Introduction: Hello, World Below

The true voyage of discovery consists not of going to new places, but of having a new pair of eyes.
—Marcel Proust (1871–1922)

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

The World Above

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

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

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

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

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

Abstractions

You may wonder how it is humanly possible to construct a complete computer system from the ground up, starting with nothing more than elementary logic gates. This must be an enormously complex enterprise! We deal with this complexity by breaking the project into *modules*, and treating each module separately, in a standalone chapter. You might then wonder, how is it possible to describe and construct these modules in isolation? Obviously they are all interrelated! As we will show

throughout the book, a good modular design implies just that: You can work on the individual modules independently, while completely ignoring the rest of the system. In fact, you can even build these modules in any desired order!

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

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

The World Below

The multi-tier collection of abstractions underlying the design of a computing system can be described *top-down*, showing how high-level abstractions can be reduced into, or expressed by, simpler ones. This structure can also be described *bottom-up*, focusing on how lower-level abstractions can be used to construct more complex ones. This book takes the latter approach: We begin with the most basic elements—

primitive logic gates—and work our way upward, culminating in the construction of a general-purpose computer system. And if building such a computer is like climbing Mount Everest, then planting a flag on the mountaintop is like having the computer run a program written in some high-level language. Since we are going to ascend this mountain from the ground up, let us survey the book plan in the opposite direction—from the top down—starting in the familiar territory of high-level programming.

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

High-Level Language Land

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

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

Jack is a simple object-based language, designed for a single purpose: to illustrate the key software engineering principles underlying the design and implementation of modern programming languages like Java and C#. Jack is presented in chapter 9, which also illustrates how to build Jack-based applications, for example, computer games. If you have any programming experience with a modern object-oriented language, you can start writing Jack programs right away and watch them execute on the computer platform developed in previous chapters. However, the goal of chapter

9 is not to turn you into a Jack programmer, but rather to prepare you to develop the compiler and operating system described in subsequent chapters.

The Road Down to Hardware Land

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

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

Hardware Land

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

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

developed hardware modules can be tested using supplied test scripts, written in a scripting language documented in appendix B.

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

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

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

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

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