A CAD Suite for High-Performance FPGA Design *

Brad Hutchings, Peter Bellows, Joseph Hawkins, Scott Hemmert,
Brent Nelson, Mike Rytting
Department of Electrical and Computer Engineering
Brigham Young University, Provo, UT 84602
hutch@ee.byu.edu

1 Introduction

This paper describes the current status of a suite of CAD tools designed specifically for use by designers who are developing high-performance configurable-computing applications. The basis of this tool suite is JHDL [1], a design tool originally conceived as a way to experiment with Run-Time Reconfigured (RTR) designs. However, what began as a limited experiment to model RTR designs with Java has evolved into a comprehensive suite of design tools and verification aids, with these tools being used successfully to implement high-performance applications in Automated Target Recognition (ATR), sonar beamforming, and general image processing on configurable-computing systems. In response to user demands (those students developing configurable-computing applications), JHDL has been modified and augmented to include:

☞ a graphical debugging tool that allows designers to simulate, debug and hierarchically navigate their designs. This tool can generate a schematic view annotated with simulation or execution data, provide a waveform view of any desired signals, and allows the designer to invoke any public methods implemented by the circuit class (via Java reflection).

☞ a schematic generator that can automatically create a high-quality schematic view of a JHDL description,

☞ an EDIF 2.0 netlist class that generates output compatible with current Xilinx M1 place and route software,

☞ simulation models and transparent run-time support for the Annapolis Microsystems WildForce platform

☞ a graphical floorplanner (under development) that will be used cooperatively with the schematic view to manually floor-plan designs.

In addition to these specific design aids, JHDL provides a unified design environment where a single, user interface can be used for both simulation and execution. This allows the designer to request either simulation or execution (or a mixture of the two) using the exact same commands for both. For example, within this unified environment, commands such as set-breakpoint, examine-variable, single-step, etc., are the same whether performing simulation or execution. This is a big advantage for designers because they can learn a single debugging environment that works for both simulation and execution –in contrast with current systems where execution and simulation environments are distinct and very different. Other design views are unified as well; for example, the schematic view can display either simulated values or values retrieved from the FPGA platform during execution using the same view and interface. Much of the flexibility of this environment is due to the dual simulation/execution environment supported in the original version of JHDL [1]; as reported previously, switching between simulation and execution mode is done by simply clicking a radio button in the circuit browser.

The remainder of the paper briefly reviews the origins of JHDL, why JHDL was adopted as a design tool, and what additional tools and capabilities were added to JHDL to make it a complete design environment.

2 JHDL as an RTR design tool

The original focus of JHDL was on Run-Time Reconfiguration (RTR). As originally published in FCCM98, it made the following contributions.

☞ JHDL used object constructors and destructors to describe circuit structures that dynamically change over time.

☞ JHDL provided a dual simulation/execution environment where a designer could easily switch between either software simulation or hardware execution with a single circuit description (JHDL program).

1. REPORT DATE **1999**	2. REPORT TYPE	3. DATES COVERED **00-00-1999 to 00-00-1999**

4. TITLE AND SUBTITLE **A CAD Suite for High-Performance FPGA Design**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Brigham Young University,Department of Electrical and Computer Engineering,Provo,UT,84602**	8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**

13. SUPPLEMENTARY NOTES **The original document contains color images.**

14. ABSTRACT **see report**

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES **13**	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**			

☞ JHDL supported simultaneous execution of hardware and software: those parts of the application that extend JHDL library classes are executed in hardware while those parts written using generic Java classes are executed on the CPU's Java virtual machine. This makes it possible to fully integrate an application GUI with a hardware description, using a single program to describe both.

Note that the current version of JHDL continues to support these features in addition to the new capabilities that are outlined in this paper.

3 Structural design and FPGAs

In its current state, JHDL is a structural design environment. That being said, the first question that enters most people's minds at this point is this: in this era of behavioral synthesis, why are we still interested in structural design? The answer is that, 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. In addition, for many applications found in the configurable-computing arena, structural capture is simply a faster, easier-to-learn and more effective way to design an application[1].

Structural design often improves the performance of configurable-computing applications because many FPGA-based applications can benefit from manual placement of at least some parts of the design. Manual placement often results in either smaller or faster circuits or both. Indeed, it is still common to see overall improvements of about 2-10x (area x speed) when a few key performance-critical data-path elements are manually placed.

Structural design is key to manual placement. Effective manual placement can only be achieved if the overall organization of the circuit is well understood and this is only possible if the designer controls (or at least understands) how the structure of the circuit is generated. For example, it is very difficult to manually place circuitry that was generated via purely behavioral synthesis. In general, the designer will not understand the circuit organization generated by the synthesis tool and as such will not be able to ascertain how circuit modules should be placed to reduce area or length of interconnect. Moreover, each synthesis run may result in a slightly different circuit structure that renders the previous placement irrelevant thus forcing the designer to redo the manual placement each time the synthesis tool is run.

Of course, one might argue that there is no need for an additional structural design tool because VHDL can be used to design structural circuits. It is possible to use VHDL and associated synthesis tools to design structural circuits; structural VHDL descriptions can be passed through the synthesis tool and the resulting circuit can then be manually placed by the designer. However, it is obviously overkill to use an expensive synthesis tool simply for netlist interchange, e.g., converting a VHDL netlist into an EDIF netlist. It can also be counterproductive because using a synthesis tool for netlist conversion is slow and complicated. Structural VHDL is also extremely verbose, making it difficult to read and to maintain. Moreover, it can be difficult to properly annotate VHDL to include manual placement directives; VHDL was not designed to support physical placement and as such all of the annotation approaches are non-standard and differ among synthesis tools (if the synthesis tool supports it all). This forces the designer to store manual placement information separately from the design which causes additional errors and maintenance headaches.

4 Programmatic Structural Design

Although VHDL is generally a poor choice as a purely structural design tool, it still demonstrates some important strengths over conventional structural design tools such as schematic capture. Because VHDL is a programming language, it is possible to write VHDL programs that programmatically generate circuit structures when the VHDL code is enumerated. That is, rather than enumerate every gate and wire in the form of a VHDL netlist, it is possible to write a VHDL program that will generate the actual circuit structure when it is enumerated. For example, in the case of an 8-bit adder (assuming that a 1-bit adder is available), rather than write the description of the circuit as an enumeration of 8 interconnected adders, the designer writes a for-generate loop that loops 8 times, instantiating an adder and connecting with its neighboring adder on each pass of the loop.

This mode of description is not only much more compact and readable, but more importantly, is much more flexible and powerful than conventional schematic capture. For example, in the case of the adder, this programmatic approach allows the size of the adder to be a variable that can be a parameter that is determined when the program is executed. This one "adder program" can generate adders of any size and is reusable by any VHDL code that wishes to build an adder of some size. This programmatic approach to structural circuit generation is usually referred to as *module gen-*

---

[1] This is not an argument for using only structural design tools for FPGAs. Rather it is an assertion that there is a place for the right kind of structural design tools in any high-performance FPGA design tool kit.

eration and it is often used to create high-performance, parameterizable data-path elements such as adders, multipliers, and more complex circuit modules. It can also be applied generally with good results to any structural design problem.

General purpose languages versus VHDL

This programmatic approach to structural circuit design can be applied even more effectively with *general-purpose* languages such as C, C++, Java, etc. General-purpose languages have some surprising advantages over VHDL even though VHDL was specifically designed as a language for describing circuits. Consider the following advantages that are provided by general-purpose programming languages:

❶ Richer Feature set. Simply put, most general-purpose languages have feature sets that make them more powerful, and easier to learn and use then VHDL. For example, console I/O in general-purpose languages is immediately available and can be used to print out informative messages during program development. This is especially handy when developing complex module generators, for example. (Console I/O is possible with VHDL but it is painful to use and limited at best). Source-level debuggers for these languages are also much more comprehensive and easier to use than their limited VHDL equivalents. All of the tools for general-purpose programming languages are cheap (often free) and are widely available, again, in contrast with VHDL. Finally, the rich set of dynamic data types in Java/C++ make it easier to develop complex programs as opposed to VHDL. As an example, we have found it relatively straightforward to manually convert MATLAB programs into standard Java, use this code to perform fixed-point simulation studies and automatically compare our results with the original MATLAB output, and then complete the design task by evolving the fixed point Java code into structural JHDL — all in the same programming environment.

❷ Simpler Syntax. The syntax of general-purpose languages such as Java is much simpler and more familiar than that of VHDL. As such, a Java-based tool such as JHDL is almost immediately accessible to a programmer who has experience with C, C++, or Java. Even for those who are unfamiliar, the syntax is still much easier to learn and unlike VHDL there is no complicated simulation environment to become familiar with. It has been our experience thus far that our students are much more productive with JHDL than VHDL when designing structural circuits. They also learn Java/JHDL much more quickly.

❸ Opportunities for Codesign. Because the hardware design language and the software design language are the same, it opens opportunities for software-hardware codesign that can significantly ease application development. The entire application can be described in Java (with JHDL libraries), including: the actual circuit description, the run-time control, the GUI (if there is one), any OS interactions such as file I/O, internet communication, etc., and any other operations that need to be performed by a host workstation. Using one description language to describe the entire application greatly eases application development and also provides additional opportunities for novel system integration and debugging tools.

❹ Object-Oriented Flexibility. Object-oriented languages such as Java are well-suited for hardware design. Circuits can be naturally described as objects (as they are in JHDL) with class structure used to control hierarchical organization. Inheritance leads to better reuse of existing code and presents a natural way to access libraries. Finally, inheritance provides a natural way of manually partitioning classes into software and hardware: hardware classes inherit from special JHDL libraries that can be simulated and netlisted. In practice, we have found the object-oriented structure of Java to be a big win in both development and debug.

In summary, general-purpose languages win out over VHDL for programmatically generating circuit structure primarily because they are more flexible, more widely available, and easier to learn and use.

5   Why We Wrote JHDL

We were motivated by three primary issues. First, we have several FPGA-based applications that can benefit from the programmatic structural design approach presented above and no such tool currently exists that met our needs. Second, based on past experience, we felt that there were real advantages to using Java as the programming language: productivity, portability, and specific Java language features such as reflection and dynamic linking. Finally, we felt that we had the opportunity and mandate to develop a CAD tool *that did exactly what we wanted it to*. We decided to take the initial version of JHDL and augment/modify/rewrite it[2] until we were satisfied with it as a tool for developing our current set of applications. We started out by creating a library that supported the Xilinx

---

[2]JHDL in its current state is essentially a complete rewrite of the earlier code.

4K series devices as it is the main device that we are using in our applications[3]. This turned out to be only the first step and many other significant changes were made to JHDL over the last year to accommodate those using JHDL as a design tool on Adaptive Computing Systems (ACS) applications found in the DARPA community. These changes were made to make JHDL easier to use, more terse, faster, and more general.

In its current state, JHDL is a complete structural design environment, including debugging, netlisting and other design aids. Circuits are described by writing Java code that programmatically builds the circuit via the JHDL libraries. Once constructed, these circuits can be debugged and verified with the design browser, a circuit verification and debugging tool. When the designer is satisfied with the functionality of the circuit, an EDIF 2.0 net-list of the circuit can be generated and this can be passed to various backends for place and route (or just routing if the entire circuit has been manually placed). JHDL currently supports Xilinx and HP Chess [8] devices. The remainder of this paper will now report on the current status of JHDL.

6   JHDL Design Strategy

JHDL is a structural design tool in which each circuit element is represented by a unique Java object. JHDL circuit objects inherit from core classes that set up the netlist and simulation model. Circuits are created by calling the constructor for the corresponding JHDL object and passing Wire objects as constructor arguments to be connected to the ports of the circuit. For example, a statement like *new and_o(a, b, q)* might create a new 2-input AND with **a** and **b** as inputs and **q** as the output. We target a specific FPGA technology by selecting the AND from the right library of primitives (e.g. *new byucc.jhdl.Xilinx.XC4000.and* vs. *new byucc.jhdl.Altera.Flex10k.and*). However, we found that this style of design tends to be very tedious and verbose in practice; therefore, the core structural design class, **Logic** class, provides a quick-hardware API to build circuits efficiently. For example, we can design a 1-bit full-adder as shown in Figure 1.

Note specifically how we build the adder logic with method calls, not constructors (e.g. *or3(and(a,b), ... co)*). These are methods inherited from the Logic class that call the appropriate constructors for us (they do not evaluate a logic function!). Each method returns the output wire for the new gate, which allows us to nest these method calls, as shown in the example. Building circuits with nested method calls makes the code much less verbose than if

[3]At the time of the code rewrite, JHDL only supported the Xilinx 6200. RIP.

```
public class FullAdder extends Logic {
 /* Define the ports for a 1-bit full adder */
 public static CellInterface cell˙interface[] = {
   in("a", 1), in("b", 1), in("cin", 1),
   out("s", 1), out("co", 1)
 };

 public FullAdder(Wire a, Wire b, Wire ci, Wire s, Wire co) {
   /* Connect wires to my ports */
   connect(''a'', a); connect(''b'', b); connect(''ci'', ci);
   connect(''s'', s); connect(''co'', co);

   /* Instantiate the logic functions */
   or_o( and(a,b), and(a,ci), and(b,ci), co ); /* co is output */
   xor_o( a, b, ci, s ); /* s is output */

   /* Map the gates to LUTs, and place. */
   map( a, b, ci, s ); place( s, ''R0C0.F'' );
   map( a, b, ci, co ); place( co, ''R0C0.G'' );
 }
```

Figure 1: JHDL Full Adder Example

we were explicitly creating every cell and every wire ourselves. The code almost looks behavioral, and is quite easy to read and understand. Furthermore, the circuit description is platform-independent, because we didn't specify a particular library from which to select the gates.

Targeting platform-specific primitive libraries

This suggests an important question: Now that JHDL supports some level of device independence, how then do I target my circuit for a particular platform? The answer is the companion to Logic, which is the **TechMapper** class. As illustrated in Figure 2, the Logic class uses a TechMapper to select gates from the library that is currently being targeted. For example, when the full-adder constructor (see Figure 1) calls the *xor_o()* method, Logic asks the current system TechMapper to fetch an xor gate. If the current TechMapper is a XC4000TechMapper, it will go to the XC4000 library; similarly, a Flex10kTechMapper[4] would go to its corresponding library. In other words, for each method available in Logic, the TechMapper decides what the appropriate action is for its target technology. This is a powerful approach that allows much of the tedium and verbosity of structural design to be hidden inside an intelligent TechMapper. Also, because the TechMapper is a JHDL system-level property, the user can retarget an entire system for a new platform (well, those parts that were created with Logic methods) simply by changing the system TechMapper before constructing the circuit.

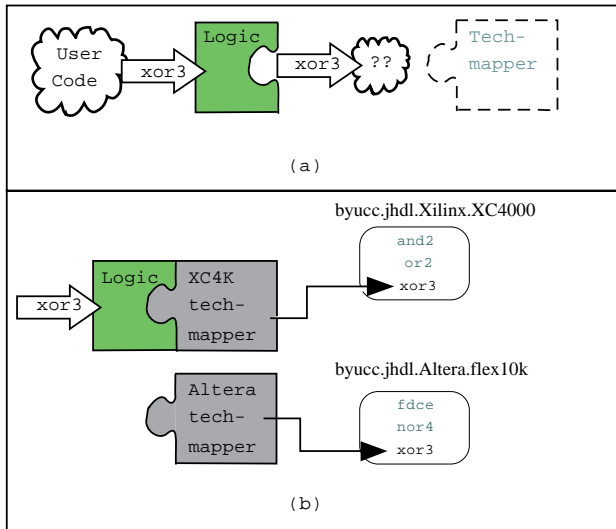[4]Altera is not currently supported in JHDL.

Figure 2: **(a)** The interaction between the Logic class and the TechMapper. **(b)** The TechMapper object makes platform-specific decisions on how to implement the logic requested by the user.

Placement annotation

A crucial element in FPGA design is circuit placement to attain desired levels of performance. In this regard JHDL follows the approach of traditional schematic-capture tools. Placement annotations are applied using string properties and annotation symbols to be interpreted by back-end tools; for example, with XC4000 technology we use FMAP symbols and RLOC property strings to map gates into LUTs and to place LUTs and flip-flops. For our full adder cell, we might place the carry logic as shown in Figure 3, resulting in the cell placement shown in Figure 4.

This placement approach can get pretty verbose, and really isn't useful to us at the front end of the tool chain; typically, such placement annotations are not evaluated until the design reaches the PAR stage. Again, the Logic class helps to reduce this verbosity by providing a general API for these two key placement functions: 1) mapping gates to LUTs, ALUs or other atomic FPGA cells; and 2) setting relative placement of atomic FPGA cells. These are the *map()* and *place()* methods shown at the end of the example full adder code in figure 1, as repeated here:

```
map( a, b, ci, s );  place( s, ''R0C0.F'' );
map( a, b, ci, co ); place( co, ''R0C0.G'' );
```

Note that both styles of placement annotation are valid in JHDL; the latter example is just an abbreviated form of the annotation code in Figure 3. The *map()* method implies

```
/* FMAP needs 4 inputs, so just repeat ci */
fmap f = new fmap( a, b, ci, ci, s );

/* Place in position 0,0, in F-LUT */
f.addProperty( RLOC , ''R0C0.F'' );

/* FMAP needs 4 inputs, so just repeat ci */
fmap g = new fmap( a, b, ci, ci, co );

/* Place in position 0,0, in G-LUT */
f.addProperty( RLOC , ''R0C0.G'' );
```

Figure 3: JHDL Placement Annotations

mapping the network of gates between the input and output wire parameters to an atomic cell; the *place()* method implies setting the relative placement of the atomic cell. Again, the exact interpretation of these method calls is left to the TechMapper, which will determine the appropriate action to take for the target technology.

Providing this placement API provides some important advantages. First, it is somewhat less verbose than the *fmap f = new fmap...* approach shown above. Second, it helps to enforce a colored design methodology where circuit structure and circuit annotations are separated; it gives a common entry point for placement annotation, making the annotations more readable. But most importantly, it provides a window of opportunity for the user to get design assistance from the TechMapper. For example, when *map()* is called, the XC4000TechMapper fully checks the network of gates for validity (i.e. intermediate fan-in or fan-out to the network). When the circuit is constructed, it will fully resolve all placement hints and report any placement conflicts. This means that placement errors are detected at the *front* of the tools chain, which helps us to design circuit architecture and layout concurrently and therefore minimize design cycles.

One final example is in order to demonstrate the power of the Logic API-TechMapper approach. The 1-bit FullAdder cell shown in Figure 1 can be easily converted to an N-bit ripple carry adder with minor modifications as shown in Figure 5.

In this example, when we call *xor_o()*, *and()*, and *or()* on the N-bit wires, the XC4000TechMapper does what makes sense - it instantiates N gates per method to implement a bit-wise function. Most Logic methods automatically adapt to the width of the Wire parameters that are passed in. Similarly, when *map()* and *place()* are called on the N-bit wires, the XC4000TechMapper infers N FMAP symbols, one per bit, and iteratively places those FMAPs in a col-
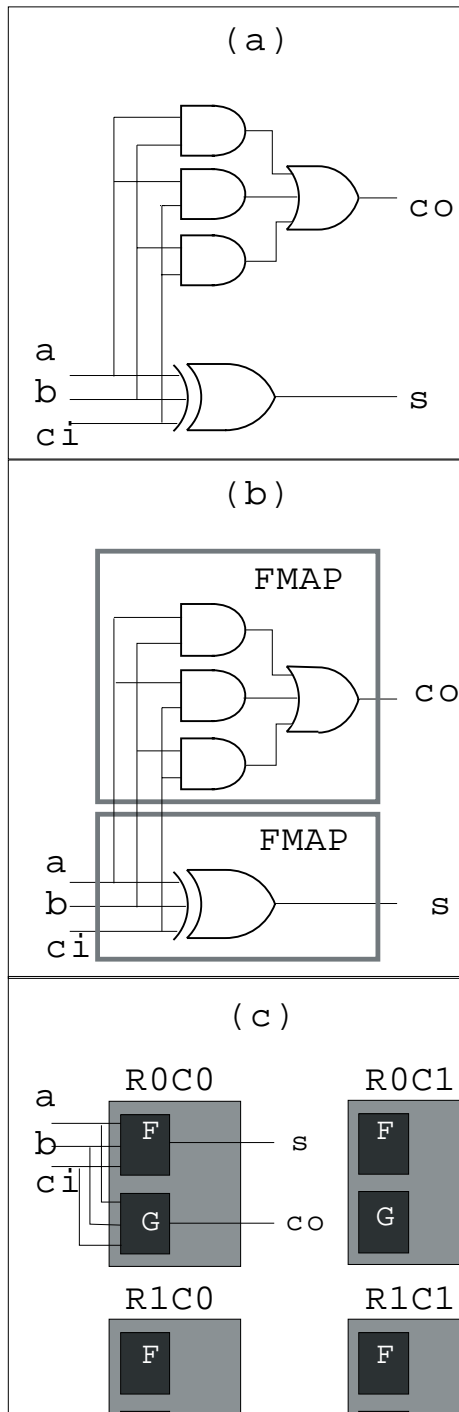
Figure 4: Examples of FPGA placement annotation: **(a)** Full-adder logic **(b)** FMAP symbols demarcate LUT boundaries **(c)** Placement annotation strings indicate relative placement of LUTs

```
public NBitAdder( Wire a, Wire b, Wire s ) {
  connect(''a'', a); connect(''b'', b); connect(''s'', s);

  int width = a.getWidth();
  /* Create the carry busses.  Bit 0 cin is grounded, and the
     cout bus is just a right-shifted version of the cin bus. */
  Wire cin  = new Wire( wire(width-1), gnd() );
  Wire cout = new Wire( nc(), cin.range(width-1, 1) );

  /* Because these wires are N bits each, each method
     creates N gates */
  xor_o( a, b, cin, s);
  or_o( and(a,b), and(a,cin), and(b,cin), cout);

  /* These instantiate 16 FMAPs, one per bit of the wires. */
  map( a, b, cin, s ); map( a, b, cin, cout );

  /* This places the FMAPs in a column, starting at R0C0.
     Placement iterates with (dx,dy) = (0,1) - bit 0 will
     be in R0C0, bit 1 in R1C0, bit 2 in R2C0, ... */
  place( s, 0, 1, ''R0C0.F'' ); place( co,0, 1, ''R0C0.G'' );
}
```

Figure 5: Parameterizable Full Adder in JHDL

umn - specifically, *s[0]* is placed at R0C0.F, *s[1]* is placed at R1C0.F, ... Thus, the TechMapper is able to provide many intelligent design shortcuts to make structural design quick, readable, and less verbose. In our example, we were able to instantiate and hand-place an arbitrary-width adder in only 6 lines of structural code.

The FloorPlanner

The floor-planner is a visualization tool currently under development that lets the user interactively place his design. The FloorPlanner allows platform-specific modules to plug in to provide the graphics and platform-dependent functions. When completed, the FloorPlanner will hierarchical placement views, a drag-and-drop interface, multiple layouts per cell, and so forth. The prototype FloorPlanner is shown in Figure 6.

In contrast to the static approach to annotation shown in the previous section, the floor-planner annotations will not be back-annotated to the source file; rather they will be stored in separate annotations files. This will allow users to have a single circuit description but with multiple layouts that are simultaneously available. The netlister will be able to incorporate these annotation files arbitrarily into its netlists. Because both the FloorPlanner and the TechMapper operate on the top-level JHDL data structures, not an intermediate file, they allow the user to do layout planning in conjunction with circuit design.
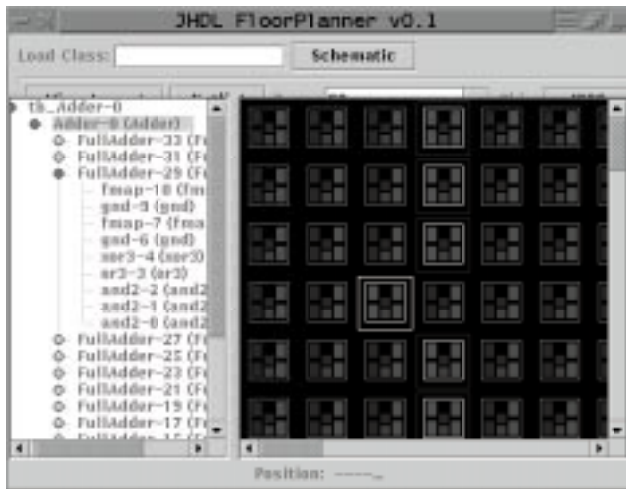
Figure 6: Screen shot of the FloorPlanner.

The Primitive Libraries

JHDL primitive libraries currently exist for the Xilinx XC4000 family. Virtex libraries are under development. The behavioral models for these cells were derived from the Xilinx libraries guide specifications in straightforward fashion. As the libraries grew and JHDL changed during development, we found it more and more difficult to maintain these libraries. Consequently, we created a pseudo-Java language that is used to describe most of the library cells. The language is just Java code that is marked up with JHDL-specific macros. These macros specify the basic structure and behavior of the gate without using any specific Java/JHDL syntax. A Perl pre-compiler translates all the macros into the JHDL syntax-du-jour, outputting compilable Java. Then, when JHDL syntax changes for any reason, we simply modify the pre-compiler output templates and recompile the macro files for all the library cells. Using this method, we have been able to track a moving JHDL target with an entire library, because we almost never have to modify any of the macro files, only the pre-compiler itself. The pre-compiler even generates basic comments and documentation files automatically (using JavaDoc). As an example, Figure 7 illustrates how a simple macro called @interface is translated into a more complex (and fully commented) JHDL port interface block.

Note that all of the Java comments are in "JavaDoc" format that is automatically compiled into stand-alone HTML class documentation.

RAM macro:

```
@class ram16x1s extends Memory {
 @interface {
  d  : in(1);
  we : in(1);
  a  : in(4);
  o  : out(1);
 }
```

Pre-compiled JHDL output:

```
final public class ram16x1s extends Memory {
 public static CellInterface cell˙interface[] = {
  in("d", 1),
  in("we", 1),
  in("a", 4),
  out("o", 1)
 };
 /** The netlist reference name for ram16x1s */
 public static final String cellname = ''ram16x1s'';
```

Figure 7: Generating a RAM Library Primitive

The Net-lister

JHDL currently includes net-listers for EDIF 2.0 (used with Xilinx M1 tools) and a proprietary HP net-list format for Chess [8]. The EDIF netlister works in conjunction with the TechMapper to include target-specific functionality such as clock- and reset-net setup, I/O pad setup, etc. The net-lister will also cooperate with the FloorPlanner to incorporate placement annotation files from either the default location or from a local override file.

7   Debugging and Verifying JHDL Designs

Debugging and verifying a JHDL design is typically done through the *design browser*. The browser allows the user to dynamically load and unload JHDL designs, cycle the global simulation clock, toggle the system reset, and trace the values of wires in the design. The design browser's interface was developed with two major goals in mind: to provide the user with tools to aid in the visualization of a textual JHDL design and to provide tools to automate debugging tasks. While textual design tools are convenient for quickly creating circuits, it is often difficult to identify structural errors in the system. For this reason the ability to graphically visualize the structure of the design is important.

The design browser addresses this need with two tools. First, the tree view provides a hierarchical view of the de-
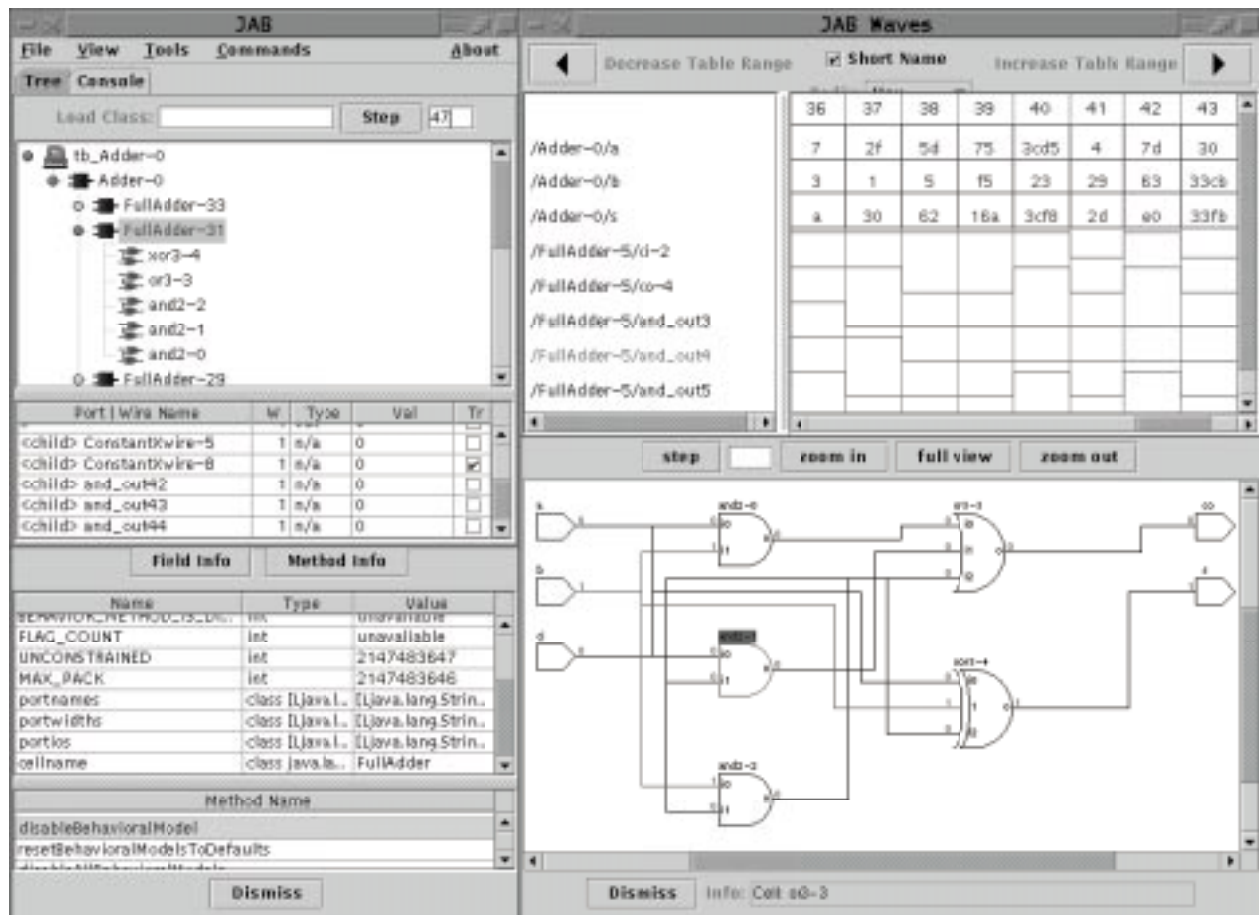
Figure 8: The Design Browser

sign's structure. Second, the schematic tool generates circuit diagrams of the design. In addition, the design browser provides a command console for textual input, a memory viewer, an object browser, and a means for directly controlling the FPGA platform via the same user interface that is used for simulation. *All of the aforementioned design browser functionality applies both to simulation and to the actual execution of the design on an FPGA platform.* For example, during simulation, the wires values displayed in either the tree view or annotated schematic view are retrieved from simulation models of the given wires. During execution, the current state of the relevant FPGA devices is retrieved via readback or some other means and the wire value is updated based on these state values. This makes the design browser equally effective as both a simulation and an execution debugging tool. A screen snapshot of the Design Browser is shown in Figure 8.

Tree View The tree view (upper left of figure) displays the circuit hierarchy in a way similar to that used to display file system hierarchies in computer systems. Cells in the hierarchy are nodes (represented by icons). Like folders in a file system, they can be 'opened' to reveal lower levels in the hierarchy. When a cell in the tree view is selected, additional information is also displayed: the cell's port and wire names, their direction or type, their width, and their current value. Figure 8 shows the tree view of the same full-adder circuit generated from the Java code shown in Figure 1.

Schematic View The schematic generation tool (lower right of figure) takes JHDL circuit descriptions and automatically generates a high-quality schematic similar to [4, 7]. This process usually takes only a few seconds for large designs. It has proven extremely useful in helping designers recognize structural errors in their designs. The designer can descend to the next lower level in the circuit hi-

erarchy by clicking the mouse within the boundaries of a block of interest and the designer can view simulation results as the simulation proceeds — current values of all circuit nodes are displayed on the schematic and are updated as they are computed. Further, a standard library of graphics blocks exists, enabling the schematic viewer to represent each block with the most suitable symbol (gates for logic functions, etc). A Java interface is also provided for users to specify graphics to be used for displaying a given cell in the schematic window. To do so the user embeds a stylized *draw()* method in the definition of his cell[5].

Object Browser   The object browser (lower left of figure) takes advantage of the reflection features available in Java to allow the user to view and manipulate public features of circuit modules in the design. These features include ports, wires, and any public fields or methods. Using the browser the user can modify the value of any native-typed field (int, short, long, boolean, String). More importantly, the user can *invoke* any public method which takes only native-typed arguments. Thus, the user can change field values at run time or call methods which, in turn, can provide specialized debug information. This last feature allows the designer to extend the run-time debug environment by providing cell-specific routines in the design of their cells. As an example, such a routine could be used to provide details on the *internal* state of a complex pipelined circuit element in a compact and understandable way without requiring the user navigate into its gate-level internals in the schematic browser.

Waveform Viewer   The design browser provides a waveform viewer (upper right of figure) to display signal values as waveforms or in a table. Multi-bit values can be viewed in one of several different radices and the window can be panned and zoomed as would be expected.

Command Console   All browser actions, normally accessed using the mouse/buttons/menus, have textual equivalents which can be typed into the command console or read in from a file. Both commands generated in the GUI and those entered at the console pass through a common command interpreter and are also logged in a command history file. The command interpreter can then source this history file or any other command file, thus helping to automate many common debugging tasks. The design hierarchy can also be navigated and browsed using console commands such as **cd** and **ls**.

---

[5]It is *stylized* because it need only specify the lines, arcs, and text which make up the module when viewed full-size. The schematic generator transparently handles all needed scaling and windowing.

Memory View Interface   The memory view interface (not shown) grew out of the need of designers to be able to examine the contents of memory components in designs during simulation. Any cell which implements the methods defined in the memory view interface can be examined using the memory viewer. Using the memory viewer the user can not only view contents of design memories at run time, he can modify them, load them from files, or dump their contents to files.

Hardware Control   A final use of the design browser is to directly control the execution of an application on an FPGA platform, e.g., WildForce, through the same user interfaces that are used to control the simulator. To do so, the design browser communicates with the FPGA platform through a control API that provides generic methods that invoke platform-specific driver functions that perform runtime operations such as setting the clock frequency, stepping the clock, loading a configuration, etc.

WildForce Platform Support

JHDL currently provides simulation and hardware support for the Annapolis Microsystems WildForce platform. The simulation and development environments leverage the basic JHDL functionality, while the runtime environment utilizes the hardware control interface supplied by the design browser and a native library (via Java Native Interface (JNI)) to access the actual Wildforce hardware.

The JHDL models are powerful and easy to use. Creating a design for the wildforce environment is identical to creating any other JHDL design with two exceptions: First, the user must extend the class **pelca**, instead of the usual classes. Second, since this **pelca** class provides a set of method calls to access the physical ports of the PE, user calls to port() can take advantage of those methods to simplify the process.

An example is shown in Figure 9. This simple design instances the adder circuit which has been shown in previous examples. It takes one 16-bit input from the lowest-order 16 bits of the left systolic bus, and takes the second input from bits 31 down to 16 of the same bus. The sum of these two numbers is then output to the lowest 16 bits of the right systolic bus of WildForce.

As can be seen from the above code segment, using a high level language makes it possible to hide much of the detail of the interface logic from the user. When using the VHDL libraries supplied with WildForce, for example, the user must supply an enable bit for each signal which has the possibility of being bi-directional. The JHDL Wildforce model, however, will automatically instance the correct type of pad for each signal tied to a PE pin and will

```
import byucc.jhdl.platforms.WildForce.*
import byucc.jhdl.base.*

public class adder extends pelca {

  public static CellInterface cell˙interface[] = {
    in("a", 16),
    in("b", 16),
    out("sum", 16)
  };

  public adder( pe parent ) {
    super(parent);

    // LeftReg() and RightReg() are methods which will return
    //   the requested wires from the Wildforce environment.
    // This frees the designer from having to understand the
    //   higher-level Wildforce models and wiring details and
    //   makes it simple to drop a design into a Wildforce PE.
    Wire a   = connect("a", LeftReg(15, 0));
    Wire b   = connect("b", LeftReg(31, 16));
    Wire sum = connect("sum", RightReg(15, 0));

    new NBitAdder(this, a, b, sum);
  }
}
```

Figure 9: The NBitAdder in the WildForce Environment

require an enable bit only if that signal is actually used as a bidirectional signal.

Another advantage of the JHDL Wildforce model is that each control signal in the processing element has a default state to which it will be set if the user does not explicitly use that signal. The value to which each unused signal defaults is signal dependent and is automatically handled by the JHDL Wildforce model. As a result, the full adder example shown in Figure 9 requires 15 lines; achieving the same functionality in VHDL requires over 45 lines. The extra lines in VHDL are required mainly to tie unused control signals to their deasserted states.

Finally, to facilitate simulation and hardware debug, the Wildforce model supplies a configuration utility as shown in Figure 10. This module is automatically started by the design browser whenever the Wildforce model is loaded and allows the user to load designs into the FPGAs, memories and FIFOs. This is done by selecting the component of interest and typing the desired file/class name into the text box.

It can be seen in Figure 10 that PE1 has been loaded with the class **NBitAdder** and the crossbar has been loaded with the configuration file **xbar.cfg**. PE2 is currently selected and about to be loaded with the class **accumulator**.
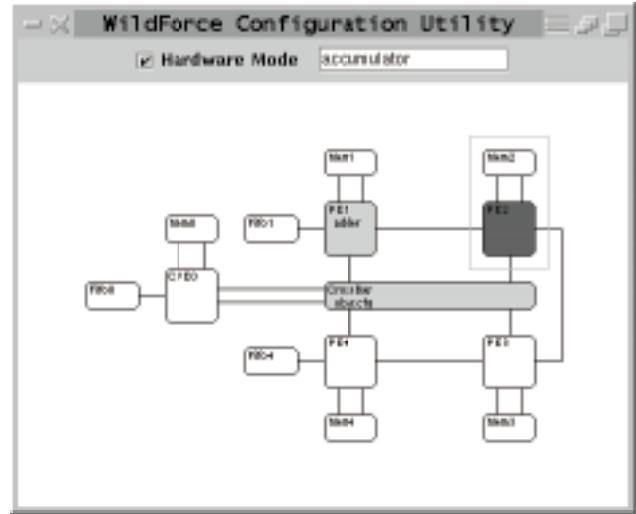

Figure 10: WildForce Configuration Utility

As shown, the system is currently set to hardware mode as denoted by the check box. This check box can be used to toggle the system between hardware mode and simulation — by simply changing the mode, the user can simulate and netlist the design or execute the design on the actual hardware. In either case the built-in features of the browser can then be used to inspect the contents of memory and the values of user defined signals in a PE. As with the other browser modules, all commands available in the configuration utility are also available from the design browser console, simplifying the creation of scripts for automated execution.

8   Applications

Our work on the DARPA-funded SLAAC project has driven much of the applications work done using JHDL to date. This application work is in two areas: automatic target recognition (ATR) and sonar beamforming. To facilitate the completion of these designs we have created a set of module generators including: fixed-point multipliers, CORDIC units, comparators, adders/subtractors, counters, finite state machines, RAM and ROM blocks, and delay lines. Most are parameterized for operand width and level of pipelining and are similar in a number of ways to those described in [3, 9].

The first major JHDL designs using these are now nearing completion for the Wildforce platform. The first, an ATR design, is a regular bit-level pipelined circuit which efficiently implements programs written in the morphological language CYTO. The second, for matched field sonar

beamforming, is less regular and is representative of many typical DSP designs. It consists of multiple state machines and datapaths, each containing a number of multiplier-accumulators and CORDIC rotators. Our experience on these projects is that the availability of such a set of module generators greatly eases the design task, making it much simpler to achieve high density, high performance designs.

Related Work Many programmatic approaches for generating hardware have been described in the literature — both for VLSI- and FPGA-based systems. The following will discuss only FPGA-related work. PamDC [2, 10] was one of the first FPGA-related tools to be based on a general purpose language (C++) that programmatically generated circuits. It was developed for use with DecPerle-1 and was used successfully on a large set of applications. The SPYDER work described in [6] uses extended C++ to generate processor configurations for a custom reconfigurable VLIW coprocessor. The C++ extensions were chosen to ensure that the resulting designs could be simulated using conventional C++ compilers. Transmogrifier C [5] is a C-based FPGA compiler. It handles many C control constructs and does LUT packing and sharing to minimize the size of the resulting circuit. Pam-BLOX [9] is a module generator library which works in conjunction with PamDC and includes generators for multiplication, CORDIC, counters, etc. Pam-BLOX modules contain placement information to ensure compact designs. Reference [3] describes a Java-based circuit-generator framework for FPGA's which provides much more than basic circuit generation. It was designed to be embedded in larger design systems and also provides support to easily retarget new devices, support for simulation and verification, and support for partial evaluation to reduce the size of generated designs when operand values are known at compile-time.

JHDL probably has more in common with PamDC than with the other listed efforts. The basic approach used by the Logic class to describe circuits is similar to that used in PamDC. However, through the technology-mapping class, JHDL provides a level of automation (see the adder example) and device independence not provided by PamDC. The debugging and verification tools associated with JHDL also provide additional functionality (graphical circuit browser, schematic view generator). JHDL also handles wires in a more general and convenient manner. However, JHDL differs from PamDC and all the other efforts on a more fundamental level as it provides a unified simulation/execution environment that allows users to access simulation of an application and *execution of that application* on an FPGA platform from the same unified tool, the design browser (this can also be accomplished by a user-written Java program if desired). Moreover, JHDL also allows the complete application, including the GUI, run-time control code and the circuit design to be captured using a single language (Java) and within a single program.

9 Concluding Remarks

Overall, the JHDL experiment has been a very positive experience. Thus far, we have been successful in using JHDL to develop large applications as noted above. Although only a simple full-adder design was used for illustrative purposes, all of the JHDL software is being used to develop much larger, computationally-challenging designs that consist of multiple Xilinx XC4085s, each connected to several memories. Although no CAD tool is ever perfect, we have noted substantial productivity gains from our students as they have become experienced JHDL users.

In our laboratory, the student users of JHDL range from the complete novice who has not even heard of VHDL, to the experienced VHDL/Synopsys user. The novices have become productive quickly in JHDL (much more quickly than when learning VHDL) and the experienced users really enjoy the ease of use and flexibility that comes from using a general-purpose programming language. To our knowledge, none of our experienced designers has requested to go back to VHDL synthesis.

In spite of early reservations, Java has turned out to be a very good language to base JHDL on. The only other choice was C++ (an object-oriented language was a requirement) and after selecting Java, it has been generally noticed that programs get written and debugged faster and the resulting programs tend to have fewer bugs (primarily, due to simplicity and Java's garbage collection) than when students work in C++.

Portability has also been unexpected bonus. JHDL has been tested on all of the standard 1.1.x Java distributions with no problems. Indeed, parts of JHDL were developed on a wide variety of machines including: NT and Windows boxes, Linux boxes, Macintosh, and HP Unix boxes. In most cases, the code has simply worked as we have moved from platform to platform. Such portability is proving to be a real advantage. Students can take the software home and run it on whatever machine they have at their disposal. More importantly, it is making it much easier to distribute JHDL widely in this research community.

We have used JavaDoc extensively to document all of the primitive libraries, as well as the Logic and TechMapper classes. In addition, libraries of complex module generators are also described using the embedded comments and the JavaDoc utility.

The primary downside of JHDL has been performance. It has not been a significant problem thus far, even on large

designs, however simulation speed remains the bottleneck for large designs. Of course, this tends to be the case no matter what tool you use. As a matter of course, we regularly profile various parts of JHDL to see where it can be sped up; however, we have found that the profiling tools for Java are not as mature nor are they as easy to use those used with C or C++, for example.

In retrospect, if the choice of which programming language to use were made today, we would still choose Java but not only because of the increased productivity. Java has some unique features such as reflection which provide an easy to use dynamic linking feature that we exploited to create the design browser and to allow the user to extend the run-time environment. In addition, Java provides excellent facilities for GUI development (AWT, Swing, etc.) and these have also made it easy to develop the user interfaces for the design browser, etc. Being able to combine the user interface with the circuit description is a powerful capability that we are just beginning to experiment with. Additional information regarding JHDL can be found at *jhdl.ee.byu.edu.*

10 Future Work

JHDL will continue to undergo development. In the future, we plan to investigate the following areas in conjunction with JHDL:

⇢ additional debugging capability including synthesis of debug circuitry to monitor the executing hardware,

⇢ interactive floorplanning to make manual placement more intuitive and easier to verify,

⇢ support for additional devices such as Altera and systems such as DEC Pamette,

⇢ state-machine synthesis to make it easier to develop controllers,

⇢ behavioral synthesis,

⇢ and, additional interfaces to allow interoperation with other CAD tools.

11 Acknowledgments

12 References

[1] P. Bellows and B. L. Hutchings. JHDL - an HDL for reconfigurable systems. In J. M. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 175–184, Napa, CA, April 1998.

[2] P. Bertin and H. Touati. PAM programming environments: Practice and experience. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 133–138, Napa, CA, April 1994.

[3] M. Chu, N. Weaver, K. Sulimma, A. DeHon, and J. Wawrzynek. Object oriented circuit-generators in java. In K. L. Pocek and J. Arnold, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machings*, pages 158–166, Napa, CA, April 1998.

[4] A. Kumar et al. Automatic generation of digital system schematic diagrams. *IEEE Design & Test of Computers*, 3(1):58–65, February 1986.

[5] D. Galloway. The transmogrifier C hardware description language and compiler for FPGAs. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 136–144, Napa, CA, April 1995.

[6] C. Iseli and E. Sanchez. A C++ compiler for FPGA custom execution units synthesis. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 173–179, Napa, CA, April 1995.

[7] C. Lageweg. Designing an automatic schematic generator for a netlist description. Master's thesis, Delft University of Technology, 1998.

[8] A. Marshall, J. Vuillemin, T. Stansfield, Igor Kostarnov, and B. L. Hutchings. A reconfigurable arithmetic array for multimedia applications. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, CA, February 1999. to appear.

[9] O. Mencer, M. Morf, and M. Flynn. Pam-blox: High performance fpga design for adaptive computing. In K. L. Pocek and J. Arnold, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machings*, pages 167–174, Napa, CA, April 1998.

[10] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. Programmable active

memories: Reconfigurable systems come of age. *IEEE Transactions on VLSI Systems*, 4(1):56–69, 1996.