7. The 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 in the book. High-level programs will be first translated into an intermediate code (Chapters 10-111), 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 recently 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* (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*. 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 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.

A virtual machine can be described as a set of virtual memory segments and an associated language for manipulating them. 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 describe, illustrate, 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.

The VM language that we present 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 Hack code. 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.

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

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 the centerpiece of two well-known 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 data structure that comes to play in many computer systems and algorithms. In particular, the VM presented in this chapter is stack-based, providing a working example of the elegance and power of this remarkably versatile data structure. As the chapter unfolds we will describe and illustrate many classical stack operations, and then implement them in our VM translator.

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 "primitive" steps -- steps that are neither "high" nor "low". In the second stage, the 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 the two compilation stages -- the exact definition of the intermediate primitive 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 primitive 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 by many compiler writers, in one way or another. Some developers went as far as defining a formal virtual machine language, most notably the *p-code* generated by several Pascal compilers in the 1970s and the *bytecode* language generated by Java compilers. More recently, the approach has been adopted by Microsoft, whose .NET infrastructure is also based on an intermediate language, running on a virtual machine called CLR.

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 re-use and language inter-operability. For example, some high-level languages are good at handling the GUI, while others excel in scientific calculations. If both languages compile into a common VM language, it is rather natural to have routines in one language call routines in the other, using an agreed-upon invocation syntax.

Another virtue 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.

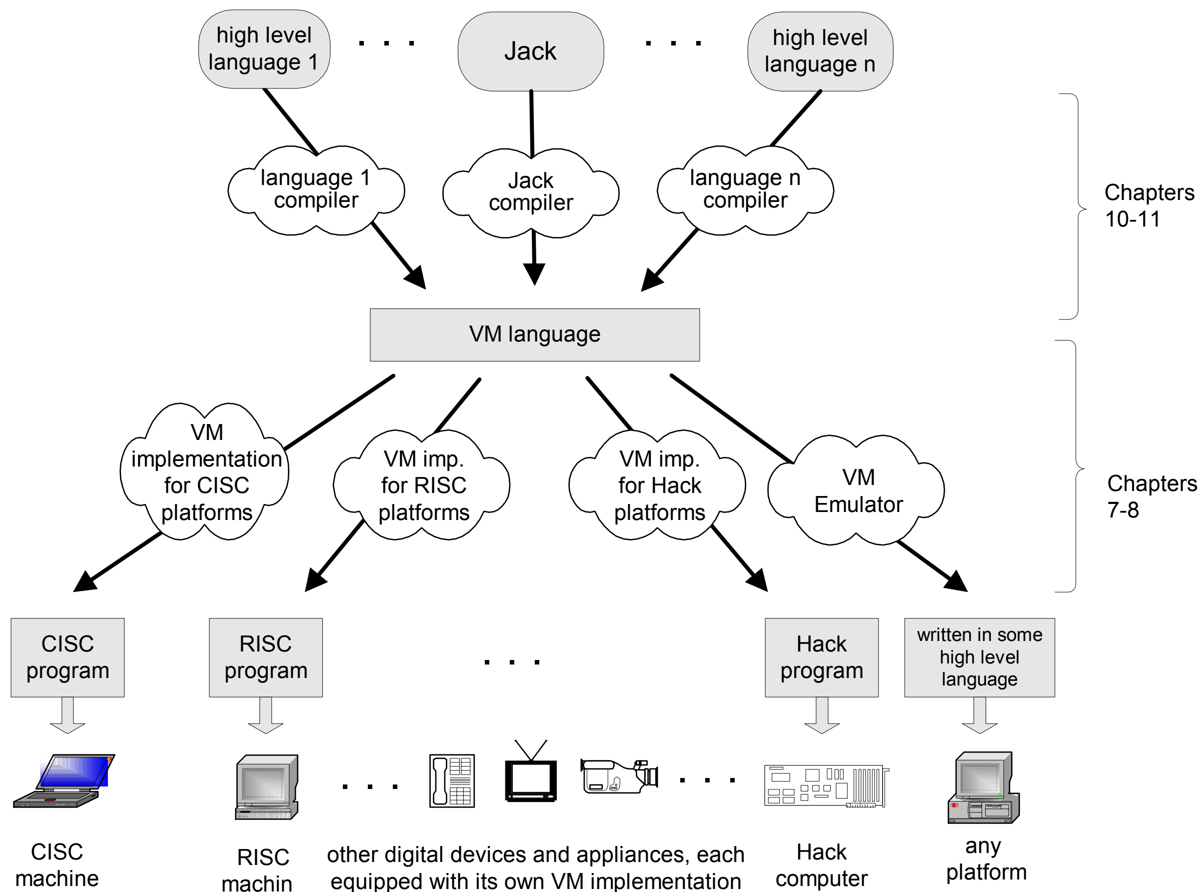


FIGURE 1: The virtual machine paradigm. Once a high-level program is compiled into VM code, the program can run on any hardware platform equipped with a suitable VM implementation. In this chapter we will start building the *VM implementation on the Hack Platform*, and use a VM emulator like the one depicted on the right. (The Jack language is introduced in chapter 9).

The Stack Machine Model

A virtual machine can be described as a set of virtual memory segments and an associated language for manipulating them. Like other 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 virtual machine architecture. 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. Taken together, 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 in this chapter.

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 Figure 2.

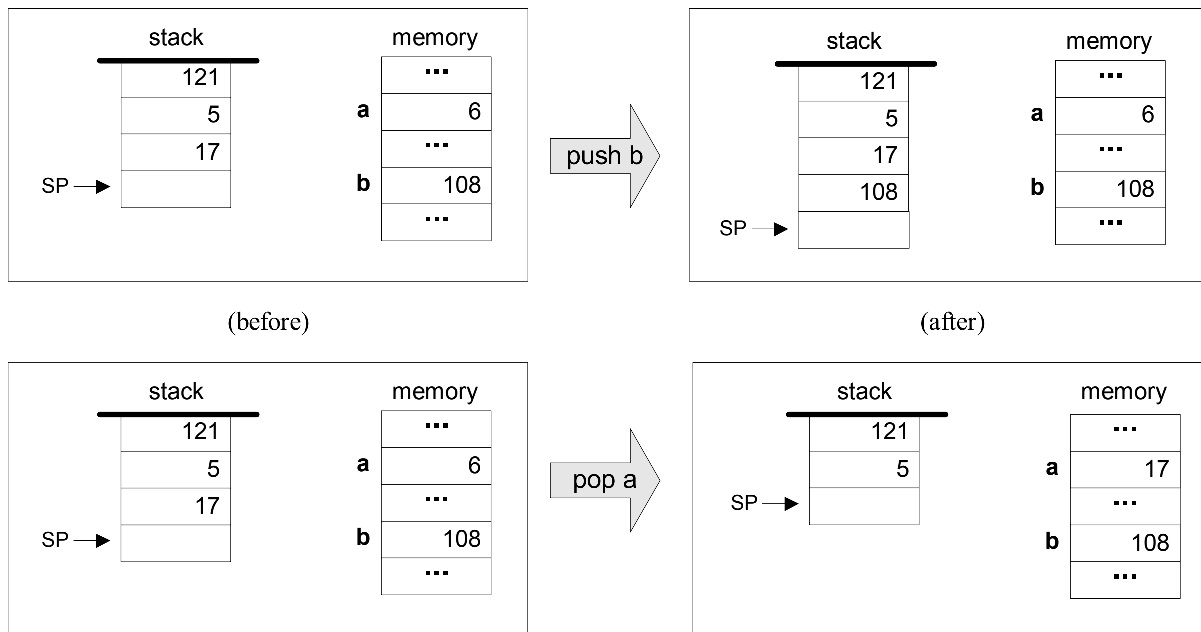


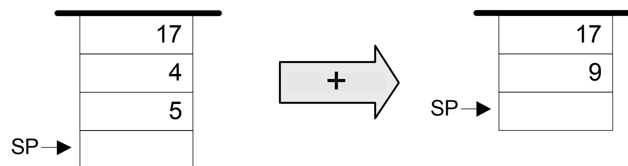
FIGURE 2: Stack processing example, illustrating the two elementary operations *push* and *pop*. Following convention, the stack is drawn upside down, as if it grows downward. The location just after the top position is always referred to by a special pointer called `sp`, or *stack pointer*. The labels *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 erases 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 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 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:



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 Figure 3.

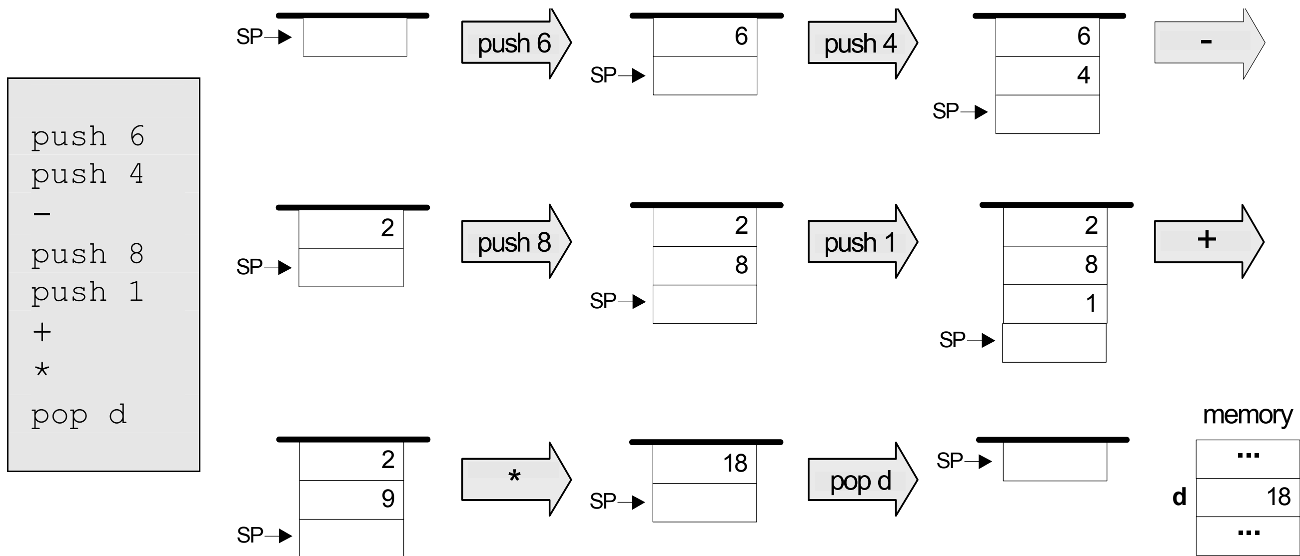


FIGURE 3: Stack-based evaluation of arithmetic expressions.
This example evaluates the expression "`d=(6-4)*(8+1)`"

In a similar fashion, every logical 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 Figure 4.

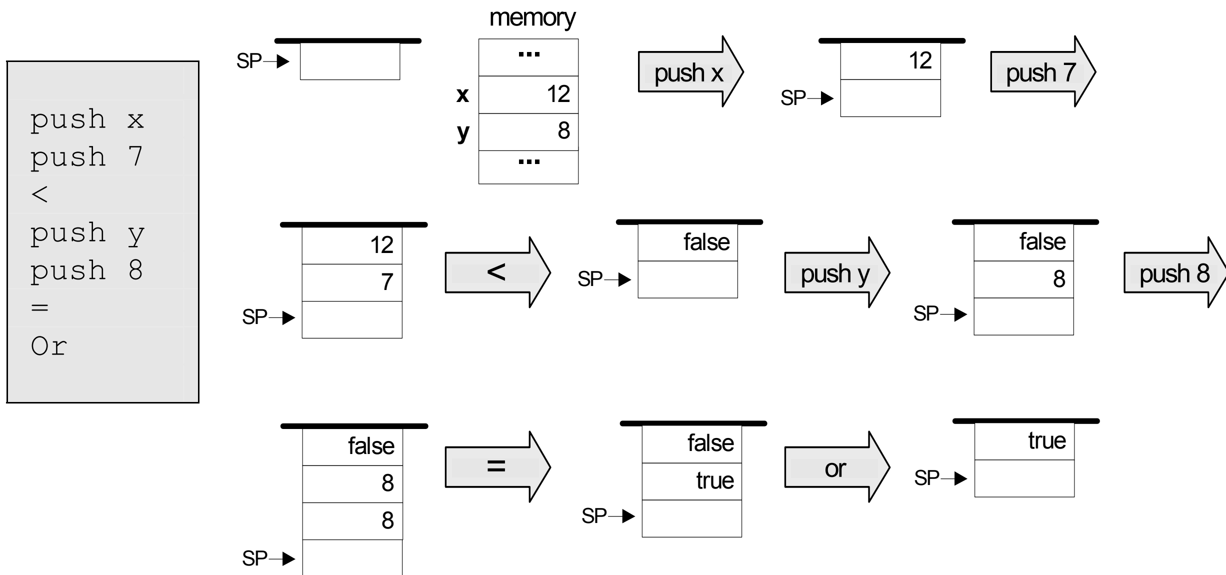


FIGURE 4: Stack-based evaluation of logical expressions.
This example evaluates the expression "`if (x<7) or (y=8) then` ..."

To sum up, the above examples illustrate a general observation: any arithmetic and Boolean expression can be transformed into a series of elementary stack operations that compute its value. Further, as we will show in Chapter 9, this transformation can be described *systematically*. Thus,

one can write a *compiler* program that translates high-level arithmetic and Boolean expressions into sequences of stack commands. Yet 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 2), illustrate them in action (section 3), and describe their implementation on the Hack platform (section 4).

2. VM Specification, Part I

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 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-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 *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*>, <*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.

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. Table 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

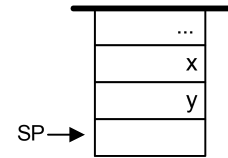


TABLE 5: Arithmetic and Logical stack commands. Throughout the table, *y* refers to the item at the top of the stack and *x* refers to the item just below it.

Three of the commands listed in Table 5 (`eq`, `gt`, `lt`) return Boolean values. The VM represents *true* and *false* as -1 (minus one, `0xFFFF`) and 0 (zero, `0x0000`), respectively.

Example: Figure 6 illustrates all the VM arithmetic commands in action. Each command is applied to an arbitrary 4-bit stack, showing the stack's state before and after the operation. We focus on the three top-most cells in the stack, noting that the rest of the stack is never affected by the current command.

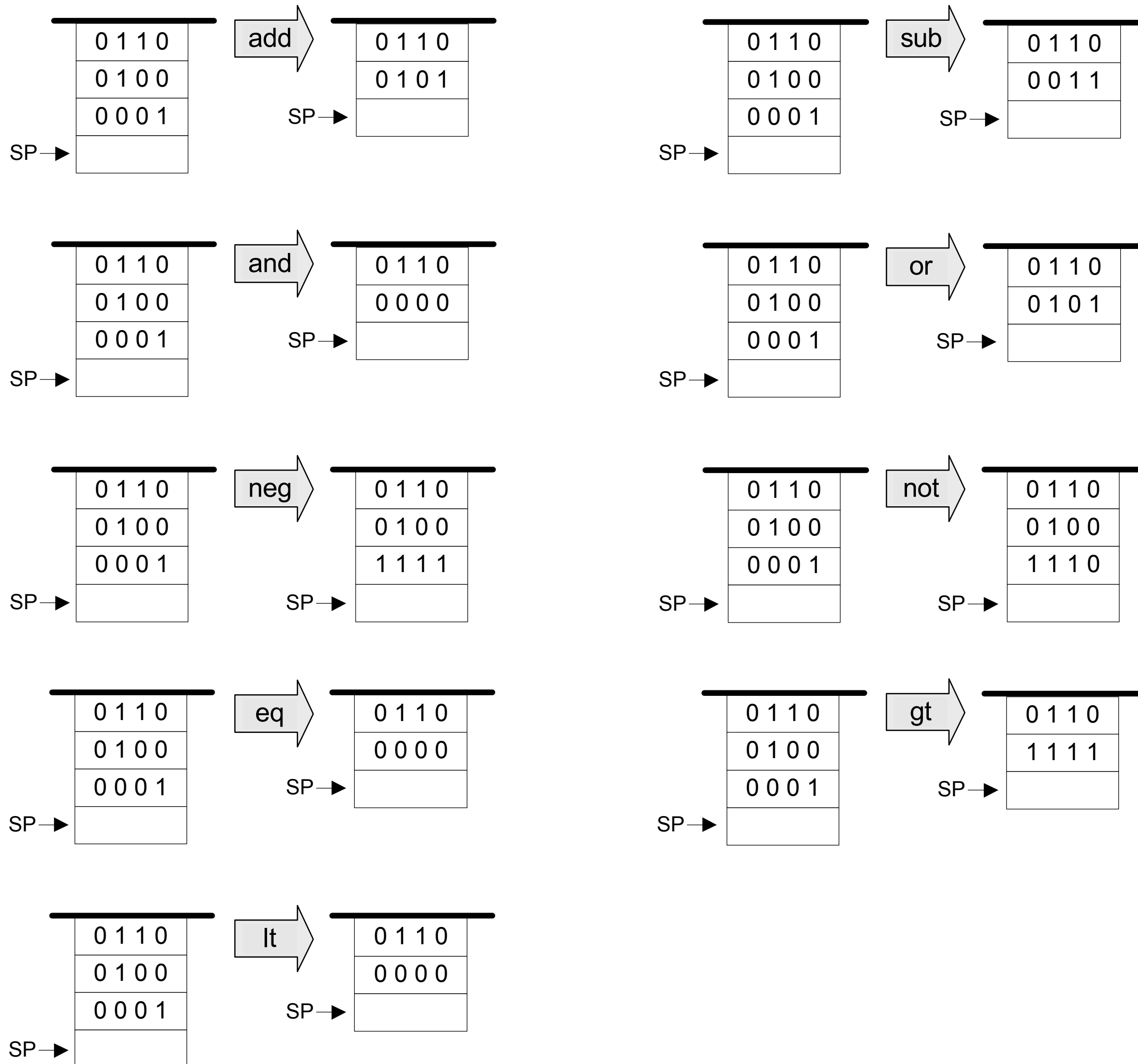


FIGURE 6: Arithmetic commands examples.

2.3 Memory Access Commands

Unlike real computer architectures, where the term "memory" refers to a collection of physical storage devices, the "memory" of a virtual machine consists of abstract devices. In particular, the VM manipulates eight *memory segments*, listed in Table 7. VM functions can access these memory segments explicitly, using VM commands. In addition, the VM manages the *stack*, but only implicitly. In other words, although the stack proper is not mentioned in VM commands, the state of the stack changes in the background, as a side effect of other commands.

Another memory element that exists in the background is the *heap*. As we elaborate later in the chapter, the heap is an area in the physical RAM where objects and arrays are stored. These objects and arrays can also be manipulated by VM commands, as we will see shortly.

Memory Segments: Each VM function sees the eight memory segments described in Table 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 `.vm` 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`	General-purpose segments that can be made to correspond to different areas in the heap. Serve various programming needs.	Any VM function can bind these segments to any area on the heap by setting the segment's base. The setting of the segment's base is done through the `pointer` segment.
`pointer`	Fixed 2-entry segment that holds the base addresses of `this` and `that`.	May be set by the VM program to bind `this` and `that` to various areas in the heap.
`temp`	Fixed 8-entry segment that holds temporary variables for general use.	May be used by the VM program for any purpose.

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

Six of the virtual memory segments have a fixed purpose, and their mapping onto the host RAM is controlled by the VM implementation. In contrast, the `this` and `that` segments are general purpose and their mapping on the host RAM can be controlled by the current VM program: `pointer 0` controls the base of the `this` segment and `pointer 1` controls the base of the `that` segment.

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.

The stack: Consider the commands sequence "`push argument 2`" followed by "`pop local 1`". This code will end up storing the value of the function's 3rd argument in its 2nd local variable (each segment's index starts at 0). The working memory of these commands 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. In addition, although every memory access operation involves the stack, individual stack elements cannot be accessed directly, except for the topmost element.

2.4 Program flow commands

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

These commands are discussed in the next chapter, and are listed here for completeness.

2.5 Function calling commands

- `function` *functionName 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 arguments
- `return` // transfer control back to the calling function

Where *functionName* is a symbol and *nLocals* and *nArgs* are non-negative integers. These commands are discussed in the next chapter, and are listed here for completeness.

2.6 The big picture

We end the first part of the VM specification with a "big picture" view of the overall translation process, from a high-level program into machine code. At the top of Figure 9 we see a Jack program, consisting of two classes (Jack is a Java-like language that will be introduced in chapter 9). Each class consists of several methods. When the Jack compiler is applied to the directory in which these classes reside, it produces two VM files. In particular, each *method* in the high-level source code translates into one *function* at the VM level.

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 low-level program does two main things. First, it emulates all the virtual memory segments shown in the figure, 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. If everything works well, i.e. if the compiler and the VM translator are implemented correctly, the target platform will end up effecting the behavior mandated by the original Jack program.

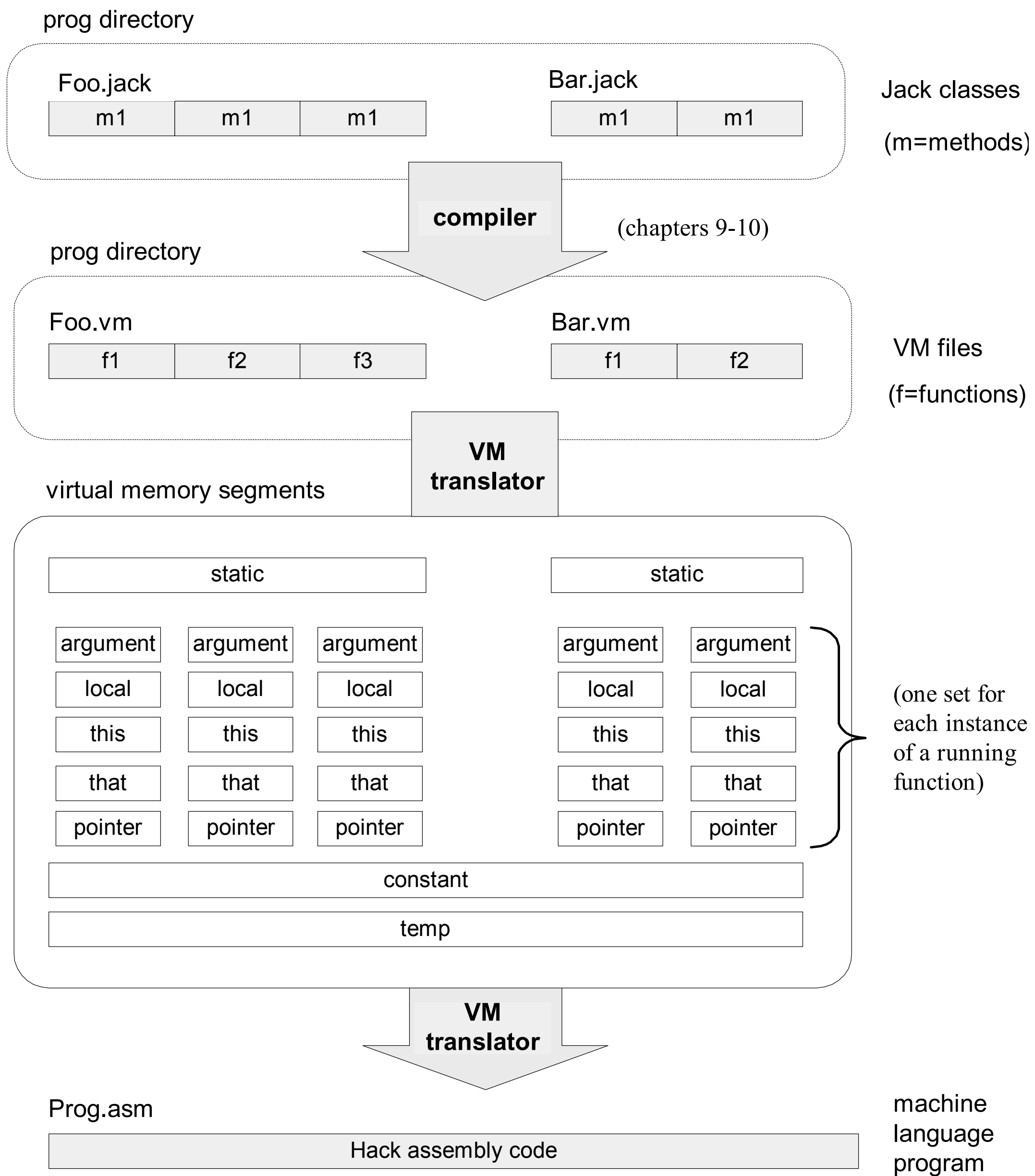


FIGURE 9: VM translation: the big picture.

3. VM Programming Examples

We now turn to illustrate the VM architecture, language, and programming style in action. We give three examples: (i) a typical arithmetic task, (ii) typical handling of object fields, and (iii) typical handling of array elements.

It's important to note 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 its translation it into VM code. We use a C-style syntax for all the high-level examples.

3.1 A Typical Arithmetic Task

Consider the multiplication algorithm shown at the top of Program 10. How should we (or more likely, the compiler) express this algorithm in the VM language? Well, given the primitive nature of the VM commands, we must think in terms of simple "goto logic," resulting in the "first approximation" version of Program 10. Next, we have to express this logic using a stack-oriented formalism. It is instructive to carry out this translation in two stages, beginning with a symbolic pseudo version of the VM language. Finally, we replace the symbols in the pseudo code with virtual memory locations, leading to the actual VM program. (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,j;
  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
```

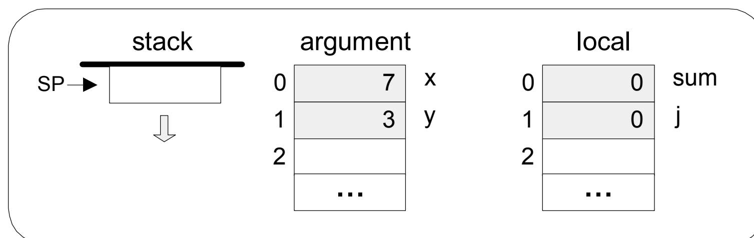
Pseudo VM code

```
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
```

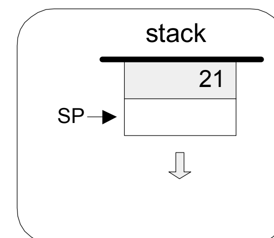
Final VM code

```
function mult 2      // 2 local variables
   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
```

Run-time example: Just after mult(7,3) is entered:



Just before mult(7,3) returns:



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

PROGRAM 10: VM programming example.

We end this example with two observations. First, let us focus on the figure at the bottom of Program 10. 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. Second, let us focus on the translation from the pseudo code to the final code. Recall that VM commands are not allowed to 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 represent `x`, `y`, `sum` and `j` as `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.

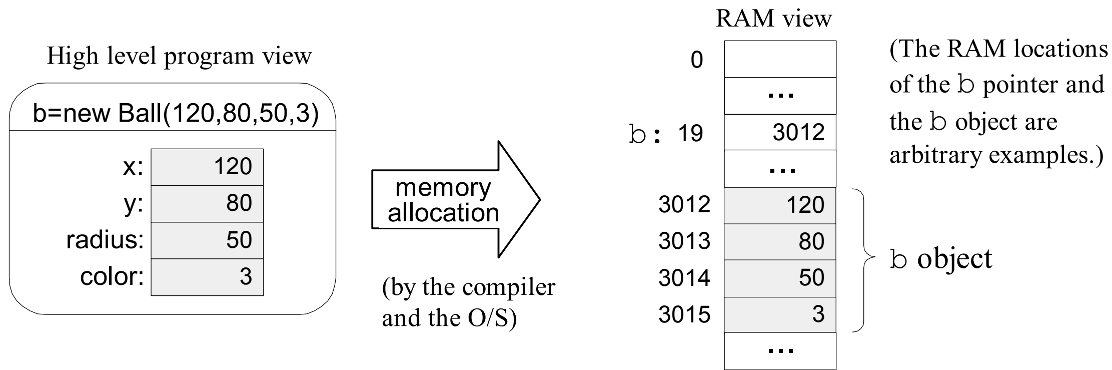
To sum up, when a VM function starts running, it assumes that it is surrounded by a private world, all of its own, consisting of initialized `argument` and `local` segments and an empty stack, waiting to be manipulated by its commands. The agent responsible for building this world for *every* VM function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

3.2 Object handling

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),* with associated code*,* that can be treated as one entity. 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 an area in the RAM called *heap*. In particular, whenever a program creates a new object using a high-level command like `b=new Ball(...)`, the compiler computes the object's size (in terms of words) and the operating system finds and allocates enough RAM space 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, as shown in Program 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 Program 11.



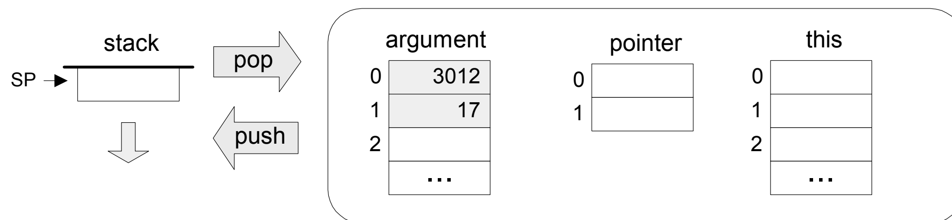
High-level code

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

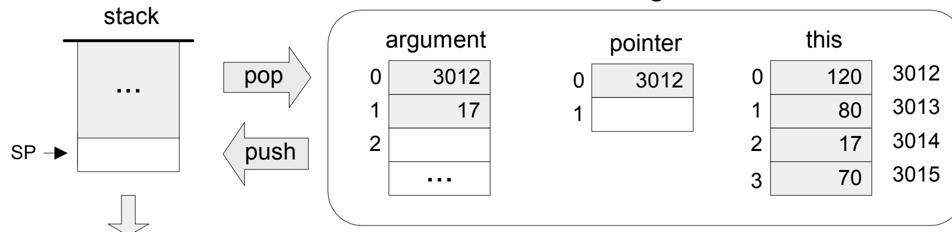
VM code

```
// 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
...
```

Run-time simulation (example): Just after resize(b,17) is entered:



Just after setting b's radius to 17:



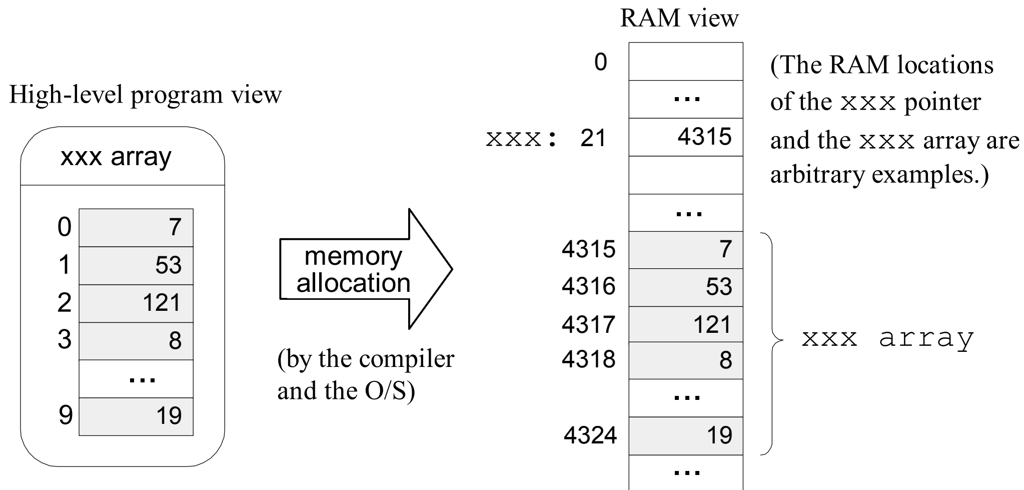
PROGRAM 11: VM-based object manipulation. (The labels at the bottom right (3012, ...) are not part of the VM state, and are given here for ease of reference.)

Note that the name of the object (which happens to be "`b`" in this example) is actually a reference to a memory cell containing the address 3012 (see Program 11). Since `b` is the first argument passed to the `resize` method, 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 heap-resident `b` object using the virtual memory segment `this`, without ever worrying about the physical address of the actual object.

3.3 Array Handling

An array is an indexed vector of objects of the same type. Suppose that a high-level program has created an array of 10 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 `foo`, 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.

In the C language, this operation can be specified using two forms of syntax: `xxx[k]=34`, and `*(xxx+k)=34`. Whereas the former expression is more intuitive for humans, the latter provides a more accurate description of what the machine is actually doing under the surface. Specifically, the C notation "`*x`" means "*the contents of the memory location addressed by* x". Hence, the command "`*(xxx+k)=34`" reads: "*set the RAM location whose address is* `(xxx+k)` *to 34*". As shown in Program 12, this is precisely what the VM code is doing, using primitive VM commands.



High-level code

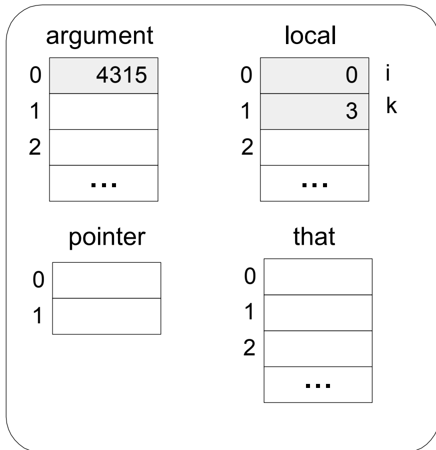
```
method foo (int[] xxx, ...) {
   int i,k;
   ...
   k=3;
   xxx[k]=34;
   ...
}
```

VM code

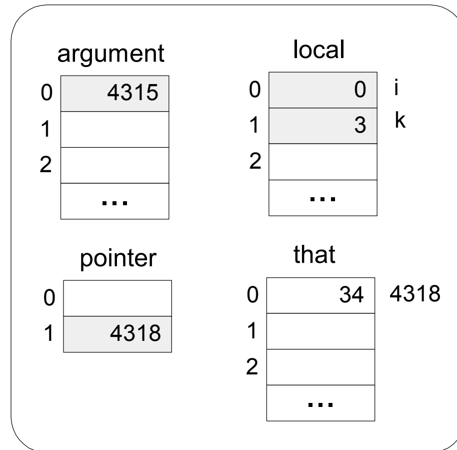
```
...
push constant 3      // set k=3
pop  local 1
push argument 0      // get xxx's base address
push local 1         // get k
add                  // put xxx+k on the stack
pop  pointer 1       // set that's base to (xxx+k)
push constant 34
pop  that 0          // *(xxx+k)=34
...
```

Run-time simulation (example):

Just after the k=3 operation:



Just after the xxx[k]=34 operation:



PROGRAM 12: VM-based array manipulation. (The symbolic and numeric labels shown in the right are not part of the VM state, and are given here for ease of reference.)

4. 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 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 machine language instructions that effect the command on the target platform.

This section describes how to implement the VM specified in the previous section on the Hack platform specified in Chapter 4. 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.

4.1 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, 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 usually recommended to provide some guidelines on how the VM should map on the target platform, rather than leaving these decisions completely to the implementer'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, 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. 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 the memory maps of I/O devices. The VM implementation should use this space as follows:

RAM addresses	*Usage*
`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 (used to store objects and arrays)
`16384-24575:`	Memory mapped I/O

TABLE 13: 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 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 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. In other words, we anticipated that the main use of the assembly language will be to develop VM translators. With that in mind, the expected use of the Hack registers in the VM context is described in Table 14.

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]`	`TEMP`	Hold the contents of the `temp` segment
`RAM[13-15]`	`(-)`	Can be used by the VM implementation as general-purpose registers.

TABLE 14: 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 location in any one of these segments should 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.

`pointer, temp`: These segments are globally fixed and are each mapped directly onto a fixed area in the RAM. Specifically, the `pointer` segment is mapped to RAM locations 3-4 (Hack registers `THIS` and `THAT`) and the `temp` segment is mapped to RAM locations 5-12 (Hack

registers R5, R6, …, R12). Thus access to `pointer i` should be translated to assembly code that accesses RAM location *i+3,* and access to *temp i* should be translated to assembly code that accesses RAM location *i+5*.

`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

To recap, Table 15 summarizes all the assembly language symbols used by VM implementations that conform to the standard mapping.

Symbol	***Usage***
SP, LCL, ARG, THIS, THAT	These pre-defined symbols point to the stack top and to the base addresses of the virtual segments local, argument, this, and that.
R13-R15	Can be used for any purpose.
"`f.j`" symbols	Each static variable *j* in file *f*`.vm` is translated into the assembly symbol `f.j`. In the subsequent assembly process, these symbols will be automatically allocated RAM locations by the Hack assembler.
Flow of control symbols (labels)	The VM commands `function`, `call`, and `label` are handled by generating symbolic labels, to be described in chapter 8.

TABLE 15: Usage of Assembly symbols in the standard mapping.

4.2 Design Suggestion for the VM implementation

The VM translator should accept a single command line parameter, `Xxx`, where `Xxx` is either a file name containing a VM program (the `.vm` extension must be specified) or the name of a directory containing one or more `.vm` files (in which case there is no extension):

```
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`,

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*.

Parser

This module handles the parsing of a single .vm file. We propose the following API:

Parser Module			
Encapsulates access to the input code. Reads a VM command, parses it, and provides convenient access to its components. In addition, 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 (enumeration)	Returns the type of the current command. C_ARITHMETIC is returned for all the arithmetic VM 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 for 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.

Code Writer

This module is responsible for translating each VM command into Hack assembly code. We propose the following API:

CodeWriter Module			
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` (enumeration), `segment` (string), `index` (int)	--	Writes the assembly code that is the translation of the given `command`, where `command` is one of the two enumerated values: `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 single `CodeWriter` for handling the output, but a separate `Parser` for handling each input file.

5. 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-stage 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, typically using interpreters. This idea came in the right time, since in the early 1980's the world of personal computers began to split into different processor and operating system camps. On the backdrop of this division, compilers that generated p-code provided a reasonable solution for running the same high-level program on multiple platforms, without having to re-compile it. This was the beginning of the *cross-platform compatibility* challenge, as well as the first attempt to address it using a VM approach.

Following the wide spread adoption of the world-wide-web in the mid 1990s, cross-platform compatibility became a universally vexing issue. Viewing this want as a business opportunity, Sun Microsystems sought to develop a new language that could potentially run on any computer and digital device hooked to the Internet. The language that emerged from this effort – *Java* – was based on a virtual machine model. Specifically, the *Java Virtual Machine* (JVM) is a specification that describes an intermediate language called *bytebode.* Files written in the bytecode language are used for dynamic 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 *Java run-time environments* became widely available "plug-ins", provided freely by Sun for practically any processor / OS combination, including game consoles and cell-phones.

Today, the JVM model and the associated bytecode language are widely used for code-mobility and interoperability over the Internet. The cornerstone of this architecture is the ubiquitous Java virtual machine, allowing Sun to market Java as a "*write once, run everywhere*" language. As a side benefit, the JVM became a means for empowering the client computer in several different ways. For example, it allows verifying the transmitted bytecodes for safety, reducing the risk of downloading malicious code.

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. Yet unlike the Java VM approach, which seeks to allow Java programs to execute on any possible hardware/OS platform, the CLR is designed to run only on top of operating systems provided by Microsoft.

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 such services as memory management, GUI utilities, string functions, math functions, and so on. One such library will be described and built in chapter 12.

6. Build it

This section describes how to build the VM translator specified 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 VM translator that implements the *stack arithmetic* and *memory access* commands described in the *VM Specification, Part I* (section 2). The VM should be implemented on the Hack computer platform, conforming to the *standard VM-on-Hack mapping* described in Section 4.1.

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 handy in this project is the visual *VM Emulator* supplied with the book. This program allows to experiment with a working VM environment before you set out to implement it yourself. For more information about this tool, refer to the *VM Emulator Tutorial*.

Contract: Write a VM-to-Hack translator. Use it to translate the test `.vm` programs supplied below, yielding corresponding `.asm` 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.

6.1 Proposed Implementation Stages

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

Stage I: Stack arithmetic: The first version of your translator should implement two things: (i) the nine stack arithmetic and logical commands, and (ii) the "`push constant`" command. Note that the latter is the *push* command for the special case where the first argument is "`constant`".

Stage II: 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. We suggest the following order:

0. You have already handled the `constant` segment;
1. Next, handle the four 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.

5.2 Test Programs

The supplied test programs and scripts are designed to support the incremental development plan described above.

Stage I: Stack Arithmetic programs:

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

Stage II: Memory Access programs:

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

We supply two test scripts for each test program. One script allows running the source `.vm` test program on the VM emulator, so that you can gain familiarity with the program's intended operation. The other script allows testing the target assembly code generated by your VM translator on the CPU emulator.

5.3 The VM Emulator

A virtual machine model can be implemented in several different ways. Translating VM programs into machine language -- as we have done so far -- is one possibility. Another implementation option is simulating the VM model in software, using a high-level language. One such simulation program, shown in Figure 16, is included in the software suite that accompanies the book.

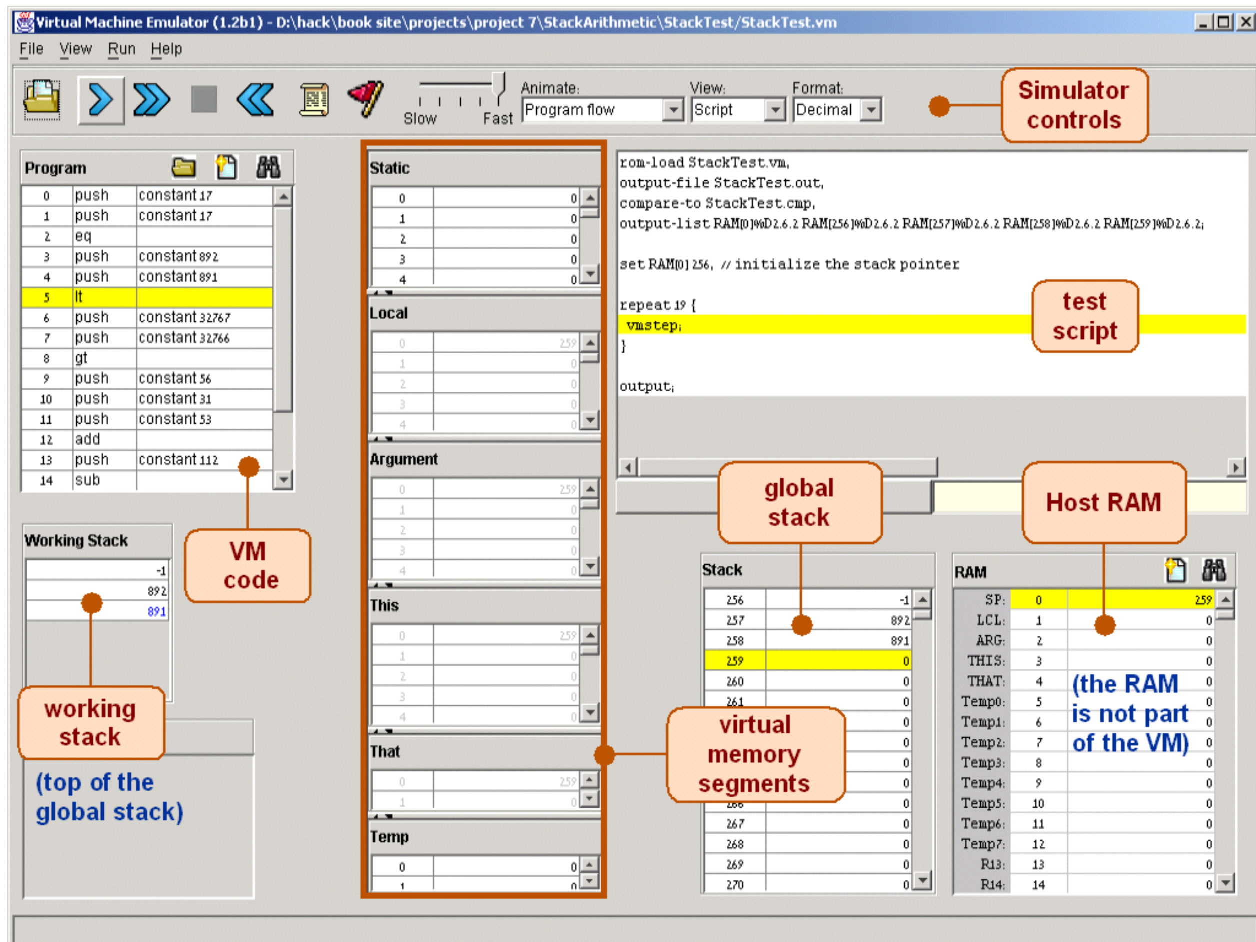


FIGURE 16: The VM emulator. This program is supplied with the book.

Our VM emulator was built with one purpose in mind: 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 is a recommended exercise, as it enables experimentation with the VM environment before you set out to implement it yourself.

5.4 Tips

Implementation: In order for any VM program to start running, it should 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 described and implemented in the next chapter. 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 are that you should not worry about these issues, since the supplied test scripts take care of them 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 supplied test `.vm` program on the VM emulator, using the VM-emulator test script, until you feel comfortable with the intended behavior of the output of this translation step.

2. Use your partial translator to translate the `.vm` program. 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 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 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.

Steps

1. Download `project7.zip` and extract its contents into a directory called `project7` on your computer, without changing the directories structure embedded in the zip file.

2. Write and test the basic VM translator in stages, as described above.