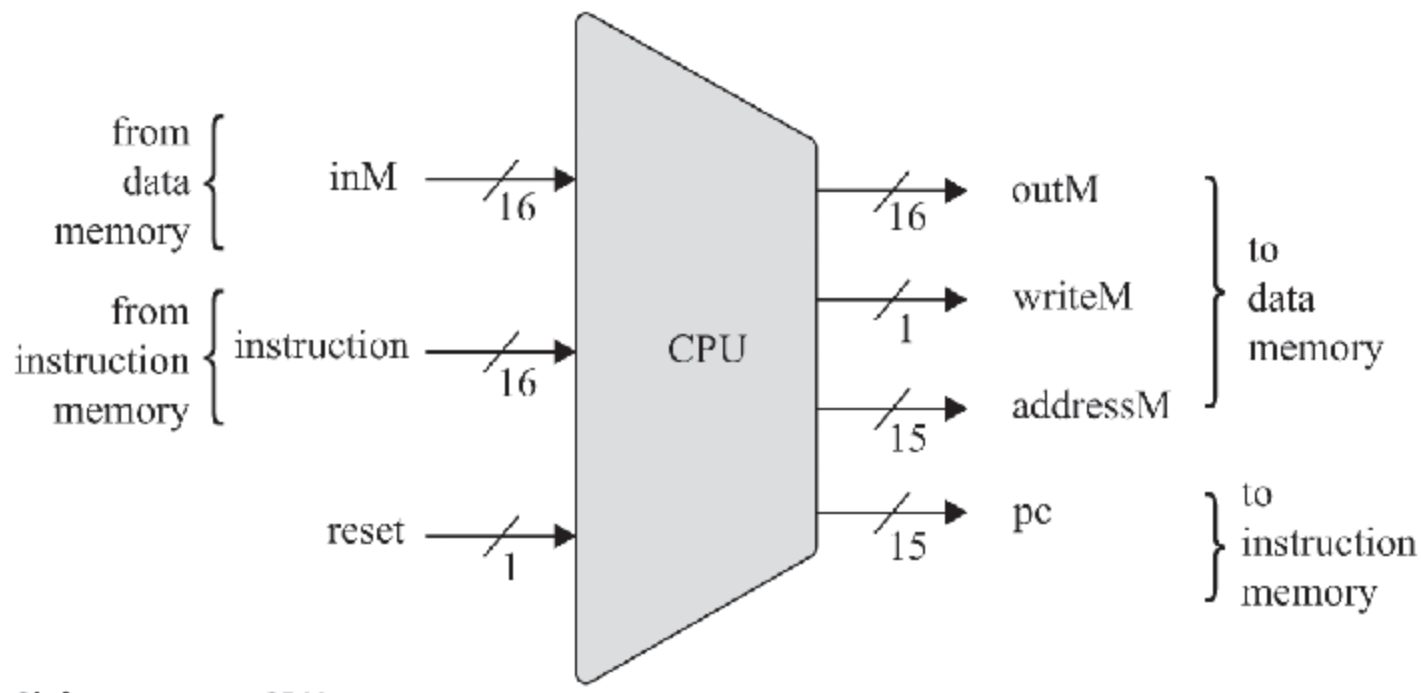
The Hack CPU consists of the ALU built in project 2 and three registers named *Data register* (D), *Address register* (A), and *Program Counter* (PC). The D register and the A register are identical to the Register chip built in project 3, and the program counter is identical to the PC chip built in project 3. While the D register is used solely for storing data values, the A register serves one of three different purposes, depending on the context in which it is used: storing a data value (like the D register), selecting an address in the instruction memory, or selecting an address in the data memory.

The Hack CPU is designed to execute instructions written in the Hack machine language. In case of an *A*-instruction, the 16 bits of the instruction are treated as a binary value which is loaded as is into the A register. In case of a *C*-instruction, the instruction is treated as a capsule of control bits that specify various micro-operations to be performed by various chip-parts

within the CPU. We now turn to describe how the CPU materializes these micro-codes into concrete actions.

5.2.2 Central Processing Unit

The Hack CPU interface is shown in figure 5.2. The CPU is designed to execute 16-bit instructions according to the Hack machine language specification presented in chapter 4. The CPU consists of an ALU, two registers named A and D, and a program counter named PC (these internal chip-parts are not seen outside the CPU). The CPU expects to be connected to an instruction memory, from which it fetches instructions for execution, and to a data memory, from which it can read, and into which it can write, data values. The inM input and the outM output hold the values referred to as M in the *C*-instruction syntax. The addressM output holds the address at which outM should be written.




```
Chip name: CPU
Input:
   instruction[16]   // Instruction to execute
   inM[16]           // The instruction's M input (contents of RAM[A])
   reset             // Signals whether to restart the program (if reset==1)
                     // or continue executing the program (if reset==0).
Output:
   outM[16]          // Written to RAM[addressM], the instruction's M output
   addressM[15]      // At which address to write?
   writeM            // Write to the memory?
   pc[15]            // Address of next instruction
```

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

If the instruction input is an *A*-instruction, the CPU loads the 16-bit instruction value into the A register. If instruction is a *C*-instruction, then (i) the CPU causes the ALU to perform the computation specified by the instruction and (ii) the CPU causes this value to be stored in the subset of {A,D,M} destination registers specified by the instruction. If one of the destination registers is M, the CPU's outM output is set to the ALU output, and the CPU's writeM output is set to 1. Otherwise, writeM is set to 0, and any value may appear in outM.

As long as the reset input is 0, the CPU uses the ALU output and the jump bits of the current instruction to decide which instruction to fetch next. If reset is 1, the CPU sets pc to 0. Later in the chapter we'll connect the CPU's pc output to the address input of the instruction memory chip, causing the latter to emit the next instruction. This configuration will realize the fetch step of the fetch-execute cycle.

The CPU's outM and writeM outputs are realized by *combinational* logic; thus, they are affected instantaneously by the instruction's execution. The addressM and pc outputs are *clocked*: although they are affected by the instruction's execution, they commit to their new values only in the next time step.

5.2.3 Instruction Memory

The Hack *instruction memory*, called ROM32K, is specified in figure 5.3.


```
Chip name:  ROM32K
Input:      address[15]
Output:     out[16]
Function:   Emits the 16-bit value stored in the address selected by the
            address input. It is assumed that the chip is preloaded
            with a program written in the Hack machine language.
```

Figure 5.3 The Hack instruction memory interface.

5.2.4 Input/Output

Access to the input/output devices of the Hack computer is made possible by the computer's *data memory*, a read/write RAM device consisting of 32K addressable 16-bit registers. In addition to serving as the computer's general-purpose data store, the data memory also interfaces between the CPU and the computer's input/output devices, as we now turn to describe.

The Hack platform can be connected to two peripheral devices: a *screen* and a *keyboard*. Both devices interact with the computer platform through designated memory areas called *memory maps*. Specifically, images can be drawn on the screen by writing 16-bit values in a designated memory segment called a *screen memory map*. Similarly, which key is presently pressed on the keyboard can be determined by probing a designated 16-bit memory register called a *keyboard memory map*.

The screen memory map and the keyboard memory map are continuously updated, many times per second, by peripheral refresh logic that is external to the computer. Thus, when one or more bits are changed in the screen memory map, the change is immediately reflected on the physical screen. Likewise, when a key is pressed on the physical keyboard, the character code of the pressed key immediately appears in the keyboard memory map. With that in mind, when a low-level program wants to read something from the keyboard, or write something on the screen, the program manipulates the respective memory maps of these I/O devices.

In the Hack computer platform, the screen memory map and the keyboard memory map are realized by two built-in chips named Screen and Keyboard. These chips behave like standard memory devices, with the additional side effects of continuously synchronizing between the I/O devices and their respective memory maps. We now turn to specify these chips in detail.

Screen: The Hack computer can interact with a physical screen consisting of 256 rows of 512 black-and-white pixels each, spanning a grid of 131,072 pixels. The computer interfaces with the physical screen via a memory map, implemented by an 8K memory chip of 16-bit registers. This chip, named Screen, behaves like a regular memory chip, meaning that it can be read and written to using the regular RAM interface. In addition, the Screen chip

features the side effect that the state of any one of its bits is continuously reflected by a respective pixel in the physical screen (1 = black, 0 = white).

The physical screen is a two-dimensional address space, where each pixel is identified by a row and a column. High-level programming languages typically feature a graphics library that allows accessing individual pixels by supplying (*row*,*column*) coordinates. However, the memory map that represents this two-dimensional screen at the low level is a one-dimensional sequence of 16-bit words, each identified by supplying an address. Therefore, individual pixels cannot be accessed directly. Rather, we have to figure out which word the target bit is located in and then access, and manipulate, the entire 16-bit word this pixel belongs to. The exact mapping between these two address spaces is specified in figure 5.4. This mapping will be realized by the screen driver of the operating system that we'll develop in part II of the book.

```
Chip name: Screen          // Screen memory map
Input:     in[16]          // What to write
           address[13]     // Where to read/write
           load            // Write-enable bit
Output:    out[16]         // Screen value at the given address
Function:  Exactly like a 16-bit, 8K RAM, plus a screen refresh side effect.
    Emits the value stored at the memory location specified by address.
    If load==1, then the memory location specified by address is set to the value of in.

    The loaded value will be emitted by out from the next time step onward.
    In addition, the chip continuously refreshes a physical screen, consisting of 256 rows
    and 512 columns of black and white pixels.

    The pixel at row r from the top and column c from the left (0 ≤ r ≤ 255, 0 ≤ c ≤ 511)
    is mapped onto the c%16 bit (counting from LSB to MSB) of the 16-bit word stored
    in Screen[r * 32 + c / 16].

    (Simulators of the Hack computer are expected to simulate the physical screen,
    the mapping, and the refresh contract).
```

Figure 5.4 The Hack Screen chip interface.

Keyboard: The Hack computer can interact with a physical keyboard, like that of a personal computer. The computer interfaces with the physical keyboard via a memory map, implemented by the Keyboard chip, whose interface is given in figure 5.5. The chip interface is identical to that of a read-only, 16-bit register. In addition, the Keyboard chip has the side effect of reflecting the state of the physical keyboard: When a key is pressed on the

physical keyboard, the 16-bit code of the respective character is emitted by the output of the Keyboard chip. When no key is pressed, the chip outputs 0. The character set supported by the Hack computer is given in appendix 5, along with the code of each character.

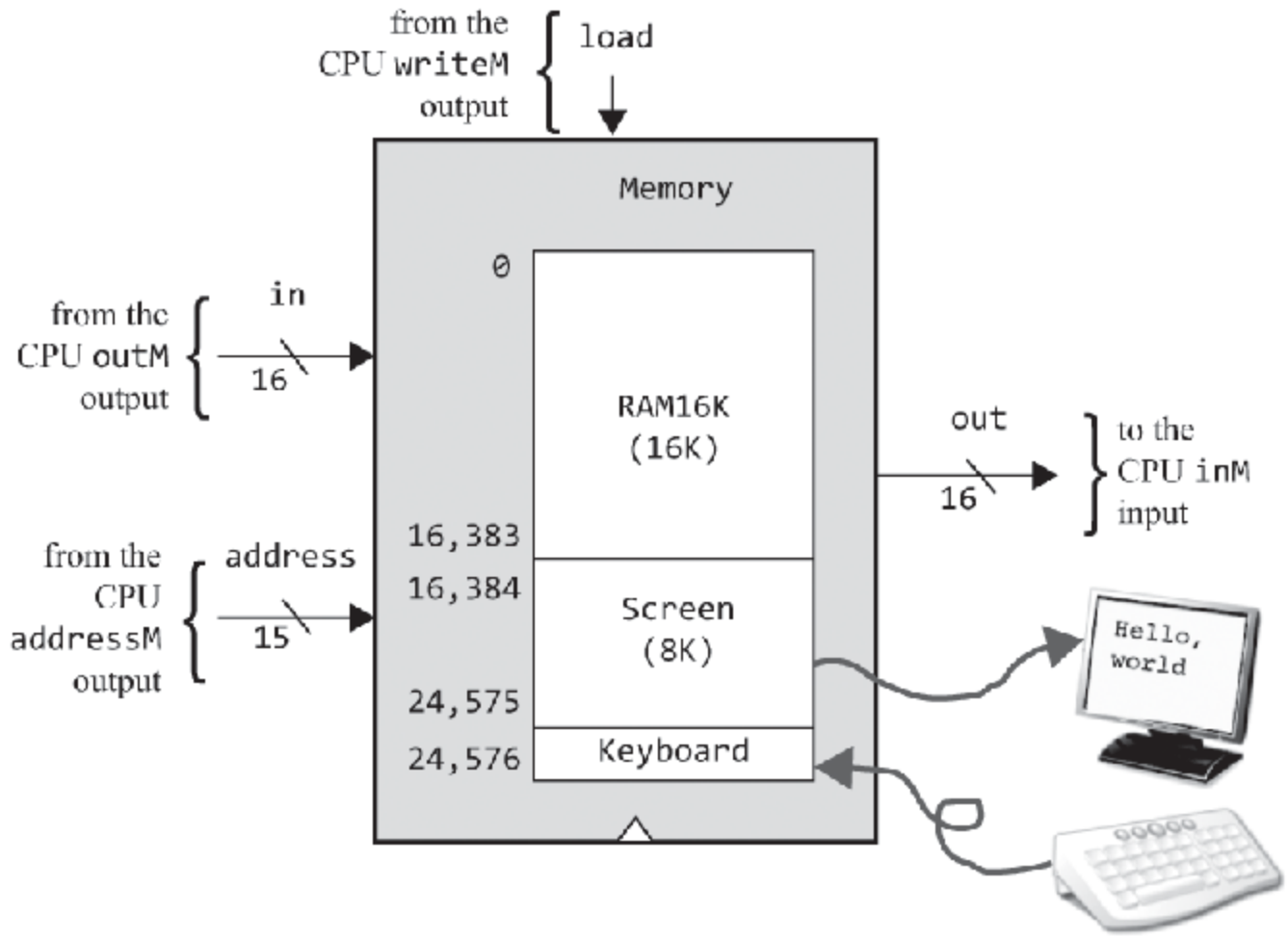
```
Chip name: Keyboard    // Keyboard memory map
Output:    out[16]
Function:  Emits the 16-bit character code of the currently pressed
           key on the physical keyboard or 0 if no key is pressed.
```

(Simulators of the Hack computer are expected to simulate this refresh contract).

Figure 5.5 The Hack Keyboard chip interface.

5.2.5 Data Memory

The overall address space of the Hack *data memory* is realized by a chip named Memory. This chip is essentially a package of three 16-bit chip-parts: RAM16K (a RAM chip of 16K registers, serving as a general-purpose data store), Screen (a built-in RAM chip of 8K registers, acting as the screen memory map), and Keyboard (a built-in register chip, acting as the keyboard memory map). The complete specification is given in figure 5.6.

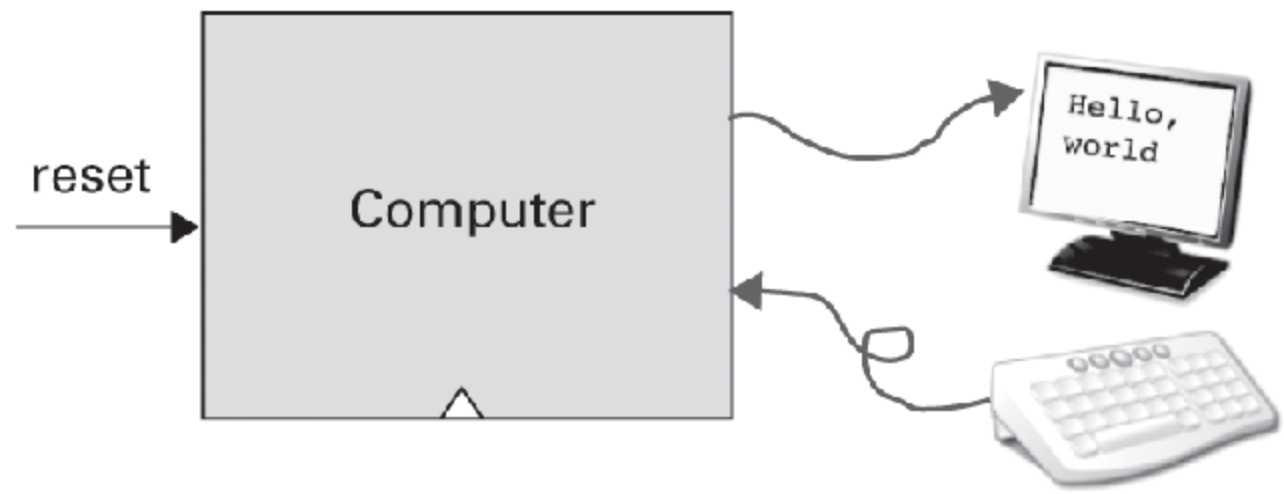



```
Chip name:  Memory          // Data memory
Input:      in[16]          // What to write
            address[15]     // Where to read/write
            load            // Write-enable bit
Output:     out[16]         // Value at the given address
Function:
    The complete address space of the Hack computer's data memory.
    Only the top 16K+8K+1 words of the address space are used.
    Accessing an address in the range 0 - 16383 results in accessing RAM16K;
    Accessing an address in the range 16384 - 24575 results in accessing Screen;
    Accessing the address 24576 results in accessing Keyboard;
    Accessing any other address is invalid.
```

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

5.2.6 Computer

The topmost element in the Hack hardware hierarchy is a computer-on-a-chip named Computer (figure 5.7). The Computer chip can be connected to a screen and a keyboard. The user sees the screen, the keyboard, and a single bit input named reset. When the user sets this bit to 1 and then to 0, the computer starts executing the currently loaded program. From this point onward, the user is at the mercy of the software.




```
Chip name: Computer
Input:     reset
Function:
```

When `reset==0`, the program stored in the computer executes.
When `reset==1`, the execution of the program restarts.
To start the program's execution, set `reset` to `1`, and then to `0`.
(It is assumed that the computer's instruction memory is loaded with a program written in the Hack machine language).

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

This startup logic realizes what is sometimes referred to as "*booting* the computer." For example, when you boot up a PC or a cell phone, the device is set up to run a ROM-resident program. This program, in turn, loads the operating system's kernel (also a program) into the RAM and starts executing it. The kernel then executes a process (yet another program) that listens to the computer's input devices, that is, keyboard, mouse, touch screen, microphone, and so on. At some point the user will do something, and the OS will respond by running another process or invoking some program.

In the Hack computer, the software consists of a binary sequence of 16-bit instructions, written in the Hack machine language and stored in the computer's instruction memory. Typically, this binary code will be the low-level version of a program written in some high-level language and translated by a *compiler* into the Hack machine language. The compilation process will be discussed and implemented in part II of the book.

5.3 Implementation

This section describes a hardware implementation that realizes the Hack computer specified in the previous section. As usual, we don't give exact building instructions. Rather, we expect readers to discover and complete the implementation details on their own. All the chips described below can be built in HDL and simulated on a personal computer using the supplied hardware simulator.

5.3.1 The Central Processing Unit

The implementation of the Hack CPU entails building a logic gate architecture capable of (i) executing a given Hack instruction and (ii) determining which instruction should be fetched and executed next. In order to do so, we will use gate logic for decoding the current instruction, an Arithmetic Logic Unit (ALU) for computing the function specified by the instruction, a set of registers for storing the resulting value, as specified by the instruction, and a program counter for keeping track of which instruction should be fetched and executed next. Since all the underlying building blocks (ALU, registers, PC, and elementary logic gates) were already built in previous chapters, the key question that we now face is how to connect these chip-parts judiciously in a way that realizes the desired CPU operation. One possible configuration is illustrated in figure 5.8 and explained in the following pages.

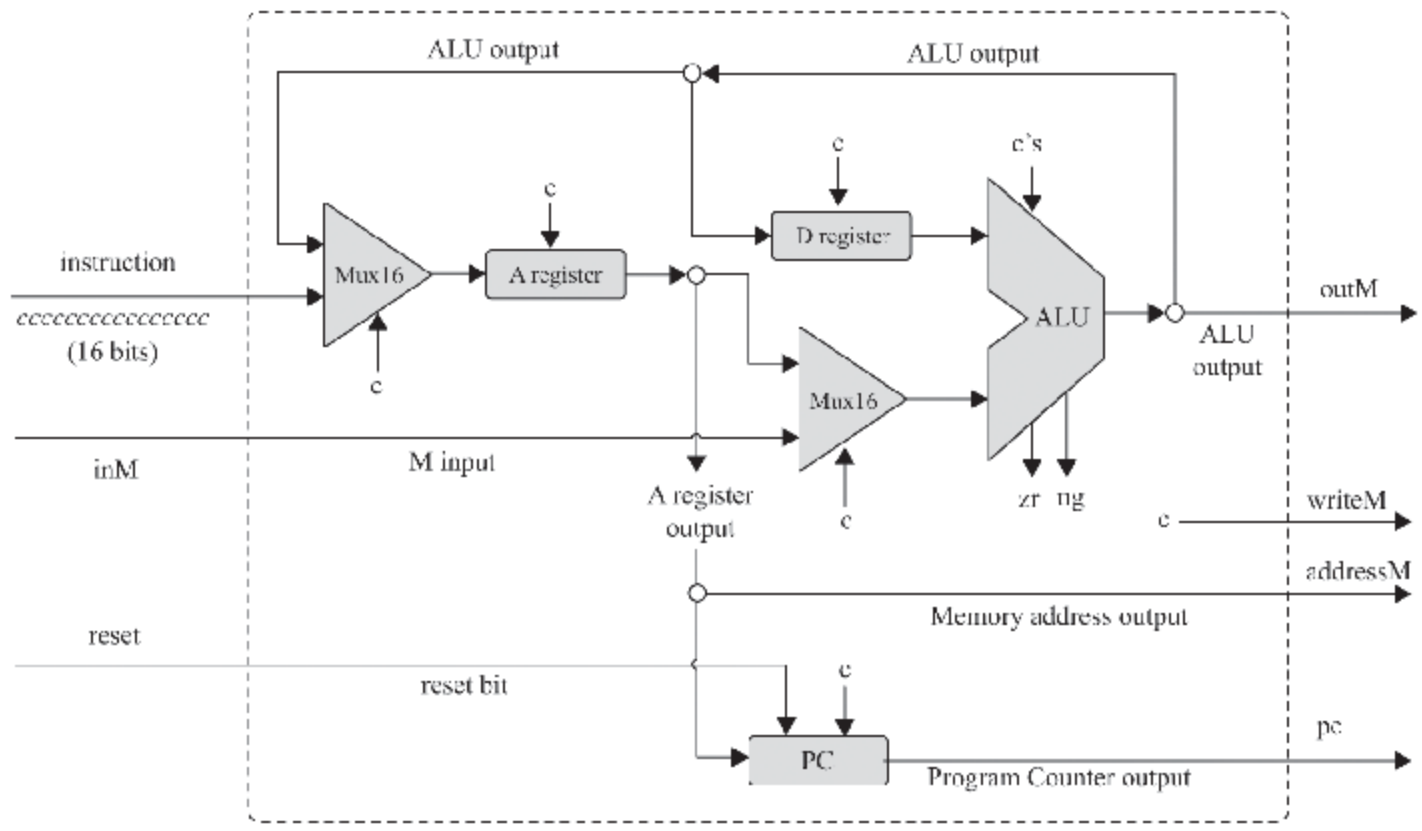


Figure 5.8 Proposed implementation of the Hack CPU, showing an incoming 16-bit instruction. We use the instruction notation *cccccccccccccccc* to emphasize that in the case of a *C*-instruction, the instruction is treated as a capsule of control bits, designed to control different CPU chip-parts. In 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.

Instruction decoding: Let's start by focusing on the CPU's instruction input. This 16-bit value represents either an *A*-instruction (when the leftmost bit is 0) or a *C*-instruction (when the leftmost bit is 1). In case of an *A*-instruction, the instruction bits are interpreted as a binary value that should be loaded into the A register. In case of a *C*-instruction, the instruction is treated as a capsule of control bits 1xxaccccccdddjjj, as follows. The a and cccccc bits code the *comp* part of the instruction; the ddd bits code the *dest* part of the instruction; the jjj bits code the *jump* part of the instruction. The xx bits are ignored.

Instruction execution: In case of an *A*-instruction, the 16 bits of the instruction are loaded as is into the A register (actually, this is a 15-bit value, since the MSB is the op-code 0). In case of a *C*-instruction, the a-bit determines whether the ALU input will be fed from the A register value or from the incoming M value. The cccccc bits determine which function the ALU will compute. The ddd bits determine which registers should accept the ALU output. The jjj bits are used for determining which instruction to fetch next.

The CPU architecture should extract the control bits described above from the instruction input and route them to their chip-part destinations, where they instruct the chip-parts what to do in order to partake in the instruction's execution. Note that every one of these chip-parts is already designed to carry out its intended function. Therefore, the CPU design is mostly a matter of connecting existing chips in a way that realizes this execution model.

Instruction fetching: As a side effect of executing the current instruction, the CPU determines, and outputs, the address of the instruction that should

be fetched and executed next. The key element in this subtask is the *Program Counter*—a CPU chip-part whose role is to always store the address of the next instruction.

According to the Hack computer specification, the current program is stored in the instruction memory, starting at address 0. Hence, if we wish to start (or restart) a program's execution, we should set the Program Counter to 0. That's why in figure 5.8 the reset input of the CPU is fed directly into the reset input of the PC chip-part. If we assert this bit, we'll effect PC=0, causing the computer to fetch and execute the first instruction in the program.

What should we do next? Normally, we'd like to execute the next instruction in the program. Therefore, and assuming that the reset input had been set "back" to 0, the default operation of the program counter is PC++.

But what if the current instruction includes a jump directive? According to the language specification, execution always branches to the instruction whose address is the current value of A. Thus, the CPU implementation must realize the following Program Counter behavior: if *jump* then PC=A else PC++.

How can we effect this behavior using gate logic? The answer is hinted in figure 5.8. Note that the output of the A register feeds into the input of the PC register. Thus, if we assert the PC's load-bit, we'll enable the operation PC=A rather than the default operation PC++. We should do this only if we have to effect a jump. Which leads to the next question: How do we know if we have to effect a jump? The answer depends on the three j-bits of the current instruction and the two ALU output bits zr and ng. Taken together, these bits can be used to determine whether the jump condition is satisfied or not.

We'll stop here, lest we rob readers of the pleasure of completing the CPU implementation themselves. We hope that as they do so, they will savor the clockwork elegance of the Hack CPU.

5.3.2 Memory

The Memory chip of the Hack computer is an aggregate of three chip-parts: RAM16K, Screen, and Keyboard. This modularity, though, is implicit: Hack

machine language programs see a *single address space*, ranging from address 0 to address 24576 (in hexadecimal: 6000).

The Memory chip interface is shown in figure 5.6. The implementation of this interface should realize the continuum effect just described. For example, if the address input of the Memory chip happens to be 16384, the implementation should access address 0 in the Screen chip, and so on. Once again, we prefer not to provide too many details and let you figure out the rest of the implementation yourself.

5.3.3 Computer

We have reached the end of our hardware journey. The topmost Computer chip can be realized using three chip-parts: the CPU, the data Memory chip, and the instruction memory chip, ROM32K. Figure 5.9 gives the details.

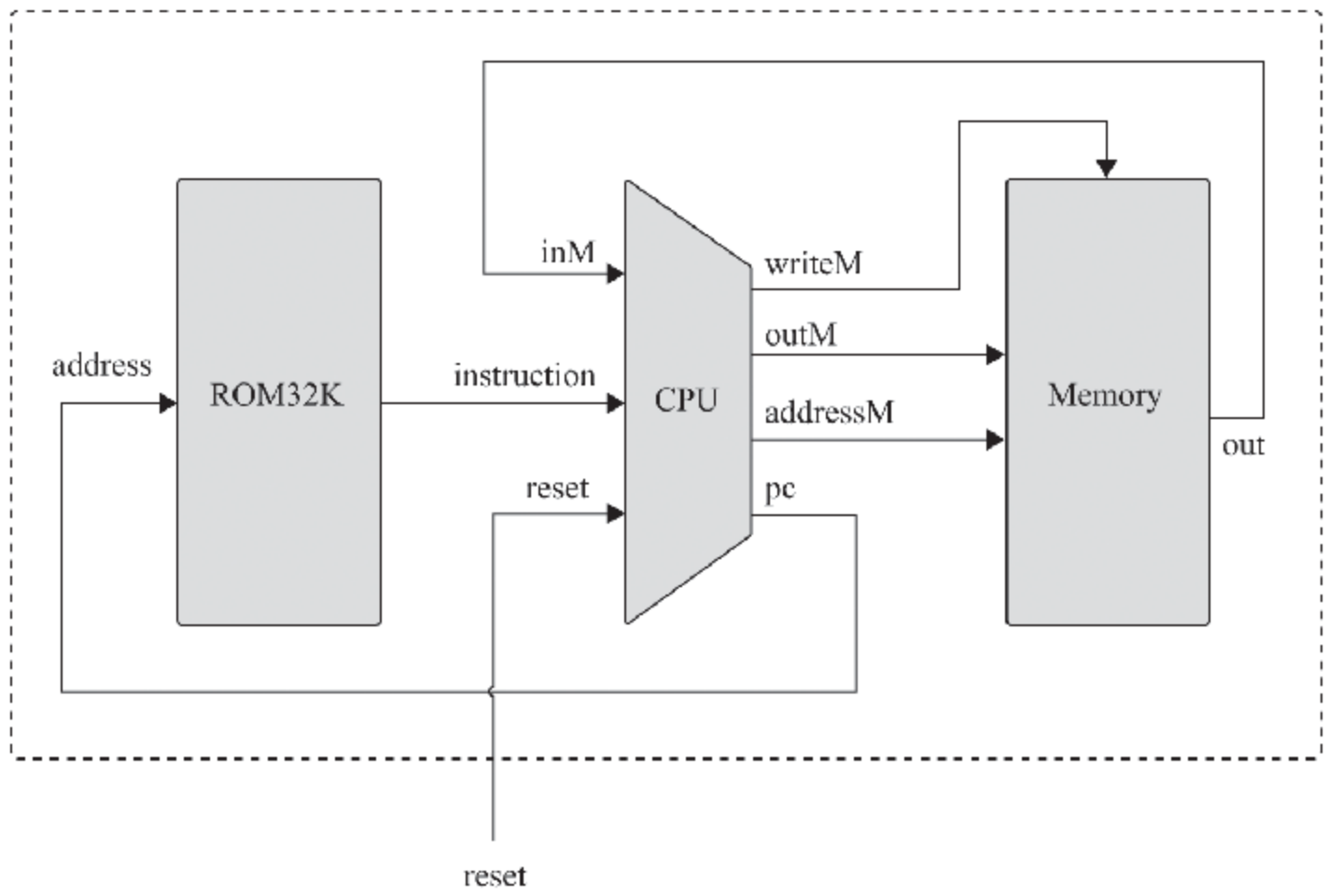


Figure 5.9 Proposed implementation of Computer, the topmost chip in the Hack platform.

The Computer implementation is designed to realize the following fetch-execute cycle: When the user asserts the reset input, the CPU's pc output emits 0, causing the instruction memory (ROM32K) to emit the first instruction in the program. The instruction will be executed by the CPU, and this execution may involve reading or writing a data memory register.

In the process of executing the instruction, the CPU figures out which instruction to fetch next, and emits this address through its pc output. The CPU's pc output feeds the address input of the instruction memory, causing the latter to output the instruction that ought to be executed next. This output, in turn, feeds the instruction input of the CPU, closing the fetch-execute cycle.

5.4 Project

Objective: Build the Hack computer, culminating in the topmost Computer chip.

Resources: All the chips described in this chapter should be written in HDL and tested on the supplied hardware simulator, using the test programs described below.

Contract: Build a hardware platform capable of executing programs written in the Hack machine language. Demonstrate the platform's operations by having your Computer chip run the three supplied test programs.

Test programs: A natural way to test the overall Computer chip implementation is to have it execute sample programs written in the Hack machine language. In order to run such a test, one can write a test script that loads the Computer chip into the hardware simulator, loads a program from an external text file into the ROM32K chip-part (the instruction memory), and then runs the clock enough cycles to execute the program. We provide three such test programs, along with corresponding test scripts and compare files:

- Add.hack: Adds the two constants 2 and 3, and writes the result in RAM[0].
- Max.hack: Computes the maximum of RAM[0] and RAM[1] and writes the result in RAM[2].
- Rect.hack: Draws on the screen a rectangle of RAM[0] rows of 16 pixels each. The rectangle's top-left corner is located at the top-left corner of the screen.

Before testing your Computer chip on any one of these programs, review the test script associated with the program, and be sure to understand the instructions given to the simulator. If needed, consult appendix 3 ("Test Description Language").

Steps: Implement the computer in the following order:

Memory: This chip can be built along the general outline given in figure 5.6, using three chip-parts: RAM16K, Screen, and Keyboard. Screen and Keyboard are available as built-in chips; there is no need to build them. Although the RAM16K chip was built in project 3, we recommend using its built-in version instead.

CPU: The central processing unit can be built according to the proposed implementation given in figure 5.8. In principle, the CPU can use the ALU built in project 2, the Register and PC chips built in project 3, and logic gates from project 1, as needed. However, we recommend using the built-in versions of all these chips (in particular, use the built-in registers ARegister, DRegister, and PC). The built-in chips have exactly the same functionality as the memory chips built in previous projects, but they feature GUI side effects that make the testing and simulation of your work easier.

In the course of implementing the CPU, you may be tempted to specify and build internal ("helper") chips of your own. Be advised that there is no need to do so; the Hack CPU can be implemented elegantly and efficiently using only the chip-parts that appear in figure 5.8, plus some elementary logic gates built in project 1 (of which it is best to use their built-in versions).

Instruction memory: Use the built-in ROM32K chip.

Computer: The computer can be built according to the proposed implementation given in figure 5.9.

Hardware simulator: All the chips in this project, including the topmost Computer chip, can be implemented and tested using the supplied hardware simulator. Figure 5.10 is a screenshot of testing the Rect.hack program on a Computer chip implementation.

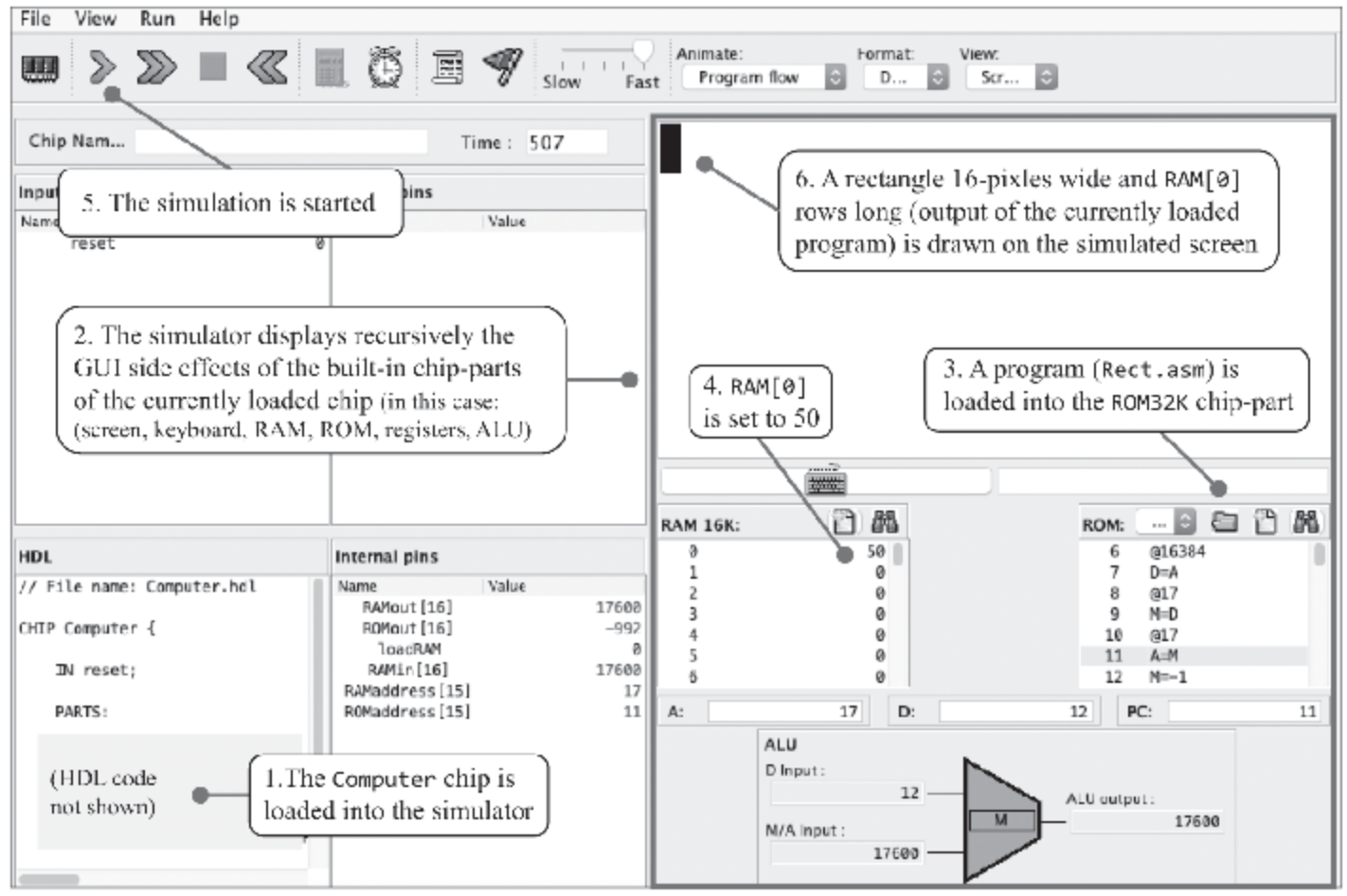


Figure 5.10 Testing the Computer chip on the supplied hardware simulator. The stored program is Rect, which draws a rectangle of RAM[0] rows of 16 pixels each, all black, at the top-left of the screen.

A web-based version of project 5 is available at www.nand2tetris.org.

5.5 Perspective

Following the Nand to Tetris spirit, the architecture of the Hack computer is minimal. Typical computer platforms feature more registers, more data types, more powerful ALUs, and richer instruction sets. Most of these differences, though, are quantitative. From a qualitative standpoint, almost all digital computers, including Hack, are based on the same conceptual architecture: the von Neumann machine.

In terms of *purpose*, computer systems can be classified into two categories: *general-purpose computers* and *single-purpose computers*. General-purpose computers, like PCs and cell phones, typically interact with a user. They are designed to execute many programs and easily switch from one program to another. *Single-purpose computers* are usually embedded in other systems like automobiles, cameras, media streamers, medical devices, industrial controllers, and so on. For any particular application, a single program is burned into the dedicated computer's ROM

(Read-Only Memory). For example, in some game consoles, the game software resides in an external cartridge, which is a replaceable ROM module encased in a fancy package. Although general-purpose computers are typically more complex and versatile than dedicated computers, they all share the same basic architectural ideas: stored programs, fetch-decode-execute logic, CPU, registers, and counters.

Most general-purpose computers use a single address space for storing both programs and data. Other computers, like Hack, use two separate address spaces. The latter configuration, which for historical reasons is called *Harvard architecture*, is less flexible in terms of ad hoc memory utilization but has distinct advantages. First, it is easier and cheaper to build. Second, it is often faster than the single address space configuration. Finally, if the size of the program that the computer has to run is known in advance, the size of the instruction memory can be optimized and fixed accordingly. For these reasons, the Harvard architecture is the architecture of choice in many dedicated, single-purpose, embedded computers.

Computers that use the same address space for storing instructions and data face the following challenge: How can we feed the address of the instruction, and the address of the data register on which the instruction has to operate, into the same address input of the shared memory device? Clearly, we cannot do it at the same time. The standard solution is to base the computer operation on a two-cycle logic. During the *fetch cycle*, the instruction address is fed to the address input of the memory, causing it to immediately emit the current instruction, which is then stored in an *instruction register*. In the subsequent *execute cycle*, the instruction is decoded, and the data address on which it has to operate is fed to the same address input. In contrast, computers that use separate instruction and data memories, like Hack, benefit from a single-cycle fetch-execute logic, which is faster and easier to handle. The price is having to use separate data and instruction memory units, although there is no need to use an instruction register.

The Hack computer interacts with a screen and a keyboard. General-purpose computers are typically connected to multiple I/O devices like printers, storage devices, network connections, and so on. Also, typical display devices are much fancier than the Hack screen, featuring more pixels, more colors, and faster rendering performance. Still, the basic

principle that each pixel is driven by a memory-resident binary value is maintained: instead of a single bit controlling the pixel's black or white color, typically 8 bits are devoted to controlling the brightness level of each of several primary colors that, taken together, produce the pixel's ultimate color. The result is millions of possible colors, more than the human eye can discern.

The mapping of the Hack screen on the computer's main memory is simplistic. Instead of having memory bits drive pixels directly, many computers allow the CPU to send high-level graphic instructions like "draw a line" or "draw a circle" to a dedicated graphics chip or a standalone graphics processing unit, also known as GPU. The hardware and low-level software of these dedicated graphical processors are especially optimized for rendering graphics, animation, and video, offloading from the CPU and the main computer the burden of handling these voracious tasks directly.

Finally, it should be stressed that much of the effort and creativity in designing computer hardware is invested in achieving better performance. Many hardware architects devote their work to speeding up memory access, using clever caching algorithms and data structures, optimizing access to I/O devices, and applying pipelining, parallelism, instruction prefetching, and other optimization techniques that were completely sidestepped in this chapter.

Historically, attempts to accelerate processing performance have led to two main camps of CPU design. Advocates of the *Complex Instruction Set Computing* (CISC) approach argued for achieving better performance by building more powerful processors featuring more elaborate instruction sets. Conversely, the *Reduced Instruction Set Computing* (RISC) camp built simpler processors and tighter instruction sets, arguing that these actually deliver faster performance in benchmark tests. The Hack computer does not enter this debate, featuring neither a strong instruction set nor special hardware acceleration techniques.

6 Assembler

What's in a name? That which we call a rose by any other name would smell as sweet.

—Shakespeare, *Romeo and Juliet*

In the previous chapters, we completed the development of a hardware platform designed to run programs in the Hack machine language. We presented two versions of this language—symbolic and binary—and explained that symbolic programs can be translated into binary code using a program called an *assembler*. In this chapter we describe how assemblers work, and how they are built. This will lead to the construction of a *Hack assembler*—a program that translates programs written in the Hack symbolic language into binary code that can execute on the barebone Hack hardware.

Since the relationship between symbolic instructions and their corresponding binary codes is straightforward, implementing an assembler using a high-level programming language is not a difficult task. One complication arises from allowing assembly programs to use symbolic references to memory addresses. The assembler is expected to manage these symbols and resolve them to physical memory addresses. This task is normally done using a *symbol table*—a commonly used data structure.

Implementing the assembler is the first in a series of seven software development projects that accompany part II of the book. Developing the assembler will equip you with a basic set of general skills that will serve you well throughout all these projects and beyond: handling command-line arguments, handling input and output text files, parsing instructions, handling white space, handling symbols, generating code, and many other techniques that come into play in many software development projects.

If you have no programming experience, you can develop a paper-based assembler. This option is described in the web-based version of project 6, available at www.nand2tetris.org.

6.1 Background

Machine languages are typically specified in two flavors: *binary* and *symbolic*. A binary instruction, for example, 1100001000000011000000000000111, is an agreed-upon package of micro-codes designed to be decoded and executed by some target hardware platform. For example, the instruction's leftmost 8 bits, 11000010, can represent an operation like "load." The next 8 bits, 00000011, can represent a register, say R3. The remaining 16 bits, 0000000000000111, can represent a value, say 7. When we set out to build a hardware architecture and a machine language, we can decide that this particular 32-bit instruction will cause the hardware to effect the operation "load the constant 7 into register R3." Modern computer platforms support hundreds of such possible operations. Thus, machine languages can be complex, involving many operation codes, memory addressing modes, and instruction formats.

Clearly, specifying these operations in binary code is a pain. A natural solution is using an agreed-upon equivalent symbolic syntax, say, "load R3,7". The load operation code is sometimes called a *mnemonic*, which in Latin means a pattern of letters designed to help with remembering something. Since the translation from mnemonics and symbols to binary code is straightforward, it makes sense to write low-level programs directly in symbolic notation and have a computer program translate them into binary code. The symbolic language is called *assembly*, and the translator program *assembler*. The assembler parses each assembly instruction into its underlying fields, for example, load, R3, and 7, translates each field into its equivalent binary code, and finally assembles the generated bits into a binary instruction that can be executed by the hardware. Hence the name *assembler*.

Symbols: Consider the symbolic instruction goto 312. Following translation, this instruction causes the computer to fetch and execute the instruction

stored in address 312, which may be the beginning of some loop. Well, if it's the beginning of a loop, why not mark this point in the assembly program with a descriptive label, say LOOP, and use the command goto LOOP instead of goto 312? All we have to do is record somewhere that LOOP stands for 312. When we translate the program into binary code, we replace each occurrence of LOOP with 312. That's a small price to pay for the gain in program readability and portability.

In general, assembly languages use symbols for three purposes:

- *Labels*: Assembly programs can declare and use symbols that mark various locations in the code, for example, LOOP and END.
- *Variables*: Assembly programs can declare and use symbolic variables, for example, i and sum.
- *Predefined symbols*: Assembly programs can refer to special addresses in the computer's memory using agreed-upon symbols, for example, SCREEN and KBD.

Of course, there is no free lunch. Someone must be responsible for managing all these symbols. In particular, someone must remember that SCREEN stands for 16384, that LOOP stands for 312, that sum stands for some other address, and so on. This symbol-handling task is one of the most important functions of the assembler.

Example: Figure 6.1 lists two versions of the same program written in the Hack machine language. The symbolic version includes all sorts of things that humans are fond of seeing in computer programs: comments, white space, indentation, symbolic instructions, and symbolic references. None of these embellishments concern computers, which understand one thing only: bits. The agent that bridges the gap between the symbolic code convenient for humans and the binary code understood by the computer is the assembler.

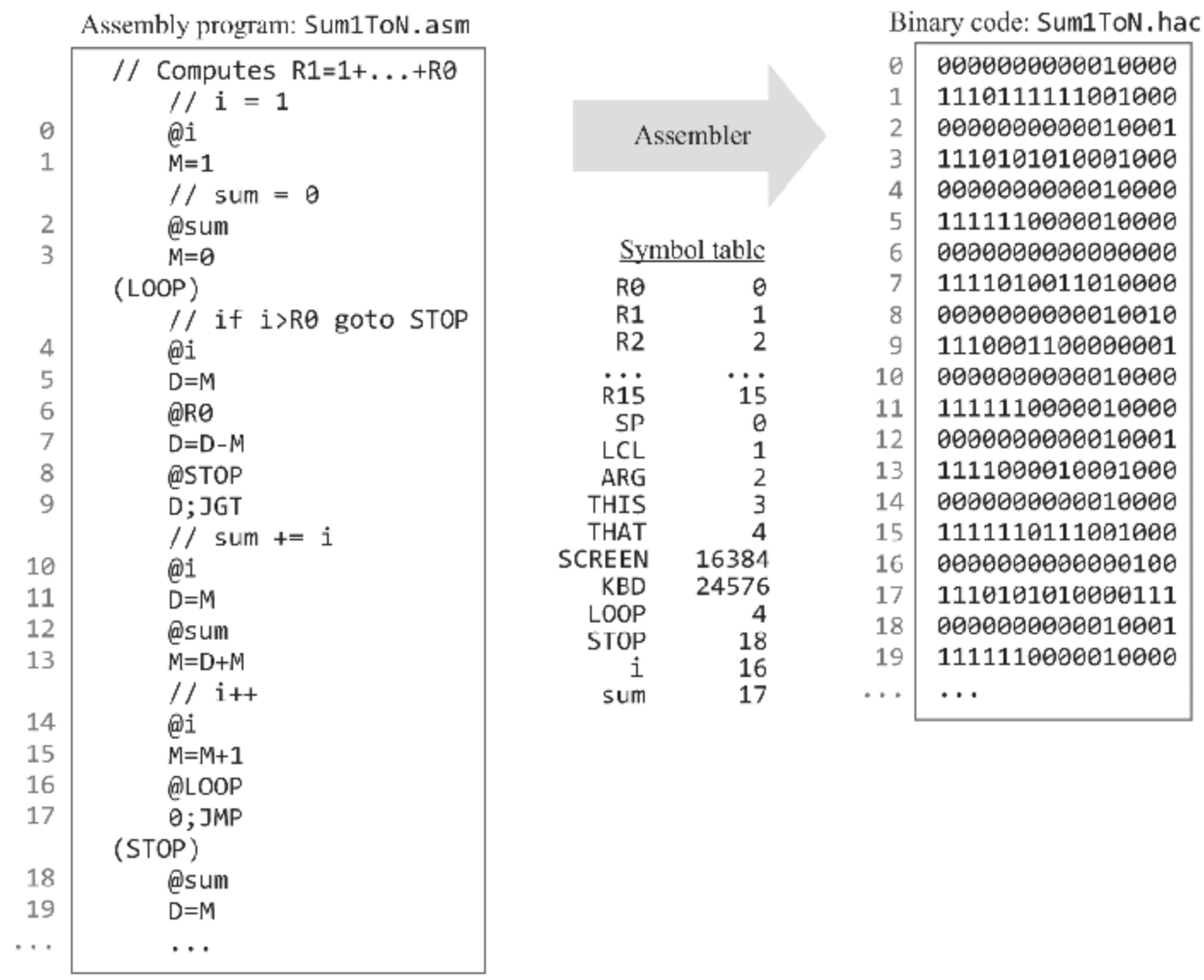


Figure 6.1 Assembly code, translated to binary code using a symbol table. The line numbers, which are not part of the code, are listed for reference.

Let us ignore for now all the details in figure 6.1, as well as the symbol table, and make some general observations. First, note that although the line numbers are not part of the code, they play an important, albeit implicit, role in the translation process. If the binary code will be loaded into the instruction memory starting at address 0, then the line number of each instruction will coincide with its memory address. Clearly, this observation should be of interest to the assembler. Second, note that comments and label declarations generate no code, and that's why the latter are sometimes called *pseudo-instructions*. Finally, and stating the obvious, note that in order to write an assembler for some machine language, the assembler's developer must get a complete specification of the language's symbolic and binary syntax.

With that in mind, we now turn to specify the Hack machine language.

6.2 The Hack Machine Language Specification

The Hack assembly language and its equivalent binary representation were described in chapter 4. The language specification is repeated here for ease of reference. This specification is the contract that Hack assemblers must implement, one way or another.

6.2.1 Programs

Binary Hack program: A binary Hack program is a sequence of text lines, each consisting of sixteen `0` and `1` characters. If the line starts with a `0`, it represents a binary *A*-instruction. Otherwise, it represents a binary *C*-instruction.

Assembly Hack program: An assembly Hack program is a sequence of text lines, each being an *assembly instruction*, a *label declaration*, or a *comment*:

- *Assembly instruction*: A symbolic *A*-instruction or a symbolic *C*-instruction (see figure 6.2).
- *Label declaration*: A line of the form (*xxx*), where *xxx* is a symbol.
- *Comment*: A line beginning with two slashes (*//*) is considered a comment and is ignored.

A-instruction

(symbolic): `@`*xxx* (*xxx* is a decimal value ranging from 0 to 32767, or a symbol bound to such a decimal value)

(binary): `0` *v v v v v v v v v v v v v v v* (*v v*... *v* = 15-bit value of *xxx*)

C-instruction

(symbolic): *dest* `=` *comp* `;` *jump* (*comp* is mandatory. If *dest* is empty, the `=` is omitted; If *jump* is empty, the `;` is omitted)

(binary): `111`*accccccdddjjj*

comp		*c*	*c*	*c*	*c*	*c*	*c*
`0`		1	0	1	0	1	0
`1`		1	1	1	1	1	1
`-1`		1	1	1	0	1	0
`D`		0	0	1	1	0	0
`A`	`M`	1	1	0	0	0	0
`!D`		0	0	1	1	0	1
`!A`	`!M`	1	1	0	0	0	1
`-D`		0	0	1	1	1	1
`-A`	`-M`	1	1	0	0	1	1
`D+1`		0	1	1	1	1	1
`A+1`	`M+1`	1	1	0	1	1	1
`D-1`		0	0	1	1	1	0
`A-1`	`M-1`	1	1	0	0	1	0
`D+A`	`D+M`	0	0	0	0	1	0
`D-A`	`D-M`	0	1	0	0	1	1
`A-D`	`M-D`	0	0	0	1	1	1
`D&A`	`D&M`	0	0	0	0	0	0
`D|A`	`D|M`	0	1	0	1	0	1
		a == 0			*a* == 1		

dest	*d*	*d*	*d*	Effect: store *comp* in:
`null`	0	0	0	the value is not stored
`M`	0	0	1	`RAM[A]`
`D`	0	1	0	`D` register (reg)
`DM`	0	1	1	`D` reg and `RAM[A]`
`A`	1	0	0	`A` reg
`AM`	1	0	1	`A` reg and `RAM[A]`
`AD`	1	1	0	`A` reg and `D` reg
`ADM`	1	1	1	`A` reg, `D` reg, and `RAM[A]`

jump	*j*	*j*	*j*	Effect:
`null`	0	0	0	no jump
`JGT`	0	0	1	if *comp* > 0 jump
`JEQ`	0	1	0	if *comp* = 0 jump
`JGE`	0	1	1	if *comp* ≥ 0 jump
`JLT`	1	0	0	if *comp* < 0 jump
`JNE`	1	0	1	if *comp* ≠ 0 jump
`JLE`	1	1	0	if *comp* ≤ 0 jump
`JMP`	1	1	1	unconditional jump

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

6.2.2 Symbols

Symbols in Hack assembly programs fall into three categories: predefined symbols, label symbols, and variable symbols.

Predefined symbols: Any Hack assembly program is allowed to use predefined symbols, as follows. `R0`, `R1`, …, `R15` stand for 0, 1, … 15, respectively. `SP`, `LCL`, `ARG`, `THIS`, `THAT` stand for 0, 1, 2, 3, 4, respectively. `SCREEN` and `KBD` stand for 16384 and 24576, respectively. The values of these symbols are interpreted as addresses in the Hack RAM.

Label symbols: The pseudo-instruction (*xxx*) defines the symbol *xxx* to refer to the location in the Hack ROM holding the next instruction in the program. A label symbol can be defined once and can be used anywhere in the assembly program, even before the line in which it is defined.

Variable symbols: Any symbol *xxx* appearing in an assembly program that is not predefined and is not defined elsewhere by a label declaration (*xxx*) is treated as a variable. Variables are mapped to consecutive RAM locations as they are first encountered, starting at RAM address 16. Thus, the first variable encountered in a program is mapped to RAM[16], the second to RAM[17], and so on.

6.2.3 Syntax Conventions

Symbols: A *symbol* can be any sequence of letters, digits, underscore (_), dot (.), dollar sign ($), and colon (:) that does not begin with a digit.

Constants: May appear only in *A*-instructions of the form @*xxx*. The constant *xxx* is a value in the range 0–32767 and is written in decimal notation.

White space: Leading space characters and empty lines are ignored.

Case conventions: All the assembly mnemonics (like A+1, JEQ, and so on) must be written in uppercase. The remaining symbols—labels and variable names—are case-sensitive. The recommended convention is to use uppercase for labels and lowercase for variables.

This completes the Hack machine language specification.

6.3 Assembly-to-Binary Translation

This section describes how to translate Hack assembly programs into binary code. Although we focus on developing an assembler for the Hack language, the techniques that we present are applicable to any assembler.

The assembler takes as input a stream of assembly instructions and generates as output a stream of translated binary instructions. The resulting code can be loaded as is into the computer memory and executed. In order to carry out the translation process, the assembler must handle instructions and symbols.

6.3.1 Handling Instructions

For each assembly instruction, the assembler

- parses the instruction into its underlying fields;
- for each field, generates the corresponding bit-code, as specified in figure 6.2;
- if the instruction contains a symbolic reference, resolves the symbol into its numeric value;
- assembles the resulting binary codes into a string of sixteen `0` and `1` characters; and
- writes the assembled string to the output file.

6.3.2 Handling Symbols

Assembly programs are allowed to use symbolic labels (destinations of goto instructions) before the symbols are defined. This convention makes the life of assembly code writers easier and that of assembler developers harder. A common solution is to develop a *two-pass assembler* that reads the code twice, from start to end. In the first pass, the assembler builds a *symbol table*, adds all the label symbols to the table, and generates no code. In the second pass, the assembler handles the variable symbols and generates binary code, using the symbol table. Here are the details.

Initialization: The assembler creates a symbol table and initializes it with all the predefined symbols and their pre-allocated values. In figure 6.1, the result of the initialization stage is the symbol table with all the symbols up to, and including, KBD.

First pass: The assembler goes through the entire assembly program, line by line, keeping track of the line number. This number starts at 0 and is incremented by 1 whenever an *A*-instruction or a *C*-instruction is encountered, but does not change when a comment or a label declaration is encountered. Each time a label declaration (*xxx*) is encountered, the assembler adds a new entry to the symbol table, associating the symbol *xxx*

with the current line number plus 1 (this will be the ROM address of the next instruction in the program).

This pass results in adding to the symbol table all the program's label symbols, along with their corresponding values. In figure 6.1, the first pass results in adding the symbols LOOP and STOP to the symbol table. No code is generated during the first pass.

Second pass: The assembler goes again through the entire program and parses each line as follows. Each time an *A*-instruction with a symbolic reference is encountered, namely, @*xxx*, where *xxx* is a symbol and not a number, the assembler looks up *xxx* in the symbol table. If the symbol is found, the assembler replaces it with its numeric value and completes the instruction's translation. If the symbol is not found, then it must represent a new variable. To handle it, the assembler (i) adds the entry <*xxx*, *value*> to the symbol table, where *value* is the next available address in the RAM space designated for variables, and (ii) completes the instruction's translation, using this address. In the Hack platform, the RAM space designated for storing variables starts at 16 and is incremented by 1 after each time a new variable is found in the code. In figure 6.1, the second pass results in adding the symbols i and sum to the symbol table.

6.4 Implementation

Usage: The Hack assembler accepts a single command-line argument, as follows,

prompt> HackAssembler *Prog*.asm

where the input file *Prog*.asm contains assembly instructions (the .asm extension is mandatory). The file name may contain a file path. If no path is specified, the assembler operates on the current folder. The assembler creates an output file named *Prog*.hack and writes the translated binary instructions into it. The output file is created in the same folder as the input file. If there is a file by this name in the folder, it will be overwritten.

We propose dividing the assembler implementation into two stages. In the first stage, develop a basic assembler for Hack programs that contain no symbolic references. In the second stage, extend the basic assembler to handle symbolic references.

6.4.1 Developing a Basic Assembler

The basic assembler assumes that the source code contains no symbolic references. Therefore, except for handling comments and white space, the assembler has to translate either *C*-instructions or *A*-instructions of the form @*xxx*, where *xxx* is a decimal value (and not a symbol). This translation task is straightforward: each mnemonic component (field) of a symbolic *C*-instruction is translated into its corresponding bit code, according to figure 6.2, and each decimal constant *xxx* in a symbolic *A*-instruction is translated into its equivalent binary code.

We propose basing the assembler on a software architecture consisting of a *Parser* module for parsing the input into instructions and instructions into fields, a *Code* module for translating the fields (symbolic mnemonics) into binary codes, and a *Hack assembler* program that drives the entire translation process. Before proceeding to specify the three modules, we wish to make a note about the style that we use to describe these specifications.

API documentation: The development of the Hack assembler is the first in a series of seven software construction projects that follow in part II of the book. Each one of these projects can be developed independently, using any high-level programming language. Therefore, our API documentation style makes no assumptions on the implementation language.

In each project, starting with this one, we propose an API consisting of several *modules*. Each module documents one or more *routines*. In a typical object-oriented language, a module corresponds to a *class*, and a routine corresponds to a *method*. In other languages, a module may correspond to a *file*, and a routine to a *function*. Whichever language you use for implementing the software projects, starting with the assembler, there should be no problem mapping the *modules* and *routines* of our proposed APIs on the programming elements of your implementation language.

The Parser

The Parser encapsulates access to the input assembly code. In particular, it provides a convenient means for advancing through the source code, skipping comments and white space, and breaking each symbolic instruction into its underlying components.

Although the basic version of the assembler is not required to handle symbolic references, the Parser that we specify below does. In other words, the Parser specified here serves both the basic assembler and the complete assembler.

The Parser ignores comments and white space in the input stream, enables accessing the input one line at a time, and parses symbolic instructions into their underlying components.

The Parser API is listed on the next page. Here are some examples of how the Parser services can be used. If the current instruction is `@17` or `@sum`, a call to `symbol()` would return the string `"17"` or `"sum"`, respectively. If the current instruction is `(LOOP)`, a call to `symbol()` would return the string `"LOOP"`. If the current instruction is `D=D+1;JLE`, a call to `dest ()`, `comp ()`, and `jump ()` would return the strings `"D"`, `"D+1"`, and `"JLE"`, respectively.

In project 6 you have to implement this API using some high-level programming language. In order to do so, you must be familiar with how this language handles text files, and strings.

Routine	*Arguments*	*Returns*	*Function*
Constructor / initializer	Input file / stream	—	Opens the input file / stream and gets ready to parse it.
`hasMoreLines`	—	boolean	Are there more lines in the input?
`advance`	—	—	Skips over white space and comments, if necessary. Reads the next instruction from the input, and makes it the current instruction. This routine should be called only if `hasMoreLines` is true. Initially there is no current instruction.
`instructionType`	—	`A_INSTRUCTION`, `C_INSTRUCTION`, `L_INSTRUCTION` (constants)	Returns the type of the current instruction: `A_INSTRUCTION` for @*xxx*, where *xxx* is either a decimal number or a symbol. `C_INSTRUCTION` for *dest=comp;jump* `L_INSTRUCTION` for (*xxx*), where *xxx* is a symbol.
`symbol`	—	string	If the current instruction is (*xxx*), returns the symbol *xxx*. If the current instruction is @*xxx*, returns the symbol or decimal *xxx* (as a string). Should be called only if `instructionType` is `A_INSTRUCTION` or `L_INSTRUCTION`.
`dest`	—	string	Returns the symbolic *dest* part of the current *C*-instruction (8 possibilities). Should be called only if `instructionType` is `C_INSTRUCTION`.
`comp`	—	string	Returns the symbolic *comp* part of the current *C*-instruction (28 possibilities). Should be called only if `instructionType` is `C_INSTRUCTION`.
`jump`	—	string	Returns the symbolic *jump* part of the current *C*-instruction (8 possibilities). Should be called only if `instructionType` is `C_INSTRUCTION`.

The Code Module

This module provides services for translating symbolic Hack mnemonics into their binary codes. Specifically, it translates symbolic Hack mnemonics

into their binary codes according to the language specifications (see figure 6.2). Here is the API:

Routine	*Arguments*	*Returns*	*Function*
`dest`	string	3 bits, as a string	Returns the binary code of the *dest* mnemonic.
`comp`	string	7 bits, as a string	Returns the binary code of the *comp* mnemonic.
`jump`	string	3 bits, as a string	Returns the binary code of the *jump* mnemonic.

All the *n*-bit codes are returned as strings of `'0'` and `'1'` characters. For example, a call to `dest ("DM")` returns the string `"011"`, a call to `comp ("A+1")` returns the string `"0110111"`, a call to `comp ("M+1")` returns the string `"1110111"`, a call to `jump ("JNE")` returns the string `"101"`, and so on. All these mnemonic-binary mappings are specified in figure 6.2.

The Hack Assembler

This is the main program that drives the entire assembly process, using the services of the Parser and Code modules. The basic version of the assembler (which we describe now) assumes that the source assembly code contains no symbolic references. This means that (i) in all instructions of type @*xxx*, the *xxx* constants are decimal numbers and not symbols and (ii) the input file contains no label instructions, that is, no instructions of the form (*xxx*).

The basic assembler program can now be described as follows. The program gets the name of the input source file, say, *Prog*, from the command-line argument. It constructs a Parser for parsing the input file *Prog*.asm and creates an output file, *Prog*.hack, into which it will write the translated binary instructions. The program then enters a loop that iterates through the lines (assembly instructions) in the input file and processes them as follows.

For each *C*-instruction, the program uses the Parser and Code services for parsing the instruction into its fields and translating each field into its corresponding binary code. The program then assembles (concatenates) the

translated binary codes into a string consisting of sixteen '0' and '1' characters and writes this string as the next line in the output .hack file.

For each *A*-instruction of type @*xxx*, the program translates *xxx* into its binary representation, creates a string of sixteen '0' and '1' characters, and writes it as the next line in the output .hack file.

We provide no API for this module, inviting you to implement it as you see fit.

6.4.2 Completing the Assembler

The Symbol Table

Since Hack instructions can contain symbolic references, the assembly process must resolve them into actual addresses. The assembler deals with this task using a *symbol table*, designed to create and maintain the correspondence between symbols and their meaning (in Hack's case, RAM and ROM addresses).

A natural means for representing this <*symbol, address*> mapping is any data structure designed to handle <*key, value*> pairs. Every modern high-level programming language features such a ready-made abstraction, typically called a *hash table*, *map*, *dictionary*, among other names. You can either implement the symbol table from scratch or customize one of these data structures. Here is the SymbolTable API:

Routine	***Arguments***	***Returns***	***Function***
Constructor / initializer	—	—	Creates a new empty symbol table.
`addEntry`	`symbol` (string), `address` (int)	—	Adds `<symbol,address>` to the table.
`contains`	`symbol` (string)	boolean	Does the symbol table contain the given `symbol`?
`getAddress`	`symbol` (string)	int	Returns the address associated with the `symbol`.

6.5 Project

Objective: Develop an assembler that translates programs written in Hack assembly language into Hack binary code.

This version of the assembler assumes that the source assembly code is error-free. Error checking, reporting, and handling can be added to later versions of the assembler but are not part of project 6.

Resources: The main tool you need for completing this project is the programming language in which you will implement your assembler. The assembler and CPU emulator supplied in nand2tetris/tools may also come in handy. These tools allow experimenting with a working assembler before setting out to build one yourself. Importantly, the supplied assembler allows comparing its output to the outputs generated by *your* assembler. For more information about these capabilities, refer to the assembler tutorial at www.nand2tetris.org.

Contract: When given to your assembler as a command line argument, a *Prog*.asm file containing a valid Hack assembly language program should be translated into the correct Hack binary code and stored in a file named *Prog*.hack, located in the same folder as the source file (if a file by this name exists, it is overwritten). The output produced by your assembler must be identical to the output produced by the supplied assembler.

Development plan: We suggest building and testing the assembler in two stages. First, write a basic assembler designed to translate programs that contain no symbolic references. Then extend your assembler with symbol-handling capabilities.

Test programs: The first test program has no symbolic references. The remaining test programs come in two versions: *Prog*.asm and *ProgL*.asm, which are with and without symbolic references, respectively.

Add.asm: Adds the constants 2 and 3 and puts the result in R0.

Max.asm: Computes max(R0, R1) and puts the result in R2.

Rect.asm: Draws a rectangle at the top-left corner of the screen. The rectangle is 16 pixels wide and R0 pixels high. Before running this program,

put a nonnegative value in R0.

Pong.asm: A classical single-player arcade game. A ball bounces repeatedly off the screen's edges. The player attempts to hit the ball with a paddle, moving the paddle by pressing the left and right arrow keys. For every successful hit, the player gains a point and the paddle shrinks a little to make the game harder. If the player misses the ball, the game is over. To quit the game, press ESC.

The supplied Pong program was developed using tools that will be presented in part II of the book. In particular, the game software was written in the high-level Jack programming language and translated into the given Pong.asm file by the *Jack compiler*. Although the high-level Pong.jack program is only about three hundred lines of code, the executable Pong application is about twenty thousand lines of binary code, most of which is the Jack operating system. Running this interactive program in the supplied CPU emulator is a slow affair, so don't expect a high-powered Pong game. This slowness is actually a virtue, since it enables tracking the graphical behavior of the program. As you develop the software hierarchy in part II, this game will run much faster.

Testing: Let *Prog*.asm be an assembly Hack program, for example, one of the given test programs. There are essentially two ways to test whether your assembler translates *Prog*.asm correctly. First, you can load the *Prog*.hack file generated by your assembler into the supplied CPU emulator, execute it, and check that it's doing what it's supposed to be doing.

The second testing technique is to compare the code generated by your assembler to the code generated by the supplied assembler. To begin with, rename the file generated by your assembler to *Prog*1.hack. Next, load *Prog*.asm into the supplied assembler, and translate it. If your assembler is working correctly, it follows that *Prog*1.hack must be identical to the *Prog*.hack file produced by the supplied assembler. This comparison can be done by loading *Prog*1.asm as a compare file—see figure 6.3 for the details.

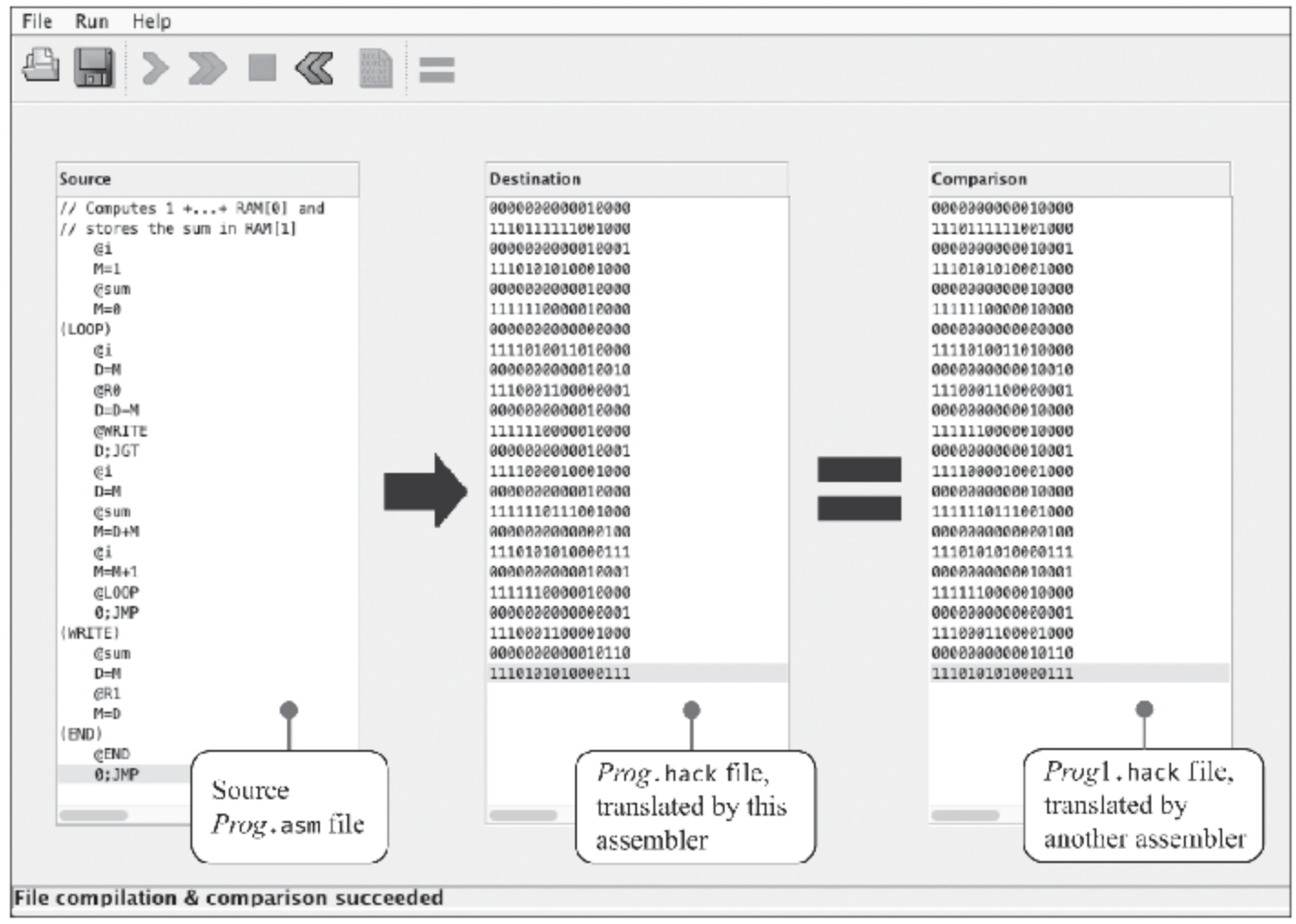


Figure 6.3 Testing the assembler's output using the supplied assembler.

A web-based version of project 6 is available at www.nand2tetris.org.

6.6 Perspective

Like most assemblers, the Hack assembler is a relatively simple translator, dealing mainly with text processing. Naturally, assemblers for richer machine languages are more elaborate. Also, some assemblers feature more sophisticated symbol-handling capabilities not found in Hack. For example, some assemblers support *constant arithmetic* on symbols, like using `base+5` to refer to the fifth memory location after the address referred to by `base`.

Many assemblers are extended to handle *macro-instructions*. A macro-instruction is a sequence of machine instructions that has a name. For example, our assembler can be extended to translate agreed-upon macro-instructions, for example, `D=M[`*addr*`]`, into the two primitive Hack instructions @*addr*, followed by `D=M`. Likewise, the macro-instruction `goto` *addr* can be translated into @*addr*, followed by `0;JMP`, and so on. Such macro-instructions can considerably simplify the writing of assembly programs, at a low translation cost.

It should be noted that machine language programs are rarely written by humans. Rather, they are typically written by compilers. And a compiler—being an automaton—can optionally bypass the symbolic instructions and generate binary machine code directly. That said, an assembler is still a useful program, especially for developers of C/C++ programs who are concerned about efficiency and optimization. By inspecting the symbolic code generated by the compiler, the programmer can improve the high-level code to achieve better performance on the host hardware. When the generated assembly code is considered efficient, it can be translated further by the assembler into the final binary, executable code.

* * *

Congratulations! You've reached the end of part I of the Nand to Tetris journey. If you completed projects 1–6, you have built a general-purpose computer system from first principles. This is a fantastic achievement, and you should feel proud and accomplished.

Alas, the computer is capable of executing only programs written in machine language. In part II of the book we will use this barebone hardware platform as a point of departure and build on top of it a modern software hierarchy. The software will consist of a virtual machine, a compiler for a high-level, object-based programming language, and a basic operating system.

So, whenever you're ready for more adventures, let's move on to part II of our grand journey from Nand to Tetris.

II Software

Any sufficiently advanced technology is indistinguishable from magic.
—Arthur C. Clarke (1962)

To which we add: "and any sufficiently advanced magic is indistinguishable from hard work, behind the scenes." In part I of the book we built the hardware platform of a computer system named Hack, capable of running programs written in the Hack machine language. In part II we will transform this barebone machine into an advanced technology, indistinguishable from magic: a black box that can metamorphose into a chess player, a search engine, a flight simulator, a media streamer, or anything else that tickles your fancy. In order to do so, we'll unfold the elaborate behind-the-scenes software hierarchy that endows computers with the ability to execute programs written in high-level programming languages. In particular, we'll focus on Jack, a simple, Java-like, object-based programming language, described formally in chapter 9. Over the years, Nand to Tetris readers and students have used Jack to develop Tetris, Pong, Snake, Space Invaders, and numerous other games and interactive apps. Being a general-purpose computer, Hack can execute all these programs, and any other program that comes to your mind.

Clearly, the gap between the expressive syntax of high-level programming languages, on the one hand, and the clunky instructions of low-level machine language, on the other, is huge. If you are not convinced, try developing a Tetris game using instructions like `@17` and `M=M+1`. Bridging this gap is what part II of this book is all about. We will build this bridge by developing gradually some of the most powerful and ambitious

programs in applied computer science: a *compiler*, a *virtual machine*, and a basic *operating system*.

Our Jack compiler will be designed to take a Jack program, say Tetris, and produce from it a stream of machine language instructions that, when executed, makes the Hack platform deliver a Tetris game experience. Of course Tetris is just one example: the compiler that you build will be capable of translating *any* given Jack program into machine code that can be executed on the Hack computer. The compiler, whose main tasks consist of *syntax analysis* and *code generation*, will be built in chapters 10 and 11.

As with programming languages like Java and C#, the Jack compiler will be *two-tiered*: the compiler will generate interim *VM code*, designed to run on an abstract *virtual machine*. The VM code will then be compiled further by a separate translator into the Hack machine language. *Virtualization*—one of the most important ideas in applied computer science—comes into play in numerous settings including program compilation, cloud computing, distributed storage, distributed processing, and operating systems. We will devote chapters 7 and 8 to motivating, designing, and building our virtual machine.

Like many other high-level languages, the basic Jack language is surprisingly simple. What turns modern languages into powerful programming systems are *standard libraries* providing mathematical functions, string processing, memory management, graphics drawing, user interaction handling, and more. Taken together, these standard libraries form a basic *operating system* (OS) which, in the Jack framework, is packaged as Jack's *standard class library*. This basic OS, designed to bridge many gaps between the high-level Jack language and the low-level Hack platform, will be developed in Jack itself. You may be wondering how software that is supposed to enable a programming language can be developed in this very same language. We'll deal with this challenge by following a development strategy known as *bootstrapping*, similar to how the Unix OS was developed using the C language.

The construction of the OS will give us an opportunity to present elegant algorithms and classical data structures that are typically used to manage hardware resources and peripheral devices. We will then implement these algorithms in Jack, extending the language's capabilities one step at a time. As you go through the chapters of part II, you will deal with the OS from

several different perspectives. In chapter 9, acting as an *application programmer*, you will develop a Jack app and use the OS services abstractly, from a high-level client perspective. In chapters 10 and 11, when building the Jack compiler, you will use the OS services as a low-level client, for example, for various memory management services required by the compiler. In chapter 12 you will finally don the hat of the OS developer and implement all these system services yourself.

II.1 A Taste of Jack Programming

Before delving into all these exciting projects, we'll give a brief and informal introduction of the Jack language. This will be done using two examples, starting with Hello World. We will use this example to demonstrate that even the most trivial high-level program has much more to it than meets the eye. We will then present a simple program that illustrates the object-based capabilities of the Jack language. Once we get a programmer-oriented taste of the high-level Jack language, we will be prepared to start the journey of realizing the language by building a virtual machine, a compiler, and an operating system.

Hello World, again: We began this book with the iconic Hello World program that learners often encounter as the first thing in introductory programming courses. Here is this trivial program once again, written in the Jack programming language:

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

Let's discuss some of the implicit assumptions that we normally make when presented with such programs. The first magic that we take for granted is that a sequence characters, say, printString ("Hello World"), can cause the computer to actually display something on the screen. How does the computer figure out *what* to do? And even if the computer knew what to do,

how will it actually do it? As we saw in part I of the book, the screen is a grid of pixels. If we want to display `H` on the screen, we have to turn on and off a carefully selected subset of pixels that, taken together, render the desired letter's image on the screen. Of course this is just the beginning. What about displaying this `H` legibly on screens that have different sizes and resolutions? And what about dealing with *while* and *for* loops, *arrays*, *objects*, *methods*, *classes*, and all the other goodies that high-level programmers are trained to use without ever thinking about how they work?

Indeed, the beauty of high-level programming languages, and that of well-designed abstractions in general, is that they permit using them in a state of blissful ignorance. Application programmers are in fact encouraged to view the language as a black box abstraction, without paying any attention to how it is actually implemented. All you need is a good tutorial, a few code examples, and off you go.

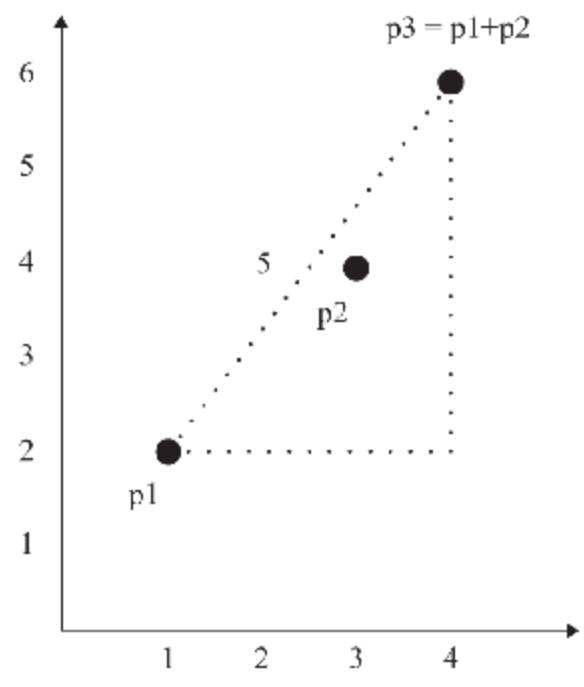
Clearly though, at one point or another, *someone* must implement this language abstraction. Someone must develop, once and for all, the ability to efficiently compute square roots when the application programmer blissfully says `sqrt(1764)`, to elicit a number from the user when the programmer happily says `x=readInt()`, to find and carve out an available memory block when the programmer nonchalantly creates an object using `new`, and to perform transparently all the other abstract services that programmers expect to get without ever thinking about them. So, who are the good souls who turn high-level programming into an advanced technology indistinguishable from magic? They are the software wizards who develop *compilers*, *virtual machines*, and *operating systems*. And that's precisely what *you* will do in the forthcoming chapters.

You may be wondering why you have to bother about this elusive behind-the-scenes scene. Didn't we just say that you can use high-level languages without worrying about how they work? There are at least two reasons why. First, the more you delve into low-level system internals, the more sophisticated high-level programmer you become. In particular, you learn how to write high-level code that exploits the hardware and the OS cleverly and efficiently and how to avoid baffling bugs like memory leaks.

Second, by getting your hands dirty and developing the system internals yourself, you will discover some of the most beautiful and powerful algorithms and data structures in applied computer science. Importantly, the

ideas and techniques that will unfold in part II are not limited to compilers and operating systems. Rather, they are the building blocks of numerous software systems and applications that will accompany you throughout your career.

The PointDemo program: Suppose we want to represent and manipulate *points* on a plane. Figure II.1 shows two such points, $p1$ and $p2$, and a third point, $p3$, resulting from the vector addition $p_3 = p_1 + p_2 = (1,2) + (3,4) = (4,6)$. The figure also depicts the *Euclidean distance* between $p1$ and $p3$, which can be computed using the Pythagorean theorem. The code in the Main class illustrates how such algebraic manipulations can be done using the object-based Jack language.




```
/** Demo: working with Point objects.
    File name:  Main.jack. */
class Main {
  function main() {
     var Point p1, p2, p3;
     let p1 = Point.new(1,2);
     let p2 = Point.new(3,4);
     let p3 = p1.plus(p2);
     do p3.print();
     do Output.printInt(p1.distance(p2));
     do Output.printInt(getPointCount());
     // Should print (4,6), 5, and 3.
     return;
  }
}
```

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

You may wonder why Jack uses keywords like var, let, and do. For now, we advise you not to dwell on syntactic details. Instead, let's focus on the big picture and proceed to review how the Jack language can be used to implement the Point abstract data type (figure II.2).

```
/** Represents a two-dimensional point.
    File name: Point.jack. */
class Point {

   // The coordinates of this point:
   field int x, y

   // The number of Point objects constructed so far:
   static int pointCount;

   /** Constructs a two-dimensional point and
       initializes it with the given coordinates. */
   constructor Point new(int ax, int ay) {
        let x = ax;
        let y = ay;
        let pointCount = pointCount + 1;
        return this;
   }

   /** Returns the x coordinate of this point. */
   method int getx() {return x;}

   /** Returns the y coordinate of this point. */
   method int gety() {return y;}

   /** Returns the number of Points constructed so far. */
   function int getPointCount() {
      return pointCount;
   }

   // Class declaration continues on top right.
```

```
   /** Returns a point which is this point plus
       the other point. */
   method Point plus(Point other) {
      return Point.new(x + other.getx(),
                       y + other.gety());
   }

   /** Returns the Euclidean distance between
       this and the other point. */
   method int distance(Point other) {
      var int dx, dy;
      let dx = x - other.getx();
      let dy = y - other.gety();
      return Math.sqrt((dx*dx) + (dy*dy));
   }

   /** Prints this point, as "(x,y)" */
   method void print() {
      do Output.printString("(");
      do Output.printInt(x);
      do Output.printString(",");
      do Output.printInt(y);
      do Output.printString(")");
      return;
   }

} // End of Point class declaration.
```

Figure II.2 Jack implementation of the Point abstraction.

The code shown in figure II.2 illustrates that a Jack class (of which Main and Point are two examples) is a collection of one or more *subroutines*, each being a *constructor*, *method*, or *function*. *Constructors* are subroutines that create new objects, *methods* are subroutines that operate on the current object, and *functions* are subroutines that operate on no particular object. (Object-oriented design purists may frown about mixing methods and functions in the same class; we are doing it here for illustrative purposes).

The remainder of this section is an informal overview of the Main and the Point classes. Our goal is to give a taste of Jack programming, deferring a complete language description to chapter 9. So, allowing ourselves the luxury of focusing on essence only, let's get started. The Main.main function begins by declaring three *object variables* (also known as *references*, or *pointers*), designed to refer to instances of the Point class. It then goes on to construct two Point objects, and assigns the p1 and p2 variables to them. Next, it calls the plus method, and assigns p3 to the Point object returned by that method. The rest of the Main.main function prints some results.

The Point class begins by declaring that every Point object is characterized by two *field* variables (also known as *properties*, or *instance variables*). It then declares a *static variable*, that is, a class-level variable associated with no particular object. The class constructor sets up the field values of the

newly created object and increments the number of instances derived from this class so far. Note that a Jack constructor must explicitly return the memory address of the newly created object, which, according to the language rules, is denoted this.

You may wonder why the result of the square root computed by the distance method is stored in an int variable—clearly a real-valued data type like float would make more sense. The reason for this peculiarity is simple: the Jack language features only three primitive data types: int, boolean, and char. Other data types can be implemented at will using classes, as we'll do in chapters 9 and 12.

The operating system: The Main and Point classes use three OS functions: Output.printInt, Output.printString, and Math.sqrt. Like other modern high-level languages, the Jack language is augmented by a set of *standard classes* that provide commonly used OS services (the complete OS API is given in appendix 6). We will have much more to say about the OS services in chapter 9, where we'll use them in the context of Jack programming, as well as in chapter 12, where we'll build the OS.

In addition to calling OS services for their effects directly from Jack programs, the OS comes to play in other, less obvious ways. For example, consider the new operation, used to construct objects in object-oriented languages. How does the compiler know where in the host RAM to put the newly constructed object? Well, it doesn't. An OS routine is called to figure it out. When we build the OS in chapter 12, you will implement, among many other things, a typical run-time memory management system. You will then learn, hands-on, how this system interacts with the hardware, from the one end, and with compilers, from the other, in order to allocate and reclaim RAM space cleverly and efficiently. This is just one example that illustrates how the OS bridges gaps between high-level applications and the host hardware platform.

II.2 Program Compilation

A high-level program is a symbolic abstraction that means nothing to the underlying hardware. Before executing a program, the high-level code must

be translated into machine language. This translation process is called *compilation*, and the program that carries it out is called a *compiler*. Writing a compiler that translates high-level programs into low-level machine instructions is a worthy challenge. Some languages, for example, Java and C#, deal with this challenge by employing an elegant *two-tier* compilation model. First, the source program is translated into an interim, abstract VM code (called *bytecode* in Java and Python and *Intermediate Language* in C#/.NET). Next, using a completely separate and independent process, the VM code can be translated further into the machine language of any target hardware platform.

This modularity is at least one reason why Java became such a dominant programming language. Taking a historical perspective, Java can be viewed as a powerful object-oriented language whose two-tier compilation model was the right thing in the right time, just when computers began evolving from a few predictable processor/OS platforms into a bewildering hodgepodge of numerous PCs, cell phones, mobile devices, and Internet of Things devices, all connected by a global network. Writing high-level programs that can execute on any one of these host platforms is a daunting challenge. One way to streamline this distributed, multi-vendor ecosystem (from a compilation perspective) is to base it on some overarching, agreed-upon virtual machine architecture. Acting as a common, intermediate run-time environment, the VM approach allows developers to write high-level programs that run almost as is on many different hardware platforms, each equipped with its own VM implementation. We will have much more to say about the enabling power of this modularity as part II unfolds.

The road ahead: In the remainder of the book we'll apply ourselves to developing all the exciting software technologies mentioned above. Our ultimate goal is creating an infrastructure for turning high-level programs—*any* program—into executable code. The road map is shown in figure II.3.

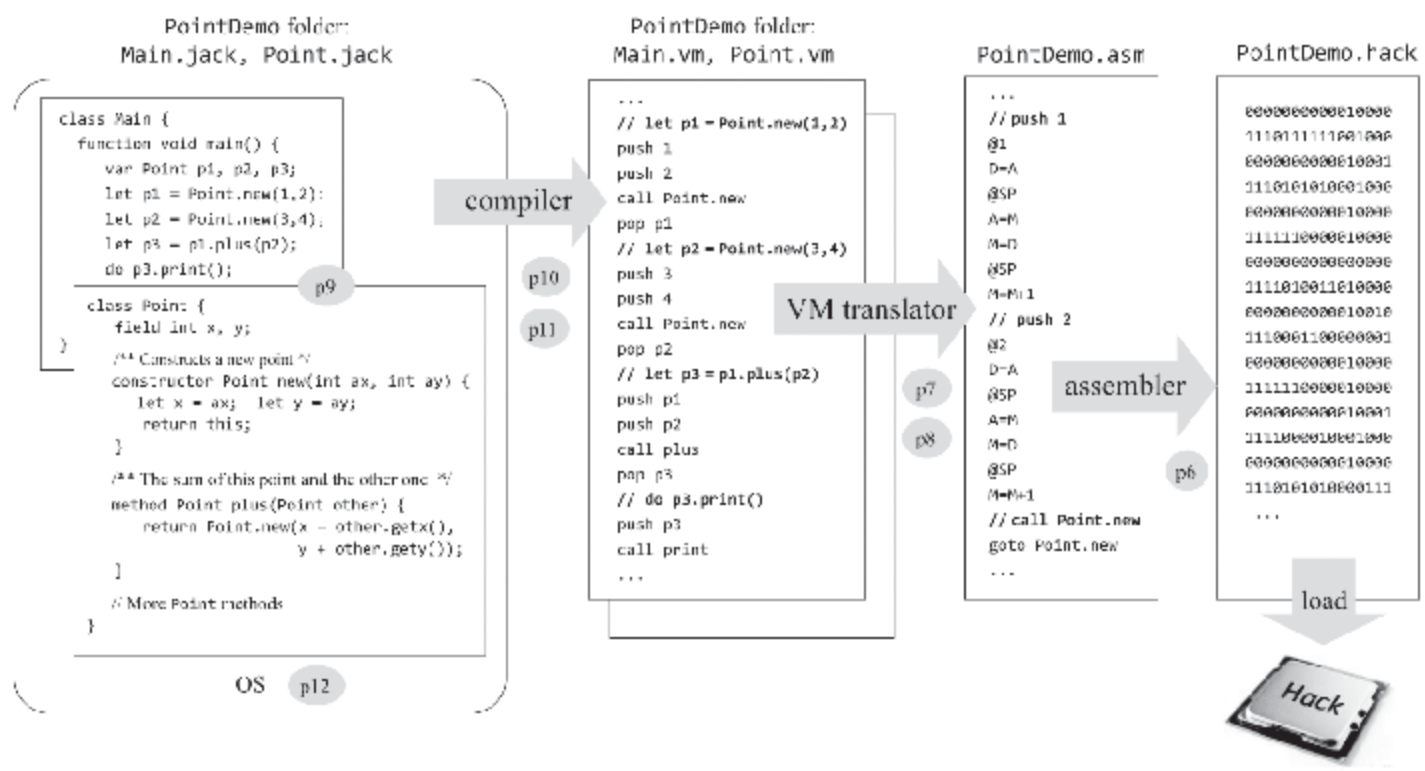


Figure II.3 Road map of part II (the assembler belongs to part I and is shown here for completeness). The road map describes a translation hierarchy, from a high-level, object-based, multi-class program to VM code, to assembly code, to executable binary code. The numbered circles stand for the projects 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.

Following the Nand to Tetris spirit, we'll pursue the part II road map from the bottom up. To get started, we assume that we have a hardware platform equipped with an assembly language. In chapters 7–8 we'll present a virtual machine architecture and a VM language, and we'll implement this abstraction by developing a *VM translator* that translates VM programs into Hack assembly programs. In chapter 9 we'll present the Jack high-level language and use it to develop a simple computer game. This way, you'll get acquainted with the Jack language and operating system before setting out to build them. In chapters 10–11 we'll develop the Jack compiler, and in chapter 12 we'll build the operating system.

So, let's roll up our sleeves and get to work!

7 Virtual Machine I: Processing

Programmers are creators of universes for which they alone are responsible. Universes of virtually unlimited complexity can be created in the form of computer programs.

—Joseph Weizenbaum, *Computer Power and Human Reason* (1974)

This chapter describes the first steps toward building a compiler for a typical object-based, high-level language. We approach this challenge in two major stages, each spanning two chapters. In chapters 10–11 we'll describe the construction of a *compiler*, designed to translate high-level programs into *intermediate code*; in chapters 7–8 we describe the construction of a follow-up *translator*, designed to translate the intermediate code into the machine language of a target hardware platform. As the chapter numbers suggest, we will pursue this substantial development from the bottom up, starting with the translator.

The intermediate code that lies at the core of this compilation model is designed to run on an abstract computer, called a *virtual machine*, or VM. There are several reasons why this two-tier compilation model makes sense, compared to traditional compilers that translate high-level programs directly to machine language. One benefit is cross-platform compatibility: since the virtual machine may be realized with relative ease on many hardware platforms, the same VM code can run as is on any device equipped with such a VM implementation. That's one reason Java became a dominant language for developing apps for mobile devices, which are characterized by many different processor/OS combinations. The VM can be implemented on the target devices by using software interpreters, or special-purpose hardware, or by translating the VM programs into the device's machine language. The latter implementation approach is taken by

Java, Scala, C#, and Python, as well as by the Jack language developed in Nand to Tetris.

This chapter presents a typical VM architecture and VM language, conceptually similar to the Java Virtual Machine (JVM) and bytecode, respectively. As usual in Nand to Tetris, the virtual machine will be presented from two perspectives. First, we will motivate and specify the VM abstraction, describing what the VM is designed to do. Next, we will describe a proposed implementation of the VM over the Hack platform. Our implementation entails writing a program called a *VM translator* that translates VM code into Hack assembly code.

The VM language that we'll present consists of arithmetic-logical commands, memory access commands called *push* and *pop*, branching commands, and function call-and-return commands. We split the discussion and implementation of this language into two parts, each covered in a separate chapter and project. In this chapter we build a basic VM translator which implements the VM's arithmetic-logical and push/pop commands. In the next chapter we extend the basic translator to handle branching and function commands. The result will be a full-scale virtual machine implementation that will serve as the back end of the compiler that we will build in chapters 10–11.

The virtual machine that will emerge from this effort illustrates several important ideas and techniques. First, the notion of having one computing framework emulate another is a fundamental idea in computer science, tracing back to Alan Turing in the 1930s. Today, the virtual machine model is the centerpiece of several mainstream programming environments, including Java, .NET, and Python. The best way to gain an intimate inside view of how these programming environments work is to build a simple version of their VM cores, as we do here.

Another important theme in this chapter is *stack processing*. The *stack* is a fundamental and elegant data structure that comes to play in numerous computer systems, algorithms, and applications. Since the VM presented in this chapter is stack-based, it provides a working example of this remarkably versatile and powerful data structure.

7.1 The Virtual Machine Paradigm

Before a high-level program can run on a target computer, it must be translated into the computer's machine language. Traditionally, a separate compiler was developed specifically for any given pair of high-level language and low-level machine language. Over the years, the reality of many high-level languages, on the one hand, and many processors and instruction sets, on the other, has led to a proliferation of many different compilers, each depending on every detail of both its source and target languages. One way to decouple this dependency is to break the overall compilation process into two nearly separate stages. In the first stage, the high-level code is parsed and translated into intermediate and abstract processing steps—steps that are neither high nor low. In the second stage, the intermediate steps are translated further into the low-level machine language of the target hardware.

This decomposition is very appealing from a software engineering perspective. First, note that the first translation stage depends only on the specifics of the source high-level language, and the second stage only on the specifics of the target low-level machine language. Of course, the interface between the two translation stages—the exact definition of the intermediate processing steps—must be carefully designed and optimized. At some point in the evolution of program translation solutions, compiler developers concluded that this intermediate interface is sufficiently important to merit its own definition as a standalone language designed to run on an abstract machine. Specifically, one can describe a *virtual machine* whose commands realize the intermediate processing steps into which high-level commands are translated. The compiler that was formerly a single monolithic program is now split into two separate and much simpler programs. The first program, still termed *compiler*, translates the high-level code into intermediate VM commands; the second program, called *VM translator*, translates the VM commands further into the machine instructions of the target hardware platform. Figure 7.1 outlines how this two-tiered compilation framework has contributed to the cross-platform portability of Java programs.

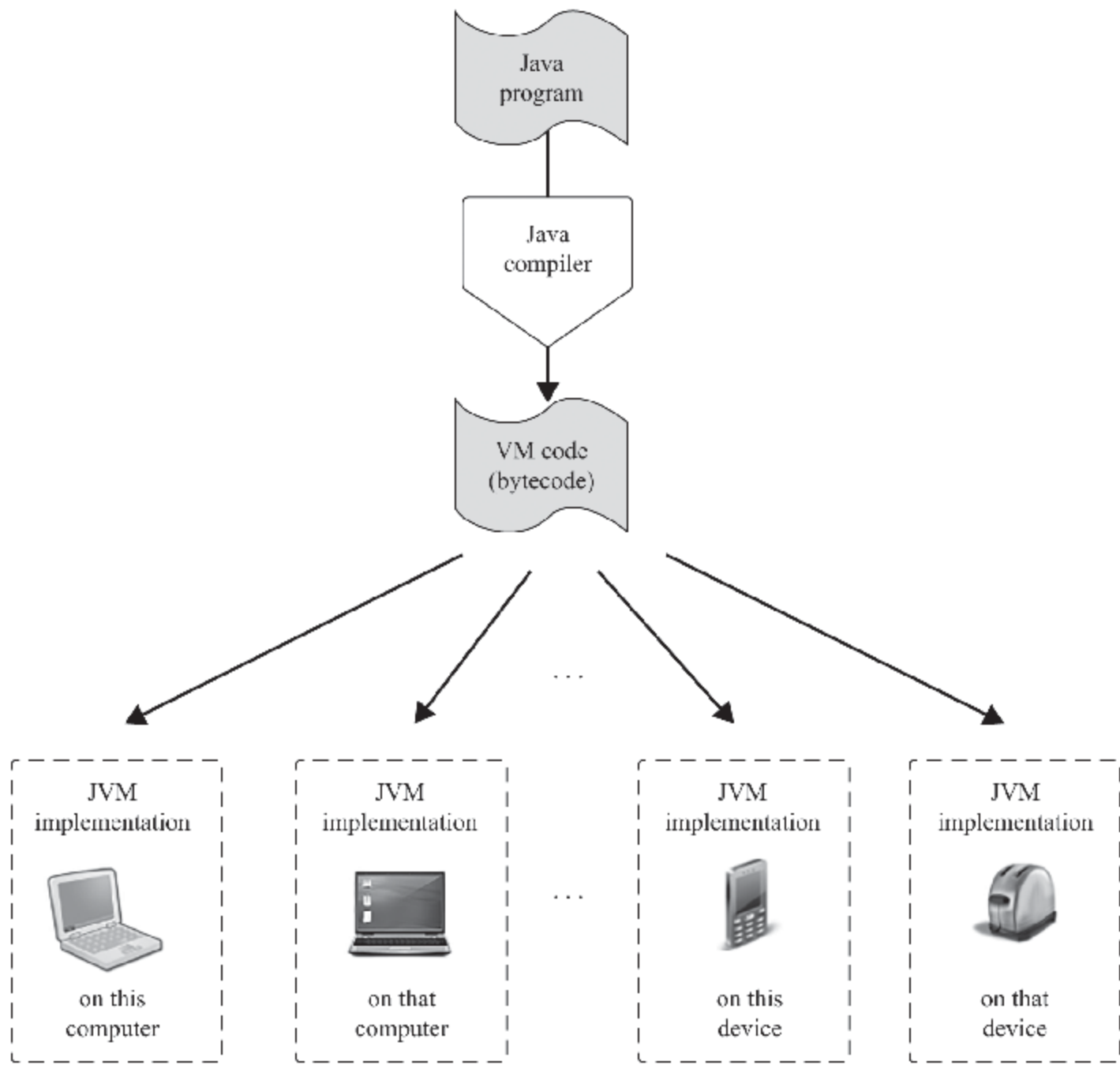


Figure 7.1 The virtual machine framework, using Java as an example. High-level programs are compiled into intermediate VM code. The same VM code can be shipped to, and executed on, any hardware platform equipped with a suitable *JVM implementation*. These VM implementations are typically realized as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

The virtual machine framework entails many practical benefits. When a vendor introduces to the market a new digital device—say, a cell phone—it can develop for it a JVM implementation, known as JRE (Java Runtime Environment), with relative ease. This client-side enabling infrastrucutre immediately endows the device with a huge base of available Java software. And, in a world like .NET, in which several high-level languages are made to compile into the same intermediate VM language, compilers for different languages can share the same VM back end, allowing usage of common software libraries and language interoperability.

The price paid for the elegance and power of the VM framework is reduced efficiency. Naturally, a two-tier translation process results, ultimately, in generating machine code that is more verbose and

cumbersome than the code produced by direct compilation. However, as processors become faster and VM implementations more optimized, the degraded efficiency is hardly noticeable in most applications. Of course, there will always be high-performance applications and embedded systems that will continue to demand the efficient code generated by single-tier compilers of language like C and C++. That said, modern versions of C++ feature both classical one-tier compilers and two-tier VM-based compilers.

7.2 Stack Machine

The design of an effective VM language seeks to strike a convenient balance between high-level programming languages, on the one hand, and a great variety of low-level machine languages, on the other. Thus, the desired VM language should satisfy several requirements coming both from above and from below. First, the language should have a reasonable expressive power. We achieve this by designing a VM language that features arithmetic-logical commands, push/pop commands, branching commands, and function commands. These VM commands should be sufficiently "high" so that the VM code generated by the compiler will be reasonably elegant and well structured. At the same time, the VM commands should be sufficiently "low" so that the machine code generated from them by VM translators will be tight and efficient. Said otherwise, we have to make sure that the translation gaps between the high-level and the VM level, on the one hand, and the VM level and the machine level, on the other, will not be wide. One way to satisfy these somewhat conflicting requirements is to base the interim VM language on an abstract architecture called a *stack machine*.

Before going on, we'd like to issue a plea for patience. The relationship between the stack machine that we now turn to describe and the compiler that we'll introduce later in the book is subtle. Therefore, we advise readers to allow themselves to savor the intrinsic beauty of the stack machine abstraction without worrying about its ultimate purpose in every step of the way. The full practical power of this remarkable abstraction will carry its weight only toward the end of the *next* chapter; for now, suffice it to say

that any program, written in any high-level programming language, can be translated into a sequence of operations on a stack.

7.2.1 Push and Pop

The centerpiece of the stack machine model is an abstract data structure called a *stack*. A stack is a sequential storage space that grows and shrinks as needed. The stack supports various operations, the two key ones being push and pop. The push operation adds a value to the top of the stack, like adding a plate to the top of a stack of plates. The pop operation removes the stack's top value; the value that was just before it becomes the top stack element. See figure 7.2 for an example. Note that the push/pop logic results in a *last-in-first-out* (LIFO) access logic: the popped value is always the last one that was pushed onto the stack. As it turns out, this access logic lends itself perfectly to program translation and execution purposes, but this insight will take two chapters to unfold.

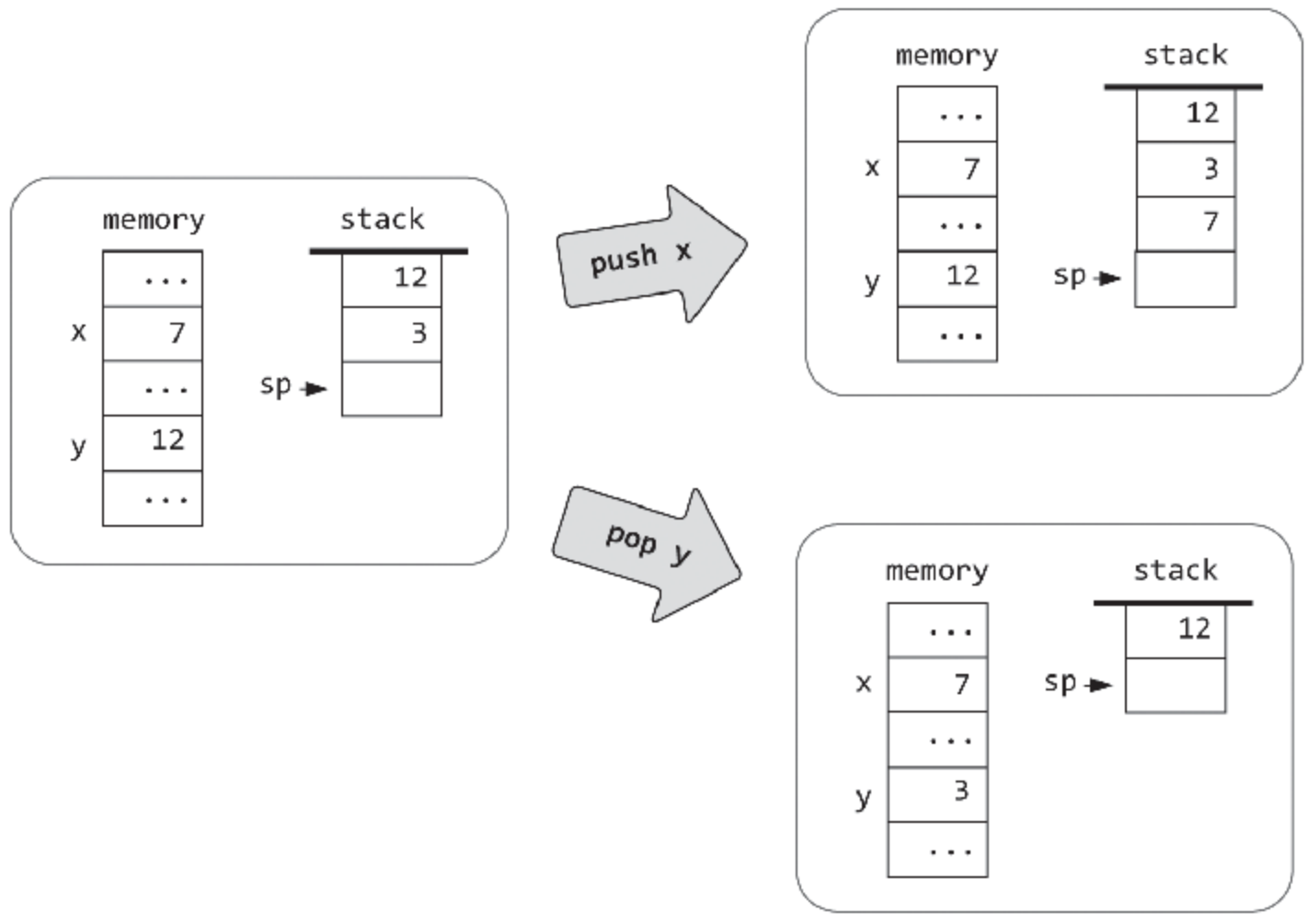
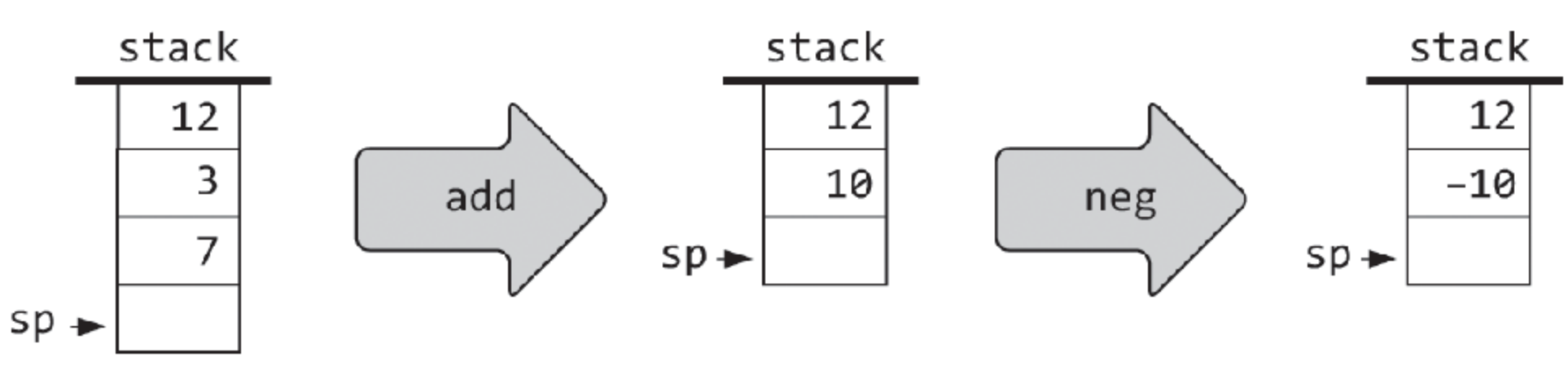


Figure 7.2 Stack processing example, illustrating the two elementary operations push and pop. The setting consists of two data structures: a RAM-like memory segment and a stack. Following convention, the stack is drawn as if it grows downward. The location just following the stack's top value is referred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

As figure 7.2 shows, our VM abstraction includes a *stack*, as well as a sequential, RAM-like memory segment. Observe that stack access is different from conventional memory access. First, the stack is accessible only from its top, whereas regular memory allows direct and indexed access to any value in the memory. Second, *reading* a value from the stack is a lossy operation: only the top value can be read, and the only way to access it entails *removing* it from the stack (although some stack models also provide a *peek* operation, which allows reading without removing). In contrast, the act of reading a value from a regular memory leaves no impact on the memory's state. Lastly, *writing* to the stack entails adding a value onto the stack's top without changing the other values in the stack. In contrast, writing an item into a regular memory location is a lossy operation, since it overrides the location's previous value.

7.2.2 Stack Arithmetic

Consider the generic operation *x op y*, where the operator *op* is applied to the operands *x* and *y*, for example, `7 + 5`, `3 – 8`, and so on. In a stack machine, each *x op y* operation is carried out as follows: first, the operands *x* and *y* are popped off the top of the stack; next, the value of *x op y* is computed; finally, the computed value is pushed onto the top of the stack. Likewise, the unary operation *op x* is realized by popping *x* off the top of the stack, computing the value of *op x*, and finally pushing this value onto the top of the stack. For example, here is how addition and negation are handled:



Stack-based evaluation of general arithmetic expressions is an extension of the same idea. For example, consider the expression `d = (2 – x) + (y + 9)`, taken from some high-level program. The stack-based evaluation of this

expression is shown in figure 7.3a. Stack-based evaluation of logical expressions, shown in figure 7.3b, follows the same scheme.

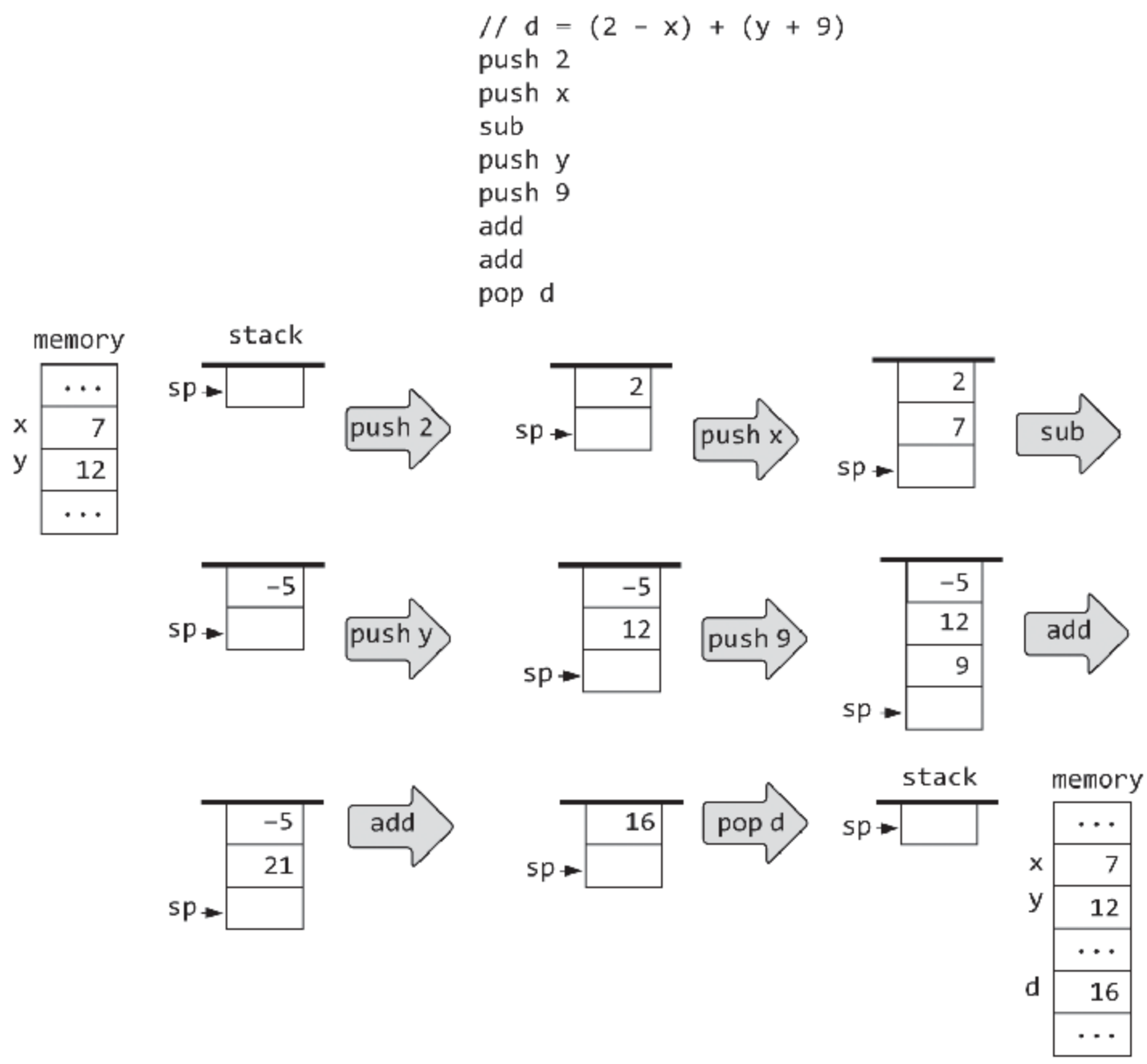


Figure 7.3a Stack-based evaluation of arithmetic expressions.

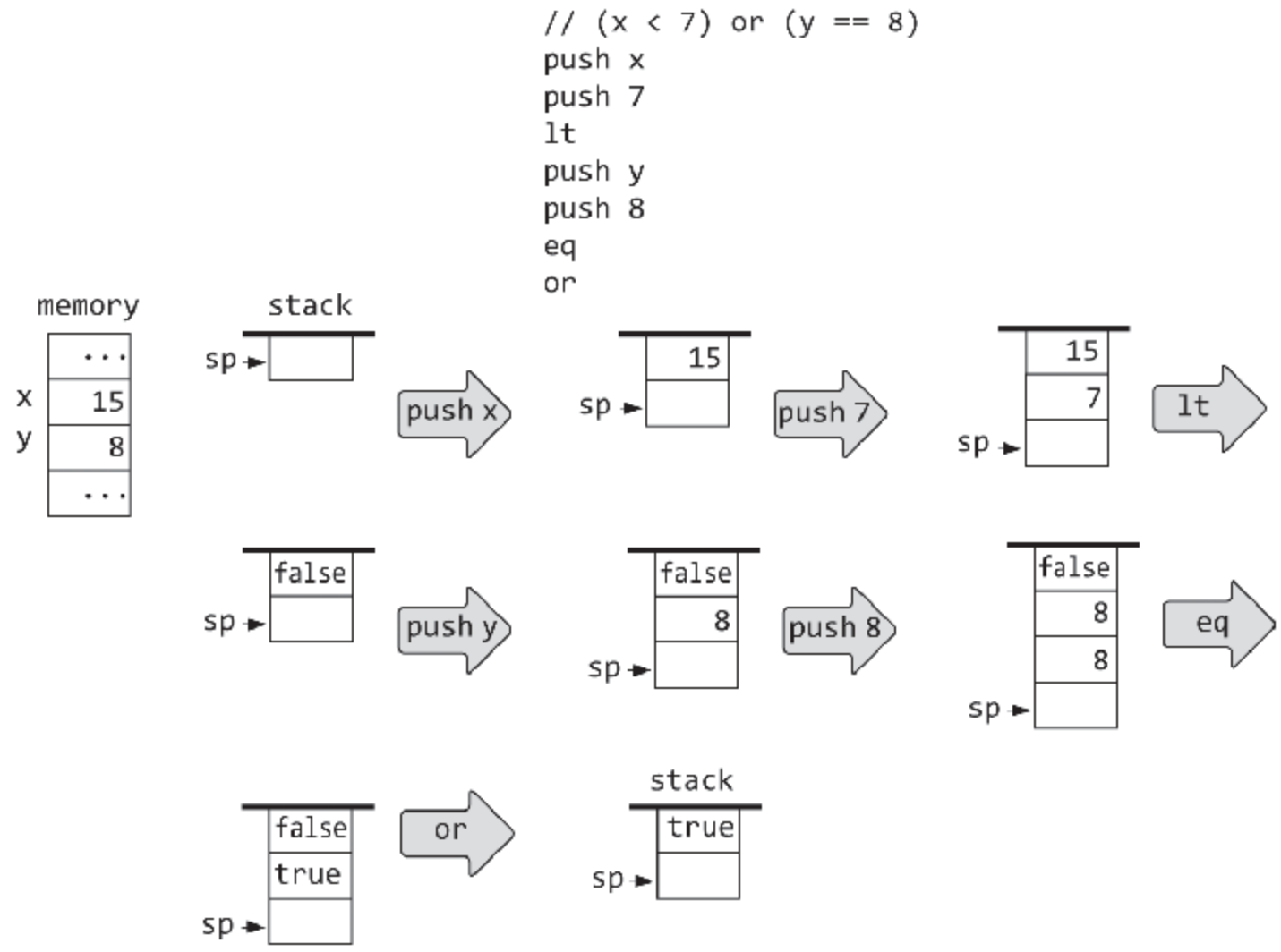


Figure 7.3b Stack-based evaluation of logical expressions.

Note that from the stack's perspective, each arithmetic or logical operation has the net impact of replacing the operation's operands with the operation's result, without affecting the rest of the stack. This is similar to how humans perform mental arithmetic, using our short-term memory. For example, how to compute the value of $3 \times (11 + 7) - 6$? We start by mentally popping 11 and 7 off the expression and calculating $11 + 7$. We then plug the resulting value back into the expression, yielding $3 \times 18 - 6$. The net effect is that $(11 + 7)$has been replaced by 18, and the rest of the expression remains the same as before. We can now proceed to perform similar pop-compute-and-push mental operations until the expression is reduced to a single value.

These examples illustrate an important virtue of stack machines: any arithmetic and logical expression—no matter how complex—can be systematically converted into, and evaluated by, a sequence of simple operations on a stack. Therefore, one can write a *compiler* that translates high-level arithmetic and logical expressions into sequences of stack commands, as indeed we'll do in chapters 10–11. Once the high-level expressions have been reduced into stack commands, we can proceed to evaluate them using a stack machine implementation.

7.2.3 Virtual Memory Segments

So far in our stack processing examples, the push/pop commands were illustrated conceptually, using the syntax push *x* and pop *y*, where *x* and *y* referred abstractly to arbitrary memory locations. We now turn to give a formal description of our push and pop commands.

High-level languages feature symbolic variables like x, y, sum, count, and so on. If the language is object-based, each such variable can be a class-level *static* variable, an instance-level *field* of an object, or a method-level *local* or *argument* variable. In virtual machines like Java's JVM and in our own VM model, there are no symbolic variables. Instead, variables are represented as entries in virtual memory segments that have names like static, this, local, and argument. In particular, as we'll see in later chapters, the compiler maps the first, second, third, … static variable found in the high-level program onto static 0, static 1, static 2, and so on. The other variable kinds are mapped similarly on the segments this, local, and argument. For example, if the local variable x and the field y have been mapped on local 1 and this 3, respectively, then a high-level statement like let x = y will be translated by the compiler into push this 3 followed by pop local 1. Altogether, our VM model features eight memory segments, whose names and roles are listed in figure 7.4.

Segment	*Role*
`argument`	Represents the function's arguments
`local`	Represents the function's local variables
`static`	Represents the static variables seen by the function
`constant`	Represents the constant values 0,1,2,3, …, 32767
`this`	Described in later chapters
`that`	Described in later chapters
`pointer`	Described in later chapters
`temp`	Described in later chapters

Figure 7.4 Virtual memory segments.

We note in passing that developers of VM implementations need not care about how the compiler maps symbolic variables on the virtual memory segments. We will deal with these issues at length when we develop the compiler in chapters 10–11. For now, we observe that VM commands access all the virtual memory segments in exactly the same way: by using the *segment name* followed by a nonnegative *index*.

7.3 VM Specification, Part I

Our VM model is *stack-based*: all the VM operations take their operands from, and store their results on, the *stack*. There is only one data type: a signed 16-bit integer. A VM *program* is a sequence of VM commands that fall into four categories:

- *Push / pop commands* transfer data between the stack and memory segments.
- *Arithmetic-logical commands* perform arithmetic and logical operations.
- *Branching commands* facilitate conditional and unconditional branching operations.
- *Function commands* facilitate function call-and-return operations.

The specification and implementation of these commands span two chapters. In this chapter we focus on the *arithmetic-logical* and *push/pop* commands. In the next chapter we complete the specification of the remaining commands.

Comments and white space: Lines beginning with // are considered comments and are ignored. Blank lines are permitted and are ignored.

Push / Pop Commands

push *segment index*	Pushes the value of *segment*[*index*] onto the stack, where *segment* is `argument`, `local`, `static`, `constant`, `this`, `that`, `pointer`, or `temp` and *index* is a nonnegative integer.
pop *segment index*	Pops the top stack value and stores it in *segment*[*index*], where *segment* is `argument`, `local`, `static`, `this`, `that`, `pointer`, or `temp` and *index* is a nonnegative integer.

Arithmetic-Logical Commands

- *Arithmetic commands*: add, sub, neg
- *Comparison commands*: eq, gt, lt
- *Logical commands*: and, or, not

The commands add, sub, eq, gt, lt, and, and or have two implicit operands. To execute each one of them, the VM implementation pops two values off the stack's top, computes the stated function on them, and pushes the resulting value back onto the stack (by *implicit operand* we mean that the operand is not part of the command syntax: since the command is designed to always operate on the two top stack values, there is no need to specify them). The remaining neg and not commands have one implicit operand and work the same way. Figure 7.5 gives the details.

Command	*Computes*	*Comment*	
add	$x + y$	integer addition	(two's complement)
sub	$x - y$	integer subtraction	(two's complement)
neg	$-y$	arithmetic negation	(two's complement)
eq	$x == y$	equality	
gt	$x > y$	greater than	
lt	$x < y$	less than	
and	x And y	bit-wise And	
or	x Or y	bit-wise Or	
not	Not y	bit-wise Not	

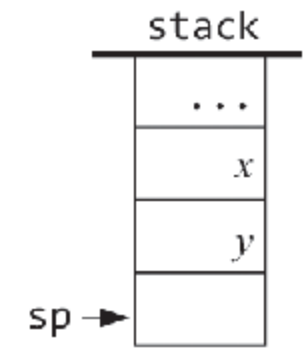


Figure 7.5 The arithmetic-logical commands of the VM language.

7.4 Implementation

The virtual machine described up to this point is an abstraction. If we want to use this VM for real, it must be implemented on some real, host platform. There are several implementation options, of which we'll describe one: a VM *translator*. The VM translator is a program that translates VM commands into machine language instructions. Writing such a program entails two main tasks. First, we have to decide how to represent the stack and the virtual memory segments on the target platform. Second, we have to translate each VM command into a sequence of low-level instructions that can execute on the target platform.

For example, suppose that the target platform is a typical von Neumann machine. In this case, we can represent the VM's stack using a designated memory block in the host RAM. The lower end of this RAM block will be a fixed base address, and its higher end will change as the stack grows and shrinks. Thus, given a fixed stackBase address, we can manage the stack by keeping track of a single variable: a *stack pointer*, or SP, which holds the address of the RAM entry just following the stack's topmost value. To initialize the stack, we set SP to stackBase. From this point onward, each push x command can be implemented by the pseudocode operations RAM[SP] = x followed by SP++, and each pop x command can be implemented by SP—— followed by x = RAM[SP].

Let us assume that the host platform is the Hack computer and that we decide to anchor the stack base at address 256 in the Hack RAM. In that case, the VM translator can start by generating assembly code that realizes SP = 256, that is, @256, D=A, @SP, M=D. From this point onward, the VM translator can handle each push x and pop x command by generating assembly code that realizes the operations RAM[SP++] = x and x = RAM[——SP], respectively.

With that in mind, let us now consider the implementation of the VM arithmetic-logical commands add, sub, neg, and so on. Conveniently, all these commands share exactly the same access logic: popping the command's operands off the stack, carrying out a simple calculation, and pushing the result onto the stack. This means that once we figure out how to implement the VM's push and pop commands, the implementation of the VM's arithmetic-logical commands will follow straightforwardly.

7.4.1 Standard VM Mapping on the Hack Platform, Part I

So far in this chapter, we have made no assumption whatsoever about the target platform on which our virtual machine will be implemented: everything was described abstractly. When it comes to virtual machines, this platform independence is the whole point: you don't want to commit your abstract machine to any particular hardware platform, precisely because you want it to potentially run on *any* platform, including those that were not yet built or invented.

Of course, at some point we have to implement the VM abstraction on a particular hardware platform (for example, on one of the target platforms mentioned in figure 7.1). How should we go about it? In principle, we can do whatever we please, as long as we end up realizing the VM abstraction faithfully and efficiently. Nevertheless, VM architects normally publish basic implementation guidelines, known as *standard mappings*, for different hardware platforms. With that in mind, the remainder of this section specifies the standard mapping of our VM abstraction on the Hack computer. In what follows, we use the terms *VM implementation* and *VM translator* interchangeably.

VM program: The complete definition of a VM *program* will be presented in the next chapter. For now, we view a VM program as a sequence of VM commands stored in a text file named *FileName*.vm (the first character of the file name must be an uppercase letter, and the extension must be vm). The VM translator should read each line in the file, treat it as a VM command, and translate it into one or more instructions written in the Hack language. The resulting output—a sequence of Hack assembly instructions—should be stored in a text file named *FileName*.asm (the file name is identical to that of the source file; the extension must be asm). When translated by the Hack assembler into binary code or run as is on a Hack CPU emulator, this .asm file should perform the semantics mandated by the source VM program.

Data type: The VM abstraction has only one data type: a signed integer. This type is implemented on the Hack platform as a two's complement 16-bit value. The VM Boolean values *true* and *false* are represented as –1 and 0, respectively.

RAM usage: The host Hack RAM consists of 32K 16-bit words. VM implementations should use the top of this address space as follows:

RAM addresses	*Usage*
0 – 15	Sixteen virtual registers, usage described below
16 – 255	Static variables
256 – 2047	Stack

Recall that according to the *Hack machine language specification* (chapter 6), RAM addresses 0 to 4 can be referred to using the symbols SP, LCL, ARG, THIS, and THAT. This convention was introduced into the assembly language with foresight, to help developers of VM implementations write readable code. The expected use of these addresses in the VM implementation is as follows:

Name	*Location*	*Usage*
SP	RAM[0]	*Stack Pointer*: the memory address just following the memory address containing the topmost stack value
LCL	RAM[1]	Base address of the `local` segment
ARG	RAM[2]	Base address of the `argument` segment
THIS	RAM[3]	Base address of the `this` segment
THAT	RAM[4]	Base address of the `that` segment
TEMP	RAM[5-12]	Holds the `temp` segment
R13 R14 R15	RAM[13-15]	If the assembly code generated by the VM translator needs variables, it can use these registers.

When we say *base address* of a segment, we mean a physical address in the host RAM. For example, if we wish to map the local segment on the physical RAM segment starting at address 1017, we can write Hack code that sets LCL to 1017. We note in passing that deciding where to locate virtual memory segments in the host RAM is a delicate issue. For example, each time a function starts executing, we have to allocate RAM space to hold its local and argument memory segments. And when the function calls another function, we have to put these segments on hold and allocate additional RAM space for representing the segments of the called function, and so on and so forth. How can we ensure that these open-ended memory segments will not overflow into each other and into other reserved RAM areas? These memory management challenges will be addressed in the next chapter, when we'll implement the VM language's function-call-and-return commands.

For now though, none of these memory allocation issues should bother us. Instead, you are to assume that SP, ARG, LCL, THIS, and THAT have been already initialized to some sensible addresses in the host RAM. Note that VM implementations never see these addresses anyway. Rather, they manipulate them symbolically, using the pointer names. For example, suppose we want to push the value of the D register onto the stack. This operation can be implemented using the logic RAM[SP++]=D, which can be expressed in Hack assembly as @SP, A=M, M=D, @SP, M=M+1. This code will execute the push operation perfectly, while knowing neither where the stack is located in the host RAM nor what is the current value of the stack pointer.

We suggest taking a few minutes to digest the assembly code just shown. If you don't understand it, you must refresh your knowledge of pointer manipulation in the Hack assembly language (section 4.3, example 3). This knowledge is a prerequisite for developing the VM translator, since the translation of each VM command entails generating code in the Hack assembly language.

Memory Segments Mapping

Local, argument, this, that: In the next chapter we discuss how the VM implementation maps these segments dynamically on the host RAM. For now, all we have to know is that the base addresses of these segments are stored in the registers LCL, ARG, THIS, and THAT, respectively. Therefore, any access to the *i*-th entry of a virtual segment (in the context of a VM "push / pop *segmentName i*" command) should be translated into assembly code that accesses address $(base + i)$ in the RAM, where *base* is one of the pointers LCL, ARG, THIS, or THAT.

Pointer: Unlike the virtual segments described above, the pointer segment contains exactly two values and is mapped directly onto RAM locations 3 and 4. Recall that these RAM locations are also called, respectively, THIS and THAT. Thus, the semantics of the pointer segment is as follows. Any access to pointer 0 should result in accessing the THIS pointer, and any access to pointer 1 should result in accessing the THAT pointer. For example, pop pointer 0 should set THIS to the popped value, and push pointer 1 should push

onto the stack the current value of THAT. These peculiar semantics will make perfect sense when we write the compiler in chapters 10–11, so stay tuned.

Temp: This 8-word segment is also fixed and mapped directly on RAM locations 5 – 12. With that in mind, any access to temp *i*, where *i* varies from 0 to 7, should be translated into assembly code that accesses RAM location $5+i$.

Constant: This virtual memory segment is truly virtual, as it does not occupy any physical RAM space. Instead, the VM implementation handles any access to constant *i* by simply supplying the constant *i*. For example, the command push constant 17 should be translated into assembly code that pushes the value 17 onto the stack.

Static: Static variables are mapped on addresses 16 to 255 of the host RAM. The VM translator can realize this mapping automatically, as follows. Each reference to static *i* in a VM program stored in file Foo.vm can be translated to the assembly symbol Foo.*i*. According to the *Hack machine language specification* (chapter 6), the Hack assembler will map these symbolic variables on the host RAM, starting at address 16. This convention will cause the static variables that appear in a VM program to be mapped on addresses 16 and onward, *in the order in which they appear in the VM code*. For example, suppose that a VM program starts with the code push constant 100, push constant 200, pop static 5, pop static 2. The translation scheme described above will cause static 5 and static 2 to be mapped on RAM addresses 16 and 17, respectively.

This implementation of static variables is somewhat devious, but works well. It causes the static variables of different VM files to coexist without intermingling, since their generated *FileName.i* symbols have unique prefix file names. We note in closing that since the stack begins at address 256, the implementation limits the number of static variables in a Jack program to $255-16+1=240$.

Assembly language symbols: Let us summarize all the special symbols mentioned above. Suppose that the VM program that we have to translate is stored in a file named Foo.vm. VM translators conforming to the standard

VM mapping on the Hack platform generate assembly code that uses the following symbols: SP, LCL, ARG, THIS, THAT, and Foo.*i*, where *i* is a nonnegative integer. If they need to generate code that uses variables for temporary storage, VM translators can use the symbols R13, R14, and R15.

7.4.2 The VM Emulator

One relatively simple way to implement a virtual machine is to write a high-level program that represents the stack and the memory segments and implements all the VM commands using high-level programming. For example, if we represent the stack using a sufficiently-large array named stack, then push and pop operations can be directly realized using high-level statements like `stack[SP++]=x` and `x=stack[--SP]`, respectively. The virtual memory segments can also be handled using arrays.

If we want this VM emulation program to be fancy, we can augment it with a graphical interface, allowing users to experiment with VM commands and visually inspect their impact on images of the stack and the memory segments. The Nand to Tetris software suite includes one such emulator, written in Java (see figure 7.6). This handy program allows loading and executing VM code as is and observing visually, during simulated run-time, how the VM commands effect the states of the emulated stack and memory segments. In addition, the emulator shows how the stack and the memory segments are mapped on the host RAM and how the RAM state changes when VM commands execute. The supplied VM emulator is a cool program—try it!

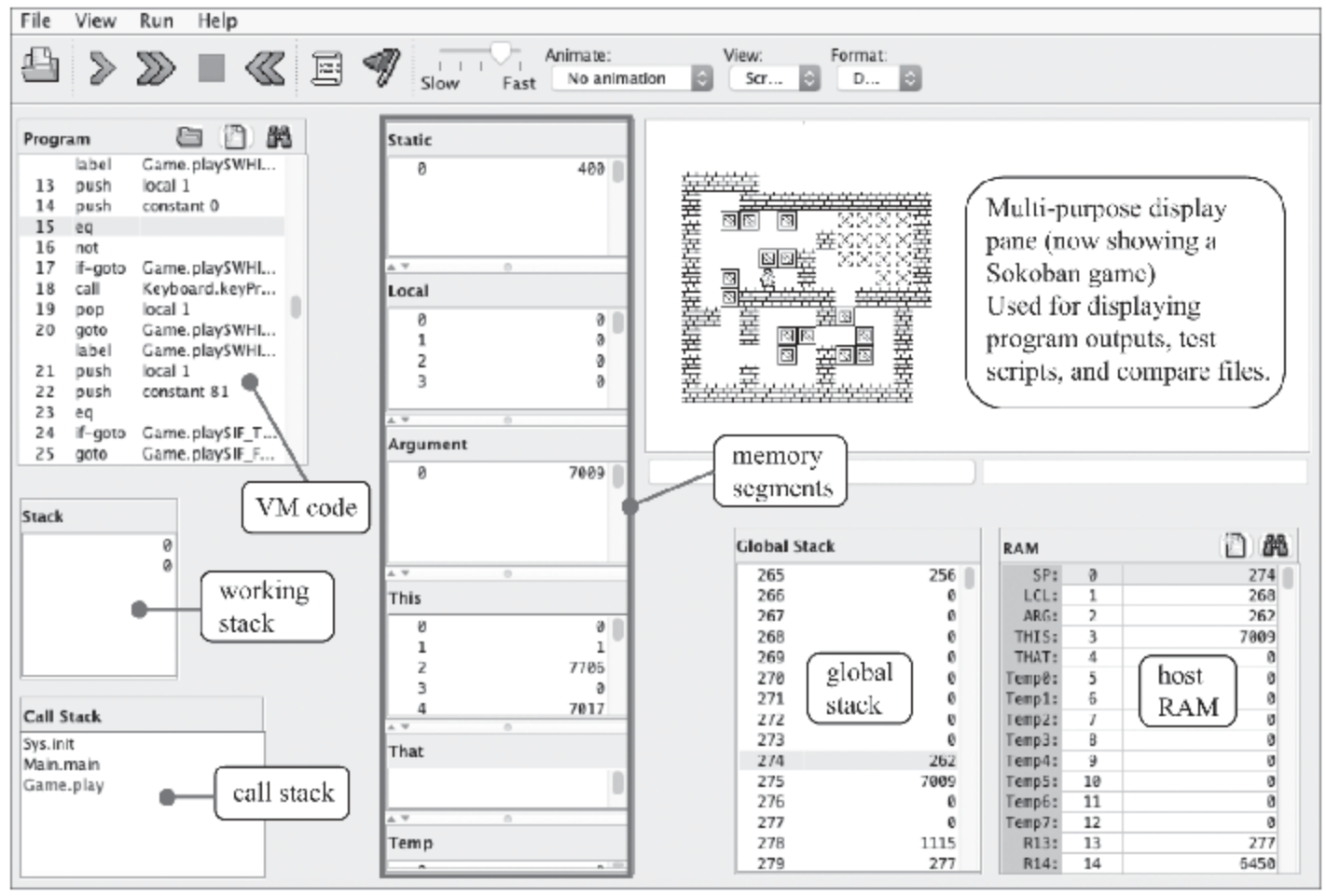


Figure 7.6 The VM emulator supplied with the Nand to Tetris software suite.

7.4.3 Design Suggestions for the VM Implementation

Usage: The VM translator accepts a single command-line argument, as follows:

prompt> VMTranslator *source*

where *source* is a file name of the form *ProgName*.vm. The file name may contain a file path. If no path is specified, the VM translator operates on the current folder. The first character in the file name must be an uppercase letter, and the vm extension is mandatory. The file contains a sequence of one or more VM commands. In response, the translator creates an output file, named *ProgName*.asm, containing the assembly instructions that realize the VM commands. The output file *ProgName*.asm is stored in the same folder as that of the input. If the file *ProgName*.asm already exists, it will be overwritten.

Program Structure

We propose implementing the VM translator using three modules: a main program called VMTranslator, a Parser, and a CodeWriter. The Parser's job is to

make sense out of each VM command, that is, understand what the command seeks to do. The CodeWriter's job is to translate the understood VM command into assembly instructions that realize the desired operation on the Hack platform. The VMTranslator drives the translation process.

The Parser

This module handles the parsing of a single .vm file. The parser provides services for reading a VM command, unpacking the command into its various components, and providing convenient access to these components. In addition, the parser ignores all white space and comments. The parser is designed to handle all the VM commands, including the *branching* and *function* commands that will be implemented in chapter 8.

Routine	*Arguments*	*Returns*	*Function*
Constructor / initializer	Input file / stream	—	Opens the input file / stream, and gets ready to parse it.
`hasMoreLines`	—	boolean	Are there more lines in the input?
`advance`	—	—	Reads the next command from the input and makes it the current command. This routine should be called only if `hasMoreLines` is true. Initially there is no current command.
`commandType`	—	`C_ARITHMETIC`, `C_PUSH`, `C_POP`, `C_LABEL`, `C_GOTO`, `C_IF`, `C_FUNCTION`, `C_RETURN`, `C_CALL` (constant)	Returns a constant representing the type of the current command. If the current command is an arithmetic-logical command, returns `C_ARITHMETIC`.
`arg1`	—	string	Returns the first argument of the current command. In the case of `C_ARITHMETIC`, the command itself (`add`, `sub`, etc.) is returned. Should not be called if the current command is `C_RETURN`.
`arg2`	—	int	Returns the second argument of the current command. Should be called only if the current command is `C_PUSH`, `C_POP`, `C_FUNCTION`, or `C_CALL`.

For example, if the current command is `push local 2`, then calling `arg1()` and `arg2()` would return, respectively, `"local"` and `2`. If the current command is `add`, then calling `arg1()` would return `"add"`, and `arg2()` would not be called.

The CodeWriter

This module translates a parsed VM command into Hack assembly code.

Routine	*Arguments*	*Returns*	*Function*
Constructor / initializer	Output file / stream	—	Opens an output file / stream and gets ready to write into it.
`writeArithmetic`	`command` (string)	—	Writes to the output file the assembly code that implements the given arithmetic-logical `command`.
`writePushPop`	`command` (`C_PUSH` or `C_POP`), `segment` (string), `index` (int)	—	Writes to the output file the assembly code that implements the given push or pop `command`.
`close`	—	—	Closes the output file / stream.
More code writing routines will be added to this module in chapter 8.			

For example, calling writePushPop (C_PUSH,"local",2) would result in generating assembly instructions that implement the VM command push local 2. Another example: Calling WriteArithmetic("add") would result in generating assembly instructions that pop the two topmost elements from the stack, add them up, and push the result onto the stack.

The VM Translator

This is the main program that drives the translation process, using the services of a Parser and a CodeWriter. The program gets the name of the input source file, say *Prog*.vm, from the command-line argument. It constructs a Parser for parsing the input file *Prog*.vm and creates an output file, *Prog*.asm, into which it will write the translated assembly instructions. The program then enters a loop that iterates through the VM commands in the input file. For each command, the program uses the Parser and the CodeWriter services for parsing the command into its fields and then generating from them a sequence of assembly instructions. The instructions are written into the output *Prog*.asm file.

We provide no API for this module, inviting you to implement it as you see fit.

Implementation Tips

1. When starting to translate a VM command, for example, push local 2, consider generating, and emitting to the output assembly code stream, a

comment like // push local 2. These comments will help you read the generated code and debug your translator if needed.

2. Almost every VM command needs to push data onto and/or pop data off the stack. Therefore, your write*Xxx* routines will need to output similar assembly instructions over and over. To avoid writing repetitive code, consider writing and using private routines (sometimes called *helper methods*) that generate these frequently used code snippets.
3. As was explained in chapter 6, it is recommended to end each machine language program with an infinite loop. Therefore, consider writing a private routine that writes the infinite loop code in assembly. Call this routine once, when you are done translating all the VM commands.

7.5 Project

Basically, you have to write a program that reads VM commands, one command at a time, and translates each command into Hack instructions. For example, how should we handle the VM command push local 2? *Tip:* We should write several Hack assembly instructions that, among other things, manipulate the SP and LCL pointers. Coming up with a sequence of Hack instructions that realizes each one of the VM arithmetic-logical and push/pop commands is the very essence of this project. That's what code generation is all about.

We recommend you start by writing and testing these assembly code snippets on paper. Draw a RAM segment, draw a trace table that records the values of, say, SP and LCL, and initialize these variables to arbitrary memory addresses. Now, track on paper the assembly code that you think realizes say, push local 2. Does the code impact the stack and the local segments correctly (RAM-wise)? Did you remember to update the stack pointer? And so on. Once you feel confident that your assembly code snippets do their jobs correctly, you can have your CodeWriter generate them, almost as is.

Since your VM translator has to write assembly code, you must flex your low-level Hack programming muscles. The best way to do it is by reviewing the assembly program examples in chapter 4 and the programs

that you wrote in project 4. If you need to consult the Hack assembly language documentation, see section 4.2.

Objective: Build a basic VM translator designed to implement the *arithmetic-logical* and *push* / *pop* commands of the VM language.

This version of the VM translator assumes that the source VM code is error-free. Error checking, reporting, and handling can be added to later versions of the VM translator but are not part of project 7.

Resources: You will need two tools: the programming language in which you will implement the VM translator and the *CPU emulator* supplied in your nand2tetris/tools folder. The CPU emulator will allow you to execute and test the assembly code generated by your translator. If the generated code runs correctly in the CPU emulator, we will assume that your VM translator performs as expected. This is just a partial test of the translator, but it will suffice for our purposes.

Another tool that comes in handy in this project is the *VM emulator*, also supplied in your nand2tetris/tools folder. We encourage using this program for executing the supplied test programs and watching how the VM code effects the (simulated) states of the stack and the virtual memory segments. For example, suppose that a test program pushes a few constants onto the stack and then pops them into the local segment. You can run the test program on the VM emulator, inspect how the stack grows and shrinks, and see how the local segment becomes populated with values. This can help you understand which actions the VM translator is supposed to generate before setting out to implement it.

Contract: Write a VM-to-Hack translator conforming to the VM Specification given in section 7.3 and to the standard VM mapping on the Hack platform given in section 7.4.1. Use your translator to translate the supplied test VM programs, yielding corresponding programs written in the Hack assembly language. When executed on the supplied CPU emulator, the assembly programs generated by your translator should deliver the results mandated by the supplied test scripts and compare files.

Testing and Implementation Stages

We provide five test VM programs. We advise developing and testing your evolving translator on the test programs in the order in which they are given. This way, you will be implicitly guided to build the translator's code generation capabilities gradually, according to the demands presented by each test program.

SimpleAdd: This program pushes two constants onto the stack and adds them up. Tests how your implementation handles the commands push constant *i* and add.

StackTest: Pushes some constants onto the stack and tests how your implementation handles all the arithmetic-logical commands.

BasicTest: Executes push, pop, and arithmetic commands using the memory segments constant, local, argument, this, that, and temp. Tests how your implementation handles these memory segments (you've already handled constant).

PointerTest: Executes push, pop, and arithmetic commands using the memory segments pointer, this, and that. Tests how your implementation handles the pointer segment.

StaticTest: Executes push, pop, and arithmetic commands using constants and the memory segment static. Tests how your implementation handles the static segment.

Initialization: In order for any translated VM program to start running, it must include startup code that forces the generated assembly code to start executing on the host platform. And, before this code starts running, the VM implementation must anchor the base addresses of the stack and the virtual memory segments in selected RAM locations. Both issues—startup code and segments initializations—are described and implemented in the next chapter. The difficulty here is that we need these initializations in place for executing the test programs in this project. The good news is that you need not worry about these details, since all the initializations necessary for this project are handled "manually" by the supplied test scripts.