Virtual Machines: Abstraction and Implementation

Shimon Schocken

Efi Arazi School of Computer Science

IDC Herzliya

July, 2009

ABSTRACT

Various forms of virtualization play key roles in the construction and usage of diverse system artifacts ranging from modern compilers to hardware migration to cloud computing. Virtualization is also a powerful cognitive apparatus, requiring abstraction skills and modeling abilities. We present a methodology, a software tool, and a set of instructional resources designed to expose students to the construction and working of virtual machines, focusing on both abstraction and implementation issues. All the resources described in this paper are freely available in open source on the web [1].

BACKGROUND

Virtualization describes a setting in which one system is made to emulate another. This capability enables many computing areas such as web-based software deployment, storage networks, cloud computing, mobile computing, software as a service, and managed code. Indeed, a search of articles whose title includes the term "virtual machine" yields thousands of hits.

Closer to home, virtualization is also a major, albeit implicit, part of the working environment of a typical CS student. Whenever a Java or a C# programmer compiles a program, he or she creates intermediate code for a virtual machine, be it the JVM or the CLR, respectively. There is no way to fully appreciate the nature of software development and deployment today – as well as the great benefits offered by managed code -- without gaining a keen appreciation of this two-tiered compilation process, and without grasping the central role that virtual machines play in it.

In addition to its practical value for modern software engineering, virtualization is an important concept on cognitive and pedagogical grounds. Designing one model to emulate another is a challenging undertaking, forcing the designer to reformulate the model's semantics in abstract terms before it can be re-expressed in the target framework. Thus, understanding or constructing a virtual machine is an excellent exercise in developing abstract thinking skills. And, abstraction is of course an important foundation of computational thinking [2,3] as well as of software engineering [4,5].

From a historical perspective, virtualization has a profound legacy. The *universal Turing machine*, as well the *Church-Turing conjecture*, can be viewed as early forerunners of virtualization. Thus, the study of virtual machines provides a natural ground for bringing important aspects of CS history into the classroom. Further, it provides a powerful illustration of how arcane theoretical artifacts evolved with time into some of the most practical enabling technologies.

Although virtual machines proper are rarely taught explicitly in CS programs, educators often view virtualization as a powerful metaphor for thinking about, and teaching, computer science. For example, the IEEE/ACM Computing Curricula Report suggests that “When discussing layers of abstraction, computer systems can be described as a hierarchy of virtual machines” [6].

Why then don’t virtual machines receive the attention that they seem to deserve in academic programs? There are several reasons for this. First, the most natural context for discussing virtual machines is compilation, yet in many CS programs compilation is no longer a required course. Second, commercial virtual machines like the JVM are quite complex, and there is simply no time (let alone academic interest) to delve into their intricate proprietary designs. Finally, the implementation of a virtual machine can be a tedious undertaking, and one can argue that a project of such magnitude is simply beyond the scope of basic CS education.

This paper describes a methodology and a set of tools designed to expose students to the abstraction and implementation of virtual machines. Entailing up to six hours of instruction and a flexible set of homework projects, the module can be easily plugged into existing courses in programming, software engineering, compilation, and computing systems. Using our materials and any programming language of their choice, students can build a complete and working virtual machine from the ground up. The remainder of this paper describes the methodology and tools that we have developed to support this undertaking, and the experience that we have had using it.

RESOURCES

The resources that we have developed consist of language specifications, software tools, lectures, projects, and tutorials. First, we describe a simple and conventional stack-based VM language that students normally learn in one hour. Programs written in this language can be executed on an interactive VM emulator that can be downloaded from our site and run on the student's PC. As we elaborate later in the paper, the VM emulator has access to a mini operating system that extends the basic VM language with many capabilities. This allows students to not only play with push and pop commands, but actually execute, and explore at the VM level, interactive applications like computer games of considerable sophistication. We note in passing that our VM language, emulator, and mini OS are conceptually similar to Java's bytecode, JVM, and JRE, respectively.

Along with these tools, we provide a set of slides and tutorials designed to support up to six hours of classroom or self-study instruction about virtual machines. Finally, we provide a set of structured VM programming exercises and projects, along with a unit-testing plan. Using these materials, students can (i) write, execute, and debug VM programs, and (ii) implement the VM itself on some target computer platform, following our proposed API and implementation plan. All the materials and tools are freely available in open-source in [1].[1]

THE VM ABSTRACTION

Our virtual machine model is stack-based: all operations are done on a stack. It is also function-based: a VM program is a collection of functions, written in our VM language.

The VM language consists of 17 commands that fall into four categories: arithmetic / logical commands that operate on the stack, memory access commands that transfer data between the stack and the memory, program flow commands that facilitate branching operations, and function calling commands that invoke functions and return from them. The commands have an intuitive stack-oriented syntax and semantics, as we now turn to illustrate. A full language description can be found in [7]. All the commands are illustrated below, in code examples.

Arithmetic/Logical commands: These commands pop their arguments off the stack, compute a function on these arguments, and push the return value onto the stack. For example, suppose that the current stack state is (…,7,15,3), 3 being the topmost element. In such a case, the `add` command will turn the stack into (…,7,18), the top element being the value of 15 + 3. Similarly, the `lt` command (less-than) will turn the same stack into (…,7,0), the top element being the truth value of 15 < 3. Altogether, our VM language features nine arithmetic/logical commands whose self-explanatory mnemonics are `add`, `sub`, `eq`, `gt`, `lt`, `and`, `or`, `neg`, `not`. The first seven commands operate on two arguments; the latter two commands operate on one argument.

[1] For reasons that we neither fully understand nor wish to contest, when searching "VM emulator" in Google, the top hit following the Wikipedia entries is our VM emulator tutorial (as of April 2009).

Memory Access commands: Data is inserted into the stack via the `push` *x* command, *x* being a constant or a variable. For example, given the stack (…,7,15,3), the command `push 6` will turn it into (…,7,15,3,6). Data is removed from the stack via the `pop` *x* command, *x* being a variable; For example, given the stack (…,7,15,3), the command `pop x` will have two side effects: the stack will turn into (…,7,15), and the value of the memory variable `x` will be set to 3.

Program flow commands: VM programs can have labels that can be used by two branching commands. The `goto` *label* command transfers control to the command following *label*. The `if-goto` *label* command pops the topmost element off the stack; If it's "true", control transfers to the command following *label*; otherwise, control continues with the command following the `if-goto` command.

Function calling commands: A VM function can invoke another VM function using the command `call` *f*, where *f* is the name of the called function. Our function calling protocol mandates that before calling another function, the calling function is required to push as many argument values as the called function requires onto the stack. Thus, when a function starts running, it knows that all the argument values that it needs are stored at the top of the stack, and it can proceed to pop and use them as needed. Our function calling protocol also mandates that before terminating its execution, a function must (i) push a return value (which many be null) onto the stack, and (ii) issue a `return` command. Thus, when control returns from the called function to the calling function, the latter can pop and use the return value. We see that the net effect on the stack is that the arguments of the called function are replaced by its return value, which is precisely the same behavior as with primitive VM commands like `add`, `eq`, etc. It is in this sense that the VM language is at once simple yet infinitely extensible: the basic language can be extended with user-defined functions at will. Importantly, the user-defined VM functions have precisely the same stack-oriented look-and-feel as the VM's primitive commands.

The Mini-OS: Exploiting the extensibility of our VM language, and willing to facilitate "industrial strength" programming, we've extended the basic VM language with an open-ended library of VM functions that presently support math operations, string processing, array handling, input/output operations, memory management, and bit-mapped graphics. Taken together, we refer to this collection as our "mini-OS".

* * *

We now turn to give some VM programming examples.

Example 1: The following VM function, which is part of our mini-OS, returns the absolute value of a supplied argument. When reading the code, recall that each VM function assumes that its argument values have already been pushed onto the stack by the calling function:

```
function Math.abs x
    // the single argument x is the top-most stack value
    push 0         // stack state: (..., x, 0)
    lt             // returns the truth value of x < 0
    if-goto negx   // if x < 0, branch
    push x         // x is non-negative; return x
    return
negx:
    push x         // x is negative, return -x
    neg
    return
```

Once the `abs` function is available, computing the expression, say, `3 + abs(x - 1)` can be done via the following sequence:

```
push 3          // stack state: (..., 3)
push x          // (..., 3, x)
push 1          // (..., 3, x, 1)
sub             // (..., 3, x-1)
call Math.abs   // (..., 3, abs(x-1))
add             // (..., 3 + abs(x-1))
```

Example 2: The following sequence draws on the screen two circles around the same center. The second circle is twice as large as the first:

```
push x
push y
push r
call Screen.drawCircle  // OS function
push x
push y
push r
push r
add
call Screen.drawCircle
```

Note that the arguments are pushed again onto the stack before `drawCircle` is called again. This is consistent with our function calling protocol, and must obeyed since every VM function consumes its arguments as a side effect of its execution.

Example 3: To compute the expression $-b+\sqrt{a^2-4\cdot a\cdot c}$**, one can use the following VM code:**

```
push b
neg
push a
push a
call Math.mult    // OS function
push 4
push a
call Math.mult
push c
call Math.mult
sub
call Math.sqrt    // OS function
add
```

The functions `abs`, `max`, `drawCircle`, `mult`, and `sqrt` are part of our mini-OS, whose complete API is published in [6].

Memory management: The VM examples that we have shown thus far can run as-is on our VM emulator, with one exception: similarly to the JVM model, our VM language has no symbolic variables. Instead of symbolic references like "x" and "y", VM functions refer to memory locations via virtual references like "local 0" and "argument 1". The first part of this reference represents the variable kind, and the second part represents that it is variable number 0,1,2,… of its kind. Specifically, our VM model assumes the existence of several such virtual memory segments with names like `local` and `argument`, each being an indexed vector holding all the variable values of a certain kind. Thus, instead of writing, say, `push x`, as we have done so far, we write `push argument 0`. The following section explains how the mapping between high-level variables like `x` and virtual references like `argument 0` is established.

A Glimpse to The Big Picture: There are two ways to think about the VM model presented here. First, it can be viewed as a stand-alone abstraction, complete with its own data structures (stack and virtual memory segments) and programming language. Taking this view, one can write VM programs directly and watch them execute on an interactive VM emulator, as we illustrate in the next section. At the same time, the VM language can also be viewed as an intermediate language like Java's Bytecode or the .NET framework's IL, i.e. the target language of a high level compiler.

Let us consider the latter view. In [6], we describe and specify a simple Java-like language, called Jack. Using Jack, students write high-level programs and translate them into the VM language using a Jack compiler. For example, here is a typical Jack method:

```
/** A library of math-oriented services */
 class Math {
   /** Returns the absolute value of its argument */
   method int abs (int x) {
      if x < 0 return -x
      else return x
   }
   // more Math functions follow.
}
```

When the Jack compiler translates this method, it maps the `x` variable to `argument 0`, since `x` happens to be the first and only argument of this method. Next, the compiler writes the VM function listed in Example 1 above, except that each occurrence of "`x`" in the VM code is replaced with "`argument 0`". The compiler sets the name of the generated VM function to `Math.abs`, since it emanates from a Jack class called `Math`.

The `Math` class, a library of commonly used mathematical functions, is part of our mini-OS. We note in passing that just like Unix is written in C, our OS is written in Jack. Following compilation, it becomes a set of executable VM functions.[2]

The Jack compiler mentioned above comes to play in our course in two ways. If one simply wants to compile Jack programs, one can download an executable version of the compiler from our web site [1]. If one wants to build the compiler itself, one can do so by following projects 9 and 10 of the course.

[2] The Jack language and our compilation model are described in a forthcoming article.

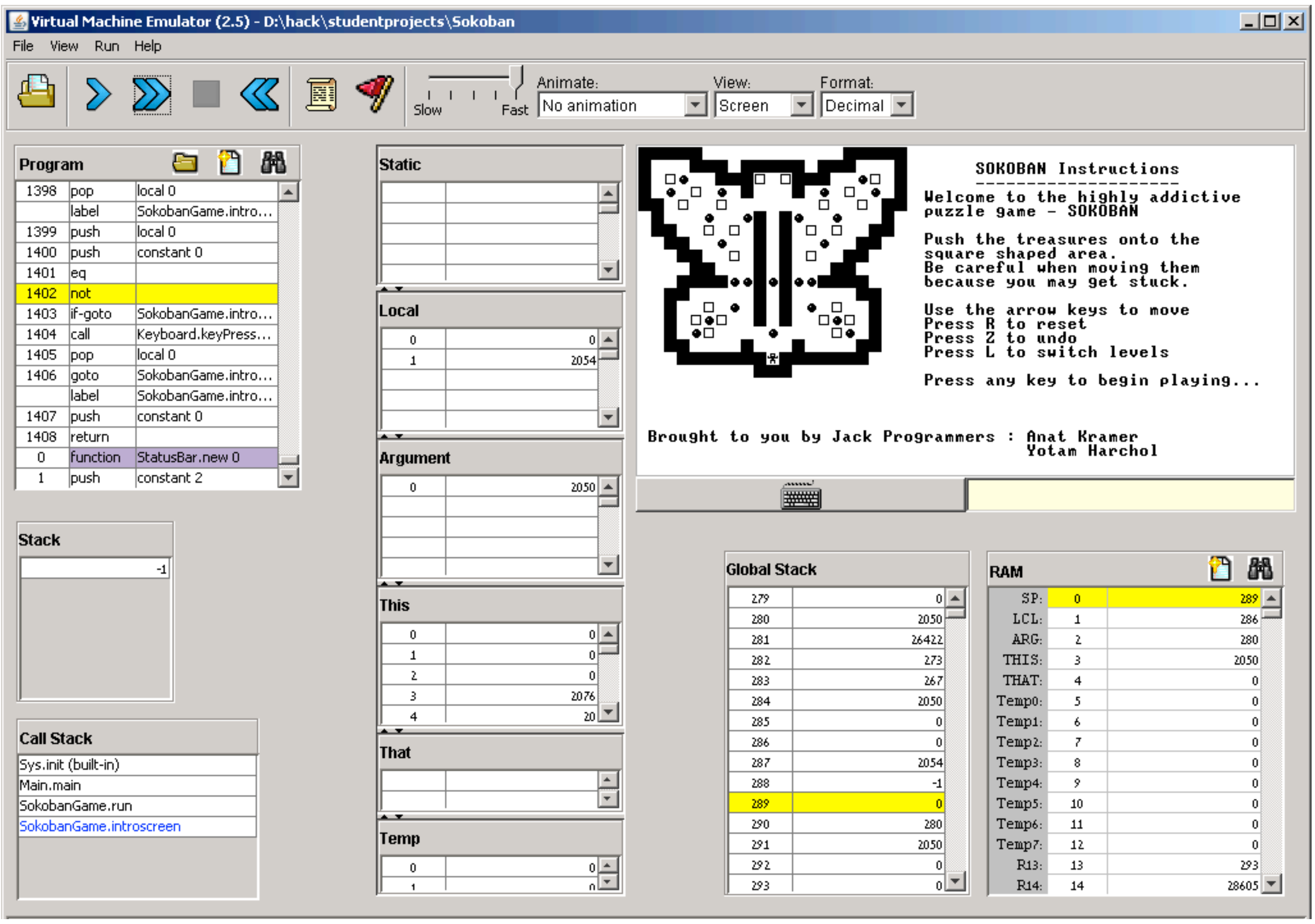
THE VM EMULATOR

As in most object-based languages, a Jack program is a collection of classes, each being a collection of methods. When applied to a Jack program, the Jack compiler translates each Jack method into a VM function, yielding a single text file that contains a sequence of VM functions.

This text file can then be loaded into, and executed by, the VM emulator. In addition to this user-level code, the emulator has access to our mini-OS, which is also implemented as a sequence of compiled VM functions. Since the OS functions are always pre-loaded to the emulator, user-level functions can always call them. And, as far as the emulator is concerned, there is no difference between user-level code and OS-level routines – they are all implemented as VM functions.

Below we see the opening screen of an interactive "Sokoban" game, written by two of our students, Yotam Harchol and Anat Kramer. The game challenges users to push spherical "treasures" into square "boxes" without getting blocked in dead ends. The game was written in Jack (about 500 lines of code) and translated by the Jack compiler into a VM program, a subset of which is shown in the screen shot below.

The game's logic interacts with the user via `Keyboard.keyPress()`, an OS function that returns (puts on the stack) the scan-code of the key currently pressed by the user. The output is drawn using other OS functions designed to draw bitmap images on a 256 rows by 512 columns screen shown in the top-right corner of the emulator's GUI. Using the emulator's controls on the top-left, one can load, inspect, and run such VM programs either in "step-wise" mode -- one VM command at a time, or in "fast-forward" mode, under program control.



Virtual Machine Emulator (2.5) - D:\hack\studentprojects\Sokoban
File View Run Help
Slow Fast

Animate: No animation | View: Screen | Format: Decimal

Program

1398	pop	local 0
	label	SokobanGame.intro...
1399	push	local 0
1400	push	constant 0
1401	eq	
1402	not	
1403	if-goto	SokobanGame.intro...
1404	call	Keyboard.keyPress...
1405	pop	local 0
1406	goto	SokobanGame.intro...
	label	SokobanGame.intro...
1407	push	constant 0
1408	return	
0	function	StatusBar.new 0
1	push	constant 2

Stack

-1

Call Stack

Sys.init (built-in)
Main.main
SokobanGame.run
SokobanGame.introscreen

Static

Local

0	0
1	2054

Argument

0	2050

This

0	0
1	0
2	0
3	2076
4	20

That

Temp

0	0
1	0

```
SOKOBAN Instructions
--------------------
Welcome to the highly addictive
puzzle game - SOKOBAN

Push the treasures onto the
square shaped area.
Be careful when moving them
because you may get stuck.

Use the arrow keys to move
Press R to reset
Press Z to undo
Press L to switch levels

Press any key to begin playing...

Brought to you by Jack Programmers : Anat Kramer
                                     Yotam Harchol
```

Global Stack

279	0
280	2050
281	26422
282	273
283	267
284	2050
285	0
286	0
287	2054
288	-1
289	0
290	280
291	2050
292	0
293	0

RAM

SP:	0	289
LCL:	1	286
ARG:	2	280
THIS:	3	2050
THAT:	4	0
Temp0:	5	0
Temp1:	6	0
Temp2:	7	0
Temp3:	8	0
Temp4:	9	0
Temp5:	10	0
Temp6:	11	0
Temp7:	12	0
R13:	13	293
R14:	14	28605

Figure 1. VM emulator screen shot, running a game developed in a student's project. The emulator requires no installation beyond downloading and running as is on a standard PC.

Importantly, the VM emulator allows the user to pause the currently running code and inspect visually the virtual machine apparatus. For example, the call stack panel, shown at the bottom left of the emulator's GUI, indicates which VM function is currently running, as well as all the functions up the calling hierarchy, waiting for the current function to return. Some of these functions, e.g. `Sys.init`, are OS-level functions, while others, like `SokobanGame.introscreen`, are user-level functions.

The current state of the stack, which happens to contain the single value -1 in the Figure 1 snapshot, is shown just above the call stack. The middle section of the emulator's GUI shows the current state of the virtual memory segments. For example, we see that the `local` segment presently contains two local variable values and the `argument` segment contains a single argument value, both belonging to the scope of the currently executing function.

The bottom right of the emulator's GUI (see Figure 1) allows us to peek into the RAM that hosts the VM implementation. For example, the first three RAM words hold the current values of the stack pointer (`SP`), the base address of the local variables segment (`LCL`), and the base address of the argument variables segment (`ARG`). If we inspect the corresponding addresses in the RAM segment shown under the "global stack" title, we see exactly how these virtual memory segments are mapped on the host RAM. Further, if we'll scroll up this "global stack" control, we will see the stored frames of all the waiting functions, i.e. copies of their stack states and memory segments, waiting to be reinstated when control returns to the calling function.

Function call-and-return is one of the most important abstractions in programming. In our compilation model, this abstraction is delivered at the VM level, and thus the VM emulator takes care of its overhead. This is done behind the scene, by storing and reinstating the memory resources of all the waiting functions up the stack. The ability to explore this dynamic data structure in-vivo has significant instructional benefits, since it allows students to demystify the function call-and-return behavior, often viewed as a black box.

IMPLEMENTATION

In addition to experimenting with VM programming and running VM programs on the supplied VM Emulator, students can also be guided to implement the VM abstraction. In other words, they can be asked to realize the stack, the virtual memory segments, and the VM language on a given hardware or software platform. This exercise serves to stretch the students' grasp of the VM model as well as their design skills. How to represent the stack? How to implement operations on the stack? How to represent and manage the stack pointer? How to parse VM commands? Many such issues must be addressed in any VM implementation project.

In order to structure this undertaking as well as its grading, we have specified a modular API for a proposed VM implementation, and developed two projects that walk the students through its gradual construction and unit-testing [6]. The first project implements stack handling, and the second project implements program control. The two projects are independent of each other, and can be done in isolation. The implementation proper can be done in any language, two popular choices being Java and Perl.

To sum up, the API and the projects describe a gradual implementation of a sequence of VM commands of increasing complexity. For example, one can settle for implementing a basic VM language consisting of nothing more than `push`, `pop`, and stack arithmetic commands. Next, one can extend this basic language with `label` and `goto` commands. Finally, one can implement the function `call` and `return` commands. Any one of these implementation projects yields a functioning language that can be debugged and unit-tested, using VM programs and test scripts available in the course web site.

SUMMARY

The approach and the resources described in this paper are presently used in many university-level CS courses as well as self-organized, web-based CS courses. Teaching the VM module requires two 3-hour lectures typically spread over two semester-weeks. The first lecture is devoted to two topics: introducing and motivating the role that virtual machines play in modern software engineering, and describing VM constructs and stack arithmetic. The second lecture is devoted to program control and implementation issues. Each lecture is accompanied by an independent project. All the lecture and project materials can be found in editable form in [1].

Students' reaction, as reflected through their course evaluation feedbacks, has been exceptionally positive. Typical comments include: "I finally understand how Java programs compile and run on my computer", "The implementation projects improved my programming skills considerably", "The VM emulator is an excellent teaching tool", etc.

Based on this experience, we feel that our VM model and teaching materials can effectively augment existing CS courses that deal with such topics as software engineering, compilation, programming languages, and computer organization. The materials are highly modular, so that different instructors can opt to teach different subsets of them, according to time and scope constraints.

Acknowledgements: I am indebted to my colleague Noam Nisan, the architect of our VM model, and to our teaching assistants Yaron Ukrainitz and Yannai Gonczarowski, who implemented our VM emulator and supported our work with great talent and dedication.

REFERENCES

[1] Nisan, N., Schocken, S. 2005. *TECS Course Web Site*. www.idc.ac.il/tecs

[2] Wing, J. M. 2006.Computational Thinking, CACM 49(3), pp. 33-35.

[3] Hazzan, O. and Tomayko, J. 2005. Reflection and Abstraction Processes in the Learning of the Human Aspects of Software Engineering, IEEE Computer 38(6), pp. 39-45.

[4] Kramer, J. 2007. Is abstraction the key to computing?, *Communications of the ACM*, Vol. 50, No. 4, pp. 37-42.

[5] Meyer, B. 2000. Object-Oriented Software Construction, Prentice-Hall.

[6] Computing Curricula 2001. Final Report, IEEE Computer Society/ACM. (2001). www.sigcse.org/cc2001, Page 27.

[7] Nisan, N., Schocken, S. 2005. The Elements of Computing Systems. Cambridge, MA: MIT Press.