Appendix A:

Hardware Description Language (HDL)

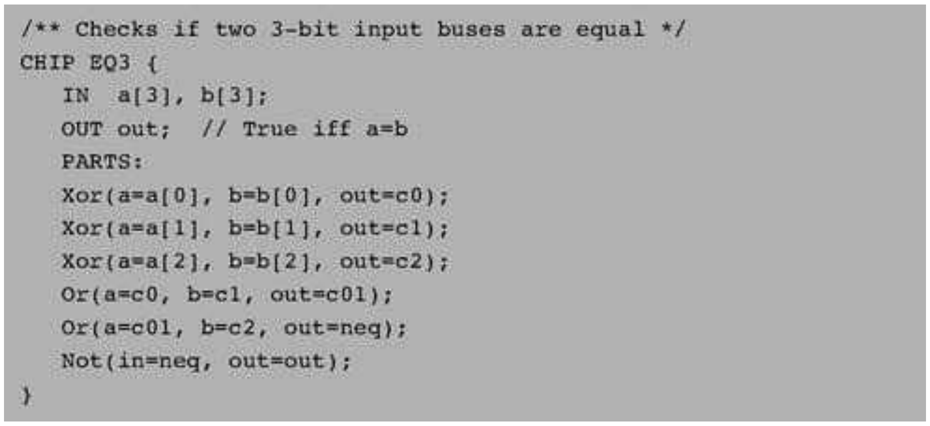
Intelligence is the faculty of making artificial objects, especially tools to make tools.
—Henry Bergson (1859-1941)

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.

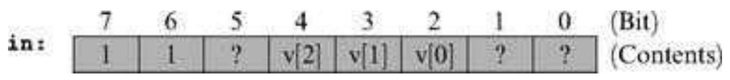
Example

```
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:

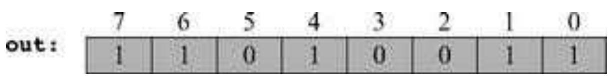
```
Foo(in[2..4]=v, in[6..7]=true, out[0..3]=x, out[2..6]=y)
```

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:



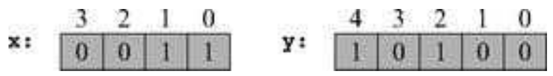
	7	6	5	4	3	2	1	0	(Bit)
in:	1	1	?	v[2]	v[1]	v[0]	?	?	(Contents)

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