Part 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 re-expresses 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. In order 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, 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 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 neither Nand, nor 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 roadmap. Starting at the bottom tier of the figure, any general-purpose computer has an architecture that includes a CPU (processor) and a RAM (memory). All CPU 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.

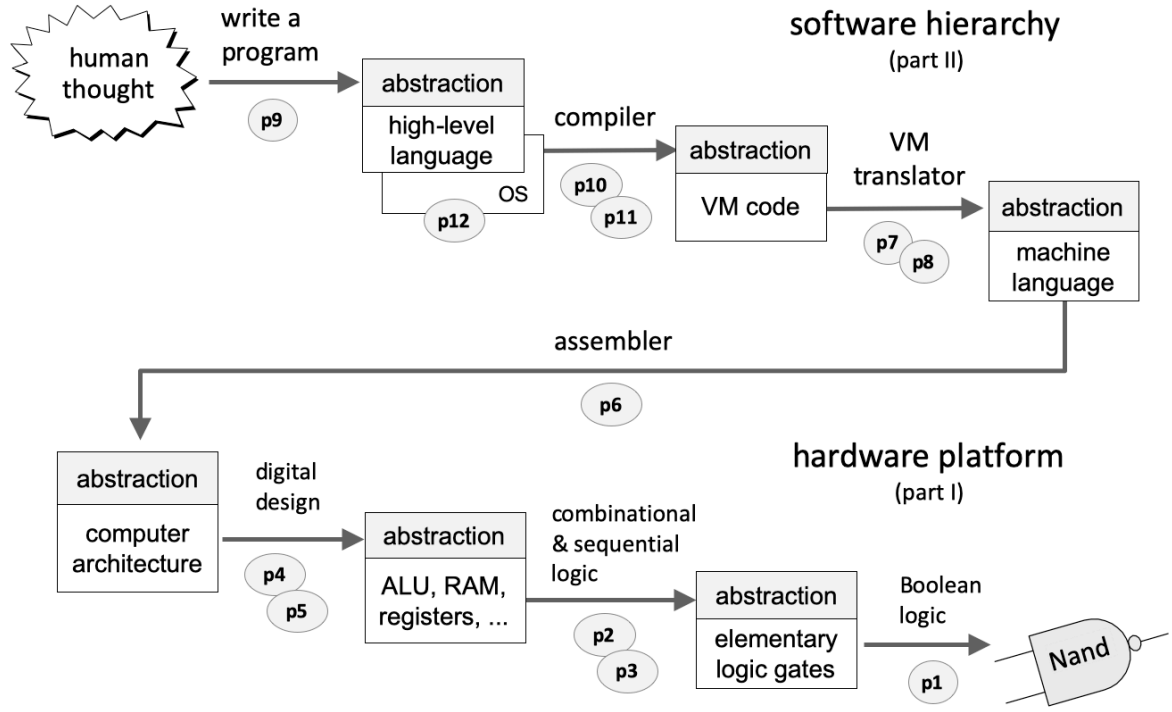

Figure I.1: Major modules of a typical computer system, consisting of a hardware platform and a software hierarchy. Each module has an *abstract view* (also called the module's *interface*), and an *implementation*. The right-going arrows signify that each module is implemented using abstract building blocks from the level below. Each circle represents a Nand to Tetris project and chapter – 12 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 hard-core 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 1 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 down-going arrow connecting the higher module with the lower one. The meaning of this arrow is very 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 stand-alone 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 very 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 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 actually build the RAM chip. How should we go about it? Following the right-

going 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 *multiplexors*, 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 some 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, quite simply, 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 stand-alone 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 30 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 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 API's, 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 12 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 upwards, leading up 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'll present some 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'll understand the module's abstraction (a rich world in its own right), you'll proceed to actually 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!