Appendix A:

Hardware Description Language (HDL)

Intelligence is the faculty of making artificial objects, especially tools to make tools.

—Henry Bergson (1859-1941)

A *Hardware Description Language* (HDL) is a formalism for defining and testing chips: objects whose interfaces consist of input and output pins that carry Boolean signals, and whose bodies are composed of interconnected collections of other, lower-level, chips. This appendix describes a typical HDL, as understood by the hardware simulator supplied with the book. Chapter 1 (in particular, section 1.1) provides essential background without which this appendix does not make much sense.

How to Use This Appendix This is a technical reference, and thus there is no need to read it from beginning to end. Instead, we recommended focusing on selected sections, as needed. Also, HDL is an intuitive and self-explanatory language, and the best way to learn it is to play with some HDL programs using the supplied hardware simulator. Therefore, we recommend to start experimenting with HDL programs as soon as you can, beginning with the following example.

A.1 Example

Figure A.1 specifies a chip that accepts two three-bit numbers and outputs whether they are equal or not. The chip logic uses Xor gates to compare the three bit-pairs, and outputs true if all the comparisons agree. Each internal part Xxx invoked by an HDL program refers to a stand-alone chip defined in a separate Xxx.hdl program. Thus the chip designer who wrote the EQ3.hdl program assumed the availability of three other lower-level programs: Xor.hdl, Or.hdl, and Not.hdl. Importantly, though, the designer need not worry about how these chips are implemented. When building a new chip in HDL, the internal parts that participate in the design are always viewed as black boxes, allowing the designer to focus only on their proper arrangement in the current chip architecture.

```
/** Checks if two 3-bit input buses are equal */
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);
}
```

Figure A.1 HDL program example.

Thanks to this modularity, all HDL programs, including those that describe high-level chips, can be kept short and readable. For example, a complex chip like RAM16K can be implemented using a few internal parts (e.g., RAM4K chips), each described in a single HDL line. When fully evaluated by the hardware simulator all the way down the recursive chip hierarchy, these internal parts are expanded into many thousands of interconnected elementary logic gates. Yet the chip designer need not be concerned by this complexity, and can focus instead only on the chip's topmost architecture.

A.2 Conventions

File extension: Each chip is defined in a separate text file. A chip whose name is Xxx is defined in file Xxx.hdl.

Chip structure: A chip definition consists of a header and a body. The header specifies the chip *interface,* and the body its implementation. The header acts as the chip's API, or public documentation. The body should not interest people who use the chip as an internal part in other chip definitions.

Syntax conventions: HDL is case sensitive. HDL keywords are written in uppercase letters.

Identifier naming: Names of chips and pins may be any sequence of letters and digits not starting with a digit. By convention, chip and pin names start with a capital letter and a lowercase letter, respectively. For readability, such names can include uppercase letters.

White space: Space characters, newline characters, and comments are ignored.

Comments: The following comment formats are supported:

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

A.3 Loading Chips into the Hardware Simulator

HDL programs (chip descriptions) are loaded into the hardware simulator in three different ways. First, the user can open an HDL file interactively, via a "load file" menu or GUI icon. Second, a test script (discussed here) can include a load Xxx.hdl command, which has the same effect. Finally, whenever an HDL program is loaded and parsed, every chip name Xxx listed in it as an internal part causes the simulator to load the respective Xxx.hdl file, all the way down the recursive chip hierarchy. In every one of these cases, the simulator goes through the following logic:

```
if Xxx.hdl exists in the current directory
  then load it (and all its descendents) into the simulator
else
  if Xxx.hdl exists in the simulator's builtIn chips directory
    then load it (and all its descendents) into the simulator
  else
      issue an error message.
```

The simulator's builtIn directory contains executable versions of all the chips specified in the book, except for the highest-level chips (CPU, Memory, and Computer). Hence, one may construct and test every chip mentioned in the book before all, or even any, of its lower-level chip parts have been implemented: The simulator will automatically invoke their built-in versions instead. Likewise, if a lower-level chip Xxx has been implemented by the user in HDL, the user can still force the simulator to use its built-in version instead, by simply moving the Xxx.hdl file out from the current directory. Finally, in some cases the user (rather than the simulator) may want to load a built-in chip directly, for example, for experimentation. To do so, simply navigate to the tools/builtIn directory—a standard part of the hardware simulator environment—and select the desired chip from there.

A.4 Chip Header (Interface)

The header of an HDL program has the following format:

```
CHIP chip name {
  IN input pin name, input pin name, ...;
  OUT output pin name, output pin name, ...;
  // Here comes the body.
}
```

■ *CHIP declaration:* The CHIP keyword is followed by the chip name. The rest of the HDL code appears between curly brackets.

■ *Input pins:* The IN keyword is followed by a comma-separated list of input pin names. The list is terminated with a semicolon.

■ *Output pins:* The OUT keyword is followed by a comma-separated list of output pin names. The list is terminated with a semicolon.

Input and output pins are assumed by default to be single-bit wide. A multi-bit bus can be declared using the notation pin *name* [*w*] (e.g., a [3] in EQ3.hdl). This specifies that the pin is a bus of width w. The individual bits in a bus are indexed 0...w—1, from right to left (i.e., index 0 refers to the least significant bit).

A.5 Chip Body (Implementation)

A.5.1 Parts

A typical chip consists of several lower-level chips, connected to each other and to the chip input/output pins in a certain "logic" (connectivity pattern) designed to deliver the chip functionality. This logic, written by the HDL programmer, is described in the chip body using the format:

PARTS:

internal chip part;
internal chip part;
...
internal chip part;

Where each internal chip part statement describes one internal chip with all its connections, using the syntax:

chip name (*connection*, ..., *connection*);

Where each connection is described using the syntax:

part's pin names = *chip's pin name*

(Throughout this appendix, the presently defined chip is called chip, and the lower-level chips listed in the PARTS section are called *parts*).

A.5.2 Pins and Connections

Each *connection* describes how one pin of a part is connected to another pin in the chip definition. In the simplest case, the programmer connects a part's pin to an input or output pin of the chip. In other cases, a part's pin is connected to another pin of another part. This internal connection requires the introduction of an *internal pin*, as follows:

Internal Pins In order to connect an output pin of one part to the input pins of other parts, the HDL programmer can create and use an internal pin, say v, as follows:

```
Part1 (..., out=v);       // out of Part1 is piped into v
Part2 (in=v, ...);        // v is piped into in of Part2
Part3 (a=v, b=v, ...);    // v is piped into both a and b of Part3
```

Internal pins (like v) are created as needed when they are specified the first time in the HDL program, and require no special declaration. Each internal pin has fan-in 1 and unlimited fan-out, meaning that it can be fed from a single source only, yet it can feed (through multiple connections) many other parts. In the preceding example, the internal pin v simultaneously feeds both Part2 (through in) and Part3 (though a and b).

Input Pins Each input pin of a part may be fed by one of the following sources:

- an input pin of the chip
- an internal pin
- one of the constants true and false, representing 1 and 0, respectively

Each input pin has fan-in 1, meaning that it can be fed by one source only. Thus Part (in1=v, in2=v, ...) is a valid statement, whereas Part (in1=v, in1=u, ...) is not.

Output Pins Each output pin of a part may feed one of the following destinations:

- an output pin of the chip
- an internal pin

A.5.3 Buses

Each pin used in a connection—whether input, output, or internal—may be a multi-bit bus. The widths (number of bits) of input and output pins are defined in the chip header. The widths of internal pins are deduced implicitly, from their connections.

In order to connect individual elements of a multi-bit bus input or output pin, the pin name (say x) may be subscripted using the syntax x[i] or x[i...j]=v, where v is an internal pin. This means that only the bits indexed *i* to *j* (inclusive) of pin *x* are connected to the specified internal pin. An internal pin (like v above) may not be subscripted, and its width is deduced implicitly from the width of the bus pin to which it is connected the first time it is mentioned in the HDL program.

The constants true and false may also be used as buses, in which case the required width is deduced implicitly from the context of the connection.

Example

```
CHIP Foo {
   IN in[8]    // 8-bit input
   OUT out[8]  // 8-bit output
   // Foo's body (irrelevant to the example)
}
```

Suppose now that Foo is invoked by another chip using the part statement:

```
Foo(in[2..4]=v, in[6..7]=true, out[0..3]=x, out[2..6]=y)
```

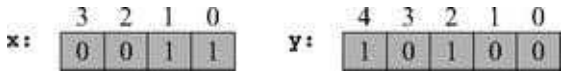
where v is a previously declared 3-bit internal pin, bound to some value. In that case, the connections in[2..4]=v and in [6..7]=true will bind the in bus of the Foo chip to the following values:

	7	6	5	4	3	2	1	0	(Bit)
in:	1	1	?	v[2]	v[1]	v[0]	?	?	(Contents)

Now, let us assume that the logic of the Foo chip returns the following output:

	7	6	5	4	3	2	1	0
out:	1	1	0	1	0	0	1	1

In that case, the connections out[0..3]=x and out[2..6]=y will yield:


x:
3 2 1 0
0 0 1 1
y:
4 3 2 1 0
1 0 1 0 0

A.6 Built-In Chips

The hardware simulator features a library of built-in chips that can be used as internal parts by other chips. Built-in chips are implemented in code written in a programming language like Java, operating behind an HDL interface. Thus, a built-in chip has a standard HDL header (interface), but its HDL body (implementation) declares it as built-in. Figure A.2 gives a typical example.

The identifier following the keyword BUILTIN is the name of the program unit that implements the chip logic. The present version of the hardware simulator is built in Java, and all the built-in chips are implemented as compiled Java classes. Hence, the HDL body of a built-in chip has the following format:

BUILTIN *Java class name:*

where Java class name is the name of the Java class that delivers the chip functionality. Normally, this class will have the same name as that of the chip, for example Mux.class. All the built-in chips (compiled Java class files) are stored in a directory called tools/builtIn, which is a standard part of the simulator's environment.

Built-in chips provide three special services:

■ *Foundation:* Some chips are the atoms from which all other chips are built. In particular, we use Nand gates and flip-flop gates as the building blocks of all combinational and sequential chips, respectively. Thus the hardware simulator features built-in versions of Nand.hdl and DFF.hdl.

```
/** 16-bit Multiplexor.
If sel = 0 then out = a else out = b.
This chip has a built-in implementation delivered by an external
Java class. */
CHIP Mux16 {
   IN a[16], a[16], sel;
   OUT out[16];
   BUILTIN Mux;  // Reference to builtIn/Mux.class, that
                 // implements both the Mux.hdl and the
                 // Mux16.hdl built-in chips.
}
```

Figure A.2 HDL definition of a built-in chip.

■ *Certification and efficiency:* One way to modularize the development of a complex chip is to start by implementing built-in versions of its underlying chip parts. This enables the designer to build and test the chip logic while ignoring the logic of its lower-level parts—the simulator will automatically invoke their built-in implementations. Additionally, it makes sense to use built-in versions even for chips that were already constructed in HDL, since the former are typically much faster and more space-efficient than the latter (simulation-wise). For example, when you load RAM4k.hdl into the simulator, the simulator creates

a memory-resident data structure consisting of thousands of lower-level chips, all the way down to the flip-flop gates at the bottom of the recursive chip hierarchy. Clearly, there is no need to repeat this drill-down simulation each time RAM4K is used as part in higher-level chips. Best practice tip: To boost performance and minimize errors, always use built-in versions of chips whenever they are available.

■ *Visualization:* Some high-level chips (e.g., memory units) are easier to understand and debug if their operation can be inspected visually. To facilitate this service, built-in chips can be endowed (by their implementer) with GUI side effects. This GUI is displayed whenever the chip is loaded into the simulator or invoked as a lower-level part by the loaded chip. Except for these visual side effects, GUI-EMPOWERED chips behave, and can be used, just like any other chip. Section A.8 provides more details about GUI-empowered chips.

A.7 Sequential Chips

Computer chips are either combinational or sequential (also called *clocked*). The operation of combinational chips is instantaneous. When a user or a test script changes the values of one or more of the input pins of a combinational chip and reevaluates it, the simulator responds by immediately setting the chip output pins to a new set of values, as computed by the chip logic. In contrast, the operation of sequential chips is clock-regulated. When the inputs of a sequential chip change, the outputs of the chip may change only at the beginning of the next time unit, as effected by the simulated clock.

In fact, sequential chips (e.g., those implementing counters) may change their output values when the time changes even if none of their inputs changed. In contrast, combinational chips never change their values just because of the progression of time.

A.7.1 The Clock

The simulator models the progression of time by supporting two operations called tick and tock. These operations can be used to simulate a series of time units, each consisting of two phases: a tick ends the first phase of a time unit and starts its second phase, and a tock signals the first phase of the next time unit. The real time that elapsed during this period is irrelevant for simulation purposes, since we have full control over the clock. In other words, either the simulator's user or a test script can issue ticks and tocks at will, causing the clock to generate series of simulated time units.

The two-phased time units regulate the operations of all the sequential chip parts in the simulated chip architecture, as follows. During the first phase of the time unit (*tick*), the inputs of each sequential chip in the architecture are read and affect the chip's internal state, according to the chip logic. During the second phase of the time unit (*tock*), the outputs of the chip are set to the new values. Hence, if we look at a sequential chip "from the outside," we see that its output pins stabilize to new values only at *tocks*—between consecutive time units.

There are two ways to control the simulated clock: manual and script-based. First, the simulator's GUI features a clock-shaped button. One click on this button (a tick) ends the first phase of the clock cycle, and a subsequent click (a tock) ends the second phase of the cycle, bringing on the first phase of the next cycle, and so on. Alternatively, one can run the clock from a test script, for example, using the command repeat n {tick, tock, output ; }. This particular example instructs the simulator to advance the clock n time units, and to print some values in the process. Test scripts and commands like repeat and output are described in detail in appendix B.

A.7.2 Clocked Chips and Pins

A built-in chip can declare its dependence on the clock explicitly, using the statement:

CLOCKED *pin*, *pin*, . . . , *pin*;

where each pin is one of the input or output pins declared in the chip header. The inclusion of an *input pin x* in the CLOCKED list instructs the simulator that changes to x should not affect any of the chip's output pins until the beginning of the next time unit. The inclusion of an output *pin x* in the CLOCKED list instructs the simulator that changes in any of the chip's input pins should not affect *x* until the beginning of the next time unit.

Note that it is quite possible that only some of the input or output pins of a chip are declared as clocked. In that case, changes in the nonclocked input pins may affect the nonclocked output pins in a combinational manner, namely, independent of the clock. In fact, it is also possible to have the CLOCKED keyword with an empty list of pins, signifying that even though the chip may change its internal state depending on the clock, changes to any of its input pins may cause immediate changes to any of its output pins.

The "Clocked" Property of Chips How does the simulator know that a given chip is clocked? If the chip is built-in, then its HDL code may include the keyword CLOCKED. If the chip is not built-in, then it is said to be clocked when one or more of its lower-level chip parts are clocked. This "clocked" property is checked recursively, all the way down the chip hierarchy, where a built-in chip may be explicitly clocked. If such a chip is found, it renders every chip that depends on it (up the hierarchy) implicitly clocked. It follows that nothing in the HDL code of a given chip suggests that it may be clocked—the only way to know for sure is to read the chip documentation. For example, let us consider how the built-in DFF chip (figure A.3) impacts the "clockedness" of some of other chips presented in the book.

Every sequential chip in our computer architecture depends in one way or another on (typically numerous) DFF chips. For example, the RAM64 chip is made of eight RAM8 chips. Each one of these chips is made of eight lower-level Register chips. Each one of these registers is made of sixteen Bit chips. And each one of these Bit chips contains a DFF part. It follows that Bit, Register, RAM8, RAM64 and all the memory units above them are also clocked chips.

```
/** D-Flip-Flop.
If load[t-1]=1 then out[t]=in[t-1] else out does not change. */
CHIP DFF {
  IN in;
  OUT out;
  BUILTIN DFF;        // Implemented by builtIn/DFF.class.
  CLOCKED in, out;    // Explicitly clocked.
}
```

Figure A.3 HDL definition of a clocked chip.

It’s important to remember that a sequential chip may well contain combinational logic that is not affected by the clock. For example, the structure of every sequential RAM chip includes combinational circuits that manage its addressing logic (described in chapter 3).

A.7.3 Feedback Loops

We say that the use of a chip entails a feedback loop when the output of one of its parts affects the input of the same part, either directly or through some (possibly long) path of dependencies. For example, consider the following two examples of direct feedback dependencies:

```
Not (in=loop1, out=loop1)    // Invalid
DFF (in=loop2, out=loop2)    // Valid
```

In each example, an internal pin (loop1 or loop2) attempts to feed the chip's input from its output, creating a cycle. The difference between the two examples is that Not is a combinational chip whereas DFF is clocked. In the Not example, loop1 creates an instantaneous and uncontrolled dependency between in and out, sometimes called data race. In the DFF case, the in-out dependency created by loop2 is delayed by the clocked logic of the DFF, and thus out (t) is not a function of in (t) but rather of in (t-1). In general, we have the following:

Valid/Invalid Feedback Loops When the simulator loads a chip, it checks recursively if its various connections entail feedback loops. For each loop, the simulator checks if the loop goes through a clocked pin, somewhere along the loop. If so, the loop is allowed. Otherwise, the simulator stops processing and issues an error message. This is done in order to avoid uncontrolled data races.

A.8 Visualizing Chip Operations

Built-in chips may be "GUI-empowered." These chips feature visual side effects, designed to animate chip operations. A GUI-empowered chip can come to play in a simulation in two different ways, just like any other chip. First, the user can load it directly into the simulator. Second, and more typically, whenever a GUI-empowered chip is used as a part in the simulated chip, the simulator invokes it automatically. In both cases, the simulator displays the chip's graphical image on the screen. Using this image, which is typically an interactive GUI component, one may inspect the current contents of the chip as well as change its internal state, when this operation is supported by the built-in chip implementation. The current version of this simulator features the following set of GUI-empowered chips:

ALU: Displays the Hack ALU's inputs and output as well as the presently computed function.

Registers (There are three of them: ARegister—address register, DRegister—data register, and PC—program counter): Displays the contents of the register and allows modifying its contents.

Memory chips (ROM32K and various RAM chips): Displays a scrollable array-like image that shows the contents of all the memory locations and allows their modification. If the contents of a memory location changes during the simulation, the respective entry in the GUI changes as well. In the case of the ROM32K chip (which serves as the instruction memory of our computer platform), the GUI also features a button that enables loading a machine language program from an external text file.

Screen chip: If the HDL code of a loaded chip invokes the built-in Screen chip, the hardware simulator displays a 256 rows by 512 columns window that simulates the physical screen. When the RAM-resident memory map of the screen changes during the simulation, the respective pixels in the screen GUI change as well, via a "refresh logic" embedded in the simulator implementation.

Keyboard chip: If the HDL code of a loaded chip invokes the built-in Keyboard chip, the simulator displays a clickable keyboard icon. Clicking this button connects the real keyboard of your computer to the simulated chip. From this point on, every key pressed on the real keyboard is intercepted by the simulated chip, and its binary code is displayed in the keyboard's RAM-resident memory map. If the user moves the mouse focus to another area in the simulator GUI, the control of the keyboard is restored to the real computer. Figure A.4 illustrates many of the features just described.

```
// Demo of GUI-empowered chips.
// The logic of this chip is meaningless, and is used merely to
// force the simulator to display the GUI effects of some other
// chips.
CHIP GUIDemo {
  IN in[16], load, address[15];
  OUT out[16];
  PARTS:
  RAM16K(in=in, load=load, address=address[0. .13], out=a);
  Screen(in=in, load=load, address=address[0. .12], out=b);
  Keyboard(out=c);
}
```

Figure A.4 HDL definition of a GUI-empowered chip.

The chip logic in figure A.4 feeds the 16-bit in value into two destinations: register number *address* in the RAM16K chip and register number address in the Screen chip (presumably, the HDL programmer who wrote this code has figured out the widths of these address pins from the documentation of these chips). In addition, the chip logic routes the value of the currently pressed keyboard key to the internal pin c. These meaningless operations are designed for one purpose only: to illustrate how the simulator deals with built-in GUI-empowered chips. The actual impact is shown in figure A.5.

A.9 Supplied and New Built-In Chips

The built-in chips supplied with the hardware simulator are listed in figure A.6. These Java-based chip implementations were designed to support the construction and simulation of the Hack computer platform (although some of them can be used to support other 16-bit platforms). Users who wish to develop hardware platforms other than Hack would probably benefit from the simulator's ability to accommodate new built-in chip definitions.

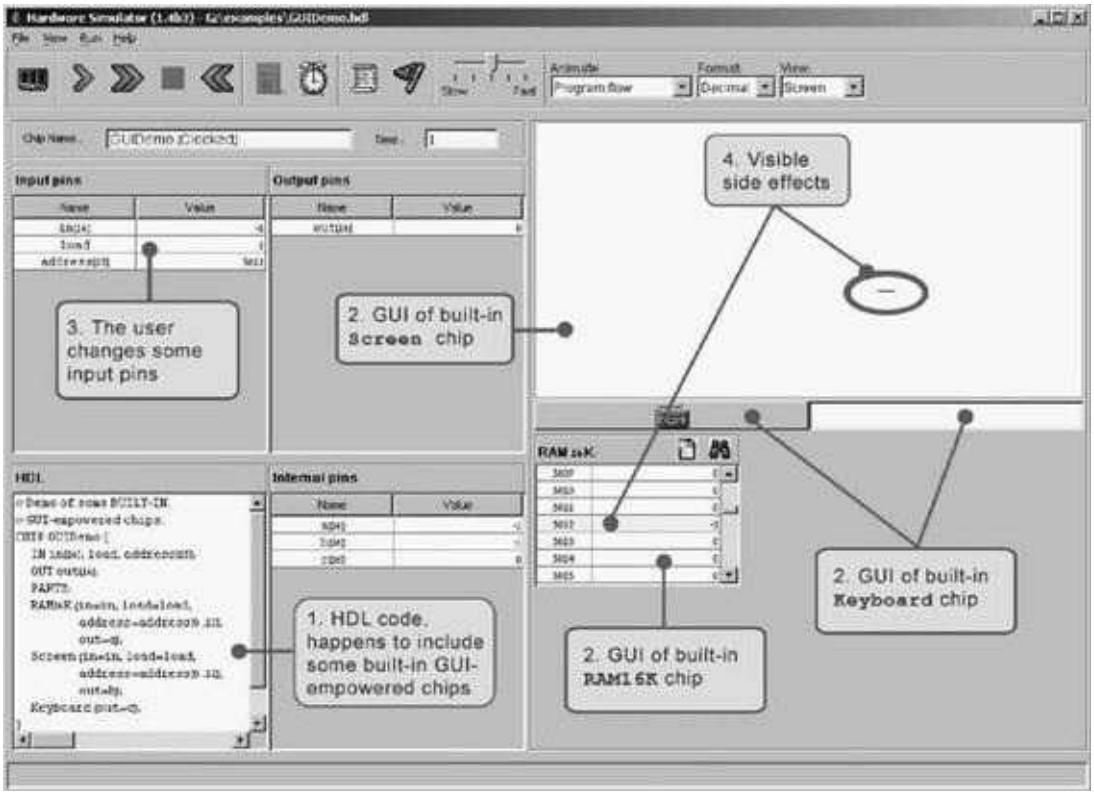


Figure A.5 GUI-empowered chips. Since the loaded HDL program uses GUI-empowered chips as internal parts (step 1), the simulator draws their respective GUI images (step 2). When the user changes the values of the chip input pins (step 3), the simulator reflects these changes in the respective GUIs (step 4). The circled horizontal line is the visual side effect of storing -1 in memory location 5012. Since the 16-bit 2's complement binary code of -1 is 1111111111111111, the computer draws 16 pixels starting at the 320th column of row 156, which happen to be the screen coordinates associated with address 5012 of the screen memory map (the exact memory-to-screen mapping is given in chapter 4).

Chip name	Specified in chapter	Has GUI	Comment
Nand	1		Foundation of all combinational chips
Not	1		
And	1		
Or	1		
Xor	1		
Mux	1		
DMux	1		
Not16	1		
And16	1		
Or16	1		
Mux16	1		
Or8way	1		
Mux4way16	1		
Mux8way16	1		
DMux4way	1		
DMux8way	1		
HalfAdder	2		
FullAdder	2		
Add16	2		
ALU	2	☑	
Inc16	2		
DFF	3		Foundation of all sequential chips
Bit	3		
Register	3		
ARegister	3	☑	Identical operation to Register, with GUI
DRegister	3	☑	Identical operation to Register, with GUI
RAM8	3	☑	
RAM64	3	☑	
RAM512	3	☑	
RAM4K	3	☑	
RAM16K	3	☑	
PC	3	☑	Program counter
ROM32K	5	☑	GUI allows loading a program from a text file
Screen	5	☑	GUI connects to a simulated screen
Keyboard	5	☑	GUI connects to the actual keyboard

Figure A.6 All the built-in chips supplied with the present version of the hardware simulator. A built-in chip has an HDL interface but is implemented as an executable Java class.

Developing New Built-In Chips The hardware simulator can execute any desired chip logic written in HDL; the ability to execute new built-in chips (in addition to those listed in figure A.6) written in Java is also possible, using a chip-extension API. Built-in chip implementations can be designed by users in Java to add new hardware components, introduce GUI effects, speed-up execution, and facilitate behavioral simulation of chips that are not yet developed in HDL (an important capability when designing new hardware platforms and related hardware construction projects). For more information about developing new built-in chips, see chapter 13.

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

```
/* 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.

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).

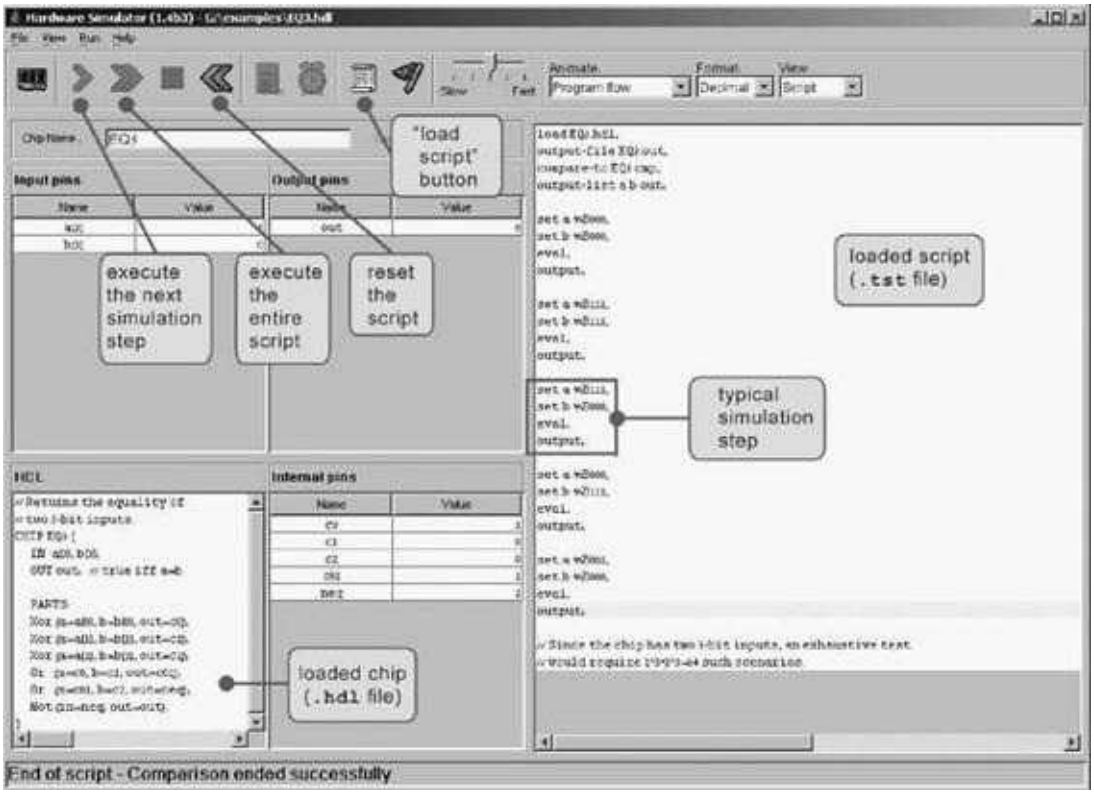


Figure B.2 Typical hardware simulation session, shown at the script's end. The loaded script is identical to EQ3.tst from figure B.1, except that some white space was added to improve readability.

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 $v1$, $v2$,...: 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 *chip Name* [*var Name*], where *var Name* 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 *chip Name*[ ]. If the internal state is a vector, the notation *chip Name* [*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 the next time unit, the Register chip will commit to this value and its output will start emitting it.

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.

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;
}
```