

6. The Virtual Machine I: Stack Arithmetic¹

*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 in the book. High-level programs will be first translated into an intermediate code (Chapters 9-10), and the intermediate code will then be translated into machine language (Chapters 6-7). This two-tier translation model is a rather old idea that recently made a significant comeback following its adoption by modern programming 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* (VM) -- an abstract computer that does not exist for real. There are many reasons why this idea makes sense, one of which being code transportability, as the VM may be implemented with relative ease on target platforms. In particular the VM may be implemented by interpreters, by special purpose hardware platforms, or, as we do here, by translating the VM programs into the machine languages of target hardware platforms. The software suite that comes with the book illustrates yet another implementation vehicle: a program that emulates the VM using an interpreter running on a standard personal computer.

The VM that we will build in chapters 6-7 is a simple virtual machine, modeled after the Java JVM paradigm. In each chapter we specify parts of the machine and then proceed to implement them on the Hack platform. In particular, in this chapter we will build a basic VM translator, capable of translating individual VM commands into Hack code, and in chapter 7 we will add program flow, multi-method, and multi-class functionality. The result will be a full-scale VM that will serve as the backend of the compiler that we will build in chapters 9-10.

The virtual machine that we are about to build illustrates several 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. It has many practical implications, e.g. using an emulator of an old generation computer platform running on a new one in order to achieve upward code compatibility. Second, the VM that we will build is based on *stack processing*. The *stack* is a fundamental data structure that comes to play in many computer systems and algorithms, and in this chapter we will illustrate it in action and then implement it in our VM translator.

¹ From *The Elements of Computing Systems*, Nisan & Schocken, MIT Press, forthcoming in 2003, www.idc.ac.il/csd

6.1 Background

The Virtual Machine Paradigm

The compilation of a high-level language into a machine language is a rather complex process. Normally, a separate compiler is written specifically for any given pair of high-level and machine level languages, and thus the compiler depends on every detail of both languages. It turns out however that for the most part, the compilation process can be broken into two nearly separate stages: in the first stage the high-level program is parsed and its commands are broken into “primitive” steps, and in the second stage these primitive steps are actually implemented in 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 these two stages 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 “primitive steps” into which high-level commands are decomposed. The compiler that was formerly a single program is now split into two separate programs. The first program, still termed *compiler*, translates the high-level language into intermediate VM code targeted for an abstract virtual machine, while the second program translates from the VM code to the machine language of the target platform.

The approach of formally defining a virtual machine language has been used in many cases of which the most famous ones are the *p-code* generated by some Pascal compilers in the 1970s, and more recently the *bytecode* generated by Java compilers. The notion of an explicit virtual machine language has several practical advantages. First, compilers for various platforms can be obtained with relative ease by replacing only the virtual machine implementation. This, in turn, allows the VM code to become transportable across different hardware platforms. This versatility is well demonstrated by the Java virtual machine (JVM), widely used for transporting Java code all over the Internet and running it on numerous clients, operating systems, and browsers. This “*write once, run everywhere*” principle works since the virtual machine itself is implemented in numerous emulators, interpreters, dedicated hardware, translators, and incremental compilers, allowing a range of tradeoffs between efficiency, memory, hardware cost, and programming effort. Some of this flexibility is illustrated in Fig. 6-1.

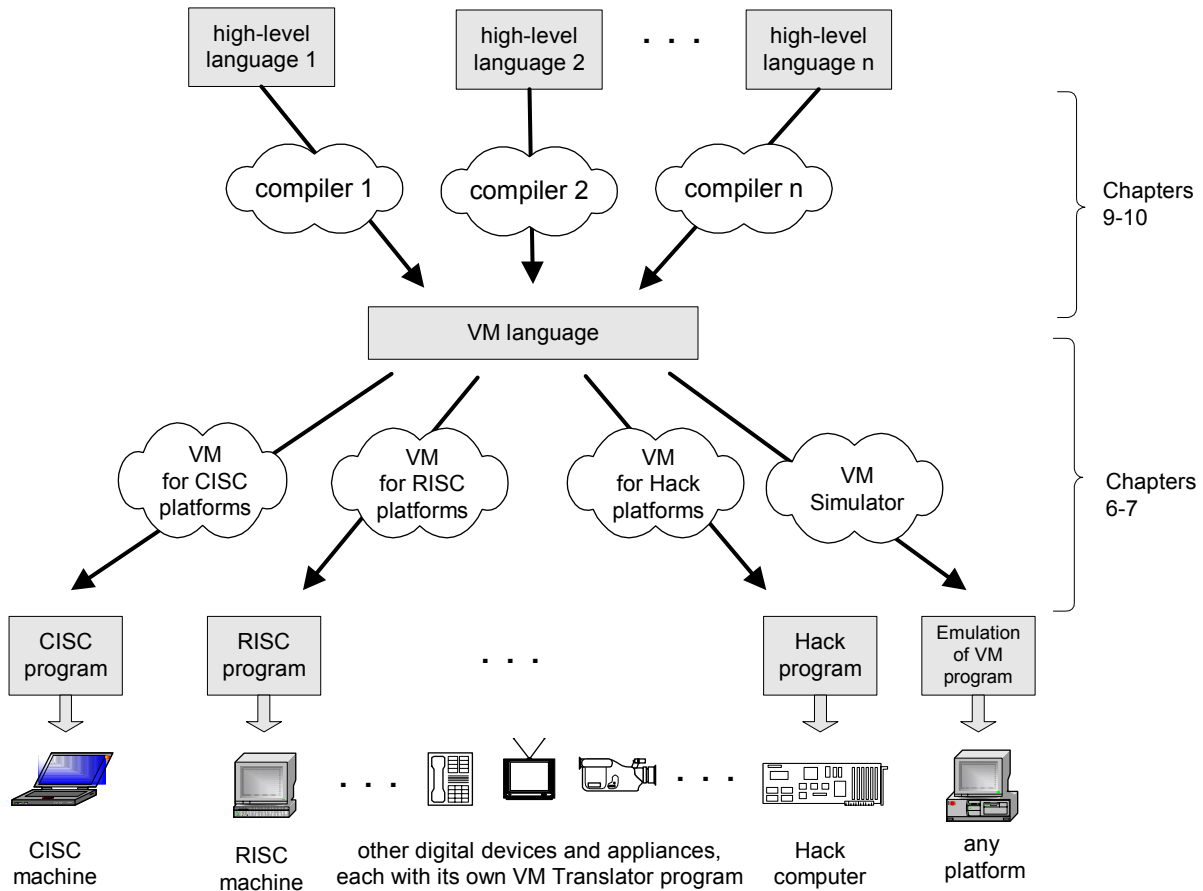


FIGURE 6-1: The virtual machine paradigm. Once a high-level program is compiled into VM code, the program can run on any hardware platform equipped with a suitable translator and run-time environment. In this book we will build the *VM for the Hack Platform* program, and use a VM emulator like the one depicted on the right.

The Stack Machine Model

There are several possible paradigms on which to base a virtual machine architecture. Perhaps the cleanest and most popular one is the *stack machine* model. This model is used in the *Java Virtual Machine* as well in the virtual machine described here. As we will see shortly, the *stack* is the centerpiece of the VM architecture -- the place where almost all the action takes place. Some VM commands remove data items from the stack, perform various operations on them, and put the resulting values onto the stack's top. Other commands transfer data items from the stack's top to designated memory locations, and vice versa. Taken together, these stack operations can implement a complete virtual machine, as powerful as any computer system.

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. This basic anatomy is illustrated in Fig. 6-2.

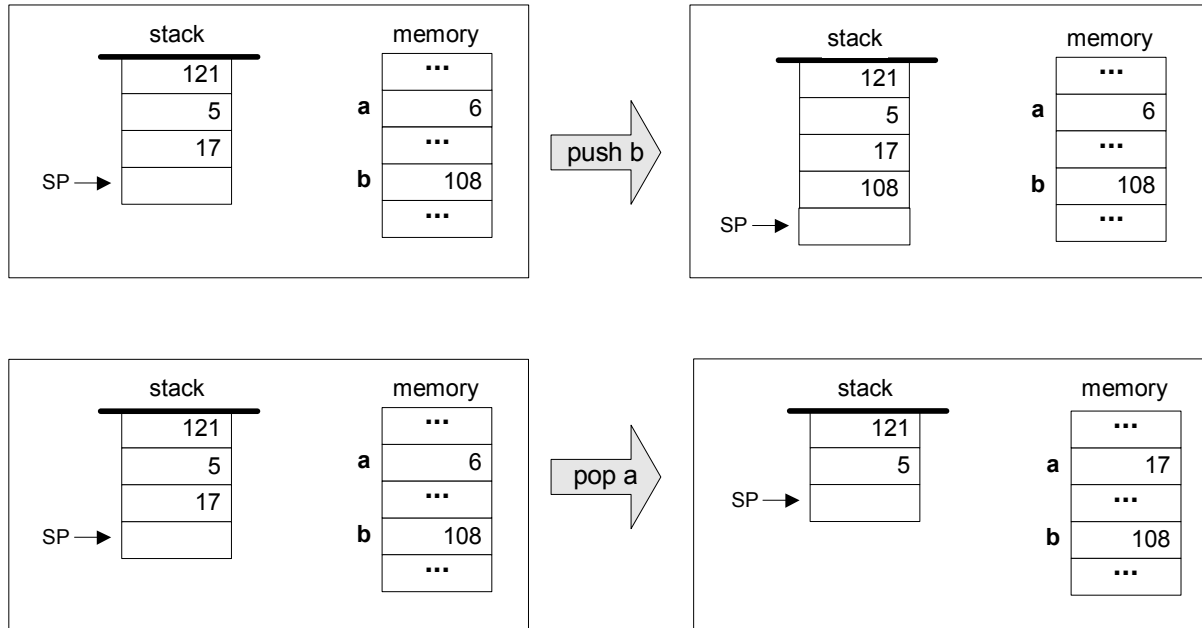


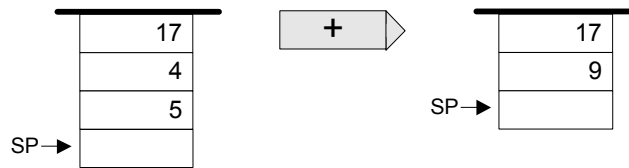
FIGURE 6-2: Stack processing example, illustrating the two elementary operations *push* and *pop*. Following convention, the stack is drawn upside down, as if it grows downward. The location just after the top position is always referred to by a special pointer called *sp*, or *stack pointer*. The labels *a* and *b* refer to two arbitrary memory addresses. The stack’s “before state” and “after state” (with respect to each operation) are shown on the left and right sides, respectively.

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 erases the location’s previous value.

Note that a stack can be easily implemented in hardware or software. Various approaches are possible, of which the simplest 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++`). The *pop* operation is implemented by first decrementing *sp* and then returning the value stored in the top position (i.e. `sp--; return stack[sp]`).

As usual in computer science, simplicity and elegance imply power of expression. The simple stack model is an extremely useful data structure that comes to play in many computer systems and algorithms. In the virtual machine architecture it serves two main purposes. First, it is used for handling all the arithmetic and logical operations of the VM. Second, it facilitates function calls and dynamic memory allocation -- the subjects of the next chapter.

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



It turns out that every arithmetic expression -- no matter how complex -- can be easily converted into, and evaluated by, a sequence of simple operations on a stack. For example, consider the expression $d = (6 - 4) * (8 + 1)$, taken from some high-level program. The stack-based evaluation of this expression is shown in Fig. 6-3.

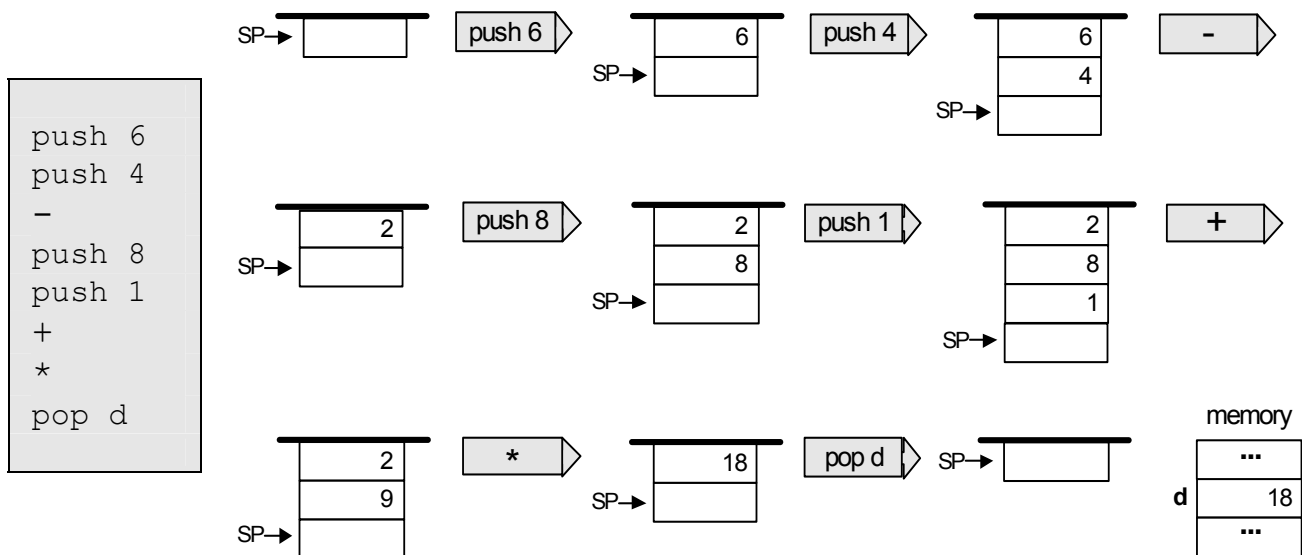


FIGURE 6-3: Stack-based evaluation of the expression “let d = (6-4) * (8+1)”

In a similar fashion, every Boolean expression can also be converted into, and evaluated by, a sequence of simple stack operations. For example, consider the high-level command “if (x<7) or (y=8) then ...”. The stack-based evaluation of this expression is shown in Fig. 6-4.

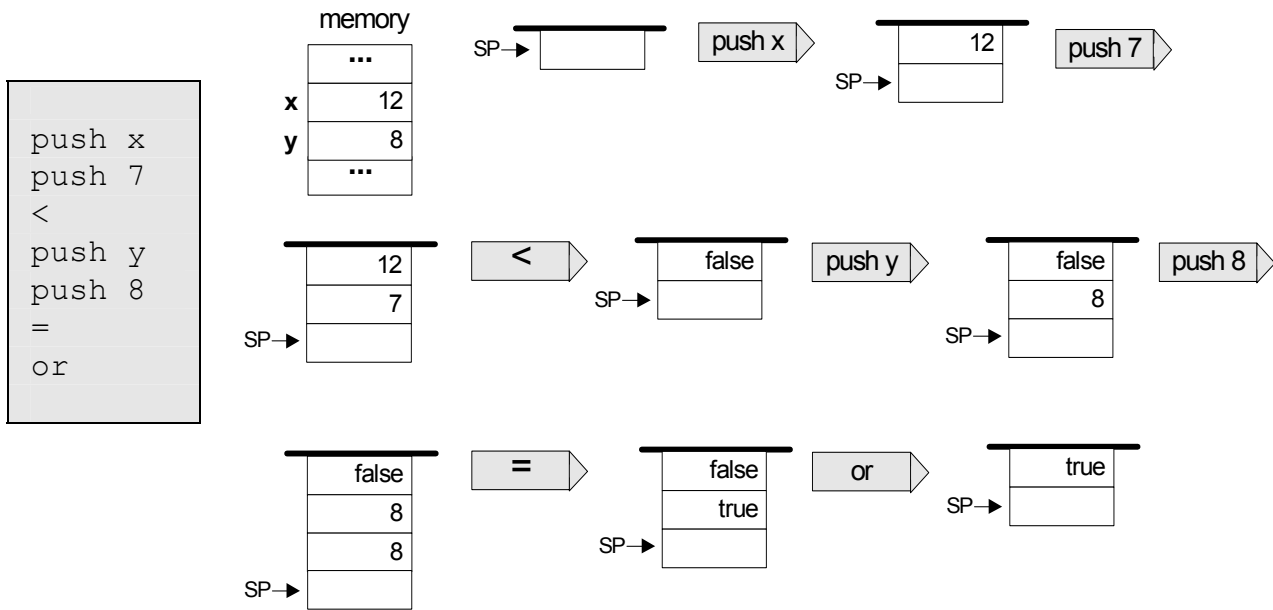


FIGURE 6-4: Stack-based evaluation of the expression “if (x < 7) or (y = 8) then ...”

To sum up, the above examples are manifestations of a general observation: any arithmetic and Boolean expression can be transformed into a series of elementary stack operations that compute its value. Further, this syntactic transformation can be described systematically, and thus it can be implemented in a computer program. In particular, as we will see in Chapter 9, one can write a *compiler* program that translates high-level arithmetic and Boolean expressions into VM commands. In this chapter we are not interested in the compilation *process*, but rather in its *results* – i.e. the VM commands that it generates. We now turn to specify these commands (section 6.2), illustrate them in action (section 6.2.6), and describe their implementation on the Hack platform (section 6.3).

6.2 Specification

6.2.1 General

The virtual machine is *stack-based*: all operations are done on a stack. It is also *function-based*: a complete VM program is composed of a collection of 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 Boolean 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 yet modular 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 that implements them only. The next chapter specifies the program flow and function calling commands, and extends the basic VM to a full-blown virtual machine implementation.

Program and command structure: A VM *program* is a collection of one or more *files* with a `.vm` extension, each consisting of one or more *functions*. From a compilation standpoint, these objects correspond, respectively, to the notions of *program*, *class*, and *method* in a Java-like language.

Within a `.vm` file, each VM command appears in a separate line, and in one of the following formats: `<command>`, `<command arg>`, or `<command arg1 arg2>`, where 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.

6.2.2 Arithmetic commands

The VM language features nine stack-oriented arithmetic 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 function's result, without affecting the rest of the stack. Table 6-5 gives the details.

Command	Return value (after popping the operand/s)	Comment
<code>add</code>	$x+y$	integer addition (2's complement)
<code>sub</code>	$x-y$	integer subtraction (2's complement)
<code>neg</code>	$-y$	negation (2's complement)
<code>eq</code>	true if $x=y$ and false otherwise	equality
<code>gt</code>	true if $x>y$ and false otherwise	greater than
<code>lt</code>	true if $x<y$ and false otherwise	less than
<code>and</code>	x and y	bit-wise
<code>or</code>	x or y	bit-wise
<code>not</code>	not y	bit-wise

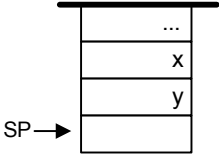


TABLE 6-5: Arithmetic and Boolean stack commands. Throughout the specification, y refers to the item at the top of the stack and x refers to the item just below it.

Three of the commands listed in Table 6-5 (`eq`, `gt`, `lt`) operate on, and return, Boolean values. The VM represents *true* and *false* as `0xFFFF` and `0x0000`, respectively.

Example: We now turn to illustrate all the VM arithmetic commands in action. We will apply each command to an arbitrary stack, and inspect the stack's state before and after each operation. We show only the three top-most cells in the stack, noting that the rest of the stack is never effected by the current command.

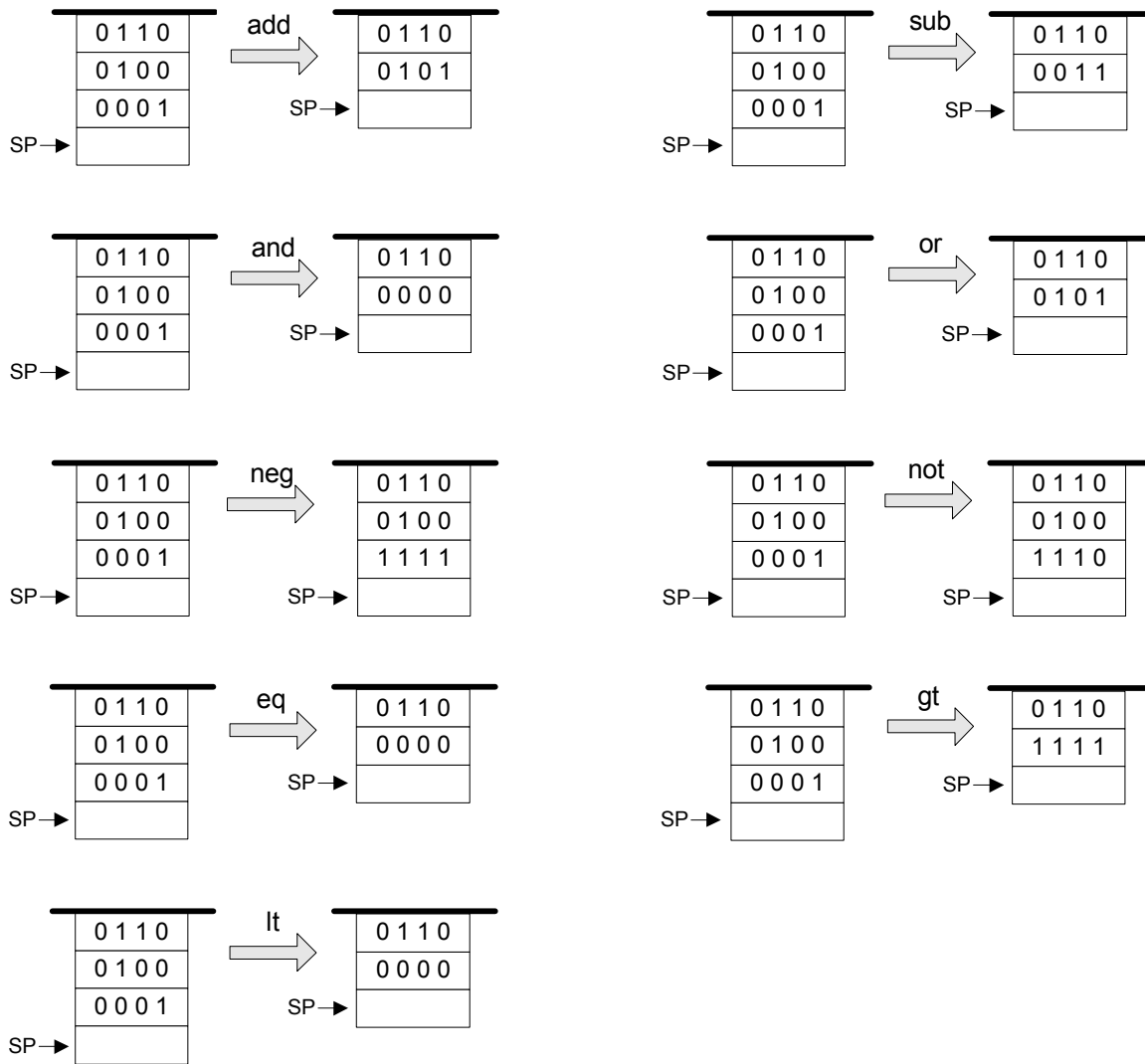


FIGURE 6-6: Arithmetic commands examples. Each operation is represented by an arrow that transforms the stack to a new state. To minimize clutter, we use a 4-bit stack. In a 16-bit stack, true and false are represented by the constants $0xFFFF$ and $0x0000$, respectively. Note that neg effects 2's complement negation, and that the commands and/or/not effect bit-wise operations.

6.2.3 Memory Access Commands

Unlike real computer architectures, where the term “memory” refers to a collection of physical storage devices, virtual machines manipulate virtual memory segments. In particular, the memory environment of each VM function consists of eight memory segments, described below. In addition, two implicit memory objects exist “in the background” but are not explicit at the VM level: the *stack* and the *heap*. The stack is the working memory of the VM function, and the heap is the RAM area where the objects and the arrays declared by the high-level program are stored. This section describes all these memory objects and the commands used to access them.

Memory Segments: Each VM function sees the eight memory segments described in Table 6-7.

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 when the function is entered.
Static	stores static variables shared by all functions in the same <code>.vm</code> file	Allocated by the VM implementation for each file; Seen by all functions in the 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 Temp	“ <i>T-segments</i> .” general-purpose segments that can be made to correspond to different areas in the heap; Serve various programming needs	Any VM function can bind any T-segment to any area on the heap by setting the segment’s base. The setting of the base is done through the <code>Pointer</code> segment.
Pointer	A 3-entry segment that holds the bases of the three T-segments: locations 0,1, and 2 store the bases of the <code>This</code> , <code>That</code> , and <code>Temp</code> segments, respectively	Can be set by the VM program. Used to bind the T-segments to various areas on the heap (e.g. for object- and array-processing).

TABLE 6-7: The memory segments seen by every VM function.

We see that five of the T-segments have a fixed purpose, and are controlled by the VM implementation. The remaining three T-segments are general purpose and their mapping on main memory is controlled by the current VM program.

Memory access commands: There are two memory access 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.

Example: As mentioned above, a VM program is a collection of one or more VM *functions*, which are similar to the high-level notion of *methods*, or *subroutines*. Each VM function can manipulate eight memory segments, three of which are depicted in Fig. 6-8.

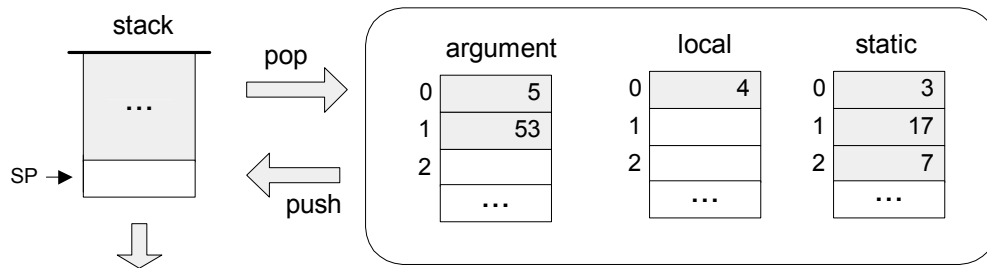


FIGURE 6-8: Some virtual memory segments. We assume (arbitrarily) that the current function has 2 arguments and 1 local variable, and that there are 3 static variables

Suppose we wish to carry out the following operations: set the function's first local variable to the value of its second argument; then set the second static variable to the same value. In the VM world this logic is done as follows:

```
push argument 1
pop local 0
push local 0
pop static 1
```

The stack: As the above example illustrates, the presence of the stack is implicit at the VM level. Unlike the eight memory segments, which are manipulated directly, the stack itself is affected in the background, as a side effect of the explicit segment manipulation commands. In addition, although every memory access involves the stack, individual stack elements cannot be accessed by VM commands, except for the topmost element.

The heap: high-level programs typically involve the manipulation of high-level data structures like *objects* and *arrays*. The allocation of memory space to the data structures declared by the high-level program is handled by the operating system, as we will see in Chapter 11. In particular, objects and arrays are stored in a RAM segment called *heap*. Unlike the virtual memory segments, the heap is implicit at the VM level, and thus objects and arrays cannot be manipulated explicitly. Instead, VM commands manipulate objects and arrays indirectly, using the Pointer and the T-segments, as we will see shortly.

The documentation of each VM implementation must specify a constant `HEAP` that specifies the beginning of the heap in the RAM. This constant will be used by the operating system code for memory allocation.

6.2.4 Program flow commands

The VM features three program flow commands:

- `label symbol` // label declaration
- `goto symbol` // unconditional branching
- `if-goto symbol` // conditional branching

These commands are discussed in chapter 7.

6.2.5 Function calling commands

The VM features three program flow commands:

- `function funcioName nLocals` // function declaration; must include
// the number of the function's local variables
- `call functionName nArgs` // function invocation; must include
// the number of the function's local variables
- `return` // transfer control back to the calling function

Where *funcioName* is a symbol and *nLocals* and *nArgs* are non-negative integers. These commands are discussed in chapter 7, and are given above in for completeness.

6.2.6 Examples

We now turn to illustrate the VM architecture, language, and programming style in action. We give three examples: (i) a typical programming task (multiplication), (ii) object fields processing, and (iii) accessing array elements. We wish to point out at the outset that VM programs are rarely written by human programmers, but rather by compilers. Therefore, it is instructive to begin each example with a high-level version of the program, and then track down the process of translating it into VM code. We use a C-style syntax for all the high-level examples.

Multiplication program

Consider the multiplication program shown in the left side of Prog. 6-9. How should we (or more likely, the compiler) write this function in the VM language? First, we note that in the VM world, we must think in terms of arguments, variables, elementary arithmetic, Boolean and assignment operations, and simple "goto logic." The algorithmic steps that result from this limited repertoire are depicted in the middle of Prog. 6-9. Next, we must express the algorithm in a stack-oriented formalism, using VM commands. However, instead of translating directly into the VM language, it is instructive to first use a symbolic version of the language, leading to the pseudo-code on the right of Prog. 6-9. (The exact semantics of the commands `function`, `label`, `goto`, `if-goto`, and `return` are described in chapter 7, but their intuitive meaning should be clear.)

High-Level Code (C style)

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

Primitive Steps

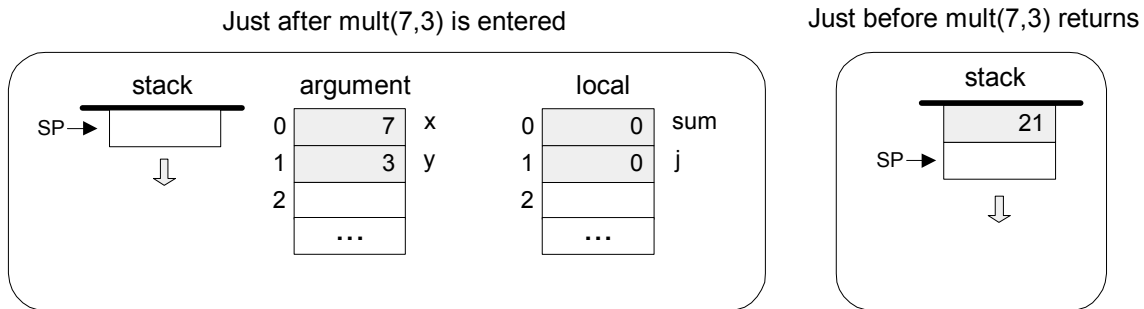
```
function: mult
    args: x,y
    vars: sum,j
    // the code
    sum=0
    j=y
loop:
    if j==0 goto end
    sum=sum+x
    j=j-1
    goto loop
end:
    return sum
```

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
```

PROGRAM 6-9: VM programming: From high-level code to pseudo VM code.

We now turn to illustrate the handling of arguments and variables at the VM level. 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 at the top of the argument segment, and (iii) the local variables that it is supposed to use are initialized to 0 and located at the top of the local segment. Recall however that VM commands are not allowed to use symbolic argument and variable names -- they are limited to making only *<segment index>* references. Therefore, the translation from the pseudo VM code to the final VM code is straightforward. First, we represent *x*, *y*, *sum* and *j* as argument 0, argument 1, local 0 and local 1, respectively. Next, we replace every occurrence of *x*, *y*, *sum* and *j* in the pseudo code with their corresponding *<segment index>* references. The details are shown in Program 6-10.



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

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
```

VM code

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

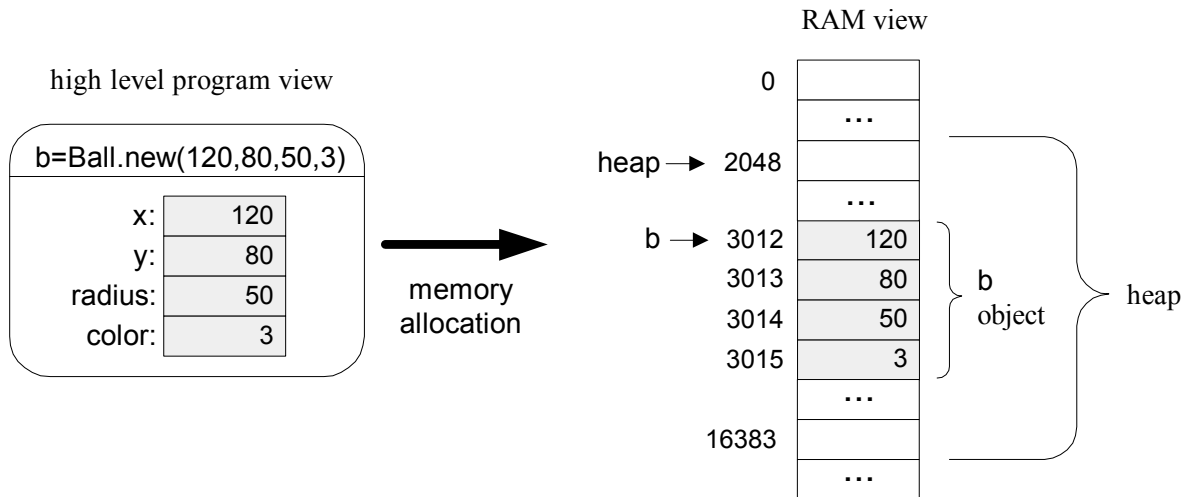
PROGRAM 6-10: VM programming: From pseudo VM code to final VM code. The translation assumes the argument and variable mappings depicted at the top of the figure.

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 building this world for *every* VM function just before it starts running is the VM implementation, as we will see in the next chapter.

Object Fields Manipulation

High-level object-oriented programming languages are designed to handle complex variables called *objects*. Technically speaking, an object is a bundle of variables (also called *fields*, or *properties*) that can be treated as one entity, e.g. passed to a method as a single parameter. For example, consider an animation program designed to juggle 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 objects, it will end up on the *heap*. The *heap* is a designated RAM segment containing serialized representations of all the objects and arrays of the currently running program. In particular, whenever the program creates a new object using a high-level command like `b=Ball.new(...)`, the compiler computes the object's size (in terms of words) and the operating system finds and allocates enough RAM locations to store it in the heap. The details of memory allocation and de-allocation will be discussed later in the book. For now, let us assume that our `b` object has been allocated RAM addresses 3012 to 3015 on the heap, as shown in Prog. 6-11. Suppose now that a certain function 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 function and its VM translation are given in Prog. 6-11.



High-level code

```
function resize (Ball b,int r) {
  ...
  let b.radius=r;
  ...
}
```

VM code

```
...
push argument 0 // get b's base address
pop pointer 0 // set base of This segment
push argument 1 // get r's value
pop this 2 // set b's third field to r
...
```

PROGRAM 6-11: VM-based object field access, assuming the memory allocation shown at the top. (The heap location in the RAM is fixed by the VM implementation)

Recall that the name of the object, which is `b`, is actually a reference to a memory cell containing the address 3012 (see Prog. 6-11). Since `b` is the first argument passed to the high-level `resize` function, the compiler will treat it as the 0th argument of the translated VM function. Hence, when we set `pointer 0` to the value of this argument, we are effectively setting the base of the VM's `This` segment to address 3012. From this point on, VM commands can access any field in the RAM-resident `b` object using the virtual memory segment `This`, without ever worrying about the physical address of the actual object. The internal details are shown in Fig. 6-12, assuming the function call `resize(b, 17)`.

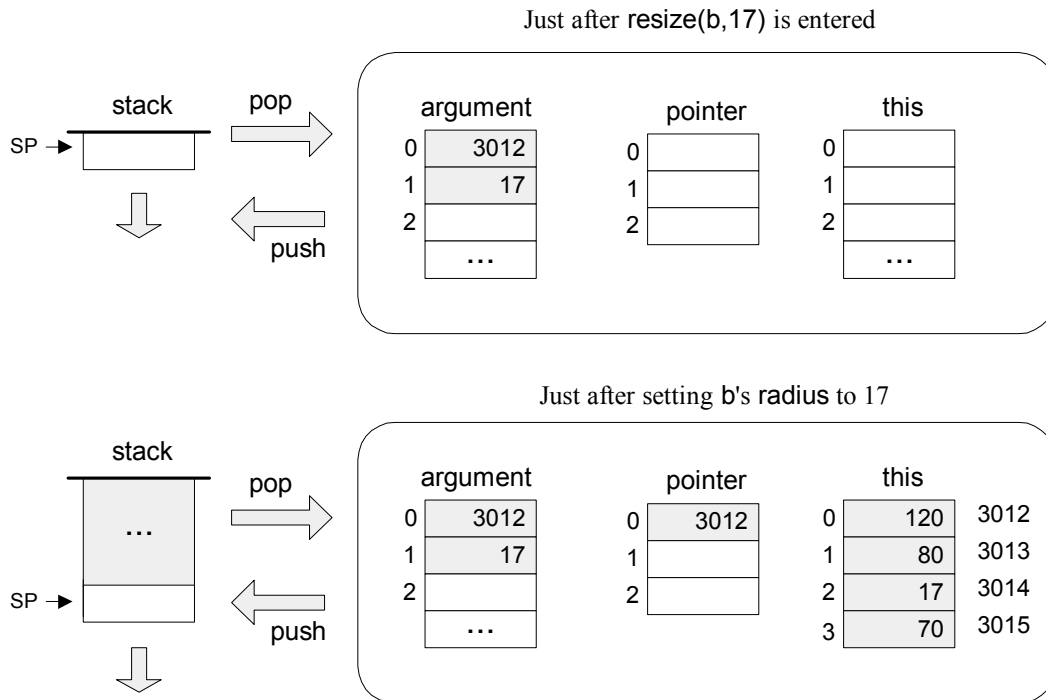
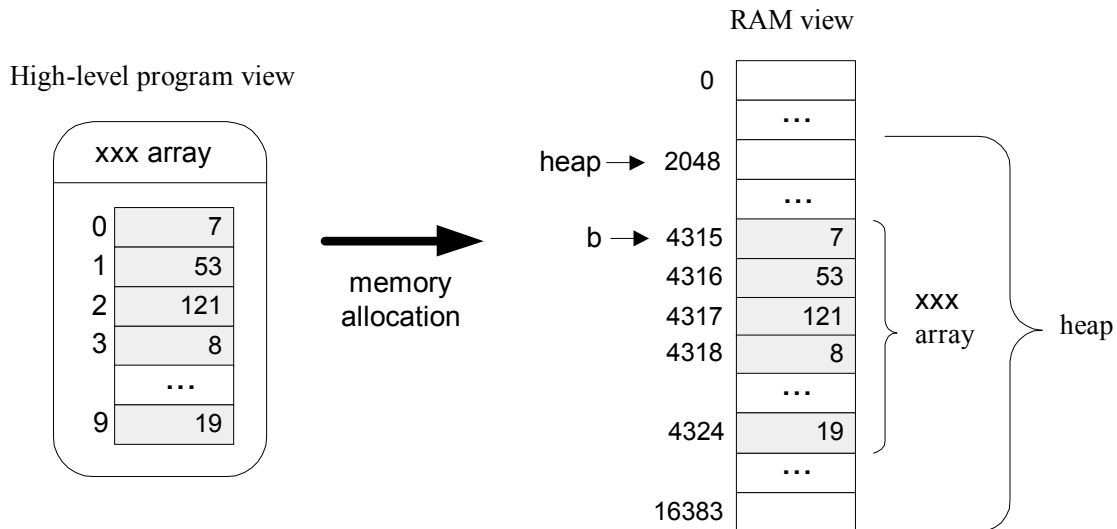


FIGURE 6-12: Example: Setting and using the This segment, assuming the memory allocation depicted in Prog. 6-11. The labels at the bottom right (3012, ...) are not part of the VM state, and are given here for ease of reference.

Array Manipulation

An array is an indexed vector of objects of the same type. Suppose that a high-level program has created an array of integers called `xxx`, and proceeded to fill it with some 10 constants. Let us assume that the array's base has been mapped on RAM address 4315 in the heap. Suppose now that a certain method in the high-level program, say `bar`, takes an array as a parameter, and, among other things, sets its k -th element to 34, where k is one of the method's local variables. Prog. 6-13 gives the details.



High-level code

```
method bar (Array xxx, ...) {
  var int i, j, k;
  let k=3;
  let xxx[k]=34;
  ...
}
```

VM code

```
...
push constant 3 // set k=2
pop local 2
push argument 0 // get xxx's base address
push local 2 // get k
add
pop pointer 1 // set That to (xxx+k)
push constant 34
pop that 0 // *(xxx+k)=34
...
```

PROGRAM 6-13: VM-based array manipulation,
assuming the memory allocation shown at the top.

Note that in the C language, the command `xxx[k]=34` can be also expressed as `*(xxx+k)=34`, which reads “set the RAM location whose address is $(xxx+k)$ to 34”. This is precisely what the VM code is doing, using primitive VM commands. The internal details are shown in Fig. 6-14.

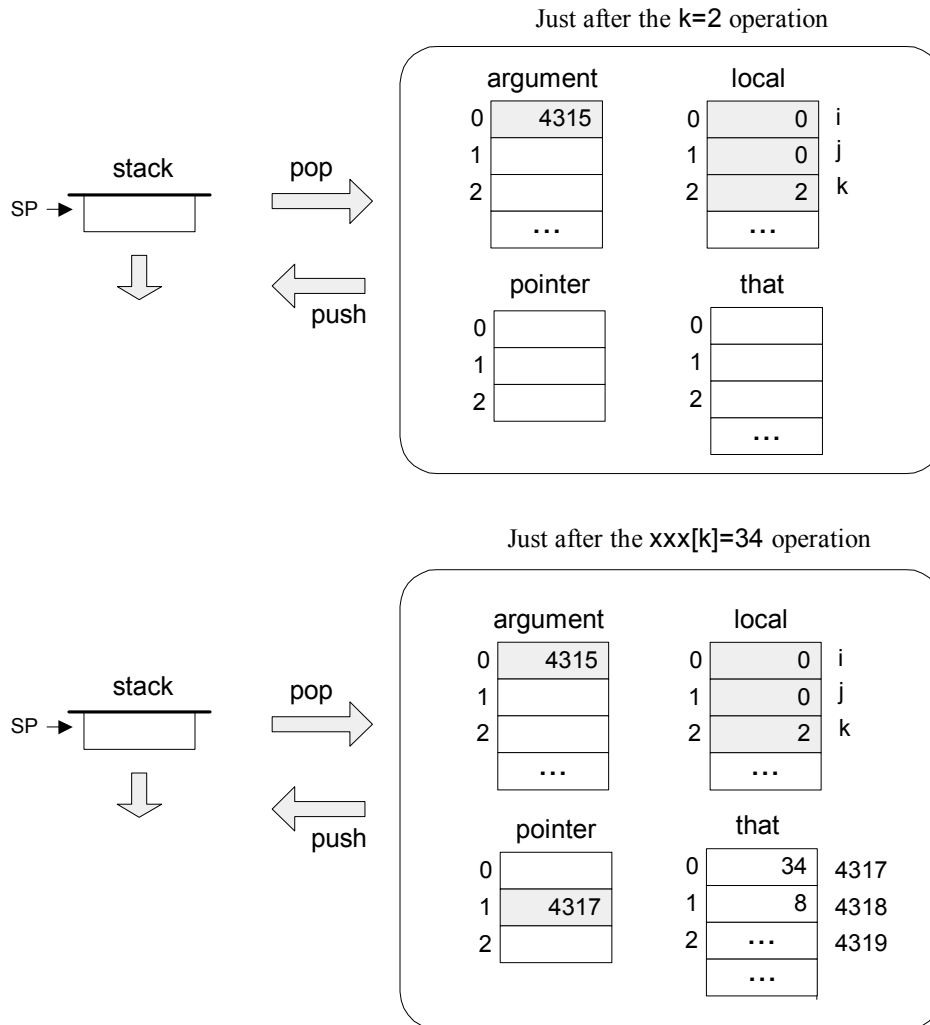


FIGURE 6-14: Using the THAT segment. The symbolic and numeric labels shown in the right are not part of the VM state, and are given here for ease of reference.

6.3 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 a VM implementation consists of two conceptual tasks. First, we have to emulate the VM world on the target hardware. 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 assembly instructions that effect the command on the target platform.

This section describes how to implement the VM specified in Section 6.2 on the Hack platform specified in Section 4.2. We start by defining a “standard” mapping from VM elements to the Hack hardware. 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.

6.3.1 Standard Mapping on the Hack Platform

If you will re-read the virtual machine specification given in Section 6.2, 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, 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. As it turns out however, it is always recommended to provide guidelines on how the VM should map on the target platform, rather than leaving these decisions to the programmer's discretion. These guidelines, called *standard mapping*, are provided for two reasons. First, we wish the VM implementation to support inter-operability with other high-level languages implemented over the target platform. 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, we now turn to specify the standard mapping of the VM on one 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. 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 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 and an unused segment. The VM implementation should use this space as follows:

<i>RAM addresses</i>	<i>Usage</i>
0–15:	16 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
16384–24575:	Memory mapped I/O

TABLE 6-15: Standard VM implementation on the Hack RAM.

Registers: The *Hack Assembly Language Specification* (Section 4.2.5) contains the following convention: RAM addresses 0 to 15 can be referred to by all assembly programs using the symbols `R0` to `R15`, respectively. In addition, the specification states that all assembly programs can refer to RAM addresses 0 to 5 (i.e. `R0` to `R5`) using the symbols `SP`, `LCL`, `ARG`, `THIS`, `THAT`, and `TEMP`. This convention was introduced into the Hack assembly by the language designers in order to help programmers write readable VM translators. In particular, the expected use of these registers is described in Table 6-16.

<i>Register</i>	<i>Location</i>	<i>Usage</i>
<code>SP</code>	<code>RAM[0]</code>	Points to the next topmost location in the stack.
<code>LCL</code>	<code>RAM[1]</code>	Points to the base of the current function's <code>local</code> segment.
<code>ARG</code>	<code>RAM[2]</code>	Points to the base of the current function's <code>argument</code> segment.
<code>THIS</code>	<code>RAM[3]</code>	Points to the base of the current <code>this</code> segment (within the heap).
<code>THAT</code>	<code>RAM[4]</code>	Points to the base of the current <code>that</code> segment (within the heap).
<code>TEMP</code>	<code>RAM[5]</code>	Points to the base of the current <code>temp</code> segment (within the heap).

TABLE 6-16: Standard mapping of the segment base pointers on the Hack RAM and related symbols in the Hack assembly language. The two left columns describe a convention in the Hack assembly language; the right column specifies its intended use in VM implementations.

In principle, the VM implementation programmer can use any RAM locations to store the various pointers needed by the VM. However, the standard mapping requires using registers `R0-R5` and referring to them using their symbolic names for two important reasons: code readability and testability. Consider for example the operation `SP--`. Programmers who will follow Table 6-16 will write this in Hack as `@SP, M=M-1`, which is clearly more readable than, say, `@R12, M=M-1`. Second, when testing the VM code that VM translators generate, our test scripts assume that the VM pointers are stored in `RAM[0]` to `RAM[5]`. To sum up, we recommend using `R0-R5` as prescribed in Table 6-16, and, while we are at it, use their symbolic names rather than anonymous “R” names.

The next ten virtual registers of the Hack assembly language (`R6` to `R15`) can be used by the VM implementation at will, as general-purpose registers.

Memory Segments Mapping

Recall that any VM function sees eight virtual memory segments. Five segments map directly on the Hack RAM, while the other three are implemented using various assembly tricks.

Local, argument, this, that, temp: 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`, `THAT`, and `TEMP`, respectively). Thus any access to the i 'th location in any one of these segments can be translated to assembly code that accesses address ($base+i$) in the RAM, where $base$ is the value stored in the register dedicated to the respective segment.

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: Recall that according to the Hack assembly language specification, whenever a new symbol is mentioned for the first time in a Hack 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 assembly code `@Xxx.3, D=M`, followed by additional assembly code that pushes *D*'s value to the stack.

Pointer: This segment, which contains a fixed number of three entries, is mapped directly onto the three registers `this`, `that`, and `temp`. Thus each VM command involving the `pointer` segment will be translated into assembly code that manipulates one of these registers, according to the segment index.

Assembly Language Symbols

To recap, Table 6-16 summarizes all the special symbols that should be used in VM implementations that conform to the standard mapping.

<i>Symbol</i>	<i>Usage</i>
SP, LCL, ARG, THIS, THAT, TEMP	These built-in labels maintain the bases addresses of the stack and the virtual segments local, argument, this, that, and temp.
R6-R15	Can be used for any purpose.
“ <i>f.j</i> ” symbols	Each static variable <i>j</i> in file <i>f.vm</i> is mapped to the symbol <i>f.j</i> (and is thus automatically allocated a RAM location by the assembler.)
flow of control symbols (labels)	The VM commands <code>function</code> , <code>call</code> , and <code>label</code> are handled by generating symbolic labels, as described chapter 7.

TABLE 6-16: Special assembly symbols prescribed by the standard mapping.

6.3.2 Design Suggestion for the VM implementation

The VM translator should accept a single command line parameter, `Xxx`, where `Xxx` is either a file name or a directory name:

```
prompt> translator Xxx
```

The translator then translates the file `Xxx.vm` or, in case of a directory, all the `.vm` files in the `Xxx` directory. The result of the translation is always a single assembly language file named `Xxx.asm`, located in the same directory as the input `.vm` file(s). The translated VM code must conform to the standard mapping (Section 6.3.1).

Program Structure

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

Parser

This module reads a `.vm` file, parses all the VM commands in it, and gives the main program convenient access to their components. In addition, the parser removes all white space and comments from the code. We propose implementing this logic as a `Parser` class with the following API.

Parser(String fileName): Constructs a `Parser` object for this file.

boolean hasMoreCommands(): Are there more commands in the input?

void advance(): Reads the next command from the input and makes it the *current command*. This method should be called only if `hasMoreCommands()` is true. Initially there is no *current command*.

int getCommandType(): Returns the constant corresponding to the current command type. There are 9 possibilities, each represented by one of the following class constants: `COMMAND_ARITHMETIC` (used for all arithmetic commands), `COMMAND_PUSH`, `COMMAND_POP`, `COMMAND_LABEL`, `COMMAND_GOTO`, `COMMAND_IF`, `COMMAND_FUNCTION`, `COMMAND_RETURN`, `COMMAND_CALL`.

String getArg1(): Returns the first argument of the current command. In the case of `COMMAND_ARITHMETIC`, the command itself (“add”, “sub”, etc.) is returned. In case of `COMMAND_RETURN`, “” is returned. The string is always returned in all-lowercase.

int getArg2(): Returns the second argument of the current command for `COMMAND_PUSH`, `COMMAND_POP`, `COMMAND_FUNCTION`, and `COMMAND_CALL`, and 0 for all other commands.

Code Writer

This module is responsible for translating each and every VM command into Hack assembly code. We propose implementing it as a `CodeWriter` class with the following API.

CodeWriter(Writer w): Constructs a `CodeWriter` object for the given output stream (`Writer`).

Void setFileName(String fileName): Informs the code writer about the name of the current file.

Void writeArithmetic(String command): writes the assembly code that is the translation of the given arithmetic command.

Void writePushPop(int command, String segment, int index): writes the assembly code that is the translation of the given `command`, where `command` is either `COMMAND_PUSH` or `COMMAND_POP`.

Void close(): closes the code writer.

In the next chapter, the code writer will be augmented with methods for writing the code of the remaining commands in the VM language.

Main Program

For each `.vm` file, the main program should open a parser and compile to a single writer. If the program's argument is a directory name rather than a file name, it should process all the `.vm` files in this directory.

6.4 Perspective

Work in progress.

6.5 Build it

This section describes how to build the basic VM translator described in this chapter. In the next chapter we will extend this basic translator with additional functionality, leading to a full-scale VM implementation.

Objective: Develop a basic VM translator that implements the *stack arithmetic* and *memory access* commands described in the *VM Specification* (section 6.2). The VM should be implemented on the Hack computer platform, conforming to the *standard mapping* described in Section 6.3.1.

Contract: Write the VM translator program. Use it to compile all the test `.vm` programs supplied below, yielding corresponding `.asm` programs written in Hack assembly. When executed on the supplied CPU Emulator, the `.asm` programs generated by your translator should deliver the results mandated by the supplied test scripts and compare files.

Proposed Implementation Stages

We recommend implementing the basic translator in two stages. This modularity will allow you to test your implementation incrementally, using step-by-step test programs that we provide. Here are the suggested stages.

Stack arithmetic: The first version of your translator should implement two things: the stack arithmetic commands as well as the “push constant” command, i.e. the *push* command for the special case where its first argument is “constant”.

Memory access commands: The next version of your translator should be a full implementation of the *push* and *pop* commands, handling all eight memory segments:

1. You have already handled the `constant` segment.
2. Next, handle the five segments `local`, `argument`, `this`, `that`, `temp`.
3. Next, handle the `pointer` segment, allowing modification of the bases of the T-segments.
4. Finally, handle the `static` segment according to the standard mapping

Test Programs

The supplied test programs are designed to support the incremental development plan described above. We supply five test programs and test scripts, as follows.

Stack Arithmetic programs:

- `simpleAdd`: Adds two constants;
- `stackTest`: Executes a sequence of arithmetic and logical operations.

Memory Access programs:

- `basicTest`: Executes *pop* and *push* operations using various memory segments;
- `pointerTest`: Executes *pop* and *push* operations using the `pointer` segment;
- `staticTest`: Executes *pop* and *push* operations using the `static` segment.

Tips

Implementation: In order for a VM program to start running, it should include a preamble startup code that forces the VM implementation to start executing it. 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 target RAM. Both issues (startup code and segments initializations) are described and implemented in the next chapter. The difficulty of course is that we need them in place in order to run the test programs given in *this* project. The good news are that you should not worry about these issues at all, since the supplied test scripts take care of them in a manual fashion, for the purpose of this chapter only.

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

1. Use your partial translator to translate the supplied test `.vm` program. The result should be a text file containing a translated `.asm` program, written in the Hack assembly language.
2. Inspect the translated `.asm` program. If there are syntax (or any other) errors, debug your translator.
3. Use the supplied `.tst` and `.cmp` files to run your translated `.asm` program on the CPU Emulator. If there are run-time errors, something is screwed up with your translator.

The fact that each program tests a very well-defined part of the translator should ease the debugging task considerably. The test programs that we supply were carefully planned to test the specific features of each stage in the implementation. Therefore, it's important to implement your translator in the proposed order, and to test it using the appropriate VM programs at each stage. Implementing a later stage before an early one may cause the test programs to fail.

Steps

1. Download the `project6.zip` file and extract its contents to a directory called `project6` on your computer.
2. Write and test the basic VM translator in stages, as described above.