

Assembler

What's in a name? That which we call a rose by any other name would smell as sweet.

—Shakespeare, from Romeo and Juliet

The first half of the book (chapters 1-5) described and built a computer's hardware platform. The second half of the book (chapters 6-12) focuses on the computer's software hierarchy, culminating in the development of a compiler and a basic operating system for a simple, object-based programming language. The first and most basic module in this software hierarchy is the assembler. In particular, chapter 4 presented machine languages in both their assembly and binary representations. This chapter describes how assemblers can systematically translate programs written in the former into programs written in the latter. As the chapter unfolds, we explain how to develop a Hack *assembler*—a program that generates binary code that can run as is on the hardware platform built in chapter 5.

Since the relationship between symbolic assembly commands and their corresponding binary codes is straightforward, writing an assembler (using some high-level language) is not a difficult task. One complication arises from allowing assembly programs to use symbolic references to memory addresses. The assembler is expected to manage these user-defined symbols and resolve them to physical memory addresses. This task is normally done using a symbol *table*—a classical data structure that comes to play in many software translation projects.

As usual, the Hack assembler is not an end in itself. Rather, it provides a simple and concise demonstration of the key software engineering principles used in the construction of any assembler. Further, writing the assembler is the first in the series of seven software development projects that accompany the rest of the book. Unlike the hardware projects, which were implemented in HDL, the software projects that construct the translator programs (*assembler*, virtual machine, and compiler) may be implemented in any programming language. In each project, we provide a language-neutral API and a detailed step-by-step test plan, along with all the necessary test programs and test scripts. Each one of these projects, beginning with the assembler, is a stand-alone module that can be developed and tested in isolation from all the other projects.

6.1 Background

Machine languages are typically specified in two flavors: symbolic and binary. The binary codes—for example, 110000101000000110000000000000111—represent actual machine instructions, as understood by the underlying hardware. For example, the instruction’s leftmost 8 bits can represent an operation code, say LOAD, the next 8 bits a register, say R3, and the remaining 16 bits an address, say 7. Depending on the hardware’s logic design and the agreed-upon machine language, the overall 32-bit pattern can thus cause the hardware to effect the operation “load the contents of Memory[7] into register R3.” Modern computer platforms support dozens if not hundreds of such elementary operations. Thus, machine languages can be rather complex, involving many operation codes, different memory addressing modes, and various instruction formats.

One way to cope with this complexity is to document machine instructions using an agreed-upon syntax, say LOAD R3,7 rather than 110000101000000110000000000000111. And since the translation from symbolic notation to binary code is straightforward, it makes sense to allow low-level programs to be written in symbolic notation and to have a computer program translate them into binary code. The symbolic language is called assembly, and the translator program assembler. The assembler parses each assembly command into its underlying fields, translates each field into its equivalent binary code, and assembles the generated codes into a binary instruction that can be actually executed by the hardware.

Symbols Binary instructions are represented in binary code. By definition, they refer to memory addresses using actual numbers. For example, consider a program that uses a variable to represent the weight of various things, and suppose that this variable has been mapped on location 7 in the computer’s memory. At the binary code level, instructions that manipulate the weight variable must refer to it using the explicit address 7. Yet once we step up to the assembly level, we can allow writing commands like LOAD R3,weight instead of LOAD R3,7. In both cases, the command will effect the same operation: “set R3 to the contents of Memory[7].” In a similar fashion, rather than using commands like goto 250, assembly languages allow commands like goto loop, assuming that somewhere in the program the symbol loop is made to refer to address 250. In general then, symbols are introduced into assembly programs from two sources:

- *Variables*: The programmer can use symbolic variable names, and the translator will “automatically” assign them to memory addresses. Note that the actual values of these addresses are insignificant, so long as each symbol is resolved to the same address throughout the program’s translation.
- *Labels*: The programmer can mark various locations in the program with symbols. For example, one can declare the label loop to refer to the beginning of a certain code segment. Other commands in the program can then goto loop, either conditionally or unconditionally.

The introduction of symbols into assembly languages suggests that assemblers must be more sophisticated than dumb text processing programs. Granted, translating agreed-upon symbols into agreed-upon binary codes is not a complicated task. At the same time, the mapping of user-defined variable names and symbolic labels on actual memory addresses is not trivial. In fact, this symbol resolution task is the first nontrivial translation challenge in our ascent up the software hierarchy from the hardware

level. The following example illustrates the challenge and the common way to address it.

Symbol Resolution Consider figure 6.1, showing a program written in some self-explanatory low-level language. The program contains four user-defined symbols: two variable names (i and sum) and two labels (loop and end). How can we systematically convert this program into a symbol-less code?

We start by making two arbitrary game rules: The translated code will be stored in the computer’s memory starting at address 0, and variables will be allocated to memory locations starting at address 1024 (these rules depend on the specific target hardware platform). Next, we build a symbol table, as follows. For each new symbol xxx encountered in the source code, we add a line (xxx, n) to the symbol table, where n is the memory address associated with the symbol according to the game rules. After completing the construction of the symbol table, we use it to translate the program into its symbol-less version.

Note that according to the assumed game rules, variables i and sum are allocated to addresses 1024 and 1025, respectively. Of course any other two addresses will be just as good, so long as all references to i and sum in the program resolve to the same physical addresses, as indeed is the case. The remaining code is self-explanatory, except perhaps for instruction 6. This instruction terminates the program’s execution by putting the computer in an infinite loop.

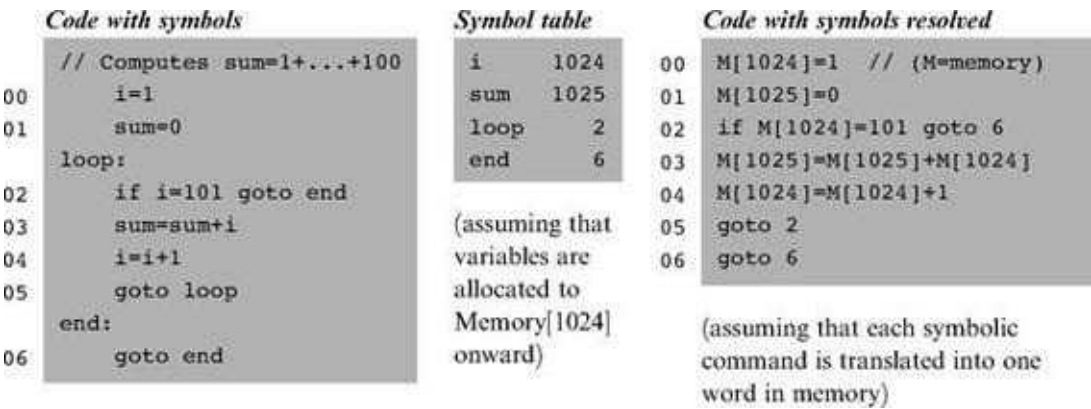


Figure 6.1 Symbol resolution using a symbol table. The line numbers are not part of the program—they simply count all the lines in the program that represent real instructions, namely, neither comments nor label declarations. Note that once we have the symbol table in place, the symbol resolution task is straightforward.

Three comments are in order here. First, note that the variable allocation assumption implies that the largest program that we can run is 1,024 instructions long. Since realistic programs (like the operating system) are obviously much larger, the base address for storing variables will normally be much farther. Second, the assumption that each source command is mapped on one word may be naïve. Typically, some assembly commands (e.g., if i=101 goto end) may translate into several machine instructions and thus will end up occupying several memory locations. The translator can deal with this variance by keeping track of how many words each source command generates, then updating its “instruction memory counter” accordingly.

Finally, the assumption that each variable is represented by a single memory location is also naïve. Programming languages feature variables of different types, and these occupy different memory spaces on

the target computer. For example, the C language data types short and double represent 16-bit and 64-bit numbers, respectively. When a C program is run on a 16-bit machine, these variables will occupy a single memory address and a block of four consecutive addresses, respectively. Thus, when allocating memory space for variables, the translator must take into account both their data types and the word width of the target hardware.

The Assembler Before an assembly program can be executed on a computer, it must be translated into the computer's binary machine language. The translation task is done by a program called the assembler. The assembler takes as input a stream of assembly commands and generates as output a stream of equivalent binary instructions. The resulting code can be loaded as is into the computer's memory and executed by the hardware.

We see that the assembler is essentially a text-processing program, designed to provide translation services. The programmer who is commissioned to write the assembler must be given the full documentation of the assembly syntax, on the one hand, and the respective binary codes, on the other. Following this contract—typically called machine language *specification*—it is not difficult to write a program that, for each symbolic command, carries out the following tasks (not necessarily in that order):

- Parse the symbolic command into its underlying fields.
- For each field, generate the corresponding bits in the machine language.
- Replace all symbolic references (if any) with numeric addresses of memory locations.
- Assemble the binary codes into a complete machine instruction.

Three of the above tasks (parsing, code generation, and final assembly) are rather easy to implement. The fourth task—symbols handling—is more challenging, and considered one of the main functions of the assembler. This function was described in the previous section. The next two sections specify the Hack assembly language and propose an assembler implementation for it, respectively.

6.2 Hack Assembly-to-Binary Translation Specification

The Hack assembly language and its equivalent binary representation were specified in chapter 4. A compact and formal version of this language specification is repeated here, for ease of reference. This specification can be viewed as the contract that Hack assemblers must implement, one way or another.

6.2.1 Syntax Conventions and File Formats

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 by the assembler 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 in section 6.2.2.

- (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 “pseudocommand” since it generates no machine code.

(The remaining conventions in this section pertain to assembly programs only.)

Constants and Symbols *Constants* must be non-negative and are 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 Text beginning with two slashes (`//`) and ending at the end of the line is considered a comment and is ignored.

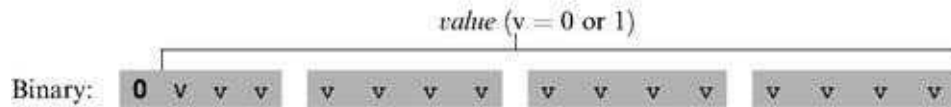
White Space Space characters are ignored. Empty lines are ignored.

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

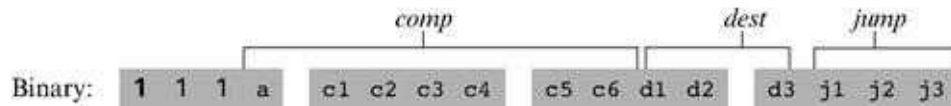
6.2.2 Instructions

The Hack machine language consists of two instruction types called addressing instruction (*A*-instruction) and compute instruction (*C*-instruction). The instruction format is as follows.

A-instruction: `@value` // Where *value* is either a non-negative decimal number
// or a symbol referring to such number.



C-instruction: `dest=comp;jump` // Either the *dest* or *jump* fields may be empty.
// If *dest* is empty, the "=" is omitted;
// If *jump* is empty, the ";" is omitted.



The translation of each of the three fields *comp*, *dest*, *jump* to their binary forms is specified in the following three tables.

<i>comp</i> (when a=0)	c1	c2	c3	c4	c5	c6	<i>comp</i> (when a=1)
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

<i>dest</i>	d1	d2	d3	<i>jump</i>	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

6.2.3 Symbols

Hack assembly commands can refer to memory locations (addresses) using either constants or symbols. Symbols in assembly programs arise from three sources.

Predefined Symbols Any Hack assembly program is allowed to use the following predefined 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 top five RAM locations can be referred to using two predefined 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 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).

6.2.4 Example

Chapter 4 presented a program that sums up the integers 1 to 100. Figure 6.2 repeats this example, showing both its assembly and binary versions.

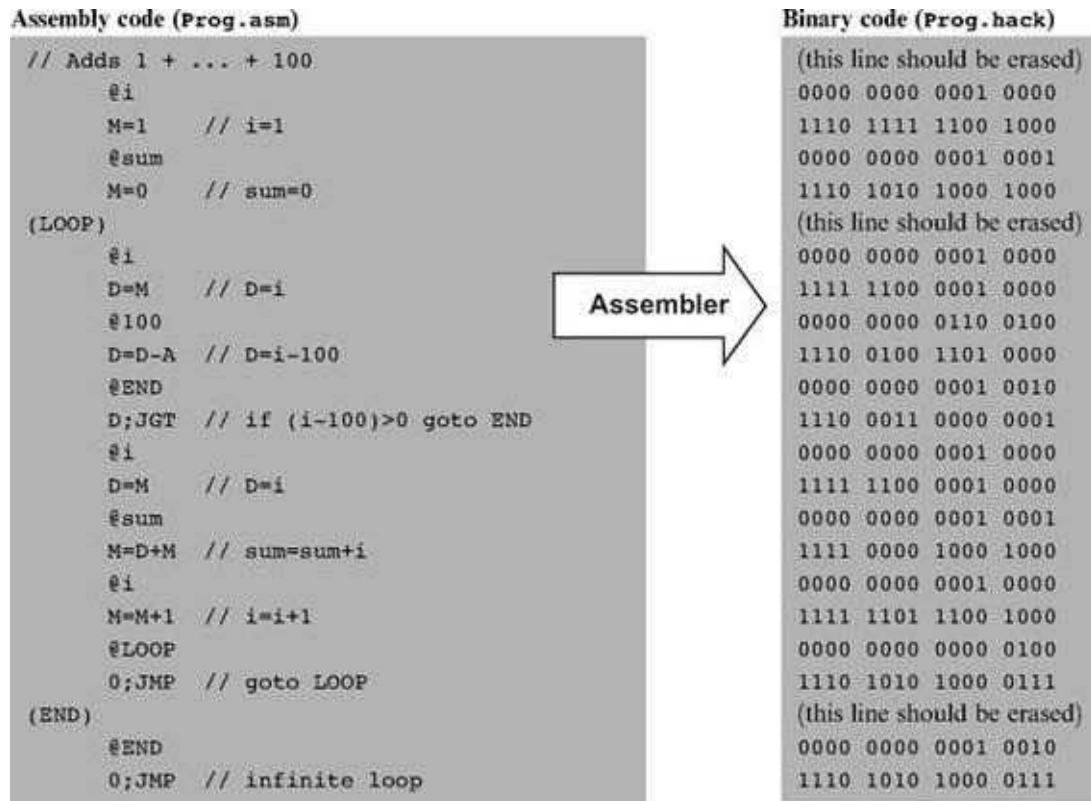


Figure 6.2 Assembly and binary representations of the same program.

6.3 Implementation

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

```
prompt> Assembler Prog.asm
```

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 assembly command is translated into its corresponding bit code according to the tables in section 6.2.2, and each symbol in the command is resolved to its numeric address as specified in section 6.2.3.

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

A Note about API Notation The assembler development is the first in a series of five software construction projects that build our hierarchy of translators (*assembler, virtual machine, and compiler*). Since readers can develop these projects in the programming language of their choice, we base our proposed implementation guidelines on language independent APIs. A typical project API describes several modules, each containing one or more routines. In object-oriented languages like Java, C++, and C#, a module usually corresponds to a class, and a routine usually corresponds to a method. In procedural languages, routines correspond to functions, subroutines, or procedures, and modules correspond to collections of routines that handle related data. In some languages (e.g., Modula-2) a module may be expressed explicitly, in others implicitly (e.g., a *file* in the C language), and in others (e.g., Pascal) it will have no corresponding language construct, and will just be a conceptual grouping of routines.

6.3.1 The Parser Module

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: 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 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	—	A_COMMAND, C_COMMAND, L_COMMAND	Returns the type of the current command: <ul style="list-style-type: none">■ A_COMMAND for @xxx where xxx is either a symbol or a decimal number■ C_COMMAND for dest=comp; jump■ 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 (8 possibilities). Should be called only when commandType() is C_COMMAND.

Routine	Arguments	Returns	Function
<code>comp</code>	—	string	Returns the <code>comp</code> mnemonic in the current <i>C</i> -command (28 possibilities). Should be called only when <code>commandType()</code> is <code>C_COMMAND</code> .
<code>jump</code>	—	string	Returns the <code>jump</code> mnemonic in the current <i>C</i> -command (8 possibilities). Should be called only when <code>commandType()</code> is <code>C_COMMAND</code> .

6.3.2 The *Code* Module

Routine	Arguments	Returns	Function
<code>dest</code>	mnemonic (string)	3 bits	Returns the binary code of the <code>dest</code> mnemonic.
<code>comp</code>	mnemonic (string)	7 bits	Returns the binary code of the <code>comp</code> mnemonic.
<code>jump</code>	mnemonic (string)	3 bits	Returns the binary code of the <code>jump</code> mnemonic.

Code: Translates Hack assembly language mnemonics into binary codes.

6.3.3 Assembler for Programs with No Symbols

We suggest building 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 input file contains no label commands, namely, 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 into the Prog.hack file.

6.3.4 The *SymbolTable* Module

Since Hack instructions can contain symbols, the symbols must be resolved into actual addresses 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 (in Hack's case, RAM and ROM addresses). A natural data structure for representing such a relationship is the classical hash table. In most programming languages, 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.

Routine	Arguments	Returns	Function
Constructor	—	—	Creates a new empty symbol table.
addEntry	symbol (string), address (int)	—	Adds the pair (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.

SymbolTable: Keeps a correspondence between symbolic labels and numeric addresses.

6.3.5 Assembler for Programs with Symbols

Assembly programs are allowed to use symbolic labels (destinations of goto commands) before the symbols are defined. This convention makes the life of assembly programmers easier and that of assembler developers harder. A common solution to this complication is to write a two-pass assembler that reads the code twice, from start to end. In the first pass, the assembler builds the symbol table and generates no code. In the second pass, all the label symbols encountered in the program have already been bound to memory locations and recorded in the symbol table. Thus, the assembler can replace each symbol with its corresponding meaning (numeric address) and generate the final binary code.

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

Initialization Initialize the symbol table with all the predefined symbols and their pre-allocated RAM addresses, according to section 6.2.3.

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 recording the ROM address into which the current command will be eventually loaded. 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 pseudocommand or a comment is encountered. Each time a pseudocommand (*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, namely, *@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 must represent a new variable. To handle it, add the pair (*Xxx*, *n*) to the symbol table, where *n* is the next available RAM address, and complete the command's translation. The allocated RAM addresses are consecutive numbers, starting at address 16 (just after the addresses allocated to the predefined symbols).

This completes the assembler's implementation.

6.4 Perspective

Like most assemblers, the Hack assembler is a relatively simple program, dealing mainly with text processing. Naturally, assemblers for richer machine languages are more complex. Also, some assemblers feature more sophisticated symbol handling capabilities not found in Hack. For example, the assembler may allow programmers to explicitly associate symbols with particular data addresses, to perform “constant arithmetic” on symbols (e.g., to use `table+5` to refer to the fifth memory location after the address referred to by `table`), and so on. Additionally, many assemblers are capable of handling macro commands. A macro command is simply a sequence of machine instructions that has a name. For example, our assembler can be extended to translate an agreed-upon macro-command, say `D=M[xxx]`, into the two instructions `@xxx` followed immediately by `D=M` (`xxx` being an address). Clearly, such macro commands can considerably simplify the programming of commonly occurring operations, at a low translation cost.

We note in closing that stand-alone assemblers are rarely used in practice. First, assembly programs are rarely written by humans, but rather by compilers. And a compiler—being an automaton—does not have to bother to generate symbolic commands, since it may be more convenient to directly produce binary machine code. On the other hand, many high-level language compilers allow programmers to embed segments of assembly language code within high-level programs. This capability, which is rather common in C language compilers, gives the programmer direct control of the underlying hardware, for optimization.

6.5 Project

Objective 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 6.2.

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 assembler tutorial (part of the book's software suite).

Contract When loaded into your assembler, a Prog.asm file containing a valid 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.

Building Plan We suggest building the assembler in two stages. First write a symbol-less assembler, namely, 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 here 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.asm is symbol-less, and Prog.asm is with symbols.

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

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

Rect: 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 (chapter 9) and translated into the

supplied assembly program by the Jack compiler (chapters 10-11). Although the original Jack program is only about 300 lines of code, the executable Pong application is about 20,000 lines of binary code, most of which being the Jack operating system (chapter 12). Running this interactive program 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 Write and test your assembler program in the two stages described previously. You may use the assembler supplied with the book to compare the output of your assembler to the correct output. This testing procedure is described next. For more information about the supplied assembler, refer to the assembler tutorial.

The Supplied Assembler The practice of using the supplied assembler (which produces correct binary code) to test another assembler (which is not necessarily correct) is illustrated in figure 6.3. Let Prog.asm be some program written in Hack assembly. Suppose that we translate this program using the supplied assembler, producing a binary file called Prog.hack. Next, we use another assembler (e.g., the one that you wrote) to translate the same program into another file, say Prog1.hack. Now, if the latter assembler is working correctly, it follows that Prog.hack = Prog1.hack. Thus, one way to test a newly written assembler is to load Prog.asm into the supplied assembler program, load Prog1.hack as a compare file, then translate and compare the two binary files (see figure 6.3). If the comparison fails, the assembler that produced Prog1.hack must be buggy; otherwise, it may be error-free.

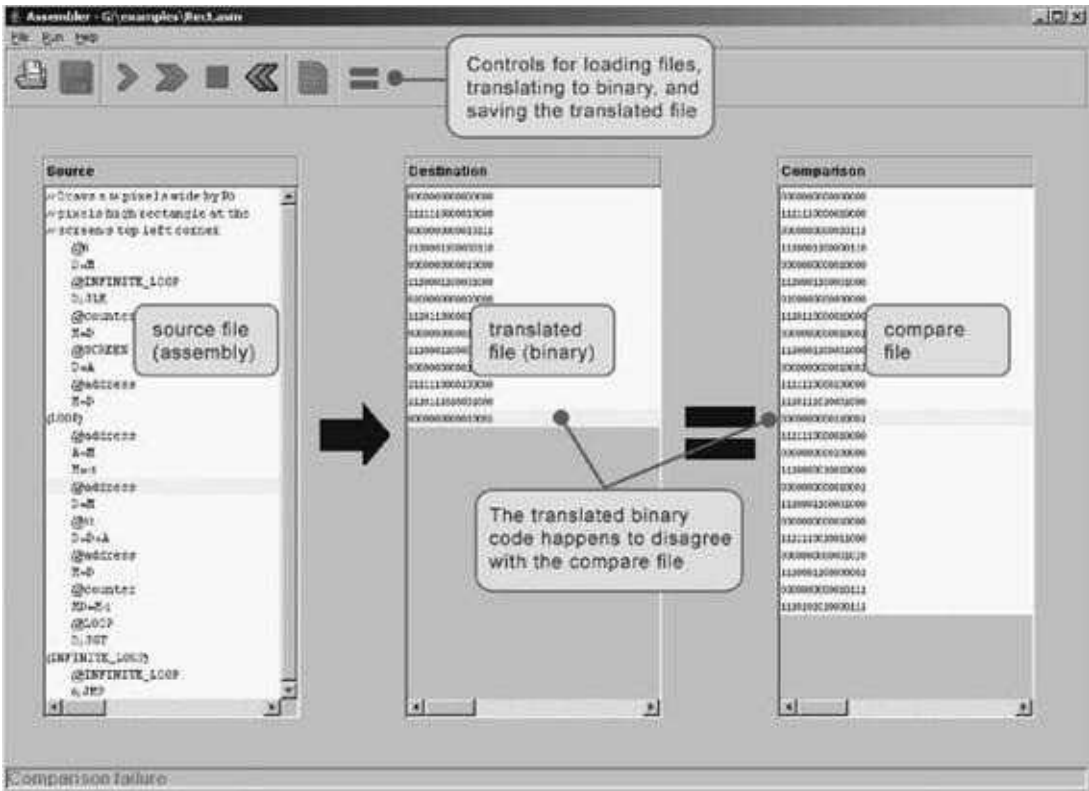


Figure 6.3 Using the supplied assembler to test the code generated by another assembler.

Virtual Machine I: Stack Arithmetic

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. High-level programs will first be 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 goes back to the 1970s. Recently, it made a significant comeback following its adoption by modern languages like Java and C#.

The basic idea is as follows: Instead of running on a real platform, the intermediate code is designed to run on a Virtual Machine. The VM is an abstract computer that does not exist for real, but can rather be realized on other computer platforms. 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, VM-based software can run on many processors and operating systems without having to modify the original source code. The VM implementations can be realized in several ways, by software interpreters, by special-purpose hardware, or by translating the VM programs into the machine language of the target platform.

This chapter presents a typical VM architecture, modeled after the Java Virtual Machine (JVM) paradigm. As usual, we focus on two perspectives. First, we motivate and specify the VM abstraction. Next, we implement it over the Hack platform. Our implementation entails 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 using Java.

A virtual machine model typically has a language, in which one can write *VM* programs. The VM language that we present here consists of four types of commands: arithmetic, memory access, program flow, and subroutine calling commands. We split the implementation of this language into two parts, each covered in a separate chapter and project. In this chapter we build a basic VM translator, capable of translating the VM's arithmetic and memory access commands into machine language. In the next chapter we extend the basic translator with program flow and subroutine calling functionality. The result is a full-scale virtual machine that will serve as the backend of the compiler that we will build in chapters 10—11.

The virtual machine that emerges 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 1930s. Over the years it had many practical implications, for example, 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 competing 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 and elegant data structure that comes to play in many computer systems and algorithms. Since the VM presented in this chapter is stack-based, it provides a working example of this remarkably versatile data structure.

7.1 Background

7.1.1 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. 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 intermediate processing steps—steps that are neither “high” nor “low.” In the second stage, the intermediate steps are translated further into 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 processing 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 can formulate a virtual machine whose instructions are the intermediate processing 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 VM 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—one way or another—in many compiler construction projects. Some developers went as far as defining a formal and stand-alone virtual machine language, most notably the p-code generated by several Pascal compilers in the 1970s. Java compilers are also two-tiered, generating a bytecode language that runs on the JVM virtual machine (also called the Java Runtime *Environment*). More recently, the approach has been adopted by the .NET infrastructure. In particular, .NET requires compilers to generate code written in an intermediate language (IL) that runs on a virtual machine called CLR (*Common Language Runtime*).

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 trade-offs among code efficiency, hardware cost, and programming effort. Second, compilers for many languages can share the same VM backend, allowing code sharing and language interoperability. For example, one high-level language may be good at scientific calculations, while another may excel in handling the user interface. If both languages compile into a common VM layer, it is rather natural to have routines in one language call routines in the other, using an agreed-upon invocation syntax.

Another benefit 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 that is equipped with a VM implementation can immediately benefit from a huge base of available software, as seen in figure 7.1.

7.1.2 The Stack Machine Model

Like most programming 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 language implementation. 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. As it turns out, 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 what follows.

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, illustrated in figure 7.2.

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 overrides 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]*).

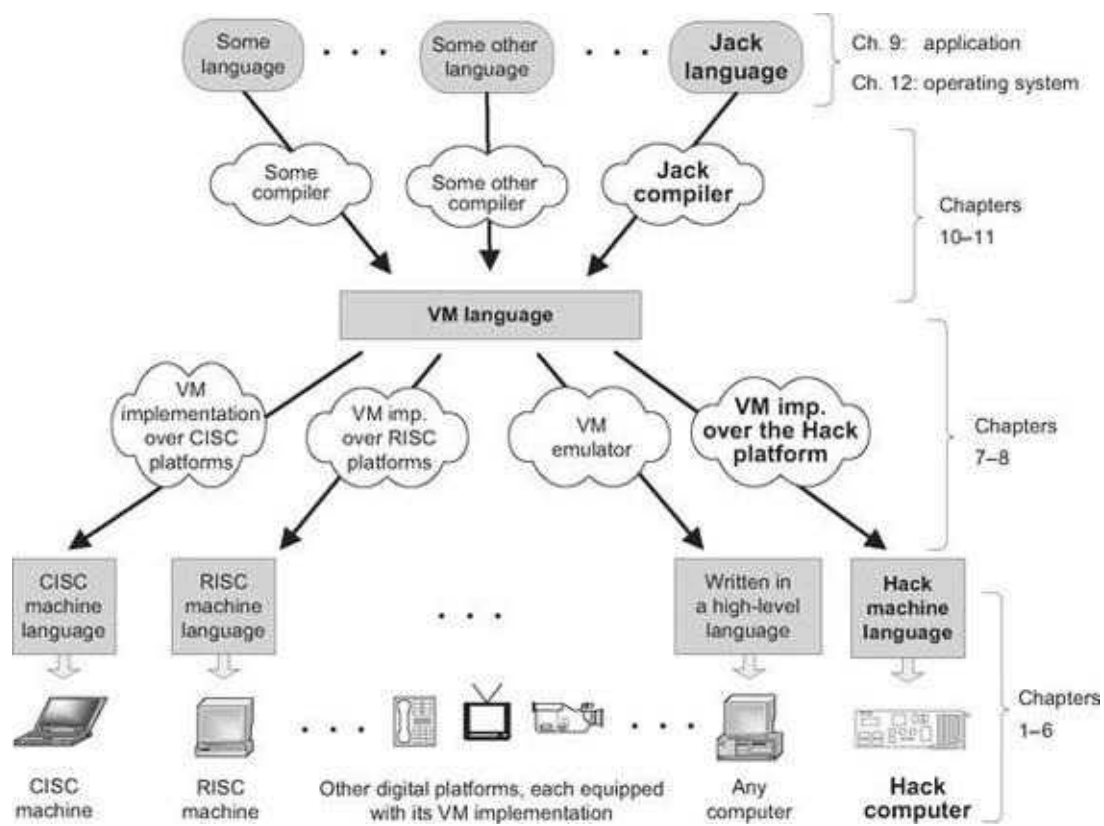


Figure 7.1 The virtual machine paradigm. Once a high-level program is compiled into VM code, the program can run on any hardware platform equipped with a suitable VM implementation. In this chapter we start building the *VM implementation on the Hack platform* and use a VM emulator like the one depicted on the right.

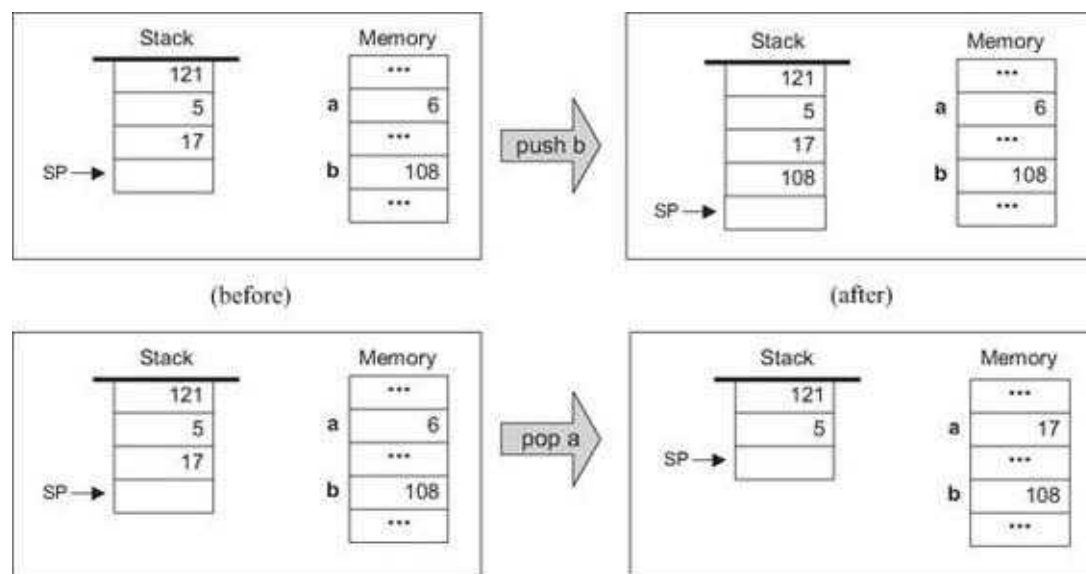
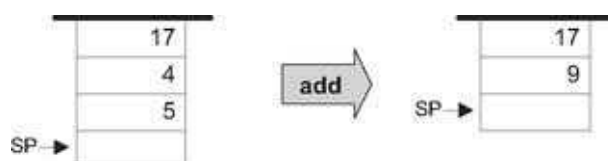


Figure 7.2 Stack processing example, illustrating the two elementary operations *push* and *pop*. Following convention, the stack is drawn upside down, as if it grows downward. The location just after the top position is always referred to by a special pointer called *sp*, or *stack pointer*. The labels *a* and *b* refer to

two arbitrary memory addresses.

As usual in computer science, simplicity and elegance imply power of expression. The simple stack model is a versatile data structure that comes to play in many computer systems and algorithms. In the virtual machine architecture that we build here, it serves two key purposes. First, it is used for handling all the arithmetic and logical operations of the VM. Second, it facilitates subroutine calls and the associated memory allocation—the subjects of the next chapter.

Stack Arithmetic Stack-based arithmetic is a simple matter: the operands 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:



The stack version of other operations (subtract, multiply, etc.) are precisely the same. For example, consider the expression $d = (2 - x) * (y + 5)$, taken from some high-level program. The stack-based evaluation of this expression is shown in figure 7.3.

Stack-based evaluation of Boolean expressions has precisely the same flavor. 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 7.4.

The previous examples illustrate a general observation: any arithmetic and Boolean expression—no matter how complex—can be systematically converted into, and evaluated by, a sequence of simple operations on a stack. Thus, one can write a compiler that translates high-level arithmetic and Boolean expressions into sequences of stack commands, as we will do in chapters 10-11. We now turn to specify these commands (section 7.2), and describe their implementation on the Hack platform (section 7.3).

7.2 VM Specification, Part I

7.2.1 General

The virtual machine is stack-based: all operations are done on a stack. It is also function-based: a complete VM program is organized in program units called 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.

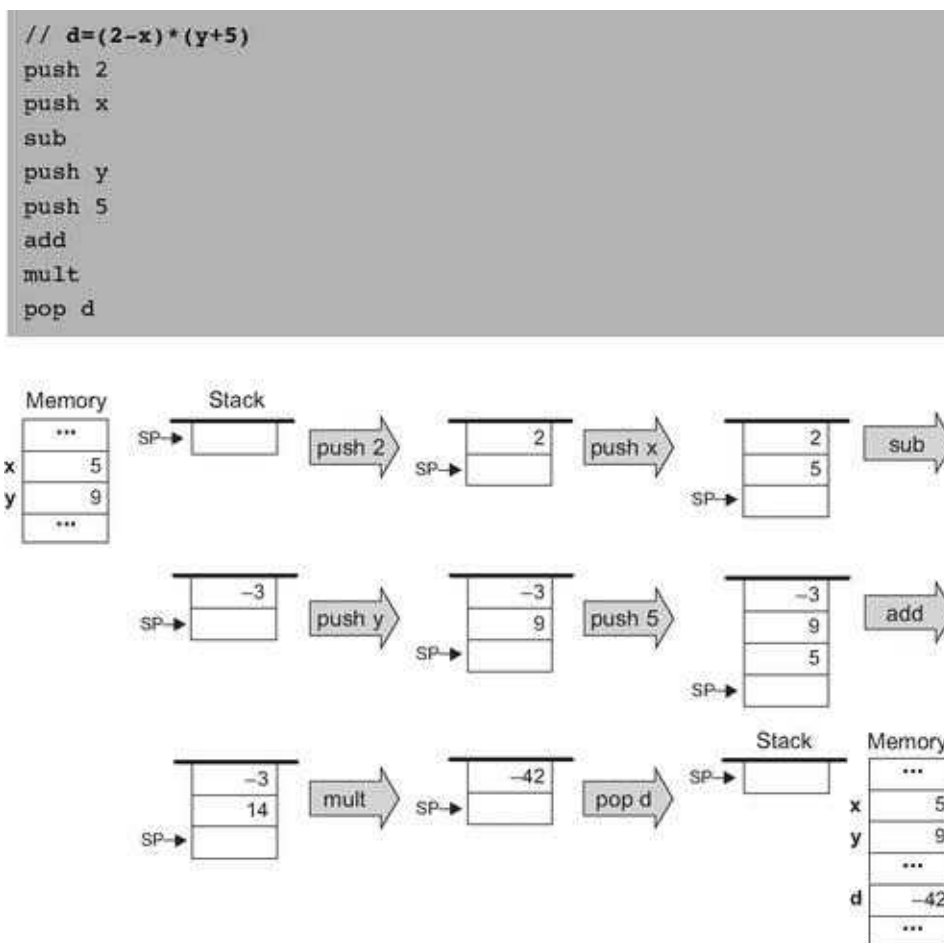


Figure 7.3 Stack-based evaluation of arithmetic expressions. This example evaluates the expression $d = (2 - x) * (y + 5)$, assuming the initial memory state $x = 5$, $y = 9$.

```
// if (x<7) or (y=8)
push x
push 7
lt
push y
push 8
eq
or
```

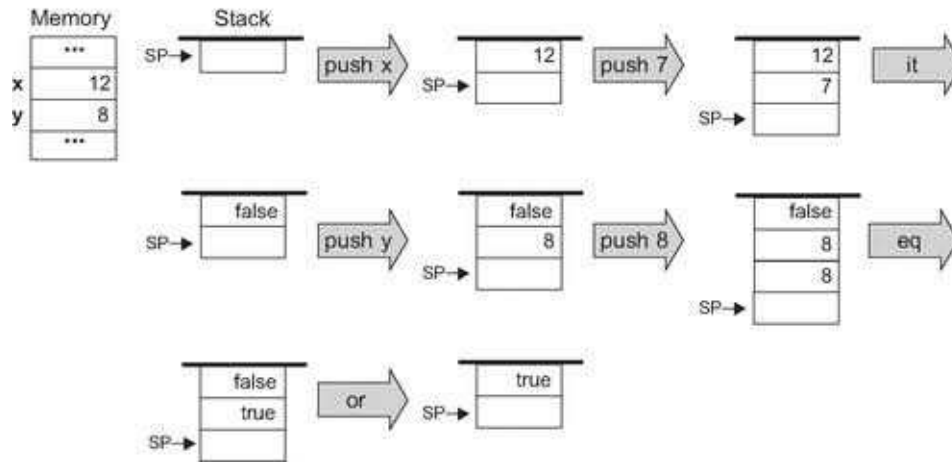


Figure 7.4 Stack-based evaluation of logical expressions. This example evaluates the Boolean expression $(x < 7) \text{ or } (y = 8)$, assuming the initial memory state $x = 12, y = 8$.

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-scale 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* (e.g., add), *command arg* (e.g., goto loop), or *command arg1 arg2* (e.g., push local 3). 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 and ignored.

7.2.2 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. Figure 7.5 gives the details.

Three of the commands listed in figure 7.5 (*eq*, *gt*, *lt*) return Boolean values. The VM represents *true* and *false* as -1 (minus one, 0xFFFF) and 0 (zero, 0x0000), respectively.

7.2.3 Memory Access Commands

So far in the chapter, memory access commands were illustrated using the pseudo-commands *pop* and *push x*, where the symbol *x* referred to an individual location in some global memory. Yet formally, our VM manipulates eight separate virtual memory segments, listed in figure 7.6.

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 $x = y$, else false	Equality
gt	true if $x > y$, else false	Greater than
lt	true if $x < y$, else false	Less than
and	$x \text{ And } y$	Bit-wise
or	$x \text{ Or } y$	Bit-wise
not	Not y	Bit-wise

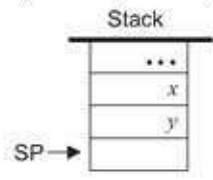


Figure 7.5 Arithmetic and logical stack commands.

Memory Access Commands All the memory segments are accessed by the same two commands:

- push *segment index* Push the value of *segment[index]* onto the stack.
- pop *segment index* Pop the top stack value and store it in *segment[index]*.

Segment	Purpose	Comments
argument	Stores the function's arguments.	Allocated dynamically by the VM implementation when the function is entered.
local	Stores the function's local variables.	Allocated dynamically by the VM implementation and initialized to 0's when the function is entered.
static	Stores static variables shared by all functions in the same .vm file.	Allocated by the VM imp. for each .vm file; shared by all functions in the .vm file.
constant	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.
this that	General-purpose segments. Can be made to correspond to different areas in the heap. Serve various programming needs.	Any VM function can use these segments to manipulate selected areas on the heap.
pointer	A two-entry segment that holds the base addresses of the this and that segments.	Any VM function can set pointer 0 (or 1) to some address; this has the effect of aligning the this (or that) segment to the heap area beginning in that address.
temp	Fixed eight-entry segment that holds temporary variables for general use.	May be used by any VM function for any purpose. Shared by all functions in the program.

Figure 7.6 The memory segments seen by every VM function.

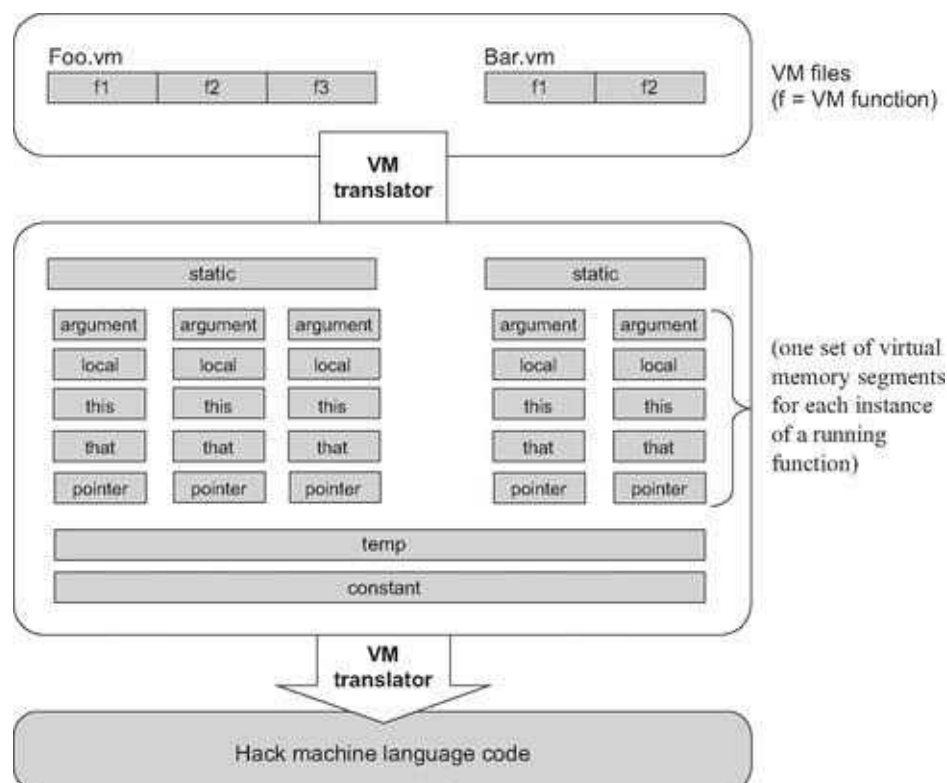


Figure 7.7 The virtual memory segments are maintained by the VM implementation.

Where *segment* is one of the eight segment names and *index* is a non-negative integer. For example, push argument 2 followed by pop local 1 will store the value of the function's third argument in the function's second local variable (each segment's index starts at 0).

The relationship among VM files, VM functions, and their respective virtual memory segments is depicted in figure 7.7.

In addition to the eight memory segments, which are managed explicitly by VM push and pop commands, the VM implementation manages two implicit data structures called *stack* and *heap*. These data structures are never mentioned directly, but their states change in the background, as a side effect of VM commands.

The Stack Consider the commands sequence push argument 2 and pop local 1, mentioned before. The working memory of such VM operations 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.

The Heap Another memory element that exists in the VM's background is the *heap*. The heap is the name of the RAM area dedicated for storing objects and arrays data. These objects and arrays can be manipulated by VM commands, as we will see shortly.

7.2.4 Program Flow and Function Calling Commands

The VM features six additional commands that are discussed at length in the next chapter. For completeness, these commands are listed here.

<code>label <i>symbol</i></code>	// Label declaration
<code>goto <i>symbol</i></code>	// Unconditional branching
<code>if-goto <i>symbol</i></code>	// Conditional branching

Program Flow Commands

<code>function <i>functionName nLocals</i></code>	// Function declaration, specifying the // number of the function's local variables
<code>call <i>functionName nArgs</i></code>	// Function invocation, specifying the // number of the function's arguments
<code>return</code>	// Transfer control back to the calling function

Function Calling Commands

(In this list of commands, *function Name* is a symbol and *nLocals* and *nArgs* are non-negative integers.)

7.2.5 Program Elements in the Jack-VM-Hack Platform

We end the first part of the VM specification with a top-down view of all the program elements that emerge from the full compilation of a typical high-level program. At the top of figure 7.8 we see a Jack program, consisting of two classes (Jack, a simple Java-like language, is described in chapter 9). Each Jack class consists of one or more methods. When the Jack compiler is applied to a directory that includes n class files, it produces n VM files (in the same directory). Each Jack method xxx within a class Yyy is translated into one *VM function* called $Yyy.xxx$ within the corresponding VM file.

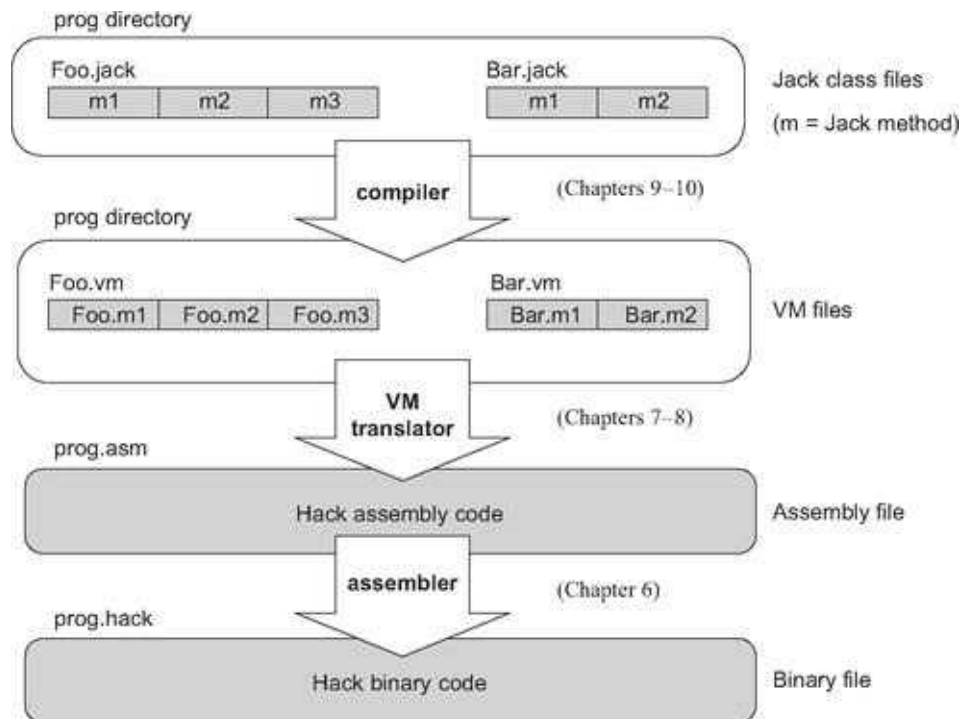


Figure 7.8 Program elements in the Jack-VM-Hack platform.

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 assembly program does two main things. First, it emulates the virtual memory segments of each VM function and file, 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—those translated from the VM commands. If all works well, that is, if the compiler and the VM translator and the assembler are implemented correctly, the target platform will end up effecting the behavior mandated by the original Jack program.

7.2.6 VM Programming Examples

We end this section by illustrating how the VM abstraction can be used to express typical programming tasks found in high-level programs. We give three examples: (i) a typical arithmetic task, (ii) typical array handling, and (iii) typical object handling. These examples are irrelevant to the VM implementation, and in fact the entire section 7.2.6 can be skipped without losing the thread of the chapter.

The main purpose of this section is to illustrate how the compiler developed in chapters 10-11 will use the VM abstraction to translate high-level programs into VM code. Indeed, VM programs are rarely written by human programmers, but rather by compilers. Therefore, it is instructive to begin each example with a high-level code fragment, then show its equivalent representation using VM code. We use a C-style syntax for all the high-level examples.

A Typical Arithmetic Task Consider the multiplication algorithm shown at the top of figure 7.9. How should we (or more likely, the compiler) express this algorithm in the VM language? First, high-level structures like `for` and `while` must be rewritten using the VM's simple "goto logic." In a similar fashion, high-level arithmetic and Boolean operations must be expressed using stack-oriented commands. The resulting code is shown in figure 7.9. (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.)

Let us focus on the virtual segments depicted at the bottom of figure 7.9. 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.

Let us now focus on the VM representation of the algorithm. Recall that VM commands cannot 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 map `x`, `y`, `sum` and `j` on 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 staging this virtual worldview for every VM function just before it starts running is the VM implementation, as we will see in the next chapter.

High-level code (C style)

```
int mult(int x, int y) {  
    int sum;  
    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
```

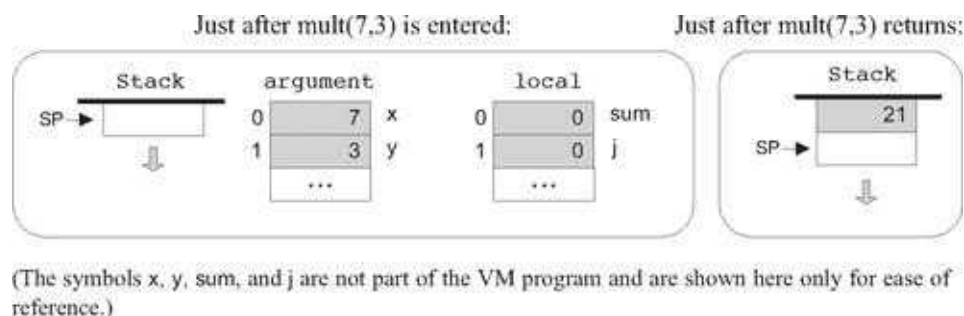
Pseudo VM code

```
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
```

Figure 7.9 VM programming example.



Array Handling An array is an indexed collection of objects. Suppose that a high-level program has created an array of ten integers called `bar` and filled it with some ten numbers. Let us assume that the array's base has been mapped (behind the scene) on RAM address 4315. Suppose now that the high-level program wants to execute the command `bar[2]=19`. How can we implement this operation at the VM level?

In the C language, such an operation can be also specified as `*(bar+2)=19`, meaning “set the RAM

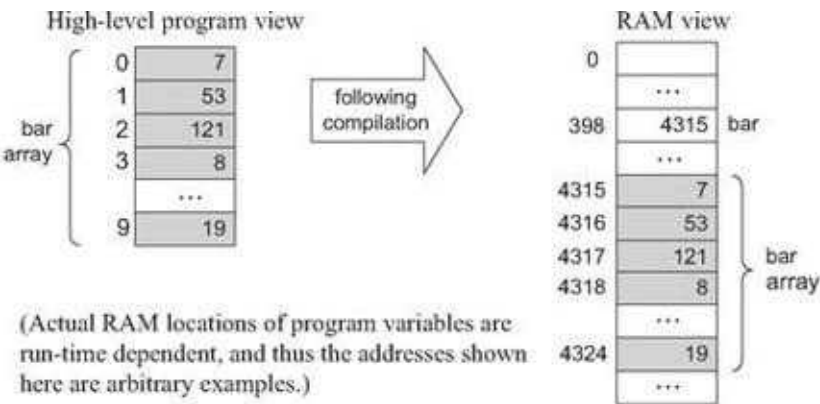
location whose address is (bar+2) to 19.” As shown in figure 7.10, this operation lends itself perfectly well to the VM language.

It remains to be seen, of course, how a high-level command like bar [2]= 19 is translated in the first place into the VM code shown in figure 7.10. This transformation is described in section 11.1.1, when we discuss the code generation features of the compiler.

Object Handling High-level programmers view objects as entities that encapsulate data (organized as *fields*, or *properties*) and relevant code (organized as *methods*). Yet physically speaking, the data of each object instance is serialized on the RAM as a list of numbers representing the object’s field values. Thus the low-level handling of objects is quite similar to that of arrays.

For example, consider an animation program designed to juggle some 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 object instances, it will be stored in the RAM. In particular, whenever a program creates a new object, the compiler computes the object’s size in terms of words and the operating system finds and allocates enough RAM space to store it (the exact details of this operation are discussed in chapter 11). For now, let us assume that our b object has been allocated RAM addresses 3012 to 3015, as shown in figure 7.11.



VM code

```
/* Assume that the bar array is the first local variable declared in the
   high-level program. The following VM code implements the operation
   bar[2]=19, i.e., *(bar+2)=19. */
push local 0      // Get bar's base address
push constant 2
add
pop pointer 1     // Set that's base to (bar+2)
push constant 19
pop that 0        // *(bar+2)=19
...
```

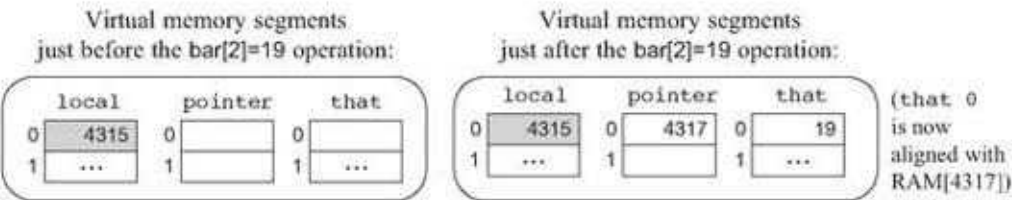


Figure 7.10 VM-based array manipulation using the pointer and that segments.

Suppose now that a certain method 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 VM representation of this logic is shown in figure 7.11.

When we set pointer 0 to the value of argument 0, we are effectively setting the base of the virtual this segment to the object's base address. From this point on, VM commands can access any field in the object using the virtual memory segment `this` and an index relative to the object's base-address in memory.

But how did the compiler translate `b.radius=17` into the VM code shown in figure 7.11? And how did the compiler know that the `radius` field of the object corresponds to the third field in its actual representation? We return to these questions in section 11.1.1, when we discuss the code generation features of the compiler.

7.3 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 platform. In particular, each data structure mentioned in the VM specification, namely, the stack and the virtual memory segments, must be represented in some way by the target platform. Second, each VM command must be translated into a series of instructions that effect the command's semantics on the target platform.

This section describes how to implement the VM specification (section 7.2) on the Hack platform. 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.

7.3.1 Standard VM Mapping on the Hack Platform, Part I

If you reread the virtual machine specification given so far, you will realize that it contains no assumption whatsoever about the architecture on which the VM can be implemented. When it comes to virtual machines, this 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.

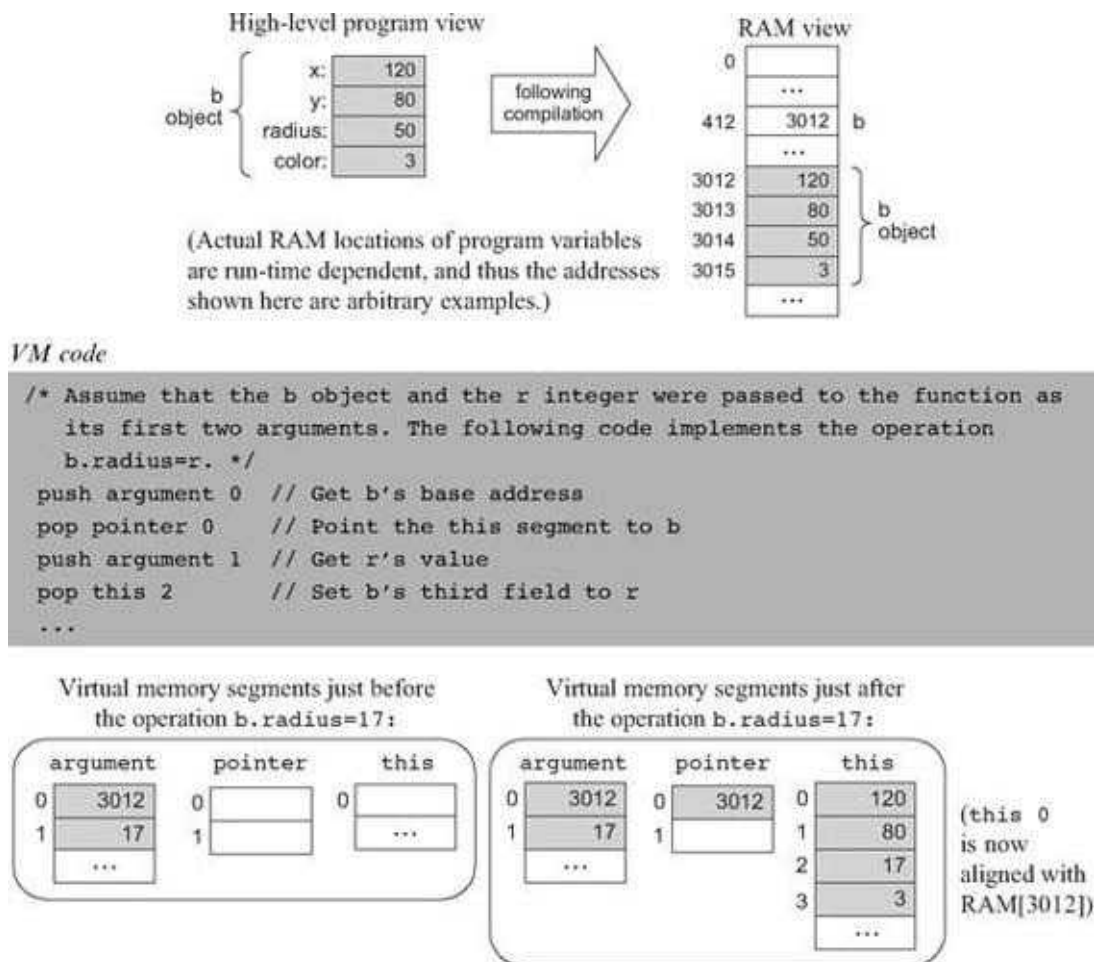


Figure 7.11 VM-based object manipulation using the pointer and this segments.

It follows that the VM designer can principally let programmers implement the VM on target platforms in any way they see fit. However, it is usually recommended that some guidelines be provided as to 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, they entail a public contract that regulates how VM-based programs can interact with programs produced by compilers that don't use this VM (e.g., compilers that produce binary code directly). Second, we wish to allow the developers of the VM implementation to run standardized tests, namely, 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 (see figure 7.7). 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 memory maps of I/O devices. The VM implementation should use this space as follows:

<i>RAM addresses</i>	<i>Usage</i>
0–15	Sixteen virtual registers, usage 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

Recall that according to the Hack Machine Language Specification, RAM addresses 0 to 15 can be referred to by any assembly program using the symbols R0 to R15, respectively. In addition, the specification states that 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. The expected use of these registers in the VM context is described as follows:

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

Memory Segments Mapping

local, argument, this, that: Each one of these segments is mapped directly on the 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 entry of any one of these segments should be translated to assembly code that accesses address $(base + i)$ in the RAM, where $base$ is the current value stored in the register dedicated to the respective segment.

pointer, temp: These segments are each mapped directly onto a fixed area in the RAM. The pointer segment is mapped on RAM locations 3-4 (also called THIS and THAT) and the temp segment on locations 5-12 (also called R5, R6,..., R12). Thus access to pointer i should be translated to assembly code that accesses RAM location $3 + i$, and access to temp i should be translated to assembly code that accesses RAM location $5 + i$.

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 $\langle constant\ i \rangle$ 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 We recap 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 predefined symbols point, respectively, to the stack top and to the base addresses of the virtual segments <code>local</code> , <code>argument</code> , <code>this</code> , and <code>that</code> .
R13-R15	These predefined symbols can be used for any purpose.
<code>xxx.j</code> symbols	Each static variable j in file <code>xxx.vm</code> is translated into the assembly symbol <code>xxx.j</code> . In the subsequent assembly process, these symbolic variables will be allocated RAM space by the Hack assembler.
Flow of control symbols	The implementation of the VM commands <code>function</code> , <code>call</code> , and <code>label</code> involves generating special label symbols, to be discussed in chapter 8.

7.3.2 Design Suggestion for the VM Implementation

The VM translator should accept a single command line parameter, as follows:

```
prompt> VMtranslator source
```

Where *source* is either a file name of the form Xxx.vm (the extension is mandatory) or a directory name containing one or more .vm files (in which case there is no extension). 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 mapping on the Hack platform.

7.3.3 Program Structure

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

The *Parser* Module

Parser: Handles the parsing of a single .vm file, and encapsulates access to the input code. It reads VM commands, parses them, and provides convenient access to their components. In addition, it 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 hasMoreCommands () 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	Returns the type of the current VM command. C_ARITHMETIC is returned for all the arithmetic commands.
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.

Routine	Arguments	Returns	Function
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.

The *Code Writer* Module

CodeWriter: 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 (C_PUSH or C_POP), segment (string), index (int)	—	Writes the assembly code that is the translation of the given command, where command is either 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 separate Parser for handling each input file and a single CodeWriter for handling the output.

7.4 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-tier compilation model. In the frontend tier, covered in chapters 10 and 11, the high-level code is translated into an intermediate code, running on a virtual machine. In the backend tier, covered in this and in the next chapter, the intermediate code is translated into the machine language of a target hardware platform (see figures 7.1 and 7.9).

The idea of formulating the intermediate code as the explicit language of a virtual machine goes back to the late 1970s, when it was used by several popular Pascal compilers. These compilers generated an intermediate “p-code” that could execute on any computer that implemented it. Following the wide spread use of the World Wide Web in the mid-1990s, cross-platform compatibility became a universally vexing issue. In order to address the problem, the Sun Microsystems company sought to develop a new programming language that could potentially run on any computer and digital device connected to the Internet. The language that emerged from this initiative—*Java*—is also founded on an intermediate code execution model called the Java Virtual Machine, or JVM.

The JVM is a specification that describes an intermediate language called *byte-code*—the target language of Java compilers. Files written in bytecode are then used for dynamic code 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 programs, also called Java Run-time Environments (JREs), are widely available for numerous processor/OS combinations, including game consoles and cell phones.

In the early 2000s, Microsoft entered the fray with its .NET infrastructure. The centerpiece of .NET is a virtual machine model called Common Language Runtime (CLR). 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 interoperate and share the software libraries of a common run-time environment.

We note in closing that a crucial ingredient that must be added to the virtual machine model before its full potential of interoperability 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 unified services like memory management, GUI utilities, string functions, math functions, and so on. One such library will be described and built in chapter 12.

7.5 Project

This section describes how to build the VM translator presented in the chapter. In the next chapter we will extend this basic translator with additional functionality, leading to a full-scale VM implementation. Before you get started, two comments are in order. First, section 7.2.6 is irrelevant to this project. Second, since the VM translator is designed to generate Hack assembly code, it is recommended to refresh your memory about the Hack assembly language rules (section 4.2).

Objective Build the first part of the VM translator (the second part is implemented in Project 8), focusing on the implementation of the stack arithmetic and memory access commands of the VM language.

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 in handy in this project is the visual VM emulator supplied with the book. This program allows experimenting with a working VM implementation before you set out to build one yourself. For more information about this tool, refer to the VM emulator tutorial.

Contract Write a VM-to-Hack translator, conforming to the VM Specification, Part I (section 7.2) and to the Standard VM Mapping on the Hack Platform, Part I (section 7.3.1). Use it to translate the test VM programs supplied here, 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.

Proposed Implementation Stages

We recommend building the translator in two stages. This will allow you to unit-test your implementation incrementally, using the test programs supplied here.

Stage I: Stack Arithmetic Commands The first version of your VM translator should implement the nine stack arithmetic and logical commands of the VM language as well as the push constant x command (which, among other things, will help in testing the nine former commands). Note that the latter is the generic push command for the special case where the first argument is constant and the second argument is some decimal constant.

Stage II: Memory Access Commands The next version of your translator should include a full implementation of the VM language's push and pop commands, handling all eight memory segments. We

suggest breaking this stage into the following substages:

0. You have already handled the constant segment.

1. Next, handle the 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.

Test Programs

The five VM programs listed here are designed to unit-test the proposed implementation stages just described.

Stage I: Stack Arithmetic

■ SimpleAdd: Pushes and adds two constants.

■ StackTest: Executes a sequence of arithmetic and logical operations on the stack.

Stage II: Memory Access

■ BasicTest: Executes pop and push operations using the virtual memory segments.

■ PointerTest: Executes pop and push operations using the pointer, this, and that segments.

■ StaticTest: Executes pop and push operations using the static segment.

For each program Xxx we supply four files, beginning with the program's code in Xxx.vm. The XxxVME.tst script allows running the program on the supplied VM emulator, so that you can gain familiarity with the program's intended operation. After translating the program using your VM translator, the supplied Xxx.tst and Xxx.cmp scripts allow testing the translated assembly code on the CPU emulator.

Tips

Initialization In order for any translated VM program to start running, it must 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 anchor the base addresses of the virtual segments in selected RAM locations. Both issues—startup code and segments initializations—are

implemented in the next project. The difficulty of course is that we need these initializations in place in order to execute the test programs given in this project. The good news is that you should not worry about these issues at all, since the supplied test scripts carry out all the necessary initializations 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 Xxx.vm program on the supplied VM emulator, using the XxxVME.tst test script, to get acquainted with the intended program's behavior.
2. Use your partial translator to translate the .vm file. 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 visible 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 VM 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.

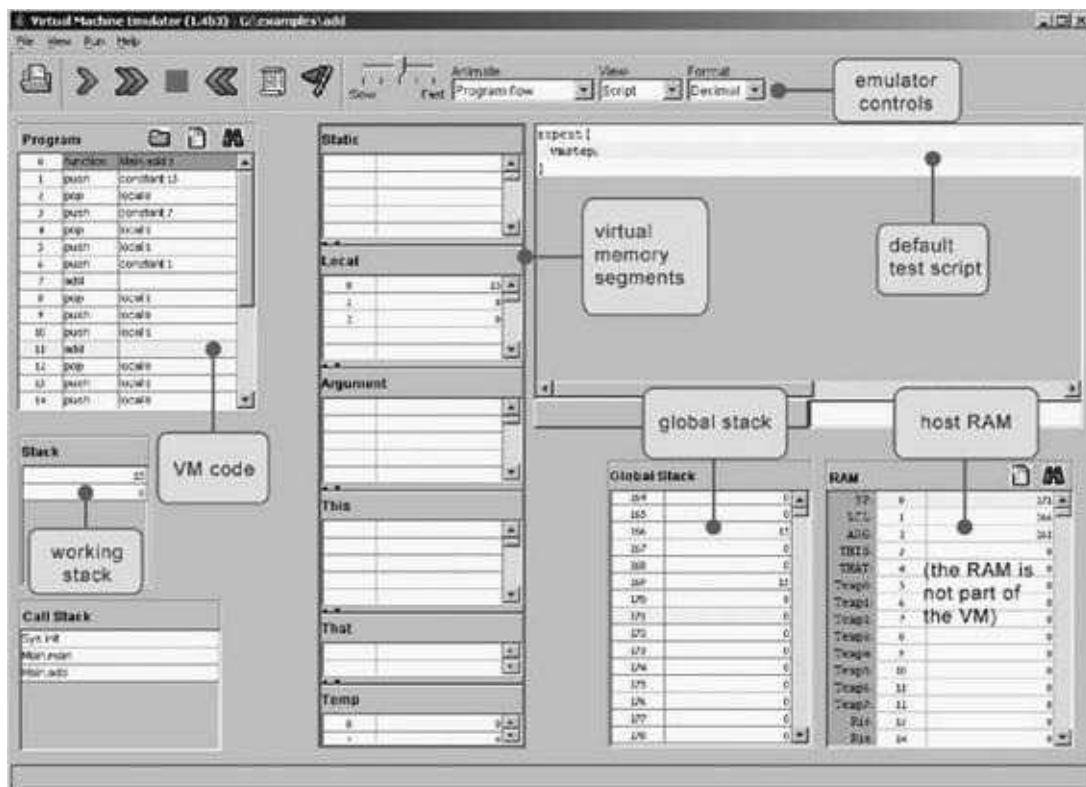


Figure 7.12 The VM emulator supplied with the book.

Tools

The VM Emulator The book's software suite includes a Java-based VM implementation. This VM emulator allows executing VM programs directly, without having to translate them first into machine language. This practice enables experimentation with the VM environment before you set out to implement one yourself. Figure 7.12 is a typical screen shot of the VM emulator in action.