

0. The Grand Tour¹

*“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. None of these things will be taught explicitly. Instead, we will engage you in the step-by-step creation of a complete computer system, from the ground up. The lessons that we wish to impart, which are far more important 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.” After teaching computer science for 30 years combined, we cannot agree more.

Computer systems are based on many layers of abstractions. Thus our voyage will consist of going from one abstraction to the other. This can be done in two directions. The *top-down* route shows how high-level abstractions (e.g. commands in an object-oriented language) can be reduced into, or expressed by, simpler ones (e.g. operations on a virtual machine). The *bottom-up* route shows how low-level abstractions (e.g. flip-flops) can be used to construct more complex ones (e.g. memory chips). This book takes the latter approach: we’ll begin with the most basic elements possible -- primitive logic gates -- and work our way upward, constructing a general-purpose computer, equipped with an operating system and a Java-like language.

If building such a computer from scratch is like climbing the Everest, then planting a flag on the mountain’s top is like having the computer run some non-trivial application programs. Since we are going to ascend this mountain from the ground up, we wish to start with a preview that goes in the opposite direction -- from the top down. Thus, the Grand Tour presented in this chapter will start at the end of our journey, by demonstrating an interactive video game running on the complete target computer. Next, we will drill through the main software and hardware abstractions that make this application work, all the way down to the bare bone transistors level.

The resulting tour will be casual. Instead of stopping to analyze each hardware and software abstraction thoroughly, we will descend quickly from one layer to the other, presenting a holistic picture that ignores many details. In short, the purpose of this chapter is to “cut through” the layers of abstraction discussed in the book, providing a high-level map into which all the other chapters can be placed.

¹ From *The Elements of Computing Systems*, Nisan & Schocken, MIT Press, forthcoming in 2003, www.idc.ac.il/csd

0. Background

The World Below

We assume that readers of this book are familiar with writing and debugging computer programs. Did you ever stop to think about the hardware and software systems that facilitate this art? Let's take a look. Suppose that we are developing some application using an object-oriented language. Typically, the process starts by abstracting the application using a set of classes and methods. Now, if we implement this design using a language like Java or C#, then the next step is to use a *compiler* to translate our high-level program into an intermediate code, designed to run on a *virtual machine* (VM). Next, if we want to actually see our program running, the VM abstraction must be realized on some real computer. This can be done by a program called *VM translator*, designed to convert VM code into the assembly language of the target computer. The resulting code can then be translated into machine language, using yet another translator, called *assembler*.

Of course *machine language* is also an abstraction -- an agreed upon set of binary codes. In order to make this abstract formalism do something for real, 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, are 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. Why? Because that's where computer science ends and physics starts, and this book is about computer science.

You may be thinking: "well, on *my* computer, compiling and running a program is much easier -- all I do is click some icons or write some commands!" Indeed, a modern computer system is like an iceberg, and most people get to see only the top. Their knowledge of computing systems is sketchy and superficial. If, however, you wish to go under the surface and investigate the systems below, then *Lucky You!* There's a fascinating world down there, below the GUI level and the OS shell. An intimate understanding of this under-world is what separates naïve programmers from professional developers -- people who can create not only end-user applications, but also new 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 computer from scratch. Our journey begins at the top of Figure 0.

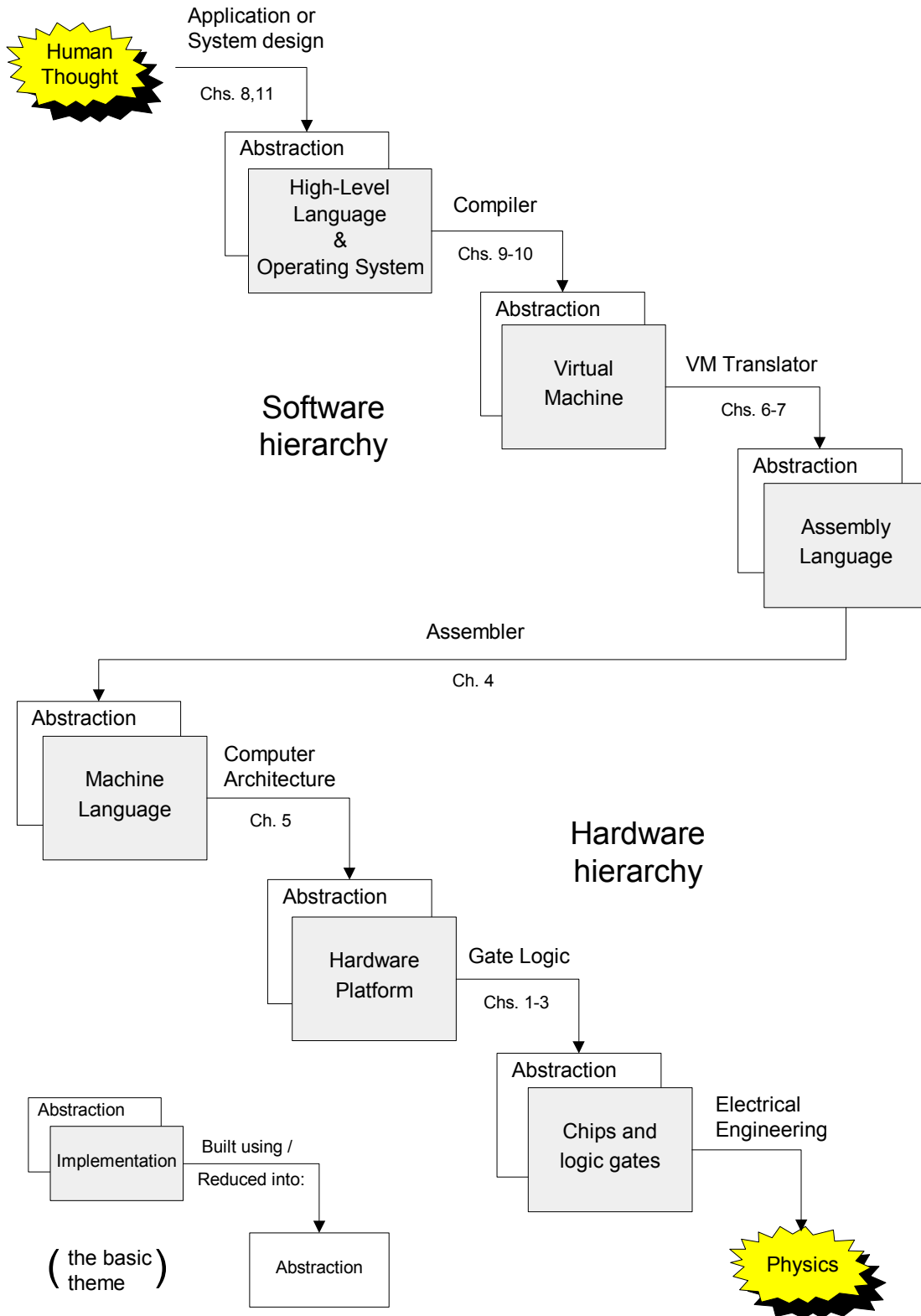


FIGURE 0: The Grand Tour (showing major stops only)

Multiple Layers of Abstraction

This book walks you through the process of constructing a complete computer system: hardware, software, and all their interfaces. You may wonder how it is possible. After all, a computer system is an enormously complex enterprise! Well, we break the project into *modules*, and we treat each module separately, in a stand-alone chapter. You might then wonder: how it is possible to describe and construct these modules in isolation? Obviously they are all inter-related! 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 problem. It turns out that people are good at this strategy thanks to a unique human faculty: the ability to create and use *abstractions*.

In computer science, an abstraction is simply a functional description of something. For example, if we are asked to develop a digital camera chip that can detect close moving objects, we can do it using low-level components like CCD chips and some routines written in the C language. Importantly, we don't have to worry about *how* these low-level hardware and software modules are implemented -- we treat them as *abstract artifacts* with predictable and well-documented behaviors. In a similar fashion, once built, our camera chip may end up being used as a building block in a variety of Driver Assistance Systems (DAS) such as adaptive cruise control, blind spot detection, lane departure warning, and so on. Now, the engineers who will build these DAS applications will care little about *how* our camera chip works. They, too, will want to use it as an off-the-shelf component with a predictable and well-documented behavior. In general then, when we operate in a particular level of a complex design, it is best to focus on that level only, "abstracting away" all the other parts of the system. This may well be the most important design principle in building large-scale computing systems.

Clearly, the notion of abstraction is not unique to computer science -- it is central to all scientific and engineering disciplines. In fact, the ability to deal with abstractions is often considered a hallmark of human intelligence in general. Yet in computer science, we take the notion of abstractions one step further. Looking "up" the construction hierarchy, the abstraction is viewed as a functional description of a given system, aimed at the people who may want to use it in constructing other, higher-level abstractions. Looking "down", the same abstraction is viewed as a complete system specification, aimed at the people who have to *implement* it. Therefore, computer scientists take special pain to define their abstractions clearly and unambiguously.

Indeed, multi-layer abstractions can be found throughout computer science. For example, the computer hardware is abstracted (read: "functionally described") by its *architecture* -- the set of machine level commands that it recognizes. The operating system is abstracted by its *system calls* -- the set of services that it provides to other programs. Applications and software systems are abstracted by their *Application Program Interfaces* -- the set of *object* and *method* signatures that they support. Other levels of abstraction are defined and documented ad-hoc, at any given design level, as the situation demands. In fact, the identification and description of abstract components is the very first thing that we do when we set out to design a new hardware or software system.

System design is a practical art, and one which is best acquired from experience. Therefore, in this book we don't expect you to engage in designing systems. Instead, we will present many classical hardware and software abstractions, and ask you to build them, following our guidelines. This is similar to saying that we don't expect you to formulate new theorems, but rather to prove the ones we supply. Continuing this analogy, you will start at the "bottom", where two primitive hardware gates will be given, not unlike axioms in mathematics. You will then gradually build more and more complex hardware and software levels of abstraction, culminating in a full-scale computer system. This will be an excellent example of an observation made by A.N. Whitehead in 1911: "civilization progresses by increasing the number of operations that can be performed without thinking about them". Note that this sentence remains silent about who's enabling the progress. Well, that's where *you* enter the picture.

In particular, as you'll progress in our journey, each chapter will provide a stand-alone intellectual unit: you need not remember the implementation details of previous systems, nor look ahead to future ones. Instead, in each chapter you will focus on two things only: the design of the current abstraction (a rich world of its own), and how it can be implemented using abstract building blocks from the level below. Using this information, we will guide you in the construction of the current abstraction, turning it into yet another "operation that we can use without thinking about it". As you push ahead, it will be rather thrilling to look back and appreciate the computer that is gradually taking shape in the wake of your efforts.

1. The Journey Starts: High-Level Language Land

The term *high-level* is normally interpreted as "close to the human" (rather than *low-level*, which is close to the machine). In this book, it means the layer at which one interacts with the computer using an object-based programming language and an operating system. There are several reasons why it is difficult to completely separate the discussion of these two subjects. First, modern operating systems are themselves written in high-level languages. Second, a running program is a collection of many routines; some come from the application, some from the OS, but from the computer's perspective they are all alike. Also, modern languages like Java include elaborate software libraries and run-time environments. These language extensions perform GUI management, multi-threading, garbage collection, and many other services that were traditionally handled by the OS.

For all these reasons, our discussion of high-level languages in chapter 8 will include many side-tours into the operating system, which will be discussed and built in chapter 11. The following is a preview of some of the underlying ideas.

The Pong Game

Video games are challenging programs. They use the computer's screen and input units extensively, they require clever modeling of geometric objects and interactive events, and they must execute efficiently. In short, video games pose a tough test to the hardware/software platform on which they run. A simple yet non-trivial example is *Pong* -- the computer game depicted in Fig. 1. In spite of its humble appearance, *Pong* is a historical celebrity: invented and built in the early 1980's, it was the first computer game that became massively popular -- a success that gave rise to a thriving computer games industry.

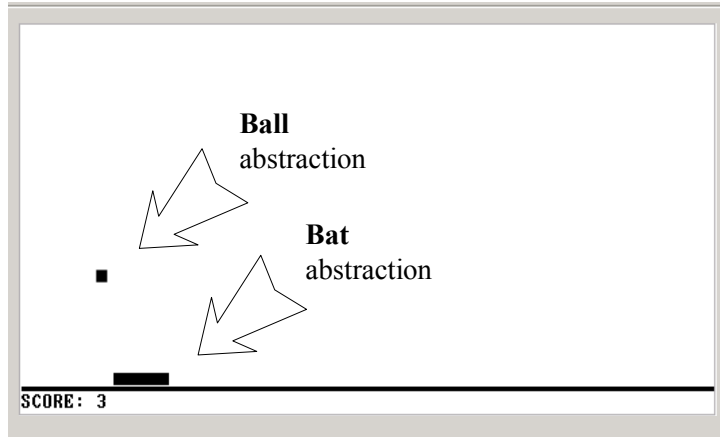


FIGURE 1: The Pong Game (annotated screen shot from a real game session, running on the *Hack* computer built in the book). A ball is moving on the screen randomly, bouncing off the screen “walls”. The user can move a small bat horizontally by pressing the keyboard’s left and right arrow keys. Each time the bat hits the ball, the user scores a point and the bat shrinks a little, to make the game harder. If the user misses and the ball hits the bottom horizontal line, the game is over.

If you will inspect the text and graphics of Fig. 1, you will have a clear understanding of *what* the Pong game is all about, but you will know nothing about *how* it is actually built. Indeed, at this point Pong will be merely an informal *abstraction* -- a theoretical artifact that exists only on paper. The fact that it’s an abstraction, though, does not mean that we have to be informal about its description. In particular, if we wish to *implement* Pong on some target computer platform, we must think hard on how to specify it formally. A good abstract specification is by far the most important deliverable in the life cycle of any application.

One reason why a formal specification is so important is because it forces us to articulate a particular *design* for the given application. Normally, the design process begins by considering various ways to break the application into lower-level abstract components. For example, a Pong application will most likely benefit from components that abstract the behaviors of graphical ball and bat objects. What should these components do? Well, the `Bat` component should probably provide such services as drawing the bat on the screen and moving it left and right. In a similar fashion, the `Ball` component should feature services for drawing the ball, moving the ball in all directions, bouncing it off other objects, and so on.

Thus, if we implement the game in some object-based language, it will make sense to describe the bat and ball objects as instances of abstract `Bat` and `Ball` *classes*. Next, the various characteristics and operations of each object can be specified in terms of *class properties* and *method signatures*, respectively. Taken together, these specifications will yield a document called the Pong Game *Application Program Interface*. This API will be a complete specification of the modules that make up Pong, aimed at people who have to either build these modules, or, alternatively, use them in the context of other systems.

A Quick Look at the High-Level Language

Once an abstraction has been formally specified, it can be implemented in many different ways. For example, Program 2 gives a possible *Jack* implementation of the bat abstraction, necessary for building the Pong game (and, in fact, many other games involving graphical bats). This being the first time that we encounter Jack in the book, a few words of introduction are in order. Jack is a simple, Java-like language that has two important virtues. First, if you have any experience in object-oriented programming, you can pick it up in just a few minutes. Second, the Jack syntax was especially designed to simplify the construction of Jack compilers, as we will see shortly.

```

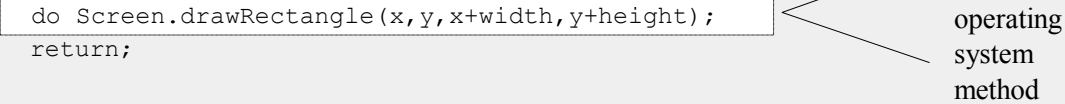
/** A Graphic Bat for a Pong Game */
class Bat {
    field int x, y;           // screen location of the bat's top-left corner
    field int width, height; // bat's width & height

    // The class constructor and most of the class methods are omitted

    /** Draws (color=true) or erases (color=false) the bat */
    method void draw(boolean color) {
        do Screen.setColor(color);
        do Screen.drawRectangle(x, y, x+width, y+height);
        return;
    }

    /** Moves the bat one step (4 pixels) to the right. */
    method void moveR() {
        do draw(false); // erase the bat at the current location
        let x = x + 4; // change the bat's X-location
        // but don't go beyond the screen's right border
        if ((x + width) > 511) {
            let x = 511 - width;
        }
        do draw(true); // re-draw the bat in the new location
        return;
    }
}

```



PROGRAM 2: High-Level implementation of the bat abstraction,
written in the *Jack* programming language.

The code of Program 2 should be self-explanatory. The `Bat` class (implementing the bat abstraction) encapsulates various bat-related services, implemented as methods. Two of these methods are shown in the figure: a “draw” method by which a bat object draws itself on the screen, and a “moveR” method by which a bat object moves itself one step to the right. Since the `Bat` class will come to play in the context of some overall program, it is likely to assume that the “drwaR” method will be invoked when the user presses the right arrow key on the keyboard.

However, this logic should not be part of the `Bat` class. Instead, it belongs to some other module in the program, e.g. one that implements a game session abstraction.

We will illustrate the design of object-based languages in Chapter 8, by specifying the Jack language and writing some sample applications in it. This will set the stage for chapters 9 and 10, in which we discuss compilation techniques and build the Jack compiler.

Peeking Inside the Operating System

The computer platform that we will build in chapter 5, called *Hack*, features a black and white screen consisting of 256 rows by 512 columns (similar to that of hand-held computers and cellular telephones). High level languages like Jack are expected to provide high-level means for interacting with this screen. Indeed, an inspection of Prog. 2 reveals two screen oriented method calls: `Screen.setColor` and `Screen.drawRectangle`. The first method sets the default screen color (i.e. the color that subsequent drawing operations will use), and the second method draws a rectangle of given dimensions at a given screen location. These methods are part of a class called `Screen`, which is part of a software layer that interfaces between the Jack language and the Hack hardware. This software layer, called the *Sack* operating system, will be described and built in Chapter 11.

Parts of the `Screen` class are shown in Program 3. Since the Sack OS is also written in Jack, the code of the `drawRectangle` function should be self-explanatory: the rectangle is drawn using a simple nested loop logic. What about the `drawPixel` function? In the Hack platform that we will build in chapter 5, the computer's screen will be *memory-mapped*. In other words, a certain area in the computer's random-access memory will be dedicated for representing the screen's contents, one bit per pixel. In addition, a refresh logic will be used to continuously re-draw the physical screen according to the current contents of its memory map. Thus, when we tell `Screen.drawPixel` to "draw" a pixel in a certain screen location, all it has to do is change the corresponding bit in the screen memory map. In the next iteration of the refresh loop (which runs several times each second), the change will be "automatically" reflected on the computer screen.

Because of their analog nature, input and output devices are always the clunkiest parts of digital computer architectures. Therefore, it is best to abstract I/O devices away from programmers, by encapsulating the operations that manipulate them in low-level OS routines. `DrawPixel` is a good example of this practice, as it provides a clean screen drawing abstraction not only for user-level programs, but also for other OS routines like `drawRectangle`.

Once again, we see the power of abstractions at work. Beginning at the top of the software hierarchy (e.g. Pong), we find programmers who draw graphical images using abstract operations like `drawRectangle`. This method signature is part of the Sack OS API, and thus one is free to invoke it in programming languages that run on top of Hack/Sack platform. When we drill down to the OS level, we see that the `drawRectangle` abstraction is implemented using the services of `drawPixel`, which is yet another, lower-level abstraction. Indeed, the abstraction-implementation interplay can run deep -- as deep as the designer wants.


```
/** An OS-level screen driver that abstracts the computer's physical screen */
class Screen {
    static boolean currentColor; // the current color

    // The Screen class is a collection of methods, each implementing one
    // abstract screen-oriented operation. Most of this code is omitted.

    /** Draws a single pixel in the current color. */
    function void drawPixel(int x, int y) {
        // Draws the pixel in screen location (x,y) by writing corresponding
        // bits in the screen memory map. The method code is omitted.    }

    /** Draws a rectangle in the current color. */
    // the rectangle's top left corner is anchored at screen location (x0,y0)
    // and its width and length are x1 and y1, respectively.
    function void drawRectangle(int x0, int y0, int x1, int y1) {
        var int x, y;
        let x = x0;
        while (x < x1) {
            let y = y0;
            while(y < y1) {
                do Screen.drawPixel(x,y);
                let y = y+1;
            }
            let x = x+1;
        }
    }
}
```

PROGRAM 3: Code segment from the Sack operating system, written in the Jack language. (In Jack, class-level methods that don't operate on any particular object are called "functions".)

The screen driver discussed above is just a small part the Sack OS. The overall operating system is an elaborate collection of software libraries, designed to manage the computer's input, output, and memory devices, as well as provide mathematical, string, and array processing services to high-level languages. Like other modern operating systems, Sack itself is written in a high level language (in our case, Jack). This may seem surprising to readers who are used to operate on top of a proprietary operating system that gives no access to its source code. We will open the OS black box in Chapter 11, where we present several geometric, arithmetic, and memory management algorithms, each being a computer science gem. These algorithms will be discussed in the context of building a Sack OS implementation.

2. The Journey Continues: the Road Down to Hardware Land

We now start crossing the great chasm between the high-level language abstraction and its low-level implementation in hardware. Before a program can actually run and do something for real, it must be translated into the machine language of some target computer. The translation process -- known as *compilation* -- is often performed in two stages. In the first stage, a *compiler* translates the high-level code into an intermediate abstraction called *virtual machine*. In the second stage, the virtual machine abstraction is implemented on the target hardware platform(s). We devote a third of the book for discussing these fundamental software engineering issues. The following is a preview of some of the ideas involved.

The Compiler at a Glance

Think about the general challenge of translating a sentence from one language to another. The first thing that you will do is use the grammar rules of the source language (perhaps implicitly) to figure out the syntactic structure of the given sentence. The translation of programming languages follows the same rationale. Each programming language has a well-documented grammar that defines how valid statements and expressions are structured in the language. Using this grammar, the compiler developer can write a program that converts the source code into some recursive data structure, designed to represent the code in a convenient way for further processing. The output of this *syntax analyzer* program (also called *parser*) can typically be described in terms of a *parse tree*. For example, Fig. 4 illustrates the parse tree of a high-level expression taken from Program 2.

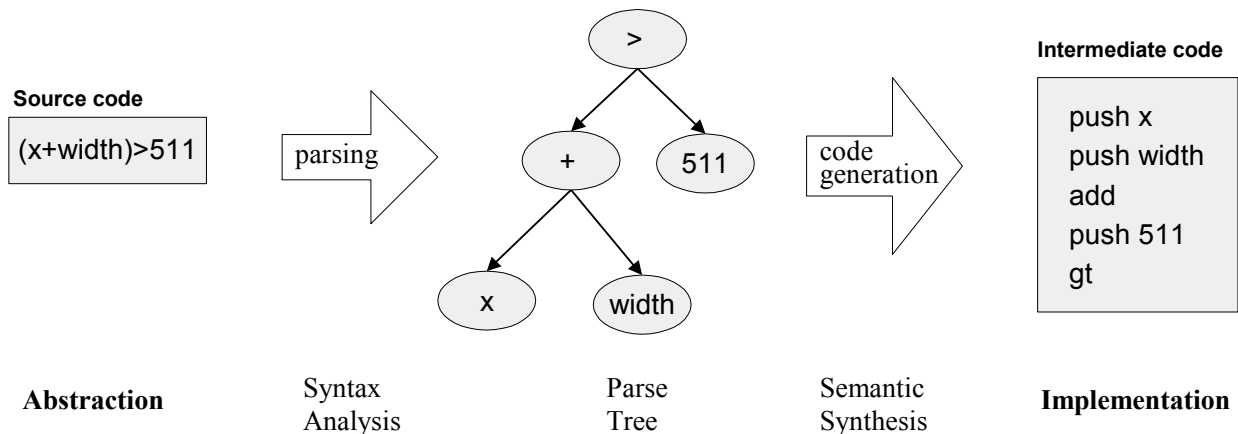


FIGURE 4: Compilation example

Once the source code has been “understood,” i.e. parsed, it can be further translated into some target language (this time, using the grammar rules of the latter) -- typically the machine language of the target computer. However, the approach taken by modern compilers, e.g. those of Java and C#, is to first break the parsed code into generic processing steps, designed to run on some abstract “machine”. Importantly, the resulting intermediate code depends on neither the source of the translation, nor on its final destination. Therefore, it is quite easy to compile it further into multiple target platforms, as needed. Of course the exact specification of the “generic processing

steps” is a key design issue. In fact, this intermediate code form is important enough so that it is often formalized as a stand-alone abstraction, called *Virtual Machine* or VM.

As it turns out, it is convenient to express the VM operations using a *postfix* format called (for historical reasons) *Right Polish Notation* or *RPN*. For example, the source expression “ $(x+width)>511$ ” is expressed in *infix* notation, meaning that operators are written between their operands, simply because that’s how human programmers are trained to think. In *postfix* notation, operators are written after the operands, as in “ $x, width, +, 511, >$ ”. This parentheses-free format is flattened and “un-nested”, and thus it lends itself nicely to low-level processing. Therefore, one thing that we want our compiler to do is translate the original code into some postfix language, as seen in the right of Fig. 4. How does the compiler achieve this translation task?

An inspection of Fig. 4 suggests that the postfix target code can be generated by the following algorithm:

- Perform a complete recursive *depth-first* processing of the parse tree;
- When reaching a terminal node x , generate the command “push x ”;
- When backtracking to an interim node from the right, generate the command which is the node’s label.

One question that comes to mind is whether this algorithm scales up to compiling a complete program rather than a single expression. The answer is *yes*. Any given program, no matter how complex, can be expressed as a parse tree. The compiler will not necessarily hold the entire tree in memory, but it will create and manipulate it using precisely the same techniques illustrated above.

The theory and practice of compilation are normally covered in a full-semester course. This book devotes two chapters to the subject, focusing on the most important ideas in syntax analysis and code generation. In chapter 9, we will build a parser that translates Jack programs into parse trees, expressed as XML files. In chapter 10, we will upgrade this parser into a compilation engine that produces VM code. The result will be a full-scale Jack compiler.

Virtual Machine Preview

To reiterate, many modern compilers don’t generate machine code directly. Instead, they generate intermediate code designed to run on an abstract computer called *Virtual Machine*. There are several possible paradigms on which to base a virtual machine architecture. Perhaps the cleanest and most popular one is the *stack machine* model, used in the *Java Virtual Machine* as well in the VM that we build in this book.

A *stack* is an abstract data structure that supports two basic operations: *push* and *pop*. The *push* operation adds an element to the “top” of the stack; the element that was previously on top is pushed “below” the newly added element. The *pop* operation retrieves and removes the top element off the stack; the element just “below” it moves up to the top position. The “add” operation removes the top two elements and puts their sum at the top. In a similar fashion, the “gt” operation (*greater than*) removes the top two elements. If the first is greater than the second, it puts the constant *true* at the top; otherwise it puts the constant *false*.

To illustrate stack processing in action, consider the following high-level code segment, taken from our bat implementation (Program 2):

```
if ((x+width)>511) {
    let x=511-width;
}
```

Fig. 5 shows how the semantics of this code can be expressed in a stack-based formalism.

```
// VM implementation of "if ((x+width)>511){let x=511-width;}"
push x      // s1: push the value of x to the stack top
push width  // s2: push the value of width to the stack top
add         // s3: pop the top two values, push their sum
push 511    // s4: push the constant 511
gt         // s5: pop the top two values, if 1st>2nd push true
if-goto L1  // s6: pop the top value, if it's true goto L1
goto L2    // s7: skip the conditional code

L1:
push 511    // s8: push the constant 511
push width  // s9: push the value of width to the stack top
sub        // s10: pop the top two values, push 1st-2nd
pop x      // s11: pop the top value into x

L2:
...
```

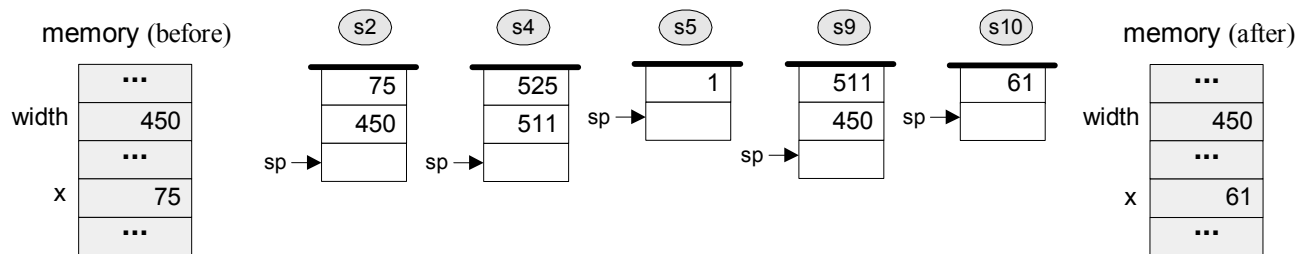


FIGURE 5: Virtual Machine code segment (top) and run-time scenario (bottom). To connect the two figures, we have annotated the VM commands and the stack images with state markers. (In stack diagrams, the next available slot is typically marked by the label *sp*, for *stack pointer*. Following convention, the stack is drawn upside down, as if it grows downward.)

The VM language and its impact on the stack are explained in the program's comments. This basic language, which provides stack arithmetic and control flow capabilities, will be developed and implemented in Chapter 6. Next, in Chapter 7, we will extend it into a more powerful abstraction, capable of handling multi-method and object-based programs as well. The resulting language will be modeled after the *Java Virtual Machine (JVM)* paradigm.

There is no need to delve further into the VM world here. Rather, it is sufficient to appreciate the general idea, which is as follows: instead of translating high level programs directly into the machine language of a specific computer, we first compile them into an intermediate code that runs on a virtual machine. The flip-side of this strategy is that in order to run the abstract VM programs for real, we must *implement* the VM on some real computer platform.

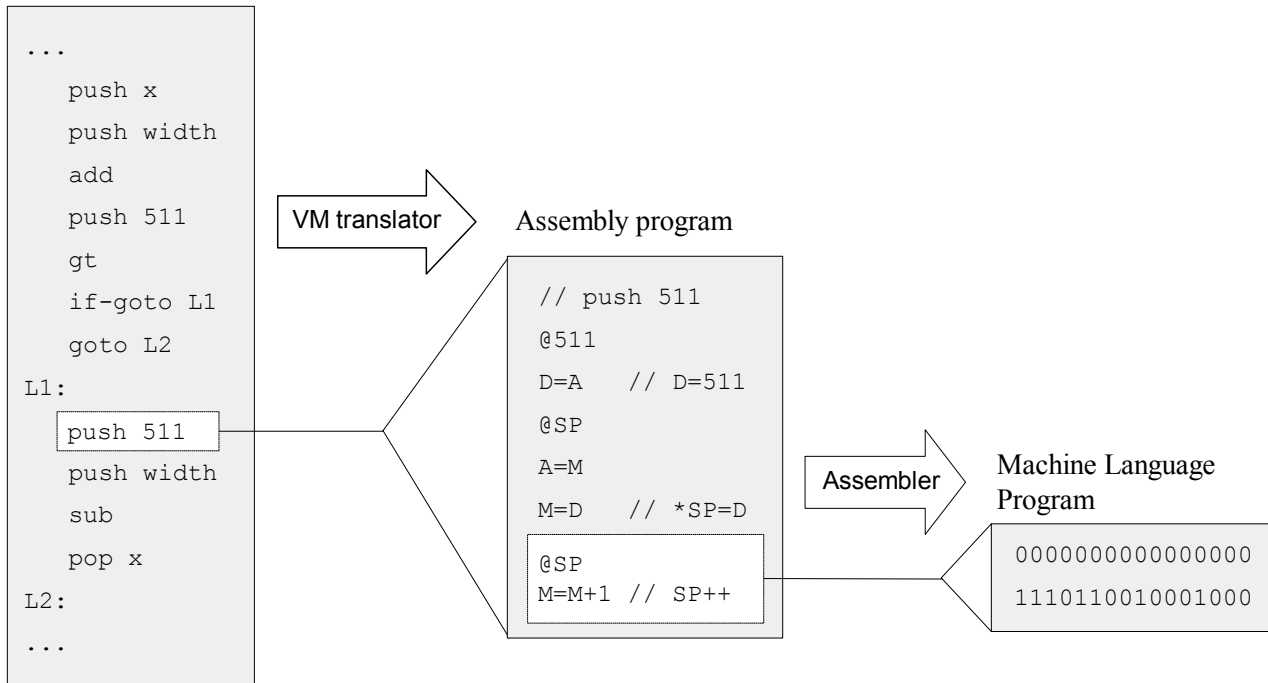
VM Implementation: One way to implement VM programs on a target hardware platform is to translate the VM code into the platform's native code. The program that carries out the translation -- *VM translator* -- is a stand-alone module which is based of two interfaces: the specification of the source VM language, and the specification of the target machine language. Yet in the larger picture of our grand tour, the VM translator can also be seen as the backend module of a two-stage compiler. First, the compiler described in the previous section translates the high level program into an intermediate VM code. Next, the VM translator translates the VM code into the native code of the target computer. This two-stage compilation model has many virtues, in particular code portability. Indeed, virtual machines and VM translators are becoming a common layer in modern software hierarchies, Java and .NET being two well-known examples.

In addition to its practical relevance, the study of virtual machine implementations is an excellent way to get acquainted with several classical computer science topics. These include program translation, push-down automata, and implementation of stack-based data structures. We will spend chapters 6 and 7 explaining these ideas and techniques, while building a VM implementation for the Hack platform. Of course Hack is just one possibility. The same VM can be realized on personal computers, cellular telephones, game machines, and so on. This cross-platform compatibility will require the development of different VM translators, one for each target platform.

Low-Level Programming Sampler

Every hardware platform is equipped with a native instruction set that comes in two flavors: *machine language* and *assembly language*. The former consists of binary instructions that humans (unlike machines) find difficult to read and write. The latter is a symbolic version of the former, designed to bring low-level programming closer to human comprehension. Yet the assembly extension is mainly a syntactical upgrade, and writing and reading assembly programs remains an obscure art. As Fig. 6 illustrates, Hack programming is no exception.

Virtual machine program



PROGRAM 6: From VM to assembly to binary code. There is no need to understand the code segments. Instead, it is enough to appreciate the big picture, which depicts a cascading translation process.

When we translate a high-level program into machine language, each high-level command is implemented as several low-level instructions. If the translator generates this code in assembly, the code has to be further translated into machine language. This translation is carried out by a program called *assembler*.

In order to read low-level code, one must have an abstract understanding of the underlying hardware platform -- in our case Hack. The Hack computer is equipped with two registers named *D* and *A* and a Random Access Memory unit consisting of 32K memory locations. The hardware is wired in such a way that the RAM chip always selects the location whose address is the current value of the *A*-register. The selected memory location -- $\text{RAM}[A]$ -- is denoted *M*. With this notation in mind, Hack assembly commands are designed to manipulate three registers named *A*, *D*, and *M*. For example, if we want to add the value stored in memory location 75 to the *D*-register, we can issue the two commands “set *A* to 75” and “set *D* to $D+M$ ”. The Hack assembly language expresses these commands as “@75” and “ $D=D+M$ ”, respectively. The rationale behind this syntax will become clear when we will build the Hack chips-set in chapters 2 and 3.

One extension that makes assembly languages rather powerful is the ability to refer to memory locations using user-defined labels rather than fixed numeric addresses. For example, let us assume that we can somehow tell the assembler that in this program, the symbol “*sp*” stands for memory location 0. This way, a high-level command like “*sp++*” could be translated into the two assembly instructions “@*sp*” and “ $M=M+1$ ”. The first instruction will cause the computer to select $\text{RAM}[0]$, and the second to add 1 to the contents of $\text{RAM}[0]$.

We end this section with Fig. 7, which describes the semantics of Program 6. This discussion is optional, and readers can skip it without losing the thread of the chapter.

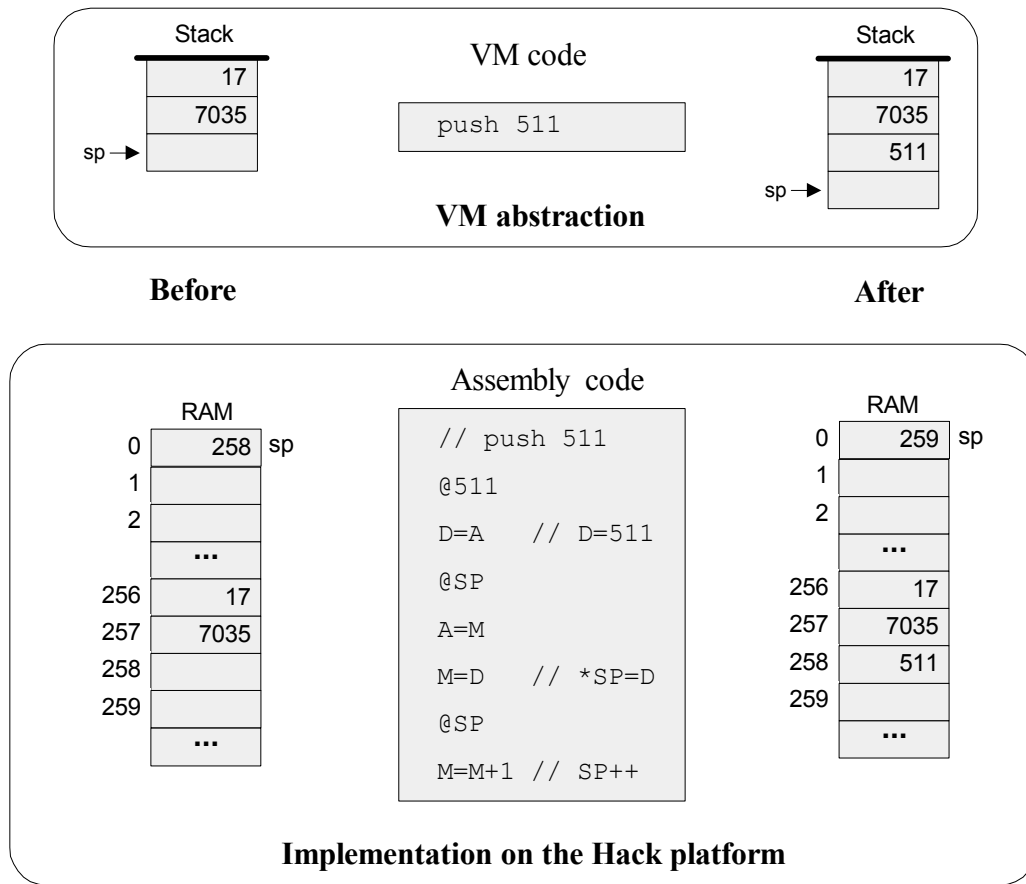


FIGURE 7: A typical abstract VM operation and its equivalent implementation on the Hack platform. The Hack code was created by the VM translator. We assume that the stack contains two arbitrary values (17 and 7035), and we track the pushing of 511 to the stack’s top. Note that among other things, the VM translator maps the stack-base and the stack-pointer on `RAM[256]` and `RAM[0]`, respectively.

Exploring the Assembly and the Machine languages

Although assembly is a low-level language that operates only a notch above the hardware, it is also an abstraction. After all, an assembly program is simply a bunch of symbols written on paper, or stored on disk. In order to turn these symbols into an executable program, we must translate them into binary instructions. This can be done rather easily, since the relationships between the machine’s binary and symbolic codes is readily available from the hardware specification.

For example, the Hack computer uses two types of 16-bit instructions. The left-most bit indicates which instruction we’re in: “0” for an *address* instruction and “1” for a *compute* instruction. In the case of an *address* instruction, the remaining 15 bits specify a number which is typically interpreted as an address. Thus, according to the language definition, the binary instruction

“0000000000010111”, whose agreed-upon assembly code is “@23”, implies the operation “set the A-register to 23” (10111 in binary is 23 in decimal). In a similar fashion, if the symbol “sp” happens to point to address 0 in the RAM, the assembly instruction “@sp” will be equivalent to “@0”, yielding “0000000000000000” in binary, which means “set the A-register to 0”.

The second Hack instruction, called *compute*, has the assembly format “dest=comp; jump”. This specification answers three questions: what to compute (*comp*), where to store the computed value (*dest*), and what to do next (*jump*). Altogether, the language specification includes 28 *comp*, 8 *dest*, and 8 *jump* directives, and each one of them can be specified using either a binary code or a symbolic mnemonic. For example, the *comp* directive “compute M-1” is coded as “0110010” in binary and as “M-1” in assembly. The *dest* directive “store the result in M” is coded as “001” in binary and as “M” in assembly. The *jump* directive “no jump” is coded as “000” in binary and as a null instruction field in assembly. Finally, the language specification says how the *comp*, *dest*, and *jump* fields should be mapped on the 16-bit machine instruction. Assembling all these codes together, we get the example shown in Fig. 8.

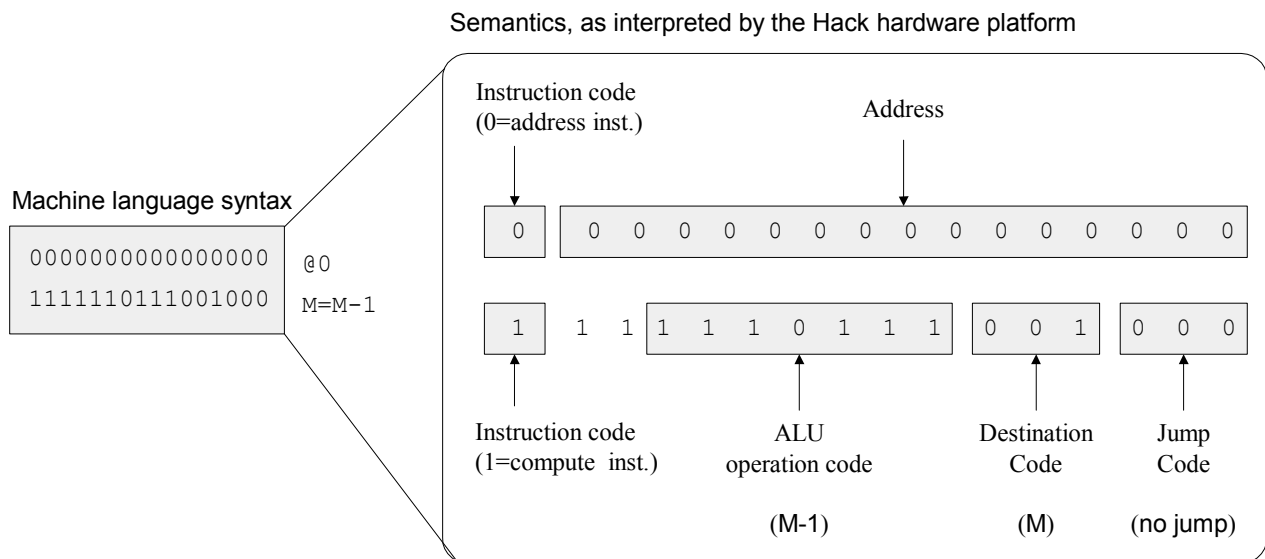


FIGURE 8: Instruction semantics in the Hack platform (example, focusing on two sample instructions). Note that the second and third most-significant bits in the *compute* instruction are not used, and are set to 1 as a language convention.

We see that the relationship between assembly and binary codes is a simple syntactical contract. Thus, if we are given a program written in assembly, we can convert each symbolic mnemonic to its respective binary code, and then assemble the resulting codes into complete binary instructions. This straightforward text processing task can be easily automated, and thus we can write a computer program to do it -- an *assembler*. The design of assembly languages, symbol tables and assemblers is the subject of Chapter 4. As the chapter progresses, we will build an assembler for the Hack platform.

We have reached a landmark in our Grand Tour -- the bottom of the software hierarchy. The next step down the abstraction-implementation route will take us into a new territory -- the top of the

hardware hierarchy. The linchpin that connects these two worlds is the *hardware architecture*, designed to realize the semantics of the machine language software.

3. The Journey ends: Hardware Land

Let us pause for a moment to appreciate where we stand in our journey. A program written in a high level language, represented in an intermediate VM code, has been translated to binary code, which should now run on a computer platform. Somehow, these various hardware/software modules (that in reality may well come from different companies) must work together flawlessly, delivering the intended program functionality. The key to success in building this remarkable complex is modular design, based on a series of contract-based, local, abstraction-implementation steps. And the most profound step in this journey is the descent from machine language to the machine itself -- the point where software finally meets hardware. One such hardware platform is seen in Diagram 9. Why did we choose this particular architecture?

Computer Architecture Tour

Almost all digital computers are built today according to a classical framework known as the *Von Neumann* model. Thus, if you want to understand computer architectures without taking a full semester course on the subject, your best bet is to study the main features of this fundamental model. In that respect, our *Hack* computer strikes a good balance between power and simplicity. On the one hand, Hack is a simple Von Neumann computer that a student can build in one or two days of work, using the chips-set that we will build in chapters 1-3. On the other hand, Hack is sufficiently general to illustrate the key operating principles and hardware elements of any digital computer.

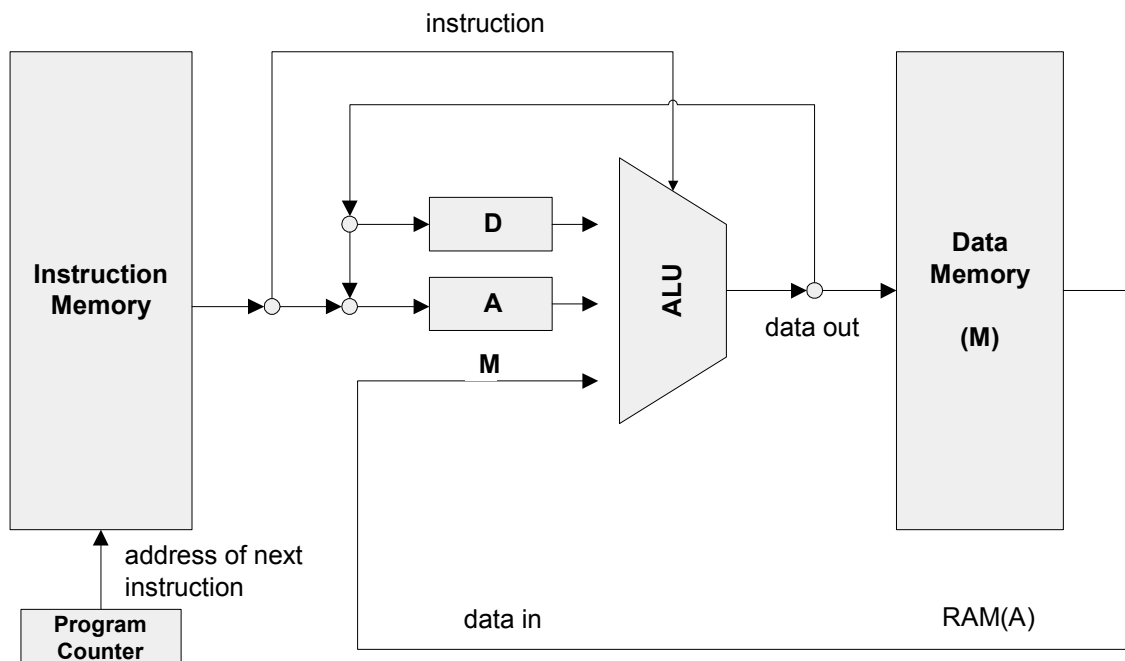


DIAGRAM 9: The Hack computer platform (overview), focusing on main chips and main data and instruction busses. To minimize clutter, the diagram does not show the control logic, the connection between the A-register and the data memory, and the connection between the A-register and the Program Counter.

The Hack computer is based on two memory units with separate address spaces, an ALU (Arithmetic Logic Unit), two registers, and a program counter. The centerpiece of the architecture is the *ALU* -- a “calculator” chip that can compute many functions of interest on its inputs. The *Instruction Memory*, containing the instructions of the current program, is designed to emit the value of the memory location whose address is the current value of the *Program Counter*. The *Data Memory*, containing the data on which the program operates, is designed to select, and emit the value of, the memory location whose address is the current value of the *A-register*. The overall computer operation, known as the *fetch-execute cycle*, is as follows.

Execute: first, the instruction that emerged from the instruction memory is simultaneously fed to both the A-register and the ALU. If it’s an *address* instruction (most significant bit = 0), the A-register is set to the instruction’s 15-bit value and the instruction execution is over. If it’s a *compute* instruction (MSB=1), then the 7 bits of the instruction’s `comp` field tell the ALU which function to compute. For example, as a convention, the code “0010011” instructs the ALU to compute the function “D-A” (the Hack ALU can compute 28 different functions on subsets of A, D, and M). The ALU output is then simultaneously routed to A, D, and M. Each one of these registers is equipped with a “load bit” that enables/disables it to incoming data. These bits, in turn, are connected to the 3 `dest` bits of the current instruction. For example, the `dest` code “101” causes the machine to enable A, disable D, and enable M to the ALU output.

Fetch: What should the machine do next? this question is determined by a simple control logic unit that probes the ALU output and the 3 `jump` bits of the current instruction. Taken together, these inputs determine if a jump should materialize. If so, the Program Counter is set to the value of the A-register (effecting a jump to the instruction pointed at by A). If no jump should occur, the Program Counter increments by 1 (no jump). Next, the instruction that the program counter points at emerges from the instruction memory, and the cycle continues.

Confused? Not to worry. We will spend all of chapter 5 explaining and building this architecture, one hardware module at a time. Further, you’ll be able to test your chips separately, making the overall computer construction surprisingly simple. The actual construction of all the hardware elements will be done using *Hardware Description Language* (HDL) and a *hardware simulator*, as we now turn to describe.

Gate Logic Appetizer

An inspection of the computer architecture from Diagram 9 reveals two types of hardware elements: *memory devices* (registers, memories, counters), and *processing devices* (the ALU). As it turns out, all these devices can be abstracted by Boolean functions, and these functions, in turn, can be realized using *logic gates*. The general subject of *logic design*, also called *digital design*, is typically covered by a full-semester course. We devote a quarter of the book to this subject (chapters 1-3), discussing the essentials of Boolean functions, combinational logic, and sequential logic. The following is a preview of some of the ideas involved.

Memory devices: A storage device, also called *register*, is a time-based abstraction consisting of a data input, a data output, and an input bit called *load*. The register is built in such a way that its output emits the same value over time, unless the load bit has been asserted, in which case the output is set to a new input value. In most computer architectures, this abstraction is implemented

using a primitive gate called *D-flip-flop*, which is capable of “remembering” a single bit over time. More complex registers are then built on top of this gate, as seen in Fig. 10.

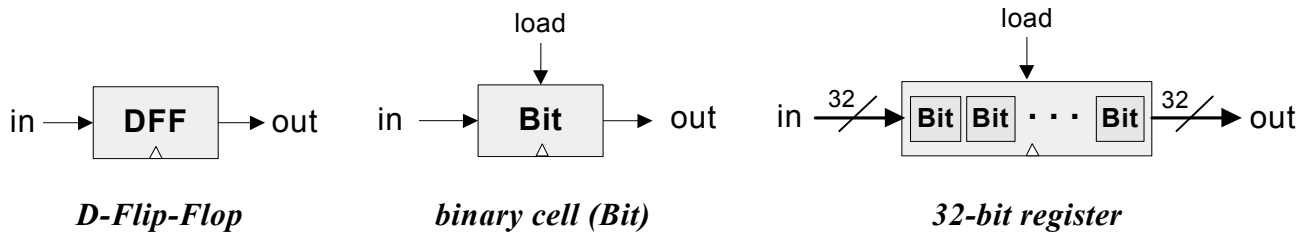


FIGURE 10: From flip-flop gates to multi-bit registers. A single-bit binary cell (also called `Bit` gate) is essentially a *D-flip-flop* with a loading capability. A multi-bit register of width w can be built from w `Bit` gates. (time-based chips are denoted by a small triangle, representing the clock input.)

What about Random-Access Memories? Well, a RAM device of length n and width w can be constructed as an array of n w -bit registers, equipped with direct-access logic. Indeed, *all* the memory devices of the computer -- registers, memories, and counters -- can be built by recursive ascent from D-Flip-Flops. These construction methods will be discussed in Chapter 3, where we use them to build all the memory chips of the Hack platform.

Processing devices: All the arithmetic operations of the ALU, e.g. $A+D$, $M+1$, $D-A$, and so on, are based on *addition*. Thus if you know how to add two binary numbers, you can build an ALU. How then do we add two binary numbers? Well, we can do it exactly the same way we learned to add decimal numbers in elementary school: we add the digits in each position, right to left, while propagating the carry to the left. Fig. 11 gives a Boolean logic implementation of this algorithm.

(Example)	(Definition)																				
$a:$ 1 0 0 1 (9)	<table style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="border-right: 1px solid black; border-bottom: 1px solid black;">a</th> <th style="border-right: 1px solid black; border-bottom: 1px solid black;">b</th> <th style="border-bottom: 1px solid black;">$Sum(a,b)$</th> <th style="border-bottom: 1px solid black;">$Carry(a,b)$</th> </tr> </thead> <tbody> <tr> <td style="border-right: 1px solid black;">0</td> <td style="border-right: 1px solid black;">0</td> <td>0</td> <td>0</td> </tr> <tr> <td style="border-right: 1px solid black;">0</td> <td style="border-right: 1px solid black;">1</td> <td>1</td> <td>0</td> </tr> <tr> <td style="border-right: 1px solid black;">1</td> <td style="border-right: 1px solid black;">0</td> <td>1</td> <td>0</td> </tr> <tr> <td style="border-right: 1px solid black;">1</td> <td style="border-right: 1px solid black;">1</td> <td>0</td> <td>1</td> </tr> </tbody> </table>	a	b	$Sum(a,b)$	$Carry(a,b)$	0	0	0	0	0	1	1	0	1	0	1	0	1	1	0	1
a		b	$Sum(a,b)$	$Carry(a,b)$																	
0		0	0	0																	
0		1	1	0																	
1		0	1	0																	
1		1	0	1																	
$b:$ 0 1 0 1 (5)																					
carry bit: 0 0 0 1																					
shifted carry bit: 0 0 0 1 0																					
sum bit: 1 1 0 0																					
$a+b:$ 1 1 1 0 (14)																					

Note: $a+b = Sum(\text{shifted carry bit}, \text{sum bit})$

FIGURE 11: Binary addition by Boolean logic

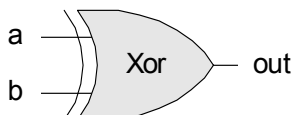
We see that binary addition can be viewed as a Boolean function, defined in terms of two simpler Boolean functions: *Sum* and *Carry*. Said otherwise, the addition operation can be implemented by an `Adder` chip, based on two lower-level chips: `Sum` and `Carry`. We note in passing that the adder chip and the ALU know nothing about “adding numbers”, neither do they know anything about “numbers” to begin with. Rather, they simply manipulate Boolean functions in a way that *effects* an addition operation (ideally, as quickly as possible).

Continuing in our reductive descent, how then should we implement the lower-level *Sum* and *Carry* abstractions? For brevity, let us focus on *Sum*. An inspection of this function's truth table reveals that it is identical to that of the standard *exclusive-or* function, denoted *Xor*. This function returns 1 when its two inputs have opposing values and 0 otherwise. The next section shows how the *Xor* abstraction can be implemented using *Hardware Description Language*.

Chip Design in a Nutshell

Like all the other artifacts encountered in our long journey, a chip can be described in two different ways. The chip *abstraction* -- also called *interface* -- is the set of inputs, outputs, and input-output transformations that the chip exposes to the outside world. The chip *implementation*, on the other hand, is a specification of a possible internal structure, designed to realize the chip interface. This dual view is depicted in Diagram 12.

Chip Abstraction (interface)



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

Possible chip Implementation

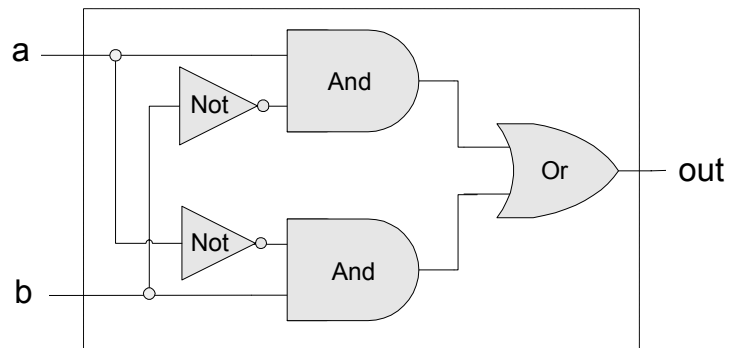


DIAGRAM 12: Chip design, using *Xor* as an example. The shown design is based on the Boolean function $Xor(a,b) = (a \text{ And } \text{Not}(b)) \text{ Or } (\text{Not}(a) \text{ And } b)$. Other *Xor* implementations are possible, some involving less gates and connections.

As usual, the chip abstraction is the right level of detail for people who want to *use* the chip as an off-the-shelf, black box component. For example, the designers of the adder chip described in the previous section need not know anything about the internal structure of *Xor*. All they need to know is the chip *interface*, as shown on the left side of Diagram 12. At the same time, the people who have to *build* the *Xor* chip must be given some building plan, and this information is contained in the chip *implementation* diagram. Note that this implementation is based on connecting *interfaces* of lower level abstractions -- those of the *Not*, *And*, and *Or* gates.

Hardware Description Language: How can we turn a chip Diagram into an actual chip? This task is commonly done today using a design tool called *Hardware Description Language*. HDL is a formalism used to define and test chips: objects whose interfaces consist of input and output pins that carry Boolean signals, and whose bodies are composed of inter-connected collections of other, lower level, chips. Program 13 gives an example.

```

CHIP Xor {
  IN a,b;
  OUT out;
  PARTS:
  Not (in=a, out=Nota);
  Not (in=b, out=Notb);
  And (a=a, b=Notb, out=aNotb);
  And (a=Nota, b=b, out=bNota);
  Or (a=aNotb, b=bNota, out=out);
}

```

PROGRAM 13: Typical HDL program, describing the Xor implementation from Diagram 12. The labels *Nota*, *Notb*, *aNotb* and *bNota* define the connections of the lower-level gates.

The HDL program gives a complete logical specification of the chip topology, describing all the lower-level components and connections of the chip architecture. This program can be simulated by a *hardware simulator*, to ensure that the structure that it implies delivers the required chip functionality. If necessary, the HDL program can be debugged and improved. Further, it can be fed into an *optimizer program*, in an attempt to create a functionally equivalent chip geometry that includes as few gates and wire crossovers as possible. Finally, the verified and optimized HDL program can be given to a fabrication facility that will stamp it in silicon.

The reader may wonder how HDL scales up to deal with realistically complex chips. Well, the Hack hardware platform consists of some 20 chips, and every one of them can be described in less than one page of HDL code. As usual, this parsimony is facilitated by modular design.

The Nand Gate: An inspection of Program 13 raises the question: And what about lower-level gates like *And*, *Or*, and *Not*? Well, they, too, can be constructed in HDL from more primitive gates. Clearly, this recursive descent must stop somewhere, and in this book it stops at the *Nand* level.

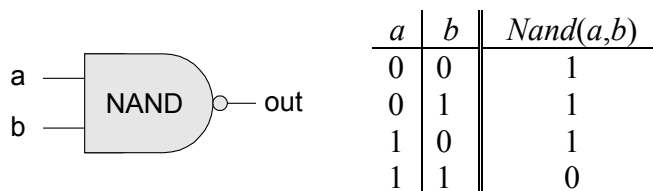


DIAGRAM 14: Nand gate (Last stop of our Grand Tour)

The *Nand* gate, implementing the trivial Boolean function depicted above, has two important properties. First, it can be modeled in silicon directly and efficiently, using 4 transistors. Second, as we will show in Chapter 1, any logic gate, and thus any conceivable chip, can be constructed recursively from (possibly many) *Nand* gates. Thus, *Nand* gates provide the cement from which all hardware systems can be built.

The Last Stop: Physics

Our Grand Tour has ended. In this book, the lowest level of abstraction that we reach is the Nand gate, which is viewed as primitive. Thus we descend no further, accepting the Nand implementation as given. Well, if we do want to peek downward, Diagram 15 shows an implementation of a Nand gate using CMOS (complementary metal-oxide semiconductor) technology. Drilling one layer lower, we reach the realm of solid-state physics, where we see how MOS transistors are constructed.

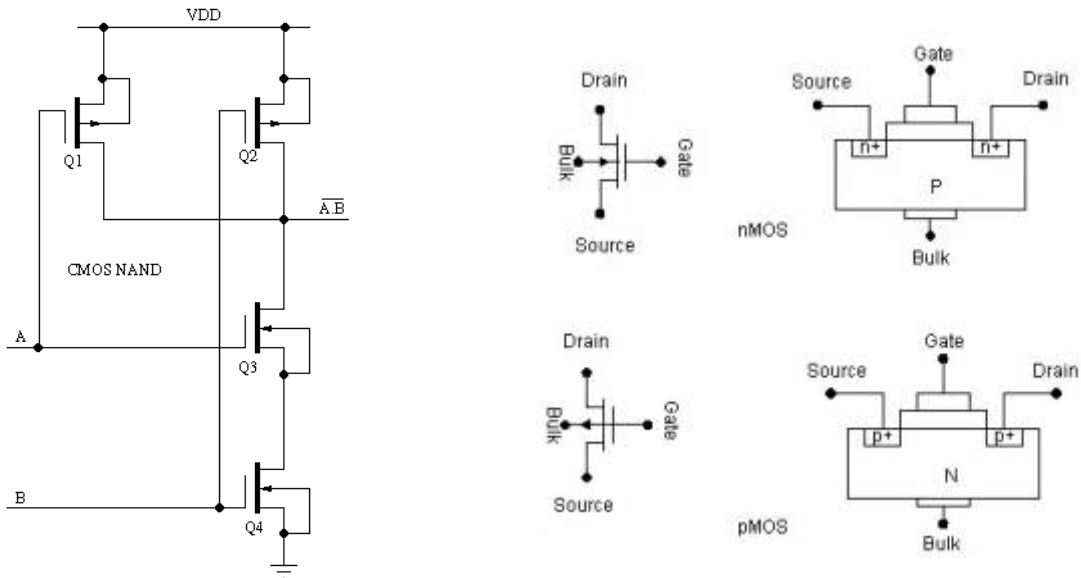


DIAGRAM 15: CMOS implementation of a Nand gate (left), based on 4 transistor abstractions. A possible MOS implementation of these transistors is shown on the right.

Asking how Nand gates are built is clearly an important question, and one that leads to many levels of additional abstractions. However, this journey will take us out of the synthetic worlds created by computer scientists, and into the natural world studied by statistical physics and quantum mechanics.

* * *

Back to the Mountain's Foot

This marks the end of our Grand Tour preview -- the descent from the high level regions of object-based software, all the way down to the bricks and mortar of the underlying hardware. In the remainder of the book we will do precisely the opposite. Starting with elementary logic gates (chapter 1), we will go bottom up to combinational and sequential chips (chapters 2-3), through the design of computer architectures (chapters 4-5) and software hierarchies (chapters 6-7), up to implementing modern compilers (chapter 9-10), high level programming languages (chapter 8), and operating systems (chapter 11). 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 count down start: **1, 0, Go!**