8 Virtual Machine II: Program 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 ended with the construction of a basic VM implementation over the Hack platform. In this chapter we continue to develop the VM abstraction, language, and implementation. In particular, we design stack-based mechanisms for handling nested subroutine calls (procedures, functions, methods) of procedural or object-oriented languages. As the chapter progresses, we extend the previously built basic VM implementation, ending with a full-scale VM translator. This translator will serve as the backend of the compiler that we will build in chapters 10 and 11, following the introduction of a high-level object-based language in chapter 9.

In any Great Gems in Computer Science contest, *stack processing* will be a strong finalist. The previous chapter showed how arithmetic and Boolean expressions can be calculated by elementary stack operations. This chapter goes on to show how this remarkably simple data structure can also support remarkably complex tasks like nested subroutine calling, parameter passing, recursion, and the associated memory allocation techniques. Most programmers tend to take these capabilities for granted, expecting the compiler to deliver them, one way or another. We are now in a position to open this black box and see how these fundamental programming mechanisms are actually implemented by a stack-based virtual machine.

8.1 Background

High-level languages allow writing programs in high-level terms. For example, $x = -b + \sqrt{b^2 - 4 \cdot a \cdot c}$ can be expressed as `x=-b+sqrt(power(b,2)-4*a*c)`,

which is almost as descriptive as the real thing. High-level languages support this power of expression through three conventions. First, one is allowed to freely define high-level operations like `sqrt` and `power`, as needed. Second, one is allowed to freely use (call) these subroutines as if they were elementary operations like `+` and `*`. Third, one is allowed to assume that each called subroutine will get executed—somehow—and that following its termination control will return—somehow—to the next command in one's code. Flow of control commands take this freedom one step further, allowing writing, say, `if ~(a=0) {x=(-b+sqrt(power(b,2)-4*a*c))/(2*a)} else {x=-c/b}`.

The ability to compose such expressions freely permits us to write abstract code, closer to the world of algorithmic thought than to that of machine execution. Of course the more abstract the high level, the more work we have to do at the low level. In particular, the low level must manage the delicate interplay between the calling subroutine (the *caller*) and the *called* subroutines—the program units that implement system- and user-defined operations like `sqrt` and `power`. For each subroutine call during runtime, the low level must handle the following details behind the scene:

- Passing parameters from the caller to the called subroutine
- Saving the state of the caller before switching to execute the called subroutine
- Allocating space for the local variables of the called subroutine
- Jumping to execute the called subroutine
- Returning values from the called subroutine back to the caller
- Recycling the memory space occupied by the called subroutine, when it returns
- Reinstating the state of the caller
- Jumping to execute the code of the caller immediately following the spot where we left it

Taking care of these housekeeping chores is a major headache, and high-level programmers are fortunate that the compiler relieves them from this duty. So how does the compiler do it? Well, if we choose to base our low level implementation on a *stack machine*, the job will be surprisingly manageable. In fact, the stack structure lends itself perfectly well to supporting all the housekeeping tasks mentioned above.

With that in mind, the remainder of this section describes how *program flow* and *subroutine calling* commands can be implemented on a stack machine. We begin with the implementation of program flow commands, which is rather simple and requires no memory management, and continue to describe the more challenging implementation of subroutine calling commands.

8.1.1 Program Flow

The default execution of computer programs is linear, one command after the other. This sequential flow is occasionally broken by branching commands, for example, embarking on a new iteration in a loop. In low-level programming, the branching logic is accomplished by instructing the machine to continue execution at some destination in the program other than the next instruction, using a *goto destination* command. The destination specification can take several forms, the most primitive being the physical address of the instruction that should be executed next. A slightly more abstract redirection command is established by describing the jump destination using a symbolic *label*. This variation requires that the language be equipped with some *labeling* directive, designed to assign symbols to selected points in the code.

This basic *goto* mechanism can easily be altered to effect conditional branching as well. For example, an *if-goto destination* command can instruct the machine to take the jump only if a given Boolean condition is true; if the condition is false, the regular program flow should continue, executing the next command in the code. How should we introduce the Boolean condition into the language? In a stack machine paradigm, the most natural approach is conditioning the jump on the value of the stack's topmost element: if it's not zero, jump to the specified destination; otherwise, execute the next command in the program.

In chapter 7 we saw how primitive VM operations can be used to compute any Boolean expression, leaving its truth-value at the stack's topmost element. This power of expression, combined with the *goto* and *if-goto* commands just described, can be used to express any flow of control structure found in any programming language. Two typical examples appear in figure 8.1.

The low-level implementation of the VM commands *label*, *goto label*, and *if-goto label* is straightforward. All programming languages, including the "lowest" ones, feature branching commands of some sort. For example, if our low-level implementation is based on translating the VM commands into assembly code, all we have to do is reexpress these *goto* commands using the branching logic of the assembly language.

8.1.2 Subroutine Calling

Each programming language is characterized by a fixed set of built-in commands. The key abstraction mechanism provided by modern languages is the freedom to extend this basic repertoire with high-level, programmer-defined operations. In procedural languages, the high-level operations are called *subroutines*,

Flow of control structure	*Pseudo VM code*

`if (cond)`
`  s1`
`else`
`  s2`
`...`
	`  VM code for computing ~(cond)`
	`  if-goto L1`
	`  VM code for executing s1`
	`  goto L2`
	`label L1`
	`  VM code for executing s2`
	`label L2`
	`  ...`

`while (cond)`
`  s1`
`...`
	`label L1`
	`  VM code for computing ~(cond)`
	`  if-goto L2`
	`  VM code for executing s1`
	`  goto L1`
	`label L2`
	`  ...`

Figure 8.1 Low-level flow of control using *goto* commands.

procedures, or *functions*, and in object-oriented languages they are usually called *methods*. Throughout this chapter, all these high-level program units are referred to as *subroutines*.

In well-designed programming languages, the use of a high-level operation (implemented by a subroutine) has the same "look and feel" as that of built-in commands. For example, consider the functions *add* and *raise to a power*. Most languages feature the former as a built-in operation, while the latter may be written as a subroutine. In spite of these different implementations, both functions should ideally look alike from the caller's perspective. This would allow the caller to weave the two operations together naturally, yielding consistent and readable code. A stack language implementation of this principle is illustrated in figure 8.2.

We see that the only difference between invoking a built-in command and calling a user-defined subroutine is the keyword `call` preceding the latter. Everything else is exactly the same: Both operations require the caller to set up their arguments, both operations are expected to remove their arguments from the stack, and both operations are expected to return a value which becomes the topmost stack element. The uniformity of this protocol has a subtle elegance that, we hope, is not lost on the reader.

```
// x+2
push x
push 2
add
...
```

```
// x^3
push x
push 3
call power
...
```

```
// (x^3+2)^y
push x
push 3
call power
push 2
add
push y
call power
...
```

```
// Power function
// result = first arg
// raised to the power
// of the second arg.
function power
// code omitted
push result
return
```

Figure 8.2 Subroutine calling. Elementary commands (like add) and high-level operations (like power) have the same look and feel in terms of argument handling and return values.

Subroutines like `power` usually use local variables for temporary storage. These local variables must be represented in memory during the subroutine's lifetime, namely, from the point the subroutine starts executing until a `return` command is encountered. At this point, the memory space occupied by the subroutine's local variables can be freed. This scheme is complicated by allowing subroutines to be arbitrarily nested: One subroutine may call another subroutine, which may then call another one, and so on. Further, subroutines should be allowed to call themselves recursively; each recursive call must be executed independently of all the other calls and maintain its own set of local and argument variables. How can we implement this nesting mechanism and the memory management tasks implied by it?

The property that makes this housekeeping task tractable is the hierarchical nature of the call-and-return logic. Although the subroutine calling chain may be arbitrarily deep as well as recursive, at any given point in time only one subroutine executes at the top of the chain, while all the other subroutines down the calling chain are waiting for it to terminate. This *Last-In-First-Out* (LIFO) processing model lends itself perfectly well to a *stack* data structure, which is also LIFO. When subroutine `xxx` calls subroutine `yyy`, we can push (save) `xxx`'s world on the stack and branch to execute `yyy`. When `yyy` returns, we can pop (reinstate) `xxx`'s world off the stack, and continue executing `xxx` as if nothing happened. This execution model is illustrated in figure 8.3.

We use the term *frame* to refer, conceptually, to the subroutine's local variables, the arguments on which it operates, its working stack, and the other memory segments that support its operation. In chapter 7, the term *stack* referred to the working memory that supports operations like *pop*, *push*, *add*, and so on. From now on, when we say *stack* we mean *global stack*—the memory area containing the frames of the

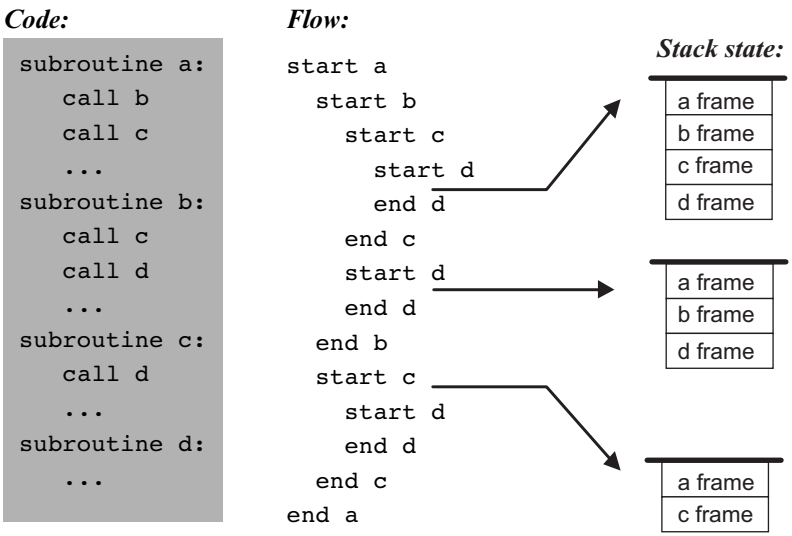

Figure 8.3 Subroutine calls and stack states associated with three representative points in the program's life cycle. All the layers in the stack are waiting for the current layer to complete its execution, at which point the stack becomes shorter and execution resumes at the level just below the current layer. (Following convention, the stack is drawn as if it grows downward.)

current subroutine and all the subroutines waiting for it to return. These two stack notions are closely related, since the working stack of the current subroutine is located at the very tip of the global stack.

To recap, the low-level implementation of the `call xxx` operation entails saving the caller's frame on the stack, allocating stack space for the local variables of the called subroutine (`xxx`), then jumping to execute its code. This last "mega jump" is not hard to implement. Since the name of the target subroutine is specified in the `call` command, the implementation can resolve the symbolic name to a memory address, then jump to execute the code starting at that address. Returning from the called subroutine via a `return` command is trickier, since the command specifies no return address. Indeed, the caller's anonymity is inherent in the very notion of a subroutine call. For example, subroutines like `power(x,y)` or `sqrt(x)` are designed to serve *any* caller, implying that the return address cannot be part of their code. Instead, a `return` command should be interpreted as follows: Redirect the program's execution to the command following the `call` command that called the current subroutine, wherever this command may be. The memory location of this command is called *return address*.

A glance at figure 8.3 suggests a stack-based solution to implementing this return logic. When we encounter a `call xxx` operation, we know exactly what the return address should be: It's the address of the next command in the caller's code. Thus, we can push this return address on the stack and proceed to execute the code of the called subroutine. When we later encounter a `return` command, we can pop the saved return address and simply *goto* it. In other words, the return address can also be placed in the caller's frame.

8.2 VM Specification, Part II

This section extends the basic VM specification from chapter 7 with *program flow* and *function calling* commands, thereby completing the overall VM specification.

8.2.1 Program Flow Commands

The VM language features three program flow commands:

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

8.2.2 Function Calling Commands

Different high-level languages have different names for program units including *functions*, *procedures*, *methods*, and *subroutines*. In our overall compilation model (elaborated in chapters 10–11), each such high-level program unit is translated into a low-level program unit called *VM function*, or simply *function*.

A function has a symbolic name that is used globally to call it. The function name is an arbitrary string composed of any sequence of letters, digits, underscore (_), dot (.), and colon (:) that does not begin with a digit. (We expect that a method `bar` in class `Foo` in some high-level language will be translated by the compiler to a VM function named `Foo.bar`). The scope of the function name is global: All functions in all files are seen by each other and may call each other using the function name.

The VM language features three function-related commands:

- `function` *f n* Here starts the code of a function named *f* that has *n* local variables;
- `call` *f m* Call function *f*, stating that *m* arguments have already been pushed onto the stack by the caller;
- `return` Return to the calling function.

8.2.3 The Function Calling Protocol

The events of calling a function and returning from a function can be viewed from two different perspectives: that of the calling function and that of the called function.

The calling function view:

- Before calling the function, the caller must push as many arguments as necessary onto the stack;
- Next, the caller invokes the function using the `call` command;
- After the called function returns, the arguments that the caller has pushed before the call have disappeared from the stack, and a *return value* (that always exists) appears at the top of the stack;
- After the called function returns, the caller's memory segments `argument`, `local`, `static`, `this`, `that`, and `pointer` are the same as before the call, and the `temp` segment is undefined.

The called function view:

- When the called function starts executing, its `argument` segment has been initialized with actual argument values passed by the caller and its `local` variables segment has been allocated and initialized to zeros. The `static` segment that the called function sees has been set to the `static` segment of the VM file to which it belongs, and the working stack that it sees is empty. The segments `this`, `that`, `pointer`, and `temp` are undefined upon entry.
- Before returning, the called function must push a value onto the stack.

To repeat an observation made in the previous chapter, we see that when a VM function starts running (or resumes its previous execution), it assumes that it is surrounded by a private world, all of its own, consisting of its memory segments and stack, waiting to be manipulated by its commands. The agent responsible for building this virtual worldview for every VM function is the VM implementation, as we elaborate in section 8.3.

8.2.4 Initialization

A VM program is a collection of related VM functions, typically resulting from the compilation of some high-level program. When the VM implementation starts running (or is reset), the convention is that it always executes an argument-less VM function called `Sys.init`. Typically, this function then calls the main function in the user's program. Thus, compilers that generate VM code must ensure that each translated program will have one such `Sys.init` function.

8.3 Implementation

This section describes how to complete the VM implementation that we started building in chapter 7, leading to a full-scale virtual machine implementation. Section 8.3.1 describes the stack structure that must be maintained, along with its standard mapping over the Hack platform. Section 8.3.2 gives an example, and section 8.3.3 provides design suggestions and a proposed API for actually building the VM implementation.

Some of the implementation details are rather technical, and dwelling on them may distract attention from the overall VM operation. This big picture is restored in section 8.3.2, which illustrates the VM implementation in action. Therefore, one may want to consult 8.3.2 for motivation while reading 8.3.1.

8.3.1 Standard VM Mapping on the Hack Platform, Part II

The Global Stack The memory resources of the VM are implemented by maintaining a global stack. Each time a function is called, a new block is added to the global stack. The block consists of the *arguments* that were set for the called function, a set of *pointers* used to save the state of the calling function, the *local variables* of the called function (initialized to 0), and an empty *working stack* for the called function. Figure 8.4 shows this generic stack structure.

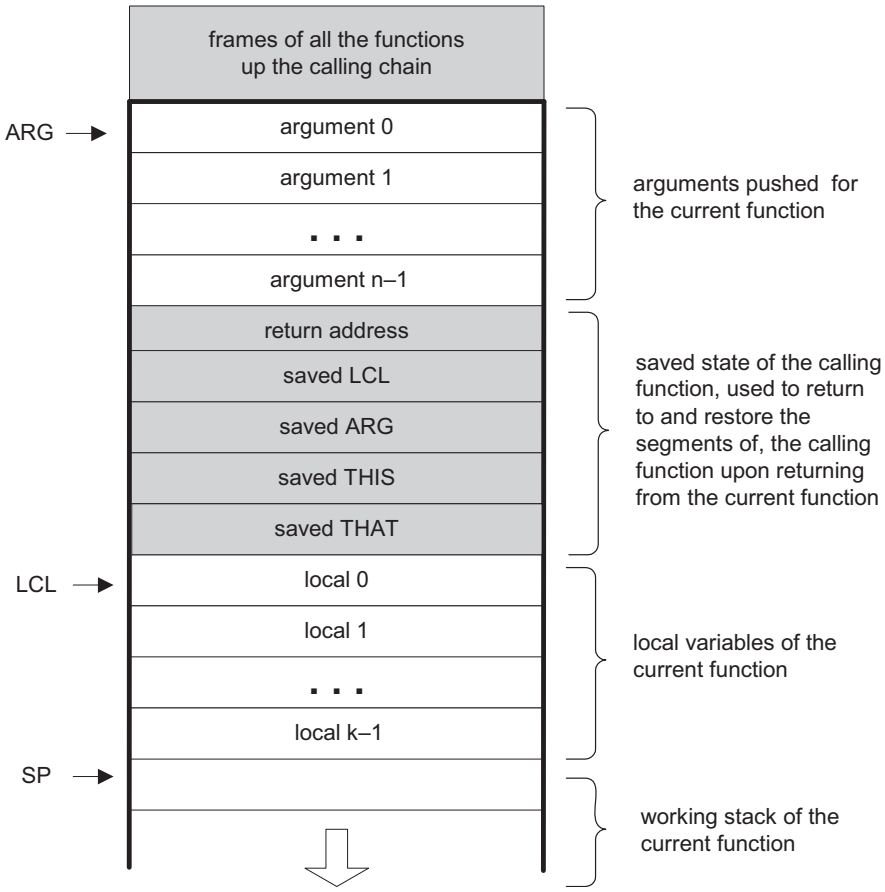

Figure 8.4 The global stack structure.

Note that the shaded areas in figure 8.4 as well as the `ARG`, `LCL`, and `SP` pointers are never seen by VM functions. Rather, they are used by the VM implementation to implement the function call-and-return protocol behind the scene.

How can we implement this model on the Hack platform? Recall that the standard mapping specifies that the stack should start at RAM address 256, meaning that the VM implementation can start by generating assembly code that sets `SP=256`. From this point onward, when the VM implementation encounters commands like `pop`, `push`, `add`, and so forth, it can emit assembly code that effects these operations by manipulating `SP` and relevant words in the host RAM. All this was already done in

chapter 7. Likewise, when the VM implementation encounters commands like `call`, `function`, and `return`, it can emit assembly code that maintains the stack structure shown in figure 8.4 on the host RAM. This code is described next.

Function Calling Protocol Implementation The function calling protocol and the global stack structure implied by it can be implemented on the Hack platform by effecting (in Hack assembly) the pseudo-code given in figure 8.5.

Recall that the VM implementation is a *translator* program, written in some high-level language. It accepts VM code as input and emits assembly code as output.

VM command	*Generated (pseudo)code emitted by the VM implementation*
`call f n` (calling a function `f` after `n` arguments have been pushed onto the stack)	`push return-address` // (Using the label declared below) `push LCL` // Save `LCL` of the calling function `push ARG` // Save `ARG` of the calling function `push THIS` // Save `THIS` of the calling function `push THAT` // Save `THAT` of the calling function `ARG = SP-n-5` // Reposition `ARG` (`n` = number of args.) `LCL = SP` // Reposition `LCL` `goto f` // Transfer control `(return-address)` // Declare a label for the return-address
`function f k` (declaring a function `f` that has `k` local variables)	`(f)` // Declare a label for the function entry `repeat k times:` // `k` = number of local variables `PUSH 0` // Initialize all of them to 0
`return` (from a function)	`FRAME = LCL` // `FRAME` is a temporary variable `RET = *(FRAME-5)` // Put the return-address in a temp. var. `*ARG = pop()` // Reposition the return value for the caller `SP = ARG+1` // Restore `SP` of the caller `THAT = *(FRAME-1)` // Restore `THAT` of the caller `THIS = *(FRAME-2)` // Restore `THIS` of the caller `ARG = *(FRAME-3)` // Restore `ARG` of the caller `LCL = *(FRAME-4)` // Restore `LCL` of the caller `goto RET` // Goto return-address (in the caller's code)

Figure 8.5 VM implementation of function commands. The parenthetical (return address) and (f) are label declarations, using Hack assembly syntax convention.

Hence, each pseudo-operation described in the right column of figure 8.5 is actually implemented by emitting assembly language instructions. Note that some of these "instructions" entail planting label declarations in the generated code stream.

Assembly Language Symbols As we have seen earlier, the implementation of *program flow* and *function calling* commands requires the VM implementation to create and use special symbols at the assembly level. These symbols are summarized in figure 8.6. For completeness of presentation, the first three rows of the table document the symbols described and implemented in chapter 7.

Symbol	***Usage***
`SP`, `LCL`, `ARG`, `THIS`, `THAT`	These predefined symbols point, respectively, to the stack top and to the base addresses of the virtual segments `local`, `argument`, `this`, and `that`.
`R13–R15`	These predefined symbols can be used for any purpose.
`Xxx.j`	Each static variable `j` in a VM file `Xxx.vm` is translated into the assembly symbol `Xxx.j`. In the subsequent assembly process, these symbolic variables will be allocated RAM space by the Hack assembler.
`functionName$label`	Each `label b` command in a VM function `f` should generate a globally unique symbol "`f$b`" where "`f`" is the function name and "`b`" is the label symbol within the VM function's code. When translating `goto b` and `if-goto b` VM commands into the target language, the full label specification "`f$b`" must be used instead of "`b`".
`(FunctionName)`	Each VM function f should generate a symbol "`f`" that refers to its entry point in the instruction memory of the target computer.
return-address	Each VM function call should generate and insert into the translated code stream a unique symbol that serves as a return address, namely the memory location (in the target platform's memory) of the command following the function call.

Figure 8.6 All the special assembly symbols prescribed by the VM-on-Hack standard mapping.

Bootstrap Code When applied to a VM program (a collection of one or more `.vm` files), the VM-to-Hack translator produces a single `.asm` file, written in the Hack assembly language. This file must conform to certain conventions. Specifically, the standard mapping specifies that (i) the VM stack should be mapped on location RAM[256] onward, and (ii) the first VM function that starts executing should be `Sys.init` (see section 8.2.4).

How can we effect this initialization in the `.asm` file produced by the VM translator? Well, when we built the Hack computer hardware in chapter 5, we wired it in such a way that upon reset, it will fetch and execute the word located in ROM[0]. Thus, the code segment that starts at ROM address 0, called *bootstrap code*, is the first thing that gets executed when the computer "boots up." Therefore, in view of the previous paragraph, the computer's bootstrap code should effect the following operations (in machine language):

```
SP=256          // Initialize the stack pointer to 0x0100
call Sys.init   // Start executing (the translated code of) Sys.init
```

`Sys.init` is then expected to call the main function of the main program and then enter an infinite loop. This action should cause the translated VM program to start running.

The notions of "program," "main program," and "main function" are compilation-specific and vary from one high-level language to another. For example, in the Jack language, the default is that the first program unit that starts running automatically is the `main` method of a class named `Main`. In a similar fashion, when we tell the JVM to execute a given Java class, say `Foo`, it looks for, and executes, the `Foo.main` method. Each language compiler can effect such "automatic" startup routines by programming `Sys.init` appropriately.

8.3.2 Example

The factorial of a positive number n can be computed by the iterative formula $n! = 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n$. This algorithm is implemented in figure 8.7.

Let us focus on the `call mult` command highlighted in the `fact` function code from figure 8.7. Figure 8.8 shows three stack states related to this call, illustrating the function calling protocol in action.

If we ignore the middle stack instance in figure 8.8, we observe that `fact` has set up some arguments and called `mult` to operate on them (left stack instance). When `mult` returns (right stack instance), the arguments of the called function have been replaced with the function's return value. In other words, when the dust clears from

```
function p
...
// Compute 4!
push constant 4
call fact 1 // 1 arg
...
```

```
function fact 2 // 2 local variables
// Returns the factorial of a given argument
   push constant 1
   pop local 0    // result=1
   push constant 1
   pop local 1    // j=1
label loop
   push constant 1
   push local 1
   add
   pop local 1    // j=j+1
   push local 1
   push argument 0
   gt
   if-goto end    // if j>n goto end
   push local 0
   push local 1
   call mult 2    // 2 arguments were pushed
   pop local 0    // result=mult(result,j)
   goto loop
label end
   push local 0
   return
```

```
function mult 2
// (2 local variables)
// Multiplies argument 0
// times argument 1.
// Code appears in
// figure 7.9.
...
// Return the result:
push local 0
return
```

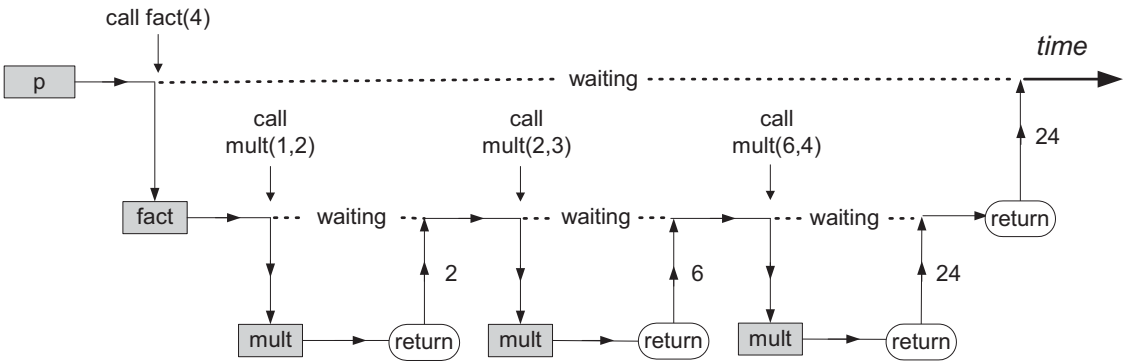

Figure 8.7 The life cycle of function calls. An arbitrary function `p` calls function `fact`, which then calls `mult` several times. Vertical arrows depict transfer of control from one function to another. At any given point in time, only one function is running, while all the functions up the calling chain are waiting for it to return. When a function returns, the function that called it resumes its execution.

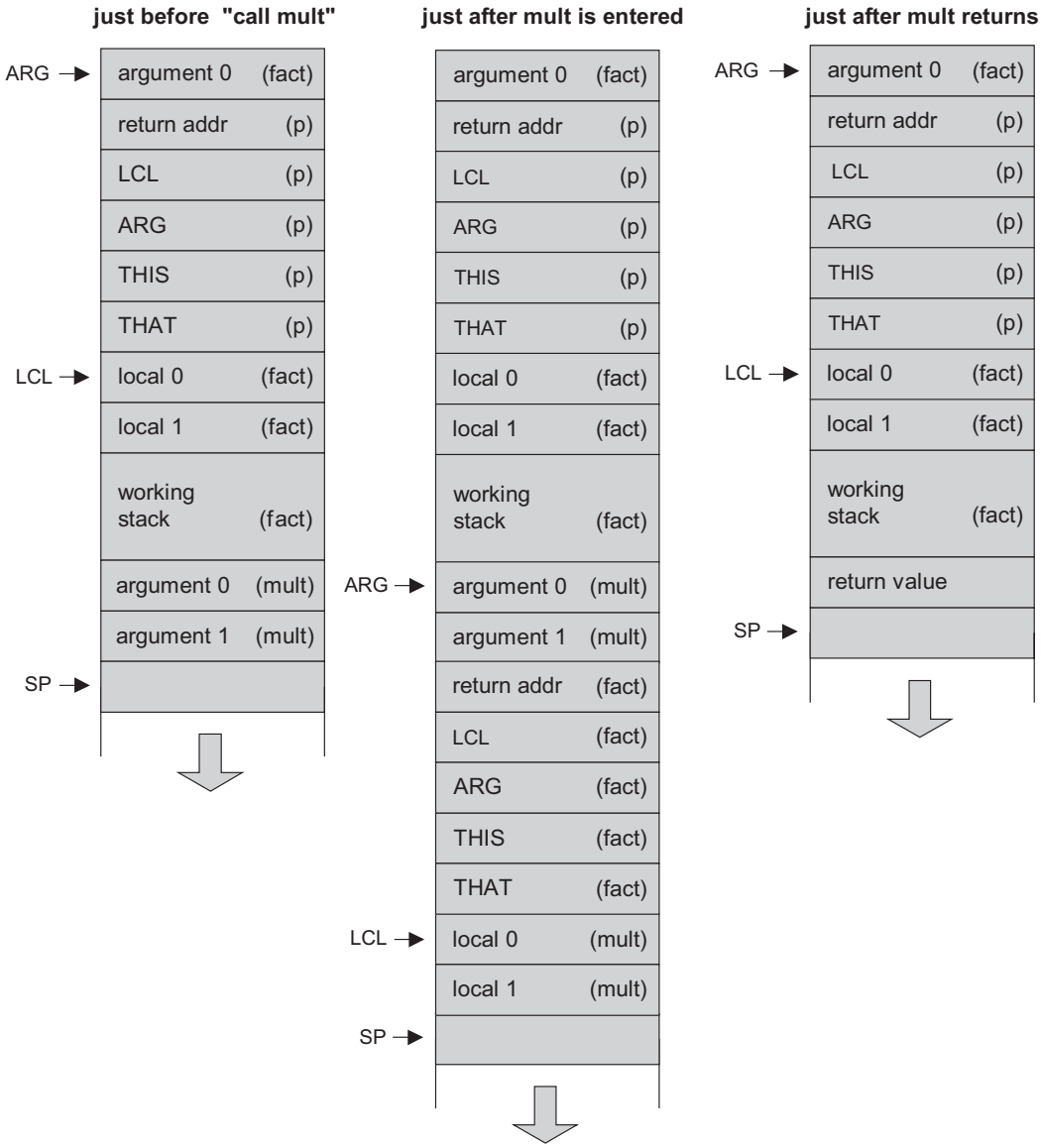

Figure 8.8 Global stack dynamics corresponding to figure 8.7, focusing on the `call mult` event. The pointers SP, ARG, and LCL are not part of the VM abstraction and are used by the VM implementation to map the stack on the host RAM.

the function call, the calling function has received the service that it has requested, and processing resumes as if nothing happened: The drama of `mult`'s processing (middle stack instance) has left no trace whatsoever on the stack, except for the return value.

8.3.3 Design Suggestions for the VM Implementation

The basic VM translator built in Project 7 was based on two modules: *parser* and *code writer*. This translator can be extended into a full-scale VM implementation by extending these modules with the functionality described here.

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

The *CodeWriter* Module The basic `CodeWriter` specified in chapter 7 should be augmented with the following methods.

CodeWriter: Translates VM commands into Hack assembly code. The routines listed here should be added to the `CodeWriter` module API given in chapter 7.

Routine	Arguments	Returns	Function
`writeInit`	—	—	Writes assembly code that effects the VM initialization, also called *bootstrap code*. This code must be placed at the beginning of the output file.
`writeLabel`	`label` (string)	—	Writes assembly code that effects the `label` command.
`writeGoto`	`label` (string)	—	Writes assembly code that effects the `goto` command.

Routine	Arguments	Returns	Function
`writeIf`	`label` (string)	—	Writes assembly code that effects the `if-goto` command.
`writeCall`	`functionName` (string) `numArgs` (int)	—	Writes assembly code that effects the `call` command.
`writeReturn`	—	—	Writes assembly code that effects the `return` command.
`writeFunction`	`functionName` (string) `numLocals` (int)	—	Writes assembly code that effects the `function` command.

8.4 Perspective

The notions of subroutine calling and program flow are fundamental to all high-level languages. This means that somewhere down the translation path to binary code, someone must take care of the intricate housekeeping chores related to their implementation. In Java, C#, and Jack, this burden falls on the VM level. And if the VM is *stack-based*, it lends itself nicely to the job, as we have seen throughout this chapter. In general then, virtual machines that implement subroutine calls and recursion as a primitive feature deliver a significant and useful abstraction.

Of course this is just one implementation option. Some compilers handle the details of subroutine calling directly, without using a VM at all. Other compilers use various forms of VMs, but not necessarily for managing subroutine calling. Finally, in some architectures most of the subroutine calling functionality is handled directly by the hardware.

In the next two chapters we will develop a Jack-to-VM compiler. Since the back-end of this compiler was already developed—it is the VM implementation built in chapters 7–8—the compiler's development will be a relatively easy task.

8.5 Project

Objective Extend the basic VM translator built in Project 7 into a full-scale VM translator. In particular, add the ability to handle the program flow and function calling commands of the VM language.

Resources (same as Project 7) You will need two tools: the programming language in which you will implement your VM translator, and the CPU emulator supplied with the book. This emulator 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 full-scale VM-to-Hack translator, extending the translator developed in Project 7, and conforming to the VM Specification, Part II (section 8.2) and to the Standard VM Mapping on the Hack Platform (section 8.3.1). Use it to translate the VM programs supplied below, yielding corresponding programs written in the Hack assembly language. When executed on the supplied CPU emulator, these assembly programs should deliver the results mandated by the supplied test scripts and compare files.

Testing Programs

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

For each program `Xxx`, 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.

Test Programs for Program Flow Commands

- `BasicLoop:` computes $1 + 2 + \cdots + n$ and pushes the result onto the stack. This program tests the implementation of the VM language's `goto` and `if-goto` commands.
- `Fibonacci:` computes and stores in memory the first n elements of the Fibonacci series. This typical array manipulation program provides a more challenging test of the VM's branching commands.

Test Programs for Function Calling Commands

- `SimpleFunction:` performs a simple calculation and returns the result. This program provides a basic test of the implementation of the `function` and `return` commands.
- `FibonacciElement:` This program provides a full test of the implementation of the VM's function calling commands, the bootstrap section, and most of the other VM commands.

The program directory consists of two `.vm` files:

- `Main.vm` contains one function called `fibonacci`. This recursive function returns the n-th element of the Fibonacci series;
- `Sys.vm` contains one function called `init`. This function calls the `Main.fibonacci` function with $n = 4$, then loops infinitely.

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

- `StaticsTest:` A full test of the VM implementation's handling of static variables. Consists of two `.vm` files, each representing the compilation of a stand-alone class file, plus a `Sys.vm` file. The entire directory should be compiled in order to produce a single `StaticsTest.asm` file.

(Recall that according to the VM Specification, the bootstrap code generated by the VM implementation must include a call to the `Sys.init` function).

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 store the base addresses of the virtual segments in selected locations in the host RAM. The first three test programs in this project assume that the startup code was not yet implemented and include test scripts that effect the necessary initializations "manually." The last two programs assume that the startup code is already part of the VM implementation.

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

1. Run the 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(s), yielding a single `.asm` text file that contains a translated 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.

Note: The supplied test programs were carefully planned to unit-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.

Tools Same as in Project 7.