# 7. Virtual Machine I: Stack Arithmetic[1]

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

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

This chapter describes the first steps toward building a *compiler* for a typical object-based high-level language. We will approach this substantial task in two stages, each spanning two chapters. High-level programs will be first translated into an intermediate code (Chapters 10-11), and the intermediate code will then be translated into machine language (Chapters 7-8). This two-tier translation model is a rather old idea that goes back to the 1970's. Recently, it made a significant comeback following its adoption by modern languages like Java.

The basic idea is as follows: instead of running on a real platform, the intermediate code is designed to run on a *Virtual Machine*. The VM is an abstract computer that does not exist for real, but can rather be realized on other computer platforms. There are many reasons why this idea makes sense, one of which being code transportability. Since the VM may be implemented with relative ease on multiple target platforms, it allows running software on many processors and operating systems without having to modify the original source code. The VM implementation on target platforms can be done in several ways, by software interpreters, by special purpose hardware, or by translating the VM programs into the machine language of the target platform.

This chapter presents a typical VM architecture, modeled after the *Java Virtual Machine* (JVM) paradigm. As usual, we focus on two perspectives. First, we will motivate and specify the VM abstraction. Next, we will implement it over the Hack platform. Our implementation will entail writing a program called *VM translator*, designed to translate VM code into Hack assembly code. The software suite that comes with the book illustrates yet another implementation vehicle, called *VM emulator*. This program implements the VM by emulating it on a standard personal computer using Java.

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

The virtual machine that will emerge from this effort illustrates many important ideas in computer science. First, the notion of having one computer emulating another is a fundamental idea in the field, tracing back to Alan Turing in the 1930's. Over the years it had many practical implications, e.g. using an emulator of an old generation computer running on a new platform in order to achieve upward code compatibility. More recently, the virtual machine model became

[1] From *The Elements of Computing Systems* by Nisan & Schocken (draft ed.), MIT Press, 2005, www.idc.ac.il/tecs

the centerpiece of two competing mainstreams -- the Java architecture and the .NET infrastructure. These software environments are rather complex, and one way to gain an inside view of their underlying structure is to build a simple version of their VM cores, as we do here.

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

## 7.1 Background

### The Virtual Machine Paradigm

Before a high-level program can run on a target computer, it must be translated into the computer's machine language. This translation -- known as *compilation* -- is a rather complex process. Normally, a separate compiler is written specifically for any given pair of high-level language and target machine language. This leads to a proliferation of many different compilers, each depending on every detail of both its source and destination languages. One way to decouple this dependency is to break the overall compilation process into two nearly separate stages. In the first stage, the high-level program is parsed and its commands are translated into intermediate steps -- steps that are neither "high" nor "low". In the second stage, the intermediate steps are translated further into the machine language of the target hardware.

This decomposition is very appealing from a software engineering perspective: the first stage depends only on the specifics of the source high-level language, and the second stage only on the specifics of the target machine language. Of course, the interface between the two compilation stages -- the exact definition of the intermediate steps -- must be carefully designed. In fact, this interface is sufficiently important to merit its own definition as a stand-alone language of an abstract machine. Specifically, one formulates a *virtual machine* whose instructions are the intermediate steps into which high-level commands are decomposed. The compiler that was formerly a single monolithic program is now split into two separate programs. The first program, still termed *compiler*, translates the high-level code into intermediate virtual machine instructions, while the second program translates this VM code into the machine language of the target platform.

This two-stage compilation model has been used -- one way or another -- in many compiler construction projects. Some developers went as far as defining a formal and stand-alone virtual machine language, most notably the *p-code* generated by several Pascal compilers in the 1970s. Java compilers are also two-tiered, generating a *bytecode* language that runs on the JVM virtual machine (sometimes called the *Java Runtime Environment*). More recently, the approach has been adopted by the .NET infrastructure. In particular, .NET requires compilers to generate code written in an *intermediate language* (IL) that runs on a virtual machine called CLR (*Common Language Runtime*).

Indeed, the notion of an explicit and formal virtual machine language has several practical advantages. First, compilers for different target platforms can be obtained with relative ease by replacing only the virtual machine implementation (sometimes called the compiler's "backend").

This, in turn, allows the VM code to become transportable across different hardware platforms, permitting a range of implementation tradeoffs between code efficiency, hardware cost, and programming effort. Second, compilers for many languages can share the same VM "backend", allowing code sharing and language inter-operability. For example, one high-level language may be good at scientific calculations, while another may excel in handling the GUI. If both languages compile into a common VM layer, it is rather natural to have routines in one language call routines in the other, using an agreed-upon invocation syntax.

Another benefit of the virtual machine approach is modularity. Every improvement in the efficiency of the VM implementation is immediately inherited by all the compilers above it. Likewise, every new digital device or appliance which is equipped with a VM implementation can immediately gain access to a huge base of available software.

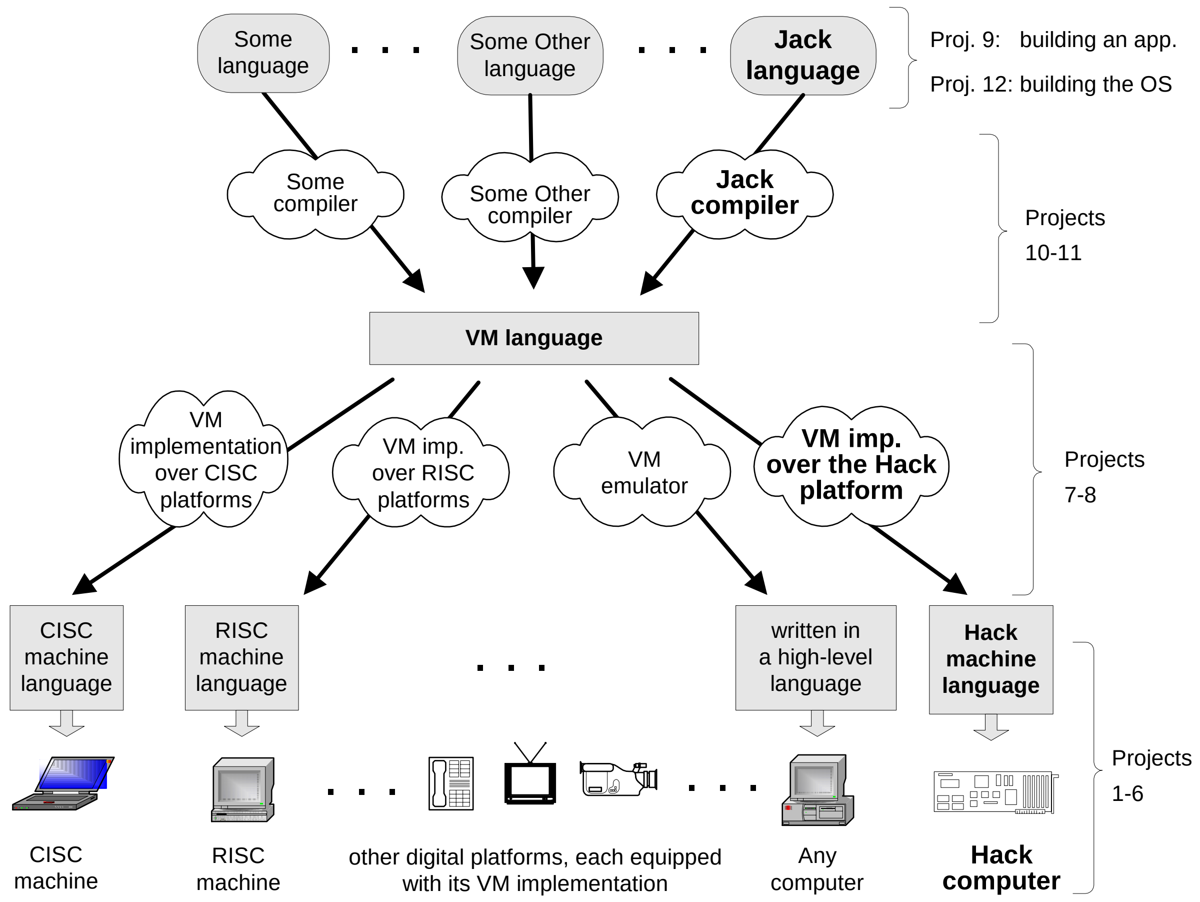


**FIGURE 7.1: The virtual machine paradigm**. Once a high-level program is compiled into VM code, the program can run on any hardware platform equipped with a suitable VM implementation. In this chapter we will start building a *VM implementation over the Hack Platform*, and use a *VM emulator* to run VM programs on a regular PC.

## The Stack Machine Model

Like most programming languages, the VM language consists of arithmetic, memory access, program flow, and subroutine calling operations. There are several possible software paradigms on which to base such a language implementation. One of the key questions regarding this choice is *where will the operands and the results of the VM operations reside*? Perhaps the cleanest solution is to put them on a *stack* data structure.

In a *stack machine* model, arithmetic commands pop their operands from the top of the stack and push their results back onto the top of the stack. Other commands transfer data items from the stack's top to designated memory locations, and vice versa. As it turns out, these simple stack operations can be used to implement the evaluation of any arithmetic or logical expression. Further, any program, written in any programming language, can be translated into an equivalent stack machine program. One such stack machine model is used in the *Java Virtual Machine* as well as in the VM described and built below.

**Elementary Stack Operations:** A stack is an abstract data structure that supports two basic operations: *push* and *pop*. The *push* operation adds an element to the top of the stack; the element that was previously on top is pushed below the newly added element. The *pop* operation retrieves and removes the top element; the element just below it moves up to the top position. Thus the stack implements a *last-in-first-out* (LIFO) storage model, illustrated in Figure 7.2.

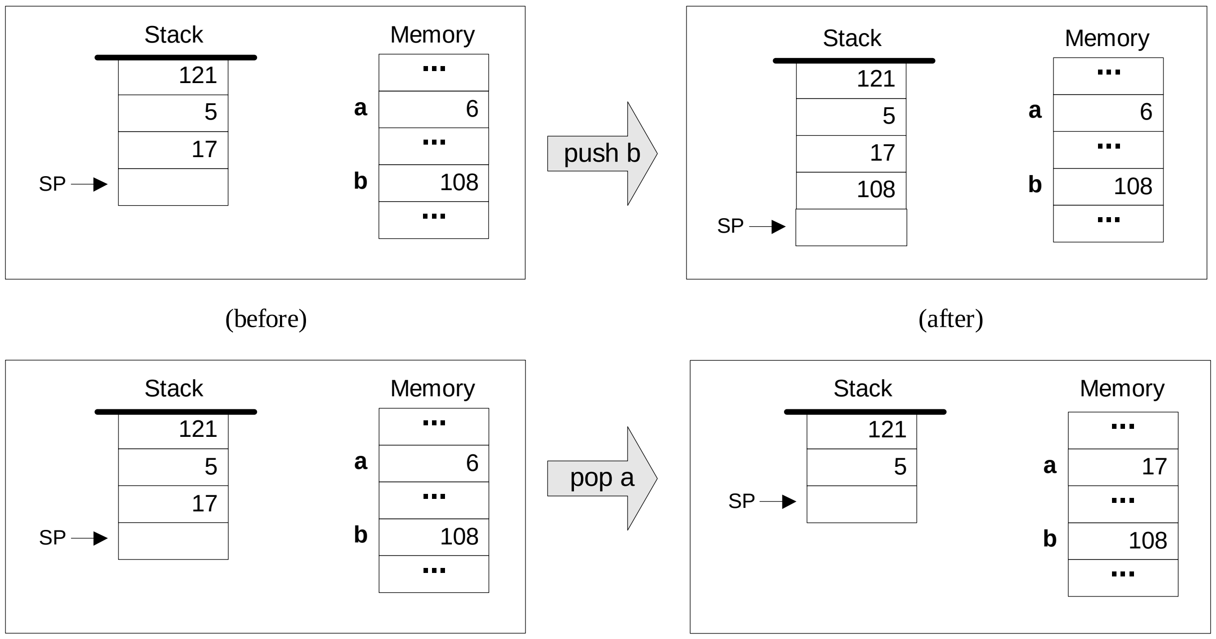


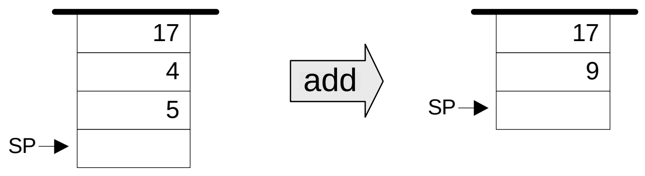
**FIGURE 7.2: Stack processing example,** illustrating the two elementary operations *push* and *pop*. Following convention, the stack is drawn upside down, as if it grows downward. The location just after the top position is always referred to by a special pointer called sp, or *stack pointer*. The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

We see that stack access differs from conventional memory access in several respects. First, the stack is accessible only from the top, one item at a time. Second, reading the stack is a lossy operation: the only way to retrieve the top value is to *remove* it from the stack. In contrast, the act of reading a value from a regular memory location has no impact on the memory's state. Finally, writing an item onto the stack adds it to the stack's top, without changing the rest of the stack. In contrast, writing an item into a regular memory location is a lossy operation, since it overrides the location's previous value.

The stack data structure can be implemented in several different ways. The simplest approach is to keep an array, say *stack*, and a *stack pointer* variable, say *sp*, that points to the available location just above the "topmost" element. The *push x* command is then implemented by storing *x* at the array entry pointed by *sp* and then incrementing *sp* (i.e. `stack[sp]=x; sp=sp+1`). The *pop* operation is implemented by first decrementing *sp* and then returning the value stored in the top position (i.e. `sp=sp-1; return stack[sp]`).

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

**Stack Arithmetic:** Stack-based arithmetic is a simple matter: the two top elements are popped from the stack, the required operation is performed on them, and the result is pushed back onto the stack. For example, here is how addition is handled:



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

```
// d=(2-x)*(y+5)
push 2
push x
sub
push y
push 5
add
mult
pop d
```

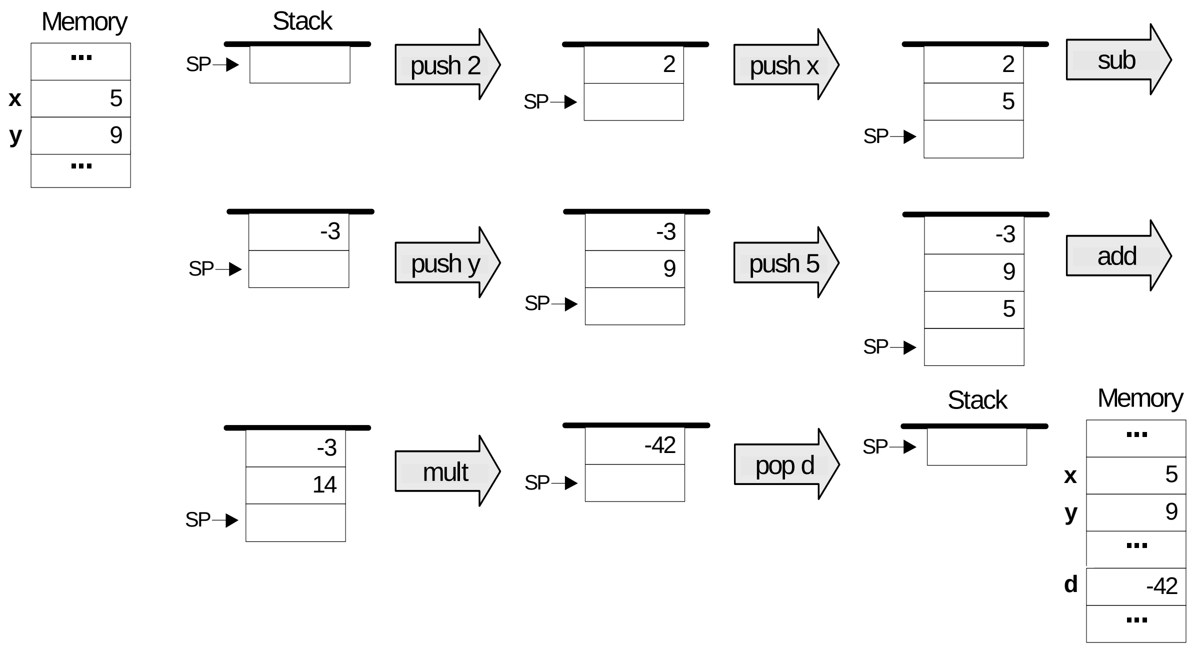


**FIGURE 7.3: Stack-based evaluation of arithmetic expressions.** This example evaluates the expression "`d=(2−x)*(y+5)`", assuming the initial memory state `x=5`, `y=9`.

Stack-based evaluation of Boolean expressions has precisely the same flavor. For example, consider the high-level command "`if (x<7) or (y=8) then` ...". The stack-based evaluation of this expression is shown in figure 7.4.

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

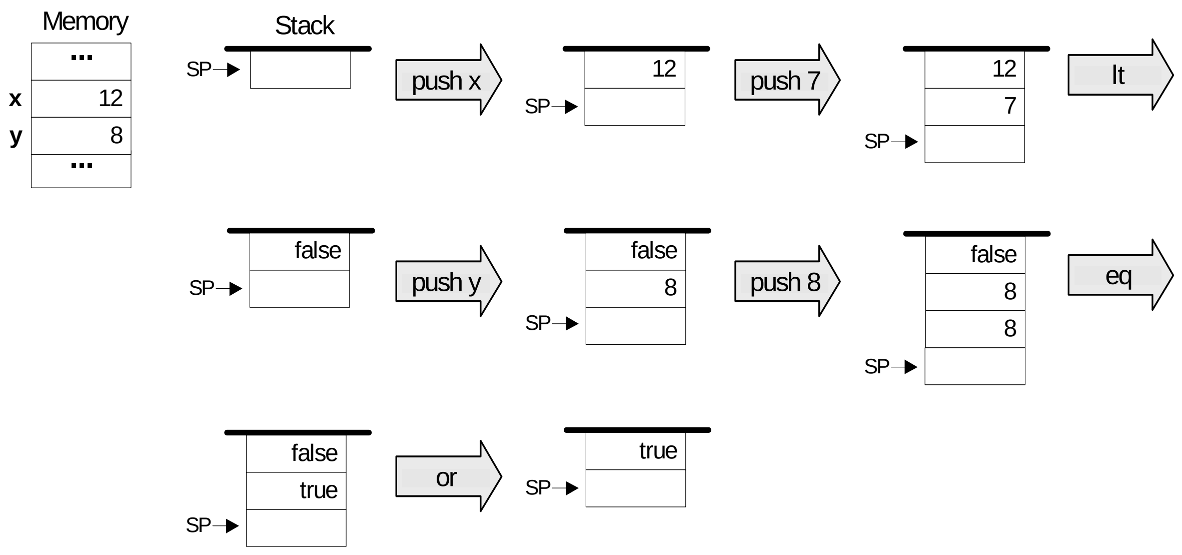


**FIGURE 7.4: Stack-based evaluation of logical expressions.** This example evaluates the boolean expression "`(x<7) or (y=8)`", assuming the initial memory state `x=12`, `y=8`.

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

## 7.2 VM Specification, Part I

### 7.2.1 General

The virtual machine is *stack-based*: all operations are done on a stack. It is also *function-based*: a complete VM program is organized in program units called *functions*, written in the VM language. Each function has its own stand-alone code and is separately handled. The VM language has a single 16-bit data type that can be used as an integer, a Boolean, or a pointer. The language consists of four types of commands:

- **Arithmetic commands** perform arithmetic and logical operations on the stack;
- **Memory access commands** transfer data between the stack and virtual memory segments;
- **Program flow commands** facilitate conditional and unconditional branching operations;
- **Function calling commands** call functions and return from them.

Building a virtual machine is a complex undertaking, and so we divide it into two stages. In this chapter we specify the *arithmetic* and *memory access* commands, and build a basic VM translator that implements them only. The next chapter specifies the program flow and function calling commands, and extends the basic translator into a full-scale virtual machine implementation.

**Program and command structure:** A VM *program* is a collection of one or more *files* with a `.vm` extension, each consisting of one or more *function*s. From a compilation standpoint, these constructs correspond, respectively, to the notions of *program*, *class*, and *method* in an object-oriented language.

Within a `.vm` file, each VM command appears in a separate line, and in one of the following formats: *command* (e.g. "`add`"), *command arg* (e.g. "`goto loop`"), or *command arg1 arg2* (e.g. "`push local 3`"). The arguments are separated from each other and from the *command* part by an arbitrary number of spaces. "//" comments can appear at the end of any line and are ignored. Blank lines are permitted and are ignored.

## 7.2.2 Arithmetic and logical commands

The VM language features nine stack-oriented arithmetic and logical commands. Seven of these commands are binary: they pop two items off the stack, compute a binary function on them, and push the result back onto the stack. The remaining two commands are unary: they pop a single item off the stack, compute a unary function on it, and push the result back onto the stack. We see that each command has the net impact of replacing its operand(s) with the command's result, without affecting the rest of the stack. Figure 7.5 gives the details.

| **Command** | **Return value** (after popping the operand/s) | **Comment** | |
|---|---|---|---|
| `add` | $x + y$ | Integer addition | (2's complement) |
| `sub` | $x - y$ | Integer subtraction | (2's complement) |
| `neg` | $-y$ | Arithmetic negation | (2's complement) |
| `eq` | true if $x = y$ and false otherwise | Equality | |
| `gt` | true if $x > y$ and false otherwise | Greater than | |
| `lt` | true if $x < y$ and false otherwise | Less than | |
| `and` | $x$ And $y$ | Bit-wise | |
| `or` | $x$ Or $y$ | Bit-wise | |
| `not` | Not $y$ | Bit-wise | |

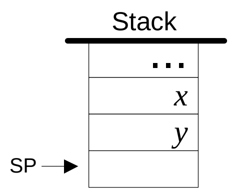


**FIGURE 7.5: Arithmetic and Logical stack commands.**

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

## 7.2.3 Memory Access Commands

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

| **Segment** | **Purpose** | **Comments** |
|---|---|---|
| `argument` | Stores the function's arguments. | Allocated dynamically by the VM implementation when the function is entered. |
| `local` | Stores the function's local variables. | Allocated dynamically by the VM implementation and initialized to 0 when the function is entered. |
| `static` | Stores static variables shared by all functions in the same `.vm` file. | Allocated by the VM implementation for each `.vm` file; shared by all functions in the `.vm` file. |
| `constant` | Pseudo-segment that holds all the constants in the range 0 ... 32767. | Emulated by the VM implementation; Seen by all the functions in the program. |
| `this` `that` | General-purpose segments. Can be made to correspond to different areas in the heap. Serve various programming needs. | Any VM function can use these segments to manipulate selected areas on the heap. |
| `pointer` | A two-entry segment that holds the base addresses of the `this` and `that` segments. | Any VM function can set `Pointer 0` (or `1`) to some address; this has the effect of aligning the `this` (or `that`) segment to the area on the heap beginning in that address. |
| `temp` | Fixed eight-entry segment that holds temporary variables for general use. | May be used by any VM function for any purpose. Shared by all functions in the program. |

**FIGURE 7.6: The eight memory segments seen by every VM function.**

**Memory access commands:** All the memory segments are managed by the same 2 commands:

| | |
|---|---|
| • `push` *segment index* | Push the value of *segment*[*index*] onto the stack; |
| • `pop` *segment index* | Pop the topmost stack item and store its value in *segment*[*index*]. |

Where *segment* is one of the eight segment names and *index* is a non-negative integer. For example, "`push argument 2`" followed by "`pop local 1`" will store the value of the function's 3rd argument in the function's 2nd local variable (each segment's index starts at 0).

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

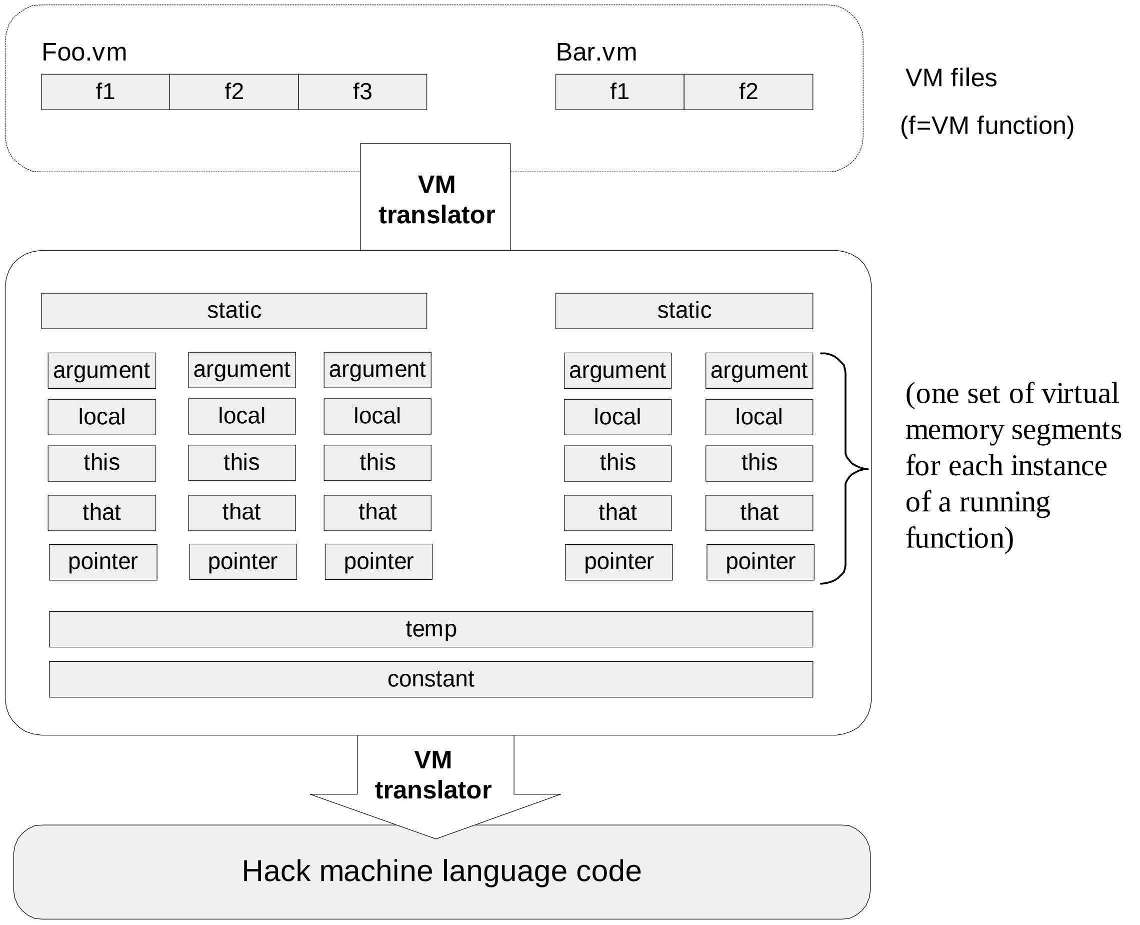


**FIGURE 7.7: The virtual memory segments** are maintained and simulated by the VM implementation.

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

**The stack:** Consider the commands sequence "`push argument 2`" and "`pop local 1`", mentioned before. The working memory of such VM operations is the *stack*. The data value did not simply jump from one segment to another -- it went through the stack. Yet in spite of its central role in the VM architecture, the stack proper is never mentioned in the VM language.

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

### 7.2.4 Program flow and function calling commands

The VM features 6 additional commands that are discussed at length in the next chapter. For completeness, these commands are listed below.

**Program Flow Commands**

- `label` *symbol* // Label declaration
- `goto` *symbol* // Unconditional branching
- `if-goto` *symbol* // Conditional branching

**Function Calling Commands**

- `function` *functionName nLocals* // Function declaration; must specify the
  // number of the function's local variables
- `call` *functionName nArgs* // Function invocation; must specify the
  // number of the function's arguments
- `return` // Transfer control back to the calling function

(In the above, *functionName* is a symbol and *nLocals* and *nArgs* are non-negative integers.)

### 7.2.5 Program elements in the Jack-VM-Hack platform

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

Next, the figure shows how the *VM translator* can be applied to the directory in which the VM files reside, generating a single assembly program. This assembly program does two main things. First, it emulates the virtual memory segments of each VM function and file, as well as the implicit stack. Second, it effects the VM commands on the target platform. This is done by manipulating the emulated VM data structures using machine language instructions -- those translated from the VM commands. If all works well, i.e. if the compiler and the VM translator and the assembler are implemented correctly, the target platform will end up affecting the behavior mandated by the original Jack program.

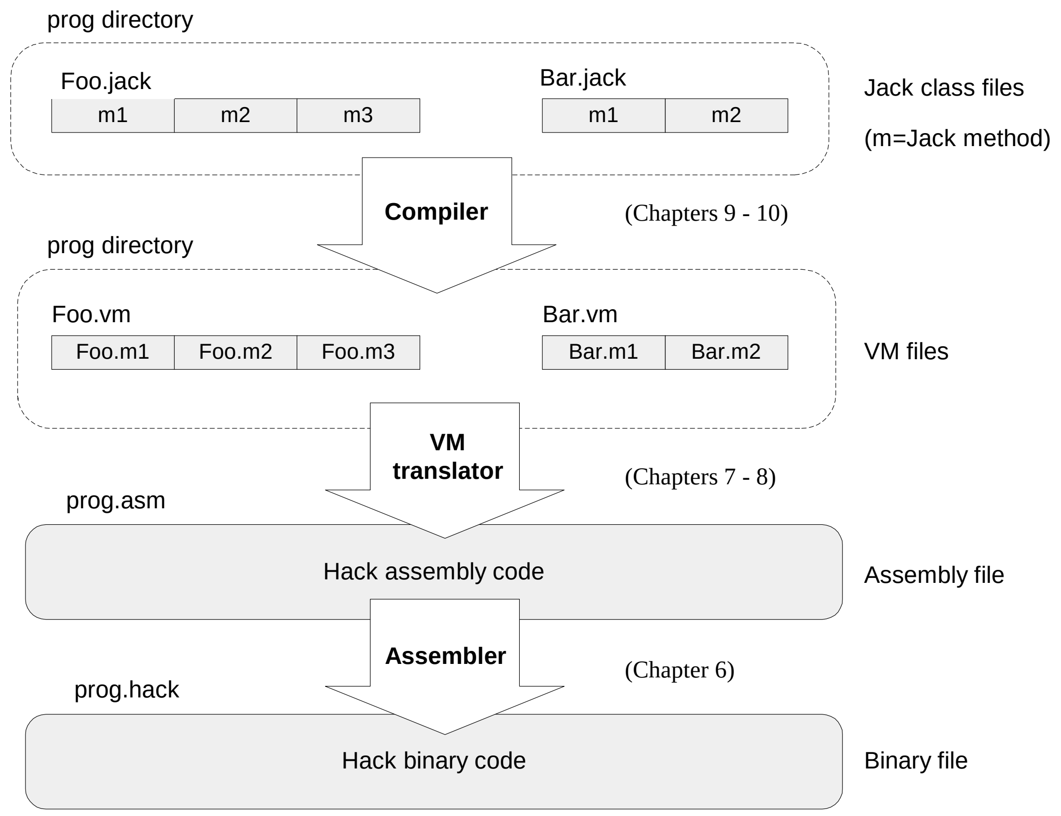


**FIGURE 7.8: Program elements in the Jack-VM-Hack platform**

### 7.2.6 VM Programming Examples

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

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

### A Typical Arithmetic Task

Consider the multiplication algorithm shown at the top of Figure 7.9. How should we (or more likely, the compiler) express this algorithm in the VM language? First, high level structures like "`for`" and "`while`" must be rewritten using the VM's simple "goto logic." In a similar fashion, high-level arithmetic and Boolean operations must be expressed using stack-oriented commands. The resulting code is shown in Figure 7.9. (The exact semantics of the VM commands `function`, `label`, `goto`, `if-goto`, and `return` are described in chapter 8, but their intuitive meaning is self-explanatory.)

***High-level code (C style)***

```
int mult(int x, int y) {
  int sum;
  sum = 0;
  for(int j = y; j != 0; j--)
      sum += x;  // Repetitive addition
  return sum;
}
```

***First approximation***

```
function mult
  args x, y
  vars sum, j
  sum = 0
  j = y
loop:
  if j = 0 goto end
  sum = sum + x
  j = j - 1
  goto loop
end:
  return sum
```

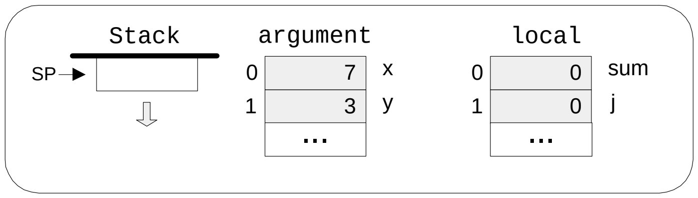
***Pseudo VM code***

```
function mult(x,y)
   push 0
   pop sum
   push y
   pop j
label loop
   push 0
   push j
   eq
   if-goto end
   push sum
   push x
   add
   pop sum
   push j
   push 1
   sub
   pop j
   goto loop
label end
   push sum
   return
```

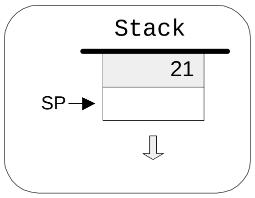
***Final VM code***

```
function mult 2       // Two local vars
   push constant 0
   pop local 0        // sum = 0
   push argument 1
   pop local 1        // j = y
label loop
   push constant 0
   push local 1
   eq
   if-goto end        // If j = 0 goto end
   push local 0
   push argument 0
   add
   pop local 0        // sum = sum + x
   push local 1
   push constant 1
   sub
   pop local 1        // j = j - 1
   goto loop
label end
   push local 0
   return             // Return sum
```

Just after mult(7,3) is entered:



Just after mult(7,3) returns:



(The symbols x, y, sum, and j are not part of the VM program, and are shown here only for ease of reference)

**FIGURE 7.9: VM programming example.** The function's arguments (7 and 3 in this example) are set up by the *caller* of the function, not shown here.

**Explanation:** Let us focus on the bottom of Figure 7.9. We see that when a VM function starts running, it assumes that (i) the stack is empty, (ii) the argument values on which it is supposed to operate are located in the `argument` segment, and (iii) the local variables that it is supposed to use are initialized to 0 and located in the `local` segment.

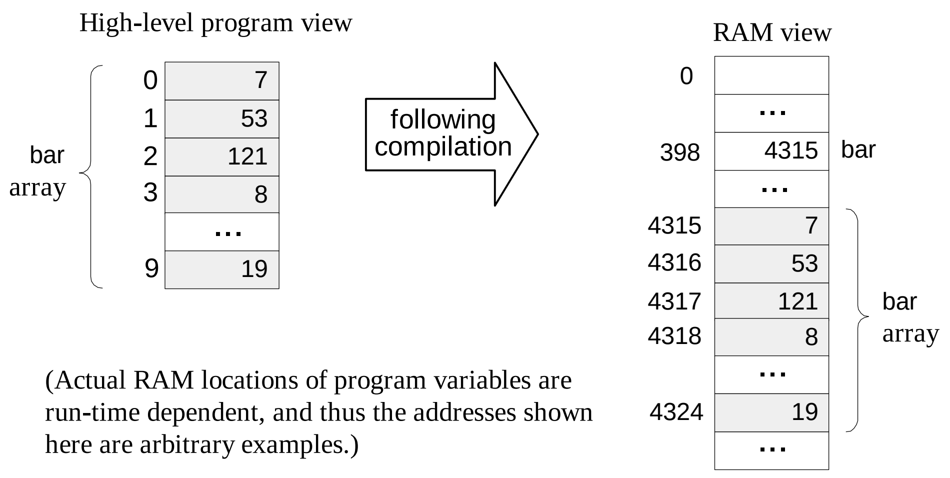
Let us now focus on the VM representation of the algorithm. Recall that VM commands cannot use symbolic argument and variable names -- they are limited to making <*segment index*> references only. However, the translation from the former to the latter is straightforward. All we have to do is map `x`, `y`, `sum` and `j` on `argument 0`, `argument 1`, `local 0` and `local 1`, respectively, and replace all their symbolic occurrences in the pseudo code with corresponding <*segment index*> references.

To sum up, when a VM function starts running, it assumes that it is surrounded by a private world, all of its own, consisting of initialized `argument` and `local` segments and an empty stack, waiting to be manipulated by its commands. The agent responsible for staging this virtual worldview for every VM function just before it starts running is the VM implementation, as we will see in the next chapter.

## Array Handling

An array is an indexed collection of objects. Suppose that a high-level program has created an array of 10 integers called `bar`, and filled it with some 10 numbers. Let us assume that the array's base has been mapped (behind the scene) on RAM address 4315. Suppose now that the high-level program wants to execute the command `bar[2]=19`. How can we implement this operation at the VM level?

In the C language, such an operation can be also specified as `*(bar+2)=19`. This C notation reads: "*set the RAM location whose address is* `(bar+2)` *to* `19`". As shown in Figure 7.10, this operation lends itself perfectly well to the VM language.



***VM code***

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

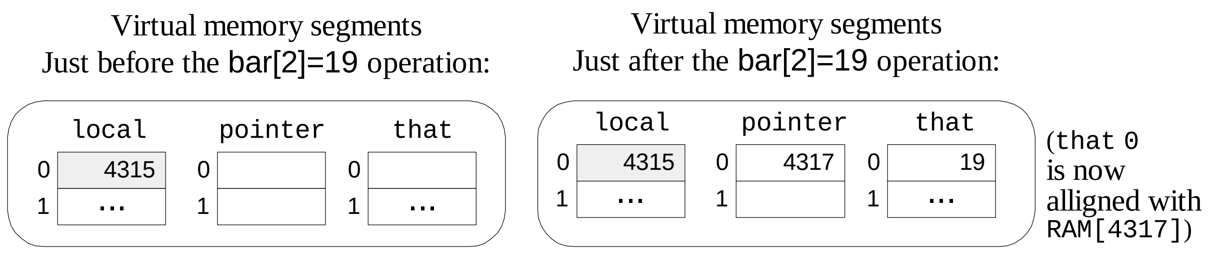


**FIGURE 7.10: VM-based Array manipulation**
using the `pointer` and `that` segments.

It remains to be seen, of course, how a high-level command like `bar[2]=19` is translated into the VM code shown in Figure 7.10 in the first place. This transformation is described in section 11.1.1 (Chapter 11), when we discuss the code generation features of the compiler.

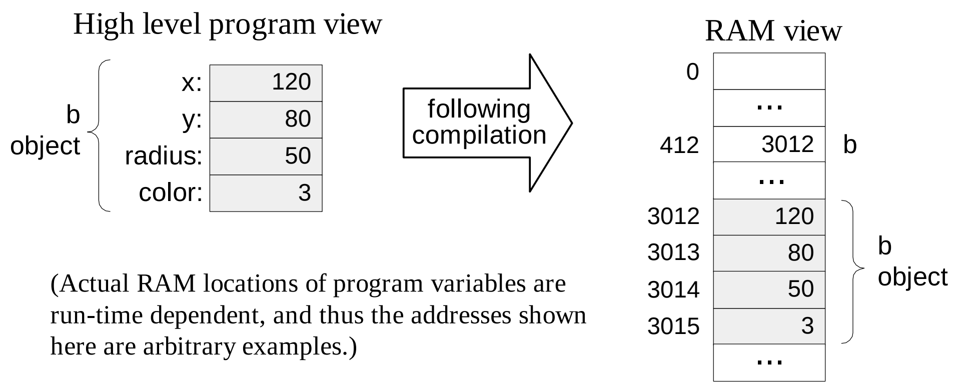
## Object handling

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

For example, consider an animation program designed to juggle some balls on the screen. Suppose that each `Ball` object is characterized by the integer fields `x`, `y`, `radius`, and `color`. Let us assume that the program has created one such `Ball` object, and called it `b`. What will be the internal representation of this object in the computer?

Like all other object instances, it will be stored in the RAM. In particular, whenever a program creates a new object, the compiler computes the object's size in terms of words and the operating system finds and allocates enough RAM space to store it (the exact details of this operation are discussed in Chapter 11). For now, let us assume that our `b` object has been allocated RAM addresses 3012 to 3015, as shown in Figure 7.11.

Suppose now that a certain method in the high-level program, say `resize`, takes a `Ball` object and an integer `r` as arguments, and, among other things, sets the ball's `radius` to `r`. The VM representation of this logic is shown in Figure 7.11.



(Actual RAM locations of program variables are run-time dependent, and thus the addresses shown here are arbitrary examples.)

***VM code***

```
/* Assume that the b object and the r integer were passed
   to the function as its first two arguments. The following
   code implements the operation b.radius=r. */
  push argument 0   // Get b's base address
  pop pointer 0     // Point the this segment to b
  push argument 1   // Get r's value
  pop this 2        // Set b's third field to r
  ...
```

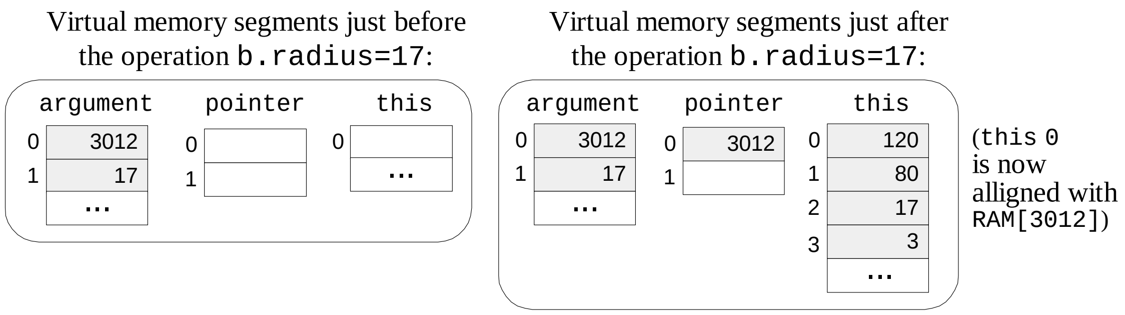

Virtual memory segments just before the operation b.radius=17:
argument
0 3012
1 17
...
pointer
0
1
this
0
...
Virtual memory segments just after the operation b.radius=17:
argument
0 3012
1 17
...
pointer
0 3012
1
this
0 120
1 80
2 17
3 3
...
(this 0 is now alligned with RAM[3012])

**FIGURE 7.11: VM-based object manipulation** using the `pointer` and `that` segments.

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

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

## 7.3 Implementation

The virtual machine that was described up to this point is an abstract artifact. If we want to use it for real, we must implement it on a real platform. Building such a VM implementation consists of two conceptual tasks. First, we have to emulate the VM world on the target platform. In particular, each data structure mentioned in the VM specification, i.e. the stack and the virtual memory segments, must be represented in some way by the hardware and low-level software of the target platform. Second, each VM command must be translated into a series of machine language instructions that affect the command's semantics on the target platform.

This section describes how to implement the VM specification (Section 7.2) on the Hack platform. We start by defining a "standard mapping" from VM elements and operations to the Hack hardware and machine language. Next, we suggest guidelines for designing the software that achieves this mapping. In what follows, we will refer to this software using the terms *VM implementation* or *VM translator* interchangeably.

### 7.3.1 Standard Mapping on the Hack Platform, Part I

If you re-read the virtual machine specification given so far, you will realize that it contains no assumption whatsoever about the architecture on which the machine can be implemented. When it comes to virtual machines, this platform-independence is the whole point: you don't want to commit to any one hardware platform, since you want your machine to potentially run on *all* of them, including those that were not built yet.

It follows that the VM designer can principally let programmers implement the VM on target platforms in any way they see fit. However, it is usually recommended to provide some guidelines as to how the VM should map on the target platform, rather than leave these decisions completely to the implementer's discretion. These guidelines, called *standard mapping*, are provided for two reasons. First, they entail a public contract that regulates how VM-based programs can interact with programs produced by compilers that don't use this VM (e.g. compilers that produce binary code directly). Second, we wish to allow the developers of the VM implementation to run standardized tests, i.e. tests that conform to the standard mapping. This way, the tests and the software can be written by different people, which is always recommended. With that in mind, the remainder of this section specifies the standard mapping of the VM on a familiar hardware platform: the Hack computer.

#### VM to Hack Translation

Recall that a VM program is a collection of one or more `.vm` files, each containing one or more VM functions, each being a sequence of VM commands. The VM translator takes a collection of `.vm` files as input and produces a single Hack assembly language `.asm` file as output (see Figure 7.7). Each VM command is translated by the VM translator into Hack assembly code. The order of the functions within the `.vm` files does not matter.

## RAM Usage

The data memory of the Hack computer consists of 32K 16-bit words. The first 16K serve as general-purpose RAM. The next 16K contain memory maps of I/O devices. The VM implementation should use this space as follows:

| *RAM addresses* | *Usage* |
|---:|---|
| 0–15 | Sixteen virtual registers, whose usage is described below |
| 16–255 | Static variables (of all the VM functions in the VM program) |
| 256–2047 | Stack |
| 2048–16483 | Heap (used to store objects and arrays) |
| 16384–24575 | Memory mapped I/O |

**FIGURE 7.12: Standard VM implementation on the Hack RAM.**

**Hack Registers:** According to the *Hack Machine Language Specification*, RAM addresses 0 to 15 can be referred to by any assembly program using the symbols `R0` to `R15`, respectively. In addition, the specification states that assembly programs can refer to RAM addresses 0 to 4 (i.e. `R0` to `R4`) using the symbols `SP`, `LCL`, `ARG`, `THIS`, and `THAT`. This convention was introduced into the assembly language with foresight, in order to promote readable VM implementations. The expected use of the Hack registers in the VM context is described in Figure 7.13.

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

**FIGURE 7.13: Usage of the Hack registers** in the standard mapping

## Memory Segments Mapping

**`local`**, **`argument`**, **`this`**, **`that`**: Each one of these segments is mapped directly on the Hack RAM, and its location is maintained by keeping its physical base address in a dedicated register (`LCL`, `ARG`, `THIS`, and `THAT`, respectively). Thus any access to the *i*-th entry of any one of these segments should be translated to assembly code that accesses address (*base*+*i*) in the RAM, where *base* is the current value stored in the register dedicated to the respective segment.

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

**`constant:`** This segment is truly virtual, as it does not occupy any physical space on the target architecture. Instead, the VM implementation handles any VM access to <*constant i*> by simply supplying the constant *i*.

**`static:`** According to the Hack machine language specification, when a new symbol is encountered for the first time in an assembly program, the assembler allocates a new RAM address to it, starting at address 16. This convention can be exploited to represent each static variable number `j` in a VM file `f` as the assembly language symbol `f.j`. For example, suppose that the file `Xxx.vm` contains the command "`push static 3`". This command can be translated to the Hack assembly commands "`@Xxx.3`" and "`D=M`", followed by additional assembly code that pushes `D`'s value to the stack. This implementation of the `static` segment is somewhat tricky, but it works.

## Assembly Language Symbols

Figure 7.14 recaps all the assembly language symbols used by VM implementations that conform to the standard mapping.

| *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` symbols | Each static variable `j` in 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. |
| Flow of control symbols | The implementation of the VM commands `function`, `call`, and `label` involves generating special label symbols, to be discussed in chapter 8. |

**FIGURE 7.14: Usage of special assembly symbols** prescribed by the VM-on-Hack standard mapping.

## 7.3.2 Design Suggestion for the VM implementation

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

```
            prompt> VMtranslator source
```

Where *source* is either a file name of the form `Xxx.vm` (the extension is mandatory) or a directory name containing one or more `.vm` files (in which case there is no extension). The result of the translation is always a single assembly language file named `Xxx.asm`, created in the same directory as the input `Xxx`. The translated code must conform to the standard VM-on-Hack mapping.

## Program Structure

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

### The *Parser* Module

**Parser:** Handles the parsing of a single `.vm` file, and encapsulates access to the input code. It reads VM commands, parses them, and provides convenient access to their components. In addition, it removes all white space and comments.

| **Routine** | **Arguments** | **Returns** | **Function** |
|---|---|---|---|
| Constructor | Input file / stream | -- | Opens the input file/stream and gets ready to parse it. |
| `hasMoreCommands` | -- | boolean | Are there more commands in the input? |
| `advance` | -- | -- | Reads the next command from the input and makes it the current command. Should be called only if `hasMoreCommands()` is true. Initially there is no current command. |
| `commandType` | -- | `C_ARITHMETIC`, `C_PUSH`, `C_POP`, `C_LABEL`, `C_GOTO`, `C_IF`, `C_FUNCTION`, `C_RETURN`, `C_CALL` | Returns the type of the current VM command. `C_ARITHMETIC` is returned for all the arithmetic commands. |
| `arg1` | -- | string | Returns the first argument of the current command. In the case of `C_ARITHMETIC`, the command itself (`add`, `sub`, etc.) is returned. Should not be called if the current command is `C_RETURN`. |
| `arg2` | -- | int | Returns the second argument of the current command. Should be called only if the current command is `C_PUSH`, `C_POP`, `C_FUNCTION`, or `C_CALL`. |

## The *CodeWriter* Module

| **CodeWriter:** Translates VM commands into Hack assembly code. | | | |
|---|---|---|---|
| **Routine** | **Arguments** | **Returns** | **Function** |
| Constructor | Output file / stream | -- | Opens the output file/stream and gets ready to write into it. |
| `setFileName` | `fileName` (string) | -- | Informs the code writer that the translation of a new VM file is started. |
| `writeArithmetic` | `command` (string) | -- | Writes the assembly code that is the translation of the given arithmetic command. |
| `WritePushPop` | `command` (`C_PUSH` or `C_POP`), `segment` (string), `index` (int) | -- | Writes the assembly code that is the translation of the given `command`, where `command` is either `C_PUSH` or `C_POP`. |
| `Close` | -- | -- | Closes the output file. |
| **Comment:** More routines will be added to this module in chapter 8. | | | |

## Main Program

The main program should construct a `Parser` to parse the VM input file and a `CodeWriter` to generate code into the corresponding output file. It should then march through the VM commands in the input file, and generate assembly code for each one of them.

If the program's argument is a directory name rather than a file name, the main program should process all the `.vm` files in this directory. In doing so, it should use a separate `Parser` for handling each input file and a single `CodeWriter` for handling the output.

## 7.4 Perspective

In this chapter we began the process of developing a compiler for a high-level language. Following modern software engineering practices, we have chosen to base the compiler on a two-tier compilation model. In the *frontend* stage, covered in chapters 10 and 11, the high-level code is translated into an intermediate code, running on a virtual machine. In the *backend* stage, covered in this and in the next chapter, the intermediate code is translated into the machine language of a target hardware platform (see Figures 1 and 9).

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

The JVM is a specification that describes an intermediate language called *bytecode* -- the target language of Java compilers. Files written in bytecode are then used for dynamic code distribution of Java programs over the Internet, most notably as applets embedded in web pages. Of course in order to execute these programs, the client computers must be equipped with suitable JVM implementations. These programs, also called *Java Run-time Environments* (*JRE*s), are widely available for numerous processor / OS combinations, including game consoles and cellphones.

In the early 2000's, Microsoft entered the fray with its ".NET" infrastructure. The centerpiece of .NET is a virtual machine model called CLR (*Common Language Runtime*). According to the Microsoft vision, many programming languages (including C++, C#, Visual Basic, and J# -- a Java variant) could be compiled into intermediate code running on the CLR. This enables code written in different languages to inter-operate and share the software libraries of a common run-time environment.

We note in closing that a crucial ingredient that must be added to the virtual machine model before its full potential of inter-operability is unleashed is a common software library. Indeed the Java virtual machine comes with the *standard Java libraries,* and the Microsoft virtual machine comes with the *Common Language Runtime*. These software libraries can be viewed as small operating systems, providing the languages that run on top of the VM with unified services like memory management, GUI utilities, string functions, math functions, and so on. One such library will be described and built in chapter 12.

## 7.5 Project

This section describes how to build the VM translator described in the chapter. In the next chapter we will extend this basic translator with additional functionality, leading to a full-scale VM implementation. Before you get started, two comments are in order. First, section 7.2.6 (*VM Programming Examples*) is irrelevant to this project. Second, since the VM translator is designed to generate Hack assembly code, it is recommended to refresh your memory about the Hack assembly language rules (Chapter 4).

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

**Resources:** You will need two tools: the programming language in which you will implement your VM translator, and the *CPU Emulator* supplied with the book. This emulator will allow you to execute the machine code generated by your VM translator -- an indirect way to test the correctness of the latter. Another tool that may come in handy in this project is the visual *VM Emulator* supplied with the book. This program allows experimenting with a working VM implementation before you set out to build one yourself. For more information about this tool, refer to the *VM Emulator Tutorial*.

**Contract:** Write a VM-to-Hack translator, conforming to the *VM Specification, Part I* (Sections 7.2.2 and 7.2.3) and to the *Standard VM-on-Hack Mapping, Part I* (Section 7.3.1). Use it to translate the test VM programs supplied below, yielding corresponding programs written in the Hack assembly language. When executed on the supplied CPU emulator, the assembly programs generated by your translator should deliver the results mandated by the supplied test scripts and compare files.

### Proposed Implementation Stages

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

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

**Stage II: Memory access commands:** The next version of your translator should include a full implementation of the VM language's `push` and `pop` commands, handling all eight memory segments. We suggest breaking this stage into the following sub-stages:

0. You have already handled the `constant` segment;
1. Next, handle the segments `local`, `argument`, `this`, and `that`;
2. Next, handle the `pointer` and `temp` segments, in particular allowing modification of the bases of the `this` and `that` segments;

3. Finally, handle the `static` segment.

## Test Programs

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

**Stage I: Stack Arithmetic:**

- `SimpleAdd`: Pushes and adds two constants;
- `StackTest`: Executes a sequence of arithmetic and logical operations on the stack.

**Stage II: Memory Access:**

- `BasicTest`: Executes `pop` and `push` operations using the virtual memory segments.
- `PointerTest`: Executes `pop` and `push` operations using the `pointer`, `this`, and `that` segments.
- `StaticTest`: Executes `pop` and `push` operations using the `static` segment.

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

## Tips

**Initialization:** In order for any translated VM program to start running, it must include a preamble startup code that forces the VM implementation to start executing it on the host platform. In addition, in order for any VM code to operate properly, the VM implementation must store the base addresses of the virtual segments in the correct locations in the host RAM. Both issues -- startup code and segments initializations -- are implemented in the next project. The difficulty of course is that we need these initializations in place in order to run the test programs given in this project. The good news is that you should not worry about these issues at all, since the supplied test scripts carry out all the necessary initializations in a manual fashion (for the purpose of this project only).

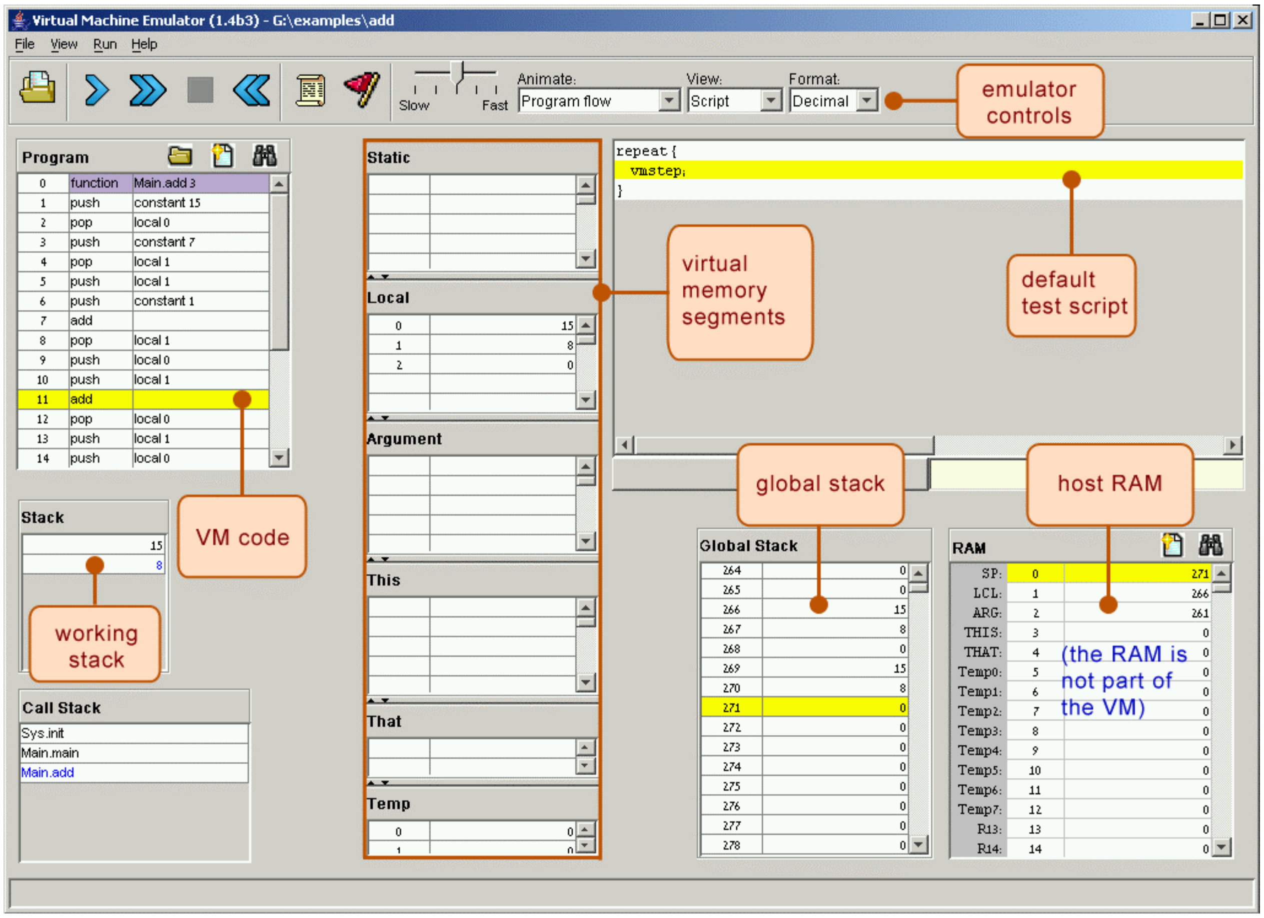
**Testing/debugging:** For each one of the five test programs, follow these steps:

1. Run the `Xxx.vm` program on the supplied VM emulator, using the `XxxVME.tst` test script, to get acquainted with the intended program's behavior.
2. Use your partial translator to translate the `.vm` file. The result should be a text file containing a translated `.asm` program, written in the Hack assembly language.
3. Inspect the translated `.asm` program. If there are visible syntax (or any other) errors, debug and fix your translator.
4. Use the supplied `.tst` and `.cmp` files to run your translated `.asm` program on the CPU emulator. If there are run-time errors, debug and fix your translator.

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

## Tools

**The VM Emulator:** The book's software includes a Java-based VM implementation. This VM Emulator was built for one purpose: illustrating how the VM works, using visual GUI and animation effects. Specifically, it allows executing VM programs directly, without having to translate them first into machine language. This practice enables experimentation with the VM environment before you set out to implement one yourself.  Here is a typical screen shot of the VM Emulator in action:



Animate: Program flow   View: Script   Format: Decimal

```
0   function  Main.add 3
1   push      constant 15
2   pop       local 0
3   push      constant 7
4   pop       local 1
5   push      local 1
6   push      constant 1
7   add
8   pop       local 1
9   push      local 0
10  push      local 1
11  add
12  pop       local 0
13  push      local 1
14  push      local 0
```

```
repeat {
  vmstep;
}
```

Stack: 15, 8

Call Stack: Sys.init, Main.main, Main.add

| Local | |
|---|---|
| 0 | 15 |
| 1 | 8 |
| 2 | 0 |

| Temp | |
|---|---|
| 0 | 0 |
| 1 | 0 |

| Global Stack | |
|---|---|
| 264 | 0 |
| 265 | 0 |
| 266 | 15 |
| 267 | 8 |
| 268 | 0 |
| 269 | 15 |
| 270 | 8 |
| 271 | 0 |
| 272 | 0 |
| 273 | 0 |
| 274 | 0 |
| 275 | 0 |
| 276 | 0 |
| 277 | 0 |
| 278 | 0 |

| RAM | | |
|---|---|---|
| SP: | 0 | 271 |
| LCL: | 1 | 266 |
| ARG: | 2 | 261 |
| THIS: | 3 | 0 |
| THAT: | 4 | 0 |
| Temp0: | 5 | 0 |
| Temp1: | 6 | 0 |
| Temp2: | 7 | 0 |
| Temp3: | 8 | 0 |
| Temp4: | 9 | 0 |
| Temp5: | 10 | 0 |
| Temp6: | 11 | 0 |
| Temp7: | 12 | 0 |
| R13: | 13 | 0 |
| R14: | 14 | 0 |

**FIGURE 7.15: The VM Emulator** supplied with the book.