Appendix B: Test Scripting Language

Mistakes are the portals of discovery.
—James Joyce (1882–1941)

Testing is a critically important element of systems development, and one that typically gets little attention in computer science education. In this book we take testing very seriously. In fact, we believe that before one sets out to develop a new hardware or software module *P*, one should first develop a module *T* designed to test it. Further, *T* should then become part of *P*'s official development's contract.

As a matter of best practice, the ultimate test of a newly designed module should be formulated not by the module's developer, but rather by the architect who specified the module's interface. Therefore, for every chip or software system specified in the book, we supply an official test program, written by us. Although you are welcome to test your work in any way you see fit, the contract is such that eventually, *your* implementation must pass *our* tests.

In order to streamline the definition and execution of the numerous tests scattered all over the book projects, we designed a uniform *test scripting language*. This language works almost the same across all the simulators supplied with the book:

- *Hardware simulator:* used to simulate and test chips written in HDL
- *CPU emulator:* used to simulate and test machine language programs
- *VM emulator:* used to simulate and test programs written in the VM language

Every one of these simulators features a rich GUI that enables the user to test the loaded chip or program interactively, using graphical icons, or batch-style, using a test script. A *test script* is a series of commands that (a) load a hardware or software module into the relevant simulator, and (b) subject the module to a series of preplanned (rather than ad hoc) testing scenarios. In addition, the test scripts feature commands for printing the test results and comparing them to desired results, as defined in supplied compare files. In sum, a test script enables a systematic, replicable, and documented testing of the underlying code—an invaluable requirement in any hardware or software development project.

Important We don't expect students to write test scripts. *The test scripts necessary to test all the hardware and software modules mentioned in the book are supplied by us and available on the book's Web site*. Therefore, the chief purpose of this appendix is to explain the syntax and logic of the supplied test scripts, as needed.

B.1 File Format and Usage

The act of testing a hardware or software module using any one of the supplied simulators involves four types of files:

`Xxx.yyy`: where `Xxx` is the module name and `yyy` is either `hdl`, `hack`, `asm`, or `vm`, standing respectively for a chip definition written in HDL, a program written in the Hack machine language, a program written in the Hack assembly language, or a program written in the VM virtual machine language;

`Xxx.tst`: this *test script* walks the simulator through a series of steps designed to test the code stored in `Xxx.yyy`;

`Xxx.out`: this optional *output file* keeps a printed record of the actual simulation results;

`Xxx.cmp`: this optional *compare file* contains a presupplied record of the desired simulation results.

All these files should be kept in the same directory, which can be conveniently named `xxx`. In all simulators, the "current directory" refers to the directory from which the last file has been opened in the simulator environment.

White space: Space characters, newline characters, and comments in test scripts (`Xxx.tst` files) are ignored. Test scripts are not case sensitive, except for file and directory names.

Comments: The following comment formats can appear in test scripts:

```
//  Comment to end of line
/*  Comment until closing */
/** API documentation comment */
```

Usage: In all the projects that appear in the book, the files `Xxx.tst`, `Xxx.out`, and `Xxx.cmp` are supplied by us. These files are designed to test `Xxx.yyy`, whose development is the essence of the project. In some cases, we also supply a skeletal version of `Xxx.yyy`, for example, an HDL interface with a missing implementation part. All the files in all the projects are plain text files that can be viewed and edited using plain text editors.

Typically, one starts a simulation session by loading the supplied `Xxx.tst` script file into the relevant simulator. Typically, the first commands in the script instruct the simulator to load the code stored in `Xxx.yyy` and then, optionally, initialize an output file and a compare file. The remaining commands in the script run the actual tests, as we elaborate below.

B.2 Testing Chips on the Hardware Simulator

The hardware simulator supplied with the book is designed for testing and simulating chip definitions written in the Hardware Description Language (HDL) described in appendix A. Chapter 1 provides essential background on chip development and testing, and thus it is recommended to read it first.

B.2.1 Example

The script shown in figure B.1 is designed to test the EQ3 chip defined in figure A.1. A test script normally starts with some initialization commands, followed by a series of *simulation steps*, each ending with a semicolon. A simulation step typically instructs the simulator to bind the chip's input pins to some test values, evaluate the chip logic, and write selected variable values into a designated output file. Figure B.2 illustrates the `EQ3.tst` script in action.

B.2.2 Data Types and Variables

Data Types Test scripts support two data types: integers and strings. Integer constants can be expressed in hexadecimal (`%X` prefix), binary (`%B` prefix), or decimal (`%D` prefix) format, which is the default. These values are always translated into their equivalent 2's complement binary values. For example, the commands `set a1 %B1111111111111111`, `set a2 %XFFFF`, `set a3 %D-1`, `set a4 -1` will set the four variables to the same value: a series of sixteen 1's, representing "minus one" in

```
/* EQ3.tst: tests the EQ3.hdl program. The EQ3 chip should
   return true if its two 3-bit inputs are equal and false
   otherwise. */
load EQ3.hdl,         // Load the HDL program into the simulator
output-file EQ3.out,  // Write script outputs to this file
compare-to EQ3.cmp,   // Compare script outputs to this file
output-list a b out;  // Each subsequent output command should
                      // print the values of the variables
                      // a, b, and out
set a %B000, set b %B000, eval, output;
set a %B111, set b %B111, eval, output;
set a %B111, set b %B000, eval, output;
set a %B000, set b %B111, eval, output;
set a %B001, set b %B000, eval, output;
// Since the chip has two 3-bit inputs,
// an exhaustive test requires 2^3*2^3=64 such scenarios.
```

Figure B.1 Testing a chip on the hardware simulator.

decimal. String values (`%S` prefix) are used strictly for printing purposes and cannot be assigned to variables. String constants must be enclosed by “”.

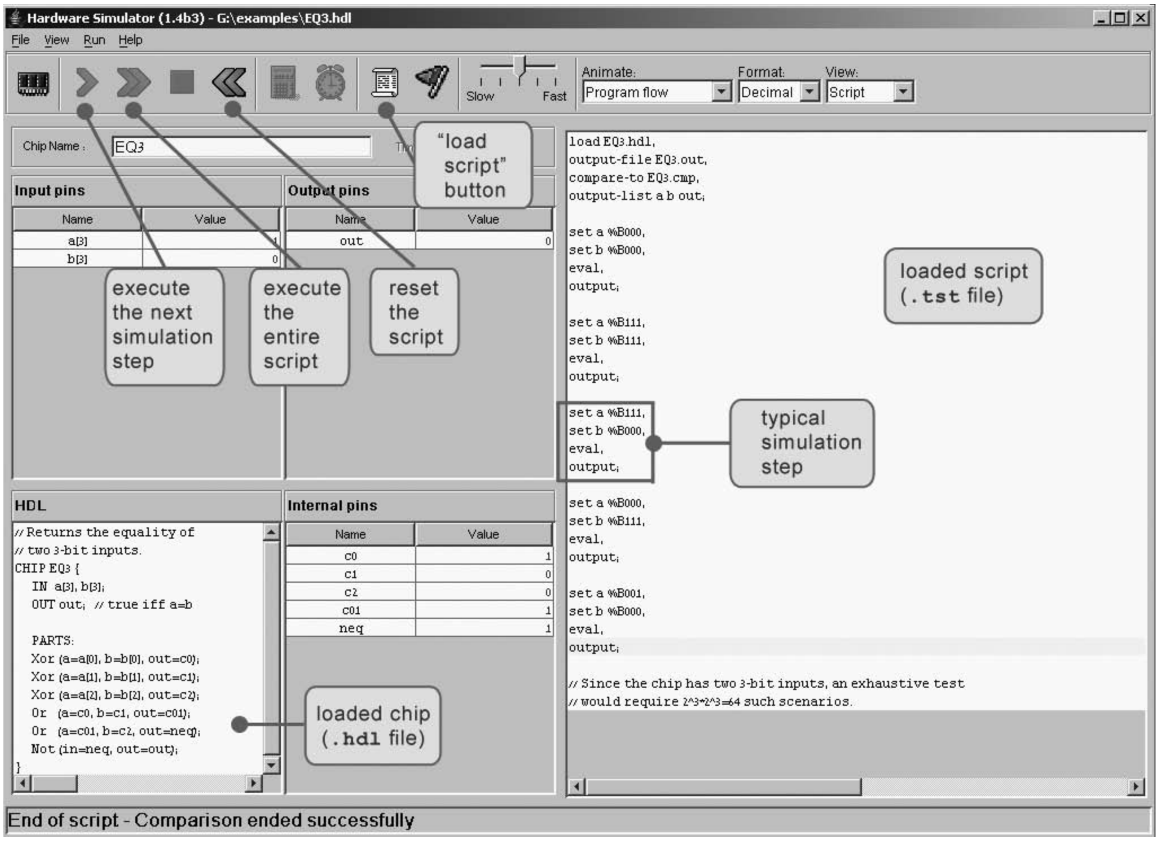
The simulator clock (used in testing sequential chips only) emits a series of values denoted 0, 0+, 1, 1+, 2, 2+, 3, 3+, and so forth. The progression of these *clock cycles* (also called *time units*) is controlled by two script commands called `tick` and `tock`. A `tick` moves the clock value from t to $t+$, and a `tock` from $t+$ to $t+1$, bringing upon the next time unit. The current time unit is stored in a system variable called `time`.

Script commands can access three types of variables: pins, variables of built-in chips, and the system variable `time`.

Pins: Input, output, and internal pins of the simulated chip. For example, the command `set in 0` sets the value of the pin whose name is `in` to 0.

Variables of built-in chips: Exposed by the chip's external implementation. See section B.2.4 for more details.

Time: The number of time units that elapsed since the simulation started running (read-only).


Hardware Simulator (1.4b3) - G:\examples\EQ3.hdl
File View Run Help
Slow Fast
"load script" button
execute the next simulation step
execute the entire script
reset the script
loaded script (.tst file)
typical simulation step
loaded chip (.hdl file)

Animate: Program flow Format: Decimal View: Script

Chip Name : EQ3

Input pins

Name	Value
a[3]	1
b[3]	0

Output pins

Name	Value
out	0

HDL

```
// Returns the equality of
// two 3-bit inputs.
CHIP EQ3 {
   IN  a[3], b[3];
   OUT out;  // true iff a=b

   PARTS:
   Xor (a=a[0], b=b[0], out=c0);
   Xor (a=a[1], b=b[1], out=c1);
   Xor (a=a[2], b=b[2], out=c2);
   Or  (a=c0, b=c1, out=c01);
   Or  (a=c01, b=c2, out=neq);
   Not (in=neq, out=out);
}
```

Internal pins

Name	Value
c0	1
c1	0
c2	0
c01	1
neq	1

```
load EQ3.hdl,
output-file EQ3.out,
compare-to EQ3.cmp,
output-list a b out;

set a %B000,
set b %B000,
eval,
output;

set a %B111,
set b %B111,
eval,
output;

set a %B111,
set b %B000,
eval,
output;

set a %B000,
set b %B111,
eval,
output;

set a %B001,
set b %B000,
eval,
output;

// Since the chip has two 3-bit inputs, an exhaustive test
// would require 2^3*2^3=64 such scenarios.
```

End of script - Comparison ended successfully

Figure B.2 Typical hardware simulation session, shown at the script's end. The loaded script is identical to `EQ3.tst` from figure B.1, except that some white space was added to improve readability.

B.2.3 Script Commands

Command Syntax A script is a sequence of commands. Each command is terminated by a comma, a semicolon, or an exclamation mark. These terminators have the following semantics:

- Comma (`,`): terminates a script command.
- Semicolon (`;`): terminates a script command and a simulation step. A simulation step consists of one or more script commands. When the user instructs the simulator

to “single-step” via the simulator’s GUI, the simulator executes the script from the current command until a semicolon is reached, at which point the simulation is paused.

- Exclamation mark (`!`): terminates a script command and stops the script execution. The user can later resume the script execution from that point onward. This option is typically used to facilitate interactive debugging.

It is convenient to organize the script commands in two conceptual sections. “Set up commands” are used to load files and initialize global settings. “Simulation commands” walk the simulator through a series of tests.

Setup Commands

`load` `Xxx.hdl`: Loads the HDL program stored in `Xxx.hdl` into the simulator. The file name must include the `.hdl` extension and must not include a path specification. The simulator will try to load the file from the current directory, and, failing that, from the simulator’s `builtIn` directory, as described in section A.3.

`output-file` `Xxx.out`: Instructs the simulator to write further output to the named file, which must include an `.out` extension. The output file will be created in the current directory.

`output-list` $v1, v2, \ldots$: Instructs the simulator what to write to the output file in every subsequent output command in this script (until the next `output-list` command, if any). Each value in the list is a variable name followed by a formatting specification. The command also produces a single header line consisting of the variable names. Each item v in the output-list has the syntax *variable format padL.len.padR*. This directive instructs the simulator to write *padL* spaces, then the current *variable* value in the specified *format* using *len* columns, then *padR* spaces, then the divider symbol “`|`”. *Format* can be either `%B` (binary), `%X` (hexa), `%D` (decimal) or `%S` (string). The default format specification is `%B1.1.1`.

For example, the `CPU.hdl` chip of the Hack platform has an input pin named `reset`, an output pin named `pc` (among others), and a chip part named `DRegister` (among others). If we want to track the values of these variables during the chip’s execution, we can use something like the following command:

```
Output-list  time%S1.5.1          // System variable
             reset%B2.1.2         // Input pin of the chip
             pc%D2.3.1            // Output pin of the chip
             DRegister[] %X3.4.4  // State of this built-in part
```

(Sate variables of built-in chips are explained here.) This command may produce the following output (after two subsequent output commands):

```
| time  |reset|  pc |DRegister[]|
|   20+ |  0  |  21 |    FFFF    |
|   21  |  0  |  22 |    FFFF    |
```

compare-to `Xxx.cmp`: Instructs the simulator that each subsequent output line should be compared to its corresponding line in the specified comparison file (which must include the `.cmp` extension). If any two lines are not the same, the simulator displays an error message and halts the script execution. The compare file is assumed to be present in the current directory.

Simulation Commands

set *variable value*: Assigns the *value* to the *variable*. The variable is either a pin or an internal variable of the simulated chip or one of its chip parts. The widths of the *value* and the *variable* must be compatible. For example, if `x` is a 16-bit pin and `y` is a 1-bit pin, then `set x 153` is valid whereas `set y 153` will yield an error and halt the simulation.

eval: Instructs the simulator to apply the chip logic to the current values of the input pins and compute the resulting output values.

output: This command causes the simulator to go through the following logic:

1. Get the current values of all the variables listed in the last `output-list` command.

2. Create an output line using the format specified in the last `output-list` command.

3. Write the output line to the output file.

4. (if a compare file has been previously declared via the `compare-to` command): If the output line differs from the current line of the compare file, display an error message and stop the script's execution.

5. Advance the line cursors of the output file and the compare file.

tick: Ends the first phase of the current time unit (clock cycle).

tock: Ends the second phase of the current time unit and embarks on the first phase of the next time unit.

repeat *num* **{***commands***}**: Instructs the simulator to repeat the *commands* enclosed by the curly brackets *num* times. If *num* is omitted, the simulator repeats the *commands* until the simulation has been stopped for some reason.

`while` *Boolean-condition* **`{`***commands***`}`**: Instructs the simulator to repeat the *commands* enclosed in the curly brackets as long as the *Boolean-condition* is true. The condition is of the form *x op y* where *x* and *y* are either constants or variable names and *op* is one of the following: =, >, <, >=, <=, <>. If *x* and *y* are strings, *op* can be either = or <>.

`echo` *text*: Instructs the simulator to display the *text* string in the status line (which is part of the simulator GUI). The text must be enclosed by “ ”.

`clear-echo`: Instructs the simulator to clear the status line.

`breakpoint` *variable value*: Instructs the simulator to compare the value of the specified *variable* to the specified *value*. The comparison is performed after the execution of each script command. If the *variable* contains the specified *value*, the execution halts and a message is displayed. Otherwise, the execution continues normally.

`clear-breakpoints`: Clears all the previously defined breakpoints.

built-in-chip *method argument(s)*: External implementations of built-in chips can expose methods that perform chip-specific operations. The syntax of the allowable method calls varies from one built-in chip to another and is documented next.

B.2.4 Variables and Methods of Built-In Chips

The logic of a chip can be implemented by either an HDL program or by a high-level programming language, in which case the chip is said to be “built-in” and “externally implemented.” External implementations of built-in chips can facilitate access to the chip’s state via the syntax *chipName*`[`*varName*`]`, where *varName* is an implementation-specific variable that should be documented in the chip API. The APIs of all the built-in chips supplied with the book (as part of the Hack computer platform) are shown in figure B.3.

For example, consider the command `set RAM16K[1017] 15`. If RAM16K is the currently simulated chip or an internal part of the currently simulated chip, this command will set its memory location number 1017 to the 2’s complement binary value of 15. Further, since the built-in RAM16K chip happens to have GUI side effects, the new value will also be displayed in the chip’s visual image.

If a built-in chip maintains a single-valued internal state, the current value of the state can be accessed through the notation *chipName*`[]`. If the internal state is a vector, the notation *chipName*`[`*i*`]` is used. For example, when simulating the built-in Register chip, one can write script commands like `set Register[] 135`. This command sets the internal state of the chip to the 2’s complement binary value of 135; in

Chip name	Exposed variables	Data type/range	Methods
`Register`	`Register[]`	16-bit (-32768...32767)	
`ARegister`	`ARegister[]`	16-bit	
`DRegister`	`DRegister[]`	16-bit	
`PC`	`PC[]`	15-bit (0..32767)	
`RAM8`	`RAM8[0..7]`	Each entry is 16-bit	
`RAM64`	`RAM64[0..63]`	"	
`RAM512`	`RAM512[0..511]`	"	
`RAM4K`	`RAM4K[0..4095]`	"	
`RAM16K`	`RAM16K[0..16383]`	"	
`ROM32K`	`ROM32K[0..32767]`	"	`load Xxx.hack/Xxx.asm`
`Screen`	`Screen[0..16383]`	"	
`Keyboard`	`Keyboard[]`	16-bit, read-only	

Figure B.3 API of all the built-in chips supplied with the book.

the next time unit, the Register chip will commit to this value and its output will start emitting it.

Built-in chips can also expose implementation-specific *methods* that extend the simulator's commands repertoire. For example, in the Hack computer, programs reside in an instruction memory unit implemented by a chip named ROM32K. Before one runs a machine language program on this computer, one must first load a program into this chip. In order to facilitate this service, our built-in implementation of ROM32K features a `load` *file name* method, referring to a text file that, hopefully, contains machine language instructions. This chip-specific method can be accessed by a test script via commands like `ROM32K load Myprog.hack`. In the chip set supplied with the book, this is the only method supported by any of the built-in chips.

B.2.5 Ending Example

We end this section with a relatively complex test script, designed to test the topmost Computer chip of the Hack platform. One way to test the Computer chip is to load a machine language program into it and monitor selected values as the computer executes the program, one instruction at a time. For example, we wrote a program that (hopefully) computes the maximum of RAM[0] and RAM[1] and writes the result to RAM[2]. The machine language version of this program is stored in the text file `Max.hack`. Note that at the very low level in which we operate, if such a program does not run properly it may be either because the program is buggy, or the hardware is buggy (and, for completeness, it may also be that the test script or the hardware simulator are buggy). For simplicity, let us assume that everything is error-free, except, possibly, for the tested Computer chip.

To test the Computer chip using the `Max.hack` program, we wrote a test script called `ComputerMax.tst`. This script loads `Computer.hdl` into the hardware simulator and then loads the `Max.hack` program into its ROM32K chip part. A reasonable way to check if the chip works properly is as follows: put some values in RAM[0] and RAM[1], reset the computer, run the clock, and inspect RAM[2]. This, in a nutshell, is what the script in figure B.4 is designed to do.

How can we tell that fourteen clock cycles are sufficient for executing this program? This can be found by trial and error, starting with a large value and watching the computer's outputs stabilizing after a while, or by analyzing the run-time behavior of the currently loaded program.

B.2.6 Default Script

The simulator's GUI buttons (single step, run, stop, reset) don't control the loaded chip. Rather, they control the progression of the loaded script, which controls the loaded chip's operation. Thus, there is a question of what to do if the user has loaded a chip directly into the simulator without loading a script first. In such cases, the simulator uses the following default script:

```
// Default script of the hardware simulator
repeat {
  tick,
  tock;
}
```

B.3 Testing Machine Language Programs on the CPU Emulator

The CPU emulator supplied with the book is designed for testing and simulating the execution of binary programs on the Hack computer platform described in chapter 5. The tested programs can be written in either the native Hack code or the assembly language described in chapter 4. In the latter case, the simulator translates the loaded code into binary on the fly, as part of the "load program" operation.

```
/* ComputerMax.tst script.
   The max.hack program should compute the maximum of
   RAM[0] and RAM[1] and write the result in RAM[2]. */

// Load the Computer chip and set up for the simulation
load Computer.hdl,
output-file Computer.out,
compare-to ComputerMax.cmp,
output-list RAM16K[0] RAM16K[1] RAM16K[2];

// Load the Max.hack program into the ROM32K chip part
ROM32K load Max.hack,
// Set the first 2 cells of the RAM16K chip part to some test values
set RAM16K[0] 3,
set RAM16K[1] 5,
output;
// Run the clock enough cycles to complete the program's execution
repeat 14 {
    tick, tock,
    output;
}

// Reset the Computer
set reset 1,
tick,         // Run the clock in order to commit the Program
tock,         // Counter (PC, a sequential chip) to the new reset value
output;
// Now re-run the program with different test values.
set reset 0,  // "De-reset" the computer (committed in next tick-tock)
set RAM16K[0] 23456,
set RAM16K[1] 12345,
output;
repeat 14 {
    tick, tock,
    output;
}
```

Figure B.4 Testing the topmost Computer chip.

As a convention, a script that tests a machine language program `Xxx.hack` or `Xxx.asm` is called `Xxx.tst`. As usual, the simulation involves four files: the test script itself (`Xxx.tst`), the tested program (`Xxx.hack` or `Xxx.asm`), an optional output file (`Xxx.out`) and an optional compare file (`Xxx.cmp`). All these files must reside in the same directory. This directory can be conveniently named `xxx`. For more information about file structure and recommended usage, see section B.1.

B.3.1 Example

Consider the multiplication program `Mult.hack`, designed to effect RAM[2] = RAM[0]*RAM[1]. A reasonable way to test this program is to put some values in RAM[0] and RAM[1], run the program, and inspect RAM[2]. This logic is carried out in figure B.5.

```
// Load the program and set up for the simulation
load Mult.hack,
output-file Mult.out,
compare-to Mult.cmp,
output-list RAM[2]%D2.6.2;

// Set the first 2 cells of the RAM to some test values
set RAM[0] 2,
set RAM[1] 5;
// Run the clock enough cycles to complete the program's execution
repeat 20 {
  ticktock;
}
output;

// Re-run the same program with different test values
set PC 0,
set RAM[0] 8,
set RAM[1] 7;
repeat 50 {    // Mult.hack is based on repetitive addition, so
  ticktock;    // greater multiplicands require more clock cycles
}
output;
```

Figure B.5 Testing a machine language program on the CPU emulator.

B.3.2 Variables

The CPU emulator, which is hardware-specific, recognizes a set of variables related to internal components of the Hack platform. In particular, scripting commands running on the CPU emulator can access the following elements:

`A`: value of the address register (unsigned 15-bit);

`D`: value of the data register (16-bit);

`PC`: value of the Program Counter register (unsigned 15-bit);

`RAM[i]`: value of RAM location *i* (16-bit);

`time`: Number of time units (also called clock cycles, or *ticktocks*) that elapsed since the simulation started (read-only).

B.3.3 Commands

The CPU emulator supports all the commands described in section B.2.3, except for the following changes:

`load` *program*: Here *program* is either `Xxx.hack` or `Xxx.asm`. This command loads a machine language program (to be tested) into the simulated instruction memory. If the program is written in assembly, it is translated into binary on the fly.

`eval`: Not applicable;

built-in-chip *method argument(s)*: Not applicable;

`ticktock`: This command is used instead of `tick` and `tock`. Each `ticktock` advances the clock one time unit (cycle).

B.3.4 Default Script

The CPU emulator's GUI buttons (single step, run, stop, reset) don't control the loaded program. Rather, they control the progression of the loaded script, which controls the program's operation. Thus, there is a question of what to do if the user has loaded a program directly into the CPU emulator without loading a script first. In such cases, the emulator uses the following default script:

```
// Default script of the CPU emulator
repeat {
  ticktock;
}
```

B.4 Testing VM Programs on the VM Emulator

Chapters 7–8 describe a virtual machine model and specify a VM implementation on the Hack platform. The VM emulator supplied with the book is an alternative VM implementation that uses Java to run VM programs, visualize their operations, and display the states of the effected virtual memory segments.

Recall that a VM program consists of one or more `.vm` files. Thus, the simulation of a VM program involves four elements: the test script (`Xxx.tst`), the tested program (a single `Xxx.vm` file or an `Xxx` directory containing one or more `.vm` files), an optional output file (`Xxx.out`) and an optional compare file (`Xxx.cmp`). All these files must reside in the same directory, which can be conveniently named `xxx`. For more information about file structure and recommended usage, see section B.1. Chapter 7 provides essential information about the virtual machine architecture, without which the discussion below will not make much sense.

Startup Code A VM program is normally assumed to contain at least two functions: `Main.main` and `Sys.init`. When the VM translator translates a VM program, it generates machine language code that sets the stack pointer to 256 and then calls the `Sys.init` function, which then calls `Main.main`. In a similar fashion, when the VM emulator is instructed to execute a VM program (collection of one or more VM functions), it is programmed to start running the `Sys.init` function, which is assumed to exist somewhere in the loaded VM code. If a `Sys.init` function is not found, the emulator is programmed to start executing the first command in the loaded VM code.

The latter convention was added to the emulator in order to assist the gradual development of the VM implementation, which spans two chapters in the book. In chapter 7, we build only the part of the VM implementation that deals with `pop`, `push`, and arithmetic commands, without getting into subroutine calling commands. Thus, the test programs associated with Project 7 consist of “raw” VM commands without the typical `function/return` wrapping. Since we wish to allow informal experimentation with such commands, we gave the VM emulator the ability to execute “raw” VM code which is neither properly initialized nor properly packaged in a function structure.

Virtual Memory Segments In the process of simulating the virtual machine’s operations, the VM emulator manages the virtual memory segments of the Hack VM (`argument`, `local`, etc.). These segments must be allocated to the host RAM—a task that the emulator normally carries out as a side effect of simulating the execution of `call`, `function`, and `return` commands. This means that when simulating "raw" VM code that contains no subroutine calling commands, we must force the VM emulator to explicitly anchor the virtual segments in the RAM—at least those segments mentioned in the current code. Conveniently, this initialization can be accomplished by script commands that manipulate the pointers controlling the base RAM addresses of the virtual segments. Using these script commands, we can effectively put the virtual segments in selected areas in the host RAM.

B.4.1 Example

The `FibonacciSeries.vm` file contains a series of VM commands that compute the first *n* elements of the Fibonacci series. The code is designed to operate on two arguments: the value of *n* and the starting memory address in which the computed elements should be stored. The script in figure B.6 is designed to test this program using the actual arguments 6 and 4000.

B.4.2 Variables

Scripting commands running on the VM emulator can access the following elements:

Contents of Virtual Memory Segments

`local[i]`: value of the i-th element of the `local` segment;

`argument[i]`: value of the i-th element of the `argument` segment;

`this[i]`: value of the i-th element of the `this` segment;

`that[i]`: value of the i-th element of the `that` segment;

`temp[i]`: value of the i-th element of the `temp` segment.

Pointers to Virtual Memory Segments

`local`: base address of the `local` segment in the RAM;

`argument`: base address of the `argument` segment in the RAM;

`this`: base address of the `this` segment in the RAM;

`that`: base address of the `that` segment in the RAM.

```
/* The FibonacciSeries.vm file contains a series of VM commands
   that compute the first n Fibonacci numbers. The program's
   code contains no function/call/return commands, and thus the
   VM emulator must be forced to initialize the virtual memory
   segments used by the code explicitly.
*/
// Load the program and set up for the simulation
load FibonacciSeries.vm,
output-file FibonacciSeries.out,
compare-to FibonacciSeries.cmp,
output-list RAM[4000]%D1.6.2 RAM[4001]%D1.6.2 RAM[4002]%D1.6.2
            RAM[4003]%D1.6.2 RAM[4004]%D1.6.2 RAM[4005]%D1.6.2;
// Initialize the stack and the argument and local segments.
set SP 256,            // Stack pointer (stack begins in RAM[256])
set local 300,         // Base the local segment in some RAM location
set argument 400;      // Base the argument segment in some RAM loc.
// Set the arguments to two test values
set argument[0] 6,     // n=6
set argument[1] 4000;  // Put the series at RAM[4000] and onward
// Execute enough VM steps to complete the program's execution
repeat 140 {
  vmstep;
}
output;
```

Figure B.6 Testing a VM program on the VM emulator.

Implementation-Specific Variables

`RAM[i]`: value of the i-th RAM location;

`SP`: value of the stack pointer;

`currentFunction`: name of the currently executing function (read only).

`line`: contains a string of the form: *current-function-name.line-index-in-function* (read only).

For example, when execution reaches the third line of the function `Sys.init`, the `line` variable contains "`Sys.init.3`". This is a useful means for setting breakpoints in selected locations in the loaded VM program.

B.4.3 Commands

The VM emulator supports all the commands described in section B.2.3, except for the following changes:

`load` *source*: Here *source* is either `Xxx.vm`, the name of a file containing one or more VM functions, or a series of “raw” VM commands, or `Xxx`, the name of a directory containing one or more `.vm` files (in which case all of them are loaded).

If the `.vm` files are located in the current directory, the *source* argument can be omitted.

`tick/tock`: Not applicable.

`vmstep`: Simulates the execution of a single VM command from the VM program, and advances to the next command in the code.

B.4.4 Default Script

The VM emulator’s GUI buttons (single step, run, stop, reset) don’t control the loaded VM code. Rather, they control the progression of the loaded script, which controls the code’s operation. Thus, there is a question of what to do if the user has loaded a program directly into the VM emulator without loading a script first. In such cases, the emulator uses the following default script:

```
// Default script of the VM emulator
repeat {
  vmstep;
}
```