



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

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.

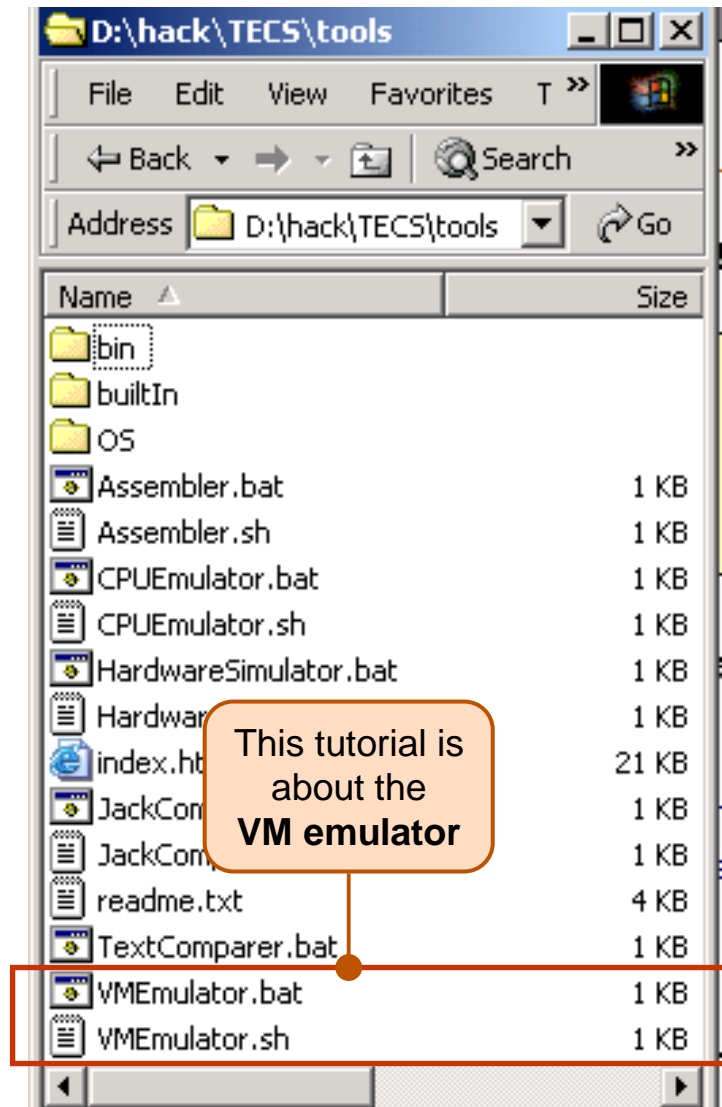
In the first part of the book/course, we build the hardware platform of a simple yet powerful computer, called Hack. In the second part, we build the computer's software hierarchy, consisting of an assembler, a virtual machine, a simple Java-like language called Jack, a compiler for it, and a mini operating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

VM Emulator Tutorial

- I. [Getting Started](#)
- II. [Using Scripts](#)
- III. [Debugging](#)

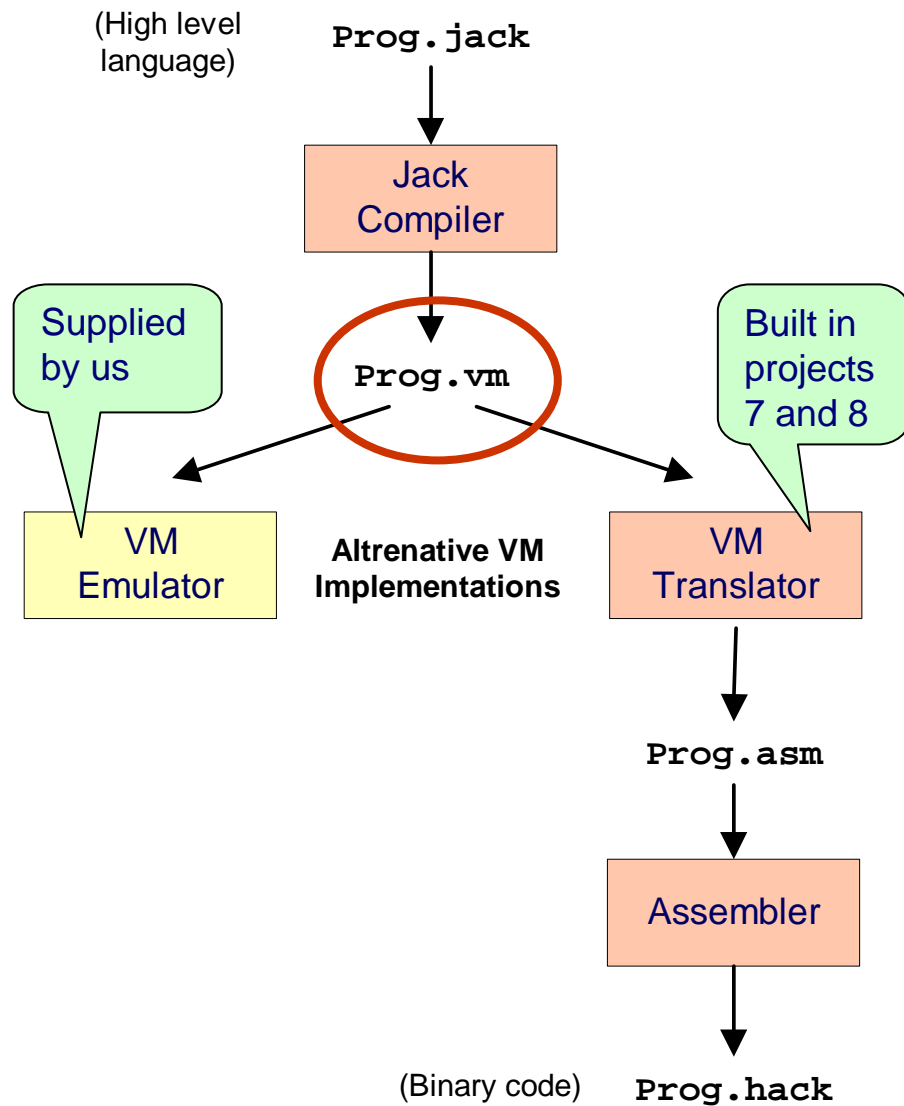
Relevant reading (from *The Elements of Computing Systems*):

- Chapter 7: *Virtual Machine I: Stack Arithmetic*
- Chapter 8: *Virtual Machine II: Program Control*
- Appendix B: *Test Scripting Language, Section 4.*

VM Emulator Tutorial

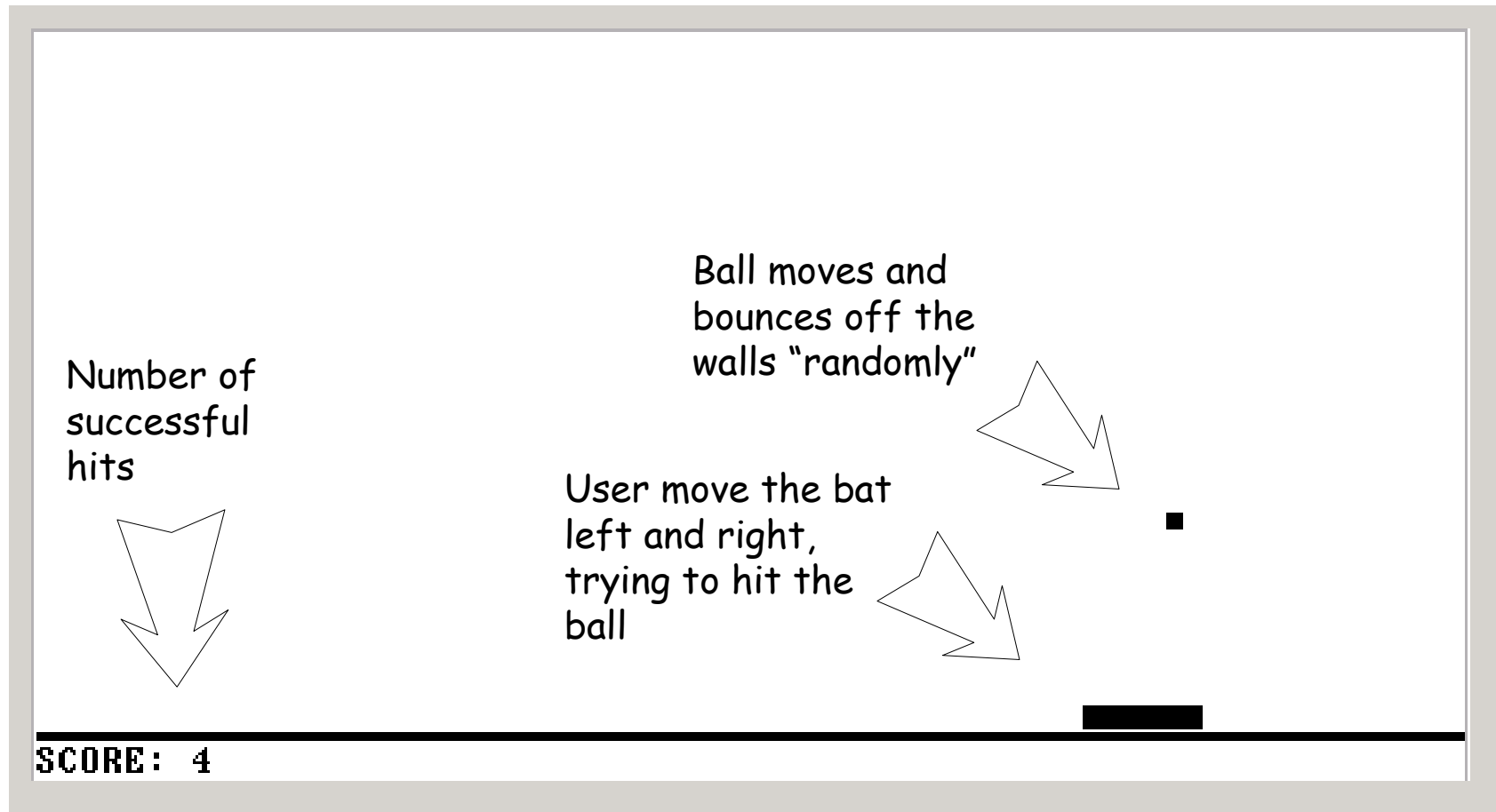


The Typical Origin of VM Programs



- 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.

Example: Pong game (user view)



Now let's go behind the scene ...

VM Emulator at a Glance

VM program
(In this example: Pong code + OS code)

Screen:
(In this example: Pong game action)

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.

global stack, as seen by the VM program

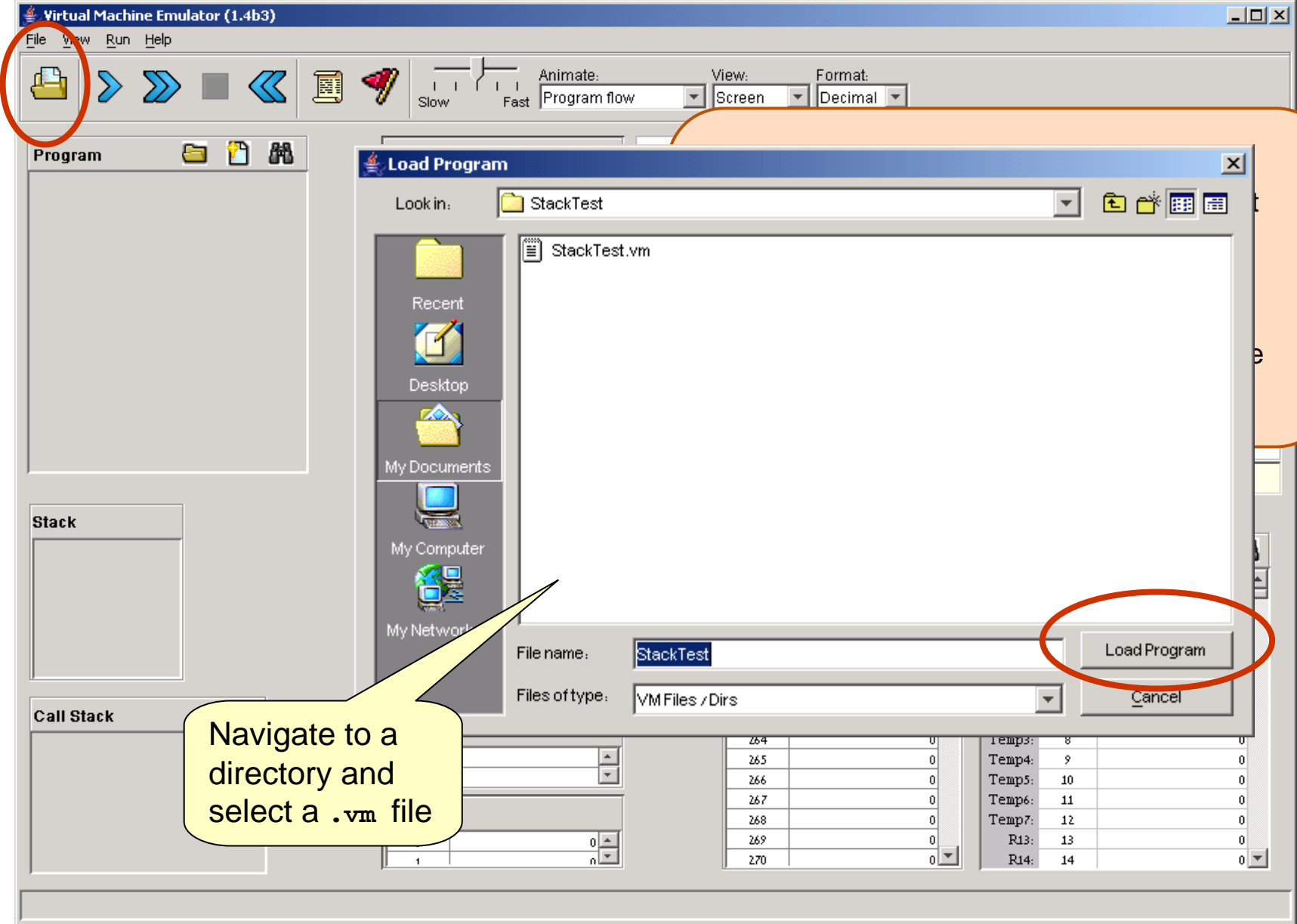
Call stack:
Hierarchy of all the functions that are currently running

Global stack:
Function frames + working stack

Host RAM:
Stores the global stack, heap, etc.

Not Part of the VM!
(displayed in the VM emulator for reference purposes)

Loading a VM Program



Running a Program

The screenshot shows the Virtual Machine Emulator (1.4b3) interface. The title bar indicates the file path: G:\projects\07\StackArithmetic\StackTest\StackTest.vm. The menu bar includes File, View, Run, and Help. The toolbar contains icons for file operations, execution (a red circle highlights the play button), and animation controls. The main window is divided into several panels:

- Program:** A table listing VM commands and their arguments.
- Script:** A text area containing the default test script.
- Local:** A table for local variables.
- Stack:** A table for the current stack.
- Call Stack:** A table for the call stack.
- That:** A table for the 'that' pointer.
- Temp:** A table for temporary registers.
- RAM:** A table for RAM memory.

Callouts provide additional information:

- Script controls:** Points to the execution and animation buttons in the toolbar.
- VM code is loaded: (read-only):** Points to the Program table, stating that 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:** Points to the Script area, stating it is always loaded, unless another script is loaded by the user.

Running a Program

The screenshot shows the Virtual Machine Emulator interface. The 'Program' list on the left contains 15 steps, with step 13 ('push constant 112') highlighted in yellow. A green arrow points from this step to the 'Global Stack' table, where row 261 (address 261, value 53) is highlighted in yellow. An orange callout box with the text 'Impact of first 13 "vmsteps"' has two lines pointing to the highlighted row in the Global Stack and the highlighted step in the Program list.

0	push	constant 17
1	push	constant 17
2	eq	
3	push	constant 892
4	push	constant 891
5	lt	
6	push	constant 32767
7	push	constant 32766
8	gt	
9	push	constant 56
10	push	constant 31
11	push	constant 53
12	add	
13	push	constant 112
14	sub	

0		0
1		0
2		0
3		0
4		0

0		0
1		0
2		0
3		0
4		0

0		0
1		0
2		0
3		0
4		0

0		0
1		0
2		0
3		0
4		0

-1	
0	
-1	
56	
84	

256		-1
257		0
258		-1
259		56
260		84
261		53
262		0
263		0
264		0
265		0
266		0
267		0
268		0
269		0
270		0

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

Loading a Multi-File Program

Virtual Machine Emulator (1.4b1)

File View Run Help

Program

Static

0	
1	
2	
3	
4	

Local

0	256
1	0
2	0
3	0
4	0

Argument

0	256
1	0

Working Stack

0	256
1	0

Call Stack

0	256
1	0

Temp

0	0
1	0

Temp2: 7

Temp3: 8

Temp4: 9

Temp5: 10

Temp6: 11

Temp7: 12

R13: 13

R14: 14

Look in: Pong

Array.vm

Ball.vm

Bat.vm

Keyboard.vm

Main.vm

File name: Pong

Files of type: VM Files / Dirs

Load Program

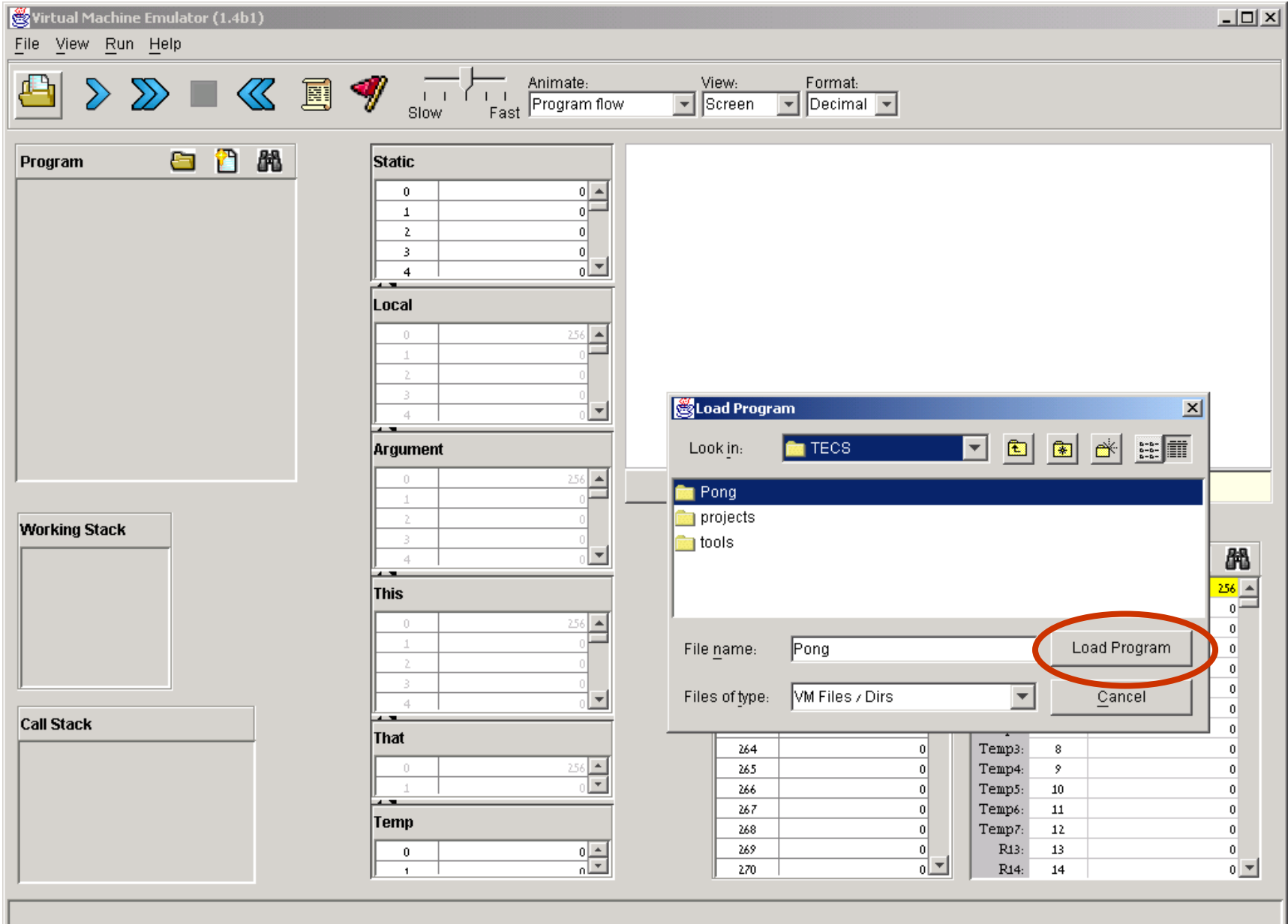
Cancel

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.

- 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*.

Loading a Multi-File Program





Virtual Memory Segments

The screenshot shows a Virtual Machine Emulator window titled "Virtual Machine Emulator (1.4b3) - G:\examples\Pong". The interface includes a menu bar (File, View, Run, Help), a toolbar with navigation and execution controls, and a main display area. On the left, a "Program" list shows instructions from line 93 to 103, with line 97 highlighted. On the right, six memory segments are displayed: Static, Local, Argument, This, That, and Temp. Each segment contains a table of indices and values. The "Static" segment has values 2064 and 2048. The "Local" segment has values 4, 0, 0, and -1. The "Argument" segment has values 361 and 16. The "This" segment has values 362, 229, 50, 7, and 2. The "That" segment has values 512 and 0. The "Temp" segment has values 512 and n.

Program

93	push	local 0
94	push	constant 1
95	add	
96	pop	local 0
	label	Math.divide\$IF_FAL...
	label	Math.divide\$IF_FAL...
97	goto	Math.divide\$WHILE_...
	label	Math.divide\$WHILE_...
	label	Math.divide\$WHILE_...
98	push	local 0
99	push	constant 1
100	neg	
101	gt	
102	not	
103	if-goto	Math.divide\$WHILE_...

Static

0	2064
1	2048

Local

0	4
1	0
2	0
3	-1

Argument

0	361
1	16

This

0	362
1	229
2	50
3	7
4	2

That

0	512
1	0

Temp

0	512
1	n

Memory segments:

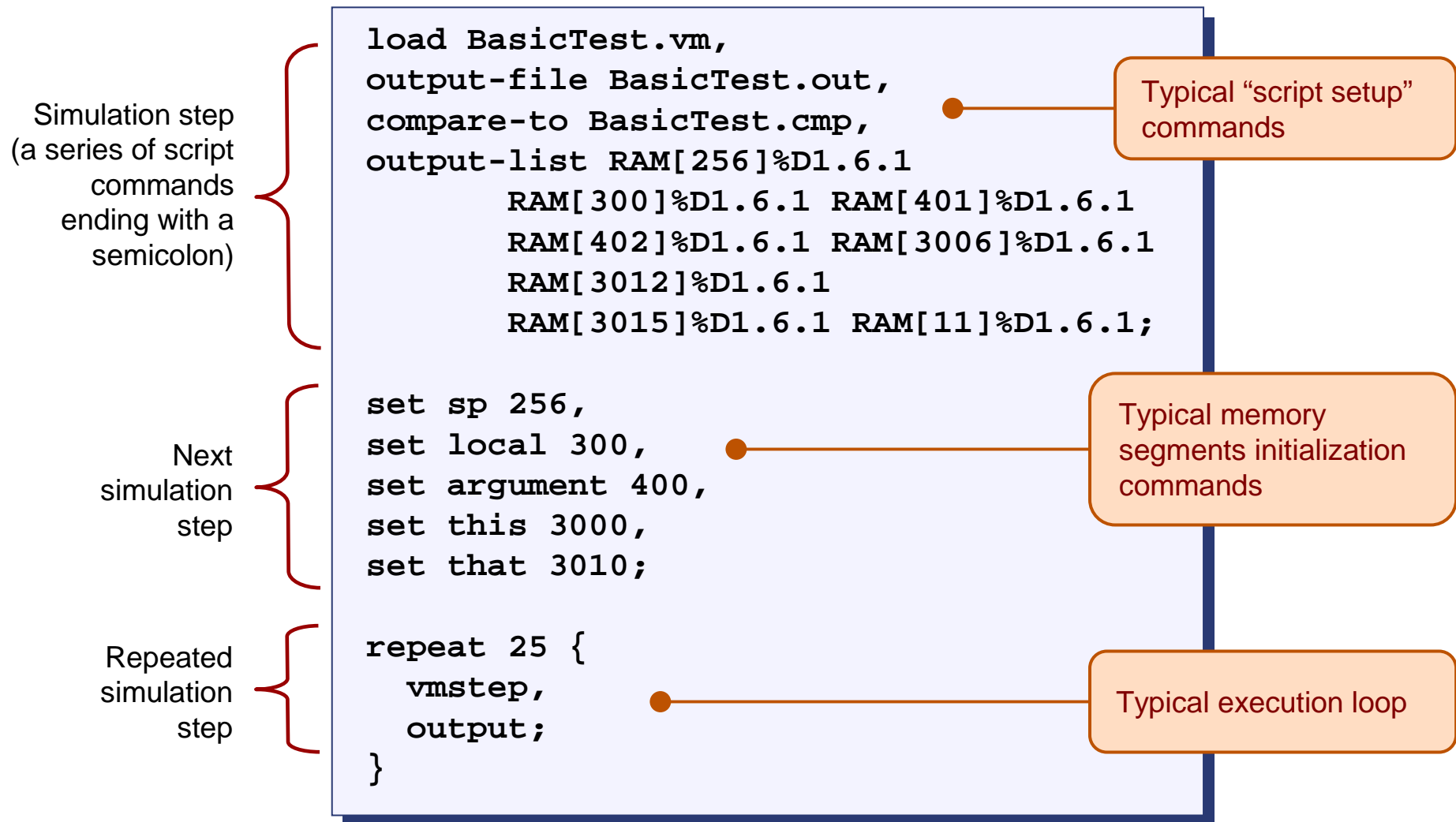
- The VM emulator displays the states of 6 of the 8 VM's memory segments;
- The **Constant** and **Pointer** segments are not displayed.

A technical point to keep in mind:

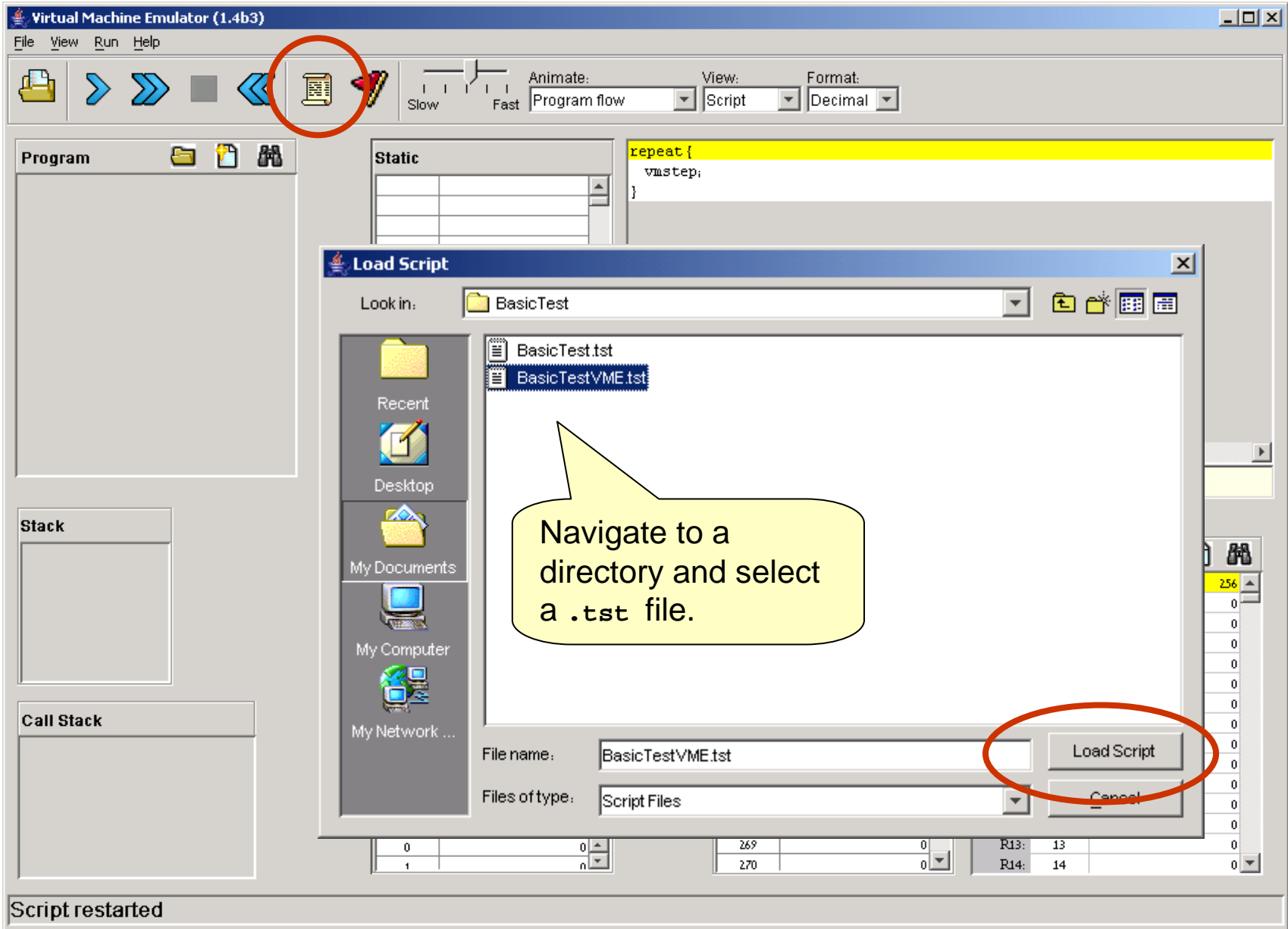
- Most VM programs include **pop** and **push** commands that operate on **Static**, **Local**, **Argument**, etc.;
- In order for such programs to operate properly, VM implementations must initialize the memory segments' bases, e.g. anchor them in selected addresses in the host RAM;
- Case 1: the loaded 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.



Typical VM Script



Loading a Script



Script Controls

The screenshot shows the Virtual Machine Emulator (1.4b3) interface. The top toolbar contains several icons for script control: a single right arrow, a double right arrow, a square, a double left arrow, a document, and a red flag. A callout points to the double right arrow icon, stating "Execution speed control". Below the toolbar, the "Program" panel has a callout pointing to the square icon, stating "Reset the script". The "Static" panel has a callout pointing to the double left arrow icon, stating "Pause the simulation". The "Argument" panel has a callout pointing to the double right arrow icon, stating "Execute step after step repeatedly". The "Call Stack" panel has a callout pointing to the single right arrow icon, stating "Execute the next simulation step". The main script editor displays the following code:

```
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;  
}
```

A callout points to the script editor, stating "Script = a series of simulation steps, each ending with a semicolon;". The "Global Stack" and "RAM" panels are also visible at the bottom right.

New script loaded: G:\projects\07\MemoryAccess\BasicTest\BasicTestVME.tst

Running the Script

Virtual Machine Emulator (1.4b3)

File View Run Help

Slow Fast Animate: Program flow View: Script Format: Decimal

Program

Static

Index	Value
0	0
1	0
2	0
3	0
4	0

Local

Index	Value
0	0
1	0
2	0
3	0
4	0

Argument

Index	Value
0	0
1	0
2	0
3	0
4	0

This

Index	Value
0	0
1	0
2	0
3	0
4	0

That

Index	Value
0	0
1	0

Temp

Index	Value
0	0
1	0

```
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;
```

Global Stack

Address	Value
256	0
257	0
258	0
259	0
260	0
261	0
262	0
263	0
264	0
265	0
266	0
267	0
268	0
269	0
270	0

RAM

Register	Value
SP:	0
LCL:	1
ARG:	2
THIS:	3
THAT:	4
Temp0:	5
Temp1:	6
Temp2:	7
Temp3:	8
Temp4:	9
Temp5:	10
Temp6:	11
Temp7:	12
R13:	13
R14:	14

Stack

Call Stack

New script loaded: G:\projects\07\MemoryAccess\BasicTest\BasicTestVME.tst

Running the Script

Virtual Machine Emulator (1.4b3) - G:\projects\07\MemoryAccess\BasicTest\BasicTest.vm

File View Run Help

Slow Fast Animate: Program flow View: Script Format: Decimal

Program

0	push	constant 10
1	pop	local 0
2	push	constant 21
3	push	constant 22
4	pop	argument 2
5	pop	argument 1
6	push	constant 36
7	pop	this 6
8	push	constant 42
9	push	constant 45
10	pop	that 5
11	pop	that 2
12	push	constant 510
13	pop	temp 6
14	push	local 0

Static

Local

Argument

This

That

Temp

```
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;
```

Stack

Call Stack

Global Stack

256	0
257	0
258	0
259	0
260	0
261	0
262	0
263	0
264	0
265	0
266	0
267	0
268	0
269	0
270	0

RAM

SP:	0	256
LCL:	1	0
ARG:	2	0
THIS:	3	0
THAT:	4	0
Temp0:	5	0
Temp1:	6	0
Temp2:	7	0
Temp3:	8	0
Temp4:	9	0
Temp5:	10	0
Temp6:	11	0
Temp7:	12	0
R13:	13	0
R14:	14	0

VM code is loaded

Running the Script

(click a few times)

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

Program

0	push	constant 10
1	pop	local 0
2	push	constant 21
3	push	constant 22
4	pop	argument 2
5	pop	argument 1
6	push	constant 36
7	pop	this 6
8	push	constant 42
9	push	constant 45
10	pop	that 5
11	pop	that 2
12	push	constant 510
13	pop	temp 6
14	push	local 0

Stack

10

Call Stack

Static

Local

0	0
1	0
2	0
3	0
4	0

Argument

0	0
1	0
2	0
3	0
4	0

This

0	0
1	0
2	0
3	0
4	0

That

0	0
1	0

Temp

0	0
1	0

Global Stack

256	10
257	0
258	0
259	0
260	0

RAM

SP:	0	257
LCL:	1	300
ARG:	2	400
THIS:	3	3000
THAT:	4	3010
Temp0:	5	0
Temp1:	6	0
Temp2:	7	0
Temp3:	8	0
Temp4:	9	0
Temp5:	10	0
Temp6:	11	0
Temp7:	12	0
R13:	13	0
R14:	14	0

```

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;
    
```

Running the Script

The screenshot shows a Virtual Machine Emulator window titled "Virtual Machine Emulator (1.4b3) - G:\projects\07\MemoryAccess\BasicTest\BasicTest.vm". The interface includes a menu bar (File, View, Run, Help), a toolbar with navigation and execution controls, and several panels for monitoring the program's state.

Program Panel: A list of 15 instructions. Instruction 10, "pop that 5", is highlighted in yellow. A large green arrow points to this instruction.

Index	Op	Arg
0	push	constant 10
1	pop	local 0
2	push	constant 21
3	push	constant 22
4	pop	argument 2
5	pop	argument 1
6	push	constant 36
7	pop	this 6
8	push	constant 42
9	push	constant 45
10	pop	that 5
11	pop	that 2
12	push	constant 510
13	pop	temp 6
14	push	local 0

Stack Panel: Shows the current stack with values 42 and 45.

Address	Value
42	42
45	45

Static Panel: Empty.

Local Panel: Shows local variables 0-4 with values 10, 0, 0, 0, 0.

Index	Value
0	10
1	0
2	0
3	0
4	0

Argument Panel: Shows argument variables 0-4 with values 0, 21, 22, 0, 0.

Index	Value
0	0
1	21
2	22
3	0
4	0

This Panel: Shows 'this' variables 2-6 with values 0, 0, 0, 0, 36.

Index	Value
2	0
3	0
4	0
5	0
6	36

That Panel: Shows 'that' variables 0-1 with values 0, 0.

Index	Value
0	0
1	0

Temp Panel: Shows 'temp' variables 0-1 with values 0, 0.

Index	Value
0	0
1	0

Global Stack Panel: Shows global stack memory addresses 256-270. Address 258 is highlighted in yellow with value 0.

Address	Value
256	42
257	45
258	0
259	0
260	0
261	0
262	0
263	0
264	0
265	0
266	0
267	0
268	0
269	0
270	0

RAM Panel: Shows RAM state with SP at 0 and LCL at 258. Other registers (ARG, THIS, THAT, Temp0-7, R13-14) are also listed.

Register	Value
SP	0
LCL	258
ARG	300
THIS	400
THAT	3000
Temp0	4
Temp1	5
Temp2	6
Temp3	7
Temp4	8
Temp5	9
Temp6	10
Temp7	11
R13	12
R14	13

Script Panel: Shows the script being executed. The line "vmstep;" is highlighted in yellow.

```
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;
```

Impact after first 10 commands are executed



View Options

The screenshot shows the Virtual Machine Emulator interface with the following components:

- Program List:**

10	pop	that 5
11	pop	that 2
12	push	constant 510
13	pop	temp 6
14	push	local 0
15	push	that 5
16	add	
17	push	argument 1
18	sub	
19	push	this 6
20	push	this 6
21	add	
22	sub	
23	push	temp 6
24	add	
- Static:**

0	0
1	0
2	0
3	0
4	0
- Local:**

0	10
1	0
2	0
3	0
4	0
- Argument:**

0	0
1	21
2	22
3	0
4	0
- This:**

0	0
1	0
2	42
3	0
- Temp:**

5	0
6	510
- Working Stack:**

472

- Call Stack:** (Empty)
- View Options:**
 - Animate: Program flow
 - View: Script
 - Format: Decimal
- Script Output:**

```

output-file BasicTest.out,
compare-to BasicTest.comp,
output-list RAM[256] RAM[300] RAM[401] RAM[401] RAM[401]
RAM[402] RAM[402] RAM[402] RAM[3012] RAM[3012]
RAM[3015] RAM[3015] RAM[3015];

set SP 256,
set local 300,
set argument 400,
set this 3000,
set that 3010,

repeat 25
  vwnstep
}
    
```
- Comparison Report:**

259	0	THIS:	3	3000
260	0	THAT:	4	3010
261	0	Temp0:	5	0
262	0	Temp1:	6	0
263	0	Temp2:	7	0
264	0	Temp3:	8	0
265	0	Temp4:	9	0
266	0	Temp5:	10	0
267	0	Temp6:	11	510
268	0	Temp7:	12	0
269	0	R13:	13	0
270	0	R14:	14	0

When the script terminates, the comparison of the script output and the compare file is reported.

- View options:
- **Script:** displays the loaded script;
 - **Output:** displays the generated output file;
 - **Compare:** displays the given comparison file;
 - **Screen:** displays the simulated screen.

End of script - Comparison ended successfully

Animation Options

Speed control
(of both execution and animation)

source

transit

destn.

data flow animation related to the last VM command (in this example: push argument 0)

Animation control:

- **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

The screenshot shows the Virtual Machine Emulator (1.4b3) interface. The main window displays a program listing on the left, a static code window in the center, and a breakpoint variables dialog box on the right. The program listing shows instructions like push, pop, add, sub, return, function, call, label, and goto. The static code window shows a repeat loop with vmstep. The breakpoint variables dialog box shows the current function as 'Main.add' and a green checkmark indicating the breakpoint is set. The interface also includes a menu bar (File, View, Run, Help), a toolbar with various icons, and a stack window at the bottom left.

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

5. Enter the value at which the break should occur

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.

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.

The screenshot shows the Virtual Machine Emulator (1.4b3) interface. The main window displays a program listing with the following code:

```
9 push local 0
10 push local 1
11 add
12 pop local 0
13 push local 1
14 push local 0
15 sub
16 return
0 function Main.main 0
1 call Main.add 0
2 return
0 function Sys.init 0
1 call Main.main 0
label Sys.init$INFINITELOOP
2 goto Sys.init$INFINITELOOP
```

The `repeat {` line is highlighted in yellow. The `currentFunction` variable in the Breakpoint Panel is set to `Main.add`. The Breakpoint Panel also shows the following values:

Variable Name	Value
local[1]	8
sp	271
line	13
currentFunction	Main.add

The RAM panel shows the following values:

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

Breakpoints in Action

Virtual Machine Emulator (1.4b3) - G:\examples\add

File View Run Help

Animate: Program flow View: Script Format: Decimal

Program

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

Static

Address	Value
0	0
1	0
2	0

Local

Address	Value
0	0
1	0
2	0

Argument

Address	Value
0	0
1	0
2	0

Stack

Call Stack

- Sys.init
- Main.main
- Main.add

Temp

Address	Value
0	0
1	0

Breakpoint Panel

Variable Name	Value
local[1]	8
sp	271
line	13
currentFunction	Main.add

RAM

Address	Value
SP:	0 269
LCL:	1 266
ARG:	2 261
THIS:	3 0
THAT:	4 0
Temp0:	5 0
Temp1:	6 0
Temp2:	7 0
Temp3:	8 0
Temp4:	9 0
Temp5:	10 0
Temp6:	11 0
Temp7:	12 0
R13:	13 0
R14:	14 0

Breakpoint reached

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

Breakpoints in Action

Virtual Machine Emulator (1.4b3) - G:\examples\add

File View Run Help

Slow Fast Animate: Program flow View: Script Format: Decimal

Program

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

Static

Local

0		15
1		7
2		0

Argument

Stack

	7
	1

Call Stack

- Sys.init
- Main.main
- Main.add

repeat {
vmstep;
}

Breakpoint Panel

Variable Name	Value
local[1]	8
sp	271
line	13
currentFunction	Main.add

RAM

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

Breakpoint reached

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

Virtual Machine Emulator (1.4b3) - G:\examples\add

File View Run Help

Slow Fast Animate: Program flow View: Script Format: Decimal

Program

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

Static

Local

0		15
1		8
2		0

Argument

Breakpoint Panel

Variable Name	Value
local[1]	8
sp	271
line	13
currentFunction	Main.add

Stack

Call Stack

- Sys.init
- Main.main
- Main.add

RAM

SP:	0	269
LCL:	1	266
ARG:	2	261
THIS:	3	0
THAT:	4	0
Temp0:	5	0
Temp1:	6	0
Temp2:	7	0
Temp3:	8	0
Temp4:	9	0
Temp5:	10	0
Temp6:	11	0
Temp7:	12	0
R13:	13	0
R14:	14	0

Breakpoint reached

A powerful debugging tool!

Breakpoints in Scripts

```
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.
```

- 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.)

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)



Ken Thompson (L) and Dennis Ritchie (R)