VM Emulator Tutorial

This program is part of the software suite that accompanies the book

*The Elements of Computing Systems*

by Noam Nisan and Shimon Schocken

MIT Press

[www.nand2tetris.org](http://www.nand2tetris.org)

This software was developed by students at the Efi Arazi School of Computer Science at IDC

Chief Software Architects: Yaron Ukrainitz and Yannai Gonczarowski
Background

*The Elements of Computing Systems* evolves around the construction of a complete computer system, done in the framework of a 1- or 2-semester course.

In the first part of the book/course, we build the hardware platform of a simple yet powerful computer, called Hack. In the second part, we build the computer’s software hierarchy, consisting of an assembler, a virtual machine, a simple Java-like language called Jack, a compiler for it, and a mini operating system, written in Jack.

The book/course is completely self-contained, requiring only programming as a pre-requisite.

The book’s web site includes some 200 test programs, test scripts, and all the software tools necessary for doing all the projects.
The Book’s Software Suite

(All the supplied tools are dual-platform: Xxx.bat starts Xxx in Windows, and Xxx.sh starts it in Unix)

Simulators (HardwareSimulator, CPUEmulator, VMEmulator):

- Used to build hardware platforms and execute programs;
- Supplied by us.

Translators (Assembler, JackCompiler):

- Used to translate from high-level to low-level;
- Developed by the students, using the book’s specs; Executable solutions supplied by us.

Other

- Bin: simulators and translators software;
- builtin: executable versions of all the logic gates and chips mentioned in the book;
- os: executable version of the Jack OS;
- TextComparer: a text comparison utility.

This tutorial is about the VM emulator
I. Getting Started

II. Using Scripts

III. Debugging

Relevant reading (from *The Elements of Computing Systems*):

- Chapter 7: *Virtual Machine I: Stack Arithmetic*
- Chapter 8: *Virtual Machine II: Program Control*
- Appendix B: *Test Scripting Language, Section 4.*
Part I: Getting Started
The Typical Origin of VM Programs

- VM programs are normally written by compilers
- For example, the Jack compiler (chapters 10-11) generates VM programs
- The VM program can be translated further into machine language, and then executed on a host computer
- Alternatively, the same VM program can be emulated as-is on a VM emulator.

![Diagram showing the typical origin of VM programs.](image-url)
Example: Pong game (user view)

Ball moves and bounces off the walls “randomly”

User move the bat left and right, trying to hit the ball

Number of successful hits

Score: 4

Now let’s go behind the scene ...
VM Emulator at a Glance

The VM emulator serves three purposes:
- Running programs
- Debugging programs
- Visualizing the VM's anatomy

The emulator's GUI is rather crowded, but each GUI element has an important debugging role.
Loading a VM Program

Let's start with a trivial VM program that manipulates only the stack (i.e. does not involve the memory segments Static, Local, Argument, etc.);

VM programs that don't manipulate memory segments can be loaded via the "load file" button.

Navigate to a directory and select a .vm file.
Running a Program

**VM code is loaded:** (read-only)
The index on the left is the location of the VM command within the VM code (a GUI effect, not part of the code).

**Default test script**
Always loaded, unless another script is loaded by the user.
Running a Program

Impact of first 13 “vmsteps”
Loading a Multi-File Program

- Most VM programs, like Pong, consist of more than one `.vm` file. For example, the Jack compiler generates one `.vm` file for each `.jack` class file, and then there are all the `.vm` files comprising the operating system. All these files must reside in the same directory.
- Therefore, when loading a multi-file VM program into the VM emulator, one must load the entire directory.

Won’t work!
Why? Because Pong is a multi-file program, and ALL these files must be loaded. Solution: navigate back to the directory level, and load it.
Loading a Multi-File Program
Part II: Virtual Memory Segments
A technical point to keep in mind:

- Most VM programs include `pop` and `push` commands that operate on `Static`, `Local`, `Argument`, etc.;
- In order for such programs to operate properly, VM implementations must initialize the memory segments’ bases, e.g. anchor them in selected addresses in the host RAM;
- Case 1: the loaded code includes function calling commands. In this case, the VM implementation takes care of the required segment initializations in run-time, since this task is part of the VM function call-and-return protocol;
- Case 2: the loaded code includes no function calling commands. In this case, the common practice is to load the code through a test script that handles the necessary initialization externally.

Memory segments:
- The VM emulator displays the states of 6 of the 8 VM’s memory segments;
- The `Constant` and `Pointer` segments are not displayed.
Part II: Using Scripts
Typical VM Script

Simulation step
(a series of script commands ending with a semicolon)

Next simulation step

Repeated simulation step

load BasicTest.vm,
output-file BasicTest.out,
compare-to BasicTest.cmp,
output-list RAM[256]%D1.6.1
   RAM[300]%D1.6.1 RAM[401]%D1.6.1
   RAM[402]%D1.6.1 RAM[3006]%D1.6.1
   RAM[3012]%D1.6.1
   RAM[3015]%D1.6.1 RAM[11]%D1.6.1;

set sp 256,
set local 300,
set argument 400,
set this 3000,
set that 3010;

repeat 25 {
   vmstep,
   output;
}

Typical "script setup" commands

Typical memory segments initialization commands

Typical execution loop
Loading a Script

Navigate to a directory and select a .tst file.
Script Controls

Execution speed control

Script = a series of simulation steps, each ending with a semicolon;

Reset the script

Pause the simulation

Execute step after step repeatedly

Execute the next simulation step
Running the Script

Loads a VM program into the emulator
Running the Script

VM code is loaded
Running the Script

The memory segments were initialized (their base addresses were anchored to the RAM locations specified by the script).

A loop that executes the loaded VM program
Running the Script

Impact after first 10 commands are executed
Part III: Debugging
When the script terminates, the comparison of the script output and the compare file is reported.
Animation Options

- **Program flow** (default): highlights the next VM command to be executed;
- **Program & data flow**: highlights the next VM command and animates data flow;
- **No animation**: disables all animation

**Usage tip:** To execute any non-trivial program quickly, select no animation.
Breakpoints: a Powerful Debugging Tool

The VM emulator keeps track of the following variables:

- **segment[i]**: Where segment is either `local`, `argument`, `this`, `that`, or `temp`
- **local, argument, this, that**: Base addresses of these segments in the host RAM
- **RAM[i]**: Value of this memory location in the host RAM
- **sp**: Stack pointer
- **currentFunction**: Full name (inc. fileName) of the currently executing VM function
- **line**: Line number of the currently executing VM command

**Breakpoints:**

- A breakpoint is a pair `<variable, value>` where `variable` is one of the labels listed above (e.g. `local[5]`, `argument`, `line`, etc.) and `value` is a valid value
- Breakpoints can be declared either interactively, or via script commands
- For each declared breakpoint, when the `variable` reaches the `value`, the emulator pauses the program’s execution with a proper message.
Setting Breakpoints

1. Open the breakpoint panel

2. Previously-declared breakpoints

3. Add, delete, or update breakpoints

4. Select the variable on whose value you wish to break

By convention, function headers are colored violet

Here the violet coloring is overridden by the yellow “next command” highlight.

A simple VM program: `Sys.init` calls `Main.main`, that calls `Main.add` (header not seen because of the scroll), that does some simple stack arithmetic.

5. Enter the value at which the break should occur
Setting Breakpoints

**Breakpoints logic:**
When `local[1]` will become 8, or when `sp` will reach 271, or when the command in line 13 will be reached, or when execution will reach the `Main.add` function, the emulator will pause the program’s execution.
Breakpoints in Action

Execution reached the `Main.add` function, an event that triggers a display of the breakpoint and execution pause.
Breakpoints in Action

Following some push and pop commands, the stack pointer (sp) became 271, an event that triggers a display of the breakpoint and execution pause.
Breakpoints in Action

Following some more execution, the second local variable (local[1]) became 8, an event that triggers a display of the breakpoint and execution pause. A powerful debugging tool!
Breakpoints in Scripts

For systematic and replicable debugging, use scripts

The first script commands usually load the .vm program and set up for the simulation

The rest of the script may use various debugging-oriented commands:

• Write variable values (output)
• Repeated execution (while)
• Set/clear Breakpoints
• Etc. (see Appendix B.)

```
load myProg.vm,
output-file myProg.out,
output-list sp%D2.4.2
   CurrentFunction%S1.15.1
   Argument[0]%D3.6.3
   RAM[256]%D2.6.2;

breakpoint currentFunction Sys.init,

set RAM[256] 15,
set sp 257;

repeat 3 {
   vmStep,
}
output;

while sp < 260 {
   vmstep;
}
output;

clear-breakpoints;

// Etc.
```
End-note on Creating Virtual Worlds

“It’s like building something where you don’t have to order the cement. You can create a world of your own, your own environment, and never leave this room.”

(Ken Thompson, 1983 Turing Award lecture)