CHAPTER 12

The JHDL Design and Debug System

Brent Nelson, Brad Hutchings
Department of Electrical and Computer Engineering
Brigham Young University

JHDL [1, 8] is a CAD environment developed at Brigham Young University for the design, debug, and runtime control of configurable computing applications based on field-programmable gate array (FPGA) technology. Developed roughly between 1997 and 2003 it was made available under an open-source license (*http://www.jhdl.org*) in approximately 2000. The term *JHDL* can refer to one of two things: (1) the JHDL circuit design language itself, or (2) the JHDL CAD system. The JHDL language is a text-based design language for algorithmic construction of structured circuits that is embedded within the Java programming language. JHDL designs are created as Java programs that access JHDL libraries to generate circuits. Within the JHDL CAD environment, circuits can be simulated, netlisted, and downloaded to the reconfigurable computing platform for execution and testing. Additional CAD tools can be built on top of the JHDL infrastructure to support higher-level circuit construction, optimization, and debugging tasks. One of the most unique features of JHDL is its runtime environment, which provides a unified simulator/hardware debugger that can be used to debug and validate a circuit through either simulation or hardware execution, and which contains many features normally found only in source-level software debuggers.

12.1 JHDL BACKGROUND AND MOTIVATION

Historically, FPGA designers have used CAD tools from three sources to develop their designs. The early tools were derived from application-specific integrated circuit (ASIC) tool flows such as schematic capture, HDL synthesis, and so on. Some were invented as new languages or language dialects specifically for FPGA design [4]. Finally, some designers have used general-purpose programming languages (GPLs) to describe FPGA circuitry [3, 9]. Although there are good reasons behind all three tool approaches, the case for using GPLs for FPGA design is quite compelling. Compared to the other two alternatives (schematic capture and HDL synthesis), GPLs are much more accessible to a larger set of users and can be applied to a much broader set of problems. In addition, GPL programming environments are less expensive, more widely available, and more mature (less buggy) than the other two alternatives.

Within the realm of GPL-based design tools, a range of approaches as well as design abstraction levels can be (and have been) supported. Sea Cucumber [11] is representative of high-level tools that *compile* standard programming language descriptions into hardware. In this case, the tie to GPLs is simply that the input specification syntax used is based on a GPL. A different approach is represented by *structural* design languages that leverage GPL language constructs to assist the user in creating a circuit from a set of building blocks (gates, wires, etc.).

JHDL is an *embedded design language* based on the Java GPL and is a structural design tool. Embedded languages like JHDL are specialized application programming interfaces (APIs) where user-defined classes and function overloading are carefully used to create the illusion of a customized circuit design language within the GPL environment. APIs allow designers to build circuits by declaring interfaces and interconnecting gates and modules, all in a structural way. Embedding does this without making any modifications to existing language syntax, which is an important point because modifying the GPL syntax negates most of the advantages of the embedding approach. Examples of past embedded languages include PAMDC [2] and Spyder [9].

As a structural design tool, JHDL constructs circuits from library primitives with the help of provided Java methods (subroutines) and module generators whose execution produces a circuit graph. This graph is then available for manipulation, including simulation and netlisting. In this era of behavioral synthesis, why are we still interested in structural design? There are three answers to this question. First, when working with FPGAs, structural design techniques often still result in circuits that are substantially smaller and faster than those developed using only behavioral synthesis tools. Second, for many applications found in the reconfigurable computing arena (especially where control over circuit placement is required), structural capture is simply a faster, easier to learn, and more effective way to design an application. Thus, it has a place in any high-performance FPGA design tool kit. Third, the circuit graph produced by the execution of a JHDL description is amenable to a variety of modifications prior to netlisting. For example, it can be programmatically modified to insert debug support features, it can be instrumented to support runtime profiling and monitoring of the final hardware, and it can be modified to support checkpointing (extraction of the hardware computation's state for later restoration) and therefore support context switching of designs on and off a configurable computing platform. None of these features are as readily performed using other approaches, especially behavioral synthesis approaches.

An overview of the design process for JHDL-based design is presented in Figure 12.1. As shown, a collection of JHDL class libraries provides the foundation for all JHDL designs. These libraries contain, at a minimum, Java classes representing primitive circuit elements. Layered on top of the device primitives library are additional libraries that contain subroutines to programmatically generate higher-level circuits from the primitives (known variously as *module generators*). A user creates a JHDL design by writing a Java program that instances these library primitives or calls the module generator subroutines that, in turn, instance primitives.

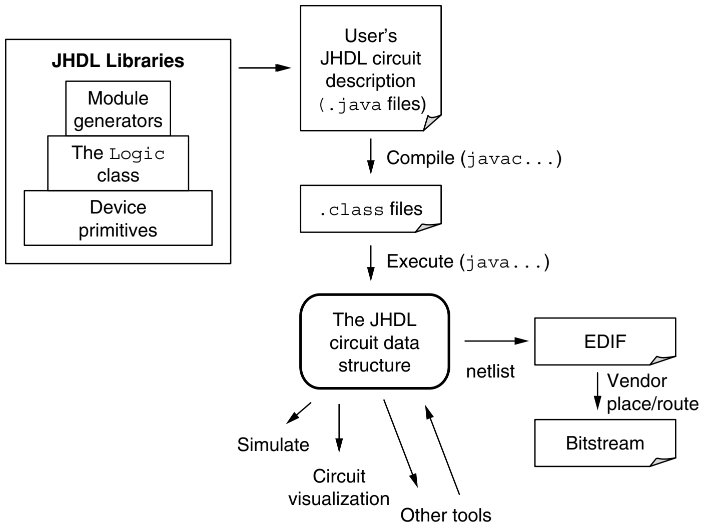


FIGURE 12.1 ■ An overview of the design process and the JHDL system.

Once a JHDL Java program has been written and compiled to a set of class files, it may be executed. The result is an in-memory data structure representing the constructed circuit in the form of a graph, in which nodes represent circuit elements and wires, and arcs represent connections between them. As shown in Figure 12.1, once this data structure has been built, various CAD tools can be applied to it to accomplish simulation, netlisting, or other desired activities. Of interest is that tools can be used to modify the JHDL circuit data structure prior to netlisting or simulation. This is shown in the figure by the arrow leading from "Other tools" up to "The JHDL circuit data structure." These modifications can be for purposes of adding debug or in-circuit monitoring features, and so on.

12.2 THE JHDL DESIGN LANGUAGE

As noted, because JHDL is *embedded*, two mechanisms are used to create the illusion of it as a customized circuit design language: classes and function overloading. The predefined classes provided by JHDL represent primitives, such as gates and wires, so one design method is to simply create instances of these primitives using the Java `new` construct. Beyond this, function overloading provides a higher level of design abstraction by allowing the designer to call parameterized functions that build the desired circuit out of primitive objects. This section describes these levels of JHDL design from a circuit designer's perspective.

12.2.1 Level-1 Design: Primitive Instantiation

The JHDL primitives library shown in Figure 12.1 is simply a package of Java classes where each class corresponds to a circuit primitive (e.g., AND, OR). Given such a library, the lowest level of JHDL design is to instance primitives from it using `new`.

Listing 12.1 shows a simple design built by instantiating primitives. The first two lines import general JHDL libraries needed by all designs. The third import makes the primitives from JHDL's Xilinx Virtex library available for use in the construction of this design.

The `mux` class is next declared by subclassing (extending) the JHDL `Logic` class. The interface ports (the named inputs and outputs for the cell) are declared using the `CellInterface` mechanism where, for example, `in ("sel", 1)` declares an input port named `sel` that is of width 1 and `out ("q", 1)` declares an output port named `q` that is also of width 1.

Listing 12.1 ■ Multiplexer example using primitive instantiation.

```
import byucc.jhdl.base.*;
import byucc.jhdl.Logic.*;
import byucc.jhdl.Xilinx.Virtex.*;

// This cell is a Java class called 'mux'
public class mux extends Logic {

  // Declare the cell's ports
  public static CellInterface[] cell_interface = {
    in("a", 1),
    in("b", 1),
    in("sel", 1),
    out("q", 1),
  };

  // This is the mux's constructor
  public mux(Node parent, Wire aw, Wire bw, Wire selw, Wire qw) {
   super(parent);
    connect("a", aw); connect("b", bw);
    connect("sel", selw); connect("q", qw);

    // The code below this point is the 'body' of the cell and builds
    //   it from primitive wire and gate objects.

    // Declare and construct local wires
    Wire a1 = new Xwire(this, 1, "a1");
    Wire a2 = new Xwire(this, 1, "a2");
    Wire selbar = new Xwire(this, 1, "selbar");

    // Invert signal "sel"
    new inv(this, selw, selbar);
    // Form AND gates
    new and2(this, aw, selbar, a1);
    new and2(this, bw, sel, a2);

    // Form OR gate for final output
    new or2(this, a1, a2, qw);
  }
}
```

The declaration of the constructor for the mux class comes next. This is a standard Java constructor method that can be called to construct a new instance of mux. The connect() calls associate a given wire with a specific port; for example, the wire parameter aw is associated with (connected to) port a.

The last section of the constructor instantiates the wires and gates needed to implement the multiplexer logic using Java new... calls. The objects being created to build the circuit are implemented by Java classes from the byucc.jhdl.Xilinx.Virtex package and represent wires and logic gates.

The problem with using primitive instantiation, as just described, is that the resulting design is specific to a particular primitive library (the example above relies on the byucc.jhdl.Xilinx.Virtex package). Designing this way limits the portability of the design between technologies, even when it is based on building blocks as simple as individual Boolean gates. Another problem with this design style is that it was specifically written for a multiplexer that has single-bit inputs and outputs—in essence, it is a fixed netlist. The Logic class overcomes these limitations.

12.2.2 Level-2 Design: Using the Logic Class and Its Provided Methods

The Logic class consists of a large collection of subroutines that can be called to create user logic. Listing 12.2 shows the design of the same multiplexer (Listing 12.1) written using methods of the Logic class. The difference between this and the previous design is that at the bottom of the constructor, rather than primitive instantiation, this version uses method calls to build the MUX circuit. These methods are available for our use because the mux class extends the predefined Logic class. In Listing 12.2, the changes from the previous MUX example are underlined, to show how the portion of the code that actually builds the logic has been changed.

Listing 12.2 ■ MUX example written using Logic class.

```
import byucc.jhdl.base.*;
import byucc.jhdl.Logic.*;
import byucc.jhdl.Xilinx.Virtex.*;

// This cell is a Java class called 'mux'
public class mux extends Logic {

  // Declare the cell's ports
  public static CellInterface[] cell_interface = {
    in("a", 1),
    in("b", 1),
    in("sel", 1),
    out("q", 1),
  };

  // This is the mux's constructor
  public mux(Node parent, Wire aw, Wire bw, Wire selw, Wire qw) {
    super(parent);
    connect("a", aw); connect("b", bw);
    connect("sel", selw); connect("q", qw);

    // The code below this point is the 'body' of the cell and builds
    //   it from Logic class subroutine calls.

    or_o(this, and(aw, not(selw)), and(bw, sel), qw);
  }
}
```

Invoking `and (a,b)` calls `byucc.jhdl.Logic.and(a,b)`, which is a subroutine that builds the desired logic (an AND gate) and returns a reference (pointer) to the output wire it created for the gate. This wire can then be used as an input to the `or_o()` call, which creates a 2-input OR gate.[1]

In addition to less verbosity, tremendous power derives from using methods (subroutines) to build circuitry in this manner. OR methods with as many inputs as desired can be created to accommodate any size OR gate and can be written to accommodate input/output wires of any width. Thus, the overloaded `or()` subroutine can handle requests for 2-input OR gates with single-bit inputs/outputs as well as requests for 8-input OR gates with 32-bit inputs/outputs.

The `Logic` class methods accomplish this using JHDL `Techmapper` classes. Figure 12.2 shows that when user code calls a `Logic` method, that method ultimately calls a `Techmapper` class object to do a technology-specific implementation of the logic it has determined should be built, and the `Techmapper` object ultimately maps the resulting logic to technology-specific primitives. This means that designs created using `Logic` class methods are completely technology independent—retargeting a design created using `Logic` to a different technology is as simple as instructing the `Logic` class object to call on a different technology's `Techmapper`. To date, `Techmappers` have been written at Brigham Young University for the Xilinx: 4K, Virtex, Virtex-II, and Virtex-II Pro technologies.

The `Logic` class contains methods to build gates, wires, registers, memories, multiplexers, adders, subtracters, and shifters, as well as methods for manipulating wires: concatenation, slicing, and so forth. Users are encouraged to use the `Logic` style of Listing 12.2 instead of the primitive style of Listing 12.1 whenever possible, as primitive instantiation is typically used only for taking advantage of device-specific features such as clock managers and memories.

[1] Note that some of the function calls have an `an_o` suffix. Functions with this suffix instantiate the gate using the provided input and output wires. Functions without this suffix instantiate both the gate and an output wire that is connected to it. In either case, the output wire is returned by the function. This approach reduces verbosity by eliminating the need to declare and construct intermediate wires.

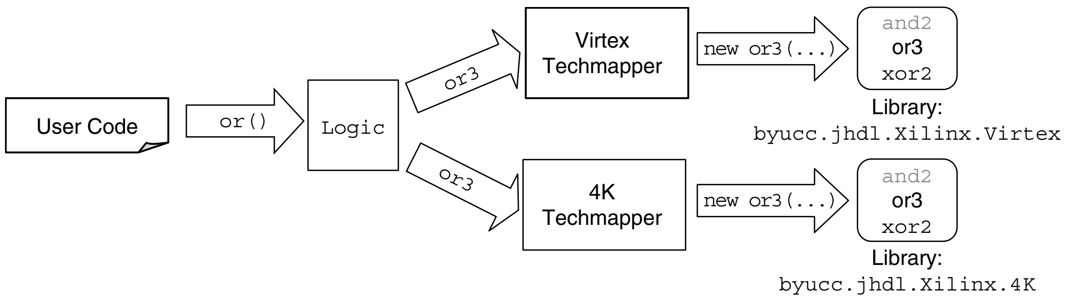


FIGURE 12.2 ■ The relationship of user code, `Logic` class methods, and `Techmapper` objects.

12.2.3 Level-3 Design: Programmatic Circuit Generation (Module Generators)

The creation of programmatic circuit generators (module generators) is a natural extension of the techniques employed by the `Logic` class. That is, Java-based subroutines that intelligently create complex hardware modules based on build time–supplied parameters can be created by any JHDL user. A very simple example that illustrates parameterized design is shown in Listing 12.3.

Listing 12.3 ■ n-bit full adder example.

```
// This design assumes the existence of a FullAdder JHDL design
//   which it instances repeatedly to build an n-bit adder.
import byucc.jhdl.base.*;
import byucc.jhdl.Logic.*;
public class NBitAdder extends Logic {
  public static CellInterface[] cell_interface = {
    param("n", INTEGER),
    in("a", "n"),
    in("b", "n"),
    out("sum", "n+1")
  };

  public NBitAdder(Node parent, Wire a, Wire b, Wire sum) {
    super(parent);  // Always call super-constructor
    int width = a.getWidth();  // Get the width of the 'a' wire
    bind("n", width);
    connect("a", a);    connect("b", b);    connect("sum", sum);

    // Create intermediate carry wires as a multi-bit wire
    Wire carries = wire(width);

    // Build and connect together needed full adders
    // The gw() method calls pull individual bits
    //   out of multi-bit wires.  The gnd() method returns
    //   a single constant '0' wire.
    for (int i=0; i < width; i++) {
      if (i==0)
        new FullAdder(this, a.gw(i), b.gw(i), gnd(),
                      sum.gw(i), carries.gw(0));
      else
        new FullAdder(this, a.gw(i), b.gw(i), carries.gw(i-1),
                      sum.gw(i), carries.gw(i));
    }
    buf_o(carries.gw(width-1), sum.gw(width));
  }
}
```

This is an `NBitAdder` design[2] that programmatically constructs a multibit adder using previously designed full adder cells (not shown). In its `CellInterface` declaration, the first line, `param("n", INTEGER)`, declares a parameter `n` that is of type `integer` (similar to a generic in VHDL—see Section 6.1.3). More precisely, `n` is declared to be an instance of the Java class `INTEGER`. All port declarations in the `CellInterface` then use `n` or `n+1` as their width. When `NBitAdder` is constructed, the `bind()` call binds the value of `n` to the width of the `a` wire. Based on this information, `connect()` calls will verify that the wires passed in to the constructor are the correct width for the ports they are being connected to. Finally, the ripple-carry adder body is constructed using a Java `for` loop that creates and interconnects `FullAdder` cells. The final `buf_o()` call connects the top carry-out bit to the most significant bit of the sum.

`NBitAdder` is a trivial example of a module generator, that is, a circuit that can be parameterized according to some set of criteria. Listing 12.4 is a slightly more complex version of the `NBitAdder` design that has been parameterized for pipelining (additions to the original design have been underlined in the source code). Here a single Boolean parameter `"pipe"` is passed into the cell constructor method to control whether a pipeline register is to be placed on the adder output. The main difference between this and the previous design is that the last few lines of the constructor body connect the adder outputs to the cell's outputs through either a register or a buffer, based on the value of the `"pipe"` parameter.

Listing 12.4 ■ `n`-bit full adder with optional pipelining.

```
... // Same imports as previous NBitAdder design
public class NBitAdder extends Logic {
  public static CellInterface[] cell_interface = {
    ... // Same ports as previous NBitAdder design
  };

  public NBitAdder(Node parent, Wire a, Wire b, Wire sum, Boolean pipe) {
```

[2] The `Logic` class contains a family of multibit adder constructor methods that would normally be called instead of this example design. Nevertheless, this design is presented here to illustrate module generator-like concepts in JHDL.

```
    ... // Main constructor body  same as previous NBitAdder design
    Wire tmpsum;

    // New code is below
    // If desired, insert pipeline latch
    if (pipe)
      (reg_o(tmpsum, sum);
    else
      buf_o(tmpsum, sum);
  }
}
```

On the surface this is similar to what can be accomplished through the use of VHDL generics: the `for-generate` and `if-generate` statements. However, parameterized circuit generation is limited in VHDL, consisting of very simple conditional circuit instantiations that are controlled by a small subset of the language dedicated solely to this purpose. In JHDL, the entire Java language can be brought to bear on this problem and sophisticated algorithms can be used to generate circuits. JHDL module generators exist for counters, comparators, accumulators, arithmetic units (multipliers, dividers, floating-point units, digit serial units), decoders, shift registers, and memories. These have employed, as a part of the module generators' calculations, simple timing and area estimation techniques, recursive tree search computations, file I/O, and the like. Such module generators have been parameterized for features such as number format, rounding/saturation/truncation modes, pipelining granularity, constant encoding methods, and resource usage (serial versus parallel implementation).

12.2.4 JHDL Is a Structural Design Language

Structural design often improves the performance of configurable computing applications because many applications that are FPGA based can benefit from manual placement of at least some parts of the design. Effective manual placement can be achieved only if the overall organization of the circuit is well understood—it is very difficult to manually place circuitry generated by behavioral synthesis.

Placement attributes can be attached to JHDL primitive circuit objects as string properties, to be interpreted by backend tools. To simplify the attaching of these attributes when `Logic` methods are used in circuit building, the `Logic` class also contains a placement API to help in the tasks of (1) mapping gates to lookup tables (LUTs), ALUs, or other atomic FPGA cells, and (2) specifying the relative placement of those cells. For example, to force a collection of gates that implement a 3-input, 1-output logic function into a single LUT, the `map()` call can be used as in:

```
map(a, b, ci, s );
```

This will force the cone of logic with `a`, `b`, and `ci` as inputs and with `s` as output into a single primitive (a LUT for most FPGA technologies). Then that primitive can be placed by specifying the location of *its output wire* in a `place()` call:

```
place( s, "R0C0.F" );
```

Note that these methods *do not* create logic themselves, but rather pack already created logic into LUTs, which they then physically place. The use of these method calls is technology specific, so a technology-specific `Techmapper` is used to determine their interpretation for the target technology at build time. This placement API acts as a window of opportunity for the user to obtain design assistance from the `Techmapper`. For example, when `map()` is called, the `Techmapper` checks the network of gates for validity (i.e., intermediate fanin or fanout to the network), and, when the circuit is fully constructed, it resolves all placement hints and reports any placement conflicts. In this way placement errors can be detected at the *front* of the tool chain rather than during place and route, which helps minimize design cycles.[3]

12.2.5 JHDL Is a Programmatic Circuit Design Language

That JHDL is also a programmatic circuit design language is perhaps the most powerful and unique feature of GPL-based circuit generation techniques. The key point is that a JHDL description, once compiled, *is* an executable Java program; it is the execution of that Java program that constructs the circuit. This gives JHDL significant advantages over HDL descriptions, which must be *parsed* by a synthesizer and a corresponding circuit then constructed.

With JHDL there is no separation between the code that represents the circuit itself and any code that might be executed to help determine how best to generate it—all of the code in a JHDL description is executable Java. In a language like JHDL there is a very clear separation between circuit generation and computation: Module instantiation is circuit generation and everything else is computation. In contrast, all code written in a VHDL or Verilog design (excepting simulation testbenches) *is* the circuit description—there is no provision for code that can be executed apart from it. This presents difficulties when computations are required, prior to circuit construction, to determine how best to generate the circuit.

At one time, designers often resorted to macro preprocessors with Verilog code to provide `for-generate`- and `if-generate`-like functionality for their designs. Similarly, some designers (including our own students) have often written C or C++ circuit generators that generate VHDL or Verilog code as output in order to work around the lack of an effective compile time computational capability in conventional HDLs. In contrast, JHDL and other GPL-based embedded languages avoid such workarounds because they provide a clean mechanism

[3] In contrast, VHDL annotation approaches for placement are nonstandard and differ from tool to tool (see Section 6.2.1). Also, VHDL placement directives are passed through to the backend tools without performing any error checking such as described previously.

for freely intermixing computational code with circuit descriptions—all based on the general-purpose computational power of the underlying GPL. One could say that languages like JHDL don't need a formal elaboration step as VHDL and Verilog do. Or one could say that the *entire* circuit construction process in JHDL is an elaboration step, albeit a much more powerful one than that provided by HDLs.

Finally, there is no *synthesizable subset* of JHDL, and thus there is no possibility for a mismatch between simulation and synthesis results due to differing CAD tools' interpretation of the description. The same circuit is constructed each time the JHDL code is executed regardless of whether it is intended for simulation or for netlisting.

12.3 THE JHDL CAD SYSTEM

As Figure 12.1 showed, the execution of a compiled JHDL design creates an in-memory structural representation of the JHDL circuit. This is a classical circuit graph where Java objects represent the cells and wires in the circuit and pointers between these objects represent connections and hierarchical parent–child relationships. The figure also showed that this circuit data structure is the entry point for the JHDL CAD system, meaning that all CAD functions and tasks, such as simulation and netlisting, use the circuit data structure via an API provided for this purpose. The result is that it is straightforward to write Java-based CAD tools for interacting with and manipulating the circuit data structure (and therefore the circuit).

12.3.1 Testbenches in JHDL

Because a JHDL design is a Java program, it needs a `main()` routine. It is the program's `main()` routine that usually acts as a testbench for JHDL designs. Listing 12.5 shows such a `main()` testbench that, like most JHDL testbenches, does three things:

- First, it creates an `HWSystem` object that is the top-level container object for the circuit and contains the simulator and netlister objects. The entire user design (testbench and device under test) exists as a child node of `HWSystem` in the resulting JHDL object hierarchy.

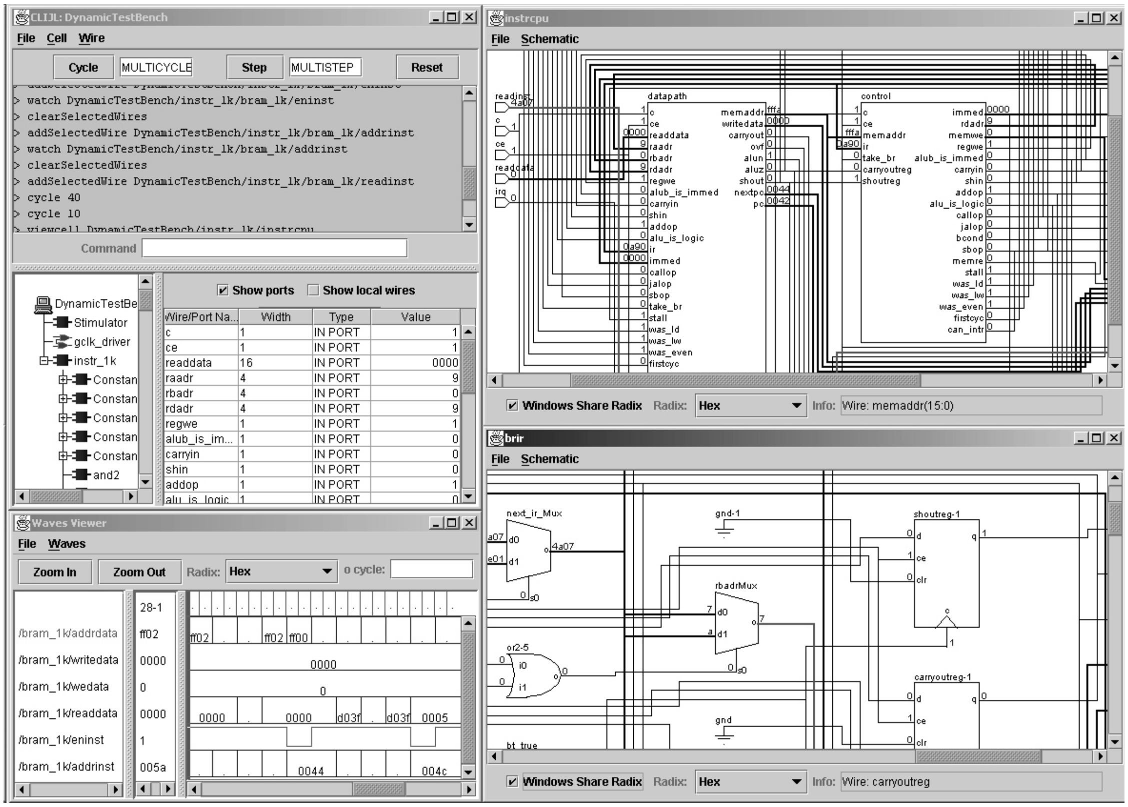
Listing 12.5 ■ A sample JHDL testbench.

```
import ...;    // Import needed packages

// Declare testbench class
public class tb_myCell extends Logic implements TestBench {

  static HWSystem hw;   // Declare a HWSystem
  private int aVal, bVal, cinVal;  // Declare some private variables
  // The main() routine for this Java program
  public static void main(String argv[]) {
    // Step 1: build a HWSystem
    hw = new HWSystem();  // Build a HWSystem
    // Step 2: Build an instance of this testbench
    tb_myCell tb = new tb_myCell(hw, ...); // Pass in some params
    // Step 3: Do something with the circuit now that the testbench
    // and DUT are built.  We can do any one of:
    // 1. Start a simulation
    // 2. Netlist the circuit
    // 3. Traverse or modify the circuit data structure
    // 4. Start a GUI-based CAD system
    // We will do the last - create a GUI-based CAD system
    // Create a new instance of cvt (the Circuit Visualization Tool)
    new cvt( tb );
  }

  // The constructor for this testbench
  public tb_myCell (Node parent, ...);  // Not all params shown
    super(parent);

    // Step 1: Specify (create) a TechMapper for Virtex
    setDefaultTechMapper(new VirtexTechMapper(true));

    // Step 2: Build wires to connect to DUT
    an = wire(1,"an");   bn = wire(1,"bn");   cinn = wire(1, "cinn");
    sn = wire(1,"sn");   coutn = wire(1,"coutn");

    // Step 3: Build mycell (the DUT)
    myCell dut = new myCell(this, an, bn, cinn, sn, coutn, "myCell");
  }
}
```

- Second, as shown in Listing 12.5, it creates a testbench object (which in turn creates the device under test).
- Third, once the JHDL circuit data structure has been created, the `main()` routine can do one of a number of things: (1) start a batch simulation, (2) call on the netlister to netlist the design, or (3) create a GUI-based interface to enable the user to interactively work with the circuit. In Listing 12.5, the `main()` routine starts up `cvt`, a graphical environment for viewing the circuit, simulation, and netlisting.

12.3.2 The `cvt` Class

Class `cvt` is a GUI-based system with widgets for navigating the design hierarchy, starting a simulation of the circuit, generating a netlist of the circuit, and so forth. The actual simulation and netlisting classes are accessed via the `HWSystem` class, and `cvt` makes calls into it to satisfy user requests. The `cvt` class implements a standard event-driven GUI system based on Swing, distributed as a part of the JHDL language. Swing was chosen for its portability and availability on all



```
> watch DynamicTestBench/instr_1k/bram_1k/eninst
> clearSelectedWires
> addSelectedWire DynamicTestBench/instr_1k/bram_1k/addrinst
> watch DynamicTestBench/instr_1k/bram_1k/addrinst
> clearSelectedWires
> addSelectedWire DynamicTestBench/instr_1k/bram_1k/readinst
> cycle 40
> cycle 10
> viewcell DynamicTestBench/instr_1k/instrcpu
```

Wire/Port Na...	Width	Type	Value
c	1	IN PORT	1
ce	1	IN PORT	1
readdata	16	IN PORT	0000
raadr	4	IN PORT	9
rbadr	4	IN PORT	0
rdadr	4	IN PORT	9
regwe	1	IN PORT	1
alub_is_im...	1	IN PORT	0
carryin	1	IN PORT	0
shin	1	IN PORT	0
addop	1	IN PORT	1
alu_is_logic	1	IN PORT	0

FIGURE 12.3 ■ The JHDL `cvt` GUI.

platforms. Class `cvt` uses the built-in Swing event mechanism for its own internal communication.

Figure 12.3 shows a screenshot of the `cvt` GUI. In the upper left is a text console window where commands may be typed. Menus and buttons above largely duplicate what can be entered in the window. Below the console is a hierarchy navigation tool. On the left is a hierarchy browser; on the right is a list of ports for the currently selected cell along with their current values (if a simulation is in progress). Beneath the browser is a waveform viewer; and on the lower right half of the figure are two different schematic viewers. This screenshot shows that the various parts of the GUI are all contained in a single pane but each can be broken out into an individually sized window if desired.

Unlike with most CAD tools, there is no *standard* JHDL CAD system. Rather, the circuit data structure API provides a mechanism for any program a user might write to interact with the circuit. The `cvt` class is simply one such example. Examples of other programs include stand-alone simulators and netlisters.

Many of the debugging experiments described later in this chapter were carried out by writing custom CAD tools to interact with the circuit data structure API. For example, a number of tools have been written that modify the circuit prior to netlisting by, for example, inserting clock managers or other special-purpose circuitry into the user's design. These tools have also instrumented designs for debug by adding scan chains to them. Finally, complete software applications that interact with the circuit during simulation and execution have been created. This last point is a unique feature of JHDL—once the circuit has been built, application software can be written that communicates with the design via the `HWSystem` API. This allows the complete application (software and hardware) to be deployed as a single Java program.

12.4 JHDL'S HARDWARE MODE

JHDL supports hardware-in-the-loop debugging with what is called *hardware mode*. Hardware mode is based on the observation that much of the data created when a JHDL circuit is built and simulated is also useful in the actual hardware debug process.

Figure 12.4 shows JHDL's dual simulation/hardware execution environment. When initially simulating a design (*left* side of figure), the simulation/runtime API provides `cvt` and simulator access to the JHDL circuit graph.

After the design's configuration bitstream is created, hardware debugging can take place using hardware mode (on the right of Figure 12.4). Loading a JHDL design in `cvt` now performs two steps: (1) the JHDL design is constructed as usual to create the internal circuit representation, and (2) the bitstream is configured onto the specified FPGA platform. Using the same `cvt` GUI as before, the user can advance execution of the design via the simulator control buttons or via commands on the command line. However, instead of cycling the simulator, these actions cause `cvt` to send clocking commands to the FPGA platform through the board's driver. After a clock command is executed, the state of the FPGA platform is retrieved using readback and *back-annotated into the JHDL circuit data structure*.

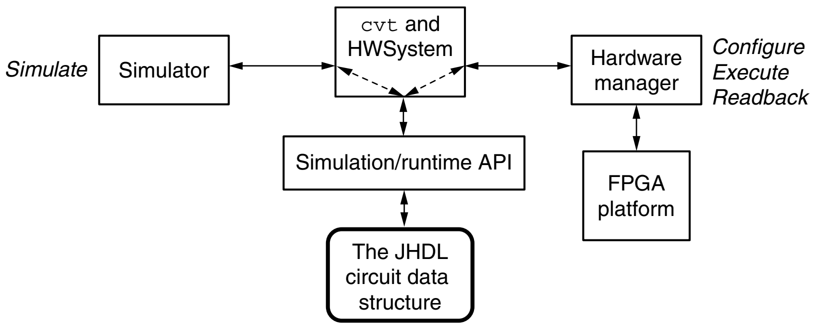


FIGURE 12.4 ■ The JHDL unified simulation/hardware execution environment.

The simulator is then used to compute the steady state of all combinational nodes in the circuit as a function of these state values, and a complete picture of the hardware execution state is now present in the JHDL circuit data structure. As a result, `cvt` can query and display the state of the circuit as normal, just as it would if the circuit were being simulated. In this case, however, it displays hardware signal values rather than simulated signal values.

JHDL's hardware mode is readily adapted to new hardware platforms given programmatic methods that exist for communicating with the board. The following capabilities are required for adaptation:

- *Configuration:* This is needed to configure the FPGA(s) on the hardware platform with bitstreams.
- *Clock control:* One or more subroutines are required to single- or multistep the clock on the board.
- *Readback:* This is necessary to read back the state from the FPGA(s) on the board.[4]

Given these capabilities, JHDL can easily be extended to communicate with the board for hardware mode operation.[5] This is achieved by modifying a thin layer of Java translation code so that standard JHDL methods can communicate with the specific C-based device driver subroutines for the board.

12.5 ADVANCED JHDL CAPABILITIES

A variety of design and debug tools have been built on top of JHDL. A few of these are described in the following sections.

12.5.1 Dynamic Testbenches

Some of the power of JHDL derives from its extensive use of the Java feature called *reflection*. The Java reflection API provides a set of methods that a Java program can use to examine the structure of a Java `.class` file. By reflecting on a Java class, a program can determine the names and type signatures of all methods in it and, if desired, dynamically load the class file and construct an object of the class.

[4] To date, JHDL supports hardware mode only on Xilinx platforms, because they contain a readback capability. Experiments have also been done to determine the cost of adding a scan chain to user designs for this purpose when readback is not available [12].

[5] An additional capability would also prove very useful—loading state into the FPGA. In the case of Xilinx FPGAs (the focus of the JHDL hardware mode work), this can be done by modifying the configuration bitstream appropriately and then reconfiguring the FPGA with that bitstream. Thus, this capability is not listed as a strict requirement in the list. A number of the debug experiments described in the next section performed bitstream modification to load state into an FPGA, but would have benefited from a simpler mechanism that did not require a reconfiguration of the FPGA.

The JHDL `dtb` class is a general-purpose testbench tool that uses reflection to automatically perform testbench functions, eliminating the need, in most cases, for the user to write code for constructing the testbench. The user runs `dtb` and specifies the name of the circuit to be constructed on the command line. `dtb` examines the corresponding file (`FullAdder.class`, for example) using reflection to determine the parameters required by its constructor. It then creates the necessary wires and calls the constructor to build an instance of the specified class, connecting it to the wires it created. When `dtb` is used, the `dtb` object itself performs all of the services required of a testbench. For example, it examines the constructed circuit, determines the clocking required, and sets up the clock for simulation. In addition, when everything has been constructed, it brings up `cvt` so that design simulation and netlisting can proceed as usual. All that is required of the user is to provide the simulation stimulus either interactively or via a script.

12.5.2 Behavioral Synthesis

Sea Cucumber [11] is a behavioral synthesis tool that was built on top of the JHDL framework and accepts a behavioral description written in Java that is compiled into bytecodes by any standard Java compiler. It parses these byte codes, discovers instruction-level parallelism, performs other common optimizations, and then synthesizes a circuit by invoking calls to the JHDL Logic library. Advantages provided by the JHDL framework include access to JHDL visualization and debugging tools to verify Sea Cucumber designs and access to JHDL netlisting modules so that the synthesized JHDL circuitry can be converted into netlists for place and route by vendor software. In fact, all of the previously mentioned JHDL features are available to Sea Cucumber, including hardware mode, dynamic test benches, and the like.

A behavioral debugger, also developed in conjunction with Sea Cucumber [7], allows the user to debug fully optimized code in the context of the original user description. It does this by traversing the JHDL circuit structure to retrieve circuit values and presenting them to the user in the context of the JHDL CAD framework.

12.5.3 Advanced Debugging Capabilities

Much of the power of JHDL comes by exploiting a single FPGA feature (readback) to access internal FPGA state and present the data to the user in some form. Because of this enhanced visibility, the current JHDL debugging environment has proved to be effective for verifying and debugging large, complex applications. However, much more powerful debugging capabilities can be achieved if a small part of the FPGA is reconfigured to implement supplemental circuitry to aid debug and validation. This is similar in spirit to the `"-g"` flag used in conventional software compilation where the compiler can enable debugging by inserting additional code. Because FPGA hardware is reconfigurable, any inserted debugging circuitry can be removed when the application is ready for deployment.

Some of the advanced debugging features that are possible via embedded debug circuitry include:

- Signals can be automatically routed to external I/O pins for viewing.
- Unused FPGA circuitry and memory can be used to implement "probe" circuits that sample and store circuit activity during circuit execution.
- Unused FPGA hardware can be used to implement complex, real-time hardware breakpoints.

As long as designers must manually modify their designs in order to embed debug circuitry, these powerful techniques may go unused. The best way to overcome this is to automate the process of synthesizing and embedding debug circuitry into user circuitry—a task best performed directly by the CAD tool environment. The ability to use a debugging tool as an integrated part of the design environment, tied to the original design specification and accessed using standard user interfaces, makes this a powerful and convenient way to develop and verify a design.

As a part of DARPA-funded research at Brigham Young University, researchers investigated a variety of advanced debug mechanisms using JHDL, all of them were aimed at providing a debug system with capabilities similar to those found in software development systems and that are significantly easier to use than manual methods. A few of these mechanisms are described in the following subsections.

Debug circuitry synthesis

In software debugging using the *gdb* symbolic debugger or similar tools, it is not uncommon for the user to temporarily change variable values as a way of determining how the program would behave *if* the variable had that different value. The work described by Graham [6] demonstrated a similar capability for hardware. First, JHDL was used to perform a readback of the FPGA's state. Changes were then made to the bitstream to reflect the user's choice for the new circuit state, and the bitstream was configured back into the FPGA. Upon resumption, the system was seen to continue execution from the previous point but with changed state values.

As another example, in the work described by Graham et al. [5], JHDL and JBits were used together to modify FPGA design bitstreams on the fly in order to rewire embedded logic analyzers to user logic in a placed-and-routed design—all within a few seconds of a mouse click. The collected data could then be viewed in the original design environment using the built-in JHDL GUI framework.

When these features first appeared in JHDL, there were no equivalent commercial debugging tools available. However, with the passage of time, commercial offerings have improved, incorporating some of the features of the original JHDL system. Altera's SignalTap and Xilinx's ChipScope now provide convenient ways to integrate customized logic analyzers into user designs (these are implemented with unused programmable circuitry), and offer separate tools that emulate a logic analyzer display on the PC's monitor. However, JHDL still differs from these products in the level of integration it provides (the debug environment *is* the design environment). Perhaps Synplicity's Identify tool comes closest to JHDL in this regard because it provides the ability to view some circuit behavior in the original VHDL context. Still, none of the commercial offerings allow the user to simultaneously integrate logic analyzers, display these results in the original design environment, and modify the current state of the circuit during debug.

Checkpointing, context switching, and remote access

Checkpointing is defined as saving the state of a computation in a way that the computation can later be restarted from that same point. It is often used in software to allow a long-running computation such as a simulation to be restarted from a known point if, for example, the system it is running on goes down. The concept of readback can easily be extended to extract the state of the entire FPGA platform.

Once this is done, checkpointing of FPGA-based computations can be supported. To do this, the JHDL `HWSystem` was augmented not only to retrieve the state of the FPGA but also to retrieve the state of all memory elements on the hardware platform (FIFOs and memories) and save that information to disk. Later, the state could be retrieved from disk and loaded back onto the FPGA platform, whereupon execution would continue from the time of the checkpoint. What is important is that a *simulation* checkpoint could be loaded onto the FPGA platform and *hardware execution* could be continued from that point. Likewise, a *hardware execution* checkpoint could be loaded into JHDL and a *simulation* continued from that point. With the availability of checkpointing in JHDL, it then became possible to time-share an FPGA platform using context switching (swapping an application off the platform to make room for another). Experiments conducted at Brigham Young University on checkpointing and context switching are described by Landaker et al. [10], to which the interested reader is referred for more information and results.

Finally, JHDL was also modified to permit remote access to an FPGA platform. In this work, the `cvt` and `HWSystem` classes were extended to include a client–server capability so that hardware mode communications with an FPGA platform could be conducted over a network.

12.6 SUMMARY

JHDL is currently in use in a variety of research projects, from module generators systems to behavioral synthesis systems to microarchitectural simulation systems. By providing a framework for the construction, simulation, netlisting, and hardware debug of FPGA-based designs, JHDL allows researchers to focus on tasks other than recreating the infrastructure that JHDL provides. Of particular importance in this regard, is that JHDL provides a target for use by synthesis tools with its primitive libraries and its `Logic` and `Techmapper` classes.

JHDL has been in use since approximately 1998 and was released under an open-source license (*http://www.jhdl.org*) in approximately 2000. Potential users can download either compiled JAR files of the JHDL system, or they can download and build JHDL from sources themselves. Documentation on the JHDL system is provided as well.

References

[1] P. Bellows, B. L. Hutchings. JHDL—An HDL for reconfigurable systems. *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, April 1998.

[2] P. Bertin, D. Roncin, J. Vuillemin. Programmable active memories: A performance assessment. In G. Borriello, C. Ebeling (eds.). *Research on Integrated Systems: Proceedings of the 1993 Symposium*, 1993.

[3] P. Bertin, H. Touati. PAM programming environments: Practice and experience. In D. A. Buell, K. L. Pocek (eds.). *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, April 1994.

[4] D. Galloway. The transmogrifier C hardware description language and compiler for FPGAs. *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, April 1995.

[5] P. Graham, B. Nelson, B. Hutchings. Instrumenting bitstreams for debugging FPGA circuits. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2001.

[6] P. S. Graham. *Logical Hardware Debuggers for FPGA-Based Systems,* Ph.D. thesis, Brigham Young University, 2001.

[7] K. S. Hemmert, J. L. Tripp, B. L. Hutchings, P. A. Jackson. Source level debugger for the Sea Cucumber synthesizing compiler. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2003.

[8] B. L. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, M. Rytting. A CAD suite for high-performance FPGA design. *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, April 1999.

[9] C. Iseli, E. Sanchez. A C++ compiler for FPGA custom execution units synthesis. *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, April 1995.

[10] W. J. Landaker, M. J. Wirthlin, B. L. Hutchings. Multitasking hardware on the SLAAC1-V reconfigurable computing system. *Proceedings of the 12th International Workshop on Field-Programmable Logic and Applications*, Springer-Verlag, September 2002.

[11] J. L. Tripp, P. A. Jackson, B. L. Hutchings. Sea Cucumber: A synthesizing compiler for FPGAs. *Proceedings of the 12th International Workshop on Field-Programmable Logic and Applications*, Springer-Verlag, September 2002.

[12] T. Wheeler, P. Graham, B. Nelson, B. Hutchings. Using design-level scan to improve FPGA design observability and controllability for functional verification. *Proceedings of the 11th International Workshop on Field-Programmable Logic and Applications*, Springer-Verlag, August/September 2001.