Appendix B: Test Scripting Language[1]

Appendix A described how to *define* and *simulate* chips. This appendix describes how to *test* and *debug* chip definitions. This is done by subjecting the HDL program representing the chip to various tests. The tests can be administered in an ad-hoc and interactive fashion, using the simulator GUI, or in a pre-planned and batch-style fashion, following a series of tests specified in a script file. This appendix describes the language in which *test scripts* are written, as understood by the hardware simulator supplied with this book. Chapter 1 provides essential background to this subject, and thus it is recommended to read it first.

The scripts that test a newly designed chip can be written either by the person who implements the chip, or by the hardware architect who ordered the chip done and specified its interface. As a matter of best practice, we recommend to use both approaches. Indeed, for every chip specified in the book, we provide an "official" test script, written by us. Thus, although you are welcome to test your chip designs in any way you see fit, the contract is such that eventually, *your* chip definitions have to pass *our* tests.

How to use this appendix: This is a technical document, and thus there is no need to read it from beginning to end. Instead, it is recommended to focus on selected sections, as needed. Like HDL, the test scripting language is rather intuitive, and the best way to learn it is to play with some sample scripts in the hardware simulator.

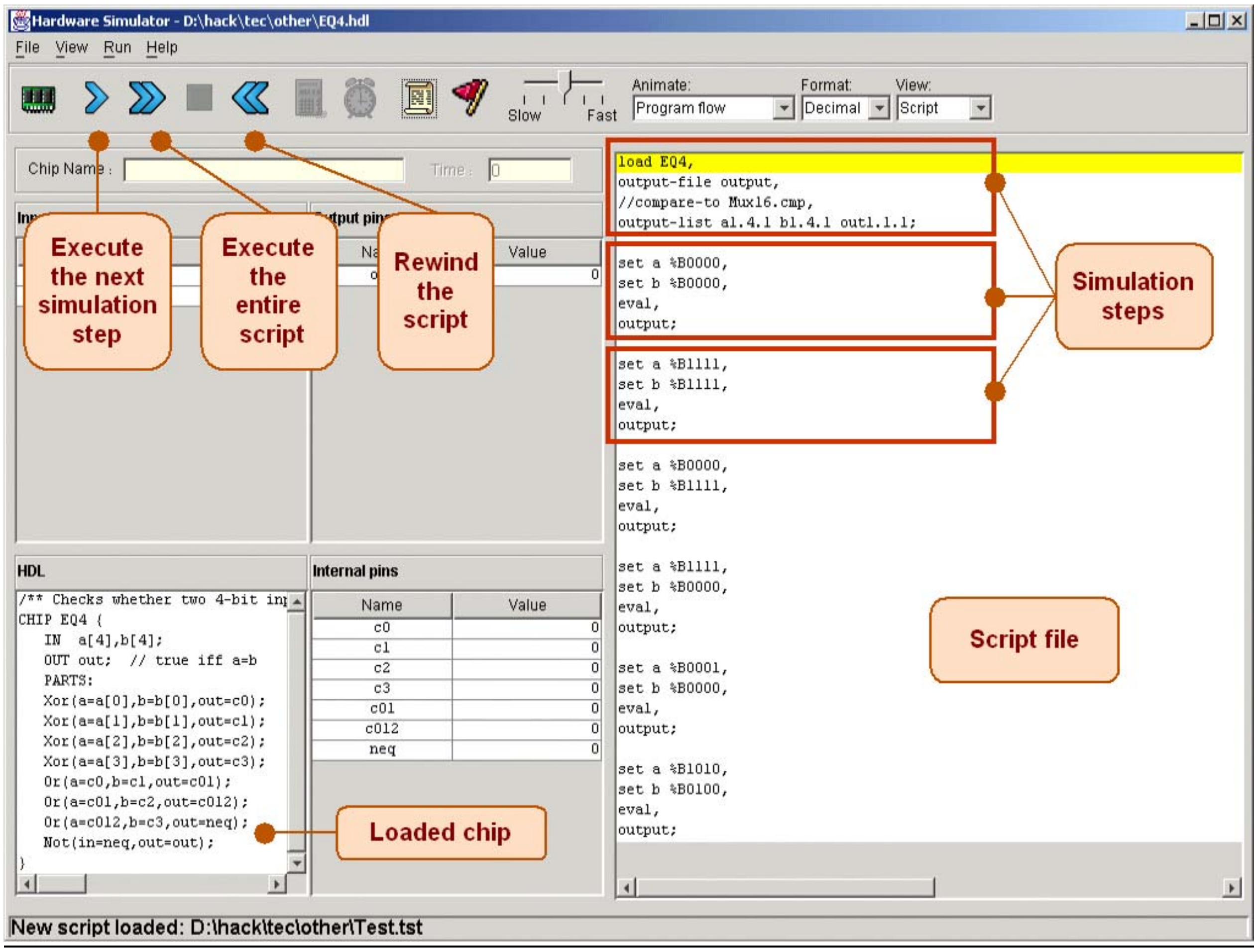
B.1 Opening Example

The following script is designed to test the EQ4 chip described in section A.1.

```
load EQ4,              // load EQ4.hdl into the simulator.
output-file EQ4.out,   // write the script outputs to this file.
compare-to EQ4.cmp,    // compare the script outputs to this file.
output-list a b out;   // each subsequent output command should
                       // print the values of a,b and out.
set a %B0000, set b %B0000, eval, output;
set a %B1111, set b %B1111, eval, output;
set a %B1111, set b %B0000, eval, output;
set a %B1111, set b %B0000, eval, output;
set a %B0000, set b %B1111, eval, output;
set a %B0001, set b %B0000, eval, output;
set a %B0101, set b %B0010, eval, output;
// Since the chip has two 4-bit inputs, an exhaustive test
// requires 2^4*2^4=256 such scenarios.
```

A test script normally starts with some initializations commands, followed by a series of *simulation steps*, each ending with a semicolon. A simulation step typically instructs the simulator to bind the chip input pins to some test values, evaluate the chip outputs, and write selected variable values into a designated output file. This logic is affected by the hardware simulator, as illustrated in Fig. B.1.

[1] From *The Digital Core*, by Nisan & Schocken, 2003, www.idc.ac.il/csd, forthcoming by MIT Press.



Hardware Simulator - D:\hack\tec\other\EQ4.hdl

Animate: Program flow
Format: Decimal
View: Script

```
load EQ4,
output-file output,
//compare-to Mux16.cmp,
output-list a1.4.1 b1.4.1 out1.1.1;

set a %B0000,
set b %B0000,
eval,
output;

set a %B1111,
set b %B1111,
eval,
output;

set a %B0000,
set b %B1111,
eval,
output;

set a %B1111,
set b %B0000,
eval,
output;

set a %B0001,
set b %B0000,
eval,
output;

set a %B1010,
set b %B0100,
eval,
output;
```

```
/** Checks whether two 4-bit in
CHIP EQ4 {
   IN  a[4],b[4];
   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);
   Xor(a=a[3],b=b[3],out=c3);
   Or(a=c0,b=c1,out=c01);
   Or(a=c01,b=c2,out=c012);
   Or(a=c012,b=c3,out=neq);
   Not(in=neq,out=out);
}
```

Internal pins

Name	Value
c0	0
c1	0
c2	0
c3	0
c01	0
c012	0
neq	0

New script loaded: D:\hack\tec\other\Test.tst

FIGURE B.1: Running a test script. Each test script *command* ends with a comma. A sequence of commands that ends with a semicolon constitutes a *simulation step* (The beginning of the *current simulation step* is highlighted with a yellow bar). The user controls the script execution by clicking the VCR buttons on the top left. Note that this particular script starts by loading the `EQ4.hdl` chip description into the simulator.

B.2 Conventions

File Extension: Test scripts are stored in text files with `.tst` extensions. Typically (but not necessarily), a script designed to test an `Xxx.hdl` program will be named `Xxx.tst`. Note however that the same chip can be tested by more than one script.

Current directory: The practice of developing and testing a chip involves at least one and typically four text files: the mandatory chip description file (`.hdl`), a test script file (`.tst`), a script-generated output file (`.out`), and a supplied compare-to file (`.cmp`). All these files should be kept in the same directory on the user's computer.

The *current directory* is defined as the *directory that contains the last file (whether chip or script) opened by the user from the simulator environment*.

Built-in chips: The built-in chips (residing in the simulator's `BuiltIn` directory) can be opened and tested by regular scripts, just like any other chips. All the built-in chips supplied with the simulator were rigorously tested, but one can experiment and re-test them for instructive purposes.

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 */
```

B.3 Data Types and Variables

Test scripts support one data type: integers. Integer constants can be expressed in hexadecimal (`%X` prefix), binary (`%B` prefix), or decimal (`%D` prefix) format, the latter being the default. These values are always translated into their equivalent 2's complement binary values. For example, the four commands `set a1 %XFFFF, set a2 %B11, set a3 %D11, set a4 -1` will set the respective variables to the binary values 1111111111111111, 11, 1011, and 1111111111111111.

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

Pins: Script commands can access the current values of all the 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: External implementations of built-in chips can expose internal variables that can be manipulated by test scripts. We delay the discussion of these variables to Section B.7.

Time: A read-only system variable, representing the number of time-units that elapsed since the simulation started running. Each rise and fall of the clock (a clock cycle, also known in our jargon as "`TickTack`") constitutes a single time-unit.

B.4 Command Structure

Command Terminators: 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.
- Semi-colon (`;`): 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 GUI, the simulator executes all the script logic from the current command until a semi-colon is reached, at which point the simulation is paused.

- Exclamation mark (`!`): terminates a script command and causes the simulator to stop the script execution. The user can later resume the script execution from that point onwards.

Command Syntax: Each command is written as a sequence of white space-separated identifiers. Identifiers are composed of any symbol except for terminators or white space. The script is not case sensitive.

It is convenient to describe the script commands in two conceptual sections. "Set up commands" are used to load files and initialize some global settings. "Simulation commands" walk the simulator through a series of tests. We now turn to describe these two categories of commands.

B.5 Set Up commands

`Load` <*chip name*>: Loads the file <*chip name*>`.hdl` into the simulator. The <*chip name*> must not contain a path specification or file extension. The simulator will try to load the chip from the current directory, and, failing that, from the simulator's `BuiltIn` directory, as described in Section A.3 (Appendix A).

`Output-file` <*file name*>: Instructs the simulator to write further output to the named file. 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 following syntax:

```
<variable name><format><padL>.<len>.<padR>
```

This directive instructs the simulator to write `padL` spaces, then the current value of `variable name` in the specified `format` using `len` columns, then `padR` spaces, then the symbol "`|`". `Format` can be either `%B` (binary), `%X` (hexa), or `%D` (decimal). The default output specification is `<variable name>%B1.1.1.`

For example, consider the command:

```
Output-list time%D0.5.2 reset%B1.1.1 PC%D2.3.2 A%X2.4.2 D%X2.4.2
```

This command may produce the following output (after two subsequent output commands):

```
| Time  |reset|   PC   |   A    |   D    |
|    37 | 0   |   21   |  001F  |  AA80  |
|    38 | 0   |   31   |  001F  |  AA80  |
```

`Compare-to` <*file name*>: Instructs the simulator that each subsequent output line should be compared to its corresponding line in the specified comparison file. If the 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.

B.6 Simulation Commands

`Set` <*variable name*> <*value*>: Assigns the *value* to the *variable*. The *variable* is either a pin or an internal variable of the simulated chip. The magnitude of *value* must match the *variable*'s width. For example, if `x` is a 16-bit pin and `y` is a single 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`: Let us assume that a *compare* file has been previously declared via the `compare-to` command. The `output` 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 the output line differs from the current line of the *compare* file, display an error msg;
5. Advance the line cursors of the *output* file and the *compare* file.

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

`Tack`: Ends the second phase of the current time unit, and advances to the first phase of the next time unit.

`Repeat` *num* {*commands*}: instructs the simulator to repeat the sequence of script *commands* enclosed in the curly brackets *num* times. If *num* is omitted, the simulator repeats the *commands* until the simulation has been stopped for some other reason.

`While` <*Boolean condition*> {*commands*}: instructs the simulator to repeat the *commands* inside 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: =, >, <, >=, <=, <>.

`Echo` <*text*>: instructs the simulator to print the *text* in the status line (part of the simulator GUI). The text must be enclosed in “”.

`Clear-echo`: instructs the simulator to clear the status line.

<Built-in chip name> <method name> <argument(s)>: Built-in chip implementations can expose methods that perform chip-specific operations. The syntax of these operations varies from one built-in chip to another and is documented in their API’s. This command option is described in detail in the next section.

B.7 Internal variables and methods of built-in chips

External implementations of built-in chips can expose internal variables via the syntax `chipName[varName]`, where `varName` is an implementation-specific variable. The exposed variables (if any) of the built-in chip should be documented in the chip's API. For example, consider our built-in version of a RAM16K chip. The API of this memory chip documents that any individual location in it can be accessed via the syntax `RAM16K[`*i*`]`, where *i* is between 0 to 16383. Such internal variables can be manipulated by test scripts, using standard `SET` commands like "`set RAM16K[1017] 15`". This particular command will set memory location number 1017 to the 2's complement binary value of 15. In addition, since the built-in `RAM16K` chip has GUI side effects, the new value will also be displayed on the chip's visual image.

If a built-in chip maintains an *internal state* (as in sequential chips), the current value of the state can be accessed through the convention `chipName[]`, but only if the internal state can be represented by a single-value variable. For example, when simulating the built-in `Register` chip, one can write script commands like `set Register[] -5324`. This command sets the internal state of the chip to the 2's complement binary value of –5324. In the next time unit, the `out` pin of the `Register` chip will start to emit this value.

Built-in chips can also expose implementation-specific *methods* that can be used in scripts as commands. For example, in the hardware platform specified in this book, programs reside in an Instruction Memory unit implemented by a chip called `ROM32K`. The contract is such that before one runs a machine language program on this computer, one must first load a program into its Instruction Memory. In order to facilitate this service, our built-in implementation of the `ROM32K` chip features a "`load` *<file name>*" method, where the *<file name>* argument is a text file that, hopefully, contains machine language instructions. This chip-specific method can be accessed by test scripts, via commands like "`ROM32K load myprog.bin`". Presently, this is the only method supplied by any of our built-in chips.

Chip Name	**Variable name/s**	**Width/Data range**
`Register`	`Register[]`	16-bit (-32768…32767)
`ARegister`	`ARegister[]`	16-bit
`DRegister`	`DRegister[]`	16-bit
`PC` (program counter)	`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]`	"
`Screen`	`Screen[0..16383]`	"
`Keyboard`	`Keyboard[]`	16-bit, read-only

TABLE B.2: Exposed internal variables of all the built-in chips supplied with the Hardware Simulator. These variables can be manipulated from test scripts.

B.8 Ending Example

We end the appendix with a relatively complex test script, designed to test the top-most `Computer` chip. One way to test the `Computer` chip is to load a program into it and monitor the outputs of the various hardware components 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.bin`. Note that at the very low level in which we operate, if such a program does not run properly it can be either because the *program* has bugs or the *hardware* has bugs (and, for completeness, it may also be that the *test script* or the *hardware simulator* have bugs). For simplicity, let us assume that the program is error-free.

To test the `Computer` chip using this program, we wrote a test script called `ComputerMax.tst`. This script loads the `Computer` chip into the hardware simulator and then loads the `max.bin` program into its `ROM` chip. 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 supplied test script is designed to do:

```
/* ComputerMax.tst script.
   The max.bin program computes the maximum of
   RAM[0] and RAM[1] and writes the result in RAM[2].
*/
// Load the chip, and set up for the simulation
load Computer, output-file output,
compare-to ComputerMax.cmp,
output-list RAM16K[0] RAM16K[1] RAM16K[2];
// Load the max.bin program into the built-in ROM32K chip
ROM32K load max.bin,
// set the first 2 cells of the built-in RAM16K chip 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,tack,
    output;
}
// Reset the Computer
set reset 1,
tick,          // run the clock in order to commit the Program
tack,          // Counter (a sequential chip) to the new reset value
output;
// Re-run the same program with different test values
set reset 0,   // "de-reset" the computer (committed in next tick-tack)
set RAM16K[0] 23456,
set RAM16K[1] 12345,
output;
Repeat 14 {
    tick,tack,
    output;
}
```

Note: We know that 14 cycles are sufficient to execute this program by trial and error, since we've experimented with this script before.