Appendix 2: Hardware Description Language

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

—Henry Bergson (1859–1941)

This appendix has two main parts. Sections A2.1–A2.5 describe the HDL language used in the book and in the projects. Section A2.6, named HDL Survival Guide, provides a set of essential tips for completing the hardware projects successfully.

A Hardware Description Language (HDL) is a formalism for defining *chips*: objects whose *interfaces* consist of input and output *pins* that carry binary signals, and whose *implementations* are connected arrangements of other, lower-level chips. This appendix describes the HDL that we use in Nand to Tetris. Chapter 1 (in particular, section 1.3) provides essential background that is a prerequisite to this appendix.

A2.1 HDL Basics

The HDL used in Nand to Tetris is a simple language, and the best way to learn it is to play with HDL programs using the supplied hardware simulator. We recommend starting to experiment as soon as you can, beginning with the following example.

Example: Suppose we have to check whether three 1-bit variables *a*, *b*, *c* have the same value. One way to check this three-way equality is to evaluate the Boolean function $\neg((a \neq b) \vee (b \neq c))$. Noting that the binary operator *not-equal* can be realized using a Xor gate, we can implement this function using the HDL program shown in figure A2.1.

interface

```
/** If the three given bits are equal, sets out to 1; else sets out to 0. */
CHIP Eq3 {
   IN  a, b, c;
   OUT out;
```

implementation

```
   PARTS:
   Xor(a=a, b=b, out=neq1);         // Xor(a,b) → neq1
   Xor(a=b, b=c, out=neq2);         // Xor(b,c) → neq2
   Or (a=neq1, b=neq2, out=outOr); // Or(neq1,neq2) → outOr
   Not(in=outOr, out=out);          // Not(outOr) → out
}
```

Figure A2.1 HDL program example.

The Eq3.hdl implementation uses four *chip-parts*: two Xor gates, one Or gate, and one Not gate. To realize the logic expressed by $\neg((a \neq b) \vee (b \neq c))$, the HDL programmer connects the chip-parts by creating, and naming, three *internal pins*: neq1, neq2, and outOr.

Unlike internal pins, which can be created and named at will, the HDL programmer has no control over the names of the input and output pins. These are normally supplied by the chips' architects and documented in given APIs. For example, in Nand to Tetris, we provide *stub files* for all the chips that you have to implement. Each stub file contains the chip interface, with a missing implementation. The contract is as follows: You are allowed to do whatever you want *under* the PARTS statement; you are not allowed to change anything *above* the PARTS statement.

In the Eq3 example, it so happens that the first two inputs of the Eq3 chip and the two inputs of the Xor and Or chip-parts have the same names (a and b). Likewise, the output of the Eq3 chip and that of the Not chip-part happen to have the same name (out). This leads to bindings like a=a, b=b, and out=out. Such bindings may look peculiar, but they occur frequently in HDL programs, and one gets used to them. Later in the appendix we'll give a simple rule that clarifies the meaning of these bindings.

Importantly, the programmer need not worry about how chip-parts are implemented. The chip-parts are used like black box abstractions, allowing the programmer to focus only on how to arrange them judiciously in order to realize the chip function. Thanks to this modularity, HDL programs can be kept short, readable, and amenable to unit testing.

HDL-based chips like Eq3.hdl can be tested by a computer program called *hardware simulator*. When we instruct the simulator to evaluate a given chip, the simulator evaluates all the chip-parts specified in its PARTS section. This, in turn, requires evaluating *their* lower-level chip-parts, and so on. This recursive descent can result in a huge hierarchy of downward-expanding chip-parts, all the way down to the terminal Nand gates from which all chips are made. This expensive drill-down can be averted using *built-in chips*, as we'll explain shortly.

HDL is a declarative language: HDL programs can be viewed as textual specifications of chip diagrams. For each chip *chipName* that appears in the diagram, the programmer writes a *chipName* (…) statement in the HDL program's PARTS section. Since the language is designed to describe *connections* rather than *processes*, the order of the PARTS statements is insignificant: as long as the chip-parts are connected correctly, the chip will function as stated. The fact that HDL statements can be reordered without affecting the chip's behavior may look odd to readers who are used to conventional programming. Remember: HDL is not a programming language; it's a specification language.

White space, comments, case conventions: HDL is case-sensitive: foo and Foo represent two different things. HDL keywords are written in uppercase letters. Space characters, newline characters, and comments are ignored. The following comment formats are supported:

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

Pins: HDL programs feature three types of *pins*: input pins, output pins, and internal pins. The latter pins serve to connect outputs of chip-parts to inputs of other chip-parts. Pins are assumed by default to be single-bit, carrying `0` or `1` values. Multi-bit *bus* pins can also be declared and used, as described later in this appendix.

Names of chips and pins may be any sequence of letters and digits not starting with a digit (some hardware simulators disallow using hyphens). By convention, chip and pin names start with a capital letter and a lowercase letter, respectively. For readability, names can include uppercase letters, for example, `xorResult`. HDL programs are stored in `.hdl` files. The name of the chip declared in the HDL statement `CHIP` *Xxx* must be identical to the prefix of the file name *Xxx*`.hdl`.

Program structure: An HDL program consists of an *interface* and an *implementation*. The interface consists of the chip's API documentation, chip name, and names of its input and output pins. The implementation consists of the statements below the `PARTS` keyword. The overall program structure is as follows:

```
/** API documentation: what the chip does. */
CHIP ChipName {
    IN inputPin1, inputPin2, ... ;
    OUT outputPin1, outputPin2, ... ;
PARTS:
   // Here comes the implementation.
}
```

Parts: The chip implementation is an unordered sequence of chip-part statements, as follows:

```
PARTS:
  chipPart(connection, ... , connection);
  chipPart(connection, ... , connection);
  . . .
```

Each *connection* is specified using the binding *pin*1 = *pin*2, where *pin*1 and *pin*2 are input, output, or internal pin names. These connections can be visualized as "wires" that the HDL programmer creates and names, as needed. For each "wire" connecting *chipPart*1 and *chipPart*2 there is an internal pin that appears twice in the HDL program: once as a *sink* in some *chipPart*1(…) statement, and once as a *source* in some other *chipPart*2 (…) statement.

For example, consider the following statements:

```
chipPart1(..., out = v,...);           // out of chipPart1 feeds the internal pin v
chipPart2(..., in = v, ...);           // in of chipPart2 is fed from v
chipPart3(..., in1 = v, ..., in2 = v,...);   // in1 and in2 of chipPart3 are also fed from v
```

Pins have fan-in 1 and unlimited fan-out. This means that a pin can be fed from a single source only, yet it can feed (through multiple connections) one or more pins in one or more chip-parts. In the above example, the internal pin v simultaneously feeds three inputs. This is the HDL equivalent of *forks* in chip diagrams.

The meaning of a = a: Many chips in the Hack platform use the same pin names. As shown in figure A2.1, this leads to statements like Xor (a=a, b=b, out=neq1). The first two connections feed the a and b inputs of the implemented chip (Eq3) into the a and b inputs of the Xor chip-part. The third connection feeds the out output of the Xor chip-part to the internal pin neq1. Here is a simple rule that helps sort things out: In every chip-part statement, the left side of each = binding always denotes an input or output pin *of the chip-part*, and the right side always denotes an input, output, or internal pin *of the implemented chip*.

A2.2 Multi-Bit Buses

Each input, output, or internal pin in an HDL program may be either a single-bit value, which is the default, or a multi-bit value, referred to as a *bus*.

Bit numbering and bus syntax: Bits are numbered from right to left, starting with 0. For example, sel=110, implies that sel[2]=1, sel[1]=1, and sel[0]=0.

Input and output bus pins: The bit widths of these pins are specified when they are declared in the chip's IN and OUT statements. The syntax is x[*n*], where x and *n* declare the pin's name and bit width, respectively.

Internal bus pins: The bit widths of internal pins are deduced implicitly from the bindings in which they are declared, as follows,

```
chipPart1(..., x[i] = u, ...);
chipPart2(..., x[i..j] = v, ...);
```

where x is an input or output pin of the chip-part. The first binding defines u to be a single-bit internal pin and sets its value to x[*i*]. The second binding defines v to be an internal bus-pin of width $j-i+1$ bits and sets its value to the bits indexed i to j (inclusive) of bus-pin x.

Unlike input and output pins, internal pins (like u and v) may not be subscripted. For example, u[i] is not allowed.

True/false buses: The constants true (1) and false (0) may also be used to define buses. For example, suppose that x is an 8-bit bus-pin, and consider this statement:

```
chipPart(…, x[0..2] = true, …, x[6..7] = true, …);
```

This statement sets x to the value 11000111. Note that unaffected bits are set by default to false (0). Figure A2.2 gives another example.

```
// Sets out = Not(in), bitwise
CHIP Not8 {
   IN  in[8];
   OUT out[8];
   ...
}
```

```
CHIP Foo {
   ...
   PARTS
   ...
   Not8(in[0..1]  = true,
        in[3..5]  = six,
        in[7]     = true,
        out[3..7] = out1,
        ...             );
   ...
}
```

Assumption: six is an internal pin, containing the value 110.

out1 is an internal pin, created by the Not8 chip-part statement.

Below: the resulting contents of the in input of Not8, and of out1.

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

	4	3	2	1	0
out1:	0	1	0	0	1

Figure A2.2 Buses in action (example).

A2.3 Built-In Chips

Chips can have either a *native* implementation, written in HDL, or a *built-in* implementation, supplied by an executable module written in a high-level programming language. Since the Nand to Tetris hardware simulator was written in Java, it was convenient to realize the built-in chips as Java classes. Thus, before building, say, a Mux chip in HDL, the user can load a built-in Mux chip into the hardware simulator and experiment with it. The behavior of the built-in Mux chip is supplied by a Java class file named Mux.class, which is part of the simulator's software.

The Hack computer is made from about thirty generic chips, listed in appendix 4. Two of these chips, Nand and DFF, are considered *given*, or *primitive*, akin to axioms in logic. The hardware simulator realizes given chips by invoking their built-in implementations. Therefore, in Nand to Tetris, Nand and DFF can be used without building them in HDL.

Projects 1, 2, 3, and 5 evolve around building HDL implementations of the remaining chips listed in appendix 4. All these chips, except for the CPU and Computer chips, also have built-in implementations. This was done in order to facilitate behavioral simulation, as explained in chapter 1.

The built-in chips—a library of about thirty *chipName*.class files—are supplied in the nand2tetris/tools/builtInChips folder in your computer. Built-in chips have HDL interfaces identical to those of regular HDL chips. Therefore, each .class file is accompanied by a corresponding .hdl file that provides the built-in chip interface. Figure A2.3 shows a typical HDL definition of a built-in chip.

```
/** 16-bit And gate, implemented as a built-in chip. */
CHIP And16 {
   IN  a[16], b[16];
   OUT out[16];
   BUILTIN And16;
}
```

Implemented by
`tools/builtInChips/And16.class`

Figure A2.3 Built-in chip definition example.

It's important to remember that the supplied hardware simulator is a general-purpose tool, whereas the Hack computer built in Nand to Tetris is a specific hardware platform. The hardware simulator can be used for building gates, chips, and platforms that have nothing to do with Hack. Therefore, when discussing the notion of built-in chips, it helps to broaden our perspective and describe their general utility for supporting any possible hardware construction project. In general, then, built-in chips provide the following services:

Foundation: Built-in chips can provide supplied implementations of chips that are considered *given*, or *primitive*. For example, in the Hack computer, Nand and DFF are given.

Efficiency: Some chips, like RAM units, consist of numerous lower-level chips. When we use such chips as chip-parts, the hardware simulator has to evaluate them. This is done by evaluating, recursively, all the lower-level chips from which they are made. This results in slow and inefficient simulation. The use of built-in chip-parts instead of regular, HDL-based chips speeds up the simulation considerably.

Unit testing: HDL programs use chip-parts abstractly, without paying attention to their implementation. Therefore, when building a new chip, it is always recommended to use built-in chip-parts. This practice improves efficiency and minimizes errors.

Visualization: If the designer wants to allow users to "see" how chips work, and perhaps change the internal state of the simulated chip interactively, he or she can supply a built-in chip implementation that features a graphical user interface. This GUI will be displayed whenever the built-in chip is loaded into the simulator or invoked as a chip-part. Except for these visual side effects, GUI-empowered chips behave, and can be used, just like any other chip. Section A2.5 provides more details about GUI-empowered chips.

Extension: If you wish to implement a new input/output device or create a new hardware platform altogether (other than Hack), you can support these constructions with built-in chips. For more information about developing additional or new functionality, see chapter 13.

A2.4 Sequential Chips

Chips can be either *combinational* or *sequential*. Combinational chips are time independent: they respond to changes in their inputs instantaneously. Sequential chips are time dependent, also called *clocked*: when a user or a test script changes the inputs of a sequential chip, the chip outputs may change only at the beginning of the next *time unit*, also called a *cycle*. The hardware simulator effects the progression of time using a simulated clock.

The clock: The simulator's two-phase clock emits an infinite sequence of values denoted 0, 0+, 1, 1+, 2, 2+, 3, 3+, and so on. The progression of this discrete time series is controlled by two simulator 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 *real time* that elapsed during this period is irrelevant for simulation purposes, since the simulated time is controlled by the user, or by a test script, as follows.

First, whenever a sequential chip is loaded into the simulator, the GUI enables a clock-shaped button (dimmed when simulating combinational chips). 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, the sequence of scripting commands repeat *n* {tick, tock, output} instructs the simulator to advance the clock *n* time units and to print some values in the process. Appendix 3 documents the *Test Description Language* (TDL) that features these commands.

The two-phased time units generated by the clock regulate the operations of all the sequential chip-parts in the implemented chip. During the first phase of the time unit (tick), the inputs of each sequential chip-part affect the chip's internal state, according to the chip logic. During the second phase of the time unit (tock), the chip outputs 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 tock—at the point of transition between two consecutive time units.

We reiterate that combinational chips are completely oblivious to the clock. In Nand to Tetris, all the logic gates and chips built in chapters 1–2, up to and including the ALU, are combinational. All the registers and memory units built in chapter 3 are sequential. By default, chips are combinational; a chip can become *sequential* explicitly or implicitly, as follows.

Sequential, built-in chips: 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 chip's input or output pins. The inclusion of an input pin *x* in the CLOCKED list stipulates that changes to *x* should affect the chip's outputs only at the beginning of the next time unit. The inclusion of an output pin *x* in the CLOCKED list stipulates that changes in any of the chip's inputs should affect *x* only at the beginning of the next time unit. Figure A2.4 presents the definition of the most basic, built-in, sequential chip in the Hack platform—the DFF.

```
/** D-Flip-Flop gate (DFF):
out[t]=in[t-1] where t is the current cycle, or time-unit. */
CHIP DFF {
   IN in;
   OUT out;
   BUILTIN DFF;
   CLOCKED in;
}
```

Implemented by `tools/builtInChips/DFF.class`

The `in` input is explicitly clocked

Figure A2.4 DFF definition.

It is possible that only some of the input or output pins of a chip are declared as clocked. In that case, changes in the non-clocked input pins affect the non-clocked output pins instantaneously. That's how the address pins are implemented in RAM units: the addressing logic is combinational and independent of the clock.

It is also possible to declare the CLOCKED keyword with an empty list of pins. This statement stipulates that the chip may change its internal state depending on the clock, but its input-output behavior will be combinational, independent of the clock.

Sequential, composite chips: The CLOCKED property can be defined explicitly only in built-in chips. How, then, does the simulator know that a given chip-part is sequential? If the chip is not built-in, then it is said to be clocked when one or more of its chip-parts is clocked. The 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) "clocked." Therefore, in the Hack computer, all the chips that include one or more DFF chip-parts, either directly or indirectly, are clocked.

We see that if a chip is not built-in, there is no way to tell from its HDL code whether it is sequential or not. *Best-practice advice*: The chip architect should provide this information in the chip API documentation.

Feedback loops: If the input of a chip feeds from one of the chip's outputs, either directly or through a (possibly long) path of dependencies, we say that the chip contains a *feedback loop*. For example, consider the following two chip-part statements:

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

In both examples, an internal pin (loop1 or loop2) attempts to feed the chip's input from its output, creating a feedback loop. The difference between the two examples is that Not is a combinational chip, whereas DFF is sequential. In the Not example, loop1 creates an instantaneous and uncontrolled dependency between in and out, sometimes called a *data race*. In contrast, in the DFF case, the in-out dependency created by loop2 is delayed by the clock, since the in input of the DFF is declared clocked. Therefore, out(t) is not a function of in(t) but rather of in(t - 1).

When the simulator evaluates a chip, it checks recursively whether its various connections entail feedback loops. For each loop, the simulator checks whether the loop goes through a clocked pin somewhere along the way. If so, the loop is allowed. Otherwise, the simulator stops processing and issues an error message. This is done to prevent uncontrolled data races.

A2.5 Visualizing Chips

Built-in chips may be *GUI empowered*. These chips feature visual side effects designed to animate some of the chip operations. When the simulator evaluates a GUI-empowered chip-part, it displays a graphical image on the screen. Using this image, which may include interactive elements, the user can inspect the chip's current state or change it. The permissible GUI-empowered actions are determined, and made possible, by the developer of the built-in chip implementation.

The present version of the hardware simulator features the following GUI-empowered, built-in chips:

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

Registers (ARegister, DRegister, PC): Displays the register's contents, which may be modified by the user.

RAM chips: Displays a scrollable, array-like image that shows the contents of all the memory locations, which may be modified by the user. If the contents of a memory location change during the simulation, the respective entry in the GUI changes as well.

ROM chip (ROM32K): Same array-like image as that of RAM chips, plus an icon that enables loading a machine language program from an external text file. (The ROM32K chip serves as the instruction memory of the Hack computer.)

Screen chip: Displays a 256-rows-by-512-columns window that simulates the physical screen. If, during a simulation, one or more bits in the RAM-resident *screen memory map* change, the respective pixels in the screen GUI change as well. This continuous refresh loop is embedded in the simulator implementation.

Keyboard chip: Displays a keyboard icon. Clicking this icon 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 appears in the RAM-resident *keyboard 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 A2.5 presents a chip that uses three GUI empowered chip-parts. Figure A2.6 shows how the simulator handles this chip. The GUIDemo chip logic feeds its in input into two destinations: register number address in the RAM16K chip-part, and register number address in the Screen chip-part. In addition, the chip logic feeds the out values of its three chip-parts to the “dead-end” internal pins a, b, and c. These meaningless connections are designed for one purpose only: illustrating how the simulator deals with built-in, GUI-empowered chip-parts.

```
// 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 its built-in chip-parts.
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 A2.5 A chip that activates GUI-empowered chip-parts.

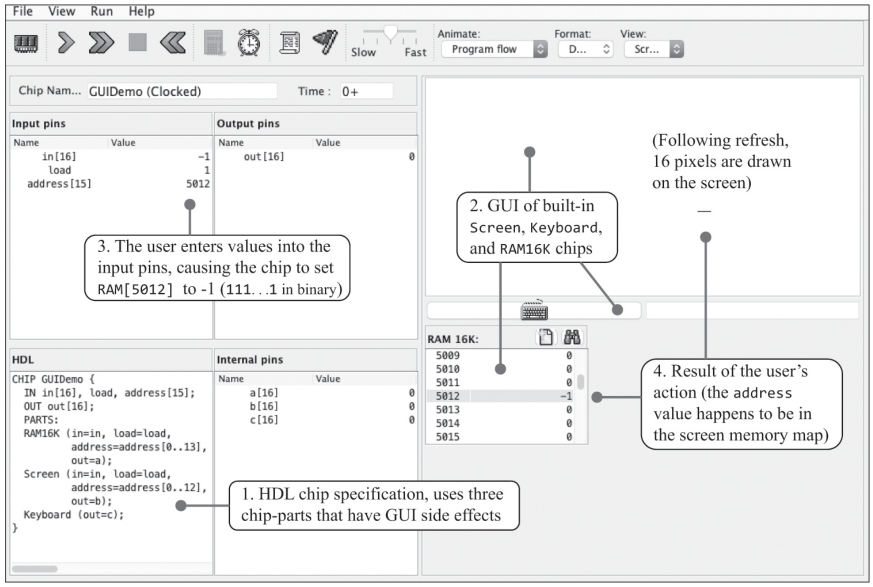

Figure A2.6 GUI-empowered chips demo. Since the loaded HDL program uses GUI-empowered chip-parts (step 1), the simulator renders their respective GUI images (step 2). When the user changes the values of the chip input pins (step 3), the simulator reflects these changes in the respective GUIs (step 4).

Note how the changes effected by the user (step 3) impact the screen (step 4). The circled horizontal line shown on the screen is the visual side effect of storing –1 in memory location 5012. Since the 16-bit two's complement binary code of –1 is 1111111111111111, the computer draws 16 pixels starting at column 320 of row 156, which happen to be the screen coordinates associated with RAM address 5012. The mapping of memory *addresses* on (*row*, *column*) screen coordinates is specified in chapter 4 (section 4.2.5).

A2.6 HDL Survival Guide

This section provides practical tips about how to develop chips in HDL using the supplied hardware simulator. The tips are listed in no particular order. We recommend reading this section once, beginning to end, and then consulting it as needed.

Chip: Your nand2tetris/projects folder includes thirteen subfolders, named 01, 02, …, 13 (corresponding to the relevant chapter numbers). The hardware project folders are 01, 02, 03, and 05. Each hardware project folder contains a set of supplied HDL *stub files*, one for each chip that you have to build. The supplied HDL files contain no implementations; building these implementations is what the project is all about. If you do not build these chips in the order in which they are described in the book, you may run into difficulties. For example, suppose that you start project 1 by building the Xor chip. If your Xor.hdl implementation includes, say, And and Or chip-parts, and you have not yet implemented And.hdl and Or.hdl, your Xor.hdl program will not work even if its implementation is perfectly correct.

Note, however, that if the project folder included no And.hdl and Or.hdl files, your Xor.hdl program will work properly. The hardware simulator, which is a Java program, features built-in implementations of all the chips necessary to build the Hack computer (with the exception of the CPU and Computer chips). When the simulator evaluates a chip-part, say And, it looks for an And.hdl file in the current folder. At this point there are three possibilities:

- No HDL file is found. In this case, the built-in implementation of the chip kicks in, covering for the missing HDL implementation.
- A stub HDL file is found. The simulator tries to execute it. Failing to find an implementation, the execution fails.
- An HDL file is found, with an HDL implementation. The simulator executes it, reporting errors, if any, to the best of its ability.

Best-practice advice: You can do one of two things. Try to implement the chips in the order presented in the book and in the project descriptions. Since the chips are discussed bottom-up, from basic chips to more complex ones, you will encounter no chip order implementation troubles—provided, of course, that you complete each chip implementation correctly before moving on to implement the next one.

A recommended alternative is to create a subfolder named, say, stubs, and move all the supplied .hdl stub files into it. You can then move each stub file that you want to work on into your working folder, one by one. When you are done implementing a chip successfully, move it into, say, a completed subfolder. This practice forces the simulator to always use built-in chips, since the working folder includes only the .hdl file that you are working on (as well as the supplied .tst and .cmp files).

HDL files and test scripts: The .hdl file that you are working on and its associated .tst test script file must be located in the same folder. Each supplied test script starts with a load command that loads the .hdl file that it is supposed to test. The simulator always looks for this file in the current folder.

In principle, the simulator's File menu allows the user to load, interactively, both an .hdl file and a .tst script file. This can create potential problems. For example, you can load the .hdl file that you are working on into the simulator, and then load a test script from another folder. When you execute the test script, it may well load a different version of the HDL program into the simulator (possibly, a stub file). When in doubt, inspect the pane named HDL in the simulator GUI to check which HDL code is presently loaded. *Best-practice advice*: Use the simulator's File menu to load either an .hdl file or a .tst file, but not both.

Testing chips in isolation: At some point you may become convinced that your chip is correct, even though it is still failing the test. Indeed, it is possible that the chip is perfectly implemented, but one of its chip-parts is not. Also, a chip that passed its test successfully may fail when used as a chip-part by another chip. One of the biggest inherent limitations of hardware design is that test scripts—especially those that test complex chips—cannot guarantee that the tested chip will operate perfectly in all circumstances.

The good news is that you can always diagnose which chip-part is causing the problem.

Create a test subfolder and copy into it only the three .hdl, .tst, and .out files related to the chip that you are presently building. If your chip implementation passes its test in this subfolder as is (letting the simulator use the default built-in chip-parts), there must be a problem with one of your chip-part implementations, that is, with one of the chips that you built earlier in this project. Copy the other chips into this test folder, one by one, and repeat the test until you find the problematic chip.

HDL syntax errors: The hardware simulator displays errors on the bottom status bar. On computers with small screens, these messages are sometimes off the bottom of the screen, not visible. If you load an HDL program and nothing shows up in the HDL pane, but no error message is seen, this may be the problem. Your computer should have a way to move the window, using the keyboard. For example, on Windows use Alt+Space, M, and the arrow keys.

Unconnected pins: The hardware simulator does not consider unconnected pins to be errors. By default, it sets any unconnected input or output pin to false (binary value 0). This can cause mysterious errors in your chip implementations.

If an output pin of your chip is always 0, make sure that it is properly connected to another pin in your program. In particular, double-check the names of the internal pins ("wires") that feed this pin, either directly or indirectly. Typographic errors are particularly hazardous here, since the simulator doesn't throw errors on disconnected wires. For example, consider the statement Foo(…, sum=sun), where the sum output of Foo is supposed to pipe its value to an internal pin. Indeed, the simulator will happily create an internal pin named sun. Now, if sum's value was supposed to feed the output pin of the implemented chip, or the input pin of another chip-part, this pin will in fact be 0, *always*, since nothing will be piped from Foo onward.

To recap, if an output pin is always 0, or if one of the chip-parts does not appear to be working correctly, check the spelling of all the relevant pin names, and verify that all the input pins of the chip-part are connected.

Customized testing: For every *chip*.hdl file that you have to complete your project folder also includes a supplied test script, named *chip*.tst, and a compare file, named *chip*.cmp. Once your chip starts generating outputs, your folder will also include an output file named *chip*.out. If your chip fails the test script, don't forget to consult the .out file. Inspect the listed output values, and seek clues to the failure. If for some reason you can't see the output file in the simulator GUI, you can always inspect it using a text editor.

If you want, you can run tests of your own. Copy the supplied test script to, say, *MyTestChip*.tst, and modify the script commands to gain more insight into your chip's behavior. Start by changing the name of the output file in the output-file line and deleting the compare-to line. This will cause the test to always run to completion (by default, the simulation stops when an output line disagrees with the corresponding line in the compare file). Consider modifying the output-list line to show the outputs of your internal pins.

Appendix 3 documents the Test Description Language (TDL) that features all these commands.

Sub-busing (indexing) internal pins: This is not permitted. The only bus-pins that can be indexed are the input and output pins of the implemented chip or the input and output pins of its chip-parts. However, there is a workaround for sub-busing internal bus-pins. To motivate the work-around, here is an example that doesn't work:

```
CHIP Foo {
     IN  in[16];
     OUT out;
     PARTS:
     Not16  (in=in, out=notIn);
     Or8Way (in=notIn[4..11], out=out);   // Error: internal bus cannot be indexed.
  }
```

Possible fix, using the work-around:

```
Not16 (in=in, out[4..11]=notIn);
Or8Way (in=notIn, out=out); // Works!
```

Multiple outputs: Sometimes you need to split the multi-bit value of a bus-pin into two buses. This can be done by using multiple out= bindings.

For example:

```
CHIP Foo {
     IN  in[16];
     OUT out[8];
     PARTS:
     Not16 (in=in, out[0..7]=low8, out[8..15]=high8);   // Splitting the out value
     Bar8Bit (a=low8, b=high8, out=out);
  }
```

Sometimes you may want to output a value and also use it for further computations. This can be done as follows:

```
CHIP Foo {
    IN a, b, c;
    OUT out1, out2;
    PARTS:
    Bar (a=a, b=b, out=x, out=out1); // Bar's output feeds the out1 output of Foo
    Baz (a=x, b=c, out=out2);        // A copy of Bar's output also feeds Baz's a input
}
```

Chip-parts “auto complete” (sort of): The signatures of all the chips mentioned in this book are listed in appendix 4, which also has a web-based version (at www.nand2tetris.org). To use a chip-part in a chip implementation, copy-paste the chip signature from the online document into your HDL program, then fill in the missing bindings. This practice saves time and minimizes typing errors.