



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.idc.ac.il/tecs

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

Chief Software Architect: Yaron Ukrainitz



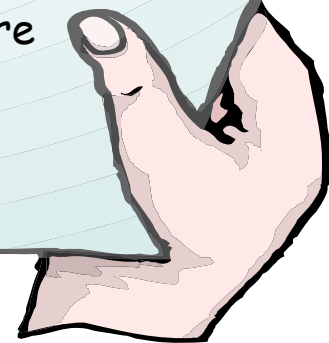
Background

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, VMEulator):

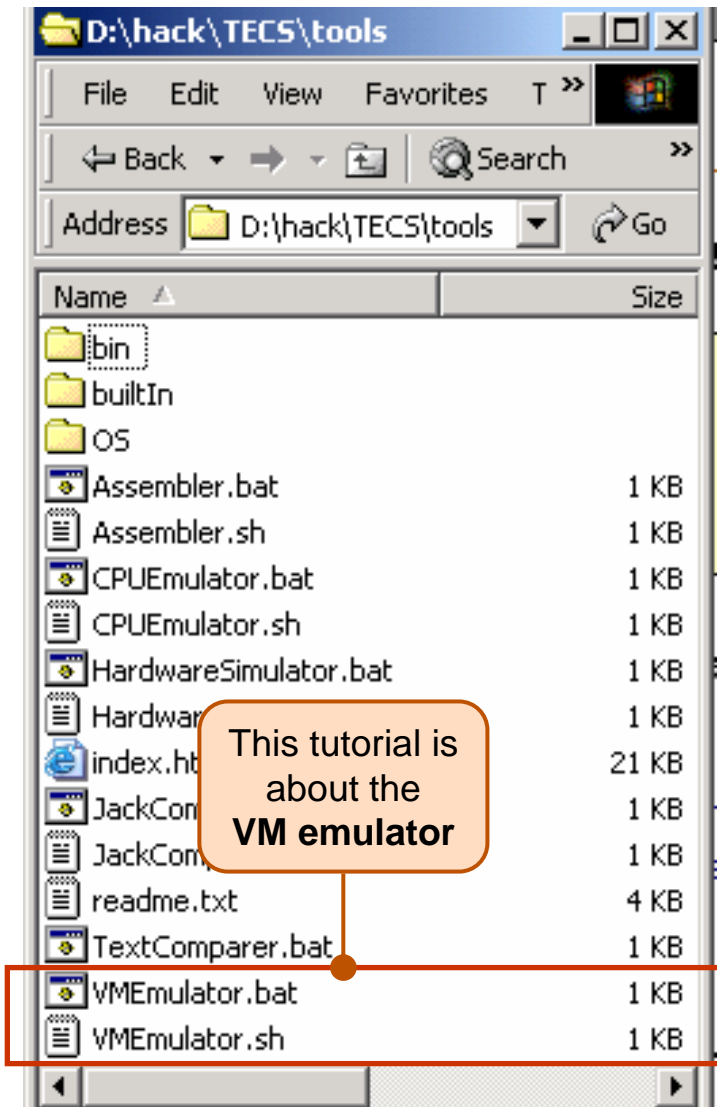
- Used to build hardware platforms and execute programs;
- Supplied by us.

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

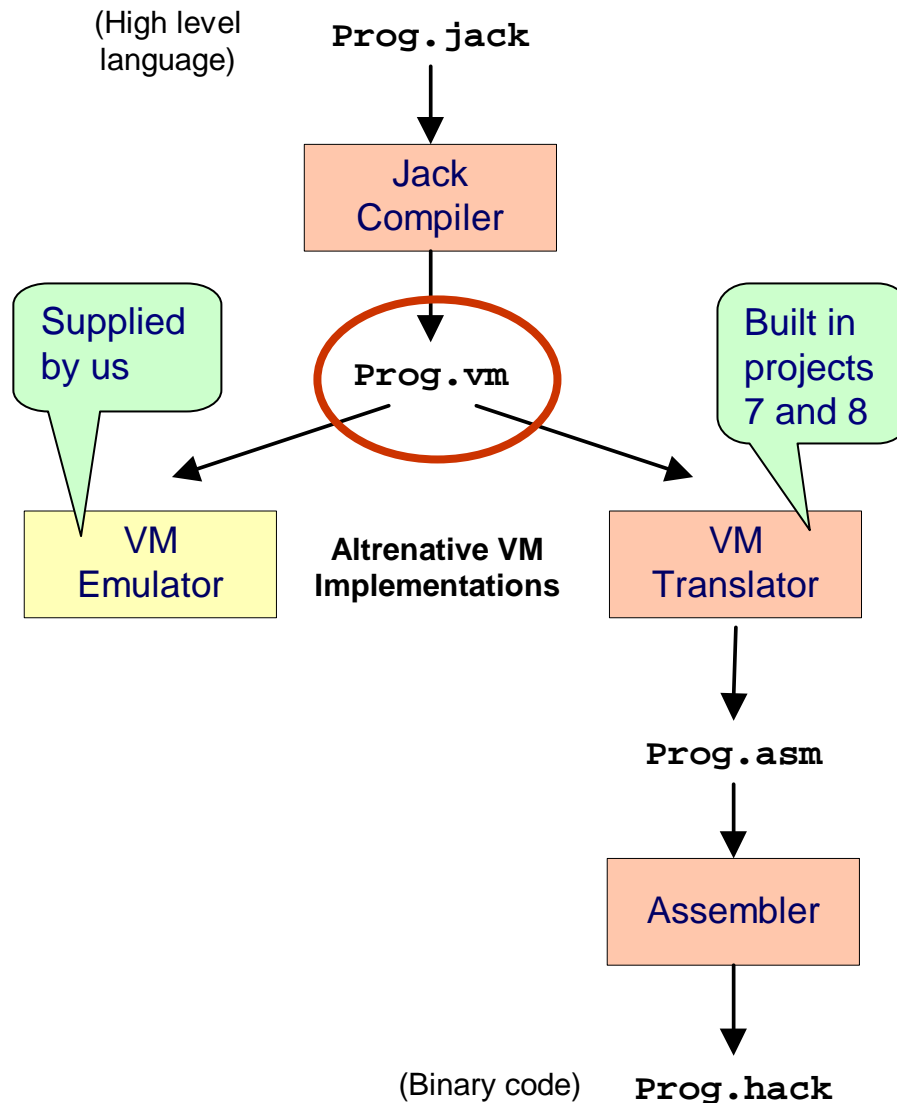
- I. [Getting Started](#)
- II. [Using Scripts](#)
- III. [Debugging](#)

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.*

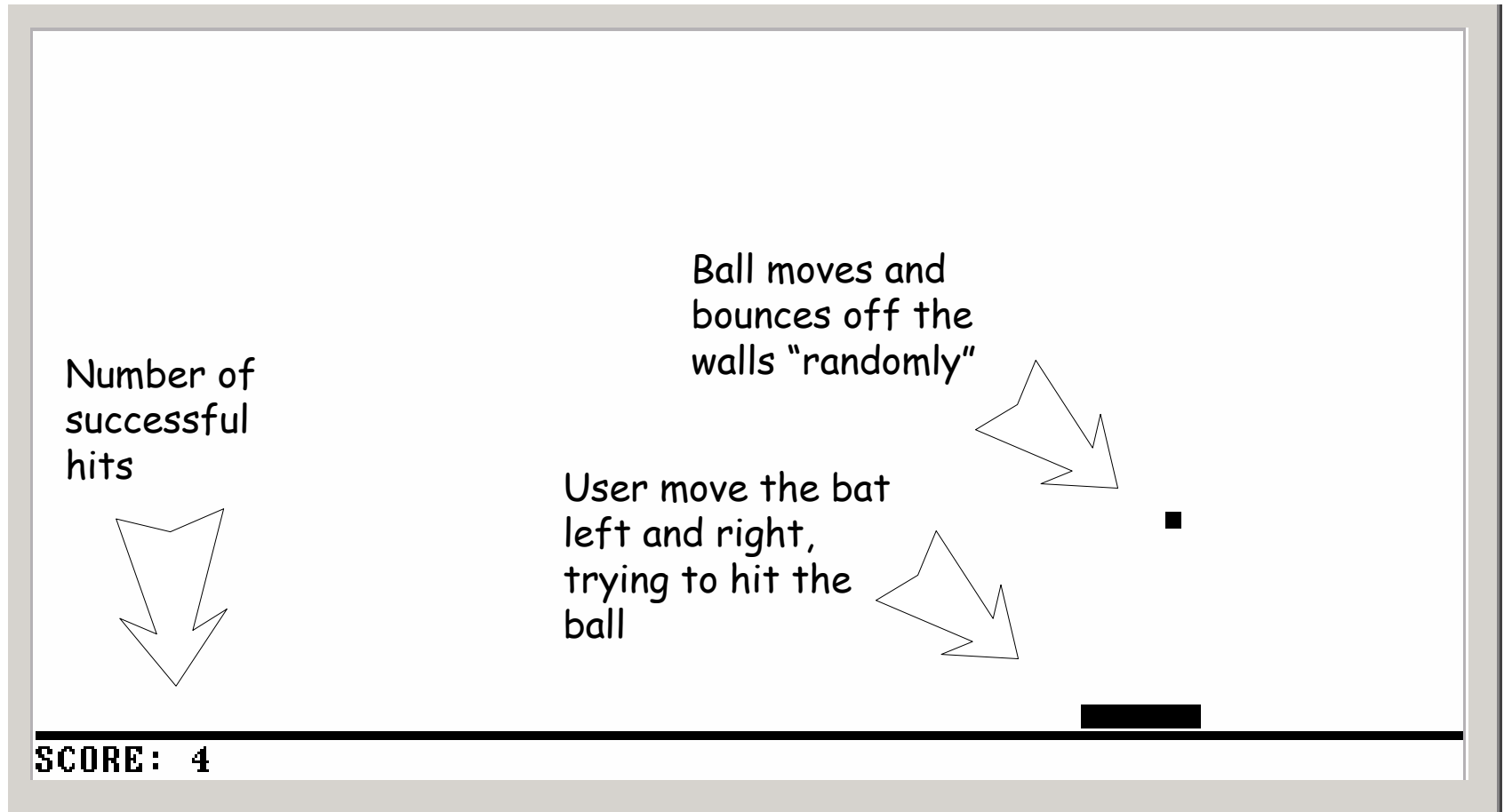


The Typical Origin of VM Programs



- 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

The screenshot shows the Virtual Machine Emulator (1.4b3) interface. The main window title is "Virtual Machine Emulator (1.4b3) - G:\examples\Pong". The menu bar includes File, View, Run, and Help. The toolbar has icons for file operations and execution controls (Slow, Fast, Animate: Program flow, View: Screen, Format: Decimal). The main area is divided into several panels:

- Program:** A list of assembly instructions with addresses 34 to 47. Instruction 41, "add", is highlighted in yellow.
- Static:** A table showing static variables at addresses 2064 and 2048.
- Stack:** A list of stack elements, with the value 2064 highlighted.
- Call Stack:** A list of function frames: Sys.init, Main.main, PongGame.run, Bat.move, Screen.drawRectangle, and Math.multiply (highlighted).
- RAM:** A table showing memory addresses and values, with a section highlighted in light blue.

Annotations and callouts:

- 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 anatomyThe 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)

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 and execution controls. A red box highlights the navigation icons (back, forward, and a play button). Callouts provide detailed information about these elements and the code being executed.

Navigation: Navigate to a directory and select a .vm file

Script controls: Script controls

Code Loading: 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: Default test script
Always loaded, unless another script is loaded by the user.

```
repeat {
  vmstep;
}
```

Index	Command	Value
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	

RAM Address	Value
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

Running a Program

Virtual Machine Emulator (1.4b3) - G:\projects\07\StackArithmetic\StackTest\StackTest.vm

File View Run Help

Animate: Program flow View: Script Format: Decimal

Program

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	

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
		0
		0
		0
		0

Stack

-1	
0	
-1	
56	
84	

Call Stack

Code

```
repeat {  
  vmstep;  
}
```

Global Stack

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

RAM

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

Impact of first 13 "vmsteps"

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

Call Stack

Temp

0	0
1	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	0
268	0	Temp7:	12	0
269	0	R13:	13	0
270	0	R14:	14	0

Load Program

Look in: Pong

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

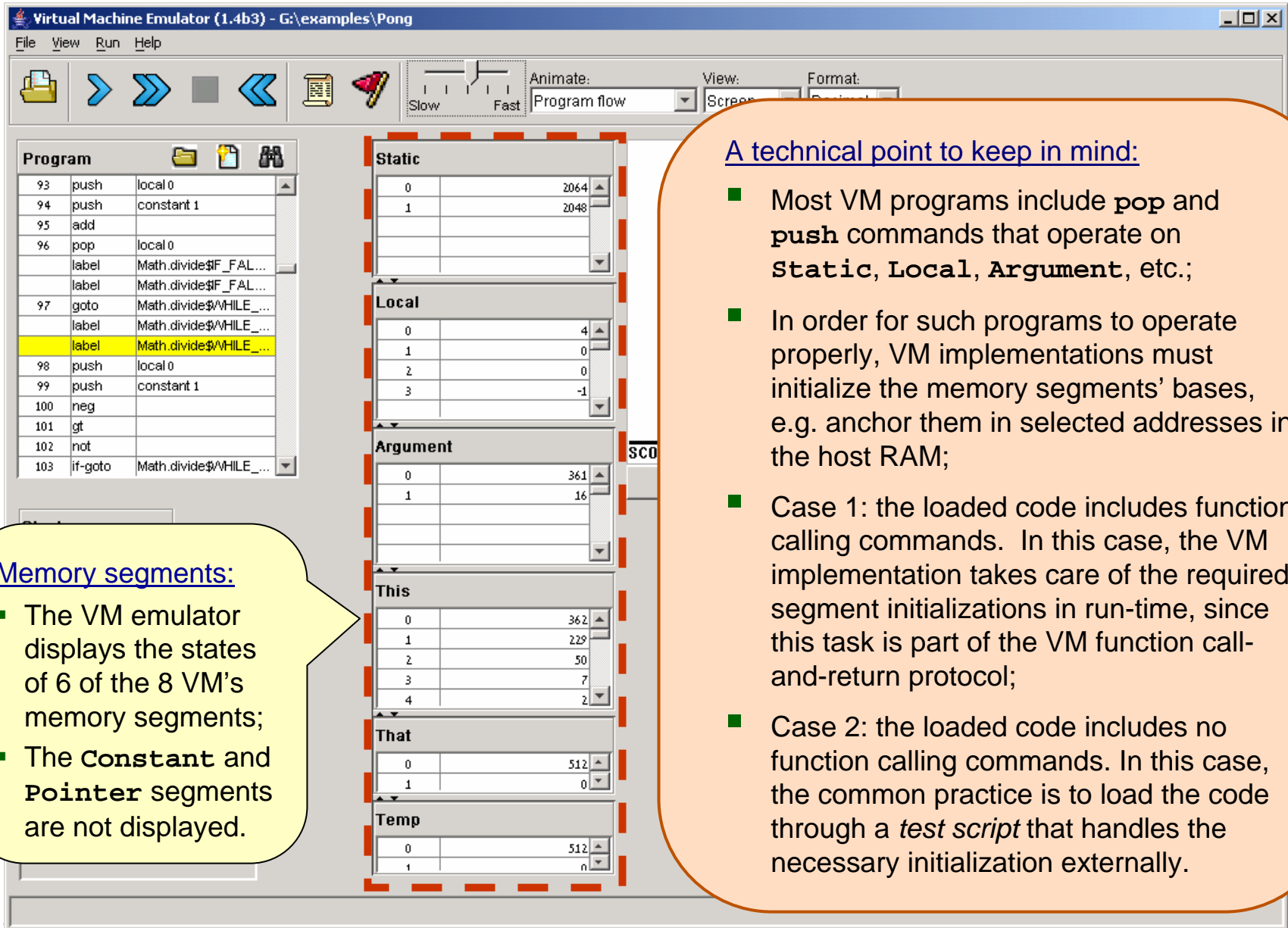
The screenshot shows the Virtual Machine Emulator (1.4b1) interface. The main window has a menu bar (File, View, Run, Help) and a toolbar with various controls. The main area is divided into several panels: Program, Static, Local, Argument, This, That, Temp, Working Stack, and Call Stack. A 'Load Program' dialog box is open, showing the 'Look in:' field set to 'TECS'. The file list shows 'Pong', 'projects', and 'tools'. The 'File name:' field contains 'Pong', and the 'Files of type:' field is set to 'VM Files / Dirs'. The 'Load Program' button is circled in red.

Static	Local	Argument	This	That	Temp
0: 0	0: 256	0: 256	0: 256	0: 256	0: 0
1: 0	1: 0	1: 0	1: 0	1: 0	1: 0
2: 0	2: 0	2: 0	2: 0	2: 0	2: 0
3: 0	3: 0	3: 0	3: 0	3: 0	3: 0
4: 0	4: 0	4: 0	4: 0	4: 0	4: 0

Temp3	Temp4	Temp5	Temp6	Temp7	R13	R14
8	9	10	11	12	13	14



Virtual Memory Segments



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:

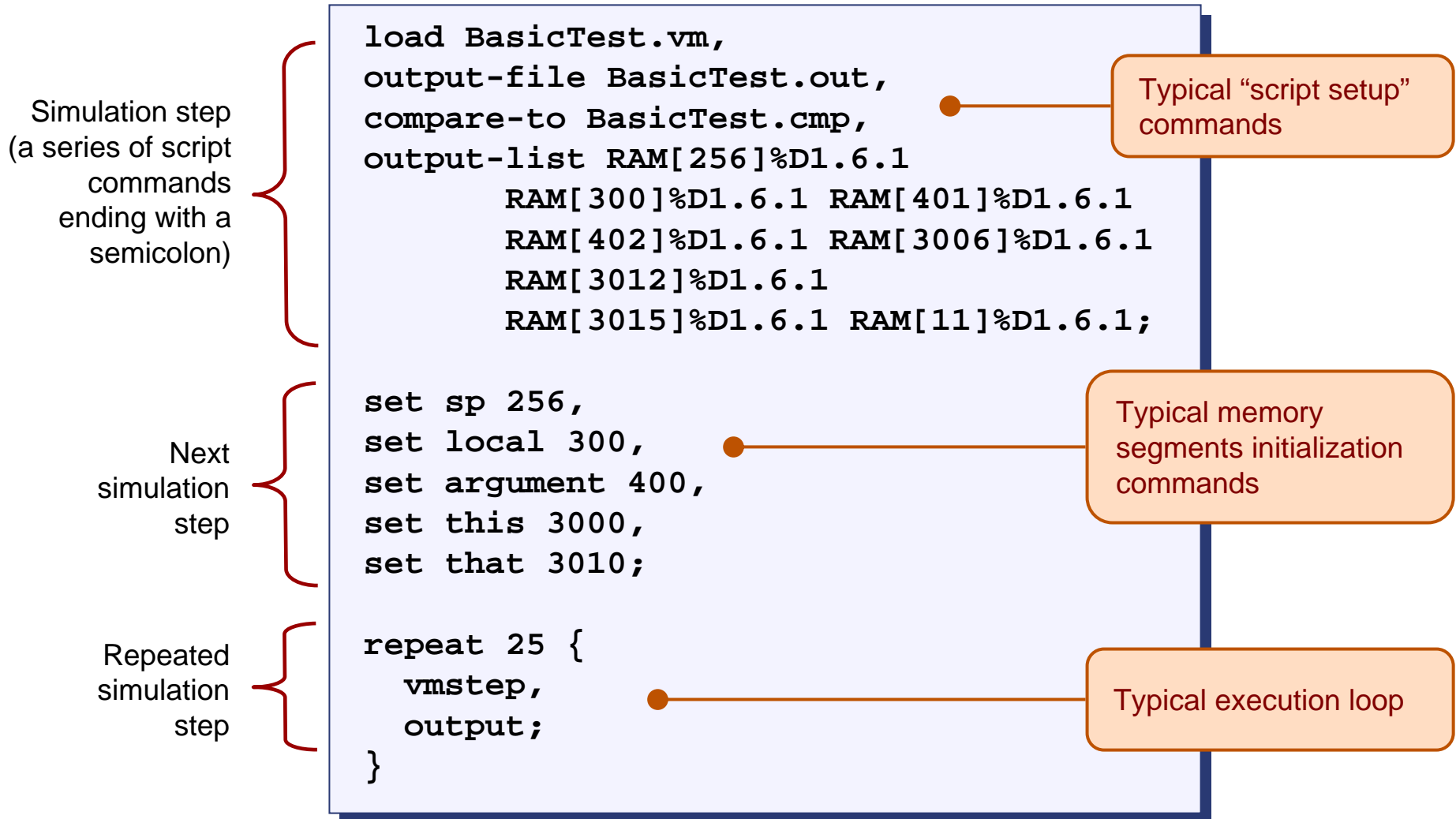
- The VM emulator displays the states of 6 of the 8 VM's memory segments;
- The **Constant** and **Pointer** segments are not displayed.

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

Virtual Machine Emulator (1.4b3)

File View Run Help

Program

Static

```
repeat {  
  vmstep;  
}
```

Look in: BasicTest

BasicTest.tst
BasicTestVME.tst

Recent
Desktop
My Documents
My Computer
My Network ...

File name: BasicTestVME.tst

Files of type: Script Files

Load Script

Cancel

Navigate to a directory and select a .tst file.

Script restarted

0	0	269	0	R13:	13
1	n	270	0	R14:	14

Script Controls

Execution speed control

The screenshot shows the Virtual Machine Emulator (1.4b3) interface. The top menu bar includes File, View, Run, and Help. The toolbar contains icons for file operations, execution (single and double arrows), a pause button, and a reset button. Below the toolbar are several panels: Program, Static, Stack, Call Stack, Global Stack, and RAM. The main window displays a script with the following content:

```
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;  
}
```

Yellow callout boxes provide the following descriptions for the controls:

- Execution speed control:** Points to the 'Animate' section with 'Slow' and 'Fast' options.
- Reset the script:** Points to the double left arrow icon.
- Pause the simulation:** Points to the square icon.
- Execute step after step repeatedly:** Points to the double right arrow icon.
- Execute the next simulation step:** Points to the single right arrow icon.

A yellow callout box explains: "Script = a series of simulation steps, each ending with a semicolon;"

At the bottom, a status bar reads: "New script loaded: G:\projects\07\MemoryAccess\BasicTest\BasicTestVME.tst"

Running the Script

The screenshot shows the VM Emulator interface with the following components:

- Program:** A list of 15 instructions. Instruction 10 is highlighted in yellow.
- Static:** An empty table.
- Local:** A table with 5 rows, indices 0-4, and values 10, 0, 0, 0, 0.
- Argument:** A table with 5 rows, indices 0-4, and values 0, 21, 22, 0, 0.
- This:** A table with 7 rows, indices 2-8, and values 0, 0, 0, 0, 0, 0, 0.
- That:** A table with 2 rows, indices 0-1, and values 0, 0.
- Temp:** A table with 2 rows, indices 0-1, and values 0, n.
- Global Stack:** A table with 11 rows, indices 256-270, and values 42, 45, 0, 0, 0, 0, 0, 0, 0, 0, 0.
- RAM:** A table with 15 rows, indices SP, LCL, ARG, THIS, THAT, Temp0-7, Temp8-14, and values 0, 300, 400, 3000, 3010, 0, 0, 0, 0, 0, 0, 0, 0, 0.

The script in the main window contains the following code:

```
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;
```

Callout boxes provide additional context:

- Impact after first 10 commands are executed:** Points to the Stack window showing values 42 and 45.
- A loop that executes the loaded VM program:** Points to the 'repeat 25' block in the script.
- The memory segments were initialized (their base addresses were anchored to the RAM locations specified by the script):** Points to the RAM window showing initialized values for SP, LCL, ARG, THIS, THAT, and Temp registers.



Animation Options

The screenshot shows the Virtual Machine Emulator (1.4b1) interface. The title bar reads "Virtual Machine Emulator (1.4b1) - G:\TECS\Pong". The menu bar includes "File", "View", "Run", and "Help". The toolbar contains icons for file operations and execution controls, including "Slow" and "Fast" buttons. The "Animate:" dropdown is set to "Program & data flow", "View:" is set to "Screen", and "Format:" is set to "Decimal".

The main window is divided into several panels:

- Program:** A list of instructions. The instruction at address 64, "add", is highlighted in yellow.
- Static:** A table of static variables.
- Argument:** A table of arguments for the current instruction.
- This:** A table of registers.
- Working Stack:** A table showing the current stack state.
- Call Stack:** A table showing the call stack.

Annotations and callouts:

- A yellow callout bubble points to the "Slow" and "Fast" buttons, containing the text: **Speed control** (of both execution and animation).
- An orange callout bubble points to the "Argument" table, containing the text: **source**.
- An orange callout bubble points to the "Working Stack" table, containing the text: **destn.**
- An orange callout bubble points to the "Argument" table, containing the text: **transit**.
- A large yellow callout bubble on the right contains the text: **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*.
- An orange callout bubble at the bottom points to the "Program" table, containing the text: **data flow animation related to the last VM command (in this example: push argument 0)**.

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 $\langle \text{variable}, \text{value} \rangle$ 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

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.

Variable Name	Value
local[1]	8
	271
	13

Name	Value
currentFunction	Main.add

Temp	Value
Temp4:	9
Temp5:	10
Temp6:	11
Temp7:	12
R13:	13
R14:	14

Breakpoints in Action

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.

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

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

Breakpoint reached

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

SP	Value
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

Temp	Value
0	0
1	0

Breakpoints in Scripts

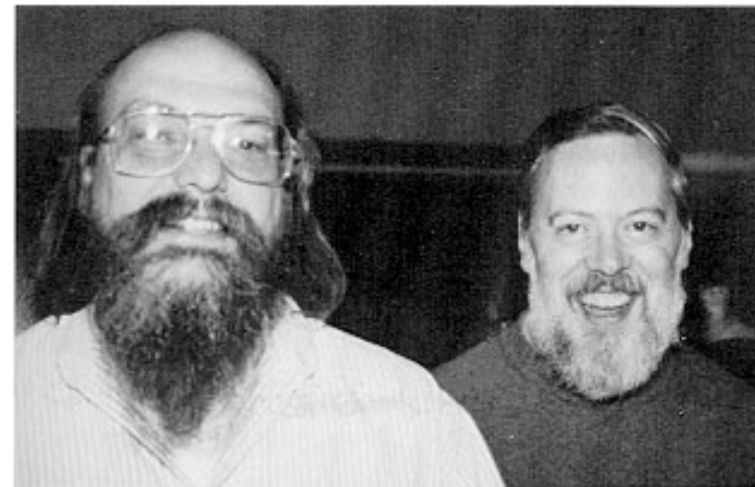
```
load myProg.vm,  
output-file myProg.out,  
output-list sp%D2.4.2  
           CurrentFunction%S1.15.1  
           Argument[0]%D3.6.3  
           RAM[256]%D2.6.2;  
  
breakpoint currentFunction Sys.init,  
  
set RAM[256] 15,  
set sp 257;  
  
repeat 3 {  
    vmStep,  
}  
output;  
  
while 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)