Chapter 7: The Virtual Machine

“*The programmer is a creator of universes for which he alone is responsible. Universes of virtually unlimited complexity can be created in the form of computer programs.*”


7. The Virtual Machine

7.1 The Approach

In the last part of the book we’ll introduce a Java-like high-level language, called Jack. The translation of Jack programs into Hack’s machine language will be a two-stage process. First, we will compile the Jack program into an intermediate language, called Vack, generating a Vack program. Next, we will translate the Vack code into Hack’s machine language. At this point the program will be expressed in native Hack code, and thus it can be loaded into, and run, on the Hack hardware. The basic process is depicted on the left side of Figure 7-1.

Vack programs are designed to run in an abstract environment called a Virtual Machine, or VM for brevity. That is to say, Vack programs access and manipulate imaginary devices that don’t exist for real. Thus, if we want to run Vack programs on a certain hardware platform, we have to materialize, or implement, the Vack VM on the target hardware. For example, in this book we wish to run Vack programs on the Hack hardware. To do so, we have to write a special program that translates each Vack command into a series of one or more Hack commands. The resulting code will be native Hack, and thus it can be executed on the Hack computer. Principally speaking, the translator program can be written in any programming language, including languages that don’t run on the target machine. The only requirement is that the input of the translator will be a Vack program, and that its output will be native Hack code that affects the program’s behavior on the Hack platform.

Why should we go through all this trouble? Why not compile the program directly from Jack to Hack? There are at least two reasons why the added complexity of a virtual machine makes sense. First, the process of compiling a high level language like Jack directly into machine language is rather complex. As it turns out, it is much easier to translate the program in two stages, using an intermediate language like Vack. Second, Vack code is portable. If all the manufacturers of digital devices in the world would equip their target machines with Vack VM implementations, the same Jack program could run on any digital device, without requiring changes in the Jack source code.

This multi-platform portability is a well-known virtue of the Java language. In the Java model, the intermediate code is called bytecode rather than Vack code, and the bytecode is executed on the JVM (Java Virtual Machine) implementation of the target machine. Since JVM implementations exist on numerous hardware platforms ranging from personal computers to cellular telephones, the same Java program can run on all these devices (barring some user-interface modifications). For that reason, software vendors don’t have to create and maintain multiple versions of the same Java program for different hardware platforms. They write the Java code once, and the JVM’s out there run it everywhere.
The chapter begins with an overview of a special data structure called stack. The stack is the backbone of the VM model, providing a variety of processing, storage and housekeeping services. We then introduce the Vack language, which consists of various stack-manipulation commands. We end the chapter with a discussion on how to implement the VM model on various hardware platforms, with special emphasis on the Hack computer. Importantly, Hack-specific issues are discussed in the last section only. The remainder of the chapter deals with general virtual machine and VM implementation issues.
7.2 The Stack

The stack is a fundamental data structure that comes to play in many computer science applications. In the VM architecture, the stack is the logical area in which Vack processing takes place. Some Vack commands remove data items from the top of the stack, perform various operations on them, and put the result back onto the stack. Other Vack commands transfer data items from the stack’s top to designated memory locations, and vice versa. The basic stack anatomy is illustrated in Figure 7-2, focusing on the two elementary stack operations push and pop.

![Stack Anatomy Diagram]

The stack is characterized by the label SP (for stack pointer), which always points to the next available slot on the stack. If we think of the stack as an array, then push x executes the two operations stack[sp]=x and sp=next(sp), whereas pop x executes the two operations sp=previous(sp) and x=stack[sp]. Importantly, these are the only two index-based operations allowed on the stack, implying that the only accessible data item in the stack is the one on the top. We use next and previous instead of + and - to emphasize that as far as the VM is concerned, the location of high memory with respect to sp is an irrelevant implementation detail.
We note in passing that SP completely defines the stack for other applications. That is to say, if you are given access to SP, you know everything you need to know about the stack. Also, we see that stack access differs from memory access in several respects. First, the stack is accessible only “from the top”, one item at a time. Second, reading the stack is a lossy operation: the only way to read the top value is to remove it from the stack. In contrast, the act of reading a regular memory location has no impact on the memory’s state. Finally, writing an item onto the stack adds it to the stack’s top, without changing the rest of the stack. In contrast, writing an item into a regular memory location is a lossy operation, since it erases the location’s previous value.

### 7.3 Stack Arithmetic

The basic data type of the Vack VM is a 16-bit binary number. Vack programs use this data type to represent integers, Booleans, and pointers. The truth-values “true” and “false” are represented by the 16-bit constants $0xFFFF$ and $0x0000$, respectively.

Vack features eight stack-oriented arithmetic commands. Some of the commands pop two items off the stack, compute a binary function on them, and push the result back onto the stack. Other commands pop a single item off the stack, compute a unary function on it, and push the result back onto the stack. The details are given in Spec. 7-3.

<table>
<thead>
<tr>
<th>command</th>
<th>operation</th>
<th>impact on the stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>addition</td>
<td>pop x; pop y; push(x+y)</td>
</tr>
<tr>
<td>sub</td>
<td>subtraction</td>
<td>pop y; pop x; push(x-y)</td>
</tr>
<tr>
<td>and</td>
<td>bit-wise and</td>
<td>pop x; pop y; push(x and y)</td>
</tr>
<tr>
<td>or</td>
<td>bit-wise or</td>
<td>pop x; pop y; push(x or y)</td>
</tr>
<tr>
<td>neg</td>
<td>2’s complement</td>
<td>pop x; push(neg(x))</td>
</tr>
<tr>
<td>not</td>
<td>bit-wise not</td>
<td>pop x; push(not(x))</td>
</tr>
<tr>
<td>eq</td>
<td>equality test</td>
<td>pop x; pop y; if x=y push(0xFFFF) else push(0x0000)</td>
</tr>
<tr>
<td>gt</td>
<td>greater-than test</td>
<td>pop y; pop x; if x&gt;y push(0xFFFF) else push(0x0000)</td>
</tr>
<tr>
<td>lt</td>
<td>less-than test</td>
<td>pop y; pop x; if x&lt;y push(0xFFFF) else push(0x0000)</td>
</tr>
</tbody>
</table>

**SPEC 7-3: Arithmetic and logical commands.** Each command has the net impact of replacing its arguments with the command’s result. The truth-values “true” and “false” are represented by the 16-bit constants $0xFFFF$ and $0x0000$, respectively.
To illustrate the arithmetic and logical commands, let us apply every one of them to an illustrative stack, and inspect the stack’s state before and after the operation (Figures 7-4 and 7-5). Since the stack is a “last in, first out” data structure, the only place where action takes place is the top. Therefore, we will focus only on three top-most cells in the stack.

**FIGURE 7-4: Arithmetic commands examples.** To minimize clutter, we assume that the stack is made up of 4-bit, rather than 16-bit, binary numbers. Each operation is represented by an arrow that transforms the stack to a new state.

**FIGURE 7-5: Comparison commands examples.** In Vack, “true” and “false” are represented by the constants 0xFFFF and 0x0000, respectively.
### 7.4 Memory Access

A program is a collection of *methods*. At any given point of time, only one method is running, i.e. engaged to the processor. We call this method the "current method". The current method has access to seven segments of memory, as follows:

<table>
<thead>
<tr>
<th>Memory Segment</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>global variables</td>
<td>shared by all methods: every method can read and modify every global variable; glbl0, ..., glbl255</td>
</tr>
<tr>
<td>local variables</td>
<td>accessible only by the current method. Initialized to zero when the method starts running; lcl0, ..., lclk</td>
</tr>
<tr>
<td>arguments</td>
<td>arguments of the current method. Initialized by the method that called the current method; arg0, ..., argn</td>
</tr>
<tr>
<td>field variables</td>
<td>fields of the current method's object, stored in the heap. As a convention, both <em>this</em> and <em>arg0</em> point to the base of this memory segment; field0, ..., fieldm</td>
</tr>
<tr>
<td>array elements</td>
<td>elements of the currently specified array; the array base points into the heap.</td>
</tr>
<tr>
<td>stack</td>
<td>The current method's working memory. Accessible to the current method only through <code>pop</code> and <code>push</code> commands.</td>
</tr>
<tr>
<td>heap</td>
<td>Stores all the objects and arrays of all the methods. Accessible to the current method only through <code>pop</code> and <code>push</code> commands.</td>
</tr>
</tbody>
</table>

**FIGURE 7-6: The method's view of the world.** In addition to the *stack* and the *heap*, which are managed implicitly by the VM implementation, each method has direct access to five memory segments. Importantly, all the pointers mentioned in the figure (SP, ARG, THIS, LCL, GLBL, ARR, and HEAP) are never used directly by method's code. The pointers are managed by the VM implementation implicitly, i.e. behind the scene, as a side-effect of the method's commands. Spec. 7-7 lists all the commands that can effect the seven memory segments mentioned above.
The method's object: Our VM is designed to support object-based languages. Therefore, each method is associated with a data structure called the method's object, implemented in the heap as a sequential segment of fields. Following the Java convention, we refer to the method's object (and to its base address in the heap) as "this." As a rule, each method must have an object; if the object is of no use to the programmer, the this pointer is set to NULL, as we will see shortly.

The method's array: A method may or may not be associated with an array, implemented in the heap as a sequential segment of elements. The method gains access to a certain array by establishing a reference to its base address via a special command, as we'll see below. This operation causes the VM implementation to store the array's base address in the pointer ARR. If the current method establishes access to a certain array, the latter is called the current array.

### SPEC 7-7: Memory-access commands, designed to operate on the seven memory segments available to the method: stack, heap, globals, locals, arguments, object fields, and array elements
(see Figure 7-6). The actual execution of each one of these commands is carried out by the VM implementation, as we explain later in the chapter.
To illustrate some of these commands in action, the following code carries out the pseudo operations \( \text{lcl7}=\text{lcl7}+\text{arg2}-153 \) and then \( \text{field5}=\text{lcl7} \):

\[
\begin{align*}
1: & \quad \text{push-local 7} \\
2: & \quad \text{push-argument 2} \\
3: & \quad \text{add} \\
4: & \quad \text{push-constant 153} \\
5: & \quad \text{sub} \\
6: & \quad \text{pop-lcl 7} \\
7: & \quad \text{push-lcl 7} \\
8: & \quad \text{pop-field 5}
\end{align*}
\]

Commands 1-3 have the net impact of pushing \( \text{lcl7}+\text{arg2} \) onto the stack. If we call this value \( v \), commands 3-5 have the net impact of pushing \( v-153 \) onto the stack (recall that both \text{add} and \text{subtract} remove their two arguments from the stack). Command 7 is necessary since the previous \text{pop} removed the result from the stack. At the end of command 8 the stack is in the same state as it was just before command 1. In other words, commands 1-8 leave the stack unchanged.

### 7.5 Program Flow

A method is a series of commands. The VM implementation executes the commands sequentially, beginning with the first one. Branching can be effected by label/goto commands, as follows:

<table>
<thead>
<tr>
<th>label c</th>
<th>use ( c ) to label the current location in the code;</th>
</tr>
</thead>
<tbody>
<tr>
<td>goto c</td>
<td>jump to label ( c ) in the code and continue execution from that location onward;</td>
</tr>
<tr>
<td>if-goto c</td>
<td>pop the topmost value from the stack; if the value is non-zero, jump to label ( c ) in the code and continue execution from that location onward.</td>
</tr>
</tbody>
</table>

**SPEC 7-8: Program flow commands.** The label \( c \) is an arbitrary string composed of letters, numbers, and the special characters "_" and "\". The language is not case-sensitive.
The mult program: We now turn to a multiplication program that illustrates many of the programming structures discussed above (Figure 7-9).

```
method mult(a,n)
  j=n;
  sum=0;
  loop: if j==0 goto end;
  sum=sum+a;
  j=j-1;
  goto loop;
end: return sum;
}
```

<table>
<thead>
<tr>
<th>informal VM code</th>
<th>psuedo code</th>
<th>Vack code</th>
</tr>
</thead>
<tbody>
<tr>
<td>method mult(a,n)</td>
<td>method mult 2 // two locals</td>
<td></td>
</tr>
<tr>
<td>push n</td>
<td>push-argument 2 // j=n</td>
<td></td>
</tr>
<tr>
<td>pop j</td>
<td>pop-local 1</td>
<td></td>
</tr>
<tr>
<td>push 0</td>
<td>push-constant 0 // sum=0</td>
<td></td>
</tr>
<tr>
<td>pop sum</td>
<td>pop-local 0</td>
<td></td>
</tr>
<tr>
<td>label loop</td>
<td>label loop</td>
<td></td>
</tr>
<tr>
<td>push 0</td>
<td>push-constant 0 // if j=0 ...</td>
<td></td>
</tr>
<tr>
<td>push j</td>
<td>push-local 1</td>
<td></td>
</tr>
<tr>
<td>eq</td>
<td>eq // ... goto end</td>
<td></td>
</tr>
<tr>
<td>if-goto end</td>
<td>if-goto end</td>
<td></td>
</tr>
<tr>
<td>push sum</td>
<td>push-local 0 // sum=sum+a</td>
<td></td>
</tr>
<tr>
<td>push a</td>
<td>push-argument 1</td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>add</td>
<td></td>
</tr>
<tr>
<td>pop sum</td>
<td>pop-local 0</td>
<td></td>
</tr>
<tr>
<td>push j</td>
<td>push-local 1 // j-j-1</td>
<td></td>
</tr>
<tr>
<td>push 1</td>
<td>push-constant 1</td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>sub</td>
<td></td>
</tr>
<tr>
<td>pop j</td>
<td>pop-local 1</td>
<td></td>
</tr>
<tr>
<td>goto loop</td>
<td>goto loop</td>
<td></td>
</tr>
<tr>
<td>label end</td>
<td>label end</td>
<td></td>
</tr>
<tr>
<td>push sum</td>
<td>push-local 0 // return sum</td>
<td></td>
</tr>
<tr>
<td>return</td>
<td>return</td>
<td></td>
</tr>
</tbody>
</table>

FIGURE 7-9: The mult program, which uses repetitive addition to implement multiplication. The informal code (left) is an intermediate conceptual stage between the C code (top) and the final Vack code (right).
Note that \texttt{mult(x,y)} is inefficient whenever \(x<y\). To improve performance, one can check the relationship between the two arguments before the loop starts and reverse their order if necessary. Introducing this change into \texttt{mult}'s code (on paper) is a good way to check one's understanding of stack-oriented programming, and thus we leave it to the reader as an exercise.

### 7.6 Method Calling

A typical application consists of multiple methods. Some methods are created by the programmer for modularity's sake; other methods enter the picture when the application invokes external API's and operating system services. When all these methods are compiled together, the result is a Vack application – a collection of Vack methods that must interact with each other in order to effect the desired processing on the virtual machine.

Each application has at least one method, called \texttt{main}, which starts running "automatically" when the application is invoked. At some point, a chain of method calling ensues: the \texttt{main} method calls method \texttt{a}, method \texttt{a} calls method \texttt{b}, and so on and so forth.

To illustrate this calling chain, we introduce a new method, called \texttt{fact}, designed to compute the factorial of a given number \texttt{n} by repetitive multiplication. Specifically, the method initializes \texttt{result=1}, then computes \texttt{result=mult(result,j)} varying \texttt{j} from \texttt{n} down to \texttt{1}. In each step of the loop, the \texttt{fact} method calls the \texttt{mult} method introduced in the previous section. Whenever such a call occurs, \texttt{fact} enters a "wait" state which persists as long as \texttt{mult} is running. When \texttt{mult} returns, \texttt{fact} resumes its execution. This delicate dance is depicted in Figure 7-10.
Chapter 7: The Virtual Machine

```c
fact(n) 
{ 
    result=1;
    j=n;
    loop: if j==1 goto end;
    result=mult(result,j);
    j=j-1;
    goto loop;
    end: return result;
}
```

FIGURE 7-10: The wait-run-return method life cycle: method p (an arbitrary method that needs factorial services) calls method fact, which then calls mult several times. Vertical arrows depict transfer of control from one method to another. Full horizontal lines depict "current method" (i.e. "running") states, whereas broken horizontal lines depict "waiting" states. At any given point of time, one method is "current" (engaged to the processor) while all the methods up its calling chain are waiting for it to return. When a method returns, the method that called it resumes its execution (which typically does something useful with the value returned by the called method).

**Arguments passing:** The key to successful inter-method interaction is proper argument passing from the "caller" to the "callee". Barring some trivial examples, all methods are designed to operate on one or more given arguments. For example, mult(x,y) is designed to compute x times y, for whatever values x and y the method's caller supplies. The notation mult(x,y) can be viewed as an interface specification that has two implications. First, the interface mult(x,y) tells programmers that if they want to receive a multiplication service, their method must (i) push the two multiplicands onto the stack, (ii) issue a call mult command, and (iii) pop the result from the stack. Second, the interface mult(x,y) tells the programmer of mult that when this method starts running, it can find the two values on which it is supposed to operate in arg1 and arg2. In other words, the contract is such that the caller sets the stage for the callee, and the callee can "hit the ground running".
**Method-calling commands:** Vack features three method-calling commands:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>call methodName c</code></td>
<td>call method <code>methodName</code>; <code>c</code> arguments were already pushed onto the stack.</td>
</tr>
<tr>
<td><code>method methodName c</code></td>
<td>here starts the code of method <code>methodName</code>; the method has <code>c</code> local variables.</td>
</tr>
<tr>
<td><code>return</code></td>
<td>terminate the method's execution.</td>
</tr>
</tbody>
</table>

**SPEC 7-11: Method-calling commands.** The label `m` is an arbitrary string composed of letters, numbers, and the special character "_". The language is not case-sensitive. `c` is an integer.
The fact program: This program computes the factorial (n!) of a given number n, using repetitive multiplication (Figure 7-12).

```java
fact(n)
{
    result=1;
    j=n;
    loop: if j==1 goto end;
    result=mult(result,j);
    j=j-1;
    goto loop;
    end: return result;
}
```

0: method fact 2    // fact has 2 local variables
1: push-constant 1    // result = 1
2: pop-local 0
3: push-argument 1    // j = n
4: pop-local 1
5: label loop        // loop:
6: push-local 1      // if j=1 goto end
7: push-constant 1
8: eq
9: if-goto end
10: push-constant 0   // push a null THIS pointer
11: push-local 0      // push result
12: pop-local 1       // push j
13: call mult 3       // call mult(result,j)
14: pop-local 0       // result = return value
15: push-local 1      // j=j-1
16: push-constant 1
17: sub
18: pop-local 1
19: goto loop         // goto loop
20: label end         // end:
21: push-local 0      // return result
22: return

**FIGURE 7-12: C and Vack implementations of the fact method.** In line 17, the mult method is called. Lines 14-16 set up the arguments just before the call. When fact returns (end of line 17), fact pops the return value and stores it in a local variable. The code of mult was given in Figure 7-9.

The method calling protocol that enables the VM implementation to support the interaction between the calling method and the called method is discussed in Spec. 7-13.
### Calling Method

1. Before the call, the calling method pushes the arguments of the called method onto the stack. The first argument must be a reference to the object of the called method. If the called method makes no use of the object, the calling method pushes a null constant as the first argument.

2. The called method is invoked via the command `call methodName`.

3. After the called method returns:
   - all the arguments of the called method have been removed from the stack;
   - the value returned by the called method (this value always exists) appears at the top of the stack;
   - The `globals`, `locals`, `arguments`, and `fields` memory segments are the same as before the call; The `array elements` segment is undefined.

### Called Method

1. Upon getting called:
   - The working stack is empty;
   - All the local variables of the called method have been initialized to zero;
   - The `arguments`, `locals` and `fields` memory segments are those of the called method. The `array elements` segment is undefined.

2. Before returning, the called method must push a return value onto the stack.

### Behind the Scene

1. When a `call methodName c` command is encountered:
   - The return address of the calling method is saved;
   - The `arguments`, `locals` and `fields` memory segments of the calling method are saved;
   - The `arguments`, `locals` and `fields` memory segments are set to those of the called method;
   - Local variables are allocated in memory for the called method and initialized to 0;
   - Control is transferred to the called method.

2. When a return command is encountered:
   - The `arguments`, `local variables`, and other junk generated by the local method are cleared from the stack. The `arguments`, `local variables` and `fields` segments of the calling method are restored.
   - Control is transferred to the return address in the calling method.

**SPEC 7-13: The method-calling protocol and its implementation.**
Chapter 7: The Virtual Machine

7.7 Behind the Scene: the VM Implementation

Up to this point of the chapter we’ve described all the functional features of Vack programming, so readers who like this language (if such perversion exists) can start writing Vack programs without additional instruction. However, recall that Vack is supposed to serve as an intermediate stage between high-level programming and machine execution. As such, Vack programs are rarely written by humans – they are generated automatically by compilers. If we want to complete the process and have these Vack programs run for real, we must translate them into the machine language of the target platform. Such translation will entail three basic services:

- All the abstract data structures that Vack programs take for granted must be simulated on RAM. In particular, the behaviors of the stack, heap, globals, locals, arguments, fields and array elements must be implemented on the target platform.

- Each Vack command must be translated into a series of machine-level commands of the target platform; These commands will operate on the implementations of the abstract data structures mentioned above.

- The method-calling protocol must be simulated and effected on the target platform.

The program that provides all these services is called the VM implementation, or simply the VM. One such program must be written for every target platform on which Vack programs are supposed to run. With that in mind, we divide the discussion of the VM implementation into two sections: general and hardware-specific. The present section discusses general implementation techniques, focusing mainly on stack management. The specific mapping between the VM implementation and the Hack hardware will be discussed in the next section.

Although the forthcoming discussion is somewhat involved, it has its rewards: the principles that guide the implementation of a virtual machine run deep, and illustrate some of the most elegant and profound ideas in software engineering. One cannot be an accomplished software developer without understanding these ideas, at least intuitively.

**Method State:** The VM model assumes that each method either runs or waits in a private and protected environment, which we call the method’s state. The method’s state consists of the method’s arguments segment, local variables segment, and field variables segment (see Figure 7-6). Thus, when one method calls another method, the VM implementation must save the state of the calling method, and construct a new state for the called method. When the called method returns, the VM implementation must abandon the state of the terminated method and restore the state of the calling method. The goal is to allow the calling method to resume its execution as if “nothing happened”.

**Method Frames:** One can think of several ways to save and restore method states in computer memory. The most elegant and efficient solution is a stack implementation, which we now turn to describe. A stack is a linear data structure that grows in one direction only. Each method that starts running on the computer receives and occupies a private area on the stack called the method frame, which persists as long as the method has not returned. Thus, if method a calls method b and method b calls method c, then as long as c is running, the stack contains three method frames.
When method c returns, its frame is abandoned, and the stack shrinks to contain two frames only. Importantly, the frame of each method contains all the information necessary to restore the state of the method that called it (in the stack metaphor – the method "just above" it). The details are given in Figure 7-15.

![Diagram of the method frame]

**FIGURE 7-15: The method frame.** The frame of the current method contains all the memory segments of the current method, as well as all the information necessary to restore access to the memory segments of the calling method (when the current method returns).

An inspection of Figure 7-15 brings to light several observations. First and foremost, note that the stack – the centerpiece of the VM -- provides a variety of services that are quite unrelated to each other. The tail of the stack contains the frames of all the methods that are waiting for the current method to terminate its execution. Within each frame, one stack area contains the memory segments that serves the method, while another area contains saved pointers that will be used to restore the memory segments of its calling method. Finally, the "cutting edge" of the stack is the working memory of the current method – this is where commands like `pop`, `push`, and `add` leave their mark. As the current method starts running, the stack grows in one direction only -- *downward*. This ensures that the frames of all the calling methods above the current method are protected, since the current method cannot touch them *by design.*
Second, note that the method's *arguments* and *local variables* are stored in the stack as continuous memory segments. Likewise, the method's *field variables* are stored in the heap as a continuous memory segment (not shown in Figure 7-15). Hence, if we want to save the state of these three memory segments, it is sufficient to save pointers to their base addresses (ARG, LCL, and THIS, respectively). In short, if $i$ represents the order of the current method in the calling methods chain, then the frame of method $i$ embeds all the information necessary to reconstruct the state of method $i-1$, et cetera, all the way up to the main method that started the calling chain.

Finally, the stack structure facilitates all the elements necessary for managing a dynamic method calling protocol. When the current method returns, we can (i) reset the pointers ARG, LCL, and THIS to their saved values, (ii) re-position SP "upward" in order to get rid of the junk of the current frame, and (iii) transfer control to the saved return address. This arrangement gives the calling method a "license to kill," since the method will start using the stack from the new (and higher) location of SP downward, running over the remains of the frame of the terminated method. One result of this setting is automatic recycling of memory resources.

Dynamic stack management is one of the most important ideas in computer science. We do hope that the reader is not left indifferent to the sheer elegance of this beautiful design.
**Method call simulation:** In order to illustrate stack management in action, we will now revisit the `fact` method, which computes the factorial of a given number \( n \) by repetitively calling the `mult` method \( n \) times. Figure 7-16 shows how the VM facilitates one such call-and-return sequence.

![Figure 7-16: Dynamic stack behavior.](image-url)

We assume that method \( p \) called method `fact`, then `fact` called `mult`. Among other things, the figure shows how the calling method bears the responsibility of setting up the arguments for the called method. This is shown at the bottom of the left stack instance, where `fact` sets the stage for `mult`, preparing its arguments. If we ignore the middle stack instance, we observe that `fact` has set up some arguments and called `mult` to operate on them (left instance). When `mult` returns (right instance), the arguments of the called method have been replaced with the method’s return value. In other words, when the dust clears from the method call, the calling method has received the service that it has requested, and processing resumes as if nothing happened: the drama of `mult`'s processing has left no trace whatsoever on the stack, except for the return value.
**Implementation of the method calling protocol:** In order to effect the stack behavior that Figure 7-16 illustrates, the VM must perform various housekeeping chores whenever a method is being called, initiated, or returns. Here are the details, in stack-oriented pseudo-code:

<table>
<thead>
<tr>
<th>Vack command</th>
<th>VM Implementation</th>
</tr>
</thead>
</table>
| call m c     | push return address  // save state  
|              | push THIS           // save state    
|              | push ARG            // save state    
|              | push LCL            // save state    
|              | ARG = SP-c-4        // reposition ARG (c=number of args) 
|              | LCL = SP+1          // reposition LCL 
|              | THIS = *ARG         // reposition THIS 
|              | goto m              // transfer control |

| method m c | repeat c times: // c = number of local variables  
|           | PUSH 0           // initialize all of them to 0 |
| return    | *ARG=pop()       // put ret. val. in the right place  
|           | SP=ARG++         // restore SP              
|           | A=LCL            // A is a temporary variable    
|           | LCL=*(--A)       // restore LCL of the calling method  
|           | ARG=*(--A)       // restore ARG of the calling method  
|           | THIS=*(--A)      // restore THIS of the calling method  
|           | goto *(--A)      // GOTO the return-address |

**SPEC 7-17:** The VM implementation of the method-calling protocol.

### 7.8 VM Implementation over the Hack platform

The previous section described the general principles and techniques underlying the implementation of the VM on *any* hardware platform. In this section we complete the picture, providing all the details necessary to map the VM on the Hack platform. The reason for specifying this standard mapping (rather than leaving it to the discretion of any VM implementation) is two-fold. First, we wish to allow inter-operability with other high-level languages implemented over the Hack platform. Second, we wish to allow the developers of the VM implementation to run well-defined tests.
The sys.init method: We assume that each application contains a `sys.init` method, designed to invoke the other methods in the application. For example, to invoke the factorial method on n=5, we use the following code:

```java
method sys.init 0 // no local variables
push-constant 0 // null THIS object
push-constant 5 // n=5
call fact 2 // invoke fact; two arguments were pushed
return
```

Method and label mapping: Vack commands are compiled by the VM into Hack assembly code. Each Vack method is compiled separately. Operating system methods are compiled and treated just like regular methods. The method name is used as a prefix symbol in the assembly language translation: each label x mentioned in the code of method m is translated to the assembly language symbol m.x.

Memory mapping of the VM constants: These are given in Spec. 7-19.

<table>
<thead>
<tr>
<th>Address</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>Here starts the bootstrap code that prepares everything and calls sys.init. Next comes the code of all the methods, in arbitrary order.</td>
</tr>
<tr>
<td>0x2000</td>
<td>SP: points to the top of the stack, i.e. to the stack location that will be filled in the next push operation; also called R0;</td>
</tr>
<tr>
<td>0x2001</td>
<td>GLBL: points to the base of the <code>global variables</code> segment; also called R1;</td>
</tr>
<tr>
<td>0x2002</td>
<td>LCL: points to the base of the <code>local variables</code> segment of the current method, within the stack; also called R2;</td>
</tr>
<tr>
<td>0x2003</td>
<td>ARG: points to the base of the <code>arguments</code> segment of the current method, within the stack; also called R3;</td>
</tr>
<tr>
<td>0x2004</td>
<td>THIS: points to the base of the <code>fields segment</code> of the object of the current method, within the heap; also called R4;</td>
</tr>
<tr>
<td>0x2005</td>
<td>ARR: points to the base of the <code>current array</code>, within the heap; also called R5;</td>
</tr>
<tr>
<td>0x2006-0x200F</td>
<td>Ten general-purpose registers, named R6-R15, that may be used at will by Hack programs (irrelevant to the VM).</td>
</tr>
<tr>
<td>0x2010</td>
<td>Here starts the <code>globals segment</code>;</td>
</tr>
<tr>
<td>0x2110</td>
<td>Here starts the <code>Stack</code>;</td>
</tr>
<tr>
<td>0x2800</td>
<td>Here starts the <code>Heap</code>;</td>
</tr>
<tr>
<td>0x4000</td>
<td>Beginning of memory-mapped I/O (irrelevant to the VM)</td>
</tr>
</tbody>
</table>

SPEC 7-19: Special addresses in the Hack/VM implementation. See Spec. 7-21 for another view of this mapping. Hack programs recognize 16 special memory locations as "registers" labeled R0..R15. The VM uses the first 6 registers, and thus they should not be touched by any Hack program written in assembly or machine language. The remaining 10 registers can be used at the programmer's discretion.
Chapter 7: The Virtual Machine

ROM

0x0000

bootstrap code
+ o/s methods code
+ app. methods code

the bootstrap code prepares everything and calls
the sys.init method. Then come the code
segments of all the application methods and the
O/S methods, in arbitrary order.

0x2000

SP

stack pointer, points to the stack’s top

0x2001

GLBL

points to the base of the global variables
segment.

0x2002

LCL

points to the base of the local variables segment
of the current method (within the stack)

0x2003

ARG

points to the base of the arguments segment of
the current method (within the stack)

0x2004

THIS

points to the base of the fields segment of the
current method's object (within the heap)

0x2005

ARR

points to the base of the current array
(within the heap)

0x2006

R6

points to the base of the current array

0x200F

R15

ten general-purpose registers.
Can be used at the programmer’s discretion

0x2010

global variables segment

Contains 256 variables that can be used by all
the methods.

0x2110

Stack

overall stack, holding all the method frames and
ending with the working stack of the current
method.

0x2800

Heap

stores all the objects and arrays data.

0x4000

I/O buffers

memory-mapped input/output segments

RAM

SPEC 7-21: the overall Hack memory map along with the VM implementation bindings.

Hack’s memory consists of 32K 16-bit words, of which 24K+1 words are actually used (8K
ROM, 8K RAM, 8K screen memory, 1 word keyboard memory).
Bootstrap code: This initialization code resides permanently in the ROM, beginning in address 0x0000, and starts executing when the computer is turned on. The bootstrap code sets up some pointers to their right values and then calls the sys.init method. The pseudo code is as follows:

```
GLBL = 0x2010  // initialize the base of the globals segment
SP = 0x2110    // initialize the stack pointer
push 0         // set up a null object for the sys.init method
call sys.init  // invoke sys.init
loop forever   // halt when sys.init returns.
```

SPEC 7-22: the bootstrap code. This code sets up some VM pointers and calls the sys.init method. The sys.init method then calls the main method of the application.