8 Virtual Machine II: Control

If everything seems under control, you're just not going fast enough.

—Mario Andretti (b. 1940), race car champion

Chapter 7 introduced the notion of a *virtual machine* (VM), and project 7 began implementing our abstract virtual machine and VM language over the Hack platform. The implementation entailed developing a program that translates VM commands into Hack assembly code. Specifically, in the previous chapter we learned how to use and implement the VM's arithmetic-logical commands and push/pop commands; in this chapter we'll learn how to use and implement the VM's branching commands and function commands. As the chapter progresses, we'll extend the basic translator developed in project 7, ending with a full-scale VM translator over the Hack platform. This translator will serve as the back-end module of the compiler that we will build in chapters 10 and 11.

In any Gre at Gems in Applied Computer Science contest, stack processing will be a strong finalist. The previous chapter showed how arithmetic and Boolean expressions can be represented and evaluated by elementary operations on a stack. This chapter goes on to show how this remarkably simple data structure can also support remarkably complex tasks like nested function calling, parameter passing, recursion, and the various memory allocation and recycling tasks required to support program execution during run-time. Most programmers tend to take these services for granted, expecting the compiler and the operating system to deliver them, one way or another. We are now in a position to open up this black box and see how these fundamental programming mechanisms are actually realized.

Run-time system: Every computer system must specify a run-time model. This model answers essential questions without which programs cannot run: how to start a program's execution, what the computer should do when a program terminates, how to pass arguments from one function to another, how to allocate memory resources to running functions, how to free memory resources when they are no longer needed, and so on.

In Nand to Tetris, these issues are addressed by the *VM language* specification, along with the *standard mapping on the Hack platform* specification. If a VM translator is developed according to these guidelines, it will end up realizing an executable run-time system. In particular, the VM translator will not only translate the VM commands (`push`, `pop`, `add`, and so on) into assembly instructions—it will also generate assembly code that realizes an envelope

in which the program runs. All the questions mentioned above—how to start a program, how to manage the function call-and-return behavior, and so on—will be answered by generating enabling assembly code that wraps the code proper. Let's see an example.

8.1 High-Level Magic

High-level languages allow writing programs in high-level terms. For example, an expression like $x = -b + \sqrt{b^2 - 4 \cdot a \cdot c}$ can be written as `x = -b + sqrt(power(b,2) - 4 * a * c)`, which is almost as descriptive as the real thing. Note the difference between primitive operations like + and − and functions like sqrt and power. The former are built into the basic syntax of the high-level language. The latter are extensions of the basic language.

The unlimited capacity to extend the language at will is one of the most important features of high-level programming languages. Of course, at some point, someone must implement functions like sqrt and power. However, the story of implementing these abstractions is completely separate from the story of using them. Therefore, application programmers can assume that each one of these functions will get executed—somehow—and that following its execution, control will return—somehow—to the next operation in one's code. Branching commands endow the language with additional expressive power, allowing writing conditional code like `if ! (a == 0){ x = (-b + sqrt(power(b,2) - 4 * a * c)) / (2 * a)} else { x = -c / b }`. Once again, we see that high-level code enables the expression of high-level logic—in this case the algorithm for solving quadratic equations—almost directly.

Indeed, modern programming languages are programmer-friendly, offering useful and powerful abstractions. It is a sobering thought, though, that no matter how high we allow our high-level language to be, at the end of the day it must be realized on some hardware platform that can only execute primitive machine instructions. Thus, among other things, compiler and VM architects must find low-level solutions for realizing branching and function call-and-return commands.

Functions—the bread and butter of modular programming—are standalone programming units that are allowed to call each other for their effect. For example, solve can call sqrt, and sqrt, in turn, can call power. This calling sequence can be as deep as we please, as well as recursive. Typically, the calling function (the *caller*) passes arguments to the called function (the *callee*) and suspends its execution until the latter completes *its* execution. The callee uses the passed arguments to execute or compute something and then returns a value (which may be void) to the caller. The caller then snaps back into action, resuming its execution.

In general then, whenever one function (the *caller*) calls a function (the *callee*), someone must take care of the following overhead:

- Save the *return address*, which is the address within the caller's code to which execution must return after the callee completes its execution;
- Save the memory resources of the caller;

- Allocate the memory resources required by the callee;
- Make the arguments passed by the caller available to the callee's code;
- Start executing the callee's code.

When the callee terminates and returns a value, someone must take care of the following overhead:

- Make the callee's *return value* available to the caller's code;
- Recycle the memory resources used by the callee;
- Reinstate the previously saved memory resources of the caller;
- Retrieve the previously saved *return address*;
- Resume executing the caller's code, from the return address onward.

Blissfully, high-level programmers don't have to ever think about all these nitty-gritty chores: the assembly code generated by the compiler handles them, stealthily and efficiently. And, in a two-tier compilation model, this housekeeping responsibility falls on the compiler's back end, which is the VM translator that we are now developing. Thus, in this chapter, we will uncover, among other things, the run-time framework that enables what is probably the most important abstraction in the art of programming: *function call-and-return*. But first, let's start with the easier challenge of handling branching commands.

8.2 Branching

The default flow of computer programs is sequential, executing one command after the other. For various reasons like embarking on a new iteration in a loop, this sequential flow can be redirected by branching commands. In low-level programming, branching is accomplished by *goto destination* commands. The destination specification can take several forms, the most primitive being the physical memory address of the instruction that should be executed next. A slightly more abstract specification is established by specifying a symbolic label (bound to a physical memory address). This variation requires that the language be equipped with a labeling directive, designed to assign symbolic labels to selected locations in the code. In our VM language, this is done using a labeling command whose syntax is label *symbol*.

With that in mind, the VM language supports two forms of branching. *Unconditional branching* is effected using a goto *symbol* command, which means: jump to execute the command just after the label *symbol* command in the code. *Conditional branching* is effected using the if-goto *symbol* command, whose semantics is: Pop the topmost value off the stack; if it's not false, jump to execute the command just after the label *symbol* command; otherwise, execute the next command in the code. This contract implies that before specifying a conditional goto command, the VM code writer (for example, a compiler) must first specify a condition. In our VM language, this is done by pushing a Boolean expression onto the stack.

For example, the compiler that we'll develop in chapters 10–11 will translate `if (n < 100) goto LOOP` into `push n, push 100, lt, if-goto LOOP`.

Example: Consider a function that receives two arguments, *x* and *y*, and returns the product *x* · *y*. This can be done by adding *x* repetitively to a local variable, say `sum`, *y* times, and then returning `sum`'s value. A function that implements this naïve multiplication algorithm is listed in figure 8.1. This example illustrates how a typical looping logic can be expressed using the VM branching commands `goto`, `if-goto`, and `label`.

High-level code

```
// Returns x * y
int mult(int x, int y) {
   int sum = 0;
   int i = 0;
   while (i < y) {
      sum += x;
      i++;
   }
   return sum;
}
```

VM code

```
// Returns x * y
function mult(x,y)
   push 0
   pop sum
   push 0
   pop i
label WHILE_LOOP
   push i
   push y
   lt
   neg
   if-goto WHILE_END
   push sum
   push x
   add
   pop sum
   push i
   push 1
   add
   pop i
   goto WHILE_LOOP
label WHILE_END
   push sum
   return
```

Figure 8.1 Branching commands action. (The VM code on the right uses symbolic variable names instead of virtual memory segments, to make it more readable.)

Notice how the Boolean condition `!(i < y)`, implemented as `push i, push y, lt, ng`, is pushed onto the stack just before the `if-goto WHILE_END` command. In chapter 7 we saw that VM commands can be used to express and evaluate any Boolean expression. As we see in figure 8.1, high-level control structures like `if` and `while` can be easily realized using nothing more than `goto` and `if-goto` commands. In general, any flow of control structure found in high-level programming languages can be realized using our (rather minimal set of) VM logical and branching commands.

Implementation: Most low-level machine languages, including Hack, feature means for declaring symbolic labels and for effecting conditional and unconditional "goto label" actions. Therefore, if we base the VM implementation on a program that translates VM

commands into assembly instructions, implementing the VM branching commands is a relatively simple matter.

Operating system: We end this section with two side comments. First, VM programs are not written by humans. Rather, they are written by compilers. Figure 8.1 illustrates source code on the left and VM code on the right. In chapters 10–11 we'll develop a *compiler* that translates the former into the latter. Second, note that the mult implementation shown in figure 8-1 is inefficient. Later in the book we'll present optimized multiplication and division algorithms that operate at the bit level. These algorithms will be used for realizing the Math.multiply and Math.divide functions, which are part of the operating system that we will build in chapter 12.

Our OS will be written in the Jack language, and translated by a Jack compiler into the VM language. The result will be a library of eight files named Math.vm, Memory.vm, String.vm, Array.vm, Output.vm, Screen.vm, Keyboard.vm, and Sys.vm (the OS API is given in appendix 6). Each OS file features a collection of useful functions that any VM function is welcome to call for their effect. For example, whenever a VM function needs multiplication or division services, it can call the Math.multiply or Math.divide function.

8.3 Functions

Every programming language is characterized by a fixed set of built-in operations. In addition, high-level and some low-level languages offer the great freedom of extending this fixed repertoire with an open-ended collection of programmer-defined operations. Depending on the language, these canned operations are typically called *subroutines*, *procedures*, *methods*, or *functions*. In our VM language, all these programming units are referred to as *functions*.

In well-designed languages, built-in commands and programmer-defined functions have the same look and feel. For example, to compute $x + y$ on our stack machine, we push x, push y, and add. In doing so, we expect the add implementation to pop the two top values off the stack, add them up, and push the result onto the stack. Suppose now that either we, or someone else, has written a *power* function designed to compute x^y. To use this function, we follow exactly the same routine: we push x, push y, and call power. This consistent calling protocol allows composing primitive commands and function calls seamlessly. For example, expressions like $(x + y)^3$ can be evaluated using push x, push y, add, push 3, call power.

We see that the only difference between applying a primitive operation and invoking a function is the keyword call preceding the latter. Everything else is exactly the same: both operations require the caller to set the stage by pushing arguments onto the stack, both operations are expected to consume their arguments, and both operations are expected to push return values onto the stack. This calling protocol has an elegant consistency which, we hope, is not lost on the reader.

Example: Figure 8.2 shows a VM program that computes the function $\sqrt{x^2 + y^2}$, also known as *hypot*. The program consists of three functions, with the following run-time behavior: main calls hypot, and then hypot calls mult, twice. There is also a call to a sqrt function, which we don't track, to reduce clutter.

```
0 function main()
// Computes hypot(3,4)
1    push 3
2    push 4
3    call hypot
4    return

5 function hypot(x,y)
// Computes sqrt(x*x + y*y)
6    push x
7    push x
8    call mult
9    push y
10   push y
11   call mult
12   add
13   call sqrt
14   return

15 function mult(x,y)
// Computes x * y (same as in figure 8.1)
16   push 0
17   pop sum
18   push 0
19   pop i
     ...
36   push sum
37   return
```

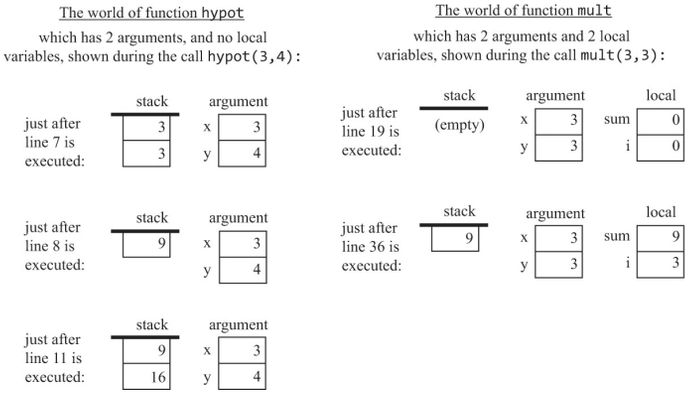

Figure 8.2 Run-time snapshots of selected stack and segment states during the execution of a three-function program. The line numbers are not part of the code and are given for reference only.

The bottom part of figure 8.2 shows that during run-time, each function sees a private world, consisting of its own working stack and memory segments. These separate worlds are connected through two "wormholes": when a function says call mult, the arguments that it pushed onto its stack prior to the call are somehow passed to the argument segment of the callee. Likewise, when a function says return, the last value that it pushed onto its stack just before returning is somehow copied onto the stack of the caller, replacing the previously pushed arguments. These hand-shaking actions are carried out by the VM implementation, as we now turn to describe.

Implementation: A computer program consists of typically several and possibly many

functions. Yet at any given point during run-time, only a few of these functions are actually doing something. We use the term *calling chain* to refer, conceptually, to all the functions that are currently involved in the program's execution. When a VM program starts running, the calling chain consists of one function only, say, main. At some point, main may call another function, say, foo, and that function may call yet another function, say, bar. At this point the calling chain is `main→foo→bar`. Each function in the calling chain waits for the function that it called to return. Thus, the only function that is truly active in the calling chain is the last one, which we call the *current function*, meaning the currently executing function.

In order to carry out their work, functions normally use *local* and *argument* variables. These variables are temporary: the memory segments that represent them must be allocated when the function starts executing and can be recycled when the function returns. This memory management task is complicated by the requirement that function calling is allowed to be arbitrarily nested, as well as recursive. During run-time, each function call must be executed independently of all the other calls and maintain its own stack, local variables, and argument variables. How can we implement this unlimited nesting mechanism and the memory management tasks associated with it?

The property that makes this housekeeping task tractable is the linear nature of the call-and-return logic. Although the function calling chain may be arbitrarily deep as well as recursive, at any given point in time only one function executes at the chain's end, while all the other functions up the calling chain are waiting for it to return. This *Last-In-First-Out* processing model lends itself perfectly to the stack data structure, which is also LIFO. Let's takes a closer look.

Assume that the current function is foo. Suppose that foo has already pushed some values onto its working stack and has modified some entries in its memory segments. Suppose that at some point foo wants to call another function, bar, for its effect. At this point we have to put foo's execution on hold until bar will terminate *its* execution. Now, putting foo's working stack on hold is not a problem: because the stack grows only in one direction, the working stack of bar will never override previously pushed values. Therefore, saving the working stack of the caller is easy—we get it "for free" thanks to the linear and unidirectional stack structure. But how can we save foo's memory segments? Recall that in chapter 7 we used the pointers LCL, ARG, THIS, and THAT to refer to the base RAM addresses of the local, argument, this, and that segments of the current function. If we wish to put these segments on hold, we can push their pointers onto the stack and pop them later, when we'll want to bring foo back to life. In what follows, we use the term *frame* to refer, collectively, to the set of pointer values needed for saving and reinstating the function's state.

We see that once we move from a single function setting to a multifunction setting, the humble stack begins to attain a rather formidable role in our story. Specifically, we now use the same data structure to hold both the working stacks as well as the frames of all the functions up the calling chain. To give it the respect that it deserves, from now on we'll refer to this hard-working data structure as the *global stack*. See figure 8.3 for the details.

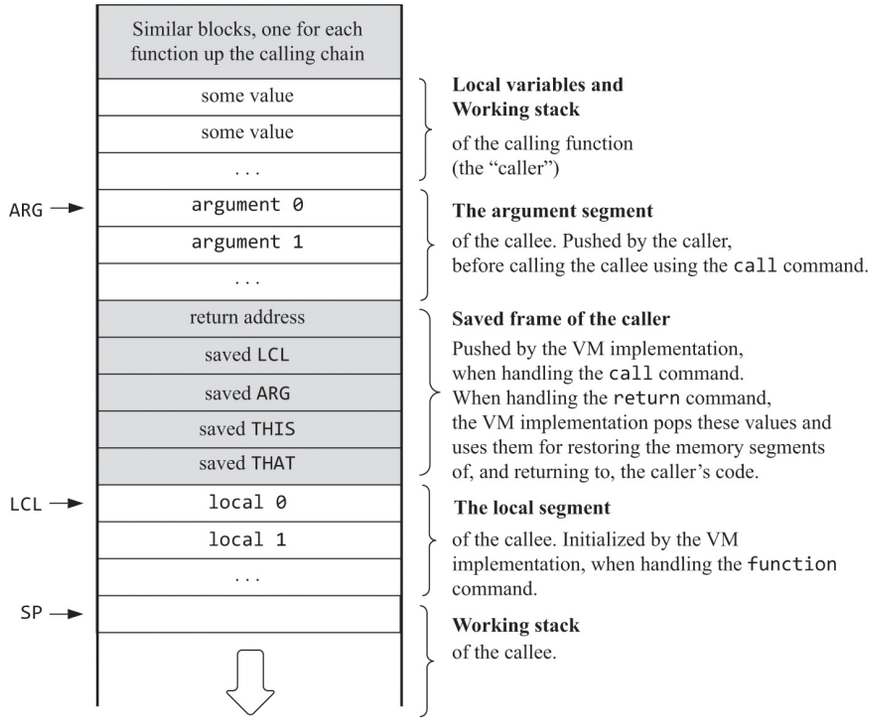

Figure 8.3 The global stack, shown when the callee is running. Before the callee terminates, it pushes a return value onto the stack (not shown). When the VM implementation handles the return command, it copies the return value onto argument 0, and sets SP to point to the address just following it. This effectively frees the global stack area below the new value of SP. Thus, when the caller resumes its execution, it sees the return value at the top of its working stack.

As shown in figure 8.3, when handling the call *functionName* command, the VM implementation pushes the caller's frame onto the stack. At the end of this housekeeping, we are ready to jump to executing the callee's code. This mega jump is not hard to implement. As we'll see later, when handling a function *functionName* command, we use the function's name to create, and inject into the generated assembly code stream, a unique symbolic label that marks where the function starts. Thus, when handling a "function *functionName*" command, we can generate assembly code that effects a "goto *functionName*" operation. When executed, this command will effectively transfer control to the callee.

Returning from the callee to the caller when the former terminates is trickier, since the VM return command specifies no return address. Indeed, the caller's anonymity is inherent in the notion of a function call: functions like mult or sqrt are designed to serve any caller, implying that a return address cannot be specified a priori. Instead, a return command is interpreted as follows: redirect the program's execution to the memory location holding the command just following the call command that invoked the current function.

The VM implementation can realize this contract by (i) saving the return address just before control is transferred to executing the caller and (ii) retrieving the return address and jumping to it just after the callee returns. But where shall we save the return address? Once again, the resourceful stack comes to the rescue. To remind you, the VM translator advances from one VM command to the next, generating assembly code as it goes along. When we encounter a call foo command in the VM code, we know exactly which command should be executed when foo terminates: it's the assembly command just after the assembly commands

that realize the call foo command. Thus, we can have the VM translator plant a label right there, in the generated assembly code stream, and push this label onto the stack. When we later encounter a return command in the VM code, we can pop the previously saved return address off the stack—let's call it *returnAddress*—and effect the operation goto *returnAddress* in assembly. This is the low-level trick that enables the run-time magic of redirecting control back to the right place in the caller's code.

The VM implementation in action: We now turn to give a step-by-step illustration of how the VM implementation supports the function call-and-return action. We will do it in the context of executing a factorial function, designed to compute *n*! recursively. Figure 8.4 gives the program's code, along with selected snapshots of the global stack during the execution of factorial(3). A complete run-time simulation of this computation should also include the call-and-return action of the mult function, which, in this particular run-time example, is called twice: once before factorial(2) returns, and once before factorial(3) returns.

High-level code

```
// Tests the factorial function
int main() {
    return factorial(3);
}

// Computes n!
int factorial(int n) {
    if (n==1)
       return 1;
    else
       return n * factorial(n-1);
}
```

VM code

```
// Tests the factorial function
function main
   push 3
   call factorial
   return
// Computes n!
function factorial(n)
   push n
   push 1
   eq
   if-goto BASE_CASE
   push n
   push n
   push 1
   sub
   call factorial
   call mult
   return
label BASE_CASE
   push 1
   return
```

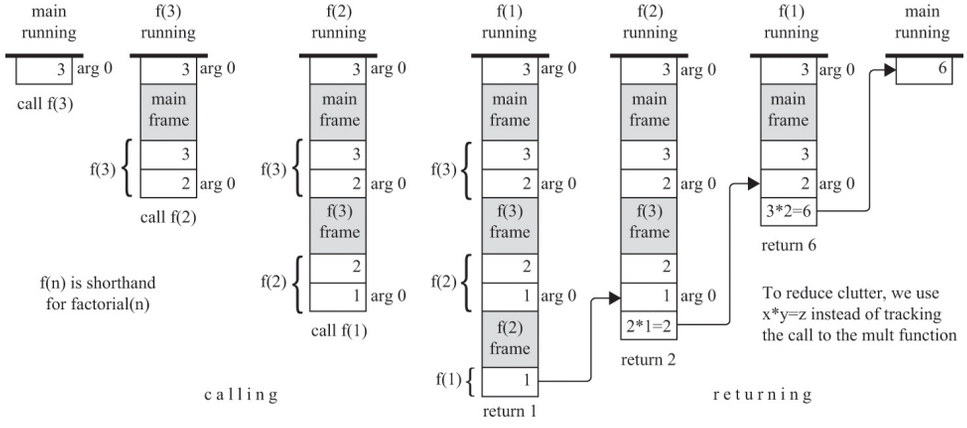

Figure 8.4 Several snapshots of the *global stack*, taken during the execution of the main function, which calls factorial to compute 3!. The running function sees only its working stack, which is the unshaded area at the tip of the global stack; the other unshaded areas in the global stack are the working stacks of functions up the calling chain, waiting for the currently running function to return. Note that the shaded areas are not "drawn to scale," since each frame consists of five words, as

shown in figure 8.3.

Focusing on the leftmost and rightmost parts of the bottom of figure 8.4, here is what transpired from main's perspective: "To set the stage, I pushed the constant 3 onto the stack, and then called factorial for its effect (see the leftmost stack snapshot). At this point I was put to sleep; at some later point in time I was woken up to find out that the stack now contains 6 (see the final and rightmost stack snapshot); I have no idea how this magic happened, and I don't really care; all I know is that I set out to compute 3!, and I got exactly what I asked for." In other words, the caller is completely oblivious of the elaborate mini-drama that was unleashed by its call command.

As seen in figure 8.4, the back stage on which this drama plays out is the global stack, and the choreographer who runs the show is the VM implementation: Each call operation is implemented by saving the frame of the caller on the stack and jumping to execute the callee. Each return operation is implemented by (i) using the most recently stored frame for getting the return address within the caller's code and reinstating its memory segments, (ii) copying the topmost stack value (the return value) onto the stack location associated with argument 0, and (iii) jumping to execute the caller's code from the return address onward. All these operations must be realized by generated assembly code.

Some readers may wonder why we have to get into all these details. There are at least three reasons why. First, we need them in order to implement the VM translator. Second, the implementation of the function call-and-return protocol is a beautiful example of low-level software engineering, so we can simply enjoy seeing it in action. Third, an intimate understanding of the virtual machine internals helps us become better and more informed high-level programmers. For example, tinkering with the stack provides an in-depth understanding of the benefits and pitfalls associated with recursion. Note that during run-time, each recursive call causes the VM implementation to add to the stack a memory block consisting of arguments, function frames, local variables, and a working stack for the callee. Therefore, unchecked use of recursion may well lead to the infamous *stack overflow* run-time debacle. This, as well as efficiency considerations, leads compiler writers to try to reexpress recursive code as sequential code, where possible. But that's a different story that will be taken up in chapter 11.

8.4 VM Specification, Part II

So far in this chapter, we have presented general VM commands without committing to exact syntax and programming conventions. We now turn to specify formally the VM *branching* commands, the VM *function* commands, and the structure of VM *programs*. This completes the specification of the VM language that we began describing in VM Specification, Part I, in chapter 7.

It's important to reiterate that, normally, VM programs are not written by humans. Rather, they are generated by compilers. Therefore, the specifications described here are aimed at compiler developers. That is, if you write a compiler that is supposed to translate programs from some high-level language into VM code, the code that your compiler generates is expected to conform to the conventions described here.

Branching Commands

- label *label*: Labels the current location in the function's code. Only labeled locations can be jumped to. The scope of the label is the function in which it is defined. The *label* is a string composed of any sequence of letters, digits, underscore (_), dot (.), and colon (:) that does not begin with a digit. The label command can be located anywhere in the function, before or after the goto commands that refer to it.
- goto *label*: Effects an unconditional goto operation, causing execution to continue from the location marked by the label. The goto command and the labeled jump destination must be located in the same function.
- if-goto *label*: Effects a conditional goto operation. The stack's topmost value is popped; if the value is not zero, execution continues from the location marked by the label; otherwise, execution continues from the next command in the program. The if-goto command and the labeled jump destination must be located in the same function.

Function Commands

- function *functionName nVars*: Marks the beginning of a function named *functionName*. The command informs that the function has *nVars* local variables.
- call *functionName nArgs*: Calls the named function. The command informs that *nArgs* arguments have been pushed onto the stack before the call.
- return: Transfers execution to the command just following the call command in the code of the function that called the current function.

VM Program

VM programs are generated from high-level programs written in languages like Jack. As we'll see in the next chapter, a high-level Jack program is loosely defined as a collection of

one or more .jack class files stored in the same folder. When applied to that folder, the Jack compiler translates each class file *FileName*.jack into a corresponding file named *FileName*.vm, containing VM commands.

Following compilation, each *constructor*, *function* (*static method*), and *method* named bar in a Jack file *FileName*.jack is translated into a corresponding VM function, uniquely identified by the VM function name *FileName*.bar. The scope of VM function names is global: all the VM functions in all the .vm files in the program folder see each other and may call each other using the unique and full function name *FileName.functionName*.

Program entry point: One file in any Jack program must be named Main.jack, and one function in this file must be named main. Thus, following compilation, one file in any VM program is expected to be named Main.vm, and one VM function in this file is expected to be named Main.main, which is the application's entry point. This run-time convention is implemented as follows. When we start running a VM program, the first function that always executes is an argument-less VM function named Sys.init, which is part of the operating system. This OS function is programmed to call the entry point function in the user's program. In the case of Jack programs, Sys.init is programmed to call Main.main.

Program execution: There are several ways to execute VM programs, one of which is using the supplied *VM emulator* introduced in chapter 7. When you load a program folder containing one or more .vm files into the VM emulator, the emulator loads all the VM functions in all these files, one after the other (the order of the loaded VM functions is insignificant). The resulting code base is a sequence of all the VM functions in all the .vm files in the program folder. The notion of VM files ceases to exist, although it is implicit in the names of the loaded VM functions (*FileName.functionName*).

The Nand to Tetris VM emulator, which is a Java program, features a built-in implementation of the Jack OS, also written in Java. When the emulator detects a call to an OS function, for example, call Math.sqrt, it proceeds as follows. If it finds a corresponding function Math.sqrt command in the loaded VM code, the emulator executes the function's VM code. Otherwise, the emulator reverts to using its built-in implementation of the Math.sqrt method. This implies that as long as you use the supplied VM emulator for executing VM programs, there is no need to include OS files in your code. The VM emulator will service all the OS calls found in your code, using its built-in OS implementation.

8.5 Implementation

The previous section completed the specification of our VM language and framework. In this section we focus on implementation issues, leading up to the construction of a full-scale, VM-to-Hack translator. Section 8.5.1 proposes how to implement the function call-and-return protocol. Section 8.5.2 completes the standard mapping of the VM implementation over the Hack platform. Section 8.5.3 gives a proposed design and API for completing the VM

translator that we began building in project 7.

8.5.1 Function Call and Return

The events of calling a function and returning from a function call can be viewed from two perspectives: that of the *calling function*, also referred to as *caller*, and that of the *called function*, also referred to as *callee*. Both the caller and the callee have certain expectations, and certain responsibilities, regarding the handling of the call, function, and return commands. Fulfilling the expectations of one is the responsibility of the other. In addition, the VM implementation plays an important role in executing this contract. In what follows, the responsibilities of the VM implementation are marked by [†]:

The caller's view:	The callee's view:
• Before calling a function, I must push onto the stack as many arguments (*nArgs*) as the callee expects to get. • Next, I invoke the callee using the command call *fileName*.*functionName nArgs*. • After the callee returns, the argument values that I pushed before the call have disappeared from the stack, and a *return value* (that always exists) appears at the top of the stack. Except for this change, my working stack is exactly the same as it was before the call [†]. • After the callee returns, all my memory segments are exactly the same as they were before the call [†], except that the contents of my static segment may have changed, and the temp segment is undefined.	• Before I start executing, my argument segment has been initialized with the argument values passed by the caller, and my local variables segment has been allocated and initialized to zeros. My static segment has been set to the static segment of the VM file to which I belong, and my working stack is empty. The memory segments this, that, pointer, and temp are undefined upon entry [†]. • Before returning, I must push a return value onto the stack.

The VM implementation supports this contract by maintaining, and manipulating, the global stack structure described in figure 8.3. In particular, every function, call, and return command in the VM code is handled by generating assembly code that manipulates the global stack as follows: A call command generates code that saves the frame of the caller on the stack and jumps to execute the callee. A function command generates code that initializes the local variables of the callee. Finally, a return command generates code that copies the return value to the top of the caller's working stack, reinstates the segment pointers of the caller, and jumps to execute the latter from the return address onward. See figure 8.5 for the details.

VM command	*Assembly (pseudo) code, generated by the VM translator*	
`call` *f nArgs* (calls a function *f*, informing that *nArgs* arguments were pushed to the stack before the call)	`push` *returnAddress*	// generates a label and pushes it to the stack
	`push LCL`	// saves `LCL` of the caller
	`push ARG`	// saves `ARG` of the caller
	`push THIS`	// saves `THIS` of the caller
	`push THAT`	// saves `THAT` of the caller
	`ARG = SP-5-`*nArgs*	// repositions `ARG`
	`LCL = SP`	// repositions `LCL`
	`goto` *f*	// transfers control to the callee
	(*returnAddress*)	// injects the return address label into the code
`function` *f nVars* (declares a function *f*, informing that the function has *nVars* local variables)	(*f*)	// injects a function entry label into the code
	repeat *nVars* times:	// *nVars* = number of local variables
	`push 0`	// initializes the local variables to 0
`return` (terminates the current function and returns control to the caller)	*frame* `= LCL`	// *frame* is a temporary variable
	retAddr `= *(`*frame*`-5)`	// puts the return address in a temporary variable
	`*ARG = pop()`	// repositions the return value for the caller
	`SP = ARG+1`	// repositions `SP` for the caller
	`THAT = *(`*frame*`-1)`	// restores `THAT` for the caller
	`THIS = *(`*frame*`-2)`	// restores `THIS` for the caller
	`ARG = *(`*frame*`-3)`	// restores `ARG` for the caller
	`LCL = *(`*frame*`-4)`	// restores `LCL` for the caller
	`goto` *retAddr*	// go to the return address

Figure 8.5 Implementation of the function commands of the VM language. All the actions described on the right are realized by generated Hack assembly instructions.

8.5.2 Standard VM Mapping on the Hack Platform, Part II

Developers of the VM implementation on the Hack computer are advised to follow the conventions described here. These conventions complete the Standard VM Mapping on the Hack Platform, Part I, guidelines given in section 7.4.1.

The stack: On the Hack platform, RAM locations 0 to 15 are reserved for pointers and virtual registers, and RAM locations 16 to 255 are reserved for static variables. The stack is mapped on address 256 onward. To realize this mapping, the VM translator should start by generating assembly code that sets SP to 256. From this point onward, when the VM translator encounters commands like pop, push, add, and so on in the source VM code, it generates assembly code that affects these operations by manipulating the address that SP points at, and modifying SP, as needed. These actions were explained in chapter 7 and implemented in project 7.

Special symbols: When translating VM commands into Hack assembly, the VM translator deals with two types of symbols. First, it manages predefined assembly-level symbols like SP, LCL, and ARG. Second, it generates and uses symbolic labels for marking return addresses and function entry points. To illustrate, let us revisit the PointDemo program presented in the introduction to part II. This program consists of two Jack class files, Main.jack (figure II.1) and Point.jack (figure II.2), stored in a folder named PointDemo. When applied to the PointDemo folder, the Jack compiler produces two VM files, named Main.vm and Point.vm. The first file

contains a single VM function, Main.main, and the second file contains the VM functions Point.new, Point.getx, ..., Point.print.

When the VM translator is applied to this same folder, it produces a single assembly code file, named PointDemo.asm. At the assembly code level, the function abstractions no longer exist. Instead, for each function command, the VM translator generates an entry label in assembly; for each call command, the VM translator (i) generates an assembly goto instruction, (ii) creates a return address label and pushes it onto the stack, and (iii) injects the label into the generated code. For each return command, the VM translator pops the return address off the stack and generates a goto instruction. For example:

VM code	Generated assembly code
`function Main.main`	`(Main.main)`
...	...
`call Point.new`	`goto Point.new`
`// Next VM command`	`(Main.main$ret0)`
...	`// Next VM command (in assembly)`
	...
`function Point.new`	`(Point.new)`
...	...
`return`	`goto Main.main$ret0`

Figure 8.6 specifies all the symbols that the VM translator handles and generates.

Symbol	*Usage*
`SP`	This predefined symbol points to the memory address within the host RAM just following the address containing the topmost stack value.
`LCL`, `ARG`, `THIS`, `THAT`	These predefined symbols point, respectively, to the base RAM addresses of the virtual segments `local`, `argument`, `this`, and `that` of the currently running VM function.
Xxx`.`*i* symbols (represent static variables)	Each reference to `static` *i* appearing in file *Xxx*`.vm` is translated to the assembly symbol *Xxx*`.`*i*. In the subsequent assembly process, the Hack assembler will allocate these symbolic variables to the RAM, starting at address 16.
functionName`$`*label* (destinations of `goto` commands)	Let `foo` be a function within the file *Xxx*`.vm`. The handling of each `label bar` command within `foo` generates, and injects into the assembly code stream, the symbol *Xxx*`.foo$bar`. When translating `goto bar` and `if-goto bar` commands (within `foo`) into assembly, the label *Xxx*`.foo$bar` must be used instead of `bar`.
functionName (function entry point symbols)	The handling of each `function foo` command within the file *Xxx*`.vm` generates, and injects into the assembly code stream, a symbol *Xxx*`.foo` that labels the entry-point to the function's code. In the subsequent assembly process, the assembler translates this symbol into the physical address where the function code starts.
functionName`$ret.`*i* (return address symbols)	Let `foo` be a function within the file *Xxx*`.vm`. The handling of each `call` command within `foo`'s code generates, and injects into the assembly code stream, a symbol *Xxx*`.foo$ret.`*i*, where *i* is a running integer (one such symbol is generated for each `call` command within `foo`). This symbol is used to mark the return address within the caller's code. In the subsequent assembly process, the assembler translates this symbol into the physical memory address of the command immediately following the `call` command.
`R13` - `R15`	These predefined symbols can be used for any purpose. For example, if the VM translator generates assembly code that needs to use low-level variables for temporary storage, `R13` - `R15` can come handy.

Figure 8.6 The naming conventions described above are designed to support the translation of multiple .vm files and functions into a single .asm file, ensuring that the generated assembly symbols will be unique within the file.

Bootstrap code: The standard VM mapping on the Hack platform stipulates that the stack be mapped on the host RAM from address 256 onward, and that the first VM function that should start executing is the OS function Sys.init. How can we effect these conventions on the Hack platform? Recall that when we built the Hack computer in chapter 5, we wired it in such a way that upon reset, it will fetch and execute the instruction located in ROM address 0. Thus, if we want the computer to execute a predetermined code segment when it boots up, we can put this code in the Hack computer's instruction memory, starting at address 0. Here is the code:

```
// Bootstrap (pseudo) code, should be expressed in machine language
SP = 256
call Sys.init
```

The Sys.init function, which is part of the operating system, is then expected to call the main function of the application, and enter an infinite loop. This action will cause the translated VM program to start running. Note that the notions of *application* and *main function* vary from one high-level language to another. In the Jack language, the convention is that Sys.init should call the VM function Main.main. This is similar to the Java setting: when we instruct the JVM to execute a given Java class, say, Foo, it looks for, and executes, the Foo.main method. In general, we can effect language-specific startup routines by using different versions of the

Sys.init function.

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

prompt> VMTranslator *source*

where *source* is either a file name of the form *Xxx*.vm (the extension is mandatory) or the name of a folder (in which case there is no extension) containing one or more .vm files. The file/folder name may contain a file path. If no path is specified, the translator operates on the current folder. The output of the VM translator is a single assembly file, named *source*.asm. If *source* is a folder name, the single .asm file contains the translation of all the functions in all the .vm files in the folder, one after the other. 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.

8.5.3 Design Suggestions for the VM Implementation

In project 7 we proposed building the basic VM translator using three modules: VMTranslator, Parser, and CodeWriter. We now describe how to extend this basic implementation into a full-scale VM translator. This extension can be accomplished by adding the functionality described below to the three modules already built in project 7. There is no need to develop additional modules.

The VMTranslator

If the translator's input is a single file, say Prog.vm, the VMTranslator constructs a Parser for parsing Prog.vm and a CodeWriter that starts by creating an output file named Prog.asm. Next, the VMTranslator enters a loop that uses the Parser services for iterating through the input file and parsing each line as a VM command, barring white space. For each parsed command, the VMTranslator uses the CodeWriter for generating Hack assembly code and emitting the generated code into the output file. All this was already done in project 7.

If the translator's input is a folder, named, say, Prog, the VMTranslator constructs a Parser for handling each .vm file in the folder, and a single CodeWriter for generating Hack assembly code into the single output file Prog.asm. Each time the VMTranslator starts translating a new .vm file in the folder, it must inform the CodeWriter that a new file is now being processed. This is done by calling a CodeWriter routine named setFileName, as we now turn to describe.

The Parser

This module is identical to the Parser developed in project 7.

The CodeWriter

The CodeWriter developed in project 7 was designed to handle the VM *arithmetic-logical* and

push / *pop* commands. Here is the API of a complete CodeWriter that handles all the commands in the VM language:

Routine	*Arguments*	*Returns*	*Function*
Constructor / initializer	Output file / stream	—	Opens an output file / stream and gets ready to write into it. Writes the assembly instructions that effect the bootstrap code that starts the program's execution. This code must be placed at the beginning of the generated output file / stream. Comment: See the *Implementation Tips* at the end of section 8.6.
`setFileName`	`fileName` (string)	—	Informs that the translation of a new VM file has started (called by the `VMTranslator`).
`writeArithmetic` (developed in project 7)	`command` (string)	—	Writes to the output file the assembly code that implements the given arithmetic-logical `command`.
`writePushPop` (developed in project 7)	`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`.
`writeLabel`	`label` (string)	—	Writes assembly code that effects the `label` command.
`writeGoto`	`label` (string)	—	Writes assembly code that effects the `goto` command.
`writeIf`	`label` (string)	—	Writes assembly code that effects the `if-goto` command.
`writeFunction`	`functionName` (string) `nVars` (int)	—	Writes assembly code that effects the `function` command.
`writeCall`	`functionName` (string) `nArgs` (int)	—	Writes assembly code that effects the `call` command.
`writeReturn`	—	—	Writes assembly code that effects the `return` command.
`close` (developed in project 7)	—	—	Closes the output file / stream.

8.6 Project

In a nutshell, we have to extend the basic translator developed in chapter 7 with the ability to handle multiple .vm files and the ability to translate VM *branching* commands and VM *function* commands into Hack assembly code. For each parsed VM command, the VM translator has to generate assembly code that implements the command's semantics on the host Hack platform. The translation of the three *branching* commands into assembly is not difficult. The translation of the three *function* commands is more challenging and entails implementing the pseudocode listed in figure 8.5, using the symbols described in figure 8.6. We repeat the suggestion given in the previous chapter: Start by writing the required assembly code on paper. Draw RAM and global stack images, keep track of the stack pointer and the relevant memory segment pointers, and make sure that your paper-based assembly code successfully implements all the low-level actions associated with handling the call, function, and return commands.

Objective: Extend the basic VM translator built in project 7 into a full-scale VM translator, designed to handle multi-file programs written in 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 8.

Contract: Complete the construction of a VM-to-Hack translator, conforming to VM Specification, Part II (section 8.4) and to the Standard VM Mapping on the Hack Platform, Part II (section 8.5.2). Use your translator to translate the supplied VM test programs, yielding corresponding programs written in the Hack assembly language. When executed on the supplied CPU emulator along with the supplied test scripts, the assembly programs generated by your translator should deliver the results mandated by the supplied compare files.

Resources: You will need two tools: the programming language in which you will implement your VM translator and the *CPU emulator* supplied in the Nand to Tetris software suite. Use the CPU emulator to execute and test the assembly code generated by your translator. If the generated code runs correctly, we will assume that your VM translator performs as expected. This partial test of the translator will suffice for our purposes.

Another tool that comes in handy in this project is the supplied *VM emulator*. Use this program to execute the supplied test VM programs, and watch how the VM code effects the simulated states of the stack and the virtual memory segments. This can help understand the actions that the VM translator must eventually realize in assembly.

Since the full-scale VM translator is implemented by extending the VM translator built in project 7, you will also need the source code of the latter.

Testing and Implementation Stages

We recommend completing the implementation of the VM translator in two stages. First, implement the *branching* commands, and then the *function* commands. This will allow you to unit-test your implementation incrementally, using the supplied test programs.

Testing the Handling of the VM Commands label, if, if-goto:

- BasicLoop: Computes $1 + 2 + \cdots +$ `argument[0]`, and pushes the result onto the stack. Tests how the VM translator handles the label and if-goto commands.
- FibonacciSeries: Computes and stores in memory the first *n* elements of the Fibonacci series. A more rigorous test of handling the label, goto, and if-goto commands.

Testing the Handling of the VM Commands call, function, return:

Unlike what we did in project 7, we now expect the VM translator to handle multi-file programs. We remind the reader that by convention, the first function that starts running in a VM program is Sys.init. Normally, Sys.init is programmed to call the program's Main.main function. For the purpose of this project, though, we use the supplied Sys.init functions for setting the stage for the various tests that we wish to perform.

- SimpleFunction: Performs a simple calculation and returns the result. Tests how the VM translator handles the function and return commands. Since this test entails the handling of a single file consisting of a single function, no Sys.init test function is needed.
- FibonacciElement: This test program consists of two files: Main.vm contains a single Fibonacci function that returns recursively the *n*-th element of the Fibonacci series; Sys.vm contains a single Sys.init function that calls Main.fibonacci with $n = 4$ and then enters an infinite loop (recall that the VM translator generates bootstrap code that calls Sys.init). The resulting setting provides a rigorous test of the VM translator's handling of multiple .vm files, the VM function-call-and-return commands, the bootstrap code, and most of the other VM commands. Since the test program consists of two .vm files, the entire folder must be translated, producing a single FibonacciElement.asm file.
- StaticsTest: This test program consists of three files: Class1.vm and Class2.vm contain functions that set and get the values of several static variables; Sys.vm contains a single Sys.init function that calls these functions. Since the program consists of several .vm files, the entire folder must be translated, producing a single StaticsTest.asm file.

Implementation Tips

Since project 8 is based on extending the basic VM translator developed in project 7, we advise making a backup copy of the source code of the latter (if you haven't done it already).

Start by figuring out the assembly code that is necessary to realize the logic of the VM commands label, goto, and if-goto. Next, proceed to implement the methods writeLabel, writeGoto, and writeIf of the CodeWriter. Test your evolving VM translator by having it translate the supplied BasicLoop.vm and FibonacciSeries.vm programs.

Bootstrapping code: In order for any translated VM program to start running, it must

include startup code that forces the VM implementation to start executing the program on the host platform. Further, for any VM code to operate properly, the VM implementation must store the base addresses of the stack and the virtual segments in the correct locations in the host RAM. The first three test programs in this project (BasicLoop, FibonacciSeries, SimpleFunction) assume that the startup code was not yet implemented, and include test scripts that effect the necessary initializations *manually*, meaning that at this development stage you don't have to worry about it. The last two test programs (FibonacciElement and StaticsTest) assume that the startup code is already part of the VM implementation.

With that in mind, the constructor of the CodeWriter must be developed in two stages. The first version of your constructor must not generate any bootstrapping code (that is, ignore the constructor's API guideline beginning with the text "Writes the assembly instructions …"). Use this version of your translator for unit-testing the programs BasicLoop, FibonacciSeries, and SimpleFunction. The second and final version of your CodeWriter constructor must write the bootstrapping code, as specified in the constructor's API. This version should be used for unit-testing FibonacciElement and StaticsTest.

The supplied test programs were carefully planned to test the specific features of each stage in your VM implementation. We recommend implementing your translator in the proposed order, and testing it using the appropriate test programs at each stage. Implementing a later stage before an early one may cause the test programs to fail.

A web-based version of project 8 is available at www.nand2tetris.org.

8.7 Perspective

The notions of *branching* and *function calling* are fundamental to all high-level languages. This means that somewhere down the translation path from a high-level language to binary code, someone must take care of handling the intricate housekeeping chores related to their implementation. In Java, C#, Python, and Jack, this burden falls on the virtual machine level. And if the VM architecture is *stack-based*, it lends itself nicely to the job, as we have seen throughout this chapter.

To appreciate the expressive power of our stack-based VM model, take a second look at the programs presented in this chapter. For example, figures 8.1 and 8.4 present high-level programs and their VM translations. If you do some line counting, you will note that each line of high-level code generates an average of about four lines of compiled VM code. As it turns out, this 1:4 translation ratio is quite consistent when compiling Jack programs into VM code. Even without knowing much about the art of compilation, one can appreciate the brevity and readability of the VM code generated by the compiler. For example, as we will see when we build the compiler, a high-level statement like `let y = Math.sqrt(x)` is translated into `push x`, `call Math.sqrt`, `pop y`. The two-tier compiler can get away with so little work since it counts on the VM implementation for handling the rest of the translation. If we had to

translate high-level statements like `let y = Math.sqrt(x)` directly into Hack code, without having the benefit of an intermediate VM layer, the resulting code would be far less elegant, and more cryptic.

That said, it would also be more efficient. Let us not forget that the VM code must be realized in machine language—that's what projects 7 and 8 are all about. Typically, the final machine code resulting from a two-tier translation process is longer and less efficient than that generated from direct translation. So, which is more desirable: a two-tier Java program that eventually generates one thousand machine instructions or an equivalent one-tier C++ program that generates seven hundred instructions? The pragmatic answer is that each programming language has its pros and cons, and each application has different operational requirements.

One of the great virtues of the two-tier model is that the intermediate VM code (e.g., Java's bytecode) can be *managed*, for example, by programs that test whether it contains malicious code, programs that monitor the code for business process modeling, and so on. In general, for most applications, the benefits of managed code justify the performance degradation caused by the VM level. Yet for high-performance programs like operating systems and embedded applications, the need to generate tight and efficient code typically mandates using C/C++, compiled directly to machine language.

For compiler writers, an obvious advantage of using an explicit interim VM language is that it simplifies the tasks of writing and maintaining compilers. For example, the VM implementation developed in this chapter frees the compiler from the significant tasks of handling the low-level implementation of the function call-and-return protocol. In general, the intermediate VM layer decouples the daunting challenge of building a high-level-to-low-level compiler into two far simpler challenges: building a high-level-to-VM compiler and building a VM-to-low-level translator. Since the latter translator, also called the compiler's *back end*, was already developed in projects 7 and 8, we can be content that about half of the overall compilation challenge has already been accomplished. The other half—developing the compiler's *front end*—will be taken up in chapters 10 and 11.

We end this chapter with a general observation about the virtue of separating abstraction from implementation—an ongoing theme in Nand to Tetris and a crucial systems-building principle that goes far beyond the context of program compilation. Recall that VM functions can access their memory segments using commands like `push argument 2`, `pop local 1`, and so on while having no idea *how* these values are represented, saved, and reinstated during run-time. The VM implementation takes care of all the gory details. This complete separation of abstraction and implementation implies that developers of compilers that generate VM code don't have to worry about how the code they generate will end up running; they have enough problems of their own, as you will soon realize.

So cheer up! You are halfway through writing a two-tier compiler for a high-level, object-based, Java-like programming language. The next chapter is devoted to describing this language. This will set the stage for chapters 10 and 11, in which we'll complete the compiler's development. We begin seeing Tetris bricks falling at the end of the tunnel.