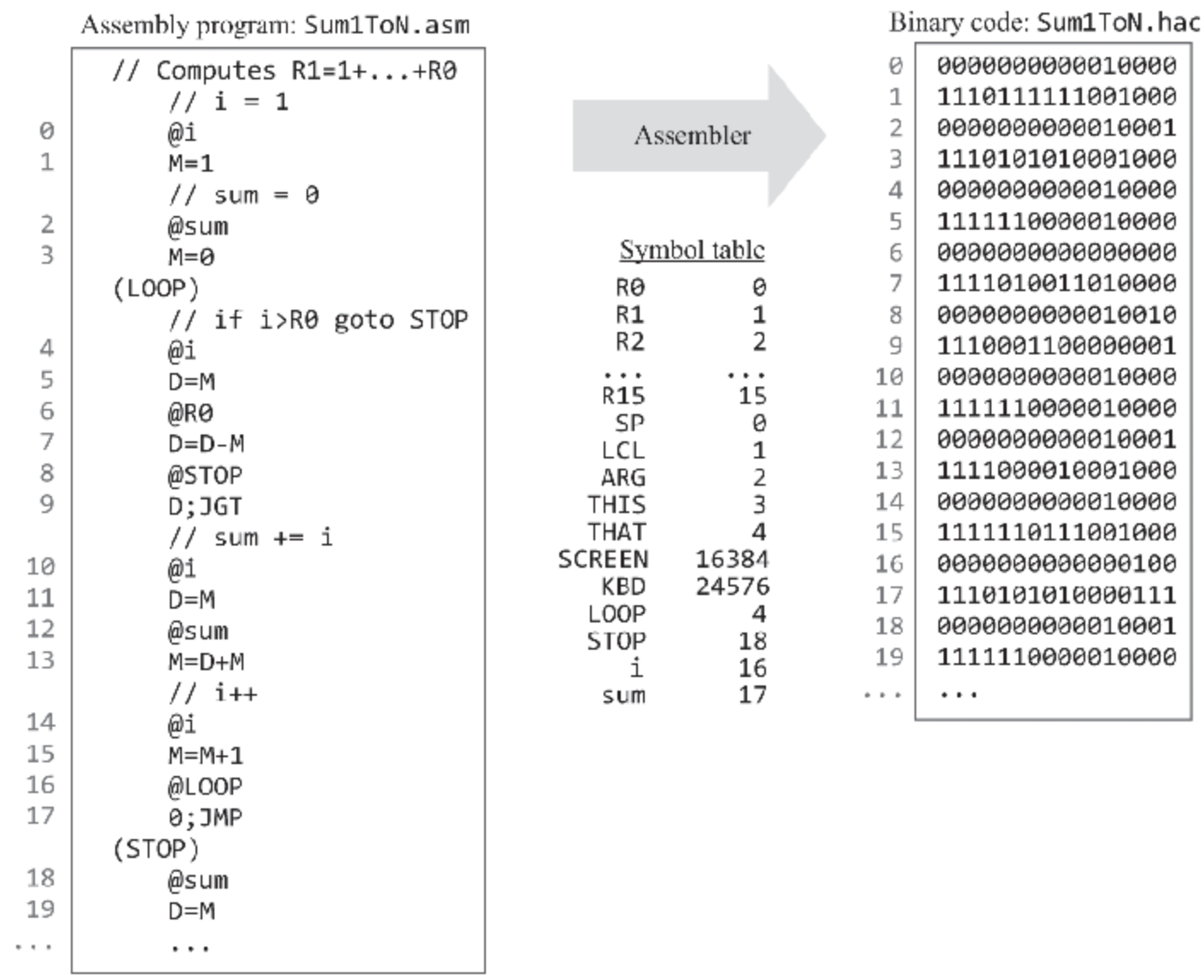


Figure 6.1 Assembly code, translated to binary code using a symbol table. The line numbers, which are not part of the code, are listed for reference.

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

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

6.2 The Hack Machine Language Specification

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

6.2.1 Programs

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

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

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

A-instruction

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

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

C-instruction

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

(binary): `111`*accccccdddjjj*

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

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

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

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

6.2.2 Symbols

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

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

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

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

6.2.3 Syntax Conventions

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

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

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

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

This completes the Hack machine language specification.

6.3 Assembly-to-Binary Translation

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

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

6.3.1 Handling Instructions

For each assembly instruction, the assembler

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

6.3.2 Handling Symbols

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

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

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

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

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

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

6.4 Implementation

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

prompt> HackAssembler *Prog*.asm

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

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

6.4.1 Developing a Basic Assembler

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

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

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

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

The Parser

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

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

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

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

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

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

The Code Module

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

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

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

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

The Hack Assembler

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

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

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

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

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

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

6.4.2 Completing the Assembler

The Symbol Table

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

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

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

6.5 Project

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

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

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

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

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

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

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

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

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

put a nonnegative value in R0.

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

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

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

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

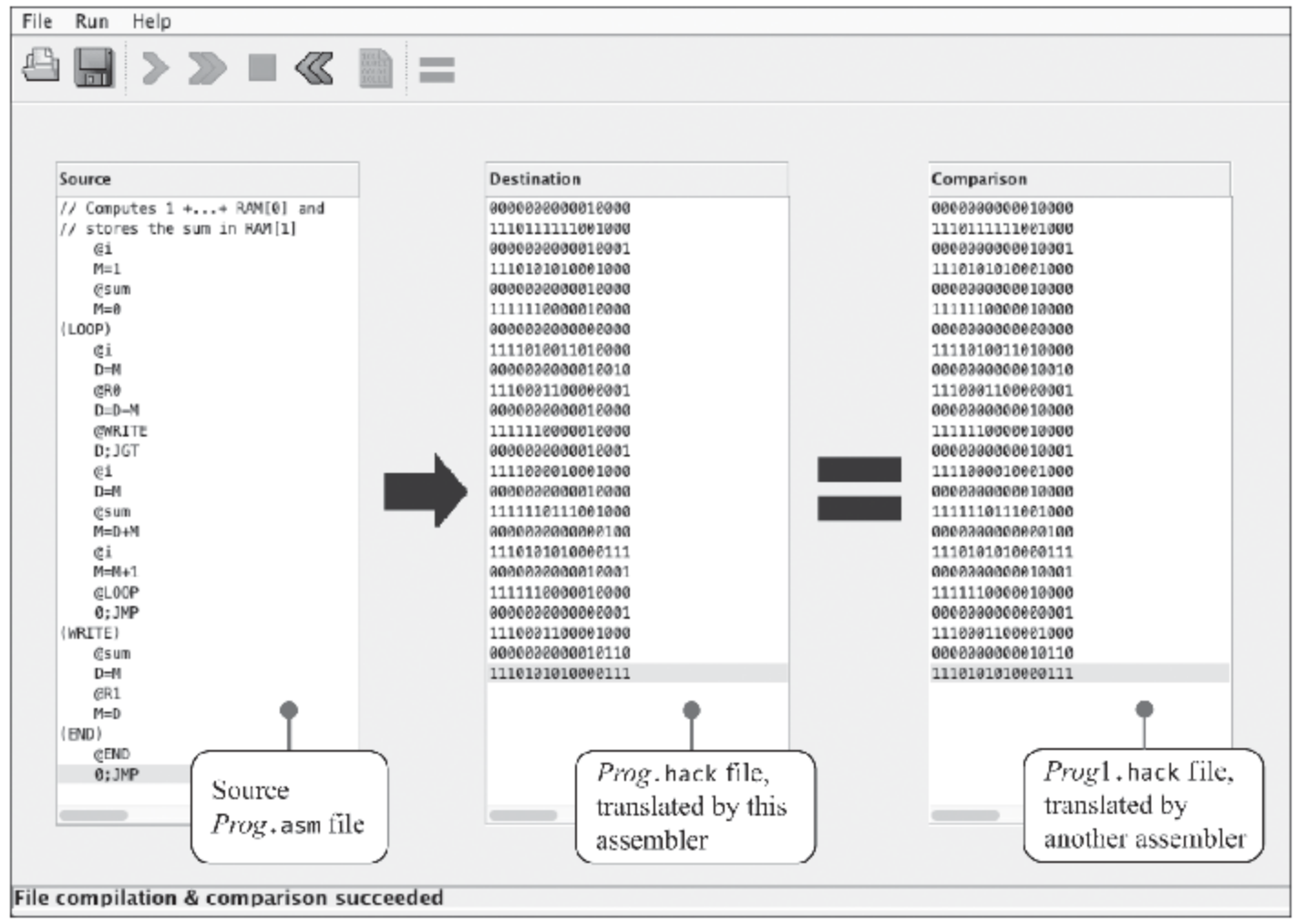


Figure 6.3 Testing the assembler's output using the supplied assembler.

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

6.6 Perspective

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

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

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

* * *

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

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

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

II Software

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

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

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

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

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

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

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

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

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

II.1 A Taste of Jack Programming

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

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

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

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

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

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

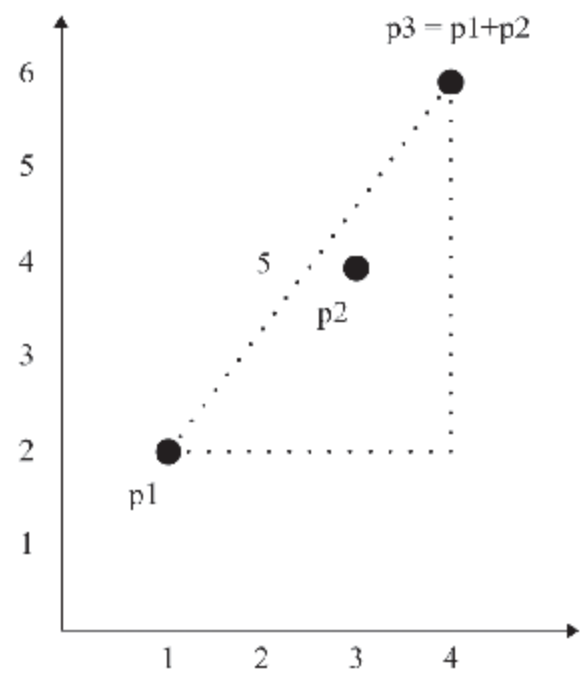
Clearly though, at one point or another, *someone* must implement this language abstraction. Someone must develop, once and for all, the ability to efficiently compute square roots when the application programmer blissfully says `sqrt(1764)`, to elicit a number from the user when the programmer happily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

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

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

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

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




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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure II.2 Jack implementation of the Point abstraction.

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

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

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

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

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

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

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

II.2 Program Compilation

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

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

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

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

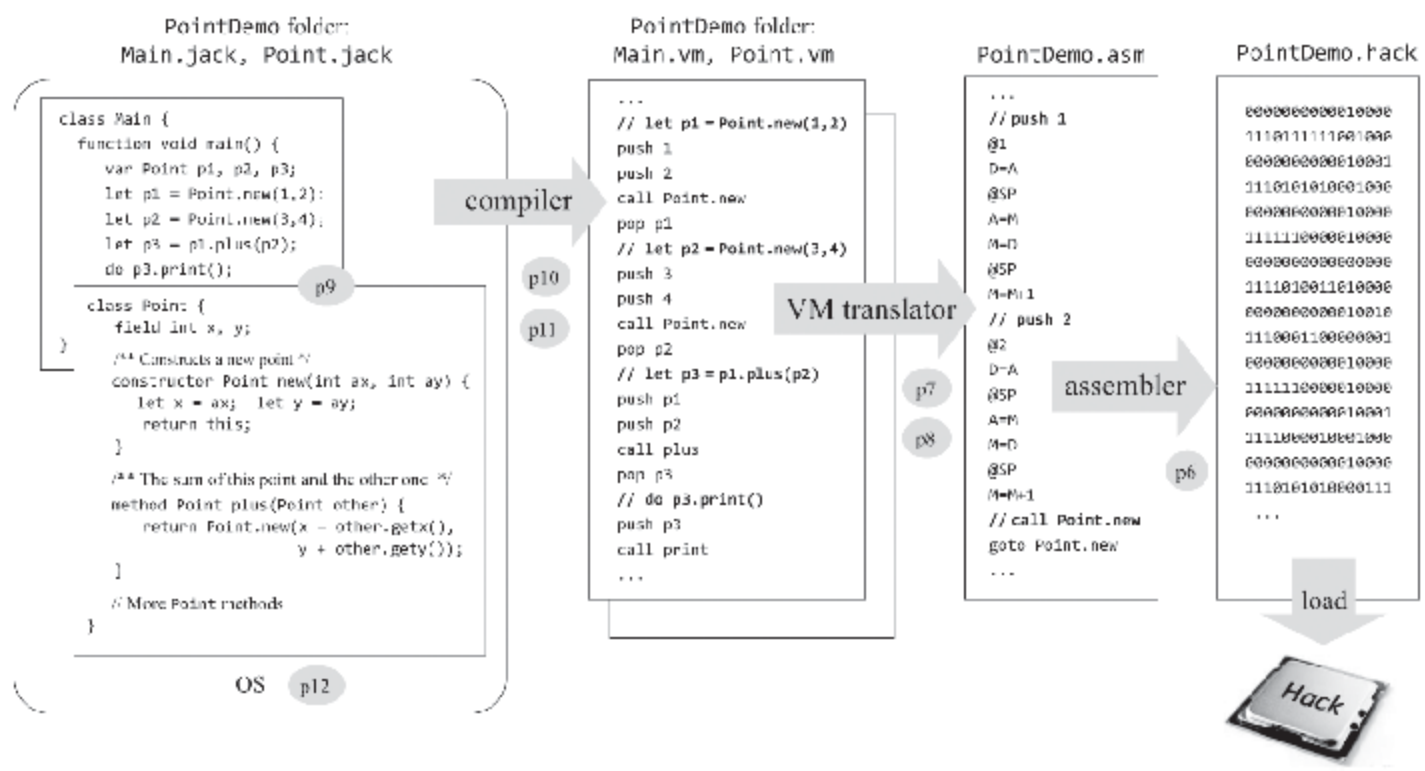


Figure II.3 Road map of part II (the assembler belongs to part I and is shown here for completeness). The road map describes a translation hierarchy, from a high-level, object-based, multi-class program to VM code, to assembly code, to executable binary code. The numbered circles stand for the projects that implement the compiler, the VM translator, the assembler, and the operating system. Project 9 focuses on writing a Jack application in order to get acquainted with the language.

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

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

7 Virtual Machine I: Processing

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

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

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

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

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

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

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

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

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

7.1 The Virtual Machine Paradigm

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

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

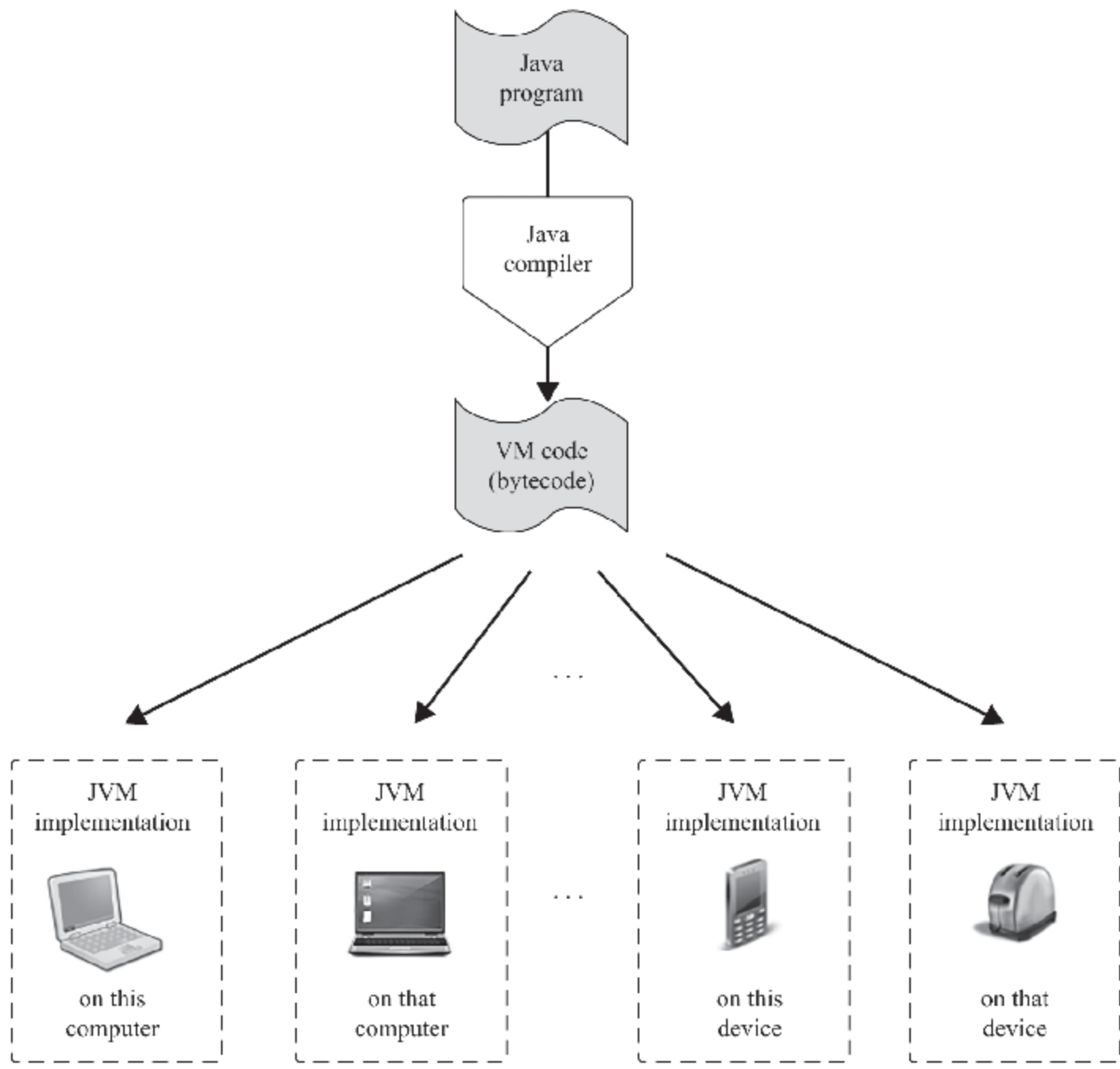


Figure 7.1 The virtual machine framework, using Java as an example. High-level programs are compiled into intermediate VM code. The same VM code can be shipped to, and executed on, any hardware platform equipped with a suitable *JVM implementation*. These VM implementations are typically realized as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

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

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

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

7.2 Stack Machine

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

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

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

7.2.1 Push and Pop

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

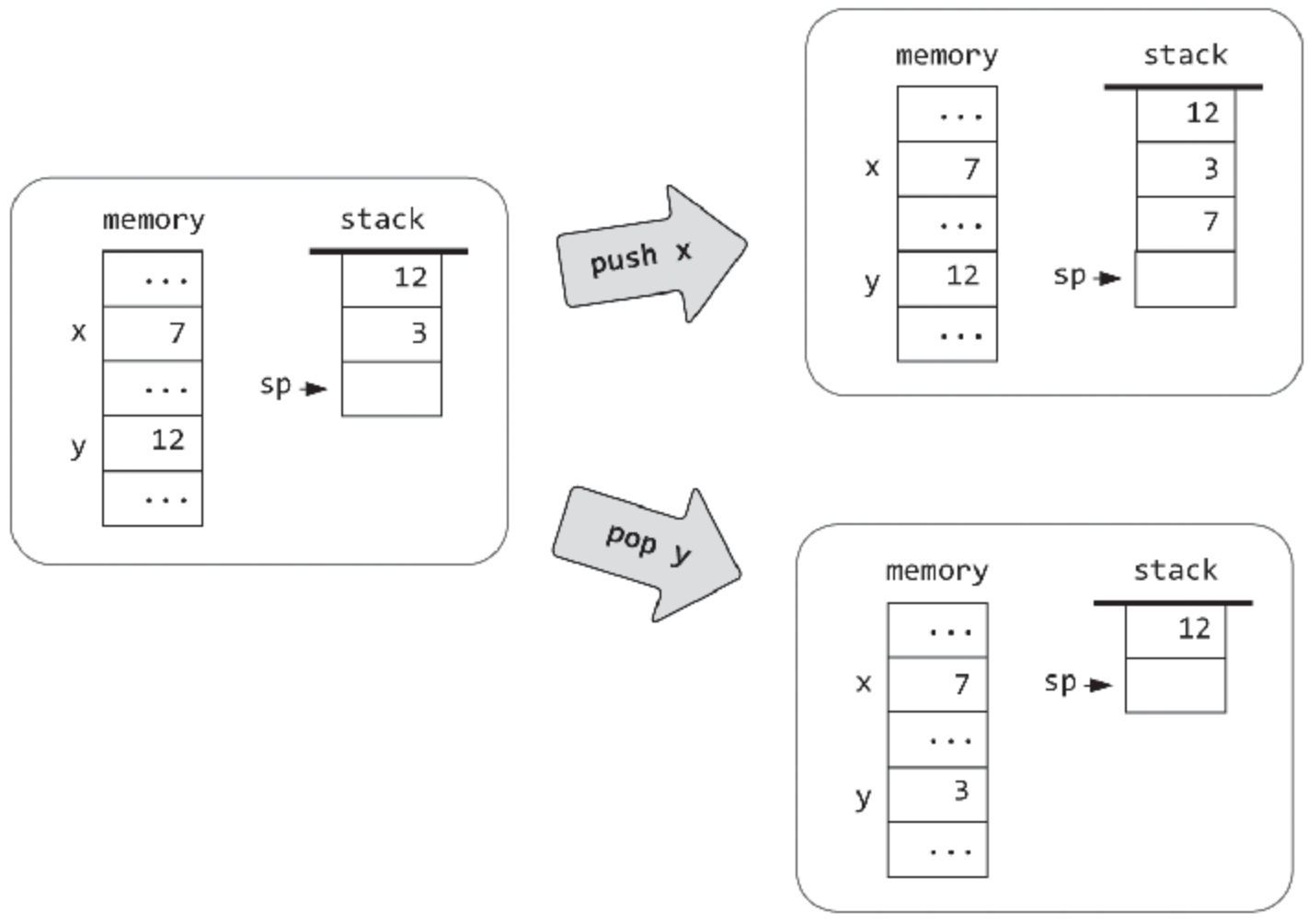
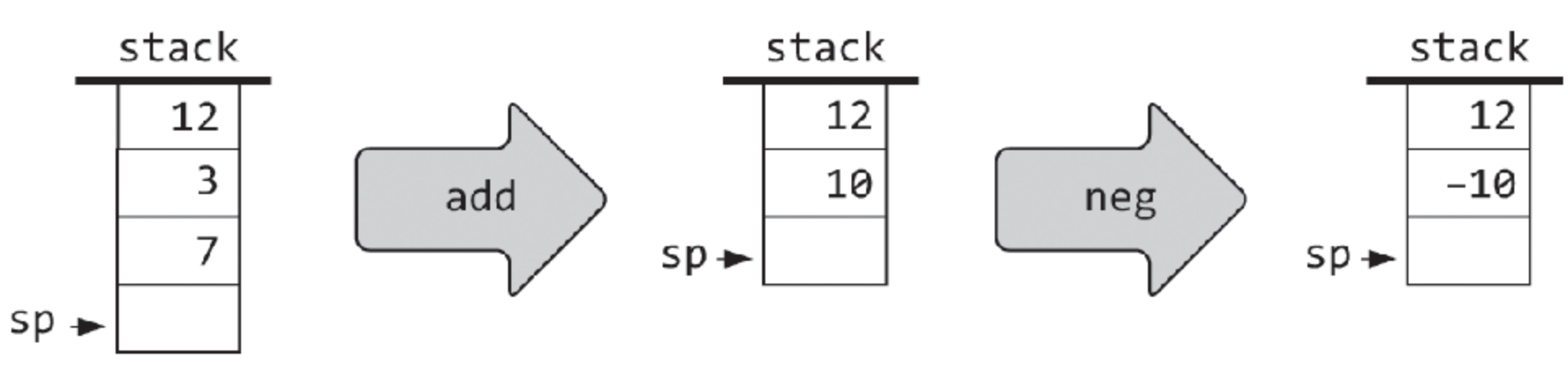


Figure 7.2 Stack processing example, illustrating the two elementary operations push and pop. The setting consists of two data structures: a RAM-like memory segment and a stack. Following convention, the stack is drawn as if it grows downward. The location just following the stack's top value is referred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

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

7.2.2 Stack Arithmetic

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



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

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

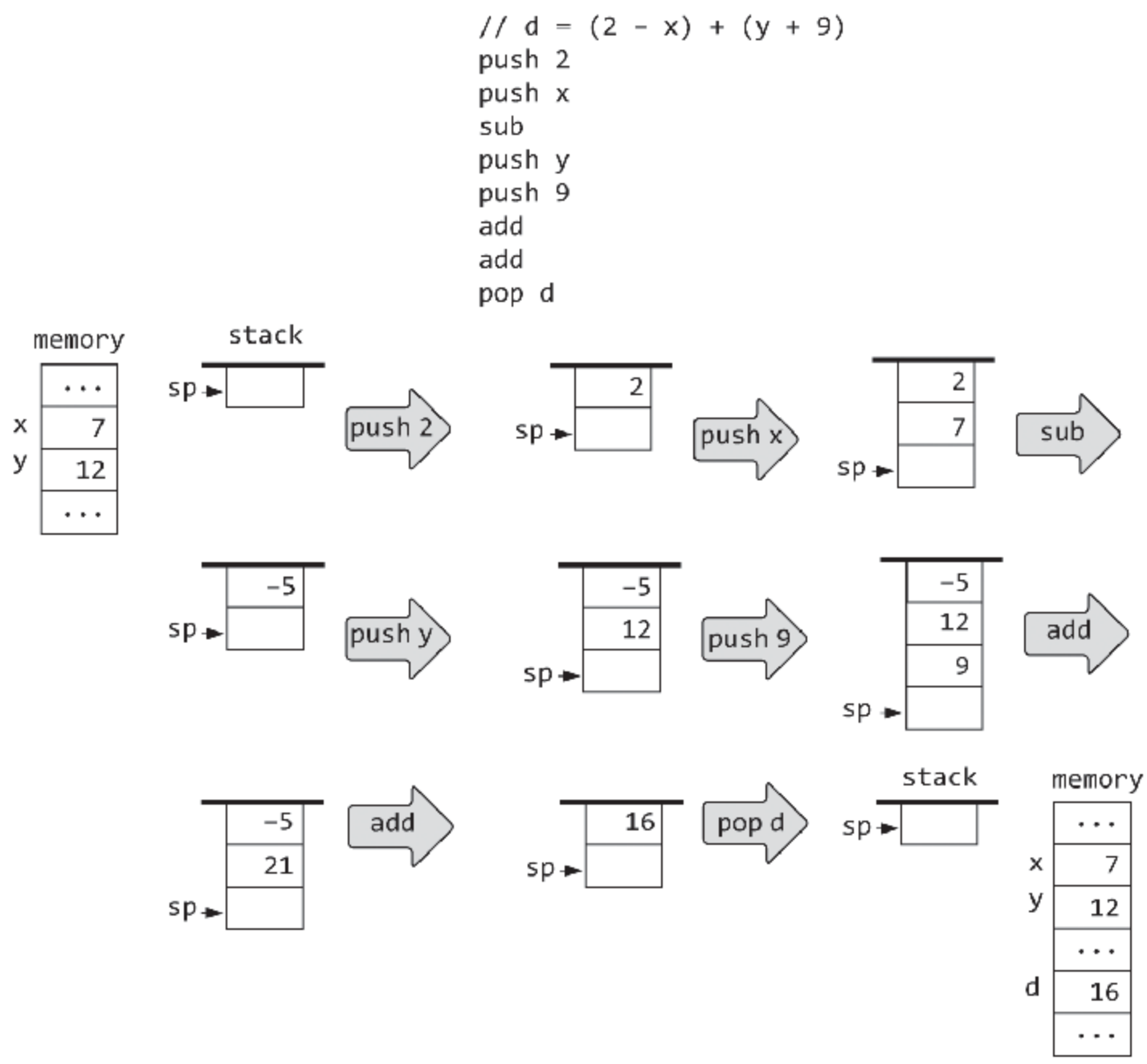


Figure 7.3a Stack-based evaluation of arithmetic expressions.

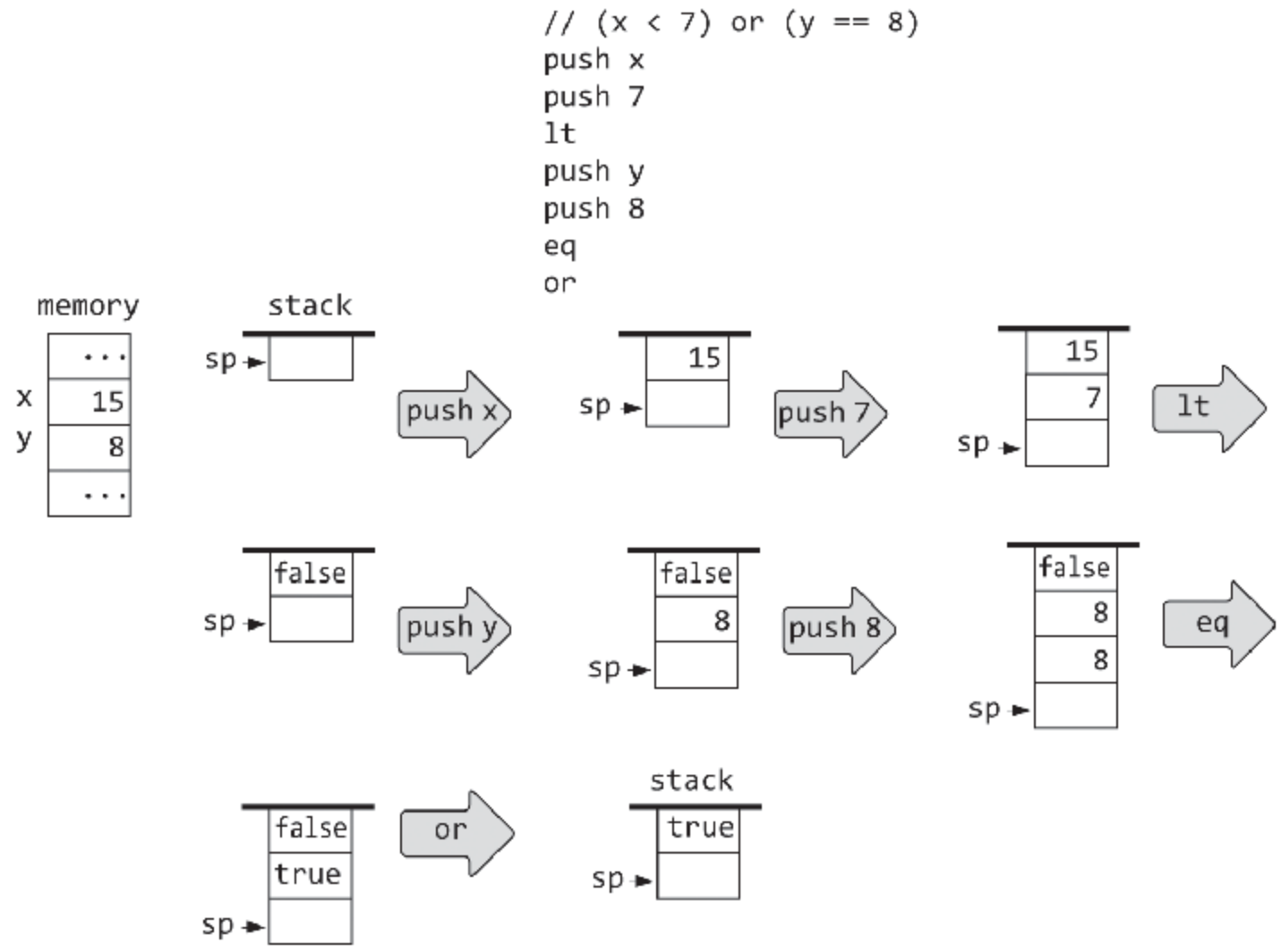


Figure 7.3b Stack-based evaluation of logical expressions.

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

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

7.2.3 Virtual Memory Segments

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

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

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

Figure 7.4 Virtual memory segments.

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

7.3 VM Specification, Part I

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

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

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

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

Push / Pop Commands

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

Arithmetic-Logical Commands

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

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

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

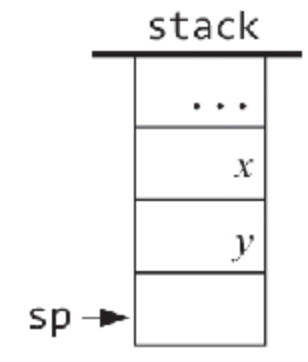


Figure 7.5 The arithmetic-logical commands of the VM language.

7.4 Implementation

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

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

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

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

7.4.1 Standard VM Mapping on the Hack Platform, Part I

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

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

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

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

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

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

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

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

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

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

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

Memory Segments Mapping

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

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

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

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

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

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

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

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

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

7.4.2 The VM Emulator

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

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

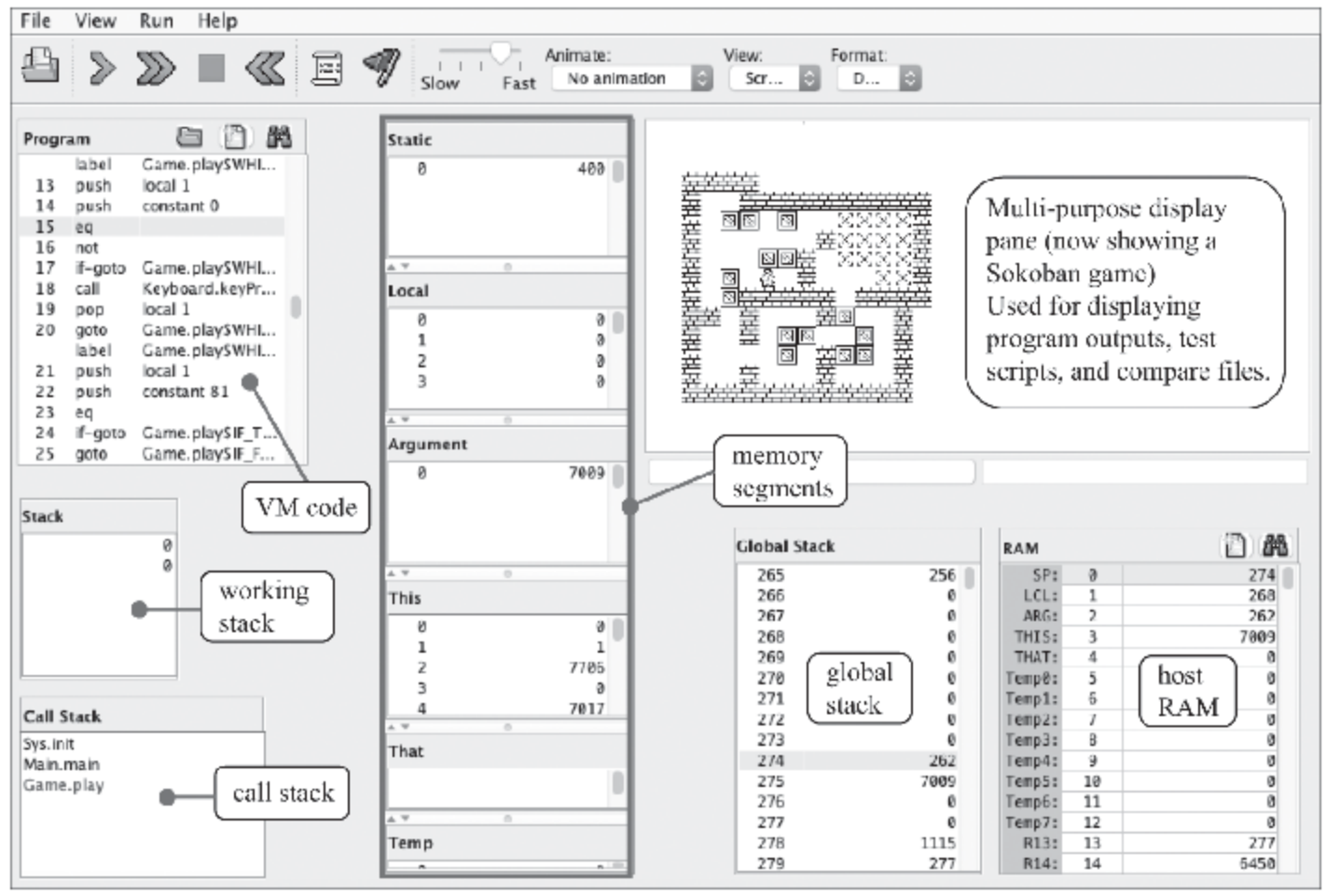


Figure 7.6 The VM emulator supplied with the Nand to Tetris software suite.

7.4.3 Design Suggestions for the VM Implementation

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

prompt> VMTranslator *source*

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

Program Structure

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

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

The Parser

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

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

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

The CodeWriter

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

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

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

The VM Translator

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

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

Implementation Tips

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

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

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

7.5 Project

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

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

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

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

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

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

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

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

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

Testing and Implementation Stages

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

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

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

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

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

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

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