

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.

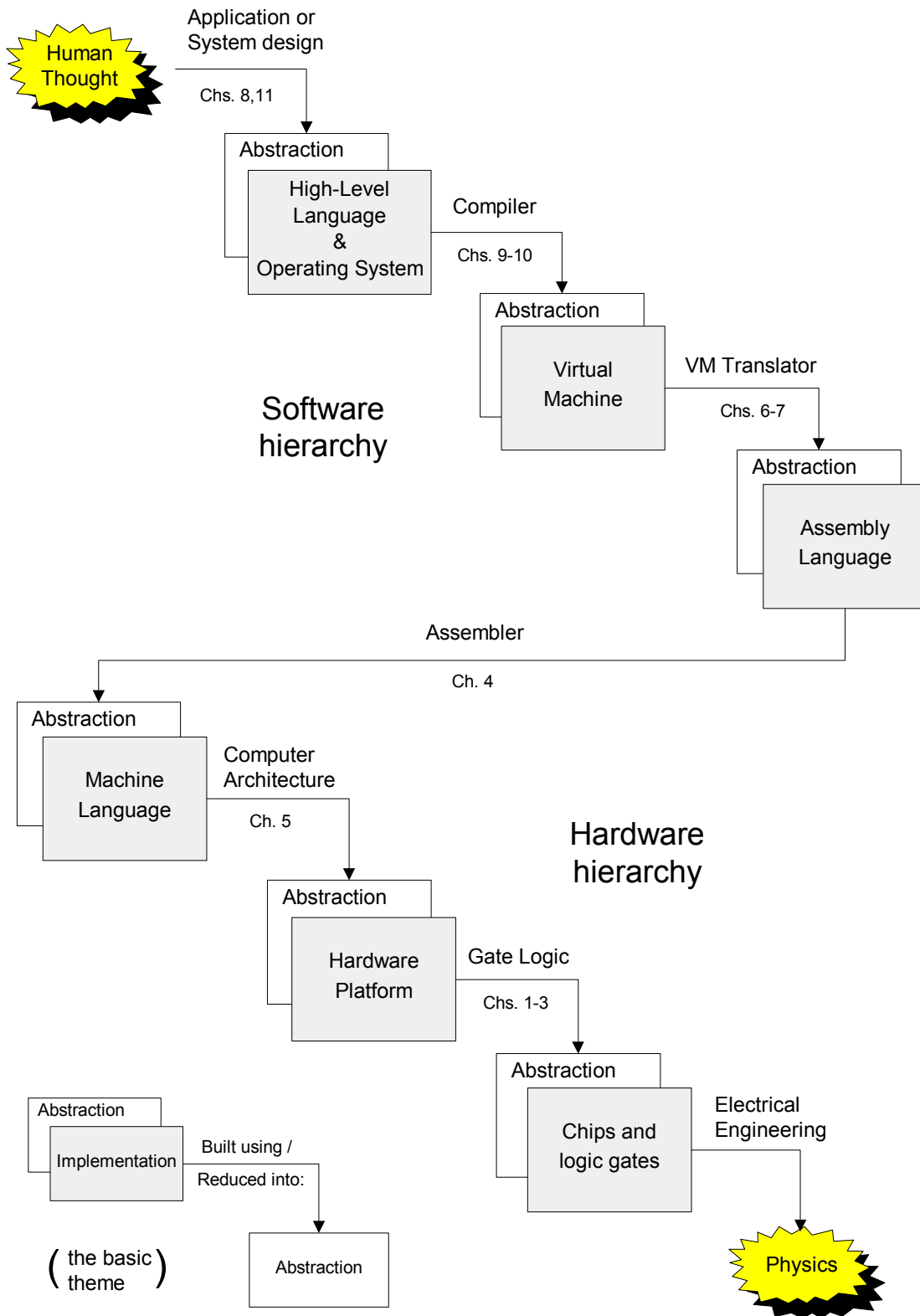


FIGURE 0: The Grand Tour (showing major stops only)

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.

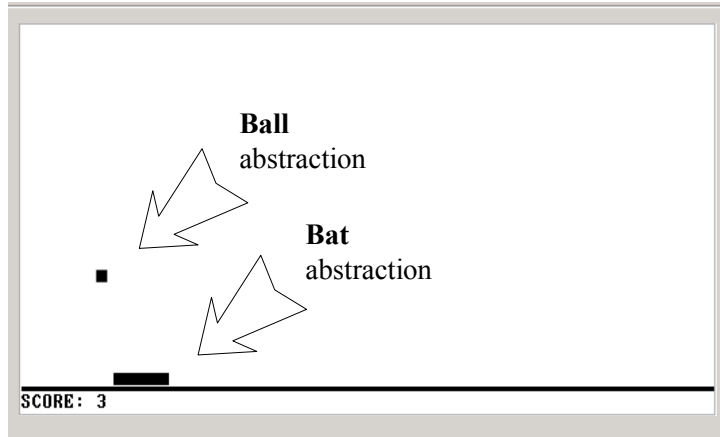


FIGURE 1: The Pong Game (annotated screen shot from a real game session, running on the *Hack* computer built in the book). A ball is moving on the screen randomly, bouncing off the screen “walls”. The user can move a small bat horizontally by pressing the keyboard’s left and right arrow keys. Each time 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.

```

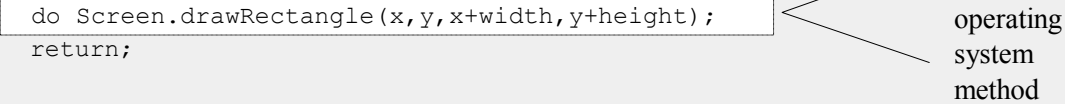
/** A Graphic Bat for a Pong Game */
class Bat {
    field int x, y;           // screen location of the bat's top-left corner
    field int width, height; // bat's width & height

    // 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.

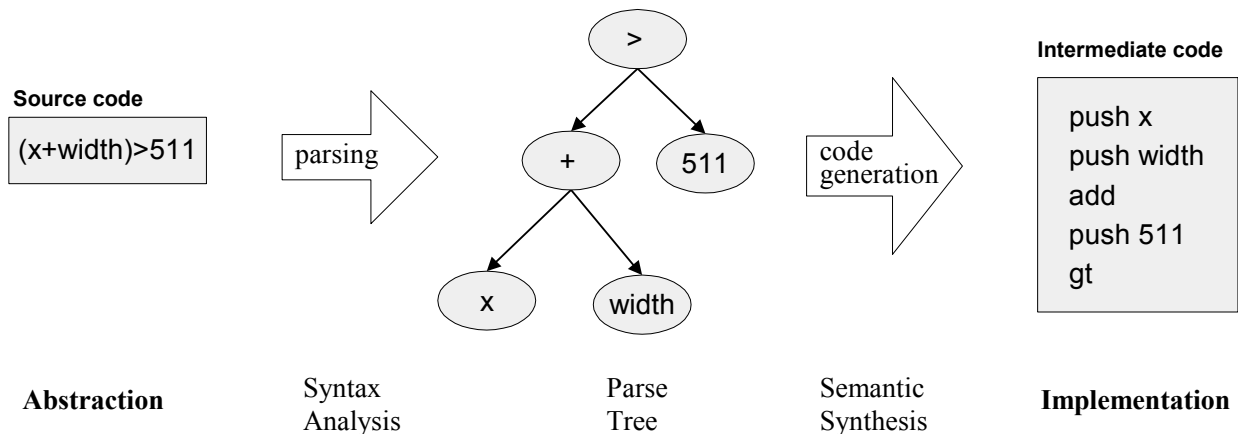


FIGURE 4: Compilation example

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:
...
```

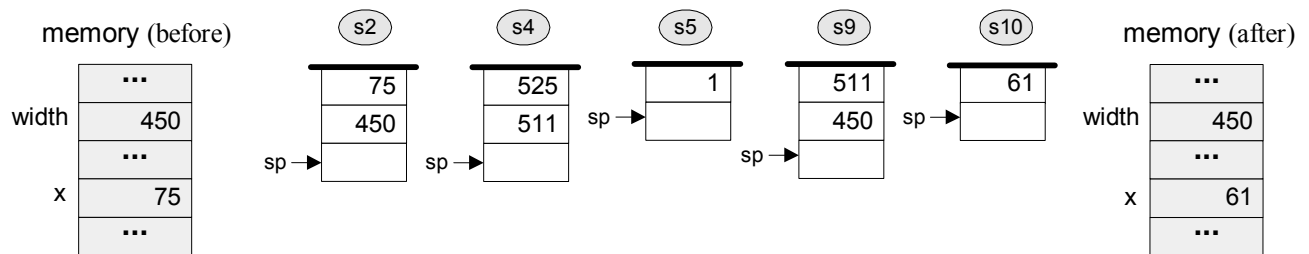


FIGURE 5: Virtual Machine code segment (top) and run-time scenario (bottom). To connect the two figures, we have annotated the VM commands and the stack images with state markers. (In stack diagrams, the next available slot is typically marked by the label *sp*, for *stack pointer*. Following convention,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

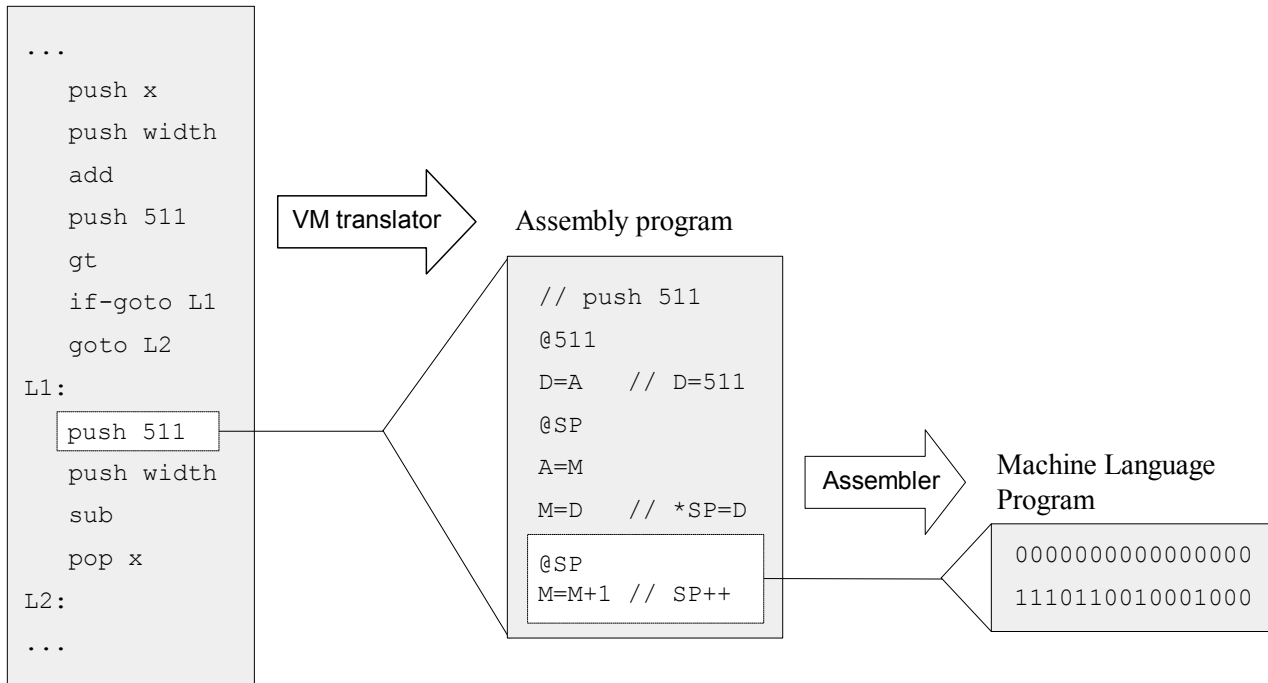
VM Implementation: One way to implement VM programs on a target hardware platform is to translate the VM code into the platform's native code. The program that carries out the translation -- *VM translator* -- is a stand-alone module which is based of two interfaces: the specification of the source VM 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.

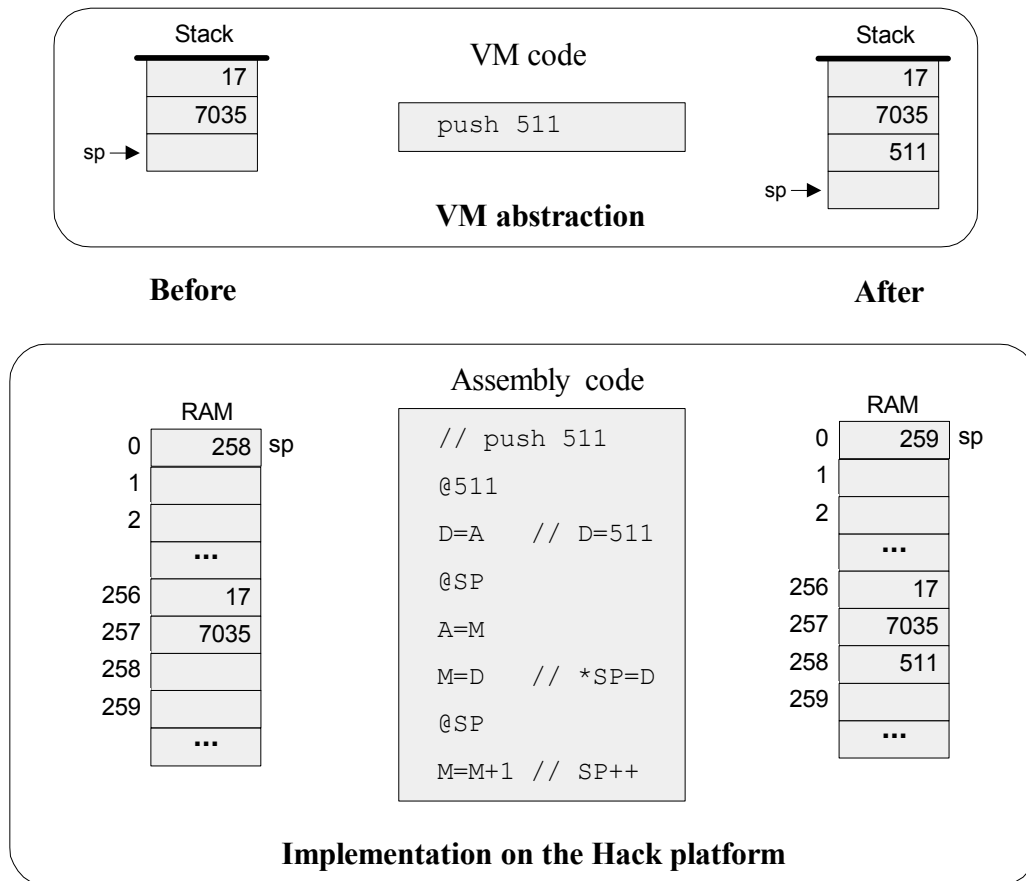


FIGURE 7: A typical abstract VM operation and its equivalent implementation on the Hack platform. The Hack code was created by the VM translator. We assume that the stack contains two arbitrary values (17 and 7035), and we track the pushing of 511 to the stack’s top. Note that among other things, the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

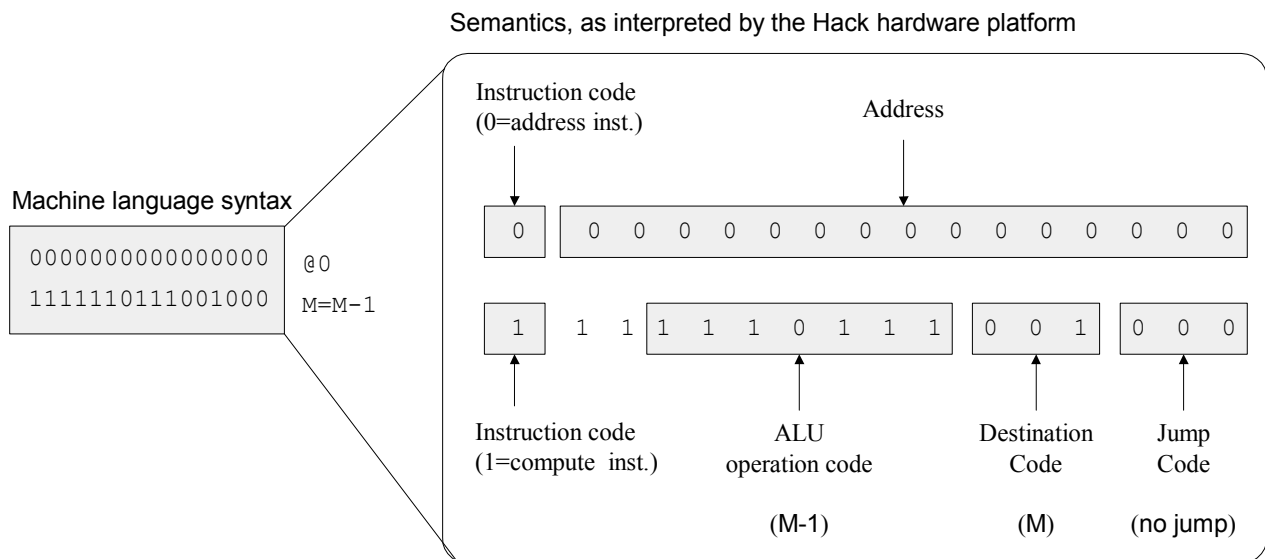


FIGURE 8: Instruction semantics in the Hack platform (example, focusing on two sample instructions). Note that the second and third most-significant bits in the *compute* instruction are not used, and are set to 1 as a language convention.

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.

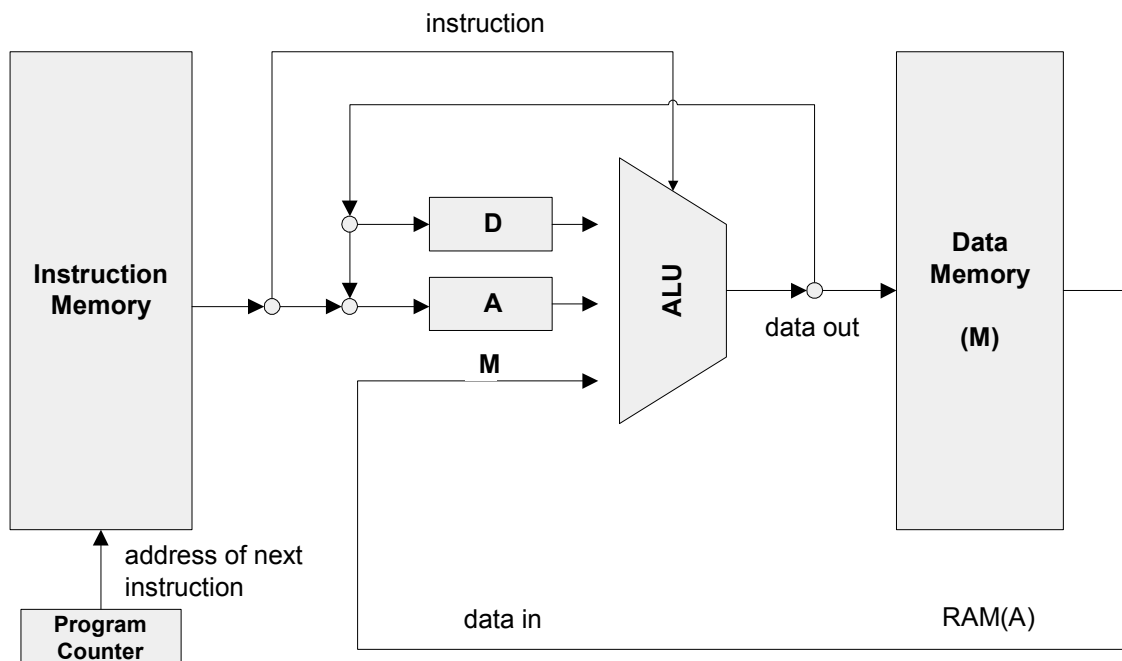


DIAGRAM 9: The Hack computer platform (overview), focusing on main chips and main data and instruction busses. To minimize clutter, the diagram does not show the control logic, the connection between the A-register and the data memory, and the connection between the A-register and the Program Counter.

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.

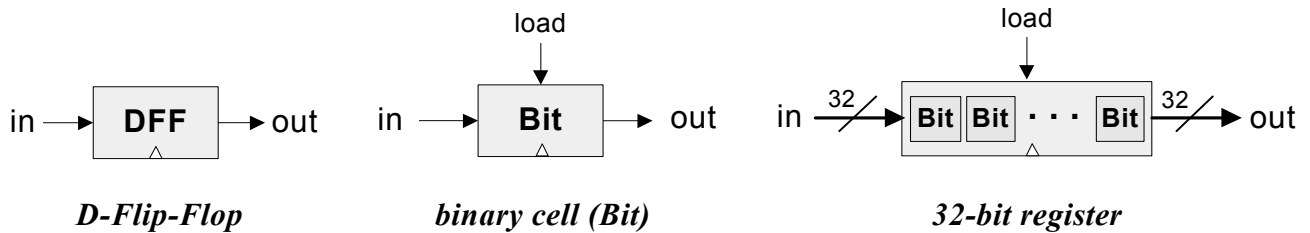


FIGURE 10: From flip-flop gates to multi-bit registers. A single-bit binary cell (also called `Bit` gate) is essentially a *D-flip-flop* with a loading capability. A multi-bit register of width w can be built from w `Bit` gates. (time-based chips are denoted by a small triangle, representing the clock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

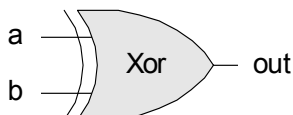
We see that binary addition can be viewed as a Boolean function, defined in terms of two simpler Boolean functions: *Sum* and *Carry*. Said otherwise, the addition operation can be implemented by an `Adder` chip, based on two lower-level chips: `Sum` and `Carry`. We note in passing that the adder chip 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

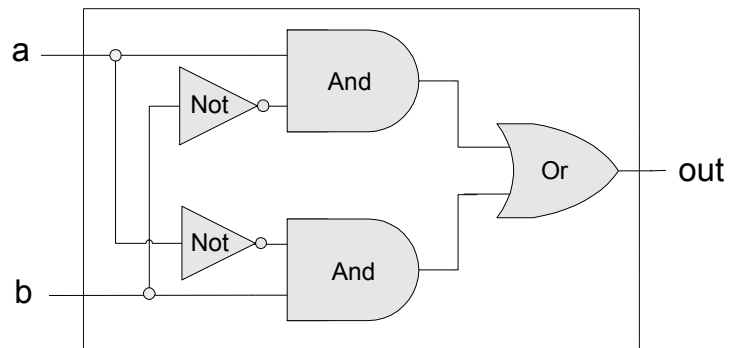


DIAGRAM 12: Chip design, using *Xor* as an example. The shown design is based on the Boolean function $Xor(a,b) = (a \text{ And } \text{Not}(b)) \text{ Or } (\text{Not}(a) \text{ And } b)$. Other *Xor* implementations are possible, some involving less gates and connections.

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.

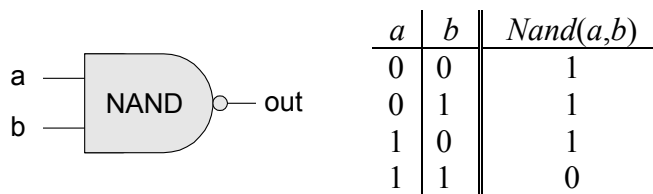


DIAGRAM 14: Nand gate (Last stop of our Grand Tour)

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.

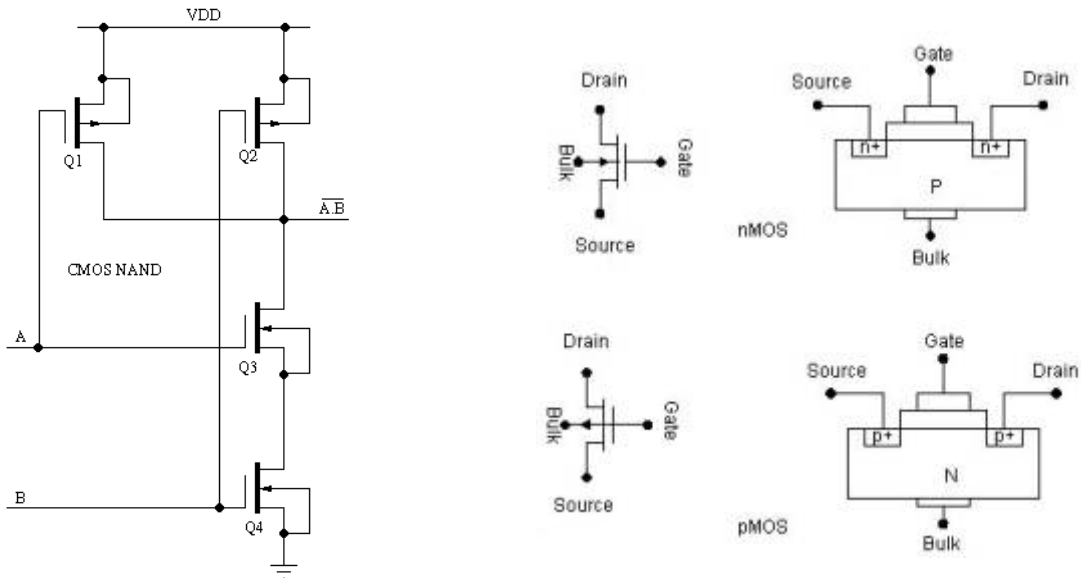


DIAGRAM 15: CMOS implementation of a Nand gate (left), based on 4 transistor abstractions. A possible MOS implementation of these transistors is shown on the right.

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!**

1. Boolean Logic¹

*Such simple things,
And we make of them something so complex it defeats us,
Almost.*

(John Ashbery, American poet, 1927-)

Every digital device – be it a personal computer, a cellular telephone, or a network router – is based on a set of chips designed to store and process information. Although these chips come in many different shapes and forms, they are all made from the same building blocks: elementary *logic gates*. The gates can be physically implemented in many different materials and fabrication technologies, but their logical behavior is consistent across all computers. In this chapter we start out with one primitive logic gate – Nand – and build all the other logic gates from it. The result will be a rather standard set of gates, which will be later used to construct our computer’s processing and storage chips. This will be done in chapters 2 and 3, respectively.

All the hardware chapters in the book, beginning with this one, have the same structure. Each chapter is focused on a well-defined *task*, designed to construct or integrate a certain family of chips. The prerequisite knowledge needed to approach this task is provided in a brief *Background* section. The next section provides a complete *Specification* of the chips abstraction, i.e. the various services that they should deliver, one way or another. Having presented the *what*, a subsequent *Implementation* section proposes guidelines and hints about *how* the chips can be implemented in practice. A *Perspective* section rounds up the chapter with comments on important topics that were left out from the discussion, with pointers to further reading and self-study. Each chapter concludes with a technical *Build It* section. This section gives step-by-step instructions for actually building the chips on your home computer, using the hardware simulator supplied with the book.

This being the first hardware chapter in the book, the *Background* section is somewhat lengthy, featuring a special section on *Hardware Description and Simulation Tools*.

1. Background

This chapter focuses on the construction of a family of simple chips called *Boolean gates*. Since Boolean gates are physical implementations of Boolean functions, we start with a brief treatment of Boolean logic and Boolean functions. We continue with a description of how Boolean gates can be inter-connected in order to achieve the complex functionality necessary for building typical hardware circuits. We conclude the background section with a description of how hardware design is actually undertaken in practice, using software simulation.

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

Boolean Functions and Boolean Algebra

Boolean algebra deals with Boolean (or binary) values that are typically labeled true/false, 1/0, yes/no, on/off, etc. We will use 1 and 0. A Boolean function is a function that operates on binary inputs and returns binary outputs. Since all computer hardware today is based on the representation and manipulation of binary values, Boolean functions play a central role in the specification, construction and optimization of hardware architectures. Hence, the ability to formulate and analyze Boolean functions is the first step toward constructing computer architectures.

Truth Table representation of a Boolean Function: The simplest way to specify a Boolean function is to provide a full enumeration of all the possible values of the function's input variables, along with the function's output for each set of inputs. This is called the *truth table* representation of the function. An example of a truth table for some arbitrary 3-input Boolean function is given in table 1.

x	y	z	$f(x, y, z)$
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

TABLE 1: Truth table of the Boolean function $f(x, y, z) = (x + y)\bar{z}$

The first three columns of Table 1 enumerate all the possible binary values of the function's variables. For each one of the 2^n possible tuples $v_1 \dots v_n$ (here $n=3$), the last column gives the value of $f(v_1 \dots v_n)$.

Boolean Expressions: In addition to the truth table specification, every Boolean function can also be specified using Boolean operations over its input variables. The basic Boolean operators that are typically used are “And” (“x And y” is 1 exactly when both x and y are 1) “Or” (“x Or y” is 1 exactly when either x or y or both are 1), and “Not” (“Not x” is 1 exactly when x is 0). We will use a common arithmetic-like notation for these operations: $x \cdot y$ (or xy) means “x And y”, $x + y$ means “x Or y”, and \bar{x} means “Not x”.

One may verify that the function whose truth-table was given in table 1 is equivalently given by the Boolean expression $f(x, y, z) = (x + y) \cdot \bar{z}$. For example, let us evaluate this expression on the inputs $x = 0$, $y = 1$, $z = 0$ (3rd row in the table). Since y is 1, it follows that $x + y = 1$ and thus $1 \cdot \bar{0} = 1 \cdot 1 = 1$. The complete verification of the equivalence between the expression and the truth table is achieved by evaluating the expression on each of the 8 possible input combinations, verifying that it yields the same value listed in the table's right column.

Canonical Representation: As it turns out, every Boolean function can be expressed using at least one Boolean expression called *canonical representation*. Starting with the function's truth table, we focus on all the rows for which the function has value 1. For each such row, we construct a term created by And-ing together *literals* (variables or their negations) that fix the values of all the row's inputs. For example, let us focus on row 3 in Table 1, where the function's value is indeed 1. Since the variable values in this row are $x = 0$, $y = 1$, $z = 0$, we construct the term $\bar{x} y \bar{z}$. Following the same procedure, we construct the terms $x \bar{y} \bar{z}$ and $x y \bar{z}$ for rows 5 and 7. Now, if we'll Or-together all these terms (for all the rows where the function has value 1), we get a Boolean expression that is equivalent to the given truth-table. Thus the canonical representation of the Boolean function shown in Table 1 is $f(x, y, z) = \bar{x} y \bar{z} + x \bar{y} \bar{z} + x y \bar{z}$.

This leads to an important conclusion: Every Boolean function, no matter how complex, can be expressed using three Boolean operators only: And, Or, and Not.

Two-input Boolean Functions: An inspection of Table 1 reveals that the number of Boolean functions that can be defined over n binary variables is 2^{2^n} . For example, the 16 Boolean functions spanned by two variables are listed in Table 2.

The 16 functions in Table 2 were constructed systematically, by enumerating all the possible 4-wise combinations of binary values in the four right columns. Each function has a conventional name that describes its underlying operation. Here are some examples: the name of the Nor function is shorthand for Not-Or: take the Or of x and y , then negate the result. The Xor function -- shorthand for "exclusive or" -- returns 1 when its two variables have opposing truth-values and 0 otherwise. Conversely, the Equivalence function returns 1 when the two variables have identical truth-values. The If x then y function (also known as $x \rightarrow y$, or "x Implies y") returns 1 when x is 0 or when both x and y are 1. The other functions are self-explanatory.

The Nand function (as well as the Nor function) has an interesting theoretical property: each one of the operations And, Or, and Not can be constructed from it (e.g. x Or $y = (x$ Nand $x)$ Nand $(y$ Nand $y)$). Since, as we have seen, every Boolean function can be constructed from And, Or, and Not operations using the canonical representation method, it follows that every Boolean function can be constructed from Nand operations alone. This result has important practical implications: once we have in our disposal a physical device that implements the Nand operation, we can implement in hardware any Boolean function.

Function	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	$x \cdot \bar{y}$	0	0	1	0
x	x	0	0	1	1
Not x And y	$\bar{x} \cdot y$	0	1	0	0
y	y	0	1	0	1
Xor	$x \cdot \bar{y} + \bar{x} \cdot y$	0	1	1	0
Or	$x + y$	0	1	1	1
Nor	$\overline{x + y}$	1	0	0	0
Equivalence	$x \cdot y + \bar{x} \cdot \bar{y}$	1	0	0	1
Not y	\bar{y}	1	0	1	0
If y then x	$x + \bar{y}$	1	0	1	1
Not x	\bar{x}	1	1	0	0
IF x then y	$\bar{x} + y$	1	1	0	1
Nand	$\overline{x \cdot y}$	1	1	1	0
Constant 1	1	1	1	1	1

TABLE 2: All the Boolean functions of two variables along with their common names, notations, and truth table definitions.

Gate Logic

A *gate* is a physical implementation of a Boolean function. Typically, gates are built from tiny switching devices, called *transistors*, wired in a certain topology designed to effect the gate functionality. Although most digital computers use electricity to represent and transmit binary data from one gate to another, any alternative technology permitting switching and conducting capabilities can be employed. Indeed, during the last 50 years researchers have built many hardware implementations of Boolean functions, including magnetic, optical, biological, hydraulic, and even tinker toy-based, mechanisms. Today, most gates are implemented as transistors etched in Silicon, packaged as *chips*. In this book we use the words *chip* and *gate* interchangeably, tending to use the term *gates* for simple chips.

The availability of alternative switching technology options, on the one hand, and the observation that Boolean algebra can be used to abstract the behavior of *any* such technology, on the other, is extremely important. Basically, it implies that computer scientists don't have to worry about physical things like electricity, circuits, switches, relays, and power supply. Instead, computer scientists can be content with the abstract notions of Boolean algebra and logic gates, trusting that someone else (the physicists and electrical engineers – god bless their souls) will figure out how

to actually realize them in hardware. Hence, a *primitive gate* (see Diagram 3) can be viewed as a black box device that implements an elementary logical operation in one way or another – we don't care how. A hardware designer starts from such primitive gates and designs more complicated functionality by inter-connecting them, leading to the construction of *composite gates*.

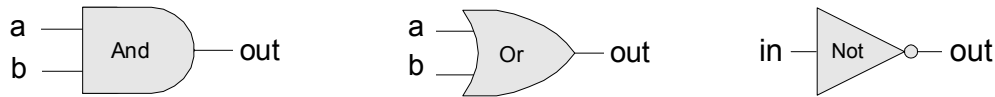


DIAGRAM 3: Standard symbolic notation of some elementary logic gates.

Primitive and Composite Gates: Since all logic gates have the same input and output semantics (0's and 1's), they can be chained together, creating *composite gates* of arbitrary complexity. For example, suppose we are asked to implement the 3-way Boolean function $\text{And}(a,b,c)$. Using Boolean algebra, we can begin by observing that $a \cdot b \cdot c = (a \cdot b) \cdot c$, or, using prefix notation, $\text{And}(a,b,c) = \text{And}(\text{And}(a,b),c)$. Next, we can use this result to construct the composite gate depicted in the right side of Diagram 4.

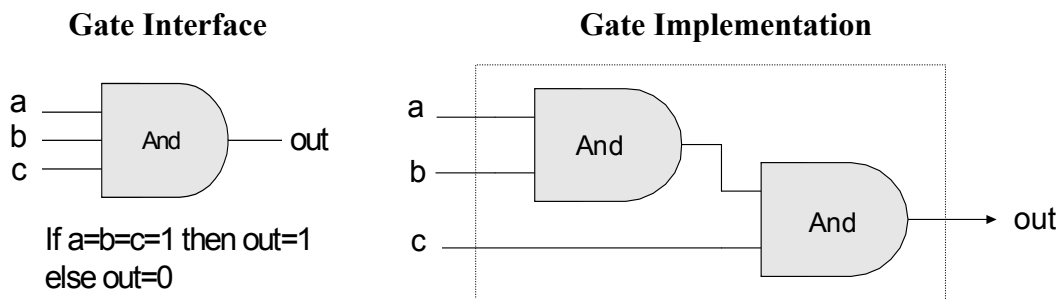


DIAGRAM 4: Composite implementation of a 3-way And gate, based on the observation $\text{And}(a,b,c) = \text{And}(\text{And}(a,b),c)$. The rectangle on the right defines the conceptual boundaries of the gate interface.

The construction described in Diagram 4 is a simple example of “gate logic,” also called “logic design”. Simply put, logic design is the art of inter-connecting elementary gates in order to implement more complex functionality, leading to the notion of *composite gates*. Since composite gates are themselves realizations of (possibly complex) Boolean functions, their “outside appearance” (e.g. left side of Diagram 4) looks just like that of primitive gates. At the same time, their internal structure can be rather complex.

We see that any given logic gate can be viewed from two different perspectives: external and internal. The right-hand side of Diagram 4 gives the gate's internal architecture, or *implementation*, whereas the left side shows only the gate *interface*, i.e. the input and output pins that it exposes to the outside world. The gate implementation diagram is relevant only to the gate designer, whereas the gate interface is the right level of detail for other designers who wish to use the gate as an abstract off-the-shelf component, without paying attention to its internal structure.

Let us consider another logic design example -- that of a Xor gate . As discussed before, $Xor(a,b)$ is 1 exactly when either a is 1 and b is 0, or when a is 0 and b is 1. Said otherwise, $Xor(a,b)=Or(And(a,Not(b)),And(Not(a),b))$. This definition leads to the logic design shown in Diagram 5.

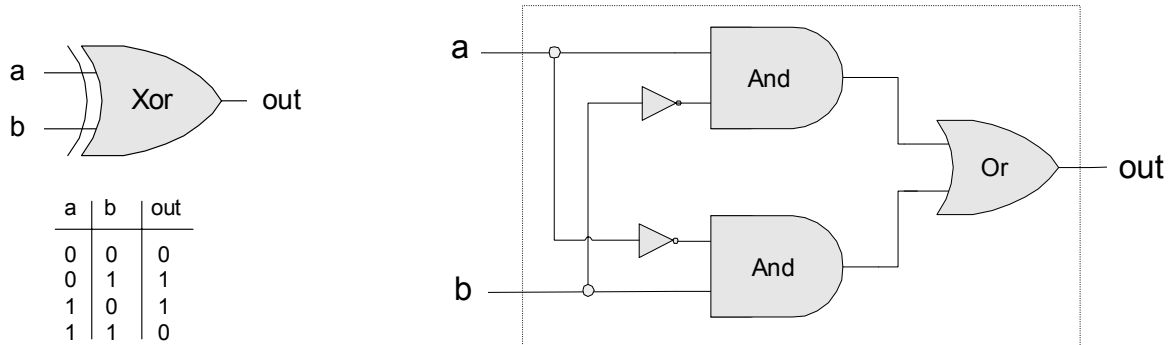


DIAGRAM 5: The Xor gate and a possible Xor implementation

We conclude this section with an important observation about the difference between *interface* and *implementation*. The gate *interface* is unique: there is only one way to describe it, and this is normally done using a truth table or some verbal specification. This interface, however, can be realized using many different *implementations*, and some of the resulting gate architectures will be better than others in terms of cost, speed, and simplicity. For example, the Xor function can be implemented using 3, rather than 5, internal gates. Thus, from a functional standpoint, the fundamental requirement of logic design is that *the gate implementation will realize the stated interface, in one way or another*. From an efficiency standpoint, the general rule is to try to do more with less, i.e. use as few gates as possible.

To sum up, the art of logic design can be described as follows: given a gate specification, find an efficient way to implement it using other gates that were already implemented. This, in a nutshell, is what the rest of the chapter is concerned with.

Actual Hardware Construction

Having described the logic of designing complex gates by composing more primitive ones, we now turn to describe how composite gates can be actually built. Let us start with an intentionally naïve example.

Suppose we open a chip production 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 plug. To get started, we pin the Xor gate diagram from Diagram 5 to our garage wall, and proceed to implement it using our hardware. First, we take two And gates, two Not gates, and one Or gate, and mount them on a board in more or less the same layout specified in the diagram. Next, we connect the

chips to each other by running copper 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 (or other people) will be asked to construct some other chips in the future, we'll be able to use these Xor gates as elementary building blocks, just like we used the And, Or, and Not gates before.

As the reader has 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 outputs will agree with its specifications. If the chip will fail to deliver the desired outputs, we will have to tinker its physical structure – a rather messy affair. Further, even if we will come up with the right design, replicating the chip assembly process many times over will be a time-consuming and error prone affair. There must be a better way!

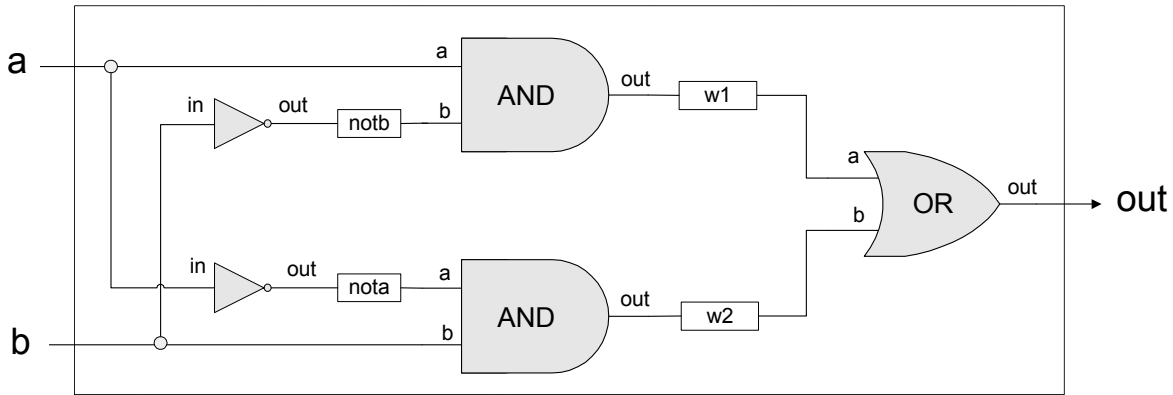
Hardware Description Language

Indeed there is. Today, hardware designers no longer build anything with their bare hands. Instead, they plan and optimize the chip architecture on a computer workstation, using a structured modeling formalism called *Hardware Description Language*, or HDL (also known as VHDL, where V stands for *Virtual*). The designer specifies the chip structure by writing an *HDL program*, which is then subjected to a rigorous battery of tests. These tests are carried out virtually, using computer simulation. That is to say, a special software tool, called *hardware simulator*, takes the HDL program as input, and builds an image of the implied chip in memory. Next, the designer can instruct the simulator to test the virtual chip on various sets of inputs, generating simulated chip outputs. The outputs can then be compared to the desired results, as mandated by the person 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 chip design (and this is perhaps the right place to point out that a Xor chip can be built with 3 elementary gates rather than 5, as we have done before). All these parameters can be simulated and quantified by the hardware simulator, helping the designer optimize the design until the simulated chip delivers desired performance levels.

Thus, using HDL, one can completely plan, debug and optimize the entire chip before a single penny is spent on actual production. When the HDL program is deemed complete, i.e. when the performance of the simulated chip satisfies the client who ordered it, 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 life cycle -- from an optimized HDL program to mass production – is typically outsourced to companies that specialize in chip fabrication.

Example: Building a Xor Gate: By definition, Xor returns true when its two inputs have opposing values, i.e. $Xor(a,b)=Or(And(a,Not(b)),And(Not(a),b))$. This logic can be expressed either graphically, as a gate diagram, or textually, as an HDL program (Fig. 10). The latter program is written in the HDL variant used throughout this book, which is completely defined in appendix A.



HDL program	Test script	Output file
<pre>// 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); }</pre>	<pre>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;</pre>	<pre>a b out ----- 0 0 0 0 1 1 1 0 1 1 1 0</pre>

FIGURE 10: HDL implementation of a Xor gate. Following a convention used in this book, the shown files are named Xor.hdl, Xor.tst, and Xor.out.

Explanation: An HDL definition of a chip consists of a *header* section and a *parts* section. The *header* section describes the chip *interface*, or *signature*, which specifies the chip name and the names of its input and output pins. The *parts* section describes the names and topology of all the lower-level chips from which this chip is constructed. Each part is represented by a *statement* that specifies the part name and the way it is connected to other parts in the design. Note that in order to write such statements legibly, the HDL programmer must have a complete description of the *interfaces* of the underlying parts. For example, Fig. 10 implies that the input and output pins of the Not gate are labeled in and out, and those of And and Or are labeled a,b and out. Without knowing this API-based convention, it would be impossible to plug these chip parts into the present code.

Inter-part connections are described 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 (wire) named `nota`, and then feeds out into it. The second statement feeds the value of the `nota` pin into the `a` input of an And gate. Note that pins may have an unlimited fan-out. For example, the input `a` is simultaneously fed into both an And gate and a Not gate. In gate diagrams, multiple connections are described using forks. In HDL, the existence of forks is implied by the code.

Testing: Rigorous quality assurance requires that chips will be tested in a specific, replicable, and well-documented fashion. With that in mind, hardware simulators are usually able to run *test scripts*, written in some scripting language. For example, the test script in Fig. 10 is written in the scripting language understood by the hardware simulator supplied with this book. Both the simulator and the scripting language are described fully in appendix B.

Let us give a brief description of the test script from Fig. 10. The first two lines of the test script instruct the simulator to load the `Xor.hdl` program and get ready to print the current values of selected variables. Next, the script lists a series of testing scenarios, designed to simulate the various contingencies under which the Xor gate will have to operate in "real life" situations. In each scenario, the script instructs the simulator to bind the gate inputs to certain data values, compute the gate 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 possible input values of the chip. The resulting output file (right side of Fig. 10) can then be viewed as a complete empirical proof that the chip is well-defined. The luxury of such certitude is not feasible in more complex chips, as we will see later.

Hardware Simulation

Although HDL is a hardware construction language, the process of writing and debugging an HDL program is quite similar to software development. The main difference is that instead of writing code in a language like Java we write it in HDL, and rather than using a compiler and a virtual machine to translate and test it, we use a *hardware simulator* instead. 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 the specifications of a given test script. There exist many commercial hardware simulators in the market, and these vary greatly in terms of cost, complexity, and ease of use. Together with this book we provide a simple (and free!) hardware simulator which is sufficiently powerful to illustrate all the key elements of the hardware design process. This simulator is all you need in order to build, test, and integrate all the chips presented in the book, leading to the construction of a powerful general-purpose computer. Fig. 11 illustrates a typical chip simulation session.

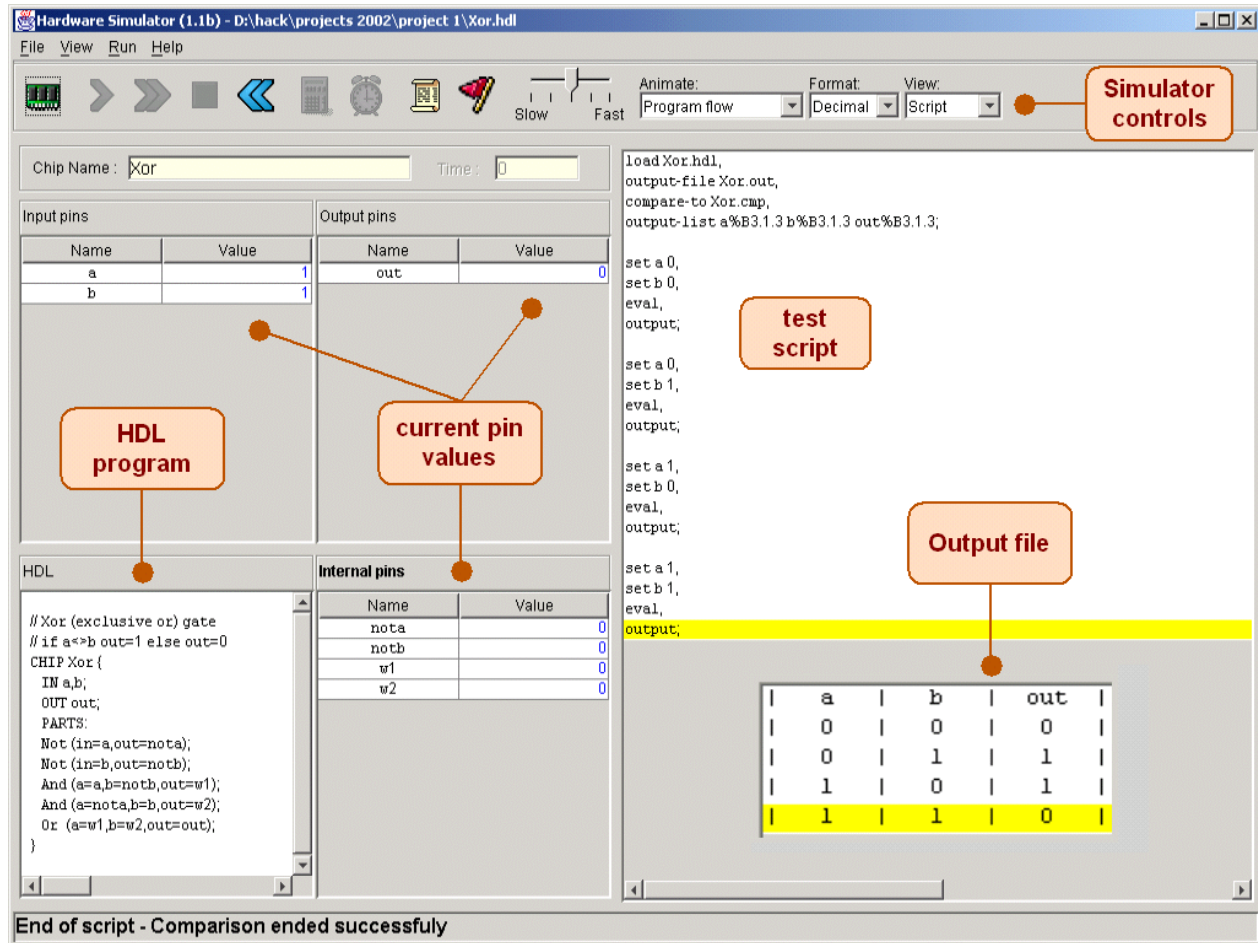


FIGURE 11: Chip Simulation. A screen shot of simulating a Xor chip on the hardware simulator. The simulator state is shown just after the test script stopped running. The pin values correspond to the last simulation step ($a = b = 1$). Note that the *output file* generated by the simulation is consistent with the Xor truth table, indicating that the HDL program provides a correct implementation of Xor. The *compare file*, not shown in the figure, has exactly the same structure and contents as that of the output file. The fact that the two files agree with each other is evident from the status message displayed at the bottom left of the screen.

2. Specification

This section specifies a typical set of gates, each designed to carry out a common Boolean operation. These gates will be used in the next chapters to construct the full architecture of a typical modern computer. Our starting point is a single primitive Nand gate, from which all other gates will be derived recursively. Importantly, we provide only the gates *specifications*, or *interfaces*, delaying *implementation* issues to a subsequent section. Readers who wish to construct the specified gates in HDL (*Hardware Description Language*) are welcome to do so, after reading Appendix A. The gates can be built and simulated on your home computer, using the hardware simulator supplied with the book.

The Nand gate

The starting point of our computer architecture is the Nand gate, from which all other gates and chips are built. The Nand gate is designed to compute the following Boolean function:

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

Throughout this section, we use "chip API boxes" to specify chips. For each chip, the API specifies the chip name, the names of its input and output pins, the function or operation that the chip effects, and an optional comment.

Chip name:	Nand
Inputs:	a,b
Outputs:	out
Function:	If a=b=1 then out=0 else out=1
Comment:	This gate is considered primitive and thus there is no need to implement it.

Basic Logic Gates

Some of the logic gates presented below are typically referred to as "elementary" or "basic." At the same time, every one of them can be composed from Nand gates alone. Therefore, they need not be viewed as primitive.

Not: The single-input Not gate, also known as "converter", converts its input from 0 to 1 and vice versa. The gate API is as follows:

Chip name:	Not
Inputs:	in
Outputs:	out
Function:	if in=0 then out=1 else out=0.

And: The And function returns 1 when both its inputs are 1, and 0 otherwise.

Chip name:	And
Inputs:	a,b
Outputs:	out
Function:	if a=b=1 then out=1 else out=0.

Or: The Or function returns 1 when at least one of its inputs is 1, and 0 otherwise.

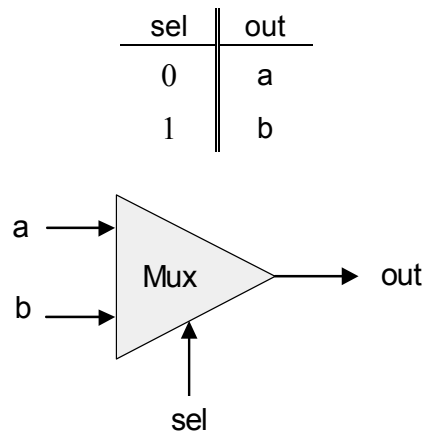
Chip name:	Or
Inputs:	a,b
Outputs:	out
Function:	if a=b=0 then out=0 else out=1.

Xor: The Xor function, also known as "exclusive or," returns 1 when its two inputs have opposing values, and 0 otherwise.

Chip name:	Xor
Inputs:	a,b
Outputs:	out
Function:	if a≠b then out=1 else out=0.

Multiplexor: A multiplexor is a 3-input gate that uses one of the inputs, called "selection bit", to select and output one of the other two inputs, called "data bits". Although a better name for this device could have been *selector*, the name *multiplexor* is commonly used since a similar device is used in telecommunications systems to combine (multiplex) several input signals over a single output wire.

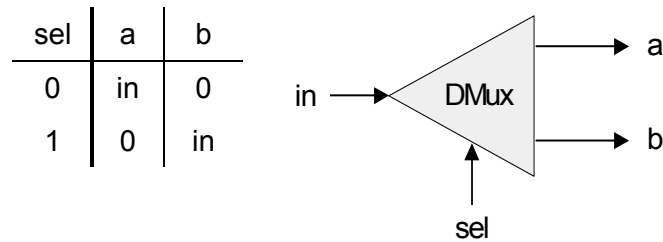
a	b	sel	out
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	1
0	0	1	0
0	1	1	1
1	0	1	0
1	1	1	1



Chip name:	Mux
Inputs:	a,b,sel
Outputs:	out
Function:	If sel=0 then out=a else out=b.

DIAGRAM 6: Multiplexor. The table at the top right is an abbreviated version of the truth table on the left.

Demultiplexor: A demultiplexor performs the opposite function of a multiplexor: it takes a single input and channels it to one of two possible output wires according to a selector input that specifies which input to chose.



Chip name:	DMux
Inputs:	in, sel
Outputs:	a, b
Function:	If sel=0 then {a=in, b=0} else {a=0, b=in}

DIAGRAM 7: Demultiplexor.

Multi-bit versions of basic gates

Computer hardware is typically designed to operate on multi-bit arrays called "buses." For example, a basic requirement of a 32-bit computer is to be able to compute (bit-wise) an And function on two given 32-bit busses. To implement this operation, we can build an array of 32 binary And gates, each operating separately on a pair of bits. In order to enclose all this logic in one package, we can encapsulate the gates array in a single chip interface consisting of two 32-bit input busses and one 32-bit output bus.

This section describes a typical set of multi-bit logic gates, as needed for the construction of a 16-bit computer. We note in passing that the architecture of n -bit logic gates is basically the same irrespective of n 's value.

When referring to individual bits in a bus, it is common to use an array syntax. For example, to refer to individual bits in a 16-bit bus named `data`, we use the notation `data[0]`, `data[1]`, ..., `data[15]`.

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

Chip name:	Not16
Inputs:	in[16]
Outputs:	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 one of the n bit-pairs drawn from its two n -bit input busses:

Chip name:	And16
Inputs:	a[16],b[16]
Outputs:	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 one of the n bit-pairs drawn from its two n -bit input busses:

Chip name:	Or16
Inputs:	a[16],b[16]
Outputs:	out[16]
Function:	For i=0..15 out[i]=Or(a[i],b[i]).

Multi-bit multiplexor: An n -bit multiplexor is exactly the same as the binary multiplexor described in Diagram 6, only the two inputs are each n -bit wide; the selector is a single bit.

Chip name:	Mux16
Inputs:	a[16],b[16],sel
Outputs:	out
Function:	if sel=0 then for i=0..15 out[i]=a[i] else for i=0..15 out[i]=b[i].

Multi-Way Versions of Basic Gates

Many two-way logic gates that accept two inputs have natural generalization to multi-way variants that accept an arbitrary number of inputs. This section describes a set of multi-way gates that will be used subsequently in various chips in our computer architecture. Similar generalizations can be developed for other architectures, as needed.

Multi-way Or: An n -way Or gate outputs 1 when at least one of its bit inputs is 1, and 0 otherwise.

Chip name:	Or8Way
Inputs:	in[8]
Outputs:	out
Function:	out=Or(in[0],in[1],...,in[7]).

Multi-way / Multi-Bit Multiplexor: An m -way n -bit multiplexor is a device that selects one of m n -bit input busses and outputs it to a single n -bit output bus. The selection is specified by a set of k control (input) bits, where $k = \log_2 m$. Diagram 8 depicts a typical example.

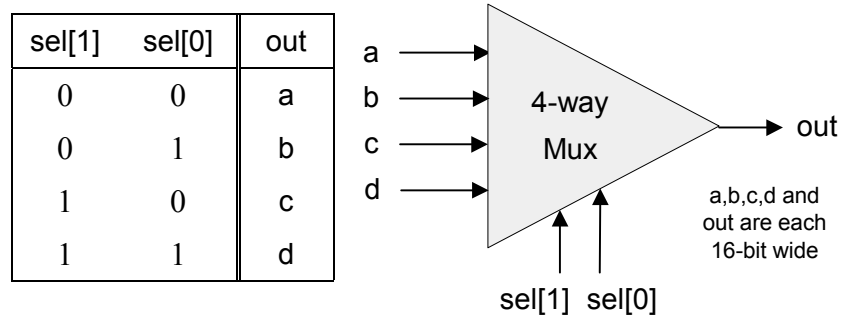


DIAGRAM 8: 4-way multiplexor. The width of the input and output busses may vary.

The computer platform that we develop in this book requires two variations of this chip: a 4-way 16-bit multiplexor, and an 8-way 16-bit multiplexor:

```

Chip name: Mux4Way16
Inputs:    a[16],b[16],c[16],d[16],sel[2]
Outputs:  out[16]
Function:  if sel=00 then out=a else if sel=01 then out=b else
               if sel=10 then out=c else if sel=11 then out=d
Comment:  The assignment operations mentioned above are all 16-bit.
               For example, "out=a" means "for i=0..15 out[i]=a[i]".
  
```

```

Chip name: Mux8Way16
Inputs:    a[16],b[16],c[16],d[16],e[16],f[16],g[16],h[16],sel[3]
Outputs:  out[16]
Function:  if sel=000 then out=a else if sel=001 then out=b else
               if sel=010 out=c ... else if sel=111 then out=h
Comment:  The assignment operations mentioned above are all 16-bit.
               For example, "out=a" means "for i=0..15 out[i]=a[i]".
  
```

Multi-way / Multi-Bit Demultiplexor: An m -way n -bit demultiplexor is a device that channels a single n -bit input into one of m possible n -bit outputs. The selection is specified by a set of k control (input) bits, where $k = \log_2 m$.

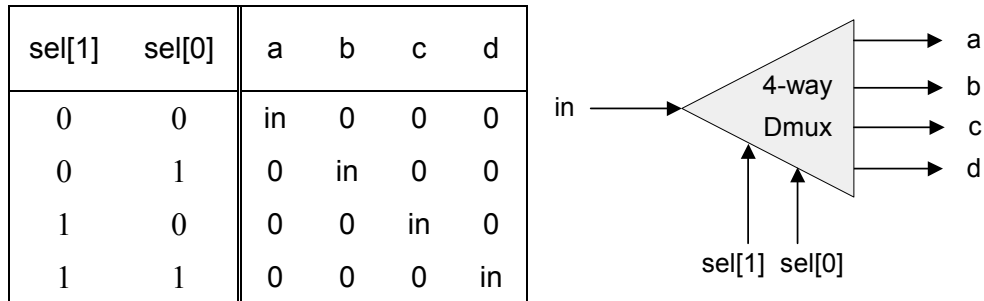


DIAGRAM 9: 4-way demultiplexor.

The computer platform that we will build requires two variations of this chip: a 4-way 1-bit demultiplexor, and an 8-way 1-bit multiplexor:

```

Chip name: DMux4Way
Inputs:    in, sel[2]
Outputs:  a, b, c, d
Function:  if sel=00 then      {a=in, b=c=d=0}
                else if sel=01 then {b=in, a=c=d=0}
                else if sel=10 then {c=in, a=b=d=0}
                else if sel=11 then {d=in, a=b=c=0}.

```

```

Chip name: DMux8Way
Inputs:    in, sel[3]
Outputs:  a, b, c, d, e, f, g, h
Function:  if sel=000 then      {a=in, b=c=d=e=f=g=h=0}
                else if sel=001 then {b=in, a=c=d=e=f=g=h=0}
                else if sel=010 ...
                ...
                else if sel=111 then {h=in, a=b=c=d=e=f=g=0}.

```

3. Implementation

Similar to the role of axioms in mathematics, *primitive* gates provide the elementary building blocks from which everything else can be built. Operationally, primitive gates have an "off-the-shelf" implementation which is supplied externally. Thus, they can be used in the construction of other gates and chips without worrying about their internal design. In the computer architecture that we are now beginning to build, we have chosen to base all the hardware on one primitive gate only: Nand. We now turn to outline the first stage of this bottom-up hardware construction project, one gate at a time. Our implementation guidelines are intentionally partial, since we want you to discover the actual gate architectures yourself. Importantly, note that each gate can be implemented in more than one way; the simpler the implementation, the better.

Not: The implementation of a unary Not gate from a binary Nand gate is rather simple. Tip: think positive.

And: Once again, the gate implementation is rather simple. Tip: think double-negative.

Or/Xor: Using some simple Boolean manipulations, these functions can be defined in terms of some of the Boolean functions implemented above. Thus, the respective gates can be built using previously-built gates.

Multiplexor / Demultiplexor: Likewise, can be built using previously-built gates.

Multi-bit Not/And/Or Gates: Since we already know how to implement the elementary versions of these gates, the implementation of their n -ary versions is simply a matter of constructing arrays of n elementary gates, having each gate operate separately on its bit inputs. This implementation task is rather boring, but it will carry its weight when these multi-bit gates will be used in the overall computer architecture, as described in subsequent chapters.

Multi-bit multiplexor: The implementation of an n -ary multiplexor is simply a matter of feeding the same selection bit to every one of n binary multiplexors. Again, a boring but useful task.

Multi-way Gates: Implementation tip: think forks.

4. Perspective

This chapter described the first steps taken in an applied digital design project. In the next chapter we will build more complicated functionality using the gates built here. Although we have chosen to use Nand as our basic building block, other approaches are possible. For example, one can build a complete computer platform using Nor gates alone, or, alternatively, a combination of And, Or, and Not gates. These constructive approaches to logic design are theoretically equivalent, just like all of geometry (and any other mathematical field) can be founded on different sets of axioms as alternative points of departure. Detailed treatments of *digital design* (also called *logic design*) techniques can be found in standard undergraduate textbooks like [Hennessy & Patterson, Appendix B] and [Mano, Chapters 2 and 3].

Throughout the chapter, we paid no attention to efficiency considerations, e.g. the number of elementary gates used in constructing a composite gate, or the number of wire cross-overs implied by the design. Such considerations are critically important in practice, and a great deal of computer science and electrical engineering expertise focuses on them. Another issue we did not address at all is the physical implementation of gates and chips using the laws of physics, e.g. the role of transistors embedded in silicon. There are of course several such implementations, each having its own characteristic (speed, power requirements, production cost, etc.) Brief discussions of these issues appear in the textbooks mentioned above. More comprehensive treatments of the technological implementations of basic gates usually require some background in electronics and physics and can be found in advanced textbooks like [Rabaey et al].

5. Build It

Objective: Implement the 15 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: The main tool that you will use in this project is the hardware simulator supplied with the book. All the gates should be implemented in the HDL language specified in appendix A.

In order to streamline and manage this construction project, we supply 45 files, as follows. For each one of the 15 gates mentioned in the chapter, we provide a skeletal .hdl program with a missing implementation part. In addition, for each gate we provide a .tst script file that tells the hardware simulator how to test it, along with the correct output file that this script should generate, called .cmp or "compare file". All these files are packed in one file named project1.zip. Your job is to complete the missing implementation parts of all the .hdl programs.

Contract: When loaded into the hardware simulator, your chip design (modified .hdl program), tested on the supplied .tst file, should deliver the behavior specified in the supplied .cmp file. If that is not the case, the simulator will let you know.

Tip: As a rule, one should implement the gates in the order in which they are presented in the book. However, since the simulator features built-in versions of all the chips described in the book, lack of .hdl versions of one gate or another should not delay the construction of more advanced chips that rely on them.

For example, consider the skeletal `Mux.hdl` program provided with this project. Suppose that for one reason or another you did not complete the implementation of this program, but you still want to use the `Mux` chip as an internal part in other chip designs. This is not a problem, thanks to the following convention. If our simulator fails to find a `Mux.hdl` file in the current directory, it automatically invokes the built-in chip implementation of `Mux`, which is pre-supplied with the simulator. This built-in `Mux` implementation -- a Java class stored in the simulator's `BuiltIn` directory -- has the same interface and functionality as those of the `Mux` chip described in the book. Thus, if you want the simulator to ignore one or more of your chip implementations, simply move the corresponding .hdl files out from the current directory.

Steps: We recommend proceeding in the following order:

0. Assuming that you've installed the book's Software Suite:
1. Read Chapter 1 and Appendix A;
2. Go through the Hardware Simulator Tutorial;
3. Create a directory called `project1` on your computer;
4. Download the `project1.zip` file and extract it to your `project1` directory;
5. Build and simulate all the chips.

2. Boolean Arithmetic¹

Counting is the religion of this generation, its hope and salvation.

(Gertrude Stein, American writer, 1874-1946)

In this chapter we build the Boolean circuits that represent numbers and perform arithmetic operations on them. Our starting point is the set of logic gates we built in the last chapter, and our ending point is a fully functional *Arithmetic Logical Unit*. The ALU is the centerpiece chip that executes all the arithmetic and logical operations performed by the computer.

1. Background

Binary Numbers: Unlike the decimal system, which is founded on base 10, the binary system is founded on base 2. When we are given a certain binary pattern, say 10011, and we are told that this pattern is supposed to represent an integer number, the decimal value of this number is computed by convention as follows:

$$(10011)_{two} = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 19 \quad (1)$$

In general, let $x = x_n x_{n-1} \dots x_0$ be a string of digits. The *value* of x in base b , denoted $(x)_b$, is defined as follows:

$$(x_n x_{n-1} \dots x_0)_b = \sum_{i=0}^n x_i \cdot b^i \quad (2)$$

The reader can verify that in the case of $(10011)_{two}$, rule (2) reduces to calculation (1).

The result of calculation (1) happens to be 19. Thus, when we press the keyboard keys labeled “1”, “9” and ENTER while running, say, a spreadsheet program, what ends up in some register in the computer is the binary code 10011. More precisely, if the computer happens to be a 32-bit machine, say, what gets stored in the register is the bit pattern 0000000000000000000000000000000010011.

Binary addition: A pair of binary numbers can be added digit-by-digit from right to left, according to the same elementary school method used in decimal addition. First, we add the two right-most digits, also called the *least significant bits* of the two binary numbers. Next, we add the resulting carry bit (which is either 0 or 1) to the sum of the next pair of bits up the significance ladder. We continue the process until the two *most significant bits* are added. If the last bit-wise addition generates a carry of 1, we can report overflow; otherwise, the addition completes successfully.

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

Positive Numbers		Negative Numbers	
0	0000		
1	0001	1111	-1
2	0010	1110	-2
3	0011	1101	-3
4	0100	1100	-4
5	0101	1011	-5
6	0110	1010	-6
7	0111	1001	-7
		1000	-8

TABLE 2: the 2's complement representation of integer numbers, assuming a 4-bit binary system.

An inspection of Table 2 suggests that an n -bit binary system with 2's complement representation has the following properties:

- The system can code a total of 2^n signed numbers, of which the maximal and minimal numbers are $2^{n-1} - 1$ and -2^{n-1} , respectively;
- The codes of all positive numbers begin with a "0";
- The codes of all negative numbers begin with a "1";
- To obtain the code of $-x$ from the code of x , leave all the trailing (least significant) 0's and the first least significant 1 intact, then flip all the remaining bits (convert 0's to 1's and vice versa). An equivalent shortcut, which is easier to implement in hardware, is to flip all the bits of x and add 1 to the result.

A particularly attractive feature of this representation is that addition of any two numbers in 2's complement is exactly identical to addition of positive numbers. Consider, for example, the addition operation $(-2) + (-3)$: using 2's complement (in a 4-bit representation) we have to add, in binary, $(1110)_{two} + (1101)_{two}$. Without paying any attention to which numbers (positive or negative) these codes represent, bit-wise addition will yield 1011 (after throwing away the 5'th overflow bit). Indeed, this is the 2's complement representation of (-5) .

In short, we see that we are able to perform addition of any two signed numbers without requiring any special hardware beyond that needed for simple bit-wise addition. What about subtraction? Recall that in the 2's complement method, the arithmetic negation of a signed number x , i.e. computing $-x$, is achieved by negating all the bits of x and adding 1 to the result. Thus subtraction can be handled by $x - y = x + (-y)$. Once again, hardware complexity is kept to a minimum.

The material implications of these theoretical results are significant. Basically, they imply that a single chip, called *Arithmetic Logical Unit*, can be used to encapsulate all the basic arithmetic and logical operators performed in hardware. We now turn to specify one such ALU, beginning with the specification of an adder chip.

2. Specification

Adders

We present a hierarchy of three adders, leading to a multi-bit adder chip:

- *Half-adder*: designed to add 2 bits;
- *Full-adder*: designed to add 3 bits;
- *Adder*: designed to add two n -bit numbers.

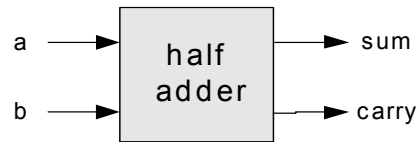
We also present a special-purpose adder, called *incrementer*, designed to add 1 to a given number.

Half Adder: The first step on our way to adding binary numbers is to be able to add two bits. This task requires the handling of four possible cases:

$$\begin{aligned} 0 + 0 &= 00 \\ 0 + 1 &= 01 \\ 1 + 0 &= 01 \\ 1 + 1 &= 10 \end{aligned}$$

We will now present a chip, called *half-adder*, that implements this addition operation. The least significant bit of the addition is called sum, and the most significant bit is called carry.

Inputs		Outputs	
a	b	carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

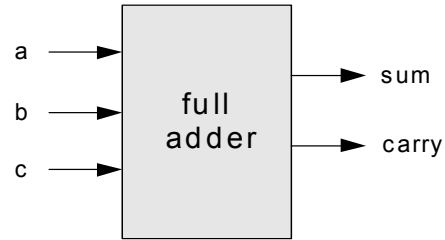


Chip name:	HalfAdder
Inputs:	a, b
Outputs:	sum, carry
Function:	sum = LSB of a+b carry = MSB of a+b

DIAGRAM 3: Half Adder, designed to add 2 bits.

Full Adder: Now that we know how to add 2 bits, we present a *full-adder* chip, designed to add 3 bits. Like the half-adder case, the full-adder chip produces two outputs: the least significant bit of the addition, and the carry bit.

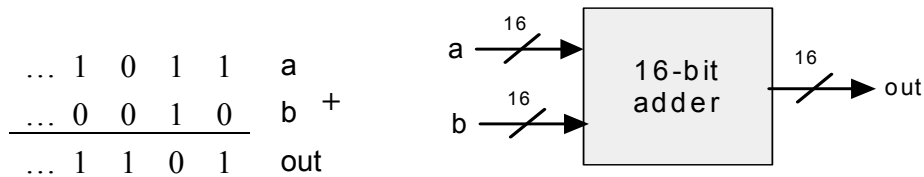
a	b	c	carry	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Chip name:	FullAdder
Inputs:	a, b, c
Outputs:	sum, carry
Function:	sum = LSB of a+b+c carry = MSB of a+b+c

DIAGRAM 4: Full Adder, designed to add 3 bits.

Adder: Memory and register chips represent integer numbers by *n*-bit patterns, *n* being 16, 32, 64, etc. – depending on the computer platform. The chip whose job is to add such numbers is called a multi-bit adder, or simply *adder*. We present a 16-bit adder, noting that our diagrams and specifications scale up as-is to any *n*-bit system.



Chip name:	Add16
Inputs:	a[16], b[16]
Outputs:	out[16]
Function:	out=a+b
Comment:	integer 2's complement addition. overflow is neither detected nor handled.

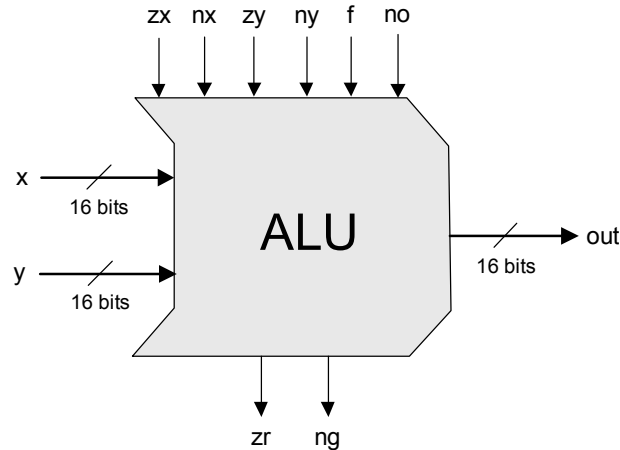
DIAGRAM 5: 16-bit adder. The example (top left) illustrates the addition of two 4-bit numbers. *n*-bit addition for any *n* is “more of the same.”

Incrementer: It is convenient to have a special purpose chip dedicated to adding the constant 1 to a given number. Here is the API of a 16-bit incrementer:

Chip name:	Incl6
Inputs:	in[16]
Outputs:	out[16]
Function:	out=in+1
Comment:	integer 2's complement addition. overflow is neither detected nor handled.

The Arithmetic Logic Unit (ALU)

The specifications of the adder chips presented so far were generic, meaning that they hold for every computer. We now turn to discuss the specific 16-bit ALU of the Hack platform. This chip is designed to compute a fixed set of functions $out = f_i(x,y)$ where x and y are the chip's two 16-bit inputs, out is the chip's 16-bit output, and f_i is an arithmetic or logical function selected from a fixed repertoire of possible functions. We instruct the Hack ALU which function to compute by setting a set of six input bits, called *control bits*, to certain binary values. The exact specification of which function the ALU computes given each setting of the control bits is given in Diagram 6, using pseudo-code.



```

Chip name: ALU
Inputs:    x[16],y[16],    // data inputs
              zx,           // zero the x input
              nx,           // negate the x input
              zy,           // zero the y input
              ny,           // negate the y input
              f,            // function code: 1 for Add, 0 for And
              no            // negate the out output
Outputs:  out[16],      // data output
              zr,          // status flag, true when the ALU output=0
              ng           // status flag, true when the ALU output<0
Function:  if zx then x=0          // 16-bit zero constant
              if nx then x=~x        // bit-wise negation
              if zy then y=0          // 16-bit zero constant
              if ny then y=~y        // bit-wise negation
              if f then out=x+y       // integer 2's complement addition
                  else out=x&y       // bit-wise And
              if no then out=~out     // bit-wise negation
              if out=0 then zr=1 else zr=0 // 16-bit equality comparison
              if out<0 then ng=1 else ng=0 // 2's-complement comparison
Comment:  overflow is neither detected nor handled.

```

DIAGRAM 6: The ALU of the Hack platform: interface and API. The ALU operation (the function computed on x and y) is determined by the six control bits. The ALU sets the output bits zr and ng to 1 when the output out is zero or negative, respectively.

Note that each one of the six control bits instructs the ALU to carry out a certain operation. Taken together, the combined effects of these operations cause the ALU to compute a variety of useful functions. Since the ALU is controlled by six control bits, it can potentially compute $2^6 = 64$ different functions. 18 of these functions are documented in Table 7.

these bits instruct how to pre-set the x input		these bits instruct how to pre-set the y input		this bit selects betw. + / And	this bit inst. how to post-set out	resulting ALU output
zx	nx	zy	ny	f	no	out=
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 And y	If no then out=-out	f(x,y)=
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

TABLE 7: The ALU truth table. Taken together, the binary operations coded by the first six columns (input control bits) in each row affect the overall function listed in the right column of that row. (We use the symbols \sim , $\&$, and $|$ to represent the operators Not, And, and Or, respectively, performed bit-wise.). The complete ALU truth table consists of 64 rows, of which only the 18 presented here are of interest.

We see that programming the ALU to compute a certain function $f(x,y)$ is done by setting the six control bits to the code of the desired function. From this point on, the internal ALU logic specified in Diagram 6 should cause the ALU to output the value $f(x,y)$ specified in Table 7. This does not happen miraculously -- it's the result of careful design.

For example, let us consider the 12th row of table 7, where the ALU is instructed to compute the function $x-1$. The zx and nx bits are 0, so the x input is neither zeroed nor negated. The zy and ny bits are 1, so the y input is first zeroed, and then negated bit-wise. Bit-wise negation of zero, $(000\dots00)_{\text{two}}$, gives $(111\dots11)_{\text{two}}$, which is the 2's complement code of -1. Thus the ALU inputs end up being x and -1. Since the f -bit is 1, the selected operation is *arithmetic addition*, causing

the ALU to calculate $x+(-1)$. Finally, since the `no` bit is 0, the output is not negated but rather left as is. To conclude, the ALU ends up computing $x-1$, which was our goal.

Does the ALU logic described in Table 6 compute every one of the other 17 functions listed in the right column of Table 7? To verify that this is indeed the case, the reader is advised to pick up some other rows in the table and prove their respective ALU operation. We note in passing that some of these computations, beginning with the function $f(x,y)=1$, are not trivial. We also note that there are some other useful functions computed by the ALU but not listed in the table.

It may be instructive to describe the thought process that led to the design of this particular ALU. First, we made a list of all the primitive operations that we wanted our computer to be able to execute (right column in Table 7). 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 certain binary operation. The resulting ALU is simple and elegant.

3. Implementation

Our implementation guidelines are intentionally partial, since we want you to discover the actual chip architectures yourself. As usual, each gate can be implemented in more than one way; the simpler the implementation, the better.

Half Adder: An inspection of Diagram 3 reveals that the functions $\text{sum}(a,b)$ and $\text{carry}(a,b)$ happen to be identical to the standard $\text{Xor}(a,b)$ and $\text{And}(a,b)$ functions. Thus, the implementation of this adder is rather trivial, using previously built gates.

Full Adder: A Full-Adder chip can be implemented from two half-adder chips and one additional simple gate. Other direct implementation options are also possible, without using half-adder chips.

Adder: The addition of two signed numbers represented by the 2's complement method as two n -bit busses can be done bit-wise, from right to left, in n steps. In step 0, the least significant pair of bits is added, and the carry bit is fed into the addition of the next significant pair of bits. The process continues until in step $n-1$ the most significant pair of bits is added. Note that each step involves the addition of 3 bits. Hence, an n -bit adder can be implemented by creating an array of n full-adder chips, and chaining them in such a way that the carry bit of each adder is fed into one of the inputs of the next adder up the significance ladder.

Incrementer: An n -bit incrementer can be implemented trivially from an n -bit adder.

ALU: Note that the ALU was carefully planned to effect all the desired ALU operations *logically*, using simple Boolean operations. Therefore, the *physical* implementation of the ALU is reduced to implementing these simple Boolean operations, following their pseudo-code specifications. Your first step will likely be to create a logic circuit that manipulates a 16-bit input according to `nx` and `zx` control bits (i.e. the circuit should conditionally zero and negate the 16-bit input). This logic can be used to manipulate the `x` and `y` inputs, as well as the `out` output.

Chips for addition and for bit-wise **And**-ing have already been built. Thus, what remains is to build logic that chooses between them according to the **f** control bit. Finally, you will need some logic that integrates all the other chips into the overall ALU.

4. Perspective

The construction of the multi-bit adder presented in this chapter was standard, although no attention was paid to efficiency. In fact, our suggested adder implementation is rather inefficient, due to the long delays incurred while the carry propagates from the least significant bit to the most significant bit. This problem can be alleviated using logic circuits that effect so-called "carry look-ahead" techniques. Since addition is one of the most prevalent operations in any given computer architecture, such low-level improvements can result in dramatic and global performance gains throughout the computer.

In any given computer, the overall functionality of the hardware/software platform is delivered jointly by the ALU and the operating system that runs on top of it. Thus, when designing a new computer system, the question of how much functionality the ALU should deliver is essentially a cost/performance issue. The general rule is that hardware implementations of arithmetic and logical operations are usually more costly, but achieve better performance. The design tradeoff that we have chosen in this book is to specify an ALU hardware with a limited functionality and then implement as many operations as possible in software. For example, our ALU features neither multiplication and division operations, nor floating point arithmetic. Some of these operations (as well as more mathematical functions) will be implemented at the operating system level, as described in Chapter 11.

Detailed treatments of Boolean arithmetic and ALU design can be found in standard undergraduate textbooks such as [Hennessy & Patterson, chapter 4].

5. Build It

Objective: Implement all the chips presented in this chapter, using previously built chips.

Tip: When your HDL programs invoke chips that you may have built in the previous project, it is recommended to use instead the built-in versions of these chips. This will ensure correctness and speed up the operation of the hardware simulator. There is a simple way to accomplish this convention: make sure that your project directory includes only the files that belong to the present project.

The remaining instructions for this project are identical to those from Chapter 1, except that every occurrence of the text "project1" should be replaced with "project2".

3. Sequential Logic ¹

“It’s a poor sort of memory that only works backward.”

Lewis Carroll (1832-1898)

All the Boolean and arithmetic chips that we built in previous chapters were *combinational*. Combinational chips compute functions that depend solely on *combinations* of their input values. These relatively simple chips provide many important processing functions (like the ALU), but they cannot *maintain state*. Since computers must be able to not only compute values but also to store and recall values, they must be equipped with memory elements that can preserve data over time. These memory elements are built from *sequential chips*.

The implementation of memory elements is an intricate art involving synchronization, clocking, and feedback loops. Conveniently, most of this complexity can be embedded in the operating logic of very low-level sequential gates called *flip-flops*. Using these flip-flops as elementary building blocks, we will specify and build all the memory devices employed by typical modern computers, from binary cells to registers to memory banks and counters. This effort will complete the construction of the chip-set needed to build an entire computer – a challenge that we take up in the next chapter.

Following a brief overview of clocks and flip-flops, section 1 introduces all the memory chips that we will build on top of them. Sections 2 and 3 describe the chips specifications and implementation, respectively. As usual, all the chips mentioned in the chapter can be built and tested using the hardware simulator supplied with the book.

1. Background

The act of “remembering something” is inherently a function of time: you remember *now* what has been committed to memory *before*. Thus, in order to employ chips that “remember” information, we must first develop some means for representing the progression of time.

The Clock: In most computers, the passage of time is marked by a master clock that delivers a continuous train of alternating signals. The exact hardware implementation is usually based on an oscillator that alternates continuously between two phases labeled “0-1”, “*low-high*”, “*tick-tock*”, etc. The elapsed time between the beginning of a “tick” and the end of the subsequent “tock” is called *cycle*, and each clock cycle is treated as a discrete time unit. The current clock phase (*tick* or *tock*) is represented by a binary signal. Using the hardware’s circuitry, this signal is simultaneously broadcast to every sequential chip throughout the computer platform.

Flip-flops: The most elementary sequential element in the computer is a device called *flip-flop*, of which there are several variants. In this book we use a variant called *data flip-flop*, or DFF, whose interface consists of a single-bit data input and a single-bit data output. In addition, the DFF has a *clock* input that continuously changes according to the master clock’s signal. Taken

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

together, the data and the clock inputs enable the DFF to implement the time-based behavior $out(t) = in(t-1)$, where in and out are the gate's input and output values and t is the current clock cycle. In other words, the DFF simply outputs the input value from the previous time unit.

As we now turn to show, this elementary behavior can form the basis of all the hardware devices in the computer that have to *maintain state*, from binary cells to registers to arbitrarily large random access memory units.

Registers: A *register* is a storage device that can “store,” or “remember,” a value over time, implementing the classical storage behavior $out(t) = out(t-1)$. A DFF, on the other hand, can only output its previous input, i.e. $out(t) = in(t-1)$. This suggests that a register can be implemented from a DFF by simply feeding the DFF output back into its input, creating the device shown in the middle of Fig. 1. Presumably, the output of this device at any time t will equal its output at time $t-1$, yielding the classical function expected from a storage unit.

Well, not so. The device shown in the middle of Fig. 1 is invalid. First, it is not clear how we'll be able to ever load this device with a new data value, since there are no means to tell the DFF when to draw its input from the in wire and when from the out wire. More generally, the rules of chip design dictate that internal pins must have a fan-in of 1, meaning that they can be fed from a single source only.

The good thing about this thought experiment is that it leads us to the correct and elegant solution shown in the right of Fig. 1. In particular, a natural way to resolve our input ambiguity is to introduce a multiplexor into the design. Further, the “select bit” of this multiplexor can become the “load bit” of the overall register chip: if we want the register to start storing a new value, we can put this value in the in input and set the $load$ bit to 1; if we want the register to keep storing its internal value until further notice, we can set the $load$ bit to 0.

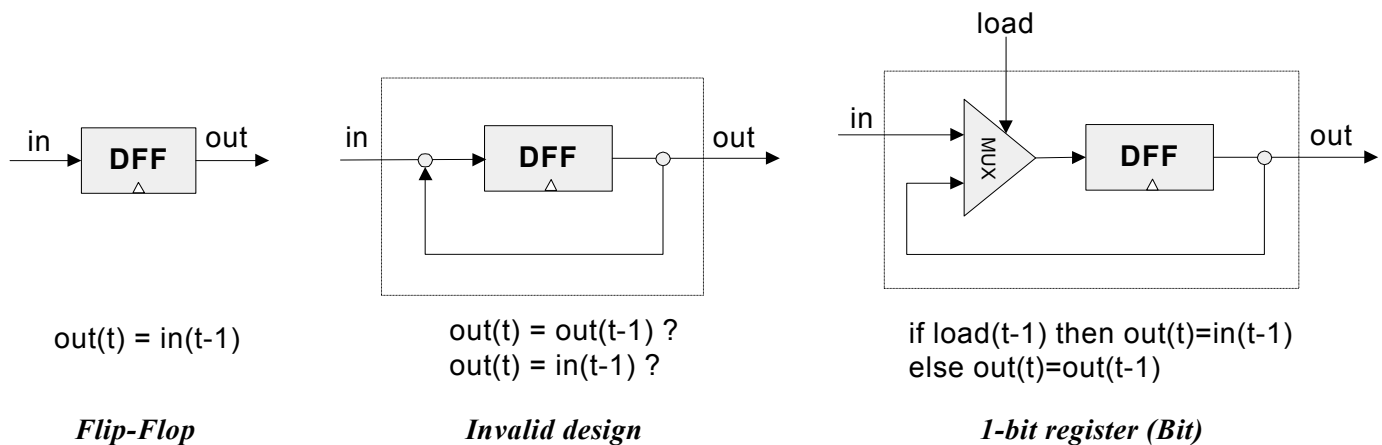


FIGURE 1: From DFF to single-bit register. The small triangle represents the clock input of the DFF. In chip diagrams, this icon states that the marked chip, as well as the overall chip that encapsulates it, are time-dependent.

Once we have the basic ability to remember a single bit over time, we can easily construct arbitrarily wide registers. This can be achieved by forming an array of as many single-bit registers as needed, creating a register that holds multi-bit values (Fig. 2). The basic design parameter of such a register is its *width* – the number of bits it holds; in modern computers, registers are usually 32-bit or 64-bit wide. The contents of such registers are typically referred to as *words*.

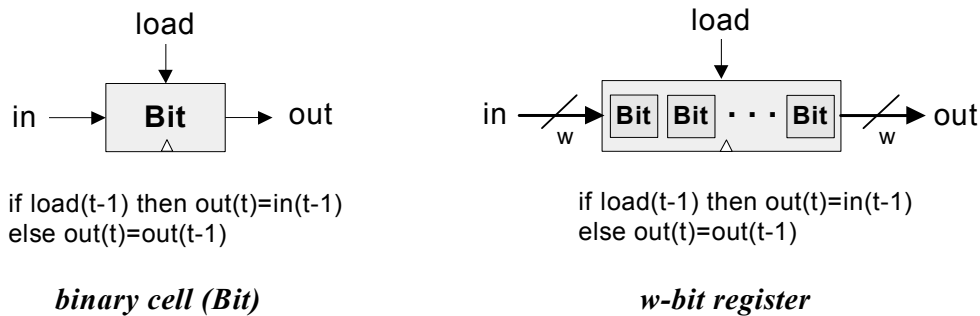


FIGURE 2: From single-bit to multi-bit registers. A multi-bit register of width w can be constructed from an array of w `Bit` chips. The operating functions of both chips is exactly the same, except that the "=" assignments are single-bit and multi-bit, respectively.

Memories: Once we have the basic ability to represent words, we can proceed to build memory banks of arbitrary length. As Fig. 3 shows, this is done by stacking together many registers to construct a *Random Access Memory* (RAM) unit. The term *random access memory* derives from the requirement that read/write operations on a RAM should be able to access randomly chosen words, with no restrictions on the order in which they are accessed. That is to say, we require that *any* word in the memory -- irrespective of its physical location -- will be accessed instantaneously, in equal speed.

This requirement can be satisfied as follows. First, we assign each word in the n -registers RAM a unique *address* (an integer between 0 to $n-1$), according to which it will be accessed. Second, in addition to stacking the n registers together, we augment the RAM chip design with a set of logic gates that, given an address j , is capable of *selecting* the individual register whose address is j .

In sum, a classical RAM device accepts three inputs: a data input, an address input, and a load bit. The *address* specifies which RAM register should be accessed in the current time unit. 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 will commit the input value in the next time unit, at which point the RAM's output will start emitting it.

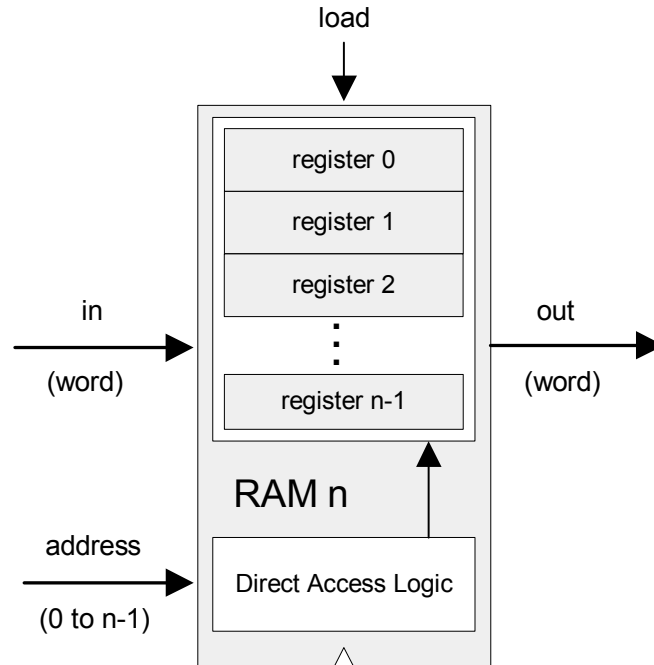


FIGURE 3: RAM chip (conceptual). The width and length of the RAM can vary.

The basic design parameters of a RAM device are its data *width* -- the width of each one of its words, and its *size* -- the number of words in the RAM. Modern computers typically employ 32- or 64-bit wide RAMs whose size is up to hundreds of millions.

Counters: A counter is a sequential chip whose state is an integer number that increments every time unit, effecting the function $out(t) = out(t-1) + c$, where c is typically 1. Counters play an important role in digital architectures. For example, most CPU's include a *program counter* that keeps track of the address of the instruction that should be executed next in the current program.

A counter chip can be implemented by combining the input/output logic of a standard register with the combinatorial logic for adding the constant 1. Typically, the counter will have to be equipped with some additional functionality, such as possibilities for resetting the count to zero, loading a new counting base, or decrementing instead of incrementing.

Time Matters

All the chips that were described above are *sequential*. Simply stated, a sequential chip is a chip that includes one or more DFF gates, either directly or indirectly. Functionally speaking, the DFF gates endow sequential chips with the ability to either maintain state (as in memory units), or to operate on state (as in counters). Technically speaking, this is done by forming feedback loops inside the sequential chip (see Fig. 4). In combinational chips, where time is neither modeled nor recognized, the introduction of feedback loops will be problematic: the output would depend on the input, which itself would depend on the output, and thus the output would depend on itself. On the other hand, there is no difficulty in feeding the output of a sequential chip back into itself, since the DFFs introduce an inherent time delay: the output at time t does not depend on itself, but

rather on the output at time $t-1$. This property guards against the uncontrolled “data races” that would occur in combinational chips with feedback loops.

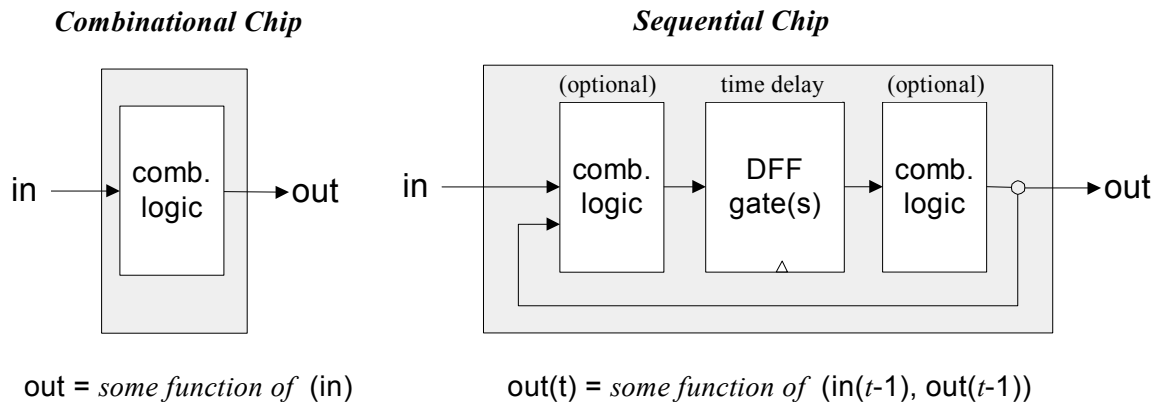


FIGURE 4: Combinational versus sequential logic (*in* and *out* stand for potentially several input and output variables). Sequential chips always consist of one layer of DFFs and optional combinational logic layers.

Recall that the outputs of combinational chips change when their inputs change, irrespective of time. In contrast, the special architecture of sequential chips implies that their outputs change only at the point of transition from one clock cycle to the next, and not within the clock cycle itself. In fact, we allow sequential chips to be in unstable states *during* clock cycles, requiring only that by the beginning of the next cycle they will output correct values.

This “discretization” of the sequential chips outputs has an important side effect: it is used to synchronize the overall computer architecture. To illustrate, suppose we instruct the arithmetic logic unit (ALU) to compute $x + y$ where x is the value of a nearby register and y is the value of a remote RAM register. Because of various physical constraints (distance, resistance, interference, random noise, etc.) the electrons representing x and y will arrive to the ALU at different times. However, being a *combinational chip*, the ALU is insensitive to the concept of time -- it continuously adds up whichever data values happen to lodge in its inputs. Thus, it will take some time before the ALU’s output will stabilize to the correct $x + y$ result. Until then, the ALU will generate garbage.

How can we overcome this difficulty? Well, since the output of the ALU is always routed to some sort of a sequential chip (a register, a RAM location, etc.), *we don’t really care*. All we have to do is ensure that the length of the clock cycle will be slightly longer than the time it takes an electron to travel the longest distance from one chip in the architecture to another. This way, we are guaranteed that by the time the sequential chip will update its state (at the beginning of the next clock cycle), the inputs that it will receive from the ALU will be correct. This, in a nutshell, is the trick that synchronizes a set of stand-alone hardware components into a well-coordinated system, as we will see in Chapter 5.

2. Specification

This section specifies a hierarchy of sequential chips:

- D-Flip-flops (DFF)
- Registers (based on DFF's)
- Memory banks (based on registers)
- Counter chips (also based on registers)

D-Flip-Flop

The most elementary storage device that we present – the basic component from which all memory elements will be designed – is the *Data Flip-Flop* gate. A DFF gate has a single-bit input and a single-bit output, as follows:



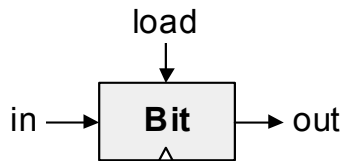
Chip name:	DFF
Inputs:	in
Outputs:	out
Function:	$out(t) = in(t-1)$
Comment:	This clocked gate has a built-in implementation and thus there is no need to implement it.

Like Nand gates, DFF gates enter our computer architecture at a very low level. Specifically, all the sequential chips in the computer (registers, memory, and counters) are based on numerous DFF gates. All these DFFs are connected to the same master clock, forming a huge distributed “chorus line”. At the beginning of each clock cycle, the outputs of *all* the DFFs in the computer commit to their inputs during the previous time-unit. At all other times, the DFFs are “latched,” meaning that changes in their inputs have no immediate effect on their outputs. This remarkable conduction feat is done in parallel, many times each second (depending on the clock frequency).

In hardware implementations, the time-dependency of the DFF gates is achieved by simultaneously feeding the master clock signal to all the DFF gates in the platform. Hardware simulators emulate the same effect in software. As far the computer architect is concerned, the end result is the same: the inclusion of a DFF gate in the design of any chip ensures that the overall chip, as well as all the chips that depend on it up the hardware hierarchy, will be “automatically” time-dependent. These chips are called *sequential*, by definition.

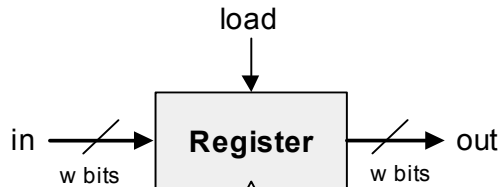
Registers

A single-bit register, which we call `Bit`, or *binary cell*, is designed to store a single bit of information (0 or 1). The chip interface consists of an input pin which carries a data bit, a `load` bit which enables the cell for writes, and an output pin which emits the current state of the cell. The interface diagram and API of a binary cell are as follows:



Chip name:	<code>Bit</code>
Inputs:	<code>in, load</code>
Outputs:	<code>out</code>
Function:	If <code>load(t-1)</code> then <code>out(t)=in(t-1)</code> else <code>out(t)=out(t-1)</code>

The API of the `Register` chip is essentially the same as that of a binary cell, except that the input and output pins are designed to handle multi-bit values:



Chip name:	<code>Register</code>
Inputs:	<code>in[16], load</code>
Outputs:	<code>out[16]</code>
Function:	If <code>load(t-1)</code> then <code>out(t)=in(t-1)</code> else <code>out(t)=out(t-1)</code>
Comment:	"=" is a 16-bit operation.

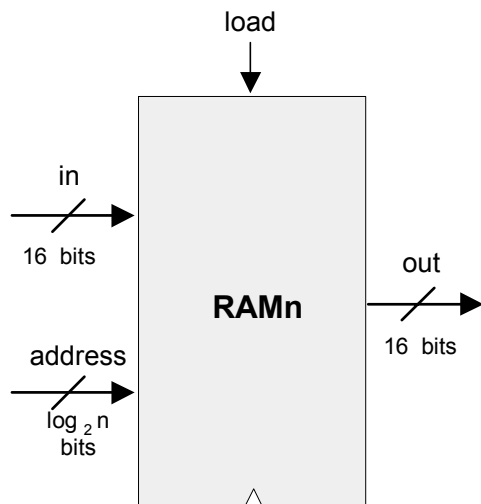
The `Bit` and `Register` chips have exactly the same read/write behavior, as follows:

Read: To read the contents of a register, we simply probe its multi-bit output.

Write: To write a new multi-bit data value d into a register, we put d in the `in` input and assert the `load` input. In the next clock cycle, the register will commit to the new data value, and its output will start emitting d .

Memory

A direct-access memory unit, also called `RAM`, is an array of n w -bit registers, equipped with direct access circuitry. The number of registers (n) and the width of each register (w) are called the memory's *size* and *width*, respectively. We will build a hierarchy of such `RAM` units, all 16-bit wide, but with varying sizes: `RAM8`, `RAM64`, `RAM512`, `RAM4K`, and `RAM16K` units. All these memory chips have precisely the same API, and thus we describe them in one parametric diagram, as follows:



Chip name: RAM n // n and k are listed below
Inputs: in[16], address[k], load
Outputs: out[16]
Function: Out(t)=RAM[address(t)](t)
 If load($t-1$) then
 RAM[address($t-1$)](t)=in($t-1$)
Comment: “=” is a 16-bit operation.

We need 5 such chips, as follows:

Chip name	n	k
RAM8	8	3
RAM64	64	6
RAM512	512	9
RAM4K	4096	12
RAM16K	16384	14

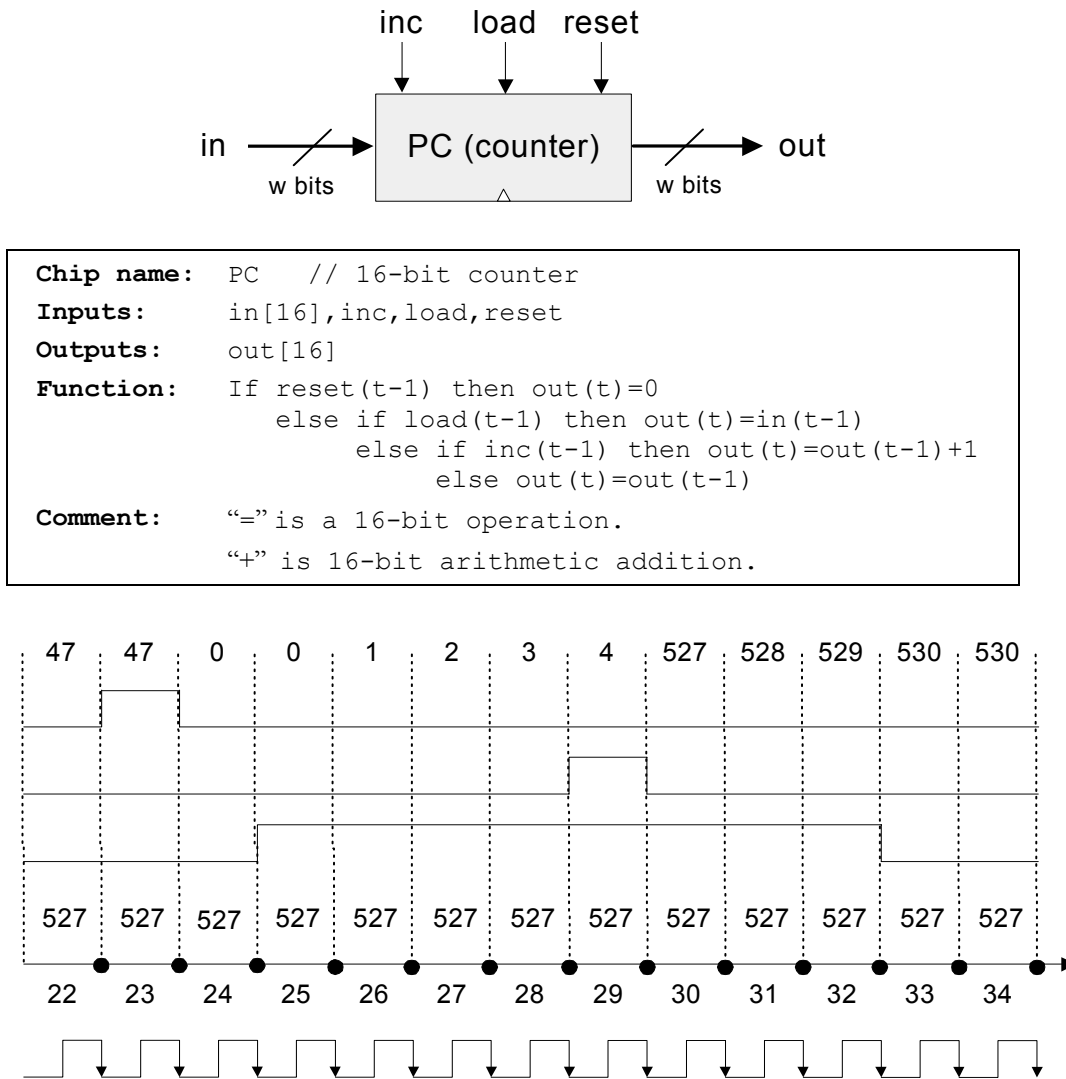
Read: To read the contents of register number m , we put m in the address input. The RAM's direct-access logic will select register number m , which will then emit its output value to the RAM's output pin. This is a combinational operation, independent of the clock.

Write: To write a new data value d into register number m , we put m in the address input, d in the in input, and assert the load input bit. The RAM's direct-access logic will select register number m , and the load bit will enable it. In the next clock cycle, the selected register will commit to the new value (d), and the RAM's output will start emitting it.

Counter

Although a *counter* is a stand-alone abstraction in its own right, it is convenient to motivate its specification by saying a few words about the context in which it is normally used. For example, consider a counter chip designed to contain the address of the instruction that the computer should fetch and execute next. In most cases, the counter has to simply increment itself by 1 in each clock cycle, thus causing the computer to fetch the next instruction in the program. In other cases, e.g. in “jump to execute instruction number n ”, we want to be able to set the counter to n , and then have it continue its default counting behavior: $n+1$, $n+2$, etc. Finally, the program's execution can be restarted anytime by simply setting the counter to 0, assuming that that's the address of the program's first instruction. In short, we need a loadable and resettable counter.

With that in mind, the interface of our Counter chip is similar to that of a register, except that it has two additional control bits, labeled *reset* and *inc*. When *inc*=1, the counter increments its state in every clock cycle, emitting the value $out(t)=out(t-1)+1$. If we want to reset the counter to 0, we assert the *reset* bit; if we want to initialize it to some other counting base d , we put d in the IN input and assert the *load* bit. The details are given in the counter API, and an example of its operation is depicted in Fig. 5.



We assume that we start tracking the counter in time unit 22, when its input and output happen to be 527 and 47, respectively. We also assume that the counter's control bits (`reset`, `load`, `inc`) are 0 -- all arbitrary assumptions.

FIGURE 5: Counter Simulation. At time 23 a `reset` signal is issued, causing the counter to emit zero in the following time-unit. The zero persists until an `inc` signal is issued at time 25, causing the counter to start incrementing, one time-unit later. The counting continues until at time 29 the `load` bit is asserted. Since the counter's input holds the number 527, the counter is reset to that value in the next time-unit. Since `inc` is still asserted, the counter continues incrementing, until time 33, when `inc` is de-asserted.

3. Implementation

Flip-Flop: DFF gates can be implemented from lower-level logic gates like those built in Chapter 1. However, in this book we view DFF gates as primitive, and thus they can be used in hardware construction projects without worrying about their internal implementation.

1-bit register (Bit): The implementation of this chip was given in Fig. 1.

Register: The construction of a w -bit Register chip from binary cells is straightforward. All we have to do is construct an array of w Bit gates and feed the register's load input to all of them.

8-Registers Memory (RAM8): An inspection of Fig. 3 may be useful here. To implement a RAM8 chip, we line up an array of 8 registers. Next, we have to build combinational logic that, given a certain address value, takes the RAM8's in input and loads it into the selected register. In a similar fashion, we have to build combinational logic that, given a certain address value, selects the right register and pipes its out value to the RAM8's out output. Tip: the combinational logic mentioned above was already implemented in Chapter 1.

n -Registers Memory: A memory bank of arbitrary length (a power of 2) can be built recursively from smaller memory units, all the way down to the single register level. This view is depicted in Fig. 6. Focusing on the right hand side of the figure, we note that a 64-register RAM can be built from an array of eight 8-register RAM chips. To select a particular register from the RAM64 memory, we use a 6-bit address, say $xxxyyy$. The MSB xxx bits select one of the RAM8 chips, and the LSB yyy bits select one of the registers within the selected RAM8. The RAM64 chip should be equipped with logic circuits that affect this hierarchical addressing scheme.

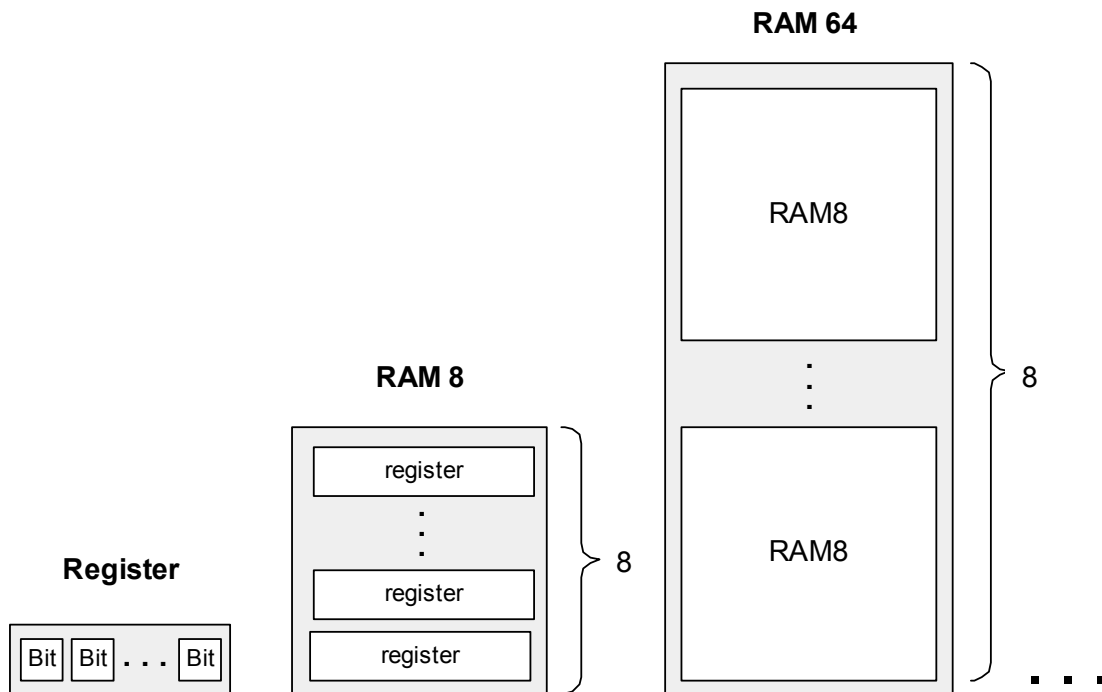


FIGURE 6: Gradual construction of memory banks by recursive ascent. A w -bit register is an array of w binary cells, an 8-register RAM an array of eight w -bit registers, a 64-register RAM an array of eight RAM8 chips, and so on. Only three more similar construction steps are necessary to build a 16K RAM chip.

Counter: A w -bit counter consists of two main elements: a regular w -bit register, and combinational logic. The combinational logic is designed to (a) compute the counting function, and (b) put the counter in the right operating mode, as mandated by the values of its three control bits. Tip: most of this logic was already built in Chapter 2.

4. Perspective

The cornerstone of all the memory systems described in this chapter is the *flip-flop* – a gate that we treated here as an atomic, primitive building block. The usual approach in hardware textbooks is to construct flip-flops from elementary combinatorial gates (e.g. Nand gates) using appropriate feedback loops. The standard construction begins by building a simple (non-clocked) flip-flop that is bi-stable, i.e. that can be set to be in one of two states. Then a clocked flip-flop is obtained by cascading two such simple flip-flops, the first being set when the clock *tics* and the second when the clock *tocks*. This “master-slave” design endows the overall flip-flop with the desired clocked synchronization functionality.

These constructions are rather elaborate, requiring an understating of delicate issues like the effect of feedback loops on combinatorial circuits, as well as the implementation of clock cycles using a two-phase binary clock signal. In this book we have chosen to abstract away these low-level considerations by treating the flip-flop as an atomic gate. Readers who wish to explore the internal structure of flip-flop gates can find detailed descriptions in [Mano, chapter 6] and [Hennessy & Patterson, appendix B].

In closing, we should mention that memory devices of modern computers are not always constructed from standard flip-flops. Instead, modern memory chips are usually very 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.

All the other chip constructions in this chapter -- the registers and memory chips that were built on top of the flip-flop gates -- were rather standard.

5. Build It

Objective: Build the chips listed below (except the first one). The only building blocks that you can use are primitive DFF gates, chips that you will build on top of them, and chips built in previous chapters.

▪ DFF	Data Flip-Flop (primitive – no need to implement)
▪ Bit	1-bit binary cell
▪ Register	16-bit
▪ RAM8	16-bit / 8-register memory
▪ RAM64	16-bit / 64-register memory
▪ RAM512	16-bit / 512-register memory
▪ RAM4K	16-bit / 4,096-register memory
▪ RAM16K	16-bit / 16,384-register memory
▪ PC	16-bit counter

Resources: The main tool that you will use in this project is the hardware simulator supplied with the book. All the chips should be implemented in the HDL language specified in appendix A.

As usual, for each chip mentioned above 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.” All these files are packed in one file named project3.zip. Your job is to complete the missing implementation parts of all the .hdl programs.

Contract: When loaded into the hardware simulator, your chip design (modified .hdl program), tested on the supplied .tst file, should deliver the behavior specified in the supplied .cmp file. If that is not the case, the simulator will let you know.

Tip: When your HDL programs invoke chips that you may have built in previous projects, it is recommended to use the built-in versions of these chips instead. This will ensure correctness and speed up the simulator’s operation. There is a simple way to accomplish this convention: make sure that your project directory includes only the files that belong to the present project.

Likewise, when constructing RAM chips from smaller ones, we recommend to use built-in versions of the latter. Otherwise, the simulator may run very slowly or even out of (real) memory space, since large RAM chips contain many tens of thousands of lower level chips, and all these chips must be simulated as software objects by the simulator. Thus, we suggest that after you complete the implementation and testing of a RAM chip, you will move its respective HDL file out from the project directory. This way, the simulator will resort to using the built-in versions of these chips.

Steps: We recommend proceeding in the following order:

0. Before starting this project, read sections 6 and 7 of Appendix A.
1. Create a directory called project3 on your computer;
2. Download the project3.zip file and extract it to your project3 directory;
3. Build and simulate all the chips.

4. Machine Language¹

“Form ever follows function”, Louis Sullivan (architect, 1856-1924)

“Form IS function”, Ludwig Mies van der Rohe (architect, 1886-1969)

A computer can be described *constructively*, by laying out its hardware platform and explaining how it is built from low-level chips. A computer can also be described *abstractly*, by specifying and demonstrating its machine language capabilities. And indeed, it is easier to get acquainted with a new computer system by first seeing some low-level programs written in its machine language. This helps us understand not only how to program the computer to do useful things, but also why its hardware was designed in a certain way. With that in mind, this chapter focuses on low-level programming in general, and on the Hack machine language in particular. This will set the stage for the next chapter, where we complete the construction of the Hack computer from the chips that we built in the previous chapters.

A machine language is an agreed-upon formalism, designed to code low-level programs as series of machine instructions. Using these instructions, the programmer can command the processor to perform arithmetic and logic operations, fetch and store values from and to the memory, move values from one register to another, test Boolean conditions, and so on. As opposed to high level languages, whose basic design goals are generality and power of expression, the goal of machine language’s design is direct execution in, and total control of, a given hardware platform. Of course, generality, power, and elegance are still desired, but only to the extent that they adhere to the basic requirement of direct execution in hardware.

Machine language is the most profound interface in the overall computer enterprise -- the fine line where hardware and software meet. This is the point where the abstract thoughts of the programmer, as manifested in symbolic instructions, are turned into physical operations performed in silicon. Thus, machine language is construed both a programming tool and an integral part of the hardware platform. In fact, just like we say that the machine language is designed to exploit a given hardware platform, we can say that the hardware platform is designed to fetch, interpret and execute, instructions written in the given machine language.

The chapter begins with a general introduction of machine language programming. Next, we give a detailed specification of the Hack machine language, covering both its binary and symbolic assembly versions. The project that accompanies this chapter deals with writing a couple of machine language programs.

Although most people will never write programs directly in machine language, the study of low-level programming is a pre-requisite to a complete understanding of the computer’s anatomy. Also, it is rather fascinating to realize how the most sophisticated software systems are, at bottom, long series of elementary instructions, each specifying a very simple and primitive operation on the underlying hardware.

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

1. Background

This chapter is language-oriented. Therefore, we can abstract away most of the details of the underlying hardware platform. In particular, in order to give a general description of machine languages, it is sufficient to focus on three main hardware elements only: a *processor*, a *memory*, and a set of *registers*.

1.1 Machines

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 designed to store data and instructions. Some computer platforms store data and instructions in the same memory device, while others employ different data and instruction memories, each featuring a separate address space. Conceptually speaking, all memories have the same structure: a continuous array of cells of some fixed width, also called *words* or *locations*, each having a unique *address*. Hence, an individual word (representing either a data item or an instruction) is specified by supplying its address. In what follows we will refer to such individual words using the notations `Memory[address]`, `RAM[address]`, or `M[address]` for brevity.

Processor: The processor, normally called *Central Processing Unit* or *CPU*, is a device capable of performing a fixed set of operations. These typically include arithmetic and logic operations, memory access operations, and control (also called *branching*) operations. The operands of these operations are the current values of registers and selected memory locations. Likewise, the results of the operations can be stored either in registers or in selected memory locations.

Registers: Memory access is a relatively slow operation requiring long instruction formats (an address may require 32 bits). For this reason, most processors are equipped with several registers, each capable of holding a single value. Located in the processor’s immediate proximity, the registers serve as a high-speed local memory, allowing the processor to quickly store and retrieve data. This setting enables the programmer to minimize the use of memory access commands, thus speeding up the program’s execution. In what follows we will refer to the registers as `R0`, `R1`, `R2`, etc.

1.2 Languages

A machine language program is a series of coded instructions. For example, a typical instruction in a 16-bit computer may be “1010001100011001”. In order to figure out what this instruction means, we have to know the rules of the game, i.e. the instruction set of the underlying hardware platform. For example, the language may be such that each instruction consists of four 4-bit fields: the left-most field codes a CPU operation, and the remaining fields represent the operation’s operands. Thus the above command may code the operation “*set R3 to R1+R9*”, depending of course on the hardware specification and the machine language syntax.

Since binary codes are rather cryptic, machine languages are normally specified using both binary codes and symbolic mnemonics (a *mnemonics* is a symbolic label that “stands for” something -- in our case binary codes). For example, the language designer can decide that the operation code

“1010” will be represented by the mnemonic “add”, and that the registers of the machine will be symbolically referred to using the symbols R0, R1, R2, ... Using these conventions, one can specify machine language instructions either directly, as “1010001100011001”, or symbolically, as, say, “ADD R3, R1, R9”.

Taking this symbolic abstraction one step further, we can allow ourselves to not only *read* symbolic notation, but to actually *write* programs using symbolic commands rather than binary instructions. Next, we can use a text processing program to parse the symbolic commands into their underlying fields (mnemonics and operands), translate each field into its equivalent binary representation, and assemble the resulting codes into binary machine instructions. The symbolic notation is called *assembly language*, or simply *assembly*, and the program that translates from assembly to binary is called *assembler*.

Since different computers vary in terms of CPU operations, number and type of registers, and assembly syntax rules, the result is a tower of Babel of machine languages, each with its own obscure syntax. Yet irrespective of this variety, all machine languages support similar sets of generic commands, as we now turn to describe.

1.3 Commands

Arithmetic and logic commands: Every computer is required to perform basic arithmetic operations like addition and subtraction as well as basic Boolean operations like bit-wise negation, bit shifting, etc. Different machines feature different sets and versions of such operations, and different ways to apply them to combinations of registers and selected memory locations. Here are some typical possibilities that can be found in various machines:

```
// In all the examples, x is a user-defined label referring to a certain memory location.
```

```
ADD R2, R3 // R2 ← R2 + R3 where R2 and R3 are registers
```

```
ADD R2, x // R2 ← R2 + x
```

```
AND R4, R5, R2 // R4 ← bit wise “And” of R5 and R2
```

```
SUBD x // D ← (D - x) where D is a register
```

```
ADD x // add the value of x to a special register called “accumulator”
```

Memory Access commands: Memory access commands fall into two categories. First, as we have just seen, in some cases arithmetic and logical commands are allowed to operate on selected memory locations. Second, all computers feature explicit *load* and *store* commands, designed to move data between the registers and the memory.

Memory access commands may use several types of *addressing modes* -- ways of specifying the address of the required memory word. As usual, different computers offer different possibilities and different notations, but three memory access modes are almost always supported:

- **Direct addressing:** The most common way to address the memory is to express a specific address or use a symbol that refers to a specific address:

```
LOAD R1, 67    // R1 ← Memory[67]
```

```
// Assume that sum refers to memory address 67
```

```
LOAD R1, sum   // R1 ← Memory[67]
```

- **Immediate addressing:** This form of addressing is used to load constants – i.e. load values that appear in the instruction proper: instead of treating the field that appears in the “load” command as an address, we simply load the value of the field itself into the register.

```
LOADI R1, 67  // R1 ← 67
```

- **Indirect addressing:** In this addressing mode the address of the required memory location is not hard-coded into the instruction; instead, the instruction specifies a memory location that holds the required memory address. This addressing mode is used to manage *pointers* in high-level programming languages. For example, consider the high-level command “`x=arr[j]`” where `arr` is an array and `x` and `j` are variables. How can we translate this command into machine language? Well, when the array `arr` is declared and initialized in the high-level program, a memory segment of the correct length is allocated to hold the array data. Second, another memory location, referred to by the symbol `arr`, is allocated to hold the *base address* of the array’s segment.

Now, when the compiler is asked to translate a reference to cell `arr[j]`, it goes through the following process. First, note that the `j`’th entry of the array should be physically stored in a memory location that is at a displacement `j` from the array’s base address (assuming, for simplicity, that each array element uses a single word). Hence the address corresponding to the expression `arr[j]` can be easily calculated by adding the value of `j` to the value of `arr`. Thus in the C programming language, for example, a command like `x=arr[j]` can be also expressed as `x=*(arr+j)`, where the notation “*n” stands for “the value of `Memory[n]`”. When translated into machine language, such commands typically yield the following code (depending on the assembly language syntax):

```
// translation of x=arr[j] or x=*(arr+j):
ADD R2, arr, j    // R2 ← arr+j
LOAD* R1, R2     // R1 ← memory[R2]
STR R1, x        // x ← R1
```

Flow of control commands: While programs normally execute in a linear fashion, one command after the other, they also include occasional branches to locations other than the next command. Branching serves several purposes including *repetition* (jump backward to the beginning of a loop), *conditional execution* (if a Boolean condition is false, jump forward to the location after the “if-then” clause), and *subroutine calling* (jump to the first command of some other code segment). In order to support these programming constructs, every machine language features means to jump to various locations in the program, both conditionally and unconditionally. In assembly languages, locations in the program can also be given symbolic names, using some syntax for specifying labels. Program 1 illustrates a typical example.

High level

```
// a while loop
while (R1>=0) {
    code segment 1
}
code segment 2
```

Low level

```
// typical translation
beginWhile:
    JNG R1,endWhile // if R1<0 goto endWhile
    here comes the translation of code segment 1
    JMP beginWhile // goto beginWhile
endWhile:
    here comes the translation of code segment 2
```

PROGRAM 1: High- and low-level branching logic. The syntax of *goto* commands varies from one language to another, but the basic idea is the same.

Unconditional jump commands like “JMP beginWhile” specify only the address of the target location. *Conditional jump* commands like “JNG R1,endWhile” must also specify a condition, expressed in some way. In some languages the condition is an explicit part of the command, while in others it is a by-product of a previous command. Here are some possible examples (noting again that the commands’ syntax is less important than their general spirit):

```
// Assume that the foo label is defined elsewhere in the program (not shown here).

JGE R1,foo // if R1>=0 then goto foo

SUB R1,R2;JEQ foo // R1←R1-R2; if (result=0) then goto foo

JZR foo // if (result of the previous command = 0) then goto foo

JMP foo // goto foo (unconditionally)
```

* * *

This ends our general and informal introduction of machine languages, and the generic commands that can typically be found in various hardware platforms. The next section will be more formal, since it describes one specific machine language -- the native code of the computer that we will build in the next chapter.

2. Hack Machine Language Specification

2.1 Overview

Hack is a typical Von Neumann platform: a 16-bit machine, consisting 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.

Memory Address Spaces: The Hack programmer is aware of two distinct memory address spaces: an *instruction memory* and a *data memory*. Both memories are 16-bit wide and have a 15-bit address space, meaning that the maximum size of each memory is 32K 16-bit words.

The CPU can only execute programs that reside in the instruction memory. The instruction memory is a read-only device, and thus programs are loaded into it using some exogenous means. For example, the instruction memory can be implemented in a ROM chip which is pre-burned with the required program. Loading a new program can be done by replacing the entire ROM chip (similar to replacing a cartridge in a game computer). In order to simulate this operation, hardware simulators of the Hack platform must provide means to load the instruction memory from a text file containing a machine language program.

Registers: The Hack programmer is aware of two registers called *D* and *A*. These general-purpose 16-bit registers can be manipulated explicitly by arithmetic and logical instructions, e.g. $A=D-1$ or $D=!A$ (where “!” means 16-bit “not”). While *D* is used solely to store data values, *A* doubles as both a data register and an address register. That is to say, depending on the instruction context, the contents of *A* can be interpreted either as a data value, or as an address in the data memory, or as an address in the instruction memory, as we now turn to explain.

First, the *A* register can be used to facilitate direct access to the data memory (which, from now on, will be often referred to as “memory”). As the next section will describe, the syntax of the Hack language is such that memory access instructions do not specify an explicit address. Instead, they operate on an implicit memory location labeled “*M*”, e.g. $D=M+1$. In order to resolve this address, the contract is such that *M* always refers to the memory word whose address is the current value of *A*. For example, if we want to effect the operation $D=Memory[516]-1$, we have to set the *A* register to 516, and then issue the instruction $D=M-1$.

Second, in addition to doubling as a general-purpose register and as an address register for the data memory, the hard working *A* register is also used to facilitate direct access to the instruction memory. As we will see shortly, the syntax of the Hack language is such that jump instructions do not specify a particular address. Instead, the contract is such that any jump operation always affects a jump to the instruction memory word addressed by *A*. For example, if we want to effect the operation “goto 35”, we set *A* to 35 and issue a “goto” command. This will cause the computer to fetch the instruction located in `InstructionMemory[35]` in the next clock cycle.

Example: Since the Hack language is quite self-explanatory, we start with an example. The only non-obvious command in the language is “@address”, where *address* is either a number or a symbol representing a number. This command simply stores the specified value into the A register. For example, if *sum* refers to memory location 17, then both “@17” and “@sum” will have the same effect: $A \leftarrow 17$.

And now to the example: Suppose we have to add all the numbers between 1 and 100, using repetitive addition. Program 2 gives a C language solution and a possible compilation into the Hack language.

C language

```
//sum the numbers 1...100
int i=1;
int sum=0;
while (i<=100){
    sum+=I;
    i++;
}
```

Hack machine language

```
//sum the numbers 1...100
    @i      // i refers to some mem. loc.
    M=1     // i=1
    @sum    // sum refers to some mem. loc.
    M=0     // sum=0
(loop)
    @i
    D=M     // D=i
    @100
    D=D-A   // D=i-100
    @end
    D;jgt   // if (i-100)>0 goto end
    @i
    D=M     // D=i
    @sum
    M=D+M   // sum=sum+i
    @i
    M=M+1   // i=i+1
    @loop
    0;jmp   // goto loop
(end)
```

PROGRAM 2: C and assembly versions of the same program.

Although the Hack syntax is more accessible than that of typical machine languages, it may still look rather obscure for readers who are not used to low-level programming. In particular, note that every operation involving a memory location requires two Hack commands: one for selecting the address on which we want to operate, and one for specifying the desired operation. Indeed, the Hack language consists of two generic instructions: an *address instruction*, also called *A-instruction*, and a *compute instruction*, also called *C-instruction*. Each instruction has a binary representation, a symbolic representation, and an effect on the computer, as we now turn to specify.

2.2 The A-Instruction

The *A*-instruction is used to set the *A* register to a 15-bit value:

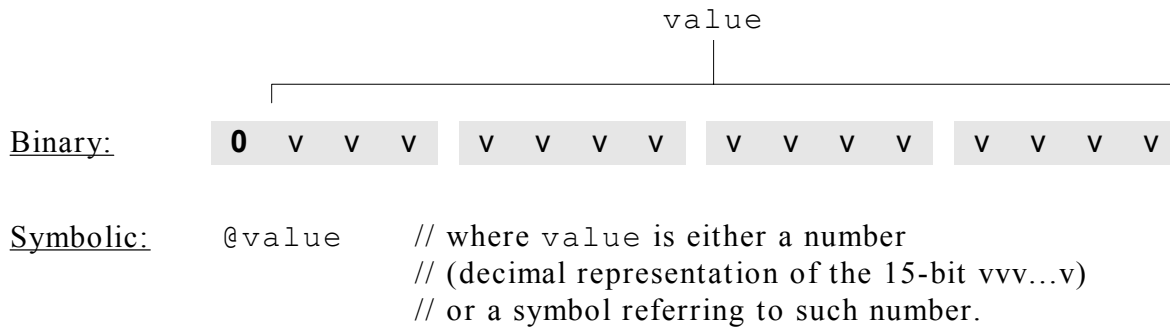


FIGURE 3: A-Instruction syntax.

This instruction causes the computer to store a constant in the *A* register. For example, the instruction @5, which is equivalent to 0000000000000101, causes the computer to store 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 designed to manipulate a certain data memory location, by first setting *A* to the address of that location. Third, it sets the stage for a subsequent *C*-instruction that involves a jump, by first loading the address of the jump destination to the *A* register. These uses will be demonstrated below.

2.3 The C-Instruction

The *C*-instruction is the programming workhorse of the Hack platform -- the instruction that gets almost everything done. The instruction code is a specification that answers three questions: (a) what to compute? (b) where to store the computed value? and (c) what to do next? Along with the *A*-instruction, these specifications determine all the possible operations of the computer.

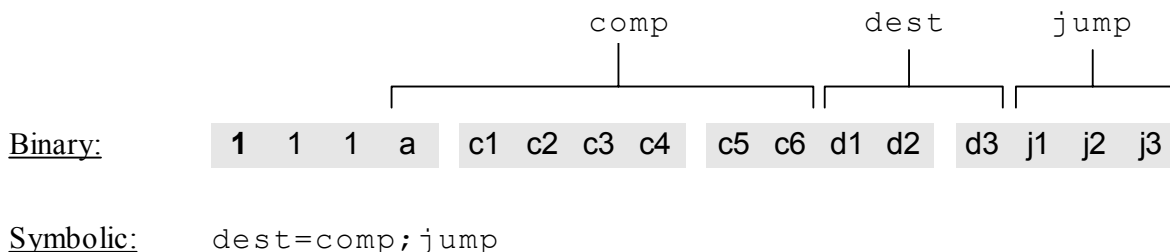


FIGURE 4: C-Instruction syntax.

The MSB is the *C*-instruction code, which is 1. The next two bits are not used. The remaining bits form three fields that correspond to the three parts of the instruction's symbolic representation. Taken together, the semantics of the symbolic instruction `dest=comp; jump` is as follow. The `comp` field instructs the CPU what to compute. The `dest` field instructs where to

store the computed value. The `jump` field specifies a jump condition. Either the `dest` field or the `jump` field or both may be empty. If the `dest` field is empty then the “=” sign may be omitted. If the `jump` field is empty then the “;” symbol may be omitted. We now turn to describe the format and semantics of each of the three fields.

The computation specification: The Hack ALU is designed to compute a fixed set of functions on the `D`, `A`, and `M` registers (where $M = \text{Memory}[A]$). 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 functions, of which only the 28 listed in Table 5 are documented in the language specification.

a=0							
mnemonic	c1	c2	c3	c4	c5	c6	
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	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	1	0	D+M
D-A	0	1	0	0	1	1	D-M
A-D	0	0	0	1	1	1	M-D
D&A	0	0	0	0	0	0	D&M
D A	0	1	0	1	0	1	D M
	c1	c2	c3	c4	c5	c6	mnemonic
a=1							

TABLE 5: The "compute" specification of the C-instruction. `D` and `A` are names of registers. `M` refers to the memory location addressed by `A`, i.e. to $\text{Memory}[A]$. The symbols “+” and “-” denote 16-bit 2’s complement addition and subtraction, while “!”, “|”, and “&” denote the 16-bit bit-wise Boolean operators `Not`, `Or`, `And`, respectively. Note the similarity between this instruction set and the ALU specification given in Table 7 of Chapter 2.

Recall that the format of the `C`-instruction is “111a cccc codd djjj”. Suppose we want to compute `D-1`, i.e. “the current value of the `D` register minus 1”. According to Table 5, this can be

done by issuing the instruction "1110 0011 10xx xxxx" (we use "x" to label bits that are irrelevant to the given example). To compute the value of $D|M$, we issue the instruction "1111 0101 01xx xxxx". To compute the constant -1, we issue the instruction "1110 1110 10xx xxxx", and so on.

The destination specification: The value computed by the `comp` part of the *C*-instruction can be simultaneously stored in several destinations, as specified by the instruction's `dest` part. 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* (i.e. in $\text{Memory}[A]$). One, more than one, or none of these bits may be asserted.

<code>d1</code>	<code>d2</code>	<code>d3</code>	<i>mnemonic</i>	<i>destination (where to store the computed value)</i>
0	0	0	null	The value is not stored anywhere
0	0	1	<i>M</i>	$\text{Memory}[A]$ (memory register addressed by <i>A</i>)
0	1	0	<i>D</i>	<i>D</i> register
0	1	1	<i>MD</i>	$\text{Memory}[A]$ and <i>D</i> register
1	0	0	<i>A</i>	<i>A</i> register
1	0	1	<i>AM</i>	<i>A</i> register and $\text{Memory}[A]$
1	1	0	<i>AD</i>	<i>A</i> register and <i>D</i> register
1	1	1	<i>AMD</i>	<i>A</i> register, $\text{Memory}[A]$, and <i>D</i> register

TABLE 6: The "destination" specification of the *C*-instruction.

Recall that the format of the *C*-instruction is "111a cccc cddd djjj". Suppose we want the computer to increment the value of $\text{Memory}[7]$ by 1, and also store the result in the *D* register. According to tables 5 and 6, this can be accomplished by the instructions:

```
0000 0000 0000 0111    // @7
1111 1101 1101 1xxx    // DM=M+1 (x=irrelevant bits)
```

The *A*-instruction causes the computer to select the memory register whose address is 7 (the so called "*M* register"). The subsequent *C*-instruction computes the value of $M+1$ and stores the result in both *D* and *M*. The role of the 3 LSB bits of the second instruction is explained next.

The jump specification: The `jump` field of the *C*-instruction tells the computer what to do next. There are two possibilities: the computer should either fetch and execute the next instruction in the program, which is the default, or it should fetch and execute an instruction located elsewhere in the program. In the latter case, we assume that the *A* register has been previously set to the address to which we want to jump.

The jump itself is performed conditionally according to the value computed in the "comp" part of this instruction. The first `j`-bit specifies whether to jump in case this value is negative, the second `j`-bit in case the value is zero, and the third `j`-bit in case it is positive. This gives 8 possible jump conditions.

j1 (<i>out</i> < 0)	j2 (<i>out</i> = 0)	j3 (<i>out</i> > 0)	Mnemonic	Effect
0	0	0	null	no jump
0	0	1	JGT	if <i>out</i> > 0 jump
0	1	0	JEQ	if <i>out</i> = 0 jump
0	1	1	JGE	if <i>out</i> ≥ 0 jump
1	0	0	JLT	if <i>out</i> < 0 jump
1	0	1	JNE	if <i>out</i> ≠ 0 jump
1	1	0	JLE	if <i>out</i> ≤ 0 jump
1	1	1	JMP	jump

TABLE 7: The "jump" specification of the C-instruction. *Out* refers to the value computed by the instruction's `comp` part, and *jump* implies "continue execution with the instruction addressed by the A register".

The following example illustrates the jump commands in action:

Logic

```
if Memory[3]=5 then
    goto 100
else goto 200
```

Implementation

```
@3
D=M // D=Memory[3]
@5
D=D-A // D=D-5
@100
D;JEQ // if D=0 goto 100
@200
0;JMP // goto 200
```

The last instruction ("`0;JMP`") effects an unconditional jump. Since the *C*-instruction syntax requires that we always effect *some* computation, we instruct the ALU to compute 0 (an arbitrary choice), which is ignored.

Conflicting uses of the A register: As was just illustrated, the programmer can use the A register in order to select either a *data memory* location for a subsequent *C*-instruction involving `M`, or an *instruction memory* location for a subsequent *C*-instruction involving a jump. Thus, in order to prevent conflicting use of the A register, we require that in well written programs, a *C*-instruction that may cause a jump (i.e. with some non-zero *j* bits) should not contain a reference to `M`.

2.4 Symbols

Assembly commands can refer to memory locations (addresses) using either constants or *symbols*. Symbols are introduced into assembly programs in three ways:

- **Predefined symbols:** A special subset of RAM (data memory) addresses can be referred to by any assembly program using pre-defined symbols, as follows.
 - **Virtual registers:** the symbols R0 to R15 are pre-defined to refer to RAM addresses 0 to 15, respectively. This syntactic convention is designed to simplify assembly programming.
 - **VM pointers:** the symbols SP, LCL, ARG, THIS, and THAT are pre-defined to refer to RAM addresses 0 to 4, respectively. Note that each of these memory locations has two labels, e.g. address 2 can be referred to using either R2 or ARG. This syntactic convention will come to play in the implementation of the virtual machine, discussed in Chapters 7 and 8.
 - **I/O Pointers:** the symbols SCREEN and KBD are pre-defined to refer to RAM addresses 16384 (0x4000) and 24576 (0x6000), respectively, which are the base addresses of the screen and keyboard memory maps. The use of these I/O devices is explained below.
- **Label symbols:** These user-defined symbols, which serve to label destinations of *goto* commands, are declared by the pseudo command “(xxx)”. This directive defines the symbol xxx to refer to the instruction memory location holding the next command in the program. A label can be defined only once and can be used anywhere in the assembly program, even before the line in which it is defined.
- **Variable symbols:** Any user-defined symbol xxx appearing in an assembly program that is not predefined and is not defined elsewhere using the “(xxx)” command is treated as a *variable*, and is mapped by the assembler to an available RAM location. Variables are mapped, as they are first encountered, to consecutive memory locations starting at RAM address 16 (0x0010).

2.5 Input / Output Handling

The Hack platform can be connected to two peripheral devices: a screen and a keyboard. Both devices interact with the computer platform through *memory maps*. This means that drawing pixels on the screen is achieved by writing binary values into a memory segment associated with the screen. Likewise, “listening” to the keyboard is done by reading a memory location associated with the keyboard. The physical I/O devices and their memory maps are synchronized via continuous refresh loops.

Screen: The Hack computer can be connected to a black-and-white screen organized as 256 rows of 512 pixels per row. The screen’s contents are represented by an 8K memory map that starts at RAM address 16384 (0x4000). Each row in the physical screen, starting at the screen’s top left corner, is represented in the RAM by 32 consecutive 16-bit words. Thus the pixel at row r from the top and column c from the left is mapped on the $c\%16$ bit (counting from LSB to

MSB) of the word located at $\text{RAM}[16384+r*32+c/16]$. To write or read a pixel of the physical screen, one reads or writes the corresponding bit in the RAM-resident memory map (1=black, 0=white). Example:

```
// Draw a single black dot at the top left corner of the screen:
@SCREEN // Set the A register to point to the memory word that is mapped
        // to the 16 left-most pixels of the top row of the screen
M=1     // Blacken the left-most pixel
```

Keyboard: The Hack computer interfaces with the physical keyboard via a single-word memory map located in RAM address 24576 (0x6000). Whenever a key is pressed on the physical keyboard, its 16-bit ASCII code appears in $\text{RAM}[24576]$. When no key is pressed, the code 0 appears in this location. In addition to the usual ASCII codes, the Hack keyboard recognizes the following keys:

Key pressed	Code	Key pressed	Code
new line	128	end	135
backspace	129	page up	136
left arrow	130	page down	137
right arrow	131	insert	138
up Arrow	132	delete	139
down arrow	133	esc	140
home	134	f1-f12	141-152

TABLE 8: Special keyboard codes in the Hack language

2.6 Syntax Conventions and Files Format

Binary code files: A binary code file is composed of text lines. Each line is a sequence of 16 “0” and “1” ASCII characters, coding a single machine language instruction. Taken together, all the lines in the file represent a machine language program. The contract is such that when a machine language program is loaded into the computer’s instruction memory, the binary code represented by the file’s n -th line is stored in address n of the instruction memory (the count of both program lines and memory addresses starts at 0).

By convention, machine language programs are stored in text files with a “hack” extension, e.g. `Prog.hack`.

Assembly language files: By convention, assembly language programs are stored in text files with an “asm” extension, e.g. `Prog.asm`. An assembly language file is composed of text lines, each representing either an *instruction* or a *symbol declaration*:

- **Instruction:** an A -instruction or a C -instruction.
- **(Symbol):** This pseudo-command causes the assembler to assign the label `Symbol` to the memory location into which the next command in the program will be stored. It is called “pseudo-command” since it generates no machine code.

Constants and symbols in assembly programs: *Constants* must be non-negative and are always written in decimal notation. A user-defined *symbol* can be any sequence of letters, digits, underscore (“_”), dot (“.”), dollar sign (“\$”), and colon (“:”) that does not begin with a digit.

Comments in assembly programs: text beginning with two slashes (“//”) and ending at the end of the line is considered a comment and is ignored.

White space in assembly programs: space characters are ignored. Empty lines are ignored.

Case conventions: All the assembly mnemonics must be written in upper-case. The rest (user-defined labels and variable names) is case sensitive. The convention is to use upper-case for labels and lower-case for variable names.

3. Perspective

The Hack machine language is almost as simple as machine languages get. Most computers usually have more instructions, more data-types, more registers, more instruction formats, and more addressing modes. At the same time, any feature not supported by the Hack machine language may still be implemented in software, at a performance cost. For example, the Hack platform does not supply multiplication and division as machine-language operations. Since these operations are obviously required by any high-level language, we will later implement them at the operating system level (Chapter 12).

One of the main characteristics that give machine languages their particular flavor is the number of memory addresses that can appear in a single command. In this respect, Hack may be described as a $\frac{1}{2}$ -address machine: we usually require two Hack instructions to perform an operation involving a single memory address: an *A*-instruction to specify the address, and a *C*-instruction to specify the operation. In comparison, most machine languages can directly specify at least one address in every machine instruction.

In terms of assembly style, we have chosen to give Hack a somewhat different look-and-feel than the mechanical nature of most assembly languages. In particular, we have chosen a high-level language-like syntax for the *C*-command, e.g. “D=M” and “D=D+M” instead of the more traditional “LOAD” and “ADD” commands, respectively. The reader should note however that these are just syntactic details. Further, one can design a macro-Hack language with commands like “D=M[address]”, “goto address”, and so on. These macro-commands can be easily translated by the assembler into the sequences “@address” followed by “D=M” and “@address” followed by “0; jmp”, and so on.

The *assembler*, which was mentioned several 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 replacing them with physical addresses of actual memory locations. We will return to this translation task in Chapter 7, in which we build an assembler for the Hack language. But first, we have to complete the construction of the Hack hardware platform, a challenge that is taken up in the next chapter.

4. Build it

Objective: To get a taste of low-level programming in machine language, and to get acquainted with the Hack computer platform. In the process of working on this project, you will get a hands-on understanding of the assembly process, and you will appreciate visually how the translated binary code executes on the target hardware.

Resources: In this project you will use two main tools supplied with the book: an *Assembler*, designed to translate Hack assembly programs into binary code, and a *CPU Emulator*, designed to run binary programs on a simulated Hack platform.

Contract: Write and test the two programs described below. When executed on the CPU Emulator, your programs should generate the results mandated by the supplied test scripts.

- **Multiplication program** (`Mult.asm`): The inputs of this program are the current values stored in `R0` and `R1` (i.e. the two top RAM locations). The program computes the product $R0 * R1$ and stores the result in `R2`. The algorithm can be iterative addition. We assume (in this program) that $R0 \geq 0$, $R1 \geq 0$, and $R0 * R1 < 32768$. Your program need not test these conditions, but rather assume that they hold. The supplied `Mult.tst` and `Mult.cmp` scripts will test your program on several representative data values.
- **I/O-Handling Program** (`Fill.asm`): This program runs an infinite loop that “listens” to the keyboard input. When a key is pressed (any key), the program blackens the screen, i.e. writes "black" in every pixel. When no key is pressed, the screen should be cleared. You may choose to blacken and clear the screen in any order, 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. Note: this program has a test script (`Fill.tst`) but no compare file – it should be checked by visibly inspecting the simulated screen.

Steps: We recommend proceeding as follows:

1. The *Assembler* and *CPU Emulator* programs needed for this project are available in the *tools* directory of the book software suite.
2. Go through the *Assembler Tutorial* and the *CPU Emulator Tutorial*.
3. Download the `project4.zip` file and extract its contents to a directory called `project4` on your computer. This will create two directories called `project4/mult` and `project4/fill`. As a rule, all the files related to each program (`.asm`, `.hack`, `.tst` and `.cmp`) must be stored in the same directory.
4. Use a text editor to write the first program in assembly, and save it as `.../mult/Mult.asm`.
5. Use the supplied Assembler (in either batch or interactive mode) to debug and translate your program. The result will be a binary file called `Mult.hack`.
6. Use the supplied CPUemulator to test your `Mult.hack` code. You can begin by loading `Mult.hack` into the simulator and testing it in interactive fashion. Then load the supplied `Mult.tst` script into the simulator, and execute it in order to run our “certified test”.
7. Repeat stages 4-6 for the second program (`Fill.asm`).

5. Computer Architecture ¹

“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 all the chips that we built in previous chapters, and integrate them into a general-purpose computer capable of running stored programs written in a machine language. The specific computer that we will build, called *Hack*, has two important virtues. On the one hand, Hack is a simple machine that can be constructed in just 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 digital computer. Therefore, building it will give you an excellent understanding of how modern computers work at the low hardware and software levels.

Following an introduction of the *stored program* concept, Section 1 gives a detailed description of the *von Neumann architecture* -- a central dogma in computer science underlying the design of almost all modern computers. The Hack platform is one example of a von Neumann machine, and Section 2 gives its exact hardware specification. Section 3 describes how the Hack platform can be implemented from available chips, in particular the ALU built in Chapter 2 and the registers and memory systems built in Chapter 3.

In the spirit of the opening quote of this chapter, the computer that will emerge from this construction will be as simple as possible, but not simpler. This means that it will have the minimal configuration necessary to run interesting programs and deliver reasonable performance. The comparison of this machine to typical computers is taken up in Section 4, which emphasizes the critical role that *optimization* plays in the design of industrial-strength computers, but not in this chapter. As usual, the simplicity of our approach has a purpose: all the chips mentioned in the chapter, culminating in the Hack computer itself, can be built and tested on a personal computer, following the technical instructions given in the chapter's last section. The result will be a minimal yet surprisingly powerful computer.

1. Background

The Stored Program Concept

Compared to all the other machines around us, the most unique feature of the digital computer is its amazing versatility. Here is a machine with finite hardware that can perform a practically infinite array of tasks, from interactive games to word processing to scientific calculations. This remarkable flexibility -- a boon that we have come to take for granted -- is the fruit of a brilliant idea called the *stored program* concept. Formulated independently by several mathematicians in the 1930s, the stored program concept is still considered the most profound invention in, if not the very foundation of, modern computer science.

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

Like many scientific breakthroughs, the basic idea is rather simple. The computer is based on a fixed hardware platform, capable of executing a fixed repertoire of instructions. At the same time, these instructions can be used and combined like building blocks, yielding arbitrarily sophisticated programs. Importantly, the logic of these programs is not embedded in the hardware platform, as it was in mechanical computers predating 1930. Instead, the program's code is stored and manipulated in the computer memory, *just 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.

The von-Neumann Architecture

The stored program concept is a key element of many abstract and practical computer models, most notably the *Universal Turing machine* (1936) and the *von Neumann machine* (1945). The Turing machine -- an abstract artifact describing a deceptively simple computer -- is used mainly to analyze the logical foundations of computer systems. In contrast, the von Neumann machine is a practical architecture and the conceptual blueprint of almost all computer platforms today.

The von Neumann architecture is based on a *central processing unit* (CPU), interacting with a *memory* device, receiving data from some *input* device, and sending data to some *output* device (figure 1). 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 very instructions that tell the computer what to do. We now turn to describe this architecture in some detail.

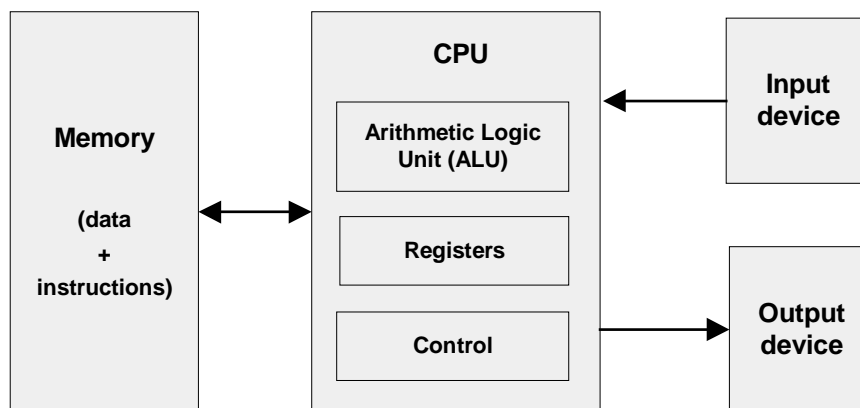


FIGURE 1: The von-Neumann Architecture (conceptual), which, at this level of detail, describes the architecture of almost all digital computers. The program that operates the computer resides in its memory, in accordance with the stored program concept.

Memory

The memory in a von-Neumann machine holds two types of information: data items and programming instructions. The two types of information are usually treated differently, and in some computers are stored in separate memory units. In spite of their different functions, both types of information are represented as binary numbers which are stored in the same generic

random-access structure: a continuous array of cells of some fixed width, also called *words* or *locations*, each having a unique *address*. Hence, an individual word (representing either a data item or an instruction) is specified by supplying its address.

Data Memory: High-level programs manipulate abstract artifacts like variables, arrays, and objects. When translated into machine language, these data abstractions become series of binary numbers, stored in the computer's data memory. Once an individual word has been selected from the data memory by specifying its address, it can be either *read* or *written* to. In the former case, we retrieve the word's value. In the latter case, we store a new value into the selected location, erasing the old value.

Instruction memory: High level programs use structured commands like `while j<100 {sum=sum+j}`. When translated into machine language, such a command becomes a series of words, each representing a single machine language instruction. These instructions are stored in the computer's instruction memory. In each step of the computer's operation, the CPU *fetches* (i.e. *reads*) a word from the instruction memory, decodes it, executes the underlying instruction, and figures out which instruction to execute next. Thus, changing the contents of the instruction memory has the effect of completely changing the computer's operation.

The instructions that reside in the instruction memory are written in an agreed upon formalism called *machine language*. Using these instructions, the programmer can command the CPU to perform arithmetic and logic operations, fetch and store values from and to the memory, move values from one register to another, test Boolean conditions, and so on. In some computers, the specification of each elementary operation (the operation code and the registers and/or memory locations on which it operates) is represented in one word. Computers with a "thin" word-width (e.g. 16-bit) may split this specification over several words.

Central Processing Unit

The CPU -- the centerpiece of the computer's architecture -- is in charge of executing the instructions of the currently loaded program. These instructions tell the CPU to carry out various calculations, to read and write values from and into the memory, and to conditionally jump to execute other instructions in the program. In order to execute these tasks, every CPU employs at least three hardware elements: an *Arithmetic-Logic Unit*, a set of *registers*, and a *control unit*.

Arithmetic-Logic Unit: the ALU is built to perform all the low-level arithmetic and logical operations featured by the computer. For instance, a typical ALU can add two numbers, test whether a number is positive, manipulate the bits in a word of data, and so on.

Registers: The CPU is designed to carry out simple calculations, quickly. In order to boost performance, the results of such calculations can often be stored locally, rather than shipped in and out of memory. Thus, every CPU is equipped with a small set of high-speed *registers*, each capable of holding a single word.

Control unit: A computer instruction is represented as a binary code, typically 16- or 32-bits wide. Before such an instruction can be executed, it must be decoded, and the information embedded in it must be used to signal various hardware devices (ALU, registers, memory) how to execute the instruction. The instruction decoding is done by the *control unit*, which is also responsible for figuring out which instruction to fetch and execute next.

The CPU operation can now be described as a repeated loop: fetch an instruction (word) from memory; decode it; execute it, fetch the next instruction, and so on. The instruction execution may involve one or more of the following micro tasks: have the ALU compute some value, manipulate internal registers, read a word from the memory, and write a word to the memory. In the process of executing these tasks, the CPU also figures out which instruction to fetch and execute next, as we describe below.

Registers

Memory access is a slow process. When the CPU is instructed to retrieve the contents of address j of the memory, the following process ensues: (a) j travels from the CPU to the RAM; (b) the RAM direct-access logic locates the memory register whose address is j ; (c) the contents of $\text{RAM}[j]$ travels back to the CPU. Registers provide the same service -- data retrieval and storage -- without the round-trip travel and search expenses. First, the registers reside physically inside the CPU chip, so accessing them is almost instantaneous. Second, there are typically only a handful of registers, compared to millions of memory cells. Therefore, machine language instructions can specify which registers they want to manipulate using just a few bits, resulting in shorter instruction formats.

Different CPUs employ different numbers of registers, of different types, for different purposes. In some computer architectures each register can serve more than one purpose:

Data registers: These registers give the CPU short-term memory services. For example, when calculating the value of $(a-b)*c$ where a , b and c are memory locations, we must first compute and remember the value of $(a-b)$. Although this result can be temporarily stored in some memory location, a better solution is to store it locally inside the CPU -- in a *data register*.

Addressing registers: The CPU has to continuously access the memory in order to read data and write data. In every one of these operations, we must specify *which* individual memory word has to be accessed, i.e. supply an address. In some cases this address appears as part of the current instruction, while in others it depends on the execution of a previous instruction. In the latter case, the address should be stored in a register whose contents can be later treated as a memory address -- an *addressing register*.

Program Counter (PC) register: When executing a program, the CPU must always keep track of the address of the next instruction that must be fetched from the instruction memory. This address is kept in a special register called *program counter*, or PC. The contents of the PC are then used as the address for fetching instructions from the instruction memory. Thus, in the process of executing the current instruction, the CPU updates the PC in one of two ways. If the current instruction contains no "goto" directive, the PC is incremented to point to the next instruction in the program. If the current instruction includes a "goto n " directive, the CPU loads n into the PC.

Input and Output

Computers interact with their external environments using a diverse array of input and output (I/O) devices. These include screens, keyboards, printers, scanners, network interface cards, CD-ROMs, etc., not to mention the bewildering array of proprietary components that embedded computers are called to control in automobiles, weapon systems, medical equipment, and so on.

There are two reasons why we will not concern ourselves here with the anatomy of these various devices. First, every one of them represents a unique piece of machinery requiring a unique knowledge of engineering. Second, and for this very same reason, computer scientists have devised various schemes to make all these devices look exactly the same to the computer. The simplest trick in this art is called *memory-mapped I/O*.

The basic idea is to create a binary emulation of the I/O device, making it “look” to the CPU like a normal segment of memory. In particular, each I/O device is allocated an exclusive area in memory, called its “memory map”. In the case of an *input* device, the memory map is made to continuously *reflect* the physical state of the device; In the case of an *output* device, the memory map is made to continuously *drive* the physical state of the device. When external events affect some input devices (e.g. pressing a key on the keyboard or moving the mouse), certain values are written in their respective memory maps. Likewise, if we want to manipulate some output devices (e.g. draw some pixels on the screen or move a robotic arm), we write some values in their respective memory maps. From the hardware point of view, this scheme requires each I/O device to provide an interface similar to that of a memory unit. From a software point of view, each I/O device is required to define an interaction contract – so that programs can access it correctly. As a side comment, given the multitude of available computer platforms and I/O devices, one can appreciate the crucial role that *standards* play in computer architectures.

We see that in a memory-mapped architecture, the design of the CPU and the overall platform can be totally independent of the number, nature, or make of the I/O devices that interact, or *will* interact, with the computer. 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 its base address (these one-time configuration tasks are typically done by the operating system). From this point onward, any program that wants to manipulate this I/O device can do so -- all it needs to do is manipulate bits in memory.

2. The Hack Hardware Platform Specification

2.1 Overview

The Hack platform is a 16-bit Von Neumann machine, consisting 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 computer can only execute programs that reside in the instruction memory. The instruction memory is a read-only device, and thus programs are loaded into it using some exogenous means. For example, the instruction memory can be implemented in a ROM chip which is pre-burned with the required program. Loading a new program can be done by replacing the entire ROM chip. In order to simulate this operation, hardware simulators of the Hack platform must provide means for loading the instruction memory from a text file containing a program written in the Hack machine language. (From now on, we will refer to the data memory and to the instruction memory as RAM and ROM, respectively.)

The Hack CPU consists of the ALU specified in Chapter 2 and three registers called *data register* (D), *address register* (A), and *program counter* (PC). D and A are general-purpose 16-bit registers that can be manipulated by arithmetic and logical instructions like $A=D-1$, $D=D|A$, and so on,

following the Hack machine language specification (chapter 4). While the *D*-register is used solely to store data values, the contents of the *A*-register can be interpreted in three different ways, depending on the instruction's context: as a data value, as a RAM address, or as a ROM address.

The Hack machine language is based on two 16-bit command types. The *address instruction* has the format “0vvvvvvvvvvvvvvvv” (each *v* is 0 or 1). This instruction causes the computer to load the 15-bit constant *vvv...v* into the *A*-register. The *compute instruction* has the format “111accccccdddjjj”. The *a*- and *c*-bits instruct the ALU which function to compute, the *d*-bits instruct where to store the ALU output, and the *j*-bits specify a jump condition, all according to the Hack machine language specification. The computer is built in such a way that the program counter (*PC*) is connected to the address input of the ROM. This way, the ROM always emits the contents of $ROM[PC]$. This value is called the *current instruction*. The overall computer operation during each clock cycle is as follows:

Execute: Parts of the current instruction are simultaneously fed to both the *A*-register and to the ALU. If it's an *address instruction* (most significant bit = 0), the *A*-register is set to the 15-bit constant embedded in the instruction, and the instruction execution is over. If it's a *compute instruction* (MSB=1), then the *a*- and *c*-bits tell the ALU which function to compute. The ALU output is then simultaneously routed to the *A* and *D* registers and to the RAM register currently addresses by *A*. Each one of these registers is equipped with a “load bit” that enables/disables it to incoming inputs. These load bits, in turn, are connected to the three *d*-bits of the current instruction. For example, “011” causes the machine to disable *A*, enable *D*, and enable $RAM[A]$ to load the ALU output.

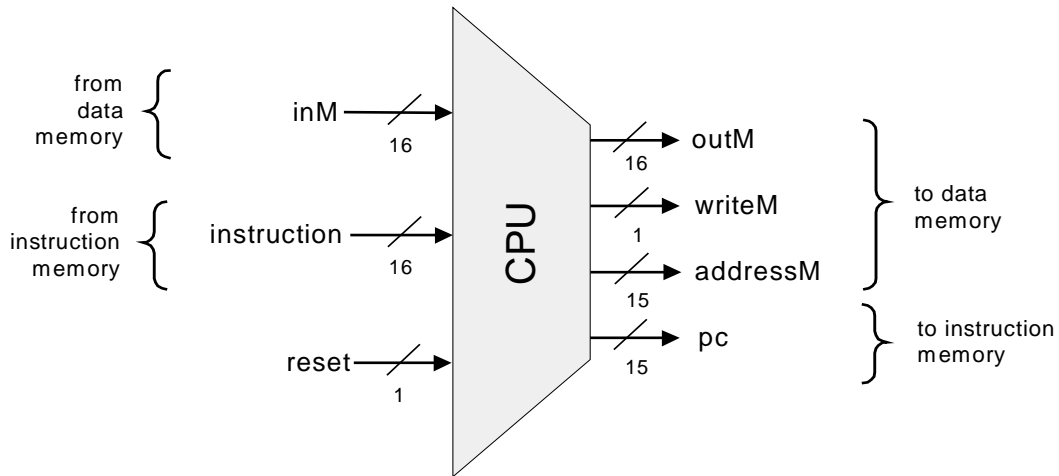
Fetch: Which instruction to fetch next is determined by the three jump bits of the current instruction and by the ALU status output bits. Taken together, these inputs determine if a jump should materialize. If so, the program counter (*PC*) is set to the value of the *A*-register; otherwise, the *PC* is incremented by 1. In the next clock cycle, the instruction that the program counter points at emerges from the ROM's output, and the cycle continues. We see that the $PC \leftarrow A$ setting causes the program flow to branch to the location specified by *A*, whereas the $PC \leftarrow PC + 1$ setting causes the program flow to continue with the next instruction in the program.

We now turn to formally specify the Hack hardware platform. Before starting, we wish to point out that most of this platform can be assembled from previously built components. The CPU is based on the *Arithmetic-Logic Unit* built in Chapter 2. The *registers* and the *program counter* are identical copies of the 16-bit register and 16-bit counter, respectively, built in chapter 3. Likewise, the ROM and the RAM chips are versions of the memory units built in Chapter 3. Finally, the *screen* and the *keyboard* devices will interface with the hardware platform through memory maps, implemented as built-in chips that have the same interface as RAM chips.

2.2 Central Processing Unit

The CPU of the Hack platform is designed to execute 16-bit instructions according to the Hack machine language specified in Chapter 4. It expects to be connected to two separate memory modules: an instruction memory, from which it fetches instructions for execution, and a data

memory, from which it can read, and into which it can write, data values. Diagram 2 gives the specification details.



```

Chip Name: CPU           // Central Processing Unit
Inputs:   inM[16],       // input from data memory (M)
             instruction[16], // instruction from instruction memory
             reset           // signals whether to re-start the current
                           // program (reset=1) or continue executing
                           // the current program (reset=0)
Outputs:  outM[16],     // output to data memory (M)
             writeM,        // write-enable the data memory
             addressM[15],  // address in data memory (of M)
             pc[15]         // address of next instruction
Function: Executes the inputted instruction according to the Hack machine
             language specification. The D and A in the language
             specification refer to CPU-resident registers, while M refers
             to the external memory location addressed by A, i.e. to
             Memory[A]. The inM input holds the value of this location.

             If the current instruction needs to write a value to M, the
             address of the target location is placed in the addressM
             output, the value is placed in outM, and the writeM control bit
             is asserted. (When writeM=0, any value may appear in outM).

             The outM and writeM outputs are combinational: they are
             affected instantaneously by the execution of the current
             instruction. The addressM and pc outputs are clocked: although
             they are affected by the execution of the current instruction,
             they commit to their new values only in the next time unit.

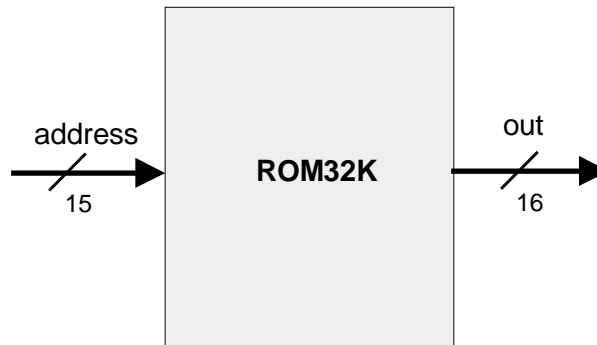
             If reset=1 then the CPU jumps to address 0 (i.e. sets pc=0 in
             next time unit) rather than to the address resulting from
             executing the current instruction.

```

DIAGRAM 2: The Central Processing Unit. This CPU can be built from the ALU and the registers built in Chapters 2 and 3, respectively.

2.3 Instruction Memory

The Hack instruction memory is implemented in a direct-access Read-Only Memory device, also called “ROM”. The Hack ROM consists of 32K addressable 16-bit registers:



Chip Name:	ROM	// 16-bit read-only 32K memory
Input:	address[15]	// Address in the ROM
Output:	out[16]	// Value of ROM[address]
Function:	out=ROM[address]	// 16-bit assignment
Comment:	The ROM is pre-loaded with a machine language program. Simulators must supply a mechanism for loading a program into the ROM.	

DIAGRAM 3: Instruction Memory.

2.4 Data Memory

Hack's *data memory* chip has the interface of a typical RAM device, like those built in Chapter 3 (see for example figure 3-3). To read the contents of register j , we put j in the memory's address input and probe the memory's out output. This is a combinational operation, independent of the clock. To write a value v into register j , we put v in the in input, j in the address input, and assert the memory's load bit. This is a sequential operation, and so register n will commit to the new value v in the next clock cycle.

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, using *memory maps*.

Memory Maps: In order to facilitate interaction with a user, the Hack platform can be connected to two peripheral devices: *screen* and *keyboard*. Both devices interact with the computer platform through *memory-mapped* buffers. Specifically, screen images can be drawn and probed by writing and reading, respectively, words in a designated memory segment called *screen memory map*. Similarly, one can check which key is presently pressed on the keyboard by probing a designated memory word called *keyboard memory map*. The memory maps interact with their respective I/O devices via peripheral logic that resides outside the computer. The contract is as follows: whenever a bit is changed in the screen's memory map, a respective pixel is

drawn on the physical screen. Whenever a key is pressed on the physical keyboard, the respective code of this key is stored in the keyboard's memory map.

We first specify the built-in chips that interface between the hardware interface and the I/O devices, and then the complete memory module that embeds these chips.

Screen: The Hack computer can be connected to a black-and-white screen organized as 256 rows of 512 pixels per row. The computer interfaces with the physical screen via a memory map, implemented by a chip called `Screen`. This chip behaves like regular memory, meaning that it can be read and written to. In addition, it features the side effect that any bit written to it is reflected as a pixel on the physical screen (1=black, 0=white). The exact mapping between the memory map and the physical screen coordinates is given in the chip API.

```
Chip Name: Screen          // memory-map of the physical screen
Inputs:   in[16],         // what to write
            load,           // write-enable bit
            address[13]     // where to write
Output:   out[16]        // screen value at the given address
Function: Functions exactly like a 16-bit 8K RAM:
            1. out(t)=Screen[address(t)](t)
            2. If load(t-1) then Screen[address(t-1)](t)=in(t-1)
            (t is the current time-unit, or cycle)
Comment:  Has the side effect of refreshing a 256 by 512 black-
            and-white screen (simulators must simulate this
            service). Each row in the physical screen is
            represented by 32 consecutive 16-bit words, starting
            with the top left corner of the screen. Thus the pixel
            at row r from the top and column c from the left
            (0<=r<=255, 0<=c<=511) reflects the c%16 bit (counting
            from LSB to MSB) of the word found in Screen[r*32+c/16].
```

DIAGRAM 4: Screen interface

Keyboard: The Hack computer can be connected to a standard keyboard, like that of a personal computer. The computer interfaces with the physical keyboard via a chip called `Keyboard`. Whenever a key is pressed on the physical keyboard, its 16-bit ASCII code appears as the output of the `Keyboard` chip. When no key is pressed, the chip outputs 0. In addition to the usual ASCII codes, the chip recognizes, and responds to, the keys listed in table 5.

Key pressed	Keyboard Output	Key pressed	Keyboard Output
new line	128	end	135
backspace	129	page up	136
left arrow	130	page down	137
right arrow	131	insert	138
up Arrow	132	delete	139
down arrow	133	esc	140
home	134	f1-f12	141-152

TABLE 5: Special keyboard keys

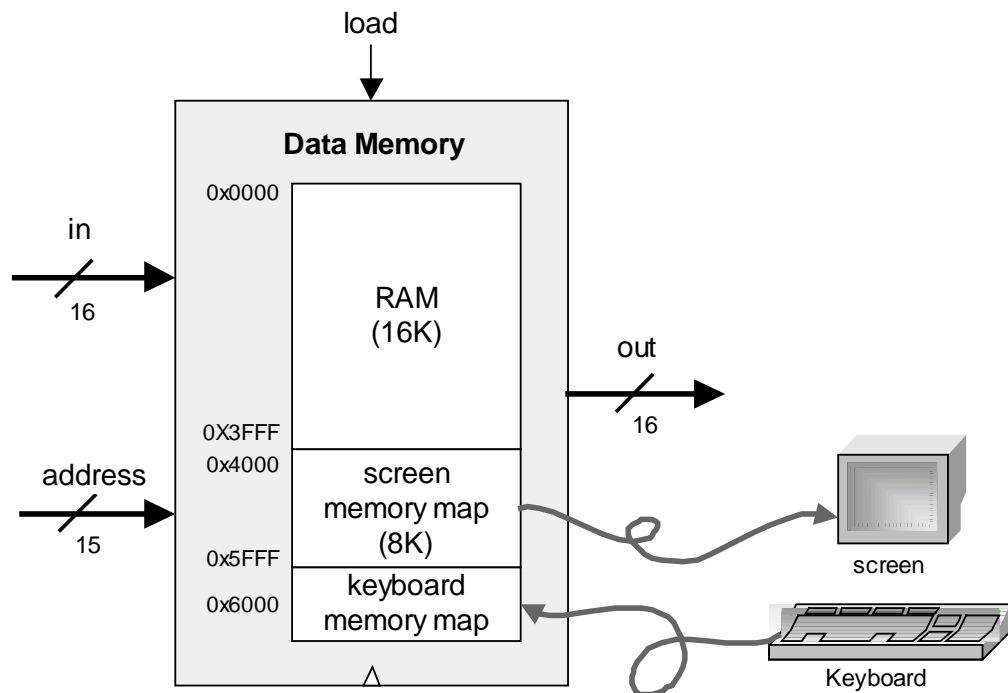
Chip Name:	<code>Keyboard</code>	<code>// Memory map of the physical keyboard. // Outputs the code of the currently // pressed key.</code>
Output:	<code>out[16]</code>	<code>// The ASCII code of the pressed key, or // one of the special codes listed in // Table 4-11, or 0 if no key is pressed.</code>
Function:	Outputs the code of the key presently pressed on the physical keyboard.	
Comment:	This chip is continuously being refreshed from a physical keyboard unit (simulators must simulate this service).	

DIAGRAM 6: Keyboard interface

Now that we've described the internal parts of the data memory, we are ready to specify the entire data memory address space.

Overall Memory: The overall address space of the Hack platform (i.e. its data memory) is provided by a chip called `Memory`. The memory chip includes the RAM (for regular data storage) and the screen and keyboard memory maps. These modules reside in a single address space that is partitioned into four sections:

- Addresses 0-16383 (0x0000-0x3FFF): Regular RAM (16K);
- Addresses 16384-24575 (0x4000-0x5FFF): Screen memory map (8K);
- Address 24576 (0x6000): Keyboard memory map (1 word);
- Addresses 24577-32767 (0x6001-0x7FFF): Unused segment.



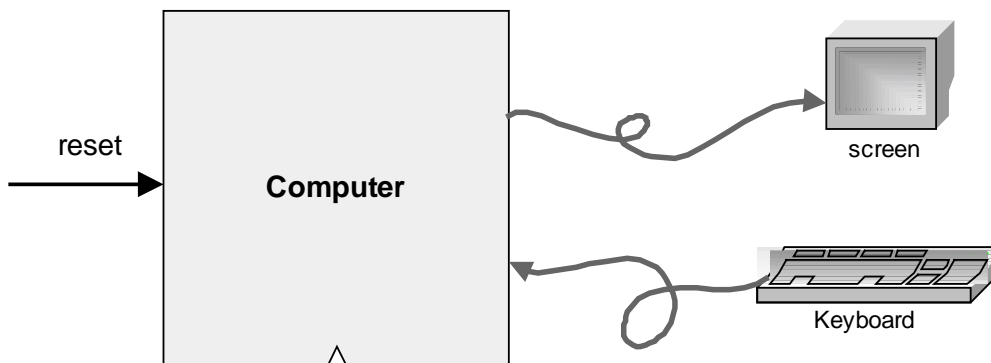
```

Chip Name: Memory      // complete memory address space
Inputs:   in[16],     // what to write
              load,       // write-enable bit
              address[15] // where to write
Output:   out[16]     // Memory value at the given address
Function: 1. out(t)=Memory[address(t)](t)
              2. If load(t-1) then Memory[address(t-1)](t)=in(t-1)
                 (t is the current time-unit, or cycle)
Comment:  Access to address>0x6000 is invalid. Access to any
                 address in the range 0x4000-0x5FFF results in
                 accessing the screen memory map. Access to address
                 0x6000 results in accessing the keyboard memory map.
                 The behavior in these addresses is described in the
                 Screen and Keyboard specifications.
  
```

DIAGRAM 7: Data Memory.

2.5 Computer

The top-most chip in the Hack hardware hierarchy is a complete computer system designed to execute programs written in the Hack machine language. This Computer chip contains all the hardware devices necessary to operate the computer, including a CPU, a data memory, an instruction memory (ROM), a screen, and a keyboard, all implemented as internal parts. In order to execute a program, the program's code must be pre-loaded into the ROM. Control of the screen and the keyboard is achieved via their memory maps, as described in their specifications.



Chip Name: Computer // top-most chip in the Hack platform
Input: reset
Function: When reset is 0, the program stored in the computer's ROM executes. When reset is 1, the execution of the program restarts. Thus, to start a program's execution, reset must be pushed "up" (1) and "down" (0).

Depending on the program's code, the screen will show some output and the user will be able to interact with the computer via the keyboard.

From this point onward the user is at the mercy of the person or company who wrote the software.

DIAGRAM 8: Computer. Top-most chip of the Hack hardware platform.

3. Implementation

This section gives general guidelines on how the Hack platform can be built to deliver the various services described in its specification (Section 2). As usual, we don't give exact building instructions, since we expect readers to come up with their own designs. All the chips can be built in HDL and simulated on a personal computer using the hardware simulator that comes with the book. As usual, technical details are given in the final "Build It" section (section 5).

Since most of the action in the Hack platform occurs in its Central Processing Unit, the main implementation challenge is building the CPU. The construction of the rest of the computer is straightforward.

3.1 The Central Processing Unit

The CPU implementation objective is to create a logic gate architecture capable of executing a given Hack instruction and fetching the next instruction to be executed. Naturally, the CPU will include an ALU capable of executing Hack instructions, a set of registers, and some control logic designed to fetch and decode instructions. Since almost all these hardware elements were already built in previous chapters, the key question here is how to connect them in order to effect the desired CPU operation. One possible solution is illustrated in diagram 9.

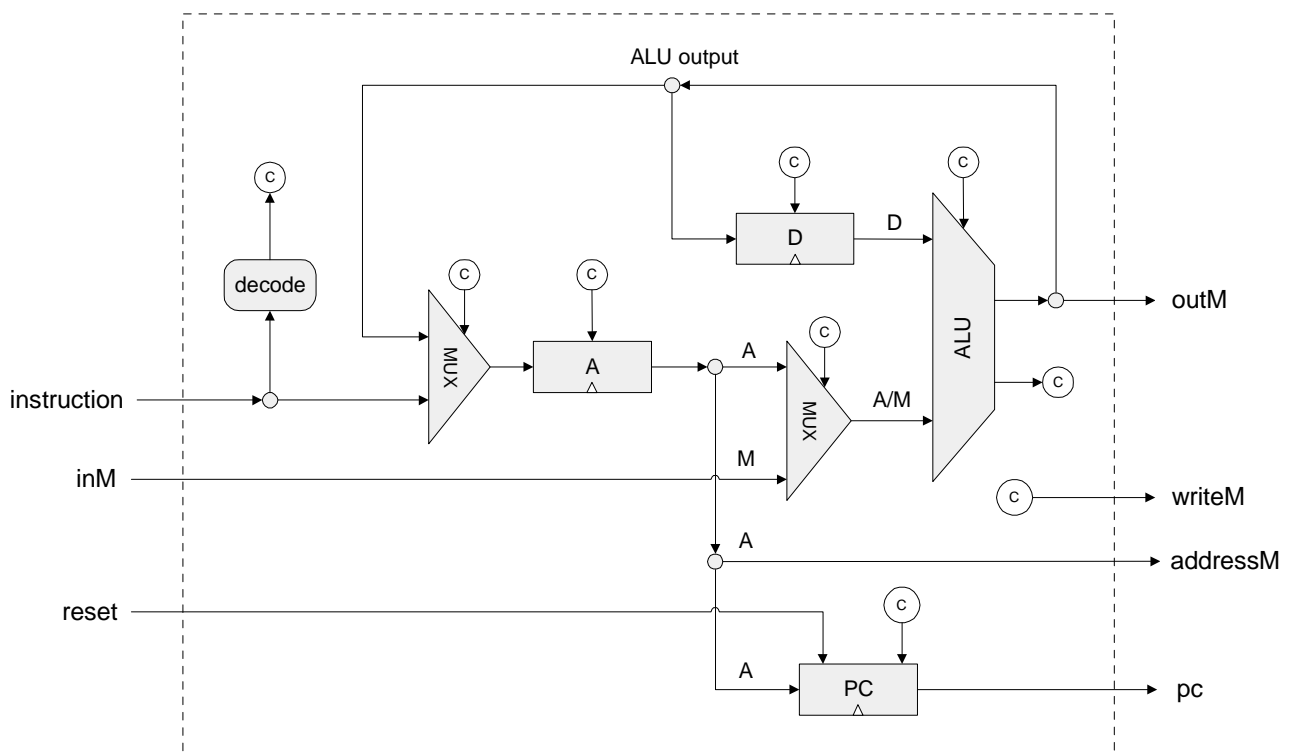


DIAGRAM 9: Proposed CPU Implementation. The diagram shows only *data* and *address paths*, i.e. wires that carry data and addresses from one place to another. The diagram does not show the CPU's *control logic*, except for inputs and outputs of control bits, labeled ©. Thus it should be viewed as an incomplete chip diagram.

The key element missing in diagram 9 is the CPU's *control logic*. The control logic is a rather simple set of gates and wires designed to perform three tasks:

- **Instruction decoding:** Figure out what the instruction means (a function of the instruction);
- **Instruction execution:** Signal the various parts of the computer what they should do in order to execute the instruction (a function of the instruction);
- **Next Instruction fetching:** Figure out which instruction to execute next (a function of the instruction and the ALU output).

(in what follows, the term "*proposed CPU implementation*" refers to diagram 9).

Instruction decoding: The 16-bit word located in the CPU's instruction input can represent either an *A*-instruction or a *C*-instruction. In order to figure out what this 16-bit word means, it can be broken into the fields "i xx a ccccc ddd jjj". The *i*-bit codes the instruction type, which is "0" for an *A*-instruction and "1" for a *C*-instruction. In case of a *C*-instruction, the *a*-bit and the *c*-bits represent the *comp* part, the *d*-bits represent the *dest* part, and the *j*-bits represent the *jump* part of the instruction. In case of an *A*-instruction, the 15 bits other than the *i*-bit should be interpreted as a 15-bit constant.

Instruction execution: The various fields of the instruction (*i*-, *a*-, *c*-, *d*-, and *j*-bits) are routed simultaneously to various parts of the architecture, where they cause different chips to do what they are supposed to do in order to execute either the *A*-instruction or the *C*-instruction, as mandated by the machine language specification. In particular, the *a*-bit determines whether the ALU will operate on the *A* register or on the *Memory*, the *c*-bits determine which function the ALU will compute, and the *d*-bits enable various locations to accept the ALU result.

Next instruction fetching: As a side effect of executing the current instruction, the CPU also determines the address of the next instruction and emits it via its *pc* output. The "seat of control" of this task is the *program counter* -- an internal part of the CPU whose output is fed directly to the CPU's *pc* output. This is precisely the *PC* chip built in chapter 3 (see figure 3-5).

Most of the time, the programmer wants the computer to fetch and execute the next instruction in the program. Thus if t is the current time-unit, the default program counter operation should be $PC(t) = PC(t-1) + 1$. When we want to effect a "*goto n*" operation, the machine language specification requires to first set the *A* register to n (via an *A*-instruction) and then issue a jump directive (coded by the *j*-bits of a subsequent *C*-instruction). Hence, our challenge is to come up with a hardware implementation of the following logic:

```
if jump(t) then PC(t) = A(t-1)
else PC(t) = PC(t-1) + 1
```

Conveniently, and actually by careful design, this jump control logic can be easily effected by the proposed CPU implementation. Recall that the *PC* chip interface (figure 3-5) has a "load" control bit that enables it to accept a new input value. Thus, to effect the desired jump control logic, we start by connecting the output of the *A* register to the input of the *PC*. The only

remaining question is when to enable the PC to accept this value (rather than continuing its steadfast counting), i.e. when does a jump need to occur. This is a function of two signals: (a) the j -bits of the current instruction, specifying on which condition we are supposed to jump, and (b) the ALU output status bits, indicating whether the condition is satisfied. Taken together, the j -bits and the ALU output status determine whether a jump needs to occur. If we have a jump, the PC must be loaded with A's output. If we don't have a jump, the PC should increment by 1.

Additionally, if we want the computer to re-start the program's execution, all we have to do is reset the program counter to 0. That's why the proposed CPU implementation feeds the CPU's reset input directly into the reset input of the PC chip.

3.2 Memory

According to its specification, the Memory chip of the Hack platform is essentially a package of three lower-level chips: RAM16K, Screen, and Keyboard. At the same time, users of the Memory chip must see a single logical address space, spanning from location 0 to 24576 (0x0000 to 0x6000 - see diagram 7). The implementation of the Memory chip should create this continuum effect. This can be done by the same technique used to combine small RAM units into larger RAM units, as we have done in Chapter 3 (see figure 3-6 and the discussion of *n-registers memory*).

3.3 Computer

Once the CPU and the Memory chips have been implemented and tested, the construction of the overall computer is straightforward. Diagram 10 depicts a possible implementation.

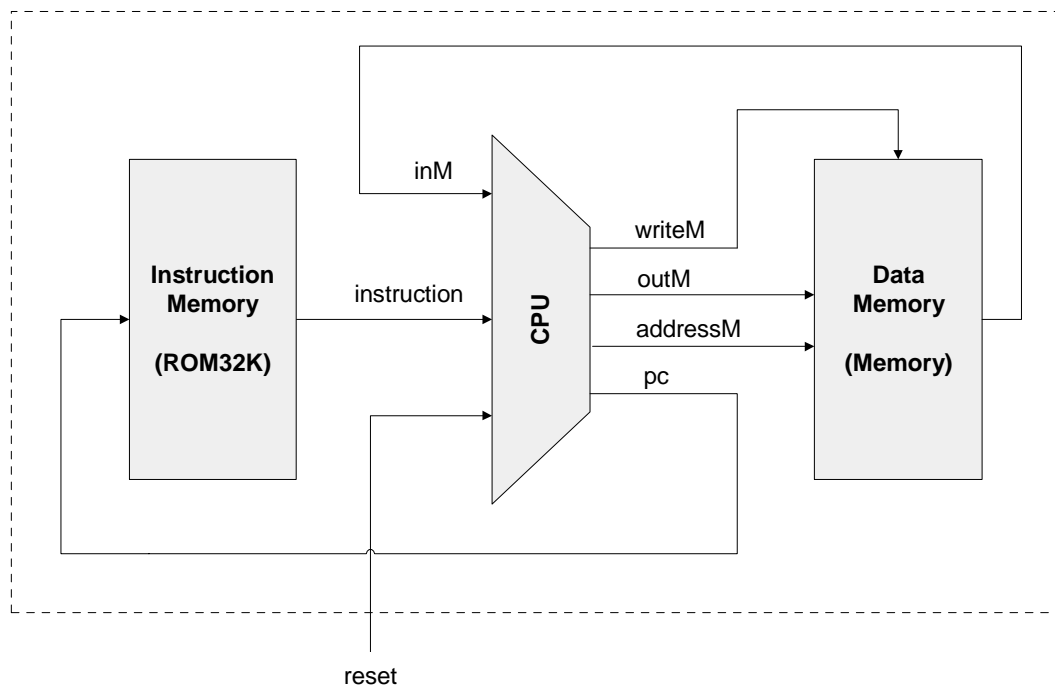


DIAGRAM 10: Proposed implementation of the top-most Computer chip.

4. Perspective

Following the general spirit of the book, the architecture of the Hack computer is rather simple and minimal. Typical computer platforms have more registers, more data types (rather than 16-bit integers only), more powerful ALU's, and more elaborate instruction sets. However, these differences are mainly quantitative. From a qualitative standpoint, Hack is quite similar to most digital computers, as they all follow the same design paradigm: the von Neumann architecture.

In terms of function, computer systems can be classified into two categories: *general-purpose computers*, designed to easily switch from executing one program to another, and *dedicated computers*, which are typically embedded in other systems like cell-phones, game-consoles, appliances, various automobile and airline control systems, factory equipment, and so on. General-purpose computers typically store data and instructions in the RAM, and are able to load programs dynamically into memory. In contrast, dedicated computers typically store software in a ROM unit: for any particular application, a single program is burned into the ROM, and is the only one executed by the embedded computer (for example, in game consoles the game software resides in an external cartridge which is simply a ROM chip encased in some fancy package). In that regard, Hack is similar to a dedicated computer. It should be noted however that general purpose and dedicated computers share the same architectural ideas: stored programs, fetch-decode-execute logic, CPU, registers, program counter, and so on.

In a similar fashion, Hack's I/O devices are as simple as possible (but not simpler). In principle, a computer can be connected to many I/O devices: printers, hard disks, digital cameras, network connections, etc. Also, typical screens are obviously much more powerful than the Hack screen, featuring more pixels, many brightness levels in each pixel, and colors. Still, the basic principle that each pixel is controlled by a memory-resident binary value is maintained: instead of a single bit controlling the pixel's black/white color, a number of bits is typically devoted to control the level of brightness of each of the three primary colors that, together, effect the pixel's ultimate color.

Likewise, the memory mapping of the Hack screen is very simplistic. Instead of mapping pixels directly into bits of memory, most modern computers employ more indirect memory maps. In particular, they allow the CPU to send higher-level graphic instructions to a graphics card that controls the screen.

Finally, it should be stressed that most of the effort and creativity in designing computer hardware is aimed at achieving better performance. Thus, hardware architecture courses evolve around such issues as implementing memory hierarchies (cache), better access to I/O devices, pipelines, parallelism, instruction pre-fetching, and other optimization techniques. Historically, the attempts to enhance the processor's performance have led to two main schools of hardware design. Advocates of the CISC (*Complex Instruction Set Computer*) approach argue for achieving better performance by providing as rich and powerful instruction sets as possible. At the same time, the RISC (*Reduced Instruction Set Computer*) camp uses simpler instruction sets in order to promote as fast a hardware implementation as possible. The Hack computer does not enter this debate, featuring neither a strong instruction set nor special hardware acceleration techniques.

5. Build It

Objective: Build the Hack computer platform, culminating in the top-most `Computer` chip. The only building blocks that you can use are the chips described in this chapter and in previous chapters, and chips that you may build on top of them.

Resources: The tools that you need for completing this project are the hardware simulator supplied with the book and the test scripts described below. The computer platform should be implemented in the HDL language specified in appendix A.

Contract: The computer platform that you build should be capable of executing programs written in the Hack machine language specified in Chapter 4. Demonstrate this capability by having your `Computer` chip run the three programs given below.

Testing: As was just said, a natural way to test your overall `Computer` chip implementation is to have it execute some sample programs written in the Hack language. In order to run such a test, one can write a test script that loads the `Computer` chip into the simulator, loads a program from an external text file into its `ROM32K` chip, and then runs the clock enough cycles to execute the program. We supply all the files necessary to run three such tests, as follows:

- **Add.hack:** this program adds the two constants 2 and 3 and writes the result in `RAM[0]`. Test scripts: `ComputerAdd.tst`, `ComputerAdd.cmp`.
- **Max.hack:** this program computes the maximum of `RAM[0]` and `RAM[1]` and writes the result in `RAM[2]`. Test scripts: `ComputerMax.tst`, `ComputerMax.cmp`.
- **Rect.hack:** this program draws a rectangle of width 16 pixels and length `RAM[0]` at the top left of the screen. Test scripts: `ComputerRect.tst`, `ComputerRect.cmp`.

Before testing your `Computer` chip on the above programs, read the relevant `.tst` file and be sure that you understand the instructions given to the simulator. Section 8 of Appendix B may be a useful reference here.

Tips: In addition to all the files necessary to test the three programs mentioned above, we supply test scripts and compare files for testing the `Memory` and `CPU` chips. It's important to complete the testing of these chips before you set out to build and test the overall `Computer` chip.

Build the computer in the following order:

- **Memory:** Composed from three chips: `RAM16K`, `Screen`, and `Keyboard`. The `Screen` and the `Keyboard` are available as built-in chips and there is no need to build them. The `RAM16K` chip was built in chapter 3. We recommend using its built-in version, as it provides a debugging-friendly GUI.
- **CPU:** Can be composed according to the proposed implementation given in diagram 9, using the `ALU` and register chips built in Chapters 2 and 3, respectively. We recommend using the built-in versions of these chips, in particular `ARegister` and `DRegister`. These chips have exactly the same functionality of the `Register` chip specified in chapter 3, plus GUI side effects.

In the course of implementing the CPU, it is allowed to specify and build some internal chips of your own. This is up to you. If you choose to create new chips not mentioned in the book, be sure to document and test them carefully before you plug them into the architecture.

- **Instruction Memory:** Use the built-in ROM32K chip.
- **Computer:** The top-most Computer chip can be composed from the chips mentioned above, using diagram 10 as a blueprint.

Steps:

1. Create a directory called `project5` on your computer;
2. Download the `project5.zip` file and extract it to your `project5` directory;
3. Build and test the chips in the order mentioned above.

6. The Assembler¹

1. Introduction

Work in progress.

2. Hack Assembly-to-Binary Translation Specification

This section gives a complete specification of the translation between the symbolic Hack assembly language to its equivalent binary representation. Unlike the language description given in chapter 4, this specification is both compact and formal. Therefore, it can be viewed as the contract that Hack assemblers must implement, in one way or another.

2.1 Syntax Conventions and Files Format

File names: By convention, programs in binary machine code and in assembly code are stored in text files with “hack” and “asm” extensions, respectively. Thus, a `Prog.asm` file is translated into a `Prog.hack` file.

Binary code (.hack) files: A binary code file is composed of text lines. Each line is a sequence of 16 “0” and “1” ASCII characters, coding a single 16-bit machine language instruction. Taken together, all the lines in the file represent a machine language program. When a machine language program is loaded into the computer’s instruction memory, the binary code represented by the file’s n -th line is stored in address n of the instruction memory (the count of both program lines and memory addresses starts at 0).

Assembly language (.asm) files: An assembly language file is composed of text lines, each representing either an *instruction* or a *symbol declaration*:

- **Instruction:** an A -instruction or a C -instruction, described below.
- **(Symbol):** This pseudo-command binds the `Symbol` to the memory location into which the next command in the program will be stored. It is called “pseudo-command” since it generates no machine code.

Constants and symbols in assembly programs: *Constants* must be non-negative and are always written in decimal notation. A user-defined *symbol* can be any sequence of letters, digits, underscore (“_”), dot (“.”), dollar sign (“\$”), and colon (“:”) that does not begin with a digit.

Comments in assembly programs: text beginning with two slashes (“//”) and ending at the end of the line is considered a comment and is ignored.

White space in assembly programs: space characters are ignored. Empty lines are ignored.

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

dest	d1	d2	d3	jump	j1	j2	j3
null	0	0	0	null	0	0	0
M	0	0	1	JGT	0	0	1
D	0	1	0	JEQ	0	1	0
MD	0	1	1	JGE	0	1	1
A	1	0	0	JLT	1	0	0
AM	1	0	1	JNE	1	0	1
AD	1	1	0	JLE	1	1	0
AMD	1	1	1	JMP	1	1	1

2.3 Symbols

Hack assembly commands can refer to memory locations (addresses) using either constants or symbols. Symbols can be introduced into assembly programs in three ways:

Predefined symbols: Any Hack assembly program is allowed to use the following pre-defined symbols:

<i>Label</i>	<i>RAM address</i>	<i>(hexa)</i>
SP	0	0x0000
LCL	1	0x0001
ARG	2	0x0002
THIS	3	0x0003
THAT	4	0x0004
R0-R15	0-15	0x0000-f
SCREEN	16384	0x4000
KBD	24576	0x6000

Note that each one of the 5 top RAM locations can be referred to using two pre-defined symbols. For example, either R2 or ARG can be used to refer to RAM[2].

Label symbols: The pseudo-command “(Xxx)” defines the symbol Xxx to refer to the instruction memory location holding the next command in the program. A label can be defined only once and can be used anywhere in the assembly program, even before the line in which it is defined.

Variable symbols: Any user-defined symbol Xxx appearing in an assembly program that is not predefined and is not defined elsewhere using the “(Xxx)” command is treated as a variable. Variables are mapped to consecutive memory locations as they are first encountered, starting at RAM address 16 (0x0010).

2.4 Example

In chapter 4 we presented a program that sums up the numbers between 1 and 100. Program 1 repeats this example, showing both its assembly and binary version.

Assembly code

```

// sum the numbers 1...100
  @i      // i=1 (allocated at 0x0010)
  M=1
  @sum    // sum=0 (allocated at 0x0011)
  M=0
(loop)
  @i      // if i-100>0 then goto end
  D=M
  @100
  D=D-A
  @end
  D;jgt
  @i      // sum+=i
  D=M
  @sum
  M=D+M
  @i      // i++
  M=M+1
  @loop   // goto loop
  0;jmp
(end)

```

Binary code

```

(this line should be erased)
0000 0000 0001 0000
1110 1111 1100 1000
0000 0000 0001 0001
1110 1010 1000 1000
(this line should be erased)
0000 0000 0001 0000
1111 1100 0001 0000
0000 0000 0110 0100
1110 0100 1101 0000
0000 0000 0001 0010
1110 0011 0000 0001
0000 0000 0001 0000
1111 1100 0001 0000
0000 0000 0001 0001
1111 0000 1000 1000
0000 0000 0001 0000
1111 1101 1100 1000
0000 0000 0000 0100
1110 1010 1000 0111
(this line should be erased)

```

PROGRAM 1: Assembly and binary representations of the same program. If the assembler is given the text file on the left, it should generate the text file given on the right.

3. Implementation

The previous section gave a complete specification of the Hack language, in both its assembly and binary versions. The program that translates assembly programs into binary programs according to this contract is called the *Hack assembler*. This section describes a proposed design for this assembler.

The assembler reads as input a text file named `Prog.asm`, containing an assembly program, and produces as output a text file named `Prog.hack`, containing the translated machine code. The name of the input file is supplied to the assembler as a command line argument:

```
prompt> Assembler Prog
```

The translation of each individual assembly command to its equivalent binary instruction is direct and one-to-one. Each command is translated separately. In particular, each mnemonic component (field) of the command is translated into its corresponding bit-code according to the tables in section 2.2, and each symbol in the command is resolved to its numeric address as explained in section 2.3.

We propose an assembler implementation based on four modules: a `Parser` module that parses the input, a `Code` module that provides the binary codes for different mnemonics, a `SymbolTable` module that handles symbols, and a main program that drives the entire translation process.

3.1 The Parser

The main function of the parser is to break each assembly command into its underlying components (fields and symbols). The API is as follows.

Parser Module			
Encapsulates access to the input code. Reads an assembly language command, parses it, and provides convenient access to the command's components (fields and symbols). In addition, removes all white space and comments.			
Routine	Arguments	Returns	Function
Constructor (initializer)	Input file (stream)	--	Opens the input file/stream and gets ready to parse it.
hasMoreCommands	--	boolean	Are there more assembly language commands in the input?
advance	--	--	Reads the next command from the input and makes it the current command. Should be called only if hasMoreCommands () is true. Initially there is no current command
commandType	--	Enumeration: <ul style="list-style-type: none"> • A_COMMAND • C_COMMAND • L_COMMAND 	Returns the type of the current command: <ul style="list-style-type: none"> • C_COMMAND for dest=comp; jump • A_COMMAND for @Xxx where Xxx is either a symbol or a decimal number • L_COMMAND (actually, pseudo-command) for (Xxx) where Xxx is a symbol.
symbol	--	string	Returns the symbol or decimal Xxx of the current command @Xxx or (Xxx) . Should be called only when commandType () is A_COMMAND or L_COMMAND .
dest	--	string	Returns the dest mnemonic in the current C-command. The 8 possible mnemonics are given in section 2.2. Should be called only when commandType () is C_COMMAND .
comp	--	string	Returns the comp mnemonic in the current C-command. The 28 possible mnemonics are given in section 2.2. Should be called only when commandType () is C_COMMAND .
jump	--	string	Returns the jump mnemonic in the current C-command. The 8 possible mnemonics are given in section 2.2. Should be called only when commandType () is C_COMMAND .

3.2 The Code Module

The `Code` module translates Hack mnemonics into their respective binary codes. The details are given in the following API.

Code Module			
Translates Hack assembly language mnemonics into binary codes.			
Routine	Arguments	Returns	Function
<code>dest</code>	string mnemonic	3 bits	Returns the 3-bit binary code of the <code>dest</code> mnemonic, as listed in section 2.2.
<code>comp</code>	string mnemonic	7 bits	Returns the 7-bit binary code of the <code>comp</code> mnemonic, as listed in section 2.2.
<code>jump</code>	string mnemonic	3 bits	Returns the 3-bit binary code of the <code>jump</code> mnemonic, as listed in section 2.2.

3.3 Assembler for programs with no symbols

We suggest building the rest of the assembler in two stages. In the first stage, write an assembler that translates assembly programs without symbols. This can be done using the Parser and Code modules just described. In the second stage, extend the assembler with symbol handling capabilities, as we explain in the next section.

The contract for the first symbol-less stage is that the input `Prog.asm` program contains no symbols. This means that (a) in all address commands of type “@Xxx” the `Xxx` constants are decimal numbers and not symbols, and (b) the file contains no label commands, i.e. no commands of type “(Xxx)”.

The overall symbol-less assembler program can now be implemented as follows. First, the program opens an output file named `Prog.hack`. Next, the program marches through the lines (assembly instructions) in the supplied `Prog.asm` file. For each *C*-instruction, the program concatenates the translated binary codes of the instruction fields into a single 16-bit word. Next, the program writes this word into the `Prog.hack` file. For each *A*-instruction of type @Xxx, the program translates the decimal constant returned by the parser into its binary representation and writes the resulting 16-bit word it into the `Prog.hack` file.

3.3 The SymbolTable Module

Since Hack instructions are allowed to use symbols, the symbols must be resolved as part of the translation process. The assembler deals with this task using a *symbol table*, designed to create and maintain the correspondence between symbols and their meaning.

The symbol table is a data structure that contains pairs of symbols and their corresponding semantics, which in our case are RAM and ROM addresses. In general, the most appropriate data

structure for representing such a relationship is the classical *hash table*. In many programming environments, such a data structure is available as part of a standard library, and thus there is no need to develop it from scratch. We propose the following API.

SymbolTable Module			
A symbol table that keeps a correspondence between symbolic labels and numeric addresses.			
Routine	Arguments	Returns	Function
Constructor	--	--	Creates a new empty symbol table
addEntry	string symbol, int address	--	Adds the pair (symbol, address) to the table.
contains	string symbol	boolean	Does the symbol table contain the given symbol?
addressOf	string symbol	int	Returns the address associated with the symbol.

3.5 Assembler for programs with symbols

Once we have a symbol table in place, the handling of symbols in the translation process is straightforward. Whenever we encounter a new symbol in the program, we allocate a numeric address to it, and add the pair (*symbol*, *address*) to the table. To translate an instruction that includes a symbol into binary code, we simply look-up the symbol in the symbol table, retrieve its numeric address, and plant it in the translated instruction. This symbol handling capability is all we need to complete the assembler's implementation

There's one complication though: in assembly programs, label symbols (used in *goto* commands) are often used before they are defined. One common solution is to write a 2-pass assembler that reads the code twice, from start to end. In the first pass, the symbol table is built and no code is generated. In the second pass, all the label symbols encountered in the program have already been bound to memory locations. Thus, in the second pass the assembler can replace them with their corresponding meanings (numbers) in order to generate the final binary code.

Recall that there are three types of symbols in the Hack language: *pre-defined symbols*, *labels*, and *variables*. The symbol table should contain and handle all these symbols, as follows:

Initialization: Initialize the symbol table with all the pre-defined symbols and their pre-allocated RAM addresses, according to Section 2.2.

First pass: Go through the entire assembly program, line by line, and build the symbol table without generating any code. As you march through the program lines, keep a running number anticipating the ROM address that will eventually be allocated to the current command. This number starts at 0 and is incremented by 1 whenever a *C*-instruction or an *A*-instruction is encountered, but does not change when a label pseudo-command or a comment is encountered. Each time a pseudo command “(xxx)” is encountered, add a new entry to the symbol table, associating xxx with the ROM address that will eventually store the next command in the program.

This pass results in entering all the program's *labels* along with their ROM addresses into the symbol table. The program's variables are handled in the second pass.

Second pass: Now go again through the entire program, and parse each line. Each time a symbolic *A*-instruction is encountered, i.e. "@xxx" where xxx is a symbol and not a number, look up xxx in the symbol table. If the symbol is found in the table, replace it with its numeric meaning and complete the command's translation. If the symbol is not found in the table, then it means that it represents a new variable. Hence, allocate the next available RAM address to it, say *n*, add the pair (xxx, *n*) to the symbol table, and complete the command's translation. The allocated RAM addresses are running, starting at address 16 (just after the addresses allocated to the pre-defined symbols).

This completes the assembler's implementation.

4. Perspective

Work in Progress.

5. Build it

Objective: To develop an assembler that translates programs written in Hack assembly language into the binary code understood by the Hack hardware platform. The assembler must implement the *Translation Specification* described in Section 2.

Resources: The only tool needed for completing this project is the programming language in which you will implement your assembler. You may also find the following two tools useful: the *assembler* and *CPU Emulator* supplied with the book. These tools allow you to experiment with a working assembler before you set out to build one yourself. In addition, the supplied assembler provides a visual line-by-line translation GUI, and allows online code comparisons with the outputs that your assembler will generate. For more information about these capabilities, refer to the supplied *Assembler Tutorial*.

Contract: When loaded into your assembler, a `Prog.asm` file containing a Hack assembly language program should be translated into the correct Hack binary code and stored in a `Prog.hack` file. The output produced by your assembler must be identical to the output produced by the assembler supplied with the book.

Testing: We suggest building the assembler in two stages. First write a symbol-less assembler, i.e. an assembler that can only translate programs that contain no symbols. Then extend your assembler with symbol handling capabilities. The test programs that we supply for this project come in two such versions (without and with symbols), to help you test your assembler incrementally.

Test Programs

Each test program, except the first one, comes in two versions: `ProgL.xxx` is symbols-less, and `Prog.xxx` is with symbols.

Add: This program adds the constants 2 and 3 and puts the result in `R0`.

Max: This program performs the operation `R2=max(R0,R1)`.

Rect: This program draws a rectangle at the top left corner of the screen. The rectangle is 16 pixels wide and `R0` pixels high.

Pong: A single-player Ping-Pong game. A ball bounces constantly off the screen's "walls." The player attempts to hit the ball with a bat by pressing the left and right arrow keys. For every successful hit, the player gains one point and the bat 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 *Pong* program was written in the *Jack* programming language (described in Chapter 9) and translated by the *Jack compiler* (described in Chapters 10-11) into the supplied assembly program. The resulting machine-level program is about 20,000 lines of code, which also include the Sack operating system (described in Chapter 12). Running this game in the CPU Emulator is a slow affair, so don't expect a high-powered Pong game. This slowness is actually a virtue, since it enables your eye to track the graphical behavior of the program. In future projects in the book this game will run much faster.

Steps: We recommend proceeding in the following order:

1. Download `project6.zip` and extract its contents into a directory called `project6` on your computer, without changing the directories structure embedded in the zip file.
2. Write and test your assembler program in the two stages described above. You may use the assembler supplied with the book to compare the output of your assembler to the correct output. This can be done by treating the `.hack` file generated by your assembler as the compare file used by the supplied assembler. For more information about the supplied assembler, go through the *Assembler Tutorial*.

7. The Virtual Machine I: Stack Arithmetic¹

*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 will approach this substantial task in two stages, each spanning two chapters in the book. High-level programs will be first translated into an intermediate code (Chapters 10-11), and the intermediate code will then be translated into machine language (Chapters 7-8). This two-tier translation model is a rather old idea that recently made a significant comeback following its adoption by modern languages like Java.

The basic idea is as follows: instead of running on a real platform, the intermediate code is designed to run on a *Virtual Machine* (VM) -- an abstract computer that does not exist for real. There are many reasons why this idea makes sense, one of which being *code transportability*. Since the VM may be implemented with relative ease on multiple target platforms, it allows running software on many processors and operating systems without having to modify the original source code. The VM implementation can be done in several ways, by software interpreters, by special purpose hardware, or by translating the VM programs into the machine language of the target platform.

A virtual machine can be described as a set of virtual memory segments and an associated language for manipulating them. This chapter presents a typical VM architecture, modeled after the *Java Virtual Machine* (JVM) paradigm. As usual, we focus on two perspectives. First, we will describe, illustrate, and specify the VM abstraction. Next, we will implement it over the Hack platform. Our implementation will entail writing a program called *VM Translator*, designed to translate VM code into Hack assembly code. The software suite that comes with the book illustrates yet another implementation vehicle, called *VM Emulator*. This program implements the VM by emulating it on a standard personal computer.

The VM language that we present consists of four types of commands: arithmetic, memory access, program flow, and subroutine-calling commands. We will split the implementation of this language into two parts, each covered in a separate project. In this chapter we will build a basic VM translator, capable of translating the VM's arithmetic and memory access commands into Hack code. In the next chapter we will extend the basic translator with program flow and subroutine-calling functionality. The result will be a full-scale virtual machine that will serve as the backend of the compiler that we will build in chapters 10-11.

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

The virtual machine that will emerge from this effort illustrates many important ideas in computer science. First, the notion of having one computer emulating another is a fundamental idea in the field, tracing back to Alan Turing in the 1930's. Over the years it had many practical implications, e.g. using an emulator of an old generation computer running on a new platform in order to achieve upward code compatibility. More recently, the virtual machine model became the centerpiece of two well-known mainstreams -- the Java architecture and the .NET infrastructure. These software environments are rather complex, and one way to gain an inside view of their underlying structure is to build a simple version of their VM cores, as we do here.

Another important topic embedded in this chapter is *stack processing*. The *stack* is a fundamental data structure that comes to play in many computer systems and algorithms. In particular, the VM presented in this chapter is stack-based, providing a working example of the elegance and power of this remarkably versatile data structure. As the chapter unfolds we will describe and illustrate many classical stack operations, and then implement them in our VM translator.

1. Background

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. This translation -- known as *compilation* -- is a rather complex process. Normally, a separate compiler is written specifically for any given pair of high-level language and target machine language. This leads to a proliferation of many different compilers, each depending on every detail of both its source and destination 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 program is parsed and its commands are translated into "primitive" steps -- steps that are neither "high" nor "low". In the second stage, the primitive steps are actually implemented in the machine language of the target hardware.

This decomposition is very appealing from a software engineering perspective: the first stage depends only on the specifics of the source high-level language, and the second stage only on the specifics of the target machine language. Of course, the interface between the two compilation stages -- the exact definition of the intermediate primitive steps -- must be carefully designed. In fact, this interface is sufficiently important to merit its own definition as a stand-alone language of an abstract machine. Specifically, one formulates a *virtual machine* whose instructions are the primitive steps into which high-level commands are decomposed. The compiler that was formerly a single monolithic program is now split into two separate programs. The first program, still termed *compiler*, translates the high-level code into intermediate virtual machine instructions, while the second program translates this VM code into the machine language of the target platform.

This two-stage compilation model has been used by many compiler writers, in one way or another. Some developers went as far as defining a formal virtual machine language, most notably the *p-code* generated by several Pascal compilers in the 1970s and the *bytecode* language generated by Java compilers. More recently, the approach has been adopted by Microsoft, whose .NET infrastructure is also based on an intermediate language, running on a virtual machine called CLR.

Indeed, the notion of an explicit and formal virtual machine language has several practical advantages. First, compilers for different target platforms can be obtained with relative ease by replacing only the virtual machine implementation (sometimes called the compiler’s “backend”). This, in turn, allows the VM code to become transportable across different hardware platforms, permitting a range of implementation tradeoffs between code efficiency, hardware cost, and programming effort. Second, compilers for many languages can share the same VM “backend”, allowing code re-use and language inter-operability. For example, some high-level languages are good at handling the GUI, while others excel in scientific calculations. If both languages compile into a common VM language, it is rather natural to have routines in one language call routines in the other, using an agreed-upon invocation syntax.

Another virtue of the virtual machine approach is modularity. Every improvement in the efficiency of the VM implementation is immediately inherited by all the compilers above it. Likewise, every new digital device or appliance which is equipped with a VM implementation can immediately gain access to a huge base of available software.

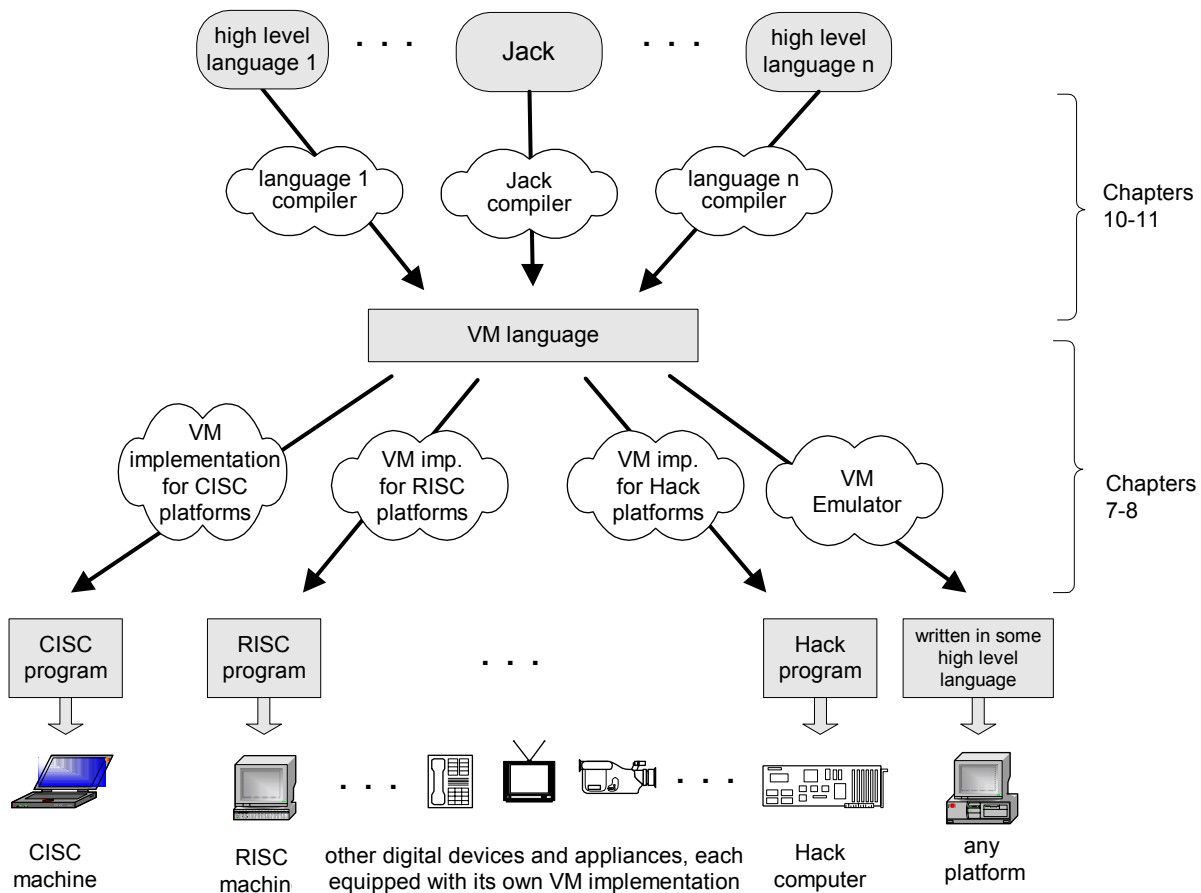


FIGURE 1: The virtual machine paradigm. Once a high-level program is compiled into VM code, the program can run on any hardware platform equipped with a suitable VM implementation. In this chapter we will start building the *VM implementation on the Hack Platform*, and use a VM emulator like the one depicted on the right. (The Jack language is introduced in chapter 9).

The Stack Machine Model

A virtual machine can be described as a set of virtual memory segments and an associated language for manipulating them. Like other languages, the VM language consists of arithmetic, memory access, program flow, and subroutine calling operations. There are several possible software paradigms on which to base such a virtual machine architecture. One of the key questions regarding this choice is *where will the operands and the results of the VM operations reside?* Perhaps the cleanest solution is to put them on a *stack* data structure.

In a *stack machine* model, arithmetic commands pop their operands from the top of the stack and push their results back onto the top of the stack. Other commands transfer data items from the stack's top to designated memory locations, and vice versa. Taken together, these simple stack operations can be used to implement the evaluation of any arithmetic or logical expression. Further, any program, written in any programming language, can be translated into an equivalent stack machine program. One such stack machine model is used in the *Java Virtual Machine* as well as in the VM described and built in this chapter.

Elementary Stack Operations: 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; the element just “below” it moves up to the top position. Thus the stack implements a *last-in-first-out* (LIFO) storage model. This basic anatomy is illustrated in Figure 2.

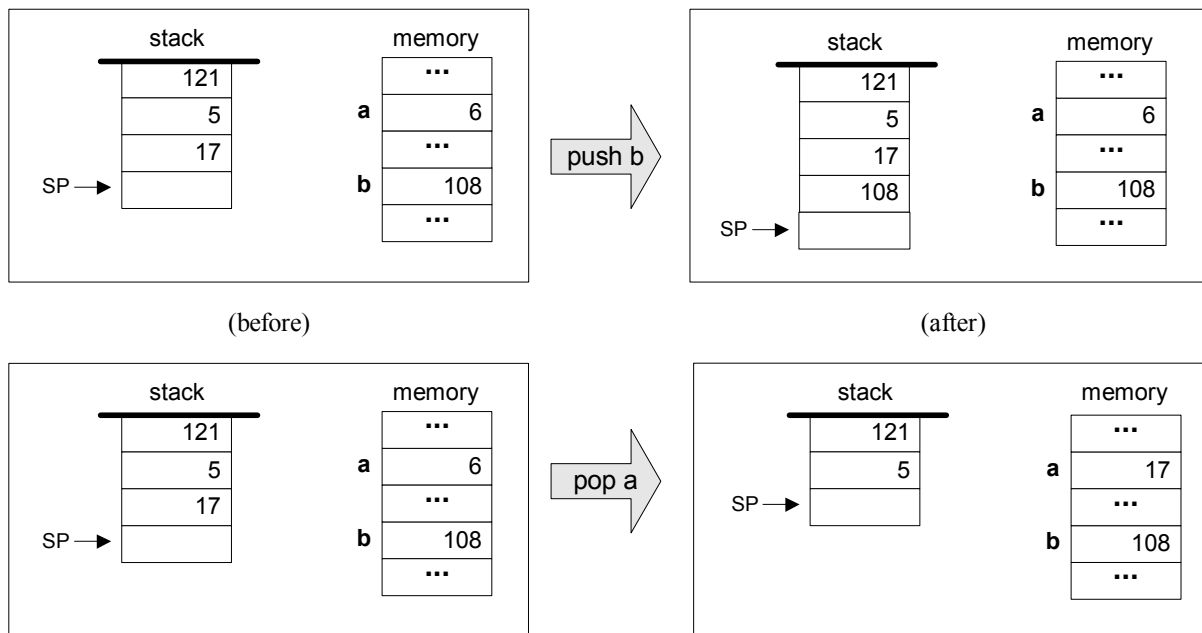


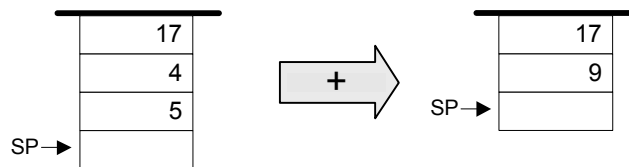
FIGURE 2: Stack processing example, illustrating the two elementary operations *push* and *pop*. Following convention, the stack is drawn upside down, as if it grows downward. The location just after the top position is always referred to by a special pointer called *sp*, or *stack pointer*. The labels *a* and *b* refer to two arbitrary memory addresses.

We see that stack access differs from conventional memory access in several respects. First, the stack is accessible only from the top, one item at a time. Second, reading the stack is a lossy operation: the only way to retrieve the top value is to *remove* it from the stack. In contrast, the act of reading a value from a regular memory location has no impact on the memory's state. Finally, writing an item onto the stack adds it to the stack's top, without changing the rest of the stack. In contrast, writing an item into a regular memory location is a lossy operation, since it erases the location's previous value.

The stack data structure can be implemented in several different ways. The simplest approach is to keep an array, say *stack*, and a *stack pointer* variable, say *sp*, that points to the available location just above the "topmost" element. The *push x* command is then implemented by storing *x* at the array entry pointed by *sp* and then incrementing *sp* (i.e. `stack[sp]=x; sp=sp+1`). The *pop* operation is implemented by first decrementing *sp* and then returning the value stored in the top position (i.e. `sp=sp-1; return stack[sp]`).

As usual in computer science, simplicity and elegance imply power of expression. The simple stack model is an extremely useful data structure that comes to play in many computer systems and algorithms. In the virtual machine architecture it serves two main purposes. First, it is used for handling all the arithmetic and logical operations of the VM. Second, it facilitates function calls and dynamic memory allocation -- the subjects of the next chapter.

Stack Arithmetic: Stack-based arithmetic is a simple matter: the two top elements are popped from the stack, the required operation is performed on them, and the result is pushed back onto the stack. For example, here is how addition is handled:



It turns out that every arithmetic expression -- no matter how complex -- can be easily converted into, and evaluated by, a sequence of simple operations on a stack. For example, consider the expression $d = (6-4) * (8+1)$, taken from some high-level program. The stack-based evaluation of this expression is shown in Figure 3.

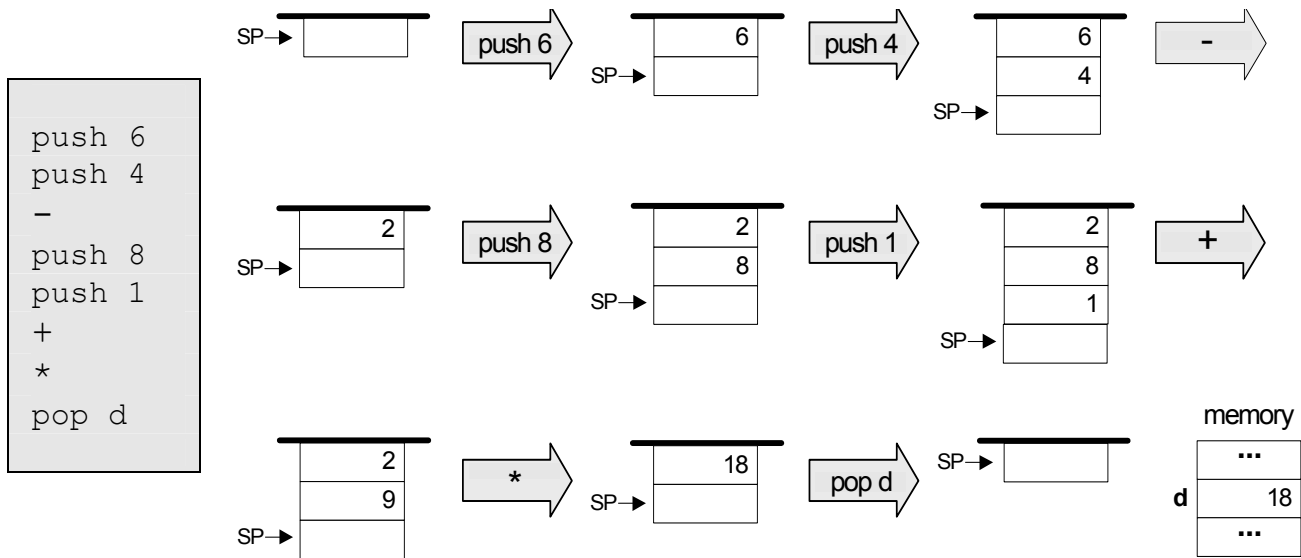


FIGURE 3: Stack-based evaluation of arithmetic expressions.

This example evaluates the expression “ $d = (6 - 4) * (8 + 1)$ ”

In a similar fashion, every logical expression can also be converted into, and evaluated by, a sequence of simple stack operations. For example, consider the high-level command “if ($x < 7$) or ($y = 8$) then ...”. The stack-based evaluation of this expression is shown in Figure 4.

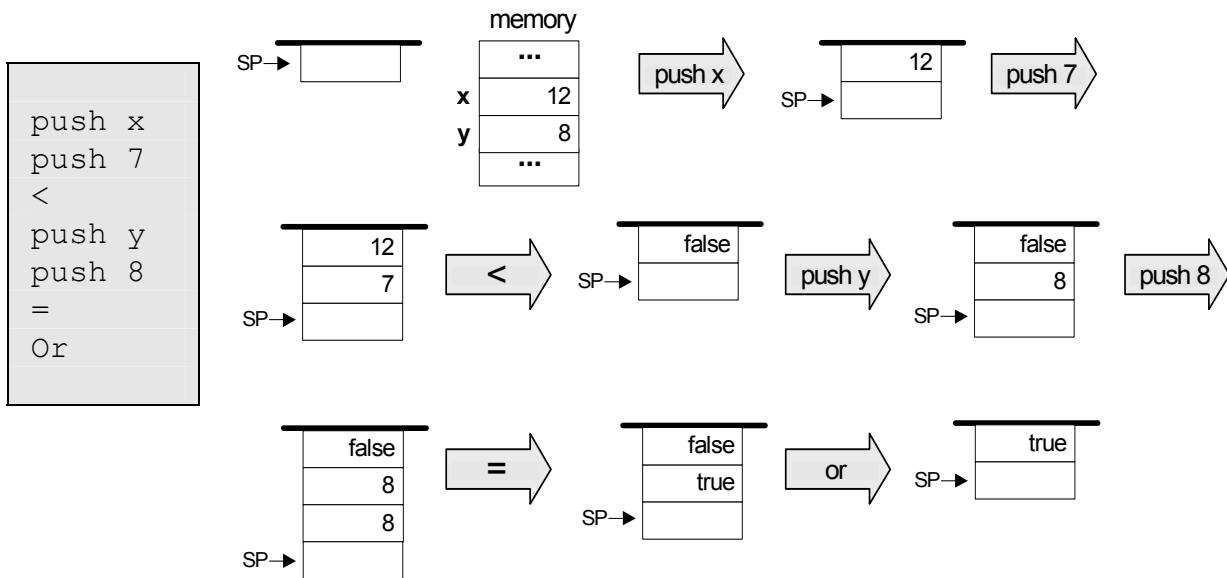


FIGURE 4: Stack-based evaluation of logical expressions.

This example evaluates the expression “if ($x < 7$) or ($y = 8$) then ...”

To sum up, the above examples illustrate a general observation: any arithmetic and Boolean expression can be transformed into a series of elementary stack operations that compute its value. Further, as we will show in Chapter 9, this transformation can be described *systematically*. Thus,

one can write a *compiler* program that translates high-level arithmetic and Boolean expressions into sequences of stack commands. Yet in this chapter we are not interested in the compilation *process*, but rather in its *results* – i.e. the VM commands that it generates. We now turn to specify these commands (section 2), illustrate them in action (section 3), and describe their implementation on the Hack platform (section 4).

2. VM Specification, Part I

2.1 General

The virtual machine is *stack-based*: all operations are done on a stack. It is also *function-based*: a complete VM program is composed of a collection of functions, written in the VM language. Each function has its own stand-alone code and is separately handled. The VM language has a single 16-bit data type that can be used as an integer, a Boolean, or a pointer. The language consists of four types of commands:

- **Arithmetic commands** perform arithmetic and logical operations on the stack;
- **Memory access commands** transfer data between the stack and virtual memory segments;
- **Program flow commands** facilitate conditional and unconditional branching operations;
- **Function calling commands** call functions and return from them.

Building a virtual machine is a complex undertaking, and so we divide it into two stages. In this chapter we specify the *arithmetic* and *memory access* commands, and build a basic VM translator that implements them only. The next chapter specifies the program flow and function calling commands, and extends the basic translator into a full-blown virtual machine implementation.

Program and command structure: A VM *program* is a collection of one or more *files* with a `.vm` extension, each consisting of one or more *functions*. From a compilation standpoint, these constructs correspond, respectively, to the notions of *program*, *class*, and *method* in an object-oriented language.

Within a `.vm` file, each VM command appears in a separate line, and in one of the following formats: `<command>`, `<command arg>`, or `<command arg1 arg2>`, where the arguments are separated from each other and from the *command* part by an arbitrary number of spaces. “//” comments can appear at the end of any line and are ignored. Blank lines are permitted.

2.2 Arithmetic and logical commands

The VM language features nine stack-oriented arithmetic and logical commands. Seven of these commands are binary: they pop two items off the stack, compute a binary function on them, and push the result back onto the stack. The remaining two commands are unary: they pop a single item off the stack, compute a unary function on it, and push the result back onto the stack. We see that each command has the net impact of replacing its operand(s) with the command's result, without affecting the rest of the stack. Table 5 gives the details.

Command	Return value (after popping the operand/s)	Comment
add	$x+y$	integer addition (2's complement)
sub	$x-y$	integer subtraction (2's complement)
neg	$-y$	arithmetic negation (2's complement)
eq	true if $x=y$ and false otherwise	equality
gt	true if $x>y$ and false otherwise	greater than
lt	true if $x<y$ and false otherwise	less than
and	x and y	bit-wise
or	x or y	bit-wise
not	not y	bit-wise

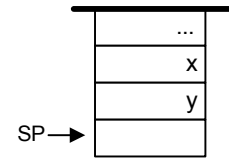


TABLE 5: Arithmetic and Logical stack commands. Throughout the table, y refers to the item at the top of the stack and x refers to the item just below it.

Three of the commands listed in Table 5 (`eq`, `gt`, `lt`) return Boolean values. The VM represents *true* and *false* as -1 (minus one, `0xFFFF`) and 0 (zero, `0x0000`), respectively.

Example: Figure 6 illustrates all the VM arithmetic commands in action. Each command is applied to an arbitrary 4-bit stack, showing the stack's state before and after the operation. We focus on the three top-most cells in the stack, noting that the rest of the stack is never affected by the current command.

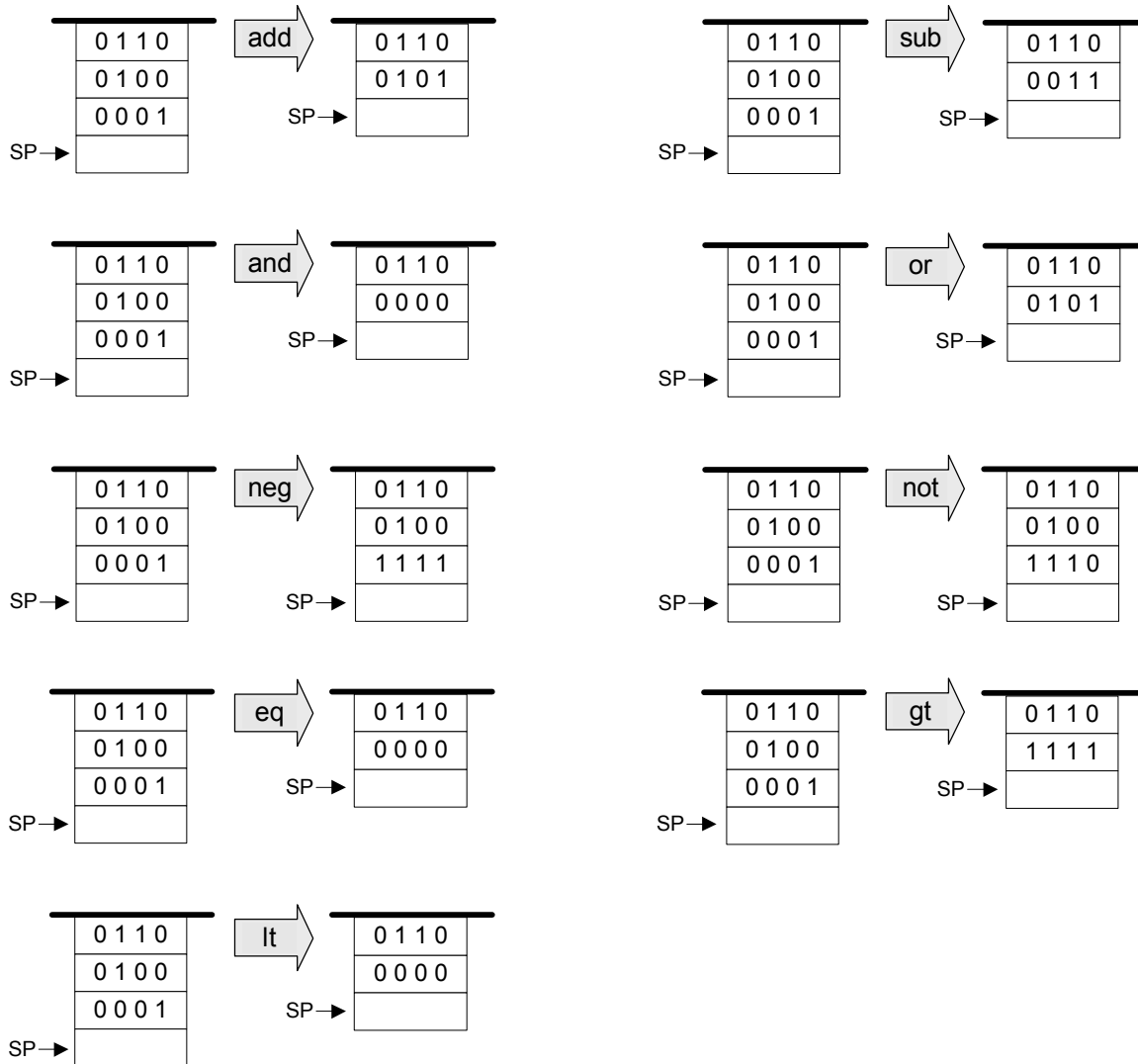


FIGURE 6: Arithmetic commands examples.

2.3 Memory Access Commands

Unlike real computer architectures, where the term “memory” refers to a collection of physical storage devices, the “memory” of a virtual machine consists of abstract devices. In particular, the VM manipulates eight *memory segments*, listed in Table 7. VM functions can access these memory segments explicitly, using VM commands. In addition, the VM manages the *stack*, but only implicitly. In other words, although the stack proper is not mentioned in VM commands, the state of the stack changes in the background, as a side effect of other commands.

Another memory element that exists in the background is the *heap*. As we elaborate later in the chapter, the heap is an area in the physical RAM where objects and arrays are stored. These objects and arrays can also be manipulated by VM commands, as we will see shortly.

Memory Segments: Each VM function sees the eight memory segments described in Table 7.

Segment	Purpose	Comments
<code>argument</code>	Stores the function's arguments.	Allocated dynamically by the VM implementation when the function is entered.
<code>local</code>	Stores the function's local variables.	Allocated dynamically by the VM implementation when the function is entered.
<code>static</code>	Stores static variables shared by all functions in the same <code>.vm</code> file.	Allocated by the VM implementation for each file; Seen by all functions in the file.
<code>constant</code>	Pseudo-segment that holds all the constants in the range 0...32767.	Emulated by the VM implementation; Seen by all the functions in the program.
<code>this</code> <code>that</code>	General-purpose segments that can be made to correspond to different areas in the heap. Serve various programming needs.	Any VM function can bind these segments to any area on the heap by setting the segment's base. The setting of the segment's base is done through the <code>pointer</code> segment.
<code>pointer</code>	Fixed 2-entry segment that holds the base addresses of <code>this</code> and <code>that</code> .	May be set by the VM program to bind <code>this</code> and <code>that</code> to various areas in the heap.
<code>temp</code>	Fixed 8-entry segment that holds temporary variables for general use.	May be used by the VM program for any purpose.

TABLE 7: The memory segments seen by every VM function.

Six of the virtual memory segments have a fixed purpose, and their mapping onto the host RAM is controlled by the VM implementation. In contrast, the `this` and `that` segments are general purpose and their mapping on the host RAM can be controlled by the current VM program: `pointer 0` controls the base of the `this` segment and `pointer 1` controls the base of the `that` segment.

Memory access commands: There are two memory access commands:

- `push segment index` push the value of `segment[index]` onto the stack;
- `pop segment index` pop the topmost stack item and store its value in `segment[index]`.

Where *segment* is one of the eight segment names and *index* is a non-negative integer.

The stack: Consider the commands sequence “push argument 2” followed by “pop local 1”. This code will end up storing the value of the function’s 3rd argument in its 2nd local variable (each segment’s index starts at 0). The working memory of these commands is the *stack*: the data value did not simply jump from one segment to another -- it went through the stack. Yet in spite of its central role in the VM architecture, the stack proper is never mentioned in the VM language. In addition, although every memory access operation involves the stack, individual stack elements cannot be accessed directly, except for the topmost element.

2.4 Program flow commands

- `label symbol // label declaration`
- `goto symbol // unconditional branching`
- `if-goto symbol // conditional branching`

These commands are discussed in the next chapter, and are listed here for completeness.

2.5 Function calling commands

- `function functionName nLocals // function declaration; must include
// the number of the function’s local variables`
- `call functionName nArgs // function invocation; must include
// the number of the function’s arguments`
- `return // transfer control back to the calling function`

Where *functionName* is a symbol and *nLocals* and *nArgs* are non-negative integers. These commands are discussed in the next chapter, and are listed here for completeness.

2.6 The big picture

We end the first part of the VM specification with a “big picture” view of the overall translation process, from a high-level program into machine code. At the top of Figure 9 we see a Jack program, consisting of two classes (Jack is a Java-like language that will be introduced in chapter 9). Each class consists of several methods. When the Jack compiler is applied to the directory in which these classes reside, it produces two VM files. In particular, each *method* in the high-level source code translates into one *function* at the VM level.

Next, the figure shows how the VM Translator can be applied to the directory in which the VM files reside, generating a single assembly program. This low-level program does two main things. First, it emulates all the virtual memory segments shown in the figure, as well as the implicit stack. Second, it effects the VM commands on the target platform. This is done by manipulating the emulated VM data structures using machine language instructions. If everything works well, i.e. if the compiler and the VM translator are implemented correctly, the target platform will end up effecting the behavior mandated by the original Jack program.

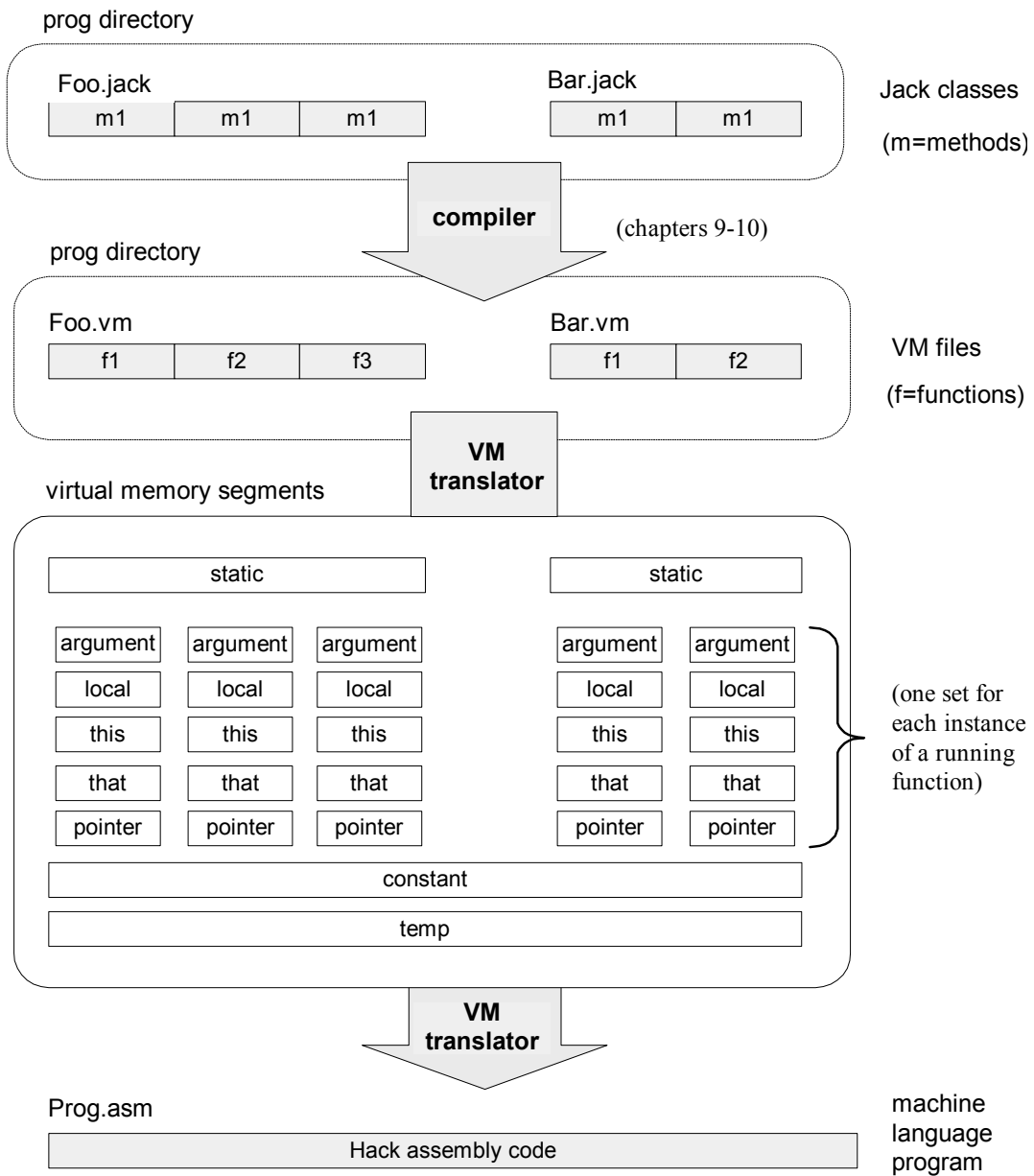


FIGURE 9: VM translation: the big picture.

3. VM Programming Examples

We now turn to illustrate the VM architecture, language, and programming style in action. We give three examples: (i) a typical arithmetic task, (ii) typical handling of object fields, and (iii) typical handling of array elements.

It's important to note at the outset that VM programs are rarely written by human programmers, but rather by compilers. Therefore, it is instructive to begin each example with a high-level version of the program, and then track down its translation it into VM code. We use a C-style syntax for all the high-level examples.

3.1 A Typical Arithmetic Task

Consider the multiplication algorithm shown at the top of Program 10. How should we (or more likely, the compiler) express this algorithm in the VM language? Well, given the primitive nature of the VM commands, we must think in terms of simple "goto logic," resulting in the "first approximation" version of Program 10. Next, we have to express this logic using a stack-oriented formalism. It is instructive to carry out this translation in two stages, beginning with a symbolic pseudo version of the VM language. Finally, we replace the symbols in the pseudo code with virtual memory locations, leading to the actual VM program. (The exact semantics of the VM commands `function`, `label`, `goto`, `if-goto`, and `return` are described in chapter 8, but their intuitive meaning is self-explanatory.)

High-Level Code (C style)

```
int mult(int x,int y) {
    int sum,j;
    sum=0;
    for(int j=y; j!=0; j--)
        sum+=x; // repetitive addition
    return sum;
}
```

First approximation

```
function mult
  args x,y
  vars sum,j
  sum=0
  j=y
loop:
  if j==0 goto end
  sum=sum+x
  j=j-1
  goto loop
end:
  return sum
```

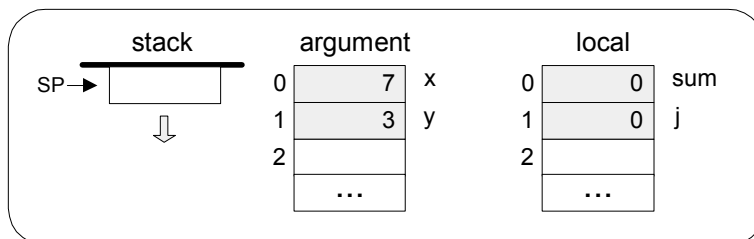
Pseudo VM code

```
function mult(x,y)
  push 0
  pop sum
  push y
  pop j
label loop
  push 0
  push j
  eq
  if-goto end
  push sum
  push x
  add
  pop sum
  push j
  push 1
  sub
  pop j
  goto loop
label end
  push sum
  return
```

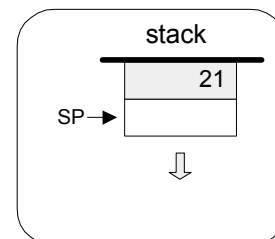
Final VM code

```
function mult 2 // 2 local variables
  push constant 0
  pop local 0 // sum=0
  push argument 1
  pop local 1 // j=y
label loop
  push constant 0
  push local 1
  eq
  if-goto end // if j==0 goto end
  push local 0
  push argument 0
  add
  pop local 0 // sum=sum+x
  push local 1
  push constant 1
  sub
  pop local 1 // j=j-1
  goto loop
label end
  push local 0
  return // return sum
```

Run-time example: Just after mult(7,3) is entered:



Just before mult(7,3) returns:



(The symbols x,y,sum,j are not part of the VM! They are shown here only for ease of reference)

PROGRAM 10: VM programming example.

We end this example with two observations. First, let us focus on the figure at the bottom of Program 10. We see that when a VM function starts running, it assumes that (i) the stack is empty, (ii) the argument values on which it is supposed to operate are located in the `argument` segment, and (iii) the local variables that it is supposed to use are initialized to 0 and located in the `local` segment. Second, let us focus on the translation from the pseudo code to the final code. Recall that VM commands are not allowed to use symbolic argument and variable names - they are limited to making `<segment index>` references only. However, the translation from the former to the latter is straightforward. All we have to do is represent `x`, `y`, `sum` and `j` as `argument 0`, `argument 1`, `local 0` and `local 1`, respectively, and replace all their symbolic occurrences in the pseudo code with corresponding `<segment index>` references.

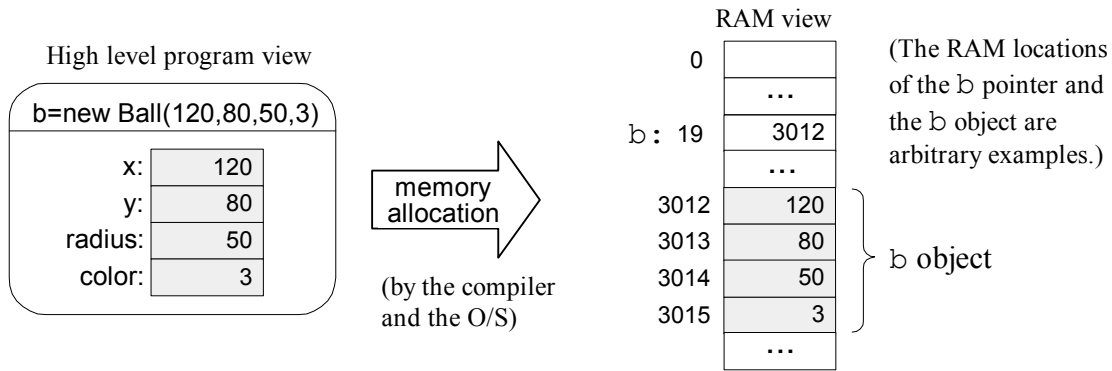
To sum up, when a VM function starts running, it assumes that it is surrounded by a private world, all of its own, consisting of initialized `argument` and `local` segments and an empty stack, waiting to be manipulated by its commands. The agent responsible for building this world for *every* VM function just before it starts running is the VM implementation, as we will see in the next chapter.

3.2 Object handling

High-level object-oriented programming languages are designed to handle complex variables called *objects*. Technically speaking, an object is a bundle of variables (also called *fields*, or *properties*), with associated code, that can be treated as one entity. For example, consider an animation program designed to juggle balls on the screen. Suppose that each `Ball` object is characterized by the integer fields `x`, `y`, `radius`, and `color`. Let us assume that the program has created one such `Ball` object, and called it `b`. What will be the internal representation of this object in the computer?

Like all other objects, it will end up on an area in the RAM called *heap*. In particular, whenever a program creates a new object using a high-level command like `b=new Ball(...)`, the compiler computes the object's size (in terms of words) and the operating system finds and allocates enough RAM space to store it in the heap. The details of memory allocation and de-allocation will be discussed later in the book. For now, let us assume that our `b` object has been allocated RAM addresses 3012 to 3015, as shown in Program 11.

Suppose now that a certain function in the high-level program, say `resize`, takes a `Ball` object and an integer `r` as arguments, and, among other things, sets the ball's `radius` to `r`. The function and its VM translation are given in Program 11.



High-level code

```

resize (Ball b,int r) {
  ...
  b.radius=r;
  ...
}
    
```

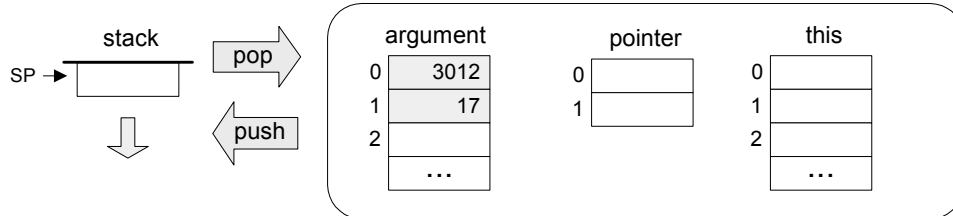
VM code

```

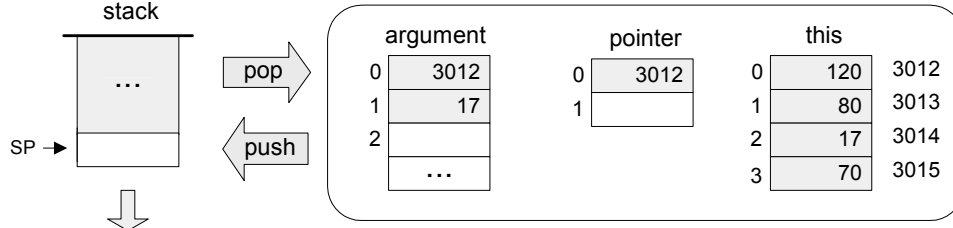
// b.radius=r
push argument 0 // get b's base address
pop pointer 0 // point the this segment to b
push argument 1 // get r's value
pop this 2 // set b's third field to r
...
    
```

Run-time simulation (example):

Just after `resize(b,17)` is entered:



Just after setting `b`'s radius to 17:



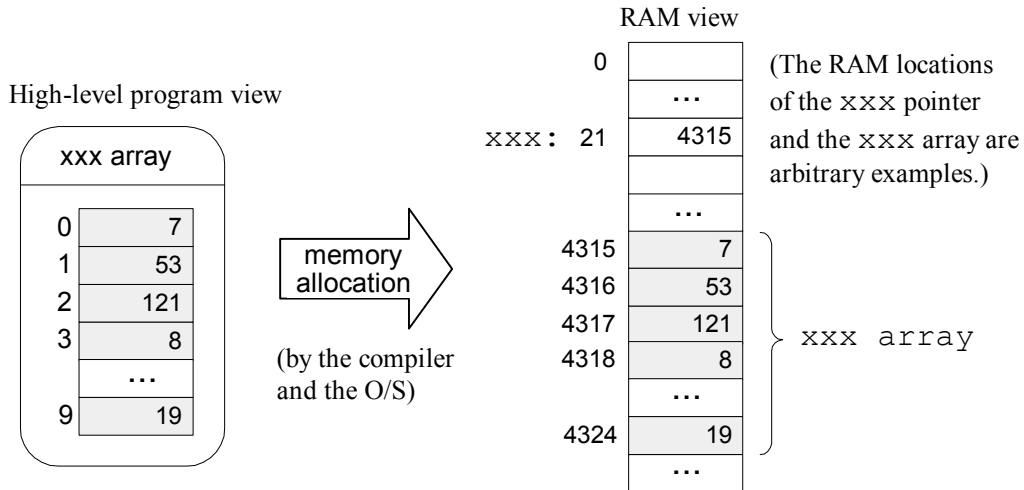
PROGRAM 11: VM-based object manipulation. (The labels at the bottom right (3012, ...) are not part of the VM state, and are given here for ease of reference.)

Note that the name of the object (which happens to be “`b`” in this example) is actually a reference to a memory cell containing the address 3012 (see Program 11). Since `b` is the first argument passed to the `resize` method, the compiler will treat it as the 0th argument of the translated VM function. Hence, when we set `pointer 0` to the value of this argument, we are effectively setting the base of the VM's `this` segment to address 3012. From this point on, VM commands can access any field in the heap-resident `b` object using the virtual memory segment `this`, without ever worrying about the physical address of the actual object.

3.3 Array Handling

An array is an indexed vector of objects of the same type. Suppose that a high-level program has created an array of 10 integers called `xxx`, and proceeded to fill it with some 10 constants. Let us assume that the array's base has been mapped on RAM address 4315 in the heap. Suppose now that a certain method in the high-level program, say `foo`, takes an array as a parameter, and, among other things, sets its k -th element to 34, where k is one of the method's local variables.

In the C language, this operation can be specified using two forms of syntax: `xxx[k]=34`, and `*(xxx+k)=34`. Whereas the former expression is more intuitive for humans, the latter provides a more accurate description of what the machine is actually doing under the surface. Specifically, the C notation “`*x`” means “*the contents of the memory location addressed by x*”. Hence, the command “`*(xxx+k)=34`” reads: “*set the RAM location whose address is (xxx+k) to 34*”. As shown in Program 12, this is precisely what the VM code is doing, using primitive VM commands.



High-level code

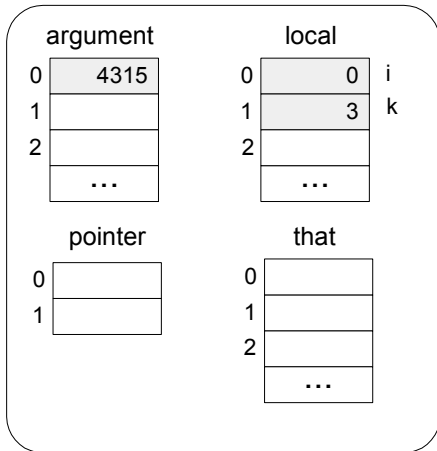
```
method foo (int[] xxx, ...) {
    int i,k;
    ...
    k=3;
    xxx[k]=34;
    ...
}
```

VM code

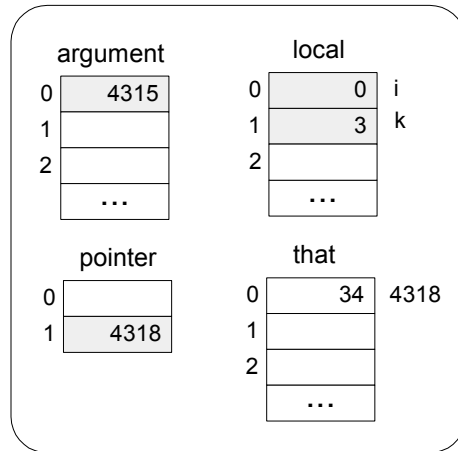
```
...
push constant 3 // set k=3
pop local 1
push argument 0 // get xxx's base address
push local 1 // get k
add // put xxx+k on the stack
pop pointer 1 // set that's base to (xxx+k)
push constant 34
pop that 0 // *(xxx+k)=34
...
```

Run-time simulation (example):

Just after the `k=3` operation:



Just after the `xxx[k]=34` operation:



PROGRAM 12: VM-based array manipulation. (The symbolic and numeric labels shown in the right are not part of the VM state, and are given here for ease of reference.)

4. Implementation

The virtual machine that was described up to this point is an abstract artifact. If we want to use it for real, we must implement it on a real platform. Building such a VM implementation consists of two conceptual tasks. First, we have to emulate the VM world on the target hardware. In particular, each data structure mentioned in the VM specification, i.e. the stack and the virtual memory segments, must be represented in some way by the hardware and low-level software of the target platform. Second, each VM command must be translated into a series of machine language instructions that effect the command on the target platform.

This section describes how to implement the VM specified in the previous section on the Hack platform specified in Chapter 4. We start by defining a “standard mapping” from VM elements and operations to the Hack hardware and machine language. Next, we suggest guidelines for designing the software that achieves this mapping. In what follows, we will refer to this software using the terms *VM implementation* or *VM translator* interchangeably.

4.1 Standard Mapping on the Hack Platform, Part I

If you re-read the virtual machine specification given so far, you will realize that it contains no assumption whatsoever about the architecture on which the machine can be implemented. When it comes to virtual machines, platform-independence is the whole point: you don’t want to commit to any one hardware platform, since you want your machine to potentially run on *all* of them, including those that were not built yet.

It follows that the VM designer can principally let programmers implement the VM on target platforms in any way they see fit. As it turns out however, it is usually recommended to provide some guidelines on how the VM should map on the target platform, rather than leaving these decisions completely to the implementer’s discretion. These guidelines, called *standard mapping*, are provided for two reasons. First, we wish the VM implementation to support interoperability with other high-level languages implemented over the target platform. Second, we wish to allow the developers of the VM implementation to run standardized tests, i.e. tests that conform to the standard mapping (this way the tests and the software can be written by different people, which is always recommended). With that in mind, the remainder of this section specifies the standard mapping of the VM on a familiar hardware platform: the Hack computer.

VM to Hack Translation

Recall that a VM program is a collection of one or more `.vm` files, each containing one or more VM functions, each being a sequence of VM commands. The VM translator takes a collection of `.vm` files as input and produces a single Hack assembly language `.asm` file as output. Each VM command is translated by the VM translator into Hack assembly code. The order of the functions within the `.vm` files does not matter.

RAM Usage

The data memory of the Hack computer consists of 32K 16-bit words. The first 16K serve as general-purpose RAM. The next 16K contain the memory maps of I/O devices. The VM implementation should use this space as follows:

<i>RAM addresses</i>	<i>Usage</i>
0–15:	16 virtual registers, whose usage is described below
16–255:	Static variables (of all the VM functions in the VM program)
256–2047:	Stack
2048–16483:	Heap (used to store objects and arrays)
16384–24575:	Memory mapped I/O

TABLE 13: Standard VM implementation on the Hack RAM.

Hack Registers: According to the *Hack Machine Language Specification*, RAM addresses 0 to 15 can be referred to by all assembly programs using the symbols `R0` to `R15`, respectively. In addition, the specification states that all assembly programs can refer to RAM addresses 0 to 4 (i.e. `R0` to `R4`) using the symbols `SP`, `LCL`, `ARG`, `THIS`, and `THAT`. This convention was introduced into the assembly language with foresight, in order to promote readable VM implementations. In other words, we anticipated that the main use of the assembly language will be to develop VM translators. With that in mind, the expected use of the Hack registers in the VM context is described in Table 14.

<i>Register</i>	<i>Name</i>	<i>Usage</i>
RAM[0]	<code>SP</code>	Stack pointer: points to the next topmost location in the stack
RAM[1]	<code>LCL</code>	Points to the base of the current VM function's <code>local</code> segment
RAM[2]	<code>ARG</code>	Points to the base of the current VM function's <code>argument</code> segment
RAM[3]	<code>THIS</code>	Points to the base of the current <code>this</code> segment (within the heap)
RAM[4]	<code>THAT</code>	Points to the base of the current <code>that</code> segment (within the heap)
RAM[5–12]	<code>TEMP</code>	Hold the contents of the <code>temp</code> segment
RAM[13–15]	(-)	Can be used by the VM implementation as general-purpose registers.

TABLE 14: Usage of the Hack registers in the standard mapping

Memory Segments Mapping

local, argument, this, that: Each one of these segments is mapped directly on the Hack RAM, and its location is maintained by keeping its physical base address in a dedicated register (`LCL`, `ARG`, `THIS`, and `THAT`, respectively). Thus any access to the i 'th location in any one of these segments should be translated to assembly code that accesses address ($base+i$) in the RAM, where $base$ is the value stored in the register dedicated to the respective segment.

pointer, temp: These segments are globally fixed and are each mapped directly onto a fixed area in the RAM. Specifically, the `pointer` segment is mapped to RAM locations 3-4 (Hack registers `THIS` and `THAT`) and the `temp` segment is mapped to RAM locations 5-12 (Hack

registers R5, R6, ..., R12). Thus access to pointer `i` should be translated to assembly code that accesses RAM location `i+3`, and access to `temp i` should be translated to assembly code that accesses RAM location `i+5`.

constant: This segment is truly virtual, as it does not occupy any physical space on the target architecture. Instead, the VM implementation handles any VM access to `<constant i>` by simply supplying the constant `i`.

static: According to the Hack machine language specification, when a new symbol is encountered for the first time in an assembly program, the assembler allocates a new RAM address to it, starting at address 16. This convention can be exploited to represent each static variable number `j` in a VM file `f` as the assembly language symbol `f.j`. For example, suppose that the file `Xxx.vm` contains the command `push static 3`. This command can be translated to the Hack assembly commands `@Xxx.3` and `D=M`, followed by additional assembly code that pushes `D`'s value to the stack. This implementation of the `static` segment is somewhat tricky, but it works.

Assembly Language Symbols

To recap, Table 15 summarizes all the assembly language symbols used by VM implementations that conform to the standard mapping.

<i>Symbol</i>	<i>Usage</i>
SP, LCL, ARG, THIS, THAT	These pre-defined symbols point to the stack top and to the base addresses of the virtual segments local, argument, this, and that.
R13-R15	Can be used for any purpose.
“ <code>f.j</code> ” symbols	Each static variable <code>j</code> in file <code>f.vm</code> is translated into the assembly symbol <code>f.j</code> . In the subsequent assembly process, these symbols will be automatically allocated RAM locations by the Hack assembler.
Flow of control symbols (labels)	The VM commands <code>function</code> , <code>call</code> , and <code>label</code> are handled by generating symbolic labels, to be described in chapter 8.

TABLE 15: Usage of Assembly symbols in the standard mapping.

4.2 Design Suggestion for the VM implementation

The VM translator should accept a single command line parameter, `Xxx`, where `Xxx` is either a file name containing a VM program (the `.vm` extension must be specified) or the name of a directory containing one or more `.vm` files (in which case there is no extension):

```
prompt> translator Xxx
```

The translator then translates the file `Xxx.vm` or, in case of a directory, all the `.vm` files in the `Xxx` directory. The result of the translation is always a single assembly language file named `Xxx.asm`,

created in the same directory as the input `xxx`. The translated code must conform to the standard VM-on-Hack mapping.

Program Structure

We propose implementing the VM translator using a main program and two modules: *parser* and *code writer*.

Parser

This module handles the parsing of a single `.vm` file. We propose the following API:

Parser Module			
Encapsulates access to the input code. Reads a VM command, parses it, and provides convenient access to its components. In addition, Removes all white space and comments.			
Routine	Arguments	Returns	Function
Constructor	Input file / stream	--	Opens the input file/stream and gets ready to parse it.
hasMoreCommands	--	boolean	Are there more commands in the input?
advance	--	--	Reads the next command from the input and makes it the current command. Should be called only if <code>hasMoreCommands()</code> is true. Initially there is no current command.
commandType	--	<code>C_ARITHMETIC</code> , <code>C_PUSH</code> , <code>C_POP</code> , <code>C_LABEL</code> , <code>C_GOTO</code> , <code>C_IF</code> , <code>C_FUNCTION</code> , <code>C_RETURN</code> , <code>C_CALL</code> (enumeration)	Returns the type of the current command. <code>C_ARITHMETIC</code> is returned for all the arithmetic VM commands.
arg1	--	string	Returns the first argument of the current command. In the case of <code>C_ARITHMETIC</code> , the command itself (“add”, “sub”, etc.) is returned. Should not be called for <code>C_RETURN</code> .
arg2	--	int	Returns the second argument of the current command. Should be called only if the current command is <code>C_PUSH</code> , <code>C_POP</code> , <code>C_FUNCTION</code> , or <code>C_CALL</code> .

Code Writer

This module is responsible for translating each VM command into Hack assembly code. We propose the following API:

CodeWriter Module			
Translates VM commands into Hack assembly code.			
Routine	Arguments	Returns	Function
Constructor	Output file / stream	--	Opens the output file/stream and gets ready to write into it.
setFileName	fileName (string)	--	Informs the code writer that the translation of a new VM file is started.
writeArithmetic	command (string)	--	Writes the assembly code that is the translation of the given arithmetic command.
WritePushPop	command (enumeration), segment (string), index (int)	--	Writes the assembly code that is the translation of the given command, where command is one of the two enumerated values: C_PUSH or C_POP.
Close	--	--	Closes the output file.
Comment: More routines will be added to this module in chapter 8.			

Main Program

The main program should construct a `Parser` to parse the VM input file and a `CodeWriter` to generate code into the corresponding output file. It should then march through the VM commands in the input file, and generate assembly code for each one of them.

If the program's argument is a directory name rather than a file name, the main program should process all the `.vm` files in this directory. In doing so, it should use a single `CodeWriter` for handling the output, but a separate `Parser` for handling each input file.

5. Perspective

In this chapter we began the process of developing a compiler for a high-level language. Following modern software engineering practices, we have chosen to base the compiler on a two-stage compilation model. In the *frontend* stage, covered in chapters 10 and 11, the high-level code is translated into an intermediate code, running on a virtual machine. In the *backend* stage, covered in this and in the next chapter, the intermediate code is translated into the machine language of a target hardware platform (see Figures 1 and 9).

The idea of formulating the intermediate code as the explicit language of a virtual machine goes back to the late 1970's, when it was used by several popular Pascal compilers. These compilers generated an intermediate “p-code” which could execute on any computer that implemented it, typically using interpreters. This idea came in the right time, since in the early 1980's the world of personal computers began to split into different processor and operating system camps. On the backdrop of this division, compilers that generated p-code provided a reasonable solution for running the same high-level program on multiple platforms, without having to re-compile it. This was the beginning of the *cross-platform compatibility* challenge, as well as the first attempt to address it using a VM approach.

Following the wide spread adoption of the world-wide-web in the mid 1990s, cross-platform compatibility became a universally vexing issue. Viewing this want as a business opportunity, Sun Microsystems sought to develop a new language that could potentially run on any computer and digital device hooked to the Internet. The language that emerged from this effort – *Java* – was based on a virtual machine model. Specifically, the *Java Virtual Machine* (JVM) is a specification that describes an intermediate language called *bytecode*. Files written in the bytecode language are used for dynamic distribution of Java programs over the internet, most notably as applets embedded in web pages. Of course in order to execute these programs, the client computers must be equipped with suitable JVM implementations. These *Java run-time environments* became widely available “plug-ins”, provided freely by Sun for practically any processor / OS combination, including game consoles and cell-phones.

Today, the JVM model and the associated bytecode language are widely used for code-mobility and interoperability over the Internet. The cornerstone of this architecture is the ubiquitous Java virtual machine, allowing Sun to market Java as a “*write once, run everywhere*” language. As a side benefit, the JVM became a means for empowering the client computer in several different ways. For example, it allows verifying the transmitted bytecodes for safety, reducing the risk of downloading malicious code.

In the early 2000's, Microsoft entered the fray with its “.NET” infrastructure. The centerpiece of .NET is a virtual machine model called CLR (*Common Language Runtime*). According to the Microsoft vision, many programming languages (including C++, C#, Visual Basic, and J# -- a Java variant) could be compiled into intermediate code running on the CLR. This enables code written in different languages to inter-operate and share the software libraries of a common run-time environment. Yet unlike the Java VM approach, which seeks to allow Java programs to execute on any possible hardware/OS platform, the CLR is designed to run only on top of operating systems provided by Microsoft.

We note in closing that a crucial ingredient that must be added to the virtual machine model before its full potential of inter-operability is unleashed is a common software library. Indeed the Java virtual machine comes with the *standard Java libraries*, and the Microsoft virtual machine comes with the *Common Language Runtime*. These software libraries can be viewed as small operating systems, providing the languages that run on top of the VM with such services as memory management, GUI utilities, string functions, math functions, and so on. One such library will be described and built in chapter 12.

6. Build it

This section describes how to build the VM translator specified in this chapter. In the next chapter we will extend this basic translator with additional functionality, leading to a full-scale VM implementation.

Objective: Develop a VM translator that implements the *stack arithmetic* and *memory access* commands described in the *VM Specification, Part I* (section 2). The VM should be implemented on the Hack computer platform, conforming to the *standard VM-on-Hack mapping* described in Section 4.1.

Resources: You will need two tools: the programming language in which you will implement your VM Translator, and the CPU Emulator supplied with the book. This emulator will allow you to execute the machine code generated by your VM Translator -- an indirect way to test the correctness of the latter. Another tool that may come handy in this project is the visual *VM Emulator* supplied with the book. This program allows to experiment with a working VM environment before you set out to implement it yourself. For more information about this tool, refer to the *VM Emulator Tutorial*.

Contract: Write a VM-to-Hack translator. Use it to translate the test `.vm` programs supplied below, yielding corresponding `.asm` 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.

6.1 Proposed Implementation Stages

We recommend implementing the translator in two stages. This modularity will allow you to test your implementation incrementally, using the step-by-step test programs that we provide.

Stage I: Stack arithmetic: The first version of your translator should implement two things: (i) the nine stack arithmetic and logical commands, and (ii) the “`push constant`” command. Note that the latter is the *push* command for the special case where the first argument is “constant”.

Stage II: Memory access commands: The next version of your translator should be a full implementation of the *push* and *pop* commands, handling all eight memory segments. We suggest the following order:

0. You have already handled the `constant` segment;
1. Next, handle the four segments `local`, `argument`, `this`, and `that`;
2. Next, handle the `pointer` and `temp` segments, in particular allowing modification of the bases of the `this` and `that` segments;
3. Finally, handle the `static` segment.

5.2 Test Programs

The supplied test programs and scripts are designed to support the incremental development plan described above.

Stage I: Stack Arithmetic programs:

- `simpleAdd`: Adds two constants;
- `stackTest`: Executes a sequence of arithmetic and logical stack operations.

Stage II: Memory Access programs:

- `basicTest`: Executes *pop* and *push* operations using various memory segments;
- `pointerTest`: Executes *pop* and *push* operations using the `pointer` and `temp` segments;
- `staticTest`: Executes *pop* and *push* operations using the `static` segment.

We supply two test scripts for each test program. One script allows running the source `.vm` test program on the VM emulator, so that you can gain familiarity with the program's intended operation. The other script allows testing the target assembly code generated by your VM translator on the CPU emulator.

5.3 The VM Emulator

A virtual machine model can be implemented in several different ways. Translating VM programs into machine language -- as we have done so far -- is one possibility. Another implementation option is simulating the VM model in software, using a high-level language. One such simulation program, shown in Figure 16, is included in the software suite that accompanies the book.

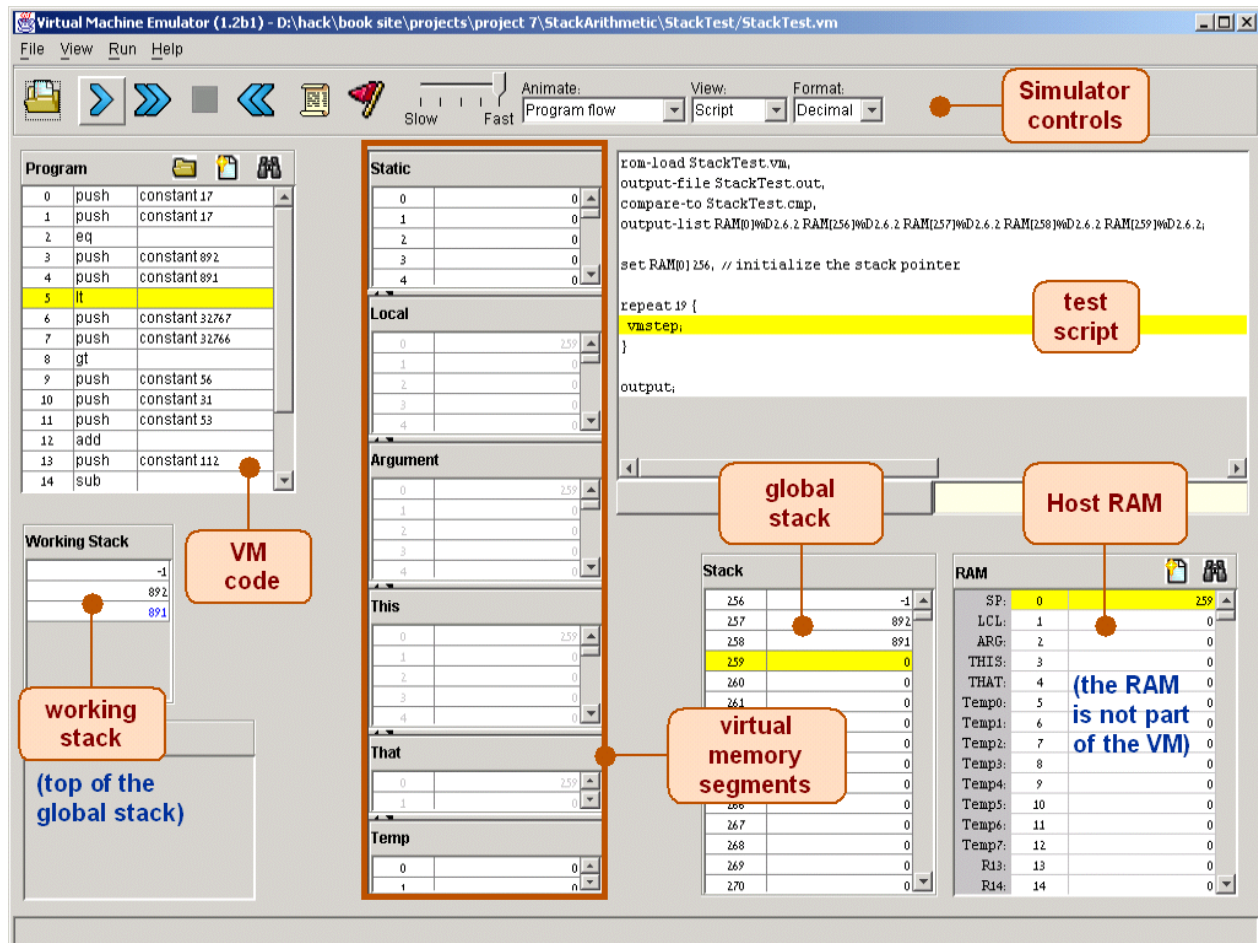


FIGURE 16: The VM emulator. This program is supplied with the book.

Our VM emulator was built with one purpose in mind: illustrating how the VM works, using visual GUI and animation effects. Specifically, it allows executing VM programs directly, without having to translate them first into machine language. This is a recommended exercise, as it enables experimentation with the VM environment before you set out to implement it yourself.

5.4 Tips

Implementation: In order for any VM program to start running, it should include a preamble startup code that forces the VM implementation to start executing it on the host platform. In addition, in order for any VM code to operate properly, the VM implementation must store the base addresses of the virtual segments in the correct locations in the host RAM. Both issues -- startup code and segments initializations -- are described and implemented in the next chapter. The difficulty of course is that we need these initializations in place in order to run the test programs given in *this* project. The good news are that you should not worry about these issues, since the supplied test scripts take care of them in a manual fashion (for the purpose of this project only).

Testing/debugging: For each one of the five test programs, follow these steps:

1. Run the supplied test `.vm` program on the VM emulator, using the VM-emulator test script, until you feel comfortable with the intended behavior of the output of this translation step.
2. Use your partial translator to translate the `.vm` program. The result should be a text file containing a translated `.asm` program, written in the Hack assembly language.
3. Inspect the translated `.asm` program. If there are syntax (or any other) errors, debug and fix your translator.
4. Use the supplied `.tst` and `.cmp` files to run your translated `.asm` program on the CPU Emulator. If there are run-time errors, debug and fix your translator.

The supplied test programs were carefully planned to test the specific features of each stage in your implementation. Therefore, it's important to implement your translator in the proposed order, and to test it using the appropriate test programs at each stage. Implementing a later stage before an early one may cause the test programs to fail.

Steps

1. Download `project7.zip` and extract its contents into a directory called `project7` on your computer, without changing the directories structure embedded in the zip file.
2. Write and test the basic VM translator in stages, as described above.

8. The Virtual Machine II: Flow Control¹

*It's like building something where you don't have to order the cement.
You can create a world of your own, your own environment,
and never leave this room.*

(Ken Thompson, 1983 Turing Award lecture)

Chapter 7 introduced the notion of a virtual machine (VM), and ended with the construction of a basic VM implementation over the Hack platform. In this chapter we continue to develop the virtual machine abstraction, language, and implementation. In particular, we focus on a variety of *stack-based* mechanisms designed to handle nested subroutine calls (procedures, methods, functions) of procedural or object-oriented languages. As the chapter progresses, we extend the previously built basic VM implementation, ending with a full-scale VM translator. This translator will serve as the backend of the compiler that we will build in chapters 10 and 11, following the introduction of a high-level object-based language in chapter 9.

In any “Great Gems in Computer Science” contest, *stack processing* will be a strong finalist. The previous chapter showed how any arithmetic and Boolean expression could be calculated by elementary stack operations. This chapter goes on to show how this remarkably simple data structure can also support remarkably complex tasks like dynamic memory management, nested subroutine calling, parameter passing, and recursion. Most people tend to take these programming capabilities for granted, expecting modern programming languages to deliver them, one way or another. We are now in a position to open this black box, and see how these fundamental programming mechanisms can be supported and implemented using a relatively simple stack-processing model.

1. Background

The previous chapter focused on the arithmetic, logical, and data management operations of a typical stack-based, virtual machine. This, of course, was just the beginning. If we want our VM to become the backend of present and future compilers, we obviously need to support program flow and subroutine handling capabilities as well. We will do this by equipping the basic VM with two additional and final sets of commands: *program flow* commands for handling conditional and unconditional branching, and *function commands* for handling subroutine calls.

The remainder of this section gives an informal introduction to both subjects. This sets the stage for section 2, which rounds up the VM specification started in Chapter 7. Sections 3 and 4 discuss how to actually complete the VM implementation, leading to a full-scale VM-to-Hack translator.

1.1 Program flow

The default execution of computer programs is linear, one command after the other. This sequential flow is occasionally broken, e.g. to embark on another iteration of a loop. In low-level

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

programming, this branching logic is accomplished by instructing to continue execution at some specified part of the program other than in the next instruction, using a “*goto destination*” command. The destination specification can take several forms, the most primitive being the physical address of the instruction that should be executed next. A slightly more abstract redirection is established by describing the jump destination using a symbolic label rather than a physical address. This variation requires that the language be equipped with some *labeling command*, designed to assign symbolic labels to selected points in the program.

The basic *goto* mechanism just described can be easily altered to affect *conditional branching* as well. Instead of jumping to some destination unconditionally, an “*if-goto destination*” command instructs to take the jump only if a certain Boolean condition is true; if the condition is false, the regular program flow should continue, executing the next command in the code. How should we introduce the Boolean condition specification into the *if-goto* mechanism? In stack-based machines, the simplest and most natural approach is to condition the jump on the value of the stack’s top element: if it’s not zero, jump to the specified destination; otherwise execute the next command in the program. Since the topmost stack value can be computed using any series of arithmetic and logical VM operations, one can condition the jump operation on arbitrarily complex Boolean expressions.

As is often the case in computer science, humble appearance often belies a great power of expression. In this case, the simple *goto* and *if-goto* commands can be used to express all the conditional and repetition constructs found in any high-level programming language. Figure 1 gives two typical examples.

<i>High-level source code</i>	<i>Compiled low-level pseudo code</i>
<pre>if (cond) s1 else s2 ... </pre>	<pre>code for computing cond if-false-goto L1 code for executing s1 goto L2 label L1 code for executing s2 label L2 ... </pre>
<pre>while (cond) s1 ... </pre>	<pre>label L1 code for computing cond if-false-goto L2 code for executing s1 goto L1 label L2 ... </pre>

Figure 1: Implementing flow of control using *goto* and *if-goto* commands.

1.2 Subroutine Calls

Any programming language is characterized by a fixed repertoire of elementary commands. The key abstraction mechanism provided by modern languages is the freedom to extend this repertoire with high-level operations, designed to meet various programming needs. Each high-level operation has an *interface* specifying how it can be used, and an *implementation* consisting of elementary commands and previously defined high-level operations. In procedural languages, the high-level operations are called *subroutines*, *procedures*, or *functions*. In object-oriented languages they are usually called *methods*, and are typically grouped into *classes*. In this chapter we will use the term *subroutine* to refer to all these high-level programming constructs.

The use of a subroutine is typically referred to as a *call* operation. Ideally, the part of the program that calls the subroutine -- the *caller* -- should treat the subroutine like any other basic operation in the language. To illustrate, the caller typically contains a sequence of commands like $\langle c1, c2, \text{call } s1, c3, \text{call } s2, c4, \dots \rangle$, where the *c*'s are elementary commands and the *s*'s are subroutine names. In other words, the caller assumes that the code of the called subroutine will get executed -- somehow -- and that following the subroutine's termination the flow of control will return -- somehow -- to the next instruction in the caller's code. The freedom to ignore these implementation details enables us to write programs in abstract terms, using high-level operations that are closer to the world of algorithmic thought than to the world of machine execution.

Of course the more abstract is the high level, the more work the low level must do. In particular, in order to support subroutine calls, VM implementations must handle several issues:

- Passing parameters to the called subroutine, and optionally returning a value from the called subroutine back to the caller;
- Allocating memory space for the local variables of the called subroutine, and freeing the memory when it is no longer needed;
- Jumping to execute the called subroutine's code;
- When the called subroutine terminates, returning (jumping back) to the command following the call operation in the caller's code.

These issues must be handled in a way that takes into account that subroutine calls can be arbitrarily nested, i.e. one subroutine may call another subroutine, which may then call another subroutine, and so on and so forth, to any desired depth. To add to the complexity, we also need to support *recursion*. This means that subroutines should be allowed to call themselves, and each recursion level must be executed independently of the other calls.

1.3 Stack-Based Implementation

We see that the low-level handling of subroutine calls is rather delicate. The property that makes this task tractable is the hierarchical structure of the call-and-return logic: the called subroutine must complete its execution before the caller can resume its own execution. This protocol implies a *Last-In-First-Out* (LIFO) structure, resembling (conceptually) a *stack* of active subroutines. All the layers in the stack are waiting for the top layer to complete its execution, at

which point the stack become shorter and execution resumes at the level just below the previous top layer.

Indeed, users of high-level programming languages often encounter terms like “call-stack,” “stack overflow,” and so on. To illustrate, figure 2 shows a method calling pattern in a high-level program, along with some run-time checkpoints and the states of the abstract call-stack associated with them.

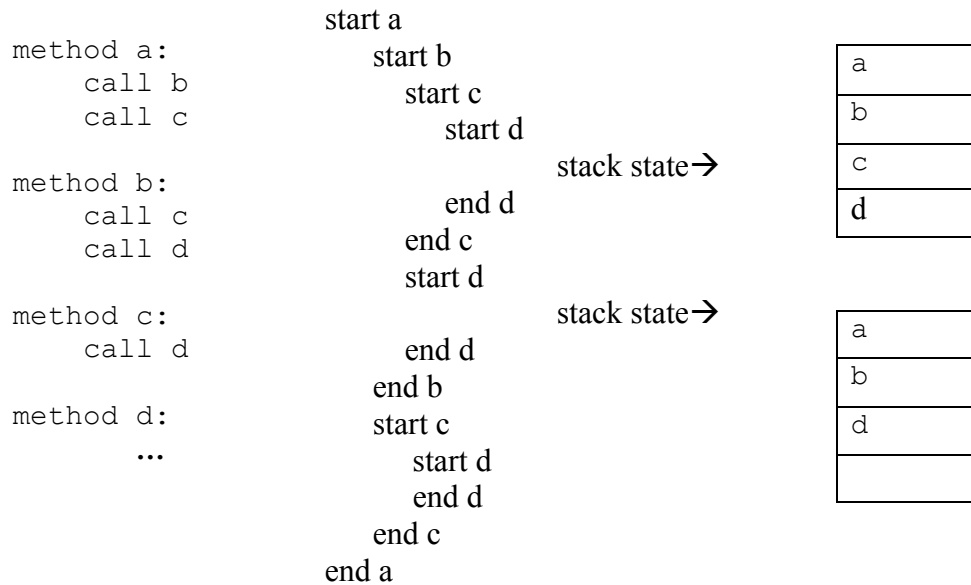


Figure 2: Subroutine calls and the abstract call-stack states generated by their execution.

It is perhaps useful to note that from this point onward, the term *stack* will be used rather freely, and the reader should be able to tell from the sentence context the which stack we are taking about. For example, the *call stack* in Figure 2 is merely a conceptual notion, listing the names of all the active subroutine that are presently running. The *global stack*, on the other hand, is a real object. In particular, note that each subroutine that has not yet returned must maintain somehow its private set of local variables, argument values, pointers, and so on. Taken together, these data items are called the *method's frame*. Where should we keep all these frames? As Figure 2 shows, we can put them on the *global stack*. The reader may wonder where the *working stack* from the previous chapter fits in -- the stack that supports the VM's push, pop, and arithmetic operations. Well, this stack can be maintained at the very top of the global stack, as we will see later.

The agent responsible for maintaining the global stack and implementing the call-and-return mechanism is the VM implementation. In order to carry out this stack, the VM implementation must handle such issues as return addresses, local variables allocation and de-allocation, and parameter passing.

Return address: The VM implementation of the “call subName” command is straightforward. Since the name of the target subroutine is specified in the command, the VM implementation has to resolve the name to a memory address -- a rather simple task -- and then jump to execute the code starting in that address.

Returning from the called subroutine via a “return” command is trickier, as the command specifies no return address. Indeed, the caller’s anonymity is inherent in the very notion of a subroutine call. For example, subroutines like `sqrt(x)` or `modulu(x,y)` are designed to serve many unknown callers, implying that the return address cannot be part of their code. Instead, a “return” command should be interpreted as follows: re-direct the program’s execution to the command following the command that called the current subroutine, wherever this command may be in the program’s code. The memory location to which we have to return is called *return address*.

One way to implement the return logic is to have the VM implementation save the return address just before the subroutine is called, and have it retrieved just after the subroutine exits. Conveniently, this store-and-recall setting lends itself perfectly to *stack* storage: the VM implementation can push the return address onto the stack when a subroutine is called, and pop it from the stack when the subroutine returns. In terms of Figure 2, the return address can be kept in the method’s frame.

Parameter passing: An important characteristic of well-designed languages is that high-level operations defined by the programmer will have the same “look and feel” as that of elementary commands. Consider for example the operations *add* and *raise to a power*. VM implementations will typically feature the former as an elementary operation, while the latter may be written as a subroutine. In spite of their different implementations, we would like to use both operations in the same way. Thus, assuming that we have already written a `Power(x,y)` subroutine that computes x to the y -th power, we would like to be able to write VM code segments like those depicted in Program 3.

<code>// x+3</code>	<code>// x^3</code>	<code>// (x^3+2)^y</code>
<code>push x</code>	<code>push x</code>	<code>push x</code>
<code>push 2</code>	<code>push 3</code>	<code>push 3</code>
<code>add</code>	<code>call power</code>	<code>call power</code>
		<code>push 2</code>
		<code>add</code>
		<code>push y</code>
		<code>call power</code>

PROGRAM 3: VM elementary commands and high-level operations have the same look-and-feel in terms of arguments usage and return values. Thus they can be easily mixed together, yielding well-designed and readable code.

Note that from the caller’s perspective, any subroutine -- no matter how complex -- is viewed and treated as a black box operation. In particular, just like with elementary VM commands, the caller expects the subroutine to remove its arguments from the stack and replace them with a return value (which may be ignored by the caller). Thus, the contract is as follows: the caller passes the arguments to the subroutine by pushing them onto the stack; the called subroutine pops

the arguments from the stack, as needed, carries out its computation, and then pushes a return value onto the stack. The result is a simple and natural parameter passing protocol requiring no memory beyond the already available stack structure.

Local variables: Subroutines rely on local variables for temporary storage. And when a subroutine is used recursively, each recursion level must maintain its own set of local variables. Note however that these variables must be stored in memory only during the subroutine call's lifetime, i.e. from the point the subroutine starts executing until it returns. At this point, the memory space allocated to the local variables can be freed. How can the VM implementation effect this dynamic memory allocation?

Once again, the hard-working stack comes to the rescue. Although the subroutine calling chain may be arbitrarily deep as well as recursive, only one subroutine executes at the end of the chain, while all the other subroutines up the calling chain are waiting. The VM implementation can exploit this Last-In-First-Out (LIFO) processing model by storing the local variables of all the waiting subroutines on the stack, and reinstate them when control returns to the subroutine to which they belong. Revisiting Figure 2, we see that local variables can be saved in, and indeed they are part of, the method's frame.

2. VM Specification, Part II

This section extends the basic VM Specification from Chapter 7 with *program flow* and *function* commands. This completes the overall VM speciation.

2.1 Program Flow Commands

The VM language features three program flow commands:

<code>label c</code>	This command labels the current location in the function's code. Only labeled locations can be jumped to from other parts of the program. The label <code>c</code> is an arbitrary string composed of letters, numbers, and the special characters “_”, “:”, and “.”. The scope of the label is the current function.
<code>goto c</code>	This command effects a "goto" operation, causing execution to continue from the location marked by the <code>c</code> label. The jump destination must be located in the same function.
<code>if-goto c</code>	This command effects a "conditional goto" operation. The stack's topmost value is popped; if the value is not zero, execution continues from the location marked by the <code>c</code> label; otherwise, execution continues from the next command in the program. The jump destination must be located in the same function.

TABLE 4: Program flow commands.

2.2 Function Commands

Each function has a symbolic name that is used globally to call it. The function name is an arbitrary string composed of letters, numbers, and the special characters “_” and “.”. (We expect that a method `bar` in class `Foo` in some high-level language will be translated by the language compiler to a VM function named `Foo.bar`).

The VM language features three function-related commands:

<code>function f n</code>	Here starts the code of a function named <code>f</code> , which has <code>n</code> local variables;
<code>call f m</code>	Call function <code>f</code> , stating that <code>m</code> arguments have already been pushed onto the stack;
<code>return</code>	Return to the calling function.

TABLE 5: Function calling commands.

The Calling Protocol

The events of calling a function and returning from a function can be viewed from three different perspectives: that of the calling function, the called function, and the VM implementation.

The *calling function* view:

1. Before calling the function, I (the caller) must push all the arguments unto the stack;
2. Next, I invoke the called function `f` using the command “`call f`”;
3. After the called function returns, the arguments that I pushed before have disappeared from the stack and the function’s *return value* (that always exists) appears at the top of the stack;
4. After the called function returns, all my memory segments (e.g. `arguments` and `locals`) are the same as before the call, except for the `Temp` segment that is now undefined.

The *called function* view:

1. Upon getting called, my `argument` segment has been initialized with values passed by the caller, my `local variables` segment has been allocated and initialized to zero, the working stack that I see is empty, and the `static` segment that I see has been set to the `static` segment of the file to which I belong. All the other memory segments are undefined and can be used as needed.
2. Just before returning, I must push a value onto the stack.

The VM implementation view:

When a function *calls* another function, I (the VM implementation) must:

- Save the return address and the segment pointers of the calling function (except for `temp` which is not saved);
- Allocate, and initialize to zero, as many local variables as needed by the called function;
- Set the `local` and `argument` segments of the called function;
- Transfer control to the called function.

When a function *returns*, I (the VM implementation) must:

- Clear the arguments and other junk from the stack;
- Restore the `local`, `argument`, `this` and `that` segments of the calling function;
- Transfer control back to the calling function, by jumping to the saved return address.

2.3 Initialization

When the VM starts running (or is reset), the VM function named “`sys.init`” gets executed.

3. Implementation

We are now ready to complete the VM implementation whose first part was specified in Chapter 7. We begin by laying out the full stack structure that must be maintained by the implementation, and how it can be mapped over the Hack platform. Next, we give design suggestions and a proposed API, leading to a full-scale virtual machine implementation, based on a VM-to-Hack translator. This program can be viewed as a stand-alone language translator, as well as the backend module of our future compiler.

3.1 The Global Stack

The “system memory” of the VM is implemented by maintaining a global stack. Each time a function is called, a new block is added to the global stack. The block consists of the *arguments* that were set for the called function, a set of *pointers* used to save the state of the calling function, the *local variables* of the called function (initialized to 0), and an empty *working stack* for the called function. Importantly, the called function sees only the tip of this iceberg, i.e. the working stack. The rest of the global stack is used only by the VM implementation and is not visible to VM functions.

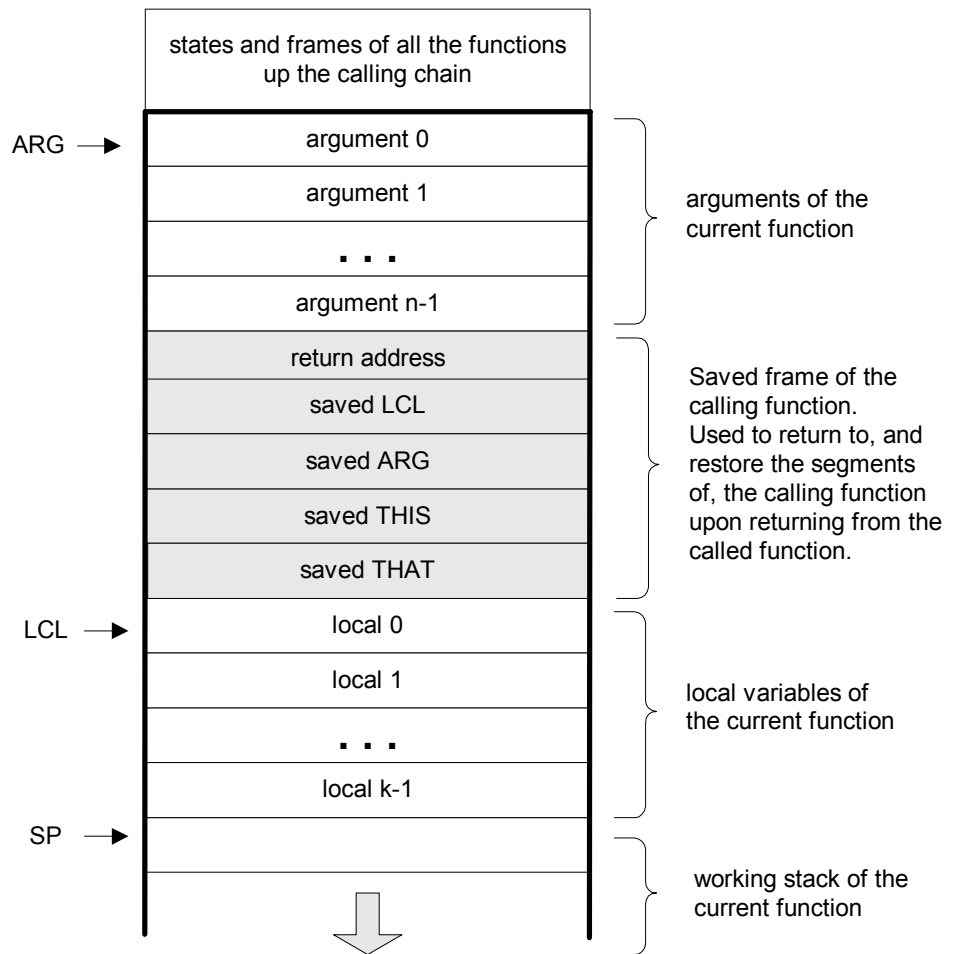


FIGURE 6: The global stack

Example: The factorial ($n!$) of a given number n can be computed by the bottom-up iterative formula $n! = 1 \cdot 2 \cdot \dots \cdot (n-1) \cdot n$. This algorithm is shown in Figure 7, along with a time-line of a typical run-time.

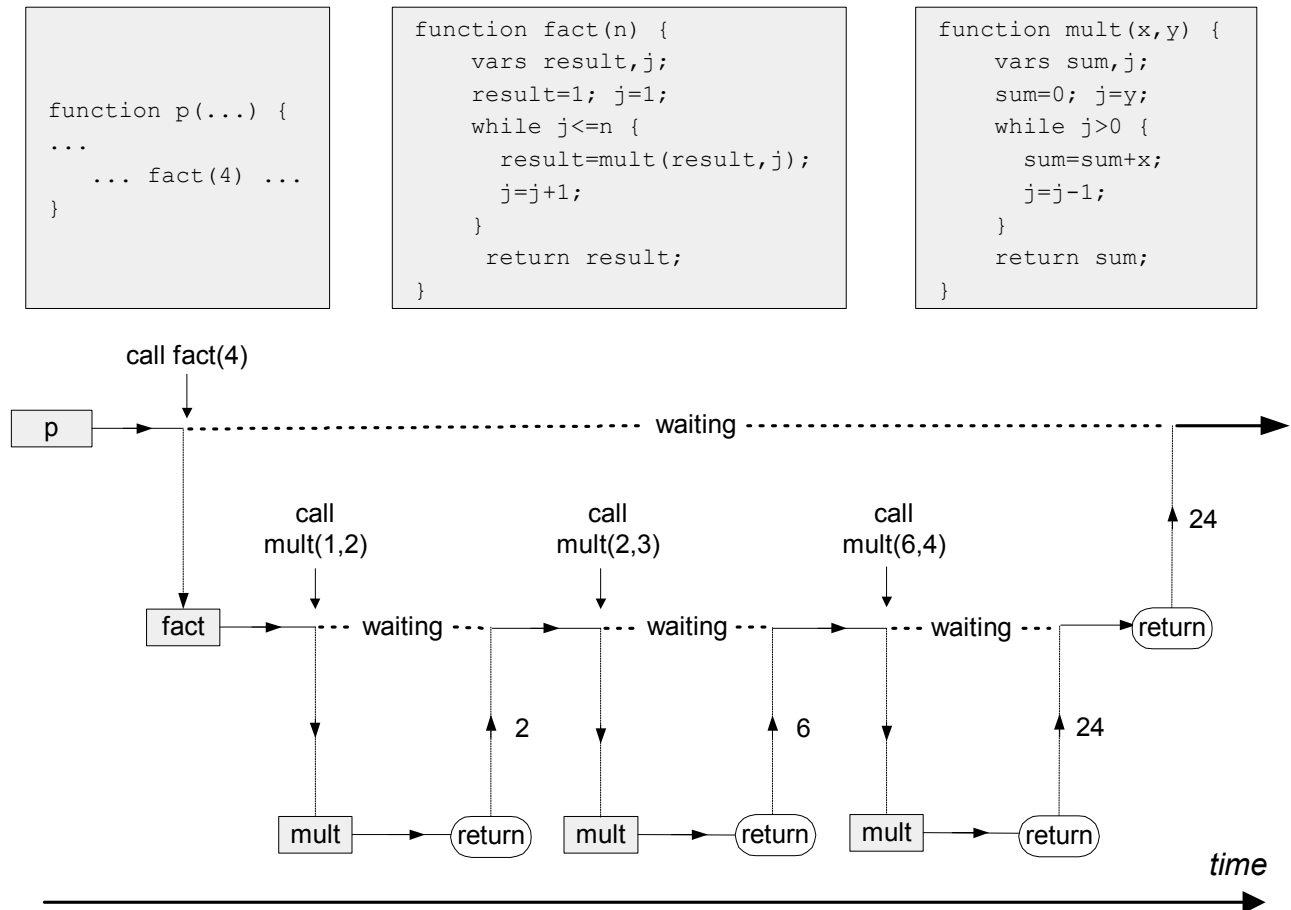


FIGURE 7: Function call-and-return routine: an arbitrary function `p` calls function `fact`, which then calls `mult` several times. Vertical arrows depict transfer of control from one function to another. At any given point of time, only one function is running, while all the functions up the calling chain are waiting for it to return. When a function returns, the function that called it resumes its execution (which typically does something useful with the value returned by the called function).

Of course our concern here is neither the `fact` nor the `mult` functions, and that's why did not bother to write them in the VM language. Rather, we wish to shed light on the hidden infrastructure that enables these functions to interact with each other through parameter passing, return values, and control re-direction. The centerpiece of this infrastructure is the global stack, as seen in Figure 8.

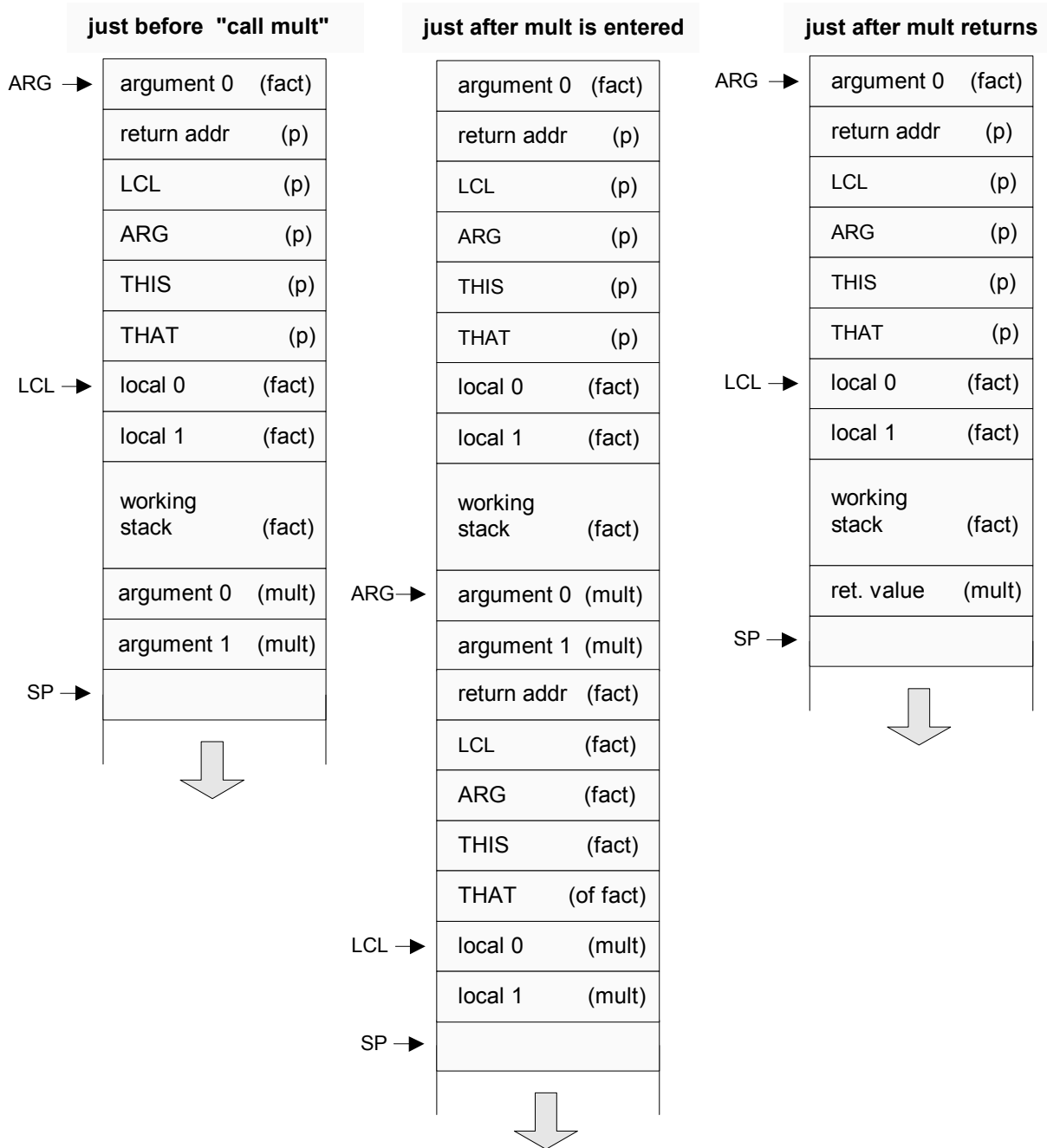


FIGURE 8: Global stack dynamics. We assume that function *p* (not seen in this figure) called *fact*, then *fact* called *mult*. If we ignore the middle stack instance, we observe that *fact* has set up some arguments and called *mult* to operate on them (left instance). When *mult* returns (right instance), the arguments of the called function have been replaced with the function's return value. In other words, when the dust clears from the function call, the calling function has received the service that it has requested, and processing resumes as if nothing happened: the drama of *mult*'s processing (middle) has left no trace whatsoever on the stack, except for the return value.

3.2 Standard Mapping on the Hack Platform, Part II

By *standard mapping* we refer to a set of guidelines on how to map VM implementations on a specific target architecture. This section completes the standard VM-on-Hack mapping whose first part was given in Chapter 7.

Function Calling Protocol

The subroutine calling mechanisms of modern programming languages (e.g. Figures 6-7) can be implemented using stack operations. Table 9 gives the details.

<i>VM command</i>	<i>VM-on-Hack Implementation action (pseudo code)</i>
<u>Calling a function:</u> call f n	<pre> push return-address // (using label below) push LCL // save LCL of calling function push ARG // save ARG of calling function push THIS // save THIS of calling function push THAT // save THAT of calling function ARG = SP-n-5 // reposition ARG (n=number of args) LCL = SP // reposition LCL goto f // transfer control (return-address) // label for the return address </pre>
<u>Function declaration:</u> function f k	<pre> (f) // declare label for function entry repeat k times: // k=number of local variables PUSH 0 // initialize all of them to 0 </pre>
<u>Returning from a function:</u> return	<pre> FRAME=LCL // FRAME is a temporary variable RET=*(FRAME-5) // save return address in a temp. var *ARG=pop() // reposition return value for caller SP=ARG+1 // restore SP for caller THAT=*(FRAME-1) // restore THAT of calling function THIS=*(FRAME-2) // restore THIS of calling function ARG=*(FRAME-3) // restore ARG of calling function LCL=*(FRAME-4) // Restore LCL of calling function goto RET // GOTO the return-address </pre>

TABLE 9: VM implementation of function commands (pseudo code).

Assembly Language Symbols

<i>Symbol</i>	<i>Usage</i>
“functionName:label” symbols	Each “label <i>b</i> ” command in a function <i>f</i> should generate a globally unique symbol <i>f:b</i> where <i>f</i> is the function name and <i>b</i> is the label symbol within the function’s code. When translating “goto <i>b</i> ” and “if-goto <i>b</i> ” commands into the target language, the full label specification <i>f:b</i> should be used instead of <i>b</i> .
“functionName” labels	Each function <i>f</i> should generate a symbol <i>f</i> that refers to its entry point in the instruction memory of the target architecture.
<i>return address</i> symbols	Each function call should generate a unique symbol that serves as a return address, i.e. the location of the command following the call command in the instruction memory of the target architecture.

TABLE 10: Special assembly symbols prescribed by the standard mapping.

Bootstrap Code

Upon reset, the Hack hardware is wired to fetch and execute the word located in ROM address 0x0000. Thus, the code segment that starts at address 0x0000, called *bootstrap code*, is the first thing that gets executed when the computer “boots up”. As a convention, we want this code to effect the following operations (in machine language):

```
SP=256          // initialize the stack pointer to 0x0100
call Sys.init   // invoke Sys.init
```

This code sets the stack pointer to its right value (as per the standard mapping) and then calls the `Sys.init` function. The contract is that `Sys.init` should then call the main function of the main program, and enter an infinite loop. Taken together, these operations should cause the translated VM program to start running.

The “main function” and the “main program” are compilation-specific and vary from one high level language to another. For example, in the Jack language, the default is that the first program unit that starts running automatically is the `main` method of a class named `Main`. In a similar fashion, when we tell the JVM to execute a given Java class, say `Foo`, it will look for, and execute, the `Foo.main` method. Such “automatic” start-up routines can be effected by the bootstrap logic described above.

3.3 Design Suggestions for the VM implementation

In chapter 7 we proposed implementing the VM translator as a main program consisting of two modules: *parser* and *code writer*. The basic translator built in Project 7 was based on basic

versions of these modules. In order to turn the basic translator into a full-scale VM implementation, we have to extend the basic *parser* and *code writer* modules with the functionality described below.

Parser

If the basic parser that you built in Project 7 does not already parse the six commands specified in this chapter, then add their parsing now. Specifically, make sure that the `commandType` method developed in Project 7 also returns the constants corresponding to the six VM commands described in this chapter: `C_LABEL`, `C_GOTO`, `C_IF`, `C_FUNCTION`, `C_RETURN`, `C_CALL`.

Code Writer

The basic `CodeWriter` specified in Chapter 7 should be augmented with the following methods.

CodeWriter Module			
Translates VM commands into Hack assembly code.			
The routines listed below should be added to the <code>CodeWriter</code> module API given in Chapter 7.			
Routine	Arguments	Returns	Function
<code>writeInit</code>	--	--	Writes the assembly code that effects the VM initialization (also called <i>bootstrap code</i>). This code should be placed in the ROM beginning in address <code>0x0000</code> .
<code>writeLabel</code>	<code>label (string)</code>	--	Writes the assembly code that is the translation of the given <code>label</code> command.
<code>writeGoto</code>	<code>label (string)</code>	--	Writes the assembly code that is the translation of the given <code>goto</code> command.
<code>writeIf</code>	<code>label (string)</code>	--	Writes the assembly code that is the translation of the given <code>if-goto</code> command.
<code>writeCall</code>	<code>functionName (string)</code> <code>numArgs (int)</code>	--	Writes the assembly code that is the translation of the given <code>Call</code> command.
<code>writeReturn</code>	--	--	Writes the assembly code that is the translation of the given <code>Return</code> command.
<code>writeFunction</code>	<code>functionName (string)</code> <code>numLocals (int)</code>	--	Writes the assembly code that is the trans. of the given <code>Function</code> command.

4. Perspective

Work in progress.

5. Build it

Objective: Extend the basic VM translator built in project 7 with the ability to handle the *program flow* and *function* commands specified in this chapter. The VM should be implemented on the Hack computer platform, conforming to the *standard mapping* described in this chapter.

Resources (same as in Project 7): You will need two tools: the programming language in which you will implement your VM Translator, and the *CPU Emulator* supplied with the book. This emulator allows executing the machine code generated by your VM Translator -- an indirect way to test the correctness of the latter. Another tool that may come handy in this project is the visual *VM Emulator* supplied with the book. This program allows experimenting with a working VM environment before you set out to implement it yourself. For more information about this tool, refer to the *VM Emulator Tutorial*.

Contract: Write a full-scale VM-to-Hack translator, extending the translator developed in Project 7. Use it to translate the test `.vm` programs supplied below, yielding corresponding `.asm` 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.

Proposed Implementation Stages

We recommend implementing the translator in two stages.

- Stage I: Implementation of *program flow* commands
- Stage II: Implementation of *function* commands

This modularity will allow you to test your implementation incrementally, using the step-by-step test programs that we provide.

Test Programs

The supplied test programs are designed to support the incremental development plan described above. We supply five test programs and test scripts, as follows.

Program Flow Test Programs

- `basicLoop`: Simple test of `goto` and `if-goto` commands. Computes the sum $1 + 2 + \dots + n$ and pushes the result onto the stack;
- `fibonacci`: A more challenging test. Computes and stores in memory the first n elements of the Fibonacci series.

Function Calling Test Programs

- `simpleFunction`: Simple test of the “function” and “return” commands. The function performs a simple calculation and returns the result.
- `FibonacciElement`: A full test of the function call commands, the bootstrap section and most of the other VM commands. The `FibonacciElement` directory consists of two `.vm` files:
 - `Math.vm` contains one recursive function called `fibonacci`. This function returns the n 'th element of the Fibonacci series;
 - `Sys.vm` contains one function called `init`. This function calls the `Math.fibonacci` function with $n=4$, and then loops infinitely.

Since the overall program consists of two `.vm` files, the entire directory should be compiled in order to create a single `FibonacciElement.asm` file (compiling each `.vm` file separately will yield two separate `.asm` files, which is not desired here).

- `StaticTest`: A full test of static variables handling. Consists of two `.vm` files, each representing the compilation of a typical class file, and a `sys.vm` file, as usual. Once again, the entire directory should be compiled in order to create a single `StaticTest.asm` file.

As prescribed by the *VM Specification* (section 2), the bootstrap code must include a call to the `Sys.init` function.

Steps

1. Download `project8.zip` and extract its contents into a directory called `project8` on your computer, without changing the directories structure embedded in the zip file.
2. Write and test the full-scale VM translator in stages, as described above.

9. The High Level Language¹

"High thoughts need a high language."

(Aristophanes, 448-380 BC)

This (work-in-progress) chapter presents an overview and specification of the *Jack* programming language. Jack is a simple object-based language that can be used to write high-level programs. It has the basic features and flavor of modern object-oriented languages like Java, with a much simpler syntax and no support for inheritance. The introduction of Jack marks the beginning of the end of our journey. In chapters 10 and 11 we will write a compiler that translates Jack programs into VM code, and in Chapter 12 we will develop a simple operating system for the Jack/Hack platform, written in Jack. This will complete the computer's construction.

The chapter starts with a very brief background. The bulk of the chapter describes and specifies the Jack language and its standard library (operating system). This is all the information needed for writing applications in the Jack language. Next, we illustrate some computer games written in Jack, and give general guidelines on how to write similar interactive applications over the Hack platform. All these programs can be compiled using a *Jack compiler* that we provide, and then run on the computer hardware that we built in chapters 1-5. Throughout the chapter, our goal is not to turn you into a Jack programmer. Instead, our "hidden agenda" is to prepare you to write the compiler and operating system described in the chapters that lie ahead.

1. Background

It's important to note at the outset that in and by itself, Jack is a rather uninteresting and simple-minded language. However, this simplicity has a purpose. First, you can learn (and unlearn) Jack very quickly -- in about an hour. Second, the Jack language was carefully planned to lend itself nicely to simple compilation techniques. As a result, one can write an elegant *Jack compiler* with relative ease, as we will do in Chapters 10 and 11. In other words, the deliberately simple structure of Jack will help us uncover the software engineering principles underlying modern languages like Java and C#. Rather than taking the compilers and run-time environments of these languages for granted, we will build a Jack compiler and a run-time environment ourselves, beginning in the next chapter. For now, let's take Jack out of the box.

2. The Jack Language

We begin with some illustrations of Jack programming, and continue with a formal language specification. The former is all that you need in order to start writing Jack programs, and the latter is a complete language reference.

2.1 Examples

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

Example 1: Hello World

Our first example is the classic Hello-World program:

```
/** Hello World program */
class Main {
    function void main(){
        do Output.printString ("HELLO WORLD");
        do Output.println();      // new line
        return;
    }
}
```

PROGRAM 1: Hello World.

This program consists of one class, called `Main`, which contains one function, called `main`, that contains a sequence of two invocations of functions from the standard library. The `Main.main()` method is where the execution starts in every program. You can see two comment formats: the `/** */` documentation comment, and the `//...` one line comment.

Example 2: Fraction

The task: Every programming language has a fixed set of primitive data types, of which Jack supports three: `int`, `char`, and `boolean`. Suppose we wish to endow Jack with the added ability to handle *rational numbers*, i.e. objects of the form n/m where n and m are integers. This can be done by creating a stand-alone class, designed to provide a fraction abstraction for Jack programs. Let us call this class `Fraction`.

Defining the class interface: A reasonable way to get started is to specify the set of properties and services that one would normally expect to see in a fraction abstraction. One such *Application Program Interface*, or *API*, is given in Program 2.

Fraction (partial API):

```
// A "Fraction" is an object representation of  $n/m$  where  $n$  and  $m$  are integers (e.g. 17/253)
field int numerator, denominator:      Represent  $n$  and  $m$ 
constructor Fraction new(int a, int b): Returns a new Fraction object.
method int getNumerator():             Returns the numerator of this object.
method int getDenominator():          Returns the denominator of this object.
method Fraction plus(Fraction other):  Returns the fraction sum of this fraction
                                        and the other one.
method void print():                   Prints this object in the format
                                        "numerator/denominator".
```

PROGRAM 2: API for the Fraction abstraction. In Jack, operations on the current object (referred to as *this* object) are represented by *methods*, and class-level operations (“static methods” in, e.g., Java) are represented by *functions*.

Using the class (Example 1): APIs mean different things to different people. If you are the programmer who has to *implement* the fraction abstraction, you can view its API as a contract that must be implemented in some way. Alternatively, if you are a programmer who needs to *use* fractions in your work, you can view the API as a library of fraction objects generators and fraction-related operations. Taking this latter view, consider the Jack code listed in Program 3.

```
/** Compute the sum of 2/3 and 1/5 */
class Main {
  function void main(){
    var Fraction a, b, c;
    let a = Fraction.new(2,3);
    let b = Fraction.new(1,5);
    let c = a.plus(b); // compute c=a+b
    do c.print(); // should print the text: "13/15"
    return;
  }
}
```

PROGRAM 3: Using the Fraction abstraction in Jack programs

Program 3 illustrates several features of the Jack language: declaration and creation of new objects, use of the assignment statement `let`, and method calling using the `do` statement. As the code implies, users of the fraction abstraction don’t have to know anything about its underlying *implementation*. Rather, they should be given access only to the fraction *interface*, or API, and then use it as a black box server of fraction-related operations.

Implementing the class: We now turn to the other player in our story -- the programmer who has to actually implement the fraction API in Jack. One possible implementation is given in Program 4. This example illustrates several additional features of object-oriented programming in Jack: a typical program structure (*classes, methods, constructors, functions*), as well as all statement types (*let, do, return, if, while*) available in Jack.

```
/** A Fraction (partial implementation) */
class Fraction {

    field int numerator, denominator;

    /** Construct a new (reduced) fraction given a numerator
    and denominator */
    constructor Fraction new(int a, int b){
        let numerator = a;
        let denominator = b;
        do reduce(); // if a/b is not reduced, reduce it.
        return this;
    }

    /* reduce this fraction - internal method (from the outside,
    a fraction always seems reduced.) */
    method void reduce() {
        var int g;
        let g = Fraction.gcd(numerator, denominator);
        if (g > 1) {
            let numerator = numerator / g;
            let denominator = denominator / g;
        }
        return;
    }

    /* compute the gcd of a and b - internal service function */
    function int gcd(int a, int b){
        var int r;
        // apply Euclid's algorithm
        while (~ (b = 0)) {
            let r = a - (b * (a / b)); // r = remainder of division a/b
            let a = b;
            let b = r;
        }
        return a;
    }
}

// the code continues in PROGRAM 4 (part II)
```

PROGRAM 4 (part I): a Fraction class implementation.

```
// continuation of the PROGRAM 4 (part I) code:

method int getNumerator(){
    return numerator;
}

method int getDenominator(){
    return denominator;
}

/** Return the sum of this fraction and the other one */
method Fraction plus(Fraction other){
    var int sum;
    let sum = (numerator * other.getDenominator())
              +(other.getNumerator() * denominator());
    return Fraction.new(sum, denominator * other.getDenominator());
}

// more methods: minus, times, div, ...

/** print this fraction */
method void print() {
    do Output.printInt(numerator);
    do Output.printString("/");
    do Output.printInt(denominator);
    return;
}

} // end Fraction
```

PROGRAM 4 (part II): a Fraction class implementation (continued)

We now turn to a formal description of the Jack language, focusing on its syntactic elements, program structure, variables, expressions, and statements.

2.2 Syntactic Elements

A Jack program is a sequence of tokens. Tokens are separated by an arbitrary amount of white space (including comments), that is ignored. Tokens can be *symbols*, *reserved words*, *constants*, and *identifiers*, as listed in Table 5.

White space and comments	<p>Space characters, newline characters, and <i>comments</i> are ignored. The following <i>comment</i> formats are supported:</p> <pre>// comment to end of line /* comment until closing */ /** API documentation comment */</pre>												
Symbols	<p>() : arithmetic grouping, and parameter/argument-lists grouping [] : array indexing; { } : program structure and statement grouping; , : variable list separator; ; : statement terminator; = : assignment and comparison operator; . : class membership; + - * / & ~ < > : operators.</p>												
Reserved words	<table> <tr> <td>class, constructor, method, function:</td> <td>Program components</td> </tr> <tr> <td>int, boolean, char, void:</td> <td>Primitive types</td> </tr> <tr> <td>var, static, field:</td> <td>Variable declarations</td> </tr> <tr> <td>let, do, if, else, while, return:</td> <td>Statements</td> </tr> <tr> <td>true, false, null:</td> <td>Constant values</td> </tr> <tr> <td>this:</td> <td>Object reference</td> </tr> </table>	class, constructor, method, function:	Program components	int, boolean, char, void:	Primitive types	var, static, field:	Variable declarations	let, do, if, else, while, return:	Statements	true, false, null:	Constant values	this:	Object reference
class, constructor, method, function:	Program components												
int, boolean, char, void:	Primitive types												
var, static, field:	Variable declarations												
let, do, if, else, while, return:	Statements												
true, false, null:	Constant values												
this:	Object reference												
Constants	<p>Positive <i>Integer</i> constants are in standard decimal notation, e.g. “1984”. Negative numbers like “-13” are not constants, but rather a unary minus operator applied to an integer constant.</p> <p><i>String</i> constants are enclosed within two quote (“) characters and may contain any characters except <i>newline</i> or <i>double-quote</i>. (These characters can be obtained using the function calls <code>String.newLine()</code> and <code>String.doubleQuote()</code>).</p> <p><i>Boolean</i> constants can be <code>true</code> or <code>false</code>.</p> <p>The constant <code>null</code> signifies a null reference.</p>												
Identifiers	<p>Identifiers are composed from arbitrarily long sequences of letters (A-Z, a-z), digits (0-9), and “_”. The first character must be a letter or “_”. The language is case sensitive.</p>												

TABLE 5: Syntactic elements of the Jack language.

2.3 Program Structure

The basic programming unit in Jack is a *class*. Each class resides in a separate file and can be compiled separately.

Class declarations have the following format:

```
class name {  
    field and static variable declarations // must precede the subroutine declarations  
    subroutine declarations // a sequence of constructor, method, and function declarations  
}
```

Each class declaration begins with a name through which the class can be globally accessed. Next comes an optional sequence of declarations of class-level *fields* and *static variables*, described below. Next comes a sequence of *subroutine* declarations, defining *methods*, *functions*, and *constructors*. A method “belongs” to an object and provides the objects’ functionality, while a function “belongs” to the class in general and is not associated with a particular object (similar to Java’s *static methods*). A constructor “belongs” to the class and generates object instances of this class.

Subroutine declarations have the following formats:

```
constructor type name (parameter-list) {  
    declarations  
    statements  
}  
  
method type name (parameter-list) {  
    declarations  
    statements  
}  
  
function type name (parameter-list) {  
    declarations  
    statements  
}
```

Each subroutine has a *name* through which it can be accessed. The subroutine’s *type* specifies the type of the value returned by the subroutine. If the subroutine returns no value, the subroutine type is declared `void`; otherwise it can be any of the primitive or object data types supported by the language (section 2.4). Constructors may have arbitrary names but must return the class’s own type, i.e. the class name.

Each subroutine contains an arbitrary sequence of local variable declarations and then a sequence of statements.

Jack programs: As in Java, a *Jack program* is a collection of one or more classes. One class must be named `Main`, and this class must include at least one function named `main`. When instructed to execute a Jack program that resides in some directory, the host platform will automatically start running the `Main.main` function.

2.4 Variables

Variables in Jack must be explicitly declared before they are used. There are several kinds of variables: fields, static variables, local variables, and parameters, each with its associated scope. Variables are typed.

Data Types

The type can either be primitive (`int`, `char`, `boolean`) as is pre-defined in the language specification, or an Object type (like `Employee`, `Car`, etc.) which is defined by the programmer, as needed.

Primitive types: Jack features three primitive data types:

- `int`: 16-bit 2's complement;
- `boolean`: *false* and *true*;
- `char`: Unicode character.

Variables of primitive types are allocated to memory when they are declared. For example, the declarations “`var int age; var boolean gender;`” will cause the compiler to create the variables `age` and `gender` and to allocate memory space to them.

Object types: Every class defines an object type. Object types are references (as in Java). The declaration of an object variable creates only a reference and only allocated memory to hold the reference. Memory for the object itself is only allocated later, if and when the programmer actually *constructs* the variable. Example:

```
// The following assumes the existence of Car and Employee classes.
// Car objects have model and licensePlate fields.
// Employee objects have name and car fields.
var Employee e, f; // creates variables e, f that contain a null reference
var Car c; // creates a variable c that contains a null reference
...
let c = Car.new("Jaguar", "007") // constructs a new Car object
let e = Employee.new("Bond", c) // constructs a new Employee object
// At this point c and e hold the references to these two objects.
let f = e; // only the reference is copied
```

PROGRAM 6: Object types.

The Jack standard library defines two built-in object types that play a role in the language syntax: arrays and strings.

Arrays: Arrays are declared using the built-in class `Array`. Arrays are single dimensional and the first index is always 0 (multi-dimensional arrays may be obtained as arrays of arrays). Array entries do not have a declared type, and different entries in the same array may have different types. The declaration of an array only creates a reference; construction of an array is done using the `Array.new(length)` function; access to array elements is done using the `a[j]` notation. The following example reads a sequence of integers and then prints their average.

```
/** Computes the average of a sequence of integers */
class Main {
  function void main() {
    var Array a;
    var int length;
    var int i, sum;

    let length = Keyboard.readInt("HOW MANY NUMBERS? ");
    let a = Array.new(length);
    let i = 0;
    while (i < length) {
      let a[i] = Keyboard.readInt("ENTER THE NEXT NUMBER: ");
      let i = i + 1;
    }
    let i = 0;
    let sum = 0;
    while (i < length) {
      let sum = sum + a[i];
      let i = i + 1;
    }
    do Output.printString("THE AVERAGE IS: ");
    do Output.printInt(sum / length);
    do Output.println();
    return;
  }
}
```

PROGRAM 7: Arrays.

Strings

Strings are declared using the built-in class `String`. The compiler also recognizes the syntax `"xxx"`. String contents can be accessed and modified using the methods of the `String` class (see below). Example:

```
var String s;
var char c;
...
let s = "Hello World";
let c = s.charAt(6);    // "W"
```

Automatic conversions: The Jack language is only weakly typed. The language does not define the results of attempted assignment or conversion from one type to another, and different compilers may allow or forbid different conversions. (This incompleteness of the definition is in order to enable simple compilers to be valid while completely ignoring all typing issues.) All compilers allow, and automatically perform, the following assignments:

- Characters and Integers are automatically converted into each other as needed, according to the Unicode specification.
- An integer can be assigned to a reference variable (of any object type), in which case it is treated as an address in memory.
- Any Object type may be converted into an Array, and vice-versa. The conversion “transforms” the fields of the object into consecutive array entries.

Variable Scopes

Jack features four kinds of variables. *Static variables* are defined at the class level, and are shared by all the objects derived from the class. For example, a `BankAccount` class may have a `totalBalance` static variable that holds the sum of all the balances of all the accounts, where each account is represented as an object of the `BankAccount` class. *Field variables* are used to define the properties of individual objects of the class, e.g. `owner` and `balance` in our banking example. *Local variables*, used by subroutines, exist only as long as the subroutine is running, and *parameters* are used to pass arguments to subroutines. For example, the function signature `function void deposit(BankAccount b, int sum)` indicates that `b` and `sum` are parameters.

Table 8 gives a formal description of all the variable kinds supported by the Jack language.

Var. Kind	Definition and Description	Declared in:	Scope
Static variables	<p>static <i>type name1, name2, ... ;</i></p> <p>Only one copy of each static variable exists, and this copy is shared by all the object instances of the class. (like <i>private static variables</i> in Java)</p>	Class declaration.	The class in which they are declared.
Field variables	<p>field <i>type name1, name2, ... ;</i></p> <p>Every object instance of the class has a private copy of the field variables. (like <i>private object variables</i> in Java)</p>	Class declaration.	The class in which they are declared, but cannot be used in functions.
Local variables	<p>var <i>type name1, name2, ... ;</i></p> <p>Local variables are allocated on the stack when the subroutine is called and are freed when it returns. (like <i>local variables</i> of Java).</p>	Subroutine declaration.	The subroutine in which they are declared.
Parameters	<p><i>type name1, type name2, ...</i></p> <p>e.g.:</p> <pre>function void drive (Car c, int miles) { ... }</pre> <p>Used as inputs of subroutines.</p>	Appear in parameter-lists as part of subroutine declarations.	The subroutine in which they are defined.

TABLE 8: Variable kinds in the Jack language.

2.4 Statements

The Jack language features 5 statements. They are defined and described in Table 9.

Statement	Syntax	Description
let	<pre>let variable=expression; or let variable [expression]=expression;</pre>	An assignment operation. The variable may be static, local, field, or a parameter. Alternatively the variable may be an array entry.
if	<pre>if (expression) { statements } else { statements }</pre>	<p>Typical <i>if</i> statement with an optional <i>else</i> clause.</p> <p>The brackets are mandatory even if <i>statements</i> is a single statement.</p>
while	<pre>while (expression) { statements }</pre>	<p>Typical <i>while</i> statement.</p> <p>The brackets are mandatory even if <i>statements</i> is a single statement.</p>
do	<pre>do function-or-method-call;</pre>	<p>Used to call a function or method for their effect, ignoring the value they return, if any.</p> <p>The function or method call follows the syntax described in section 2.6.</p>
return	<pre>return expression; or return;</pre>	<p>Used to return values from subroutines.</p> <p>The second form must be used by functions and methods that return a void value.</p> <p>Constructors must return the expression <i>this</i>.</p>

TABLE 9: Jack statements.

2.5 Expressions

Expression Syntax: Jack expressions are defined recursively according to the rules given in Table 10.

A *Jack expression* is one of the following:

- ❑ A *constant*
- ❑ A *variable name* in scope (the variable may be *static*, *field*, *local*, or a *parameter*)
- ❑ The `this` keyword, denoting the current object. Cannot be used in functions.
- ❑ An *array element* using the syntax `name[expression]`, where *name* is a variable name of type `Array` in scope.
- ❑ A *subroutine call* that returns a non-void type (see Section 2.6).
- ❑ An expression prefixed by one of the unary operators `{-,~}`:
 - expression*: arithmetic negation
 - ~expression*: boolean negation (bitwise for integers)
- ❑ An expression of the form *expression operator expression* where *operator* is one of the binary operators `{+,-,*,/,&,|,<,>,>=}`:

<code>+ - * /</code>	<code>:</code>	integer arithmetic operators
<code>& </code>	<code>:</code>	boolean And, Or (bitwise for integers) operators
<code>< > =</code>	<code>:</code>	comparison operators
- ❑ An expression in parenthesis: *(expression)*

TABLE 10: Jack expressions are either atomic or derived recursively from simpler expressions

Order of evaluation and operator priority: Operator priority is *not* defined by the language, except that expressions in parentheses are evaluated first. Thus an expression like `2+3*4` may yield either 20 or 14, whereas `2+(3*4)` is guaranteed to yield 14. The need to use parentheses in such expressions makes Jack programming a bit cumbersome. At the same time, the lack of operator priority makes the writing of Jack compilers simpler.

2.6 Subroutine calls

Subroutine calls invoke methods, functions, and constructors for their effect, using the general syntax `subroutineName(argument-list)`. The number and type of the arguments must match those of the subroutine's parameters, as defined in its declaration. The parentheses must appear even if the argument list is empty. Each argument may be an expression of unlimited complexity. For example, consider the function declaration “function int sqrt(int n)” in class `Math`, designed to return the integer part of the square root of its single parameter. Such a function can be invoked using calls like `Math.sqrt(17)`, `Math.sqrt(a*sqrt(c-17)+3)` and so on.

Within a class, methods are called using the syntax *methodName(argument-list)*, while functions and constructors must be called using their full-names, i.e. *className.subroutineName(argument-list)*. Outside a class, the class functions and constructors are also called using the full-name syntax, while methods are called using the syntax *var.methodName(argument-list)*, where *var* is a previously defined object variable. Program 11 gives some examples.

```
class Bar {
  ...
}

class Foo {
  ...
  function void f() {
    var Bar b;
    ...
    ... g(5,7)    // call to method g of class Foo (on this object)
    ... Foo.p(2)  // call to function or constructor p of class Foo
    ... Bar.h(3)  // call to function or constructor h of class Bar
    ... b.q()     // call to method q of class Bar (on object b)
    ...
  }
  ...
}
```

PROGRAM 11: Subroutine call examples.

Object Construction and Disposal: As mentioned, when a variable of an object type is declared, only a reference to an object of this type is allocated. To actually create the object, a class constructor must be called. There must be at least one constructor for every class that defines a type. Constructors may have arbitrary names, but it is customary to call one of them “new”. When constructors are used they are called just like any other class function using the format

```
let <var> = <class_name>.<constructor_name>(<parameter_list>);
```

When the constructor is called, the compiler automatically allocates memory space for the new object, and assigns the allocated space’s base address to *this*. Then the constructor body is executed, in order to initialize the object into a valid state.

Objects can be de-allocated and their space reclaimed using the `Memory.deAlloc(object)` function of the standard library. Convention calls for every class to contain a `dispose()` method that properly encapsulates this de-allocation.

Example: Consider a class named `List`, designed to hold a linked-list. Each link in the list holds an element and a reference to the rest of the list. Program 12 gives a standard `List` implementation.

```
/** A List class (partial implementation) */
class List {
    field int data;
    field List next;

    constructor List new(int car, List cdr){
        let data = car;
        let next = cdr;
        return this;
    }
    ...

    method void dispose() {
        if (~next = null) {
            do next.dispose();
        }
        do Memory.deAlloc(this);
        return;
    }
}

// create a list holding the numbers (2,3,5):
function void create235(){
    var List v;
    let v=List.new(5,null);
    let v=List.new(2,List.new(3,v));
    // ...
    do v.dispose();
    return;
}
```

PROGRAM 12: Examples of object construction and destruction code in Jack. The function constructs the null-terminated linked-list (2,3,5) and then disposes it.

3. Jack Standard Library / Operating System

The Jack language comes with a standard library that may also be viewed as an interface to an underlying operating system. The library is a collection of Jack classes, and must be provided in every implementation of the Jack language. The standard library includes the following classes:

- **Math:** Provides basic mathematical operations;
- **String:** Implements the `String` type and basic string-related operations;
- **Array:** Defines the Array type and allows construction and disposal of arrays;
- **Output:** Handles text based output;
- **Screen:** Handles graphic screen output;
- **Keyboard:** Handles user input from the keyboard;
- **Memory:** Handles memory operations;
- **Sys:** Provides some execution-related services.

This section specifies the subroutines that are supposed to be in these classes.

Math

This class enables various mathematical operations.

- Function `int abs(int x)`: Returns the absolute value of `x`.
- Function `int multiply(int x, int y)`: Returns the product of `x` and `y`.
- Function `int divide(int x, int y)`: Returns the integer part of the `x/y`.
- Function `int min(int x, int y)`: Returns the minimum of `x` and `y`.
- Function `int max(int x, int y)`: Returns the maximum of `x` and `y`.
- Function `int sqrt(int x)`: Returns the integer part of the square root of `x`.

String

This class implements the `String` data type and various string-related operations.

- Constructor `String new(int maxLength)`: Constructs a new empty string (of length zero) that can contain at most `maxLength` characters.
- Method `void dispose()`: Disposes this string.
- Method `int length()`: Returns the length of this string.
- Method `char charAt(int j)`: Returns the character at location `j` of this string.
- Method `void setCharAt(int j, char c)`: Sets the `j`'th element of this string to `c`.
- Method `String appendChar(char c)`: Appends `c` to this string and returns this string.
- Method `void eraseLastChar()`: Erases the last character from this string.

- Method `int intValue()`: Returns the integer value of this string (or at least of the prefix until a non numeric character is found).
- Method `void setInt(int j)`: Sets this string to hold a representation of `j`.
- Function `char backSpace()`: Returns the backspace character.
- Function `char doubleQuote()`: Returns the double quote (“) character.
- Function `char newLine()`: Returns the newline character.

Array

This class enables the construction and disposal of arrays.

- Function `Array new(int size)`: Constructs a new array of the given size.
- Method `void dispose()`: Disposes this array.

Output

This class allows writing text on the screen.

- Function `void moveCursor(int i, int j)`: Moves the cursor to the `j`'th column of the `i`'th row, and erases the character located there.
- Function `void printChar(char c)`: Prints `c` at the cursor location and advances the cursor one column forward.
- Function `void printString(String s)`: Prints `s` starting at the cursor location, and advances the cursor appropriately.
- Function `void printInt(int i)`: Prints `i` starting at the cursor location, and advances the cursor appropriately.
- Function `void println()`: Advances the cursor to the beginning of the next line.
- Function `void backSpace()`: Moves the cursor one column back.

Screen

This class allows drawing graphics on the screen. Column indices start at 0 and are left-to-right. Row indices start at 0 and are top-to-bottom. The screen size is hardware-dependant (over HACK: 256 rows * 512 columns).

- Function `void clearScreen()`: Erases the entire screen.
- Function `void setColor(boolean b)`: Sets the screen color (white=false, black=true) to be used for all further `drawXXX` commands.
- Function `void drawPixel(int x, int y)`: Draws the `(x,y)` pixel.
- Function `void drawLine(int x1, int y1, int x2, int y2)`: Draws a line from pixel `(x1,y1)` to pixel `(x2,y2)`.
- Function `void drawRectangle(int x1, int y1, int x2, int y2)`: Draws a filled rectangle where the top left corner is `(x1, y1)` and the bottom right corner is `(x2,y2)`.

- Function void `drawCircle`(int x, int y, int r): Draws a filled circle of radius r around (x,y)

Keyboard

This class allows reading inputs from the keyboard.

- Function char `keyPressed`(): Returns the character of the currently pressed key on the keyboard; if no key is currently pressed, returns 0.
- Function char `readChar`(): Waits until a key is pressed on the keyboard and released, and then echoes the key to the screen and returns the character of the pressed key.
- Function String `readLine`(String message): Prints the message on the screen, reads the next line (until a newline character) from the keyboard, echoes the line to the screen, and returns its value. This method handles user backspaces.
- Function int `readInt`(String message): Prints the message on the screen, reads the next line (until a newline character) from the keyboard, echoes the line to the screen, and returns the integer until the first non numeric character in the line. This method handles user backspaces.

Memory

This class allows direct access to the main memory.

- Function int `peek`(int address): Returns the value of the main memory at this address.
- Function void `poke`(int address, int value): Sets the value of the main memory at this address to the given value.
- Function Array `alloc`(int size): Allocates the specified space on the heap and returns a reference to it.
- Function void `dealloc`(Array o): De-allocates the given object and frees its memory space.

Sys

This class supports some execution-related services.

- Function void `halt`(): Halts the program execution.
- Function void `error`(int errorCode): Prints the error code on the screen and halts.
- Function void `wait`(int duration): Waits approximately *duration* milliseconds and returns.

4. Writing Jack Applications

Jack is a general-purpose language that can be implemented over different hardware platforms. In the next two chapters we will develop a Jack compiler that ultimately generates Hack code, and thus it is natural to discuss Jack applications in the context of the Hack platform.

Recall that the Hack computer is equipped with a 256 rows by 512 columns screen and a standard keyboard. These I/O devices, along with the Jack classes that drive them, enable the development of interactive programs with a graphical GUI. Figure 13 gives some examples.

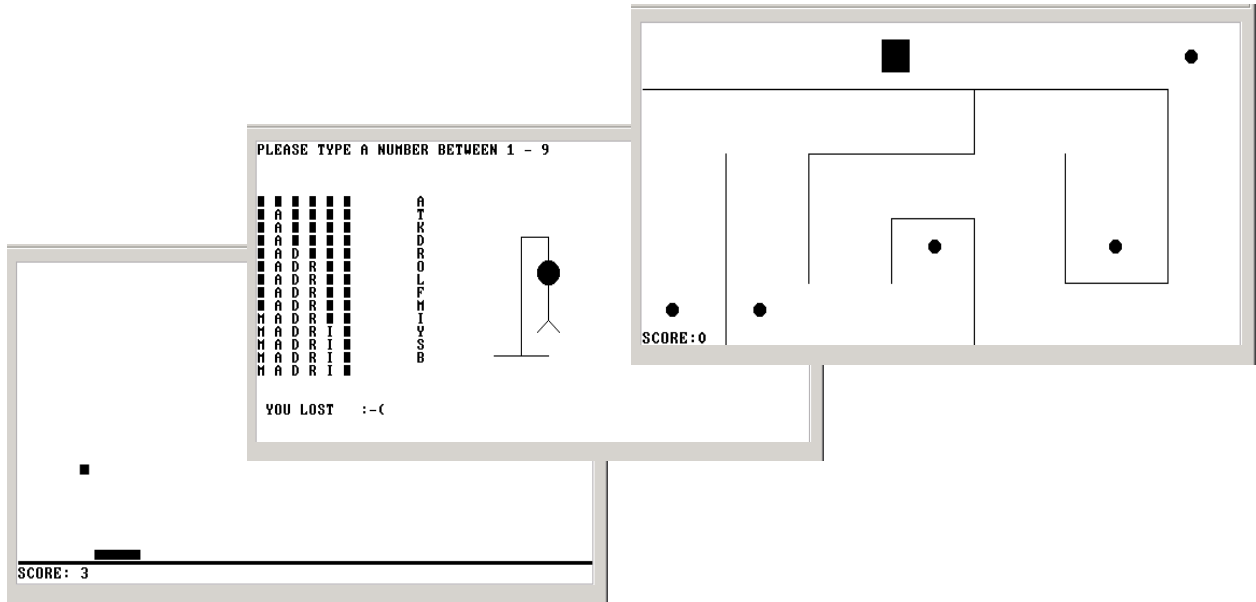


FIGURE 13: Screen shots of three computer games written in Jack, running on the Hack computer. Left to right: a single player “Pong” game, after scoring three points (the author’s record), a “Hangman” game, where the user has to guess the name of the capital city hidden behind the squares (the 1-9 number determines the game level), and a maze game, in which the user scores points by moving the square over the dots.

The Pong game: A ball is moving on the screen randomly, bouncing off the screen “walls”. The user can move a small bat horizontally by pressing the 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.

The Pong game provides a good illustration of Jack programming over the Hack platform. First, it is a non-trivial program, requiring several hundreds lines of Jack code organized in several classes. Second, the program has to carry out some non-trivial mathematical calculations, in order to compute the direction of the ball’s movements. Third, the program must animate the movement of graphical objects on the screen (the bat and the ball), requiring extensive use of the language’s graphics drawing methods. Finally, in order to do all of the above *quickly*, the program must be efficient, meaning that it has to do as few real-time calculations and screen drawing operations as possible.

Other applications: Of course Pong is just one example of the numerous applications that can be written in Jack over the Hack platform. Since the Jack screen resembles that of a cellular telephone, it lends itself nicely to the computer games that one normally finds on cellular telephones, e.g. *Snake* and *Tetris*. In general, the more event- and GUI-driven is the application, the more “catchy” it will be.

Having said that, recall that the Jack/Hack platform is in fact a general-purpose computer. As such, one can use it to develop any application that involves inputs, outputs, and calculations, not necessarily in this order. For example, one can write a program that inputs student names and grades, and then prints the average grade and the name of the student who got the maximal grade, and so on. We now turn to describe how such applications can be planned and developed.

The development of Jack applications over a target platform requires careful planning (as always). In the specification stage, the application designer must consider the physical limitations of the target hardware, and plan accordingly. For example, the physical dimensions of the computer’s screen limits the size of the graphical images that the program can handle. Likewise, one must consider the language’s range of input/output commands, in order to have a realistic appreciation of what can and cannot be done.

Although the language’s capabilities can be extended at will (by modifying its supporting libraries, also written in Jack), readers will perhaps want to hone their programming skills elsewhere. After all, we don’t expect Jack to be part of your life beyond this course. Therefore, it is best to view the Jack/Hack platform as a given environment, and make the best out of it. That’s precisely what programmers do when they write software for embedded devices and dedicated processors that operate in restricted environments. Instead of viewing the constraints imposed by the host platform as a problem, they view it an opportunity to display their resourcefulness and ingenuity. That’s why some of the best programmers in the trade were first trained on primitive computers.

Specification: The specification stage should start with some description of the application’s behavior and, in the case of graphical and interactive applications, some hand-written drawings of typical screens. Using object-oriented analysis and design practices, the designer should then create an object-based architecture of the application. This requires the identification of *classes*, *fields*, and *subroutines*, resulting with a well-defined API (like Program 1).

Implementation: Next, one implements the API in Jack and tests it on the target platform. Following compilation into VM code, the program can be tested in two alternatives ways. Either the VM code can be executed directly on a VM emulator, or it can be translated into Hack code and ran on the hardware platform.

5. Perspective

The Jack programming language is certainly more “clunky” than what you would expect from a modern programming language. However, its basic features and semantic level are not so different from other modern programming languages. Jack is an “object-based” language: supports objects and classes but not inheritance. In this respect it is somewhere between procedural languages (like Pascal or C) and object-oriented languages (like Java or C++). Additionally, Jack’s primitive type system is pretty weak, and moreover, Jack is not “strongly

typed” – i.e. type conformity in assignments and operations is not strictly enforced. All these differences from normal programming languages are in order to simplify compiler construction.

The standard library -- operating system – is quite far from any reasonable operating system. The major deficiencies are the total lack of concurrency -- multi-threading or multi-processing, and the total lack of any permanent storage – files or even communication. The operating system does provide graphic and textual I/O similar to standard ones, although in very basic forms. It provides a standard implementation of strings, as well as standard memory allocation and de-allocation. Additionally, it provides the basic arithmetic operations of multiplication and division that are normally implemented in hardware.

6. Build It

Unlike most projects in the book, this project does not involve building part of the computer’s hardware or software systems. Rather, you have to choose an application of your choice, specify it, and then implement it in Jack on the Hack platform.

Objective: The major objective of this project is to get acquainted with the Jack language, for two purposes. First, you have to know Jack intimately in order to write the Jack compiler in Projects 10 and 11. Second, you have to be familiar with Jack’s supporting libraries in order to write the computer’s operating system in Project 12. The best way to gain this knowledge is to write a Jack application.

Contract: Adopt or invent an application idea, e.g. a simple computer game or some other interactive program. Then specify and build the application.

Steps

1. Download the supplied `os.zip` file and extract its contents to a directory named `project9` on your computer (you may want to give this directory a more descriptive name, e.g. the name of your program). The resulting set of `.vm` files constitutes an implementation of the computer’s operating system, including all the supporting Jack libraries.
2. Write your Jack program (set of one or more Jack classes) using a plain text editor. Put each class in a separate `ClassName.jack` file. Put all these `.jack` files in the same program directory described in step 1.
3. Compile your program using the supplied Jack Compiler. This is best done by applying the compiler to the name of the program directory. This will cause the compiler to translate all the `.jack` classes in that directory to corresponding `.vm` files. If a compilation error is reported, debug the program and re-compile until no error messages are issued.
4. At this point, the program directory should contain three sets of files: (a) your source `.jack` files, (b) a compiled `.vm` file for each one of your source `.jack` files, and (c) the supplied operating system `.vm` files. To test the compiled program, invoke the *VM Emulator* and direct it to load the program by selecting the program directory name. Then run the program. In case of run-time errors or undesired program behavior, fix the program and go to stage 3.

Deliverables: You are expected to deliver a `readme.txt` text file that tells users everything they have to know about using your program, and a zip file containing all your source Jack files. Do not submit any `.vm` files.

10. The Compiler I: Syntax Analysis¹

*"Neither can embellishments of language be found
without arrangement and expression of thoughts,
nor can thoughts be made to shine without the light of language."*

Cicero (106 BC - 43 BC)

This chapter is work in progress. In this chapter we start the process of building a compiler for the Jack high-level language. The process of compilation is usually partitioned into two conceptual parts: syntactic understanding of the program structure, and semantic generation the compiled code. This chapter deals with the first issue, that of *parsing* a program written in the Jack language as to “understand its structure”. The second part, code generation, is the subject of chapter 11.

The concept of “understanding the structure” of a program needs some explanation. When humans reads a program, they immediately see the “structure” of the program: where classes and methods begin and end, what are declarations, what are statements, what are expressions and how they are built, and so on. Notice that this is a complex nested structure: classes contain methods that contain statements that contain expressions, etc. The allowable structure of programs may be formalized, and programming languages today have formal syntax rules, usually given as a “context free language”.

Parsing a program that was written according to these rules means determining the exact correspondence between the program and the syntax rules. This correspondence is usually hierarchal, and may be specified by a “derivation tree” for the program. Compilers often keep an explicit data structure that corresponds to this tree and use this data structure for code generation. Alternatively, they may generate this information implicitly and use it on the fly for code generation. Since in this chapter we do not generate any code yet, we have chosen to explicitly output the parsed structure in a particular format, specifically in XML. This will demonstrate the correct parsing of the program, in a way that is easily displayed in any web browser. In the next chapter we will simply replace the parts of the current program that output the parsing in XML with parts that do actual code generation.

It is worthwhile to note that the same types of syntax rules used for specifying programming languages are also used for specifying the syntax of many other types of files. While most programmers will never need to write a real compiler, it is very likely that they will often need to parse files of some other type with a complex syntax. This parsing will be done in the same way that the parsing of a programming language is done.

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

1. Background

Lexical Analysis

In its plainest syntactic form, a program is simply a sequence of characters, stored in a text file. The first step in the syntax analysis of the program is to group the characters into *words*, also called *tokens*, while ignoring white space and comments. This step is usually called “lexical analysis,” “scanning,” or “tokenizing”. Once a program has been tokenized, the tokens (rather than the characters) are viewed as its basic atoms. Thus the tokens stream becomes the main input of the compiler. Program 1 illustrates the tokenizing of a typical code fragment, taken from a Java or C program.



PROGRAM 1: Lexical Analysis, also called *tokenizing*, converts the input text into a list of tokens. These tokens are then taken to be the elementary atoms from which the program is made.

As Program 1 illustrates, there are several distinct types of tokens: `while` is a keyword; `count` is an identifier; `<=` is an operator, `100` is a constant, and so on. Also notice that white space (blanks and newline characters) is eliminated in the tokenizing process, and so are comments.

In general, each programming language specifies the types of tokens it allows, as well as the exact syntax rules for combining them into programmatic structures. For example, some languages may specify that “++” is an operator, while other languages may not. In the latter case, an expression containing two consecutive + operators will be considered invalid.

Context Free Languages

Once we have lexically analyzed a program into a stream of tokens, we are now faced with the main challenge of parsing it into its formal structure. We first need to consider how the formal syntax of languages is specified. There is a rich theory called “formal languages” that discusses many types of languages, including the formalisms used to specify them. Almost all programming languages, as well as most other formal languages used for describing the syntax of complex files types, use a formalism known as “context free grammars”.

A *context free grammar* is a specification of allowable syntactic elements, and rules for composing them from other syntactic elements. For example, the grammar of the Java language allow us to combine the atoms `100`, `count`, and `<=` into the pattern `count<=100`. In a similar fashion, we can observe that according to the Java grammar, the input pattern `count<=100` is valid, i.e. it is consistent with the language's rules. Indeed, every language has a dual perspective. From a constructive standpoint, the grammar specifies allowable ways to combine *words*, also called *terminals*, into higher-level syntactic elements, called *non-terminals*. From an analytic standpoint, the grammar is also a prescription for doing the reverse: decomposing a given input pattern into non-terminals, lower-level non-terminals, and eventually into terminals that cannot be decomposed further. These terminals correspond to the tokens of the lexical analysis phase.

The syntactic structure of the language -- the context free grammar -- is a set of rules that specify how non-terminals can be derived from other non-terminals and terminals. The grammar may be recursive. There may be more than one possible rule for deriving any particular non-terminal, and the different alternatives are usually indicated using the “|” notation. Grammar 2 gives an example of a small part of the context-free grammar of the C-language.

```

...
statement:  whileStatement
           |  ifStatement
           |  ...          // other statement possibilities follow
           |  '{' statementSequence '}'

whileStatement: 'while' '(' expression ')' statement

ifStatement: ...          // if definition comes here

statementSequence: ' ' // null, i.e. the empty sequence
                  | statement ';' statementSequence

expression: ... // definition of an expression comes here

...          // more definitions follow

```

GRAMMAR 2: A Context Free Grammar is a set of rules that describes the syntactic structure of a language. Here we see part of the C language grammar.

What does Grammar 2 mean? First, the grammar implies that *statement*, *whileStatement*, *ifStatement*, *expression*, and *statementSequence* are non-terminals, whereas “while”, ‘{’, ‘}’, and ‘;’ are terminals. Further, the grammar implies that a *statement* in the C language may be one of several forms, including a *while* statement, an *if* statement, other possibilities not shown here for lack of space, and any sequence of statements enclosed in curly brackets. The resulting grammar is highly recursive, allowing nested structures like the following example:

```

while (some expression) {
    some statement;
    some statement;
    while (some expression) {
        while (some expression)
            some statement;
        some statement;
    }
}

```

Parsing: The act of checking whether a grammar “accepts” an input text as valid (according to the grammar rules) is called *parsing*. As a side effect of the parsing process, the entire syntactic structure of the input text is uncovered. Since the grammar rules are hierarchical, the result is a tree-oriented data structure, called *parse tree* or *derivation tree* (whether the tree is stored in memory or recognized on-line is a different issue that will be addressed later). For example, if we apply Grammar 2 to the tokenized version of Program 1, we will obtain the parse tree depicted in Figure 3.

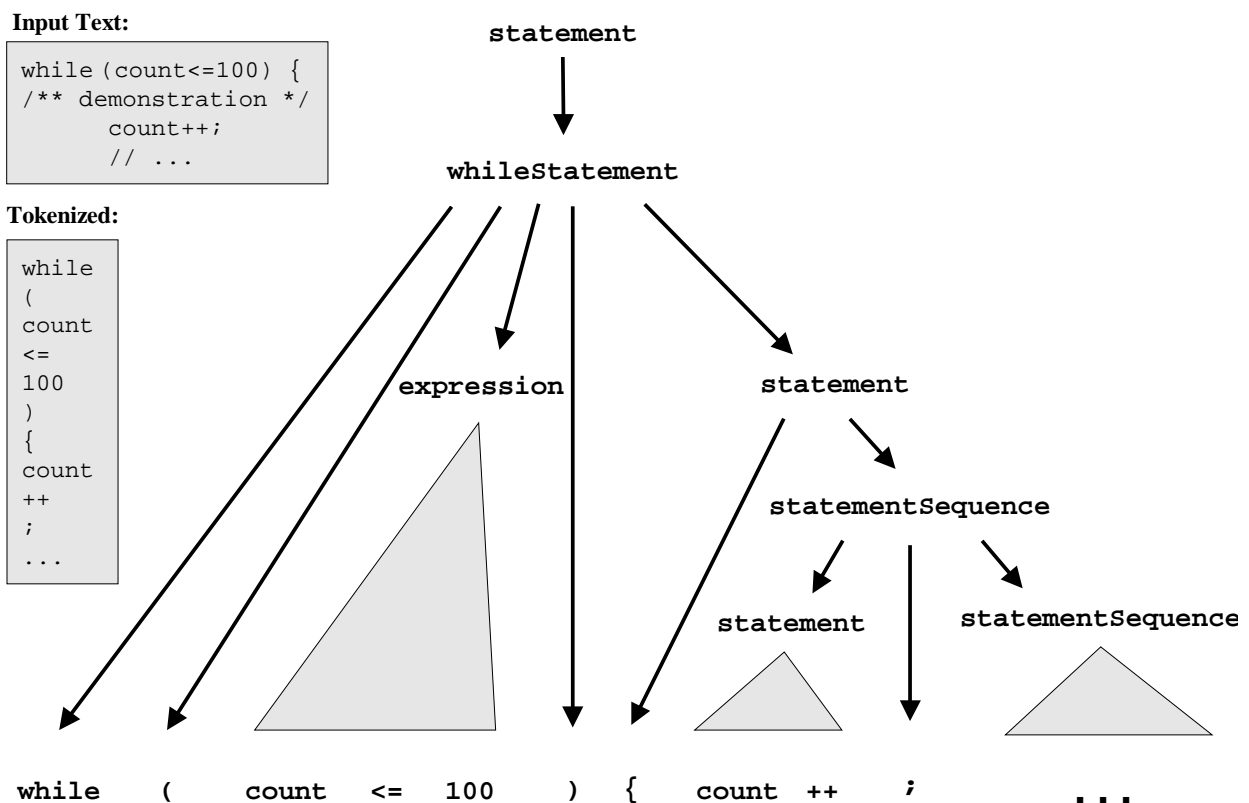


FIGURE 3: Parse tree of Program 1 according to Grammar 2. Solid triangles represent lower-level parse trees. The input of the parsing process is the tokenized version of the program.

Recursive Descent Parsing

The last section ended with a description of a *parse tree*. We now turn to describe the algorithms that can be used to construct such trees from given input programs, according to the syntax rules of a given language. There are general algorithms that can do that for any context free language. The general algorithms are not efficient enough for practical use on very long programs, and there are more efficient parsing algorithms that apply to certain restricted classes of context free languages, classes that contain the syntax rules of essentially all program languages. These more efficient algorithms usually run “online” – they parse the input as they read it, and do not have to keep the entire input program in memory. There are essentially two types of strategies for this parsing. The simple strategy works top-down, and this is the one we present here. The more advanced algorithms work bottom-up, and are not described here since they require a non-trivial elaboration of theory.

The top-down approach to parsing, also called *recursive descent parsing*, parses the input stream recursively, using the nested structure prescribed by the language grammar. Let us consider how a *parser* program that implements this strategy can be constructed. For every non-terminal building block of the language, we can equip the parser with a recursive procedure designed to parse that non-terminal. If the non-terminal consists of terminal atoms only, the procedure will simply read them. Otherwise, for every non-terminal building block, the procedure will recursively call the procedure designed to parse the non-terminal. The process will continue recursively, until all the terminal atoms have been reached and read.

For example, suppose we have to write a recursive descent parser that implements Grammar 2. Since the grammar has five derivation rules, the parser implementation can consist of five major procedures: `parseStatement()`, `parseWhileStatement()`, `parseIfStatement()`, `parseStatementSequence()`, and `parseExpression()`. The parsing logic of these procedures should follow the syntactic patterns found in the corresponding grammar rules. Thus `parseStatement()` should probably start its processing by determining what is the first token. Having established the token’s identity, the procedure could determine which statement we are in, and then call the parsing procedure associated with this statement type.

For example, if the input stream were Program 1, the procedure will establish that the first token is `while`, and then call the procedure `parseWhileStatement()`. According to the corresponding grammar rule, this procedure should next attempt to read the terminals “`while`” and ““(”, and then call `parseExpression()` to parse the non-terminal *expression*. After `parseExpression()` would return (having read and parsed the “`count<=100`” sequence in our example), the grammar dictates that `parseWhileStatement()` should continue parsing the remainder of the while statement. In particular, the grammar states that it should attempt to read the terminal “)” and then recursively call `parseStatement()` to parse the non-terminal statement. This call would continue recursively, until at some point only terminal atoms are read.

LL(0) grammars: Recursive parsing algorithms are simple and elegant. If you will think about them, you will realize that the only thing that complicates matters is the existence of several alternatives for parsing non-terminals. For example, when `parseStatement()` attempts to parse a statement, it does not know in advance whether this statement is a while-statement, an if-statement, a curly-bracket enclosed statement list, and so on. The span of possibilities is determined by the underlying grammar, and in some cases it is easy to tell which alternative we are in. For example, consider Grammar 2. If the first token is “`while`”, it is clear that we are faced with a while

statement, since this is the only alternative that starts with a “while” token. This observation can be generalized as follows: whenever a non-terminal has several alternative derivation rules, the first token specifies without ambiguity which rule to use. Grammars that have this property are called $LL(0)$ grammars, and they can be handled simply and neatly by recursive descent algorithms.

When the first token does not suffice to resolve the element’s type, it is possible that a “look ahead” to the next token will settle the dilemma. Such parsing can obviously be done, but as we need to look ahead at more and more tokens down the stream, things start getting complicated. The Jack language grammar, which we now turn to present, is “almost” $LL(0)$, and thus it can be handled rather simply by a recursive descent algorithm. The only exception is the parsing of expressions, where just a little look ahead is necessary.

2. Specification

In this chapter we will write a syntax analyzer for the Jack programming language. In the next chapter we will add the functionality of code generation to the syntax analyzer, and obtain a full compiler. The main purpose of the syntax analyzer is to read a Jack program and “understand” its structure according to the Jack language syntax specification. The meaning of “understanding” is that the program “knows” at each point the meaning of what it is reading: an expression, a statement, a variable name, etc. It has to have this knowledge in a complete recursive sense. This is what will be needed for later enabling the code generation.

One way to demonstrate that the analyzer has “understood” the programmatic structure of the input is to have it print the text in a way that provides a visual image of the program structure. Therefore, while syntax analyzers are normally not stand-alone programs, we define our syntax analyzer as having a specific output: an XML description of the program. Thus the syntax analyzer you build here will output an XML file whose structure reflects the structure of the underlying program. In the next chapter you will replace the parts of the program that output the XML code with software that generates executable VM code instead.

Usage: The Jack syntax analyzer accepts a single command line argument that specifies either a file name or a directory name:

```
prompt> JackCompiler source
```

If *source* is a file name of the form `xxx.jack`, the analyzer compiles it into a file named `xxx.xml`, created in the same folder in which the input `xxx.jack` is located. If *source* is a directory name, all the `.jack` files located in this directory are compiled. For each `xxx.jack` file in the directory, a corresponding `xxx.xml` file is created in the same directory.

Jack Language Syntax

The functional specification of the Jack language was given in chapter 9. We now turn to give a formal specification of the Jack language *syntax*, using a context free grammar. The grammar is based on the following conventions:

- **'xxx'** quoted boldface is used for characters that appear verbatim (“terminals”)
- xxx regular typeface is used for names of language constructs (“non-terminals”)
- () parentheses are used for grouping of language constructs
- x | y means that either x or y can appear
- x? means that x appears 0 or 1 times
- x* means that x appears 0 or more times

Input: The input to the Jack syntax analyzer is simply a stream of characters. This stream should be tokenized into a stream of tokens according to the rules specifying the lexical elements in the table. These tokens may be separated by an arbitrary amount of white space (space and newline characters) and comments, which are ignored. Comments are of the standard formats `/* comment until closing */`, `/** API comment */`, and `// comment to end of line`.

The complete language grammar is given in Grammar 4.

Lexical elements	There are five types of lexical elements in the Jack language:
keyword:	'class' 'constructor' 'function' 'method' 'field' 'static' 'var' 'int' 'char' 'boolean' 'void' 'true' 'false' 'null' 'this' 'let' 'do' 'if' 'else' 'while' 'return'
symbol:	{' ' '(' ')' '[' ']' '.' ',' ';' '+' '-' '*' '/' '%' ' ' '<' '>' '=' '~' }
integerConstant:	a decimal number in the range 0 .. 32767
stringConstant:	'" sequence of ASCII characters not including double quote or newline "'
identifier:	sequence of letters, digits, and underscore ('_') not starting with a digit
Program structure:	A program is a collection of classes, each appearing in a separate file. The compilation unit is a class, and is given by the following context free syntax:
class:	'class' className '{' classVarDec* subroutineDec* '}'
classVarDec:	('static' 'field') type varName (',' varName)* ';'
type:	'int' 'char' 'boolean' className
subroutineDec:	('constructor' 'function' 'method') ('void' type) subroutineName '(' parameterList ')' subroutineBody
parameterList:	((type varName) (',' type varName)*)?
subroutineBody:	{' varDec* statements '}'
varDec:	'var' type varName (',' varName)* ';'
className:	Identifier
subroutineName:	Identifier
varName:	Identifier
Statements	
statements:	statement*
statement:	letStatement ifStatement whileStatement doStatement returnStatement
letStatement:	'let' varName ('[' expression ']')? '=' expression ';'
ifStatement:	'if' '(' expression ')' '{' statements '}' ('else' '{' statements '}')?
whileStatement:	'while' '(' expression ')' '{' statements '}'
doStatement:	'do' subroutineCall ';'
returnStatement:	'return' expression? ';'
Expressions:	
expression:	term (op term)*
term:	integerConstant stringConstant keywordConstant varName varName '[' expression ']' subroutineCall '(' expression ')' unaryOp term
subroutineCall:	subroutineName '(' expressionList ')' (className varName) '.' subroutineName '(' expressionList ')'
expressionList:	(expression (',' expression)*)?
op:	'+' '-' '*' '/' '%' ' ' '<' '>' '='
unaryOp:	'-' '~'
keywordConstant:	'true' 'false' 'null' 'this'

GRAMMAR 4: Complete grammar of the Jack language

XML Output Format

The output of the syntax analyzer should be an XML description of the program. Figure 5 gives a detailed example. Basically, the analyzer has to recognize two major types of language constructs: terminal elements, and non-terminal elements. These constructs are handled as follows.

Non-terminals: Whenever a non-terminal element of type xxx of the language is encountered, the analyzer should generate the output:

```
<xxx>  
    recursive code for the body of the xxx element  
</xxx>
```

Where xxx is one of the following (and only the following) non-terminals of the Jack grammar:

- class, classVarDec, subroutineDec, parameterList, subroutineBody, varDec
- statements, whileStatement, ifStatement, returnStatement, letStatement, doStatement
- expression, term, expressionList

Terminals: Whenever a terminal element of type xxx of the grammar is encountered, the following output should be generated:

```
<xxx> terminal </xxx>
```

Where xxx is one of the five terminals specified in the “lexical elements” part of the Jack grammar: keyword, symbol, integerConstant, stringConstant, identifier.

```

Class Bar {
  method Fraction foo(int y) {
    var int temp; // a variable
    let temp = (xxx+12)*-6 ;
    ...
  }
}

```

```

<class>
  <keyword> class </keyword>
  <identifier> Bar </identifier>
  <symbol> { </symbol>
  <subroutineDec>
    <keyword> method </keyword>
    <identifier> Fraction </identifier>
    <identifier> foo </identifier>
    <symbol> ( </symbol>
    <parameterList>
      <keyword> int </keyword>
      <identifier> y </identifier>
    </parameterList>
    <symbol> ) </symbol>
    <subroutineBody>
      <symbol> { </symbol>
      <varDec>
        <keyword> var </keyword>
        <keyword> int </keyword>
        <identifier> temp </identifier>
        <symbol> ; </symbol>
      </varDec>
      <statements>
        <letStatement>
          <keyword> let </keyword>
          <identifier> temp </identifier>
          <symbol> = </symbol>
          <expression>
            <term>
              <symbol> ( </symbol>
              <expression>
                <term>
                  <identifier> xxx </identifier>
                </term>
              <symbol> + </symbol>
              <term>
                <integerConstant> 12 </integerConstant>
              </term>
            </expression>
            <symbol> ) </symbol>
          </term>
          <symbol> * </symbol>
          <term>
            <symbol> - </symbol>
            <term>
              <integerConstant> 6 </integerConstant>
            </term>
          </term>
        </expression>
        <symbol> ; </symbol>
      </letStatement>
      ...
    </subroutineBody>
  </subroutineDec>
  </symbol> } </symbol>

```

FIGURE 5: Analyzer input (top) and output (bottom)

3. Implementation

We suggest to arrange the implementation of the syntax analyzer in three modules:

- `JackAnalyzer`: a main driver that organizes and invokes everything;
- `JackTokenizer`: a tokenizer;
- `CompilationEngine`: a recursive top-down syntax analyzer.

These modules handle the syntax of the language. In the next chapter we will extend this implementation with two additional modules that handle the language's semantics: a *symbol table* and a *VM-code writer*. This will complete the construction of a full compiler for the Jack language.

JackAnalyzer

The analyzer program operates on a given *source*. If *source* is a file name of the form `Xxx.jack`, the analyzer compiles it into a file named `Xxx.xml`, created in the same folder in which the input `Xxx.jack` is located. If *source* is a directory name, all the `.jack` files located in this directory are compiled. For each `Xxx.jack` file in the source directory, the analyzer creates a corresponding `Xxx.xml` file in the same directory. The logic is as follows:

For each source `Xxx.jack` file:

1. Create a *tokenizer* from the `Xxx.jack` file
2. Open an `Xxx.xml` file and prepare it for writing
3. Compile(INPUT: *tokenizer*, OUTPUT: *output file*)

Where *output file* refers to the `Xxx.xml` file.

JackTokenizer

The tokenizer removes all comments and white space from the input stream and breaks it into Jack-language tokens, as specified in the Jack grammar.

Routine	Arguments	Returns	Function
Constructor	input file / stream	--	Opens the input file/stream and gets ready to tokenize it
hasMoreTokens	--	Boolean	do we have more tokens in the input?
advance	--	--	gets the next token from the input and makes it the current token. This method should only be called if <i>hasMoreTokens()</i> is true. Initially there is no current token..
tokenType	--	KEYWORD, SYMBOL, IDENTIFIER, INT_CONST, STRING_CONST	returns the type of the current token
keyWord	--	CLASS, METHOD, FUNCTION, CONSTRUCTOR, INT, BOOLEAN, CHAR, VOID, VAR, STATIC, FIELD, LET, DO, IF, ELSE, WHILE, RETURN, TRUE, FALSE, NULL, THIS	returns the keyword which is the current token. Should be called only when <i>tokenType()</i> is KEYWORD.
symbol	--	char	returns the character which is the current token. Should be called only when <i>tokenType()</i> is SYMBOL.
identifier	--	string	returns the identifier which is the current token. Should be called only when <i>tokenType()</i> is IDENTIFIER
intVal		int	returns the integer value of the current token. Should be called only when <i>tokenType()</i> is INT_CONST
stringVal		string	returns the string value of the current token, without the double quotes. Should be called only when <i>tokenType()</i> is STRING_CONST.

CompilationEngine

This module effects the actual compilation into XML form. It gets its input from a `JackTokenizer` and writes its parsed XML structure into an output file/stream. This is done by a series of `compilexxx()` methods, where `xxx` is a corresponding syntactic element of the Jack grammar. The contract between these methods is that each `compilexxx()` method should read the syntactic construct `xxx` from the input, `advance()` the tokenizer exactly beyond `xxx`, and output the XML parsing of `xxx`. Thus, `compilexxx()` may only be called if indeed `xxx` is the next syntactic element of the input.

In the next chapter, this module will be modified to output the compiled code rather than XML.

Routine	Arguments	Returns	Function
Constructor	Input stream/file Output stream/file	--	creates a new compilation engine with the given input and output. The next method called must be <code>compileClass()</code> .
<code>CompileClass</code>	--	--	compiles a complete class.
<code>CompileClassVarDec</code>	--	--	compiles a static declaration or a field declaration.
<code>CompileSubroutine</code>	--	--	compiles a complete method, function, or constructor.
<code>compileParameterList</code>	--	--	compiles a (possibly empty) parameter list, not including the enclosing “()”.
<code>compileVarDec</code>	--	--	compiles a var declaration.
<code>compileStatements</code>	--	--	compiles a sequence of statements, not including the enclosing “{ }”.
<code>compileDo</code>	--	--	Compiles a do statement
<code>compileLet</code>	--	--	Compiles a let statement
<code>compileWhile</code>	--	--	Compiles a while statement
<code>compileReturn</code>	--	--	compiles a return statement.
<code>compileIf</code>	--	--	compiles an if statement, possibly with a trailing else clause.
<code>CompileExpression</code>	--	--	compiles an expression.

<code>CompileTerm</code>	--	--	compiles a <i>term</i> . This method is faced with a slight difficulty when trying to decide between some of the alternative rules. Specifically, if the current token is an identifier, it must still distinguish between a variable, an array entry, and a subroutine call. The distinction can be made by looking ahead one extra token. A single look-ahead token, which may be one of “[“,“(“,“.”, suffices to distinguish between the three possibilities. Any other token is not part of this term and should not be advanced over.
<code>CompileExpressionList</code>	--	--	compiles a (possibly empty) comma-separated list of expressions.

4. Perspective

In this chapter we have side-stepped almost all of the formal language theory studied in a typical compilation course. We were able to do this by choosing a very simple syntax for Jack that could be easily compiled using recursive descent techniques. In particular, our grammar for expressions did not mandate the usual operator precedence (e.g. of multiplication over addition). This avoided the need for bottom-up parsing of “LR” languages, usually used in other programming languages.

In reality, programmers rarely write syntax analyzers by hand. Instead, they use, so called, “compiler-compilers” (such as “yacc”) utilities. These programs receive as input a context free grammar, and produce as output a syntax analysis code for this grammar. Following the “show me” spirit of this book, we have chosen not to use such black boxes in the implementation of our compiler.

5. Build it

In this project you will build a syntactic analyzer for the Jack language. In the next chapter, we will extend this analyzer into a full-scale Jack compiler.

Objective: Develop a syntactic analyzer that parses Jack programs according to the Jack grammar. The output of the analyzer should be written in XML format, following the example given in Figure 5.

Resources: The main tool that you need is the programming language in which you will implement the analyzer. You will also need the supplied `TextComparer` utility, which allows comparing the output files generated by your analyzer to the compare files supplied by us. If you want to inspect the XML code generated by the analyzer, you will also need an XML viewer (any standard Web browser should do the job).

Contract: Write the Jack analyzer program in two stages, as described below. Use it to parse all the .jack files mentioned below. For each source .jack file, your analyzer should generate an .xml output file. The generated files should be identical to the supplied .xml compare-files.

Test Programs

We supply three test programs, as follows.

Square Dance A simple interactive game that will be used to test your compiler in both projects 10 and 11. Although the details of the game are irrelevant to the compilation process, we describe it briefly. *Square Dance* is a trivial “game” that enables moving a black square around the screen using the keyboard’s four arrow keys. While moving, the size of the square can be increased and decreased by pressing the “z” and “x” keys, respectively. To quit the game, press the “q” key. The game implementation is organized in three classes:

- ❑ Class Main: Initializes a new game and starts it;
- ❑ Class Square: Implements an animated square. A square object has a screen location and size properties, and methods for drawing, erasing, moving, and size changing;
- ❑ Class SquareGame: Runs the game according to the game rules.

We provide three sets of test files for this program:

- ❑ Input source code: Main.jack, Square.jack, SquareGame.jack
- ❑ Tokenizer output (compare files): MainT.xml, SquareT.xml, SquareGameT.xml
- ❑ Analyzer output (compare files): Main.xml, Square.xml, SquareGame.xml

Expressionless Square Dance: An identical copy of the *Square Dance* game, except that each expression in the latter is replaced with a single identifier (a variable name in scope). This version of the program is especially useful in the project’s second stage, in which it is advised to first implement a *Parser* that handles everything except expressions. The replacement of the expressions with variables required the introduction of some illegal variable castings into the source code, and so this version of the game cannot be compiled using the Jack Compiler. Still, it follows all the Jack grammar rules. The provided test files have the same names as those of the *Square Dance* program.

Array test: A single-class Jack program that computes the average of a user-supplied sequence of integers. This program uses some array notation not used in the *Square Dance* program, and therefore it is recommended to test it only after successful testing of the latter. The provided test files include the source code (Main.jack), the compare file for the *Tokenizer* output (MainT.xml) and the compare file for the *Parser* output (Main.xml).

Implementation Tips

- ❑ Since the output files that your tokenizer and analyzer will generate will have the same names and extensions as those of the supplied compare files, we suggest putting them in separate directories.

- Since each one of the test programs focuses on different aspects of the Jack language, it is recommended to perform the tests in the following order: *Expressionless Square Dance*, then *Square Dance*, then *Array test*.
- All the source test files are written in Jack. If you want, you can compile the *Square* and *Array* programs using the supplied Jack compiler, and then run them on the supplied VM emulator. These activities are completely irrelevant to the analyzer implementation, but they serve to highlight the fact that the test programs are not just plain text (although this is perhaps the best way to think about them in the context of this project).

Stage 1: Tokenizer

First, implement a Jack tokenizer. In order to test this stage, have your tokenizer output an XML file describing the list of the parsed tokens. When applied to a text file containing Jack code, the tokenizer should produce a list of tokens, each printed in a separate line along with its classification: *symbol*, *keyword*, *identifier*, *integer constant*, or *string constant*. The classification should be recorded using XML tags. For example, consider the text:

```
let x=5+yy; let city="Paris";
```

This input should generate the following output:

```
<keyword> let </keyword>
<identifier> x </identifier>
<symbol> = </symbol>
<integerConstant> 5 </integerConstant>
<symbol> + </symbol>
<identifier> yy </identifier>
<symbol> ; </symbol>
<keyword> let </keyword>
<identifier> city </identifier>
<symbol> = </symbol>
<stringConstant> Paris </stringConstant>
<symbol> ; </symbol>
```

Note that the tokenizer throws away the double quote characters. That's OK.

A slight difficulty, and a solution: Four of the symbols used in the Jack language (<, >, ", &) are also used for XML markup, and thus they cannot appear as data in XML files. To solve the problem, have your tokenizer output these tokens as `<`, `>`, `"`, and `&`, respectively. For example, in order for the text "`<symbol> & </symbol>`" to be displayed in an XML viewer, the source XML should be written as "`<symbol> & </symbol>`".

Testing: For each source file `xxx.jack`, have your tokenizer give the output file the name `xxxT.xml`. For each one of the three test programs, apply your tokenizer to every class file in the test program. This should generate an `.xml` output file for each one of the source `.jack` files. Next, use the supplied `TextComparer` utility to compare the generated output to the supplied `.xml` compare files.

Stage 2: Parser

Next, implement the *Compilation Engine* (also referred to as *Parser*). Write each method of the engine, as specified in the API, and make sure that it emits the correct XML output.

Implementation tips:

- ❑ The indentation of XML code is only for readability. XML viewers and the supplied *TextComparer* utility ignore white space.
- ❑ Note that conceptually speaking, the output of the tokenizer is embedded within the XML output. In other words, the parser builds the language “super structure” on top of the terminal tokens.
- ❑ You may want to start by writing a parser that can handle everything except expressions. For example, assume that the only expressions that the input source code can contain may be single identifiers, and handle everything else. Next, extend the parser to handle expressions as well.

Testing: For each source file `xxx.jack`, have your parser give the output file the name `xxx.xml`. For each one of the three test programs, apply your parser to every class file in the test program. This should generate an `.xml` output file for each one of the source `.jack` files. Next, use the supplied *TextComparer* utility to compare the generated output to the supplied `.xml` compare files.

11. The Compiler II: Code Generation¹

“The syntactic component of a grammar must specify, for each sentence, a deep structure that determines its semantic interpretation...”

Noam Chomsky (b. 1928), U.S. mathematical linguist

(This chapter is work in progress). In this chapter we complete the development of the Jack compiler. The overall compiler is based on two modules: the VM backend developed in chapters 7 and 8, and the Syntax Analyzer and Code Generator developed in chapters 10 and 11, respectively. Although the second module seems to consist of two separate sub-modules, they are usually combined into one program, as we will do in this chapter. Specifically, in chapter 10 we built a Syntax Analyzer that “understands” -- parses -- source Jack programs. In this chapter we extend this program into a full-scale compiler that converts each “understood” Jack operation and construct into equivalent series of VM operations on equivalent VM constructs.

1. Background

A program is composed of operations that manipulate data. When we compile a program into a lower level language we must first consider how the data items are mapped into the lower level language and then how each possible operation is translated into a sequence of low-level operations.

1.1. Mapping of data items

Symbol Table

A typical high-level program contains many identifiers. Whenever the compiler encounters any such identifier, it needs to know what it stands for. Is it a variable name, a class name, or a function name? If it's a variable, is it a field of an object, or an argument of a function? What type of variable is it -- an integer, a string, or some other type? The compiler must resolve these questions in order to map the construct that the identifier represents onto a construct in the target language. For example, consider a C function that declares a local variable named `sum` as a `double` type. If we translate this program into the machine language of some 32-bit computer, the `sum` variable will have to be mapped on a pair of two consecutive addresses, say `RAM[3012]` and `RAM[3013]`. Thus, whenever the compiler will encounter high-level statements involving this identifier, e.g. `sum+=i` or `printf(sum)`, it will have to generate machine language instructions that operate on `RAM[3012]` and `RAM[3013]` instead.

We see that in order to generate target code correctly, the compiler must keep track of all the identifiers introduced by the source code. For each identifier, we must record what the identifier stands for in the source language, and on which construct it is mapped in the target language. This information is usually recorded in a “housekeeping” data structure called *symbol table*.

¹ From *The Elements of Computing Systems*, Nisan & Schocken, MIT Press, forthcoming in 2004, www.idc.ac.il/csd

Whenever a new identifier is encountered in the source code for the first time (e.g. in variable declarations), the compiler adds its description to the table. Whenever an identifier is encountered elsewhere in the program, the compiler consults the symbol table to get all the information needed for generating the equivalent code in the target language.

The basic symbol table solution is complicated slightly due to the fact that most languages allow different parts of the program to use the same identifiers for different purposes. For example, two C functions may declare a local variable named x for two completely different purposes. The programmer is allowed to re-use such symbols freely in different program units, since the compiler is clever enough to map them on completely different objects in the target language, as implied by the program's context (and consistent with the programmer's intention). Specifically, in most languages each identifier has a well defined *scope*, i.e. the region of the program in which the identifier is recognized. Whenever the compiler encounters an identifier x in a program, it treats x as the one currently in scope, and generate the appropriate code accordingly. The complication in handling different scopes comes from the fact that they can usually be nested within each other. The convention in most languages is that inner-scoped definitions always hides more outer-scoped ones.

Thus in addition to all the relevant information that must be recorded about each identifier, the symbol table must also reflect in some way the identifier's scope. The classic data structure for this purpose is a list of *hash tables*, each reflecting a single scope nested within the next one in the list. When the compiler fails to find the identifier in the table of the current scope, it looks it up in the next table, from inner scopes outward. Thus if x appears undeclared in a certain code segment (e.g. a method), it may be that x is declared in the code segment that owns the current segment (e.g. a class).

To sum up, depending on the scoping rules of the compiled language, the symbol table can be implemented as a list of two or more hash tables.

Allocation of Variables

One of the basic challenges faced by every compiler is how to map the various types of variables of the source program onto the memory of the target platform. This is not a trivial task. First, different variable *types* require different amounts of memory, so the mapping is not one-to-one. Second, different *kinds* of variables have different life cycles. For example, a single copy of each static variable should be kept "alive" for the complete duration of the program. In contrast, each object instance of the class should have a different copy of all its instance variables (*fields*). Also, each time a function is being called, a new copy of its local variables must be created -- a need which is clearly seen in recursion. In short, memory allocation of variables is an intricate task.

That's the bad news. The good news is that we have already handled these difficulties. The VM that we created in Chapters 8-9 has built-in mechanisms for representing and handling the standard kinds of variables needed by high-level languages: static, local, arguments, and fields of objects. All the allocation and manipulation details were already handled at the VM level. Recall that this functionality was not achieved easily. In fact, we had to work rather hard to create a VM implementation that maps all these constructs and behaviors on a flat RAM structure and a primitive instruction set, respectively. Yet this effort was worth our while: for any given

language L , any L -to-VM compiler is now completely relieved from low-level memory management; all it has to do is map source constructs on respective VM constructs – at this point a rather simple translation task. Further, any improvement in the way the VM implementation manages memory will immediately affect any compiler that depends on it. That’s why it pays to develop efficient VM implementations and continue to improve them down the road.

Arrays

Compilers usually implement arrays as sequences of consecutive memory locations. The array name is usually treated as a pointer to the beginning of the array’s allocated memory block. In some languages (e.g. Pascal), the entire memory space is allocated when the array is declared. In other languages (e.g. Java), the array declaration results in the allocation of a single pointer only. The array proper is created in memory later, when the array is explicitly constructed during the program’s execution. This type of *dynamic memory allocation* is done from the heap, using the memory management services of the operating system. Figure 1 offers a snapshot of the memory organization of a typical array.

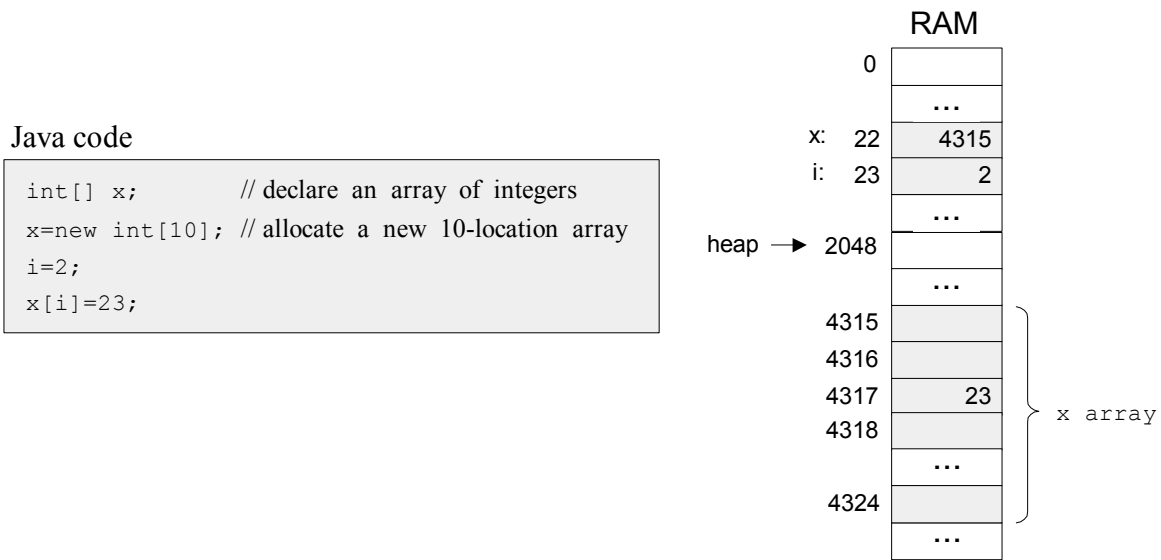


FIGURE 1: Array creation and manipulation. All the addresses in the example were chosen arbitrarily (except that in the Hack platform, the heap indeed begins at address 2048). Note that the basic operation illustrated is $*(x+i)=23$.

Thus storing the value 23 in the i 'th location of array x can be done by the following pointer arithmetic:

```

push x
push i
+
pop addr    // at this point addr points to x[i]
push 23
pop *addr   // store the topmost stack element in RAM[addr]
```

Explanation: The fact that the first four pseudo-commands make variable $addr$ point to the desired array location should be evident from Figure 1. In order to complete the storage operation, the target language must be equipped with some sort of an indirect addressing mechanism. Specifically, instead of storing a value in some memory location y , we need to be able to store the value in the memory location whose address is the current contents of y . In the example above, this operation is carried out by the “pop *addr” pseudo-command. Different

virtual machines feature different ways to accomplish this indirect addressing task. For example, the VM built in chapters 7-8 handles indirect addressing using its `pointer` and `that` segments.

Objects

Object-oriented languages allow the programmer to encapsulate data and the code that operates on the data within programming units called *objects*. This level of abstraction does not exist in low-level languages. Thus, when we translate code that handles objects into a primitive target language, we must handle its underlying data and code explicitly. This will be illustrated in the context of Program 2.

```
/** A Bank Account */
class BankAccount {

    static int sysID;

    // Fields:
    int id;
    String owner;
    int balance;

    private int nextID() {
        return ++sysID;
    }

    /** construct a new bank account with 0 balance */
    public BankAccount(String name) {
        id = nextID();
        owner = name;
        balance = 0;
    }

    /** deposit money in this bank account*/
    public void deposit(int amount) {
        balance = balance + amount;
    }

    // more methods come here

    public static void main(String args[]) {
        BankAccount joeAcct;
        sysID = 0;
        ...
        joeAcct = new BankAccount("joe");
        joeAcct.deposit(5000);
        ...
    }
    ...
} // BankAccount
```

PROGRAM 2

Object data (construction): The data kept by each object instance is essentially a list of fields. As with array variables, when an object-type variable is declared, the compiler typically only allocates a reference (pointer) variable. The memory space for the object proper is allocated only when the object is created via a call to the class constructor. The space for the new object must ultimately be allocated by the operating system that must provide some service like “`alloc(size)`”

that finds a free memory block of the required size and returns a pointer to its base. When compiling a constructor like `BankAccount(String name)`, the compiler generates code that (i) requests the operating system to find a memory block to store the new object, and (ii) sets a pointer to the base of the allocated block to be called, within the constructor, “this”. From this point onward, the object’s fields can be accessed linearly, using an index relative to its base. Thus statements like `let owner=b` can be easily compiled, as we now turn to explain.

Object data (usage): The previous paragraph focused on how the compiler generates code that creates new objects. We now describe how the compiler handles commands that manipulate the data encapsulated in existing objects. For example, consider the handling of a statement like `let balance=balance+amount` within the method `deposit`. First, an inspection of the symbol table will tell the compiler that `amount` is an argument, while `balance` is the 2nd field of the `BankAccounts` class (starting the field count from 0). Using this information, the compiler can generate code effecting the operation `*(this+2) = *(this+2) + (argument 1)`. Of course the generated code will have to accomplish this operation using the target language.

Object code: The encapsulation of methods within object instances is a convenient abstraction that is not implemented for real. Unlike the fields data, of which different copies are indeed kept for each object instance, only one copy of each method is actually kept at the target code level. Thus the trick that stages the code encapsulation abstraction is to have the compiler force the method to always operate on the desired object. A standard simple way to handle this is by passing the object reference as a “hidden” argument. I.e. a method call like `xxx.m(y)` is actually compiled as if it were written as `m(xxx,y): “push xxx, push y, call m”`. A syntactic detail that has to be handled is making sure that the called method `m` is really the one defined for `xxx`’s class. In object-oriented languages this determination must be done in run-time due to the possibility of method overriding in a sub-class. When run-time typing is out of the picture, e.g. in languages like Jack, or if `m` was somehow declared not to be virtual, then all that is needed is to ensure that the called method `m` belongs to the correct class. E.g. in our example, if `xxx` was a variable of class `xxx`, then we may call the method named `xxx.m`.

1.2. Command translation

We now turn to describe how commands are translated. There are two elements to consider: expression evaluation and flow control.

Expression Evaluation

How should we generate code for evaluating high level expressions like `x+f(2,y,-z)*5`? First, we must “understand” the syntactic structure of the expression, e.g. convert it into a parse tree like the one depicted in Figure 3. This was already handled in chapter 10. Next, we traverse the tree and generate the target code. Clearly, the choice of the code generation algorithm will depend on the target language into which we are translating.

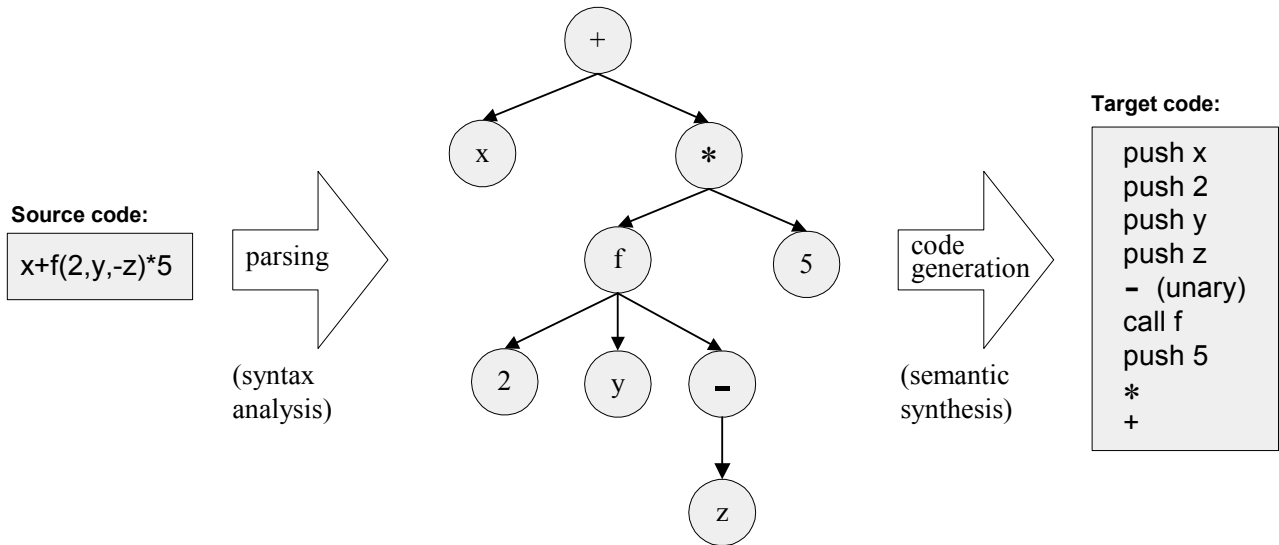


FIGURE 3: Code generation for expressions is based on a syntactic understanding of the expression, and can be easily accomplished by recursive manipulation of the expression tree. Note that the parsing stage was carried out in Chapter 10.

The strategy for translating expressions into a stack-based language is based on a postfix (depth-first) traversal of the corresponding expression tree. This simple strategy is described in Algorithm 3.

Code(exp):

if exp is a number n then output "push n "
 if exp is a variable v then output "push v "
 if exp = (exp1 op exp2) then Code(exp1); Code(exp2) ; output "op"
 if exp = op(exp1) then Code(exp1) ; output "op"
 if exp = f(exp1 ... expN) then Code(exp1) ... Code(expN); output "call f"

ALGORITHM 4: A recursive postfix traversal algorithm for evaluating an expression tree by generating commands in a stack-based language.

The reader can verify that when applied to the tree in Figure 3, Algorithm 4 yields the desired stack-machine code.

Flow Control

Structured programming languages are equipped with a variety of high-level control structures like `if`, `while`, `for`, `switch`, and so on. In contrast, low-level languages typically offer two control primitives: conditional and unconditional `goto`. Therefore, one of the challenges faced by the compiler is to translate structured code segments into target code that includes these primitives only. Figure 5 gives two examples.

<i>Source code</i>	<i>Generated code</i>
<code>if (cond)</code>	code for computing <code>~cond</code>
<code>s1</code>	if-goto L1
<code>else</code>	code for executing <code>s1</code>
<code>s2</code>	goto L2
...	label L1
	code for executing <code>s2</code>
	label L2
	...
<code>while (cond)</code>	label L1
<code>s1</code>	code for computing <code>~cond</code>
...	if-goto L2
	code for executing <code>s1</code>
	goto L1
	label L2
	...

FIGURE 5: Compilation of control structures

Two features of high-level languages make the compilation of control structures slightly more challenging. First, control structures can be nested, e.g. `if` within `while` within another `while` and so on. Second, the nesting can be arbitrarily deep. The compiler deals with the first challenge by generating unique labels, as needed, e.g. by using a running index embedded in the label. The second challenge is met by using a recursive compilation strategy. The best way to understand how these tricks work is to discover them yourself, as you will do when you will build the compiler implementation described below.

2. Specification

Usage: The Jack compiler accepts a single command line argument that specifies either a file name or a directory name:

```
prompt> JackCompiler source
```

If *source* is a file name of the form `xxx.jack`, the compiler compiles it into a file named `xxx.vm`, created in the same folder in which `xxx.jack` is located. If *source* is a directory name, all the `.jack` files located in this directory are compiled. For each `xxx.jack` file in the directory, a corresponding `xxx.vm` file is created in the same directory.

Standard mapping over the Virtual Machine

This section lists a set of conventions that must be followed by every Jack-to-VM compiler.

File and function naming: Each `.jack` class file is compiled into a separate `.vm` file. The Jack subroutines (functions, methods, and constructors) are compiled into VM functions as follows:

- ❑ A Jack subroutine `xxx()` in a Jack class `yyy` is compiled into a VM function called `yyy.xxx`.
- ❑ A Jack *function* or *constructor* with k arguments is compiled into a VM function with k arguments.
- ❑ A Jack *method* with k arguments is compiled into a VM function with $k+1$ arguments. The first argument (argument number 0) always refers to the `this` object.

Returning from void methods and functions:

- ❑ VM functions corresponding to void Jack methods and functions must return the constant 0 as their return value.
- ❑ When translating a “do subName” statement that invokes a *void* function or method, the caller of the corresponding VM function must remember to pop (and ignore) the returned value, which is always the constant 0.

Memory allocation and access:

- ❑ The static variables of a Jack class are allocated to, and accessed via, the VM’s `static` segment of the corresponding `.vm` file.
- ❑ The local variables of a Jack subroutine are allocated to, and accessed via, the VM’s `local` segment.
- ❑ Before calling a VM function, the caller must push the function’s arguments onto the stack. If the VM function corresponds to a Jack method, the first pushed argument must be the object on which the method is supposed to operate.
- ❑ Within a VM function, arguments are accessed via the VM’s `argument` segment.
- ❑ Within VM functions corresponding to Jack *methods* or *constructors*, access to the fields of the *this* object is obtained by first pointing the VM’s `this` segment to the current object (using “pointer 0”) and then accessing individual fields via “*this index*” references. For VM functions corresponding to Jack *methods*, the base of the `this` segment is passed as the 0’t^h argument and code for setting it is automatically inserted by the compiler at the beginning of the VM function. For constructors, the base of the `this` segment is obtained and set when the space for the object is allocated. The code for this allocation is automatically inserted by the compiler at the beginning of the constructor’s code.
- ❑ Within a VM function, access to array entries is obtained by pointing the VM’s `that` segment to the address of the desired array location.

Constants:

- ❑ `null` and `false` are mapped to the constant 0. `True` is mapped to the VM constant `-1` (that is obtained via “push constant 0” followed by “neg”).

Use of Operating system functions:

When needed, the compiler should use the following built-in functions, provided by the operating system:

- ❑ Multiplication and division is handled using the OS functions `Math.multiply()` and `Math.divide()`.
- ❑ String constants are handled using the OS constructor `String.new(length)` and the OS method `String.appendChar(nextChar)`.
- ❑ Constructors allocate space for constructed objects using the OS function `Memory.alloc(size)`.

3. Implementation

3.1. *Compilation Example*

We start with a simple example. This examples shows (parts of) a Jack class, the constructed symbol tables, and the generated VM code.

```

// High-level (Jack) code
// Some common sense was sacrificed in this banking example in order
// to create non-trivial and easy-to-follow compilation examples.
class BankAccount {
    // class variables
    static int nAccounts;
    static int bankCommission; // as a percentage, e.g. 10 for 10 percent
    // account properties
    field int id;
    field String owner;
    field int balance;

    method int commission(int x) { /* code omitted */ }

    method void transfer(int sum, BankAccount from, Date when) {
        var int i, j; // some local variables
        var Date due; // some other date variable
        // ... omitted code
        let balance = (balance + sum) - commission(sum * 5);
        return;
    }
    // ... more methods
}

```

Class-scope symbol table

Name	type	Kind	#
nAccounts	int	Static	0
bankCommission	int	Static	1
id	int	Field	0
owner	String	Field	1
balance	int	Field	2

Method-scope (transfer) sym. table

name	Type	kind	#
this	BankAccount	argument	0
sum	int	argument	1
from	BankAccount	argument	2
when	Date	argument	3
i	int	var	0
j	int	var	1
due	Date	var	2

```

// VM pseudo code
function BankAccount.commission
// ... code omitted
function BankAccount.transfer
push this-that-was-passed-as-argument
pop this-segment-base
// .. code omitted
push balance
push sum
add
push this
push sum
push 5
call multiply
call commission
sub
pop balance
push 0
return
// ... code omitted

```

```

// VM code
function BankAccount.commission 0
// ... code omitted
function BankAccount.transfer 3
push argument 0
pop pointer 0
// ... code omitted
push this 2
push argument 1
add
push argument 0
push argument 1
push constant 5
call Math.multiply 2
call BankAccount.commission 2
sub
pop this 2
push constant 0
return
// ... code omitted

```

PROGRAM 6: Symbol table and code generation example.

Compilation examples for arrays and objects can be found as examples in chapter 8.

3.2. Suggested Design

We now turn to propose a software architecture for the compiler. This architecture builds upon the Syntax Analyzer described in chapter 10. In fact, the current architecture is based on gradually evolving the Syntax Analyzer into a full-scale compiler. The overall compiler can thus be constructed using five modules:

- A main driver that organizes and invokes everything (`JackCompiler`);
- A tokenizer (`JackTokenizer`);
- A symbol table (`SymbolTable`);
- An output module for generating VM commands (`VMWriter`);
- A recursive top-down compilation engine (`CompilationEngine`).

Class `JackCompiler`

The program receives a name of a file or a directory, and compiles the file, or all the Jack files in this directory. For each `xxx.jack` file, it creates a `xxx.vm` file in the same directory. The logic is as follows:

For each `xxx.jack` file in the directory:

1. Create a tokenizer from the `xxx.jack` file
2. Create a VM-writer into the `xxx.vm` file
3. `Compile(INPUT: tokenizer, OUTPUT: VM-writer)`

Class `JackTokenizer`

The API of the tokenizer is given in chapter 10.

Class `SymbolTable`

This module provides services for creating, populating, and using a *symbol table*. Recall that each symbol has a scope from which it is visible in the source code. In the symbol table, each symbol is given a running number (index) within the scope, where the index starts at 0 and is reset when starting a new scope. The following kinds of identifiers may appear in the symbol table:

<i>Static:</i>	Scope: class.
<i>Field:</i>	Scope: class.
<i>Argument:</i>	Scope: subroutine (method/function/constructor).
<i>Var:</i>	Scope: subroutine (method/function/constructor).

When compiling code, any identifier not found in the symbol table may be assumed to be a subroutine name or a class name. Since the Jack language syntax rules suffice for distinguishing between these two possibilities, and since no “linking” needs to be done by the compiler, these identifiers do not have to be kept in the symbol table.

A symbol table that associates names with information needed for Jack compilation: type, kind, and running index. The symbol table has 2 nested scopes (class/subroutine).

Routine	Arguments (type)	Returns	Function
Constructor	--	--	Creates a new empty symbol table
startSubroutine	--	--	Starts a new subroutine scope (i.e. erases all names in the previous subroutine's scope.)
define	name (String) type (string) kind (STATIC, FIELD, ARG, or VAR)	--	Defines a new identifier of a given <i>name</i> , <i>type</i> , and <i>kind</i> and assigns it a running index. <code>STATIC</code> and <code>FIELD</code> identifiers have a class scope, while <code>ARG</code> and <code>VAR</code> identifiers have a subroutine scope.
varCount	kind (STATIC, FIELD, ARG, or VAR)	int	Returns the number of variables of the given <i>kind</i> already defined in the current scope.
kindOf	name (String)	(STATIC, FIELD, ARG, VAR, NONE)	Returns the <i>kind</i> of the named identifier in the current scope. Returns <code>NONE</code> if the identifier is unknown in the current scope.
typeOf	name (String)	String	Returns the <i>type</i> of the named identifier in the current scope.
indexOf	name (String)	int	Returns the <i>index</i> assigned to named identifier.

Comment: you will probably need to use two separate hash tables to implement the symbol table: one for the class-scope and another one for the subroutine-scope. When a new subroutine is started, the subroutine-scope table should be cleared.

VMWriter

This class writes VM commands into a file. It encapsulates the VM command syntax.

Emits VM commands into a file			
Routine	Arguments (type)	Returns	Function
Constructor	Output file / stream	--	Creates a new file and prepares it for writing VM commands
writePush	Segment (CONST, ARG, LOCAL, STATIC, THIS, THAT, POINTER, TEMP) Index (int)	--	Writes a VM push command
writePop	Segment (CONST, ARG, LOCAL, STATIC, THIS, THAT, POINTER, TEMP) Index (int)	--	Writes a VM pop command
WriteArithmetic	command (ADD, SUB, NEG, EQ, GT, LT, AND, OR, NOT)	--	Writes a VM arithmetic command
WriteLabel	label (String)	--	Writes a VM label command
WriteGoto	label (String)	--	Writes a VM label command
WriteIf	label (String)	--	Writes a VM If-goto command
writeCall	name (String) nArgs (int)	--	Writes a VM call command
writeFunction	name (String) nLocals (int)	--	Writes a VM function command
writeReturn	--	--	Writes a VM return command
close	--	--	Closes the output file

Class CompilationEngine

This class does the compilation itself. It reads its input from a `JackTokenizer` and writes its output into a `VMWriter`. It is organized as a series of `compilexxx()` methods, where `xxx` is a syntactic element of the Jack language. The contract between these methods is that each `compilexxx()` method should read the syntactic construct `xxx` from the input, `advance()` the tokenizer exactly beyond `xxx`, and emit to the output VM code effecting the semantics of `xxx`. Thus `compilexxx()` may only be called if indeed `xxx` is the next syntactic element of the input. If `xxx` is a part of an expression and thus has a value, then the emitted code should compute this value and leave it at the top of the VM stack.

The API of this module is identical to the API of the Syntax Analyzer's compilation engine, specified in chapter 10. We suggest gradually morphing the syntax analyzer into a full compiler.

4. Perspective

The fact that Jack is a relatively simple language permitted us to side-step several compilation issues. Here we mention some of the most significant ones.

While Jack looks like a typed language, this is hardly the case: all data types are 16-bit long, and the semantics of the language allow compilers to ignore almost all type information (with the single exception that a method call `x.m()` must know `x`'s type). In most other languages the type system has significant implications for the compiler: different amounts of memory must be allocated for different types; conversion from one type into another requires specific operations; the compilation of a simple expression like `x+y` strongly depends on the types of `x` and `y`; and so on. In particular Jack compilers need not determine the types of expressions, e.g. array entries in Jack are not typed.

Another significant simplification is that Jack does not support inheritance. This major simplification implies that all method calls can be determined statically at compile-time, rather than treating them as virtual methods whose location is only determined at run-time according to the run-time type of the object.

The lack of real typing, of inheritance and of public class fields, allows a truly independent compilation of classes. A class in Jack can be compiled without any access to the code of any other class: the fields of other classes are never accessed and all linking to methods of other classes is "late", done just by name.

Many other simplifications of the Jack language are not very significant and can be relaxed with little effort (but also little pedagogic gain). E.g. one may easily add a "for" statement to Jack, or character constants 'c'.

Finally, as usual, we did not pay any attention to optimization. Optimization is, of course, a main focus of attention in the code generation part of any compilation course.

5. Build it

Project 10 guidelines will be published in the web site.

12. The Operating System¹

“Civilization progresses by extending the number of operations that we can perform without thinking about them”

(Alfred North Whitehead, *Introduction to Mathematics*, 1911)

(This chapter is work in progress.) In previous chapters of this book we described and built the hardware architecture of a computer platform, called *Hack*, and the software hierarchy that makes it usable. In particular, we introduced an object-based language, called *Jack*, and described how to write a compiler for it. Other high-level programming languages can be specified on top of the Hack platform, each requiring its own compiler.

The last major interface which is missing in this puzzle is an *operating system*. The OS is designed to close gaps between the computer's software and hardware systems, and to make the overall computer more accessible to programmers and users. For example, our computer is equipped with a bitmap screen. In order to output the text “Hello World”, several hundreds pixels must be drawn at specific locations on the computer's screen. To do so, we can consult the hardware specification, and write commands that put the necessary bits in the RAM segment that controls the screen's output. Needless to say, high-level programmers will need a better interface with the screen. They will want to use commands like `print('Hello World')`, and let someone else worry about the details. And that's where the operating system enters the picture.

Throughout the chapter, the term “operating system” is used rather loosely. In fact, the OS services that we describe comprise an operating system in a very minimal fashion, aiming to encapsulate various hardware-provided services in a software-friendly way and extending high-level languages with some mathematical functions and abstract data types. The dividing line between an operating system in this sense and a “standard language library” is not very clear. Usually, standard libraries associated with particular programming languages include both interfaces to the underlying services of the operating system and other libraries and services related to the programming language and its indented uses.

Indeed, the simple operating system we build here may alternatively be viewed as a standard library for the Jack language. It is packaged a collection of Jack classes, each providing a set of related services via Jack subroutine calls. The resulting OS has many features resembling those of industrial strength operating systems, but it lacks numerous OS features such as process handling or disk management.

Operating systems are usually programmed in a high level language and compiled into binary form like any other program. Indeed the operating system described here is written completely in Jack. Unlike normal programs written in a high-level language, the operating system code must be aware of the hardware platform it is running on. For example, in order to implement the various encapsulated I/O services, it must directly access the I/O devices. The Jack programming language was defined with sufficient “lowness” in it, permitting an intimate closeness to the hardware, when needed.

¹ From *The Elements of Computing Systems*, Nisan & Schocken, MIT Press, forthcoming in 2004, www.idc.ac.il/csd

The title of this chapter is somewhat misleading, since we discuss only the OS elements needed for our computer platform. The chapter starts with a background section that describes the underlying algorithms and programming techniques. Next, we specify the complete Sack OS API, and give guidelines on how to implement it in Jack. Section 4 mentions briefly some of the elements of normal operating systems that were not discussed, and Section 5 walks you through a complete implementation of the OS.

The chapter embeds two key lessons, one in software engineering and one in computer science. First, we describe and illustrate the important interplay between high-level languages, compilers, and operating systems. Second, we present a series of elegant and efficient algorithms, each being a little computer science gem.

1. Background

1.1 Mathematical Operations

Almost every computer system must support mathematical operations like addition, multiplication, and division. Normally, *addition* is implemented in hardware, at the ALU level, as we have done in Chapter 3. Other operations like *multiplication* and *division* can be implemented in either hardware or software, depending on the computer's purpose and cost/performance requirements. This section shows how multiplication, division, and square root operations can be implemented efficiently in software, at the operating system level. It should be noted that hardware implementation of these operations are based on the same algorithms presented below.

A word about algorithmic efficiency is in order here. Mathematical algorithms operate on n -bit binary numbers, with typical computer architectures having $n=16$ (as in Hack), 32 or 64. As a rule, we seek algorithms whose running time is proportional (or at least “polynomial”) in this parameter n . Algorithms whose running time is proportional to the *value* of n -bit numbers are unacceptable, since these values are exponential in n . For example, suppose we implement the multiplication operation $x \cdot y$ using the “repeated addition” algorithm *for* $i = 1 \dots y$ $\{sum = sum + x\}$. If we use this algorithm on a 64-bit computer, y can be as large as 1,000,000,000,000,000 (still smaller than the maximal value 2^{63}). In such cases, this naïve algorithm may run for years, even on the fastest computers. On the other hand, the running time of the multiplication algorithm that we present below is proportional to the number of bits n , and thus will require only dozens or a few hundreds of operations on a 64-bit architecture for any value of y .

We will use the standard “Big-Oh” notation, $O(n)$, to describe the running time of algorithms. Readers who are not familiar with this notation should simply read $O(n)$ as “in the order of magnitude of n ”. With that in mind, we now turn to present a multiplication $x \cdot y$ algorithm for n -bit numbers whose running time is $O(n)$ rather than $O(y)$, which is exponentially larger.

Multiplication

Consider the standard multiplication method taught in elementary school. To compute 356 times 27, we write the two numbers one on top of the other. Next, we multiply each digit of 356 by 7. Next, we "shift to the left" one position, and multiply each digit of 356 by 2. Finally, we sum up the numbers in the two rows and obtain the result. The binary version of this technique -- Algorithm 1 -- is exactly the same.

The "steps"	The algorithm explained (first 4 of 16 iteration) (ignoring some leading zeros, to save clutter)
$\begin{array}{r} 1\ 0\ 1\ 1 \\ \underline{1\ 0\ 1} \\ 1\ 0\ 1\ 1 \\ 0\ 0\ 0\ 0 \\ \underline{1\ 0\ 1\ 1} \\ 1\ 1\ 0\ 1\ 1\ 1 \end{array} = 5\ 5$	$\begin{array}{r} x: 0\ 0\ 0\ 1\ 0\ 1\ 1 \\ y: 0\ 0\ 0\ 0\ 1\ 0\ 1 \\ \hline 0\ 0\ 0\ 1\ 0\ 1\ 1 \\ 0\ 0\ 1\ 0\ 1\ 1\ 0 \\ 0\ 1\ 0\ 1\ 1\ 0\ 0 \\ \hline 1\ 0\ 1\ 1\ 0\ 0\ 0 \\ x \cdot y: 0\ 1\ 1\ 0\ 1\ 1\ 1 \end{array} \begin{array}{l} j^{\text{th}} \text{ bit of } y \\ \\ \\ \\ \\ \text{sum} \end{array}$

multiply(x,y):

```

initialize sum = 0
initialize shiftedX = x
for j = 0...(n-1) do
    if (j'th bit of y) = 1 then
        sum = sum + shiftedX
        shiftedX = shiftedX * 2

```

ALGORITHM 1: Multiplication.

This algorithm performs $O(n)$ addition operations on n -bit numbers, where n is the number of bits in x and y . (Note that $shiftedX * 2$ can be efficiently obtained by either adding $shiftedX$ to itself or shifting its bit representation one place to the left.)

Division

The naïve way to compute x / y is to repeatedly subtract y from x until it is impossible to continue (i.e. until $x < y$). The running time of this algorithm is clearly proportional to the quotient, and may be as large as $O(x)$, which is exponential in the number of bits n . To speed up this algorithm, we can try to subtract large chunks of y 's from x in each iteration. For example, if $x=891$ and $y=5$, we can tell right away that we can deduct a hundred 5's from x and the remainder will still be greater than 5, thus shaving 100 iterations from the naïve approach. Indeed, this is the rationale behind the school method for long division. Formally, in each iteration we try to subtract the largest possible shift of y , i.e. $y \cdot T$ where T is the largest power of 10 such that $y \cdot T \leq x$. The binary version of this algorithm is precisely the same, except that T is a power of 2 instead of 10.

It is an easy exercise to formally write down this school algorithm for division, as we have done for multiplication. We find it more illuminating to provide the same logic in the form of a recursive program that is probably easier to implement:

```
divide (x,y):  
  // Integer part of x/y, where x and y are natural numbers.  
  if y>x return 0  
  q = divide(x, 2*y)  
  if (x - 2*q*y) < y  
    return 2*q  
  else  
    return 2*q + 1
```

ALGORITHM 2: Division.

The running time of this algorithm is determined by the depth of the recursion. Since in each level of recursion the value of y is multiplied by 2, and since we terminate once $y > x$, it follows that the recursion depth is bounded by the number of bits in x . Each recursion level involves a constant number of addition, subtraction, and multiplication operations, and thus the total running time of the algorithm requires $O(n)$ such operations.

Algorithm 2 may be considered sub-optimal since each multiplication operation also requires $O(n)$ addition and subtraction operations. However, careful inspection reveals that the product $2*q*y$ can be computed without any multiplication. Instead, we can rely on the value of this product in the previous recursion level, and use a few addition operations to establish its current value.

Square Root

Square roots can be computed efficiently in a number of different ways, e.g. using the Newton-Raphson method or a Taylor series expansion. For our purposes though, a simple binary search will suffice. The square root function $y = \sqrt{x}$ has two convenient properties. First, it is monotonically increasing. Second, its inverse function $x = y^2$ is something that we already know how to compute (multiplication). Taken together, these properties imply that we have all we need to compute square roots using binary search.

```

sqrt(x):    // Compute the integer part of  $y = \sqrt{x}$  :
// Find  $y$  such that  $y^2 \leq x < (y+1)^2$  :
initialize low = 0
initialize high = square root of the largest n-bit number
while low < high do
    med = (low + high) / 2
    if med * med > x
        high = med - 1
    else
        low = med
return low

```

ALGORITHM 3: Square root computation using binary search.

Note that each loop iteration takes a constant number of arithmetic operations. Since the difference between *high* and *low* shrinks by a factor of 2 in each iteration, the total number of iterations is at most the logarithm of the initial value of *high-low*, which is at most n . Thus the total running time is $O(n)$ arithmetic operations.

1.2 String representation of numbers

Computers represent numbers in memory using binary codes. Yet humans are used to dealing with numbers in a decimal notation. Thus, when humans have to read or input numbers, and only then, a conversion to or from decimal notation must be performed. Typically, this service is implicit in the character handling services provided by the operating system. In other words, programmers can write high level code that operates directly on the decimal or textual representation of numbers, assuming that the OS will perform the necessary conversions, as needed. We now turn to describe how some of these OS services are actually implemented.

Of course the only subset of characters which is of interest here are the 10 digits which represent actual numbers. The ASCII codes of these characters are as follows:

Character:	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'
ASCII code:	48	49	50	51	52	53	54	55	56	57

As gleaned from the table, single digit characters can be easily converted into their numeric representation, and vice versa, as follows. To compute the ASCII code of a given digit $0 \leq x \leq 9$, we can simply add x to 48 – the code of '0'. Conversely, the numeric value represented by an

ASCII code $48 \leq y \leq 57$ is obtained by $y - 48$. And once we know how to convert single digits, we can proceed to convert any given integer. These conversion algorithms can be based on either an iterative or a recursive logic, so we present one of each.

```
// Convert a number to a string.
```

```
toString(n):
```

```
    lastDigit = n % 10
    c = character representing lastDigit
    if n < 10
        return c (as a string)
    else
        return toString(n/10).append(c)
```

```
// Convert a string to a number.
```

```
toInt(s):
```

```
    n = 0
    for i = 1 .. length of s
        d = integer value of the digit s[i]
        n = n * 10 + d
    return n
```

(Assuming that $s[1]$ is the most significant digit character of s .)

ALGORITHMS 4-5: String-numeric conversion

1.3 Memory Management

Dynamic Memory Allocation: Some of the memory required for a program's execution is explicitly defined in the program code. For example, *static variables* are allocated when the program starts running, *local variables* are allocated when a subroutine starts running, and so on. Other memory is dynamically requested during the program's execution. For example, memory should be allocated dynamically to accommodate the construction of new objects or arrays whose size is determined only during run-time. This dynamic memory allocation is typically done by the operating system. When a running program constructs a new object of a certain size, enough RAM space must be located in memory and then allocated to store the new object. When the program declares that the object is no longer needed, its RAM space may be recycled. The RAM segment from which memory is dynamically allocated is called the *heap*.

Operating systems use various techniques for handling dynamic memory allocation and deallocation. These techniques are implemented in two functions traditionally called `alloc()` and `dealloc()`. We present two memory allocation algorithms: a basic one and an improved one.

Basic memory allocation algorithm: The data structure that this algorithm manages is a single pointer, called *free*, which points to the beginning of the heap segment that was not yet allocated. Algorithm 6-a gives the details.

```
// Objects and arrays are stored on the heap.
Initialization: free=heapBase
// Allocate a memory segment of size words:
alloc(size):
    pointer = free
    free += size
    return pointer
// De-allocate the memory space of a given object:
deAlloc(object):
    do nothing
```

ALGORITHM 6-a: Basic Memory Allocation Scheme (wasteful)

Algorithm 6-a is clearly wasteful, as it does not reclaim the space of decommissioned objects.

Improved memory allocation algorithm: This algorithm manages a linked list of available memory blocks, called *freeList*. Each block is characterized by two “housekeeping” fields: the block’s length, and a pointer to the next block in the *freeList*. These fields can be kept in the two memory locations preceding the block itself. For example, the implementation can use the convention `b.length==x[-1]` and `b.next==x[-2]`.

When asked to allocate a memory segment of size n , the algorithm has to search the *freeList* for a suitable block. There are two well-known strategies for doing this. *Best-fit* finds the block whose size is the closest (from above) to the required size, while *first-fit* finds the first block that is long enough. Once the block has been found, the required memory segment is taken from it. Next, this block is updated in the *freeList*, becoming the part that remained after the allocation (if no memory was left in the block, the entire block is eliminated from the *freeList*).

When asked to reclaim the memory of an unused object, the algorithm inserts the de-allocated block into the *freeList*. The details are given in Algorithm 6-b.


```
// Objects and arrays are stored on the heap.
initialization:
    freeList = heapBase+2
    freeList.length = heapEnd-(heapBase+2)
    freeList.next = null

// Allocate a memory segment of size words:
alloc(size):
    1. use methods like best-fit or first-fit
       to locate a free block in freeList
    2. return the base address of that block

// De-allocate the memory space of a given object:
deAlloc(object):
    Append the object to the freeList
```

ALGORITHM 6-b: Improved Memory Allocation Scheme (with memory recycling)

After a while, dynamic memory allocation schemes like Algorithm 6-b may create a block fragmentation problem. Hence, some kind of “defrag” operation should be considered, i.e. merging memory segments that are physically consecutive in memory but logically split into different blocks in the *freeList*. The defragmentation operation can be done each time an object is de-allocated, or when `alloc()` cannot find an appropriate block, or according to some other intermediate or ad-hoc condition.

1.4 Variable length arrays and Strings

The memory allocation operations considered above allocate fixed length memory blocks. This is exactly appropriate for arrays in high level programming languages. Most programming languages also provide data-types that have variable length – most commonly strings. Strings contain arrays of characters, whose length may vary. String objects are usually provided by the standard library of the programming language (e.g. the `String` and `StringBuffer` classes in Java or the `strXXX` functions in C).

Indeed, the implementation of variable length strings can be done by creating a `String` class that provides the string abstraction and related services. The standard data structure used in this context typically contains an array of characters that holds the string contents, and the current length of the string. Array locations beyond the current length are not considered part of the string contents. When such a data structure is first constructed, some maximum possible length must be defined for it, and the array is allocated to be in this size.

1.5 Input/Output Management

An important part of the functionality of an operating system is handling the various I/O devices connected to the computer, encapsulating the details of interfacing them, and providing convenient access to their basic functionality. We will describe only the very basic elements of

handling the I/O devices available in Hack: a screen and a keyboard. We will divide the issue of handling the screen into two logically separate steps: handling graphics, and handling character output.

1.5.1 Graphics output

Pixel drawing: Most computers today use *raster*, also called *bitmap*, display technologies. The only primitive operation that can be physically performed in such an output device is that of drawing a single pixel (a *pixel* refers to a single “dot” on the screen). Pixels are specified using (*column*, *row*) coordinates. The usual convention is that columns are numbered from left to right (like the *x*-axis in high school) while rows are numbered from the top down (opposite of the *y*-axis in high school). Thus the top left pixel is located in screen location (0,0).

The low level drawing of a single pixel is a hardware-specific operation that depends on the particular interface of the screen or the underlying graphics card. If the screen interface is based on a memory map, then drawing a pixel is achieved by writing the proper value into the RAM location that represents the required pixel in memory.

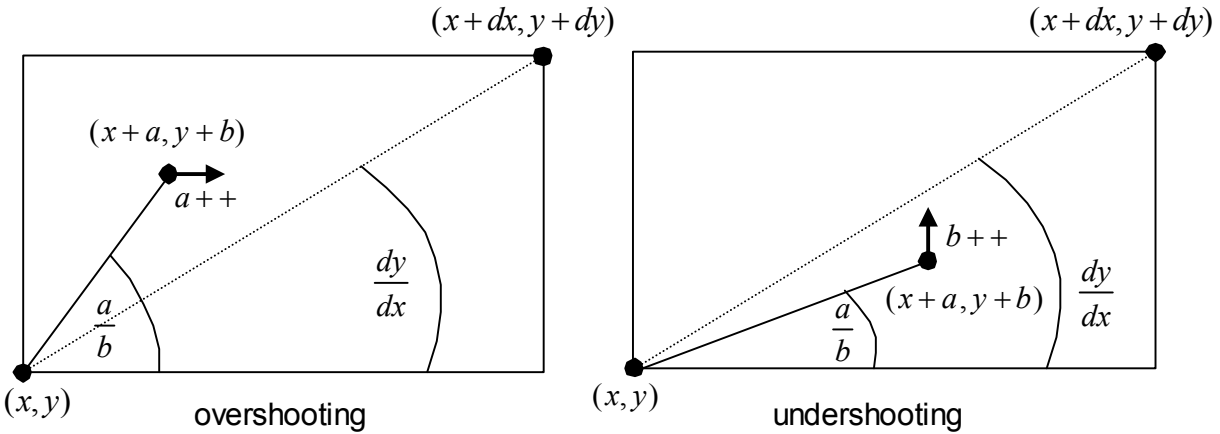
```
drawPixel (x,y):  
// Hardware-specific. Assuming a memory mapped screen:  
  
Write a pre-determined value in the RAM  
location corresponding to screen location (x,y).
```

ALGORITHM 7: Drawing a pixel.

Now that we know how to draw a single pixel, we turn to describe how to draw lines and circles.

Line drawing: Recall that the only elementary drawing operation supported by computers is that of drawing a single pixel. Hence, when asked to draw a line between two screen locations, the best that we can possibly do is approximate the line by drawing a series of pixels along the imaginary line that connects the two points. Note that the “pen” that we use can move in four directions only: up, down, left, and right. Thus the drawn line is bound to be jagged, and the only way to make it look good is to use a high-resolution screen. Since the receptor cells in the human eye’s retina also form a grid of “biological pixels,” there is a limit to the image granularity that the human eye can resolve. Thus, high-resolution screens and printers can fool the human eye to believe that the lines drawn by pixels or printed dots are actually smooth. In fact they are always jagged.

The procedure for drawing a line from location (x1,y1) to location (x2,y2) starts by drawing the (x1,y1) pixel, and then zigzagging in the direction of the (x2,y2) pixel, until it is reached. See Algorithm 8a for the details.



```

drawLine(x,y,x+dx,y+dy):
// Assuming  $dx, dy \geq 0$ .
initialize  $(a, b) = (0, 0)$ 
while  $a \leq dx$  or  $b \leq dy$  do
    drawPixel( $x + a, y + b$ )
    if  $a/dx < b/dy$  then  $a++$  else  $b++$ 

```

ALGORITHM 8-a: Line Drawing

Algorithm 8-a is applicable only for $dx, dy \geq 0$. To extend it into a general-purpose line drawing routine, one also has to take care of the three other possibilities: $dx, dy < 0$, $dx > 0, dy < 0$, and $dx < 0, dy > 0$.

An annoying feature of this algorithm is the use of division operations (a/dx , b/dy) in each loop iteration. This division operation is not only time-consuming -- it also requires floating point operations rather than simple integer arithmetic. A possible solution is to replace the $a/dx < b/dy$ condition with the equivalent $a*dy < b*dx$, which requires only integer multiplication. Further, careful inspection reveals that this latter condition may be checked without using any multiplication. As shown in Algorithm 8-b, this may be done by maintaining a variable that updates the value of $a*dy - b*dx$ each time either a or b are modified.

```

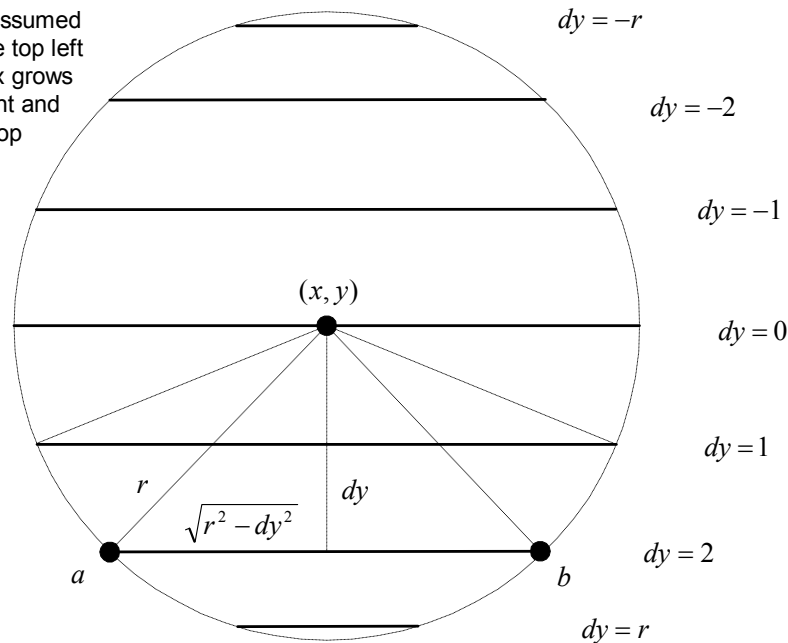
// To test whether  $a/dx < b/dy$ ,
// maintain a variable  $adyMinusbdx$ , and test whether  $adyMinusbdx < 0$ :
Initialization: set  $adyMinusbdx = 0$ 
When  $a++$  is performed: set  $adyMinusbdx = adyMinusbdx + dy$ 
When  $b++$  is performed: set  $adyMinusbdx = adyMinusbdx - dx$ 

```

ALGORITHM 8-b: Testing whether $a/dx < b/dy$

Circle drawing: There are several ways to draw circles on a bitmap screen. We present an algorithm that uses three routines already implemented in this chapter: *multiplication*, *square root computation*, and *line drawing*.

Point (0,0) is assumed to be at the top left corner. Thus x grows from left to right and y grows from top to bottom.



$$\text{point } a = (x - \sqrt{r^2 - dy^2}, y + dy)$$

$$\text{point } b = (x + \sqrt{r^2 - dy^2}, y + dy)$$

drawCircle(x,y,r):

for each $dy \in -r \dots r$ do

drawLine from $(x - \sqrt{r^2 - dy^2}, y + dy)$ to $(x + \sqrt{r^2 - dy^2}, y + dy)$

ALGORITHM 9: Circle Drawing

The algorithm is based on drawing a series of horizontal lines (like the typical line ab in the figure), one for each row in the range $y - r$ to $y + r$. Since r is specified in pixels, the algorithm ends up drawing a line in every screen row along the circle's north-south diameter, and thus the resulting circle is completely filled. A trivial version of this algorithm can yield an empty circle as well.

1.5.2 Character Output

All the output that we described so far was graphical: pixels, lines, and circles. We now turn to describe how characters are printed on the screen. Well, pixel by pixel. The first step before writing characters on a screen is to divide the physical pixel-oriented screen into a logical character-oriented screen suitable for drawing complete characters. For example, consider a

physical 256 rows by 512 columns screen. If we allocate a grid of 11*8 pixels for drawing a single character (11 rows, 8 columns), then our screen can show 23 lines, each holding 64 characters (with 3 extra rows of pixels left unused). Note that this calculation accounts for the requisite spacing, since the 11*8 allocation includes a 1-pixel space between adjacent lines and a 2-pixels space between adjacent characters.

Now, for each character that we want to display on the screen, we have to design a suitable bitmap. For example, Figure 10 gives a possible bitmap for the letter “A”.

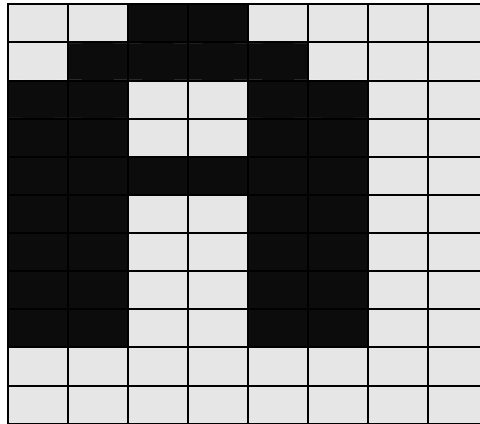


FIGURE 10: Character bitmap of the letter “a”.

Characters are usually drawn on the screen one after the other, from left to right. For example, the two commands `print("a")` and `print("b")` probably mean that the programmer wants to see the image “ab” drawn on the screen. Thus the character-writing package must maintain a “cursor” object that keeps track of the screen location where the next character should be drawn on the logical “character screen”. The cursor information consists of “line” and “column” counts. For example, the character screen described in the previous paragraph is characterized (excuse the pun) by $0 \leq \text{line} \leq 22$ and $0 \leq \text{column} \leq 63$. Drawing a single character at location $(\text{line}, \text{column})$ is achieved by writing the bitmap of the character onto the box of pixels at rows $\text{line} * 11 \dots \text{line} * 11 + 10$, and columns $\text{column} * 8 \dots \text{column} * 8 + 7$. After a character is drawn, the cursor should be moved one step to the right (i.e. column is increased by 1), and when a new line is requested, row is increased by 1 and column is reset to 0. When the bottom of the screen is reached, there is a question of what to do next, the common solution being to effect a “scrolling” operation. Another possibility is erasing the screen and starting over from the top left corner (i.e. setting the cursor to (0,0).)

To conclude, we know how to write characters on the screen. Writing other types of data is now easy: strings are written character by character; numbers are written by first converting them to strings, and so on.

1.5.3 Keyboard Handling

Handling user-supplied character input is more involved than meets the eye. When interacting with a computer, a human user presses a key on the keyboard for some variable duration of time. Yet the program that manages the interaction with the user wants to accept this single character

input independently of the time that elapsed between the “key press” and “key release” events.. Further, we usually want to give some feedback to the user. First, we typically want to display some graphical cursor at the screen location where the input “goes”. Second, we typically want to echo the actual input by displaying it on the screen at that point.

In the “raw” form of keyboard access, the program gets direct data from the keyboard indicating which key is *currently* pressed by the user. The access to this raw data depends on the specifics of the keyboard interface. For example, if the keyboard is represented using a memory map, we can simply inspect the contents of the relevant RAM area to determine which key is presently pressed. The details of this inspection can then be incorporated into the implementation of Algorithm 11.

```

keyPressed (x,y):
  // Depends on the specifics of the keyboard interface.
  if a key is pressed on the keyboard
    return the ASCII value of the key
  else
    return 0

```

ALGORITHM 11: Capturing (“raw”) keyboard input.

Usually, an input typed by the user is considered final only after the “enter” key has been pressed, yielding the *new-line* character. Further, users may backspace and erase their previously entered characters until this event takes place. These requirements, along with the “raw” input form supplied by the `keyPressed` routine, can be used to implement the “cooked” form of character input expected by human users. The details are shown in Algorithms 12 and 13.

```

// Read and echo a single character.
readChar():
  display the cursor
  while no key is pressed on the keyboard
    do nothing // wait till the user presses a key
  c = code of currently pressed key
  while a key is pressed
    do nothing // wait for the user to let go
  print c at the current cursor location
  move the cursor one position to the right
  return c

```

```

// Read a “line” (until new-line).
readLine():
  s = empty string
  repeat
    c = readChar()
    if c == new-line character
      print new-line
      return s
    else if c == backspace character
      remove last character from s
      move the cursor 1 pos. back
    else
      s = s.append(c)
  return s

```

ALGORITHMS 12-13: Capturing (“cooked”) keyboard input.

2. The Sack OS Specification

This section duplicates “The Jack Standard Library” section from Chapter 9. The various services of the Sack operating system are organized in eight modules, as follows:

- **Math:** Provides basic mathematical operations;
- **String:** Implements the `String` type and basic string-related operations;
- **Array:** Defines the `Array` type and allows construction and disposal of arrays;
- **Output:** Handles text based output;
- **Screen:** Handles graphic screen output;
- **Keyboard:** Handles user input from the keyboard;
- **Memory:** Handles memory operations;
- **Sys:** Provides some execution-related services.

This section specifies the subroutines that are supposed to be in these classes.

Math

This class enables various mathematical operations.

- Function void `init()`.
- Function int `abs(int x)`: Returns the absolute value of `x`.
- Function int `multiply(int x, int y)`: Returns the product of `x` and `y`.
- Function int `divide(int x, int y)`: Returns the integer part of the `x/y`.
- Function int `min(int x, int y)`: Returns the minimum of `x` and `y`.
- Function int `max(int x, int y)`: Returns the maximum of `x` and `y`.
- Function int `sqrt(int x)`: Returns the integer part of the square root of `x`.

String

This class implements the `String` data type and various string-related operations.

- Constructor `String new(int maxLength)`: Constructs a new empty string (of length zero) that can contain at most `maxLength` characters.
- Method void `dispose()`: Disposes this string.
- Method int `length()`: Returns the length of this string.
- Method char `charAt(int j)`: Returns the character at location `j` of this string.
- Method void `setCharAt(int j, char c)`: Sets the `j`'th element of this string to `c`.
- Method `String appendChar(char c)`: Appends `c` to this string and returns this string.
- Method void `eraseLastChar()`: Erases the last character from this string.

- Method `int intValue()`: Returns the integer value of this string (or at least of the prefix until a non numeric character is found).
- Method `void setInt(int j)`: Sets this string to hold a representation of j.
- Function `char backSpace()`: Returns the backspace character.
- Function `char doubleQuote()`: Returns the double quote (“) character.
- Function `char newLine()`: Returns the newline character.

Array

This class enables the construction and disposal of arrays.

- Function `Array new(int size)`: Constructs a new array of the given size.
- Method `void dispose()`: Disposes this array.

Output

This class allows writing text on the screen.

- Function `void init ()`.
- Function `void moveCursor(int i, int j)`: Moves the cursor to the j'th column of the i'th row, and erases the character located there.
- Function `void printChar(char c)`: Prints c at the cursor location and advances the cursor one column forward.
- Function `void printString(String s)`: Prints s starting at the cursor location, and advances the cursor appropriately.
- Function `void printInt(int i)`: Prints i starting at the cursor location, and advances the cursor appropriately.
- Function `void println()`: Advances the cursor to the beginning of the next line.
- Function `void backSpace()`: Moves the cursor one column back.

Screen

This class allows drawing graphics on the screen. Column indices start at 0 and are left-to-right. Row indices start at 0 and are top-to-bottom. The screen size is hardware-dependant (over HACK: 256 rows * 512 columns).

- Function `void init ()`.
- Function `void clearScreen()`: Erases the entire screen.
- Function `void setColor(boolean b)`: Sets the screen color (white=false, black=true) to be used for all further drawXXX commands.
- Function `void drawPixel(int x, int y)`: Draws the (x,y) pixel.

- Function void **drawLine**(int x1, int y1, int x2, int y2): Draws a line from pixel (x1,y1) to pixel (x2,y2).
- Function void **drawRectangle**(int x1, int y1, int x2, int y2): Draws a filled rectangle where the top left corner is (x1, y1) and the bottom right corner is (x2,y2).
- Function void **drawCircle**(int x, int y, int r): Draws a filled circle of radius $r \leq 181$ around (x,y).

Keyboard

This class allows reading inputs from the keyboard.

- Function void **init** () .
- Function char **keyPressed**(): Returns the character of the currently pressed key on the keyboard; if no key is currently pressed, returns 0.
- Function char **readChar**(): Waits until a key is pressed on the keyboard and released, and then echoes the key to the screen and returns the character of the pressed key.
- Function String **readLine**(String message): Prints the message on the screen, reads the next line (until a newline character) from the keyboard, echoes the line to the screen, and returns its value. This method handles user backspaces.
- Function int **readInt**(String message): Prints the message on the screen, reads the next line (until a newline character) from the keyboard, echoes the line to the screen, and returns the integer until the first non numeric character in the line. This method handles user backspaces.

Memory

This class allows direct access to the main memory.

- Function void **init** () .
- Function int **peek**(int address): Returns the value of the main memory at this address.
- Function void **poke**(int address, int value): Sets the value of the main memory at this address to the given value.
- Function Array **alloc**(int size): Allocates the specified space on the heap and returns a reference to it.
- Function void **deAlloc**(Array o): De-allocates the given object and frees its memory space.

Sys

This class supports some execution-related services.

- Function void **init**(): Calls the **init** functions of the other OS classes (where appropriate) and then calls the `Main.main()` method.
- Function void **halt**(): Halts the program execution.

- Function void `error(int errorCode)`: Prints the error code on the screen and halts.
- Function void `wait(int duration)`: Waits approximately *duration* milliseconds and returns.

3. Implementation

This section provides some hints and suggestions for implementing the various classes of the Sack OS using Jack over the Hack platform.

Initialization: In each OS class that requires class-level initialization, the class-level initialization code is embedded in a single `init` routine. This routine is then called (once) by the OS's `Sys.init` routine, as part of the “booting” process. This is explained further below, in the `Sys` class implementation tips.

Math: The multiplication and division algorithms 1 and 2 are designed for natural (non-negative) numbers only. A simple way of handling negative numbers is doing all calculations on absolute values and then setting the sign appropriately. For the multiplication algorithm, this is not really needed since it turns out that if the input numbers are given in 2's complement then the results will be correct with no further ado.

Note that in each iteration j of the multiplication Algorithm 1, the j^{th} bit of the second number is extracted. We suggest to encapsulate this operation as follows:

`bit(x, j)`: Returns true if the j^{th} bit of the integer x is 1 and false otherwise.

The `bit(x, j)` function can be easily implemented using shifting operations. Alas, Jack does not support shifting. Instead, to speed up this function implementation in Jack, it may be convenient to define a fixed static array of length 16, say `twoToThe[j]`, whose j^{th} location holds the value 2 to the power of j . This array may be initialized once (in `Math.init`), and then used, via bitwise Boolean operations, in the implementation of `bit(x, j)`.

In the division Algorithm 2 we multiply y by a factor of 2 until $y > x$. A detail that needs to be taken into account is that y can overflow. This overflow can be detected by noting when y becomes negative.

For computing the square root (Algorithm 3), notice that $182^2 = 33124 > 32767 = 2^{15} - 1$, and thus the binary search can be limited to the range 0..181.

String: The ASCII codes of newline, backspace and doubleQuote are 128, 129 and 34 respectively.

Array: Note that the “new” function is not really a constructor, despite the fact that it looks like one. Therefore, memory space for a new array should be explicitly allocated using a call to `Memory.alloc()`. Similarly, de-allocation must be done explicitly.

Output: We suggest using character maps of 11*8, leading to 23 lines of 64 characters each. Since building character maps of all ASCII characters is quite a burden, we supply such maps for you (except for one or two characters which are left as an exercise). In particular, we supply Jack code that gives for each printable ASCII character an array that holds its bit map (using a “font” that we created). The array holds 11 entries, each corresponding to a row of pixels. Each row is given as a binary number whose bits represent the 8 pixels in the row.

There is no need to implement scrolling.

Memory: The `peek` and `poke` functions should provide direct access to the underlying memory. As it turns out, the Jack language includes a trapdoor that enables the programmer to gain complete control of the computer’s memory. This hacking trick can be exploited to enable the implementation of `peek` and `poke` using plain Jack programming. The trick is based on an anomalous use of reference variables (pointers). Specifically, the Jack language does not prevent the programmer from assigning a constant to a reference variable. This constant can then be treated as an absolute memory address. In particular, when the reference variable happens to be an array, this trick can give convenient and direct access to the entire computer memory:

```
// To create a Jack-level "proxy" of the RAM:
var Array memory;
let memory=0;
// From this point on we can use code like:
let x = memory[j] // where j is any RAM address
let memory[j] = y // where j is any RAM address
```

PROGRAM 14: A trapdoor enabling complete control of the RAM from Jack.

Following the first two lines of Program 14, the base of the `memory` array points to the first address in the computer's RAM. To set or get the value of the RAM location whose physical address is `j`, all we have to do is manipulate the array entry `memory[j]`. This will cause the compiler to manipulate the RAM location whose address is `0+j`, which is precisely what we want to do.

Recall that in Jack, arrays are not allocated space on the heap at compile-time, but rather at run-time, when the array's `new` method is called. Here, however, a `new` initialization will defeat the purpose, since the whole idea is to anchor the array in a particular address rather than let the OS allocate it to an address in the heap that we don't control. In short, this hacking trick works because we use the array variable without initializing it “properly”, as we would do in normal usage of arrays.

The higher level functions `alloc` and `deAlloc` manipulate the heap. Recall that the standard implementation of the VM over the Hack platform specified that the heap resides at RAM locations 2048-16383.

Screen: Drawing a pixel on the screen is done by directly accessing its memory map using `Memory.peek()` and `Memory.poke()`. Recall that the memory map of the screen on the hack

platform specifies that the pixel at column c and row r ($0 \leq c \leq 511$, $0 \leq r \leq 255$) is mapped to the $c\%16$ bit of memory location $16384 + r*32 + c/16$. Notice that drawing a single pixel requires changing a single bit in the accessed word, a task that can be achieved in Jack using bit-wise operations.

The only tricky element in the other graphic operations here is avoiding overflow. Overflow in the line drawing Algorithm 8 will not occur if you use the suggested efficient implementation for determining whether $a/dx < b/dy$.

The specification of the `drawCircle` routine limits circle radiuses to be at most 181. This eliminates the possibility of overflow when using the suggested circle drawing Algorithm 9.

Keyboard: Recall that the memory map of the keyboard on the Hack platform is at memory location 24576. The method `keyPressed()` provides “raw” access to this memory location and can be implemented easily using `Memory.peek()`. The other methods provide the required “cooking”.

Sys: An application program written in Jack is a collection of classes. One class must be named `Main`, and this class must include a method named `main`. In order to start running the application program, the `Main.main()` method should be invoked. Now, it should be understood that the operating system is itself a program. Thus, when the computer boots up, we want to start running the operating system program first, and then we want the OS to start running the application program. Indeed, the VM specification states a bootstrap code that automatically invokes a VM function called `Sys.init()`. This `Sys.init()` function, which is part of the OS’s `Sys` class, should then call the `init()` methods of the other OS classes (libraries), and then call the `Main.main()` method of the application program.

The `Sys.wait` function can be implemented pragmatically, under the limitations of the Hack platform. In particular, you can use a loop that runs approximately n milliseconds before it (and the method) returns. You will have to time your specific computer to obtain a one millisecond wait (this constant varies from one CPU to another). As a result, your `Sys.wait()` function will not be portable, but that’s life.

The `Sys.halt` function can be implemented by entering an infinite loop.

4.Perspective

The standard class library presented in this chapter was given the name “operating system” due to its main conceptual goal: encapsulation of the gory hardware details, omissions, and idiosyncrasies in a clean software packaging. However, the gap between what we called here an operating system and an “industrial strength” operating system is rather large.

Our “operating system” completely lacks some of the very basic components most closely associated with operating systems. The Hack/Jack system supports no multi-threading or multi-processing; in contrast the very kernel of most operating systems is devoted to exactly that. The Hack/Jack system has no mass storage devices; in contrast the main information kept and handled by operating systems is the file system. The Hack/Jack system has neither a textual user interface

(as in a Unix shell or a DOS window), nor a graphical one (windows, mouse, icons, etc.); in contrast this is the operating system aspect that users expect to see and interact with. Numerous other services commonly found in operating systems are not present here: security, communication, and more.

A central feature of most operating systems is that their code is somehow more “privileged” than user code – the hardware platform forbids non-system code to perform various operation that are allowed to OS code. Consequently, access to operating system services requires a mechanism that is more elaborate than a simple function call. Further, programming languages usually wrap these OS services in regular functions or methods. In our case there is no difference between normal code and OS code, and OS system services run in the same “user mode” as that of the application program.

Our operating system does however include some of the most fundamental OS services, e.g. managing memory, driving I/O, handling initialization, as well as supplying mathematical functions not implemented in hardware. Additionally, our operating system supplies some services that are normally found in the standard libraries of programming languages, e.g. the *String* abstraction. Consistent with the spirit of this book, all these OS services are described and implemented in the simplest possible way, but not simpler.

The algorithms that we presented for multiplication and division are very standard. However, in most cases these algorithms, or a variant thereof, are implemented in hardware rather than in software. The running time of the presented multiplication and division algorithms is $O(n)$ addition operations. Since adding two n -bit numbers requires $O(n)$ bit operations (gates in hardware), multiplication and division require $O(n^2)$ bit operations. There are algorithms whose running time is asymptotically significantly faster. For a large number of bits, these algorithms are more efficient.

5. Build It

This project describes a modular implementation of the entire Sack operating system. The OS is implemented as a collection of eight classes. Each of these classes can be implemented in isolation. Further, the classes may be developed and incrementally tested in any particular order.

Objective: Implement the Sack operating system and test it by executing application programs that use OS services.

Resources: The main tool that you need in this project is Jack -- the language in which you have to develop the OS. This implies that you will also need the supplied *Jack compiler*, to compile your OS implementation as well as the supplied test programs. In order to facilitate partial testing of the OS, you will also need the *supplied OS*, consisting of 8 `.vm` files (one for each OS class). Finally, you will need the supplied *VM Emulator*. This program will be used as the platform on which the actual test takes place. In order to start the project on the right foot, we also supply skeletal Jack files for each OS class.

Contract: Write the OS implementation and test it by running all the test programs and testing scenarios described below.

Recommended Testing Strategy

Here are the project materials:

- **Skeletal OS classes:** We supply one Jack file for each OS class. This file includes the “signatures” (interfaces) of all the functions and methods that should be implemented, with empty implementations. Your job is to provide the missing implementations according to the class API and the suggested algorithms.
- **Test programs:** For each OS class, we supply a separate test program written in Jack. In addition, we supply the Jack code of the *Pong* game as a “master” test.

We suggest that each class be developed and unit-tested in isolation. This can be done by compiling the OS class that you write and then putting the resulting `.vm` file in a directory that contains the other 7 `.vm` OS files and the `.vm` files of the respective test program. In particular, after implementing an OS class, you may follow these steps:

- 1) Copy your implemented OS class (Jack file) into the directory that contains the corresponding supplied test program (a collection of one or more Jack files);
- 2) Compile the entire directory using the *supplied* Jack Compiler;
- 3) Copy all the *supplied* OS `.vm` files (except the one that you have just compiled) into the directory.
- 4) At this point the directory should contain an executable program consisting of the eight `.vm` files related to the OS and one `.vm` file for each class in the test program;
- 5) Execute this program by opening the entire directory in the VM Emulator;
- 6) Proceed to test if the OS services are working properly according to the guidelines given below for each OS class.

After testing successfully each OS class in isolation, test your entire OS implementation using the *Pong* game. Put all your OS `.jack` files in the *Pong* directory, compile the directory, and execute the game in the VM Emulator. If the game works, that’s pretty good.

Testing

Memory, Array, Math: In addition to the requisite `.jack` files, the test materials for each of these classes also include a `.tst` test script and a compare `.cmp` file for execution on the VM Emulator. To test your implementation of each one of these three OS classes, execute the given test scripts on the VM Emulator and make sure that the comparison ends successfully. Note that `Memory.alloc` and `Memory.deAlloc` are not fully tested, since a full test depends on internal implementation details not visible in user-level testing. Thus it is recommended to test these two methods using step-by-step debugging in the VM Emulator.

The remaining test programs do not include test scripts. They should be compiled and executed on the VM Emulator as is.

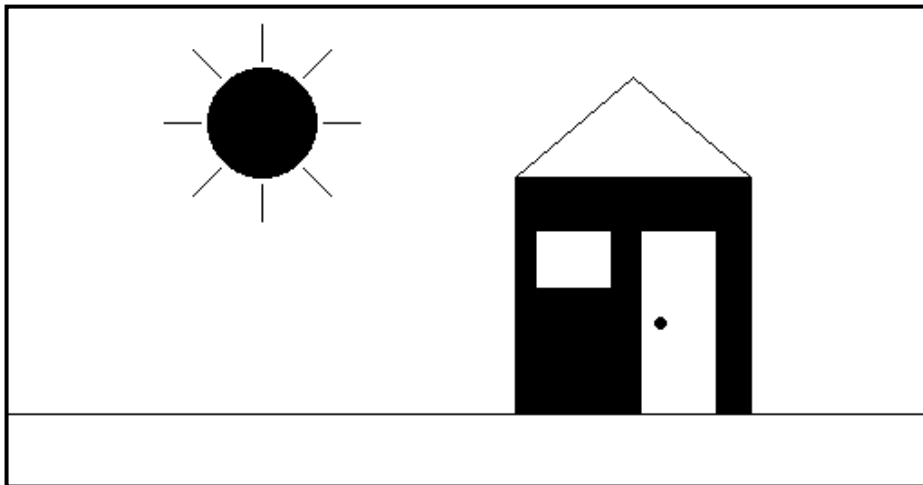
String: Execution of the corresponding test program should yield the following output:

```
new,appendChar: abcde
setInt: 12345
setInt: -32767
length: 12
charAt[2]: 99
setCharAt(2,'-'): ab-de
eraseLastChar: ab-d
intValue: 456
intValue: -32123
backSpace: 129
doubleQuote: 34
newLine: 128
```

Output: Execution of the corresponding test program should yield the following output:

```
AB
0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
!#$%&'()*+,-./:;<=>?@[\\]^_`{|}~
-12346789
CD
```

Screen: Execution of the corresponding test program should yield the following output:



Keyboard: This OS class is tested using a test program that effects some program-user interaction. For each function in the `Keyboard` class (`keyPressed`, `readChar`, `readLine`, `readInt`) the program requests the user to press particular keys on the keyboard. If the function is implemented correctly and the correct keys are pressed, the program prints the text “ok” and proceeds to test the next function. If not, the program repeats the request for the same function. If all requests end successfully, the program prints ‘Test ended successfully’, at which point the screen may look as follows:

```
keyPressed test:
Please press the 'Page Down' key
ok
readChar test:
(Verify that the pressed character is echoed to the screen)
Please press the number '3':
ok
readLine test:
(Verify echo and usage of 'backspace')
Please type 'JACK' and press enter:
ok
readInt test:
(Verify echo and usage of 'backspace')
Please type '-32123' and press enter: ");
ok

Test completed successfully
```

Sys: Only two functions in this class can be tested: `Sys.init` and `Sys.wait`. The supplied test program tests the `Sys.wait` function by requesting the user to press any key, waiting for two seconds (using `Sys.wait`) and then printing another message on the screen. The time that elapses from the moment the key is released until the next message is printed should be two seconds.

The `Sys.init` function is not tested explicitly. However, recall that it performs all the necessary OS initializations and then -- by definition -- calls the `Main.main` function of each test program. Therefore, we can assume that nothing would work properly unless `Sys.init` is implemented correctly. A simple way to test `Sys.init` in isolation is to run *Pong* using your `Sys.vm` file.

Appendix A: Hardware Description Language (HDL)¹

*Intelligence is the faculty of making artificial objects,
especially tools to make tools.*

Henry Bergson, (1859-1941)

A *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. This appendix describes a typical HDL, as understood by the hardware simulator supplied with the book. Chapter 1 provides essential background to this appendix, and thus it is recommended to read it first.

How to use this appendix: This is a technical document, and thus there is no need to read it from beginning to end. Instead, it is recommended to focus on selected sections, as needed. Also, HDL is an intuitive and self-explanatory language, and the best way to learn it is to play with some HDL programs in the hardware simulator. Therefore, we recommend to start experimenting with HDL programs as soon as you can, beginning with the following example.

1. Example

The following HDL program specifies a chip that accepts two 4-bit numbers and outputs whether they are equal or not. The chip logic uses `Xor` gates to compare the 4 bit-pairs, and then outputs true if all the comparisons are “equal”.

```
/** Returns 1 if the two inputs are equal and 0 otherwise. */
CHIP EQ4 {
  IN  a[4],b[4];    // 4-bit busses
  OUT out;          // true iff a=b
  PARTS:
  Xor(a=a[0],b=b[0],out=c0);
  Xor(a=a[1],b=b[1],out=c1);
  Xor(a=a[2],b=b[2],out=c2);
  Xor(a=a[3],b=b[3],out=c3);
  Or(a=c0,b=c1,out=c01);
  Or(a=c01,b=c2,out=c012);
  Or(a=c012,b=c3,out=neq);
  Not(in=neq,out=out);
}
```

Each internal part `Xxx` invoked by an HDL program is in itself a stand-alone chip defined in a separate `Xxx.hdl` program like the one listed above. Thus the chip designer who wrote the above program assumed the existence of three other chips: `Xor.hdl`, `Or.hdl`, and `Not.hdl`.

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

Importantly, though, the chip designer need not worry about *how* these chips are implemented. The internal parts are always viewed as black boxes, allowing the designer to focus only on their proper arrangement in the current chip architecture.

Thanks to this modularity, all HDL programs, including those that describe high-level chips, can be kept short and readable. For example, a complex chip like RAM32K can be implemented using a few internal parts, each described in a single HDL line. When fully evaluated by the hardware simulator all the way down the recursive chip hierarchy, these internal parts are expanded into many thousands of inter-connected elementary logic gates. Yet the chip designer need not be concerned by this complexity, and can focus instead only on the chip's topmost architecture.

Comment: the EQ4.hdl program is not supplied with the book. If you want to experiment with it, you have to create the EQ4.hdl text file and load it into the hardware simulator.

2. Conventions

File Extension: Each chip is defined in a separate text file. A chip whose name is Xxx is defined in file Xxx.hdl.

Chip structure: A chip definition consists of a *header* and a *body*. The header provides a full specification of the chip *interface*, while the body describes the chip *implementation*. The header acts as the chip's API, or public documentation. The body should not interest people who use the chip as internal part in other chip definitions.

Syntax conventions: HDL is case-sensitive. By convention, HDL keywords are written in upper-case letters.

Identifiers naming: Names of chips and pins may be any sequence of letters and digits not starting with a digit. By convention, chip and pin names start with a capital letter and a lower-case letter, respectively. For readability, upper-case letters can be used in the middle of identifier names.

White space: Space characters, newline characters, and comments are ignored.

Comments: The following three comment formats are supported:

```
// comment to end of line
/* comment until closing */
/** API documentation comment */
```

3. Loading Chips into the Simulator

HDL programs (chip descriptions) are loaded into the simulator environment in three different ways. First, the user can open an HDL file interactively, via a "load file" menu or GUI icon. Second, a test script (discussed below) can include a "load Xxx.hdl" command, which has the same effect. Finally, whenever an HDL program is loaded and parsed (e.g. EQ4.hdl), every chip name Yyy listed in it as internal part (e.g. XOR) causes the simulator to load the respective

`Yyy.hdl` file (e.g. `Xor.hdl`), all the way down the recursive chip hierarchy. In every one of these cases, the simulator goes through the following logic:

```

if <chip name>.hdl exists in the current directory
  then load it (and all its descendents) into the simulator
else
  if <chip name>.hdl exists in the simulator's BuiltIn chips directory
    then load it (and all its descendents) into the simulator
  else
    issue an error message.

```

The simulator's `BuiltIn` directory contains executable versions of all the chips specified in the book, except for the highest-level chips (`CPU`, `Memory`, and `Computer`). Hence, one may construct and test a chip before all, or even any, of its lower-level chip parts have been implemented: the simulator will automatically invoke their built-in versions instead. Alternatively, if a lower-level chip `Xxx` has been implemented by the user in HDL, the user can still force the simulator to use its built-in version instead, by simply moving the `Xxx.hdl` file out from the current directory. Finally, in some cases the user (rather than the simulator) may want to load a built-in chip directly, e.g. for experimentation. To do so, navigate to the `BuiltIn` directory – a standard part of the hardware simulator environment -- and select the desired chip from there.

4. Chip Header (Interface)

The header of an HDL program has the following format:

```

CHIP <chip name> {
  IN <input pin name>, <input pin name>, ... ;
  OUT <output pin name>, <output pin name>, ... ;
  // Here Comes the Body
}

```

- **CHIP declaration:** The `CHIP` keyword is followed by the chip name. The rest of the HDL code appears between curly brackets.
- **Input pins:** The `IN` keyword is followed by a comma-separated list of the names of the chip input pins. The list is terminated with a semi-colon.
- **Output pins:** The `OUT` keyword is followed by a comma-separated list of the names of the chip output pins. The list is terminated with a semi-colon.

Input and output pins are assumed by default to be single-bit wide. A multi-bit *bus* can be declared using the notation `<pin name> [w]`. (e.g. “a [4]” in `EQ4.hdl`). This specifies that the pin is a bus of width w . The individual bits in a bus are indexed $0 \dots w-1$, from right to left (i.e. index 0 refers to the least significant bit).

5. Chip Body (Implementation)

Parts

A chip typically consists of several lower-level chips, connected to each other and to the chip input/output pins in a certain topology that forms the chip logic. This logic, designed by the HDL programmer to deliver the chip's desired functionality, is described in the chip *body* using the following format:

```
PARTS:
<internal chip part>;
<internal chip part>;
. . .
<internal chip part>;
```

Each one of these statements describes one internal chip with all its connections, using the following syntax:

```
<chip name>(<connection>, ... , <connection>);
```

Throughout this document, the presently defined chip is called *chip*, and the lower level chips listed in the PARTS section are called *parts*.

Pins and Connections

The syntax of a *connection* specification is:

```
<part's pin name> = <chip's pin name>
```

Connections describe how a part is connected to the overall chip architecture: each connection describes how one pin of the part is connected to another pin in the chip definition. In the simplest case, one may connect the part's pin to an input or output pin of the chip. In other cases, we have to connect the part's pin to another pin of another part. This is done by defining an *internal pin* (a chip level object), and connecting the pins of the two parts to it. Thus, the definition of an internal pin is essentially the same as creating and naming a wire that connects an output pin of one chip to the input pin of another.

Internal pins: In order to connect an output pin of *Part1* to the input pins of other parts, the HDL programmer can create and use an *internal pin*, say *v*, as follows:

```
Part1(...,out=v);      // out of Part1 is piped into v
Part2(in=v,...);      // v is piped into in of Part2
Part3(a=v, b=v,...);  // v is piped into a and b of Part 3
```

An internal pin (like *v* above) acts like a pipe that receives a value from one part and feeds it into one or more other parts. Internal pins are created as needed when they are specified the first time in the HDL program, and require no other definition. Each internal pin has fan-in 1 and unlimited fan-out. In other words, an internal pin can be fed from a single source only, yet it can feed (through multiple connections) many other parts. In the above example, the internal pin *v* simultaneously feeds both *Part2* (through *in*) and *Part3* (through *a* and *b*).

Input pins: Each input pin of a part may be fed by one of the following sources:

- An input pin of the chip;
- An internal pin;
- One of the constants `true` and `false`, representing 1 and 0, respectively.

Each input pin has fan-in 1, meaning that it can be fed by one source only. Thus `Part(a=v,b=v,...)` is a valid statement (assuming that both `a` and `b` are input pins of the part), whereas `Part(a=v,a=u,...)` is not.

Output pins: Each output pin of a part may feed one of the following destinations:

- An output pin of the chip;
- An internal pin.

Buses

Each pin used in a connection -- whether input, output, or internal -- may be a *multi-bit bus*. The *widths* (number of bits) of input and output pins are defined in the chip header. The widths of internal pins are deduced implicitly by the simulator, as explained below.

In order to connect individual elements of a multi-bit bus input or output pin, the pin name (say `x`) may be sub-scripted using the syntax `x[n..m]=v`, where `v` is an internal pin. This means that only the bits indexed `n` to `m` (inclusive) of pin `x` are connected to the specified internal pin. An internal pin (like `v` above) may not be subscripted, and its width is deduced implicitly from the width of the bus pin to which it is connected the first time it is mentioned in the HDL program.

The constants `true` and `false` may also be used as buses, in which case the required width is deduced implicitly from the context of the connection.

Example: Consider the following chip:

```
CHIP Foo {
    IN in[8]          // 8-bit input
    OUT out[8]       // 8-bit output
    // Foo's body (irrelevant to the example)
}
```

Suppose now that `Foo` is invoked by another chip using the part statement:

```
Foo(in[2..4]=v, in[6..7]=true, out[0..3]=x, out[2..6]=y)
```

Where `v` is a previously declared 3-bit internal pin, bound to some values. In that case, the connections `in[2..4]=v` and `in[6..7]=true` will bind the `in` bus of the `Foo` chip to the following values:

in:	Bit:	7	6	5	4	3	2	1	0
	Contents:	1	1	?	v[2]	v[1]	v[0]	?	?

Now, let us assume that following its processing, the `F00` chip returns the following set of values (an arbitrary assumption):

out:	Bit:	7	6	5	4	3	2	1	0
	Contents:	1	1	0	1	0	0	1	1

In that case, the connections `out[0..3]=x` and `out[2..6]=y` will yield:

x:	Bit:	3	2	1	0
	Contents:	0	0	1	1

y:	Bit:	4	3	2	1	0
	Contents:	1	0	1	0	0

6. Built-In Chips

The hardware simulator features a library of built-in chips that can be used as internal parts by higher-level chips. Built-in chips are implemented in code written in a programming language like Java, operating behind an HDL interface. Thus, a built-in chip has a standard HDL header (interface), but its HDL body (implementation) declares it as built-in. For example, consider the following built-in version of a typical `Register` chip:

```
/** 16-bit register.
If load[t]=1 then out[t+1]=in[t] else out does not change */
CHIP Register {
    IN in[16], load;
    OUT out[16];
    BUILTIN Register; // refers to register.class, a compiled Java class
    CLOCKED in, load; // this command is explained later in this appendix
}
```

The identifier following the keyword `BUILTIN` is the name of the program unit that delivers the chip functionality. The present version of the hardware simulator is built in Java, and all the built-in chips are implemented as compiled Java classes. Hence, the HDL body of a built-in chip has the following format:

```
BUILTIN <Java class name>;
```

Where `<Java class name>` is the name of the Java class that models the intended chip behavior. Normally, this class will have the same name as that of the chip, e.g. `Register.class`. All the built-in chips (compiled Java class files) are stored in a directory called `BuiltIn`, which is a standard part of the simulator's environment.

Built-in chips provide three special services:

- **Foundation:** Some chips are the atoms from which all other chips are built. In particular, we use the `Nand` gate and the `D-Flip-Flop` gate as the building blocks of all combinational and sequential chips, respectively. Thus the hardware simulator features built-in versions of `Nand.hdl` and `DFF.hdl`.
- **Certification & Efficiency:** In order to modularize the development and testing of hardware construction projects, the chips that participate in the project can be made available in built-in versions. Thus one may construct a chip before constructing its lower-level parts – the simulator will automatically invoke their built-in versions. Additionally, it makes sense to use built-in versions even for chips that were already constructed, since the former are typically much faster and more space-efficient than the latter (simulation-wise). For example, consider a `RAM4K` chip. When you write and debug the file `RAM4K.hdl`, the simulator creates a memory-resident data structure consisting of thousands of lower-level chips, all the way down to the `D-Flip-Flop` gates at the bottom of the recursive chip hierarchy. Although this top-down drilling must be done when you *develop* and *test* the `RAM4K` chip, there is no need to repeat it each time the chip is used as part in higher-level chips, e.g. `RAM16K`. **Best practice tip:** To boost performance and minimize errors, always use the supplied built-in versions of chips whenever they are available.
- **Visualization:** Some high-level chips, e.g. memory units, are easier to understand and debug if their operation can be inspected visually. To facilitate this service, selected built-in chips can be endowed with GUI side effects. This GUI is displayed whenever the chip is loaded into the simulator or invoked as a lower-level part by the loaded chip. Except for these visual side effects, GUI-empowered chips behave, and can be used, just like any other chip. Section 8 contains more details about GUI-empowered chips.

7. Sequential Chips

Computer chips are either *combinational* or *sequential* (also called *clocked*). The operation of combinational chips is instantaneous. Thus, when a user or a test script changes the values of one or more of the input pins of a combinational chip and presses the “eval” button, the simulator responds by immediately setting the chip output pins to a new set of values, as computed by the chip logic. In contrast, the operation of sequential chips is clock-regulated. In particular, when the inputs of a sequential chip change, the outputs of the chip may change to new values only at the beginning of the next time unit, as effected by the simulated clock.

In fact, sequential chips may change their output values when the time changes even if none of their inputs changed. In contrast, combinational chips never change their values just because of the progression of time.

The Clock

The simulator models the progression of time by a built-in device, called *clock*, which is controlled by “Tick” and “Tock” operations. These operations generate a series of *time units*, each consisting of *two phases*: a “Tick” ends the first phase of a time unit and starts its second phase, and a “Tock” moves to the first phase of the next time unit. The *real time* that elapsed during this period is irrelevant for simulation purposes, since we have full control over the clock. In other words, either the simulator’s user or a test script can issue *Ticks* and *Tocks* at will, causing the clock to generate a series of simulated time units.

The two-phased time units regulate the operations of *all* the sequential chip parts in the simulated chip architecture, as follows. During the first phase of the time unit (*Tick*), the inputs of each sequential chip in the architecture are read and affect the chip’s internal state, according to the chip logic. During the second phase of the time unit (*Tock*), the outputs of the chip are set to the new values. Hence, if we look at a sequential chip “from the outside,” we see that its output pins stabilize to new values only at “Tocks” – between consecutive time units.

There are two ways to control the simulated clock: manual and script-based. First, the simulator's GUI features a clock-shaped button called “TickTock”. A “Tick” (one click on this button) ends the first phase of the clock cycle, and a “Tock” (subsequent click) ends the second phase of the cycle, bringing on the first phase of the next cycle, and so on. Alternatively, one can run the clock from a test script, e.g. using the command `repeat n {tick,tock,output;}`. This script command instructs the simulator to advance the clock *n* time units, and to print some values in the process. Test scripts and commands like `repeat` and `output` are described in detail in appendix B.

Clocked Chips and Pins

A built-in chip can declare its dependence on the clock explicitly, using the statement:

```
CLOCKED <pin>, <pin>, ..., <pin>;
```

Where each `pin` is either an input pin or an output pin, as declared in the chip header. The inclusion of an *input pin* *x* in the `CLOCKED` list instructs the simulator that changes to *x* should not effect any of the chip’s output pins until the beginning of the next time unit. The inclusion of an *output pin* *x* in the `CLOCKED` list instructs the simulator that changes in any of the chip’s input pins should not effect *x* until the beginning of the next time unit. Note that it is quite possible that only some of the input or output pins of a chip are declared as clocked. In that case, changes in the non-clocked input pins may affect the non-clocked output pins in a combinational manner, i.e. independent of the clock. In fact, it is also possible to have the `CLOCKED` keyword with an empty list of pins, signifying that even though the chip may change its internal state depending on the clock, changes to any of its input pins may cause immediate changes to any of its output pins.

The “clocked” property of chips: A primitive (built-in) chip is said to be explicitly clocked when it includes a `CLOCKED` statement. A composite (not built-in) chip is said to be implicitly clocked when one or more of its lower-level chip parts are clocked. This property is checked recursively, all the way down the chip hierarchy, where a built-in chip may be explicitly clocked. If such a chip is found, it renders every chip that depends on it (up the hierarchy) implicitly clocked. It follows that nothing in the HDL code of a composite chips suggest that they may be clocked – the user can know that only from the chip documentation.

Example: The built-in D-Flip-Flop chip is defined as follows:

```
/** Clocked D-Flip-Flop. out[t+1]=in[t] */
CHIP DFF {
  IN in;
  OUT out;
  BUILTIN DFF; // implemented by a DFF.class Java program.
  CLOCKED in,out; }
```


Every sequential chip in our computer architecture depends in one way or another on (typically numerous) DFF chips. For example, the RAM64 chip is made up from eight RAM8 chips. Each one of these chips is made from eight lower-level Register chips. Each one of these registers is made from many Bit chips. And each one of these chips contains a DFF part. It follows that Bit, Register, RAM8, RAM64 (and all the memory units above them) are also *clocked*, or *sequential*, chips.

It's important to remember though that a sequential chip may well contain combinational logic which is not effected by the clock. For example, the structure of every sequential RAM chip includes combinational circuits that manage its addressing logic (described in Chapter 3).

Feedback Loops

We say that the use of a chip entails a feedback loop when the output of one of its parts affects the input of the same part, either directly or through some (possibly long) path of dependencies. For example, consider the following two examples of direct feedback dependencies:

```
Not(in=loop1, out=loop1) // invalid
DFF(in=loop2, out=loop2) // valid
```

In both examples, an internal pin (`loop1` and `loop2`) attempts to feed the chip's input from its output, creating a cycle. The difference between the two examples is that `Not` is a *combinational* chip whereas `DFF` is *sequential*, or *clocked*. Thus, `loop1` creates an instantaneous and uncontrolled dependency, whereas the dependency that `loop2` creates is delayed by the clock dependency of the underlying pins (as defined in the `DFF` logic). In general, we have the following:

Valid/invalid Feedback loops: When the simulator loads a chip, it checks recursively if its various connections entail feedback loops. For each loop, the simulator checks if the loop goes through a clocked pin, somewhere along the loop. If so, the loop is allowed. Otherwise, the simulator stops processing and issues an error message. This is done in order to avoid the uncontrolled “data races” that occur in combinational feedback loops from outputs to inputs. The only way to fix a chip with a combinational feedback loop is to redesign its logic.

8. Visualizing Chip Operations

Built-in “GUI-empowered” chips feature visual side effects, designed to illustrate internal chip operations using graphics and animation. Like any other chip, a GUI-empowered chip can come to play in two possible ways. First, the user can load it directly into the simulator. Second, and more typically, the built-in chip is invoked by the simulator automatically, whenever it is used as *part* in more complex chips. In both cases, the simulator displays the chip's graphical image on the screen. Using this image, which is actually an executable GUI component, the user may inspect the current contents of the chip as well as change its internal state, when this operation is supported by the chip implementation.

The present version of the simulator features the following set of GUI-empowered chips:

ALU: Displays the ALU's inputs and output as well as the presently computed function.

Registers (`ARegister` -- address register, `DRegister` -- data register, and `PC` -- program counter): Displays the contents of the registers and allows to modify them.

Memory chips (`RAM` and `ROM`): Displays a scrollable array-like image that shows the contents of all the memory locations, and allows the user to manipulate them. If the contents of a memory location change during the simulation, the respective entry in the GUI changes as well. In the case of the `ROM` chip (which serves as the instruction memory of our computer platform), the GUI also features a button that enables the user to load into it a machine language program from an external text file.

I/O chips (`Screen` and `Keyboard`): If the HDL code of a loaded chip invokes the built-in `Screen` chip, the hardware simulator displays a 256 rows by 512 columns window that simulates the physical screen. When the RAM-resident memory-map of the screen changes during the simulation, the respective pixels in the screen GUI change as well, via a "refresh logic" embedded in the simulator implementation.

If the HDL code of a loaded chip invokes the built-in `Keyboard` chip, the simulator displays a clickable keyboard icon. Clicking this button connects the real keyboard of your computer to the simulated chip. From this point on, every key pressed on the real keyboard is intercepted by the simulated chip. If the user moves the mouse focus to another area in the simulator GUI, the control of the keyboard is restored to the real computer.

Example: To illustrate how the simulator deals with GUI-empowered chips, consider the following HDL program, which uses the built-in chips `RAM16K`, `Screen`, and `Keyboard`:

```
// Demo of GUI-empowered chips.
// The logic of this chip is meaningless, and is used merely to
// force the simulator to display some GUI-empowered chips.
CHIP GUIDemo {
    IN in[16],load, address[15];
    OUT out[16];
    PARTS:
    RAM16K(in=in,load=load,address=address[0..13],out=a);
    Screen(in=in,load=load,address=address[0..12],out=b);
    Keyboard(out=c);
}
```

The chip logic feeds the 16-bit `in` value into two destinations: register number `address` in the `RAM16K` chip and register number `address` in the `Screen` chip (presumably, the HDL programmer who wrote this code has figured out the widths of these address pins from the API's of these chips). In addition, the chip logic routes the value of the currently pressed keyboard key to the internal pin `c`. These meaningless operations are designed for one purpose only: illustrating how the simulator deals with built-in GUI-empowered chips. The actual impact is shown in Figure 1.

The screenshot shows the Hardware Simulator (1.1b) interface. The main window displays the chip name 'GUIDemo (Clocked)' and a time of 3. The interface is divided into several sections:

- Input pins:** A table showing the values for 'in[16]', 'load', and 'address[15]'. The 'address[15]' value is 5012.
- Output pins:** A table showing the value for 'out[16]', which is 0.
- HDL:** A code editor showing the HDL program for 'demo GUI-empowered chips'. The program defines a chip 'GUIDemo' with input pins 'in[16]', 'load', and 'address[15]', and output pins 'out[16]'. It includes three built-in GUI-empowered chips: 'RAM16K', 'Screen', and 'Keyboard'.
- Internal pins:** A table showing the values for 'a[16]', 'b[16]', and 'c[16]'. The 'a[16]' value is -1.
- RAM 16K:** A memory map showing the values for addresses 5009 through 5015. The value at address 5012 is -1.
- GUIs:** The simulator displays the GUIs for the built-in chips. The 'Screen' chip GUI shows a horizontal line, and the 'Keyboard' chip GUI shows a keyboard icon. The 'RAM16K' chip GUI shows a memory map with the value -1 at address 5012.

Four callouts are present in the image:

- 1. HDL code invokes 3 built-in GUI-empowered chips**: Points to the HDL code in the code editor.
- 2. GUI of built-in Screen chip**: Points to the GUI of the built-in Screen chip.
- 3. The simulator user enters test values into the chip input pins**: Points to the 'address[15]' input pin value.
- 4. outputs**: Points to the 'out[16]' output pin value and the 'RAM 16K' memory map.

FIGURE 1: GUI-empowered Chips. Since the loaded HDL program uses GUI-empowered chips as internal parts (step 1), the simulator draws 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). The tiny horizontal line (circled) is the visual side effect of storing -1 in memory location 5012. Since the 16-bit 2's complement binary code of -1 is 1111111111111111, the computer draws 16 pixels starting at the 320th column of row 156, which happen to be the screen coordinates associated with address 5012 of the memory map (the exact memory-to-screen mapping is given in Chapter 4).

9 List of Built-In Chips

Name	Specified in chapter	Has GUI	Comment
Nand	1		Foundation of all combinational chips
Not	1		
And	1		
Or	1		
Xor	1		
Mux	1		
DMux	1		
Not16	1		
And16	1		
Or16	1		
Mux16	1		
Or8way	1		
Mux4way16	1		
Mux8way16	1		
DMux4way	1		
DMux8way	1		
HalfAdder	2		Foundation for all sequential chips
FullAdder	2		
Add16	2		
ALU	2	<input checked="" type="checkbox"/>	
Incl6	2		
DFF	3		
Bit	3		
Register	3		
ARegister	3	<input checked="" type="checkbox"/>	
DRegister	3	<input checked="" type="checkbox"/>	
RAM8	3	<input checked="" type="checkbox"/>	Identical operation to Register, with GUI
RAM64	3	<input checked="" type="checkbox"/>	
RAM512	3	<input checked="" type="checkbox"/>	
RAM4K	3	<input checked="" type="checkbox"/>	
RAM16K	3	<input checked="" type="checkbox"/>	
PC	3	<input checked="" type="checkbox"/>	
ROM32K	5	<input checked="" type="checkbox"/>	
Screen	5	<input checked="" type="checkbox"/>	GUI allows loading a program from a text file
Keyboard	5	<input checked="" type="checkbox"/>	GUI connects to a window on the actual screen
			GUI connects to the actual keyboard

TABLE 2: All the built-in chips supplied with the present version of the Hardware Simulator.
A built-in chip has an HDL interface but is implemented as an executable Java class.

In future versions of the simulator, we plan to release a simulator-extension API, allowing programmers to write additional built-in chip implementations. Needless to say, the present simulator version can still execute any desired chip *written in HDL*; the ability to create *built-in chip implementations in other languages* is an optional luxury designed to add GUI effects, improve execution speed, and facilitate behavior chip simulation before it is built in HDL.

Appendix B: Test Scripting Language¹

Mistakes are the portals of discovery

(James Joyce, 1882-1941)

Appendix A described how to *define* and *simulate* chips. This appendix describes how to *test* and *debug* chip definitions. This is done by subjecting the HDL program representing the chip to various tests. The tests can be administered in an ad-hoc and interactive fashion, using the simulator GUI, or in a pre-planned and batch-style fashion, following a series of tests specified in a script file. This appendix describes the language in which *test scripts* are written, as understood by the hardware simulator supplied with this book. Chapter 1 provides essential background to this subject, and thus it is recommended to read it first.

The scripts that test a newly designed chip can be written either by the person who implements the chip, or by the hardware architect who ordered the chip done and specified its interface. As a matter of best practice, we recommend to use both approaches. Indeed, for every chip specified in the book, we provide an “official” test script, written by us. Thus, although you are welcome to test your chip designs in any way you see fit, the contract is such that eventually, *your* chip definitions have to pass *our* tests.

How to use this appendix: This is a technical document, and thus there is no need to read it from beginning to end. Instead, it is recommended to focus on selected sections, as needed. Like HDL, the test scripting language is rather intuitive, and the best way to learn it is to play with some sample scripts in the hardware simulator.

1. Example

The following script is designed to test the EQ4 chip described in section A.1.

```
load EQ4.hdl,           // load the HDL program into the simulator.
output-file EQ4.out,   // write the script outputs to this file.
compare-to EQ4.cmp,   // compare the script outputs to this file.
output-list a b out;  // each subsequent output command should
                    // print the values of a,b and out.
set a %B0000, set b %B0000, eval, output;
set a %B1111, set b %B1111, eval, output;
set a %B1111, set b %B0000, eval, output;
set a %B1111, set b %B0000, eval, output;
set a %B0000, set b %B1111, eval, output;
set a %B0001, set b %B0000, eval, output;
set a %B0101, set b %B0010, eval, output;
// Since the chip has two 4-bit inputs, an exhaustive test
// requires 2^4*2^4=256 such scenarios.
```

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

A test script normally starts with some initializations commands, followed by a series of *simulation steps*, each ending with a semicolon. A simulation step typically instructs the simulator to bind the chip input pins to some test values, evaluate the chip outputs, and write selected variable values into a designated output file. This logic is affected by the hardware simulator, as illustrated in Fig. 1.

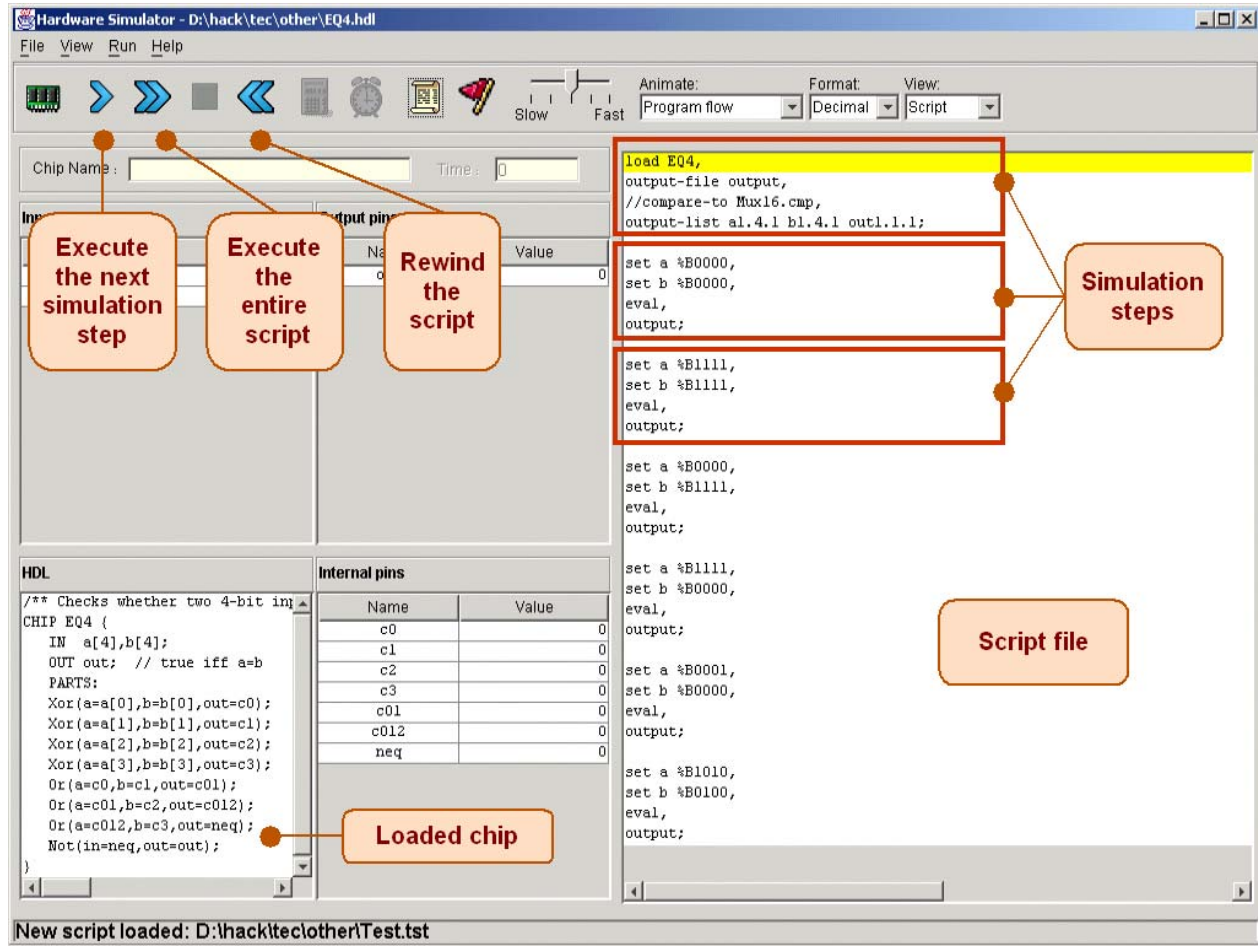


FIGURE 1: Running a test script. Each test script *command* ends with a comma. A sequence of commands that ends with a semicolon constitutes a *simulation step* (The beginning of the *current simulation step* is highlighted with a yellow bar). The user controls the script execution by clicking the VCR buttons on the top left. Note that this particular script starts by loading the EQ4.hdl chip description into the simulator.

2. Conventions

File Extension: Test scripts are stored in text files with .tst extensions. Typically (but not necessarily), a script designed to test an xxx.hdl program will be named xxx.tst. Note however that the same chip can be tested by more than one script.

Current directory: The practice of developing and testing a chip involves at least one and typically four text files: the mandatory chip description file (.hdl), a test script file (.tst), a script-generated output file (.out), and a supplied compare-to file (.cmp). All these files should be kept in the same directory on the user's computer.

The *current directory* is defined as the *directory that contains the last file (whether chip or script) opened by the user from the simulator environment*.

Built-in chips: The built-in chips (residing in the simulator's `BuiltIn` directory) can be opened and tested by regular scripts, just like any other chips. All the built-in chips supplied with the simulator were rigorously tested, but one can experiment and re-test them for instructive purposes.

White space: Space characters, newline characters, and comments are ignored.

Comments: The following comment formats are supported:

```
// comment to end of line
/* comment until closing */
/** API documentation comment */
```

3. Data Types and Variables

Test scripts support one data type: integers. Integer constants can be expressed in hexadecimal (`%X` prefix), binary (`%B` prefix), or decimal (`%D` prefix) format, the latter being the default. These values are always translated into their equivalent 2's complement binary values. For example, the four commands `set a1 %XFFFF`, `set a2 %B11`, `set a3 %D11`, `set a4 -1` will set the respective variables to the binary values `1111111111111111`, `11`, `1011`, and `1111111111111111`.

Script commands can manipulate three types of variables: *pins*, *variables of built-in chips*, and the system variable *time*.

Pins: Script commands can access the current values of all the input, output, and internal pins of the simulated chip. For example, the command `set in 0` sets the value of the pin whose name is `in` to `0`.

Variables of built-in chips: External implementations of built-in chips can expose internal variables that can be manipulated by test scripts. We delay the discussion of these variables to Section 7.

Time: A read-only system variable, representing the number of time-units that elapsed since the simulation started running. Each rise and fall of the clock (a clock cycle, also known in our jargon as "TickTock") constitutes a single time-unit.

4. Command Structure

Command Terminators: A script is a sequence of commands. Each command is terminated by a comma, a semicolon, or an exclamation mark. These terminators have the following semantics:

- Comma (`,`): terminates a script command.

- Semi-colon (;): terminates a script command and a simulation step. A simulation step consists of one or more script commands. When the user instructs the simulator to "single-step" via the GUI, the simulator executes all the script logic from the current command until a semi-colon is reached, at which point the simulation is paused.
- Exclamation mark (!): terminates a script command and causes the simulator to stop the script execution. The user can later resume the script execution from that point onwards.

Command Syntax: Each command is written as a sequence of white space-separated identifiers. Identifiers are composed of any symbol except for terminators or white space. The script is not case sensitive.

It is convenient to describe the script commands in two conceptual sections. "Set up commands" are used to load files and initialize some global settings. "Simulation commands" walk the simulator through a series of tests. We now turn to describe these two categories of commands.

5. Set Up commands

Load *<hdl file>*: Loads the file into the simulator. The *<hdl file>* must include the `.hdl` extension and must not contain a path specification. The simulator will try to load the file from the current directory, and, failing that, from the simulator's `BuiltIn` directory, as described in Section A.3 (Appendix A).

Output-file *<file name>*: Instructs the simulator to write further output to the named file, which must include an `.out` extension. The output file will be created in the current directory.

Output-list *<v1, v2, ... >*: Instructs the simulator what to write to the output file in every subsequent output command in this script (until the next `output-list` command, if any). Each value in the list is a variable name followed by a formatting specification.. The command also produces a single header line consisting of the variable names. Each item *v* in the output-list has the following syntax:

```
<variable name><format><padL>.<len>.<padR>
```

This directive instructs the simulator to write `padL` spaces, then the current value of `variable name` in the specified `format` using `len` columns, then `padR` spaces, then the symbol "|". `Format` can be either `%B` (binary), `%X` (hexa), or `%D` (decimal). The default output specification is `<variable name>%B1.1.1`.

For example, consider the command:

```
Output-list time%D0.5.2 reset%B1.1.1 PC%D2.3.2 A%X2.4.2 D%X2.4.2
```

This command may produce the following output (after two subsequent output commands):

	Time		reset		PC		A		D	
	37		0		21		001F		AA80	
	38		0		31		001F		AA80	

Compare-to *<file name>*: Instructs the simulator that each subsequent output line should be compared to its corresponding line in the specified comparison file (which must include the `.cmp` extension). If any two lines are not the same, the simulator displays an error message and halts the script execution. The compare file is assumed to be present in the current directory.

6. Simulation Commands

Set *<variable name>* *<value>*: Assigns the *value* to the *variable*. The *variable* is either a pin or an internal variable of the simulated chip. The magnitude of *value* must match the *variable*'s width. For example, if *x* is a 16-bit pin and *y* is a single bit pin, then `set x 153` is valid whereas `set y 153` will yield an error and halt the simulation.

Eval: Instructs the simulator to apply the chip logic to the current values of the input pins and compute the resulting output values.

Output: Let us assume that a *compare* file has been previously declared via the `compare-to` command. The `output` command causes the simulator to go through the following logic:

1. Get the current values of all the variables listed in the last `output-list` command;
2. Create an output line using the format specified in the last `output-list` command;
3. Write the output line to the *output file*;
4. If the output line differs from the current line of the *compare* file, display an error msg;
5. Advance the line cursors of the *output* file and the *compare* file.

Tick: Ends the first phase of the current clock cycle (time unit).

Tock: Ends the second phase of the current time unit, and advances to the first phase of the next time unit.

Repeat *num* {*commands*}: instructs the simulator to repeat the sequence of script *commands* enclosed in the curly brackets *num* times. If *num* is omitted, the simulator repeats the *commands* until the simulation has been stopped for some other reason.

While *<Boolean condition>* {*commands*}: instructs the simulator to repeat the *commands* inside the curly brackets as long as the *Boolean condition* is true. The condition is of the form *<x op y>* where *x* and *y* are either constants or variable names, and *op* is one of the following: `=`, `>`, `<`, `>=`, `<=`, `<>`.

Echo *<text>*: instructs the simulator to print the *text* in the status line (part of the simulator GUI). The text must be enclosed in `""`.

Clear-echo: instructs the simulator to clear the status line.

<Built-in chip name> **<method name>** **<argument(s)>**: Built-in chip implementations can expose methods that perform chip-specific operations. The syntax of these operations varies from one built-in chip to another and is documented in their API's. This command option is described in detail in the next section.

7. Internal variables and methods of built-in chips

External implementations of built-in chips can expose internal variables via the syntax `chipName[varName]`, where `varName` is an implementation-specific variable. The exposed variables (if any) of the built-in chip should be documented in the chip's API. For example, consider our built-in version of a RAM16K chip. The API of this memory chip documents that any individual location in it can be accessed via the syntax `RAM16K[i]`, where i is between 0 to 16383. Such internal variables can be manipulated by test scripts, using standard SET commands like `"set RAM16K[1017] 15"`. This particular command will set memory location number 1017 to the 2's complement binary value of 15. In addition, since the built-in RAM16K chip has GUI side effects, the new value will also be displayed on the chip's visual image.

If a built-in chip maintains an *internal state* (as in sequential chips), the current value of the state can be accessed through the convention `chipName[]`, but only if the internal state can be represented by a single-value variable. For example, when simulating the built-in Register chip, one can write script commands like `set Register[] -5324`. This command sets the internal state of the chip to the 2's complement binary value of -5324. In the next time unit, the out pin of the Register chip will start to emit this value.

Built-in chips can also expose implementation-specific *methods* that can be used in scripts as commands. For example, in the hardware platform specified in this book, programs reside in an Instruction Memory unit implemented by a chip called ROM32K. The contract is such that before one runs a machine language program on this computer, one must first load a program into its Instruction Memory. In order to facilitate this service, our built-in implementation of the ROM32K chip features a "load <file name>" method, where the <file name> argument is a text file that, hopefully, contains machine language instructions. This chip-specific method can be accessed by test scripts, via commands like "ROM32K load myprog.hack". Presently, this is the only method supplied by any of our built-in chips.

Chip Name	Variable name/s	Width/Data range
Register	Register[]	16-bit (-32768...32767)
ARegister	Aregister[]	16-bit
DRegister	Dregister[]	16-bit
PC (program counter)	PC[]	15-bit (0..32767)
RAM8	RAM8[0..7]	Each entry is 16-bit
RAM64	RAM64[0..63]	"
RAM512	RAM512[0..511]	"
RAM4K	RAM4K[0..4095]	"
RAM16K	RAM16K[0..16383]	"
ROM32K	ROM32K[0..32767]	"
Screen	Screen[0..16383]	"
Keyboard	Keyboard[]	16-bit, read-only

TABLE 2: Exposed internal variables of all the built-in chips supplied with the Hardware Simulator. These variables can be manipulated from test scripts.

8. Ending Example

We end the appendix with a relatively complex test script, designed to test the top-most `Computer` chip. One way to test the `Computer` chip is to load a program into it and monitor the outputs of the various hardware components as the computer executes the program, one instruction at a time. For example, we wrote a program that (hopefully) computes the maximum of `RAM[0]` and `RAM[1]` and writes the result to `RAM[2]`. The machine language version of this program is stored in the text file `max.hack`. Note that at the very low level in which we operate, if such a program does not run properly it can be either because the *program* has bugs or the *hardware* has bugs (and, for completeness, it may also be that the *test script* or the *hardware simulator* have bugs). For simplicity, let us assume that the program is error-free.

To test the `Computer` chip using this program, we wrote a test script called `ComputerMax.tst`. This script loads the `Computer` chip into the hardware simulator and then loads the `max.hack` program into its ROM chip. A reasonable way to check if the chip works properly is as follows: put some values in `RAM[0]` and `RAM[1]`, reset the computer, run the clock, and inspect `RAM[2]`. This, in a nutshell, is what the supplied test script is designed to do:

```
/* ComputerMax.tst script.
   The max.hack program computes the maximum of
   RAM[0] and RAM[1] and writes the result in RAM[2].
*/
// Load the chip, and set up for the simulation
load Computer.hdl, output-file Computer.out,
compare-to ComputerMax.cmp,
output-list RAM16K[0] RAM16K[1] RAM16K[2];
// Load the max.hack program into the built-in ROM32K chip
ROM32K load max.hack,
// set the first 2 cells of the built-in RAM16K chip to some test values
set RAM16K[0] 3,
set RAM16K[1] 5,
output;
// run the clock enough cycles to complete the program's execution
Repeat 14 {
    tick,tock,
    output;
}
// Reset the Computer
set reset 1,
tick,          // run the clock in order to commit the Program
tock,         // Counter (a sequential chip) to the new reset value
output;
// Re-run the same program with different test values
set reset 0,   // "de-reset" the computer (committed in next tick-tock)
set RAM16K[0] 23456,
set RAM16K[1] 12345,
output;
Repeat 14 {
    tick,tock,
    output;
}
```

Note: We know that 14 cycles are sufficient to execute this program by trial and error, since we've experimented with this script before.